Aspects of algorithms and complexity

Tromp, J.T.

Publication date
1993
Document Version
Final published version

Aspects of Algorithms and Complexity

Aspects of Algorithms and Complexity

Academisch Proefschrift

ter verkrijging van de graad van doctor aan
de Universiteit van Amsterdam op gezag van
de Rector Magnificus prof. dr P.W.M. de Meijer
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op donderdag 16 december 1993 te 13.30 uur

door

Johannes Theodorus Tromp

geboren te Alkmaar

Promotor: prof. dr. ir. P.M.B. Vitányi

Promotie-commissie:
prof. dr. K. Apt
dr. P. van Emde Boas
prof. dr. J. van Leeuwen
dr. G. Tel
dr. L. Torenvliet

Faculteit Wiskunde en Informatica
Universiteit van Amsterdam

Partial support for the research described in this thesis was received as a CWI fellowship (oio plaats), and from the Dutch organization for scientific research NWO under NFI project Aladdin, project number NF 62-376.

to my parents

Acknowledgements

The following publications form the basis of the chapters of this thesis.

Chapter 2: A. Blum, T. Jiang, M. Li, J. Tromp, M. Yannakakis, *Linear Approximation of Shortest Superstrings*, Proc. of the 23rd annual ACM Symposiom on Theory of Computing (STOC91), New Orleans, May 1991, pp. 328–336. *Journal of the ACM*, to appear.

Chapter 4: G. Kissin, J. Tromp, *The energy complexity of threshold and other functions*, CWI Technical Report CS-R9101, January 1991.

Chapter 5: J. Tromp, P. van Emde-Boas, *Associative Storage Modification Machines*, in: "Complexity Theory", Ambos-Spies, Homer, and Schöning (editors), Cambridge University Press.

Chapter 6: J. Tromp, *How to Construct an Atomic Variable*, Proc. of the 3rd International Workshop on Distributed Algorithms, *Lecture Notes in Computer Science 392*, Springer-Verlag, pp. 292–302, 1989.

Chapter 7: M. Li, J. Tromp, P.M.B. Vitányi, *How to Share Concurrent Wait-Free Variables*, (under revision for *Journal of the ACM*).

Chapter 8: J. Tromp, J-H. Hoepman, *Binary Snapshots*, Proc. of the 5th International Workshop on Distributed Algorithms, *Lecture Notes in Computer Science 725*, Springer-Verlag, pp. 18–25 1993.

Chapter 9: J. Tromp, *On Update-Last Schemes*, Parallel Processing Letters 3(1), pp. 25–28, 1993.

The co-authors of the above papers have greatly contributed to the thesis. I would like to thank all of them for their inspiring cooperation, and the referees of the journal publications for their helpful comments.

My greatest debt is to my promotor Paul Vitányi, whose guidance and support have been most stimulating. In Evangelos Kranakis I found an excellent colleague and tutor. I thank Ming Li and Amos Israeli for giving me the opportunity to cooperate with them in Waterloo and Haifa respectively, and making those stays so enjoyable both in work and play.

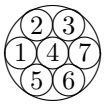
My stay at CWI has been enlivened by numerous guests, each of which added a different facet to my scientific world view: Danny Krizanc, Gloria Kissin, Peter Clote, Hyunyong Shin, Amnon Shaham, Shlomo Moran, Dany Breslauer, Joerg Keller, Peter Gacs, Vladimir Uspensky, Philippas Tsigas, Marina Papatriantafilou, and Alessandro Panconesi.

Our department secretary Marja Hegt has been quite tolerant of my sometimes clumsy behaviour in administrative matters.

In addition, I want to thank all the following people for sharing their time and ideas with me: Victor Allis, Krzysztof Apt, Andries Brouwer, Harry Buhrman, David Chaum, Matthijs Coster, Shlomi Dolev, Herman Ehrenburg, Willem Jan Fokkink, Peter Grunwald, Petra van Haaften, Sibsankar Haldar, Eugene van Heijst, Ted Herman, Ray Hirschfeld, Henk de Koning, Karst Koymans, Pieter van Langen, Jan van Leeuwen, Jan van de Lune, Jeroen van Maanen, Lambert Meertens, Raymond Michiels, Eric Ristad, Marius Schilder, Anneke Schoone, Lex Schrijver, Jan van der Steen, Gerard Tel, Leen Torenvliet, K. Vidyasankar, and Freek Wiedijk.

The cover illustration is a graphical depiction of the game-theoretical value of all up-to-4-ply positions in the game of connect-4. The largest disc represents the (empty) initial board, and is colored white to indicate a 1st player win. Drawn positions are colored light-green, and lost (to the 1st player) positions green.

The seven sub-discs of all but the smallest discs represent positions resulting from a move in one of the seven columns of the board, as follows:



Contents

List of Figures

1
Introduction

The papers collected in subsequent chapters give a fair representation of my research at CWI and the several places I visited abroad. The diversity in subject matter reflects on the one hand the multitude of interests I like to pursue and on the other hand the failure to remain focussed on a single problem area in which to make more extensive explorations.

With such diversity, a title as general as "Aspects of Algorithms and Complexity" seem inevitable, but also holds a promise of finding the links and relations that connect them. To this end, let us consider the concepts involved in some more detail.

1.1 Algorithms

An algorithm is normally understood to be a "recipe" for solving a (computational) problem. That is, a step by step explanation of how to get from a problem instance to a solution. A classical example is the problem of sorting. Here, a problem instance is a list of numbers, like $5, 2, 8, 3, 5$, and a solution is a permutation (re-ordering) of that list, in which the numbers are non-decreasing, like $2, 3, 5, 5, 8$. The problem instance is called the *input* and the solution the *output*. The solution is also sometimes called the *answer*, in particular when the problem instance can be considered a question.

The word 'algorithm' is formally reserved to those recipes that always yield an output in a finite number of steps, in other words, that always *terminate*. The problems considered in Chapters 2, 3, and 4 are of this type. But the word is sometimes also used for processes that aren't even supposed to terminate, like the workings of an elevator. For such processes, the word 'protocol' is more appropriate. A protocol is like a rule of behaviour. The goal of a protocol is not to find a solution to a problem instance, but rather to insure a particular, desirable, behaviour in a system. An elevator protocol must insure for instance

that people get to the floor they want to in a reasonable amount of time. A research field known as 'distributed computing' is devoted to the study of algorithms and protocols for communication, cooperation and competition between multiple, more or less independent, computing agents. Chapters 7,8, and 9 deal with some problems of this type.

As mentioned earlier, an algorithm prescribes a sequence of steps to lead from a given input to an output. This still leaves open the questions of what a step is, how the input is given, and finally how the output is obtained and/or interpreted. We see that an algorithm is not complete without a specification of its operating environment. The classical notion of an algorithm is that of a program running inside a box called computer, with some input device reading the input symbol by symbol, and an output device writing the output symbol by symbol. In this framework, the input and output are words of some input, respectively, output language. A so called 'machine model' specifies the form of programs that a box will run and what operations it can perform in a single step. In some models the steps are executed strictly in sequence, one after the other. These models are said to be *sequential*. A *parallel* model is one where the steps are not necessarily executed in sequence, but is also used for models that are in a sense more powerful than the simplest sequential ones.

The computer will have some form of *storage space* of unlimited capacity, like a tape, where intermediate results are kept. The program usually consists of a list of instructions executed in order except for branches, and each instruction modifies only a small fixed-size part of the storage. This is basically the machine model that Alan Turing envisioned as the machine-equivalent of a human working with pen and paper, and known as a 'Turing machine'. Turing's goal was to have a formal basis for deciding what it means for an input-output function to be 'computable'. He could probably not imagine the large variety of alternative machine models that have been proposed since as a subject of study in its own right. One such model is investigated in Chapter 5.

We can also view the Turing machine as a box that's travelling over a single tape that initially holds the input, and whose contents is taken as output when the box goes into a halting state. We can then replace the tape by an arbitrary graph and allow the box to drop various types of markers on the nodes to get a class of algorithms known as 'bug automata'. These are well suited to labyrinth exploration problems and come into action in Chapter 3.

In a more physical view, algorithms can take the form of *circuits* built from wires and gates. Since the number of inputs of a circuit is *hard-wired* and thus fixed, it is more precise to say that an algorithm corresponds to a *family* of circuits, working on larger and larger inputs. The gates are the steps of the algorithm, and are clearly not executed in sequence. In Chapter 4 we will see an example of a circuit family.

1.2 Complexity

For many problems studied in theoretical computer science, the question of whether it can be solved at all is not interesting, as the answer is invariably yes. The question becomes interesting only when the class of algorithms considered is reduced. Usually we are interested in those algorithms that are the least

complex. A complexity measure is then a way of quantifying how complex an algorithm is. One type of complexity that is of great practical importance is conceptual complexity: how hard is it to understand an algorithm? This however is rather difficult if not impossible to quantify and thus appears only in informal discussion.

If the algorithm can be written down as a sequence of symbols in a standard language, then an obvious complexity measure is the length of that sequence. Combining this with the idea of looking at all algorithms and inputs that produce a given output leads to a very interesting notion of the inherent description complexity of that output. The theory dealing with this notion, known under the names of Kolmogorov Complexity and Algorithmic Information Theory, will not be dealt with in this thesis.

Most complexity measures concern the use of resources during execution of the algorithm, the two most important ones being time and space. On a sequential machine, time is simply measured as the number of steps taken to go from the input to the output. On parallel machines, a parallel notion of time is introduced that reflects the simultaneous execution of steps. For the total number of steps executed, irrespective of timing, the term 'work' is used instead. These notions also apply to circuits, although they carry different names. The (parallel) time of a circuit is taken to be the maximum distance from an input to an output node, and is called the 'depth' of the circuit. used for this purpose, and called 'depth'. The amount of work done by a circuit equals its gate count.

Space is measured as the maximum size of the storage space used. In some machine models the precise formalization of this requires some care to ensure 'compatibility' with other models (not surprising considering the many different forms that storage space comes in). In the case of circuits space can be taken as the (rectangular) area needed by an embedding of the circuit in the plane (with limited cross-over).

In order to proceed from resources like time and space to the corresponding complexity measures, we express their use in terms of the size of the input. Size is defined to be a natural number that is roughly proportional to the length of the input in some standard notation. When inputs are words in some input language, then their size is simply defined as their length. If an input is a graph for instance, then the number of nodes plus the number of edges is a reasonable definition of size. Usually, the bigger the input, the more resources are needed. A complexity measure tells you how quickly the use of resources grows with input size: it is a function that gives for each size the maximum amount of resource used, over all inputs of that size. Apart from this *worst-case* measure, one can also consider an *average case* complexity, where the average of resource used is taken over all inputs.

Chapter 4 considers an practically significant resource for circuits, namely energy consumption. In conventional technologies, whenever the input to a circuit changes, some subset of the gates and wires will switch and the energy dissipated hereby in the from of heat is proportional to the sum area of all switching elements.

In *randomized* algorithms, choices can be made based on the outcome of random coin flips. Randomness is used either to ensure a high probability of giving the correct answer, or to limit the (expected) time to find the correct answer. In either case, number of coin flips is a useful resource.

Closely related to randomness, is the resource of *queries*. A query is a question that the algorithm can ask, from some set of allowed questions, to always be given the right answer. Like a random coin flip, this is one external bit of information, except it can only come out one way.

The last 'resource' we'll discuss is the size of the output. This becomes interesting when for each input, more than one output solution is possible. The closer in size a solution is to the minimal one, the better. Chapter 2 is exclusively concerned with this output approximation complexity for a specific problem.

1.3 Trade-offs

Complexity measures can rarely be considered in isolation. This is because one resource can often be minimized at the cost of several others. Most resources can be minimized at the cost of making the time exponential in the size of input. Since exponential time (or for that matter exponential anything) is considered an extremely bad property for an algorithm to have, this trade-off is hardly ever even considered. We are interested in trade-offs between resources that all remain polynomial.

In Chapter 3, a trade-off is established between time and space for a particular exploration-type problem. It is shown that various different algorithms have the same space-time-product complexity, up to constant factors. Sometimes algorithms are designed with a single parameter that can be varied to produce any desired trade-off of resources between two extremes.

In the domain of circuits, the familiar time-space trade-off translates into a depth-area tradeoff. It is known for instance that a logarithmic depth circuit on n inputs needs on the order of $n \log(n)$ area to be embedded in the plane (assuming the n inputs lie on a convex boundary). If the depth is relaxed, then often a linear embedding is possible. For circuits, logarithmic depth is considered just as desirable as polynomial time is for sequential machines. Thus, in Chapter 4, the resource of energy is minimized under the requirement of logarithmic depth.

1.4 Overview

In Chapter 2, we consider the following problem: given a collection of strings $s_1, \ldots, s_m$, find the shortest string s such that each s_i appears as a substring (a consecutive block) of s. Although this problem is known to be NP-hard, a simple greedy procedure appears to do quite well and is routinely used in DNA sequencing and data compression practice, namely: repeatedly merge the pair of (distinct) strings with maximum overlap until only one string remains. Let n denote the length of an optimal (shortest) superstring. A common conjecture states that the above greedy procedure produces a superstring of length $O(n)$ (in fact, $2n$), yet the only previous nontrivial bound known for *any* polynomial-time algorithm is a recent $O(n \log n)$ result.

We show that the greedy algorithm does in fact achieve a constant factor approximation, proving an upper bound of $4n$. Furthermore, we present a simple

modified version of the greedy algorithm that we show produces a superstring of length at most $3n$. We also show the superstring problem to be *MAX SNP-hard*, which implies that a polynomial-time approximation scheme for this problem is unlikely.

In Chapter 3, we examine the space complexity of flood-filling. Fill algorithms are commonly used for changing the color of a region of pixels. A flood-fill algorithm (FFA) is given a *seed* pixel from which it starts exploring the region delimited by a *boundary* of arbitrary shape. Most known FFAs can be notoriously memory hungry, using in the worst case even more space than is devoted to storing the screen image. While such regions never show up in practice, it may be of interest to find an FFA with minimal worst-case memory requirements. We present an FFA that uses only a constant amount of space, in addition to that in which the image is stored. The price it pays for this memory friendliness is a possible lack of speed—in the worst case time is quadratic in the number of pixels. It thus achieves the same space-time product of $O(n^2)$ as do the common FFAs with linear space and linear time, illustrating a well-known time-space tradeoff.

Chapter 4 turns to the study of a *hardwired* algorithm; a novel construction is described that yields fast, minimum energy VLSI circuits that compute k-threshold and count-to-k functions. The results are obtained in the Uniswitch Model of switching energy.

In Chapter 5, we move from the study of algorithms to the study of computational models. We present a parallel version of the storage modification machine. This model, called the Associative Storage Modification Machine (ASMM), has the property that it can recognize in polynomial time exactly what Turing machines can recognize in polynomial space. The model therefore belongs to the Second Machine Class, consisting of those parallel machine models that satisfy the parallel computation thesis. The Associative Storage Modification Machine obtains its computational power from following pointers in the reverse direction.

Chapters 6,7 and 8 consider the space and time complexity of constructing certain types of shared memory out of simpler building blocks. The protocols involved are designed to be wait-free: any operation on the constructed memory can be completed with only a bounded number of accesses to the simpler memory objects, irrespective of the relative execution speeds. Such implementations, where processors need not wait for each other to get access to memory, help to exploit the amount of parallelism inherent in distributed systems.

We present solutions to the problem of simulating an atomic single-reader, single-writer variable with non-atomic bits. The first construction, for the case of a 2-valued atomic variable (bit), achieves the minimal number of non-atomic bits needed. The main construction of a multi-bit variable avoids repeated writing (resp. reading) of the value in a single write (resp. read) action on the simulated atomic variable. It improves on existing solutions of that type in simplicity and in the number of non-atomic bits used, both in presence and in accesses per read/write action. We show how to verify these constructions by machine, based on atomicity-testing automata.

Chapter 7 presents a construction of an multi-user atomic variable directly from single-writer, single-reader atomic variables. It uses a linear number of control bits, and a linear number of accesses per Read/Write running in con-

stant parallel time.

In Chapter 8 we consider the *atomic snapshot* object in its simplest form where each cell contains a single bit. We demonstrate the 'universality' of this binary snapshot object by presenting an efficient linear-time implementation of the general multi-bit atomic snapshot object using an atomic binary snapshot object as a primitive. Thus, the search for an efficient (sub-quadratic or linear time) wait-free atomic snapshot implementation may be restricted to the binary case.

In the final Chapter, number 9, we introduce the notion of Update-Last Scheme as a distributed method of storing an index, and derive exact bounds on their space complexity.

2

Linear Approximation of Shortest Superstrings

2.1 Introduction

Given a finite set of strings, we would like to find their shortest common superstring. That is, we want the shortest possible string s such that every string in the set is a substring of s.

The question is NP-hard [5, 6]. Due to its important applications in data compression [14] and DNA sequencing [8, 9, 13], efficient approximation algorithms for this problem are indispensable. We give an example from the DNA sequencing practice. A DNA molecule can be represented as a character string over the set of nucleotides $\{A, C, G, T\}$. Such a character string ranges from a few thousand symbols long for a simple virus to approximately 3×10^9 symbols for a human being. Determining this representation for different molecules, or *sequencing* the molecules, is a crucial step towards understanding the biological functions of the molecules. With current laboratory methods, only small fragments (chosen from unknown locations) of at most 500 bases can be sequenced at a time. Then from hundreds, thousands, sometimes millions of these fragments, a biochemist *assembles* the superstring representing the whole molecule. A simple greedy algorithm is routinely used [8, 13] to cope with this job. This algorithm, which we call GREEDY, repeatedly merges the pair of (distinct) strings with maximum overlap until only one string remains. It has been an open question as to how well GREEDY approximates a shortest common superstring, although a common conjecture states that GREEDY produces a superstring of length at most two times optimal [14, 15, 16].

From a different point of view, Li [9] considered learning a superstring from randomly drawn substrings in the Valiant learning model [17]. In a restricted sense, the shorter the superstring we obtain, the smaller the number of samples are needed to infer a superstring. Therefore finding a good approximation bound for shortest common superstring implies efficient learnability or inferability of DNA sequences [9]. Our linear approximation result improves Li's

$O(n \log n)$ approximation by a multiplicative logarithmic factor.

Tarhio and Ukkonen [15] and Turner [16] established some performance guarantees for GREEDY with respect to the "compression" measure. This basically measures the number of symbols saved by GREEDY compared to plainly concatenating all the strings. It was shown that if the optimal solution saves l symbols, then GREEDY saves at least $l/2$ symbols. But, in general this implies no performance guarantee with respect to optimal length since in the best case this only says that GREEDY produces a superstring of length at most half the total length of all the strings.

In this chapter we show that the superstring problem *can* be approximated within a constant factor, and in fact that algorithm GREEDY produces a superstring of length at most $4n$. Furthermore, we give a simple modified greedy procedure MGREEDY that also achieves a bound of $4n$, and then present another algorithm TGREEDY, based on MGREEDY, that we show achieves $3n$.

The rest of the chapter is organized as follows: Section 2.2 contains notation, definitions, and some basic facts about strings. In Section 2.3 we describe our main algorithm MGREEDY with its proof. This proof forms the basis of the analysis in the next two sections. MGREEDY is improved to TGREEDY in Section 2.4. We finally give the $4n$ bound for GREEDY in Section 2.5. In Section 2.7, we show that the superstring problem is *MAX SNP-hard* which implies that there is unlikely to exist a polynomial time approximation scheme for the superstring problem.

2.2 Preliminaries

Let $S = \{s_1, \ldots, s_m\}$ be a set of strings over some alphabet Σ. Without loss of generality, we assume that the set S is "substring-free" in that no string $s_i \in S$ is a substring of any other $s_j \in S$. A *common superstring* of S is a string s such that each s_i in S is a substring of s. That is, for each s_i, the string s can be written as $u_i s_i v_i$ for some u_i and v_i. We will use n and $\mathrm{OPT}(S)$ interchangeably for the length of the *shortest* common superstring for S. Our goal is to find a superstring for S whose length is as close to $\mathrm{OPT}(S)$ as possible.

Example. Assume we want to find the shortest common superstring of all words in the following sentence: "Alf ate half lethal alpha alfalfa". The word "alf" is a substring of both "half" and "alfalfa", so we can immediately eliminate it. Our set of words is now $S_0 = \{$ ate, half, lethal, alpha, alfalfa $\}$. A trivial superstring is "atehalflethalalphaalfalfa" of length 25, which is simply the concatenation of all substrings. A shortest common superstring is "lethalphalfalfate", of length 17, saving 8 characters over the previous one (a compression of 8). Looking at what GREEDY would make of this example, we see that it would start out with the largest overlaps from "lethal" to "half" to "alfalfa" producing "lethalfalfa". It then has 3 choices of single character overlap, two of which lead to another shortest superstring "lethalfalfalphate", and one of which is lethal in the sense of giving a superstring that is one character longer. In fact, it is easy to give an example where GREEDY outputs a string almost twice as long as the optimal one, for instance on input $\{c(ab)^k, (ba)^k, (ab)^k c\}$.

For two strings s and t, *not necessarily distinct*, let v be the longest string

such that $s = uv$ and $t = vw$ for some *non-empty* strings u and w. We call $|v|$ the (amount of) *overlap* between s and t, and denote it as $ov(s,t)$. Furthermore, u is called the *prefix* of s with respect to t, and is denoted $pref(s,t)$. Finally, we call $|pref(s,t)| = |u|$ the *distance* from s to t, and denote it as $d(s,t)$. So, the string $uvw = pref(s,t)t$, of length $d(s,t)+|t| = |s|+|t|-ov(s,t)$ is the shortest superstring of s and t in which s appears (strictly) before t, and is also called the *merge* of s and t. For $s_i, s_j \in S$, we will abbreviate $pref(s_i,s_j)$ to simply $pref(i,j)$, and $d(s_i,s_j)$ and $ov(s_i,s_j)$ to $d(i,j)$ and $ov(i,j)$ respectively. The overlap between a string and itself is called a *self-overlap*. As an example of self-overlap, we have for the string $s =$ undergrounder an overlap of $ov(s,s) = 5$ Also, $pref(s,s) =$ undergro and $d(s,s) = 8$. The string $s =$ alfalfa, for which $ov(s,s) = 4$, shows that the overlap is not limited to half the total string length.

Given a list of strings $s_{i_1}, s_{i_2}, \ldots, s_{i_r}$, we define the superstring $s = \langle s_{i_1}, \ldots, s_{i_r} \rangle$ to be the string $pref(i_1,i_2)pref(i_2,i_3)\cdots pref(i_{r-1},i_r)s_{i_r}$. That is, s is the shortest string such that $s_{i_1}, s_{i_2}, \ldots, s_{i_r}$ appear *in order* in that string. For a superstring of a substring-free set, this order is well-defined, since substrings cannot 'start' or 'end' at the same position, and if substring s_j starts before s_k, then s_j must also end before s_k. Define $first(s) = s_{i_1}$ and $last(s) = s_{i_r}$. In each iteration of GREEDY the following invariant holds:

CLAIM **2.1** For two distinct strings s and t in GREEDY's set of strings, neither $first(s)$ nor $last(s)$ is a substring of t.

PROOF. Initially, $first(s) = last(s) = s$ for all strings, so the claim follows from the fact that S is substring-free. Suppose that the invariant is invalidated by a merge of two strings t_1 and t_2 into a string $t = \langle t_1, t_2 \rangle$ that has, say, $first(s)$ as a substring. Let $t = u\ first(s)\ v$. Since $first(s)$ is not a substring of either t_1 or t_2, it must properly 'contain' the piece of overlap between t_1 and t_2, *i.e.*, $|first(s)| > ov(t_1,t_2)$ and $|u| < d(t_1,t_2)$. Hence, $ov(t_1,s) > ov(t_1,t_2)$; a contradiction. □

So when GREEDY (or its variation MGREEDY that we introduce later) chooses s and t as having the maximum overlap, then this overlap $ov(s,t)$ in fact equals $ov(last(s), first(t))$, and as a result, the merge of s and t is $\langle first(s), \ldots, last(s), first(t), \ldots, last(t) \rangle$. We can therefore say that GREEDY *orders* the substrings, by finding the shortest superstring in which the substrings appear in that order.

We can rephrase the above in terms of permutations. For a permutation π on the set $\{1, \ldots, m\}$, let $S_\pi = \langle s_{\pi(1)}, \ldots, s_{\pi(m)} \rangle$. In a shortest superstring for S, the substrings appear in some total order, say $s_{\pi(1)}, \ldots, s_{\pi(m)}$, hence it must equal S_π.

We will consider a traveling salesman problem on a weighted directed complete graph G_S derived from S and show that one can achieve a factor of 4 approximation for TSP on that graph, yielding a factor of 4 approximation for the shortest-common-superstring problem. Graph $G_S = (V, E, d)$ has m vertices $V = \{1, \ldots, m\}$, and m^2 edges $E = \{(i,j) : 1 \le i,j \le m\}$. Here we take as weight function the distance $d(,)$: edge (i,j) has weight $d(i,j) = d(s_i,s_j)$, to obtain the *distance graph*. This graph is similar to one considered by Turner in the end of his paper [16]. Later we will take the overlap $ov(,)$ as the weight function to obtain the *overlap graph*. We will call s_i the string *associated* with

vertex i, and let $\mathit{pref}(i,j) = \mathit{pref}(s_i, s_j)$ be the string associated with edge (i,j).

As examples we draw in Figure 2.1 the overlap graph and the distance graph for our previous example $S_0 = \{$ ate, half, lethal, alpha, alfalfa $\}$. All edges not shown have overlap 0. Note that the sum of the distance and overlap weights on an edge (i,j) is the length of the string s_i.

Notice now that $\mathrm{TSP}(G_S) \leq \mathrm{OPT}(S) - \mathit{ov}(\mathit{last}(s), \mathit{first}(s)) \leq \mathrm{OPT}(S)$, where $\mathrm{TSP}(G_S)$ is the cost of the minimum weight Hamiltonian cycle on G_S. The reason is that turning any superstring into a Hamiltonian cycle by overlapping its last and first substring saves on cost by charging $\mathit{last}(s)$ for only $d(\mathit{last}(s), \mathit{first}(s))$ instead of its full length.

We now define some notation for dealing with directed cycles in G_S. Call two strings s, t equivalent, $s \equiv t$, if they are cyclic shifts of each other, *i.e.*, if there are strings u, v such that $s = uv$ and $t = vu$. If c is a directed cycle in G_S with vertices $i_0, \ldots, i_{r-1}$ in order around c, we define $\mathit{strings}(c)$ to be the equivalence class $[\mathit{pref}(i_0, i_1)\mathit{pref}(i_1, i_2) \cdots \mathit{pref}(i_{r-1}, i_0)]$ and $\mathit{strings}(c, i_k)$ the rotation starting with $\mathit{pref}(i_k, i_{k+1})$, *i.e.*, the string $\mathit{pref}(i_k, i_{k+1}) \cdots \mathit{pref}(i_{k-1}, i_k)$, where subscript arithmetic is modulo r. Let us say that an equivalence class $[s]$ has *periodicity* k ($k > 0$), if s is invariant under a rotation by k characters ($s = uv = vu, |u| = k$). Obviously, $[s]$ has periodicity $|s|$. A moment's reflection shows that the minimum periodicity of $[s]$ must equal the number of distinct rotations of s. This is the size of the equivalence class and denoted by $\mathit{card}([s])$. Furthermore, it is easily proven that if $[s]$ has periodicities a and b, then it has periodicity $\gcd(a, b)$ as well. (See, *e.g.*, [4].) It follows that all periodicities are a multiple of the minimum one. In particular, we have that $|s|$ is a multiple of $\mathit{card}([s])$.

In general, we will denote a cycle c with vertices $i_1, \ldots, i_r$ in the order by "$i_1 \to \cdots \to i_r \to i_1$." Also, let $w(c)$, the *weight* of cycle c, equal $|s|, s \in \mathit{strings}(c)$. For convenience, we will say that s_j is in c, or "$s_j \in c$" if j is a vertex of the cycle c.

Now, a few preliminary facts about cycles in G_S. Let $c = i_0 \to \cdots \to i_{r-1} \to i_0$ and c' be cycles in G_S. For any string s, s^k denotes the string consisting of k copies of s concatenated together.

CLAIM **2.2** Each string s_{i_j} in c is a substring of s^k for all $s \in \mathit{strings}(c)$ and sufficiently large k.

PROOF. By induction, s_{i_j} is a prefix of $\mathit{pref}(i_j, i_{j+1}) \cdots \mathit{pref}(i_{j+l-1}, i_{j+l})\, s_{i_{j+l}}$ for any $l \geq 0$ (addition modulo r). Taking $k = \lceil |s_{i_j}|/w(c) \rceil$ and $l = kr$ we get that s_{i_j} is a prefix of $\mathit{pref}(i_j, i_{j+1}) \cdots \mathit{pref}(i_{j+kr-1}, i_{j+kr}) = \mathit{strings}(c, i_j)^k$, which itself is a substring of s^{k+1} for any $s \in \mathit{strings}(c)$. □

CLAIM **2.3** If each of $\{s_{j_1}, \ldots, s_{j_r}\}$ is a substring of s^k for some string $s \in \mathit{strings}(c)$ and sufficiently large k, then there exists a cycle of weight $|s| = w(c)$ containing all these strings.

PROOF. In a (infinite) repetition of s, every string s_i appears as a substring at every other $|s|$ characters. This naturally defines a circular ordering of the strings $\{s_{j_1}, \ldots, s_{j_r}\}$ and the strings in c whose successive distances sum to $|s|$. □

CLAIM **2.4** The superstring $\langle s_{i_0}, \cdots, s_{i_{r-1}} \rangle$ is a substring of $\mathit{strings}(c, i_0) s_{i_0}$.

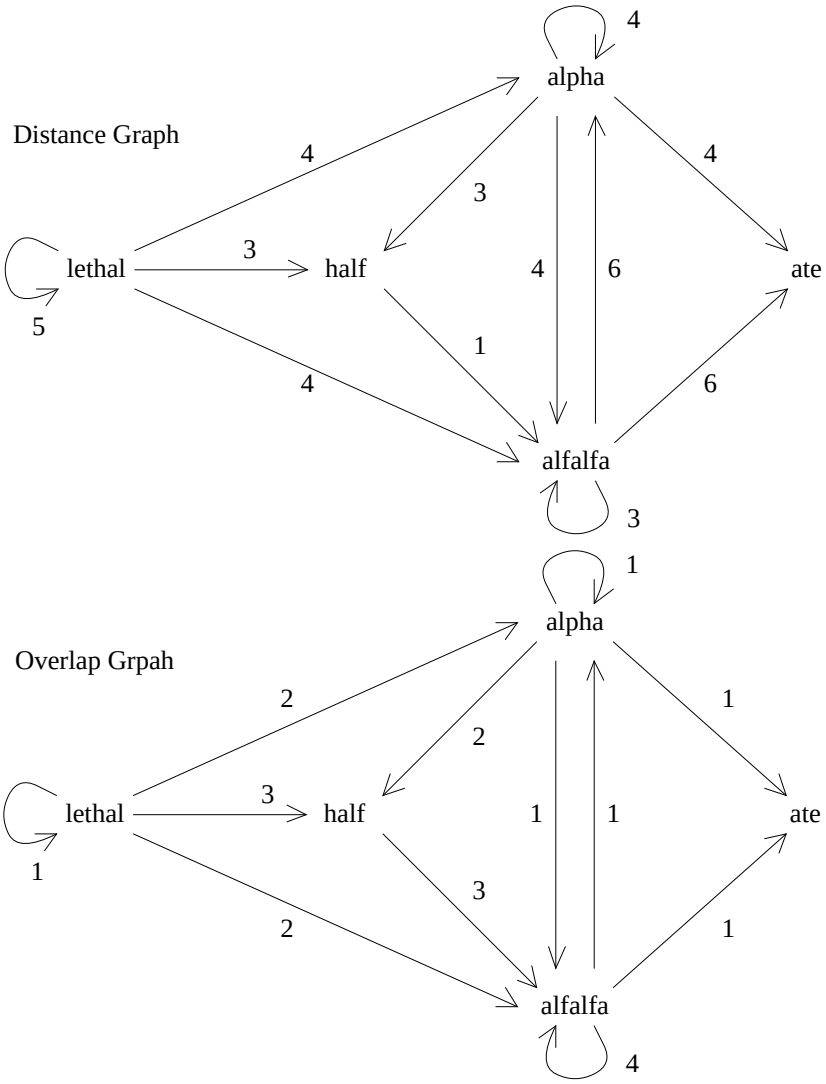


FIGURE 2.1. The overlap and distance graphs.

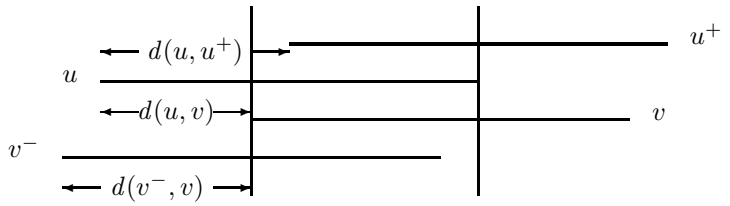


FIGURE 2.2. Strings and overlaps

PROOF. String $\langle s_{i_0}, \ldots, s_{i_{r-1}} \rangle$ is clearly a substring of $\langle s_{i_0}, \ldots, s_{i_{r-1}}, s_{i_0} \rangle$, which by definition equals $pref(i_0, i_1) \cdots pref(i_{r-1}, i_0) s_{i_0} = strings(c, i_0) s_{i_0}$. □

CLAIM **2.5** If $strings(c') = strings(c)$, then there exists a third cycle $\tilde{c}$ with weight $w(c)$ containing all vertices in c and all those in c'.
PROOF. Follows from claims 2.2 and 2.3. □

CLAIM **2.6** There exists a cycle $\tilde{c}$ of weight $card(strings(c))$ containing all vertices in c.
PROOF. Let u be the prefix of length $card(strings(c))$ of some string $s \in strings(c)$. By our periodicity arguments, $|u|$ divides $|s| = w(c)$, and $s = u^j$ where $j = w(c)/|u|$. It follows that every string in $strings(c) = [s]$ is a substring of u^{j+1}. Now use Claim 2.3 for u. □

The following lemma has been proved in [15, 16]. Figure 2.2 gives a graphical interpretation of it. In the figure, the vertical bars surround pieces of string that match, showing a possible overlap between v^- and u^+, giving an upper bound on $d(v^-, u^+)$.

LEMMA **2.7** *Let u, u^+, v^-, v be strings, not necessarily different, such that $ov(u, v) \geq \max\{ov(u, u^+), ov(v^-, v)\}$. Then, $ov(u, v) + ov(v^-, u^+) \geq ov(u, u^+) + ov(v^-, v)$, and $d(u, v) + d(v^-, u^+) \leq d(u, u^+) + d(v^-, v)$.*

That is, given the choice of merging u to u^+ and v^- to v or instead merging u to v and v^- to u^+, the best choice is that which contains the pair of largest overlap. The conditions in the above Lemma are also known as "Monge conditions" in the context of transportation problems [1, 3, 7]. In this sense the Lemma follows from the observation that optimal shipping routes do not intersect. In the string context, we are transporting 'items' from the ends of substrings to the fronts of substrings.

2.3 A $4 \cdot \mathrm{OPT}(S)$ bound for a modified greedy algorithm

Let S be a set of strings and G_S the associated graph. Now, although finding a minimum weight Hamiltonian cycle in a weighted directed graph is in general a hard problem, there *is* a polynomial-time algorithm for a similar problem known as the *assignment problem* [10]. Here, the goal is simply to find a decomposition of the graph into cycles such that each vertex is in exactly one cycle and the

total weight of the cycles is minimized. Let $\mathrm{CYC}(G_S)$ be the weight of the minimum assignment on graph G_S, so $\mathrm{CYC}(G_S) \leq \mathrm{TSP}(G_S) \leq \mathrm{OPT}(S)$.

The proof that a modified greedy algorithm MGREEDY finds a superstring of length at most $4 \cdot \mathrm{OPT}(S)$ proceeds in two stages. We first show that an algorithm that finds an optimal assignment on G_S, then opens each cycle into a single string, and finally concatenates all such strings together has a performance ratio of at most 4. We then show (Theorem 2.10) that in fact, for these particular graphs, a greedy strategy can be used to find optimal assignments. This result can also be found (in a somewhat different form) as Theorem 1 in Hoffman's 1963 paper [7].

Consider the following algorithm for finding a superstring of the strings in S.

Algorithm Concat-Cycles

1. On input S, create graph G_S and find a minimum weight assignment C on G_S. Let C be the collection of cycles $\{c_1, \ldots, c_p\}$.

2. For each cycle $c_i = i_1 \to \cdots \to i_r \to i_1$, let $\tilde{s}_i = \langle s_{i_1}, \ldots, s_{i_r} \rangle$ be the string obtained by opening c_i, where i_1 is arbitrarily chosen. The string $\tilde{s}_i$ has length at most $w(c_i) + |s_{i_1}|$ by Claim 2.4.

3. Concatenate together the strings $\tilde{s}_i$ and produce the resulting string $\tilde{s}$ as output.

THEOREM **2.8** *Algorithm Concat-Cycles produces a string of length at most* $4 \cdot OPT(S)$.

Before proving Theorem 2.8, we first need a preliminary lemma giving an upper bound on the amount of overlap possible between strings in different cycles of C. The lemma is also implied by the results in [4].

LEMMA **2.9** *Let c and c' be two cycles in a minimum weight assignment C with $s \in c$ and $s' \in c'$. Then, the overlap between s and s' is less than $w(c) + w(c')$.*

PROOF. Let $x = \mathit{strings}(c)$ and $x' = \mathit{strings}(c')$. Since C is a minimum weight assignment, we know $x \neq x'$. Otherwise, by Claim 2.5, we could find a lighter assignment by combining the cycles c and c'. In addition, by Claim 2.6, $w(c) \leq \mathit{card}(x)$.

Suppose that s and s' overlap in a string u with $|u| \geq w(c) + w(c')$. Denote the substring of u starting at the i-th symbol and ending at the j-th as $u_{i,j}$. Since by Claim 2.2, s is a substring of t^k for some $t \in x$ and large enough k and s' is a substring of $t'^{k'}$ for some $t' \in x'$ and large enough k', we have that $x = [u_{1,w(c)}]$ and $x' = [u_{1,w(c')}]$. From $x \neq x'$ we conclude that $w(c) \neq w(c')$; assume without loss of generality that $w(c) > w(c')$. Then

$$u_{1,w(c)} = u_{1+w(c'),w(c)+w(c')} =$$

$$u_{1+w(c'),w(c)} u_{w(c)+1,w(c)+w(c')} = u_{1+w(c'),w(c)} u_{1,w(c')}.$$

This shows that x has periodicity $w(c') < w(c) \leq \mathit{card}(x)$, which contradicts the fact that $\mathit{card}(x)$ is the minimum periodicity. □

PROOF. **(of Theorem 2.8.)** Since $C = \{c_1, \ldots, c_p\}$ is an optimal assignment, $\mathrm{CYC}(G_S) = \sum_{i=1}^{p} w(c_i) \leq \mathrm{OPT}(S)$. A second lower bound on $\mathrm{OPT}(S)$ can be determined as follows: For each cycle c_i, let $w_i = w(c_i)$ and l_i denote the length of the longest string in c_i. By Lemma 2.9, if we consider the longest string in each cycle and merge them together optimally, the total amount of overlap will be at most $2\sum_{i=1}^{p} w_i$. So the resulting string will have length at least $\sum_{i=1}^{p} l_i - 2w_i$. Thus $\mathrm{OPT}(S) \geq \max(\sum_{i=1}^{p} w_i, \sum_{i=1}^{p} l_i - 2w_i)$.

The output string $\tilde{s}$ of algorithm Concat-Cycles has length at most $\sum_{i=1}^{p} l_i + w_i$ (Claim 2.4). So,

$$\begin{aligned} |\tilde{s}| &\leq \sum_{i=1}^{p} l_i + w_i \\ &= \sum_{i=1}^{p} l_i - 2w_i \; + \; \sum_{i=1}^{p} 3w_i \\ &\leq \mathrm{OPT}(S) \; + \; 3 \cdot \mathrm{OPT}(S) \\ &= 4 \cdot \mathrm{OPT}(S). \end{aligned}$$

□

We are now ready to present the algorithm MGREEDY, and show that it in fact mimics algorithm Concat-Cycles.

Algorithm MGREEDY

1. Let S be the input set of strings and T be empty.
2. While S is non-empty, do the following: Choose $s, t \in S$ (not necessarily distinct) such that $ov(s, t)$ is maximized, breaking ties arbitrarily. If $s \neq t$, then remove s and t from S and replace them with the merged string $\langle s, t \rangle$. If $s = t$, then just remove s from S and add it to T.
3. When S is empty, output the concatenation of the strings in T.

We can look at MGREEDY as choosing edges in the overlap graph ($V = S, E = V \times V, ov(,)$). When MGREEDY chooses strings s and t as having the maximum overlap (where t may equal s), it chooses the directed edge from *last*(s) to *first*(t) (see Claim 2.1). Thus, MGREEDY constructs/joins paths, and closes them into cycles, to end up with a collection of disjoint cycles $M \subset E$ that cover the vertices of G_S. We will call M the assignment created by MGREEDY. Now think of MGREEDY as taking a list of all the edges sorted in the decreasing order of their overlaps (resolving ties in some definite way), and going down the list deciding for each edge whether to include it or not. Let us say that an edge e *dominates* another edge f if e precedes f in this list and shares its head (or tail) with the head (or tail, respectively) of f. By the definition of MGREEDY, it includes an edge f if and only if it has not yet included an edge dominating f.

THEOREM **2.10** *The assignment created by algorithm MGREEDY is an optimal assignment.*

PROOF. Note that the overlap weight of an assignment and its distance weight add up to the total length of all strings. Accordingly, an assignment is optimal

(i.e., has minimum total weight in the distance graph) if and only if it has maximum total overlap. Among the maximum overlap assignments, let N be one that has the maximum number of edges in common with M. We shall show that $M = N$.

Suppose this is not the case, and let e be the edge of maximum overlap in the symmetric difference of M and N, with ties broken the same way as by MGREEDY. Suppose first that this edge is in $N \setminus M$. Since MGREEDY did not include e, it must have included another adjacent edge f that dominates e. Edge f cannot be in N (since N is an assignment), therefore f is in $M \setminus N$, contradicting our choice of the edge e. Suppose that $e = k \to j$ is in $M \setminus N$. The two N edges $i \to j$ and $k \to l$ that share head and tail with e are not in M, and thus are dominated by e. Since $ov(k, j) \geq \max\{ov(i, j), ov(k, l)\}$, by Lemma 2.7, $ov(i, j) + ov(k, l) \leq ov(k, j) + ov(i, l)$. Thus replacing in N these two edges with $e = k \to j$ and $i \to l$ would yield an assignment N' that has more edges in common with M and has no less overlap than N. This would contradict our choice of N. □

Since algorithm MGREEDY finds an optimal assignment, the string it produces is no longer than the string produced by algorithm Concat-Cycles. (In fact, it could be shorter since it breaks each cycle in the optimum position.)

2.4 Improving to $3 \cdot \mathrm{OPT}(S)$

Recall that in the last step of algorithm MGREEDY, we simply concatenate all the strings in set T without any compression. Intuitively, if we instead try to overlap the strings in T, we might be able to achieve a bound better than $4 \cdot \mathrm{OPT}(S)$. Let TGREEDY denote the algorithm that operates in the same way as MGREEDY except that in the last step, it merges the strings in T by running GREEDY on them. We can show that TGREEDY indeed achieves a better bound: it produces a superstring of length at most $3 \cdot \mathrm{OPT}(S)$.

THEOREM **2.11** *Algorithm TGREEDY produces a superstring of length at most* $3 \cdot OPT(S)$.

PROOF. Let $S = \{s_1, \ldots, s_m\}$ be a set of strings and s be the superstring obtained by TGREEDY on S. Let $n = \mathrm{OPT}(S)$ be the length of a shortest superstring of S. We show that $|s| \leq 3n$.

Let T be the set of all "self-overlapping" strings obtained by MGREEDY on S and C be the assignment created by MGREEDY. For each $x \in T$, let c_x denote the cycle in C corresponding to string x, and let $w_x = w(c_x)$ be its weight. For any set R of strings, define $||R|| = \sum_{x \in R} |x|$ to be the total length of the strings in set R. Also let $w = \sum_{x \in T} w_x$. Since $\mathrm{CYC}(G_S) \leq \mathrm{TSP}(G_S) \leq \mathrm{OPT}(S)$, we have $w \leq n$.

By Lemma 2.9, the compression achieved in a shortest superstring of T is less than $2w$, i.e., $||T|| - n_T \leq 2w$. By the results in [15, 16], we know that the compression achieved by GREEDY on set T is at least half the compression achieved in any superstring of T. That is,

$$||T|| - |s| \geq (||T|| - n_T)/2 = ||T|| - n_T - (||T|| - n_T)/2 \geq ||T|| - n_T - w.$$

So, $|s| \leq n_T + w$.

For each $x \in T$, let s_{i_x} be the string in cycle c_x that is a prefix of x. Let $S' = \{s_{i_x} | x \in T\}$, $n' = \mathrm{OPT}(S')$, $S'' = \{strings(c_x, i_x)s_{i_x} | x \in T\}$, and $n'' = \mathrm{OPT}(S'')$.

By Claim 2.4, a superstring for S'' is also a superstring for T, so $n_T \leq n''$, where $n_T = \mathrm{OPT}(T)$. For any permutation π on T, we have $|S''_\pi| \leq |S'_\pi| + \sum_{x \in T} w_x$, so $n'' \leq n' + w$, where S'_π and S''_π are the superstrings obtained by overlapping the members of S' and S'', respectively, in the order given by π. Observe that $S' \subseteq S$ implies $n' \leq n$. Summing up, we get

$$n_T \leq n'' \leq n' + w \leq n + w.$$

Combined with $|s| \leq n_T + w$, this gives $|s| \leq n + 2w \leq 3n$. □

2.5 GREEDY achieves linear approximation

One would expect that an analysis similar to that of MGREEDY would also work for the original GREEDY. This turns out not to be the case. The analysis of GREEDY is severely complicated by the fact that it continues processing the "self-overlapping" strings. MGREEDY was especially designed to avoid these complications, by separating such strings. Let $GREEDY(S)$ denote the length of the superstring produced by GREEDY on a set S. It is tempting to claim that

$$GREEDY(S \cup \{s\}) \leq GREEDY(S) + |s|.$$

If this were true, a simple argument would extend the $4 \cdot \mathrm{OPT}(S)$ result for MGREEDY to GREEDY. But the following counterexample disproves this seemingly innocent claim. Let

$$S = \{ca^m, a^{m+1}c^m, c^m b^{m+1}, b^m c\}, s = b^{m+1}a^{m+1}.$$

Now $GREEDY(S) = |ca^{m+1}c^m b^{m+1}c| = 3m + 4$, whereas $GREEDY(S \cup \{s\}) = |b^m c^m b^{m+1}a^{m+1}c^m a^m| = 6m + 2 > (3m + 4) + (2m + 2)$.

With a more complicated analysis we will nevertheless show that

THEOREM **2.12** *GREEDY produces a string of length at most* $4 \cdot OPT(S)$.

Before proving the theorem formally, we give a sketch of the basic idea behind the proof. If we want to relate the merges done by GREEDY to an optimal assignment, we have to keep track of what happens when GREEDY violates the maximum overlap principle, i.e. when some self-overlap is better than the overlap in GREEDY's merge. One thing to try is to charge GREEDY some extra cost that reflects that an optimal assignment on the new set of strings (with GREEDY's merge) may be somewhat longer than the optimal assignment on the former set (in which the self-overlapping string would form a cycle). If we could just bound these extra costs then we would have a bound for GREEDY. Unfortunately, this approach fails because the self-overlapping string may be merged by GREEDY into a larger string which itself becomes self-overlapping, and this nesting could go arbitrarily deep. Our proof concentrates on the innermost self-overlapping strings only. These so called culprits form a linear order in the final superstring. We avoid the complications of higher level self-overlaps

by splitting the analysis in two parts. In one part, we ignore all the original substrings that connect first to the right of a culprit. In the other part, we ignore all the original substrings that connect first to the left of a culprit. In each case, it becomes possible to bound the extra cost. This method yields a bound of $7 \cdot \mathrm{OPT}(S)$. By combining the two analyses in a more clever way, we can even eliminate the effect of the extra costs and obtain the same $4 \cdot \mathrm{OPT}(S)$ bound as we found for MGREEDY. A detailed formal proof follows.

We will need some notions and lemmas. Think of both GREEDY and MGREEDY as taking a list of all edges sorted by overlap, and going down the list deciding for each edge whether to include it or not. Call an edge *better* (*worse*) if it appears before (after) another in this list. Better edges have at least the overlap of worse ones. Recall that an edge dominates another iff it is better and shares its head or tail with the other one.

At the end, GREEDY has formed a Hamiltonian path

$$s_1 \to s_2 \to \cdots \to s_m$$

of 'greedy' edges. (w.l.o.g., the strings are renumbered to reflect their order in the superstring produced by GREEDY.) For convenience we will usually abbreviate s_i to i. GREEDY does not include an edge f iff

1. f is dominated by an already chosen edge e, or
2. f is not dominated but it would form a cycle.

Let us call the latter "bad back edges"; a bad back edge $f = j \to i$ necessarily has $i \leq j$. Each bad back edge $f = j \to i$ corresponds to a string $\langle s_i, s_{i+1}, \ldots, s_j \rangle$ that, at some point in the execution of GREEDY, has more (self) overlap than the pair that is merged. When GREEDY considers f, it has already chosen all (better) edges on the greedy path from i to j, but not yet the (worse) edges $i-1 \to i$ and $j \to j+1$. The bad back edge f is said to *span* the closed interval $I_f = [i, j]$. The above observations provide a proof of the following lemma.

LEMMA **2.13** *Let e and f be two bad back edges. The closed intervals I_e and I_f are either disjoint, or one contains the other. If $I_e \supset I_f$ then e is worse than f (thus, $ov(e) \leq ov(f)$).*

Thus, the intervals of the bad back edges are nested and bad back edges do not cross each other. *Culprits* are the minimal (innermost) such intervals. Each culprit $[i, j]$ corresponds to a *culprit string* $\langle s_i, s_{i+1}, \ldots, s_j \rangle$. Note that, because of the minimality of the culprits, if $f = j \to i$ is the back edge of a culprit $[i, j]$, and e is another bad back edge that shares head or tail with f, then $I_e \supset I_f$, and therefore f dominates e.

Call the worst edge between every two successive culprits on the greedy path a *weak link*. Note that weak links are also worse than all edges in the two adjacent culprits as well as their back edges. If we remove all the weak links, the greedy path is partitioned into a set of paths, called *blocks*. Every block consists of a nonempty culprit as the middle segment, and (possibly empty) left and right *extensions*. The set of strings (nodes) S is thus partitioned into three sets S_l, S_m, S_r of left, middle, and right strings. The example in Figure 2.3 has 7 substrings, of which 2 by itself and the merge of 4, 5, and 6

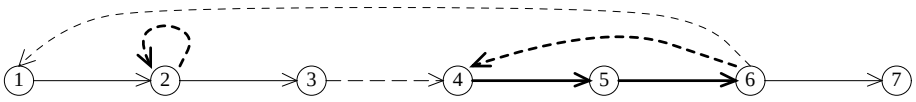


FIGURE 2.3. Culprits and weak links in Greedy merge path.

form the culprits (indicated by thicker lines). Bad back edges are $2 \to 2$, $6 \to 4$, and $6 \to 1$. The weak link $3 \to 4$ is the worst edge between culprits [2] and $[4, 5, 6]$. The blocks in this example are thus $[1, 2, 3]$ and $[4, 5, 6, 7]$, and we have $S_l = \{1\}, S_m = \{2, 4, 5, 6\}, S_r = \{3, 7\}$.

The following lemma shows that a bad back edge must be from a middle or right node to a middle or left node.

LEMMA **2.14** *Let $f = j \to i$ be a bad back edge. Node i is either a left node or the first node of a culprit. Node j is either a right node or the last node of a culprit.*

PROOF. Let $c = [k, l]$ be the leftmost culprit in I_f. Now either $i = k$ is the first node of c, or $i < k$ is in the left extension of c, or $i < k$ is in the right extension of the culprit c' to the left of c. In the latter case however, I_f includes the weak link, which by definition is worse than all edges between the culprits c' and c, including the edge $i - 1 \to i$. This contradicts the observation preceding Lemma 2.13. A similar argument holds for s_j. □

Let C_m be the assignment on the set S_m of middle strings (nodes) that has one cycle for each culprit, consisting of the greedy edges together with the back edge of the culprit. If we consider the application of the algorithm MGREEDY on the subset of strings S_m, it is easy to see that the algorithm will actually construct the assignment C_m. Theorem 2.10 then implies the following lemma.

LEMMA **2.15** *C_m is an optimal assignment on the set S_m of middle strings.*

Let the graph $G_l = (V_l, E_l)$ consist of the left/middle part of all blocks in the greedy path, i.e. $V_l = S_l \cup S_m$ and E_l is the set of non-weak greedy edges between nodes of V_l. Let M_l be a maximum overlap assignment on V_l, as created by MGREEDY on the ordered sublist of edges in $V_l \times V_l$. Let $V_r = S_m \cup S_r$, and define similarly the graph $G_r = (V_r, E_r)$ and the optimal assignment M_r on the right/middle strings. Let l_c be the sum of the lengths of all culprit strings. Define $l_l = \sum_{i \in S_l} d(s_i, s_{i+1})$ as the total length of all left extensions and $l_r = \sum_{i \in S_r} d(s_i^R, s_{i-1}^R)$ as the total length of all right extensions. (Here x^R denotes the reversal of string x.) The length of the string produced by GREEDY is $l_l + l_c + l_r - o_w$, where o_w is the summed block overlap (i.e. the sum of the overlaps of the weak links).

Denoting the overlap $\sum_{e \in E} ov(e)$ of a set of edges E as $ov(E)$, define the cost of a set of edges E on a set of strings (nodes) V as

$$cost(E) = ||V|| - ov(E).$$

Note that the distance plus overlap of a string s to another equals $|s|$. Because an assignment (e.g. M_l or M_r) has an edge from each node, its cost equals its distance weight. Since V_l and V_r are subsets of S and M_l and M_r are optimal

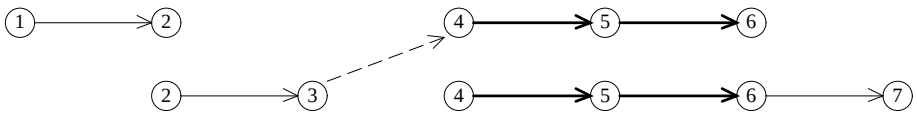


FIGURE 2.4. Left/middle and middle/right parts with weak links.

assignments, we have $cost(M_l) \leq n$ and $cost(M_r) \leq n$. For E_l and E_r we have that $cost(E_l) = l_l + l_c$ and $cost(E_r) = l_r + l_c$.

We have established the following (in)equalities:

$$\begin{aligned} l_l + l_c + l_r &= (l_l + l_c) + (l_c + l_r) - l_c \\ &= cost(E_l) + cost(E_r) - l_c \\ &= ||V_l|| - ov(E_l) + ||V_r|| - ov(E_r) - l_c \\ &= cost(M_l) + ov(M_l) - ov(E_l) + \\ &\quad cost(M_r) + ov(M_r) - ov(E_r) - l_c \\ &\leq 2n + ov(M_l) - ov(E_l) + ov(M_r) - ov(E_r) - l_c. \end{aligned}$$

We proceed by bounding the overlap differences in the above equation. Our basic idea is to charge the overlap of each edge of M to an edge of E or a weak link or the back edge of a culprit in a way such that every edge of E and every weak link is charged at most once and the back edge of each culprit is charged at most twice. This is achieved through combining the left/middle and middle/right parts carefully as shown below. For convenience, we will refer to the union operation for multisets (*i.e.*, allowing duplicates) as the *disjoint union*.

Let V be the disjoint union of V_l and V_r, let E be the disjoint union of E_l and E_r, and let $G = (V, E)$ be the disjoint union of G_l and G_r. Thus each string in $S_l \cup S_r$ occurs once, while each string in S_m occurs twice in G. We modify E to take advantage of the block overlaps. Add each weak link to E as an edge from the last node in the corresponding middle/right path of G_r to the first node of the corresponding left/middle path of G_l. This procedure yields a new set of edges E'. Its overlap equals $ov(E') = ov(E_l) + ov(E_r) + o_w$. A picture of (V, E') for our previous example is given in Figure 2.4.

Let M be the disjoint union of M_l and M_r, an assignment on graph G. Its overlap equals $ov(M) = ov(M_l) + ov(M_r)$. Every edge of M connects two V_l nodes or two V_r nodes; thus, all edges of M satisfy the hypothesis of the following lemma.

LEMMA **2.16** *Let N be any assignment on V. Let $e = t \rightarrow h$ be an edge of $N \setminus E'$ that is not in $V_r \times V_l$. Then e is dominated by either*

1. *an adjacent E' edge, or*
2. *a culprit's back edge with which it shares the head h and $h \in V_r$, or*
3. *a culprit's back edge with which it shares the tail t and $t \in V_l$.*

PROOF. Suppose first that e corresponds to a bad back edge. By Lemma 2.14, h corresponds to a left node or to the first node of a culprit. In the latter case, e is dominated by the back edge of the culprit (see the comment after Lemma

2.13). Therefore, either h is the first node of a culprit in V_r (and case 2 holds), or else $h \in V_l$. Similarly, either t is the last node of a culprit in V_l (and case 3 holds) or else $t \in V_r$. Since e is not in $V_r \times V_l$, it follows then that case 2 or case 3 holds. (Note that if e is in fact the back edge of some culprit, then both cases 2 and 3 hold.)

Suppose that e does not correspond to a bad back edge. Then e must be dominated by some greedy edge since it was not chosen by GREEDY. If the greedy edge dominating e is in E' then we have case 1. If it is not in E', then either h is the first node of a culprit in V_r or t is the last node of a culprit in V_l, and in both cases f is dominated by the back edge of the culprit. Thus, we have case 2 or 3. □

While Lemma 2.16 ensures that each edge of M is bounded in overlap, it may be that some edges of E' are double charged. We will modify M without decreasing its overlap and without invalidating Lemma 2.16 into an assignment M' such that each edge of E' is dominated by one of its adjacent M' edges.

LEMMA **2.17** *Let N be any assignment on V such that $N \setminus E'$ does not contain any edges in $V_r \times V_l$. Then there is an assignment N' on V satisfying the following properties.*

1. *$N' \setminus E'$ has also no edges in $V_r \times V_l$,*
2. *$ov(N') \geq ov(N)$,*
3. *each edge in $E' \setminus N'$ is dominated by one of its two adjacent N' edges.*

PROOF. Since N already has the first two properties, it suffices to argue that if N violates property 3, then we can construct another assignment N' that satisfies properties 1 and 2, and has more edges in common with E'.

Let $e = k \to j$ be an edge in $E' - N$ that dominates both adjacent N edges, $f = i \to j$, and $g = k \to l$. By Lemma 2.7, replacing edges f and g of N with e and $i \to l$ produces an assignment N' with at least as large overlap. To see that the new edge $i \to l$ of $N' \setminus E'$ is not in $V_r \times V_l$, observe that if $i \in V_r$ then $j \in V_r$ because of the edge $f = i \to j$ ($N \setminus E'$ does not have edges in $V_r \times V_l$), which implies that k is in V_r because of the E' edge $e = k \to j$ (E' does not have edges in $V_l \times V_r$), which implies that also $l \in V_r$ because of the N edge $g = k \to l$. □

PROOF. **(of Theorem 2.12.)** By Lemmas 2.16 and 2.17, we can construct from the assignment M another assignment M' with at least as large total overlap, and such that we can charge the overlap of each edge of M' to an edge of E' or to the back edge of a culprit. Every edge of E' is charged for at most one edge of M', while the back edge of each culprit is charged for at most two edges of M': for the M' edge entering the first culprit node in V_r and the edge coming out of the last culprit node in V_l. Therefore, $ov(M) \leq ov(M') \leq ov(E') + 2o_c$, where o_c is the summed overlap of all culprit back edges. Denote by w_c the summed weight of all culprit cycles, i.e., the weight of the (optimal) assignment C_m on S_m from Lemma 2.15. Then $l_c = w_c + o_c$. As in the proof of Theorem 2.8, we have $o_c - 2w_c \leq n$ and $w_c \leq n$. (Note that the overlap of a culprit back edge is less than the length of the longest string in the culprit cycle.) Putting

everything together, the string produced by GREEDY has length

$$\begin{aligned} l_l + l_c + l_r - o_w &\leq 2n + ov(M_l) - ov(E_l) + ov(M_r) - ov(E_r) - l_c - o_w \\ &\leq 2n + ov(M') - ov(E') - l_c \\ &\leq 2n + 2o_c - l_c \\ &= 2n + o_c - w_c \\ &\leq 3n + w_c \\ &\leq 4n. \end{aligned}$$

□

2.6 Which algorithm is the best?

Having proved various bounds for the algorithms GREEDY, MGREEDY, and TGREEDY, one may wonder what this implies about their relative performance. First of all we note that MGREEDY can never do better than TGREEDY since the latter applies the GREEDY algorithm to an intermediate set of strings that the former merely concatenates.

Does the $3n$ bound for TGREEDY then mean that it is the best of the three? This proves not always to be the case. In the example $\{c(ab)^k, (ab)^{k+1}a, (ba)^k c\}$, GREEDY produces the shortest superstring $c(ab)^{k+1}ac$ of length $n = 2k + 5$, whereas TGREEDY first separates the middle string to end up with something like $c(ab)^k ac(ab)^{k+1}a$ of length $4k + 6$.

Perhaps then GREEDY is always better than TGREEDY, despite the fact that we cannot prove as good an upper bound for it. This turns out not to be the case either, as shown by the following example. On input $\{cab^k, ab^k ab^k a, b^k dab^{k-1}\}$, TGREEDY separates the middle string, merges the other two, and next combines these to produce the shortest superstring $cab^k dab^k ab^k a$ of length $3k + 6$, whereas GREEDY merges the first two, leaving nothing better than $cab^k ab^k ab^k dab^{k-1}$ of length $4k + 5$.

Another greedy type of algorithm that may come to mind is one that arbitrarily picks any of the strings and then repeatedly merges on the right the string with maximum overlap. This algorithm, call it NAIVE, turns out to be disastrous on examples like

$$\{abcde, bcde\#a, cde\#a\#b, de\#a\#b\#c, e\#a\#b\#c\#d, \#a\#b\#c\#d\#e\}.$$

Instead of producing the optimal $abcde\#a\#b\#c\#d\#e$, NAIVE might produce $\#a\#b\#c\#d\#e\#a\#b\#c\#de\#a\#b\#cde\#a\#bcde\#abcde$ by picking $\#a\#b\#c\#d\#e$ as a starting point. It is clear that in this way superstrings may be produced whose length grows quadratically in the optimum length n.

2.7 Lower bound

We show here that the superstring problem is *MAX SNP-hard*. This implies that if there is a polynomial time approximation scheme for the superstring

problem, then there is one also for a wide class of optimization problems, including several variants of maximum satisfiability, the node cover and independent set problems in bounded-degree graphs, max cut, etc. This is considered rather unlikely.[1]

Let A, B be two optimization (maximization or minimization) problems. We say that A *L-reduces* (for *linearly reduces*) to B if there are two polynomial time algorithms f and g and constants α and $\beta > 0$ such that:

1. Given an instance a of A, algorithm f produces an instance b of B such that the cost of the optimum solution of b, $opt(b)$, is at most $\alpha \cdot opt(a)$, and
2. Given any solution y of b, algorithm g produces in polynomial time a solution x of a such that $|cost(x) - opt(a)| \leq \beta |cost(y) - opt(b)|$.

Some basic facts about L-reductions are: First, the composition of two L-reductions is also an L-reduction. Second, if problem A L-reduces to problem B and B can be approximated in polynomial time with relative error ϵ (i.e., within a factor of $1 + \epsilon$ or $1 - \epsilon$ depending on whether B is a minimization or maximization problem) then A can be approximated with relative error $\alpha\beta\epsilon$. In particular, if B has a polynomial time approximation scheme, then so does A. The class MAX SNP is a class of optimization problems defined syntactically in [11]. It is known that every problem in this class can be approximated within *some* constant factor. A problem is MAX SNP-hard if every problem in MAX SNP can be L-reduced to it.

THEOREM **2.18** *The superstring problem is MAX SNP-hard.*

PROOF. The reduction is from a special case of the TSP with triangle inequality. Let TSP(1,2) be the TSP restricted to instances where all the distances are either 1 or 2. We can consider an instance to this problem as being specified by a graph H; the edges of H are precisely those that have length 1 while the edges that are not in H have length 2. We need here the version of the TSP where we seek the shortest Hamiltonian path (instead of cycle), and, more importantly, we need the additional restriction that the graph H be of bounded degree (the precise bound is not important). It was shown in [12] that the TSP(1,2) problem (even for this restricted version) is MAX SNP-hard.

Let H be a graph of bounded degree D specifying an instance of TSP(1,2). The hardness result holds for both the symmetric and the asymmetric TSP (i.e., for both undirected and directed graphs H). We let H be a directed graph here. Without loss of generality, assume that each vertex of H has outdegree at least 2. The reduction is similar to the one of [5] used to show the NP-completeness of the superstring decision problem. We have to prove here that it is an L-reduction. For every vertex v of H we have two letters v and v'. In addition there is one more letter #. Corresponding to each vertex v we have a string $v\#v'$, called the *connector* for v. For each vertex v, enumerate the edges out of v in an arbitrary cyclic order as $(v, w_0), \ldots, (v, w_{d-1})$ (*). Corresponding to the ith edge (v, w_i) out of v we have a string $p_i(v) = v'w_{i-1}v'w_i$, where subscript arithmetic is modulo d. We will say that these strings are *associated* with v.

[1] In fact, Arora et al. [2] have recently shown that MAX SNP-hard problems do not have polynomial time approximation schemes, unless P = NP.

Let n be the number of vertices and m the number of edges of H. If all vertices have degree at most D then $m \leq Dn$. Let k be the minimum number of edges whose addition to H suffices to form a Hamiltonian path. Thus, the optimal cost of the TSP instance is $n - 1 + k$. We shall argue that the length of the shortest common superstring is $2m + 3n + k + 1$. It will follow then that the reduction is linear since m is linear in n.

Consider the distance-weighted graph G_S for this set of strings, and let G_2 be its subgraph with only edges of minimal weight (2). Clearly, G_2 has exactly one component for each vertex of H, which consists of a cycle of the associated p strings, and a connector that has an edge to each of them. We need only consider 'standard' superstrings in which all strings associated with some vertex form a subgraph of G_2, so that only the last p string has an outgoing edge of weight more than 2 (3 or 4). Namely, if some vertex fails this requirement, then at least two of its associated strings have outgoing edges of weight more than 2, thus we do not increase the length by putting all its p strings directly after its connector in a standard way. A standard superstring naturally corresponds to an ordering of vertices $v_1, v_2, \ldots, v_n$.

For the converse there remains a choice of which string q succeeds a connector $v_i \# v_i'$. If H has an edge from v_i to v_{i+1} and the 'next' edge out of v_i (in (*)) goes to, say v_j, then choosing $q = v_i' v_{i+1} v_i' v_j$ results in a weight of 3 on the edge from the last p string to the next connector $v_{i+1} \# v_{i+1}'$, whereas this weight would otherwise be 4. If H doesn't have this edge, then the choice of q doesn't matter. Let us call a superstring 'Standard' if in addition to being standard, it also satisfies this latter requirement for all vertices.

Now suppose that the addition of k edges to H gives a Hamiltonian path $v_1, v_2, \ldots, v_{n-1}, v_n$. Then we can construct a corresponding Standard superstring. If the out-degree of v_i is d_i, then its length will be $\sum_{i=1}^{n}(2 + 2d_i + 1) + k + 1 = 3n + 2m + k + 1$.

Conversely, suppose we are given a common superstring of length $3n + 2m + k + 1$. This can then be turned into a Standard superstring that is no longer. If $v_1, v_2, \ldots, v_n$ is the corresponding order of vertices, it follows that H cannot be missing more than k of the edges (v_i, v_{i+1}). □

Since the strings in the above L-reduction have bounded length (4), the reduction applies also to the maximization version of the superstring problem [15, 16]. That is, maximizing the total compression is also MAX SNP-hard.

2.8 Open problems

We end the chapter with several open questions raised from this research:

(1) Obtain an algorithm which achieves a performance better than 3 times the optimum.

(2) Prove or disprove the conjecture that GREEDY achieves 2 times the optimum.

Bibliography

[1] N. Alon, S. Cosares, D. Hochbaum, and R. Shamir. An algorithm for the detection and construction of Monge Sequences. *Linear Algebra and its Applications 114/115, 669–680, 1989.*

[2] A. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. *Proc. 33rd IEEE Symp. Found. Comp. Sci.*, 1992, 14-23.

[3] E. Barnes and A. Hoffman. On transportation problems with upper bounds on leading rectangles. *SIAM Journal on Algebraic and Discrete Methods 6, 487–496, 1985.*

[4] N. Fine and H. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.* 16, 1965, 109-114.

[5] J. Gallant, D. Maier, J. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences 20, 50–58, 1980.*

[6] M. Garey and D. Johnson. Computers and Intractability. *Freeman, New York, 1979.*

[7] A. Hoffman. On simple transportation problems. *Convexity: Proceedings of symposia in pure mathematics, Vol 7, American Mathematical Society, 317–327, 1963.*

[8] A. Lesk (Edited). Computational Molecular Biology, Sources and Methods for Sequence Analysis. *Oxford University Press, 1988.*

[9] M. Li. Towards a DNA sequencing theory. *31st IEEE Symp. on Foundations of Computer Science, 125–134, 1990.*

[10] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[11] C. Papadimitriou and M Yannakakis. Optimization, approximation and complexity classes. *20th ACM Symp. on Theory of Computing*, 229–234, 1988.

[12] C. Papadimitriou and M Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research.* To appear.

[13] H. Peltola, H. Soderlund, J. Tarhio, and E. Ukkonen. Algorithms for some string matching problems arising in molecular genetics. *Information Processing 83 (Proc. IFIP Congress, 1983) 53–64.*

[14] J. Storer. *Data compression: methods and theory. Computer Science Press, 1988.*

[15] J. Tarhio and E. Ukkonen. A Greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science 57 131–145 1988*

[16] J. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation 83 1–20 1989.*

[17] L. G. Valiant. A Theory of the Learnable. *Comm. ACM* 27(11) 1134–1142 1984.

3

On Labyrinth Problems and Flood-Filling.

3.1 Introduction

Consider a property that pixels in a frame buffer may or may not possess, such as having a particular color, or, more generally, a color from a selected range. We can identify regions of connected pixels that share this property. Each region is delimited by a set of pixels lacking the property, constituting its *boundary*. We are concerned with *4-connected*[1] regions, where each pixel connects to its 4 horizontally and vertically adjacent neighbours. In contrast, the boundary of a region consists of a number of 8-connected components, one external component, and zero or more internal ones. Given a designated region, a filling algorithm is to apply to each pixel in the region a certain operation, such as setting the color to a specified value, exactly once.

3.1.1 Properties and Operations

There is one, very important, restriction we put on the possible combination of property and operation, namely that the operation **invalidates** the property. So, the operation "make green" is allowed for the property "red", but not for the property "red or green".

We pose this restriction with the aim of finding a constant space filling algorithm. To see why this is necessary, assume there is a third colour, blue, and consider some arbitrary green region with a blue boundary. With a property "red or green" defining the regions, the operation "make green" is of course ineffective on our green-only region. If there had been red pixels in the region however, they would need to be found by the filling algorithm and colored green. This reduces the task of the filling algorithm to one of visiting all the

[1]Note that we diverge here from the graph-theoretic interpretation of having so many disjoint paths between any 2 points.

green pixels and detecting termination. Now, since the question of whether this can be done using less than a linear amount of space is known to be a hard open problem, there is little hope of finding an algorithm for unrestricted filling using only a constant amount of memory.

Furthermore, is it widely believed that no finite "bug" automaton[2] (possibly equipped with a fixed number of markers) can traverse all nodes of an arbitrary planar graph embedded in a grid, let alone detect the termination of its traversal. This means that the restriction we impose on the filling operation is probably vital for a constant space solution.

It may seem from the above that such problems arise only if the number of colors exceeds 2, but this is not necessarily the case. Consider a so called *pattern-fill*, where the operation depends on the pixel's coordinates. A simple example is a monochrome image, where the property is "white", and the operation is to make the pixel black iff both its x and y coordinate are even. In other words, change the color of a white region into what looks like light gray. Using a 2 by 2 square of 4 pixels, 1 black and 3 white, as a building block, it is clear that we can again construct arbitrarily shaped regions where the required operation is always ineffective.

As a result of the restriction, pixels which have been operated on will be indistinguishable from the boundary. In effect, the boundary can be said to expand into the region.

3.1.2 Terminology

From now, we will abstract from the details of the distinguishing property and desired operation by assuming the existence of the following two functions:

get() a boolean function evaluating the property for the pixel whose address it is given, and

set() a procedure that performs the operation on the pixel whose address it is given.

Let R, at any time, denote the set of pixels that remain to be filled, The boundary of R, denoted $b(R)$ is just the set of pixels in $\bar{R}$ that neighbour a pixel in R. We assume w.l.o.g. that an FFA applies get() only to pixels in $R \cup b(R)$, i.e. it will not cross the boundary. On those in R it returns true, and on those in $b(R)$ it returns false.

The existing algorithms discussed here are, from a technical viewpoint, *propagation* algorithms. They start from a given seed pixel and explore the entire region containing it; propagating, as it were, from the seed pixel to the region's boundary. In the formulation of Fishkin and Barsky [1], 'filling is the composite process of finding and SETting the pixels in the region.' This distinction seems a natural one for familiar FFAs, which can be written purely in terms of get() to serve as algorithms that merely traverse a designated region. The correct propagation of our algorithm however very much depends on the SETting of pixels. Therefore, we will refrain from the above compositional view and simply refer to all algorithms discussed here as FFAs.

---

[2]Imagine a finite automaton as a bug walking in a graph, at each step seeing which of the 4 direction are possible.

We are interested in the *space complexity* and, to a lesser extent, the *time complexity* of FFAs. These are defined to be the maximum space (respectively time) used for filling a region of area n (that is, consisting of n pixels).

For the moment we assume a RAM-like machine model in which pixel-addresses fit in a fixed number of machine words. This is a reasonable assumption for all practical purposes where the regions will certainly be smaller than 2^{32} by 2^{32} pixels.

In the next section, we will present two straightforward examples of well known FFAs and show their space complexity to be linear in the area of the filled region.

3.2 Depth First Filling

Consider the following recursive flood fill algorithm:

```
df_fill(p)
{
  if (!get(p))          /* not in region */
    return;
  set(p);
  df_fill(p-DX);
  df_fill(p+DX);
  df_fill(p-DY);
  df_fill(p+DY);
}
```

For conciseness, we have flattened the 2 dimensions of the pixel plane into one linear address space; using two different offsets, DX and DY, to move to horizontally, resp. vertically adjacent neighbours. This closely resembles the way in which pixels are commonly stored in computer memory. Furthermore, let us call the negative x-direction "left", the positive x-direction "right", the negative y-direction "up", and the positive y-direction "down".

Most of the fill algorithms in use today are more or less sophisticated variations on this simple procedure. See Fishkin and Barsky [1] for a comprehensive survey of such algorithms. They use as a measure of speed the average number of times a pixel in the region is visited. Clearly, for the above simple FFA, every pixel properly inside the filled region is visited once from each side (4 times) and those neighbouring the boundary 1, 2, or 3 times. The pixels on the boundary itself are also visited of course, but for simplicities' sake, are excluded from the above average. Let us mention that the new FFA proposed in [1] achieves an average of only 1.05 visits per pixel.

Now let's look at the memory usage. While there are no explicit data structures, it takes a lot of stack space to remember all the recursive invocations. Assume that the seed pixel is in the bottom-right corner of a rectangular region. The filling then propagates first to the left boundary, then one pixel up, then back to the right boundary, and so on. Since back-tracking occurs only on the original rectangle boundary, the depth of the recursion can be seen to equal the area of the region, hence the memory usage is proportional to that. Permuting the order of recursive calls doesn't help; for any order there will

be two opposite directions that both come before a third, like left and right come before up in the ordering above, and an appropriate starting point in a rectangle will thus show the same behaviour for the permuted order fill.

In conclusion: recursive or stack-based FFAs have linear space complexity.

3.3 Breadth First Filling

In the following algorithm the stack implicit in the previous algorithm is replaced by a first-in, first-out queue.

```
bf_fill(p)
{
  enque(p);
  while (not_empty()) {
    p = deque();
    if (get(p)) {
      set(p);
      enque(p-DX);
      enque(p+DX);
      enque(p-DY);
      enque(p+DY);
    }
  }
}
```

This algorithm is a big improvement over the previous one in terms of memory-usage, at least in practice. It will fill the rectangle or for that matter any convex region using $O(\sqrt{n})$ space. This is because the fill tends to expand in a diamond-shaped form, where only pixels in or neighbouring the outermost pixel-layer of the diamond are in the queue.

It is tempting to conjecture that the space complexity of the breadth first fill is in fact $O(\sqrt{n})$. In trying to prove this, one runs into difficulties with regions where the diamond is not just expanding away from the seed pixel but also has to move back towards it.

The picture in Figure 3.1 was constructed to exhibit this behaviour in the extreme. Consider changing the white region inside the big diamond to black. Starting from the center, the fill will spread out evenly in all four directions, then split again fourfold, and after one more split will arrive, almost simultaneously, at the 64 'outlets'.

From there 64 small diamonds will develop in parallel, the total circumference of which is on the order of the area of the region. This shows breadth-first (queue based) FFAs, too, suffer from a linear worst case space complexity.

3.4 The price of disconnecting

If we trace the need for using all that memory in stack and queue based FFAs, we see that the neighbours of a just-filled pixel are remembered to ensure that the regions those neighbours are part of, ultimately get filled too. Recall that

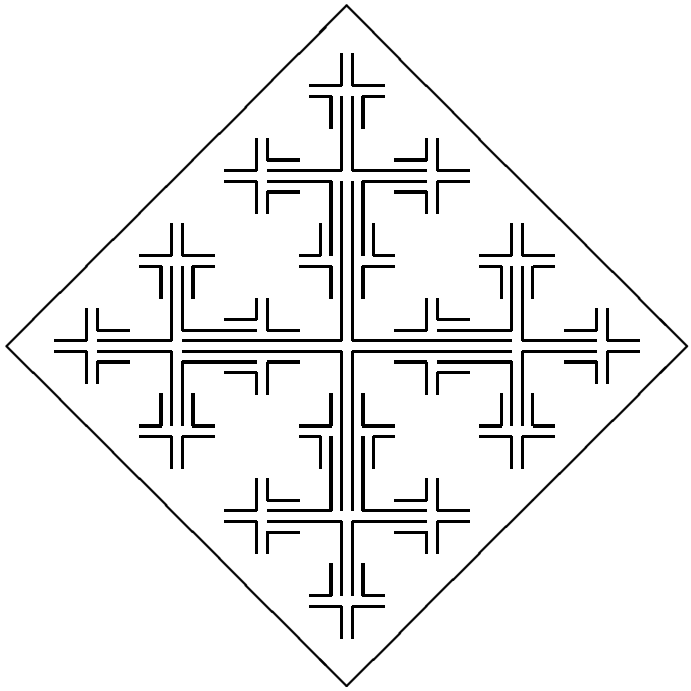


FIGURE 3.1. A worst case example region for breadth first fill.

R, at any time, denotes the set of pixels that remain to be filled. Clearly, if R consists of c components, then at least c different pixels must be remembered, one in each component, in order to be able to complete filling all of R. Whenever a pixel is filled that is a cutting point of its component (a cutting point in a connected graph is a point whose removal leaves more than one component), additional pixels must be remembered. On the other hand, if we only fill pixels that leave R connected, then we just move to any of its unfilled neighbours and continue, without having to remember any pixel other than where we are. This reduces the problem of filling to that of finding a non-cutting pixel. If we can find such a pixel in time $T(n)$, then filling all the pixels in R can be done in time $nT(n)$ with no additional storage.

3.5 On planar embeddings

Let's consider the problem in a slightly different, more general, setting. A planar embedding of a (planar) graph can be succinctly represented by a triple (V, H, r). V is the set of vertices in the graph. $H \subseteq V \times V$ is the set of *half-edges*. A half-edge $h = (v, w)$ is that half of the edge $\{v, w\}$ which is adjacent to v. Of course, we require that H be symmetric; a half edge (v, w) cannot exist without a corresponding other half (w, v). Finally, $r : H \to H$ is a function that orders all half-edges around each vertex in a clock-wise manner. If the half-edges adjacent to a vertex v of degree d are, in clock-wise order, $h_0, h_1, \ldots, h_{d-1}$, then

$r(h_i) = h_{(i+1) \bmod d}$. Half-edges adjacent to the same vertex are also said to be adjacent. Note that the vertices are in one-one correspondence to the *orbits* of r. Let us call such a triple (V, H, r) a *labyrinth*.

We introduce some more terminology that will be useful later. Labyrinth $L' = (V', H', r')$ is a *sublabyrinth* of $L = (V, H, r)$ iff $V' \subseteq V$, $H' \subseteq H \cap V' \times V'$, and $\forall h \in H' : r'(h) = r^i(h), i > 0$ minimal such that $r^i(h) \in H'$. When $L = (V, H, r)$ is a labyrinth and $V' \subseteq V$, then $L|V'$ is the sublabyrinth (V', H', r') with $H' = H \cap V' \times V'$. Also, with $v \in V$, $L \setminus v$ denotes $L|(V - \{v\})$. An *edge* in a labyrinth is a set of two corresponding half-edges $\{(v, w), (w, v)\}$. A *path* is defined in the natural way. A labyrinth is *connected* if for any distinct vertices v, w there is a path from v to w. A *connected component*, or just component, of a labyrinth $L = (V, H, r)$, is a maximal connected sublabyrinth $L' = (V', H', r')$. For a half-edge $h = (v, w)$ in labyrinth $L = (V, H, r)$, $C(h)$ denotes the connected component of $L \setminus v$ containing w.

3.5.1 Exploring the labyrinth

An *explorer* is placed in the labyrinth on an arbitrary half-edge, equipped with a finite set of distinct markers. At any time, movement is possible in three directions

right Move from a half-edge h to $r(h)$.

left Move from a half-edge h to $r^{-1}(h)$.

cross Move from a half-edge (v, w) to its counterpart (w, v).

Marker manipulation is achieved through the primitive operations **drop** marker (on current half-edge), **pick** marker (from current or adjacent half-edge), and the tests **carrying** marker, and **here** marker (is marker on current half-edge?). Note that markers are placed at half-edges, not vertices. We further allow some more involved operations whose implementation in terms of the above primitives is straightforward. Of these, the **near** test checks whether any adjacent half-edge contains a given marker, the **replace** operation picks up a given marker at an adjacent half-edge to drop another marker in its place, and the **goto** operation moves to an adjacent half-edge containing a given marker. Finally, a **halt** operation is available to signal the completion of a task.

The explorer is a finite automaton expressed as a procedure with no variables other than a fixed set of markers.

3.5.2 Finding a non-cutting point

The problem we're interested in is that of finding a non-cutting vertex. The solution is an explorer which, started on an any initial half-edge of an arbitrary but finite labyrinth, always halts in a finite number of steps on a half-edge adjacent to a non-cutting vertex. Our proposed initial solution is:

```
markers last,out
loop
   if carrying last then
     drop-left last; drop out
   elsif near last then
        if at-left out then
           move-here out
           if here last then
              halt
           endif
        else left; pick last,out
        endif
   else left
   endif
   cross
end
```

The following discussion serves both as an explanation of the workings of the explorer, and as a proof of its correctness.

What the explorer does is to perform a series of *connectivity tests* (con-test for short) $T_0, T_1, \ldots$ at vertices $v_0, v_1, \ldots$. Connectivity test T_i determines whether vertex v_i is a cutting point or not. At the start of T_i, marker *last* is dropped on a half-edge adjacent to v_i. If the con-test fails, i.e. v_i is a cutting point, then *last* is picked up and dropped at a neighbouring vertex v_{i+1}, where the process is repeated. Let h_0 be the half-edge on which explorer is initially placed and $h_{i+1} = (v_{i+1}, v_i)$ be the half-edge on which the explorer arrives at v_{i+1} for con-test T_{i+1}.

We first consider what happens when *last* is dropped at a non-cutting point $v = v_i$. This case is illustrated in Figure 3.2(a). Since removal of v leaves the labyrinth connected, there are paths connecting all of the neighbours of v in $L \setminus v$, indicated by the dotted lines. These paths form, in this case, 3 faces F_1, F_2, F_3, one between every 2 adjacent half-edges of v (in case v lies on the boundary of the labyrinth, then one of these faces is the infinite external face.) Suppose h_i, the half-edge on which explorer arrives at v, is the bottom one. Explorer will drop marker *last* at the next left half-edge, and marker *out* at h_0. Then it will cross, i.e. leave v through the *out* edge. Since it's no longer carrying *last*, it will just alternate **left** and **cross** until it gets back to v. That is, it will follow a *right-hand walk* along the boundary of face F_1, returning to v on the half-edge right of *out*. Next, the tests **near** *last* and **at-left** *out* will succeed (the at-left is of course from the viewpoint of v), and *out* will move one half-edge to the right. This process will repeat until *out* moves to *halt*, at which point explorer will halt. This shows that non-cutting points are correctly identified.

Next consider the case where $v = v_i$ is a cutting point, illustrated in Figure 3.2(b). So $L \setminus v$ is not connected and consists of multiple components. Again suppose that h_i is the bottom half-edge. The dotted lines indicate that $C(h_i)$ contains two more neighbours of v. Each other component of $L \setminus v$ is internal to one of the faces determined by $C(h_i)$. Our explorer starting from h_i, will traverse faces just as in case (a), until a non-empty face is traversed. In (b) this is when the top face has just been traversed and explorer returns to v.

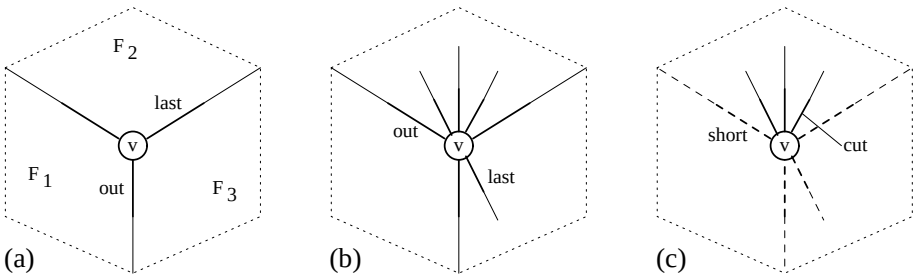


FIGURE 3.2. Connectivity tests

Now the test **near** *last* will succeed but **at-left** *out* will fail. There are other half-edges between the one where explorer returned to v and the one where it left v (*out*), constituting one or more components different from $C(h_i)$. The explorer now picks up *last* and crosses the half-edge on its left, moving to v_{i+1} to start another con-test.

3.5.3 Termination

For convenience, let's write C_i for $C(h_i)$. The crucial observation is that C_i is a proper sublabyrinth of C_{i+1}. Since the labyrinth is finite, so is the sequence of nested components:

$$C_1 \subset C_2 \subset \cdots \subset C_r,$$

with $r \leq n = |V|$, and T_r is successful. This proves termination of the explorer.

3.5.4 Improving Time Complexity

Next let's consider the time complexity. Induction easily shows that during T_i, only C_i is visited in addition to vertex v_i itself. Furthermore, each half-edge is visited exactly once for each adjacent face traversed during T_i, hence at most twice. Together with the fact that the number of edges in a planar graph is linear in the number of vertices, this gives an upper bound of $O(n^2)$.

But we can do better. Consider the following extension to our procedure.

```
markers last,out,short,cut
loop
   if carrying last then
     drop-left last; drop out
   elsif near last then
        if at-left out then
           move-here out
           if here last then
              halt
           endif
        else replace out by short; left; drop cut; pick last
        endif
   elsif here short goto cut
   else left
   endif
   cross
end
```

When a con-test T_i fails, a marker *short* is left in place of *out* and another marker *cut* is dropped on the half-edge leading to v_{i+1}. (the **replace** operation includes a possible excursion to the neighbour where these two markers were dropped earlier in order to pick them up.) Refer to Figure 3.2(c). During T_{i+1}, the only way for our initial explorer to traverse component C_i again is to arrive at v on the half-edge right of *short*, and leave v over *short* itself. It would then follow the exact same path that also led to T_i failing. The new explorer short-circuits this by detecting the presence of *short* and proceeding immediately to *cut*. This only speeds up the con-test and doesn't invalidate its correctness. The effect of this shortcut is the same as if all half-edges from *short* up to *cut* were removed, as indicated by the dashed lines. With the shortcut mechanism, T_{i+1} only visits $C_{i+1} \setminus C_i$, plus v_i. Hence, successive con-tests only overlap in single points, and using the same planarity argument, we obtain an $O(n)$ upper bound on the time complexity of the improved explorer.

3.6 The constant space FFA

Having solved the subproblem of finding a non-cutting point, we now return to flood filling. The above explorer is easily translated into an FFA. The current position as well as each marker is represented by a pair (p, d) of a pixel address and direction. Carried markers have a special value. At the start of the loop, all 4 neighbouring pixels are tested with `get()`. If the number of neighbours belonging to the region is 0, then the FFA is finished after setting the current pixel. If the number is 1 then a slight optimization is possible; since the current pixel has only one neighbour in the region it is a non-cutting point and can be set, without even disturbing the current con-test (but pick up the *last* marker if it happens to be located at the current pixel). Otherwise, the FFA behaves similar to the labyrinth explorer. It uses constant space, and, since it can find no more than n non-cutting points, runs in time $O(n^2)$.

3.7 Conclusion

The memory-hunger of existing flood-fill algorithms is not an unavoidable evil. At the cost of increasing the worst-case running time by a factor n, memory-use can be reduced to a constant: a reduction by the same factor n. An interesting open problem is whether $\Theta(n^2)$ is a lower bound on the space-time complexity product for FFAs and if so, whether this product can also be achieved by FFAs with complexities strictly between these two extremes.

Bibliography

[1] K.P. Fishkin, B.A. Barsky, *An Analysis and Algorithm for Filling Propagation*, Proceedings of Graphics Interface '85", pp. 203–212.

[2] Armin Hemmerling, *Labyrinth Problems: Labyrinth-Searching Abilities of Automata*, Teubner-Texte zur Mathematik; 114.

4

The Energy Complexity of Threshold and other functions.

In this chapter we obtain upper bounds on the switching energy needed to compute threshold and counting functions with VLSI circuits.

Switching energy is theoretically interesting because it is believed [14] to be intrinsic to computation and a fundamental complexity measure of VLSI computations. Energy is practically motivated in VLSI design because energy consumed by a circuit is transformed into heat. How well a circuit can dissipate heat determines its operational limitations. Thus, the less heat produced the better. Further, energy considerations determine a significant portion of the overall costs of a computer [15].

Common to all physical devices is the switching energy [14] consumed when a wire or gate changes state from 1 to 0 or vice versa. The amount of switching energy consumed is proportional to the area switched.

The results in this chapter are obtained in the *Uniswitch Model* (USM) of energy consumption, described in the next section. USM was first defined by Kissin [9] and has become the primary model for the asymptotic analysis of switching energy ([12], [16], [18], [1]). Kissin [8] also described the first energy saving design technique, by obtaining energy-efficient circuits for *OR*, *AND*, *Compare* and *Addition* functions. Lengauer and Mehlhorn [13] showed that n-input functions realizable in $AT^2 = O(n^2)$ require $\Omega(AT)$ switching energy, where A is area and T is time in the Thompson model [17]. Aggarwal et al [1] improved the result of Lengauer and Mehlhorn to obtain an $\Omega(n^2)$ energy bound for the class of *transitive* functions [19]. Leo [12] showed that, for a specialized circuit basis, the parity function requires $\Omega(A)$ average switching energy, where A is the area of the parity circuit. Tyagi [18] studied the average energy consumption of logic level structures such as Programmable Logic Arrays (PLA).

USM measures the differences between two stable states of a circuit. Race conditions (aka hazards) are neglected; they are the domain of the *Multiswitch Models*, which are defined and discussed in [7] and [6]. USM provides a lower

bound on the total energy consumed by a circuit.

The rest of this chapter is organized as follows. Section 4.1 describes and motivates the Uniswitch Model of energy consumption, aka *uniswitch energy*. (The term *energy* refers to switching energy for the duration of this chapter). The definition of a circuit is extended to include wires with bandwidth greater than 1. In Section 4.2, upper bounds are obtained in USM. In particular, a fast VLSI circuit is described for count-to-k functions, which is optimal in consuming $O(n)$ worst case uniswitch energy. The count-to-k construction also yields bounds on k-threshold functions. Conclusions follow in Section 4.3.

4.1 The setting

The Uniswitch Model of energy consumption defines an energy cost measure for VLSI circuits. USM measures the differences between states of a circuit. The following discussion sets the stage for a precise definition of USM.

A *VLSI circuit* is a combinational circuit [2] embedded in a plane as in [4]. Salient assumptions of the VLSI circuit model that are important to USM are as follows. A circuit is an acyclic directed graph of gates (nodes) and wires (edges). The number of input (fanin 0) nodes is denoted n. The maximum number of edges on a path from an input node to an output (fanout 0) node is called the *depth* of the circuit. A 'monowire' in a VLSI circuit has bandwidth 1 and width λ (λ is the standard name for *line width*). A wire has some bandwidth $b \leq B$ and width $2\lambda b - 1$, consisting of b parallel monowires separated at distance λ. We use B to denote a maximum bandwidth. This nonstandard notion of wire makes it easier to deal with values in a range $0, 1, 2, \ldots, k$ with $k < 2^B$. At most a constant number of wires (at least 2) can overlap at any point in a VLSI circuit. A node has constant area at least λ^2, bounded fanin (number of input wires), and bounded fanout (number of output wires). A non-input node computes a function of the values on its input wires in constant time (see discussion below). Gates are separated by distances of at least λ.

The k-threshold circuit described in this chapter uses a circuit basis that includes addition, subtraction, minimum, and comparison functions of $2 \leq B$-bit numbers, where $k < 2^B$ are constants. These functions can be decomposed into standard boolean subcircuits that use only a constant amount of resources. In the analysis that follows, node complexity is therefore neglected.

Write a circuit C as $C = (V, W)$, where V is the set of nodes and W the set of wires. A *legal state*, (hereafter also called *state* or *stable state*) of a circuit $C = (V, W)$, is a function $V \cup W \rightarrow \{0, 1\}^{\leq B}$, that attributes consistent values to the nodes and wires of C. That is, the value of a node equals that of all outgoing wires, and the value of a non-input node equals the corresponding function of the values on the input wires. Numbers are equated with their binary representations. Obviously, the value of any node or wire in a stable state is completely determined by the input vector. For a node v (wire w) and an input vector x, let $v(x)$ ($w(x)$) denote the value of the node (wire) in the stable state induced by input x.

The switching energy of a circuit C is defined as a distance metric on the set of 2^n possible inputs vector. We are interested in what happens when one input vector x to C is replaced by another input vector y. A wire w for which

$w(x) \neq w(y)$ is said to *switch*. Let $h_{w,x,y}$ be the Hamming distance between $w(x)$ and $w(y)$. A wire w of length l_w that switches, involves a switching area of $h_{w,x,y}A_w$, where $A_w = l_w\lambda$ is the area of one of the constituent monowires of w. Letting p be the area accounting for 1 unit of *switching energy*, this accounts for A/p switching energy. Similarly, a node v of area A that switches ($v(x) \neq v(y)$), accounts for A/p switching energy (for convenience we assume that all of the area of the node is involved in the switching).

The total switching energy consumed by C when switching from input x to input y is

$$E(C,x,y) = (\sum_{w\in W} h_{w,x,y}A_w + \sum_{v\in V, v(x)\neq v(y)} A_v)/p.$$

Obviously, $E(C,x,x) = 0$, $E(C,x,y) = E(C,y,x)$, and $E(C,x,z) \leq E(C,x,y) + E(C,y,z)$, so that $E(C)$ is indeed a distance metric. Also, $E(C)$ is bounded by A_C/p, with A_C the total area of C.

The *worst case uniswitch energy* of a circuit C is defined as

$$E_{worst}(C) = \max_{x,y\in\{0,1\}^n} E(C,x,y),$$

while the *average case uniswitch energy* is defined as

$$E_{avg}(C) = \sum_{x,y\in\{0,1\}^n} E(C,x,y)/2^{2n},$$

where 2^{2n} is the number of possible input pairs (x,y). This definition of $E_{avg}(C)$ assumes that the input vector is distributed uniformly over $\{0,1\}^n$.

A function $f : \{0,1\}^* \to \{0,1\}^*$ is *energy efficient* iff there exists a family $C = (C_n)_{(n\in\mathbf{N})}$ of circuits with C_n realizing $f|\{0,1\}^n$ in $O(\log n)$ depth, such that $E_{worst}(C_n) = O(n)$.

Throughout this chapter, $\log n$ means $\log_2 n$.

4.1.1 Model Motivation

The intent of this section is to motivate the *Uniswitch Model* in light of physical considerations. USM is a good model for obtaining lower bounds because it conservatively estimates a circuit's switching behaviour. Thus, a lower bound in USM is an equally valid lower bound on *multiswitch* energy.

USM takes no notice of how a circuit arrives at a particular state. This is the domain of the *Multiswitch Models*, which are discussed in [7] and [6]. However, in order to discuss the relevance of using USM to obtain upper bounds, the following multiswitch notions are introduced.

The switching behaviour of physical circuits is influenced by various delay functions, such as gate delay δ, wire delay Δ and input delay I. δ determines the switching speed of a gate. Δ determines the time to transmit bits along a wire. I determines when an input value arrives at an input port.

Let (C_n, δ, Δ, I) denote a *circuit scheme*, where C_n is a VLSI circuit with gate delay δ, wire delay Δ, and input delay I. A circuit scheme (C_n, δ, Δ, I) exhibits the *uniswitch property* if each node or wire of C_n switches at most once when C_n changes from one input setting to another, according to δ, Δ and I. Otherwise, (C_n, δ, Δ, I) exhibits the multiswitch property.

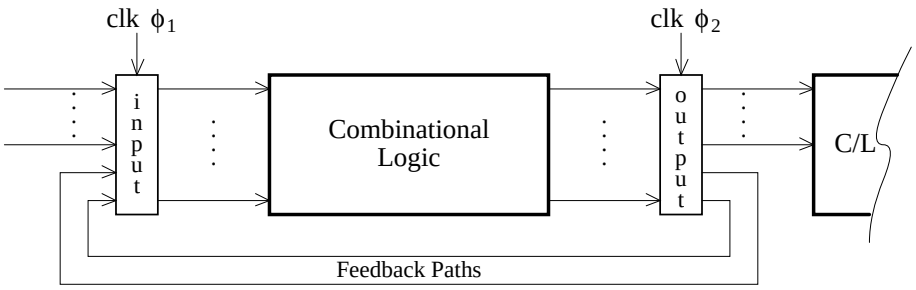


FIGURE 4.1. Preferred Sequential Circuit: a Finite State Machine

Using USM to obtain upper bounds is justified for circuit schemes that exhibit the uniswitch property. For example, if each node of a circuit receives its inputs at the same time, race conditions cannot arise. The uniswitch property is thus ensured. Some real circuits have this timing property. Where race conditions derive solely from a circuit's asynchrony (i.e.. the paths to a node vary in length), a circuit scheme can acquire the uniswitch property if the circuit can be made synchronous. A "bad" input schedule can be offset by varying gate delays. These approaches to designing circuit schemes that achieve the uniswitch property are discussed in [7] and [6]. Further, according to C. Mead [15], many CMOS designs are synchronized to ensure that the corresponding circuit schemes have the uniswitch property.

USM is the first step in the systematic asymptotic analysis of switching energy consumption in VLSI circuits. As such, USM is justified as an upper bound model. In addition, USM is motivated by designers' practical efforts to prevent hazards and thus ensure the uniswitch property. USM is used for upper bound analysis in [18] and [1].

USM is defined for acyclic circuits. The study of combinational circuits (without loops) has a long and distinguished history. Krohn and Rhodes' [10, 11] seminal work in this area showed that each sequential machine (with loops) can be decomposed into structures consisting of only combinational circuits and flip-flops.

A recommended architecture for sequential machines is the finite state machine in which the combinational logic is isolated from the looping structure [14]. See Figure 4.1. This architecture lends itself to analysis of the combinational logic distinct from the looping buffers.

4.2 Worst case upper bounds

4.2.1 Energy-Efficient k-Threshold Circuit

In this section, a novel energy-efficient threshold circuit is described, which generalizes the techniques of the energy-efficient *OR* and *AND* circuits described in [8] ,[5]. The energy-efficient *OR* circuit used the observation that it is sufficient to turn on one *OR* input to turn on the output. Thus, when many inputs to *SOR* (Smart *OR* circuit) are "1", all but one of these "1" signals are 'killed'. In a completely analogous manner, it is sufficient to turn off one *AND* input

in order to turn off the output. Thus, when many inputs are turned off, only one 0 signal must propagate all the way to the output. The principle idea is that the input may provide more information than the function requires, and suppressing unneeded input bits results in energy savings. This idea was also used in [8] to design energy-efficient comparator circuits.

A threshold function, T_k, is defined on a boolean vector $x = x_1x_2 \ldots x_n$ as follows:

$$T_k(x) = \begin{cases} 1 & \text{if } \sum x_i \geq k \\ 0 & \text{otherwise} \end{cases}$$

Hence, at most k input bits that are "1" must reach the output; the rest can be "killed". The energy-efficient threshold circuit, Thr_k, described below, effectively uses only necessary information, killing off the rest, resulting in only linear worst case switching energy consumption.

Let's define a sum-to-k function on $x = x_1x_2 \ldots x_n$ as:

$$S_k(x) = \min(k, \sum x_i).$$

Circuit Thr_k consist of a comparator bolted onto a circuit Cnt_k, which uses $(k+1)$-ary logic to compute S_k. The comparator compares the output of Cnt_k with the constant k. Circuit Cnt_k contains 2 types of nodes: +-nodes that sum the inputs, and min-nodes that 'kill' inputs exceeding k.

Like the *SOR* circuit, the Cnt_k circuit is laid out so that, for any input, the area of non-zero wires is at most linear in the input size.

The following recurrence describes the boolean sum-to-k function $S_k|\{0,1\}^{2n}$ in terms of $S_k|\{0,1\}^n$, addition, and minimum.

$$\begin{aligned} & S_k(x_1, \ldots, x_{2n}) \\ &\quad = \min(k, S_k(x_1, \ldots, x_n) + S_k(x_{n+1}, \ldots, x_{2n})) \\ &\quad = S_k(x_1, \ldots, x_n) + \min(k - S_k(x_1, \ldots, x_n), S_k(x_{n+1}, \ldots, x_{2n})) \\ &\quad = \min(S_k(x_1, \ldots, x_n), k - S_k(x_{n+1}, \ldots, x_{2n})) + S_k(x_{n+1}, \ldots, x_{2n}) \end{aligned}$$

The structure of the Cnt_k circuit directly reflects the above recurrence. We'll next describe it's layout for the case that n is a power of 2. The generalization to other values of n can be obtained in a straightforward manner by taking the circuit for the next power of 2, hardwiring the excess inputs to 0, and simplifying the result. For notational convenience, number the $n = 2^l$ inputs $0, 1, \ldots, n-1$ and identify them with their length l binary expansion. A binary string s of length less than l then naturally correspond to the block of $2^{l-|s|}$ consecutive inputs having s as a prefix. In particular, the empty string ϵ identifies the sequence of all inputs. The above recurrence can now be rewritten as

$$\begin{aligned} S_k(s) &= \min(k, S_k(s0) + S_k(s1)) \\ &= S_k(s0) + \min(k - S_k(s0), S_k(s1)) \\ &= \min(S_k(s0), k - S_k(s1)) + S_k(s1) \end{aligned}$$

Figures 4.2 and 4.3 show the layout of circuit Cnt_k on input block s. The first one shows the case where $|s| = l - 1$, i.e. part of the bottom layer of $Cnt_k(\epsilon)$. There are $2^{|s|} = n/2$ such parts in the bottom layer. The second shows part of the middle/upper layers of $Cnt_k(\epsilon)$, where $|s| < l - 1$.

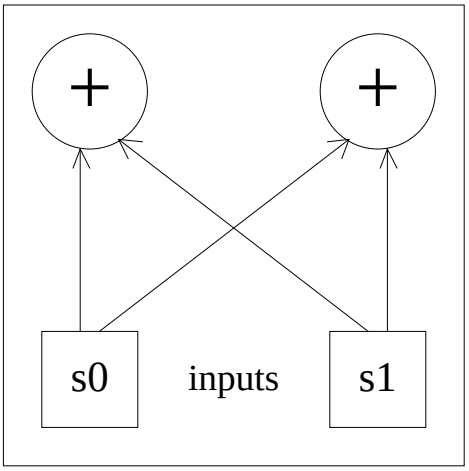


FIGURE 4.2. Bottom Layer of an Embedding of Circuit Cnt_k

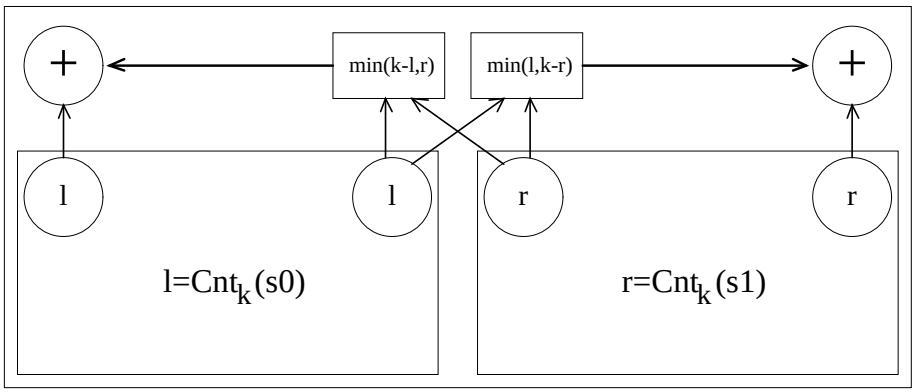


FIGURE 4.3. Middle/Upper Layer of an Embedding of Circuit Cnt_k

Each sub-circuit $Cnt_k(s)$ occupies a rectangular area above the input block s. It computes $S_k(s)$ which appears at both the top-left and top-right of the subcircuit, as the outputs of the two addition nodes there. Apart from these addition nodes, it adds onto the lower layers 6 short (constant length) wires, 2 constant-area subtract-and-minimize nodes (which have the constant threshold k hardwired into them), and 2 long wires.

All wires in layers B and up have constant bandwidth $B = \lceil \log(k+1) \rceil$, sufficient to carry any value in the range $\{0, 1, \ldots, k\}$. For the bottom $B-1$ layers, where $|s| > l-B$, $Cnt_k(s)$ sums over $2^{l-|s|}$ inputs, hence its wires can do with bandwidth $l - |s| + 1 \leq B$. This shows that the input node density is not limited so much by the wire-width as by the size of the non-input nodes. The width available to these nodes doubles at every layer, while their number of input bits increases by only 2. Hence, the inputs can be spaced $c\lambda$ apart, with c a small constant depending on the implementation of the non-input nodes in the bottom layers.

THEOREM **4.1** *For constant k, S_k is energy efficient.*

PROOF. Take Cnt_k for increasingly larger n as a circuit family. Since the bottom layer has depth 1 and each additional layer adds only 2 to the depth, the depth of the entire circuit is $2l - 1 = O(\log n)$ as required. It remains to show that $E_{worst}(Cnt_k) = O(n)$. The nodes have bounded area, since they have at most $2B$ input bits. The total number of nodes in Cnt_k, including input nodes, is

$$\sum_{|s|<l} 4 = 4(n-1).$$

It follows that the node energy of the circuit is only linear. The number of vertical wires is

$$\sum_{|s|=l-1} 4 + \sum_{|s|<l-1} 6 = 2n + 6(n/2 - 1) = 5n - 6.$$

These too have only a linear total area making the 'vertical wire energy' of the circuit linear.

So we are left to consider the 'long wire energy'. By the triangle inequality, it suffices to show that $E(Cnt_k, x, 0) = O(n)$. The simple but crucial observation is that if the long wire in $Cnt_k(s)$ running directly above $Cnt_k(s0)$ (i.e. the left long wire shown in Figure 4.3) carries a non-zero value, then $S_K(s) > S_K(s0)$. Similarly for the long wire above $Cnt_k(s1)$. More generally, if the long wire running directly above $Cnt_k(s)$ is non-zero, then $S_K(t) > S_K(s)$ for any t which is a proper prefix of s. Now imagine any vertical cross-section of the circuit. The number of non-zero long wires though this cross-section is then at most $S_K(\epsilon)$ which clearly cannot exceed k. Thus, the total length of non-zero long wires is bounded by k times the width of the circuit Cnt_k, which is $knc\lambda$, and the area involved is no more than $B\lambda$ times that. With k,c,B, and λ all being constant, this proves that $E(Cnt_k, x, 0) = O(n)$ and thereby the theorem. □

Since adding a comparator to Cnt_k adds only constant area and 1 to depth, we have

COROLLARY **4.2** For constant k, T_k is energy efficient.

4.3 Conclusions

We have described a construction that yields fast, minimum energy VLSI circuits to compute threshold functions and to count up to a constant. Energy bounds for general counting functions, including majority, remain open.

Bibliography

[1] Aggarwal, A., A. Chandra, P. Raghavan, *Energy Consumption in VLSI Circuits*, Proceedings of 20th ACM STOC, May 1988, pp. 205-216.

[2] Borodin, A., *On Relating Time and Space to Size and Depth*, SIAM Journal of Computing, Vol. 6, No. 4, December 1977, pp. 733-744.

[3] Brent, R.P., H.T. Kung, *A Regular Layout for Parallel Adders*, IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982, pp. 260-264.

[4] Brent, R.P., H.T. Kung, "The Area-Time Complexity of Binary Multiplication", JACM, Vol. 28, No. 3, July 1981, pp. 521-534.

[5] Kissin, G., *Upper and Lower Bounds on Switching Energy in VLSI*, JACM, Vol. 38, No. 1, January 1991.

[6] Kissin, G., *Models of Multiswitch Energy*, CWI Quarterly, Vol. 3, No. 1, March 1990, pp. 45-66.

[7] Kissin, G., *Modeling Energy Consumption in VLSI Circuits*, PhD Thesis, Department of Computer Science, University of Toronto, 1987.

[8] Kissin, G., *Functional Bounds on Switching Energy*, Proceedings of 1985 Chapel Hill Conference on Very Large Scale Integration, May 1985, pp. 181-196.

[9] Kissin, G., *Measuring Energy Consumption in VLSI Circuits: a Foundation*, Proceedings of 14th ACM STOC, May 1982, pp. 99-104.

[10] Krohn, K., J. Rhodes, "Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines", Transactions of American Mathematical Society, Vol. 116, 1965, pp. 450-464.

[11] Krohn, K., J. Rhodes, *Results on Finite Semigroups*, Proceedings of the National Academy of Science, USA, Vol. 53, 1965, pp. 499-501.

[12] Leo, J., *Energy Complexity in VLSI*, M.S. Thesis, University of Nymequen, The Netherlands, February 1984.

[13] Lengauer, T., K. Mehlhorn, *On the Complexity of VLSI Computations*, Proceedings of CMU Conference on VLSI, Computer Science Press, October 1981, pp. 89-99.

[14] Mead, C., L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

[15] Mead, C., private communication

[16] Snyder, L., A. Tyagi, *The Energy Complexity of Transitive Functions*, Proceedings of 24th Allerton Conference on Communication, Control and Computing, October 1986, pp. 562-572.

[17] Thompson, C., *A Complexity Theory for VLSI*, PhD Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1980.

[18] Tyagi, A., *Energy Complexity of Logic Level Structures,* Proceedings of Stanford Conference on Advanced Research in VLSI," MIT Press, 1987.

[19] Vuillemin, J., *A Combinatorial Limit to the Computing Power of VLSI Circuits*, IEEE Transactions on Computers, Vol. C-30, No. 2, 1983, pp. 135-140.

5

Associative Storage Modification Machines

5.1 Introduction

The Storage Modification Machine (*SMM*) is a machine model introduced by Schönhage in 1977 [16]. The model has its predecessor in the Kolmogorov-Uspenskii machine (*KUM*) [10]. Schönhage advocates his model as *a model of extreme flexibility*.

The model resembles the Random Access Machine (*RAM*) [1] as far as it has a stored program and a potentially infinite memory structure where it stores its data. Whereas the *RAM* uses an infinite sequence of storage registers, each capable of storing an arbitrarily large integer, the *SMM* operates on a directed graph by creating nodes and (re)directing pointers. The main difference between the *SMM* and the *KUM* is that the *KUM* operates on undirected instead of directed graphs.

We can approximately model an *SMM* by a Pascal program that uses records of pointers to records to describe the directed graph[1]:

```
type pointer = ^node;
     node    = record a,b: pointer end;
var  head    :  pointer;
```

In contrast with Pascal, pointers are not allowed to be *nil* or undefined; they must always point to some node. The (finite) set of pointer names, in the example $\{a, b\}$, is called the *alphabet* of *directions*, denoted Δ. The pointers in the graph are labeled with the elements of Δ such that each node in the digraph has, for each direction $\delta \in \Delta$, exactly one outgoing δ-pointer. The graph thus has regular outdegree $|\Delta|$. To complete the analogy between an

[1] In Pascal `^T` denotes the type 'pointer to T'; a value of this type is the address of an object of type T. Indirection through a pointer is written as `p^`, which refers to the object at which p points.

SMM and a Pascal program, the latter must be restricted to the use of only one variable; the pointer `head`. By repeated application of the Pascal `new` statement, the program can create an arbitrarily large data structure. This is addressed with expressions like `head^.b^.a^.b^.b^.a`. Similarly, the *SMM* addresses its storage with words (strings) over Δ, like *babba*. In the *SMM* model, there is only a conceptual head pointer—at any time, one node, call the *center*, is distinguished as the one from where addressing starts. Thus the center, whose identity can change dynamically, is addressed by the empty word ϵ, and other nodes are addressed by following pointers starting from the center.

It has been established that from the perspective of computational complexity theory the *SMM* (if equipped with the correct space measure [12, 21]) is computationally equivalent to the other standard sequential machine models like the Turing machine and the *RAM*. This equivalence amounts to the fact that these models simulate each other with polynomially bounded overhead in time and constant factor overhead in space, thus satisfying the so-called *invariance thesis* [17, 22].

For most sequential models there have been proposed parallel machine models based on the classical sequential version. For the Turing machine Savitch [15] has proposed a parallel version based on parallel recursive branching; a model based on nondeterministic forking on a shared set of tapes was described by Wiedermann [24, 25], but this model turns out to be polynomially equivalent in time and space with the standard sequential devices. The richness of parallel models based on the *RAM* is even much greater, which makes it hard, if not impossible to refer to a small set of representative models. There are models based on shared memory and alternative models based on local storage and message passing. Hybrid combinations occur as well. Within each class there exist more refined distinctions like the resolution strategy for resolving write conflicts in shared memory models, the available arithmetic instructions and the mechanism for restricting the number of processors activated during a computation. Moreover, there exist sequential models which become computationally equivalent to parallel models due to their power to create and manipulate exponentially large values in a linear number of steps in the uniform time measure. Also, by exploiting the alternating mode of computation [5], some standard sequential devices become computationally equivalent to the parallel machines.

For a more detailed survey of parallel models we refer to [20, 22]. For the purpose of the present exposition, it suffices to give some impression of the overall landscape of parallel machine models.

It turns out that most parallel models proposed in the literature belong to the so-called *Second Machine Class* consisting of machine models which obey the *Parallel Computation Thesis*. This thesis expresses that the class of languages recognized in nondeterministic polynomial time on the parallel device is equal to the class *PSPACE* of languages recognized in polynomial space on a sequential device. Conversely all languages in *PSPACE* are recognized in deterministic polynomial time on the parallel machine. In our reading the Parallel Computation Thesis entails the equivalence of deterministic and nondeterministic polynomial time on the parallel model. The models for which the thesis was originally formulated obey this more restricted thesis as well. And indeed those models for which nondeterministic polynomial time seems to exceed *PSPACE*

nowadays are held to be more powerful.

Not all parallel models obey the above parallel computation thesis. Some weak models turn out to be polynomial time equivalent to the sequential models (the parallel Turing machine proposed by Wiedermann, and its equivalents [24, 25] being a typical example). Other models, like the *P–RAM* presented by Fortune and Wyllie [7] deviate from the thesis by recognizing exponentially time bounded languages in polynomial nondeterministic time on the parallel device; some parallel devices even recognize arbitrary languages in constant time [13]. The second machine class therefore represents a frequently occurring version of the power of uniform unrestricted parallelism rather than the union of all possible parallel machine models. Second machine class members can be characterized as providing the right mixture of exponential growth potential together with the proper degree of uniformity. The exponential growth potential is required for the implementation of the transitive closure algorithm on a directed graph of exponential size (which models the computation graph of some *PSPACE*-bounded machine), or the direct solution of the *PSPACE*-complete problem *QBF* in polynomial time. The uniformity is required for performing the simulation of a polynomial-time computation of the nondeterministic version of the parallel machine in polynomial space. See [22] for more details on the standard strategies for proving membership in the second machine class.

In this chapter we propose (as far as we know for the first time) a parallel version of the storage modification machine which belongs to the second machine class. To our knowledge few parallel versions of pointer machines have been investigated in the complexity theory literature. The earliest reference known to us concerns a parallel version of the Kolmogorov-Uspenskii machine which was proposed by Barzdin [2, 3]. This machine operates like an irregular cellular array of finite state automata in a graph which is dynamically changed by the individual nodes interacting with their neighbourhood. A single computation step resembles a parallel rewrite step in a graph grammar derivation. In this model all nodes are active in every computation step; if their neighborhood matches the pattern required by the instruction the node will transform its environment. The Hardware Modification Machine (*HMM*) introduced by Dymond and Cook [6] behaves in a similar way. This model indeed has been investigated for its complexity behavior. From Lam and Ruzzo [11] it follows that the machine is equivalent with constant factor time overheads with a restricted version of the *P–RAM* of Fortune and Wyllie. From this result one can observe that the *HMM* represents another example of the class of devices which are located beyond the second machine class - its nondeterministic version accepts *NEXPTIME* in polynomial time.

The computational power of our *ASMM* model originates from the possibility of traversing pointers in their *reverse* order. By using reverse directions, an *ASMM* can address, from a given node x, all the nodes that are associated with x by pointing to x (hence the name[2]). More than one node can be reached on a path by traversing pointers in the reverse direction. Note that at this point it is crucial that we have based ourselves on the *SMM* rather than the older *KUM* model; in an undirected graph traversing pointers in the reverse direction

[2]compare with *content-addressable associative memory*

makes no sense.

As in the standard *SMM* model the finite control accesses the storage structure by means of a single center node. The power of traversing reversed pointers is used only in two types of instructions: the *new* and the *set* instruction. The first argument of the above two instructions is a path which now may contain reverse pointers. This path therefore no longer denotes a single node but a set of nodes (which in fact may be empty). The action described by the instruction now will be performed for all nodes in this set in parallel. The second argument of the *set* instruction is required to be a path consisting of forward pointers only; it therefore always denotes a single node. Therefore the action performed by the two instructions above is deterministic.

Our model may be considered to be a member of the class of sequential machines which operate on large objects in unit time and obtain their power of parallelism thereof. Other models of this character are the vector machines of Pratt and Stockmeyer [14], the *MRAM* proposed by Hartmanis and Simon [9] and simplified by Bertoni et al. [4], and also the *EDITRAM* presented by Stegwee et al. [18, 22].

Evidently our model is one among a number of possible alternatives for designing a parallel version of the *SMM* model. In the conclusion of this chapter we discuss another alternative suggested by an anonymous referee.

Following [22] we denote the class of languages accepted in polynomial time by the *ASMM* model by *ASMM−PTIME*. The class of languages accepted in polynomial time by nondeterministic *ASMM* devices is denoted by *ASMM−NPTIME*. The class *PSPACE* as indicated above, denotes the class of languages recognized in polynomial space on a Turing machine. The fact that the *ASMM* is a true member of the second machine class is now expressed by the equality:

$ASMM{-}PTIME = ASMM{-}NPTIME = PSPACE$

In the proof of this equality we use the well known *PSPACE*-complete problem:

QUANTIFIED BOOLEAN FORMULAS (*QBF*) [19] :

INSTANCE: A formula of the form $Q_1 x_1 \ldots Q_n x_n [P(x_1, \ldots, x_n)]$,
where each Q_i equals $\forall$ or $\exists$, and where $P(x_1, \ldots, x_n)$
is a propositional formula in the boolean variables $x_1, \ldots, x_n$.

QUESTION: does this formula evaluate to *true*?

5.2 The *SMM* and the *ASMM* models

Our *ASMM* model is based on the Storage Modification Machine as introduced by Schönhage in 1970 [16]. The *SMM* model resembles the *RAM* model as far as it has a stored program and a similar flow of control. It has a single storage structure, called a Δ-*structure*. Here Δ denotes a finite alphabet consisting of at least two symbols. We denote the reverse of a direction $a \in \Delta$ as $\bar{a}$. Furthermore, $\bar{\Delta} = \{\bar{\alpha} | \alpha \in \Delta\}$ is the set of reverse directions and we let $\tilde{\Delta} = \Delta \cup \bar{\Delta}$.

A Δ-structure X is a finite directed graph each node of which has $k = |\Delta|$ outgoing edges which are labeled by the k elements of Δ. In Schönhage's formalization, a Δ-structure is a triple (X, c, p), where X denotes the finite set

of nodes, $c \in X$ is the *center*, and $p : X \times \Delta \to X$ is the pointer mapping; $p(x, \alpha) = y$ means that the α-pointer from x goes to y.

There exists a map p^* from Δ^* to X defined as follows: For the empty string ϵ one has $p^*(\epsilon) = c$, and otherwise $p^*(wa) = p(p^*(w), a)$ is the end-point of the a-labeled pointer starting in $p^*(w)$.

The map p^* does not have to be surjective. Nodes which can not be reached by tracing a word w in Δ^* starting from the center c will turn out to play no subsequent role during the computations of the *SMM*. In the *ASMM* model pointers can be traversed in the opposite direction, and therefore these nodes no longer can be disregarded as being garbage.

The storage of an *SMM* or an *ASMM* is a dynamically changing Δ-structure, which initially consists of a single node, the center. The *ASMM*'s operation is described by a *program*, which is a finite sequence of *labels* and *instructions*. Labels can be used in control flow statements; they should occur exactly once in case the machine is deterministic. *Nondeterminism* is introduced by allowing multiple occurrences of the labels referred to in jump or conditional jump instructions. Consequently we only consider nondeterminism in the flow of control. An alternative would be to design instructions that manipulate the data in a nondeterministic manner, but such instructions easily lead to a more powerful model.

In the text below we separate labels and instructions by a colon, whereas instructions are ended by semicolons.

The instruction repertoire of the *SMM* and the *ASMM* includes the *common* instructions (the λ's are labels and $\beta \in \{0, 1\}$)

input λ_0, λ_1;
output β;
goto λ;
halt;

The *input* instruction reads an input bit β and transfers control to λ_β. The other instructions are straightforward.

Furthermore there exist three *internal* instructions which operate on memory - in this case a Δ-structure X. For the *SMM* the arguments in these instructions are strings over Δ. For the *ASMM* the single argument of *new* and the first argument of *set to* are strings over $\tilde{\Delta}$; the other arguments (second argument of *set to* and both arguments of the *if* instruction) are strings over Δ. All arguments are finite strings which are written literally in the program. We first describe their meaning for the *SMM*:

1. *new* W: creates a new node which will be located at the end of the path traced by W; if $W = \epsilon$ the new node will become the center; otherwise the last pointer on the path labeled W will be directed towards the new node. All outgoing pointers of the new node will be directed to the former node $p^*(W)$

2. *set* W *to* V: redirects the last pointer on the path labeled by W to the former node $p^*(V)$; if $W = \epsilon$ this simply means that $p^*(V)$ becomes the new center; otherwise the structure of the graph is modified.

3. *if* $V = W$ (*if* $V \neq W$) *then* $\langle instr \rangle$: depending on whether $p^*(V)$ and $p^*(W)$ coincide or not, the conditional instruction $\langle instr \rangle$ (conditional jump suffices) is executed or skipped.

In the *ASMM* model the Δ-structure can be addressed by words (also called *paths*) over the alphabet of normal and reverse directions $\tilde{\Delta}$. Every word $W \in \tilde{\Delta}^*$ addresses the (possibly empty) set of all the nodes reachable from the center by following the consecutive directions and reverse directions in W.

The notion of 'addressing' is formalized by the mapping $P : \tilde{\Delta}^* \rightarrow 2^X$, defined by:

$$\begin{aligned} P(\epsilon) &= \{c\} \\ P(W\alpha) &= \{p(x,\alpha) | x \in P(W)\} \\ P(W\bar{\alpha}) &= \{x | p(x,\alpha) \in P(W)\}. \end{aligned}$$

Note: It will often be convenient to give a name to an address path $V \in \Delta^*$. In the code fragments presented in this chapter, we will use paths having such a name v as a prefix, in addition to fully explicit paths. This serves two purposes. First, fixed nodes that have been given descriptive names can be addressed by their name rather than some arbitrary path (we say that a node is fixed iff it has a constant address). Second, if we are using one of the pointers from a fixed node to traverse part of the graph, it can be given a name that more closely resembles its function: that of a variable. We will use variable names without specifying which path they stand for, omitting the details of the creation of spare nodes to provide the required[3] pointers.

A node x is said to be *directly* addressable if it is reachable from the center by normal (non-reversed) directions, i.e. $\exists\, V \in \Delta^* : P(V) = \{x\}$.

In order to facilitate the descriptions of the internal instructions, we define a mapping $Q : \tilde{\Delta}^* \rightarrow 2^X$, from a path to the set of nodes from which the last pointer on this path originates, by:

$$\begin{aligned} Q(\epsilon) &= \emptyset \\ Q(W\alpha) &= P(W) \\ Q(W\bar{\alpha}) &= P(W\bar{\alpha}). \end{aligned}$$

The *new* and *set* change the Δ-structure from (X, c, p) to (X', c', p') as follows:

new W;
: Here, $W \in \tilde{\Delta}^*$ determines where new nodes are inserted. If $W = \epsilon$, then a new center c' is created such that $X' = X \cup \{c'\}$ and $p'(c', \delta) = c$ for all $\delta \in \Delta$. Otherwise, if $W = U\tilde{\alpha}$ ($\tilde{\alpha}$ is either α or $\bar{\alpha}$), then for every node $u \in Q(W)$ a new node x_u is created such that $X' = X \cup \{x_u | u \in Q(W)\}$, $p'(u, \alpha) = x_u$, $\forall\, \delta \in \Delta\; p'(x_u, \delta) = p(u, \alpha)$, and $c' = c$. All other pointers remain unchanged.

[3] In the case of the *ASMM*, when we use an address like $v\bar{x}$ with v a variable name, it is desirable for v not to be an x-pointer, i.e. that the address that v stands for doesn't end with the direction x.

set W *to* V;
Here, $W \in \tilde{\Delta}^*$ determines which pointers are redirected to the node determined by $V \in \Delta^*$. If $W = \epsilon$, then $c' = P(V)$ becomes the new center. Otherwise, if $W = U\tilde{\alpha}$, then for every node $u \in Q(W)$, $p'(u, \alpha) = P(V)$ and $c' = c$. In both cases X' is the restriction of X to the nodes which are reachable from c'.

The third internal instruction is the *if* statement. Since both paths in this instruction consist of forward pointers only, the meaning of this instruction is equal for the *SMM* and the *ASMM*.

The *time complexity* we use is simply the number of instructions executed. We do not concern ourselves with the *space complexity*; see [12, 21] for a discussion of the space complexity of the *SMM*.

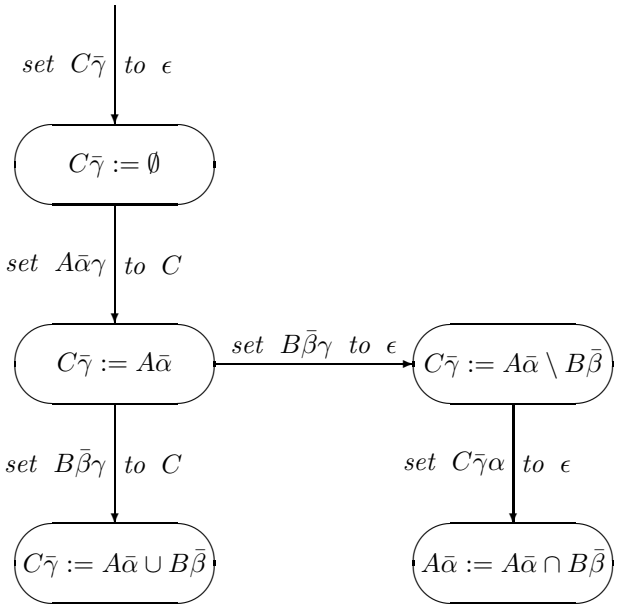
5.3 An illustration of the power of associativity

We demonstrate the power of the *ASMM* model by showing the capability to manipulate arbitrarily large sets in constant time.

The model allows the following natural representation of sets. If W is a word over Δ, and $\alpha \in \Delta$ a direction, then $P(W\bar{\alpha})$ is the set of all nodes having their α-pointer directed to the node $P(W)$. Assume that our alphabet is $\Delta = \{A, B, C, \alpha, \beta, \gamma\}$ and that the A, B, and C-pointers from the center go to three different nodes $P(A)$, $P(B)$ and $P(C)$, none of which is the center. We will now consider the sets $P(A\bar{\alpha})$, $P(B\bar{\beta})$ and $P(C, \bar{\gamma})$ and see how the standard set operators can be applied to them by using appropriate *set to* instructions. We have chosen A, B and C to be directions so that the instructions with which we will implement the set operators cannot affect the addressing of the nodes $P(A)$, $P(B)$ and $P(C)$. As long as no such interference exists, we can generalize to the case where A, B and C are not elements of Δ but words over Δ.

The instruction *set* $A\bar{\alpha}\beta$ *to* B; has the effect of adding to $P(B\bar{\beta})$ the set $P(A\bar{\alpha})$, while the instruction *set* $A\bar{\alpha}\beta$ *to* ϵ; removes from $P(B\bar{\beta})$ the nodes which are also in $P(A\bar{\alpha})$.

The figure below now shows how the standard set operators, shown as assignment statements in the boxes, can be implemented in terms of *set to* instructions. The center ϵ is used to direct pointers away from A or C.



The following program illustrates how in linear time a set $P(\bar{\alpha})$ of exponential size can be constructed (with a singleton alphabet):

new $\bar{\alpha}$;
set $\bar{\alpha}\bar{\alpha}$ *to* ϵ;
⋮
new $\bar{\alpha}$;
set $\bar{\alpha}\bar{\alpha}$ *to* ϵ;

Initially only the center exists, so all nodes point to the center. If at some point 2^k nodes exist, all of which point to the center, then after the *new* instruction, each of these 2^k nodes now points to one of 2^k newly created nodes, which again point to the center. Next the *set* instruction makes all 2^{k+1} nodes point to the center. Hence after k repetitions of these two instructions the size of the set $P(\bar{\alpha})$ has become 2^k.

In the next section we will see how these and similar constructions are used to process large amounts of data in parallel.

5.4 *PSPACE = ASMM−(N)PTIME*

The proof of membership in the Second Machine Class is usually split into two parts:

LEMMA **5.1** *PSPACE* ⊆ *ASMM−PTIME*

We prove this by sketching an *ASMM* which solves the *PSPACE*-complete problem *QBF* in polynomial time.

LEMMA **5.2** *ASMM−NPTIME* ⊆ *PSPACE*

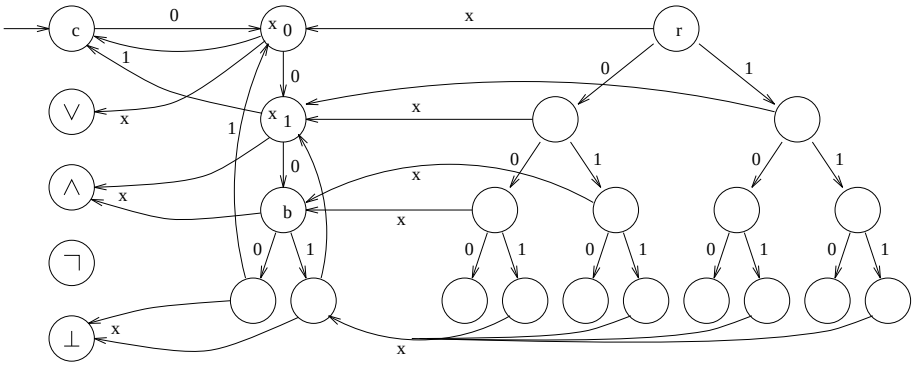


FIGURE 5.1. storage structure for $\exists x_0 \forall x_1 : x_0 \wedge x_1$

We prove this by showing how to simulate t steps of a nondeterministic *ASMM* on a Turing machine using $O(t^2)$ space.

5.4.1 QBF ∈ ASMM−TIME(n^2)

The *ASMM* algorithm we present for solving *QBF* in polynomial time proceeds in 8 stages. Let $X = \{x_0, \ldots, x_{k-1}\}$ be the set of variables in the formula of length n, let $\Delta = \{0, 1, x\}$ and let c be the center. Basically, the algorithm expands the formula by rewriting the quantifiers, one by one, innermost first, as follows:

$$\forall x_i F(x_i) \Longrightarrow F(0) \wedge F(1),$$
$$\exists x_i F(x_i) \Longrightarrow F(0) \vee F(1).$$

The resulting, fully expanded formula, can be viewed as a tree. It consists of a complete binary tree T of depth k, with an instance of the formula body B rooted at each leaf of T. In each such instance, the variable references can be replaced by the truth values assigned to them along the path from the root to the leaf. The algorithm does little more than to build and evaluate this tree.

Figure 5.1 depicts the structure built for the example formula $\exists x_0 \forall x_1 : x_0 \wedge x_1$. The part on the right represents the expanded formula (where some of the x-pointers have been omitted for clarity). We now briefly summarize each of the 8 stages:

1. Build a list of nodes $c, x_0, x_1, \ldots, x_{k-1}, b$ linked through the 0-pointer. Using the $0, 1$-pointers, build a representation of the formula body as a binary tree B rooted at b. The non-leaf nodes of B represent the connectives (and, or, not) while the leaves represent instances of variables. Create the nodes $\wedge, \vee, \neg, \perp$ that represent the different types of nodes in B.

2. Build a complete binary tree T of depth k using the $0, 1$-pointers. For a node at depth i, its 0-subtree represents the case $x_i = 0$ and its 1-subtree the case $x_i = 1$.

3. Build 2^k copies of B rooted at the leaves of T.

4. For every leaf u of B representing an instance of x_i, let the 2^k copies of u direct their x-pointer to either u or c depending on the connective of u's parent and the value assigned to x_i.

5. For every non-leaf u of B, let the 2^k copies of u direct their x-pointer to either u or c depending on the connectives of u and its parent.

6. For every x_i, let the 2^i nodes of T at level i direct their x-pointer to either x_i or c depending on the quantifiers of x_i and x_{i-1}.

7. Evaluate all copies of B in parallel.

8. Evaluate T.

Recalling Section 5.2, we address the nodes $\wedge, \vee, \neg, \bot$ by their name, and use variable-names v, w for the purpose of traversing X and B.

The data structure constructed can be roughly divided in two parts: the linearly sized input representation (on the left in Fig5.1), and the exponentially sized formula expansion on the right. For a node $v \in X \cup B$, let $C(v)$ its set of copies. For $v = x_i$, these are the 2^i nodes at depth i in T, while nodes in B all have 2^k copies. The two parts, left and right, are connected only by the x-pointers that go from a node in some $C(v)$ to either v or to the center c. Thus the following invariant holds throughout the execution:

$$\forall v \in X \cup B : \; v\bar{x} \subseteq C(v).$$

Furthermore, the x-pointer from r, the root of T, remains anchored to x_0 until evaluation is completed.

For a clear understanding of the construction, it is important to distinguish between truth-values and their representation. Conceptually, the algorithm works with truth values, 1 (true) and 0 (false). The leaves of the 2^k copies of B are assigned truth values in the obvious way according to which variable they represent. The other nodes are assigned default truth values, which are 0 for a disjunction, and 1 for a conjunction or a negation (recall that the quantifiers have been transformed into disjunction and conjunction). Next, a bottom-up process repeatedly changes defaults, that are in disagreement with their evaluated children, into the correct evaluation. In order to facilitate this process, we use a mixed representation of truth values, as follows.

Call a node $u' \in C(u)$ *active* if its x-pointer is directed to u or *passive* if its x-pointer is directed to c. A truth value is represented by the active state iff that truth value disagrees with the default of its parent, and with the passive state otherwise. To summarize:

parent type	$\wedge$	$\vee$	$\neg$
parent default	1	0	1
active value	0	1	1

As an example, suppose a $\neg$ node $u' \in C(u)$ has a parent $\vee$ node. The parent gets a default value of 0, which is to be changed into a 1 iff either of its children evaluates to 1. Thus, 1 is the active value for u'. The value 0 is passive for u', since it agrees with the default 0 value of its $\vee$ parent. Since a $\neg$ node is 1 by default, u' is active by default, hence its x-pointer is initially directed to u.

The representation of the truth value at a node u therefore depends on the type of the logical connective associated to the parent of u in the tree. This holds also for the nodes in the tree T which are associated to the variables x_i. In this tree the copies of the variables have been treated as logical connectives according to the type of the quantifier binding this variable. Note that by keeping the x-pointer of r directed to x_0, it is made active by default.

In the algorithm above stages 1, 2 and 3 are used for building the tree; during stage 4 the truth values are assigned to all variable occurrences in the copies of B, and in stages 5 and 6 all intermediate nodes are given their default values. During the final two stages the entire tree is evaluated.

We next describe each of the above stages in some more detail.

In stage 1 the input is examined and used to construct a linearly sized list and tree representing the formula. We represent the type of a node $u \in X \cup B$ by directing its x-pointer to one of the special nodes $\vee, \wedge, \neg, \bot$. As noted before, these four symbols will also be used as paths addressing the nodes. The leaves of B are of type $\bot$ and have their 1-pointer directed to the appropriate x_i. Existentially quantified x_i have $type(x_i) = \vee$ and universally quantified x_i have $type(x_i) = \wedge$.

When traversing the list of $x_i's$, the algorithm needs to be able to detect its end. Since the nodes in B already use the 1-pointer to point to their children (or single child in case of a $\neg$ node), we have the x_i direct their 1-pointer to the center, and thus by comparing vx with ϵ can tell whether v addresses a node in X or in B.

In stage 2 the parallel power of the machine is used to build an exponentially large tree in linear time. This is achieved by the piece of code below:

new v;	create r, root of T
set vx *to* 0;	classify it
set v *to* 0;	start X traversal
λ : *new* $v\bar{x}0$;	0-children for $C(x_i)$
set $v\bar{x}0x$ *to* $v0$;	classify in $C(x_{i+1})$
new $v\bar{x}1$;	1-children for $C(x_i)$
set $v\bar{x}1x$ *to* $v0$;	classify in $C(x_{i+1})$
set v *to* $v0$;	advance to x_{i+1}
if $v1 = \epsilon$ *then goto* λ;	repeat for all x_i

The construction of 2^k copies of B in stage 3 proceeds analogously. Note that by now all the leaves of T have their x-pointer directed to b. Traversing B in preorder, we do the following at each node v:

if $vx = \bot$ *goto* λ_2;	do nothing at leaves
if $vx = \neg$ *goto* λ_1;	$\neg$ node has no 0-child
new $v\bar{x}0$;	create 0-child
set $v\bar{x}0x$ *to* $v0$;	classify in $C(v0)$
λ_1: *new* $v\bar{x}1$;	create 1-child
set $v\bar{x}1x$ *to* $v1$;	classify in $C(v1)$
λ_2:	

In stage 4, all the x-pointers in the copies of leaves of B are installed. Again omitting the details of how to traverse B, let w be a leaf of B ($wx = \bot$), and $w1 = x_i$ the variable it represents. We show how to install the x-pointers in all

copies $C(w)$ of w. We assume that w has the active value 1. The case for 0 is analogous. The code fragment

set v *to* 0;	start traversal of the x_j
λ : *set* $v0\bar{x}$ *to* ϵ;	clear $C(x_{j+1})$
if $v \neq w1$ *then set* $v\bar{x}0x$ *to* $v0$;	if $i \neq j$ then take both children
set $v\bar{x}1x$ *to* $v0$;	if $i = j$ then take only 1-children
set v *to* $v0$;	advance to x_{j+1}
if $v1 = \epsilon$ *then goto* λ;	repeat for all x_j

ends with $b\bar{x}$ equal to the set of leaves of T which have 1, the active value, assigned to x_i. In a similar fashion we can traverse the path from b to w, to end up with the active nodes of $C(w)$ pointing to w and the passive ones pointing to c.

The next two stages, 5 and 6, prepare the evaluation by giving default values to copies of non-leaves of B and nodes in T. This is most easily done by first setting $v\bar{x} = C(v)$ for all v in X or internal in B. Then, in a second pass, those copies that should be passive by default are taken care of by executing the following instruction for the appropriate v in X and B:

set $v\bar{x}$ *to* ϵ;	clear $v\bar{x}$

Now all that's left to be done is the evaluation itself. This is done bottom up—by a post-order traversal of B and then from x_{k-1} back to x_0. With the other cases being analogous, we restrict ourselves to the evaluation of an $\wedge$-node $w \in X \cup B$. Let v be its parent. The default value of w is 1, which is passive if v has type $\wedge$, or active if v has type $\vee, \neg$. The value of w should become 0 if either of its children has value 0, which is active for them. It should now be clear that the code fragment

if $vx = \wedge$ *then goto* λ_1;	passive or active default?
set $w0\bar{x}\bar{0}x$ *to* ϵ;	change to passive if 0-child disagrees
set $w1\bar{x}\bar{1}x$ *to* ϵ;	change to passive if 1-child disagrees
goto λ_2;	
λ_1 : *set* $w0\bar{x}\bar{0}x$ *to* w;	change to active if 0-child disagrees
set $w1\bar{x}\bar{1}x$ *to* w;	change to active if 1-child disagrees
λ_2 :	

evaluates node w. The technique used here is essentially the same as in section 5.3 for computing a union. Because of our symmetric representation, it works for both $\vee$ and $\wedge$.

When evaluation is complete, the root of T, r, will have its x-pointer directed to either x_0 or c, depending on the truth value of the input formula and on the default truth value for x_0 which in turn depends on its quantifier. In order to test whether the x-pointer from r is directed to x_0, we will use the fact that $x_0\bar{x} \subseteq \{r\}$ and $p(x_0, x) \in \{\wedge, \vee\}$. The instruction

set $0\bar{x}xx$ *to* ϵ;

changes $p(x_0, x)$ to ϵ iff $0\bar{x}$ is nonempty, which is equivalent to $p(r, x) = x_0$. Combining all this information yields the value of the formula.

Regarding the time complexity, the most time-consuming stage is number 4, where for each leaf of B, both X and B are traversed, requiring at most n^2 steps. Hence the complete algorithm runs in quadratic time.

5.4.2 $ASMM{-}NTIME(t) \subseteq SPACE(t^2)$

The simulation which proves this inclusion is relatively straightforward and employs previously known methods [14, 9]. We can write down in polynomial space a trace of the computation containing information on the sequence of instructions executed. Since the machine being simulated is nondeterministic this trace is guessed. Next it is verified by means of a system of recursive procedures and some other arrays containing polynomially sized information that this trace indeed represents an accepting computation. The *if*, *new* and *set to* statements pose the main problems, since their impact on the Δ-structure requires repeated recomputation of the current state of the Δ-structure. In polynomial space we cannot explicitly store the possibly exponentially large Δ-structure of the ASMM-machine, so an implicit representation is called for. This will consist of three arrays, and three mutually recursive functions. The arrays are

1. $instr[i]$ holds the instruction executed at step i
2. $nodes[i]$ holds the number of nodes at time i
3. $center[i]$ holds the center at time i

The simulation starts at time 0 and has step i ($i \geq 1$) leading to time i. Each array is of length t, the number of steps to be simulated, and each array element fits in t bits since the number of nodes can at most double after each step. Every node will have a unique number, and the resulting ordering of nodes is used for numbering nodes created by a *new* instruction. More precisely, a *new* W; instruction at step i is simulated as follows:

If $W = \epsilon$, then $center[i] = nodes[i-1]$ and $nodes[i] = nodes[i-1] + 1$.

Otherwise, if $W = U\tilde{\alpha}$, then $center[i] = center[i-1]$ and $nodes[i] = nodes[i-1] + |Q(W)|$. Semantically, if $Q(W) = \{x_0 < x_1 < \ldots < x_{k-1}\}$, then at time i, $p(x_j, \alpha) = nodes[i-1] + j$, for $j < k = |Q(W)|$.

For all other instructions, $nodes[i] = nodes[i-1]$ and $center[i] = center[i-1]$, except that the instruction *set* ϵ *to* V; sets $center[i]$ to $P(V)$. In order to compute $P(V)$ and to simulate the *if* instruction, we use the following functions:

$p(x, \alpha, i)$ returns the number of the node $p(x, \alpha)$ at time i

$P(x, W, i)$ returns whether $x \in P(W)$ at time i

$Q(x, W, i)$ returns whether $x \in Q(W)$ at time i.

These functions satisfy the equations

$$\begin{aligned} Q(x, \epsilon, i) &= \mathit{false} \\ Q(x, U\alpha, i) &= P(x, U, i) \\ Q(x, U\bar{\alpha}, i) &= P(x, U\bar{\alpha}, i) \end{aligned}$$

$$\begin{aligned} P(x,\epsilon,i) &= (x == center[i]) \\ P(x,U\alpha,i) &= (\exists\ 0 \le y < nodes[i] : P(y,U,i) \wedge p(y,\alpha,i) == x) \\ P(x,U\bar{\alpha},i) &= P(p(x,\alpha,i),U,i) \\ p(x,\alpha,0) &= 0 \end{aligned}$$

which shows that they can be easily computed, apart from the case $p(x,\alpha,i)$ for positive values of i. The action of p in this case depends on the value of $instr[i]$, the only interesting values of which are *new* and *set*.

Consider first the case $instr[i] = new\ W$. If $x \ge nodes[i-1]$ then (using $Q(y,W,i)$) the difference $x - nodes[i-1]$ can be used to find the y in $Q(W)$ which 'generated' and now points to x (unless $W = \epsilon$, in which case $p(x,\alpha,i) = center[i-1]$). Now $p(x,\alpha,i) = p(y,\delta,i-1)$, where δ is the direction W ends in. On the other hand, suppose $x < nodes[i-1]$. If $W = U\tilde{\alpha}$ (i.e. α-pointers may have changed) and $Q(x,W,i-1)$, then x has generated $p(x,\alpha,i) = nodes[i-1] + |\{y < x | q(y,W,i-1)\}|$. Otherwise $p(x,\alpha,i) = p(x,\alpha,i-1)$.

Second and last, consider the case $instr[i] = set\ W\ to\ V$. If $W = U\tilde{\alpha}$ and $Q(x,W,i-1)$, then $p(x,\alpha,i)$ is the unique y satisfying $P(y,V,i-1)$. Otherwise $p(x,\alpha,i) = p(x,\alpha,i-1)$.

These functions can easily be coded on a Turing Machine using recursion (stack frames). The recursion depth is bounded by ct, where c is a constant depending only on the maximum path length of the ASMM program. Each stack frame holds a return address and some node numbers and counters each of which fits in t bits. Together with the three arrays, space $O(t^2)$ suffices for the simulation of t steps of the ASMM.

5.5 Conclusion

Of all the parallel models which have been shown to belong to the Second Machine Class, the *ASMM* is the first to obtain its power from the use of associative addressing, thus making it an interesting addition to the realm of Second Machine Class devices. It provides another example that a small modification of a machine model can enforce a substantial increase in computational power. In [4] it was shown that this increase is provoked by adding multiplicative instructions to the unit-time standard *RAM* model. Similarly the *EDITRAM* model obtains its power from introducing a few edit operators that are available on most real life text editors anyhow. In the *ASMM* model it turns out that traversing pointers in the reverse direction is all we need to obtain full parallel power. At the same time, the fact that the storage structure of the *ASMM* is manipulated by a finite program that interacts with the Δ-structure by means of a single center seems to be the main reason why the machine has not become too powerful. As shown by Lam and Ruzzo [11], a model where the nodes become independently active finite automata becomes equivalent with a restricted version of the *P–RAM* of Fortune and Wyllie. This suffices for making the nondeterministic version more powerful than *PSPACE* (except for the unlikely case that *PSPACE = NEXPTIME*). This situation resembles the relation between the *SIMDAG* described by Goldschlager [8], where a single processor broadcasts its instructions to a collection of peripheral processors and the *P–RAM* model of Fortune and Wyllie [7] where the local processors

are independent.

Clearly there are other models which could serve as a parallelized version of the *SMM*. In our model the set-to instruction is rather limited. Since its second argument addresses a single node, it cannot be used for setting different pointers to different destinations. This severely limits the scope of proofs that our machine is indeed so powerful. A more conventional approach, based on the construction of the transition graph of a polynomial space bounded Turing machine, and the computation of its transitive closure by pointer jumping—as suggested by the referee—is rendered infeasible by the limitation of the set-to instruction. Overcoming this limitation would require a different flavour of set-to instruction. A natural possibility is to allow the conventional set-to instruction of the *SMM* to be executed in parallel with respect to many different 'centers', the latter being specified by a third argument which is a string in $\tilde{\Delta}$. This model has some drawbacks, however. One is the possibility of conflicts arising when a pointer must be set to one node when addressed through one center, and to another node when addressed through another center. Resolving this problem would probably detract from the elegance of the model, one of its prime features. Another problem is that it becomes harder to manage all the pointers, since there is no simple way in which to direct a bunch of them to some fixed node where they can be 'out of the way'. Thus it is not a strict generalization of our model, although it should be possible to simulate our set-to instruction with this new one by keeping around an extra direction to always point to the real center.

Bibliography

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] Barzdin', Ya. M., *Universal pulsing elements*, Soviet Physics-Doklady 9 (1965) 523–525.

[3] Barzdin', Ya. M., *Universality problems in the theory of growing automata*, Soviet Physics-Doklady 9 (1965) 535–537.

[4] Bertoni, A., Mauri, G. and Sabadini, N., *Simulations among classes of random access machines and equivalence among numbers succinctly represented*, Ann. Discr. Math. 25 (1985) 65–90.

[5] Chandra, A.K., Kozen, D.C. and Stockmeyer, L.J., *Alternation*, J. Assoc. Comput. Mach. 28 (1981) 114–133.

[6] Dymond, P.W. and Cook, S.A., *Hardware complexity and parallel computation*, Proc. 21st Ann. IEEE Symp. Foundations of Computer Science, 1980, pp. 360–372.

[7] Fortune, S. and Wyllie, J., *Parallelism in random access machines*, Proc. 10th Ann. ACM Symp. Theory of Computing, 1978, pp. 114–118.

[8] Goldschlager, L.M., *A universal interconnection pattern for parallel computers*, J. Assoc. Comput. Mach. 29 (1982) 1073–1086.

[9] Hartmanis, J. and Simon, J., *On the structure of feasible computations*, in Rubinoff, M. and Yovits, M.C. (Eds.), Advances in Computers, Vol. 14 , Acad. Press, New York, 1976, pp. 1–43.

[10] Kolmogorov, A.N. and Uspenskii, V.A., *On the definition of an algorithm*, Uspehi Mat. Nauk 13 (1958) 3–28 ; AMS Transl. 2nd ser. 29 (1963) 217–245.

[11] Lam, T.W. and Ruzzo, W.L., *The power of parallel pointer manipulation*, Proc. 1st Ann. ACM Symp. Parallel Algorithms and Architectures, 1989, pp. 92–102

[12] Luginbuhl, D.R. and Loui, M.C., *Hierarchies and space measures for pointer machines*, Inf. and Comput., 1993, to appear; also: Report UILU-ENG-88-2245, Department of Electr. Engin., University of Illinois at Urbana-Champaign, 1988.

[13] Parberry, I., *Parallel speedup of sequential machines: a defense of the parallel computation thesis*, SIGACT News 18, nr. 1, 1986, pp. 54–67.

[14] Pratt, V.R. and Stockmeyer, L.J., *A characterization of the power of vector machines*, J. Comput. Syst. Sci. 12 (1976) 198–221.

[15] Savitch, W.J., *Recursive Turing machines*, Inter. J. Comput. Math. 6 (1977) 3–31.

[16] Schönhage, A., *Storage modification machines*, SIAM J. Comput. 9 (1980) 490–508.

[17] Slot, C. and van Emde Boas, P., *The problem of space invariance for sequential machines*, Inf. and Comp. 77 (1988) 93–122.

[18] Stegwee, R.A., Torenvliet, L. and van Emde Boas, P., *The power of your editor*, Report RJ 4711 (50179), IBM Research Lab., San Jose, Ca., 1985.

[19] Stockmeyer, L., *The polynomial time hierarchy*, Theor. Comp. Sci. 3 (1977) 1–22.

[20] van Emde Boas, P., *The second machine class 2: an encyclopaedic view on the Parallel Computation Thesis*, in: Rasiowa, H. (Ed.), Mathematical Problems in Computation Theory, Banach Center Publications, Vol. 21, Warsaw, 1987, pp. 235–256.

[21] van Emde Boas, P., *Space measures for storage modification machines*, Inf. Proc. Lett. 30 (1989) 103–110.

[22] van Emde Boas, P., *Machine models and simulations*, in: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science, North-Holland Publ. Comp. 1990, pp. 1–66.

[23] Wagner, K. and Wechsung, G., *Computational Complexity*, Mathematische Monographien Vol. 19, VEB Deutscher Verlag der Wissenschaften, Berlin (DDR), 1986, also: Reidel Publ. Comp., Dordrecht, 1986.

[24] Wiedermann, J., *Parallel Turing machines* , Techn. Rep. RUU-CS-84-11, Dept. of Computer Science, University of Utrecht, Utrecht, 1984.

[25] Wiedermann, J., *Weak parallel machines; a new class of physically feasible parallel machine models*, I.M. Havel & V. Koubek (Eds.), proc. Mathematical Foundations of Computer Science 1992, Springer Lecture notes in Computer Science 629 (1992) pp. 95–111.

6
How to Construct an Atomic Variable

6.1 Introduction.

Communication plays a vital role in any distributed system, allowing multiple processors to share and exchange information. Conventionally this communication is based on mutually exclusive access to a shared *variable*. This is the case not only in a shared memory system, but also at the two endpoints of a link in a message based system. Unfortunately, this exclusive nature of access may force one user of such a variable to wait for another user and therefore impedes the parallelism inherent in distributed systems. In the last years interest has focussed on *wait-free* variables, which can be accessed concurrently without any form of waiting. The question is how to construct such variables in terms of lower-level hardware, like flip-flops.

Peterson was one of the first to investigate this question in [15], giving a construction for a single-writer, multi-reader, multi-bit atomic variable from single-writer, multi-reader atomic bits. Later, Lamport [10], sparked off interest in the subject by developing a precise theory and formalisms—apart from presenting some solutions to subproblems. It is worth noting that most papers use the word "register" instead of variable.

The ultimate goal is to build a variable accessible to any fixed number of users—each having write and read capabilities—which can hold any fixed number of values, and whose accesses behave atomically. The latter means that for any sequence of operations on the variable, the partial precedence order (defined later) among those operations must have a total extension (for external consistency) such that each read operation returns the value from the write operation which is the last to precede it in the total order (for internal consistentency).

The construction of this "ultimate" variable is not done directly from the most primitive kind of variable. Rather, this task is more conveniently split into two subtasks: the construction of atomic, multi-bit, single-writer, single-reader

Paper	used	write	read
[15]	$3b + 10$	$3b + 13$	$3b + 23$
[9]	$4b + 39$	$b + 5$	$b + 26$
this	$4b + 8$	$b + 2$	$b + 4$

TABLE 6.1. worst case number of bits

variables and next the construction of atomic, multi-bit, multi-user variables from the former type.

This partition can be justified by the nature of the problems involved. In the first construction (as in [10]), the multi-bit value is to be *distributed* over a multiple number of bits. In the second construction (as in [20]), the value of the multi-user variable is *replicated* among all users[1] along with control information which allows each user to identify the most recently written value. This is the more complex problem, as witnessed by the fact that many proposed (and often proven) solutions were later found to be erroneous. The interested reader is referred to [2, 8, 14, 16, 20].

In this chapter we attack the first problem, and also give special attention to the case of constructing a single-bit atomic variable.

6.2 Comparison with Related Work

There are basically two approaches that can be taken in order to construct a multi-bit variable from a linear number of single bits. The first was taken by G.L. Peterson in [15] and involves keeping 3 copies of the multi-bit value, called the *tracks* (in the original paper, they are called *buffers*). Apart from the 3 tracks, there are some *control* bits which we collectively call the *switch*. In this approach the writer writes the new value to all three tracks. The reader reads from all tracks, but in a different order. The switch allows the reader to determine which track was read without interference from the writer.

In Kirousis et. al. [9], the second approach was taken. The idea is that the writer and the reader access only a single track, and that the switch ensures that they never access the same track simultaneously. The price to be paid for the reduced number of track-accesses is the necessity of using four tracks.

Both papers mentioned have protocols for multiple readers but we'll consider the case of one only.

In this chapter a simplification of the single reader construction of [9] is presented. Table 6.2 gives a comparison of these constructions for a b-bit atomic variable. The "used" column displays the total number of safe bits used in the construction ("space complexity"). The "write" ("read") column gives the worst case number of safe bits that must be accessed in a write (read) action on the atomic variable. A trade-off between time and space is clearly visible.

We also present a solution to the special case of constructing an atomic bit with a minimal number of non-atomic bits. This problem was solved in-

[1] to have a communication path between any pair of users, we need to maintain $\Omega(n^2)$ copies.

dependently and earlier by J. Burns and G. Peterson [3], using a very similar construction. The use of finite state machines for (automated) correctness verification is related to the work by Clarke and Emerson (e.g. [4]) and is new to this area.

6.3 Preliminaries

In this chapter we consider variables which can be written by one user, called the *writer*, and read by another, the *reader*. Both users may be accessing the variable concurrently without ever having to wait for one another. This means that no assumptions are made about the relative speed of the users, and that the correct operation of the variable is not impaired by halting either one. As stated in the introduction, we aim to construct an atomic, multi-bit, single-writer, single-reader variable. The objects we use in this construction are *safe*, single-bit, single-writer, single-reader variables, or simply *safe bits*. These are the mathematical counterparts of flip-flops, in the sense that real-life flip-flops can be argued to satisfy the *safety* property. Before giving rigorous definitions for the notions of safe and atomic, we first state some preliminary definitions.

In order to distinguish the accesses to the constructed atomic variable (the *higher level*) from the accesses to the safe bits (the *lower level*), we call the former *actions*, and the latter *subactions*. As we will see, each higher-level action is composed of a number of subactions—where the *wait-free* condition requires this number to be bounded.

Let V be a variable and $\mathcal{A}$ (the set of accesses) be the union of a set of writes $\mathcal{W}$ to V and a set of reads $\mathcal{R}$ from V. The result of a read is a value which is said to be *returned* by that read. Each access $a \in \mathcal{A}$ occupies a *time interval* $(s(a), f(a))$, where $s(a)$ is the start time and $f(a)$ the finish time of access a. All start and finish times are assumed to be pairwise distinct. We define a precedence relation $\rightarrow$ on $\mathcal{A}$ as follows: $a \rightarrow b$ iff $f(a) < s(b)$. We say that a *overlaps* b, or a and b overlap, if they are $\rightarrow$-incomparable, that is, $a \not\rightarrow b \wedge b \not\rightarrow a$. *Complete overlap* of a by b means that $s(b) < s(a) < f(a) < f(b)$. We assume that the set $\{a | a \rightarrow b\}$ is finite for any action b (finite history assumption). We call the pair $(\mathcal{A}, \rightarrow)$ a *run*. The writes in $\mathcal{W}$ are totally ordered by $\rightarrow$, and so are the reads in $\mathcal{R}$, in accordance with the requirement that a user can only perform one access at a time.

We relate the reads to the writes in terms of a reading function. A partial function $\pi : \mathcal{R} \rightarrow \mathcal{W}$ is a reading function if for every read $r \in \mathcal{R}$ on which π is defined, $\pi(r)$ writes to V the value returned by r. Unless explicitly stated, the reading function will be total (non-total reading functions will be needed in the definition of safety). We call the triple $(\mathcal{A}, \rightarrow, \pi)$ a *system execution*.

We can now define atomicity:

DEFINITION **6.1** A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ of the variable V is atomic iff there is a total extension $\rightarrow'$ of $\rightarrow$ consistent with π, i.e. for every read $r \in \mathcal{R}$, $\pi(r)$ is the last write preceding r in the total order $\rightarrow'$.

In the case of a single writer, a simplification of the general definition above can be given which avoids the use of a total ordering, [10, 14, 1]:

DEFINITION **6.2** A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ of the single-writer variable

V is atomic iff the following three properties hold for all $r, r_1, r_2 \in \mathcal{R}$ and $w \in \mathcal{W}$:

A0 not $(r \rightarrow \pi(r))$

A1 if $r_1 \rightarrow r_2$, then not $(\pi(r_2) \rightarrow \pi(r_1))$

A2 not $(\pi(r) \rightarrow w \rightarrow r)$

Equivalence of definitions 6.1 and 6.2 is shown by proving ([10])

$$\exists\text{consistent total extension } \rightarrow' \qquad \Leftrightarrow \qquad \rightarrow \text{ satisfies } A0, A1, A2$$

First of all, if any of the three conditions on $\rightarrow$ is violated, then it will clearly be impossible to find a consistent extension $\rightarrow'$, thus proving the first direction. To prove the converse, we construct the following $\rightarrow'$: Merge the reads into the totally ordered set of writes, such that each write w is immediately followed by the reads r with $\pi(r) = w$, putting the reads in some total order that extends their partial precedence order. Naturally, this order is consistent with π. By A1, the merging process preserves the ordering among reads. A0 and A2 ensure that the precedence between a read and a write is also extended. If $w \rightarrow r$, then by A2, $\neg(\pi(r) \rightarrow w)$, so by the construction of $\rightarrow'$ we have $w \rightarrow' r$. If $r \rightarrow w$, then by A0, $\pi(r) \rightarrow w$, so again by the construction of $\rightarrow'$ we have $r \rightarrow' w$. □

Thus, in atomic runs, the partially ordered set of accesses can be linearized while respecting the logical read/write order. In addition to definition 6.2, we have:

Definition **6.3** σ is *regular* iff A0 and A2 hold. σ is *safe* iff $(\mathcal{A} - \mathcal{R}', \rightarrow, \pi)$ is atomic, where $\mathcal{R}' = \{r \in \mathcal{R} | \exists w \in \mathcal{W}(w \not\rightarrow r \wedge r \not\rightarrow w)\}$ is the set of reads which overlap a write (in which case π is left undefined).

Thus, in a safe run, a read overlapping a write may return any value in the domain of the variable. The other actions will then be totally ordered, such that each non-overlapping read returns the value written by the last preceding write.

In a regular run, a read may return the value of either the last completed write or of any of the overlapping writes. Thus, during a long write, one read may obtain the new value, and the next read the old value. This violation of A1 is called a *new-old inversion*, and is what distinguishes a regular run from an atomic one.

Definition **6.4** Variable V is atomic (regular,safe) iff for each of its runs $(\mathcal{A}, \rightarrow)$, there exists a reading function π, such that the system execution $(\mathcal{A}, \rightarrow, \pi)$ is atomic (regular,safe).

It is clear from these definitions that an atomic variable is regular, and that a regular variable is safe. Whereas an atomic bit appears to change its value in a single, indivisible time instant, the value of a changed regular bit *flickers* during the change, and only stabilizes to the complementary value when the changing write finishes.

We call a reading function *normal* if it satisfies A0, i.e. it doesn't map a read to a write which starts after the read finishes. In practice, only normal reading functions are considered.

6.3.1 Making Safe Bits Regular

A well known technique to make a safe bit regular ([10]), is to avoid writing the old value to the bit, i.e. only *changing* it. Then for a read overlapping a write, the value it returns, be it a 0 or a 1, will always be either the value of the last completed write, or the value of an overlapping write. The safe bits used in our constructions are only changed, not overwritten with the old value, and so we can assume for them the existence of a total reading function, as for regular bits.

6.4 Problem Statement

It remains to define what it means to construct an atomic variable from safe bits. (Recall that in this chapter all variables are single-writer, single-reader, hence two users). Such a construction is defined by an *architecture* and a pair of *protocols*, one for each user. The architecture specifies the number of safe bits, their names and how they are connected among the two users. Each safe bit can be connected to the reader and the writer in one of only two ways: changed by the writer and read by the reader, or changed by the reader and read by the writer. The user that can change a bit is said to be the *owner*.

A protocol specifies how the writer (reader) can change (read) the atomic variable in terms of changes to and reads from the safe bits. In addition, the protocols may make use of *local variables*, which can be viewed as safe bits that are changed and read by the same user. These are however not considered part of the architecture, which specifies only *shared* bits.

We consider only *wait-free* protocols, i.e., the number of safe bit accesses in a single protocol execution must be bounded by a fixed constant. This requirement forbids solutions in which a user might have to wait for a safe bit to change value.

A read or write action on the atomic variable consists of an execution of the corresponding protocol. We use the terms "action" and "protocol execution" interchangeably. A construction is initialized by an initial write that sets the atomic variable to the value 0. This allows the definition of a reading function on every read action. All other shared bits and local variables are also initialized to 0. Finally, each run of the construction must satisfy the atomicity criterion.

In the next section we consider the special case of a 2-valued atomic variable. After proving that 3 safe bits are needed to construct an atomic bit, we develop a construction that achieves this lower bound, followed by a proof of correctness. The general case of a b-bit (2^b-valued) atomic variable is dealt with in Section 6.6.

6.5 Optimal Construction of Atomic Bits

6.5.1 A Lower Bound on the Number of Safe Bits needed to Construct an Atomic Bit

We demonstrate that 3 safe bits are required in a construction of an atomic bit, in particular, 2 bits owned by the writer, and 1 owned by the reader.

The reason that a single bit (call it V), owned by the writer doesn't suffice, is exemplified by a run $(\mathcal{A}, \rightarrow, \pi)$, where $\mathcal{A} = \{w, r_1, r_2\}$, and $\rightarrow = \{(r_1, r_2)\}$, i.e. write w (which changes the atomic bit from its initial value 0 to 1), overlaps two reads r_1 and r_2. Without loss of generality, we can assume that the writer changes V at least once during the execution of its protocol. We may also assume that, depending on the values obtained by reading V, but independent of the values of local variables at the start of a read, the reader protocol can return both a 0 and a 1. Consider now the case when both reads occur during the first time that w changes V. Then due to the flickering of V, it is possible for r_1 to return 1 while r_2 returns 0. But now there is no reading function which satisfies all three atomicity conditions.

The example shows that a communication channel from the writer to the reader of only one safe bit is too narrow—at least 2 safe bits are necessary.

In [10], Lamport has shown the necessity of two-way communication, i.e., the reader must own at least 1 bit. The sequence of safe bits changed during the protocol execution of the writer, must therefore depend on information that it receives from the reader.

LEMMA **6.5** *A construction of an atomic bit from non-atomic, safe bits requires at least* 2 *bits owned by the writer, and at least* 1 *bit owned by the reader.*

The next few sections deal with the development of a solution meeting these bounds. Let us first describe the architecture of the construction.

6.5.2 The Architecture

We aim to attain the optimal number of shared safe bits, which is 3. As shown above, the reader will be the owner of one of these, which we'll call "R." One of the two bits owned by the writer will be used to hold the value of the simulated atomic bit and is called "V." The other bit owned by the writer is named "W". To sum up, we have the following 3 safe bits:

6.5.3 The Protocols

In the protocols, we make use of the following statements. The owner of a safe bit B can execute the statement "`change B`" to change its value. Remember that during this change, the value may flicker between 0 and 1. Local variables have lower case names to distinguish them from the shared bits. In all the protocols presented here, the local variables are 2-valued (bits), and are used to hold a copy of V. For this purpose there is a statement "`read loc := V`," whose effect is to read V and store the result in the local variable *loc*. Given the regularity of a changed-only safe bit, we know that the changes to and reads from it obey the conditions A0 and A2 (see definition 6.4), for some reading function. There is the conditional "`if` test `then` statement.", with the obvious semantics. The test is either "`W==R`" or "`W<>R`." Performing such a test is done

by first reading the flag owned by the other user (e.g. the writer reads R in its test). This read is implicit in the test, and is not stated explicitly in the protocol as a separate `read` statement. In order to be able to compare this value against the value of one's own flag, we assume that the owner of a flag keeps track of its value. This abbreviated notation for tests will prove to make the protocols more concise and readable. The final statement in our repertoire is "`return loc`," with *loc* again a local bit. It is used by the reader to exit the execution of its protocol and to specify the return value.

We can now state the protocol:

```
WRITER PROTOCOL                 READER PROTOCOL

change V                        1. if W==R then return v
if W==R then change W           2. read x := V
                                3. if W<>R then change R
                                4. read v := V
                                5. if W==R then return v
                                6. read v := V
                                7. return x
```

6.5.4 Handshaking

A handshaking mechanism is employed to let the reader detect when a change of V is finalised. After changing V, the writer performs a handshake by trying to make $W \neq R$. The basic plan for the reader is to perform a handshake (trying to make $W = R$) in between two reads of V. If afterwards, W is found to still equal R in value, then the read returns the second value, and until the next writer's handshake is detected, future reads can return this same value with no risk of new-old inversions. If, on the other hand, W is found to have been changed by the writer, then the first value, x is returned. In that case, v, which might be returned in line 1 of the next read action, is assigned the current value of V, since that cannot form a new-old inversion with x.

6.5.5 Proof of Correctness

Let $(\mathcal{A}, \rightarrow)$ be a run of the atomic bit construction. Then we can find a lower level regular run for each of the three safe bits, consisting of all the changes to and reads of that safe bit, and the precedence relation defined from their start and finish times. Let π' be a reading function that makes the run on V regular. Let $\mathcal{W}$ be the set of write actions in $\mathcal{A}$, and $\mathcal{R}$ the set of read actions in $\mathcal{A}$. We must prove the atomicity of $\sigma = (\mathcal{A}, \rightarrow, \pi)$ for some reading function π. We define π in a natural way as follows. Let $r \in \mathcal{R}$ be any read action and let *loc* be the local variable returned by r. We can define ρ, the subread of r's return value, as the last subread from V into *loc* before r returns. E.g. if r returns in line 7 then ρ is the read in line 2 of r, and if r returns in line 1, then ρ is the read in line 4 or line 6 of some earlier read action. Let w be the write action which executed $\pi'(\rho)$ in the first line of its protocol. Then we define $\pi(r) = w$.

Proof of A0 Intuitively, since the underlying bits are safe, a read action can only return the value of a past or concurrent (overlapping) write action. We formally prove A0 by contradiction: Assume that for some $r \in \mathcal{R}$, $r \rightarrow \pi(r)$.

Let ρ be the subread of r's return value as above. Then $f(\rho) < f(r)$ and by definition of π, $s(\pi(r)) < s(\pi'(\rho))$. Together these imply that $\rho \to \pi'(\rho)$, which contradicts the safety of π'.

Proof of A1 The proof is again by contradiction. Let $r_1, r_2 \in \mathcal{R}$ be such that $r_1 \to r_2$ and $\pi(r_2) \to \pi(r_1)$. By the finite history assumption, we can let r_1 be the first such read action. Let $\rho_i, i \in \{1, 2\}$ be the subread from V of r_i's return value. For notational convenience, we use the element-of-set symbol $\in$ to denote that a safe bit access is part of a read or write action. Then $\rho_1 \in r_1$ if r_1 returned in line 5 or line 7. Otherwise, if r_1 returned in line 1, then by the minimality of r_1, ρ_1 is in the immediately preceding read action. Defining ω_i as $\pi'(\rho_i)$, we also have $\omega_i \in \pi(r_i)$. From $\pi(r_2) \to \pi(r_1)$ follows $\omega_2 \to \omega_1$. According to the reader protocol, $r_1 \to r_2$ implies $\rho_1 = \rho_2$ or $\rho_1 \to \rho_2$. Since $\pi(r_1) \neq \pi(r_2)$, $\rho_1 \neq \rho_2$ hence $\rho_1 \to \rho_2$. Since the run on V is regular, we have $s(\omega_1) < f(\rho_1)$ (from A0) and $s(\rho_2) < f(\omega_1)$ (from A2). Since ω_1 lasts throughout the time interval $[f(\rho_1), s(\rho_2)]$,

all reads from W between $f(\rho_1)$ and $s(\rho_2)$ obtain the same value. (6.1)

We now consider all three possible cases of the position of ρ_1.

`read x := V` **in line 2** Then r_1 returned in line 7 of its protocol execution, after seeing $W \neq R$ in line 5. However, at that point, the value of R is the same as the value read from W in line 3. Because ρ_2 is either the read in line 6 of this protocol execution, or a later read, we have found a contradiction with (6.1) above.

`read v := V` **in line 4** Then r_1 returned in line 5 of its protocol execution, after seeing W equal to R. Since $\rho_1 \to \rho_2$, ρ_2 must be part of some later read action which sees W different from R in line 1 of its protocol execution. This contradicts (6.1) again.

`read v := V` **in line 6** Then r_1 returned in line 1 of its protocol execution (which immediately succeeds that of ρ_1) after seeing W equal to R. This case therefore reduces to the previous one.

We have shown that the assumed violation of A1 leads to a contradiction.

Proof of A2 The proof is once again by contradiction. Let $r \in \mathcal{R}, w \in \mathcal{W}$ be such that $\pi(r) \to w \to r$. Let ρ be the read from V of r's return value as usual, and ω the write to V in w. From $\pi'(\rho) \in \pi(r)$ and $\omega \in w$ follows $\pi'(\rho) \to \omega$. By regularity of V, $\neg(\omega \to \rho)$, in other words, $s(\rho) < f(\omega)$. Hence $\rho \notin r$, and r must have returned in line 1 of its protocol execution after seeing $W = R$. This means the interval $[f(\omega), s(r)]$ is not free from changes to R, since write w executes line 2 of its protocol in this interval. So R must have been changed between ρ and r. According to the reader protocol, this is done in line 3, and is followed by a `read v := V` statement. This read between ρ and r contradicts the definition of ρ. This completes the proof. □

Lemma 6.5 and the given construction prove the following

THEOREM **6.6** 3 *safe bits are necessary and sufficient to construct a single-reader, single-writer, atomic bit.*

6.6 The 4-track Protocol

We return to the general problem of constructing a b-bit atomic variable with a linear number of safe bits. The safe bits are divided into a number of *control bits*, collectively called the *switch*, and several b-bit tracks, whose purpose is to hold the values of the atomic variable, We start out with a proof of why 4 tracks are necessary, a result due to Burns and Peterson [3]. It is a variation on proofs showing the impossibility of wait-free consensus ([6, 3, 1]).

Consider the initial state of a construction, where both the reader and the writer are about to start their protocol. In this state, the choice of which track the reader will access in its next action is not fixed yet, since a write using a new track could or could not occur before the read starts. Such a state, from which different runs lead to different choices of track to read are called 'bivalent', while states in which the choice is fixed are called 'univalent'. For a wait-free construction, there can be no infinite run of bivalent states (we restrict attention to runs where the read and write protocol are each executed only once). Thus, from the starting state we can reach a bivalent state, say S, both of whose successors S_w and S_r are univalent, where the writer performs subaction x to get from state S to S_w, and the reader perform subaction y to get from state S to S_r. The choice of track is fixed in S_w and S_r, but different. If S_{wr} is the state reached from S_w by a subaction of the reader, then the choice of track in S_{wr} is also different from that in S_r. So the reader's local state must be different between S_{wr} and S_r. It follows that x is a write subaction, and y a read subaction, accessing the same variable. To the writer, states S_w and S_{rw}, reached from S_r by executing x, are indistinguishable, while the reader will access different tracks from these states. If the writer now finishes its protocol and proceeds to write a second and third value, then these 2 new values will have to be written to new tracks to ensure collision-freedom. This shows that 4 tracks are necessary. It remains to show that 4 tracks suffice.

We conveniently split the 4 tracks into 2 groups T_0, T_1 of 2 tracks $T_{i,0}, T_{i,1}$ each. In order to avoid collisions, the writer always tries to go to the group other than where it sees the reader. The reader in turn wants recent values, hence it tries to go to the group where it sees the writer. Both the reader and the writer use part of the switch to signal the other user about the group they are in. For the moment this involves an atomic bit W for the writer, and an atomic bit R for the reader. In addition, the switch has two *trackdisplays* D_0, D_1, one for each group, displaying the most recently completed track. For the moment, these too are atomic bits. Later we will show how to use safe bits instead. Now when the writer completes a write action, the new value will be on track T_{W,D_W}.

In summary, the architecture consists of 4 tracks of b safe bits each and the following 4 atomic bits, which comprise the switch:

We can now informally state the writer protocol. The writer starts by reading R, the group that the reader is in, and compares it to W, the writer's group. If they are equal, then the reader must have left the other group, so the writer simply writes to a track in that other group, and changes W afterwards. It chooses the displayed track so that it doesn't have to change the trackdisplay. If R is different from W, then the writer writes to the other track in its group and changes the trackdisplay D_W afterwards.

The reader protocol is then as follows: The reader starts by reading W to see if the writer has vacated the reader's group. In that case the reader changes R and follows the writer to the other group.

Next, the reader reads the trackdisplay D_R of its group. It then reads the track T_{R,D_R} and returns the obtained value.

In a programming language, not unlike the one introduced in section 6.5.3, the above protocols look like:

```
WRITER PROTOCOL                  READER PROTOCOL

1. if R==w then                  if W<>r then
2.   w := 1-w                      r := 1-r
3.   write track T[w,d[w]]        change R
4.   change W                    endif
5. else                          read d := D[r]
6.   d[w] := 1-d[w]              read track T[r,d]
7.   write track T[w,d[w]]
8.   change D[w]
9. endif
```

The lower-case local variables hold copies of the similarly named shared bits. An array notation is used for the tracks and displays instead of the index notation that we reserve for the text. The access to a track has been compressed to a single statement since we can ignore how many bits must be changed and in what order. For notational convenience, we do not mention the value to be written in the writer protocol or the value to be read in the reader protocol. Since each protocol execution involves exactly one track access, the meaning should be obvious.

Consider a run of the above construction. Each action contains lower-level accesses to the atomic bits of the switch and to the safe bits of a track. By definition 6.1, the partial order on the accesses to each atomic bit can be extended to a total one. Intuitively, the accesses to different atomic bits can then also be totally ordered. In [1], it was shown that this is indeed the case, if the precedence relation is defined in terms of a global time[2]. Using this total ordering on all atomic bit accesses, we can model a run by a sequence of *state transitions*, each transition corresponding to an atomic bit access. In this model, the states of the writer are:

0 idle, i.e., before the atomic read of R in line 1,

[2]This *global time assumption* is equivalent to the *interval axiom*: if $a \rightarrow b \wedge c \rightarrow d$, then $a \rightarrow d$ or $c \rightarrow b$.

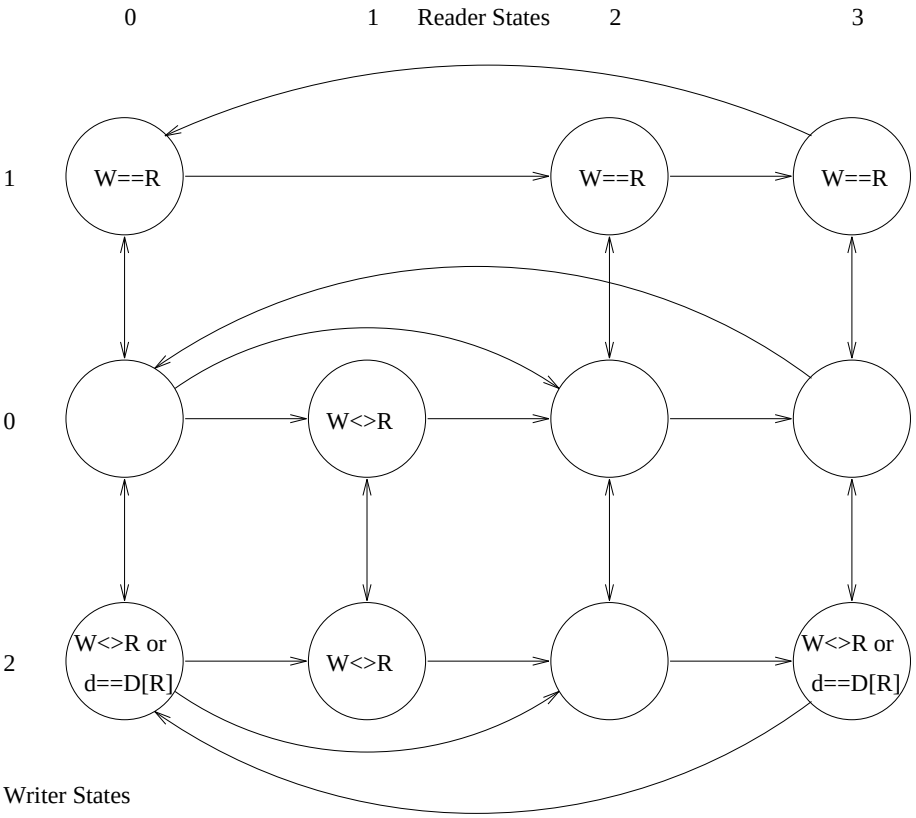


FIGURE 6.1. state diagram of 4-track construction

1 between the atomic read of R and the atomic change of W in line 4, when it is writing track $T_{1-W,D_{1-W}}$,

2 between the atomic read of R and the atomic change of D_W in line 8, when it is writing track $T_{W,1-D_W}$.

Thus, the writer is always moving from state 0 to either state 1 or state 2 (depending on the outcome of the test), and then back to state 0. The states of the reader are:

0 idle, i.e., before the atomic read of W in line 1,

1 between the atomic read of W and the atomic change of R in line 3,

2 just before the atomic read of D_R in line 5,

3 after the atomic read of D_R, when it is reading track $T_{R,d}$.

Thus the reader is always moving from state 0 to either state 1 and then to state 2 or directly to state 2, then on to state 3, and finally back to state 0.

Now figure 6.1 shows all possible transitions in a run of the 4-track construction. It can be easily checked that the invariants in the nodes hold. Note that it is impossible for the writer and the reader to be in state 1 simultaneously.

LEMMA **6.7** *The 4-track construction is collision-free.*

Proof. We denote the combined writer and reader state in a pair (ws, rs). Collisions can only occur in states $(1,3)$ and $(2,3)$, when both the writer and the reader are accessing a track.

In the former case, the writer is in group $w = 1 - W$, while the reader is in group $r = R$. From the diagram we see that $W = R$ in state $(1,3)$, so the users are accessing tracks in different groups.

In state $(2,3)$, the writer writes on track $d_w = 1 - D_W$ in group $w = W$, while the reader reads from track d in group R. The diagram shows that either $W \neq R$ or $d = D_R$, so the users are again accessing different tracks. □.

6.6.1 Correctness

Given lemma 6.7, it remains to show that for every run $(\mathcal{A}, \rightarrow)$, there exists a reading function π such that $\sigma = (\mathcal{A}, \rightarrow, \pi)$ satisfies the three atomicity conditions. As before we may assume that the set of all atomic bit accesses is totally ordered by $\rightarrow$, hence we can use the state model.

Lemma 6.7 allows us to define the reading function π as the "union" of the four reading functions that make each track atomic. This means that a read is mapped to the write which was the last to access the track from which the read obtained its value. We now prove each of the three conditions in turn.

Proof of A0 The reading function is obviously normal by the safety of the track-bits.

Proof of A2 The proof is by contradiction. Let $r \in \mathcal{R}, w \in \mathcal{W}$ be such that $\pi(r) \rightarrow w \rightarrow r$.[3] Assume without loss of generality that w writes on track $T_{0,0}$ and that $D_1 = 0$ at time $f(w)$. Then at the same time, $W = 0$ and $D_0 = 0$.

Consider now the 4 possible tracks that r can read from:

$T_{0,0}$ This contradicts the assumption that $\pi(r)$ precedes w.

$T_{0,1}$ In this case, r reads $d = 1$ from D_0, which requires that the writer changes D_0 to 1 between $f(w)$ and the read of D_0 by r. But according to the writer protocol, this change is preceded by the writing of track $T_{0,1}$, implying $w \rightarrow \pi(r)$ and hence leading to a contradiction.

The last two cases are similar and we need only show that the track read by r must have been written after w.

$T_{1,0}$ In this case, r reads 1 from W, which requires that the writer changes W to 1. This is preceded by the writing of track $T_{1,0}$.

$T_{1,1}$ In this case, r reads $d = 1$ from D_1, which requires that the writer changes D_1 to 1. This is preceded by the writing of track $T_{1,1}$.

In all three cases, we see that r cannot read a value older than that of w, because the display (W, D_0, D_1) doesn't change until the new track has been written. In other words, once the display is set, every new read action must read either the track on display or a more recently written one.

[3] It will be clear from context whether we mean the write action w or the similarly denoted writer's local copy of W.

Proof of A1 We claim that A1 follows from A2 and show this by deriving a violation of A2 from a violation of A1. Let $r_1, r_2 \in \mathcal{R}$ be such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$. By definition of π and lemma 6.7, r_1 accesses some track, say $T_{0,0}$, after $\pi(r_1)$ does so. But since r_1 cannot access the track until after $\pi(r_1)$ changes an atomic bit (W or D_0), we have that $f(\pi(r_1)) < f(r_1) < s(r_2)$, hence with $w = \pi(r_1)$, $\pi(r_2) \rightarrow w \rightarrow r_2$, violating A2. □

6.6.2 Space Complexity

Now that the 4-track construction has been proven correct, we consider its "space complexity." Using the 3 safe bit construction to implement each of the four atomic bits, we see that 12 bits suffice for the switch. But we can do better, because those atomic bits are used in a special way. In particular, since the W and R bits are used for handshaking, there is exactly one atomic read of W between an atomic change of W and an atomic change of R (and vice versa). Hence there is at most one atomic change of W between two consecutive atomic reads of W (and vice versa). With the trackdisplay bits D_0, D_1 the situation is more complicated. When the reader changes groups (say, to 0), and atomically reads D_0, there can be at most one atomic change of D_0 before the writer leaves group 0.

We will show that, because of these properties, we can implement any of the four atomic bits, call it B, with 2 safe bits B_0, B_1. The problem with safe bits is their flickering. If, for example, R was only a safe bit, then while being changed by the reader, the writer could first see the new value, change groups, then see the old value and write to the displayed track in the old group. With 2 safe bits, the following scheme can be applied to alleviate the flickering problem. We represent the value of atomic bit B as the exclusive-or of 2 safe bit values: $B = B_0 \oplus B_1$. The change of atomic bit B is then replaced by a change of safe bit B_b, where b is the old value of B. Thus, B_0 and B_1 are changed alternatingly. For the purpose of reading B, two local copies b_0, b_1 of B_0 and B_1 are kept. Normally then, an atomic read of B is replaced by a safe read of B_b into b_b, where $b = b_0 \oplus b_1$ is the old value of B. In this case, new-old inversions are eliminated, since the flickering bit is no longer examined once the new value is obtained. This procedure suffices for reading W and R, since the handshaking ensures that each safe bit change is noticed by the other user. It also suffices if the reader sees the writer in the same group and wants to read the trackdisplay, because the writer will change the display at most once (before moving to the other group). If on the other hand the reader sees the writer in the other group, then any local copies it would have of the trackdisplay bits in that other group might be out of date. In this case it can simply read both safe bits of that display one after the other, because again the writer will change the display at most once before moving to the other group.

The new architecture of the switch is as follows:

Writer $\rightarrow$ $D_{1,1}$ $\rightarrow$ **Reader** group 1 trackdisplay

Writer $\rightarrow$ W_0 $\rightarrow$ **Reader**

Writer $\rightarrow$ W_1 $\rightarrow$ **Reader** writer's group

Writer $\leftarrow$ R_0 $\leftarrow$ **Reader**

Writer $\leftarrow$ R_1 $\leftarrow$ **Reader** reader's group

The corresponding protocols are:

```
WRITER PROTOCOL                  READER PROTOCOL

1. if R[1-w]==W[1-w] then        if W[r]<>R[r] then
2.    w := 1-w                     change R[r]
3.    write track T[w,x[w]]        r := 1-r
4.    change W[1-w]                read d[0] := D[r,0]
5. else                            read d[1] := D[r,1]
6.    x[w] := 1-x[w]             else read d[d[0]⊕d[1]]
7.    write track T[w,x[w]]           := D[r,d[0]⊕d[1]]
8.    change D[w,1-x[w]]         endif
9. endif                         read track T[r,d[0]⊕d[1]]
```

The writer's local variables d_0, d_1 have been renamed to x_0, x_1 to emphasize that x_i now represents the eXclusive-or of $D_{i,0}$ and $D_{i,1}$. The reader's local variable d has been replaced by d_0 and d_1, where d_i is meant to hold a copy of $D_{r,i}$. All shared and local variables are initialized to 0 as usual. Because the switch now consists of eight safe bits, we call it the "Safe Byte Switch."

We can now state the main theorem:

THEOREM **6.8** *A single-reader, single-writer b-bits atomic variable can be constructed from* $4b + 8$ *safe bits (4 tracks and a safe byte).*

We postpone the proof of correctness of the new construction to Section 6.7.3.

6.7 The Atomicity Automaton

In this and the next few sections we discuss the use of machines (computers) as an aid in designing and verifying atomic variable constructions.

The verification is based on a generic automaton which embodies the three atomicity properties of system executions (in the single-reader, single-writer case). Figure 6.2 shows a picture of the automaton. The transitions of this automaton represent the starts and ends of read and write actions, while the nodes represent the "atomicity state" of a run on the atomic variable. The latter corresponds to the set of values that the next-ending read can return without violating atomicity—its size is shown inside each node.

The nodes can be divided into four groups, depending on whether each user is idle or busy accessing the variable. When both users are idle, the atomicity state of the run is fixed by the current value of the variable—this being the

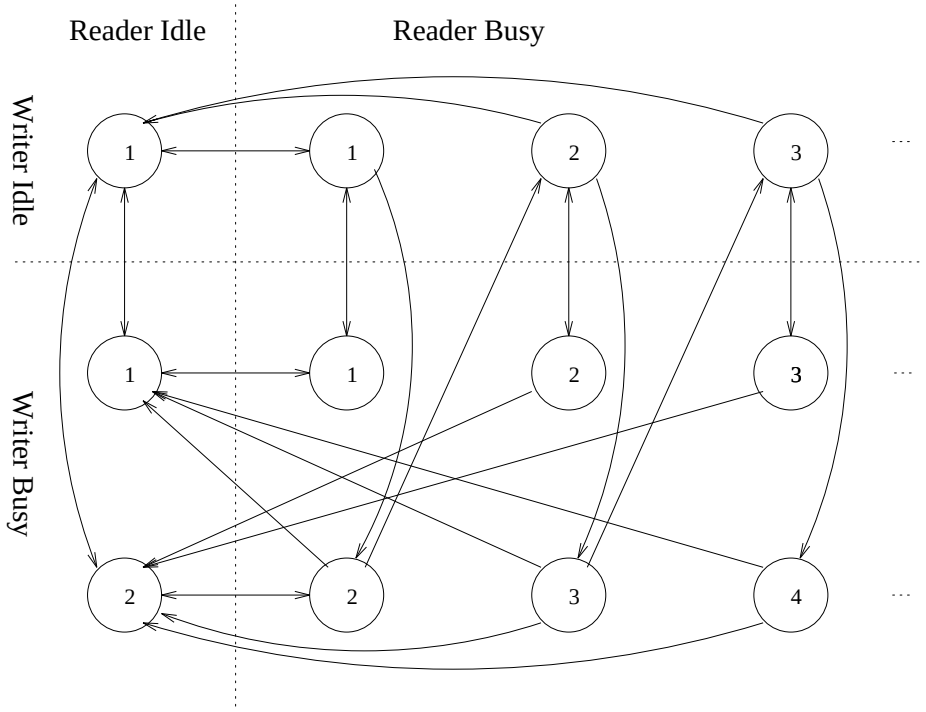


FIGURE 6.2. the general atomicity automaton

only value that a newly started read is allowed to return. This explains the single node in this group.

When the writer is busy and the reader idle, we can distinguish between two states: either the reader has read the new value that is being written, or it hasn't. Hence there are two nodes in this group. In the former case subsequent read actions must return the same value as the last read action in order to prevent new-old inversions (condition A1). Hence the set size of one.

When the reader is busy, there are many possible states, depending on the number of writes that overlap the read. As the nodes progress to the right, the set of values that the current read action is allowed to return grows. Of course, while the picture suggests an infinite progression of nodes, its size is in fact limited by the number of values that the atomic variable can hold (its domain-size).

In the group where only the reader is busy, there are two start-of-write transitions emanating downward from each node. As can be deduced from the resulting set sizes, the upper transition corresponds to the write of a value already in the set of permitted return values. In this case, while the set size remains the same, it is now no longer required to map the read action to the current write action (if the read action decides to return its value). This means that the next read action will not be able to combine with the current one to create a new-old inversion.

Alternatively, if the value of the new write is outside the set, then this value is added to it, but if now the read decides to return the new value, then the atomicity state represented by the middle-left node is reached. The other leftward

transitions from the right-bottom nodes to the left bottom node correspond to return values written by earlier write actions.

6.7.1 Using the Automaton for Verification of a given Run

For verification, we include in the state information the actual set of values of *completed write actions* that are valid return values for the next ending read. The value returned by a read can then be verified as follows: If it is in the above set, then we take the leftward transition to the the bottom-left or top-left node (in case writer is idle), and reduce the set to contain only the value of the last completed write. Otherwise, if it is the value currently being written by the writer, then we take the leftward transition to the middle-left node and empty the set (since now even the value of the last completed write is invalid). If the returned value satisfies neither of these cases, then the run is non-atomic. The set is further maintained at the completion of a write, by either adding the written value to the set if the reader is busy, or changing the set to the singleton with that value if the reader is idle.

For atomic bit constructions, we know that the values written are alternatingly 1 and 0. This means that the size of the set of permitted return values is bounded by 2. The size of the atomicity automaton shrinks acccordingly. In the group with only the reader busy, there are only two essentially different nodes—either the reader can return the current value of the atomic bit, or it can return both 0 and 1. In the group where both users are busy, there are only three nodes. In one, it can return either the old or the new value with two different transitions. In the second, it can return both 0 and 1 as old values so there is only one such transition. In the third node, it must return the new value. Figure 6.3 shows this reduced automaton, with explicit mention of which value is returned by a read (if the writer is idle, then "new" means "current.") There are two nodes from which a single new-labelled transition emanates to the left. From these nodes atomicity can be violated if the read returns the other value.

6.7.2 Verifying the Atomic Bit Construction

A program has been written to systematically search all states of the atomic bit construction. The state information involves the following:

- position of writer in its protocol, i.e., writer state
- position of reader in its protocol, i.e., reader state
- values of the reader's local variables
- values of the three safe bits
- position in automaton, i.e., atomicity state

We now explain how the safety of the shared bits is modelled. A safe write is modelled by two separate transitions representing the start and the finish of that write. A read, on the other hand, is represented by a single transition, as if it occurred in a single time instant. This can be done for the following reason. If a read from a safe bit overlaps a write on the same bit, then either 0 or 1 can

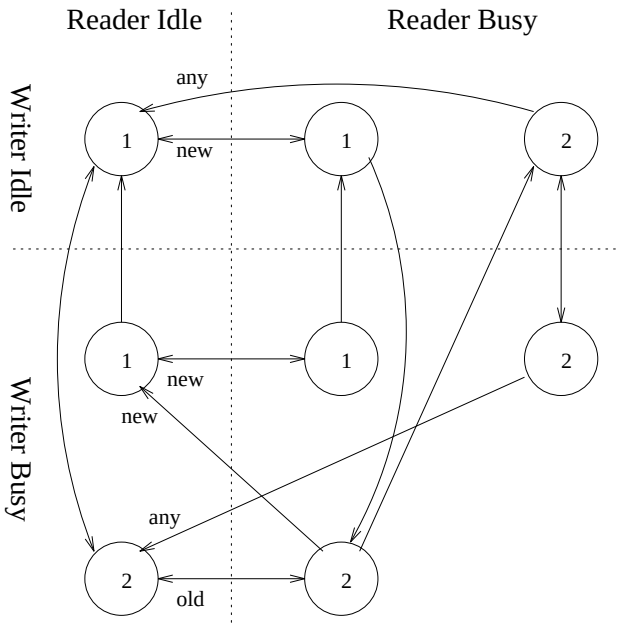


FIGURE 6.3. the atomic bit automaton

be returned, so the read might as well have occurred completely within that write. If no write overlaps the read, then the value returned must be that of the last preceding write, and it clearly doesn't matter how long the read lasts.

In summary, if a read occurs between two consecutive writes, then there is a single transition corresponding to the return of the current value, and if it occurs between the start and finish of a write, then there are two different transitions, one for each value that can be returned.

This model captures the essence of safe bits. It leads to 3 writer states and 7 reader states. The program starts by putting the initial state in an otherwise empty set. Then it repeatedly takes an element from the set, and replaces it by all states that result from the removed one by a single transition and are not yet in the set. Additionally, the program keeps track of the shortest path from the initial state to each visited one. If some transition is the return of a value which is invalid according to the automaton, then the program prints out a description of the shortest path to the failing state, revealing the shortcomings of the construction being verified. Otherwise, if the set becomes empty, then some statistics are printed such as the number of visited states for each combination of writer state and reader state.

The program helped the design of the atomic bit construction by making it easy to try out various alternatives, and immediately getting a diagnosis of possible problems.

6.7.3 Verifying the Safe Byte Switch Construction

Like in the proof of the 4-track construction with the 4 atomic bits, we must first establish that the new construction is collision free, that is, we must prove

lemma 6.7 again. For this we again need an invariant to hold under all possible runs based on a state diagram. In this case however, we cannot assume that the switch bit accesses can be linearized, since they are only safe. Instead we adopt the safe bit model of the previous section. This entails redefining the reader and writer states. Since the simplicity of the invariants depends rather heavily on the exact form of the protocols. we base the proof on the following reformulated protocols, that are semantically equivalent to those of Section 6.6.2 (local assignments have moved and indices changed accordingly):

```
WRITER PROTOCOL                    READER PROTOCOL

1. if R[1-w]==W[1-w] then          if W[r]<>R[r] then
2.   write track T[1-w,x[1-w]]      r := 1-r
3.   w := 1-w                      change R[1-r]
4.   change W[1-w]                 read d[0] := D[r,0]
5. else                            read d[1] := D[r,1]
6.   write track T[w,1-x[w]]       else read d[d[0]⊕d[1]]
7.   x[w] := 1-x[w]                     := D[r,d[0]⊕d[1]]
8.   change D[w,1-x[w]]            endif
9. endif                           read track T[r,d[0]⊕d[1]]
```

Note that when a safe bit is changed, the local copy already holds the new value—this is the property that ensures the most simple invariants. We proceed to enumerate the essential positions of the users in their protocols.

The states of the writer are:

0 idle, i.e., before the safe read of R_{1-w} in line 1

1 between the safe read of R_{1-w} and the safe change of W_{1-w} in line 4, when it is writing track $T_{1-w,x_{1-w}}$

2 changing safe bit W_{1-w} in line 4

3 between the safe read of R_{1-w} and the safe change of $D_{w,1-x_w}$ in line 8, when it is writing track $T_{w,1-x_w}$

4 changing safe bit $D_{w,1-x_w}$ in line 8

Thus, the writer is always moving from state 0 to either state 1 followed by state 2 or to state 3 followed by state 4 (depending on the outcome of the test), and then back to state 0.

The states of the reader are:

0 idle, i.e., before the safe read of W_r in line 1

1 between the safe read of W_r and the safe read of $D_{r,0}$ in line 4, when it is changing safe bit R_{1-r}

2 between the safe read of $D_{r,0}$ and the safe read of $D_{r,1}$ in line 5

3 between the safe read of W_r and the safe read of $D_{r,d_0\oplus d_1}$ in line 7

4 reading track $T_{r,d_0\oplus d_1}$ in line 9

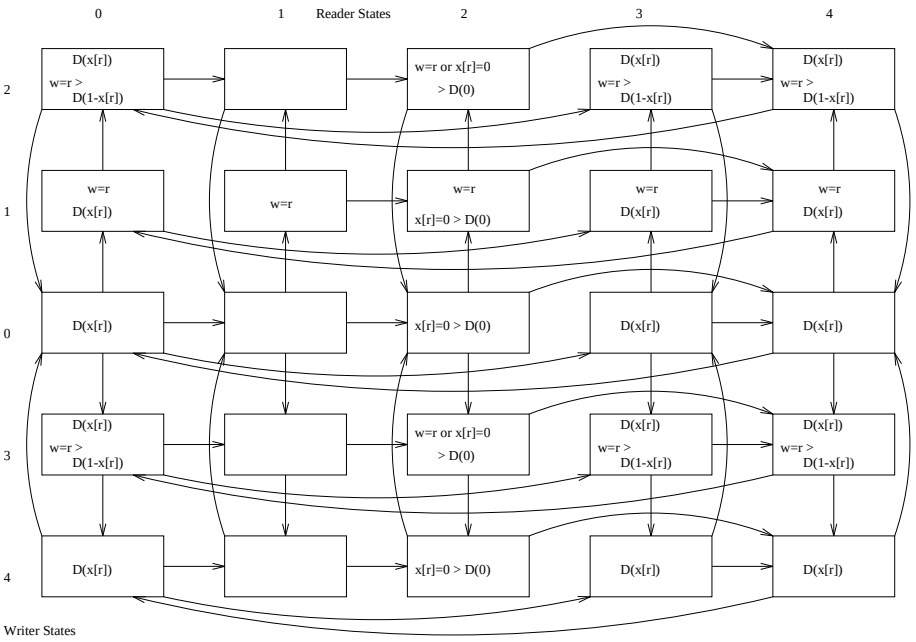


FIGURE 6.4. state diagram of 4-track construction with safe byte switch

Thus the reader is always moving from state 0 to either state 1, followed by state 2 or to state 3, then on to state 4, and finally back to state 0.

Altogether, there are now $5 \times 5 = 25$ states, which are pictured in figure 6.4, along with all possible transitions. Each node contains a (possibly empty) set of formulas, which are to be conjuncted, together with the invariant $W_w = R_w$ which holds for all nodes. To solve the potential ambiguity of this formula which arises when the reader is changing R_w (the writer changes W_{1-w}), we make the following definition:

DEFINITION **6.9** If a safe bit B is being changed then in formulas, B refers to its *new* value.

The notation $D(i)$ is used as an abbreviation of $d_i = D_{r,i}$ and expresses that a local display bit of the reader matches the shared one (in the reader's group). A greater-than sign ($>$) is used to denote implication. Starting from the initial state $(0, 0)$ (both users idle), each invariant can be manually checked by considering the possible predecessors of a node.

Another state space search program was used to construct this diagram. It's state also includes information on the validity of the tracks (relative to the next end-of-read), as derived from the atomicity automaton. With the help of this information, the program actually verifies the correctness of the construction. In this chapter however, we neglect this extra state, since including it would make the diagram overly complex, making manual inspection practically impossible.

It is now easy to see from the diagram that lemma 6.7 holds. Potential collisions can occur in states $(1, 4)$ and $(3, 4)$. In the first case, $w = r$ and the writer writes on a track in group $1-w$, so there is no collision. In the second case, the writer writes on track $T_{w,1-x_w}$. If $w \neq r$, then we are done. Otherwise, the

reader and writer are in the same group, so $w = r$, implying $d_{1-x_r} = D_{r,1-x_r}$. Along with $d_{1-x_r} = D_{r,1-x_r}$, this leads to $x_r = D_{r,0} \oplus D_{r,1} = d_0 \oplus d_1$, so the reader is on the other track—no collision.

6.8 Correctness of Safe Byte Switch Construction

Given lemma 6.7, it remains to show that for every run $(\mathcal{A}, \rightarrow)$, there exists a reading function π such that $\sigma = (\mathcal{A}, \rightarrow, \pi)$ satisfies the three atomicity conditions. We choose the reading function π to map a read action to the write action that last writes to the track before the read action reads from that track. As before, this can be viewed as the union of the four reading functions that make each track atomic, according to lemma 6.7.

We now prove each of the three conditions in turn.

Proof of A0 The reading function is obviously normal by the safety of the track-bits.

Proof of A1 The proof is by contradiction. Let $r_1, r_2 \in \mathcal{R}$ be such that $r_1 \rightarrow r_2$ are two consecutive read actions and $\pi(r_2) \rightarrow \pi(r_1)$. Assume that $\pi(r_1)$ writes on track $T_{0,0}$, that $W_0 = W_1 = R_0 = 0$ (using the invariant $W_w = R_w$) and that $D_{1,0} = D_{1,1} = 0$ at time $f(\pi(r_1))$.[4]

Consider now the 4 possible tracks that r_2 can read from:

$T_{0,0}$ This contradicts the assumption that $\pi(r_2)$ precedes $\pi(r_1)$, which cannot be the case if r_1 and r_2 read from the same track.

$T_{0,1}$ Note that r_2 didn't change groups thus taking the else branch. Examination of figure 6.4 reveals that $d_0 = D_{0,0}$ at time $f(r_1)$ (Recall that $x_i = D_{i,0} \oplus D_{i,1}$). In order for r_2 to read track $T_{0,1}$, it must have seen a change in $D_{0,0}$, which it reads in line 7. But $\pi(r_1)$ ends by changing either W_1 (line 4) or $D_{0,1}$ (line 8). Hence a write action later than $\pi(r_1)$ started changing $D_{0,0}$ before r_2 accessed its track, and this write action must have written to that track, contradicting $\pi(r_2) \rightarrow \pi(r_1)$.

$T_{1,0}$ In this case r_2 did change groups, taking the then branch. So in line 1, it saw W_0 set ($W_0 \neq R_0 = 0$). Given that $\pi(r_1)$ doesn't change W_0, a later write action must have started changing it, following the writing on track $T_{1,0}$. This again contradicts $\pi(r_2) \rightarrow \pi(r_1)$.

$T_{1,1}$ Again r_2 changed groups and took the then branch. Also, it saw either $D_{1,0}$ or $D_{1,1}$ set, which requires that a write action later than $\pi(r_1)$ has already scribbled on track $T_{1,1}$, contradicting $\pi(r_2) \rightarrow \pi(r_1)$.

Proof of A2 The proof is once again by contradiction. Let $r \in \mathcal{R}, w \in \mathcal{W}$ be such that $\pi(r) \rightarrow w \rightarrow r$. Assume that w writes on track $T_{0,0}$, that $W_0 = W_1 = R_0 = 0$ and that $D_{1,0} = D_{1,1} = 0$ at time $f(w)$.

Consider now the 4 possible tracks that r can read from:

$T_{0,0}$ This contradicts our choice of the reading function π, since $\pi(r)$ must either equal w or succeed it.

[4] Other cases are analogous.

$T_{0,1}$ Since $\pi(r) \rightarrow w$, track $T_{0,1}$ is not written to between w and its access by r. Study of the writer protocol then shows that $D_{0,0}$ and $D_{0,1}$ remain constant during that time. Because w and r access different tracks, r did not read both $D_{0,0}$ and $D_{0,1}$—it took the else branch. From figure 6.4 we obtain that, with r in state 3, $d_{x_r} = D_{r,x_r}$. But then either d_{1-x_r} already equals $D_{r,1-x_r}$, or it will do so after the read in line 7, in contradiction with $\pi(r) \neq w$.

$T_{1,0}$ Since $\pi(r) \rightarrow w$, track $T_{1,0}$ is not written to between w and its access by r. Study of the writer protocol now shows that W_0 and W_1 remain 0 during that time. But then in line 1, r reads either W_1 (and moves to group 0), or W_0, in which case it remains in group 0, a contradiction.

$T_{1,1}$ Since $\pi(r) \rightarrow w$, track $T_{1,1}$ is not written to between w and its access by r. Study of the writer protocol now shows that $D_{1,0}$ and $D_{1,1}$ remain 0 during that time. We conclude that r takes the else branch. Again with r in state 3, we have $d_0 = d_{x_r} = D_{r,x_r} = 0$. But then either already $d_1 = 0$, or this will hold after the read in line 7, in contradiction with $\pi(r) \neq w$.

□

6.9 Conclusions

We have presented and proven correct the following two constructions:

- an atomic bit from 3 safe bits
- an atomic b-bit variable from $4b + 8$ safe bits

The first achieves the optimal number of non-atomic bits needed (optimal space complexity). The second needs only 2 extra bit accesses in a write action, and at most 4 extra bit accesses in a read action on the atomic variable (in addition to the b accesses to the bits on a track), making its time complexity very near (if not equal) to optimal. The cost for this "speed" is in the space complexity, which is about a factor 4/3 from optimal, since Peterson showed the sufficiency of 3 tracks. A main advantage of the 4-track construction as given here, is its simplicity and transparency—the purpose of the bits in the architecture and the workings of the protocols can be easily understood. We have developed a finite state verification methodology for concurrent wait-free shared variable constructions, whose succesful application provides additional practical support.

Bibliography

[1] K. Abrahamson, *On achieving consensus using shared memory*, Proc. 7th ACM Symposium on Principles of Distributed Computing, 1988, pp. 291–302.

[2] B. Bloom, *Constructing Two-writer Atomic Registers*, IEEE Transactions on Computers, vol. 37, pp. 1506–1514, 1988.

[3] B. Chor, A. Israeli, M. Li, *On processor coordination using asynchronous hardware*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 86–97, 1987.

[4] E.M. Clarke, E.A.Emerson, A.P. Sistla, *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach*, Proc. 10th ACM Symposium on Principles of Programming Languages, pp. 117–126, 1983.

[5] L. Lamport, *On Interprocess Communication Parts I and II*, Distributed Computing, vol.1, 1986, pp. 77–101

[6] M. Loui, H.H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, pp. 163–183 in: Advances in Computing Research, JAI Press, 1987.

[7] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *Atomic Multireader Register*, Proc. 2nd International Workshop on Distributed Computing, Springer Verlag Lecture Notes in Computer Science 312, pp. 278–296, 1987.

[8] M. Li, J. Tromp, P.M.B. Vitányi, *How to Share Concurrent Wait-Free Variables*, Chapter 7 (also under revision for *Journal of the ACM*).

[9] B. Awerbuch, L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *On Proving Register Atomicity*, Proc. 8th Conference on Foundations of Software Tech-

nology and Theoretical Computer Science, Springer Verlag Lecture Notes in Computer Science 338, pp. 286–303, 1988.

[10] G.L. Peterson and J.E. Burns, *Concurrent reading while writing II: the multiwriter case*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 383–392, 1987.

[11] G.L. Peterson and J.E. Burns, *The Ambiguity of Choosing*, Proc. 8th ACM Symposium on Principles of Distributed Computing, pp. 145–157, 1989.

[12] G.L. Peterson, *Concurrent reading while writing*, ACM Transactions on Programming Languages and Systems, vol.5, No.1, 1983, pp. 46–55

[13] J.E. Burns, G.L. Peterson, *Sharp Bounds for Concurrent Reading While Writing*, Technical Report, Georgia Institute of Technology GIT-ICS-87/31

[14] R. Schaffer, *On the correctness of atomic multi-writer registers*, Technical Report MIT/LCS/TM-364, MIT lab. for Computer Science, June 1988.

[15] P.M.B. Vitányi, B. Awerbuch, *Atomic Shared Register Access by Asynchronous Hardware*, Proc. 27th IEEE Symposium on Foundations of Computer Science, pp. 233–243, 1986. (Errata, Ibid.,1987)

7

How to Share Concurrent Wait-Free Variables

7.1 Introduction

In [10] Lamport has shown how an atomic variable—one whose accesses appear to be indivisible—shared between one writer and one reader, acting asynchronously and without waiting, can be constructed from lower level hardware rather than just assuming its existence. There arises the question of the construction of multi-user atomic variables of this type (see [20], on which the current chapter is based). In this chapter we will supply a uniform solution to such problems, given Lamport's construction, and derive the implementations by transformations from the specification.

7.1.1 Informal Problem Statement and Main Result

Usually, with asynchronous readers and writers, atomicity of operations is simply assumed or enforced by synchronization primitives like semaphores. However, active serialization of asynchronous concurrent actions always implies waiting by one action for another. In contrast, our aim is to realize the maximum amount of parallelism inherent in concurrent systems by avoiding waiting altogether in our algorithms. In such a setting, serializability is *not* actively enforced, rather it follows from the way the executions of the algorithm by the various processes interact. Any one of the references, say [10] or [20], describes the problem area in some detail.

The point of departure is the solution of the following problem. (We keep the discussion informal.) Consider two processors that are asynchronous and do not wait for one another. A flip-flop is a boolean variable that can be read by one processor and written by the other. Suppose, one is given atomic flip-flops as building blocks, and is asked to implement a k-bit atomic variable, that can be written by one processor and read by the other. Of course, a buffer consisting of k flip-flops suffices to hold such a value. If, however, the implementation

allows the reader to read and return the value held by the same buffer that the writer may simultaneously access for writing, then either the writer or the reader might do all of its accesses while the other is sleeping halfway. Thus, the reader would obtain a value consisting of half the new value and half the old one. Obviously, this violates atomicity. The problem then is to design a protocol that provides exclusive access to buffers without waiting. Correct implementations of atomic multi-bit variables from single bits can be found in [15, 10, 26].

These atomic variables serve as the building blocks of our construction of an n-user variable; a variable shared between n users each of which can atomically execute both read and write operations.

At the outset we state our main result:

THEOREM **7.1** *An atomic n-user variable is implemented wait-free from $O(n^2)$ atomic 1-reader 1-writer variables each with $O(n)$ control bits; using $O(n)$ accesses per Read/Write running in $O(1)$ parallel time.*

Our notion of parallel time allows a set of accesses to different variables to proceed in arbitrary order in one time-unit.

7.1.2 Comparison with Related Work.

Related constructions are given by [17, 9, 3, 13, 7] for the single-reader to multi-reader case, and by [20, 14, 16, 8] for the multi-reader to multi-writer case. (see also Section 7.6.) The latter problem is the more difficult one. The solutions in references [20, 14] are known to be incorrect [16]. There has been no previous attempt to implement an n-user variable directly from single reader variables.

The algorithm uses $O(n)$ accesses to single-reader variables per operation, and each single-reader variable stores two copies of the value of the constructed variable together with $O(n)$ bits of control information. In order to compare our algorithm to one using n multi-reader variables, the latter could be combined with n copies of a single-reader to multi-reader algorithm. This doesn't increase the number of variables used, since one single-reader register from each process to each other process is both necessary and sufficient. It does however increase their size, as well as the number of accesses per operation, by a factor of n. This should be kept in mind with the comparisons below.

The multi-writer algorithm in [16], which patches [14], uses $\Theta(n^2)$ accesses to multi-reader variables per operation. Following [7], recent work [4, 21, 22, 23, 24] provides a general method for replacing unbounded timestamps by bounded timestamps in concurrent systems of multi-reader variables. The best construction, [24], uses multi-reader variables of size $\Theta(n)$, and $\Theta(n)$ accesses per operation, and can be applied directly to give a multi-writer variable.

A more recent construction than ours, [8], comes up with a direct solution that is optimal in space (logarithmic control bit complexity) as well as number of variable accesses per read/write (linear). They do not however achieve constant parallel time and have a rather more complicated protocol. We believe the construction presented here is relatively simple and transparent. Both problems above are solved by simplifications of our main solution.

The basis of our proof-technique was developed in [1]. Our model and terminology is based on [6], which defines and motivates the notion of linearizability. Bloom [2] presented an elegant 2-writer construction. In [5], Herlihy consid-

ers more powerful shared objects that have no wait-free implementations from variables.

7.1.3 Multi-user Variable Construction

In this section we consider the problem of constructing an n-user variable from single-reader variables and state the correctness condition such a construction has to satisfy.

Throughout the chapter, the n users are indexed with the set $I = \{0, \ldots, n-1\}$. The variable constructed will be called ABS (for abstract).

A construction consists of a collection of shared variables $R_{i,j}, i, j \in I$, and two procedures, *Read* and *Write*. Both procedures have an input parameter i, which is the index of the executing user, and in addition, Write takes a value to be written to ABS as input. A return statement must end both procedures, in the case of Read having an argument which is taken to be the value read from ABS.

A procedure contains a declaration of local variables and a body. A local variable appearing in both procedures can be declared *static*, which means it retains its value between procedure invocations. The body is a program fragment comprised of atomic statements. Access to shared variables is naturally restricted to assignments from $R_{j,i}$ to local variables and assignments from local variables to $R_{i,j}$, for any j (recall that i is the index of the executing user). No other means of inter-process communication is allowed. In particular, no synchronization primitives can be used. Assignments to and from shared variables are called writes and reads respectively, always in lower case. The space complexity of a construction is the maximum size, in bits, of a shared variable.

A loop of the type **for** $j \in I$ signifies a parallel loop whose iterations[1] can be executed in arbitrary order. Shared variables accessed in different iterations of a parallel loop must be disjoint since a user can only execute one operation at a time on a given shared variable. As the time complexity of the Read or Write procedure we take the maximum number of shared variable accesses outside parallel loops plus for each such loop the maximum number of shared variable accesses in a single iteration of that loop.

A construction must satisfy the following constraint.

Wait-Freeness Each procedure must be free from unbounded loops.

Given a construction, we are interested in properties of its executions, which the following notions help formulate. A *state* is a configuration of the construction, comprising values of all shared and local variables, as well as program counters. Note that we need a somewhat liberal notion of program counter to characterize the execution of a parallel loop. In between invocations of the Read and Write procedure, a user is said to be idle, and its program counter has the value 'idle', One state is designated as *initial state*. All users must be idle in this state.

A state t is an *immediate successor* of a state s if t can be reached from s through the execution of a procedure statement by some user in accordance

[1]This is a slight abuse of the term, since the word iteration suggests sequential behaviour.

with its program counter. Recall that n denotes the number of users of the constructed variable ABS. A state has at least n immediate successors: If a user is idle, it can invoke either the Read or Write procedure. And if it is within one of these procedures, there is at least one atomic statement to be executed next (possibly more during the execution of a parallel loop).

A *history* of the construction is a finite or infinite sequence of states $t_0, t_1, t_2, \ldots$ such that t_0 is the initial state and t_{i+1} is an immediate successor of t_i. Transitions between successive states are called the *events* of a history. With each event is associated the index of the executing user, the relevant procedure statement, and the values manipulated by the execution of the statement.

An event a *precedes* an event b in history h, $a \prec_h b$, if a occurs before b in h. The subscript h is omitted when clear from context. Call a finite set of events of a history an event-set. Then we similarly say that an event-set a precedes an event-set b in a history, $a \prec_h b$, when each event in a occurs before all those in b. The relation $\prec_h$ on event-sets constitutes what is known as an *interval order*. That is, a partial order satisfying the interval axiom $a \prec b \wedge c \prec d \wedge c \not\prec b \Rightarrow a \prec d$. This implication can be seen to hold by considering the last event of c and the earliest event of b. See [10] for an extensive discussion on models of time.

Of particular interest are the sets consisting of all events of a single procedure invocation, which we call an *operation*. An operation is either a Read operation or a Write operation. It is *complete* if it includes the execution of the final **return** statement of the procedure. Otherwise it is said to be *pending*. A history is complete if all its operations are complete. Note that in the final state of a complete finite history, all users are idle. The *value* of an operation is the value written to ABS in the case of a Write, or the value read from ABS in the case of a Read.

The following crucial definition expresses the idea that the operations in a history appear to take place instantaneously somewhere during their execution interval. A more general version of this is presented and motivated in [6]. To avoid special cases, we introduce the notion of a *proper* history as one that starts with an initializing Write operation that precedes all other operations.

Linearizability A complete proper history h is *linearizable* if the partial order $\prec_h$ on the set of operations can be extended to a total order which obeys the semantics of a variable. That is, each Read operations returns the value written by that Write operation which last precedes it in the total order.

A construction is *correct* if it satisfies Wait-Freeness and all its complete proper histories are linearizable.

7.1.4 The Tag Function

While the definition of linearizability is quite clear, it is convenient to transform it into an equivalent specification from which the first algorithm can be directly derived. The idea behind the following lemma was first expressed by Lamport in Proposition 3 of [10], for the case of a single writer. In [17], the equivalent conditions given by Lamport's proposition are in fact taken as the definition of linearizability (often called atomicity in the register construction literature).

The Atomicity Criterion of [1] is the first generalization of Lamport's proposition to the case of multiple readers. A further generalization appears in [18] for the case of a variable having several fields which can be written independently.

LEMMA **7.2** *A complete proper history h is linearizable iff there exist a function mapping each operation in h to a rational number, called its tag, such that the following 3 conditions are satisfied:*

Uniqueness *different Write operations have different tags.*

Integrity *for each Read operation there exists a Write operation with the same tag and value, that it doesn't precede.*

Precedence *if one operation precedes another, then the tag of the latter is at least that of the former.*

PROOF. $\Rightarrow$ Let a complete proper history h be linearizable. Then there is some total order $<$ extending $\prec_h$. Assign to each operation a tag which is the number of Write operations preceding it in $<$. This clearly satisfies Uniqueness. For any Read operation R, the Write operation W that precedes it last in $<$ has the same tag. Also, because $<$ obeys the semantics of a variable, W and R have the same value. From the facts that $<$ extends $\prec_h$, $W < R$, and $<$ is acyclic, we conclude that $\neg R \prec_h W$. So Integrity is satisfied as well. Finally, for operations $A \prec_h B$, we necessarily have $A < B$ and thus the tag of B is at least that of A.

$\Leftarrow$ Suppose we are given a complete proper history h and a function *tag* satisfying the three conditions. Using Uniqueness, totally order the Write operations according to their tags. Next, we insert all Read operations in this total order: for each Write operation in turn, insert immediately after it those Read operations having the same tag, in any order extending $\prec_h$. By Integrity, the result is a total order $<$ on all operations, that obeys the semantics of a variable. It remains to show that $<$ extends $\prec_h$. Suppose $A \prec_h B$ are two operations. By Precedence, A's tag is at most that of B. If A's tag is less than B's, or A and B are Read operations with the same tag, then $A < B$ follows from the construction of $<$. In the remaining case A and B have equal tags and at least one of them is a Write operation. By Uniqueness, one is a Read operation, and the other is the unique Write operation with the same tag. Finally, we use Integrity to conclude that A is the Write, and B the Read operation. Thus, $A < B$ follows again from the construction of $<$. □

7.2 The Basic Unbounded Construction

Figure 7.1 shows Construction 0, which is the unbounded solution of [20]. We present it here as an aid in understanding Construction 1, and give only a sketchy proof.

The Write and Read procedures are given after the declaration of the type of the shared variables $R_{i,j}$. The initial state of the construction has all $R_{i,j}$ containing $(0, 0)$.

The tag function called for in lemma 7.2 is built right in to this construction. Each operation starts by collecting value-tag pairs from all users. By executing

```
type I : 0..n − 1
       shared : record
                    value : ABStype
                    tag   : integer
                end
```

```
procedure Write(i, v)
var j : I
    t : integer
    from : array[0..n − 1] of shared
begin
    for j ∈ I do from[j] := R_{j,i}
    select t such that (∀j : t > from[j].tag) ∧ t ≡ i   (mod n)
    from[i] := (v, t)
    for j ∈ I do R_{i,j} := from[i]
end
```

```
procedure Read(i)
var j, max : I
    from : array[0..n − 1] of shared
begin
    for j ∈ I do from[j] := R_{j,i}
    select max such that ∀j : from[max].tag ≥ from[j].tag
    from[i] := from[max]
    for j ∈ I do R_{i,j} := from[i]
    return from[i].value
end
```

FIGURE 7.1. Construction 0

line 3 of either procedure, the operation picks a value and tag for itself. It finishes after distributing this pair to all users. It is not hard to see that the three conditions of lemma 7.2 are satisfied for any complete proper history. Integrity and Precedence are straightforward to check. Uniqueness follows since tags of Write operations of different users are not congruent modulo n, while tags of Write operations of a single user strictly increase (based on the observation that each $R_{i,i}.tag$ is nondecreasing).

7.3 Solution Method

The only problem with Construction 0 is that the number of tags is infinite. With a finite number of tags comes the necessity to re-use tags and hence to distinguish old tags from new ones.

In Construction 1, we introduce a shooting mechanism to provide aging information in addition to the tags. At the start of an operation, a user sets up a 'target' that gets 'shot at' by Write operations. A tag can be considered old once its associated target has received sufficiently many shots. The shooting mechanism also serves another purpose, which is that of approximating a snap-shot, i.e. an instantaneous picture of a set of shared variables. In Construction 0, an operation collects information on values and tags of all users by reading their variables one after another, in arbitrary order (the first line in either procedure). Since these read events are interleaved with events of other users, in particular write events, the picture it gets this way can be very distorted. In Construction 1, with the additional information to collect, there is a need to limit the amount of distortion. If, after the information-collecting period, the target that was set up has received sufficiently many shots, then the operation will *abort*, i.e. terminate without executing the remainder of the procedure. Aborting operations do not change or make use of any tags and thus have very limited interaction with non-aborting operations. The latter in turn will have got a good, if not instantaneous, picture of the shared state. This in principle allows them to distinguish old tags by inspection of the associated targets. This distinction is however not yet made in Construction 1. With all the added unbounded counters, Construction 1 merely paves the way to our final, bounded, solution. In Section 7.3.1, we discuss Construction 1 and in particular the shooting mechanism, in some more detail. Section 7.3.2 introduces some notational conventions. The correctness proof is given in Section 7.3.3. Finally, Section 7.4 shows how Construction 1 can be changed into an equivalent one using only bounded counters.

7.3.1 Construction 1

Figure 7.2 shows the data-structure and procedures of Construction 1. The Write procedure turns out to be an extension of the Read procedure which is why the two are more conveniently shown together. The line indicated '(Read only)' is unique to the Read procedure, making the remaining lines effectively unique to the Write procedure. The initial state of Construction 1 has 0 in all fields of all shared and static variables.

Let's look at the data structures used in the construction. The *value* and

```
type I : 0..n − 1
     shared : record
                  value,prev : ABStype
                  tag  : integer
                  ss : 0..1
                  shoot,heal : array[0..1][0..n − 1] of integer
              end
```

```
procedure Read(i) / Write(i, v)
var j : I
    t : integer
    s : 0..1
    from,tmp : array[0..n − 1] of shared
    static me : shared
begin
    s := 1 − me.ss
s:  for j ∈ I do me.heal[s][j] := R_{j,i}.shoot[s][i]
h:  for j ∈ I do R_{i,j} := me
r:  for j ∈ I do from[j] := R_{j,i}
t:  for j ∈ I do tmp[j] := R_{j,i}
    if ∃j ∈ I : tmp[j].shoot[s][i] − me.heal[s][j] ≥ 3
    then return tmp[j].prev
    select max such that ∀j : from[max].tag ≥ from[j].tag
    me.prev, me.value, me.tag, me.ss :=
          me.value, from[max].value, from[max].tag, s
p:  for j ∈ I do R_{i,j} := me
    (Read only) return me.value
    for j ∈ I, s ∈ {0..1} do
        if me.shoot[s][j] − from[j].heal[s][i] < 6
        then me.shoot[s][j]+ := 1
    select t such that t − me.tag ∈ {1, . . . , n} ∧ t ≡ i   (mod n)
    me.value, me.tag := v, t
w:  for j ∈ I do R_{i,j} := me
end
```

FIGURE 7.2. Construction 1

tag fields have exactly the same function as in Construction 0. The *prev* field is used to remember values of former operations, which are used by aborting Read operations. Two sets of heal counters, $heal[0][0..n-1]$ and $heal[1][0..n-1]$, are used to hold targets. The *ss* (shoot-selector) field selects which of the two sets holds the target associated with the current value-tag pair. A second set is needed since new operations must set up a target before they can compute a new tag. Together with the heal counters, the shot counters, $shoot[0..1][0..n-1]$, implement the shooting mechanism. User j shoots at a target $heal[s][0..n-1]$ of user k by making his counter $shoot[s][k]$ larger than the counter $heal[s][j]$ of user k, up to a maximum of 6.

Now let's consider the procedures. Note that the lines are no longer numbered. Instead, the lines involving shared variable access are identified by one of the characters s,h,r,t,p and w, which are mnemonic shorthands for setup, heal, read, test, propagate and write, respectively.

At the start of an operation, call it a, user i sets up a new target in the available heal counter set $(1 - me.ss)$ by catching up with each user's shot counter. It then writes out the target in line h so that the other users can start shooting it. After collecting every one's data in line r, it proceeds to test how many times it's target has been shot. More precisely, if any user has increased its shot counter at least 3 times since it was previously read in line s, then a will abort. The j in the **return** statement is meant to be any j satisfying the condition of the test, but could be chosen as the minimal index for the sake of definiteness. It can be shown that a completely 'contains' an operation b of user j with the value $tmp[j].prev$. Thus, a can be imagined to have occurred right before or after b in a linearization, depending on whether it's a Write or Read operation. If no user shot the target 3 times, then user i sets *max* to an index of the largest visible tag. It then saves the old value in *prev*, changes its value and tag to that of *max*, and associates its target with the new value-tag pair. In line p, record *me* is written out. The purpose of the Write operations propagating the value-tag pair of *max* is to ensure that the maximum tag visible to one user is at most n larger than the maximum tag visible to any other user. The Read procedure ends after line p by returning the value copied from *max*.

The Write procedure continues by shooting all visible targets, that is, increasing all its shot counters that are not already 6 ahead of their corresponding heal counter. User i next chooses a tag unique to it which is larger than all visible ones. This is paired with the argument v the Write procedure, and all is written out in line w.

7.3.2 Notational Conventions

The following notions are used in the proof. Where necessary, assume a fixed, but arbitrary history. The m'th non-aborting operation of user i is denoted N_i^m. If $a = N_i^m$ then a^{+r} denotes N_i^{m+r}, i.e. the r-th next non-aborting operation by user i following a, assuming it exists. If $a = N_i^m$ then a^{-r} denotes N_i^{m-r}, i.e. the r-th previous non-aborting operation by user i preceding a. Use of this notation depends on the assumption that $r < m$. Since all shot counters are initialized to 0, and increase at most by one per non-aborting Write operation, the value assigned by an operation a to one of its shot counters provides a lower bound on m. We'll use the notation only where it is justified on these grounds.

The events of an operation a involving shared variable access constitute up to 6 events-sets, or *phases*:

$$a.\mathrm{s} \prec a.\mathrm{h} \prec a.\mathrm{r} \prec a.\mathrm{t} \prec a.\mathrm{p} \prec a.\mathrm{w},$$

in accordance with the labelled lines of the Read and Write procedure. Aborting operations consist of only the first 3 phases, while a non-aborting Read operation has the first 5. The n events in a phase $a.\mathrm{c}$ (c one of s,h,r,t,p or w) are denoted $a.\mathrm{c}_j$ with $j \in I$, and are called c-events.

For a shared variable read event e, define p-*Last*(e) to be the operation containing the last p-event preceding e that accesses the same shared variable. If such an event does not exist, then p-*Last*(e) is defined to be the non-operation $\perp$.

For a an operation and *exp* an expression consisting of (symbolic or explicit) constants and local variables. define *exp*@a as the final value of that expression in the procedure invocation corresponding to a. Array indices i, j, k, s, t refer to symbolic constants defined in the context, not to the local variables. Define

$$\mathit{value}@\perp = \mathit{prev}@\perp = \mathit{tag}@\perp = \mathit{ss}@\perp = \mathit{shoot}[\,][\,]@\perp = \mathit{heal}[\,][\,]@\perp = 0,$$

in accordance with the initialization of the construction. Define *exp*@a.c (c one of s,h,r,t,p or w) as the value of the expression *exp* after completion of line c of the procedure invocation corresponding to a. By convention, the prefix $me.$ is omitted when *exp* is a field of me.

7.3.3 Correctness of Construction 1

Construction 1 clearly satisfies Wait-Freeness as all loops range over $I = \{0, \ldots, n-1\}$. To prove correctness, we must therefore show that each complete proper history is linearizable. The proof is based on Lemma 7.2. First we need some preparatory claims.

Claim **7.3** All shared tag, heal and shot counters are nondecreasing in the course of a history.

Proof. A shared variable $R_{i,j}$ is changed only when me is written to it, in an h-, p-, or w-event of user i, so a nondecreasing counter in the static local variable me of user i implies a corresponding nondecreasing counter in $R_{i,j}$ for all $j \in I$. The $me.\mathit{shoot}$ counters are clearly nondecreasing and therefore so are the shared shot counters. Each heal counter $me.\mathit{heal}[s][j]$ of user i is only changed by assignment from $R_{j,i}.\mathit{shoot}[s][i]$ and is thus also nondecreasing. It remains to show that $me.\mathit{tag}$ is non-decreasing. Consider the new tag $\mathit{from}[\mathit{max}].\mathit{tag}$ that is assigned to $me.\mathit{tag}$ prior to line p. By the selection of max, this is at least $\mathit{from}[i].\mathit{tag}$ which by lines h,r is just a copy of $me.\mathit{tag}$. Thus, $me.\mathit{tag}$ doesn't decrease in this assignment. In the other assignment, prior to line w, $me.\mathit{tag}$ only increases. □

Corollary **7.4** If event e assigns v_e to a shared tag, heal, or shot counter, and event f assigns v_f from the same shared counter, then $e \prec f \Rightarrow v_f \geq v_e$ and $v_f < v_e \Rightarrow f \prec e$.

COROLLARY **7.5** Let $a \prec b$ be non-aborting operations by users i and j respectively. If b is a Read operation, then

$$tag@b \geq tag@a,$$

and if b is a Write operation, then

$$tag@b \geq tag@a + 1.$$

PROOF. By the selection of max, $a \prec b$, and corollary 7.4, $from[max].tag@b.\mathrm{p} \geq from[i].tag@b.\mathrm{r} \geq tag@a$. For b a non-aborting Read operation, $tag@b = from[max].tag@b.\mathrm{p}$. For b a non-aborting Write operation, $tag@b \geq tag@b.\mathrm{p}{+}1 = from[max].tag@b.\mathrm{p}{+}1$. □

CLAIM **7.6** The difference $me.shoot[s][j]{-}from[j].heal[s][i]$ as well as $tmp[j].shoot[s][i]{-}me.heal[s][j]$ between corresponding shot and heal counter as computed in the Read and Write procedure is between 0 and 6 (inclusive).

PROOF. Using the 0 initialization of the construction, claim 7.3, and the way shot counters are increased, differences less than 0 or greater than 6 are easily shown to lead to a contradiction. □

CLAIM **7.7** Let a be an operation by user i, $b = \text{p-}Last(a.\mathrm{t}_j)$, and $c = \text{p-}Last(a.\mathrm{r}_j)$. Let $s = from[j].ss@a.\mathrm{r}$. Then

1. $tmp[j].prev@a.\mathrm{t} = prev@b$, $s = ss@c$
2. if $c = \perp$ then $from[j].tag@a.\mathrm{r} = 0$
   else $tag@c.\mathrm{p} \leq from[j].tag@a.\mathrm{r} \leq tag@c$
3. for all $k \in I$, $from[j].heal[s][k]@a.\mathrm{r} = heal[s][k]@c$
4. for all $k \in I, z \in \{0,1\}$, if $c = \perp$ then $from[j].shoot[z][k]@a.\mathrm{r} = 0$
   else $shoot[z][k]@c.\mathrm{p} \leq from[j].shoot[z][k]@a.\mathrm{r} \leq shoot[z][k]@c$

PROOF.

1. Only the p-events of user j change $R_{j,i}.prev$ and $R_{j,i}.ss$.
2. In case $c = \perp$, no tag has overwritten the initial 0. In case $c \neq \perp$, the first inequality follows directly from the definition of c and corollary 7.4. For the second, note that after c, the value of $R_{j,i}.tag$ remains $tag@c$ until $c^{+1}.\mathrm{p}_i$ (if any), which by definition of c doesn't precede $a.\mathrm{r}_j$.
3. After $c.\mathrm{p}_i$ (or from the start of history if $c = \perp$), the value of ss in user j's me remains s at least until $c^{+1}.\mathrm{p}$ (if any), hence only its $heal[1{-}s]$ counters are changed at least until a new operation starts after c^{+1}.
4. Analogous to item 2.

□

CLAIM **7.8** Let $a, b = a^{+1}$ be two non-aborting operations by user i.

1. $prev@b = value@a$
2. $\forall j \in I, s \in \{0,1\} : shoot[s][j]@b \leq shoot[s][j]@a + 1$
3. If a and b are Write operations then $tag@b \geq tag@a + n$.

PROOF.

1. Since b doesn't abort, and aborting operations don't change $me.value$, $prev@b = value@b.\mathrm{s} = value@a$.
2. Similarly.
3. Since $tag@b \equiv tag@a \equiv i \pmod{n}$, their difference is a multiple of n, and by corollary 7.5, it is positive.

□

CLAIM **7.9** Let a be an aborting operation by user i, and let j be the index for which the abortion condition holds. Then there exists a non-aborting Write operation b by user j, such that

$$a.\mathrm{s}_j \prec b \prec a.\mathrm{t}_j \wedge tmp[j].prev@a = value@b.$$

PROOF. Let $c = \text{p-}Last(a.\mathrm{t}_j)$, $b = c^{-1}$, and $d = c^{-2}$ (recall Section 7.3.2 on notation). Then $d \prec b \prec c.\mathrm{p} \prec a.\mathrm{t}_j$ and by claims 7.7,7.8, $tmp[j].prev@a = prev@c = value@b$. Also, with $s = 1 - ss@a$, by abortion of a, claims 7.7 and 7.8,

$$heal[s][j]@a + 3 \leq tmp[j].shoot[s][i]@a \leq shoot[s][i]@c \leq shoot[s][i]@d + 2.$$

This shows that $heal[s][j]@a < shoot[s][i]@d$, hence not $d \prec a.\mathrm{s}_j$. Combined with $d \prec b$ this yields $a.\mathrm{s}_j \prec b$. □

The following claim will be used in later sections.

CLAIM **7.10** If a is an operation a by user i, and w_1, w_2, w_3 are non-aborting Write operations by user k, such that

$$a.\mathrm{h}_k \prec w_1.\mathrm{r}_i \prec w_2 \prec w_3.\mathrm{w}_i \prec a.\mathrm{t}_k,$$

then a aborts.

PROOF. Let $s = ss@a$. Claim 7.6 and the assumption of the claim give

$$shoot[s][i]@w_1.\mathrm{r} \geq from[i].heal[s][k]@w_1.\mathrm{r} \geq heal[s][k]@a.$$

According to the shooting mechanism, induction on m shows that $shoot[s][i]@w_m \geq heal[s][k]@a + \min(m, 6)$. Since $w_3.\mathrm{w}_i \prec a.\mathrm{t}_k$, corollary 7.4 implies

$$tmp[k].shoot[s][i]@a.\mathrm{t} \geq shoot[s][i]@w_3 \geq heal[s][k]@a + 3,$$

hence a aborts. □

LEMMA **7.11** *Any complete proper history h of Construction 1 is linearizable.*

PROOF. The proof is based on the tag lemma. We show that there is a function $\tau()$, mapping each operation in h to a rational number, that satisfies Uniqueness, Integrity, and Precedence. Let a be an operation by user i. If a doesn't abort, then simply set $\tau(a) = tag@a$. Otherwise, if a aborts, let b be the operation given by claim 7.9. Now set $\tau(a) = tag@b$ if a is a Read operation, or set $\tau(a) = tag@b - \epsilon_a$, if a is a Write operation, where $0 < \epsilon_a < 1$ is a fraction unique to a.

Uniqueness Let a and b be different Writes operations by users i and j respectively. If either aborts, then its tag has a unique fractional part and is therefore different from the other operation's tag. Suppose neither aborts. Then $\tau(a) = tag@a = i \pmod n$, and $\tau(b) = tag@b = j \pmod n$. If $i \neq j$ then Uniqueness follows immediately. In case $i = j$, one Write operation must precede the other, and Uniqueness follows from corollary 7.5.

Integrity For aborting Read operations, Integrity follows from claim 7.9. The value-tag pair that a non-aborting Read operation a copies must originate from a non-aborting Write operation b. Clearly, $\neg(a \prec b)$. Combined with the definition of τ, this proves Integrity.

Precedence Consider two operations $a \prec b$. We must show that $\tau(a) \leq \tau(b)$. If a aborts, then by claim 7.9 and definition of τ, there exists a j and a non-aborting operation a' such that $a' \prec a.\mathrm{t}_j \prec b$ and $\tau(a) \leq \tau(a')$, in which case it would suffice to show Precedence for $a' \prec b$. So without loss of generality we can assume that a doesn't abort. If b doesn't abort, then Precedence follows from corollary 7.5. Suppose b aborts. By claim 7.9, there exist a j and a non-aborting operation b' such that $a \prec b.\mathrm{s}_j \prec b'$. Then we use the definition of τ and corollary 7.5 to show Precedence: If b is a Write operation then $\tau(b) = tag@b' - \epsilon_b \geq tag@a + 1 - \epsilon_b > \tau(a)$. If b is a Read operation then $\tau(b) = tag@b' \geq tag@a = \tau(a)$.

□

7.4 Bounding the counters

Having proven Construction 1 correct, we will make a correctness preserving transformation that renders all variables bounded. The transformation is based on 3 key lemmas. The first formalizes the idea that a tag, whose target is seen to have been shot sufficiently many times, can be considered old, and ignored in the selection of a maximum tag. The second shows that the remaining, 'live', tags are in a bounded range, which is the basis for bounding the tags. Finally, the third shows that the perceived number of times a target is shot is bounded both from below and above, which is the basis for bounding the heal and shot counters.

7.4.1 Old tags

LEMMA **7.12** *Let a be a non-aborting operation by user i. Let $j, k \in I$, and $s = from[j].ss@a.\mathrm{r}$. If*

$$from[k].shoot[s][j]@a.\mathrm{r} - from[j].heal[s][k]@a.\mathrm{r} \geq 6$$

(a sees 6 shots by k on j's target), then

$$from[k].tag@a.\mathrm{r} > from[j].tag@a.\mathrm{r}$$

PROOF. Let $b = \text{p-}Last(a.\mathrm{r}_j)$ and $c = \text{p-}Last(a.\mathrm{r}_k)$. Using in succession claim 7.7, the assumption of the claim, and claim 7.7 again:

$$shoot[s][j]@c \geq from[k].shoot[s][j]@a.\mathrm{r} \geq$$

$$from[j].heal[s][k]@a.\mathrm{r} + 6 = heal[s][k]@b + 6$$

This shows the existence of

$$w_3 \prec w_4 \prec w_5 \prec w_6.\mathrm{w}_i \prec a.\mathrm{r}_k,$$

where $w_m, 3 \leq m \leq 6$ is the *first* non-aborting Write operation by user k such that

$$shoot[s][j]@w_m = heal[s][k]@b + m.$$

By corollary 7.4, claim 7.8, and the tag choice in Construction 1,

$$from[k].tag@a.\mathrm{r} \geq tag@w_6 \geq$$

$$tag@w_4 + 2n \geq from[max].tag@w_4.\mathrm{p} + 2n + 1.$$

If $b = \perp$, then $from[j].tag@a.\mathrm{r} = 0$ and the lemma follows immediately. Otherwise, by claim 7.7 and the tag choice in Construction 1,

$$from[j].tag@a.\mathrm{r} \leq tag@b \leq from[max].tag@b.\mathrm{p} + n.$$

Thus, to prove the lemma it suffices to show that

$$from[max].tag@w_4.\mathrm{p} + 2n + 1 > from[max].tag@b.\mathrm{p} + n. \qquad (7.1)$$

Since by its definition, b doesn't abort, corollary 7.4 implies $b.\mathrm{r} \prec b.\mathrm{t}_k \prec w_3.\mathrm{w}_j \prec w_4$. Let Write operation w be the originator of the tag $from[max].tag@b.\mathrm{p}$. We have $\neg(b.\mathrm{r} \prec w.\mathrm{w})$. Since $b.\mathrm{r} \prec w_4$ and $w.\mathrm{p} \prec w.\mathrm{w}$, it must therefore be that $w.\mathrm{p} \prec w_4$. This shows

$$from[max].tag@w_4.\mathrm{p} \geq tag@w.\mathrm{p} \geq tag@w - n = from[max].tag@b - n,$$

which immediately implies 7.1. □

DEFINITION **7.13** In the context of a Read or Write procedure, define

$$\begin{aligned} alive(j) \quad \equiv \quad & (\forall k \in I : from[k].shoot[s][j] - from[j].heal[s][k] < 6), \\ & \text{where } s = from[j].ss \end{aligned}$$

COROLLARY **7.14** For any non-aborting operation a, $alive(max)@a.\mathrm{p}$.

This shows that the choice of max can be restricted to those $j \in I$ for which $alive(j)$ holds.

7.4.2 Range of alive tags

The parameter m in the next lemma serves to prepare for a later simplification of the construction, for the case of only one writer.

LEMMA **7.15** *Let a be a non-aborting operation by user i. Let $j \in I$ and $s = \mathit{from}[j].\mathit{ss}@a.\mathrm{r}$. Let $1 \le m \le n$ be the number of users with Write operations. If $\mathit{alive}(j)@a.\mathrm{r}$ then*

$$\mathit{from}[\mathit{max}].\mathit{tag}@a.\mathrm{p} - \mathit{from}[j].\mathit{tag}@a.\mathrm{p} \le 10mn$$

PROOF. Assume, to the contrary, that $\mathit{from}[\mathit{max}].\mathit{tag}@a.\mathrm{p} > \mathit{from}[j].\mathit{tag}@a.\mathrm{p} + 10mn$. Let W be the set of all non-aborting Write operations w such that $\mathit{tag}@w > \mathit{from}[j].\mathit{tag}@a.\mathrm{p} \wedge \neg(a.\mathrm{r} \prec w)$. Since a reads a tag which is more than $10mn$ greater than j's, and new tags are chosen in increments of at most n, we must have $|W| > 10m$. Therefore, some user, say k, has at least 11 operations, say $w_0 \prec w_1 \prec \cdots \prec w_{10}$, in W.

We claim that

$$\mathit{from}[j].\mathit{heal}[s][k]@w_1 \ge \mathit{heal}[s][k]@b. \tag{7.2}$$

Otherwise corollary 7.4 gives $b \neq \perp$ and $w_0 \prec w_1.\mathrm{r}_j \prec b.\mathrm{h}_k \prec b.\mathrm{r}_k$, from which corollary 7.4 and claim 7.7 imply $\mathit{tag}@w_0 \le \mathit{from}[k].\mathit{tag}@b.\mathrm{r} \le \mathit{tag}@b.\mathrm{p} \le \mathit{from}[j].\mathit{tag}@a.\mathrm{p}$, contradictory to the definition of $w_0 \in W$. As the proof of claim 7.10 shows, 7.2 implies $\mathit{shoot}[s][j]@w_6 \ge \mathit{heal}[s][k]@b + 6$. On the other hand, the assumption $\mathit{alive}(j)@a.\mathrm{r}$ and claim 7.7 give $\mathit{from}[k].\mathit{shoot}[s][j]@a.\mathrm{r} < \mathit{from}[j].\mathit{heal}[s][k]@a.\mathrm{r}+6 = \mathit{heal}[s][k]@b+6$. Together, using corollary 7.4, these inequalities show that

$$a.\mathrm{h}_k \prec a.\mathrm{r}_k \prec w_6.\mathrm{w}_i \prec w_7.\mathrm{r}_i.$$

Since a doesn't abort, claim 7.10 implies

$$a.\mathrm{r} \prec a.\mathrm{t}_k \prec w_9.\mathrm{w}_i \prec w_{10},$$

in contradiction to the definition of $w_{10} \in W$. □

The lemma shows that all alive tags are from $10mn$ to 0 less than the maximum. The following is an easy consequence.

COROLLARY **7.16** Let a be a non-aborting operation by user i. Let $1 \le m \le n$ be the number of users with Write operations. Let $j, k \in I$ such that $\mathit{alive}(j)@a.\mathrm{r} \wedge \mathit{alive}(k)@a.\mathrm{r}$. Then

$$-10mn \le \mathit{from}[k].\mathit{tag} - \mathit{from}[j].\mathit{tag} \le 10mn.$$

7.4.3 Bounds on perceived shots

LEMMA **7.17** *Let a be a non-aborting operation by user i. Let $j, k \in I$, and $s = \mathit{from}[j].\mathit{ss}@a.\mathrm{r}$. Then*

$$-4 \le \mathit{from}[k].\mathit{shoot}[s][j]@a.\mathrm{r} - \mathit{from}[j].\mathit{heal}[s][k]@a.\mathrm{r} \le 9$$

PROOF. Assume, to the contrary, that the difference is outside $\{-4,\ldots,9\}$. We will reach the required contradiction by showing that the conditions of claim 7.10 hold, implying that a aborts.

Let $b = \text{p-}\mathit{Last}(a.\mathrm{r}_j)$ and $c = \text{p-}\mathit{Last}(a.\mathrm{r}_k)$. In the first case, assume the difference is under -4. Then by claim 7.7, $s = ss@b$ and

$$\mathit{heal}[s][k]@b = \mathit{from}[j].\mathit{heal}[s][k]@a.\mathrm{r} \geq \mathit{from}[k].\mathit{shoot}[s][j]@a.\mathrm{r} + 5.$$

So $b \neq \bot$ and $b.\mathrm{s}_k$ read a shot counter from user k that's at least 5 greater than what a read in $a.\mathrm{r}_k$. This shows the existence of

$$a.\mathrm{r}_k \prec w_1.\mathrm{w}_i \prec w_2 \prec w_3 \prec w_4 \prec w_5.\mathrm{w}_j \prec b.\mathrm{s}_k,$$

where $w_m, 1 \leq m \leq 5$ is the *first* non-aborting write action by user k such that

$$\mathit{shoot}[s][j]@w_m = \mathit{from}[k].\mathit{shoot}[s][j]@a.\mathrm{r} + m.$$

Consequently, the conditions of claim 7.10 hold:

$$a.\mathrm{h}_k \prec a.\mathrm{r}_k \prec w_2 \prec w_4 \prec b.\mathrm{s}_k \prec b.\mathrm{p}_i \prec a.\mathrm{r}_j \prec a.\mathrm{t}_k.$$

This proves the first inequality of the lemma.

In the other case, assume the difference is over 9. Then by claim 7.7

$$\mathit{shoot}[s][j]@c \geq \mathit{from}[k].\mathit{shoot}[s][j]@a.\mathrm{r} \geq$$

$$\mathit{from}[j].\mathit{heal}[s][k]@a.\mathrm{r} + 10 = \mathit{heal}[s][k]@b + 10.$$

This shows the existence of

$$w_7 \prec w_8 \prec w_9 \prec w_{10}.\mathrm{w}_i \prec a.\mathrm{r}_k,$$

where $w_m, 7 \leq m \leq 10$ is the *first* non-aborting Write operation by user k such that

$$\mathit{shoot}[s][j]@w_m = \mathit{heal}[s][k]@b + m.$$

Claim 7.6 shows that

$$\mathit{from}[j].\mathit{heal}[s][k]@w_7.\mathrm{r} \geq \mathit{shoot}[s][j]@w_7 - 6 = \mathit{heal}[s][k]@b + 1.$$

As the proof of item 3 of claim 7.7 shows, this can only be if $d \prec w_7.\mathrm{r}_j$, with d being N_j^1 in case $b = \bot$ and b^{+1} otherwise.

Thus, using b's definition,

$$a.\mathrm{h}_k \prec a.\mathrm{r}_j \prec d.\mathrm{p}_i \prec w_7.\mathrm{r}_j \prec w_8.$$

Again the conditions of claim 7.10 hold. This proves the second inequality of the lemma. □

7.4.4 Equivalence with bounded counters

For notational convenience, we introduce three binary operators $\ominus$, $\oplus$ and $\odot$:

DEFINITION **7.18** For any integers a, b, $a \ominus b$, $a \oplus b$ and $a \odot b$ are uniquely defined by the equations

$$\begin{array}{rlll} -4 \leq & a \ominus b & < 10 & \wedge \quad a \ominus b \equiv a - b \pmod{14} \\ -4 \leq & a \oplus b & < 10 & \wedge \quad a \oplus b \equiv a + b \pmod{14} \\ -10n^2 \leq & a \odot b & < 10n^2 + n & \wedge \quad a \odot b \equiv a - b \pmod{20n^2 + n} \end{array}$$

By Corollary 7.14 and a simple reordering of terms, the assignment

select max such that $\forall j : from[max].tag \geq from[j].tag$

of Construction 1 is equivalent to:

select $max \in A$ such that $\forall j \in A : from[max].tag - from[j].tag \geq 0$
where $A = \{j \in I : alive(j)\}$

which, by corollary 7.16 and lemma 7.17, is in turn equivalent to

select $max \in A$ such that $\forall j \in A : from[max].tag \odot from[j].tag \geq 0$
where $A = \{j \in I : \forall k \in I : from[k].shoot[z][j] \ominus from[j].heal[z][k] < 6$
where $z = from[j].ss\}$

Furthermore, the subtractions referred to in claim 7.6 are also equivalent to $\ominus$.

Let Construction 2 be the result of replacing these 3 program fragments (selection of max and shot-heal differences) by their equivalent parts. Obviously, Construction 2 is also correct.

We next consider our bounded solution, Construction 3, shown in figure 7.3. It is identical to Construction 2 except for the type of tags and shot/heal counters, and the way they are increased. Note that in the 2nd to last line, the selection of t is possible (and unique) because $20n^2 + n$ is a multiple of n. The initial state of Construction 3 has 0 in all fields of all shared and static variables, like Constructions 1 and 2.

To prove Construction 3 correct, it suffices to show that any of its histories, say $h_3 = S_0, S_1, S_2, \ldots$, is *equivalent* to a history $h_2 = T_0, T_1, T_2, \ldots$ of Construction 2, in the sense that corresponding states S and T are the same, up to *congruence* of tags and shot/heal counters. That is, when tag_3 is the value of some local or shared variable of type *tagtype* in state S_r, and tag_2 is the value of that same variable in state T_r, then we want $tag_3 \equiv tag_2 \pmod{20n^2+n}$. Similarly, when $shot_3$ is the value of some local or shared variable of type *shottype* in state S_r, and $shot_2$ is the value of that same variable in state T_r, then we want $shot_3 \equiv shot_2 \pmod{14}$. We prove the existence of h_2 by induction on the length of h_3.

If h_3 consists of only the initial state S_0 then it is clearly equivalent to the h_2 consisting only of the initial state T_0 of Construction 2. Now suppose $h_3 = S_0, S_1, \ldots, S_r, S_{r+1}$ is a history of Construction 3, and the one event shorter history $S_0, S_1, \ldots, S_r$ is equivalent to a history $T_0, T_1, \ldots, T_r$ of Construction 2. In particular, S_r is the same as T_r, up to congruence. Let T_{r+1} be the state reached from T_r by execution of the same procedure statement by the same user as in the event $S_r \rightarrow S_{r+1}$. One can check in a straightforward manner,

by considering all procedure statements, that T_{r+1} must then be the same as T_{r+1}, up to congruence. The test for abortion comes out the same because of congruence. Similarly, the same choices of *max* are available in T_r as in S_r. The increment of $me.shoot[s][j]$ in the 3rd to last line preserves congruence, as does the selection of t in the next line. Other statements are even more straightforward.

This completes the induction argument and shows

THEOREM **7.19** *Any complete proper history of Construction 3 is linearizable.*

7.5 Complexity

First let's consider the time complexity of Construction 3. Since all shared variable accesses occur in phases, each of which consists of n reads or writes in parallel, the time complexity of both the Read and Write operation is bounded by the number of phases, which is clearly $O(1)$.

Next consider the space complexity of Construction 3, which is the size in bits of the type *shared*. This can be split into two parts: the data size and the control size.

The data size is $2 \times sizeof(\text{ABS})$. This factor 2 overhead can be traced back to the use of single reader shared variables in our construction. In fact, the version of Construction 3 using multi-reader variables can be easily modified to do away with the *prev* data field (as sketched in the next section).

The control size concerns all the other fields in the shared variables. Note first that from the values read from $R_{j,i}$, user i never uses any of the counters $heal[1-ss][k]$, where $k \neq i$. Thus, in addition to the single counter $heal[1-ss][i]$, only a single heal counter set needs to be stored in $R_{j,i}$, of which the missing first index is understood to be $R_{j,i}.ss$. This leads to a control size of

$$\lceil \log 20n^2 + n \rceil + 1 + (3n+1)\lceil \log 14 \rceil \leq 12n + o(n).$$

7.6 Subproblems

The final construction presents a solution to the problem of implementing a multi-user variable from single-reader variables. Many other papers have considered the intermediate level of a single-writer multi-reader variable, which splits the problem into two subproblems. We show that *projections* of our construction yield solutions to those two subproblems with competitive complexity measures.

The first projection is obtained by collapsing the row of shared variables $R_{i,0}, \ldots, R_{i,n-1}$ into a single multi-reader variable R_i. Each loop

for $j \in I$ **do** $R_{i,j} := me$

is replaced by the single write

$R_i := me$

```
type I : 0..n − 1
       tagtype : −10n²..10n² + n − 1
       shottype : −4..9
       shared : record
                    value,prev : ABStype
                    tag  : tagtype
                    ss : 0..1
                    shoot,heal : array[0..1][0..n − 1] of shottype
                end
```

```
procedure Read(i) / Write(i, v)
var j : I
    t : tagtype
    s : 0..1
    from,tmp : array[0..n − 1] of shared
    static me : shared
begin
    s := 1 − me.ss
    for j ∈ I do me.heal[s][j] := R_{j,i}.shoot[s][i]
    for j ∈ I do R_{i,j} := me
    for j ∈ I do from[j] := R_{j,i}
    for j ∈ I do tmp[j] := R_{j,i}
    if ∃j ∈ I : tmp[j].shoot[s][i] ⊖ me.heal[s][j] ≥ 3
    then return tmp[j].prev
    select max ∈ A such that ∀j ∈ A : from[max].tag ⊙ from[j].tag ≥ 0
    where A = {j ∈ I : ∀k ∈ I : from[k].shoot[z][j] ⊖ from[j].heal[z][k] < 6
                        where z = from[j].ss}
    me.prev, me.value, me.tag, me.ss :=
      me.value, from[max].value, from[max].tag, s
    for j ∈ I do R_{i,j} := me
    (Read only) return me.value
    for j ∈ I, s ∈ {0..1} do
        if me.shoot[s][j] ⊖ from[j].heal[s][i] < 6
        then me.shoot[s][j]⊕ := 1
    select t ∈ tagtype such that t ⊙ me.tag ∈ {1, . . . , n} ∧ t ≡ i  (mod n)
    me.value, me.tag := v, t
    for j ∈ I do R_{i,j} := me
end
```

FIGURE 7.3. Construction 3

while each read from $R_{j,i}$ is replaced by a read from R_j. The result is a solution to the problem of implementing a multi-user variable from single-writer multi-reader variables, since each of its histories corresponds to a history of Construction 3 in which all writes of a parallel loop happen to be consecutive events. The time complexity is still constant, while the space complexity is $sizeof(shared) = 2 \times sizeof(\text{ABS}) + 16n + o(n)$. The *prev* field can be made redundant by letting an aborting operation return $tmp[j].value$ instead of $tmp[j].prev$. This can be proven correct by reformulating claim 7.9 and the Precedence part of lemma 7.11. (The idea is that the value returned by a in claim 7.9 is then the value of c or the value propagated by c, and a completely contains the part of c up to c.p).

The second projection is more involved, but yields a large space savings. Assume that only user 0 executes Write operations. Then all shoot counters of the remaining users, as well as all heal counters $heal[0..1][1..n-1]$, remain 0, and can therefore be omitted. The *prev* field of the Read-only users will never be used and can also be omitted. Furthermore, 0 is easily seen to be always alive and therefore user 0 needs neither heal counters nor the shoot-selector ss. User 0 also never aborts and can always choose $max := 0$.

By corollary 7.16, and because new tags are always chosen to be $0 \pmod n$, only tags $\{-10n, -9n, -8n, \ldots, 9n, 10n\}$ ever occur, and n can be factored out.

The result of removing all these redundancies is shown in figure 7.4 as Construction 4.

The space complexity (taking into account possible savings) is $2 \times sizeof(\text{ABS}) + O(n)$ for the shared variables of user 0, and $sizeof(\text{ABS}) + O(1)$ for the shared variables of the remaining users. This space complexity is the same as that of [9], and, apart from the data field used in the shared variables of Read only users, also the same as that of [14, 13, 17].

7.7 Conclusion

Our construction shows that shared memory can be implemented, using replication, from simple bounded memory cells, with only a small constant factor increase in access time.

To this end, the unbounded solution of Vitányi and Awerbuch, for a long time the only recognized correct solution, is refined by adding a powerful, in itself unbounded, shooting mechanism. This mechanism allows slow, potentially confused operations to safely abort, and allows the remaining operations to interpret the unbounded timestamps of Vitányi and Awerbuch as bounded quantities. The end result follows by showing that the shooting mechanism itself is easily bounded.

```
type shared : record
                value,prev : ABStype
                tag  : −10..10
                ss : 0..1
                shoot,heal : array[0..1][0..n − 1] of −4..9
              end
```

```
procedure Write(i, v)
var j : I
    from : array[0..n − 1] of shared
    static me : shared
begin
    for j ∈ I do from[j] := R_{j,i}
    me.prev := me.value
    for j ∈ I do R_{i,j} := me
    for j ∈ I, s ∈ {0..1} do
        if me.shoot[s][j] ⊖ from[j].heal[s][i] < 6
        then me.shoot[s][j]⊕ := 1
    me.value, me.tag := v, (me.tag + 10) mod 21 − 10
    for j ∈ I do R_{i,j} := me
end
```

```
procedure Read(i)
var j : I
    s : 0..1
    from,tmp : array[0..n − 1] of shared
    static me : shared
begin
    s := 1 − me.ss
    me.heal[s][0] := R_{0,i}.shoot[s][i]
    R_{i,0} := me
    for j ∈ I do from[j] := R_{j,i}
    tmp[0] := R_{0,i}
    if tmp[0].shoot[s][i] ⊖ me.heal[s][0] ≥ 3
    then return tmp[0].prev
    select max ∈ A such that ∀j ∈ A : from[max].tag ⊙ from[j].tag ≥ 0
    where A = {0} ∪ {j ∈ I : from[0].shoot[z][j] ⊖ from[j].heal[z][0] < 6
                             where z = from[j].ss}
    me.value, me.tag, me.ss := from[max].value, from[max].tag, s
    for j ∈ I do R_{i,j} := me
    return me.value
end
```

FIGURE 7.4. Construction 4; single to multi-reader

Bibliography

[1] B. Awerbuch, L. Kirousis, E. Kranakis, P.M.B. Vitányi, *A proof technique for register atomicity*, Proc. 8th Conference on Foundations of Software Technology & Theoretical Computer Science, Lecture Notes in Computer Science 338, pp. 286–303, 1988.

[2] B. Bloom, *Constructing Two-writer Atomic Registers*, IEEE Transactions on Computers, vol. 37, pp. 1506–1514, 1988.

[3] J.E. Burns and G.L. Peterson, *Constructing Multi-reader Atomic Values From Nonatomic Values*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 222–231, 1987.

[4] D. Dolev and N. Shavit, *Bounded Concurrent Time-Stamp Systems Are Constructible*, Proc. 21th ACM Symposium on Theory of Computing, pp. 454–466, 1989.

[5] M.P. Herlihy, *Impossibility and Universality Results for Wait-Free Synchronization*, Proc. 7th ACM Symposium on Principles of Distributed Computing, 1988.

[6] M.P. Herlihy, J. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, ACM TOPLAS vol. 12, No. 3, pp. 463–492, 1990.

[7] A. Israeli and M. Li, *Bounded Time-Stamps*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 371–382, 1987.

[8] A. Israeli and A. Shaham, *Optimal Multi-Writer Multi-Reader Atomic Register*, Proc. 11th ACM Symposium on Principles of Distributed Computing, pp. 71–82, 1992.

[9] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *Atomic Multireader Register*, Proc. 2nd International Workshop on Distributed Computing, Lecture Notes in Computer Science 312, pp. 278–296, July 1987.

[10] L. Lamport, *On Interprocess Communication Parts I and II*, Distributed Computing, vol. 1, pp. 77–101, 1986.

[11] M. Li and P.M.B. Vitányi, *A very simple construction for atomic multiwriter register*, Techn. Report TR 01-87, Aiken Comp. Lab., Harvard University, November 1987. Also, pp. 488-505 in: *Proc. International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989.

[12] N. Lynch and M. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, Proc. 6th ACM Symposium on Principles of Distributed Computing, 1987.

[13] R. Newman-Wolfe, *A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 232–248, 1987.

[14] G.L. Peterson and J.E. Burns, *Concurrent reading while writing II: the multiwriter case*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 383–392, 1987.

[15] G.L. Peterson, *Concurrent reading while writing*, ACM Transactions on Programming Languages and Systems, vol. 5, No. 1, pp. 46–55, 1983.

[16] R. Schaffer, *On the correctness of atomic multi-writer registers*, Technical Report MIT/LCS/TM-364, MIT lab. for Computer Science, June 1988.

[17] A.K. Singh, J.H. Anderson, M.G. Gouda, *The Elusive Atomic Register Revisited*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 206–221, 1987.

[18] J. Anderson, *Composite Registers*, Distributed Computing, to appear.

[19] K. Vidyasankar, *Converting Lamport's Regular Register to an atomic register*, Information Processing Letters, vol. 28, pp. 287–290, 1988.

[20] P.M.B. Vitányi, B. Awerbuch, *Atomic Shared Register Access by Asynchronous Hardware*, Proc. 27th IEEE Symposium on Foundations of Computer Science, pp. 233–243, 1986. (Errata, Ibid.,1987)

[21] C. Dwork, O. Waarts, *Simple and efficient bounded concurrent timestamping, or, Bounded concurrent timestamp systems are comprehensible!,* , Proc. 24th ACM Symposium on Theory of Computing, pp. 655–666, 1992.

[22] A. Israeli, M. Pinhasov, *A Concurrent Time-Stamp Scheme which is Linear in Time and Space*, Proc. 6th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 647, pp. 95–109, 1992.

[23] R. Gawlick, N. Lunch, N. Shavit, *Concurrent Timestamping Made Simple*, Proc. 1st Israel Symposium on Theory of Computing and Systems, pp. 171–183, 1992.

[24] C. Dwork, M. Herlihy, S. Plotkin, O Waarts, *Time-Lapse Snapshots*, Proc. 1st Israel Symposium on Theory of Computing and Systems, pp. 154–170, 1992.

[25] U. Abraham, *On interprocess communication and the implementation of a multi-writer atomic register*, unpublished manuscript.

[26] J. Tromp, *How to Construct an Atomic Variable*, Chapter 6, also in Proc. 3rd International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 392, pp. 292–302, 1989.

8
Binary Snapshots

8.1 Introduction

Consider a concurrent shared memory system. A snapshot memory object shared between n processes is a vector of n memory cells, one 'owned' by each process. All processes can independently and concurrently write to (*update*) the cell they own, and all processes can 'instantaneously' collect (*scan*) all values in the vector in a single operation.

The problem of implementing a wait-free atomic snapshot object was independently proposed and solved by Anderson [6, 7, 8] and Afek et al. [1]. Anderson gives an exponential time[1] solution to this problem using single-writer multi-reader registers, and also considers the multi-writer case in which more than one process may update a particular cell. In his solution for the multi-writer case he uses the single-writer snapshot object as a primitive, so his solution does not rely on multi-writer multi-reader registers. Afek et al. give a polynomial time implementation of a single-writer atomic snapshot object, also using single-writer multi-reader registers. They also consider the multi-writer case, but give a solution using multi-writer multi-reader registers instead.

The atomic snapshot memory object is a powerful tool to construct other atomic wait-free objects, for instance counters, logical clocks, or bounded concurrent time-stamp schemes. Aspnes and Herlihy [3] give a general method to convert a sequential specification of a shared memory object that satisfies certain constraints to a wait-free implementation of that object using an atomic snapshot memory object as a primitive. They also give a a polynomial-time implementation of a wait-free atomic snapshot object.

The main question remains whether it is possible to deterministically imple-

[1] In the literature on this subject the time complexity is usually measured by the number of shared register accesses per operation as a function of the number of processes.

ment an atomic snapshot object with single-writer multi-reader registers such that the time complexity of both the update and the scan operations is linear. Much research has focused on affirming this, by imposing certain restrictions on the applicability of the solutions. In [12], Kirousis et al. present a linear-time solution for the case in which no two scans ever overlap. Dwork et al. [9] introduce the weaker time-lapse snapshot object, and give a linear time implementation of this object. Time-lapse snapshots satisfy the same properties atomic snapshots do, except that the former allow concurrent scans to contradict each other. In [11], Israeli et al. present linear-time implementations for either the update or the scan operations, or for unbalanced systems in which the number of updaters is substantially smaller than the number of scanners, or vice versa. Finally, Attiya et al. [4] introduce the lattice agreement decision problem and show that a solution to this problem can be converted to a wait-free atomic snapshot implementation.

In this chapter we take a similar approach, and reduce the general atomic snapshot problem to a simpler one. We present a bounded, linear time construction of a wait-free implementation of the general atomic snapshot object from an atomic wait-free *binary* snapshot object (where each cell can contain only two values) and a small amount of safe and regular single-writer registers. Thus the search for an efficient atomic snapshot implementation may be restricted to the binary case.

We will use a proof technique proposed in [5], also used in [14] to prove the correctness of some atomic register constructions. The technique is a derivation of Lamport's system as described in [13], where his two precedence relations *precedes* and *can affect* are replaced by a single interval order. We first present the model in Section 8.2, then we state the atomic wait-free snapshot problem in Section 8.3. The protocol is presented in Section 8.4, and is proven correct in Section 8.5.

8.2 The Model

A concurrent shared memory system is a collection of sequential processes communicating asynchronously through shared memory data structures. At any time a process is executing at most one action. A process can at any time decide to start a new action when it is idle, or to finish an ongoing action. The start time of an action a is denoted by $s(a)$ and the finish time by $f(a)$.

We model an execution of the shared memory system by a tuple $\langle \mathcal{A}, \rightarrow \rangle$, where $\mathcal{A}$ is the set of all executed actions ordered by $\rightarrow$ such that a precedes b, $a \rightarrow b$, if $f(a) < s(b)$. We require for any execution $\langle \mathcal{A}, \rightarrow \rangle$ that for any $a \in \mathcal{A}$ there are only a finite number of actions $b \in \mathcal{A}$ with $\neg(a \rightarrow b)$. This way we require an execution to start at some point in time, rather than extending into the infinite past [13, 5]. With this definition, $\rightarrow$ is a special kind of partial order called an *interval order* (i.e. a transitive binary relation such that if $a \rightarrow b$ and $c \rightarrow d$ then $a \rightarrow d$ or $c \rightarrow b$). Now we have abstracted away from the actual time an action occurred, and we can specify the behaviour of actions involving access to the shared memory in terms of the interval order.

If one wishes to implement a certain *compound* shared memory object, one first assumes a set of *primitive* shared memory objects used in the implemen-

tation. Every operation on the compound object is implemented by a *protocol* which invokes actions on these primitive objects. Using the compound object will result in an *implementation* execution. Since every operation on the compound object is implemented by a sequence of actions on the primitive objects, an implementation execution induces a *basic* execution $\langle \mathcal{A}, \rightarrow \rangle$ on the shared memory system. In an implementation execution we model an operation as the set of actions it invokes. The implementation execution itself is modeled by a tuple $\langle \mathcal{O}, \stackrel{o}{\rightarrow} \rangle$, where $\mathcal{O}$ contains all operations invoked during the execution, and where for operations $A, B \in \mathcal{O}$, $A \stackrel{o}{\rightarrow} B$ iff all actions $a \in A$ precede all actions $b \in B$ in $\langle \mathcal{A}, \rightarrow \rangle$.

8.3 Atomic Snapshot Memories

A snapshot memory object on n processes is a vector of n memory cells. A process P_i can both write a new value to the i-th cell in the vector or instantaneously collect all values in the vector in a single operation. In the first case it performs an *update-operation*, in the latter case it performs a *scan-operation*.

We require our implementation to be *wait-free* to allow maximal concurrency, and failure-resilience in the case of crash-failures. An implementation is wait-free if and only if all update and scan operations performed by any process will complete in an a priori bounded number of steps, regardless of the behaviour of the other processes.

Secondly, we require our implementation to be *atomic*. This means that all operations must appear to take effect at one instant of time during the actual time the operation executed[2]. This allows us to 'shrink' the actual execution interval of an operation to a point, and we require a scan to return the values written by the most recent preceding updates. The next paragraph formalises this.

Let $\mathcal{O}$ be the set of all scan and update operations invoked in an implementation execution $\langle \mathcal{O}, \stackrel{o}{\rightarrow} \rangle$ of a snapshot object. Assume for ease of presentation that $\mathcal{O}$ includes n initialising updates, one per processor, that precede all other operations in $\mathcal{O}$. The implementation of an atomic snapshot object is *correct* if for any of its executions $\langle \mathcal{O}, \stackrel{o}{\rightarrow} \rangle$ we can extend $\stackrel{o}{\rightarrow}$ to a total order $\stackrel{o}{\Longrightarrow}$ such that for all scan operations $S \in \mathcal{O}$, S returns for any cell i the value written by the last update $U_i \in \mathcal{O}$ executed by P_i preceding S in $\stackrel{o}{\Longrightarrow}$.

8.4 The Solution

In the next two sections we give our implementation of the n process wait-free atomic snapshot object. The *architecture* describes all primitive shared memory objects used by the *protocols*—one for each type of operation on the shared memory object. The architecture also specifies the initial values of the primitive objects, the operations each process is allowed to perform on them, and the type of values it holds.

---

[2] Although here we refer to the global time model for its more intuitive appeal, we will actually prove atomicity by linearization (cf. [13], and the previous section).

The intuition behind our implementation is quite straightforward: Suppose update operations of P_i write the new value alternately to two registers $val_i[0]$ and $val_i[1]$ (this idea was independently put forward by Haldar and Vidyasankar [10]), after which they use an update on the binary snapshot to inform the scans of the position they wrote to. A scan first performs a scan on the binary snapshot, and tries to read the values from the registers val_i at the positions returned by the binary scan. As later updates may overwrite values before they are read by a concurrent scan, updates perform a scan operation as well, the result of which they write in the register $view_i$. A scan uses a *handshaking* mechanism to detect overwriting updates, in which case it copies the view written by an interfering update.

8.4.1 The Architecture

Our implementation of an n process atomic snapshot memory—with cells of type T—will use one n process binary atomic snapshot object with operations $B\text{-}Update_i$ and $B\text{-}Scan_i$, performed by process P_i. Each cell of this binary snapshot object is initially 0. In addition to this, our n-process atomic snapshot protocol will use the following shared registers. For each $i \in \{1, \ldots, n\}$:

- 2 safe registers of type T, $val_i[0]$ and $val_i[1]$, written by process P_i and read by all. Initially, $val_i[1]$ may be arbitrary, but $val_i[0]$ must be initialised to the desired initial value of cell i of the snapshot vector.
- 1 regular register, $view_i$ (an n-value vector with elements of type T), written by process P_i and read by all, initially arbitrary.
- for each $j \in \{1, \ldots, n\}$: a safe bit c_{ij} (the 'complement'-bit), an atomic bit s_{ij} (the 'start'-bit) and a regular bit e_{ij} (the 'end'-bit). All written by process P_i, read by process P_j and initially 0.

8.4.2 The Protocols

Each of the n processes P_i can execute both updates and scans according to the following protocols

```
Procedure Update_i(value)
    b := 1 − b
    write val_i[b] ← value
    B-Update_i(b)
    for j ∈ {1, ..., n} do
        write s_ij ← (read c_ji)
    write view_i ← Scan_i
    for j ∈ {1, ..., n} do
        write e_ij ← s_ij
```

```
Procedure Scan_i
    for j ∈ {1, ..., n} do
        write c_ij ← 1−(read s_ji)
    b[1..n] := B-Scan_i
    for j ∈ {1, ..., n} do
        read v[j] ← val_j[b[j]]
        if c_ij = (read s_ji) = (read e_ji)
        then return (read view_j)
    return v[1..n]
```

The Update-protocol uses local variables j (ranging over $\{1, \ldots, n\}$), and b, a static bit variable initially 0, which retains its value in between successive invocations of the protocol. The Scan-protocol uses local variables b (an n-bit vector), j (ranging over $\{1, \ldots, n\}$), and v (an n-value vector with elements of type T).

A few words on the programming notation are in order. Some assignments involve both a write and a read or Scan. These are to be executed sequentially, the read/Scan first and then the write. E.g. 'write $s \leftarrow$ (read c)' is shorthand for ' read $t \leftarrow c$; write $s \leftarrow t$'. This should not to be confused with read-modify-write operations that execute atomically. We assume that the value of a shared register written by a process also belongs to that process's local state. This means that the value of for instance the shared variable c_{ij} in the Scan-protocol need not be explicitly read. The return statements in the Scan-protocol serve to return the indicated value to the caller, and to terminate the protocol immediately.

The for loops are indexed over a set to make clear that the n loop bodies may be interleaved arbitrarily. Since the registers accessed in the loop bodies are all disjoint, such a for statement can also be interpreted as a do-in-parallel construct. Thus the parallel time complexity [2] of snapshots equals the parallel time complexity of binary snapshots (up to a constant factor).

8.5 Proof of Correctness

To prove correctness we assume the usual correctness conditions on the read write registers that we use in our implementation. We also assume the correctness of the atomic binary snapshot object used by our implementation. I.e. in an execution $\langle \mathcal{A}, \rightarrow \rangle$ we assume there exists a total order $\Rightarrow$ extending $\rightarrow$ such that every binary scan BS returns for bit i the value written by the last binary update BU_i executed by P_i preceding BS in $\Rightarrow$.

We write U for *Update* and S for *Scan*. For operation $O \in \{U, S, BU, BS\}$, O_i^x denotes the x-th execution of O by process P_i, including scans S_i that are invoked by some update U_i^y. These scans are sometimes written as US_i^y. Note that BS_i^x is invoked by S_i^x, and BU_i^x is invoked by U_i^x. $\mathcal{O}$ contains all invocations S_i^x and U_i^x for $i \in \{1, \ldots, n\}$ and $x \geq 0$. Note that this also includes updates U_i^0 that wrote the initial values for the cells i, and scans US_i^x invoked by updates U_i^x.

If scan S_i^x sees $c_{ij} =$ (read s_{ji}) = (read e_{ji}), then process P_j (or some update U_j) is said to *interfere* with S_i^x. A scan is *direct* if no process interferes with it. S_i^x *contains* S_j^y iff $s(S_i^x) < s(S_j^y) < f(S_j^y) < f(S_i^x)$. The next lemma shows that direct scans will return correct values.

LEMMA **8.1** *Assume P_j does not interfere with some scan S_i^x, and let S_i^x scan the value $b[j]$ updated by some U_j^y, i.e. $BU_j^y \Rightarrow BS_i^x \Rightarrow BU_j^{y+1}$. Then the value $val_j[b[j]]$ read by S_i^x was written there by U_j^y.*

PROOF. Assume scan S_i^x does not see $c_{ij} =$ (read s_{ji}) = (read e_{ji}) and that BS_i^x scanned the value $b[j]$ for cell j updated by BU_j^y, i.e. $BU_j^y \Rightarrow BS_i^x \Rightarrow BU_j^{y+1}$.

The write of $val_j[b]$ by U_j^y precedes BU_j^y in $\rightarrow$, and the read of $val_j[b]$ by S_i^x follows BS_i^x in $\rightarrow$. Since $BU_j^y \Rightarrow BS_i^x$, we have $\neg(BS_i^x \rightarrow BU_j^y)$, so the write of $val_j[b]$ by U_j^y precedes the read of it by S_i^x. So if S_i^x does not read the value written by U_j^y it must be concurrent with or occur after a write to $val_j[b]$ by a later update U_j^z Note that this later update cannot be U_j^{y+1}, since this update will write to $val_j[1-b]$.

Suppose the read of $val_j[b]$ is concurrent with or occurs after a write to it

by an update U_j^z, $z > y+1$. Now $BS_i^x \Rightarrow BU_j^{y+1}$, so by a similar argument as before the read of c_{ij} by U_j^{y+1} occurs after the write of c_{ij} by S_i^x in $\rightarrow$. U_j^{y+1} writes the value of c_{ij} to s_{ji} and later to e_{ji} before S_i^x reads these, since the read of $val_j[b]$ by S_i^x is concurrent with or occurs after a write to it by update U_j^z. Now the values of c_{ij}, s_{ji} and e_{ji} must be equal, and as long as S_i^x does not finish, c_{ij} will not change. This implies that any later writes to s_{ji} and e_{ji} will not change their value and thus, as they are atomic and regular, S_i^x should see $c_{ij} = (\text{read } s_{ji}) = (\text{read } e_{ji})$, a contradiction. □

The next lemma shows that scans that cannot collect the values directly due to interfering updates can copy the result from such an interfering update. This interfering update will have stored the result, called a view, of a direct scan contained in the interfered scan.

LEMMA **8.2** *If process P_j interferes with scan S_i^x, then the view S_i^x copied from $view_j$ is the result of a direct scan S_k^z, contained in S_i^x.*

PROOF. Since S_i^x sees $c_{ij} = (\text{read } s_{ji})$ and s_{ji} is atomic and S_i^x sets $c_{ij} = 1-(\text{read } s_{ji})$, there must be an update U_j^y that changed s_{ji} after S_i^x read s_{ji}. This implies that the scan US_j^y of U_j^y started after S_i^x did. Note that after U_j^y changes s_{ji}, e_{ji} holds the old value of s_{ji} which is unequal to the current value of s_{ji}. Then if S_i^x also sees $c_{ij} = (\text{read } e_{ji})$, U_j^y must have written e_{ji} before or concurrent with the read of e_{ji} by S_i^x. This implies that S_i^x reads $view_j$ after the result of US_j^y was written to it by U_j^y. This also shows that S_i^x contains this US_j^y. Note that $view_j$ must be a regular register, since views written by later updates may interfere with the read of the view by S_i^x. □

We conclude by proving the correctness of our implementation of the atomic snapshot object. The implementation is obviously wait-free.

THEOREM **8.3** *For any execution $\langle \mathcal{O}, \xrightarrow{o} \rangle$ there exists a total extension $\overset{o}{\Longrightarrow}$ of $\xrightarrow{o}$ such that any scan S_i^x with $U_j^y \overset{o}{\Longrightarrow} S_i^x \overset{o}{\Longrightarrow} U_j^{y+1}$ returns for cell j the value written by U_j^y.*

PROOF. For direct scans S_i^x, let $\beta(S_i^x) = BS_i^x$. For indirect scans S_i^x that copied the view collected by a direct scan S_j^y (see lemma 8.2), let $\beta(S_i^x) = BS_j^y$. Finally, for updates, let $\beta(U_i^x) = BU_i^x$.

For any two $A, B \in \mathcal{O}$, define $A \overset{o}{\Longrightarrow} B$ if $\beta(A) \Rightarrow \beta(B)$. Note that neither $A \overset{o}{\Longrightarrow} B$ nor $B \overset{o}{\Longrightarrow} A$ iff $\beta(A) = \beta(B)$. By lemma 8.2, $\beta(S)$ occurs inside S for any indirect scan S. This implies that if $A \xrightarrow{o} B$ we have $\beta(A) \Rightarrow \beta(B)$ and thus $A \overset{o}{\Longrightarrow} B$. So $\overset{o}{\Longrightarrow}$ extends $\xrightarrow{o}$. Now extend $\overset{o}{\Longrightarrow}$ to a total order.

If for some scan S_i^x, $U_j^y \overset{o}{\Longrightarrow} S_i^x \overset{o}{\Longrightarrow} U_j^{y+1}$, then $BU_j^y \Rightarrow \beta(S_i^x) \Rightarrow BU_j^{y+1}$ by the definition of β and $\overset{o}{\Longrightarrow}$ (Note that if $\beta(A) = \beta(B)$, then both A and B are scans). If S_i^x is a direct scan, then $\beta(S_i^x) = BS_i^x$ and by lemma 8.1 the theorem is proved. If S_i^x is not a direct scan, then it copied the result from a direct scan S_k^z, and thus $\beta(S_i^x) = BS_k^z$. But again by lemma 8.1 the theorem is satisfied. □

8.6 Future Research

Further research might be directed at finding an implementation of atomic binary snapshots with subquadratic or linear time complexity.

It is interesting to note that all atomic snapshot implementations we are aware of use at least $O(n)$ registers with $O(nv)$ size (where v is the maximal number of bits contained in any cell of the snapshot object). However, Dwork et al. [9] have shown that for time-lapse snapshots $O(n^2)$ registers with size $O(n+v)$ suffice. It is an interesting open question whether registers with size $O(nv)$ are necessary to implement atomic snapshot objects.

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, *Atomic snapshots of shared memory*, Proc. 9th ACM Symposium on Principles of Distributed Computing, pp. 1–13, 1990.

[2] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi, *Wait-free test-and-set*, Proc. 6th International Workshop on Distributed Algorithms, LNCS 647, pp. 85–94. Springer Verlag, 1992.

[3] J. Aspnes and M. P. Herlihy, *Wait-free data structures in the asynchronous pram model*, Proc. of the 2nd Ann. Symp. on Parallel Algorithms and Architectures, pp. 340–349, 1990.

[4] H. Attiya, M. Herlihy, and O. Rachman, *Efficient atomic snapshots using lattice agreement*, Proc. 6th International Workshop on Distributed Algorithms, LNCS 647, pp. 35–53. Springer Verlag, 1992.

[5] B. Awerbuch, L. M. Kirousis, E. Kranakis, and P. M. B. Vitányi, *On proving register atomicity*, Proc. of the 8th Conf. on Foundations of Software Technology and Theoretical Computer Science, pp. 286–303, 1988.

[6] J. H. Anderson, *Composite registers*, Technical Report TR-89-25, Department of Computer Science, The University of Texas at Austin, 1989.

[7] J. H. Anderson, *Multiple-writer composite registers*, Technical Report TR-89-26, Department of Computer Science, The University of Texas at Austin, 1989.

[8] J. H. Anderson, *Composite registers*, Proc. 9th ACM Symposium on Principles of Distributed Computing, pp. 15–29, 1990.

[9] C. Dwork, M. Herlihy, S. A. Plotkin, and O. Waarts, *Time-lapse snapshots*, Israel Symposium Theory of Computing and Systems, LNCS 601, pp. 154–170. Springer Verlag, 1992.

[10] S. Haldar and K. Vidyasankar, *Elegant constructions of atomic snapshot variables*, Unpublished manuscript, 1992.

[11] Amos Israeli, Amnon Shaham, and Asaf Shirazi, *Linear-time snapshot protocols for unbalanced systems*, Technical Report CS-R9236, CWI, 1992.

[12] L. M. Kirousis, P. Spirakis, and P. Tsigas, *Reading many variables in one atomic operation: Solutions with linear or sublinear complexity*, Proc. 5th International Workshop on Distributed Algorithms, LNCS 579, pp. 229–241. Springer Verlag, 1991.

[13] L. Lamport, *On interprocess communication part i: basic formalism*, Distributed Computing 1(2), pp. 77–85, 1986.

[14] M. Li, J. Tromp, and P. M. B. Vitányi, *How to share concurrent wait-free variables*, Chapter 7 (also under revision for *Journal of the ACM*).

9
On Update-Last Schemes

9.1 Introduction

Let $X = X_1 \times \cdots \times X_n$ be the product of n domains, one for each $i \in I = \{1, \ldots, n\}$. Elements of the X_i are called *labels*, and those of X *label vectors*. We say that two label vectors l and l' are i-equivalent, $l \equiv_i l'$, whenever $\forall_{j \neq i} : l_j = l'_j$.

DEFINITION **9.1** An Update-Last scheme is a partial function $last : X \to I$ with a non-empty domain $U \subseteq X$ such that

$$\forall\, l \in U, i \in I\ \exists\, l' \in U : l \equiv_i l' \wedge last(l') = i,$$

In this chapter we derive exact bounds on the domain sizes $|X_i|$ that allow the existence of such a scheme.

One can think of an Update-Last scheme as providing a method whereby each of a set of n objects can be made a 'leader' by choosing its label appropriately. The objects can be either active entities that carry out the label-inspection and choice-making themselves, or passive objects in some system that wants to keep track of which object is special. The vector of labels can be seen as a way of storing an index in a distributed fashion.

The above formalization is intended to capture the essence of such methods, which may appear in different forms and shapes. The possibility of *last* not being total accommodates methods where not all possible label combinations can arise.

The main motivation for this work comes from the implementation of wait-free multi-writer registers from single-writer ones. Update-Last schemes immediately provide for *serial* implementations that work correctly as long as no two operations overlap: a writer tags each new value with a label that shows this value to be the last-written or current one, while a reader simply collects all value-label pairs and returns the value of the last writer. Conversely, any serial

implementation of a multi-writer register where the number of writers does not exceed the number of values induces an Update-Last scheme. The exact bounds proved here on label domain sizes thus yield a lower bound on the space complexity of real concurrent implementations. The multi-writer implementation of Israeli and Shaham [4] is the first optimal one in the sense of achieving polynomial label domain size, i.e. *logarithmic* label size when measured in bits.

9.2 Related Work

Israeli and Li [3] introduced *time-stamp schemes* as a method of representing a total order on a dynamic set of items, and proved a lower bound of $2^n - 1$ on the size of the label domain. Li and Vitanyi [5] present an Update-Last scheme where each X_i is of size n, and argue that for the purpose of implementing multi-writer registers, one does not need the full functionality of time-stamp schemes.

The unlabeled (non-indexed) case, where *last* gets as input a set of n different labels from a single domain, and maps to one of them, was considered by Cori and Sopena [2]. They proved a tight bound of $2n - 1$ on the size of the label domain. The proof of our main theorem was in part inspired by their proof.

Another surprising application is in the simulation of a DFA by a (fully-connected) asynchronous cellular automaton [1]. Such an ACA has one node for each letter in the alphabet, which is activated when that letter appears in the input. Upon activation, the node changes its state according to the states of all its neighbours. In the simulation, part of a node's state is the state of the simulated DFA, and an Update-Last scheme allows an activated node to identify the node holding the current DFA state.

9.3 Characterizing Update-Last schemes

THEOREM **9.2** *There exists an Update-Last scheme on label vector space X iff* $\sum_{i \in I} 1/|X_i| \leq 1$.

PROOF.

$\Rightarrow$ Let U_i be $U/\equiv_i$ and U' be the disjoint union of all U_i. Each element in U_i is an equivalence class of label vectors differing only in component i. Such a class can also be thought of as a label vector in which component i has been blanked.

For a fixed i, consider the number of pairs $(l, x) \in U \times U_i$ with $l \in x$. There is one such pair for each $l \in U$ and hence $|U|$ in total. On the other hand, for $x \in U_i$, there are at most $x_i \stackrel{\text{def}}{=} |X_i|$ such pairs. This gives $|U| \leq |U_i| x_i$, or $1/x_i \leq |U_i|/|U|$. By definition of U', we also have that $|U'| = \sum_{i \in I} |U_i|$. Thus,

$$\sum_{i \in I} 1/x_i \leq \sum_{i \in I} |U_i|/|U| = |U'|/|U|.$$

Given that $last : X \to I$ is an Update-Last scheme, there exists a function $new : U' \to U$ satisfying

$$\forall x \in U_i : new(x) \in x \wedge last(new(x)) = i.$$

Note that new is invertible (one-one), since $new^{-1}(l)$ is just the equivalence class of l w.r.t. $\equiv_{last(l)}$. Hence $|U'| \leq |U|$ and the result follows.

$\Leftarrow$ W.l.o.g. assume that $X_i = \{0, 1, \ldots, x_i - 1\}$. Given that $\sum_{i \in I} 1/|X_i| \leq 1$, we can partition the $[0, 1)$ interval into n disjoint half-open intervals such that i's interval has length at least $1/x_i$. Now define the total function $last(l)$ to be the index whose interval contains $(\sum_{i \in I} l_i/x_i) \bmod 1$ ($x \bmod 1$ denotes the fractional part of x). To see that $last$ is an Update-Last scheme, note that for any $l \in U$ and $i \in I$, the set $\{last(l') : l' \equiv_i l\}$ consists of x_i points evenly spread around the $[0, 1)$ interval and thus necessarily intersects i's interval. □

Putting $y_i = 1/|X_i|$, the condition of the theorem becomes $\sum_{i \in I} y_i \leq 1$. By standard convexity arguments, $\prod_{i \in I} y_i$ is maximal (and hence $\prod_{i \in I} |X_i|$ minimal) under this condition when all y_i equal $1/n$. This proves the following lower bound on the number of label vectors:

COROLLARY **9.3** An Update-Last scheme with label vector space X satisfies $\prod_{i \in I} |X_i| \geq n^n$.

This proves the optimality of the construction presented in [5], which is essentially an Update-Last scheme with $|X_i| = n, i = 1, \ldots, n$.

9.4 Further Work

There are several directions in which Update-Last schemes can be generalized. One would be a scheme where i can choose its label in a way that makes j become last, for all $i, j \in I$. Theorem 1 shows a way to do this when all $|X_i|$ equal n, but in general the condition $\sum_{i \in I} 1/|X_i| \leq 1$ will not be sufficient any more.

More interesting, perhaps, is the study of general schemes where i can choose its label so as to satisfy some constraint on a function of all the labels, where this function might represent the state of a shared object.

Bibliography

[1] R. Cori, Y. Metivier, W. Zielonka, " Asynchronous mappings and asynchronous cellular automata", LaBRI technical report 89-97, Université Bordeaux, December 1989.

[2] R. Cori, E. Sopena, "Some Combinatorial aspects of Time Stamp Systems", submitted to Journal of Algorithms.

[3] A. Israeli, M. Li, "Bounded Time-Stamps", Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, pp. 371–382, 1987.

[4] A. Israeli, A. Shaham, "Optimal Multi-Writer Multi-Reader Atomic Register", Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing, pp. 71–82, 1992.

[5] M. Li, P.M.B. Vitányi, "Optimality of Wait-Free Atomic Multi-Writer Variables", technical report CS-R9128, CWI, June 1991.

Samenvatting (Dutch)

Het proefschrift bundelt een kleine verscheidenheid aan studies binnen de theoretische informatica. De meeste verhandelen over zogenaamde *algorithmen*, d.w.z. stapsgewijze methoden voor het oplossen van een probleem. U kunt hierbij denken aan de methode die U op de lagere school heeft geleerd voor het uitvoeren van staartdelingen. Het aftrekken van een enkel cijfer van een andere is dan een goed voorbeeld van een (primitieve) stap. Het aftrekken van de hele noemer van een gedeelte van de teller zou dat niet zijn, daar deze operatie meer moeite zal kosten naarmate de noemer groter wordt (uit meer cijfers bestaat). Het idee van een stap is dat het een constante hoeveelheid werk vertegenwoordigt, onafhankelijk van de mogelijke grootte van het probleem.

De aard van de stappen waarin een algorithme geformuleerd is kan zeer sterk uiteenlopen, evenals de 'werkomgeving' waarbinnen het uitgevoerd dient te worden. De verschillende hoofdstukken geven een aardige indruk van de mogelijke variatie.

Het interessante vraagstuk bij veel problemen is hoeveel middelen er voor nodig zijn om het op te lossen. De 2 meest bekeken middelen zijn tijd en geheugen(ruimte). Zo gebruikt de klassieke staartdeling bijvoorbeeld een hoeveelheid tijd (aantal stappen) en geheugen (papieroppervlak) die proportioneel is met zowel het aantal cijfers van de teller als van de noemer. We zeggen dan: de tijds (of geheugen) complexiteit van staartdelen is van de order het produkt van aantal cijfers in noemer en aantal cijfers in teller. Een complexiteits maat zegt altijd hoe het gebruik van middelen toeneemt met de grootte van de probleem instantie.

Naast geheugen en tijd komen ook minder bekende middelen aan bod, zoals energiegebruik in electronische circuits.

Zelfs de grootte van de oplossing kan men als te minimaliseren 'middel' zien, ingeval die oplossing niet uniek is. Zo wordt in Hoofdstuk 2 het volgende probleem bestudeerd: gegeven een verzameling woorden $w_1, w_2, \ldots, w_n$, zoek een woord w waar al alle w_i deel van uit maken. Zo zijn bijvoorbeeld 'stoel' en

'kist' deelwoorden van 'kistoel'. Onder een algemeen aanvaarde assumptie is het niet mogelijk om altijd het kortst mogelijke *superwoord* in redelijke tijd te vinden. Om precies te zijn, is er voor willekeurige k geen algorithme, dat van een verzameling woorden van totale lengte m altijd het kortste superwoord oplevert binnen m^k stappen.

Er is wel een bekend algorithme dat in iets als m^3 stappen een redelijk kort superwoord weet te vinden. Dit algorithme wordt wel gebruikt bij het 'DNA-sequencen', het bepalen van de exacte basen-volgorde van DNA-moleculen. Voorheen was niet bekend of de lengte van superwoorden gevonden met dit algorithme wel begrensd is tot een constant aantal keer de kortst mogelijke lengte. In dit hoofdstuk wordt afgeleid dat dit inderdaad zo is, en wel met een grens van 4 maal de kortste.

In het volgende hoofdstuk staat de geheugen-complexiteit van 'plaatjes inkleuren' centraal. Er wordt een algorithme afgeleid dat dit probleem met een constante hoeveelheid geheugen (interne ruimte) oplost, in tegenstelling tot de in praktijk (b.v. tekenprogramma's) gebruikte methoden die geheugen proportioneel in de grootte van het in te kleuren figuur nodig hebben. Bij gebruik van slechts een constante hoeveelheid geheugen is de werkomgeving als een gigantisch doolhof in de kleuren wit en zwart, waarbij de hele witte omgeving van de startpositie (die dus begrensd is door zwarte stukken) zwart geverfd moet worden.

Hoofdstuk 4 gaat vervolgens in op de schakel-energie van circuits die de binaire (2 waardig; 0 of 1) of-functie (en generalisaties daarvan) berekenen op n inputs. Door een kleine redundantie in de hoeveelheid schakel-elementen kan de hoeveelheid bedrading die omschakelt bij verandering van de inputs met een factor l gereduceerd worden, waarbij l het aantal cijfers van n is.

In Hoofdstuk 5 wordt een nieuw soort computer voorgesteld, waarvan het geheugen bestaat uit 'cellen' die elkaar aan kunnen wijzen, elke cel wijst bijvoorbeeld naar 3 andere, met een rode, groene, en blauwe wijzer. Op elk moment van de berekening is er een speciale cel van waaruit de andere ge-adresseerd kunnen worden. Niet alleen enkele cellen kunnen ge-adresseeerd worden, ook kan vanuit een cel alle cellen die er naar wijzen, bevoorbeeld met de groene wijzer, tegelijk ge-adresseerd worden. Op een verzameling ge-adresseerde cellen kunnen allerlei operaties worden uitgevoerd, zoals het creëren van nieuwe cellen, of het omleggen van wijzers. Dit blijkt een bijzonder krachtige soort computer op te leveren, die bijvoorbeeld in een relatief klein aantal stappen het schaakspel zou kunnen oplossen (hoewel het aantal cellen wat in die berekening aangemaakt wordt wel verschrikkelijk groot wordt.)

In de laatste 4 hoofdstukken komen communicatie protocollen aan bod, die verschillende processen de indruk geven dat ze gezamenlijk één geheugen delen. Wat de een erin schrijft kan een ander dan lezen. Het begint heel eenvoudig met een enkel bit (2-waardig geheugenelement) dat door slechts een proces kan worden geschreven en door een ander gelezen. Vanuit deze bouwstenen kunnen geheugens met veel meer mogelijkheden worden gebouwd: meer waardes, meer lezers, meer schrijvers, enz. Het unieke van deze protocollen is dat ze *wachtvrij* zijn, wat wil zeggen dat het ene proces nooit op het ander hoeft te wachten om zijn geheugen-operatie te kunnen voltooien. Er kan rustig tegelijkertijd in gelezen en geschreven worden, zonder dat er onzin uit komt.

Curriculum Vitae

Johannes Theodorus Tromp

13 mei 1966:	geboren te Alkmaar
1978–1984:	Atheneum Han Fortman College te Heerhugowaard
1984–1989	Doctoraal Informatica Universiteit van Amsterdam
1989–1993	Onderzoeker in opleiding in het project “Algorithms and Complexity” van het CWI te Amsterdam