June 1985

Report No. STAN-CS-85-1055

A Programming and Problem-Solving Seminar

by

Ramsey W. Haddad and Donald E. Knuth

Department of Computer Science

Stanford University
Stanford, CA 94305

Computer Science Department

A PROGRAMMING AND PROBLEM-SOLVING SEMINAR

by

Ramsey W. Haddad and Donald E. Knuth

This report contains edited transcripts of the discussions held in Stanford's course **CS204,** Problem Seminar, during winter quarter **1985.** Since the topics span a large range **of** ideas in computer science, and since most of the important research paradigms and programming paradigms were touched on during the discussions, these notes may be of interest to graduate students of computer science at other universities, as well as to their professors and to professional people in the "real world."

The present report is the sixth in a series of such transcripts, continuing the tradition established in **STAN-CS-77-606** (Michael J. **Clancy, 1977),** STAN-CS-79-707 (Chris Van Wyk, **1979), STAN-CS-U-863 (Allan** A. Miller, **1981), STAN-CS-83-989** (Joseph S. Weening, **1983), STAN-CS-83-990** (John D. Hobby, **1983).**

The production of this report was supported in part by National Science Foundation grant MCS-83-00984.

Table of Contents

Index to problems in the six seminar transcripts

STAN-CS-77-606	Map drawing
	Natural language numbers
	Efficient one-sided list access
	Network Design
	Code generation
STAN-CS-79-707	The Camel of Khowârizm
	Image cleanup
	Cubic spline plotting
	Dataflow computation with Gosper numbers
	Kriegspiel endgame
STAN-CS-81-863	Alphabetic Integers
	King and queen versus king and rook
	Unrounding a sequence
	Grid layout
	File transmission protocols
STAN-CS-83-989	Penny flipping
	2 x 2 x 2 Rubik's cude
	Archival maps
	Fractal maps
	Natural language question answering
STAN-CS-83-990	Bulgarian solitaire
	Restricted planar layout
	Dynamic Huffman coding
	Garbage collection experiments
	PAC-land and PAC-war
This report	Monotonic Squares
	Code-breaking
	Hardware fault detection
	Distributed stability
	High-tech art

Cast of characters

Professor

DEK	Donald E. Knuth

Teaching Assistant

RWH	Ramsey W. Haddad

Students

DEA	Dimitrios E. Andivahis
MJB	Marianne J. Baudinet
PCC	Pang-Chieh Chen
RFC	Roger F. Crew
MDD	Mike D. Dixon
ARG	Anil R. Gangolli
AG	Andrew R. Golding
AAM	Arif A. Merchant
KAM	Kathy A. Morris
KES	Karin E. Scholz
AGT	Andrew G. Tucker
AJU	Amy J. Unruh
JFW	John F. Walker
RMW	Richard M. Washington
JIW	John I. Woodfill

Lecturing Visitors

SB	Susan Brenner
MK	Matthew S. Kahn
DK	David Kriegman

Introduction

Notes for Tuesday, January 8

DEK started out the first class by giving an overview of the history and goals of "CS204 — Problem Seminar". The course has had a long history. It started out under George Pólya; some of the other instructors, besides DEK, have been George Forsythe, Bob Floyd, Bob Tarjan, and Ernst Mayr. Over the years, the title of the course has varied to include various combinations of "Problem Solving" and "Programming". These various names all reflect the nature of the course, which is: using computer programming as one of the tools brought to bear in problem solving.

This course has many goals. DEK indicated that the main one was for the students to gain a better practical understanding of the problem solving process. The students will also be exposed to many general problem solving and programming techniques, representative of many different areas and paradigms within Computer Science. The course will also give students experience in working in teams to solve problems.

DEK said that he was already done with the hardest part of the course (for him). This involved choosing the 5 problems that will be attacked during the quarter. He has looked at the problems enough to tell that they should lead to interesting sub-problems. (Note that none of these problems has previously been solved.) He thinks that this is a very good set of problems, perhaps his best ever!

Then, DEK went over some administrative details. He mentioned that one of the thrills about this course is to see students in their teams enthusiastically discussing Computer Science with each other. (He had mentioned earlier that this course was no longer required, so that students who did not enjoy the class would not be trapped into taking it, and hence possibly spoiling the atmosphere for the others.) He also informed the students that they had already earned **A** grades, hence they only needed to concentrate on getting the most that they could out of the course.

Course Schedule:

Week	Tuesday	Thursday
Jan. 8	Intro & P 1	Problem 1
Jan. 15	Problem 1	Problem 1
Jan. 22	Problem 2	Problem 2
Jan. 29	Problem 2	Lecture by Kahn
Feb. 5	Problem 3	Demonstrate P 2
Feb. 12	Problem 3	Devices for P 5
Feb. 19	Problem 3	Problem 3
Feb. 26	Problem 3	Problem 4
Mar. 5	Problem 4	Problem 4
Mar. 12	Problem 4	Display P 5

Next, DEK pointed out that this course was being video-taped for the first time. Unlike many other courses, where the video tapes are erased at the end of the quarter, the tapes for this course would be the first volume of a new Stanford videotape archive, which will consist of courses chosen because their unique content will not be repeated in later years. These video tapes will be available to people both inside and outside the University, just as technical reports are now available. A tech report will be published to chronicle the class meetings, but the video tapes will give other people a better chance to see the problem solving process in action in a way that a written report just can't capture.

DEK mentioned that work in this course occasionally leads to the publication of a paper describing some interesting discoveries that the class has come up with. These are usually small discoveries, but he pointed out that computer science is built up of many small discoveries, rather than a few big ones. Besides progressing by weaving together small discoveries, Computer Science also progresses by generalizing the techniques that led to the small discoveries, then applying them to other problems.

Rules of the Game

Problems: There are five problems and we will take them in order, spending about two weeks on each. Problem 5 should be done by each student working alone, and everybody should begin working on it soon, while doing the other ones. Students should work in groups of two or three on problems **l-4,** with no two students working together on more than two of the four assignments; then everybody will get to know almost everybody else. We stress cooperation and camaraderie, not concealment and competition!

Grading: Students should hand in a well-documented listing of their computer programs for each problem, along with a writeup that describes the approaches taken. This writeup should include a discussion of what worked or didn't work, and (when appropriate) it should also mention promising approaches to take if there were extra time to pursue things further. The written work will be graded on the basis of style, clarity, and originality, as well as on program organization, appropriateness of algorithms, efficiency, and the correctness of the results. These grades will be given on an A-E scale for the sake of information, but each student's overall grade for the course will be either "A" or "nothing."

Class notes: Classroom discussions will mostly involve the homework problems, but we will try to emphasize general principles of problem solving that are illustrated by our work on the specific problems that come up. Everyone is encouraged to participate in these discussions, except that nobody but Knuth will be allowed to talk more than three (3) times per class period. After class, the TA will prepare notes about what went on, so that everybody will be able to participate freely in the discussions instead of worrying about their own note-taking.

Caveat: This course involves more work than most other S-unit courses at Stanford.

Problem 1

Monotonic squares: ***Find all positive integers n such that the decimal digits of n and n^2 are both in nondecreasing order from left to right.***

Notes for Tuesday, January 8

After noting that $38^2 = 1444$ and $1^2 = 1$ were solutions, DEK asked for any first thoughts about how to go about solving the problem. Someone mentioned that $12^2 = 144$ is also a solution.

MDD noted that $n = 1..7$ all work and that n = 8..9 dont.

It was quickly seen that $n = 10..11$ fail, and that $n = 12..13$ work.

DEK noted that this method could give us our first algorithm:

```
Algorithm 1:
    1. Start with n = 1.
    2. If $n$ and $n^2$ both have nondecreasing digits, then print $n$.
    3. Increment n.
    4. Goto 2.
```

KES wondered whether there might be an infinite number of solutions.

DEK said that if that was true, then we wouldnt want to use this first algorithm.

RMW wondered how we could hope to get all solutions if there were an infinite number of them.

Many people noted that we might be able to characterize an infinite number of solutions by giving a finite-length description of a grammar, or function, that described an infinite set of numbers.

DEK noted that even if there were only a finite number of solutions, the largest n that was a solution might still be very big. Hence, we should look at better ways of finding our answer.

AAM suggested that we try to analyze the ns from right to left.

Notational diversion: Let $s = n^2$. Let $n = n_j ... n_2 n_1 n_0$. Let $s = s_k ... s_2 s_1 s_0$.

DEK wrote on the board that if $n_0 = 3$ then se = 9. Also, $n_0 = 9 \Longrightarrow$ so = 1 and $n_0 = 7 \Longrightarrow s_0 = 9$.

Following AAM's lead, he then started examining in further detail the case when $n_0 = 9$, and hence $s_0 = 1$. He tried to see what we could derive about n_1 and s_1.

$$(n_1 9)^2 \doteq (10 \times n_1 + 9)^2 = 100 \times n_1^2 + 180 \times n_1 + 81$$

Hence, $s_1 \equiv 8 \times n_1 + 8 \pmod{10}$. T his means that s_1 is even. Since $s_1 \leq$ se, we can conclude that $s_1 = 0$. Hence all the s_i's for $i \geq 1$ are equal to zero. This makes it clear that there are no solutions with $n_0 = 9$.

DEK ended the first class with a reminder for the students to form into teams.

Notes for Thursday, January 10

Thursday, January 10

After singing a round of "Happy Birthday" to both DEK and MJB, the class settled down to work on Problem **1.**

DEK asked if anyone had completely solved the problem yet. No one had.

ARG said that he could show an infinite number of solutions.

MDD/RFC said that they had found several such families.

Noting that MDD and RFC had done some work as a team, DEK asked if everyone had formed teams. It turned out that no one else had. RWH's expression betrayed disapproval at the idea of having to grade **15** separate project reports.

AG said that after working on the problem, he kept on coming to a dead end when he tried to think of a way to prove that there were no further solutions other than some set of discovered solutions.

Notational diversion: $6^{(i)}$ means i *6's* in a row. 6^* means an arbitrary number ***of 6's in a row.*** 6^i ***means 6*** raised to ***the ith power.***

The class now looked closer at the families of infinite solutions. The one that ARG had found is $n = 6^*7$. The first three examples are: $7^2 = 49$, $67^2 = 4489$, and $667^2 = 444889$.

DEK asked if ARG could ***prove*** that $6^{(m)}7^2 = 4^{(m+1)}8^{(m)}9$.

ARG said that it could be proved by induction.

DEK agreed that it probably could be shown by induction after looking at:

$$\begin{aligned} 6667^2 &= (6000+667)^2 \\ &= 36000000 + 8004000 + 667^2 \\ &= 44004000 + 444889 \\ &= 44448889. \end{aligned}$$

MDD pointed out that besides this family he had found the infinite classes: $n = 3^*4$, $n = 3'5$, $n = 3'7$, and $n = 3^*6^*7$. Some specific examples from these classes are: $3334^2 = 11115556$, $3335^2 = 11122225$, $3337^2 = 11135569$, and $33333667^2 = 1111133355666889$.

DEK was amazed by the $n = 3^*6^*7$ solution and asked if there was a proof for it.

MDD said that he only had empirical evidence, and that he hadn't tried proving it yet.

MJB said that she had noticed that in all of the solutions that she had found (even those not in the infinite classes), only the digits **1, 3,** or *6* ever appeared as the left-most digit in n (other than in one digit solutions). No one had a proof that her observation was true of all solutions.

AJU pointed out that $n = 16^*7$ also seemed to be a class of infinite solutions.

DEK re-raised two questions. Can we ***prove*** that the interesting two-parameter solutions of the form $n = 3^{(m)}6^{(p)}7$ always work? How will we know when we have all of the solutions?

AG suggested that maybe some sort of pumping lemma would work. Since some of the people were unfamiliar with pumping lemmas, AG explained that pumping lemmas say that after a solution string gets bigger than a certain length, then by repeating certain parts of the string, new solutions could be generated.

DEK continued that this arises in context free languages. It can be shown to be true by looking at the derivation of a long enough solution string. Since there are a fixed number of non-terminal characters and production rules in the grammar, once a string gets to be a certain size then you know that some non-terminal character must have been expanded into itself (and some other text) once. If the character can be expanded into itself once then it can be done twice, three times, . . . Each time, however, pumps up the string by inserting repetitions of a sub-string.

DEK was not sure that this idea would be useful unless we could show that squaring is somehow a context free operation. Since squaring involves "carrying", he doubted that this was so, unless the monotonicity restriction on the digits somehow forced it. He thought it was unlikely.

He wanted to return to the case of n = **3*6*7,** so he started building up a chart [see completed chart below].

AAM rushed in to class, late, carrying a printout hot off the presses. He said that he believed that the classes n = **16*7,** n = **3*4,** n = **3*5,** and n = **3*6*7** were a complete set of the infinite classes of solutions.

DEK noted that if this was our conjecture, then we may be able to prove it without using computers.

KES agreed, pointing out that we could use induction to prove that the infinite sets were solutions (like we had with n = 6*7). And that we could use an extension of the method we used on Tuesday (showing that no solutions have n_0 = 9) to show that there were no other solutions.

DEK completed the chart that he had started earlier:

$7^2 = 49$	$37^2 = 1369$	$337^2 = 113569$
$67^2 = 4489$	$367^2 = 134689$	$3367^2 = 11336689$
$667^2 = 444889$	$3667^2 = 13446889$	$33667^2 = 1133466889$
$6667^2 = 44448889$	$36667^2 = 1344468889$	$336667^2 = 113344668889$

DEK then challenged the class to figure out the pattern (thus possibly allowing us to do a proof by induction).

No one did, but RMW suggested that n = 337 should be regarded as an anomaly.

DEK suggested that we look for an algebraic explanation.

KAM noted that $6^{(n)}7 = \frac{2\times10^{n+1}+1}{3}$.

From this, DEK thought that it might be a good idea to play with $n = 3^{(m)}6^{(p)}7$ by multiplying it by *3.* This yielded: $3 \times n = 10^{(m-1)}10^{(p)}1$. Hence:

$$(3 \times n)^2 = (10^{m+p+1} + 10^{p+1} + 1)^2$$
$$= 10^{2\times m+2\times p+2} + 2\times10^{m+2\times p+2} + 2\times10^{m+p+1} + 10^{2\times p+2} + 2 \times 10^{p+1} + 1$$

DEK noted that we can see that this is divisible by 9, since its digits sum to 9 (besides already knowing it because of how we generated the number). So now let's compute $\frac{(3\times n)^2}{9}$.

MDD mentioned that it would be good if we had a notation for a string of **1's.**

DEK said that was an excellent idea; he remembered reading that such numbers have been called ***rep-units.***

Thursday, January 10

Notational diversion: A rep-unit: $R_m = \frac{10^m-1}{9} = 1^{(m)}$

DEK mentioned that rep-units are the base-ten analogs of Mersenne numbers $2^n - 1$, and that he thought he had read that R_{17} is prime. [After class he looked at the literature and found a note by Yates in ***Journal of Recreational Math*** 9 **(1976), 278-279,** stating that R_2 and R_{19} and R_{23} are prime but all other $R_n < R_{1000}$ are composite. Also $R_{17} = 2071723 \cdot 5365222357$.]

Continuing from above:

$$(3 \times n)^2 = (10^{2\times m+2\times p+2} - 1) + 2 \times (10^{m+2\times p+2} - 1) + 2 \times (10^{m+p+1} - 1) \\ + (10^{2\times p+2} - 1) + 2 \times (10^{p+1} - 1) + 9;$$

$$n^2 = \frac{(3 \times n)^2}{9} \\ = R_{2\times m+2\times p+2} + 2 \times R_{m+2\times p+2} + 2 \times R_{m+p+1} + R_{2\times p+2} + 2 \times R_{p+1} + R_1.$$

From this it is obvious that $(3^{(m)}6^{(p)}7)^2$ has monotonically non-decreasing digits. Indeed, when we look at things this way, we have a surprisingly nice characterization of such numbers:

A number satisfies our non-decreasing digits criteria $\Longleftrightarrow$ it is the sum of up to 9 R_m's.

Now we have a direct way to show that our infinite sets of possible solutions are actually all solutions, and we don't have to use induction.

AG brought up the fact that we still were at a dead end in trying to show that there were no other infinite sets of solutions.

DEK started speculating about what our results using rep-units might imply about the solutions to this problem in bases other than base 10. It seems that the infinite classes arise because the rep-units are gotten by dividing by 9, and 9 is a perfect square. Hence, if this is the source of the infinite classes, we would expect no infinite sets of solutions for base 8, but we would for base 17.

MDD reported that he had looked at bases 2 and 3, but that nothing interesting had happened.

DEK returned to the point brought up by AG and KES. We still needed to show that we can describe all possible solutions. He asked the class if they had any ideas about how to show that we weren't missing anything.

AAM said that we could do a search and prune the suffixes that wouldn't go any further (plus the infinite ones).

DEK wondered how he would decide when to prune.

AAM indicated that he would be able to tell by the occurence of certain patterns.

DEK asked if AAM would have the computer do the induction.

AAM said no.

DEK pointed out that if we were going to have a program decide when to prune, we could either give the program all the axioms of math from which the program would have to decide when it could prune the tree, or we could give it just a few carefully selected rules. If we are to give it a carefully selected subset, what subset is adequate for the task at hand?

AAM tried to explain what he meant, by using the following example: If you were trying to find n_3 such that n_3334 was an allowable suffix, then, since $334^2 = 111556$, we know that:

$$0 < 8 \times n_3 + 1 \equiv s_3 \quad (\mathrm{mod\ IO}) \leq 5$$

Since n is non-decreasing, then we know that either $n_3 = 0$ or $n_3 = 3$.

DEK then led a short discussion on how the problem would be different if only n^2 had to have non-decreasing digits. He admitted that in this case, he had no idea how to go about getting a provably complete list of solutions.

DEK said that unless something unexpectedly interesting popped up in solving this problem, we would try to wrap it up on Tuesday, and start on the next problem for Thursday. He wondered if it would be possible to have the computer generate the conjectures about the infinite classes, and then have the computer try to prove that every element of the class was a solution by using some technique (for example: a proof based on rep-units). One scheme for doing this might be to generate a conjecture and try to prove or disprove it whenever you repeat a digit. This would be more interesting than having us find the classes, prove their correctness using our intelligence, and then use the computer just to exhaustively disprove the existence of any other infinite classes. If we automated more of it, then we could easily have the computer solve the problem for other bases.

Notes for Tuesday, January 15

After getting no response to his inquiry about whether anyone had solved the main problem that we had left from the last class, DEK asked if anyone remembered what the main problem that was left was.

ARG said that last class we had conjectured that there were only a few infinite classes of solutions, but that we still needed to prove it.

DEK agreed that, yes, we had already proved the hardest infinite class of solutions ($n = 3^*6^*7$), and that we could easily prove the other three classes ($n = 16^*7$, $n = 3^*4$ and $n = 3^*5$). We could also find a bunch of solutions that didn't fit into an infinite class. (DEK admitted to having given $38^2 = 1444$ as the example solution in the original handout, because it wasn't in an infinite class.) But, the link that we were missing is to show that there are no other infinite classes of solutions.

What was needed is a way to have the program discard, for example, all the cases $n = 23^{(i)}4$ without exhaustively trying all values of i. DEK asked if anyone had any ideas of how to crack this problem.

RFC said that he had discovered a pumping lemma, which said that as you repeatedly pump up a digit in n, you will get to a point where the first bunch of digits of n^2 and the last bunch of digits remain the same, while some digits in the middle of n^2 get pumped up.

He gave DEK a write-up of what he had found, and DEK began figuring it out and explaining it to the class. RFC's work was done in terms of an arbitrary base that was one larger than a perfect square. In presenting it to the class, DEK concentrated on base 10.

RFC's theorem is that, for $3 \times a < 10$ and for $m, n \geq F(b,c)$:

$$(c \times 10^n + 3 \times a \times R_n + b)^2 \text{ is sorted} \iff (c \times 10^m + 3 \times a \times R_m + b)^2 \textbf{ is sorted.}$$

RFC made the disclaimer that this handled only the infinite classes with a single digit that gets repeated.

DEK thought that, nevertheless, it would be a big advance if we could show something like this. So, he started in on proving it:

Notational *diversion:* $Q_n = 3 \times R_n = \frac{10^n-1}{3} = 3^{(n)}$.

Now, we wish to prove

$$(c \times 10^n + a \times Q_n + b)^2 \text{ is sorted} \iff (c \times 10^m + a \times Q_m + b)^2 \textbf{ is sorted.}$$

DEK said that he made this replacement, because it would allow him to use the multiplicative formulas:

$$\begin{aligned} Q_m \times Q_n &= \frac{10^{m+n} - 10^m - 10^n + 1}{9} \\ &= R_{m+n} - R_m - R_n \end{aligned}$$

$$\begin{aligned} Q_n \times Q_n &= R_{2\times n} - 2 \times R_n \\ &= R_n \times 10^n - R_n \end{aligned}$$

$$\begin{aligned} Q_n \times 10^m &= \frac{10^n - 1}{3} \times 10^m \\ &= \frac{10^{m+n} - 10^m}{3} \\ &= Q_{m+n} - Q_m \end{aligned}$$

RFC suggested that leaving the formula as is (with only R_n's) might be simpler. Then, after squaring, collect terms with the same power of ten.

$$\begin{aligned} (c \times 10^n + a \times Q_n + b)^2 = {} & c^2 \times 10^{2\times n} + 2 \times a \times c \times 10^n \times Q_n \\ & + a^2 \times (R_n \times 10^n - R_n) + b^2 \\ & + 2 \times b \times c \times 10^n + 2 \times a \times b \times Q_n \\ = {} & 10^{2\times n} \times c^2 \\ & + 10^n \times \big(R_n \times (a^2 + 6 \times a \times c) + 2 \times b \times c\big) \\ & + \big(R_n \times (6 \times a \times b - a^2) + b^2\big) \end{aligned}$$

RFC indicated that he had a lemma that would help in showing the theorem from this point. The lemma says that $k \times R_n$ has some stuff at the begining, some stuff at the end, and a lot of repeated digits in the middle, for every integer k. More explicitly it says:

$$R_n \times k = l \times 10^n + d \times R_n - l$$

Here l and d are numbers that are dependent on k. They will be rewritten as $l(k)$ and $d(k)$. Note that $d(k)$ is a single digit. Continuing:

$$R_n \text{ x } k = l(k) \text{ x } 10^n + d(k) \text{ x } R_n - l(k)$$
$$(10^n - 1) \text{ x } k = (10^n - 1) \times (9 \times l(k) + d(k))$$
$$k = 9 \text{ x } I(k) + d(k)$$

Remembering that $d(k)$ is a single digit yields:

$$d(k) = 1 + (k-1) \bmod 9;$$
$$I(k) = \left\lfloor \frac{k-1}{9} \right\rfloor .$$

Hence, the lemma is true.

Now we can return to the theorem. First, we have an l_1 and d_1 based on $k_1 = a^2 + 6 \text{ x } a \text{ x } c$. We also have an l_2 and d_2 based on $k_2 = 6 \text{ x } a \text{ x } b - a^2$. Hence:

$$(c \times 10^n + a \text{ x } Q_n + b)^2 = 10^{2\times n} \text{ x } (c^2 + l_1)$$
$$+ 10^n \times (d_1 \times R_n + (2 \text{ x } b \text{ x } c - l_1 + l_2))$$
$$+ (d_2 \times R_n + (b^2 - l_2))$$

The resulting number will look like:

$$??\overbrace{d_1d_1\cdots d_1?\cdots??}^{n \text{ digits}}\overbrace{d_2d_2\cdots d_2?\cdots??}^{n \text{ digits}}$$

If n increases by 1, one more d_1 and one more d_2 will be inserted.

DEK noted that if $k_2 = 6 \text{ x } a \text{ x } b - a^2 \leq 0$ then we might need a slightly modified version of our lemma.

RWH pointed out that by our definition of a, this could only happen if $b = 0$.

So, we now had an intuition of why the theorem was true.

AG claimed that this result could be easily extended to the $n = 3^{(m)}6^{(p)}7$ *case,* also.

RMW said that you would need to do $m > p$ and $m < p$ as separate cases.

AG suggested that just a few cases wouldn't work.

DEK considered $(c \times 10^m + Q_m + Q_p + b)^2$, with $m > p$. He wondered if there might be a way to see if we have similar behavior for strings from the classes with two parameters.

RJC said that he had a related result for this case, but that it required both m and p to get large together.

MDD said that this was probably because n^2 has different digits in it when you have more 6's than 3's in n, than it has when you have more 3's than 6's.

DEK said that this could be seen from the formula for $n = 3^*6^*7$ in terms of R_i's that we derived last class. Since he didn't remember the formula, he re-derived it using Q_i's:

$$(Q_m + Q_p + 1)^2 = (R_{2\times m} - 2 \text{ x } R_m) + (R_{2\times p} - 2 \text{ x } R_p) + 1$$
$$+ 2 \times (R_{m+p} - R_m - R_p) + 6 \times R_m + 6 \times R_p$$
$$= R_{2\times m} + 2 \times R_{m+p} + R_{2\times p} + 2 \times R_m + 2 \times R_p + 1$$

He pointed out that while you knew the relative ordering of the rest of the R_i's, *you* had to know m and p before you knew where the $R_{2\times p}$ term fit in with the others. Hence, a pumping lemma type argument couldn't be done without multiple cases.

DEK wanted to find out what approaches other people had looked at. Since ARG had a printout in front of him, he was singled out next.

ARG said that he had tried to confirm that there weren't any other infinite classes of solutions. He tried this using a computer program that pruned off possibilities using the following "Suffix Test": If n is an m-digit nondecreasing number, and if the lowest m digits of n^2 are not nondecreasing, then adding any digits to the left of the digits in n will never give you a number whose square has nondecreasing digits.

After a little discussion it became clear that some teams had used a slightly stronger suffix test that also pruned if any of the lowest m digits of n^2 *was* a 0.

ARG noted that $\mathbf{n} = 3^{(m)}8$ yielded $n^2 = 1^{(m)}42^{(m-1)}44$ which didn't violate the suffix test. But, n = **23*8** violates the suffix test, n = **13*8** doesn't violate the **suffix** test (some people pointed out that it does violate the stronger version), and n = **113*8** violates the **suffix** test. ARG also mentioned that n = **9*** didn't violate the suffix test (again, people mentioned that this violates the stronger suffix test).

DEK started looking at what could be derived.

$$\begin{aligned}(10^n \times a + Q_n + 5)^2 &= a^2 \times 10^{2\times n} + (R_{2\times n} - 2 \times R_n) + 25 \\ &\quad + 2 \times a \times 10^n \times Q_n + 10 \times a \times 10^n + 10 \times Q_n \\ &= a^2 \times (R_{2\times n+1} - R_{2\times n}) + (R_{2\times n} - 2 \times R_n) + 25 \\ &\quad + 2 \times a \text{ x } (R_{2\times n} - R_n) + 10 \times a \times (R_{n+1} - R_n) + 30 \times R_n \\ &= a^2 \times R_{2\times n+1} + (-a^2 + 1 + 2 \times a) \text{ x } R_{2\times n} + 10 \text{ x } a \times R_{n+1} \\ &\quad + R_n \times (-2 - 12 \times a + 30) + 25\end{aligned}$$

for the case a = 1, we get:

$$(10^n \times 1 + Q_n + 5)^2 = R_{2\times n+1} + 2 \times R_{2\times n} + 10 \times R_{n+1} + 16\text{X } R_n + 25.$$

· So, now we want to know if this is sorted or not. We already know that: a number satisfies our non-decreasing digits criteria $\Longleftrightarrow$ it is the sum of up to 9 R_m's.

This brings up the subproblem: ***Given*** a number x, what ***is the*** smallest number of rep-units that **it takes to** [additively) represent ***the*** number?

This is similar to problems such as calculating how to make change using the fewest number of coins. DEK mentioned that for U.S. currency the greedy algorithm works. That is, you give the largest coin that you can, and then **recurse** on the remainder. He pointed out that it doesn't work for all sets of coins, and he asked if anyone could come up with a counter-example.

MDD mentioned that if we also had a 4 cent coin, then the greedy algorithm would give non-optimal change for 8 cents.

DEK said that another case was if we didn't have nickels, then the greedy algorithm would give non-optimal change for ***30*** cents. Getting back to the problem at hand, it turns out that using rep-units would give **a** coin system that works with the greedy algorithm.

To demonstrate, DEK asked AJU for a random number.

AJU suggested *2344.* The class thought that this was too easy, since it was nondecreasing. DEK said that the number wasn't random enough.

MDD suggested 383.

DEK showed that:

$$\begin{aligned} 383 &= 3 \times R_3 + 50 \\ &= 3 \times R_3 + 4 \times R_2 + 6 \times R_1 \end{aligned}$$

DEK said that there many ways to show that the greedy algorithm worked for a coinage system based on rep-units. (He noted that this probably implied that it was a useful fact.)

Assuming that $n < R_{m+1}$, then $n \leq R_{m+1} - 1 = 10 \times R_m$, and hence it follows that $n/R_m \leq 10$. Hence, the greedy algorithm will never use more than **10** of any one R_m in its solution. Also, once **10** of some R_m are used, then we are done giving change.

In this numbering system, 383 is represented as $(3,4,6)$. Counting in this numbering system goes as follows:

$(2,9,9,9,8)$
$(2,9,9,9,9)$
$(2,9,9,9,A)$
$(2,9,9,A,0)$
$(2,9,A,0,0)$
$(2,A,0,0,0)$
$(3,0,0,0,0)$
$(3,0,0,0,1)$

But, back to our claim that the greedy algorithm gives us the smallest number of coins. Let's look at our previous example of $(10^n + Q_n + 5)^2$:

$$\begin{aligned} (1,2,0,\dots,0,10,16,0,\dots,0,25) &= (1,2,0,\dots,0,11,6,0,\dots,0,24) \\ &= (1,2,0,\dots,1,1,6,0,\dots,0,23) \\ &= (1,2,0,\dots,1,1,6,0,\dots,2,1) \end{aligned}$$

The greedy algorithm gives **14** rep-units in the representation of this number. In general

we can replace: (a,, . . . , $a_{k+1}, a_k, a_{k-1}, \dots, a_0), a_k \geq 10$

with: $(a_n, \dots, a_{k+1} + 1, a_k - 10, a_{k-1}, \dots, a_0$ -1)

The only time there might be a problem is with $a_0 = 0$. But it turns out that even then, we will use no more coins in the new combination than in our old combination. Hence, we can see that by repeatedly applying this transform, we will eventually get to our canonical form. Since, we did this from an arbitrary starting combination, and we never increased the number of coins used, then our canonical form (the form produced by the greedy algorithm) is at least as good as any other combination of the same value.

ARG complained that this still didn't tell him what to do if something passed the suffix test. He still didn't see how he could know that adding more digits to the left of the number wouldn't make it work.

Thursday, January **17**

DEK explained that when you are solving a problem you should always keep looking for interesting side problems. He quoted Prof. Kahn of the Art Department (who will be talking to us later this quarter in connection with Problem **5),** as saying "I send you out to mine for silver, if you bring back gold, well that's fine too." He said that on- Thursday we would try to wrap up the discussion on the Problem **1.** For then, one thing that people might want to explore is how pumping might work for multiple parameter classes. We will also try to figure out what we learned from this problem.

Notes for Thursday, January **17**

AG asked if the class could have editing privileges to their comments in the class notes. ARG complained that he objected to **RWH's** use of the phrase "ARG complained" in the class notes. DEK, noting that he himself would probably disagree with some of the things that RWH attributed to him, assured them that changes could be made before the final tech report went out. They should send RWH any specific requests for changes.

DEK wanted to know what the final status was for each group.

KAM (reporting for her team whose other members were KES and ARG) said that ARG had written a program that used a generate-and-test methodology to find the solutions. He used his "suffix test" to prune the search space. The program generated a number of conjectures about infinite classes of solutions, and conjectures about infinite classes of non-solutions that had to be verified by hand.

She made the conjecture that in any base, if n = $s_1d^*s_2$ was a solution, then $d\times s_1d^*s_2$ probably had a lot of zeros in it. It is true for the base ten solutions. She sighted the example $n = 16^*7$: $6 \times n = 100^*2$. An implication of this, which she thought was connected to n being a solution, was that in doing long-hand multiplication, you would have very few (nonzero) terms that you would actually have to sum together.

DEK agreed that this was interesting, but thought that it would be hard to prove this in general.

AG (working with RMW, JFW, AJU and MJB) said that his group had done things similar to **KAM's** group. They had used their program to minimize the work they would have to do. After pruning using a suffix test, it generated conjectures that they had to try to prove/disprove by hand.

AGT (working alone) used pretty much the same idea. His program had generated a whole bunch of things that he was still going to have to prove by hand.

DEK mentioned that later in the class he would talk more about solving problems where the work is shared between humans and computers.

AAM (working with PCC) had his program do some proofs by induction. The program used an arithmetic wherein it could multiply a pattern with a single parameter times a number, such as multiplying $36^{(m)}7$ by 3. This allowed them to have their program prove/disprove whether or not any pattern with a single parameter was a solution pattern.

KES said that she had looked at the same idea. She, too, had been able to figure out how to have the computer multiply a single-parameter pattern by a number. But, she said that she couldn't figure out how to have the program multiply two patterns together.

Since, in the induction proof you needed to square the pattern, and no matter how you decomposed the squaring operation, you would have to multiply a pattern by a pattern she wondered if **AAM** had figured out how to multiply two patterns.

AAM said that he hadn't. Rather, he had gotten around this problem by decomposing the squaring operation so that the **multipication** of two patterns could be solved using the induction hypothesis.

DEK pointed out that the formulas that we had involving multiplying Q_m by Q_n gave us hope that there might be a way to do the multiplication of patterns directly.

AAM thought that maybe the two-parameter patterns could be multiplied using the induction trick.

DEK was skeptical, noting that there were many areas of Computer Science where it was possible to have automatic formulation of induction for one loop. But, the techniques all seem to break down when trying to do induction on two loops.

AAM still insisted that it could be done.

DEK opined that the complexity of doing it would be at least the square of the complexity for a single loop.

DEA (working with JIW) had looked at all suffixes of length at most ten. The suffixes that couldn't be thrown away fell into a few patterns. They proved/disproved the one-parameter ones by computer and the two-parameter ones by hand. One interesting two-parameter pattern that had to be disproved by hand, since it passed the suffix test, *was:* $n = 3^{(m)}56^{(p)}7$ when $m \leq$ p.

They had also looked at base **17,** but there were too many infinite classes of solutions. Bases 5 and 6 weren't as bad. They each had only a few solutions.

Clarifying a suggestion from last class, AG proposed a way to use our single pumping lemma to get double pumping. For example, when dealing with the pattern $n = 3^{(m)}6^{(p-1)}7$, the pattern should be looked at as $n = 3^{(m+p)} + b$ if $m > p$, and as $n = c \times 10^p + 6^p + 1$ if $m \leq p$.

MDD (working with RFC) reported that RFC had refined his pumping lemmas and had written a program to use them. MDD had written a program similar to AAM/PCC that can do the inductive proofs/disproofs on the single-parameter patterns. MDD said that the complexity of automating proofs for the two-parameter ones looks very bad. His system generates its hypothesis when the same digit gets repeated **4** times in a pattern.

RFC reported on the current state of his double pumping lemma. He could show (for both positive and negative *k)* that for $m_1, m_1 + k, m_2, m_2 + k \geq F_k(b, c)$:

$$(c \times 10^{m_1+m_1+k} + 3^{(m_1)}6^{(m_1+k)} + b)^2 \text{ is sorted} \Longleftrightarrow$$
$$(c \times 10^{m_2+m_2+k} + 3^{(m_2)}6^{(m_2+k)} + b)^2 \text{ is sorted.}$$

There was a discussion about how this related to showing the more general claim, that for $m_1, n_1, m_2, n_2 \geq F(b,c)$:

$$(c \times 10^{m_1+n_1} + 3^{(m_1)}6^{(n_1)} + b)^2 \text{ is sorted} \Longleftrightarrow$$
$$(c \times 10^{m_2+n_2} + 3^{(m_2)}6^{(n_2)} + b)^2 \text{ is sorted.}$$

The result was the following:

RFC's actual pumping lemma above gives you an infinite number of equivalence classes. If you plot n against m then each equivalence class consists of the points along a diagonal ray. Using the original (one-parameter) pumping lemma, you can get equivalence classes that correspond to horizontal (to the right) and vertical (up) rays. These horizontal and vertical rays collapse many of our diagonal classes together, so that we are left with a much smaller number of equivalence classes. Figuring out exactly how many equivalence classes are left would require looking more closely at the $F()$**'s.**

(After class, DEK came up with another way of seeing the difference between the two pumping lemmas. It looks like if "is sorted" is replaced by "passes the **suffix** test" in RFC's lemmas, the proofs will still go through. It was observed that: $3^{(m)}56^{(n)}7$ passes the suffix test $\Longleftrightarrow m \leq n$. Therefore the following is false:

$Q_{m+n} + 2 * 10^n + Q_n + 1$ either satisfies the suffix condition ***for all m,*** $n > N(2,1)$ **or** fails ***to*** satisfy ***it for all such m, n.***

But, it is true that:

For all ***fixed k,*** $Q_{m+n} + 2 \times 10^n + Q_n + 1$ either satisfies ***the*** suffix condition ***for*** all m, n ***with*** $m = n + k$ ***and*** $n > N_k(d,b)$ or fails ***to*** satisfy ***it for*** all ***such m, n.***

The suffix condition holds for $k < 0$ but not for $k > 0$. Hence the first statement above is false, while the second is true for any fixed ***k.)***

DEK now wanted to discuss what we had learned, besides the particular solution to this particular problem. He noted that we had learned a little about number theory, and a little about rep-units. We also had learned that sometimes solving a problem would involve an interaction between computer calculations and hand calculations. Sometimes, the program can only solve up to a point, and we have to finish off the rest by hand.

We saw that in having a computer prove things, we could find a small theory (a subset of mathematics) for the computer to operate within. DEK said that this was related to the idea of "inductive closure"; sometimes you can't prove a theorem directly by induction, but it turns out to be easy to prove by induction a more general theorem of which the first theorem is just a special case.

JIW, who had been looking through printouts for an example of a base that was not just one larger than a perfect square and yet still had an infinite class of solutions, reported that $n^2 = (4^{(m)}5)_9^2 = (2^{(m+1)}6^{(m)}7)_9$ was a solution to the problem in base 9.

DEK observed that this n was of the form $\left\lceil \frac{9^{m+1}}{2} \right\rceil$.

(After class, DEK noted that there was a solution of the form $n = ab^*cd^*e$, where a, c and e are all non-empty. This happens in base **100** with: a = 3, ***b*** **= 33, c = 36,** ***d*** **= 66** and e = **67.)**

Problem **1,** Review of Solutions

The consensus is that the set of solutions for base **10** is:

$\{1, 2, 3, 6, 12, 13, 15, 16, 38, 116, 117, 3^*4, 3^*5, 16^*7, 3^*6^*7\}$.

Here is what I think your programs do; if I'm wrong, then your reports were even worse than I thought.

Everyone had pretty much the same initial framework. They had a search tree where the integers associated with the children of a node were built by prefixing a digit to the integer associated with the node. A node was pruned if it failed some version of the suffix test. (Hence, the descendants of the node 4 would be 14, 24, 34, and 44. 14 and 24 would be pruned by the suffix test. 34 is a valid solution. 44 is not a solution, but it can't be pruned, since it passes the suffix test.) Other than this initial framework, most of the approaches did some things slightly differently from the others. For example, the search of this tree could never be stopped by exhaustion; there were many techniques for being able to stop.

ARG/KAM/KES: When a number had a digit repeated four times, their program assumed that repeating the digit more times would not change anything; hence, it didn't look at any numbers with a digit repeated more than four times. Thus, its search tree became finite. (It did, however, try putting smaller digits to the left of the repeated digit.)

For each number that it found, having a digit that was repeated four times and that passed the suffix test, the program printed a conjecture about the infinite class of numbers formed by "pumping" the quadruply-repeated digits. These conjectures were of the form: "all numbers of this form are solutions" or "no numbers of this form are solutions (even though they pass the suffix test)".

Singly infinite classes were proved and disproved by appeal to **RFC's** pumping lemmas. Doubly infinite classes were done by hand.

MJB/AG/AJU/JFW/RMW: Their approach was slightly different from that above. The search program also assumed infinite classes after three repetitions of a digit, but it didn't try putting any prefixes on a number once it had a thrice-repeated digit.

For each infinite class of n's, they (by hand) determined and inductively proved the properties of xn^2 (where xn means n prefixed by x); sometimes this broke up into multiple cases. Then, prefixes to these classes were examined by different runs of a separate program. This program had as input one of the infinite classes, and it also had knowledge of the form of xn^2. It checked which x's ($0 \leq x \leq 999$) would cause violations of the suffix test. For each x that didn't violate the suffix test, they (by hand) used the previously derived equations for xn^2 to determine whether the whole class was a solution, or whether the wholeclass failed as a solution.

DEA/JIW: Their program stopped expanding nodes when it reached depth 9 in the tree. It printed out a list of the unexpanded nodes that weren't solutions, but that still hadn't failed the suffix test. It also printed out a list of all of the solutions that it had encountered so far.

By hand, they identified the infinite classes that were manifest in their output. Then they disproved the bad classes and proved the good classes by induction. With the help of a separate program, they verified that adding prefixes to any of these infinite classes could not result in any new solutions.

PCC/AAM: Their program stopped expanding a node when the number had eight digits in it. For each infinite class of n's that hadn't been pruned, they determined by hand formulas for what the n^2's looked like. The singly infinite classes were inductively proved by a computer program that performed the squaring symbolically. The doubly infinite ones were proved by hand.

RFC: His program stopped repeatedly prefixing a digit when an appropriate pumping lemma could be applied. It was able to print out formulas for which infinite classes were solutions and disprove those which weren't.

MDD: If his program tried to repeat digits that weren't multiples of three, then it would let them repeat as often as they liked, correctly assuming that eventually they would violate the suffix condition. If it was about to repeat a digit that was a multiple of **3,** it switched to symbolic math. When squaring the number symbolically, there were times when the form of the output would have multiple cases. The program automatically figured out the n's and n^2's for all of the cases, and the conditions when each case applied. His program was thus able to throw out bad infinite classes not only by the suffix test, but also by symbolically showing that nothing in the class was a solution.

One conjecture that ***many*** people would have liked to have proved was that a digit, d, could be repeated infinitely in a base b solution only if for some carry digit, c, we have $c \times b = d \times d + c$. Or, equivalently, if d^2 is divisible by $b - I$. If we also conjecture that $d = b - 1$ is never infinitely repeatable, even though it satisfies the equation, then it follows that a base, b, has infinitely many solutions only if $b - 1$ is divisible by a perfect square (greater than **1).** We probably would also like to conjecture that the "only if" should be replaced by "if and only if". Hence, our final conjecture is:

A base, b, has classes of infinitely many solutions iff $b - 1$ is divisible by ***the square of a prime number.*** The ***infinitely repeatable digits will be those*** $0 < d < b - 1$ ***such that*** d^2 is divisible by $b - 1$.

Later Results: Half of proof of conjecture

After the class had finished working on Problem **1,** DEK continued to be interested in the conjecture. He and RWH looked at it further. The following proof resulted.

We have a conjecture that:

A ***base, b, has classes of infinitely many solutions iff*** $b - 1$ ***is divisible by the square of*** a prime number. The infinitely repeatable digits will be those $0 < d < b - 1$ such that d^2 is divisible by $b - 1$.

Here, we will have a constructive proof of the ***if*** portion.

Assume $0 < d < b - 1$ and that d^2 is divisible by $b - 1$. **Let** $c = \frac{d^2}{b-1}$.

$$
\begin{aligned}
x &= d^{(n)} \\
&= d \times R_n \\
&= d \times \left(\frac{b^n - 1}{b - 1}\right) \\
&= d \times \left(\frac{b^n - 1}{d^2/c}\right) \\
&= \frac{c}{d} \times (b^n - 1) \\
x + 1 &= \frac{c}{d} \times \left(b^n + \left(\frac{d}{c} - 1\right)\right)
\end{aligned}
$$

$$
\begin{aligned}
(x+1)^2 &= \frac{c^2}{d^2} \times \left(b^{2\times n} + 2 \times (t - 1) \times b^n + \left(\frac{d}{c} - 1\right)^2\right) \\
&= \frac{c^2}{d^2} \times \left((b^{2\times n} - 1) + 2 \times \left(\frac{d}{c} - 1\right) \times (b^n - 1) + \left(\frac{d^2}{c^2}\right)\right) \\
&= c \times R_{2\times n} + 2 \times (d - c) \times R_n + R_1
\end{aligned}
$$

Hence, $(x + 1)^2$ will be monotonic if $d - c \geq 0$ and $2 \times d - c + 1 < b$.

Since $c = \frac{d^2}{b-1}$ and $d < b - 1$, we have $\frac{d}{c} = \frac{b-1}{d} > 1$. Thus, $d - c > 0$.

Since $d \neq b - 1$, then we know that:

$$
\begin{aligned}
0 &< (d - (b-1))^2 \\
0 &< d^2 - 2 \times d \times (b - 1) + (b - 1) \times (b - 1) \\
2 \times d \times (b - 1) - d^2 &< (b - 1) \times (b - 1) \\
2 \times d - \frac{d^2}{(b-1)} &< (b - 1) \\
2 \times d - c + 1 &< b
\end{aligned}
$$

So, what we have exactly shown is that any digit of the form $0 < d < b - I$, such that d^2 is divisible by $b - 1$, spawns at least one infinite class of solutions in base b. It easily follows from this that base b has infinitely many solutions if $b - 1$ is divisible by the square of a prime number.

There are informal arguments that suggest that no other *d's* can work for b. A rigorous formal proof looks messy.

Thursday, January 17

Problem 2

Code-breaking: Decipher the two secret messages in Appendix A (see page 41).

Each message is 256 bytes long, and each byte of ciphertext is given as a decimal number between 0 and 255. One of the secret messages consists of English text; the other is a syntactically and semantically correct PASCAL program. [But you don't know in advance which message is which.)

The original plain text messages-i.e., the secrets-consisted of 256 bytes of ASCII code, using only the standard 94 visible characters (code numbers 33-126) together with the special codes null [O), line-feed (10), carriage-return (13), and space (32).

The enciphering was done as follows: Let the plaintext codes be $p_0 \ldots p_{255}$ and the ciphertext codes be $c_0 \ldots c_{255}$. There is an unknown function f such that $p_k + k = f(c_k + k)$ for $0 \le k \le 255$, where the addition is modulo 256. This function f is a permutation of the set {0, 1, . . . ,255) such that $f(x) \ne x$ and $f(f(x)) = x$ for all x. (Thus, in particular, it follows that $p_k \ne c_k$, and that $c_k + k = f(p_k + k)$. The coding scheme in UNIX's ‘crypt ’ function is a special case of this method.)

The unknown functions f are different in each message.

A third secret message will be unveiled in class on February 5; we will work as a group to decipher it, using any computer tools that the class has developed to decode the first two messages.

Notes for Thursday, January 17

DEK started out by comparing the decoding process to the game of Adventure. In both there are secrets to be found out by piecing together bits of evidence. He warned that the problem might turn out to be too easy, and that some people might have the answer by Tuesday. He asked that people who found the answer not tell others, so that the others could also experience the challenge and enjoyment of figuring out the answer.

He said that one of the coded messages was English text, and that the other was Pascal code. He said that he didn't remember which was which, and that even if he did, he wouldn't tell. DEK asked if anyone knew why he was telling us anything at all about the contents of the messages?

KES suggested that from this we would know something about the frequencies of letters in the original message and that this could help in the deciphering.

DEK pointed out that besides this, you also knew that the original pattern had to make sense. He said that lots of f()'s would satisfy the conditions determined by ASCII codes or frequencies, but there was only one decoding function that would yield a message that made sense.

AG insisted that DEK could only claim that there *was* a *high probability* that only one decoding would make sense. DEK concurred that there was a small probability that there could be two sensible decodings, though it would be quite amazing if it happened.

DEK pointed out that the better people's programs were, the less work the human would have to do. But a human was necessary to test for “making sense.”

He also gave some further information on the coding. First, he said that there were no comments in the Pascal code; it was hard enough to write an interesting program in

256 bytes without wasting some of them on comments. Second, he said that he had made no special attempts at being tricky. Third, the $f()$ had been generated using a random number generator that was based on the time that the encoding was done.

To give the students some of the motivation behind the problem, he mentioned that this code was related to the Enigma cipher used by the Germans during WWII. Alan Turing had spent a lot of time during WWII decoding messages in this code so that the British could keep tabs on their enemy. The reason that $f(f(x)) = x$ was so that the same hardware could be used to do the encoding and the decoding.

MDD said that he thought that he could tell which text was the Pascal, and which was the English. All (correct) Pascal programs start with the word 'program'. Hence, $p_1 = p_6 + 5$. (The letter 'r' is 5 more than 'm' in ASCII code.)

$$\begin{aligned} c_1 &= f(p_1 + 1) - 1 \\ &= f(p_6 + 6) - 6 + 5 \\ &= c_6 + 5 \end{aligned}$$

Since the second ciphertext has $c_1 = 198 = 193 + 5 = c_6 + 5$, he concluded that the second ciphertext was the Pascal code.

DEK pointed out that this was a special case of the "probable word" method. In this particular case you know the position of the word, but sometimes it is also useful to use the fact that a specific word will probably appear somewhere in the text, and then try to find out where it could possibly appear.

Notes for Tuesday, January 22

DEK mentioned once again that Problem 2 made it necessary to combine human intuition with computer calculation and bookkeeping. The human is essential in the loop because there are many possible decoding functions that don't violate the mathmatical restrictions that arise, but there is only one way to decode the messages so that they "make sense". It would be very hard to have a computer program exploit the fact that the decoded message makes sense; but a human is easily able to exploit this knowledge.

Considering that $f(f(x)) = x$ and $f(x) \neq x$, DEK wondered how many possible $f()$'s there were for the case of a ten-character alphabet/code.

KES suggested the Stirling Number: $\left[{10 \atop 5} \right]$.

DEK explained to the class that this was the number of permutations of **10** elements that have exactly 5 cycles. After some discussion, it was decided that this wasn't the answer, since it wouldn't just count the number of ways to have 5 cycles of length 2, but, rather, it would also count permutations with cycles of other lengths.

AG suggested that $\frac{10!}{2^5 \times 5!}$ was the answer.

RMW thought it might be $\binom{10}{2}$.

RFC disagreed, suggesting: $\binom{10}{2}\binom{8}{2}\binom{6}{2}\binom{4}{2}\binom{2}{2} = \frac{10!}{2^5} = \binom{10}{2,2,2,2,2}$.

AAM believed it was yet another formula: $\binom{10}{5} \times 5! = \frac{10!}{5!}$.

MDD pointed out that **RFC's** formula counts many possible $f()$'s multiple times. After some discussion, it was agreed that $\frac{10!}{2^5 \times 5!} = 945$ was the right one.

Tuesday, January 22

DEK wanted to generate a random **f()**. To do this, we were going to generate a random permutation of the digits from **0** to 9, and then pair up adjacent ones. These would give us the cycles for **f()**. In order to generate a random permutation we needed a random number generator. **DEK's** favorite was the digits of π. With help from JFW (who could easily have gone on listing digits of π for a lot longer), we had π = **3.141592653589793..** .

So we started with:

```
0 1 2 3 4 5 6 7 8 9
```

Since the first random number was 3, the first digit for our permutation was the **4th** **(3 + 1)** number: 3. DEK asked how we should update our list of unused digits.

JFW suggested that we take out the digit just used, and move the digits that were after it to the left.

RMW thought that it would be faster to swap the digit just used, with the last digit unused.

JFW conceded that this would be better. With that settled, we continued:

```
0 1 2 9 4 5 6 7 8   3
0 8 2 9 4 5 6 7   1 3
0 8 2 9 7 5 6   4 1 3
0 6 2 9 7 5   8 4 1 3
0 6 2 9 7   5 8 4 1 3
0 6 2 9   7 5 8 4 1 3
0 6 9   2 7 5 8 4 1 3
9 6   0 2 7 5 8 4 1 3
9   6 0 2 7 5 8 4 1 3
  9 6 0 2 7 5 8 4 1 3
```

A digit in the random sequence would have been ignored in accordance with the following rule: With ***m* unused digits, ignore the random number *r* if** $r \geq m \times \lfloor \frac{10}{m} \rfloor$. This allowed us to wrap the random numbers around without biasing the results in favor of the digits towards the left.

Our resulting **f()** was the permutation: **(9 6)(0 2)(7 5)(8 4)(1 3).**

Next, DEK wondered what the chance was that a message of length **10** would not use a particular one of the 2-cycles when it was encoded?

AGT said that this means that two digits can't be used. Hence, $\left(\frac{8}{10}\right)^{10}$ is the probability that a given **2-cycle** is not used.

With this answered, DEK wondered, what was the expected number of 2-cycles not used?

MDD cautioned that since the probabilities for each 2-cycle not being used were dependent, then we could not just add the probabilities in our calculations.

ARG noted that we can still add averages, even of dependent things.

DEK concurred, saying that this was one of the beauties of averages. It might also be one of the reasons that they are abused in so many meaningless ways, he noted.

After some discussion, MJB got the answer: $5 \times \left(\frac{8}{10}\right)^{10}$.

DEK now wanted to get back to our particular case: When we have **256** different possible values for each character.

ARG generalized MJB's equation to: 128 x $\left(\frac{254}{256}\right)^{256}$. But, he objected, this was done assuming that the text is random. In our case, the original text letters come from a small subset of $\{0, I, . . . , 255\}$, and they also appear with very widely varying probabilities.

DEK deferred the objection until later. For now, he wanted to try and figure out how much of f() we would know, if we were given both the plain text and the cipher text for one message. (Presumably, we might have other cipher texts that were encoded using f(), and we wanted to know f() so that we could decode them.) So, DEK wanted an approximation to $\left(\frac{254}{256}\right)^{256}$.

MJB suggested that it could be re-written as $\left(1-\frac{2}{256}\right)^{256}$.

DEK noted that if it was re-written as $(I- \frac{2}{n})^{n}$, then for large n, we could approximate this by $e^{-2} \approx \frac{1}{7.5}$. So the number that we want is roughly $\frac{128}{7.5} \approx 16$. This means that we won't use about **16** of the 2-cycles in f().

ARG reported that for the Pascal example, only **110** of the pairs were used.

DEK wondered if ARG had figured out how many pairs were used without decoding the whole message.

ARG said that he had decoded the whole Pascal message.

DEK returned to ARG's earlier objection that our preceding mathematical analysis was incorrect since our text was not random. He asked what the result would be if our f() was random but our text was not. The ensuing discussion revealed nothing important.

Notational diversion: $c'_k = (c_k + k) \bmod 256$.

DEK asked how we could figure out how many different values of f() were used. It turned out that we could do this by just counting how many different numbers appeared in the set of c'_k's. (Note that this is different from our earlier problem of counting how many 2-cycles were used; that number can't be determined for a particular cipher text until the decoding has been done.)

DEK re-iterated the point that, since many values of f() aren't used, including a number of the 2-cycles, methods that require the complete $f()$ to be found are doomed to failure.

DEK returned to the discovery mentioned in the last class by MDD, that if $p_i = p_{i+k} + k$, then they both use the same value of $f()$ when being encoded, hence we also know that $c_i = c_{i+k} + k$.

DEK pointed out that for Pascal, 'program' has this property between the first 'r' and the 'm'; in the language C, the keyword 'include' has this property for two pairs of letters: 'i' with 'd', and 'n' with 'l'.

The class pointed out that there were a bunch of other Pascal words that had this property, for example: 'output', 'string', and '`writeln`'.

AG reported that half the words in the on-line Unix dictionary had this property. Many people tried to explain this away by saying that any words ending in 'ing' or 'ed' would work.

DEK calculated that for two adjacent letters, α, and β, there was a $\frac{25}{26^2} \approx \frac{1}{26}$ chance that $\alpha = \beta + 1$. In words of length 6, there were $\binom{6}{2} = 15$ potential pairs of letters to have this interaction. Hence, using rough numbers, about $\frac{15}{26}$ of all words will have this property. This suggests that AG's empirical observation may not be as odd as it sounds.

DEK also pointed out that most on-line dictionaries don't have all the variant forms of words in them, so that the Unix dictionary probably didn't have all the words formed by just adding 'ing' or 'ed' to a verb.

Getting back to the problem of decoding, DEK wondered if, just using the mathematical properties that we knew about the encoding, we could answer questions of the form: Is it possible that p_{20} = 'a' = 97, given that $c_{20} = 215$?

First, we would know that it implied that $f(117) = 195$ as well as $f(195) = 117$.

ARG suggested that we could then go through the rest of the message and see if this caused any contradictions by implying that some plain text character had an ASCII code outside of our allowed range.

DEK calculated the probability that the 2-cycle is not used anywhere else: $\left(\frac{254}{256}\right)^{256} \approx e^{-2} \approx \frac{1}{7}$.

ARG also suggested that you could do the same thing by proposing a whole string of plain text and then seeing if it led to a contradiction.

At this point, DEK wondered whether there where ten tricks that needed to be used, or only one general trick.

MDD said that one other trick that he had used was that, because $c'_{138} = c'_{255} = 184$ and they were so far apart, there was only one possibility for $f(184)$.

The general formulation that we come up with was essentially equivalent to this:

We start out with a complete (undirected) graph of 256 nodes $n_0, n_1, \ldots, n_{255}$. The meaning of an edge between n_i and n_k is that we haven't ruled out the possibility that $f(i) = k$ (or equivalently that $f(k) = i$). Note that the graph being undirected corresponds to the fact that $f(f(x)) = x$. Also note that the non-existence of self loops in the complete graph corresponds to the fact that $f(x) \neq x$.

We can delete a lot of the edges by the following method. Since, we know the cipher text, any conclusion of the form $p_k \neq y$ allows us to delete the edge between nodes $y + k$ and $c_k + k$. When we have done this for all of the y's that we know do not appear anywhere in our plain text, then we will pretty much have exhausted all the direct implications about the function $f()$ that we can get purely mathematically.

The last thing to note about this graph is that there is a one to one correspondence between the perfect matchings of the graph and the possible $f()$'s. Hence, from this perspective, MDD was able to conclude the value of $f(184)$ because, after deleting edges from the graph by the method above, there was an edge that was in every possible perfect matching of the remaining graph. Also, once this table was constructed, checking (from the example further above) whether $p_{20} = 97$ is possible or leads to a contradiction can be done by just checking to see whether or not there is an edge between n_{117} and n_{195} in the graph.

DEK repeated for the third time that these mathematical calculations would not be sufficient by themselves to solve the problem. A system to use while decrypting would need to have interactive support that would allow a human (using the knowledge that the message "made sense") to make hypotheses. The system would have to have a way of updating its data structures so that fruitless hypotheses could be retracted.

DEK started the class by checking who had already broken the codes. ARG, RFC, and KAM had each broken at least the Pascal program.

DEK went back to his favorite theme about the importance of having a human in the loop for solving this type of problem. Only a human can recognize if a hypothesized solution is correct, so there is no way to totally automate the problem solving (given the present state of work in artificial intelligence).

He wanted to find out what progress people had made.

PCC (working with DEA and **AAM**) said that their program allowed you to conjecture that a string of characters started at some position, and then it saw if there were any contradictions.

AAM clarified that the program printed out a separate copy of the partial decoding for each position that the string could start at without causing a contradiction. In each of these separate partial decodings, new letters could be implied from the string that was conjectured. If the implied letters caused an unlikely string to appear in the partial decoding, then that particular partial decoding could be pruned. If too many possible partial decodings were left, then the original conjecture would have to be modified.

DEK commented that this is similar to how search is done in some databases. For example, for on-line card catalogs, you can say “give me all titles with the word ‘program’ in it”. It might come back and tell you that it had found **1023** such references. So, you then would have to give a more focused request.

AAM mentioned that you know beforehand that some keywords such as ‘program’, ‘begin’ and ‘end’ were in the Pascal program. But, you don’t know about keywords like ‘while’; these may or may not be present.

He continued that, if you only have a few partial decodings left, then you can choose one of them and do a depth first search.

ARG noted that two different paths are not necessarily disjoint. Maybe there are two different places that the string can fit in at the same time without causing any contradictions. Hence, there might be two paths that lead to the same state.

DEK, switching to questions of display, asked what people had used in their output routines to fill in unknown characters.

DEA said that they used underscores.

DEK wondered how they displayed the fact that there was a carriage return in the text.

AAM noted that they hadn’t gotten far enough in the decoding for that to happen. He said that they just printed ***chr*(13)** and so he supposed that it would just print a real carriage return.

In **KAM’s** program, she used two characters to display a letter. Normal letters were printed with a space as one of the letters. Special characters were printed differently:

‘\L’ – linefeed
‘\R’ – carriage return
‘\O’ – nulls
‘0 ‘– unknown

KES (working with RMW and JIW) said that in their system, the main goal was to minimize the time it took for a human to use the system to do the decoding. So the user interface – the display in particular – was crucial. She said that they viewed ease of use by the human as more important than having powerful commands.

DEK interjected that he possibly disagrees with that last claim.

KES continued that they are trying to get a terminal independent package that would let them use reverse video in their output.

We started looking at output from KAM's program, using a computer terminal that had been brought to class. She said that it finds words from a table of words, calculates the inter-letter distances, and sees what fits.

DEK noted that the output had a grid.

KAM said that this was so that a person who wanted to add a specific letter to a specific place in the text could calculate the position that she wanted to add it at.

DEK said that it was too bad that we didn't have a mouse. Traditional programming languages aren't equipped with the I/O capabilities to handle one. This was partly because I/O is the hardest thing to standardize due to the rapid pace of change in hardware.

DEK observed that KAM's method of using two printed letters to represent one letter was not too distracting. He noted that many programs have to deal with text as text. For these programs' input, every non-text character in its character set becomes very valuable.

KES mentioned that their system also printed two letters to display each letter. However, they typed the second letter below, instead of to the right of, the first letter. This way, the normal characters were printed right next to each other.

DEK returned to the grid of numbers in KAM's output.

KAM said that by adding the number from a letter's row with the number from its column, the user could get the exact position of a letter. After some discussion, she conceded that it might be better if she had the row numbers be multiples of ten.

AAM suggested that another way to select a specific character would be to move a cursor to it.

RMW pointed out that the program would need to be screen-oriented to do that.

AG suggested that this could be accomplished by using EMACS as the top level of your program. EMACS could then invoke various LISP functions that you had written.

DEK suggested that also showing the grid on the right and bottom of the text would be helpful.

DEK said that in his program for this problem, he had the program try to fit a word in the text, and then show him all of the possible places that the word could start. He would then be able to tell the program to assume that the word started at the $k^{\rm th}$ one of these possibilities, instead of referring to a particular grid position. Of course, if there were too many possibilities it became inconvenient to figure out what k was by counting.

DEK recounted that when he was using an early version of TEX to write a book, every two pages he would think of a way that TEX could be improved. The moral of the story was that for interactive systems, the designer and programmer should also use the system, otherwise the result might still have a lot of things that, from the user's perspective, could be done better.

DEK returned to looking at KAM's output. He noticed that there was a help message.

KAM said that she had put that in because she kept on forgetting her one letter commands.

MJB explained one of the features of their system. She had made a dictionary of Pascal keywords of length > 2. Then she ran a program to find which of these words had the property that $p_i = p_{i+k} + k$. The ones that she found for $k = 1$ were: ‘function’, ‘output’, ‘input’, ‘packed’, ‘procedure’, and ‘**const**’. She also found many others for different k’s.

DEK wanted to know how they used this information.

KAM said that they had a command, ‘g’, that told the program to search for the next place (starting from co) where any one of these words could fit.

DEK asked what they did if putting the word in led to implausible implications elsewhere in the message.

KAM said that their program could undo one step. This was done by keeping a copy of the state before the most recent change.

DEK asked how much information we would need to keep around if we wanted to be able to backtrack more than just one step.

AG suggested that it would probably be simplest to save the state every once in a while. And to also store the logical dependencies.

DEK asked AG to be more concrete about what he would save.

AG said that you would need to save your partial **f()**.

DEK agreed that you could then reconstruct the partially decoded cipher text from **f()**. DEK pointed out that in moving down the search tree, you were only filling in more slots of **f()**, and that in backtracking you would unfill these slots. He wondered whether this observation could lead to a good method of handling the backtracking.

RFC pointed out a possible problem. Suppose that at one point you knew that **f**(67) ∈ (122,135). Then you assumed that **f**(135) = 28. This implies that **f**(28) = 135 and hence that **f**(67) = 122. If you ever retract **f**(135) = 28 then you need to retract the consequence that **f**(67) = 122.

AG noted that he wasn’t really advocating the idea that he had mentioned earlier. He thought that it would be easier just to store things at periodic checkpoints.

DEK said that this type of thing might become an issue for Problem *3.* In backing up from level 6 to level 5, it might be better to store level 4, back up to it, and re-apply the transform to get from level 4 to level 5, rather than to store level 5 itself.

KES mentioned that the assumption you want to retract might not be the most recent one that you made.

KAM said that her program could handle this, since you can tell the program to change a specific letter. It will undo the previous map caused by the old letter. Then it will try to add the new letter. If there is a contradiction, then you won’t be able to add the new letter.

RFC mentioned that arbitrary **undo-ing** becomes a problem if you figure out values of **f()** by process of elimination.

AAM and RMW didn’t see why there was any complication.

Here is an example that shows why a truth maintenance system might be needed if you want to allow arbitrary undo-ing and if you use process of elimination in filling in the values of **f()**.

You know that: **f**(67) $\in \{122, 135\}$, **f**(72) $\in \{45, 77, 122\}$

Now assume: $f(28) = 135 \Rightarrow f(135) = 28 \Longrightarrow f(67) = 122 \Rightarrow f(122) = 67$.

Now assume: $f(77) = 55 \Rightarrow f(55) = 77 \Longrightarrow f(72) = 45 \Rightarrow f(45) = 72$.

Now retract: $f(28)$ and hence $f(135), f(67)$, and $f(122)$. Note that the reason that you assumed that $f(72) = 45$ is no longer valid, but that the system will now not let you assume that $f(28) = 45$.

One way around this might be to retract everything that was done after the thing that you are retracting. Then, re-assume all the assumptions that you still want to keep. But, this may prove to be too inefficient.

Notes for Tuesday, January 29

DEK started the class by describing new ground rules that we were going to adopt for the solutions to Problem 2. Each group will be given a cipher text (hint: the plaintext is English text) and 45 minutes to see how far they can get. Their efforts will be video-taped. The whole class will get to see an edited overview of all of the groups at work.

DEK pointed out that, since they would only have 50 minutes, the students should worry about increasing the speed of their programs. He surmised that with their current programs, the students could probably get answers to the ciphers in a couple of hours.

Pointing out that the groups were now competitors, DEK hoped that we could still share ideas with each other. The most important point was to find the current bottle-neck in the deciphering process.

AJU said that one problem was in guessing what words were likely to appear.

DEK pointed out that this meant that S-person teams might thus have an advantage over 2-person teams (having native speakers of English would also be helpful). He wondered what sort of deciphering aids people might want.

KES suggested that it would be nice if there was a dictionary that was indexed according to letters in the middle of the words.

DEK pointed out that there are some such dictionaries that are intended for crossword puzzle solvers. You could look up, for instance, all of the five letter words of the form ?u??e’. He pointed out that a lot of the words in such dictionaries were ones that he had never heard of.

JFW said that we would like to know what words occured with high frequencies.

DEK mentioned that in Volume 3 of “The Art of Computer Programming”, he had shown a data base structure that was based on the 31 most common words in English. He said that ‘the' was the most common.

AAM said that a problem with this, especially if the word is short, is that you can then have too many possible paths to choose between.

DEK asked the class to try to think of ways to speed-up the process of finding the answer. He said that they should try to use their experience from solving the two sample cipher texts to find general strategies of attack.

AAM pointed out that one can use partial decodings to get a suggestion of what to try next.

DEK mentioned that retrieval of facts from partial information is one thing that is still much easier for humans to do than for computers. One would need to have an on-line dictionary of the type mentioned by KES to allow computers to do this well.

AAM said that even this wouldn't help us in deciphering a few good starting points.

DEK agreed that it seems that there is a problem in getting a "critical mass" of information in solving the problem. He asked for any suggestions for how to get off to a good start.

AG suggested that using the cases where $c_i = c_{i+k} + k$ allows you to get starting points that have a higher probability of being right.

There was also a suggestion that it might be fruitful to look for endings such as 'ing', 'ed', and 'tion'.

DEK suggested that we try now to work on a coding of some text from "Wuthering Heights". It was decided that we try to use KAM's current program.

While the modem and cameras were being set up, DEK mentioned that it might be interesting to try to figure out a coding scheme that would defeat all of the tricks that we were using, so that encrypted files would be more secure.

KAM's program had a command that would first find a place in the cipher text such that $c_i = c_{i+k} + k$, and then conjecture a word that might fit there.

The program made its first conjecture. ARG noted that the conjecture was very improbable since it implied that there was a **'$'** in the text.

DEK pointed out that it was very useful to rule out things that are improbable, instead of just ruling out things that are impossible.

KAM mentioned that, besides adding more words to her list, she was thinking of adding word fragments to it.

DEK noted that in this case the **'$'** was a give-away since we knew that it shouldn't occur in Wuthering Heights. But what would we do about things like **'2's** which may very well occur?

Looking at the display produced by KAM's program, RFC mentioned that it would be useful if the consequences of the most recent assumption were highlighted for easy identification. This would simplify the checking for improbable consequences.

KAM tried to see if there were any nulls at the end of the text. There weren't. From the partial text that we had so far, she also tried adding spaces around words and after commas.

DEK said that looking for the nulls was perfectly legal. He asked if there was anything besides a space that might be after a comma. Some people suggested carriage-returns or quotation marks.

AG said that it would be nice if you had a way to know which values had the most consequences so that you could know which values to try to guess first.

AAM said that this whole problem could be viewed as a constraint satisfaction problem, and that there were general purpose constraint satisfaction programs (such as one called ALICE) that could be used here.

DEK said that there was nothing wrong with using other people's code in solving Problem 2, but that he thought that a special purpose program would be more useful for this problem than any general purpose one.

Returning to KAM's attempt at decoding, it was noted that the current assumptions implied that the string 'qe' appeared in the text. This was considered unlikely.

AG remarked that this reminded him of a Scientific American article he had read on random text generation. It was an extension of the idea of having a monkey type randomly at a typewriter. In this case, there was an N^{th} order monkey whose random letter distribution could depend on the previous $N - 1$ letters.

DEK returned to the text at hand. He looked through the trace of when he broke this particular code. He mentioned that in the search for probable words, all of a sudden, the first real clue popped up. This happened one time when he had the program try to fit ‘, and’ into the text. He had a string that looked like: ‘, and ?a?he? ’. Since it seemed like this was a descriptive passage, he guessed that the word was ‘rather’. We then had `‘, and rather ?or????'.` From the writing style of the passage, he was able to guess that an occurrence of ‘,?y?t??’ was ‘, yet’.

In ending the class, DEK mentioned that we had noticed that there were at least two distinct strategies that came into play at different points in the code-breaking. First there was finding an initial wedge, and then there was an “endgame”. There were thus many different operations that one might like to perform: finding where a word could fit, trying out words, retracting them, determining how probable various possibilities were, etc...

Notes for Thursday, February 7

On Monday, the six teams each had 45 minutes to use their code breaking programs in trying to break a coded message. (The same coded message was used in each.) Today, the class started off by looking at edited video-tapes of how each of the groups had fared.

The first team was MJB/KAM. They were unable to solve it, although they got a good start. One command that they realized that they would have liked to have had available was a command that gives all of the possible letters that can fit at a specified place in the text; they added the command after the Monday code-breaking session.

Next up were AGT/AJU/JFW. They didn’t finish in the allotted time, either. DEK mentioned that one interesting thing about this group was that their methodology of code-breaking was able to count out incorrect assumptions very early. They had a lot of unfilled parts of the text, but what they had filled in was amazingly correct.

It was **RFC/ARG’s** turn to try their hand at breaking the code. They were able to break it in a little under 37 minutes. DEK noted that the advantages of cursor control within a screen-oriented system became apparent. Also, RFC/ARG were able to look at the different possibilities that were still available. They could also see the interdependencies of things, so that they could see when retracting an assumption would cause them to lose a lot of good (plausible) text.

DEA/PCC/AAM were close, but no cigar. DEK said that having a team of people who were all of foriegn origin didn’t seem to be a handicap, **nohow.** Contrariwise, they seemed to have a better familiarity with English Literature. Their main problem appeared to be that their user interface required a lot of typing to enter commands.

MDD/AG were able to break the code in a little over 38 minutes. DEK liked the way that the user interface allowed you to just type over the text that was already there. The program would beep, if something that you were trying to type caused a contradiction.

A lot of use was also made of highlighting. In posing this problem to us, DEK said that he hadn't foreseen anyone using this high a level of interaction. He noted that this group wasn't the only one for which the word "nohow" was the last word decoded; it seemed to be the hardest word in the text.

Finally, RMW/KES/JIW waltzed in and polished off the code in 22 minutes flat. DEK thought that the ability of this program to display the possible letters for more than one position at the same time was very useful.

DEK noted that 3 teams had broken the code and that 3 teams hadn't. Had anyone noticed any factor that seperated the two groups?

RMW pointed out that the teams that broke the code had screen oriented programs, while the other teams didn't.

DEK mentioned that this was further evidence that screen oriented programs really are easier to use. Unfortunately, the programs aren't as portable that way. It takes time before new ideas are understood well enough to be able to standardize them.

AAM noted that breaking the coded Pascal program was the easiest.

DEK said that in writing it he wanted something that was interesting, but still took less than 256 bytes. DEK asked why the program was interesting.

RFC said that he had run the program and it had printed itself out.

DEK mentioned that throughout the history of computing there has always been an interest in self-reproducing programs. He thought that this was the shortest such Pascal program that could be run on SAIL's Pascal. He could shorten it a bit for Pascals that didn't have a limit on line length.

AAM said that he had suspected from the beginning that the program was a self-reproducing program.

DEK remembered that Bob Floyd devised an Algol W program that fit on an 80 column card and that caused itself to be punched out on another card.

RFC noted that the Pascal program used ASCII codes in some places. Filling these in required one to figure out what the purpose of the program was.

DEK mentioned that the other text was from a paper on how to break the Unix crypt command. The paper also gave modifications of the code and how to break them. [Reeds and Weinberger, ***AT&T Bell Laboratories Technical Journal*** **63 (1984), 1673–1683.**]

He recalled a story about when some coding program had first become available at Stanford. John McCarthy was the first to use it to encipher several of his files. Legend has it that he forgot what keys he had used, and that no one has been able to decode those files since.

AG brought up a point that he had mentioned in an earlier class. He said that there were two possible decodings of one of the texts; the text could either end with a '!' or with a '.'.

DEK said that very little technical literature uses exclamation points. He used to use a lot of them himself, until Bob Floyd dissuaded him of the practice.

He asked for ideas on how to improve this type of ciphering.

RFC suggested that we encipher the text twice with two different keys.

RMW wanted to remove the restriction on having cycles of length 2. (DEK mentioned that this was essentially equivalent to RFC's suggestion, since he thought any permutation is a product of two permutations that have only 2-cycles.)

MDD pointed out that compressing our data before coding it would make things much harder, because **ASCII** code would be obliterated.

AAM suggested that shorter messages would be harder to decode.

RMW thought that the way the coding function depended on position should be changed.

Problem 2, Review of Solutions

There were a lot of possible ideas to use in solving Problem 2. Clearly then, no one could be expected to use them all. Especially since powerful systems could be built using only a subset of these ideas. Naturally, the different groups chose different subsets. Here is a survey of some of the ideas:

Data structures

Maintain table of possibilities. Most of the groups used a 256 by 256 array to keep track of the possible $(x,$ f(z)) pairs. Some groups used it to repesent which $f()$'s were ***initially disallowed*** by illegal character codes. Other groups used the array to keep track of values that are ***currently disallowed*** because of their current assumptions. The advantage to keeping track of the implications of your current assumptions, is that it enables you to make some of the automated inferences mentioned below. One way to look at the 256 by 256 array is as an adjacency matrix of a 256 node (undirected) graph; noting that there is a one-to-one correspondence between perfect matchings of the graph and allowable $f()$'s helps motivate a few of the automated inference schemes mentioned below.

Automated inference

Disallow illegal character codes. All of the groups had their programs disallow $f()$'s that would have caused illegal **ASCII** codes (those corresponding to 'unprintable' characters) to appear in the plain text. If you were keeping the 256 by ***256*** array, the information on illegal characters would let you rule out large parts of the array.

Enforce **f** $()$ ***'s basic restrictions.*** That is, enforce that $f(x) \neq x$ and $f(f(x)) = x$. Enforcing $f(f(x)) = x$ has two parts. First, this equation is why the 256 node graph is undirected and, hence, it implies that the 256 by 256 matrix should always be symmetric. Second, it implies that there is a unique a such that $f(a) = b$. Thus, when assuming a relationship of the form $f(a) = b$ (which is the same as assuming that $f(b) = a$ or that the edge (a, b) is in the perfect matching), you can then delete edges by the rule: if (a, b) is in the perfect matching, then no other edge that uses either a or b can be in the matching. All such edges can be deleted from the array. Some people didn't delete them, but rather only forbade their later use.

Enforce properties of perfect matchings – nodes of degree 0; ***degree 1.*** Actually deleting edges that violate $f()$'s basic restrictions allows you to use another rule: If node a has only one edge, (a, b), coming out of it, then it is clear (if your current assumptions are true) that (a, b) must be in any allowable perfect matching; hence you can automatically assume that $f(a) = b$ and $f(b) = a$. Lastly, if a node has no edges coming out of it, then you have made a faulty assumption. Three groups used this type of automated inference.

Enforce properties of perfect matchings - nodes of degree 2. More use can be made of the properties of perfect matchings. The next most obvious rule that can be thought of by looking at the problem from the perfect matching angle is: if a node, a, has only two edges coming out of it, (a, *b)* and (a, c), then, clearly, one of these edges must be in the matching; hence, the edge *(b,* c) can not be. So, *(b,* c) can be deleted from the graph. No one made use of this.

Enforce properties of perfect matchings - more involved properties. Other rules can be thought of, but they start to get complicated/expensive to implement. For example, if adding edge (a, *b)* to the perfect matching (and hence deleting all other edges to ***a*** and *b*) breaks the graph into components, any of which have an odd number of nodes, then *(a, b*) can't be in any perfect matching, and it can be deleted from the graph. Some of the rules mentioned above are just disguised forms of special cases of this rule.

Truth maintenance. Many of these automated deduction schemes that we are talking about find out the consequences of ***assumptions.*** Thus, a program using them should have a way of undoing the consequences of an assumption that is later retracted. This can be done by keeping data structures that allow ***real truth maintenance.*** Alternately, you can do the ***truth maintenance by recomputation.*** Lastly, you can do ***no truth maintenance;*** that is, you let unsupported consequences stay around until they are individually retracted.

Auto-filter out unlikely characters. In breaking the codes, many people would try strings out and see that they caused unlikely letters to appear, then they would retract the string. By automatically ruling out the unlikely characters, things can be sped up; that is, allow the user to tell the program: assume that there are no **'#'s** in the message. This can help in a few ways: (1) it saves the human the time of scanning the resulting text for the character; (2) 'search for string' can automatically skip more possibilities; (3) the ruling out of another character can be used by the automated deduction routines to further restrict things. The only ones to apply a variant of this idea were AAM/DEA/PCC; their feature allows you optionally to assume that a fixed set of unlikely characters isn't used in the text. They use the idea only to gain benefits (1) and (2).

Auto-filter out unlikely character pairs. Here, the idea is to have the program look at the remaining possible characters for two adjacent positions and rule out some of the characters in one because of the absence of certain characters in the other. The information from this can propagate in two ways: (1) deleting a character is equivalent to deleting an edge in the 256 by 256 array - this can allow more deductions to be made based on the properties of perfect matchings; (2) deleting a character from position x based on the absence of certain characters at position x + 1 might allow you then to use this same rule to delete possible characters from position x - 1 (or even from position x + 1) [see Waltz filtering/Relaxation methods in the AI literature]. KES/RMW/JIW use this idea in a restricted way. They allowed the user to say "fill in <CR><LF> pairs". This is equivalent to saying that: <CR> is not followed by anything other than an <LF>, and <LF> is not preceded by anything other than <CR>. Because of the limited nature of the prohibitions, no propagation of type (2) is possible, but they do do propagation of type (1).

Case restrictions. There were times when the code-breakers knew that a character was upper case, or that it was lower case, or that it was some sort of punctuation or separating character. Thus, it might have useful to have a command such as: assume that the character at position x is a lower case letter. This would have ruled out possibilities which, as above, could lead to the propagation of constraints, yielding useful information elsewhere in the cipher. No one did this, though. AAM/DEA/PCC did allow some of these types of assumptions to be used, but just for the purpose of printing out a subset of the possible characters for a position.

Probabilistic reasoning. **A** few groups mentioned that, if they had more time, they might have liked to play around with the idea of using automated reasoning using probabilities (or Mycin-like confidence factors).

Displaying the current status

Screen-oriented I/O. The programs which could have worked just as well on teletypes weren't as easy to use as the ones that took advantage of the cursor addressing and other screen functions. The use of ***highlighting by reverse-video*** also made things much easier on the user. Communication via the visual cortex was much more direct.

Show currently possible. Either show what ***characters for a position*** or ***values for an*** $f(x)$ are currently possible. Some people showed ***letters for several positions simultaneously,*** which seemed to be very useful. MDD/AG also showed which possibilities had been ruled out only because of your previous assumptions.

Show most constrained. Which ***character*** or ***function values*** have the least possibilities left? Which have less than ***k*** possibilities left?

Show positions with same function value. There are a few reasons for wanting to do this. It will tell you where you might need **a** word that satisfies the "program" property. It will, also, give you an idea of how useful it will be to conclude the value of this function.

Show conflict reason. If the program is saying that you can't do something that you want to do, then you naturally would like to know why you can't do it. Perhaps you can then retract something, which will allow you to do what you wanted; or else you might decide that what you had wanted to do was wrong.

Print verbatim decipherment. This prints the text decoded so far in the normal form it would appear; that is, with line breaks controlled by the <CR><LF>s in the text. This eases the perception of the structure of the text.

Print current map of f **().** This function, which deals explicitly with the values of f**(),** didn't turn out to be very useful.

Show stack. AGT/AJU/JFW had a command which allowed them to look at a stack of the previous assumpions; these assumptions could then be retracted, if they desired.

Other major commands

Lookup strings. KAM/MJB do it ***from a list of words with the "program" property.*** AAM/DEA/PCC do it ***from a list of the 20 most common words.*** **MDD/AG** do it ***from the*** UNIX ***dictionary via*** grep.

Search for possible positions for string. Given a string, find the next place that the string could occur in the text, without causing a contradiction. MDD/ARG allowed the strings to have ***wildcards***; each wildcard could match an arbitrary (or unspecified) character.

Assume values for f **().** The values could be specified by ***entering the numbers*** or ***adding a string at a position.***

Retract character; retract assumption. One interesting thing to note, is that those programs that enforced properties of perfect matchings were not always able to retract a character. This would happen when **a** character was known indirectly as a result of process of elimination. Such a character is forcibly implied by the ***combination*** of several assumptions (and some are even forced by the initial disallowed information). The program has no way of knowing which ***one*** of these assumptions should be retracted unless you explicitly retract that assumption.

Save/Restore State. This allows the user to save a state. Later, the state can be returned to without doing a lot of tedious retractions.

Program Features

AGT/AJU/JFW: Maintain table of possibilities – initially disallowed and currently disallowed. Disallow illegal character codes. Enforce $f()$'s basic restrictions. Show currently possible – values for an $f(x)$. Show most constrained – function values. Show stack. Search for possible positions for string. Assume values for $f()$ – adding a string at a position; entering the numbers. Retract assumption.

KAM/MJB: Maintain table of possibilities – currently disallowed. Disallow illegal character codes. Enforce $f()$'s basic restrictions. Show currently possible – characters for a position. Print verbatim decipherment. Print current map of $f()$. Lookup strings – from a list of words with the "program" property. Search for possible positions for string. Assume values for $f()$ – adding a string at a position. Retract character.

AAM/DEA/PCC: Disallow illegal character codes. Enforce $f()$'s basic restrictions. Auto-filter-out – unlikely characters. Show currently possible – characters for several positions simultaneously; case restrictions. Lookup words – from a list of the 20 most common words. Search for possible positions for string. Assume values for $f()$ – adding a string at a position. Retract character.

MDD/AG: Maintain table of possibilities – currently disallowed. Disallow illegal character codes. Enforce $f()$'s basic restrictions. Enforce properties of perfect matchings – nodes of degree 0; degree 1. Truth maintenance – no truth maintenance. Screen-oriented I/O – highlighting by reverse-video. Show currently possible – characters for a position; characters ruled out only because of current assumptions. Show most constrained – character. Show conflict reason. Lookup words – from UNIX dictionary via grep. Search for possible positions for string. Assume values for $f()$ – adding a character at a position. Retract character. Save/Restore state.

RFC/ARG: Maintain table of possibilities – currently disallowed. Disallow illegal character codes. Enforce $f()$'s basic restrictions. Enforce properties of perfect matchings – nodes of degree 0; degree 1. Truth maintenance – truth maintenance by recomputation. Screen-oriented I/O. Show currently possible – characters for a position. Show most constrained – character. Show positions with the same function value. Print verbatim decipherment. Search for possible positions for string – wildcards allowed. Assume values for $f()$ – adding a character at a position. Retract character. Save/Restore state.

KES/RMW/JIW: Maintain table of possibilities – currently disallowed. Disallow illegal character codes. Enforce $f()$'s basic restrictions. Enforce properties of perfect matchings – nodes of degree 0; degree 1. Truth maintenance – truth maintenance by recomputation. Auto-filter-out – unlikely pairs (<CR> <LF >) . Screen-oriented I/O – terminal independant package; highlighting by reverse-video. Show currently possible – characters for several positions simultaneously. Show most constrained – character. Show positions with the same function value. Show conflict reason. Search for possible positions for string. Assume values for f()– adding a character at a position. Retract character.

In summary, there seem to be two main factors that influenced how well programs did: **(1)** amount of automated inference, and (2) ease of perception and use. Among the three programs that weren't able to solve the deciphering problem in **50** minutes, the **AAM/DEA/PCC** program was able to get much further than the other two; this seems to be due to that program's feature of allowing unlikely characters to be filtered out, and due to its ability to display the currently possible characters for several positions simultaneously. The major differences between the three programs that were able to solve the cipher in time, and the three that weren't, was that the three that were able to solve it had both: screen-oriented input/output, and did automatic deductions that made use of the properties of perfect matchings. The main feature that probably allowed **KES/RMW/JIW** to outperform the two other programs that solved the problem, was that they were able to display the currently possible characters for several positions simultaneously.

Appendix A. The secret messages.

The ciphertexts.

Ciphertext **#1:**

```
027 111 238 000 034 107 159 105 112 001 110 148 097 194 193 241
140 219 151 089 160 224 202 136 212 166 046 181 017 162 248 190
025 012 244 064 191 053 174 182 018 013 079 049 255 146 053 227
006 250 249 067 074 196 041 220 101 100 129 190 216 188 051 239
062 001 245 255 234 233 122 043 172 192 191 160 019 156 053 105
110 105 099 194 081 089 060 028 177 067 025 099 115 108 074 193
106 084 083 113 136 042 146 175 086 100 071 131 016 139 129 128
050 148 128 192 146 111 218 188 210 189 201 065 253 090 006 217
083 032 122 160 159 239 196 253 128 045 229 088 196 174 191 039
056 068 140 186 089 211 137 145 012 060 038 054 002 001 005 035
029 199 009 008 094 005 004 014 003 116 023 084 254 218 001 205
034 242 051 138 112 080 073 129 064 212 043 170 084 216 233 067
253 214 250 214 221 100 095 186 109 184 039 012 002 037 148 215
158 010 020 228 010 115 100 071 130 097 102 113 017 137 217 011
122 001 125 126 130 240 122 083 129 146 152 107 167 155 172 046
219 200 011 180 081 124 013 121 141 021 085 218 241 090 211 032
```

Ciphertext **#2:**

```
102 198 003 019 224 107 193 019 000 173 211 143 142 077 055 054
157 048 104 176 175 123 164 174 197 020 050 018 206 166 150 113
136 043 141 137 064 138 250 043 068 142 185 109 194 127 083 105
203 229 152 164 120 176 121 230 185 172 103 211 051 188 047 153
015 184 058 052 093 144 090 149 066 006 146 254 179 078 150 000
175 049 035 144 143 149 109 051 242 250 105 113 008 072 092 212
089 110 088 009 210 067 163 143 240 040 175 014 236 020 106 232
054 168 073 168 192 023 149 015 165 012 067 057 124 013 008 182
161 254 167 188 221 156 022 138 193 085 046 045 208 181 180 112
195 142 152 151 102 253 155 071 214 177 212 069 142 157 092 165
170 003 164 042 000 147 227 192 126 049 249 201 154 071 245 159
121 090 070 147 073 239 255 050 069 121 093 187 200 206 018 247
196 008 161 111 119 119 220 230 238 217 147 250 056 114 232 246
050 152 008 007 012 085 215 030 232 081 053 242 149 181 110 221
179 177 133 198 096 239 024 004 144 206 018 000 008 227 005 191
199 054 100 161 047 204 016 067 074 112 013 104 096 227 074 185
```

Appendix A

Ciphertext from Wuthering Heights:

```
1‘79 139 222 233 177 211 141 131 059 213 186 231 225 184 228 000
192 032 004 252 157 125 246 047 067 044 242 037 021 064 253 233
006 233 132 033 040 001 097 210 153 209 244 240 231 000 018 043
155 081 020 099 016 108 242 184 089 108 041 253 105 089 130 168
097 164 095 107 100 145 159 135 023 144 010 215 097 037 177 255
160 040 099 001 156 042 076 086 004 239 025 237 248 239 122 245
033 024 019 027 127 100 150 188 178 227 046 252 229 128 150 193
252 053 113 060 245 009 139 023 241 049 242 161 190 239 230 088
208 056 191 199 195 111 187 167 085 052 183 195 010 161 248 193
129 117 190 151 242 180 169 183 104 166 225 242 236 151 201 247
071 131 014 224 177 068 153 247 031 149 031 239 228 032 240 116
123 210 021 028 119 024 252 058 015 114 250 253 057 019 064 035
016 210 218 077 118 208 242 059 130 146 006 041 201 173 020 051
064 087 062 136 192 124 181 142 079 222 228 045 019 059 177 000
171 170 082 089 072 210 002 162 231 235 020 161 213 202 254 210
071 085 116 031 101 040 028 255 230 027 193 229 065 115 156 030
```

Ciphertext for final contest:

```
204 118 254 095 012 015 057 164 100 046 168 029 043 147 189 156
210 192 049 226 130 027 149 032 139 042 182 021 180 210 183 016
122 019 018 224 220 001 150 104 008 214 097 213 011 107 126 246
245 005 095 094 227 029 163 159 108 155 209 203 202 159 103 220
140 009 182 022 203 167 212 161 138 246 140 061 156 249 144 205
161 132 021 142 035 193 160 032 233 147 220 039 038 219 037 182
087 033 015 253 153 029 153 036 050 199 101 128 127 013 143 196
238 230 229 057 103 068 082 230 229 229 037 036 254 008 204 096
052 249 088 230 020 042 127 030 010 059 077 243 006 161 090 095
222 105 224 184 067 153 137 005 169 153 248 188 123 008 244 127
077 207 250 110 070 207 192 074 086 203 020 196 180 141 015 025
211 181 154 181 237 134 235 171 006 184 004 127 013 118 124 215
214 185 160 218 129 055 090 031 205 210 007 013 218 161 225 002
072 246 011 200 250 089 212 035 221 199 244 192 061 016 052 179
070 056 113 067 016 080 056 141 144 109 223 141 100 121 134 053
035 116 050 201 127 220 138 092 178 108 194 249 021 087 033 147
```

The plaintexts (don't peek).

Plaintext #1:

```
File encryption is roughly equivalent in protection
 to putting the contents<CR><LF>
of your file in a safe, a locked desk, or an unlocked
 desk.<CR><LF>
The technical contribution of this paper is that UNIX's
 ‘crypt’<CR><LF>
is rather more like the last than the first.<CR><LF>
<NULL><NULL><NULL><NULL>
```

Plaintext #2:

```
program M(output);const a=';
begin writeln(b,chr(39),a,chr(39),chr(59));
writeln(chr(98),chr(61),chr(39),b,chr(39) ,a>;end.';<CR><LF>
b='program M(output);const a=';
begin writeln(b,chr(39),a,chr(39),chr(59));
writeln(chr(98),chr(61),chr(39),b,chr(39),a);end.<CR><LF>
<NULL><NULL><NULL><NULL><NULL><NULL><NULL>
```

Plaintext from Wuthering Heights:

```
<CR><LF>
many a country squire; rather slovenly, perhaps, yet<CR><LF>
not looking amiss with his negligence, because he has<CR><LF>
an erect and handsome figure, and rather morose.<CR><LF>
Possibly, some people might suspect him of a degree<CR><LF>
of underbred pride; I have a sympathetic c
```

Plaintext for final contest:

```
“I know what you're thinking about," said Tweedledum;
 “but it isn't so,<CR><LF>nohow.’’<CR><LF><CR><LF>
“Contrariwise," continued Tweedledee, “if it was so,
 it might be;<CR><LF>
and if it were so, it would be; but it isn't, so it ain't.
 That's logic.’’<CR><LF><CR><LF>
-- LEWIS CARROLL<CR><LF><NULL><NULL>
```

Tuesday, February 5

Problem 3

Hardware fault detection: Appendix B [on page 66) describes a circuit that multiplies two 8-bit numbers with 19 levels of logic, using a total of 401 gates. {The definition is given in an ad hoc language where, for example, ‘x[0..7]’ is used as an abbreviation for ‘$x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$’. Although 754 gates are actually specified, it's easy to reduce the number to 417, by eliminating cases where at least one input to a gate is the constant 0; we also remove gates that don't contribute to the final output, in order to get down to 401.)

The purpose of this problem is to find as small as possible a set of test inputs (x_1, y_1), (x_2, y_2), . . . , (x_m, y_m) such that, if a chip for the circuit correctly multiplies these m inputs, we can be sure that it has no “single-stuck-at faults.” A single-stuck-at fault occurs when the answer is wrong due to a gate whose output is always “stuck” at 0 or 1, although all other gates are functioning correctly.

Since there are 401 gates, there are 802 potential stuck-at faults; we want to find m inputs that will detect them all, where m is as small as possible. The instructor believes that there may be a solution with $m < 16$.

Notes for Tuesday, February 5

DEK started talking about the next task – Problem 3, which is related to combinatorial algorithms. The problem itself can be expressed as a type of covering problem that is NP-complete. The class is expected only to solve it for a specific circuit — a given parallel multiplier. Hence, it was possible that people might finish it before they got their Ph.D.'s. In fact, there is no requirement to get a provably optimal solution, the only requirement is to get a solution that requires less than 16 test, sets; this gives hope that somebody might finish by the due date.

DEK asked what was meant in regard to the circuit when we said “parallel”.

RFC said that the idea is that we do a lot of things at the same time.

DEK noted that the goal was to minimize the number of levels of logic; this would minimize the propagation delay. In an adder, for example, you want to minimize the carry propagation time. Our multiplier is parallel for the same goal, to have high speed.

But, the problem at hand does not involve any design of circuits. We are not even talking about checking a design; there are ways to prove the logical correctness of designs. Rather, we are going to assume that we will be given manufactured chips that correspond to the parallel multiplier in the appendix. These chips will either be perfectly alright, or they will have one fault. The type of fault, that a chip might have, is that one (and only one) of its gates might be stuck outputting a constant logical level (either a 0 or a 1) .

What the students have to do is figure out a test set for the circuit that has as few tests as they can get away with. (DEK believes that the best that one can do is somewhere between 10 and 16 tests.) Notice that by saying “test set”, you are being told that it doesn't make sense to use conditional test sequences for this problem; in fact, the sequence that the tests are performed in will be irrelevant (you do all see why, don't you?). Note that if you had to actually determine which gate was faulty, then you would quite probably end up with a test sequence where previous test results would influence the selection of which tests to perform in the future.

So: each test will cover some of the fault cases. The tests taken as a set should cover all of the fault cases. It's alright for some fault cases to be covered by more than one test.

DEK asked if anyone knew how to do parallel adds efficiently.

AG said that you would need to do "carry look aheads".

DEK wanted to be more specific. We looked at the case of binary addition of 4-bit numbers.

In adding two binary digits, x_i and y_i, we get a sum, z_i, and a carry, w_i. Their relationships are expressed by the equations:

$$z_i = x_i \oplus y_i$$
$$w_i = x_i \cdot y_i$$

Thus, we have:

$$\begin{array}{ccccc}
 & \text{53} & x_2 & x_1 & x_0 \\
+ & \text{Y3} & y_2 & \text{Yl} & \text{Yo} \\
\hline
 & z_3 & z_2 & z_1 & z_0 \\
w_3 & w_2 & w_1 & w_0 & \\
\hline
a_4 & a_3 & a_2 & a_1 & a_0
\end{array}$$

But, to calculate the a_i's we would need several iterations of addition. It would be good if we could calculate c_i's such that:

$$a_i = z_i \oplus c_{i-1}$$

MJB suggested that:

$$c_i = (x_i \cdot \text{Yi}) \vee (c_{i-1} \cdot x_i) \vee (c_{i-1} \cdot y_i)$$

RFC simplified this to:

$$c_i = (xi \cdot \text{Yi}) \vee (z_i \, . \, c_{i-1})$$
$$c_0 = w_0$$

DEK suggested that we unwrap the recurrence for c_3:

$$\begin{aligned}
c_3 &= w_3 \vee z_3 c_2 \\
&= w_3 \vee z_3(w_2 \vee z_2 c_1) \\
&= w_3 \vee z_3 w_2 \vee z_3 z_2 w_1 \vee z_3 z_2 z_1 w_0
\end{aligned}$$

DEK wanted an efficient way to compute c_i. He noted that there was a time when people were discovering lots of algorithms that were much better than the previously known ones. Algorithms that had been $O(n^2)$ were improved to $O(n \log n)$, and $O(n^3)$ ones were improved to $O(n^2)$. Analyzing all the techniques by which this happened led to the-realization that they were all done by what came to be known as "divide and conquer". With this in mind, how could we efficiently compute:

$$c_7 = w_7 \vee z_7w_6 \vee \cdots \vee z_7z_6z_5z_4z_3z_2z_1w_0$$

AAM suggested maybe a bit-slicing architecture that combined ripple-through and carry-look-ahead would work.

ARG thought that some sort of binary tree type approach would work.

DEK mentioned that sometimes when doing things in parallel, when what you are computing depends on a number that has not yet been calculated, you calculate two results, one based on each possible value of the unknown number. When the value of the number becomes known, then you select the right result.

He suggested another way, based on re-writing c_7 as:

$$\begin{aligned} c_7 = &\,(w_7 \vee z_7w_6 \vee z_7z_6w_5 \vee z_7z_6z_5w_4) \\ &\vee (z_7z_6z_5z_4)(w_3 \vee z_3w_2 \vee z_3z_2w_1 \vee z_3z_2z_1w_0) \end{aligned}$$

If we define:

$$\begin{aligned} C_{i,j} &= w_i \vee z_iw_{i-1} \vee \ldots \vee z_iz_{i-1}\ldots z_{j+1}w_j \\ D_{i,j} &= z_iz_{i-1}\ldots z_j \end{aligned}$$

then, we can compute $c_7 = C_{7,0}$ by the recurrences:

$$\begin{aligned} C_{i,j} &= C_{i,k} \vee (D_{i,k} \,.\, C_{k-1,j}), \text{ where } k = \frac{i+j+1}{2} \\ D_{i,j} &= D_{i,k} \,.\, D_{k-1,j} \end{aligned}$$

From this, we can see that we only need $2 \lg n + c$ levels of logic to compute c_n. DEK asked if anyone thought that this could be improved.

ARG said that it couldn't be improved if we only allowed a constant number of inputs for a gate.

DEK agreed that if we allowed an arbitrary number of inputs, then we could just re-express our formula in sum-of-products form and then we would only need 2 levels of logic.

ARG pointed out that the leftmost output bit depends on all of the input bits.

DEK continued that with n levels of two-input gates, we could only "use" 2^n inputs. He mentioned that Bob Floyd teaches a course that contains proofs about tight upper and lower bounds for the number of levels of logic for parallel addition and multiplication.

DEK now switched to parallel multiplication, since this is the function performed by the circuit that we are looking at. He pointed out that the parallel multiplication needs parallel addition of the products.

He pointed out that the constuction of the circuit that is described in the handout will lead to 'and'gates with a hard-wired zero as one of its inputs. Such a gate can be removed from the design, since it will always output a zero; it also ***should*** be removed from the circuit because it is untestable. (Note that removing one gate may then allow you to remove another gate that you previously couldn't have.)

DEK claimed that n n-bit numbers could be added in $O(\log n)$ levels of gates. No one believed him.

He explained that a full-adder could be viewed as a way to transform 3 bits into 2 bits. For a full-adder $x_i + y_i + z_i = sum_i + 2 \cdot carry_i$. By sticking n such circuits in parallel, we have a circuit for n-bit wide numbers such that $X + Y + Z = SUM + 2 \cdot CARRY$.

Thus, if we had n n-bit numbers, Xi,. . . , X_n, we could add the numbers together to yield two n-bit numbers in O(log n) levels of logic by the following method:

1. Initialize i to 0.
2. If $n + 2i + 1 \leq 3i + 3$ then STOP.
3. Form X_{n+2i+1} as SUM from the inputs $X_{3i+1}, X_{3i+2}, X_{3i+3}$.
4. Form X_{n+2i+2} as $2 \cdot CARRY$ from the inputs $X_{3i+1}, X_{3i+2}, X_{3i+3}$.
5. Increment i.
6. Goto 2.

It is clear that we use $n - 2$ of our n-bit wide circuits. But, how many levels of logic gates are used? (Note that our n-bit-wide circuits have a constant number of levels.)

The recurrence that we get for the number of levels is $T(3n) = 1 + T(2n)$.

Assume that $T(n) = c \cdot \log n$.

Hence, $c \cdot \log 3n = 1 + c \cdot \log 2n$.

Yielding $c \cdot \log \frac{3}{2} = 1$.

Finally, $T(n) = \frac{\log n}{\log \frac{3}{2}} = \log_{1.5} n$.

In summary, we need $O(\log_{1.5} n)$ levels of logic to combine the n given n-bit numbers into two n-bit numbers. We then need another $O(\log n)$ levels to add these 2 numbers yielding our final n-bit answer.

DEK ended the class by suggesting that the students should first write a program that builds an internal representation of the circuit and throws away the unneeded gates.

Notes for Tuesday, February 12

DEK wanted to explore techniques for Problem 3 by looking at a small example. The example chosen was a simple adding circuit that had three inputs, x, y and z and two outputs, u (the sum) and v (the carry). The circuit was built up by five gates as follows:

$$a = x \oplus y$$
$$b = x \wedge y$$
$$C = a \wedge z$$
$$U = a \oplus z$$
$$v = b \vee c$$

DEK wanted to build up a table of what triples of inputs, xyz, would be needed to cover each of the possible stuck-at-faults. He wanted to start with gate a stuck-at-O.

ARG mentioned that our test input would have to satisfy a = x $\oplus$ y = 1.

DEK asked if that was all that we needed.

KES added that we also needed the output of a to propagate to a visible output.

RMW noted that for this particular case, *a* would propagate to u regardless of what *z* was.

Hence, we could fill in the first entry of our table; it was $x\bar{x}z$. The next entry that we wanted to fill was for a stuck-at-l.

DEA quickly saw that xxz was the answer.

Next we went to solve *b* stuck-at-O. We needed to force *b* = x A y = 1, and also insure that *b* was visible.

RMW noted that x A y = 1 implied that a = x $\oplus$ y = 0. Hence, c was zero. Thus, *b* is visible.

DEK observed that we were using both forward and backward propagation of constraints in trying to solve this. Now, what do we need as an input to test *b* stuck-at-l?

RMW said that $x\bar{x}0$ or $00z$ would work.

DEK verified RMW's answer with the following reasoning: For *b* to be visible, we must have c = a A *z* = 0. There are two cases. From the a = x A y = 0 case we can see that the input $00z$ would work. For the case *z* = 0, we can see that any text input of the form $x\bar{x}0$ would cover the fault.

MDD went through the analysis for the c stuck-at-0 case. We need c = a $\wedge z =$ 1, so the input *z* must be a 1. We also need a = x $\oplus$ y = 1; fortunately this makes *b* = 0, and hence c is visible. Final answer: $x\bar{x}1$.

Analyzing the c stuck-at-l case fell to RFC. He noted that, we still need to be able to reach the output, so we need *b* = x $\wedge$ y = 0, again. We also want c = a $\wedge z =$ 0. Hence, we have two cases. We can have *z* = 0. Alternately, if x and y are both zero, then c would be 0 and visible. Hence, $x\bar{x}0, 00z$ were the answers.

The cases for u proved easy. We turned to v.

RFC interrupted by saying that he didn’t think that the answer for c stuck-at-1 was right.

DEK asked if RFC liked the answer for *b* stuck-at-l (which has the same answer as c stuck-at-l).

RFC did believe that the answer for *b* stuck-at-l was right.

DEK said that while he might not have complete faith in the answers for these two faults, he did believe that they should have the same answer, since covering them seemed to lead to the same equations.

He mentioned that for many logic puzzles it is really hard to get a good intuition as to what is right and what is wrong. Frequently, you just have to crank it out by hand to see if it is right. DEK re-checked the answer in question by hand and found that it was alright.

RWH suggested that for v it might be easier to take into account that this gate was the carry output. This would save us a lot of tedious calculation.

DEK admonished RWH for this idea. The whole reason that we were doing this by hand was so that we could gain insight in to how to have a computer do it. The computer would need to work on 400 gates and it wouldn’t be able to take into account the ‘meaning’ of the various bits in deciding on what set of test inputs to use. It would need a mechanical method.

DEK noted that there were some circuits that could be made to have zeros as outputs, regardless of their inputs.

AG suggested that this implied that there were **simpifications** that could have been made to the circuit, beforehand.

DEK said that in building up the circuit for this problem from the description, he expected us to make two types of simplifications to the circuits. First, many gates which had a constant as one of its inputs could be short-circuited around. Second, any gate which didn’t have an effect on the final output could be removed.

MDD mentioned that he thought that he had found an ‘and’gate in the **400** gate circuit which was constant, because one of its inputs was always zero.

AG pointed out that if such “always zero” gates are simplified out prior to the analysis, then their stuck-at-one faults cannot be detected. To be consistent, one should remove either all such “always” gates, or none of them.

DEK said that we could come back to this after we finished filling in the table. We still needed to fill in the value for v stuck-at-l.

AJU analyzed it as follows: ***b*** and c must be 0, to force v to 0. So, we either must have $x\bar{x}0$ or $00z$ as our input.

The final table looked like:

gate	**Stuck-at-0**	Stuck-at-l
a	$x\bar{x}z$	xxz
b	$11z$	$00z, x\bar{x}0$
c	$x\bar{x}1$	$00z, x\bar{x}0$
u	$x\bar{x}0, xx1$	$x\bar{x}1, xx0$
v	112, $x\bar{x}1$	$00z, x\bar{x}0$

DEK pointed out that while we have a complete table, on a larger circuit we might not be able to compute a complete table. We also might not need a complete table, since we don’t need all solutions.

[From this table one generalization that we can draw is: If you cover a gate for both stuck-at-0 and stuck-at-l faults, and the output of this gate is the input to ***only one other*** gate (and this other gate is ‘and’ or ‘or’), then this other gate automatically has both its stuck-at-0 and stuck-at-l faults covered.

There are many other relationships of the form: If you cover faults ***m*** and ***n***, then faults o and ***p*** are also automatically covered. This suggests a side-problem: Find the smallest subset of the faults, such that, covering this set of faults implies that all of the other faults are also covered.]

We returned to examine the possibility of having a gate which, for a particular fault, was untestable. MDD said that he had found such a gate in either a **2-bit** by 2-bit multiplier or in a **3-bit** by **3-bit** multiplier, though he couldn’t remember which.

AG insisted that this really shouldn’t happen if you have simplified the circuit as much as possible.

Tuesday, February 12

Many people pointed out that the task of simplifying a circuit as much as possible is, in general, a hard problem.

DEK tried to give people an intuitive idea of how this problem could arise. If for a multiplication circuit there is some gate that can only be tested when the output is all l's, and if the number with all l's happens to be a prime, then no input will be able to test this gate. Showing that the gate can not be tested, however, would require the use of number theory or something equivalent.

While MDD was unable to reconstruct the example that he had come across, AG had a circuit in which two different gates computed the exact same value by two different techniques.

AAM pointed out that if we had then had an ‘exclusive-or’ gate that used these two gates as input, then we would never be able to test the gate for stuck-at-O. (Of course, it is not clear why we would then *care* if the gate was stuck-at-O.)

RMW didn’t think that there was a way to tell if a gate couldn’t be tested for a fault without running through and trying all the inputs.

Some other people seemed to think that some back-and-forth propagation methods might suffice.

DEK said that Problem *3* was rich enough that he didn’t expect to see everyone come up with the same approach. For example, using the C language might allow you to test 32 different test inputs in parallel by using bit-wise operations on words; this efficiency might make different approaches more feasible than they would be in some other language.

DEK now wanted to turn to the dual problem. Given an input for the circuit, what stuck-at faults will it detect? We started with the input 011.

MDD noted that it was only possible to detect faults that were the opposite of the correct output of a gate.

DEK asked if we could detect all such faults with this input.

ARG pointed out that the value had to reach the output.

So, with x = 0, y = 1 and z = 1 we have:

$a = x \oplus y = 1$	maybe it detects:	a stuck-at-0
$b = x \wedge y = 0$		*b* stuck-at-1
$c = a \wedge z = 1$		c stuck-at-0
$U = a \oplus z = 0$		u stuck-at-l
$V = b \vee c = 1$		v stuck-at-0

RMW said that it was obvious that the faults for u and v would be detected.

DEK asked which others would be detected?

AG mentioned that an input of 1 to an ‘or’ gate masks the other input from being visible. (Hence *b* stuck-at-1 is not detected in this case.) An input of 0 to an ‘and’ gate similarly masks the other input.

A value going into an ‘xor’ gate could only be masked if the other input was the same function of the inputs as the value or its complement.

DEK suggested that there might be a way of working strictly downwards through the circuit, starting from the input, that would allow you, upon reaching the outputs, to know exactly which gates could have a fault detected by the input.

MDD thought that this could be done by simultaneously trying both cases for a gate.

DEK mentioned that he thought that a subroutine, which figured out what faults could be detected by a given input, would be an interesting one to have. He moved on to another approach. Can we characterize when the output at v depends on the output of gate a? He explained that every function could be re-written using **‘xor’** and ‘and’ such that we could derive an equation of the form:

$$v(x, y, z) = a \cdot f(x, y, z) \oplus g(x, y, z)$$

So when is a visible?

$$\begin{aligned} v &= b \oplus c \oplus b \cdot c \\ &= x \cdot y \oplus c \oplus x \cdot y \cdot c \\ &= x \cdot y \oplus a \cdot z \oplus x \cdot y \cdot a \cdot z \\ &= a \cdot (z \oplus x \cdot y \cdot z) \oplus x \cdot y \end{aligned}$$

ARG noticed that a was visible when $f = 1$, invisible when $f = 0$.

AAM pointed out that finding all conditions when $f = 1$ was a hard problem.

DEK ended the class with the observation that every problem that anyone has ever solved is a special case of some unsolvable problem, but we still manage to solve the special cases.

Notes for Tuesday, February 19

DEK opened the class by reminding people that there were only 23 more days before Problem 5 was due.

Then he rewrote the equations for a circuit from the last class:

$$\begin{gathered} x \oplus y \rightarrow a \\ x \wedge y \rightarrow b \\ a \oplus z \rightarrow u \\ a \wedge z \rightarrow c \\ b \vee c \rightarrow v \end{gathered}$$

He noted that since we were dealing with combinational circuits in this problem, the circuits could be represented as **DAGs** (directed acyclic graphs). He asked how people represented these in their programs.

ARG was using LISP. Each gate had a symbol associated with it, and the information about the gate was stored on its property list. The circuit above would be represented as:

```
IO -   type : input  inputs : nil      outputs: (GO  Gl)
11 -   type : input  inputs : nil      outputs: (GO  Gl)
12 -   type : input   inputs : nil     outputs : (G2  G3)
GO -   type:  xor     inputs : (IO 11) outputs : (G2  G3)
etc...
```

DEK asked if that was what everyone used. Everyone sat quietly, smiling. DEK asked if everyone used LISP.

RMW explained that he was using C, but that his data structure was essentially the same as **ARG's.**

DEK noted that both people had extra unnecessary redundancy in their data structures to allow some operations to be carried out efficiently. DEK asked ARG how the name 'GO' got assigned.

ARG explained that the numbers were assigned in the order that the gates were generated by his procedures.

DEK asked if there was any numerical ordering on the gates that might be useful.

ARG noted that a gate's inputs are generated before the gate is. Hence, the numbers for the gates feeding any given gate's inputs are lower than the number for the gate itself.

DEK asked how many gates his circuit had.

ARG said that, after the simplifications that we were expected to make, he got 401 gates.

DEK mentioned that if you used our construction to build a 16 by 16 bit parallel multiplier, the number of gates would go up to 1722; for 32 by 32 bits, it's 6933 gates. There was some confusion as to whether inputs should be counted as gates.

DEK wanted to discuss the possibility that a gate, that we hadn't simplified out of the circuit, had a solid 0 as its output (i.e., 0 output regardless of input).

MDD conjectured that there was at least one such 'and' gate, G, for each multiplier over a certain size. He had looked at multipliers up to 5 by 5 bits.

RMW said that there was a gate, G', that couldn't be checked, because its output only went to G. Its output would be masked.

MDD pointed out that while we couldn't detect a stuck-at-0 fault for G, we could still detect a stuck-at-1 fault for it.

DEK mentioned that there was indeed a G for the 8 by 8 bit multiplier, but he didn't think that there was a G'. He asked whether we should prove that the gate always outputs a 0, and remove it from our circuit, or whether we should settle for just detecting all the detectable faults.

JFW pointed out that any fault that was untestable, was also irrelevant to the proper functioning of the circuit. All that we should worry about is whether the circuit works right.

DEK decided to tell the class which gate caused the problem. He thought that an actual proof that it would always be 0 would take 1 or 2 pages.

Let $a_0, \ldots, a_7$ be the 8 partial products that we have to add together to get the final product. DEK reminded us of how we did the addition. We compute:

$$a_0 + a_1 + a_2 \rightarrow a_8 + a_9$$

$$a_3 + a_4 + a_5 \rightarrow a_{10} + a_{11}$$

$$a_{15} + a_{16} + a_{17} \rightarrow a_{18} + a_{19}$$

It turns out that the high order bit of a_{19}, which is a carry bit from the addition: $a_{15} + a_{16} + a_{17}$, will always be 0. But, to show this you need to know that $a_0, \ldots, a_7$ came from a multiplication; it isn't true for all choices of $a_0, \ldots, a_7$.

AAM pointed out that with an arbitrarily complex circuit, we wouldn't be able to prove by hand which gates would have constant outputs.

DEK mentioned that **JFW's** point, that what we can't test is irrelevant to the proper functioning of the circuit, was a good one. The other alternative to finding a test set that just verifies that the **401-gate** circuit works correctly, is for us to simplify G out of the circuit. DEK wanted everyone to be working on the same problem, so we had a vote to determine which way we would go. By 8 to 3 (and 4 abstentions) we decided to leave any gates, with possibly untestable faults, in the circuit.

Thus we reformulated the problem slightly: Our test inputs $(x_1, y_1), \ldots, (x_m, y_m)$ should have the property that a chip that correctly multiplies all test inputs will correctly multiply all inputs, assuming that each chip is either correct or has a single stuck-as fault.

AG said that this meant that everyone would have to prove by brute force that a redundant gate's output is always zero.

DEK responded that there were a variety of methods that are useful in doing this problem. Some of them might possibly bog down in the combinatorics. When he himself worked on the problem, he found that he needed to use a combination of different methods.

DEK wanted to return back to the question of data structures. He asked RMW how he could be using the same data structure as ARG, if they were using different languages (C doesn't have property lists).

RMW said that he wasn't using property lists, but that he was representing gates out of the same basic elements; for each gate, he had a description of its inputs, outputs, and **type.**

DEK suggested another method of storing the information which was more **compiler-**like.

gut e:	***type***	***op***	***aux***
1:	input	7	
2:	input	8	
3:	input	11	
4:	fetch	1	0
5:	fetch	2	0
6:	xor	10	
7:	fetch	1	4
8:	fetch	2	5
9:	and	**13**	
10:	fetch	6	0
11:	fetch	3	0
12:	xor	14	
13:	output	9	0
14:	output	12	0

Note that for any gate, we have a list of its outputs via ***op*** and ***aux*** links. From this, it is clear that we could do this problem in FORTRAN if we really had to.

DEK said that this was not the only data structure that we might need. If we have a heuristic search that is going forward and backward, we might want to propagate information about gates that are assumed to be constant, or are assumed to have relations to other gates. By doing this, if we ever get $B \wedge \bar{B}$ we will know that it's 0 even though we don't know what B is.

DEK asked for any ideas about how we might represent this kind of information. Everyone sat quietly, not smiling.

KES suggested that we could do it the way it is done in Data Flow Analysis. Here, you keep track of a whole bunch of expressions that are true as you go through a tree.

DEK pointed out that here we have a very special case. We only need to group things into equivalence classes. What is a good way to group things into equivalence classes?

AG mentioned that we could use UNION-FIND.

DEK agreed that UNION-FIND was a good way to represent equivalence classes. We will only need a slight modification of it here (since we want to be able to express $x_1 = x_3 = \bar{x}_4$). He mentioned that we would be using this for branching, so we either needed to save everything so that we could back up, or we just had to be able to undo things. DEK said that there was a data structure for UNION-FIND where it is easy to undo the UNIONS.

All the elements of an equivalence class are joined in a circular list. They each have a pointer to a header node, which has stored in it a count of the number of elements in the equivalence class. To check the equivalence of two symbols, we only need to do a FIND on each and see if they have the same header node. To do a UNION, we only need to merge two lists, and update the pointers to the header in the nodes of the shorter list. DEK asked what the most efficient way to do a UNION was, given a pointer to p in one equivalence class, and a pointer to q in the other.

AAM said that we should make p point to the node that q pointed to, and make q point to the node that p pointed to.

ARG mentioned that we also needed to modify the header information.

DEK agreed that we still needed to modify the pointers to the header, and the count for the header.

AAM suggested that if we added one more level of indirection, we wouldn't have to update as many pointers.

DEK agreed that there were more efficient UNION-FIND algorithms than the one he was presenting. The average time for this one is $O(n)$, the worst case is $O(n\,.\,\log n)$. It is because we keep track of the sizes of our equivalence classes, and merge the smaller one into the larger one that the worst time is $O(n \cdot \log n)$. **Tarjan** has shown that there is a more efficient algorithm that is $O(n\,.\,a(n))$ where $a(n)$ (which is related to the inverse of Ackerman's function) grows so slowly that $a(n) \leq 5$ for all practical values of n.

Now, DEK wanted to determine what information we needed to keep around to undo a UNION. Clearly, we would want to swap back the pointers for p and q. Hence, we needed to be able to locate p and q. So, if we kept a stack of p's and q's, what else would we need to undo a UNION?

MDD suggested that it would be good to have the old header back.

RMW said that it would be easy enough to just add a pointer to it to our stack.

DEK asked if we would be able to get away with only saving p and the old header in our stack.

KES saw that we could do this. Since the old header still had its count in it, we could easily figure out what q was.

DEK noted that if one wasn't careful, one could reverse the sense of the equivalence class, but this didn't really matter ($x_1 = x_3 = \bar{x}_4$ might have been changed to $\bar{x}_1 = \bar{x}_3 = x_4$).

MDD said that he didn't see how this could be used to represent the possibilities for the gates feeding into an 'or' gate that you knew was outputting a 1.

DEK agreed that this didn't cover that. He only claimed that this was a way that we could represent equivalence classes. For cases like the 'or' gate, one would have to make an assumption, which one could later undo if it proved fruitless.

We will frequently be interested in sub-problems such as: find an input such that $g_5 = 1$, $g_6 = 0$, and $g_{300} = 1$. This problem is similar in nature to theorem proving problems. We want to find a set of axioms such that certain statements are theorems. When one actually starts working on it, you will see that the mix between what is known, what is assumed, and what is derived based on what is known and assumed can get very confusing.

As in all methods that do branching, it probably pays to do some work to limit the number of branches that we have to follow.

AG suggested that forward-chaining would also be useful in doing this sub-problem. If by backward chaining from some gates high up (close to the inputs) in the circuit, we find conditions on the inputs, we can then propagate these conditions forward to gates lower in the circuit.

AAM wondered if DEK had actually used the two data structures that he had discussed in class.

DEK had. His program had been written in WEB.

Notes for Thursday, February 21

Today, DEK wanted to look at other aspects of Problem 3. He wanted to explore the structure of the problem and how the problem related to other problems.

First, he wanted to tidy up the discussion that we had had a few weeks ago: given a set of inputs, what faults does it detect? DEK wanted to know how people in the class had solved this.

ARG wanted to know what kind of analysis we could assume that we had already pre-computed.

DEK wanted us to look at the problem assuming that we just knew the inputs. He presented a sketch of a method. For each gate, G,, whose two inputs come from gates G_i and G_j, you should solve the following sub-problem: if G_n were an output, find $S_n = \{G_p^q \mid$ we can detect G_p stuck-at-q by looking at the output of $G_n\}$. You would only compute S_n after computing S_i and S_j, so that you could use them in the computation of S_n.

We started out considering the circuit:

$$G_1: \qquad x \wedge y \rightarrow v_1$$

$$G_2: \qquad x \oplus z \to v_2$$
$$G_3: \qquad v_1 \vee v_2 \to v_3$$

with the inputs x = 0, y = 1 and $z = 1$. What was S_1?

ARG said that we could detect G_1 stuck-at-l by looking at G_1.

So, $S_1 = \{G_1^1\}$. DEK added that for G_2 we could detect G_2 stuck at zero; $S_2 = \{G_2^0\}$. Also, $S_3 = \{G_3^0, G_2^0\}$. DEK claimed that there was a way to compute these sets, so we started to explore that problem.

KES had reservations about the iterative framework that was proposed. She was worried that a stuck-at fault might fan-out to many gates, propagate a while, and then fan-in and cancel itself out. She didn't think that this approach would be able to handle it.

DEK thought that we would be able to handle this, if we did things right. We examined a general 'or' gate:

$$G_n: \qquad v_i \vee v_j \to v_n$$

We wanted to compute S_n.

ARG saw that $G_n^{\bar{v}_n}$ was certainly in S_n.

DEK pointed out that, for our example, S_3 picked up S_2 but not S_1.

After some discussion, it was conjectured that

$$S_n = \{G_n^{\bar{v}_n}\} \cup (S_i \text{ if } v_j = 0) \cup (S_j \text{ if } v_i = 0).$$

AAM thought that there must be a term of the form $S_i - S_j$ somewhere in the expression if this method solved KES's objection.

DEK wanted to check it out algebraically.

RFC re-emphasized AAM's point by saying that if $v_i = 0$ and $v_j = 1$ were the correct values, then we wouldn't be able to detect the case when both values were in error.

KES suggested that we look at all four possible cases of v_i and v_j.

DEK decided to check it out by algebra. We wanted to find out when an error from G_k would be detectable at G_n. Remember from a previous discussion, that the value at any gate, g_i, can be expressed as a polynomial of the value at another gate, g_k, by an equation of the form:

$$g_i = g_k \cdot f_{i,k} \oplus h_{i,k},$$

and that a stuck-at fault at gate k will be detectable at gate i iff $f_{i,k} = 1$. So, let's compute the value of $f_{n,k}$.

$$
\begin{aligned}
g_i &= g_k \,.\, f_{i,k} \oplus h_{i,k} \\
g_j &= g_k \cdot f_{j,k} \oplus h_{j,k} \\
g_n &= g_k \,.\, f_{n,k} \oplus h_{n,k} \\
&= g_i \vee g_j \\
&= (g_k \,.\, f_{i,k} \oplus h_{i,k}) \vee (g_k \,.\, f_{j,k} \oplus h_{j,k}) \\
&= \overline{(g_k \cdot f_{i,k} \oplus h_{i,k}) \vee (g_k \,.\, f_{j,k} \oplus h_{j,k})} \oplus 1 \\
&= \overline{(g_k \cdot f_{i,k} \oplus h_{i,k}) \,.\, (g_k \,.\, f_{j,k} \oplus h_{j,k})} \oplus 1 \\
&= (g_k \cdot f_{i,k} \oplus h_{i,k} \oplus 1) \,.\, (g_k \,.\, f_{j,k} \oplus h_{j,k} \oplus 1) \oplus 1 \\
&= g_k \,.\, (f_{i,k} \,.\, f_{j,k} \oplus f_{i,k} \cdot h_{j,k} \oplus f_{i,k} \oplus f_{j,k} \cdot h_{i,k} \oplus f_{j,k}) \oplus (h_{i,k} \oplus 1) \cdot (h_{j,k} \oplus 1) \oplus 1
\end{aligned}
$$

By also noting that the values v_i satisfy

$$v_i = v_k \cdot f_{i,k} \oplus h_{i,k}, \quad \text{hence}$$
$$h_{i,k} = v_i \oplus v_k \cdot f_{i,k},$$

we can conclude that:

$$
\begin{aligned}
h_{n,k} &= (h_{i,k} \oplus 1) \,.\, (h_{j,k} \oplus 1) \oplus 1 \\
&= h_{i,k} \vee h_{j,k} \\
f_{n,k} &= f_{i,k} \cdot f_{j,k} \oplus f_{i,k} \cdot h_{j,k} \oplus f_{i,k} \oplus f_{j,k} \,.\, h_{i,k} \oplus f_{j,k} \\
&= (f_{i,k} \vee f_{j,k}) \oplus f_{i,k} \,.\, h_{j,k} \oplus f_{j,k} \,.\, h_{i,k} \\
&= (f_{i,k} \vee f_{j,k}) \oplus f_{i,k} \,.\, (v_j \oplus v_k \cdot f_{j,k}) \oplus f_{j,k} \cdot (v_i \oplus v_k \cdot f_{i,k}) \\
&= (f_{i,k} \vee f_{j,k}) \oplus v_i \cdot f_{j,k} \oplus v_j \cdot f_{i,k}
\end{aligned}
$$

Considering that:

$$
\begin{aligned}
G_k^{\bar{v}_k} \in S_n &\iff G_k \text{ stuck-at-}\bar{v}_k \text{ is detectable at } G_n \\
&\iff g_n \neq v_n \text{ when } g_k = \bar{v}_k \\
&\iff g_n = (\bar{v}_k \,.\, f_{n,k} \oplus h_{n,k}) \neq (v_k \,.\, f_{n,k} \oplus h_{n,k}) = v_n \\
&\iff f_{n,k} = 1
\end{aligned}
$$

we can derive the formula for S_n in terms of S_i and S_j for an ‘or’ gate:

$$
S_n = \begin{cases}
\{G_n^{\bar{v}_n}\} \cup (S_i \cup S_j) & \text{if } v_i = 0 \text{ and } v_j = 0 \\
\{G_n^{\bar{v}_n}\} \cup (S_j - S_i) & \text{if } v_i = 0 \text{ and } v_j = 1 \\
\{G_n^{\bar{v}_n}\} \cup (S_i - S_j) & \text{if } v_i = 1 \text{ and } v_j = 0 \\
\{G_n^{\bar{v}_n}\} \cup (S_i \cap S_j) & \text{if } v_i = 1 \text{ and } v_j = 1
\end{cases}
$$

Similar recurrences can be derived for ‘and’ and ‘xor’ gates. The one for ‘and’ gates is:

$$
S_n = \begin{cases}
\{G_n^{\bar{v}_n}\} \cup (S_i \cap S_j) & \text{if } v_i = 0 \text{ and } v_j = 0 \\
\{G_n^{\bar{v}_n}\} \cup (S_i - S_j) & \text{if } v_i = 0 \text{ and } v_j = 1 \\
\{G_n^{\bar{v}_n}\} \cup (S_j - S_i) & \text{if } v_i = 1 \text{ and } v_j = 0 \\
\{G_n^{\bar{v}_n}\} \cup (S_i \cup S_j) & \text{if } v_i = 1 \text{ and } v_j = 1
\end{cases}
$$

Thursday, February 21

The one for ‘xor’ gates is simply:

$$S_n = \{G_n^{\bar{v}_n}\} \cup ((S_i \cup S_j) - (S_i \cap S_j))$$

DEK wanted to change perspectives. Now, we wanted to look at the problem as a covering problem. We looked at the table for the full-adder that we had been discussing in previous classes. It shows which inputs cover which stuck-at faults:

	a		*b*		c		u		*v*	
xyz	sa0	sa1	sa0	sa1	sa0	sa1	sa0	sa1	sa0	sa1
000		•		•		•		•		•
001		•		•		•	•			•
010	.			•		•	•			•
011	•				•			•	•	
100	•			•		•	•			•
101	•				•			•	•	
110		•	•					•	•	
111		•	•				•		•	

We wanted to find the optimal solution. That is, the smallest number of rows that were needed to cover all of the columns. DEK asked for ideas about how to solve it.

KAM suggested that, from each row, we count how many faults it covers, and then take the row with the highest number.

DEK agreed that this “greedy” approach was one widely used method.

AAM suggested checking to see if any column only had one input that covered it. Then, we would know right away that we should take this input.

ARG noted that we should apply this rule at any step where it was applicable.

KES joked that we could always convert this problem into a long boolean expression for which we would *only* have to find the minimum way to satisfy it.

DEK pointed out that this at least would give us an upper bound on the complexity of the problem.

AG suggested that we never use an input if another input ‘dominates’ it.

It was suggested that the greedy approach might already incorporate this. DEK pointed out that this wasn’t quite the case, since applying this rule to the inputs 011 and 101 would let us rule one of them out, and then we could apply AAM’s rule.

DEK mentioned that we didn’t really need the optimal solution. Besides, for most problems that arise naturally, the greedy method works very well. It usually only messes up for devious hand-crafted cases. He referred us to a 1960 book by Prof. McCluskey, which he said had one of the best discussions on practical ways to solve covering problems. He said that there are a bunch of good methods even though this problem is *NP*-hard.

AAM mentioned that for large cases, though, it might be prohibitively time-consuming to even just calculate the type of table that we had above.

DEK suggested that some approximate techniques could work reasonably by just trying random inputs. You would try 100 random inputs, and then choose the one of those that covered the most new faults. He himself had tried this idea on our 8 by 8 bit multiplier. He found that this worked fine for most of the gates, but that there were some faults which could only be detected by 1 or 2 different inputs.

Notes for Tuesday, February 26

DEK mentioned that he had been able to find a solution that required 14 tests. He asked how other people had fared.

ARG/AG didn't have a set of tests that completely covered the faults. They were still missing 10 tough-to-get faults.

PCC/JIW/DEA had been able to cover all but 2 faults. They had checked that the two faults were coverable by exhaustively trying all tests; on that run, however, they didn't keep track of what tests covered those faults. It took too much time to try to re-run it.

KES/MJB/AAM had covered all but 18 of the "little peckers". They thought they would be able to get the rest after getting rid of the "final bug" in their program.

KAM/RFC/MDD had a solution that used only 13 inputs.

AJU/RMW still had 15 faults to cover.

JFW/AGT did not have a cover yet.

DEK wanted to know how ARG/AG had gotten down to having 10 faults left to cover.

ARG said that their partial cover had been generated by a variety of different ways. Some of the tests were picked at random, some were chosen by hand, and some were generated by a program.

DEK asked if a lot of them had been chosen at random.

ARG replied that they had only tried a few random ones to detect faults that were not detected by the tests already chosen.

ARG didn't remember how many had been chosen by the program. The program had used a non-exhaustive forcing scheme that would occasionally give up.

DEK wanted to know how they had generated tests by hand.

ARG said that they would enter the test into a program, which then checked to see if the test covered anything new.

What DEK really wanted to know was how they *decided* what test to enter.

ARG had used the knowledge that the circuit was a multiplier.

DEK mentioned that he had, as a first approach, tried a random greedy method. He later found out that there was one gate with only 2 test cases that could detect it stuck at 1, and only 2 test cases that could detect it stuck at 0.

AAM asked if the gate was close to the high-order output bits.

DEK concurred that it was near the end of the circuit. To find tests to cover the last 10 faults, he had had to use an exhaustive search. All of the other faults he had been able to cover through a backtracking technique. The backtracking worked well for gates near the beginning of the circuit. For the last 10 cases, however, the backtracking took forever.

This reminded DEK of a program he had written as an undergraduate. He had tried to find a solution to a puzzle where you remove pegs from a board by jumping other pegs over them; the object was to be left with only one peg. The program worked fine on a 4 by 4 test case. Then he gave the program the actual puzzle. After letting it run for several hours, he stopped it to see how far it had gotten. Doing a few quick calculation, he realized that it would take the program another 200 centuries to get the solution.

Tuesday, February 26

DEK found that 1 of the 10 hard faults had only two inputs that would test it:

$$199 \times 247 = 247 \times 199 = 49153 \approx 3 \times 2^{14}$$

Some of the tests that covered other hard cases were:

$$149 \times 220 = 32780$$
$$147 \times 223 = 32781$$
$$231 \times 142 = 32802$$
$$149 \times 110 = 16390$$

DEK noted that these cases seemed to have a lot of consecutive carries. This suggests that for the next higher case (a 16 by 16 bit multiplier), we should try to factor numbers near 2^{31} and try these as inputs.

DEK suggested that the two faults that PCC/JIW/DEA had not covered were probably covered by tests on this list. But, he wondered how they had gotten as far as they had.

PCC said that they had started looking at smaller cases. For a 4 by 4 bit multiplier, they had an optimal solution with 5 tests. For a 5 by 5 bit, they were able to get by with 7 tests.

DEK wondered if they had been able to use their results from the 4 by 4 bit multiplier in looking for a solution for the 8 by 8 bit multiplier.

PCC reported that they had tried to recognize patterns of pairs in the solutions for the 4 by 4 case. They had found 70 different sets of 5 tests that covered the 4 by 4 case.

DEK thought that looking for something that was generalizable was usually a good strategy, but it might not work too well in this problem. Intuitively he wouldn't trust it, because our circuit isn't quite built up recursively: The 4 by 4 is not a sub-part of our 8 by 8.

JIW noted that running the 4 by 4 exhaustively had taken 20 minutes. The 5 by 5 took too long; they didn't let it run to completion.

DEK mentioned that Marshall Hall (who had been his professor at Cal Tech) had made a famous observation: Getting another order of magnitude of computing resources only enables you to solve combinatorial problems that are one more case harder.

PCC said that the 4 by 4 case hadn't helped them too much. They got most of their test cases by good guessing on the part of DEA.

DEA explained that they had a fast simulator that, given an input test, tells you what gates get covered.

DEK asked how they-did their fast search.

DEA explained that they had used C.

DEK wanted to know what their algorithm was. How did it differ from the one presented in the last class?

JIW said that they had run all 802 cases in parallel using bit-wise operations on full words.

DEK asked KES/MJB/AAM how many tests they needed to cover the faults that they were able to cover.

AAM suggested that if we were only interested in a few gates, then there might be a symbolic way to find inputs for those gates.

KES explained that they had spent the first two weeks of the problem in a theoretical vacuum. They had used an elaborate probabilistic scheme. They had assumed that each input covered about the same number of faults, and they had used some heuristics to guide their search. They hadn't foreseen that a few cases might be hard to cover. But, some of the tools that they built in the early stages did turn out to be useful later. Now, they were trying to work on forcing gates to particular values.

AAM said that they had originally tried a greedy approach. They hadn't wanted to manipulate constraints backward and forward. Instead, they set one bit of the input at a time, trying to cover new gates as they did so. They used a heuristic estimate of the number of remaining faults that would be covered by different values of the bits. The estimates turned out to be good when only few faults had been covered yet.

DEK explained that when he had multiple tests for covering some one of the hard inputs, he chose the one that covered the most other gates. It turned out that the input 147 x 223 covered two of the hard cases.

MDD said that the program he wrote with KAM and RFC (which was the only one to find a full cover, and better than DEK's to boot) wasn't really as clever as some of the ideas mentioned; for instance, it doesn't take advantage of the fact that the circuit is a multiplier. Their program has two stages: first, they try to find a sufficient cover; then, they try to minimize it.

DEK asked if anyone else had used this two-stage approach.

ARG/AG had.

MDD reported that they first found a sufficient solution with 20 tests.

DEK asked if their final 13 tests was just a subset of these 20.

MDD said that it wasn't. For instance, they ran the first stage on different configurations of the circuit. These different configurations were gotten by interchanging the two inputs to some gates. This caused the program to search the circuit in a different order.

RFC explained that from different runs they eventually had a file with 100 tests: five groups of 20.

MDD continued that a greedy covering algorithm was only able to cut this down to 14 tests. To get their final answer of 13, they tinkered with the set of 100 tests. They guessed that one of the greedy algorithm's early choices was a bad one, so they removed it from the file.

DEK wondered if anyone had independently verified that their cover was correct. He had only checked that it had one of 199 x 247 or 247 x 199.

AAM wondered how they had generated their sufficient cover.

DEK agreed that this was the interesting part.

MDD joked that they had "used Pascal".

AAM conceded that his original choice of LISP had been a poor one.

ARG agreed, saying that he had changed from LISP to C, since the Vax's LISP had poor running time and a poor compiler.

MDD continued that they didn't use any of the symbolic methods that had been described in class. Each gate was specified as either being a 0 or a 1 or unknown.

RFC said that they had kept flags of which faults had been covered already.

DEK asked if that means that their program said, "here's a gate that I've covered for stuck-at-O, but not for stuck-at-l, so now I'll try to force it to 0."

MDD explained that they started at the outputs and worked toward the inputs. That is, they decided that they wanted to force a gate to a specific value, and then they proceeded to try to force it.

DEK mentioned that in his solution, it was when trying to force the value of an exclusive-or gate that he used his symbolic **(in)equality** method based on UNION-FINDs.

MDD said that they just branched to handle that.

RFC elaborated that if they wanted to force, for example, an AND gate to 0, they had a FORCE-VALUE procedure that they called to do it. If they were trying to see if there was a stuck-at fault at this gate, then they would also call a FORCE-VISIBLE procedure to make sure that this gate was visible. Also, in calling the FORCE-VALUE procedure, they would set a DETECT parameter to indicate that a fault at the gate was going to be detectable; hence, if any other gates could be checked for faults easily, they would know to do it.

DEK wanted to know how they did the FORCE-VISIBLE.

MDD explained that the routine looked at the path through the circuit from the gate that they were inspecting to an output bit. If the path went through an AND gate, then the other input to the gate would be **FORCE-VALUEd** to 1; for an OR gate, the other input would be **FORCE-VALUEd** to 0. These were just heuristics that might not work if the gates were wired up in funny ways.

RFC pointed out that FORCE-VISIBLE was actually implemented recursively. To FORCE-VISIBLE a gate, besides **FORCE-VALUEing** one of its inputs, you would also FORCE-VISIBLE a gate that its output went to.

DEK concluded that the motto of this problem seemed to be that, "greed might not always win, but dishonesty might."

KAM mentioned that she wrote the code that would then be called to check which gates were ***actually*** covered.

DEA commented that his group also tried something similar. First, they generated a path from a gate they were testing to be faulty to an output gate. Then they worked forwards and backwards, setting the values of gates until they were done with setting all gates to their desired values, or until they decided it was impossible.

RFC said that the general idea was just to get a good test. It didn't have to be optimal.

DEK pointed out that forward and backward propagation were both used in searching in AI tasks. It can be viewed as just a special case of hypothesis validation. It is similar to having a model, and then designing an experiment that might be able to refute it. When DEK was exploring this problem, Mike Genesereth showed him a paper where this idea was applied to symbolic integration.

AAM wondered how **MDD/RFC/KAM** continued after covering one fault. That is, after they have some partial restrictions on the inputs so that a certain fault is detected, how do they continue? How do they decide what fault to try to cover next?

MDD replied that they do this in the same way that they picked the gate that they just succeeded in covering. They simply choose the next highest numbered gate (closest to the output) that is uncovered.

DEK asked how they make use of the knowledge they keep about what still needs to be done. "We all pretty much understand your heuristics for visibility. And that you then have a final test afterwards to find out what you actually accomplished. This method has the advantages of all good heuristics; it is usually true, it requires only simple control, and it reduces the search. But, how do you decide what else will be fruitful?"

MDD explained that they try to pick up extra gates wherever they can.

DEK wanted to know exactly how they decided on the inputs.

MDD said that the first step was to assume that the faults on the current gate were visible. Then, you would try to see if you could also detect something else.

RFC gave an example. If you had an and gate that you were verifying for stuck-at-l, and you were thus **FORCE-VALUEing** it to 0, with the **DETECTable** parameter set, then you would want at least one of its inputs to be a 0. By making the other input a 1, you might be able to detect more faults. Say that gate feeding the right input hasn't had its stuck-at-l fault covered yet, but the gate that feeds the left input has had its stuck-at-l fault covered. Then, you would want to FORCE-VALUE the left gate to 1 (making the other gate visible) with the DETECT parameter off, since this gate won't be visible. For the right gate, you would FORCE-VALUE it to 0 with the DETECT parameter set.

If both gates needed to be **FORCE-VALUEd** with the DETECT parameter set, they would choose the one on the left. This is why re-arranging the circuit (switching which gate fed the left input and which fed the right input) would result in a different answer.

In FORCE-VISIBLE, RFC mentioned that they choose to go through XOR gates, wherever possible.

DEK mentioned that he had done a study of the gates and their visibilities. He found that in only eight cases would this heuristic fail.

AAM mentioned that in their computations of probabilities that a fault will be propagated forward towards an observable gate, they found, that in only **5** cases did their probability drop below **.95.**

DEK concluded the class by noting that once again the class had come up with many good ideas for this problem.

Notes for Thursday, February 28

By the next class, RMW/AJU had found a solution with only 10 tests! RMW explained that they had started an exhaustive search from 255 x 255 and stopped when they had all but 5 faults covered. This only took them to about 215 x 0, or roughly ten thousand cases. Then they threw in a bunch of randomly generated tests in order to cover the rest.

AJU had a program that checked the covers. It added tests one at a time and saw how many tests could be removed. She noted that you could sometimes remove two after adding one, since one fault might be covered in many ways.

DEK noted that this was a generalized version of dominance. You have a sufficient cover of size k. You try to add a $(k+1)$st element to see if you can wipe out some of the previous k elements.

RMW opined that their solution was not very elegant.

DEK replied that you can't quarrel with success.

AG noted that the method seemed related to 2-optimizing of travelling salesman paths.

Problem 3, Review of Solutions

There is one obvious way to go about solving this problem; unfortunately, it is computationally infeasible. First, for every possible input, you could find out what faults it covers. Then you would just have to solve a set cover problem with 2^{16} sets having elements drawn from a pool of 802 possible faults.

There were numerous attempts to find a small number of tests that would cover all of the faults, without going through all of the possibilities.

One idea is to ***limit*** the number ***of possible*** inputs that are considered. One way to do this was to sample ***randomly*** from all of the possible inputs. Some people used heuristic methods to have their programs generate tests that were expected to test a ***large number of*** faults. Another thing people tried was to have a program analyze the circuit and try to produce inputs that would test particular faults; this was done by either heuristic search, or exhaustive ***search.***

When people had a set of inputs that were sufficient to cover all of the faults, they wanted to minimize these sets. Rather than find actual minimum covers, people were satisfied with near-minimum covers gotten through greedy algorithms, use of the idea of dominance, or through ***hill-climbing*** methods.

MJB/AAM/KES: Unfortunately, this group hadn't foreseen that some of the faults would be hard to detect (only a few of the 2^{16} inputs would detect them). Thus, they used heuristic techniques based on probabilities that worked well in covering a large number of the faults, but which were too crude (due to independence assumptions that turned out to be bad approximations) to focus in and cover the last 18. They tried to generate inputs one at a time. At each step in generating an input, they would specify one more bit (working from high to low order) such as to try to maximize the expected number of previously uncovered faults that the input would cover.

AGT/JFW: They started with a brute force search to find a cover. For each new input that they looked at, they saved time by only checking which previously uncovered faults it covered. They were able to get a cover with 35 inputs. By re-ordering the inputs, and feeding them back through the same routine, they found that they could get by with only **21** of them.

To cut the number down further, they decided that for the restricted set of inputs that they were now considering, they should figure out exactly which faults were covered by which tests. They were able to reduce to a solution with **14** tests, through a cycle of: (1) add new random test, (2) remove any tests that were completely redundant.

```
247 x 199   231x142   255 x 239   251 x 211   255 x 159   126 x 223
196x 173    222 x 191  182 xl83   154 x 203   162 x 51    180 x 221
90x 139     254 x 95
```

RFC/MDD/KAM: They use a method that starts with faults, and then tries to determine inputs that make those faults detectable. Through various modifications in ordering of some decisions that the method makes, they were able to generate multiple sufficient covers. Then, a greedy approach was used on the combination of covers to find this solution with **13** tests:

```
119 x 7     251 x 249   254 x 239   147 x 223   183 x 222   249 x 157
255 x 255   219 x 152   171 x 96    218 x 236   218 x 158   247 x 199
102 x 208
```

DEA/PCC/JIW: Through random sampling, they were able to cover all but ≈ 30 faults, using ≈ 20 inputs. After gaining some intuition from carefully analyzing the 4 by **4** bit multiplier, they were able to guess inputs that left them with only 7 more faults to cover. By using a program that propagated constraints backwards and forwards from gates that they wanted to cover, they covered 4 more faults, leaving 3 (correcting a bug that was later discovered allowed it to get the 2 of the 3 that were detectable). By exhaustive search, they found that one fault was uncoverable, and they found covers for the last two faults.

Their covering procedure used the heuristic: Cover first those faults that are covered by the least number of inputs (of the ones that we are considering).

Their solution (**12** inputs) was:

```
159 x 249 119 x 255 231 x 231 199 x 247 255 x 143 255 x 129
149 x 220  247 x 71   255 x 191  170 x 254 240 x 60   191 x 254
```

AG/ARG: They tried a number of different ways, each with varying degrees of success. First, they tried to compute an expression for each fault describing what inputs would detect it; this proved to be computationally infeasible. They tried to have routines that would start at uncovered faults and try to find inputs that would cover them via forward and backward propagation of constraints; this was foiled by Franz LISP stack problems. After re-writing the code with various changes (e.g. when only 5 inputs remain unspecified, then evaluate the 32 choices by brute force) they were able to get most of the faults (running it in a non-exhaustive mode). After guessing a few possible inputs, they had only 10 faults left to cover. Running their propagation program in an exhaustive but time-bounded mode covered a few more faults. After trying all of the possible 16-bit inputs (2516) with 1 to **4** zeros, they were left with one last fault. Their forcing algorithm managed to cover it, running in a time-unlimited mode. Hence, they had a sufficient cover. Using a greedy method to reduce it, they got a final answer of:

```
252 x 135 63 x 163   147 x 223 199 x 247 255 x 151 191 x 95
219 x 239 110 x 255 191 x 173 179 x 251 249 x 249
```

AJU/RMW: By exhaustively trying a bunch of inputs counting down from 255 x 255, they were able to cover most of the faults. By adding a few random inputs, and a partial cover found earlier, they found a sufficient cover. Their method of getting a small cover was to add tests to the cover one by one and see how many of the old tests they could remove. They got the following final result:

```
124 x 235 182 x 203 214 x 154 221 x 234 231 x 142 247 x 138
247 x 199 251 x 179 255 x 159 255 x 173
```

Appendix B. Gate Definitions.

```
% parallel 8-bit multiplier

module H = % full-adder: x+y+z=u+2v
 input x,y,z;
 x⊕y→a; x∧y→b; a⊕z→u; a∧z→c; b∨c→v;
 output u,v.

module R = % redundant 16-bit adder: x+y+z=a+2b
 input x[0..15],y[0..15],z[0..15];
 for i=0..15: H(x[i],y[i] ,z[i])→a[i],b[i]; endfor
 output a[0..15],0,b[0..14].

module B[k] = % multiply byte by bit and shift left k
 input x[0..7],y;
 for i=0..7: x[i]∧y→a[i]; endfor
 output 0[0..k-1],a[0..7],0[k..15].

module C[k] = % carry propagator, shifting by k
 input x[0..15],y[0..15];
 for i=0..k-1: 0→a[i]; 0→b[i]; endfor
 for i=k..15: y[i]∧x[i-k]→a[i]; y[i]∧y[i-k]→b[i]; endfor
 for i=0..15: a[i]∨x[i]→c[i]; endfor
 output c[0..15],b[0..15].

module A = % parallel 16-bit adder: x+y=g
 input x[0..15],y[0..15];
 for i=0..15: x[i]⊕y[i]→a[i]; x[i]∧y[i]→b[i]; endfor
 C[1](0,b[0..14],0,a[0..14])→c[0..31];
 C[2](c[0..31])→d[0..31];
 C[4](d[0..31])→e[0..31];
 C[8](e[0..31])→f[0..31];
 for i=0..15: a[i]⊕f[i]→g[i]; endfor
 output g[0..15].

module M = % multiply byte by byte: xy=b
 input x[0..7],y[0..7];
 for i=0..7: B[i](x[0..7],y[i])→a[i,0..15]; endfor
 for i=0..5: R(a[3i,0..15],a[3i+1,0..15],a[3i+2,0..15])→
                 a[2i+8,0..15],a[2i+9,0..15]; endfor
 A(a[18,0..15],a[19,0..15])→b[0..15];
 output b[0..15].
```

Problem 4

Distributed stability: [This problem *was suggested by E. W. Dijkstra's note, "Self-stabilizing systems in spite of distributed control," in Communications of the ACM* 17 *(1974), 643-644; see his Selected Writings on Computing [Springer, 1982), 41–46,* for the interesting first draft of this *work.]*

Consider a circular arrangement of n processors $P_0, P_1, \ldots, P_n$, *where the left neighbor of* P_i *is* P_{i-1} *and the right neighbor is* P_{i+1}; *subscripts are computed modulo n. Processor* P_i *is in one* of m_i states {O,l,. . . ,$m - 1$); *let* p_i *be the state of* P_i. *All state changes in the system are of the form* $p_i \leftarrow f_i(p_{i-1}, p_i, p_{i+1})$; *i.e., each processor can change state only as a function of its current state and the states of its neighbors. If* $f_i(p_{i-1}, p_i, p_{i+1}) \neq p_i$, *we say that* P_i *is able to "move."*

A configuration of states $p_0 p_1 \ldots p_{n-1}$ *in which* $k > 0$ *different processors are able to move can be followed by* $2^k - 1$ *different configurations, since any subset of the moves might be made. Certain configurations are said to be "good"; the others are "bad." We wish to find a definition of goodness and a sequence of functions* $f_0\ f_1 \ldots f_{n-1}$ *such that the following five properties are satisfied:*

(1) At least one processor can move, in any *configuration* $p_0 p_1 \ldots p_{n-1}$.

(2) At most one processor can move, in any good configuration.

(3) Each move *from* a good configuration leads to another good configuration.

(4) No sequence of moves takes a bad configuration into itself.

(5) Any cycle of moves includes a move by each processor.

Property (4) implies that an arbitrary configuration must lead eventually into a good one. Property (5) excludes a trivial situation where some processors "starve"; if $f_i(x, y, z) = 0$ *for all* $i > 0$ *while* $f_0(x, y, z) \neq y$, *and if the good configurations are those with* $p_1 = \ldots = p_{n-1} = 0$, *properties (1)–(5) hold but most of the processors can make at most one move.*

Here, for example, is Dijkstra's first solution, which works when all m_i *have the same value* m, *where* $m \geq n \geq 2$:

$$f_0(x, y, z) = \textbf{if } x = y \textbf{ then } y + 1 \textbf{ else } y$$
$$f_i(x, y, z) = x, \qquad \text{for } 1 \leq i < n.$$

[States are evaluated mod m.) The good configurations are those $p_0 p_1 \ldots p_{n-1}$ *in which* $p_0 = \cdots = p_{k-1} = p_k + 1 = \cdots = p_{n-1}$, *for some* $0 \leq k \leq n$. (When $m = n = 3$ the good *configurations form the cycle*

$$000 \rightarrow 100 \rightarrow 110 \rightarrow 111 \rightarrow 211 \rightarrow 221 \rightarrow 222 \rightarrow 022 \rightarrow 002 \rightarrow 000$$

and the same pattern occurs in general.) Properties (1)–(5) are *not difficult to verify, although (4) takes some thought. This construction does not work when* $m = n - 1$; *for example, when* $m = 3 = n - 1$, *there's a bad cycle*

$$0120 \rightarrow 1201 \rightarrow 2012 \rightarrow 0120$$

with all processors moving simultaneously.

Dijkstra also found a solution for $m = 3$ in which $n \geq 3$ can be arbitrarily large:

$$
\begin{aligned}
f_0(x,y,z) &= \textbf{if } y+1 = z \textbf{ then } y-1 \textbf{ else } y\\
f_i(x,y,z) &= \textbf{if } (y+1=x) \vee (y+1=z) \textbf{ then } y+1 \textbf{ else } y, \qquad \textit{for } 1 \leq i < n-1;\\
f_{n-1}(x,y,z) &= \textbf{if } x = z \textbf{ then } x+1 \textbf{ else } y.
\end{aligned}
$$

The proof in this case is more difficult [Dijkstra left it to the reader!); the good cycle when $n = 4$ begins

$$0002 \rightarrow 0001 \rightarrow 0011 \rightarrow 0111 \rightarrow 2111 \rightarrow 2211 \rightarrow 2221 \rightarrow \ldots$$

And now for Problem 4: Let M_n be the minimum value of ***the*** product $m_0 m_1 \ldots m_{n-1}$, ***the*** total number ***of*** configurations. Dijkstra ***has shown*** that $M_n \leq 3^n$; ***it can be shown*** that $M_n > 2^n$ ***when n*** > 4. Is ***it*** possible ***to prove*** that limsup $M_n^{1/n} < 3$, ***and/or*** that lim inf $M_n^{1/n} > 2$? We will ***try to find the best*** bounds ***on M_n when n is large. [Computer*** programming ***may or may not be*** helpful ***in*** this ***problem.}***

Notes for Thursday, February 28

DEK gave us a little background on the problems from which Problem 4 descended. He first came across one of these when visiting Dijkstra in the Netherlands. Dijkstra was excited about a problem having to do with distributed control. Was it possible, starting from any state, to have a mechanism whereby many machines would come to a synchronized state? He found some ways, which he published without proof. DEK was surprised that the result hadn't become more well known and that more hadn't been done with it. Recently, he had been reading Dijkstra's "Selected Writings in Computer Science". This book was partly a travelogue, and like all of Dijkstra's books, this one had no Index. Hence, DEK had to read the whole book to see what Dijkstra had said about him. (Had he made any desparaging remarks about visits to DEK's house?)

But, anyway, Dijkstra's paper had been reprinted in this book. It was from this paper that DEK made up Problem 4. Originally, the problem was going to be: Can we decrease Dijkstra's solution from having all of the processors having 3 states, to all of the processors having 2 states? DEK had mentioned the problem to RWH, the night before he handed it out. The next day, RWH reported to DEK that he could show that it wasn't possible with all 2-state processors.

Fortunately, over that same evening, DEK had used the Science Citation Index to see who had referenced Dijkstra's paper. He had found a dozen references to the paper. Only one turned out to be relevant. In his Ph.D. thesis, Maurice Tchuente showed that, if all the processors had the same number of states, you needed 3-state processors to meet all of Dijkstra's conditions. *[See RAIRO Inf. Theor.* 15 (1981), 47–66.] So, DEK generalized the problem. Now, the processors could have different numbers of states. We wanted to minimize the product of these, that is, the total number of states of the system. We were interested in the limiting values as ***n*** got large.

We would have a ring of n processors which we would write linearly as $p_0 \ldots p_{n-1}$. Also, we would have the simplifying assumption that there was a master clock. For each processor, we would have a function,

$$p'_j = f_j(p_{j-1}, p_j, p_{j+1});$$

processor P_j in state p_j was said to be able "to move", if $p_j \neq p'_j$.

ARG wanted to know if the $f_j()$'s had to be time independent.

DEK said yes. He also said that we would ignore times where nothing moved.

RFC wondered if the functions had to return single values.

DEK, again, said yes.

ARG noted that this meant that if there was ever a time when nothing moved, then everything would be stuck forever.

DEK clarified that if a processor *could* move, that didn't mean that it had to. Hence, if there was a situation where nothing could move, then the system would be stuck in that situation forever. But, if there was a time interval where some processor could move, but none of them actually did, then the processor that could move would still be able to move at the next time interval. DEK mentioned that with no loss of generality, we were going to assume that at least one processor does move at each time step.

DEK resumed his explanation of the problem. At each time interval we want a number of conditions to be true:

At least one processor can move.

In a 'good' configuration, exactly one processor can move. (Thus, if the processors are sharing a global resource, this fact can be used to control access to the resource; it would prevent interference.)

Any move from a 'good' configuration yields another 'good' configuration.

No sequence of 'bad' moves brings a 'bad' configuration into itself. (There is no cycle of 'bad' moves.)

Any cycle of moves includes a move by each processor. (No processor starves.)

Dijkstra's first solution didn't depend on the right neighbor. Also, there was only one distinguished processor. The rest of them had the same function. These functions were:

$$f_0(x, y, z) = \begin{cases} y + 1 & \text{if } x = y; \\ y & \text{otherwise.} \end{cases}$$

$$f_i(x, y, z) = x$$

This solution gave m-state per processor solutions, where m $\geq n \geq 2$.

After looking at an example we concluded that the 'good' configurations were of the form:

$$ss \ldots s(s-1) \ldots (s-1)(s-1)$$

or,

$$ss \ldots ss$$

Thursday, February 28

RMW noted that any state of the form

ss . . . so

could also be considered a ‘good’ configuration.

DEK replied that there may be more than one set of configurations that we could call the ‘good’ configurations. All that was necessary was that each one of the ‘good’ configurations lead to another. The hard part is showing that a ‘bad’ configuration can’t lead to itself. DEK mentioned that Dijkstra’s original problem statement was slightly different; in it, Dijkstra had required that you get from any ‘good’ configuration to any other. DEK had relaxed the condition for this problem.

DEK wanted to look at what would happen to this solution with n = 3 processors and only m = 2 states per processor. The ‘good cycle’ is:

$$111 \rightarrow 011 \rightarrow 001 \rightarrow 000 \rightarrow 100 \rightarrow 110 \rightarrow 111$$

But, 010 and 101 can go to many configurations, and hence they are ‘bad’. Unfortunately, they can go to each other. Hence, the ‘solution’ doesn’t work for this case. (Note that Dijkstra never claimed that it did. We are looking at it to try to understand why the solution breaks down for these values of n and m.)

DEK mentioned that this would be a solution if we could force things so that only one processor moved at a time. Then the ‘bad’ configurations would feed into ‘good’ configurations.

ARG objected that the processors couldn’t know if they were the only ones moving.

DEK concurred that if they could, it would mean that we would have already solved the problem.

DEK mentioned that Dijkstra was finally able to do it with m = 3 states for each of the n processors. The rest of today, however, we would try to disprove that a solution was possible for m = 2 states for each of the n processors. First let’s look at $n = 3$.

AG suggested using the Gray Code.

DEK explained that a “Gray code” is a permutation of the 2^n binary numbers with n bits, such that only one bit changes between any two adjacent numbers. This yields a cycle of ‘good’ configurations:

$$000 \rightarrow 001 \rightarrow 011 \rightarrow 010 \rightarrow 110 \rightarrow 111 \rightarrow 101 \rightarrow 100 \rightarrow 000$$

Since all of the configurations are declared to be ‘good’ and we can easily define the functions such that this works, this case turned out to be easy. Now, let’s look at $n = 4$ processors.

AAM objected that DEK had said that this was impossible with m = 2 states, yet we had just shown that we could trivially do it.

DEK explained that for sufficiently large n, it can’t be done; but, it can be done for a few small n’s.

DEK suggested that without loss of generality, we could assume that the first element of a cycle of ‘good’ configurations was 0000.

RFC mentioned that, similarly, without loss of generality, we could assume that the second element was 1000.

After much discussion, the class concluded that the cycle of ‘good’ configurations must be:

$$0000 \to 1000 \to 1100 \to 1110 \to$$

$$1111 \to 0111 \to 0011 \to 0001 \to 0000$$

DEK wondered about the other eight configurations. He suggested that he thought that they could be formed into another cycle of ‘good’ configurations that was parallel to and co-existed with the one that we had found. Hence, we could get by with $m = 2$ for $n = 4$. By the next class, however, DEK hoped that a student could show a proof that $m = 2$ and $n = 5$ is impossible.

Notes for Tuesday, March 5

Today, DEK wanted to start class by going through a proof of Dijkstra’s n-state solution. We already could see how to disprove proposed solutions by raising counterexamples, but we needed to see how we could prove a solution, too.

$$f_0(x,y,z) = \begin{cases} y+1 & \text{if } x = y; \\ y & \text{otherwise.} \end{cases}$$

$$f_i(x,y,z) = x$$

Thus we could see the good cycles, which looked like:

$$444433 \to 444443 \to 444444 \to 544444 \to \ldots$$

But besides knowing that we will stay in a good cycle once we enter one, we must show that we will always enter one.

RFC suggested that we could look at the differences between adjacent processors. These would look like:

$$000010 \to 000001 \to 000000 \to 100000 \to .*=$$

So, maybe we could re-write the $f()$’s in terms of the differences between adjacent processors.

DEK agreed that he liked the idea of looking at the differences, but he didn’t really see how to use them for our current purpose.

$$q_i = p_{i-1} - p_i$$
$$f(p_l, p_m, p_n) = p'$$
$$g(q_l, q_m, q_n) = q'$$

But, DEK doubted that there was enough information available local to q_m to adequately define the $g()$’s. DEK looked at 450123 → 445013. It seemed like this approach would take too long to iron out, since it seemed that the behavior of the q_i’s was not local enough.

MDD suggested that we try to keep track of the number of different numbers that were active at the processor's, and show that it goes down to 2.

RMW pointed out that this wasn't strictly true for the good cycles, since occasionally the number goes down to 1 and then back up to 2.

DEK suggested that one good way to prove something true is to try to prove that it is false. In other words, try to find a counter-example. Either we would succeed in the disproof (so, Dijkstra made a mistake – we all do), or we would get stuck. In getting stuck, we would probably notice something useful. DEK threw out two questions to guide the disproof: (1) Can we have an infinite bad sequence where all the processors move? (2) Can we have an infinite sequence where p_0 never moves?

RMW said that the second one was impossible.

DEA said that this was because the other processors could just copy the value of the processor to its left. Eventually everyone will have the value that p_0 starts out with.

DEK wondered if we could show this formally.

RMW thought that we could find a recurrence relation to show that all the processors would have the value of p_0 by time t.

DEK wondered how long we could keep it going.

JFW suggested that it would happen by $t = \frac{n^2+n}{2}$.

RMW thought that that was about right.

Doing the recurrence the lazy way gave a bound of $t = 2^n$. Surely we could do better. DEK suggested putting tick marks between adjacent processors that had different values.

$$0 \;{}_{\wedge}\, x_1 \;{}_{\wedge}\, x_2 \;{}_{\wedge}\, x_3 \;{}_{\wedge}\, x_4$$

JFW noted that any tick mark can only move to the right. Sometimes a tick mark can disappear or more than one tick mark can move at a time, but the worst case is for one tick mark to move to the right at each time step.

DEK numbered the tick marks for easier identification.

$$0 \underset{0}{{}_{\wedge}} x_1 \underset{1}{{}_{\wedge}} x_2 \underset{2}{{}_{\wedge}} x_3 \underset{3}{{}_{\wedge}} x_4$$

So the maximum number of time steps is $1 + 2 + 3 + 4 + \ldots = O(n^2)$. We now knew for sure that if there was an infinite cycle of bad processors, it must include moves by p_0.

RMW pointed out that the sequence:

$$14321 \rightarrow 21432 \rightarrow 32143 \rightarrow 43214 \rightarrow 04321$$

shows that the number of distinct numbers can increase between two bad configurations; this ruled out the hope of an easy proof based on the number of distinct numbers.

DEK continued that since we know that p_0 must move in our hypothesized infinite sequence of bad configurations, we know that at some time $p_0 = p_{n-1}$. So there must, at some point, be the sequence:

$$ax_1 \ldots x_{n-1} \rightarrow\rightarrow ay_1 \ldots y_{n-2}a \rightarrow (a \;\; 1)y'_1 \ldots y'_{n-1}$$

RMW pointed out that either all the y_i''s were equal to a, or a $\in \{x_1, \ldots, x_{n-1}\}$.

AAM agreed that this must be the case, since the a at the processor on the right, must have originated from somewhere on the left.

DEK re-emphasized that assuming we could construct an infinite cycle of bad processors, we know that a $\in \{x_1, \ldots, x_{n-1}\}$. We still need at least one more observation to make this all work.

JFW pointed out that the process would continue:

$$(a+1)y_1' \ldots y_{n-1}' \rightarrow\rightarrow (a+1)y_1'' \ldots y_{n-2}''(a+1)$$

DEK pointed out that from our previous reasoning we thus knew that (a + 1) $\in \{y_1', \ldots, y_{n-1}'\}$.

JFW saw that this implied that (a + 1) $\in \{x_1, \ldots, x_{n-1}\}$. Continuing this line of reasoning, we would need every number from 0 to $n-1$ to be in $\{x_1, \ldots, x_{n-1}\}$.

DEK summed up that, since this was impossible, we were done showing that there was no infinite sequence of bad configurations. Since one of the n states was missing from $\{x_1, \ldots, x_{n-1}\}$, eventually p_0 would have to go move from that state into the next higher state. At that point in time, though, all the processors would have to be at the same state – that is, we would have to be in a good configuration. [Side note: since p_0 would have to move at least once every $\binom{n}{2}$ moves, and since if p_0 moves n times then we are in a good configuration, we can get a bound that we will be in a good configuration after at most roughly $n\binom{n}{2}$ moves; this number can surely be lowered. So, what is the tightest bound we can get for it?]

DEK pointed out that Dijkstra had another method with 3 states for each processor. Did anyone have a good intuition as to why that one worked? The main problem with these distributed systems is that, in a bad configuration, locally it can look like two different processors can move. Dijkstra gets by this in the case with n states by essentially having a "filter" at one point.

RFC said that in the S-state case, looking at the $p_i - p_{i-1}$'s works.

ARG mentioned that it is always easy to show that once you are in a cycle of good configurations, you stay there. The hard part is showing that you never get stuck in a cycle of bad configurations.

DEK agreed with this. He pointed out that he didn't think that the 3-state solution had ever been proved in print.

$$0001 \rightarrow 0011 \rightarrow 0111 \rightarrow 2111 \rightarrow 2211 \rightarrow 2221 \rightarrow 2220 \rightarrow 2200 \rightarrow$$

He wondered how we could show that the bad configurations go to good configurations. He mentioned that RWH -had described it roughly this way: put in the ticks between processors with unequal states (except p_0 and p_{n-1}); note that these ticks move in waves back and forth from the left end to the right and back to the left; in a bad configuration, you have multiple ticks; if there is more than one tick, the waves will eventually collide, either merging into one, or cancelling each other out.

Can we beat 3^n states? If we do, we will need to stick in at least one **2-state** processor. Note that replacing two 3-state processors by one 2-state and one **4-state** processor, reduces the total number of states.

Thursday, March 7

DEK wondered if in the last few minutes of class we could show that you can't solve this problem with only **2-state** machines. We started with assumed good configurations

$$x_1x_2x_3x_4x_5 \rightarrow \bar{x}_1x_2x_3x_4x_5$$

ARG pointed out that only p_2 or p_5 could move next. Without loss of generality:

$$x_1x_2x_3x_4x_5 \rightarrow \bar{x}_1x_2x_3x_4x_5 \rightarrow \bar{x}_1\bar{x}_2x_3x_4x_5$$

DEK agreed that, in a 2-state system, we would never want the same processor to move twice in a row.

AGT mentioned that if we now allowed x_1 to move, we would be in the configuration $x_1\bar{x}_2x_3x_4x_5$. Because of previous decisions, we could only move p_2, but this would bring us back to our starting state.

DEK said that we could generalize this to: if p_i moved followed by p_{i+1}, then this must be followed by p_{i+2}; or, if p_i moved followed by p_{i-1}, then this must be followed by p_{i-2}. Once the movement started in a certain direction, then it had to stay in that direction. (Remember, we are only talking about the all 2-state case.)

So, if we had a configuration where three different places could move, we would have three waves that just kept chasing each other.

DEK said that people were free to use computers if they could help, but sometimes in Computer Science you just use pencils to solve problems.

Notes for Thursday, March 7

DEK pointed out that in our proof of Dijkstra's n-state solution in the previous class, we had proved only that there was no infinite sequence of different bad configurations. For our proof to have been complete, we also needed to show that there were no configurations in which no processors could move. For this case it is obvious, so we won't prove it, but for other cases it might not be so obvious.

Now DEK wanted to see if there was a solution with five **2-state** processors. We had proved the previous day that for (sufficiently many) **2-state** processors, the movement of processors for any sequence of good states must march around in the same direction. That is, our good cycle must look like:

$$x_0x_1x_2x_3x_4 \rightarrow \bar{x}_0x_1x_2x_3x_4 \rightarrow \bar{x}_0\bar{x}_1x_2x_3x_4 \rightarrow \bar{x}_0\bar{x}_1\bar{x}_2x_3x_4 \rightarrow \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3x_4 \rightarrow$$
$$\bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \rightarrow x_0\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \rightarrow x_0x_1\bar{x}_2\bar{x}_3\bar{x}_4 \rightarrow x_0x_1x_2\bar{x}_3\bar{x}_4 \rightarrow x_0x_1x_2x_3\bar{x}_4 \rightarrow$$

This good cycle forces many values for our f_i ()'s

DEA now exhibited an example of a bad cycle:

$$\frac{x_0}{?}\,\frac{\bar{x}_1}{Y}\,\frac{\bar{x}_2}{N}\,\frac{\bar{x}_3}{N}\,\frac{x_4}{?}\ \mathrm{j}\quad \frac{x_0}{Y}\,\frac{x_1}{N}\,\frac{\bar{x}_2}{Y}\,\frac{\bar{x}_3}{N}\,\frac{x_4}{?}\ \mathrm{j}\quad \frac{x_0}{Y}\,\frac{x_1}{N}\,\frac{x_2}{N}\,\frac{\bar{x}_3}{?}\,\frac{x_4}{?}\ \mathrm{j}$$
$$\frac{\bar{x}_0}{N}\,\frac{x_1}{Y}\,\frac{x_2}{N}\,\frac{\bar{x}_3}{?}\,\frac{x_4}{Y} \rightarrow \frac{\bar{x}_0}{?}\,\frac{x_1}{Y}\,\frac{\mathrm{x2}}{N}\,\frac{\bar{x}_3}{Y}\,\frac{\bar{x}_4}{N} \rightarrow \frac{\bar{x}_0}{?}\,\frac{x_1}{Y}\,\frac{x_2}{N}\,\frac{x_3}{N}\,\frac{\bar{x}_4}{?} \rightarrow$$

This completes the proof that five 2-state processors cannot achieve stability.

DEK pointed out that if every processor able to move was forced to move, then these counter-examples would be easier to find. As it was, this was like doing the word puzzles: change 'SOBER' to 'DRUNK'.

RFC said that with a slight variation, you could show that any larger circuit can't have four **2-state** processors in a row.

DEK mentioned that when he had posed the problem he had thought that you might be able to pump stuff from the right and from the left, through a 2-state channel. But, now it looks like we can prove a general result that if the system has three 2-state processors, then there must be a **2-state** processor through which the movement is uni-directional.

Let's take a closer look at this situation. Let us represent the states of our **2-state** processors by: a_i, b_i, and c_i. We will use α_i, β_i, and γ_i to represent strings of 3-state processors. We can start with assumed good configurations

$$a_0\alpha_0 b_0\beta_0 c_0\gamma_0 \rightarrow \bar{a}_0\alpha_0 b_0\beta_0 c_0\gamma_0 \rightarrow$$

We will assume that the baton goes to the right. It is possible for a to move again before b does. But from this, we can see that γ can't move now for either a = 0 or a = 1. So the baton cannot be passed to the left by a. (At this point DEK mentioned that many of his European friends are horrified by our anthropomorphism of computers.) So, if the baton can't go the left now, then:

$$a_0\alpha_0 b_0\beta_0 c_0\gamma_0 \rightarrow \bar{a}_0\alpha_0 b_0\beta_0 c_0\gamma_0 \rightarrow\rightarrow a_1\alpha_1 b_0\beta_0 c_0\gamma_0 \rightarrow a_1\alpha_1 \bar{b}_0\beta_0 c_0\gamma_0$$

MDD pointed out that if the baton couldn't go to the left for a, then it can can never go to the left through any 2-state processor; specifically, it can't go to the left for b or c.

Continuing the sequence gives:

$$\begin{aligned}&a_1\alpha_1 b_0\beta_0 c_0\gamma_0 \rightarrow a_1\alpha_1 \bar{b}_0\beta_0 c_0\gamma_0 \rightarrow\rightarrow a_1\alpha_1 b_1\beta_1 c_0\gamma_0 \rightarrow a_1\alpha_1 b_1\beta_1 \bar{c}_0\gamma_0 \rightarrow\rightarrow\\ &a_1\alpha_1 b_1\beta_1 c_1\gamma_1 \rightarrow \bar{a}_1\alpha_1 b_1\beta_1 c_1\gamma_1 \rightarrow\rightarrow\rightarrow a_i\alpha_i b_i\beta_i c_i\gamma_i \rightarrow\rightarrow\rightarrow\end{aligned}$$

Eventually it would cycle.

To see that there might be cases where $a_i \neq \bar{a}_{i-1}$, DEK showed that with the sequence of the number of states in the processors being $[2,4,2,4,2,4]$, a good cycle was constructible where the baton would go between processors in the sequence

$$(0,1,0,1,2,3,2,3,4,5,4,5).$$

ARG didn't think that we would be able to prove what we wanted this way, since we knew that there was a solution with four 2-state processors.

DEK agreed that we would probably need to assume that α, β, and γ were non-empty. He added that while the $[2,4,2,4,2,4]$ system that he mentioned above had two good cycles, he had written a program to search for bad cycles; it had a bad cycle of length 32.

MDD (who had worked with RFC/ARG on this problem) said that they had also come to the conclusion that a 2-state processor could either reflect the baton, or else the baton could go across the 2-state processor in one direction – but then, the baton could never come back in the opposite direction.

DEK mentioned that Dijkstra had a $[2,4,4, \ldots ,4,2]$ solution where the two processors on the ends don't look at each other. Hence, this was actually a straight line solution as opposed to a circular solution. It might be possible to show that a straight line solution needs many **4-state** processors.

AAM said that if we had $[\ldots, p_0, 2, 2, p_1, \ldots]$ then information would be lost in passing the baton from p_0 to p_1 through the **2-state** processors.

DEK explained another approach that he had tried. One way to view the problem is to say that you want to be able to get rid of multiple waves. So, you need a structure of processors that filters k-**waves** to $\lceil \frac{k}{2} \rceil$-waves on every lap. This proves to be hard, though it is easy to get a filter that goes to $\lfloor \frac{k}{2} \rfloor$-waves; unfortunately, such a system goes dead – in the steady-state no processors can move.

DEK pointed out that one of the great things about research is that you don't need to stick to the original problem. There are many interesting sub-problems that can be worked on. For example: (1) How many states do you need in a system where the baton is always passed to the right? Is Dijkstra's n-state solution optimal? (2) What is the best that we can do for 5 processors?

There are many different properties between states to look at. For example, each state has 1, 3, 7, 15, . . . successor states. Or, you can look at the $a_0\alpha_0b_0\beta_0c_0\gamma_0$ type stuff.

RFC suggested finding all the places where there can be shifts such as:

$$\begin{aligned} a_1\alpha_1b_k\beta_kc_k\gamma_k &\to a_1\alpha_1\bar{b}_k\beta_kc_k\gamma_k \\ \text{or } &\to \bar{a}_1\alpha_1b_k\beta_kc_k\gamma_k \\ \text{or } &\to \bar{a}_1\alpha_1\bar{b}_k\beta_kc_k\gamma_k \end{aligned}$$

MDD listed some of the conditions that must be satisfied here:

$$\begin{aligned} a_1 &= a_k \\ \text{for their rightmost states } \gamma_k &= \gamma_1 \\ b_k &= b_0 \\ \text{for their rightmost states } \alpha_1 &= \alpha_{k+1} \end{aligned}$$

RFC said that they had tried to do a disproof for systems with four 2-state processors, with other processors between them. But there was no apparent way to force a counterexample.

DEK said that there -have been times when a CS204 problem wasn't solved until a while after the course was over.

Notes for Tuesday, March 12

Today was the last day on problem 4. DEK asked if anyone had been able to solve this problem. No one had. DEK smiled. He had finally stumped some of the best minds in the world. Well, did anyone have any partial results?

KAM thought that ARG might, but ARG was one of the five students absent from class today (shame! shame!).

JFW said that he thought that someone had discovered a bug in what ARG had done, anyway.

DEK mentioned that at the end of today's class, the students would have to fill out course evaluations. Since no one had any results, this class might not be one of the more interesting ones. DEK hoped that students would remember some of the other, more interesting classes when filling out the course evaluations.

It seemed like no one had been able to get any partial results. DEK exclaimed that this class was better at deciding when to quit on a problem than he was. At least five times, he had worked on real hard problems that he finally gave up on; three of those times, he had the right idea about how to solve them within an hour after giving up. Once, he had thought that he had a proof that $P = NP$ in four pages; he found a terrible oversight. Since then, he has had to come to grips with the fact that he'll probably never solve that problem.

Surely, there must be more powerful results than what we had last class. Then, we believed that having three **2-state** processors would prevent the system from working.

DEK noted that starting from any arbitrary configuration, any possible sequence must lead to a good cycle. Hence we know that, in particular, if we always move the leftmost processor that can move, we will eventually end up in a good cycle. We will use $\alpha^{(0,1)}$ in

$$0\alpha 1 \rightarrow\rightarrow 0\alpha^{(0,1)}1$$

to mean that we have exhausted all of the leftmost moves of the block of processors a, given that a has left neighbor 0, and right neighbor 1.

DEK suggested that we try working on a simpler problem. If we can't figure out the simpler one, then we can forget the harder one. If we can get the simpler one, then we will have some idea as to what will and what won't work on the harder one. Let's look at the case with two 2-state processors. A good cycle will look like:

$$\begin{aligned} 0\alpha_0 1\beta_{n-1} \rightarrow 0\alpha_0 0\beta_{n-1} \rightarrow\rightarrow 0\alpha_0 0\beta_0 \rightarrow 1\alpha_0 0\beta_0 \rightarrow\rightarrow \\ 1\alpha_1 0\beta_0 \rightarrow 1\alpha_1 1\beta_0 \rightarrow\rightarrow 1\alpha_1 1\beta_1 \rightarrow 0\alpha_1 1\beta_1 \rightarrow\rightarrow \\ 0\alpha_2 1\beta_1 \rightarrow 0\alpha_2 0\beta_1 \rightarrow\rightarrow 0\alpha_2 0\beta_2 \rightarrow 1\alpha_2 0\beta_2 \rightarrow\rightarrow \end{aligned}$$

Note that, according to our previous definition:

$$\alpha_1 = \alpha_0^{(1,0)}$$

$$\beta_1 = \beta_0^{(1,1)}$$

Also, we can see that α can't move in the situations

$$0\alpha_{2k}0,\ 0\alpha_{2k}1,\ 1\alpha_{2k+1}0,\ 1\alpha_{2k+1}1$$

The following must hold:

$$1\alpha_{2k}0 \rightarrow\rightarrow 1\alpha_{2k+1}0$$

Tuesday, March 12

$$0\alpha_{2k+1}1 \to\to 0\alpha_{2k+2}1$$

From our good cycle, what happens to the following configurations are not determined:

$$1\alpha_{2k}1,\ 0\alpha_{2k+1}0$$

JIW pointed out that we already knew that if α and β were each single 2-state processors, then we already knew that we could make this work.

DEK pointed out that maybe we could still get some result which says, for instance, that the period, p, must be short, say 4 or 8. Clearly p is even.

RFC agreed that p must be even, since we already had forced α to be dead in configurations $1\alpha_{2k+1}0$ and for it to be able to move in configurations $1\alpha_{2k}0$.

DEK said that this was a small theorem, but that at this point, we'd take anything. He pointed out that the rules for the β's where the same except that they had the rightmost bit complemented.

DEK was about to continue. He had been presenting an analysis that he had done on paper before class. But, he noticed a mistake in his previous work. It wasn't immediately apparent to him how to fix it. No one in the class had any ideas, either.

JIW (who had worked with DEA and ARG) reported that they had tried to prove that no processor could swallow a baton. Hence, they had a conjecture that you never lose an active wave.

AAM asked if anyone had tried to construct a solution of size 5.

That suggested to DEK that maybe we should talk about how we could use a computer to get data on these things. Let's assume that we are in a programming contest, and are given the problem: Determine whether a certain system satisfies the constraints of Problem 4.

He ran through some of the things that we knew had to be true about the system. (1) Every configuration has to have at least one successor. (2) Every good configuration has exactly 1 successor, and that successor is a good configuration. (3) There is no cycle of moves that takes a bad configuration to itself. (4) In every cycle, each processor must move at least once.

(1) could be checked very easily, and (4) could be checked very easily once we knew what the cycles were. But how do we know if there is a possible labelling of configurations as good and bad, which satisfies (2) and (3)?

JFW suggested that we compute the transitive closure of the configurations. Each configuration in a cycle would have to be labelled 'good', so if any of them were of degree > 1, then this set of $f_i()$'s was not a valid system.

DEK said that we should let x → y mean that y is a successor configuration of x, and x $\to^+$ y mean that y is a successor of x under transitive closure. Then, we could just re-state what we were saying as: if x $\to^+$ x, then x had better be a 'good' configuration. If we call the rest 'bad', then we have all of our conditions satisfied. How fast is it?

JFW offered $n^2 \log n$ or n^3.

DEK said that $\frac{n^3}{\log n}$ was the bound at one time. He mentioned that the time for transitive closure was the same as that for matrix multiplication. So we could do it in $n^{2.49}$ (though no one would really use the $O(n^{2.49})$ algorithm for any reasonably sized problem).

RFC noted that *n* was the number of configurations for the system. Couldn't we do it without enumerating all of the configurations?

AAM suggested a different method for labelling the configurations as 'good' and 'bad'. Find the **configurations** of outdegree > 1; they had to be 'bad'. Find all configurations that can lead to the 'bad' ones and label these 'bad'. The rest should be **labelled** 'good'.

DEK noted that (2) was true. How would we check **(3)?**

AAM continued by suggesting that we do searches starting from 'bad' configurations, keeping track of where we get.

DEK asked how we know if the 'bad's were loop free.

AAM said that we would use the search to check if the 'bad' configurations just formed trees.

RFC noted that this could be done by depth-first search.

DEK thought of another way. We could just iteratively keep removing from our graph any 'bad' source vertices (those with no predecessor). Eventually we would get stuck; if any 'bad' **verticies** remained, then we would know that there was a cycle of 'bad' configurations. This was only O(n), and hence clearly better than the method that required transitive closures.

RFC asked again whether it would be possible to avoid expanding out all of the configurations, since *n* grows exponentially with the number of processors.

DEK thought that it might be possible to do symbolically. As a last note, DEK mentioned that people should bring in any interesting mistakes from Problem **5,** since . graphics mistakes can often be just as interesting as the final product.

Problem 4, Review of Solutions

Even though no one was able to solve the problem, the students were asked to turn in any partial solutions that they had. ARG (working with DEA and **AAM)** was the only one to turn anything in.

ARG made the conjecture that there were no working schemes with more than a constant number of **2-state** processors (as *n* gets large).

ARG gives an argument for the proposition that: in any valid scheme with 2-state processors, every good cycle is ***quasi-unidirectional*** (once the baton is passed through one **2-state** processor, it doesn't pass through the previous 2-state processor, again, until it has gone through all of the other 2-state processors). The main argument holds for three or -more **2-state** processors; the cases for either one or two 2-state processors are trivially true.

Finally, he was able to show that: in any valid scheme allowing 2N non-adjacent 2-state processors, the least common multiple, ***L*,** of the numbers of states in each of the blocks intervening between the 2-state processors must be at least N + 1. Otherwise, it is easy to construct a cycle of bad configurations that look good locally to each of the processors that can move, and, hence, is infinite.

Thursday, January 31

Problem 5

High-tech art: Here is your chance to make a contribution to fine art, by means of the art of programming: Produce a machine-aided self protrait.

You may use any technique or medium that you like, provided only that you use a computer in some essential and preferably innovative way; i.e., you should create something that would be humanly impossible without the help of a computing machine that you have programmed for this problem. (Programming should be a key ingredient; merely using some existing software like “MacPaint” to prepare a sketch is not the idea.)

Note: Problem 5 is different from the other four in that all students are expected to work individually; after all, self-portraits are very personal things. Prof. Matt Kahn of the Art department will lead a special presentation about how to express yourself artistically, during our class session on January 31; then on February 14 we will take time out from Problem 3 to discuss special equipment that exists at Stanford for inputting or outputting images to or from computers. We'll scour the campus for possibilities.

The portraits will all be exhibited and discussed during our final class session, on March 14. You should begin thinking now about what sort of self-portrait you have subconsciously been wishing that you had time to create; and you should find time to create it, while working on the other four problems!

Notes for Thursday, January 31

Mentioning that one of the great things about working at a university like Stanford was the opportunity for interdisciplinary cooperation, DEK introduced our guest speaker, Prof. Matt Kahn (MK) of the Art Department.

MK, noting that he had a different background from the students, hoped that his presentation would be helpful. He hoped that he could visually define what a ‘portrayal’ was. His goal was to stimulate, not to inform. If ‘good teaching’ was showing people how to do things, this class would fall under ‘bad teaching’.

[During the slide show, the notes are a little sparodic due to the fact that all of the lights were turned off.]

MK noted that art was older than farming. The tradition of visible communication is old, extending back to the times of cave dwellers.

Looking at art from different times, one can see that the available techniques influenced the content. Art is something that some of us spend our whole lives doing. It is not possible, MK pointed out, to just waltz in, do a great work of art, and then waltz out. Technology gives us the impression that this is possible. But, no matter how sophisticated the technique, the effect will only be as sophisticated as the user.

MK mentioned that photography was a means by which we can accurately represent faces. It can still be an art form, though. This is because the selection of the content, angle, framing and lighting of a photograph has a great effect on the final work.

He quoted the saying that, “the plaster cast is the same as the original, except in everything.”

MK pointed out that nature is magnificent; in showing it, light becomes important. In showing change, the context is important. It allows the portrayal of feeling, not just the portrayal of the physical. It expresses a point of view.

MK mentioned that all people go to great lengths to portray and project a self-image. They do this in their costumes, hairstyles, makeup. We are more conscious of this in cultures other than our own. But, we too bring various props to bear, just as an actor or a clown uses make-up.

In sculpture, distortions of the face are used to symbolize extensions of our selves. The handling of heads takes infinite forms. In art you should express what you want to say. Editorial options are editorial obligations. All art is self-portrative; it is a vehicle for statements about our selves.

With the slide show over, DEK mentioned that he hoped **MK's** points would be useful to us.

KES said that it is hard for non-artists to understand by themselves some of the points that MK mentioned, so it is good that MK brought them up. She found his comments, about the effect of technology on technique, interesting. The greatest challenge is to combine them.

MK agreed that yes, this was part of what he was trying to convey when he brought up the use of cameras. We make our statements through our mechanical tools. When your tool is new and exciting, you will show enthusiasm for it.

AG said that it would be difficult for us to capture "our whole essence," since we were unfamiliar with the methods for doing so.

What MK had meant was that with a new method or a new concept, you should notice the scope of what you are dabbling in. He wanted us to know that there was a possible objective that we could try to achieve with it. He expressed the belief that somebody in this class will come close. Caring a lot is the first step.

AAM, noting that most of the slides that we saw were in color, wondered if artists did much work in black and white.

MK pointed that many artists do indeed work in black and white. There is a lot of drawing, print making, etching and black and white photography (though the last of these has tonal variations). MM said that every new technology allows new forms of expression. If a statement has validity then making it is what is important.

AG asked if we should restrict ourselves to a single portrait or if we could use moving pictures.

MK pointed out that the use of movement allows you to express rhythm. He recommended that we don't talk to each other about our ideas until we were all done. Then we could be surprised by the originality of each other's ideas. He didn't want to set parameters on what form our projects could take. He asked DEK whether color was an option.

DEK said that the equipment at our disposal doesn't have color available. He mentioned that for work in film, having color makes the job easier. For us, however, we have to specify how everything is to be controlled; for us, color would make the job harder.

MK mentioned that color frequently seduces the artist away from making a statement that has any meaning. He warned us not to get so seduced by the technology at our disposal that the final work wouldn't be a reflection of our selves, personally. Most art is very gratifying .

KES noted MK's point that we should not try to conceal the nature of our medium. She complained that she didn't want to use just black and white, as this wouldn't allow her to fully express her self image. She was going to have to figure out how to overcome the limitations of the medium.

MK mentioned that paint is the most used medium because it is so versatile. He expressed the hope that the medium that we are working with is versatile enough for us.

DEK said that the final result did not have to be totally produced by computer; it could be mixed with human work done by hand. The requirement was that computer programming be involved. First, you would have to decide what type of statement you wanted to make; then, you could decide to what extent you wanted to make the computer your entire medium.

MK said that this brought to his mind an image of a 4 year old child drawing with crayons on a TV screen as the picture was changing. He asked if people have any ideas of what they wanted to do. He was glad that this project wasn't being done in teams; this would have diluted our ingenuity and inventiveness.

DEK brought up the entrepeneurial aspect – you could possibly build a tool that other people would be able to do portrayals on. MK frowned at the idea that the noble goal of expressing oneself through art might be corrupted by the base profit motive.

DEK mentioned that at NYU ten years ago, programming was taught to some artists. This lead to the development of the Artspeak language; the artists could then use this language to communicate with the computer, thus enabling them to express their ideas in this strange (to them) new medium.

AAM thought that sometimes "Modern Art" was being inventive just to be strange.

MK responded that while there is some that he would object to because it was not being done by serious artists, he was heartened by it when done by serious artists. Too many artist do recipe-book art, and it is important for artists to investigate and explore.

DEK was bothered by critics who were negative about the work of Leonardo Da Vinci. The critics, themselves, weren't as good as Leonardo; who were they to be critical?

MK said this happens because all good art challenges the person who experiences it.

AG said that criticism seemed to be an interpretational art in itself.

MK agreed that was the case in such things as whole books devoted to analysis. But, the obligation of a newspaper reviewer is entirely different.

AAM said that it seemed to him that when something gets interpreted too much, it seems like people are reading meaning into the work that isn't there.

MK disagreed. He has never feared that he is seeing more than the artist intended.

KES thought that what the art work means ***to*** you was more important than what the artist may or may not have meant.

MJB wondered if an artist sometimes doesn't know what he is expressing until after he has completed his expression of the idea.

MK agreed that there is a balance between the anticipated and the unanticipated. He, himself, doesn't do sketches of a painting before doing the painting itself; he doesn't want to ruin the spontaneity.

KES noted that there were parallels in this to what happened in solving Problem 1. If you think of all the ideas first, and then go out to do the programming and write-up, the fun of the last part is destroyed.

MK ended by saying that the process of doing art was itself important. He cautioned, however, that we not make the process more important than the product. Finally, he noted that the process should be reflected in the product.

Notes for Thursday, February 14

DEK introduced Susan Brenner, henceforth SB, who was going to give us a talk on the human face.

SB said that this talk had many goals. She was going to talk about caricatures. Even though caricatures use few lines, most people agree that they are a better representation of a face than the face itself. This is because caricatures involve a semantic bandwidth compression. Even though they involve a distortion of reality, humans can still recognize who the person is.

In her work, SB was not trying to model the whole process of caricature as done by humans. Rather, she was just trying to capture the main visualisation process. SB was going to describe work that she had been done as a Computer Graphics Thesis at MIT – part of DARPA funded work on tele-conferencing.

Much psychological work has been done on human recognition of faces. Humans can recognize faces well. They also interpret many abstract things as faces. However, they find it hard to recognize faces from upside-down pictures or from negatives of pictures, even though all of the same information is there. Other research has shown that the presence of a strong feature, such as a large scar, can mask other features, thus interfering with the recognition process. It also turns out that people will, at first, have difficulty distinguishing members of a new race; this ability, though, is improved over time. People are adept at separating out the essential features of a face regardless of angles or other complications.

Compared to these human abilities, machine vision is not even close.

Computer systems for matching faces by computer have been motivated by two different applications. One, is to have a computer *verify* that a person is who he claims to be; this clearly has many uses in security systems. The other is to be able to give a computer a picture of someone's face, and have the computer *determine* who the person is from a large data-base of people; this would be useful for law-enforcement.

For example, one system by Bledsoe had human operators identify 16 different features on someone's face. He stored the distances between the features. A system by Kelly did feature identification automatically. These techniques involved numerical measurement of a face, but not regeneration of the face.

SB pointed out that a caricature had to be of a particular person, not of a stereotype of a class of people. She mentioned that portrait artists, animators, and cartoonists all had various heuristics for drawing faces. Da Vinci tried to catalog all the types of noses that people have. Dürer drew a coordinate grid over an ideal face, and then applied transformations to it.

Caricaturists prefer to work from a file of photographs of the same person. But, SB found in trying to do caricatures, you shouldn't try to draw while looking at the face; this results in a drawing with too many details. Looking also inhibits the visualization process.

One approach to computer caricaturing would be to start with a **collaging** program. You would load a trace of a picture and then exaggerate various features, such as lengthening the face or widening the eyes. However, this resulted in a method that was piece-meal and iterative. SB wanted something better.

She went to the idea of having a line representation which would then be subjected to more and more distortion. In doing this she came across the hypothesis that a caricturist selects unique features and amplifies them. But this raised the questions: what characterizes a face? what should be measured?

Nixon, for example, has four features that are always enhanced in his caricatures: his nose, his hairline, his forehead, and his jowls. Leaving out a key feature such as one of these four doesn't hinder recognition, but replacing it with a contradictory feature does make recognition harder. The features that various artists chose to distort were the same, but the degree of distortion was different.

The features to distort depend on the context, the beholder, and the face. SB quoted Gombrich as saying something like, "It is not really the perception of likeness for which we are originally programmed, but rather, unlikeness; the departure from the norm is what sticks out in our mind."

Her technique starts with a digitized image. Then she gets a line drawing of the face. To get a line drawing, SB tried filtering the image using a Laplacian and using a histogram-based technique due to Kanade, but the resolution wasn't good enough. She therefore indicates points on the face, which the computer then draws a spline through.

Originally, she thought that she would have to do the distortion with respect to an 'ideal' face. But, given any two faces, her method can distort one face with respect to the other. This is done by computing the distances between features on the two faces. Then, a new face can be constructed by exaggerating the differences between the two faces. A similar approach can be applied to generic lip positions. Then, by controlling an enunciation coefficient one can make the caricature 'speak'; this can be done by doing a real-time correlation of the lips and the sound.

DEK said that one of the main purposes of today's class was to remind people that there are only **28** days left until this project is due. He also mentioned that one of the purposes of this particular project was to show the world that computer scientists aren't entirely nerds. DEK mentioned that the use of a computer allows people who lack the manual skills, that most artists need, to still make art.

He mentioned that once you had an image on the computer, there were many things that you could do to it. Computers were good at distortion and repetition, among other things.

There are several books on computers and art. DEK mentioned that one way to "be creative" on demand was to stimulate yourself by looking at books or magazines that might jog your imagination.

DEK showed us some examples of using computers in the production of art work. These were things he had done when taking a course taught by MK.

The first one was an example were the computer had generated the instructions about how to draw the final product, which was a study of how red and blue combined to form purple. There was a rectangle of evenly spaced red and blue dots. The computer had been used to make the pattern of colors regular in some places and random in others. DEK had expected that the random areas were where we would see purple, while in the regular areas, we would see the blue and red as separate. He found out when he produced the piece, that he had it exactly backwards; the regular pattern appeared purple.

The second piece had a pattern that was repeated at places that were selected by a computer program.

The third was a piece showing Bach. Contoured to the shape of his face, was a musical score that Bach had written. DEK said that he had found that many people liked a grid of the shape of Bach's head, which he had produced as an intermediate step, better than they liked the final product. The grid had been produced by first inputing three-dimensional coordinates using a sensing device with pins. Then, DEK had interpolated other points. Finally, he used a crude hidden-line removal algorithm. The hidden-line removal did not draw the perimeter of any polygonal face of the 3-D object which had its normal going into the paper; this would have worked perfectly, if the object had been convex.

The fourth work was some three-color needlepoint. The computer had been used to generate the color patterns from photographs of **DEK's** family. The algorithm to reduce the pictures down to three colors was based on a method by Floyd and Steinberg.

Now it was David Kriegman's (DK) turn. He was going to tell us how to get our images into a computer. He showed us a camera that was capable of getting an image that was 256 by 256 pixels, and had 256 levels per pixel. He asked us to try not to clog up · COYOTE with the resulting 64K byte files.

The system is turned on by three 'on' switches. It runs on a stripped down Sun system. A handout describes how to use it. The account we were going to be using was **cs204,** and the program name was 204. He asked the class to try to get any images that they might need in the next couple of weeks, since he couldn't guarantee that the equipment would be available/operational for much longer than that.

AG asked if you could take pictures of photographs with the camera.

DK said that people had gotten good results taking pictures of photographic slides. You might want to change lenses, though; there are a bunch of different lenses available.

DK needed a volunteer to demonstrate the equipment on. SB was kind enough to volunteer RWH for this joyous task. The format of the files has dimensions of the picture (256 by 256) in ASCII on the the first line. Then the files have **64K** bytes, with one byte per pixel in the image. When **FTPing** these files, DK said that you should use binary mode, so that you don't risk losing the high order bit.

He mentioned that they also had other software available to do filtering and edge detection. It could be loaded by typing 'b [coyote]**:/mnt/mobi/sun/src/iptest'**; a '?' would give a list of commands. There was no documentation for this though.

It was suggested that interesting effects could be achieved by double-exposures of faces.

Problem 5, Review of Solutions

At the end of this section is a selection made up of the students' final portraits and some of their intermediate products. Before looking at the portraits, let's discuss the wide variety of techniques that the students used and tools that they built in order to produce their self-portraits.

All of the students started out with some digitized portraits of themselves. There are a number of ideas that were common to many of the self-portraits.

Edge operators. Some of the students wrote routines implementing edge detection operators based on differencing the values of adjacent pixels. DEA and KES used them for some of their intermediate products. This idea contributed to some **RFC's** and **AGT's** final work.

Picture editing. Many students developed programs to cut pieces from various pictures and put them together; laying one piece on top of another could either occlude the piece on the bottom, or the bottom piece could also be seen (like a double exposure). Some routines, like the one used to produce **DEA's** final portrait, performed a specific overlaying of pictures. At the other extreme, RFC implemented an interactive picture editor.

Reflecting. PCC, AGT and MJB experimented with reflecting various parts of their faces.

Projecting. Another operation that many students performed on their pictures was to project the images onto surfaces that weren't parallel to the plane of the paper. MJB, ARG, AJU, and RMW made use of this in their final portraits.

Before we finally get to the pictures, let's say a few words about each one.

DEA first tried to have the computer transform a digitized photograph into **a** 'sketch' by using edge operators. He was dissatisfied with the results, and he settled on a weighted double-exposure of two pictures. The weighting scheme favored one picture toward the center, and the other picture toward the edges.

MJB experimented with 'wrapping' her face onto a cube, and also with reflecting it. Thus, the top portrait made use of three-fourths of her original digitized image, while the bottom one made use of only half of it.

PCC produced his final result using multiple iterations of combining pictures by reflecting and by weighted double-exposures. Note that his three faces each have a pair of -glasses, yet there are only four lenses in the picture.

RFC used his picture editor to produce his portraits. He was able to use a mouse to cut polygonal regions out of pictures, which he then pieced together. The source pictures that he used were his original picture, an edge-filtered version of his picture, and a pure black picture.

MDD's original idea was to have his picture resemble a 'scribbled' drawing with one continuous line. He ended up using multiple splines whose density is proportional to the darkness of the original portrait. His final product was actually a three color portrait, which can't be reproduced here; an intermediate black-and-white version is therefore shown.

ARG's portrait was produced by projecting an image of his face onto a sphere and then repeatedly merging the picture with a shrunken verion of itself.

AG's picture is made up of pieces of several pictures. To project his face onto the frisbee, AG had to have his computer program represent the shape of a frisbee. Once he had this, he tried to have the computer generate a simulated picture of a frisbee; he found this unsatisfactory, and he ended up using a photo of a real one.

AAM started with his original portrait and then had routines add in a few devilish features.

KAM's final picture was made by applying transformations to a picture, brightening/darkening pictures, and overlapping pictures. The overlapping gives an illusion of depth.

KES's final product was actually a three-dimensional figure that was molded using edge-detection filtered photographs from multiple angles as guides. Reproduced here are some intermediate products.

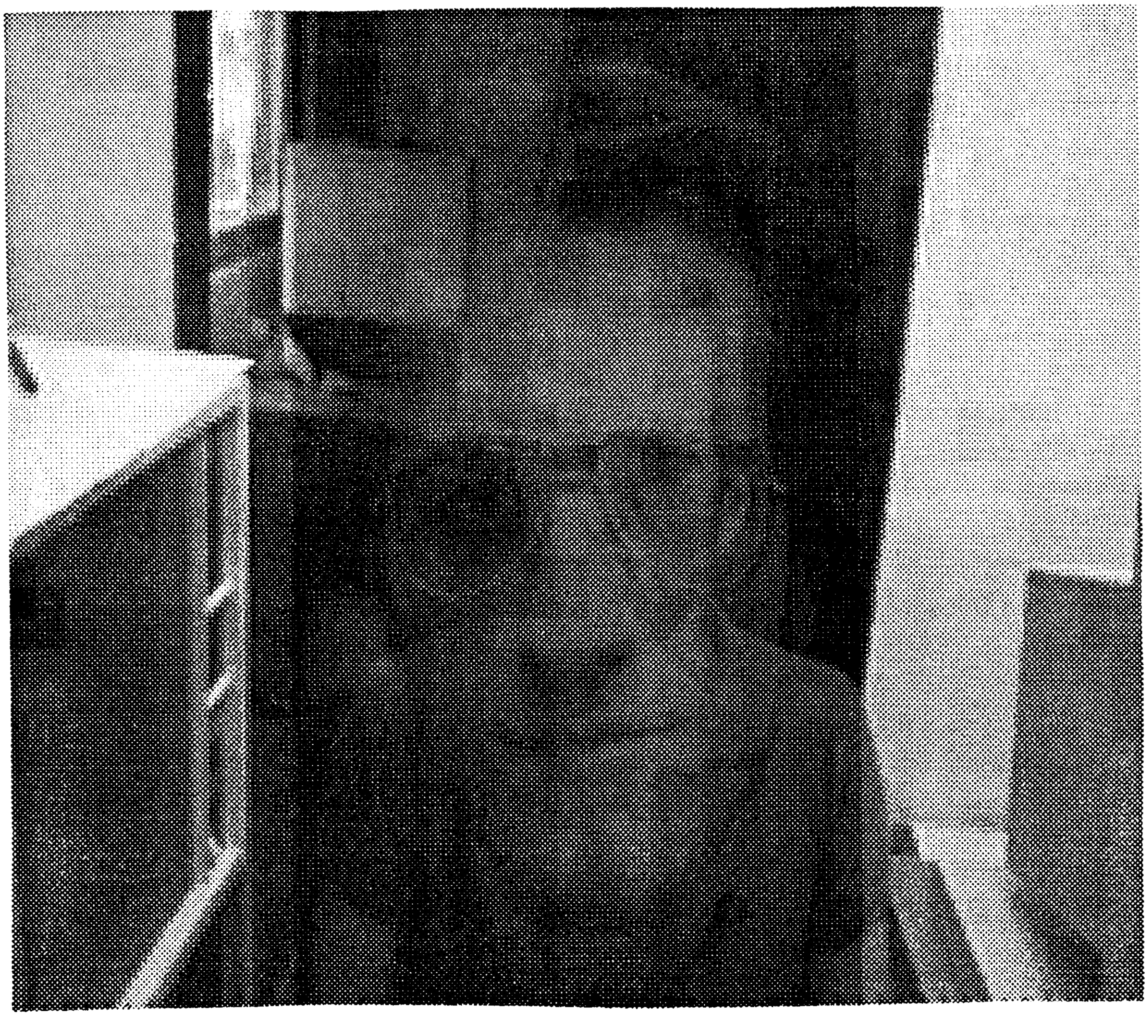
AGT started by enhancing the contrast of his picture and clipping off the background on the left side of his original portrait (he forgot the right side). Three of the pictures were produced by just reflecting this modification of the original. The fourth, the one on the bottom-right, was produced by overlaying the picture above it with that same picture run through an edge-detection routine.

AJU had routines that rotated pictures and overlayed them. While overlaying, she treated white as being transparent; this yielded an interesting 3-D effect.

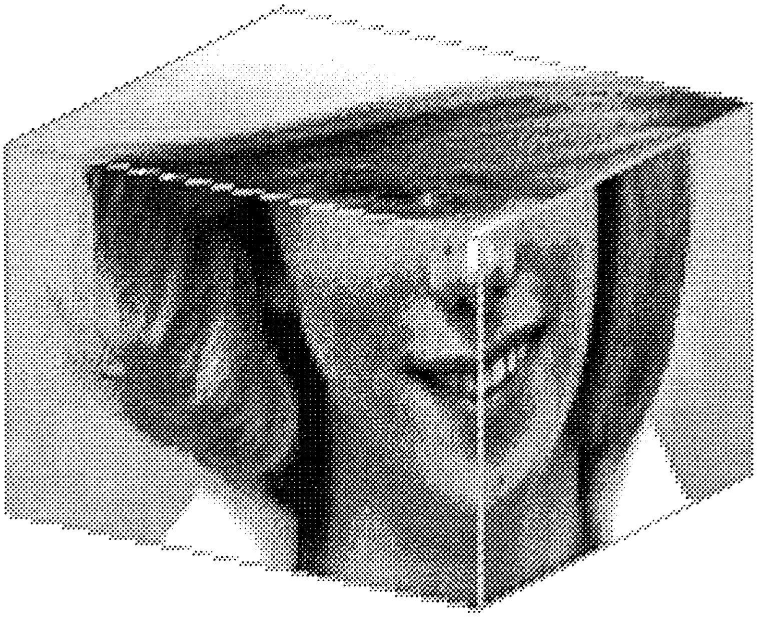
JFW took his original picture, and had a program add the herring-bone pattern, that can be clearly seen in the background, to his portrait. He didn't use modular addition; his numbers reflected back rather than wrapping around.

RMW took his original picture and cut it into four horizontal strips. He used these to create (within the computer) a three dimensional object made up of four blocks; each block had one of the strips on each of its four sides. The final result was produced by looking at the resulting object from four different views.

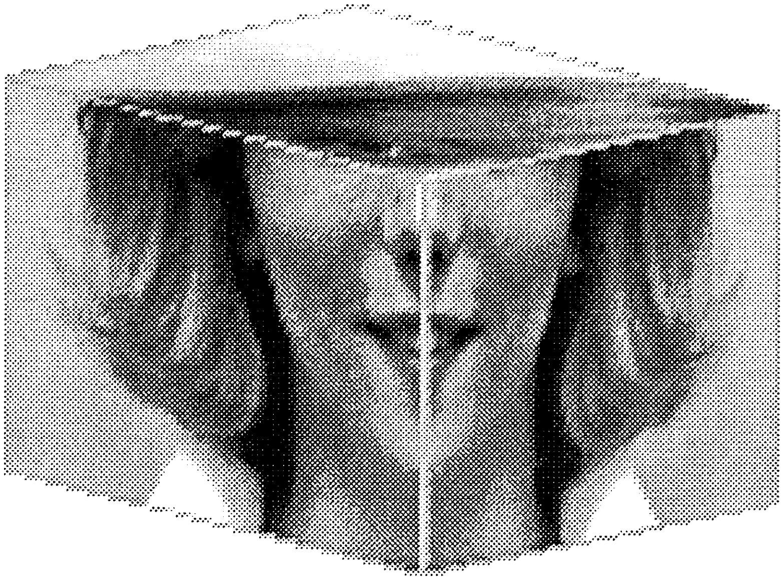
JIW's self-portrait has his face, drawn in sand, being blown away by the wind. The lighter points in the picture are considered to be higher. It is easier for the randomized simulated wind to blow sand down-hill than to blow it uphill.

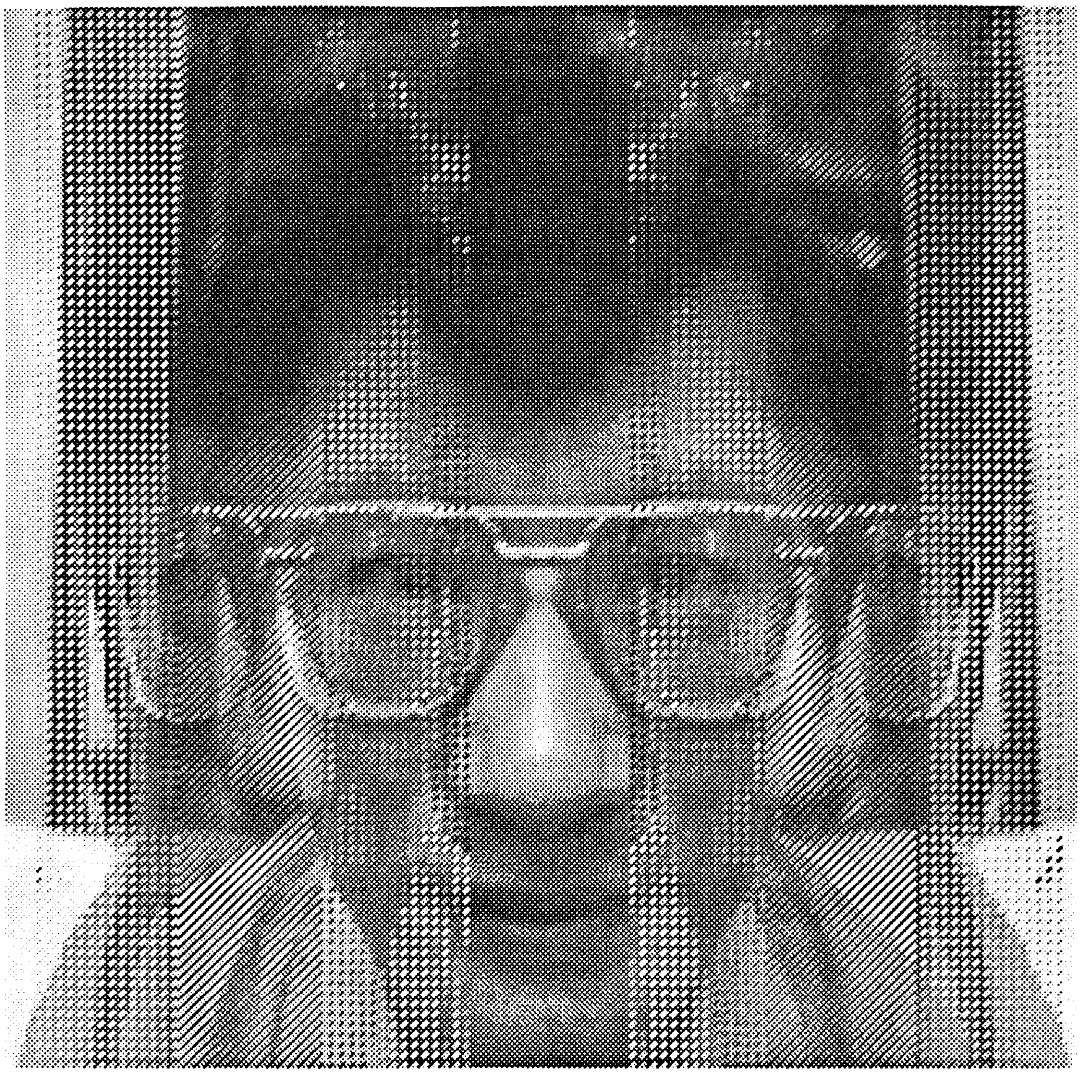


DEA — Dimitrios E. Andivahis .



MJB — Marianne J. Baudinet

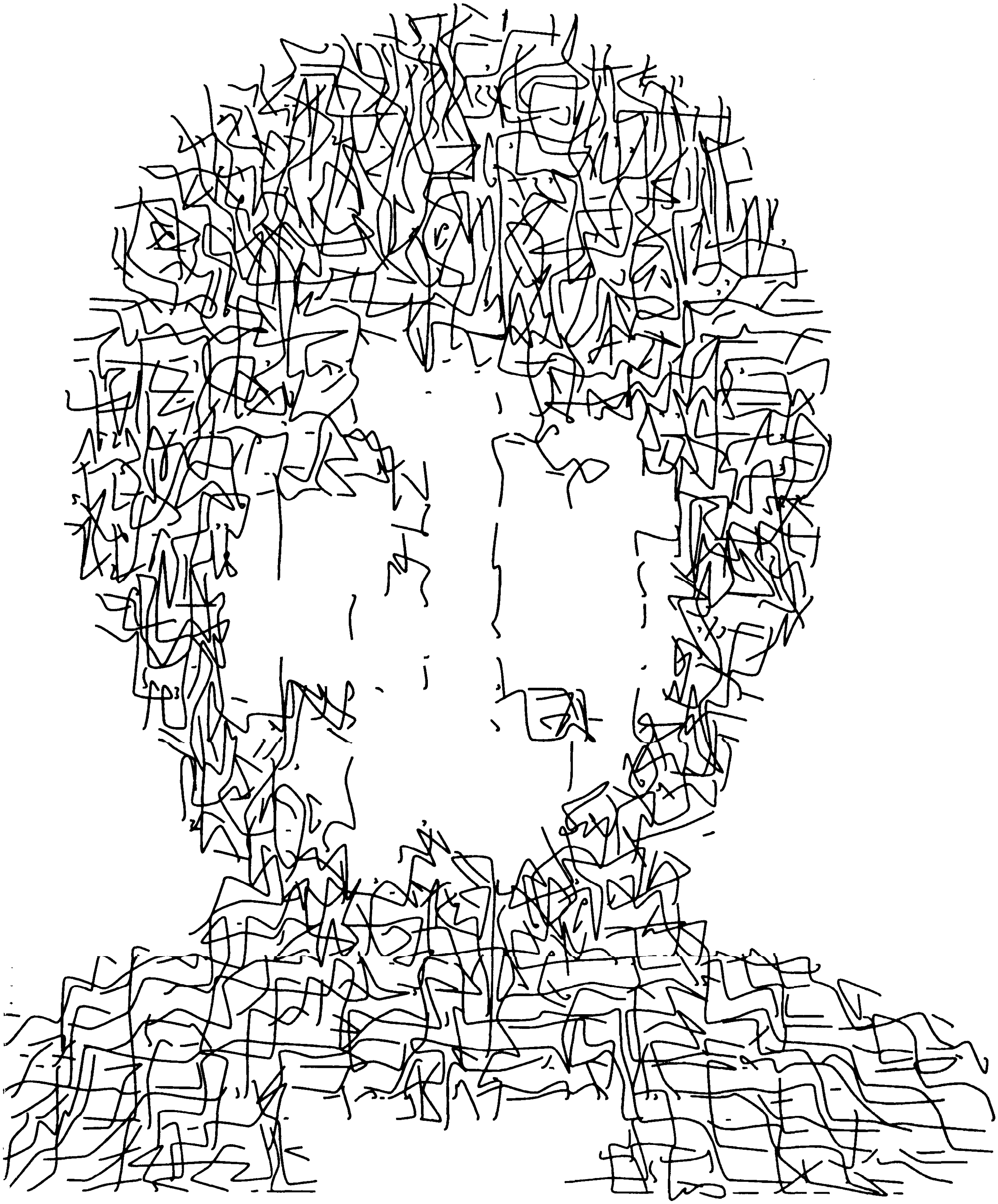




PCC — Pang-Chieh Chen

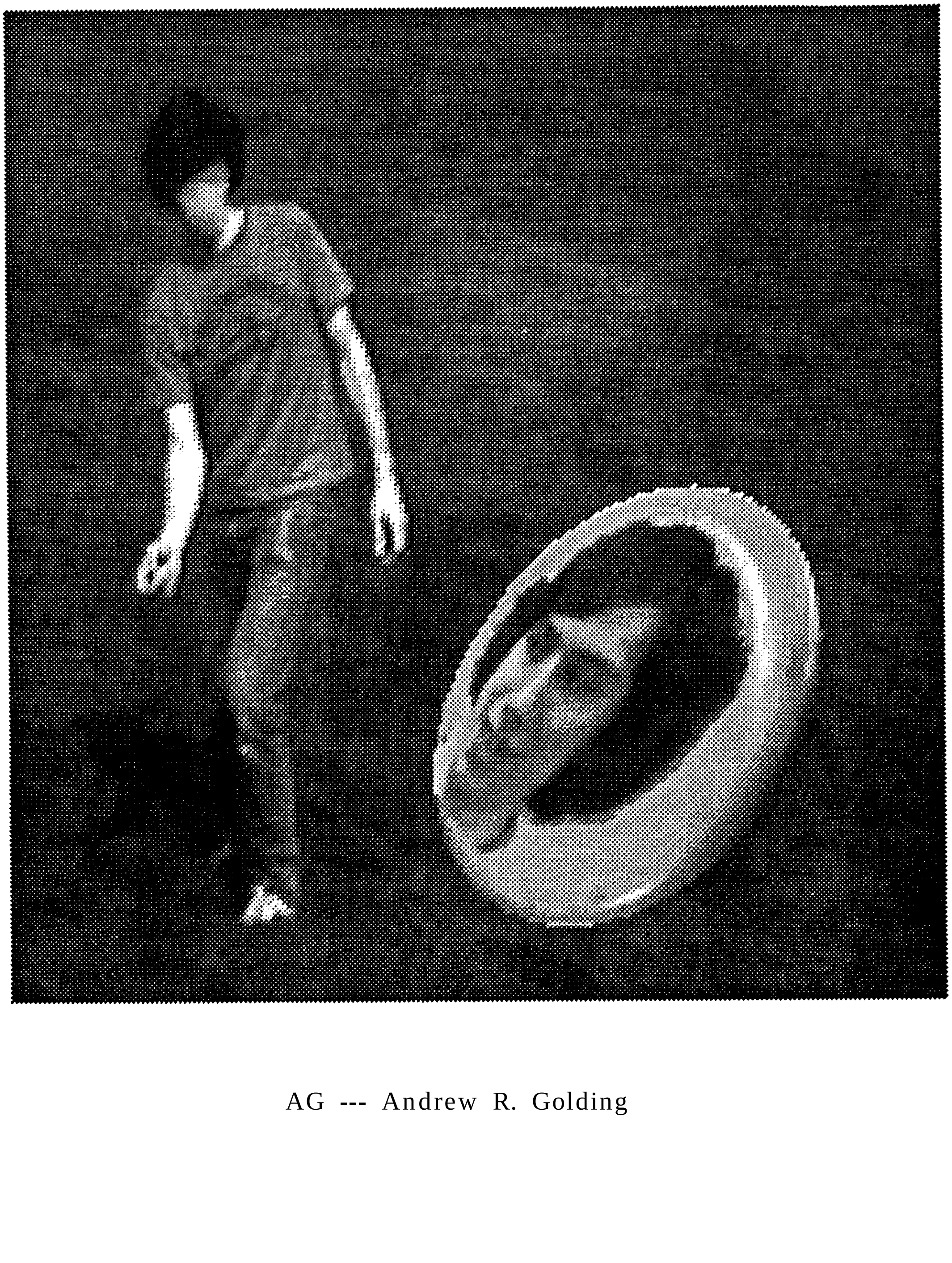
RFC —Roger F. Crew

RFC -- Roger F. Crew



MDD ---- Mike D. Dixon

ARG -- Anil R. Gangolli

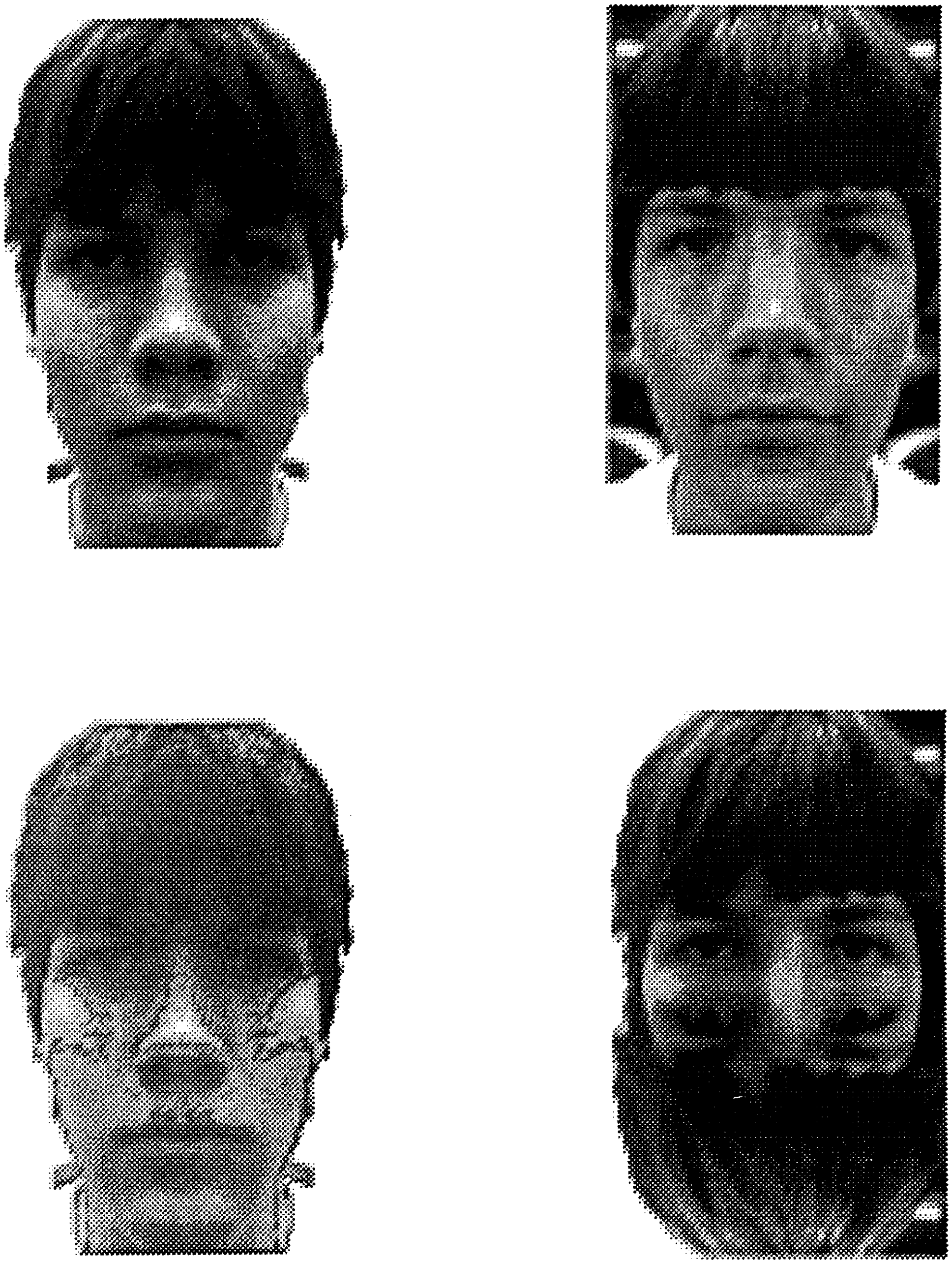


AG --- Andrew R. Golding

AAM — Arif A. Merchant

KAM — Kathy A. Morris

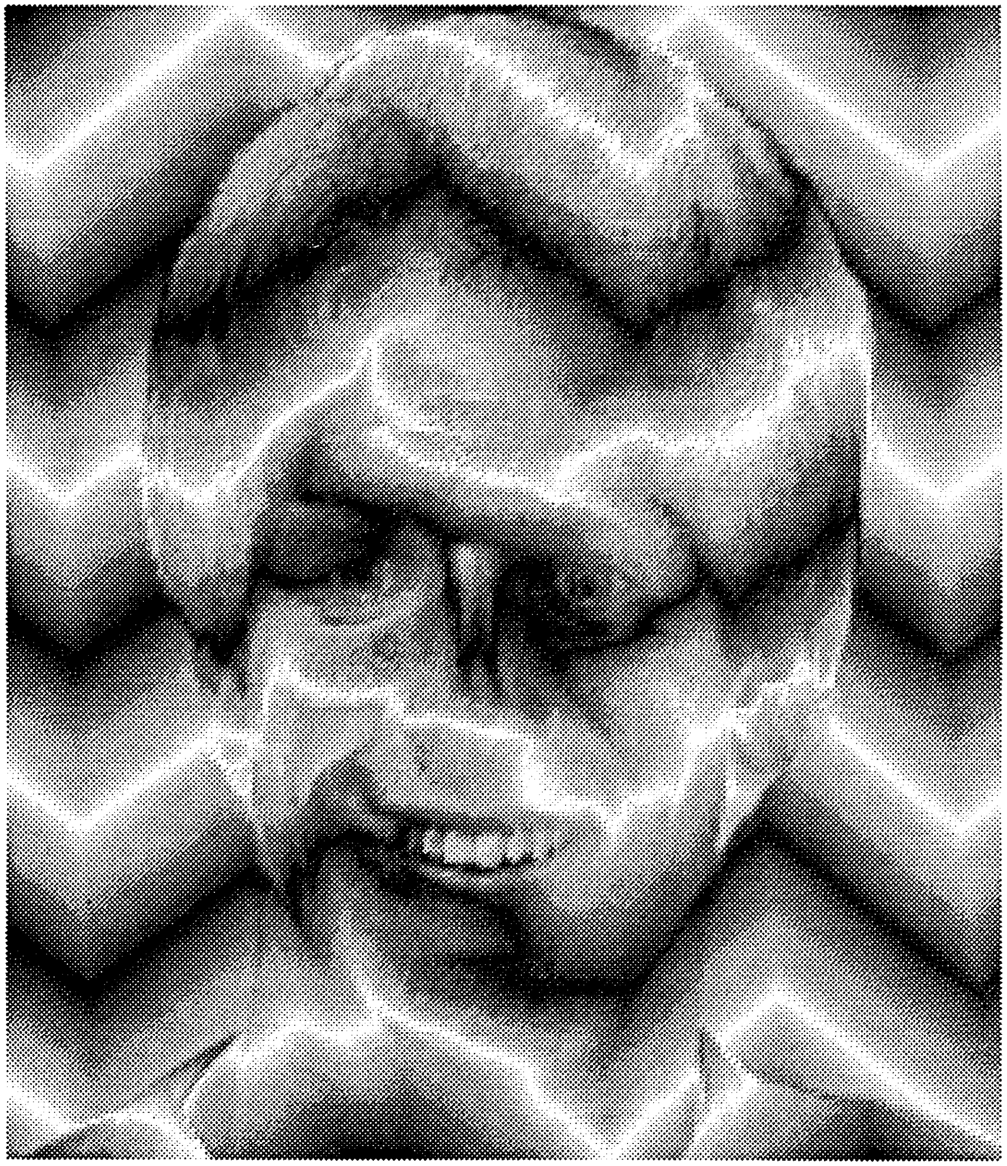
KES - - Karin E. Scholz



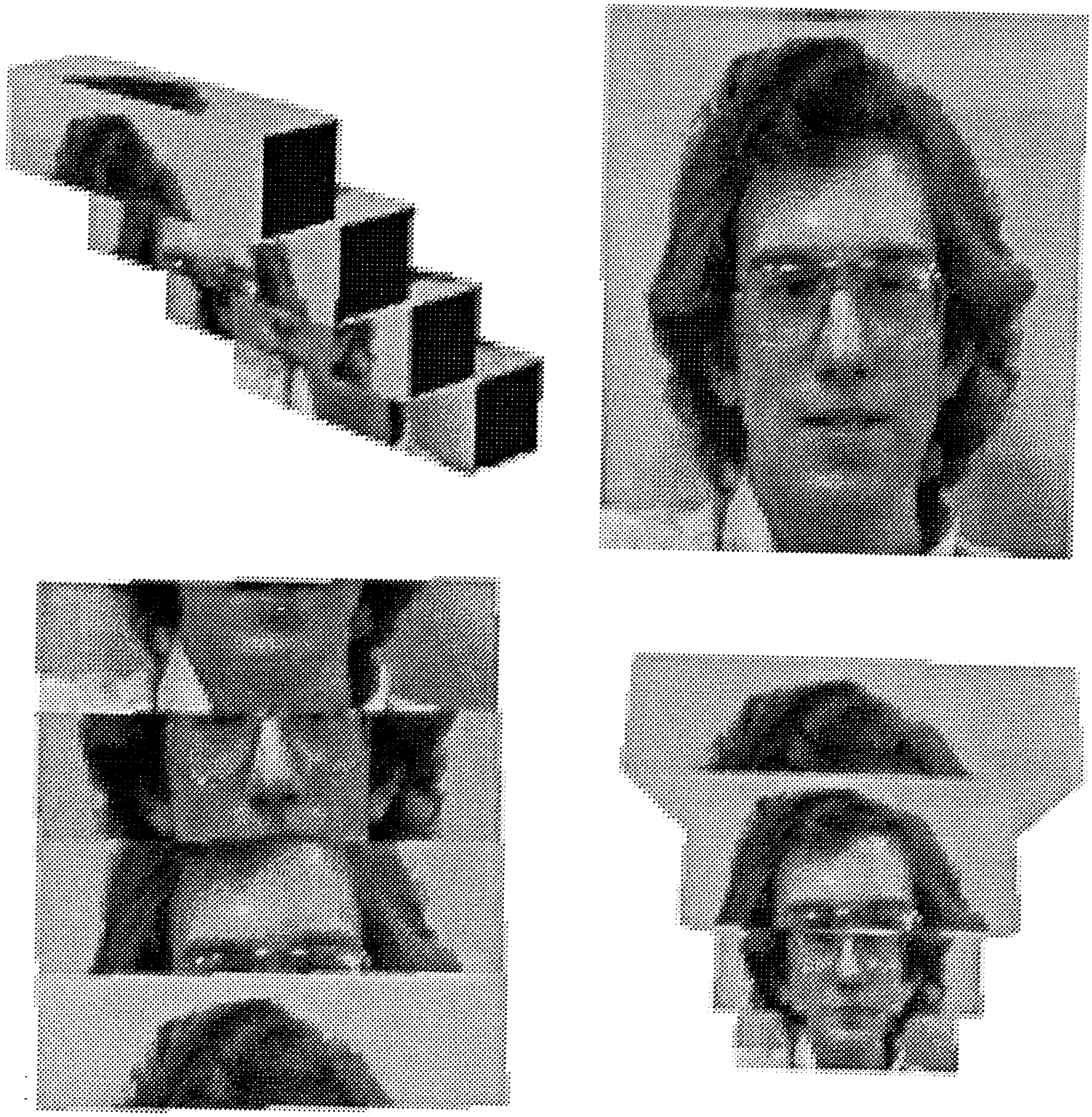
AGT —Andrew G. Tucker

AJU — Amy J. Unruh

Problem 5



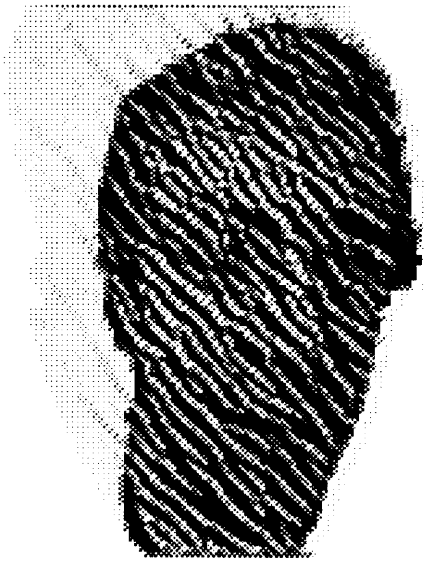
JF VV -- John F. Walker



RMW — Richard M. Washington

JIW —John I. Woodfill

State 110



State 010

State 001

State 011

'The lone and level sands stretch far away.