February 199 1 Report No. STAN-CS-9 1- 1350

A Programming and Problem Solving Seminar

by

E. Chang, S.J. Phillips, J.D. Ullman

Department of Computer Science

Stanford University
Stanford, California 94305

A PROGRAMMING AND PROBLEM SOLVING SEMINAR

by

Edward Chang, Steven J. Phillips and Jeffrey D. Ullman

This report contains transcripts of the classroom discussions of Stanford's Computer Science problem solving course for Ph.D. students, CS304, during Winter quarter 1990, and the first CS204 class for undergraduates, in the Spring of 1990. The problems, and the solutions offered by the classes, span a large range of ideas in computer science. Since they constitute a study both of programming and research paradigms, and of the problem solving process, these notes may be of interest to students of computer science, as well as computer science educators.

The present report is the ninth 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-81-863 (Allan A. Miller, 1981), STAN-CS-83-989 (Joseph S. Weening, 1983), STAN-CS-83-990 (John D. Hobby, 1983), STAN-CS-85-1055 (Ramsey W. Haddad, 1985), STAN-CS-87-1154 (Tomas G. Rokicki, 1987), and STAN-CS-89-1269 (Kenneth A. Ross, 1989).

Contents

1 Introduction **2**
1.1 Data Sheet 2
1.2 The Participants 13

2 Superabundant Numbers **14**

3 Tiling a Hall **23**

4 Constructing a Star Map 29

5 Generalized Hi-Q **35**

6 Playing God **41**

7 CS204 Problem Statements **50**

8 CS204 Class Notes **56**

9 Conclusions – JDU 94

Chapter 1

Introduction

The original descriptions of the course problems appear on the following pages. These descriptions were handed out on the first day of class.

1.1 Data Sheet

Time/Place: Tuesdays and Thursdays, 11am–12:15pm, in Terman 101.

Instructor: Jeffrey D. Ullman, 338MJH, ullman@cs, 723-1512.

Office Hours: Rosemary Napier (see below) can schedule appointments Monday and Wednesday afternoons, after 2:15 pm.

TA: Steven J. Phillips, 408MJH, phillips@cs, 723-0618.

Course Secretary: Rosemary Napier, 340MJH, rfn@sail, 723-3825.

Course Format: We shall generally follow the pattern set by Don Knuth for the course. There will be five problems, with two weeks for each problem. The TA will take notes and distribute them to the class; we plan to turn the notes into a TR, eventually.

CS304 should be both challenging and fun. Students are encouraged to cooperate in teams of two or three to solve the problems. We recommend the traditional policy that two people should not team up on more than one problem. Cooperation among teams working on separate programs is reasonable, and will not be regarded as "cheating."

Grading: We plan to follow the DEK policy that all participating in the course will receive an "A" grade. Solutions to the problems, including code, will be graded by the TA for your own information. When handing in a solution, please document your code reasonably, but especially include a clear description of the data structures and algorithms used, and an analysis of the running time, where appropriate. It is also helpful to describe

briefly any failed approaches and anything you would do to improve matters if you had had more time.

Computing Resources: I assume that class members will be able to find cycles on departmental machines. Let me know if you are having problems getting enough computing; I'll find something for you. There are no restrictions regarding programming language.

Caveat: Apparently, CS304 traditionally requires more work than typical 3-unit courses. That may be true this year as well. I hope you will find it worth the effort.

CS304 PROBLEM #1: Superabundant Numbers

Due Monday, Jan. 22

The abundance of an integer *n* is the sum of the divisors of *n* (including *n* itself), divided by *n*. Integer n is *k-abundant* if its abundance is at least *k*.

For example, the sum of the divisors of 6 is 6 + 3 + 2 + 1 = 12, and $12/6 = 2$, so 6 is 2-abundant. As another example, the sum of the divisors of 120 is

$$120 + 60 + 40 + 30 + 24 + 20 + 15 + 12 + 10 + 8 + 6 + 5 + 4 + 3 + 2 + 1 = 360$$

so 120 is 3-abundant. It happens that 6 is the smallest 2-abundant number and 120 is the smallest 3-abundant number. They happen to be exactly 2- and 3-abundant, respectively, but it is generally possible that the smallest k-abundant number has abundance greater than *k*.

Your assignment, should you choose to accept it, is to write a program that finds the smallest k-abundant number for $k = 1, 2, \ldots$. How high can you go?

I'm not sure whether the following approach will turn out to be best, but you might consider the investigation started by SJP, who looked for those numbers that are more abundant than any smaller number. In his honor, let's call them Phillips numbers. Can you find the first m Phillips numbers for some large m? What is the density of Phillips numbers? That is, how many Phillips numbers are there between the first k-abundant number and the first *(k* + l)-abundant number?

CS304 PROBLEM #2: Tiling a Hall

Due Monday, Feb. 5

This problem is based on what, according to my son Peter, was the Princeton undergraduate programming contest problem for 1989. He says that the contest was won by an outsider, Nate Thurston (who is Bill Thurston's son, and goes to Reed College), whose solution was declared to be "constant time." I'm not so sure that is right, since by my calculation, it takes $\Omega(\log n)$ time just to read *n*, the length of the hall. Well, no matter: let's see what you folks can do with it.

The Problem

Given an integer n and a collection of "tiles," determine whether a "hall of length n," that

is, an n *x* 4 rectangle can be completely tiled by some combination of the tiles. Each tile may be used as many times as needed, but only in the orientation(s) given in the data. That is, tiles may not be rotated, slid horizontally (across the hall), or mirror-imaged. They may be moved up or down the hall. By "completely tiled," we mean that the rectangle must be completely covered, and no tiles may overlap.

Tiles

A tile is a subset of a 4 x 4 array of squares. We use 1 to represent a square that is "present" and 0 to represent absence. When drawing tiles, we shall use the orientation in which vertical corresponds to the long dimension of the hall and horizontal corresponds to the width of the hall. For example, suppose we want to tile the hall with dominos. Then the seven tiles in Fig. 1 must be given as input.

0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000
1000	0100	0010	0001	0000	0000	0000
1000	0100	0010	0001	1100	0110	0011
(a)	(b)	(C)	(d)	(e)	(f)	(g)

Fig. 1. Seven tiles representing a domino.

Tiles (a), (b), (c) and (d) 1ow us to use a domino vertically, in the first, second, third, and fourth columns of the hall, respectively. Tile (e) lets us lay a domino horizontally, flush left, (f) 1et s us lay it horizontally in the middle two columns, and (g) lets us lay a domino horizontally flush right.

Tiles need not be connected. Here is a possible tile.

0010
1000
0111
0010

We shall, however, guarantee that tiles are presented as low in the grid as possible. That is, you can count on at least one square in the bottom row being 1.

Input Format

You will be given a file consisting of tiles, one to a line, followed by a line with a #, and one or more lines cont aining hall lengths. A tile is writ ten as sixteen O's and 1 's, representing the rows of the 4 x 4 array, from the top. For example the problem consisting of the tiles of Fig. 1, in order, with the two hall lengths 57 and 90 would be coded as the file

```
0000000010001000
0000000001000100
0000000000100010
0000000000010001
0000000000001100
0000000000000110
0000000000000011
#
57
90
```

For this example, any hall can be tiled by dominoes, so the correct answer is "Yes, Yes."

How Bad Can Tiling Problems Be?

An interesting side issue is how hard can an instance of this tiling problem be? One way to measure difficulty is to ask what is the shortest hall that can be tiled by a set of tiles. Of course, some sets of tiles do not allow *any n x* 4 hall to be tiled, but among those that do, what set of tiles has the largest minimum-length hall that it is possible to tile? There are some other possible measures of badness that we shall discuss after we've had a chance to understand the problem.

CS304 PROBLEM #3: Constructing a Star Map

Due Monday, Feb. 19

This problem is a modification of one suggested by Bob Floyd, who says he posed it in the 1973 edition of CS304 (then CS204), but "nobody did much with it." My version is, up to a point, simpler. At the end of this description, I'll discuss the Floyd version, and some sample data will be provided if you care to tackle it.

Trantor, the center of the galactic empire, is at galactic coordinate $(0, 0, 0)$, of course. The Earth in this coordinate system is at $(10, 10, 0)$. Astronomers on the two planets are cooperating to produce a 3-dimensional map of the stars in a recently discovered cubic globular cluster.† The stars of this cluster occupy the cube with opposite corners at $(0, 10, 0)$ and $(1, 11, 1)$; that is, its center is at $(.5, 10.5, .5)$, slightly closer to Earth than Trantor. Figure 1 shows a perspective of the situation.

A photograph of the CGC is taken from Trantor, with the camera pointed along the y-axis, so the CGC appears in the first quadrant of the photo. The scale of the photo is such that the CGC appears to be projected on a plane one unit in the y direction from Trantor, i.e., the equation of the plane is y = 1. As a result, the photo is approximately, in $1/10$th scale. It is important to understand that stars near the "back" of the CGC will tend to appear slightly closer to the origin than stars near the "front."

Another photo of the CGC is taken from Earth, with the camera aimed in the $-x$

†: Current astrophysical theory says that the cubic globular clusters are the galaxies that, at the creation of the universe, were taken out of the box last.

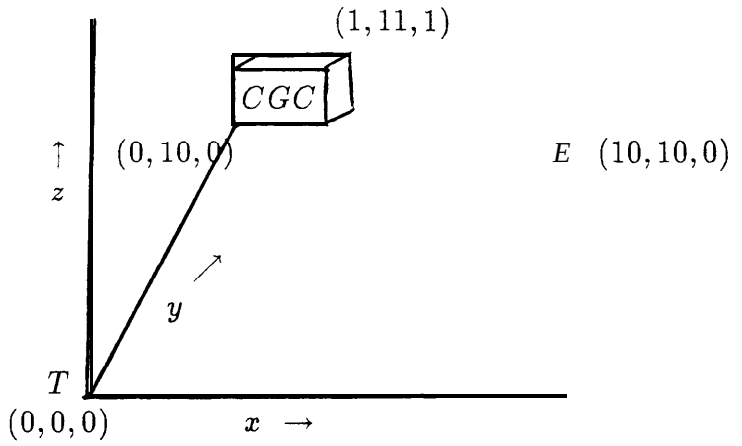


Fig. **1.** View of Trantor, Earth and the cubic globular cluster.

direction. Again, the scale of the photo is such that the CGC appears to be projected onto a plane one unit from Earth. This plane is in the $-x$ direction from Earth; that is, its equation is $x = 9$. Again the scale is about $1/10$th, but the CGC appears slightly larger (by about 10%) from Earth than Trantor. Figure 2 shows the geometry of the photographs, viewed from above.

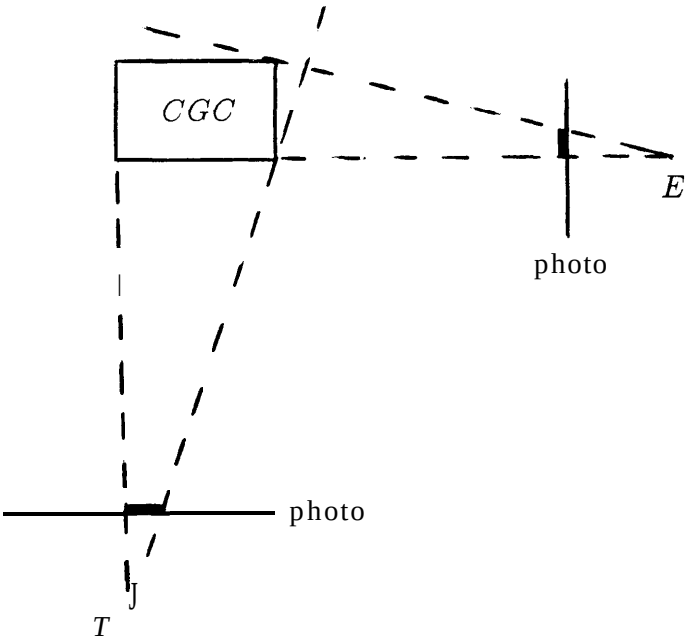


Fig. 2. Making the photographs.

Data

Unfortunately, due to imperfections in the photographic process, the stars do not necessarily appear in their proper positions. All horizontal and vertical positions have been

"jiggled" by adding a random number chosen uniformly in the range $(-10^{-6}, +10^{-6})$. Note that has roughly the effect of moving stars by up to 10^{-5} units, since the scale of the photos is about $1/10$th.

We shall provide a file containing the observations of 10,000 stars, one to a line. A sample line is

```
8.280807e-02 7.724439e-02 1.042154e-01 9.302261e-02
```

The separating characters are tabs. The first number is the horizontal (x-axis) projection of the star onto the photo at Trantor, and the second number is the vertical (z-axis) projection at Trantor. The third number is the horizontal (y-axis) projection at Earth, and the fourth is the vertical (z-axis) projection at Earth.

It is important to remember that the data carries the answer. That is, the ith star in the view from Trantor is the same as the ith star in the view from Earth. Given this "hint," it is a simple matter to estimate the true location of the star from its two views, compensating as best one can for the random noise. However, your program **must not** take advantage of the hint. To be safe, you should take some prefix of the data (working with all 10,000 stars is tough), separate the Earth and Trantor views (awk is good for this step), sort one of them (sort is good here), and read the data that way. Let your program try to figure out which Trantor star best matches which Earth star.

The Problem

Your task is to take the first **n** stars, for as large an **n** as you can, and reconstruct the star map. That is, you should determine the galactic coordinates of each star. As I see it, you face three demons.

1. You need to remember enough calculus that you can figure out which star from Trantor most closely matches which star from Earth. (Again, remember either to permute the order of the stars as seen from one of the planets, or write your program without bias in favor of matching the ith stars from each planet.)

2. As n gets large, the best matches are not always the true matches. For all 10,000 stars, about 10% of the matches will be false. Sometimes there is nothing you can do. For example, two nearby stars may be "jiggled" enough that they get switched; i.e., their images at one planet match the other star at the other planet. Other times, one can hope to correct, because certain matches lead to implausibly large distances between the observations of "matching" stars at the two planets.

3. Also as n gets large, comparing each star at Trantor with each star at Earth becomes painfully slow.

The Original Variant

In the original Floyd problem, there was no jiggling of data, i.e., projections were exact, to the precision of the computer. However, he didn't tell the students where the second photograph was taken. If you'd like to tackle this problem, we'll also provide an image taken from a space station somewhere in space. The camera was aimed at the point $(0, 10, 0)$, and the scale is such that the image appears to be 1 unit in the direction the

camera points, just as for the other two observations.

The data for this variant will consist of 100 stars, in the same format as was described above, with no perturbation of data. The first two numbers are the horizontal and vertical positions from Trantor, and the second two are the horizontal and vertical positions from the space station. However, in this data, the stars *have* been permuted, and you should not assume that the two observations on a line are of the same star. Your problem is to find the position of the space station and reconstruct the star map.†

CS304 PROBLEM #4: Generalized Hi-Q

Due Monday, March 5

Last summer, Peter, Scott, and I drove the car back from sabbatical in Princeton (well actually, Scott read comics in the back seat, Peter drove, and I held the TripTik). Somewhere in the middle of Kansas, or some state that looks remarkably like Kansas, we stopped at the only restaurant whose existence the AAA acknowledged for 30 miles in either direction.

While we were waiting for our chickenburgers or something, we found that a game board had been put on each table. The board (Fig. 1) had 15 holes arranged in a triangle, and a supply of pegs. To start, you put a peg in each hole. Then, you removed pegs from certain holes, depending on your route; remove this peg if you are traveling west, this one if you are on an odd-numbered interstate, and so on. Then, you had to remove all the other pegs by jumping. That is, if *a*, *b*, and c are three consecutive holes along any diagonal or horizontal line, *a* is empty, and *b* and c have pegs, you can move the peg in c to *a* and take the peg in *b* off the board.

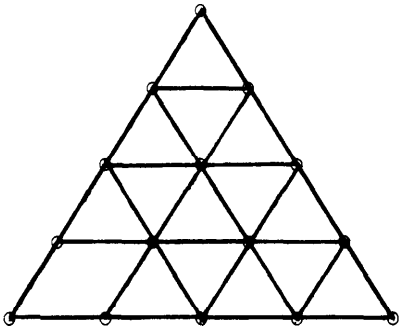


Fig. 1. Game board from Kansas restaurant.

We tried the game without success until our chickenburgers arrived. Then we tried the chickenburgers; nothing much there either. Peter and I agreed that if we had a laptop along, we could have solved the problem easily, because there are only 2^{15} configurations to worry about. Scott pointed out that one of the Teenage Mutant Ninja Turtles (the one

†: Actually, this problem is still easier than the one posed by Prof. Floyd, but if I told you why, I'd be giving away the store, so I'll discuss it in class by and by, instead.

with the superpower to solve meaningless games quickly) could have solved the problem without a laptop. However, this game reminded me of one I played when I a kid, called Hi-Q. This puzzle had 45 holes arranged as in Fig. 2, with only the center hole left empty initially. Jumps were allowed horizontally and vertically. I never had much success with that one either.

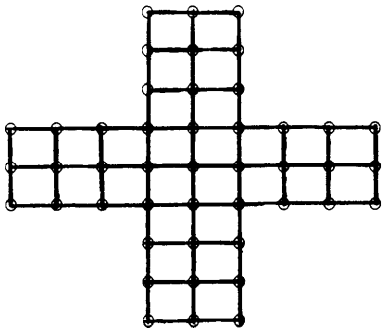


Fig. 2. Hi-Q board.

We're going to develop a solver for Hi-Q type problems, where jumping in horizontal and vertical directions only is permitted. (Aside: do problems with a hexagonal grid and three jumping directions reduce to the square grid problem?) The generality comes from the shape of the board. We shall assume boards are located in the first quadrant of the plane, and we shall give the contents of the columns, starting at column 0 and proceeding right, until the last column that contains a hole. Each column will be given on a line, and will consist of the y-coordinates of holes in that column. We use *i-j* as a shorthand for the list $i, i+1, \ldots, j$.

Following the columns will be a line containing the keyword empty and a list of one or more pairs of x-y coordinates representing the initially empty holes. The list of pairs can be spread over several lines, but each pair is on one line, surrounded by parentheses. Pairs are separated by white space. For example, the Hi-Q board itself could be represented, among other ways, as in Fig. 3. If the eight corners of the board, instead of the middle, were left empty initially, then the lines

```
empty (0,3) (0,5) (3,8) (5,8)
(8,5) (8,3) (5,0) (3,0)
```

would replace the last line in Fig. 3.

You can be certain that the board will be presented flush against the x and y axes. Also, we shall guarantee that the dimension of a board will not exceed 20 x 20; i.e., the grid points are a subset of $\{0, \ldots, 19\}$ x $\{0, \ldots, 19\}$. You may also assume that the board is connected.

The object of the game is to remove all but one peg. In the original Hi-Q game, you got extra credit for leaving the last peg in the center, but we shall not make such stipulations part of the input. If you cannot remove all but one peg, try to leave as few pegs as possible. We're not going to put a time limit on your program, but after about 10 minutes, everyone will lose interest and walk away.

```
3,495
5,4,3
3-5
O-8
0,1,2,3,4,5,6,7,8
5,6-8,0-4
3-5
3-5
3-5
empty (4,4)
```

Fig. 3. Input format example.

Some Additional Questions

Can you characterize the configurations for which there is a solution? A quick test for a solutionless configuration, even if it missed some, would be a big advantage in searching for a solution, so perhaps a better way to phrase the question is: find a polynomial time test with a high probability of recognizing a solutionless configuration and a very low probability of declaring a configuration with a solution to be solutionless.

Finally, consider one-dimensional boards. Can you give an (efficient) algorithm to test whether a configuration on a one-dimensional board has a solution? Is the problem formally intractable?

CS304 PROBLEM #5: Playing "God"

Playoff Thursday March 15
Writeup due by Monday, March 19

In the days before Dungeons *and Dragons,* one of the things nerdy teenagers did was play a card game called "God." On each round, one player was selected to be the god. The god made up a rule whereby cards could be played on a pile, which was constructed in a line, so all played cards were visible. An example of a (too simple) rule is "only a red card can be played on top of a black card, and vice versa." To begin play, all cards but one were dealt to the players, and the last was turned up to start the pile.

In turn, players offered cards for the top of the pile. If the play meets the god's rule, then the card is allowed to stay on the pile; otherwise it is withdrawn to the player's hand. Play ends when one player gets rid of his last card. That player, and the god, each score an amount equal to the sum of the cards remaining in the other players' hands.

Selection of God Rules

The scoring system suggests that the god should pick a rule that is nontrivial, but deducible with some effort. If a rule is too easy, then everyone will catch on quickly and get rid of their cards at approximately the same time. The score will tend to be low, as no one will

be caught with many cards. If the rule is too hard, everyone will play as if at random, and the expected number of cards with which anyone is caught will be low as well. (Aside: how low?) Ideally, from the point of view of the god, one player should figure out the rule immediately, and get rid of his cards quickly, while the other players are stumped and get rid of cards only by luck.

There are some constraints on legal rules that the god may use. First, while many good rules are "O-memory," in the sense that playability of a card depends only on the previous card played, it is permissible to use a rule in which the playability of a card depends on the entire pile. Example: "The sum of the cards on the stack, modulo 13, must be a prime." However, the following must be satisfied.

1. The rule depends only on the stack contents. No "I pass them crossed" or "Anything Sally plays is OK, but anything anybody else plays is wrong."
2. There must be at least 10 cards playable in any situation. This rule must be understood to apply on the assumption that an infinite supply of cards exists. Otherwise, with any rule and any stack of 51 cards there is only one card playable. An example of an illegal rule: "each card must be of one higher rank than the previous card on the stack, with Ace following King and Deuce following Ace." (Only 4 cards are playable in any situation.)

Even with rule (2), it is possible that the game will reach a situation where there are no legal moves for any player. Example: "A letter card must follow a number card, and vice versa."

Some Example Rules

Consider the following possibilities, for example.

1. If the rank of the card played is equal to or higher than the rank of the top card, then the two cards must be of the same color; otherwise, they must be of different colors.
2. The difference in the ranks of the cards must be no greater than 3, in the "end-around" sense (e.g., Deuce is distance 3 from Queen).
3. A letter card may not be played if either of the top two cards are letter cards.
4. Face cards (J, Q, K) playedmust be of the same color as the top card. Other cards must be of a minor suit (C, D) if the top card is a major suit (H, S), and vice versa.

The Problem

You should write a program to run on neon that will play "solitaire" God, interacting with a controller program being built by SJP. Your program starts with 51 cards, and the Ace of Spades is assumed to be the initial card on the pile. You make a play by writing a card in a file of your own directory named `play.gd`. A card is represented by two characters, the first being the rank (A, 2, 3,. . . ,9, T, J, Q, K), and the second the suit (C, D, H, S). A serial number, in decimal, followed by at least one blank, must precede the card. For example, if your 105th play is the six of spades, you (re)write your file to contain the six characters

```
105 6S
```

The controller will then rewrite a file in its own directory (presumably SJP's directory) named out come. gd, to cont ain

1. The serial number of the play,
2. The outcome (Y if successful, N if not), and
3. A recap of the current state of the pile, consisting of cards separated by blanks, from the bottom.

For example, the controller might respond with the file contents

```
105 Y AS TC 8D JH 3S KC 4D 5H 6S
```

meaning the play of the six of spades was accepted, and the pile now consists of the nine cards listed.

Scoring

Your score will be the number of plays you make, until you either run out of cards, or reach a state where no card in your hand is playable. The controller will stop after 51 consecutive unsuccessful plays, but we shall verify that during that time you have tried every card remaining in your hand. (Nuance: if your program believes it knows the rule, it might try to drive the pile to a state in which no move is possible, to lower your score.) Points are awarded to the team with the low score; the number of points is the sum of the differences between that team's score and the other teams' scores.

Writing God Programs

We shall write some god programs of our own, but we shall also accept god programs, as C functions to be compiled with the controller, from those teams that wish to write them. Remember, the quality of a god program is measured by how high a score the winner can achieve.

Your function should have two parameters *rank* and suit of type char, representing a single card, and it should return int 1 (if the play is accepted) or int 0 (if not). You should assume that the first card on the pile is the Ace of Spades, and remember (using global variables whose names begin with gd) whatever you need to know about the state of the pile.

1.2 The Participants

Professor

JDU Jeffrey D. Ullman

Teaching Assistant

SJP Steven J. Phillips

Students

SQ Sean Quinlan
ET Eric Torng
AS Anton Schwarz
VG Vineet Gupta
DK Daphne Koller
DC David Cyrluk
AT Alberto Torres
MG Michael Greenwald
SR Scott Roy
AH Alan Hu
DP Dan Pehoushek

Chapter 2

Superabundant Numbers

January 9

The second scheduling of CS304 got off to a good start, with eleven enthusiastic participants in attendence. JDU explained that the course had been planned for Autumn, but had clashed with the comprehensive exams. Quick introductions were made all round, and a list of the names of all participants (and their abbreviations for these notes) will be available soon.

Before we start the (approximately) mathematical details of the day's discussion, it is appropriate to quote from *A programming and problem solving seminar,* by Kenneth A. Ross and Donald E. Knuth, for a little history of the course:

> George Pólya originated the idea of a problem solving course many years ago, and George Forsythe had promoted the idea in the computer science department. The course [has] esisted [at Stanford for] 21 years.

We launched quickly into a discussion of superabundant numbers, the topic of problem 1. SR and AT had a head start, being among the brave few who defied the comprehensive exams and attended the one lecture of CS304 in Autumn, but the discussion soon broadened to include most of the class.

Let us denote the abundance of a number n by $Ab(n)$, and let p denote an arbitrary prime. JDU noted that $Ab(1)$ = **1,** *Ab(6)* = 2, and *Ab(120)* = 3. Put more suggestively, the first numbers that are respectively 1, 2 and 3 abundant are **1** !, 3! and 5!. By Engineers Induction, the first n-abundant number is $(2n-1)!$. But no such luck — although AH found it intuitive that factorials should have high abundance, they are not always the most abundant. [What is the smallest factorial that is not a Phillips number?]

AT gave us the derivation

$$\begin{aligned} Ab(p_i^k) &= (1 + p_i + \ldots + p_i^k)/p_i^k \\ &= \frac{p_i^{k+1} - 1}{p_i^k(p-1)} \end{aligned}$$

and erased the subscripts after JDU said that the simplest notation is the best. Thus

$$Ab(p^k) = \frac{p}{p-1} - \frac{1}{p^k(p-1)}.$$

DK showed that

$$\begin{aligned} Ab(p^k q^l) &= (1 + p + \ldots + p^k)(1 + q + \ldots + q^l)/p^k q^l \\ &= Ab(p^k)Ab(q^l), \end{aligned}$$

which gives us an expression for abundance in terms of prime abundance. DK and others noticed that this multiplicative property generalises to any two numbers that are relatively prime. DK described a dynamic programming technique for calculating a table of abundances in an analogous way to the sieve of Eratosthenes.

AH brought up the question of whether arbitrary abundances can be obtained, and SR gave an outline of why the answer is affirmative: $Ab(p^k)$ tends to $p/(p-1)$ as $k \to \infty$, so

$$\begin{aligned} Ab(\prod p_i^{k_i}) &\approx \prod(p_i/(p_i - 1)) \text{ for large } k_i \\ &= \prod(1 + 1/(p_i - 1)) \\ &> 1 + \sum 1/(p_i - 1), \end{aligned}$$

and VG noted that since p_i, the i'th prime, is about i log i,

$$Ab(\prod_{i=1}^{r} p_i^{k_i}) \text{ diverges with } \int_e^r \frac{dx}{x \log x} = \log \log r.$$

With that out the way, we decided to determine order statistics on the smallest k-abundant numbers. We assumed without justification that the approximation above works well enough when each $k_i = 1$, and obtained

$$Ab(\prod_{i=1}^{r} p_i^{k_i}) \approx e^{\sum 1/p_i} \approx \log r.$$

VG noted that $n > 2$' so that $Ab(n) = O(\log \log n)$. The rest of the class did some trickier calculations for a while that turned out only to be a harder way of showing that $Ab(n) = O(\log \log n)$. A highlight was DP showing that $n = r^r \Rightarrow r \approx \log n / \log \log n$. [Take logs of both sides, divide by log r and substitute once for r.]

SR mentioned the problem of representation of the large integers that will be needed in the programs, and it was decided that the best to deal only with the prime factorisation of a number. AT stated that a search in some number theory books revealed no formula for perfect numbers, and he expects there to be no formula for Phillips numbers or for the first number whose abundance is more than some variable. JDU concluded by asking how big the abundance gap can be between adjacent Phillips numbers.

January 11

The following fact, derived on the 9'th, was left out of the notes: The multiplicative increase in the abundance when you replace a number with k factors p by one with $k + s$ factors p is $1 + \frac{p^s - 1}{p^{k+s+1} - p^s}$.

We started the day with computing troubles: AH had been reprimanded for using Neon for background brute force computation of Phillips numbers, but there is no other readily available source of cycles for such computations. JDU found the idea of having a computer

on which no-one is allowed to program pretty ridiculous, and resolved to find out before the next class what computing resources can be used. He said that the new HP workstations will probably be up and running by the end of the quarter [round of laughter], and in the meanwhile all class members are being given accounts on Nimbin. A mailing list has been set up; the address is cs304@nimbin.

Some programs had been written since the first class (and last quarter too), with the following results: AH had computed the first 26 Phillips numbers, finding that π_{26} = 166,320 and the first 4-abundant number is 27,720. His results are in his directory ajh/public/phillips. MG computed Phillips numbers up to abundance 10 last quarter, using carefully optirnised search methods and incremental calculation of abundances and values of products of primes. AH ran his program overnight, while MG's program took only 2 minutes of CPU time. Said JDU, "Algorithms do matter!" AH ably defended his program, saying he likes some quick-and-dirty data to work with and make conjectures about, as a first step in solving a problem. DP also did some programming, and handed out a list of "lower bounds on the minimum largest prime factor in k abundant numbers" which indicated about how many primes are needed to obtain abundances up to 18. His method was to calculate the abundance of high powers of successive initial segments of the list of all primes. He also handed out a list of estimates of the first 4, 5, 6 and 7 abundant numbers, obtained by search and trial and error. He will mail everyone a list of the first 32K prime numbers; VG pointed out that the program "primes" in the games directory produces primes very fast.

The next order of business was the discussion of some very interesting facts about Phillips numbers that will be helpful in focussing the search for high abundance. AT noted again that

$$Ab(p^k) = \frac{p}{\mathrm{p}-1} - \frac{1}{p^k(p-1)}$$

and that it easily follows that any Phillips number should be a product of consecutive primes starting with 2, and SQ added that the powers in the prime expansion must be monotonically decreasing. A few attempts were made to prove the last statement, which was widely believed to be true, but we might have to wait for the write-ups before seeing a rigorous proof of it. In the attempt more formulae similar to the one at the top of this handout we derived, and JDU said that formulae for the increase of abundance obtained by adding factors of a prime are very useful for inclusion in a program. DP stated that in the prime expansion of a Phillips number the exponent of a small prime should be about the log to the base of that prime of the largest prime in the expansion. Proof by inspection. . .

SR. posed two questions which were quickly answered by the class: can successive Phillips numbers have decreasing powers of small primes? (yes) and about how many different prime factors does a large Phillips number have? (log n/ log log n proved in the last class).

While discussing how to obtain a Phillips number from previous ones, SR came up with a conjecture: each Phillips number can be expressed as the product of a single prime and a smaller Phillips number. Again this was widely accepted as true, but not proved. AT provided a counter-example which was refuted, then he proved the conjecture in the case that the Phillips number has a prime factor that appears in no smaller Phillips number:

Let $\rho = pl$ be a Phillips number, with p a factor of no previous Phillips number. If l is not a Phillips number then there is some Phillips number m such that $m < l$ and $Ab(m) > Ab(l)$,

so

$$Ab(mp) = Ab(m)Ab(p) > Ab(l)Ab(p) \geq Ab(lp),$$

which is a contradiction. SR wondered why the argument doesn't work if *p* is a factor of *m* — the reason is that then $Ab(mp) \neq Ab(m)Ab(p)$.

SJP invented a dubious form of lottery to help divide the class into teams. The results: Team 1 is DK, DP, VG; Team 2 is SR, ET, AS; Team 3 is MG, DC, SQ; and Team 4 is AT, AH.

JDU described the concept of the "profile of a Phillips number" – the exponentially decreasing graph of number of factors versus the index of a prime, in the prime expansion of the Phillips number. He asked if there could be a continuous "optimal profile" if non-integer powers of primes were allowed, and conjectured that a Phillips number would be close to this optimal profile. MG thought that starting from an optimal profile might not help much, as one would still need to search for nearby Phillips numbers.

SR asked if there is an asymptotic relationship between powers of 2 and 3 in Phillips numbers. JDU suggested that perhaps $Ab(2^k) \approx Ab(3^m)$ or something similar.

After some more discussion of DP's idea relating the powers of small primes to the largest prime in a Phillips number, the class concluded that approximate and exact relationships between the prime factors of Phillips numbers are exactly what is needed to guide the search of a program to efficiently seek out abundant numbers.

January 16

Some preclass banter:

> This is a really hard problem. *SR*
>
> You ain't seen nothin' yet! *JDU*

JDU started the class with a thorough algorithmic treatment of the problem of multiplying large numbers of primes to display very abundant numbers. Let's say we have *n* k-bit numbers. An obvious way to form their product is to multiply the first two, then multiply the product by the third, then by the fourth and so fifth. A problem with this is that if we aren't using exact arithmetic, errors in early multiplications propagate a long way. A better way is do the multiplications as if the numbers were the leaves of a binary tree.

Let $M(x, y)$ be the time taken to multiply a x-bit number by a y-bit number. The linear approach uses time

$$\sum_{i=1}^{n-1} M(k, ki),$$

while the divide and conquer (tree) approach uses time

$$\sum_{i=1}^{\log n} \frac{n}{2^i} M(k2^{i-1}, k2^{i-1}).$$

If $M(x,y)$ = xy then both are $\Theta(k^2n^2)$. However the situation changes when $M(x, y) >$ xy. The Schönhage-Strassen algorithm does multiplication in just over *n* log *n* time, but is the best algorithm only for numbers of more than 25000 bits. This is experimental evidence

based on Doug McElroy's work on calculating π to a gazillion bits (JDU); McElroy took about a month to implement the algorithm.

A simpler fast multiplication algorithm is due to Karatsuba and Offman, and works better than the trivial algorithm for numbers of more than 500 bits. If we have 2 $2k$-bit numbers AB and CD, the product is

$$(AC)2^{2k} + (AD + BC)2^k + BD.$$

Doing the multiplications in the obvious way gives us the recurrence $M(2k) = 4M(k) + O(k)$, which is no improvement. However we can get by with only 3 multiplications:

$$(A + B)(C + D) = AC + AD + BC + BD,$$

so we have

$$(AD + BC) = (A + B)(C + D) - AC - BD.$$

We now have the recurrence $M(2k) = 3M(k) + O(k)$, so $M(k) = O(k^{\log_2 3})$, and log, 3 is about 1.59.

In the tree approach the last term in the sum dominates and is about $(k2^{\log n-1})^{1.59} = \binom{kn}{2}^{1.59} = O((kn)^{1.59})$.

In the linear approach the numbers we are multiplying are not the same size, but we have $M(k, ki) \leq M(k, k) = ik^{1.59}$, so the time in this case is $\sum ik^{1.59} = O(k^{1.59}n^2)$. We see that the tree scheme is better if we are using fast multiplication.

SQ asked if this was useful for this problem, since one can use floating point numbers. He asked how to do comparisons efficiently between numbers represented by their prime factors, and why we would need to multiply our numbers out anyway. JDU replied that floating point overflow will be a problem because of the large numbers this problem deals with, and that comparison should probably be done by cancelling common terms and then multiplying out. Multiplying out will also be useful for presenting the very abundant numbers. (Impress your friends with your pages of digits, or cover the walls of your bedroom with a 25-abundant number. Trivia question: How abundant a number do we really need to cover the walls of a typical dorm room?)

SR asked if lisp systems use fast multiplication for their infinite precision rational arithmetic, and MG said it has been done in some experimental systems, but not on anything we have available.

MG has been using exact rationals, but has hit some combinatorial walls, so his team are working out what precision is needed in the calculations, as a function of the desired abundance. Their method is to examine the rationals that will arise in the computation, and use enough precision to distinguish between them and to represent their differences. JDU warned that he thinks one may need 2" bits to represent the difference between two n-bit rationals, or at least $2n$ bits.

SR decided we should ignore implementation details for a while, and said can we talk about how to solve the problem. Each team described their current approaches to the problem:

Team 2 which is SR, ET, and AS spoke first. SR described how they have focused on the problem of optirnising k and j in $n = p^k q^j$, keeping n constant.

$$AB(p^r n) = Ab(n)(1 + \sigma(p^r, k)), \text{ and}$$

$$AB(\frac{n}{q^s}) = \frac{Ab(n)}{(1 + \sigma(a^s, j-s))},$$

where $\sigma(a^i, j) = \frac{a^i-1}{a^i(a^{j+1}-1)}$. JDU observed that σ is a curious function, and is not well defined for some arguments; SR explained it as a notational convenience, apologised and continued:

$$\begin{aligned} Ab(\frac{p^r}{q^s}) < Ab(n) \text{ iff } & \frac{1 + \sigma(p^r, k)}{1 + \sigma(q^s, j-s)} < 1 \\ \text{iff } & \sigma(p^r, k) < \sigma(q^s, j-s) \\ \text{iff } & \frac{1}{p^{k+1}-1} < \frac{1}{q^{j-s+1}-1} \\ \text{iff } & q^{j-s+1} < p^{k+1} \\ \text{iff } & j < \frac{\log p}{\log q}(k+1) + s - 1 \\ \text{iff } & j \leq c(k+1) - 1. \end{aligned}$$

The last line follows from letting s tend to **0.** Similarly if we consider $Ab(\frac{q^s}{p^r}n)$ we **get** $j \geq c(k + 1) - 1$. *So* $j + 1 = c(k + 1)$, **or** $\frac{j+1}{k+1} = \frac{\log p}{\log q}$, i.e. the exponent + **1** is inversely proportional to the log of the prime. Thus the continuous profile has f(p) $= \frac{f(2)+1}{log_2 p} - 1$, where **f(p)** is the exponent of p.

SR stated that in the discrete case this analysis gives a band of about 3 possible values for f(p) given **1** exponent. This relates to MG's attack on the problem, where he used two exponents (of 2 and 3) to bound the choices for exponents of larger primes.

Team 1 which is now DK, SQ and VG. DK explained that they set the largest prime, and used it to restrict the size of the exponents of smaller primes. They still have exponential blowup in work with increase in abundance, but JDU thinks this is unavoidable. SQ calls their method "horizontal bounding". Let p_r be the largest prime, and let $p \approx \sqrt{p_{r+1}}$. Consider the largest pair of primes with exponents of at least 2: if their product is more than $p_r + 1$ then removing one factor of each and multiplying by $p_r + 1$ reduces the size of the number, and increases the abundance (this last fact was only implied, check it). This gives us the estimate that the point at which the exponents switch from 1 to 2 is around $\sqrt{p_r}$. JDU conjectured that the point where the i'th step occurs is about $p_r^{1/i}$. SQ noted that this analysis can be done between each pair of steps to further limit the search.

Team 3 now consisting of MG, DC, DP. (Tearn changes were due to language incompatibilities.) MG said they were trying to calculate numbers of high abundance without calculating all smaller Phillips numbers. They are not optimistic about getting more than H-abundant Phillips numbers. They can find all Phillips numbers up to abundance **11:** there aren't too many. There are only 482 Phillips numbers of abundance less than 9, and somewhere around 800 of abundance less than 10. JDU said there seems to be a fundamental exponent of growth of Phillips numbers in terms of abundance, of the number of Phillips numbers, and of growth of the number of primes in a

Phillips number. Are the exponents the same, or related in a simple way? SQ noted that the continuous profile can be used to get very abundant numbers (not necessarily Phillips numbers) very easily.

VG derived a fact that he found while trying unsuccessfully to prove SR's conjecture: Let $p^k | \pi_n$, and $p^{k+1} \not| \pi_n$. Let π_r be the largest Phillips number less than π_n/p, and assume that π_n/p is not a Phillips number. Then $Ab(\pi_r) \geq Ab(\pi_n/p)$, and $Ab(\pi_r p) < Ab(\pi_n)$. Suppose that t is the exponent of p in π_r. Then

$$Ab(\pi_r p) = Ab(\pi_r) \frac{p^{t+2} - 1}{p^{t+2} - p} < Ab(\pi_n),$$

$$Ab(\pi_n/p) = Ab(\pi_n) \frac{p^{k+1} - p}{p^{k+1} - 1} \leq Ab(\pi_r),$$

and so

$$Ab(\pi_n) \frac{p^{k+1} - p}{p^{k+1} - 1} \frac{p^{t+2} - 1}{p^{t+2} - p} < Ab(\pi_n)$$

so that $k < t + 1$, i.e. $k \leq t$. Put into words, if π_n is not p times a previous Phillips number then the largest Phillips number less than π_n/p has at least as many factors of p as π_n.

The last order of business was computing resources. JDU announced that some workstation managers would each give one student an account on their workstations. Interested students should send a request to mumick@cayuga, gangolli@wolvesden, phipps@solitary or plotkin@goblin.

January 18

Today we discussed the progress of all the groups, and JDU started a riot by providing a simple way to generate lots of large Phillips numbers. First the progress reports:

Team 3 (MG, DC, DP) ran a program overnight, to look for H-abundant numbers, but forgot to set a flag to limit the precision used in the calculations, so had no success. They determine bounds on the abundance of a number using a certain number of primes, then search for the first number with more abundance, which must be a Phillips number. They have encountered some of the combinatorial problems inherent in the search for abundant numbers — one cannot get around the fact that the number of primes needed is exponential in the abundance. (Experimentally they have found that the number of primes needed for abundance k is about F_k, the k'th Fibonacci number.) They are trying to avoid other problems stemming from doing calculations with large infinite precision numbers, and are still working on ways to bound the precision needed for floating point operations. SQ proposed they use logarithms of the large quantities, both to keep sizes reasonable and to speed operations by replacing multiplications by additions.

The other teams all seem to be working from the continuous optimal profile. MG pointed out that the Phillips numbers jiggle a lot around the profile. DK and MG tried to prove bounds on the amount of jiggling, without success. Team 2 (SR, AS, ET) noted that the continuous curve f(p) = $\frac{c}{\log p} - 1$ dips below the z-axis, giving a cutoff point for the last prime. One can calculate a formula for c in terms of abundance, or do binary search. The resulting profile gives a lower bound on the size of a number with the required abundance, so we have a good starting point for searching. AH asked if a Phillips number can always

be obtained by one of the 2' jiggles (rounding up and down) around the profile (no). AS truncates each prime, then increments primes which have been truncated the most until the desired abundance is reached. He conjectures this gives Phillips numbers (the famous Schwarz conjecture), but doesn't always get the *first* number with the required abundance. There was general agreement that this "tweaking the profile" approach isn't really producing the goods.

When JDU first thought about the problem about a decade ago he thought "Aha! A knapsack problem." We have a collection of items, the primes p_i (the i'th occurence of the prime p). We have derived formulas for the relative abundance increase arising from another factor of a prime. Let $v(p,i)$, the value of p_i, be the log of this abundance increase. Let $w(p,i)$, the weight of p_i, be logp. The task is to pack the knapsack, whose weight capacity is the maximum size number we want to consider, with as much value as possible.

The greedy technique is to add items with the best value to weight ratio — this works well at first but isn't good when the knapsack is almost full. An interesting point is that the sequence of numbers produced by the greedy algorithm consists only of Phillips numbers. This is easily seen by looking at a graphical representation of the problem. Label the x-coordinate with weight, and the y-coordinate with value/weight. We can represent a number by drawing a rectangle for each prime factor, with the width being the factor's weight and the height being its value/weight ratio. The value (log of the abundance) of the number is the total area of the rectangles. If we add items (prime factors) in the greedy order, what we get is an infinite descending staircase. The number at the end of any stair must be a Phillips number, because all unused items have a smaller value/weight ratio than the chosen items, so replacing chosen items by unused items will lower the graph, thus reducing the total abundance.

The TA lost track of the 10 simultaneous discussions at this point. Some people found it hard to believe that there is such an easy way to generate guaranteed Phillips numbers; JDU had to resort to proof by intimidation.

The combinatorial problems arise again when we wish to find the *first* E-abundant Phillips number. One way is to use the greedy algorithm to get close to the required abundance, then to fiddle with the remaining space in the knapsack. DK thinks it is dumb to reduce the problem to a NP-complete problem; the question now is whether one can find a polynomial time algorithm algorithm for this particular knapsack problem.

January 23

JDU started the class with a proof that the greedy algorithm produces Phillips numbers, similar to the proof given in the class notes of 1/18. The numbers generated by the greedy algorithm form a strict subsequence of the Phillips numbers, which the class dubbed the Ullman numbers. JDU conjectured that the Ullman numbers are pretty sparse in the Phillips numbers. DP observed that experimentally there are not many Phillips numbers per new prime. There is at least one Ullman number with each new prime, so the Ullman numbers in fact form a significant proportion of the Phillips numbers. SR commented that experimentally it seems like not much tweaking is needed to get from an Ullman number to a nearby Phillips number. AT tried unsuccessfully to prove that adding a greedy step to a Phillips number always produces a Phillips number.

Todays progress report is summarised in the table below. Team 2 are searching only for

Ullman numbers so far, as their method based on the discredited Schwarz conjecture led up a blind alley. Team 3 can achieve greater abundance, but are still concentrating on the problem of guaranteeing numerical safety without infinite precision. Team 4 is using the SR-hypothesis.

Team	Abundance	Runtime	Comments
1	first 22	13 hr @ 10 mip	$2^{20}3^{12}5^{8}7^{7}\ldots13^{5}\ldots83^{3}\ldots661^{2}\ldots231431.$ 3 x 10^9 numbers searched
2	23.00048	15 mins	Ullman number
3	first 18	1.5 hr @ 1 mip	$2^{18}3^{10}5^{7}7^{6}\ldots13^{4}\ldots37^{3}\ldots211^{2}\ldots24499$ Searched 4 x 10^5 numbers
4	first 11	8 hrs	$2^{10}3^{6}5^{4}\ldots29^{2}\ldots487$ Found all smaller Phillips #'s

Team 3 uses horizontal and vertical constraints to limit searching, so has a search space about a hundredth of team 1's, and their program is roughly ten times faster.

Numerical errors are significant in this problem — a number whose abundance is calculated as 23.00000001 might not even be 23-abundant. The absolute error in adding n numbers is about $\sqrt{n}$ times the error in each (JDU). DP asked how difficult it is to get accurate logs. According to SQ most Unix systems guarantee accuracy to the last bit. Some systems have an e-log routine, for taking $\log(1+\epsilon)$, but the class found that it disagreed with the standard log and was thought to be faulty.

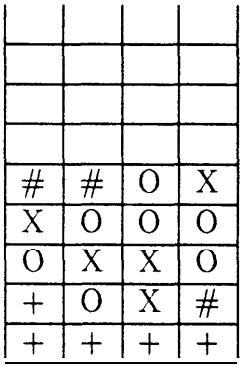
We then started on problem 2, tiling a hall, the topic of the next chapter.

Chapter 3

Tiling a Hall

January 23 continued

The problem can be summarised as follows: there is a hall of width 4. We are given a collection of tiles (subsets of a 4 x 4 square) and a hall length, and must decide whether the hall can be completely covered with tiles. Note that the 4 x 4 squares that contain the tiles can and will overlap, but the tiles cannot overlap, though they will interlock. For example, if #, X, 0 and + represent different tiles, then a fully tiled hall of length 5 could be:



JDU described how the problem can be viewed as an automaton problem: the state is the contents of the top 4 rows giving $2^{16} - 1$ states. (SR: "Oh, this is easy. 2^{16} is small.") A transition corresponds to laying a tile in the first 4 unfilled rows of the hall — if the hall is tilable, then it can be tiled with a sequence of tiles each of which helps fill the first unfilled row of the hall. The nFA will have several choices per state. There is 1 input symbol, 0, and a the label of a transition is the number of rows of the hall that are completely filled by that transition. In the above example, if the 0 tile was the last to be laid down, the transition would be labelled with 4 0's.

Al Aho gave a CSD colloquium talk recently about Bell Labs' experience with grep. The problem is to generate automata for pattern recognition. Before working on Unix, Thompson wrote an efficient nFA simulator, but found that converting to a dFA first to allow a quick scan worked better most of the time. On some strings (such as "a....." which matches an 'a' a certain distance from the end of a string) the dFA has an exponential number of states. The current grep, egrep, creates a dFA lazily, producing states only when they are needed, and buffering recently used states.

DP asked if the question is to minimise the number of holes in a hall (no, just to answer yes or no to "is a hall of length n tilable?") JDU pointed out that doing an nFA simulation

is fine if the required hall length is *35,* but not if it is 35 gazillion. If arithmetic can be done in time independent of the size of the numbers (laughter in the class) then there is a constant time (independent of *n* but depending of the tiles) solution to the problem. SR said we need a kind of pumping lemma, and AS realised we will need to find cycles in the nFA. AH noted that the simple cycles will have less than 2^{16} states. MG noted that often all long halls will be tilable, and AT suggested using a Mealy machine, with transitions on tiles, that outputs 0 when a row is full.

JDU said we will be looking for cycles with a fixed number of 0's. DK and SR asked why one can't use a connectivity test to see whether the final state (hall full) is reachable from the start state (hall empty). The reason is that this answers "is a hall of any length tilable?" instead of "is a hall of length *n* tilable?"

SJP did a random team assignment: Team **1** is SR, DC. Team 2 is MG, DP, ET. Team 3 is AS, AH, DK. Team 4 is VG, AT, SQ.

January 25

JDU led the class with a discussion of the use of closed semirings in path-finding problems. A closed semiring is an algebraic system based on a ring. It consists of a domain D, +, and x. + is infinitely associative and commutative, while x is asssociative and distributes over infinite sums: $a \sum_{i=1}^{\infty} b_i = \sum_{i=1}^{\infty} ab$; if $\sum_{i=1}^{\infty} b_i$ exists. Similarly for multiplication on the right. 0 and 1 are the identities for + and x respectively. 0 is a x-annihilator. The Kleene star operator $a^* = 1 + a + a^2 + \ldots$ makes sense.

We wish to consider a graph labelled with elements of the domain, and compute

$$\sum_{\text{path } p_{n \to m}} \text{label(p)}$$

where label(p) is the product of the labels along the path **p** from *n* to m.

Some examples of closed semirings are:

- The Boolean domain $D = \{0, 1\}$, + = V, ⊠ = A, $a^* = 1$. In this case the above sum is 1 iff the nodes *n* and m are connected.
- D is the non-negative reals, + = minimum, x = addition, so that $a^* = \min(0, a, a^2, \ldots) = 0$. The sum gives the length of the shortest path from *n* to *m.*
- D is the set of regular of expressions, + is union, x is concatenation, and a* is the usual Kleene closure. + has identity ϕ while x has identity ϵ. The sum gives a regular expression for all paths from *n* to *m.*

JDU went on to say that probably none of these closed semirings will help us with our hall tiling, but he does think that there is a structure that will be helpful.

Next he described the Kleene algorithm for calculating the above sum for an arbitrary graph. Take a node k, its arcs labelled a_i from its predecessors p_i and arcs labelled b_j to its successors s_j, and perhaps a self-loop labelled c. (If there is no self-loop we can insert one labelled with the multiplicative identity). For each arc u from a predecessor p_i to a successor s_j replace its label d by $d + a_i c^* b_j$. Some care must be taken if u is a loop. This procedure effectively eliminates the node k, which can therefore be removed from the graph. We continue this till we are left only with nodes n and m.

SR noted that if a set of tiles can tile hall lengths a and b with $(a, b) = 1$, then they can tile any hall beyond a certain length, certainly beyond ab. VG proved this: consider the numbers $0, a, 2a, \ldots, (b-1)a$: since (a, b) = 1 this forms a complete residue system modulo b, so adding multiples of b to these numbers gives all numbers past $(b-1)a$.

Thus the problem reduces (SR) to finding the hcf of all cycle lengths, then determining the threshold at which all multiples of the hcf are tilable. This gives a constant time solution (assuming arithmetic is constant time). JDU asked how many equations must be solved, and how high the threshold can be. We are writing programs, not just theorising, so the constant in "constant time" matters.

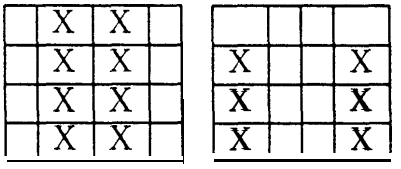
AT did some calculations on the number of states in the nFA. Recall that the state encodes the contents of the top 4 rows of the hall. The first row cannot be full, so 2^{12} states are eliminated. If the bottom row is empty the top row must be empty, eliminating 2^8 states. If the top row is not empty then the leftmost bottom square must be full. This works out to $2^{15} - 2^{11}$ states.

MG and DP form all combinations of tiles inside the 4x4 square. They can then fill the bottom row on each move, so the state needs to encode only the top 3 rows of the hall, giving 2^{12} states. The number of composite tiles is no more than (the number of tiles)*, and is certainly no more than 2^{15}. SR noted that this seems equivalent to forming the original nFA and collapsing the e-moves. MG says it is doing the collapsing once instead of many times.

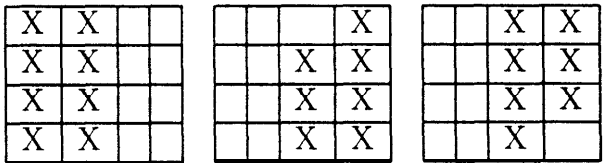
DC thinks that one shouldn't do all this work for easy instances of the problem. JDU asked if there are ways to try for an easy solution first, before constructing the large nFA.

Everyone decided they wanted a time trial at the end of the fortnight, where each group brings a collection of tiles to test everyone else's programs. A tentative date for the competition is Tuesday February 8.

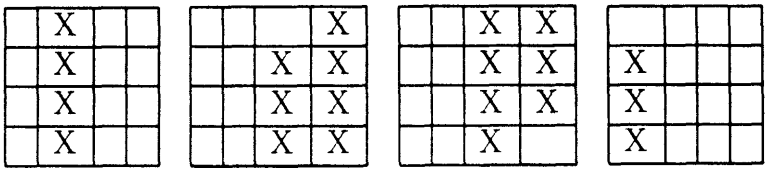
JDU thinks that finding the hardest set of tiles (those with the largest finite minimum tilable hall length) may be harder than than the original problem. AT gave some small sets of hard tiles. A pair of tiles with minimum hall length 12 is:



Three tiles with minimum hall length 28 are:



Lastly four tiles with muinimum hall length 84 are:



January 30

Class started with abundant numbers again. ET, SR and AS described their program for finding the first k-abundant numbers. They generate a list of millions of primes, then a list of millions of Ullman numbers, and then do efficient search to find the smallest k-abundant number for any $k < 32$. The speed of their approach is limited mainly by the generation of primes, which is the best we could hope for. Search is restrained by finding the first k-abundant Ullman number as a first guess, then using the current best guess to severely limit the choice of factors that can be deleted from a number or added to a number such that the number remains feasible (such that other factors can be deleted or inserted to regain a number smaller than the best guess, with abundance > *k).* For π_{21} 1398 numbers are searched, while π_{23} requires 336 numbers to be searched, with only 3 updates to the best guess. π_{28} takes about 5 minutes to calculate (most of the time spent reading in the list of Ullman numbers) and contains about 7 million digits.

Before we say goodbye to abundant numbers, a correction needs to be made to an earlier set of class notes. The offending line was

> "[Team 2]'s method based on the discredited Schwartz conjecture led up a blind alley."

AS pointed out that

> the "Schwartz conjecture" says that you can come up with a really good subsequence of Phillips numbers by starting with the number one and adding successive powers of primes in the order in which they help you achieve high abundance relative to size, where the helpfulness of prime p at power k is given by the coefficient of the optimal curve on which (p,k) lies.

The conjecture thus appears to be no more than a statement that the greedy technique works.

NOW on to problem 2. SQ asked that the hall length be limited to 2^{31}. SR said that the Kleene algorithm will give a variable for each minimal cycle in the graph. MG has been trying to reduce the nested expressions, like $(4 + 6^*)^*$ given by the Kleene algorithm, without success yet.

JDU introduced semilinear sets. Consider vectors $[a_1, \ldots a_k]$ over integers. A vector addition system (VAS) is a set of vectors $v_1, \ldots$, v,. We are given a vector w and asked for non-negative integers $b_1 \ldots b_n$ such that $\sum b_i v_i = w$ The decidability of this problem was open for a long time, till Ernst Mayr found an algorithm, whose time seems non-elementary (not bounded by any tower of 2's). Fortunately we need only consider VAS's of dimension 1, i.e. semilinear sets. Given a basis v_0 and periods $v_1 \ldots v_n$ then $[v_0 : v_1, \ldots, v_n]$ represents $\{ v_0 + \sum_i b_i v_i \mid \text{integers } b_i \}$. Actually only 1 period is needed, as a semilinear set is a union of semilinear sets with 1 period. The point of all this is that the expressions generated by the Kleene algorithm are semilinear sets. Thus we need to develop routines for manipulating these sets (especially for reducing expressions like $(a + b^*)^*$ as soon as they appear).

SQ said it isn't obvious how to reduce such an expression efficiently; said JDU, "I'm not going to write your code for you!"

DP asked how to turn an arbitrary graph into a set of tiles. JDU described a method: for each arc $i \rightarrow j$ in the graph, add a tile with j in binary in the top half, and i in binary in

the bottom. This lets us encode an arbitrary graph of up to 255 vertices. AS noticed that the tiles could interfere in unexpected ways, which we could avoid if each tile had two 1's in the bottom of the leftmost column. This lets us safely encode graphs of 64 states. JDU pointed out that a weight 4 code will prevent interferences (each number is coded with by a number with exactly 4 1's) giving us 70 states to use.

DP asked if one can get a graph that somehow encodes an NP-hard problem, such as forcing the programs to look for a Hamiltonian path when given a certain hall length. JDU thinks not. If semilinear set operations are done in constant time then there is a cubic time algorithm. The question then is how fast the semilinear sets can grow.

We philosophized about the difficulty of the problem for a while. VG think this problem is harder to get started on than problem 1. JDU and SJP still aren't sure how hard a set of tiles can be. JDU thinks that there is a program that will work quickly in practice. SQ noted a trade-off: if there are lots of tiles there will be lots of interference, so an easy solution. JDU thinks the semilinear sets won't blow up in practice, as the simplifications will be taking gcd's of lots of numbers. JDU suggested using a brute force program and a complicated solution running in parallel, to give a fairly quick solution for both easy and hard instances of the problem.

One last correction is in order: the idea of combining tiles before constructing the NFA was wrongly accredited in the last class notes. The idea is in fact due to ET.

February 1

The first topic of discussion was the difficulty of the tiling problem. DC finds it similar to AC-unification (unification with some operators associative and commutative) which is NP-hard. He has been trying to reduce the unification problem to the problem of determining if two NFA graphs are equivalent. JDU thinks the equivalence problem is easy: form the cross-product automaton, each state of which corresponds to a pair of states, one from each automaton. The equivalence problem reduces to determining whether any pairs of the form (final, non-final) are reachable in the product automaton.

SQ described some attempts at reducing expressions representing hall lengths. The notation derives from production systems. e is the non-terminal representing hall lengths. He considered expressions given by:

$$e ::= 1^p \mid (1^q e)^*$$

or

$$e ::= (1^p (1^q)^*)^*.$$

The latter reduces to

$$(1^p)^* \mid (1^{p+q})^* \mid (1^{p+2q})^* \mid \ldots \mid (1^{p+(n-1)q})^*$$

DK used some simpler notation to describe reduction of expressions given by

$$e ::= e^* \mid e_1 \vee e_2 \mid e_1 + e_2$$

She noted that

$$(a \vee b) + c = (a + c) \vee (b + c),$$

$$(a \vee b)^* = a^* \vee b^*, \text{ and}$$

$$(e_1 + e_2^* + e_3^*)^* = 0 \vee (e_1 + e_1^* + e_2^* + e_3^*).$$

Any expression given by the above productions can be reduced to an expression of the form

$$0 \vee (e_1 + e_1^* + e_2^* + e_3^* + \ldots) \vee (f_1 + f_1^* + f_2^* + f3^* + \ldots) \vee \ldots$$

Both DK and SQ are describing how to evaluate the result of operations on semilinear sets. JDU's notation for semilinear sets may be useful here.

MG described how he implemented these semilinear sets. He represents a semilinear set as a table of small integers and all multiples of a single factor beyond a threshold. When reducing a semilinear set to this form, he keeps a table of small integers modulo the length of the smallest cycle. More precisely, to represent all values of the expression ax + *by* + cz, say with (a, *b, c)* = 1, a < *b* and a < c, he steps through all low values of the expression, till all conjugacy classes mod a have been obtained. All integers beyond the last generated value are values of the expression. A similar result holds if (a, b, c) $\neq$ 1. An example: 6x + $10y$ + 152 generates the integers $0, 10, 15, 20, 25, 30$ and 35, which span all the conjugacy classes of 6, so all integers greater than 35 are values of the expression. The runtime of this algorithm is quadratic in the size of the smallest cycle in the equation.

If we implement Kleene's algorithm always working with the smallest cycles first, we need only small tables to represent all the sets, and the semilinear set operations should be efficient.

DC and SR described a method based on forming an equation for each set of cycles in the graph (hence an exponential number of equations). Each equation is of the type *ax* + *by* + $cz = n$, where a, b, and c are cycle lengths. If we are given a value for n, we must find non-negative integer values of the variables x, y and z that satisfy the equation. In two dimensions we can draw a graph corresponding to the equation, and we wish to find a lattice point in the first quadrant through which the graph passes. SR generalised this to higher dimensions, and thinks that finding an integer solution is exponential time (in the number of variables or the cycle lengths?)

VG said there is a well known algorithm for solving such equations.

DP starts with the state space of size N, and each arc labelled 1. The graph of the NFA can be easily converted to this form, assuming we form combinations of tiles before constructing the NFA. For each state he records all the distances $< k$ in which that state can be reached from a fixed state. This takes $O(Nk)$ time. In a similar way in order N^2k time one can record all the cycle lengths $\leq k$ through each node. Then there is a number c, $N^2 \leq$ c $\leq N^2$ + N, such that all tilable hall lengths are of the form c + $\sum_i$(cycle-length$_i$) where the sum is over any collection of cycles in the graph. The reason is that the whole graph can be traversed once within c steps.

Chapter 4

Constructing a Star Map

February 6

> I assume you want to talk mainly about progress on problem 2 today. *JDU*
> Progress? What progress? *SR*

That said, we moved on to problem 3. In Bob Floyd's original version of the problem the cluster was spherical, probably with a Poisson distribution for the radius and uniform distribution for the angle. One was given 2 views, and had to work out where the second view was from. SR noted that as long as there are lots of stars, and they are distributed reasonably, then the distance of the second viewpoint can be easily determined by the apparent size of the cluster. MG noted that if one knows the position of a reference point, say the corner $(0,10,0)$ of the cluster, then the density of stars around that point gives an indication of the viewer position. AS pointed out that the edges of the view will be corners of the square.

After making the cluster, JDU jitters the position of each star's image on the film by 10^{-6} of a unit. The jitter is uniformly distributed in a cube of side 2 x 10^{-6}. The error is independent for each view. Each star image then defines a tube of radius 10^{-5} inside the galaxy within which the real star can lie. We get a possible error in combining the two views if a tube crosses two tubes from the other plate.

DK described the problem in terms of a bipartite graph, with a node for each star image on each plate, where an edge is drawn between two images if they are on different plates and their tubes cross. What we are looking for is a kind of minimum weight matching in this graph, where the weight of an edge depends on the extent to which two tubes intersect. AH suggested RMS weighting. DK noted that one can get a partial matching by a greedy method, then jiggle it (by augmenting paths) giving a polynomial time (in 10 000 stars) solution. Further comments were lost in the ensuing chaos.

JDU showed that one cannot resolve 2 stars that are coplanar with both viewpoints — one gets a quadrilateral where we don't know at which pair of opposite corners the two stars lie.

JDU has a program that generates a cluster of any number of stars with any jiggle size. The output gives the earth and Trantor co-ordinates of each star together, so the correct answer is known, and the programs' results can be easily checked against the correct solution. SQ asked whether it is better to give an impossible solution that has lots of good matches,

or a bad feasible solution. The answer is that we will only know once both strategies have been implemented and tested on random clusters.

With 10000 stars there is about a 10% chance of error for each star. The probability that a tube of a star from plate A meets a wrong tube from plate B is the probability that 10000 bullets of radius 10^{-5} hit a tube of width 10^{-5}. The total bullet hole are is 10000 x $(10^{-5})^2 = 10^{-6}$, while the tube area is 10^{-5}, so the probability of a bullet hitting the tube about is 1 in 10. This has been verified experimentally.

SR decided that the JDU's version of the problem is too easy, and wants to try Floyd's version. Said DK, "I like a problem once in a while that we can actually solve."

VG asked if the mystery viewpoint will be at a lattice point. JDU: no, but the size of the image limits the possible viewpoints. How many exact points are needed to determine the viewer position? 3 and 4 were popular guesses. The shape of the image gives hints about the orientation. If the cubic case is too easy JDU will construct some spherical clusters. The clusters for this problem will contain about 100 stars, which might not be enough to define the cubic shape well.

SR suggested dividing the plates into subsquares, and using the density of the subsquares in the two views. JDU asked if one can get from a small feasible region for the second view to a smaller region by local approximation. AT brought up the orientation of the camera. It was agreed that the bottom of the plate should be parallel to the xy-plane, and there should be no views from directly above or below.

SQ suggested chunking views (with the chunk size depending of the jiggle size), then using the solution to the first problem to determine the feasibility of each of a finite number of views, and choosing the best match.

DP said that FFT's (Fourier transforms) may be useful. Maybe small moves only change some of the frequencies in the transform.

Teams were selected: (1) SR, MG, DK. (2) DC, AT, AS. (3) AH, SQ, DP. (4) VG, ET.

There was a little discussion on problem 2. SQ thinks he can make some nasty sets of tiles. DP thinks he can't. Test2 and Test3, JDU's test sets of tiles, turned out to be easy. The example random set of tiles turned out to tile every hall length. All the groups will have programs running by Thursday, though most aren't confident about using them in a competition.

February 8

We did a little elementary geometry today, determining the distance between rays from the two viewpoints Earth and Trantor. Let a star be seen at (x_1, z_1) from Trantor, and let a star be at position (y_2, z_2) as seen from Earth. The view from Trantor is along the y axis, and the Earth view is along the x axis, so the two rays are parametrised by $(x_1 t, t, z_1 t)$ and $(10 - s, 10 + y_2 s, z_2 s)$. The distance between a pair of points, one on each line, is the square root of

$$(10 - s - x_1 t)^2 + (10 + y_2 s - t)^2 + (z_2 s - z_1 t)^2.$$

We want to minimise this expression; its two partial derivatives are

$$2(-(10 - s - x_1 t) + y_2(10 + y_2 s - t) + z_2(z_2 s - z_1 t)) \text{ and}$$

$$2(-x_1(10 - s - x_1 t) - (10 + y_2 s - t) - z_1(z_2 s - z_1 t)).$$

Setting both to zero gives us a pair of linear equations in two unknowns, which is easy to solve. Said DP: "We can do this. You don't need to finish it."

So we returned for a last time to problem 2. We had a terminal set up for demonstration of the tiling programs, but only team 2 (MG, DP, ET) had a program ready to demonstrate. Their program sets up the FA determined by the precombined tiles, throws out useless states, derives information about cycles from which it determines the behaviour for all large hall-lengths, and answers small hall-length queries by simulation. The program is fast on small tile sets, but a large random set of tiles makes a huge number of combined tiles and hence a huge FA, and their program is slow in this case.

Getting back to problem 3, SR described a way to determine Earth's position in the hard version of the problem (where Earth's position is unknown). Take three well separated stars near the boundary. They cannot match all triples of stars in the other image — the corresponding images must have separation at least the largest height in the triangle formed by the stars in Earth's view. Thus we can try matching the three stars against all feasible triples in the other view (in each orientation). Each match determines a position of Earth which can be checked for feasibility. DK suggested using density information to reduce the number of triples that need to be considered. SR suggested using 6 stars on the outside of one view, to force three of the stars to be near the outside on the other view, thus eliminating most of the search.

ET and VG noted that if the stars are symmetric in the cluster there will be lots of "correct" orientations or positions for Earth. However all we are looking for is one correct position — we don't care how many there are.

There was a lot of loud and fractured discussion about using the clustering of stars to determine the Earth's position. AH thinks there won't be large clusters. JDU gave numerical support for this: if n darts are thrown uniformly at a dart board that is divided into n blocks, the expected largest number of darts in a block is about $\frac{\log n}{\log \log n}$.

AH suggested first solving a l-dimensional version of the problem, where all views are from the xy-plane. JDU thinks this is not much easier than the general problem. He suggested scattering test views on a sphere, guessing which one is best, then moving small distances while improving the correlation of the two views. Does this converge to a solution? Are there local minima? (AT, DC: Yes. If the cluster is roughly symmetric there will be local minima across lines of symmetry.) How do we rate the guesses? AH thinks one must be careful rating the guesses — if the measure of goodness is the sum distances of the tubes then an infeasible solution looks too good. AH suggested calculating the effective jiggle in the star positions induced by moving the viewpoint a little.

February 13

Some more progress has been made on the tiling problem. DP and MG improved their program. DK's program works well on small sets of tiles, but is slow if there are lots of cycles. MG described how the gcd of all cycle lengths can be obtained by simulation to length $2n - 1$, where n is the number of states in the graph: for any cycle length k in the graph, there is a cycle through the start node of length no more than $2n - 1$, including the cycle. Then to answer queries we would need to simulate out to at most n^2, after which every multiple of the gcd will be tilable.

JDU related the ideas behind the test cases: test 2 has lots of cycles, and was designed to

kill a program that tried to solve the problem by conversion to a deterministic automaton. Tests 1, 5 and 6 were meant to be one cycle with a single cycle branching off it, with the cycle lengths relatively prime. Errors in the encoding made some smaller cycles possible, but the current versions should be as intended. Tests 3 and 4 are random, and are supposed to have huge state spaces, though 3 is trivial and can tile any hall length.

AT SQ and VG implement the Kleene algorithm on the fly, lazily. They keep a variable for each state, and equations in terms of these variables determined by the edges in the graph. For instance $A = C + 1B$ would represent a state A with null transition to C and transition on **1** to B. They repeatedly expand the start state, solving recurrence relations whenever they appear. This gives a starring operation, which can be evaluated using tables as described by MG. Variables are expanded in a breadth-first manner, the shortest terms being expanded first. This technique is having problems as it is running out of memory.

Returning to problem 3, SR introduced a novel way of looking at the problem. Consider the plane through the center of the cluster and the 2 viewpoints. If we are dealing with orthographic projections, instead of perspective views, then the distance of a star from this plane is independent of the viewpoint. Since the bottom of the camera is aligned with the xy-plane, the position of the described plane determines 1 or 2 possible positions of the unknown viewpoint. Hence we can rotate a line (the image of the plane) around the center of each view, and have a probable match when the function relating the number of stars to distance from the line is the same in the two views. In other words we transform a view into the density function of number of stars in terms of distance. Finding the viewpoint can then be done without matching individual stars, instead just counting the number of stars in bands. SR doesn't know how much the distribution changes as the lines are rotated. He suggested making the stars fuzzy, so that the distribution changes continuously as a function of the angle of the line.

DP asked about FFT's and JDU talked about the Fourier transform. Given a time sequence al, $a_2, \ldots$, say a digitised sound signal: the n'th frequency is given by $\sum_{i=0}^{\infty} a_i \cos(\frac{2\pi i}{n})$. It is large if the signal has a component of that frequency. For example the component of frequency 2 is given by

$$a_1 - a_2 + a_3 - a_4 + \ldots$$

In 2 dimensions this generalises to $\sum a_{ij} \cos(\frac{2\pi i}{x}) \cos(\frac{2\pi j}{n})$. In reality the transform should be done in the complex domain, rather than in the real domain as described, so that we get both the magnitude and the phase shift at each frequency.

SR wants to compute the distance transform for all rotations of the line in a small time, and hopes that it can be done fast, like the Fourier transform.

It is getting late in the fortnight, so some programs need to be written. VG and ET have started programming the simpler version of the problem, where the viewpoints' positions are known.

February 15

SR continued developing his technique for solving the hard star problem. We choose k bands of stars parallel to the line (which is the image of the plane through the two viewpoints and the center of the cluster). A value of the distance transform is a vector of the number of stars in each band; the number of different values is $\leq 2kn$, since each star can cross into or out of each band at most once. We can construct all these values, sort them in the two

views, and in time linear in the number of values we can run up the sorted lists and find the best match. Constructing all the distance values could be done quickly by constructing the set of events (eg. a star crossing into a band), then sorting by angle.

How many bands are needed? The number of stars in a band can be approximated by a normal distribution (if the number of bands is quite small) with variance $\sigma^2 = 2np(1-p)$, where p is the probability that a star will be in the given band. We want to find the probability that the number of stars in this band in different views is equal; if we normalise the distributions to have mean 0, then we want the probability that the sum of 2 normally distributed variables with mean 0 is 0, which if we assume the variables are independent is $\frac{1}{2\sqrt{\pi np(1-p)}}$. The expected number of matches is then no more than $(2kn)^2(\frac{1}{2\sqrt{\pi np(1-p)}})^k$ which is less than 1 for large enough *k*. SR conjectures that choosing *k* around 6 or 10 will be good enough for the problem.

AH pointed out that the normal distribution approximation gets worse as the number of non-overlapping bands increases; if we use overlapping bands then the variables are not independent so our analysis breaks down. We also have variations induced by changes in perspective in the two views, so we would have to make do with approximate matching of the distance vectors. If we get a few almost matches we could check them using a program for the easy version of the problem. VG thinks this approach might get too many matches, as we expect roughly the same number of stars in each band, and perspective induces errors around 10%.

JDU asked if the plane must always appear level in one of the views. The answer is no — take any non-level plane through the Earth view and the center of the galaxy; rotating Trantor around this plane would make Trantor's view of the plane rock back and forth around the horizontal.

VG gave some surprising results for the simple problem. Taking 10 000 stars with 10^{-6} or 10^{-5} error on the plates his team's program found the original star positions. Even with 10^{-4} error the original positions were found with only 3 flips. JDU found many more errors this in his testing and analysis. With 10^{-6} jiggle on the plates each image on Earth's plate defines a tube of width 4×10^{-6} in Trantor's view. The fraction of the total area of the view covered by the tube is 4×10^{-5}, so with 10 000 stars we expect 0.4 stars to hit the tube. VG confirmed that their program had produced possible match lists of average length 1.4, but they found that matching stars to the nearest unmatched star on the other plate, and pairing off the closest pairs of stars first, gave the original cluster without errors. Their program produces a list of possible matches for each star in Trantor, then sorts each list by closeness of the matches, and sorts the Trantor stars by the closeness of their best match. It then runs through the stars matching each with its best unmatched neighbour. Even with 10^{-5} jiggle this simple scheme gave a perfect matching.

They derive the lists of possible matches by considering Trantor in the view from Earth. As seen in this view, the tube of a star in Trantor's view extends from Trantor through the cluster, and the section of the cluster it cuts through is determined by the angle between the ray and the y-axis (the angle between the ray and the bottom of the cluster, in Earth's view). They sort the stars in Earth's view by this angle, or equivalently by $\frac{y}{x+1}$. The possible matches for a star S in Trantor's view are then all stars in a range of angles around the angle defined by the S's ray.

AH asked if the uncanny accuracy they obtain could be because the star positions were

given in correctly sorted order in the input, and don't get permuted during the program execution. JDU suggested that the problem was porting the random star generator between machines; VG's team had also changed the generator slightly (using double precision instead of single precision). Said JDU: "I will send you *my* [jittered and permuted] data, and I bet you won't get it right!" We'll see the results in Tuesday's class.

February 20

Tuesday's enigma was solved — when VG's group changed the error variable in the star generator from single to double precision they caused the error to be read in as 0. With a correct generator they now get 8 flips in 100 stars, and about 1000 errors in 10 000 stars, as expected.

MG pointed out that if the position (0,10,0) (the corner of the cluster) is known in the mystery view then the viewpoint can be easily determined to be one of at most 4 positions. His team is working on the version of the problem where the view is centered on the center of the (round) cluster.

SR discussed matching small numbers of stars in the bands given by his method, to exactly determine the viewpoint. Given two matchable stars at (r_1, θ_1) and (r_2, θ_2) in polar co-ordinates, let the line representing the plane be at angles α and β from the horizontal in the two views. Then the distance from the first star to its plane is $r_1 \sin(\theta_1 - \alpha)$, this must equal the corresponding expression $r_2 \sin(\theta_2 - \beta)$ for the second star. Choosing one more star in each view to pair off gives two equations in 2 unknowns α and β, which could be solved numerically.

MG has programs that display clusters of stars, and generate views of (the center of) clusters from mystery viewpoints. So far his team is only considering orthographic projections.

Again class and problem boundaries do not coincide, so will proceed to the next chapter for the second half of today's class.

Chapter 5

Generalized Hi-Q

February 20 continued

JDU introduced problem **4** by describing some HiQ facts he had learnt from Thane Plambeck. A **1970** paper by De Bruijn gives an easily computable attribute of sets of pegs that remains invariant under peg jumping. Consider the plane colored with three colors so that all three colors are present in any three cells that are horizontally or vertically adjacent. Take the sum over all the pegs of the color of the cell they are in, where the sum of two different colors is the third color, and any color added to itself is 0. The colors then form the non-zero elements of GF(4), the Galois field on 4 elements. $GF(p)$ consists of the integers modulo p, and $GF(p^k)$ is all degree $k-1$ polynomials with coefficients over $GF(p)$. Thus $GF(p^k)$ looks like 0, 1, x, x + 1. To determine the product operation in GF(4) we need to find an irreducible polynomial of degree k over $GF(p)$; in our case x^2 + x + 1 is the only choice. It is easy to see it is irreducible since substituting 0 or 1 for x doesn't give 0. Setting this polynomial to 0 gives us x^2 = x + 1, which determines the product operation. Now the color sum is just $\sum_{\text{pegs }(i,j)} x^{i+j}$. As an example of why this is invariant under jumps: say there are pegs on (i,j) and $(i+1,j)$ but $(i$ + 2,j) is empty. Performing the jump gives a peg only on (i + 2, j), and $x^{i+2+j} = x^2x^{i+j} = (x+1)x^{i+j} = x^{i+j+1} + x^{i+j}$. One can also use the sum $\sum x^{i-j}$.

VG noted that if the sum is 0 then there is no final configuration with only one peg.

JDU asked if one can characterise how sparse a region of the plane must be so that there is no solution. In the 1 dimensional case the set of configurations is a regular set (this is an exercise in Manna's 1974 book). Thane Plambeck has a DFA with about 17 states that recognises the set.

AT had made progress in this direction. If there is a hole of length 3 then the problem is unsolvable, and if there is a hole of length 2 it can only be spanned by moving in tiles from both directions. If there is an even number of pegs in a block on say the left end, they must all be jumped right (otherwise a hole of length 3 or a peg isolated by a hole of length 2 will result). If there is an odd block of pegs on both ends the problem is unsolvable. This gives a linear time algorithm for solving the 1 dimensional case.

Thane Plambeck had a regular expression something like $(11)^*(01)^*00(10)^*(11)^*$ for solvable peg configurations in 1 dimension. He showed that the problem is unsolvable if there are two holes of length 2.

SR asked how hard the brute force simulation method is in the 2 dimensional case (very

exponential). JDU suggested thinking of running things backwards — this gives a grammar for solvable peg arrangements. Unfortunately the grammar is context sensitive. There is a theorem that if all productions increase the size of the expression then the language is context free, but DK noted that the size of the equivalent context free grammar can be exponential.

AT described some progress on problem 2. His team's program still runs out of memory on sets of tiles that give lots of short cycles. He gave some notation for semilinear sets: 3; $0, 7, \infty$ represents $3^* + 7.3^*$. Then 4 . 3; $0, 7, \infty$ gives 3; ∞, 4,11. He noted that the representation is efficient: $|C_1 + C_2| < |C_1| + |C_2|$, $|C^* = 1|$, **and** $|C_1 C_2| = \min(|C_1|, |C_2|)$.

February 22

With JDU away for the day we talked about HiQ with Thane Plambeck (hereafter TP). AT first gave a sequence of theorems about the 1-D game:

1. Let A be a peg, then A can not go to the left without using a peg already to its left.
2. Let A, B be pegs where B is to the right of A, then B can not go more than 1 position to the left of A without using a peg already to the left of A. Proof by induction on the number of pegs between A and B.
3. A configuration with a hole of length 3 is unsolvable.
4. A configuration with 2 or more holes of length 2 is not solvable. Proof by induction on the length of the gap between the holes.
5. In a configuration with an extreme odd cluster, both possible moves involving only pegs in the cluster lead to unsolvable configurations. Proof by induction on the size of the cluster.
6. A configuration with an extreme odd cluster and a hole of length 2 is unsolvable.
7. A configuration with 2 odd extreme clusters is unsolvable. (The first move leaves one of the previous cases.)
8. In a solvable configuration with an even extreme cluster, the move suggested by AT's algorithm leads to a solvable configuration, since this move must be made in any solution.
9. AT's algorithm works. 7 and 8 give partial correctness, and the number of pegs is decreased at each stage, giving total correctness.
10. The length of any solvable configuration is 1 or even.

AT then gave a long case analysis to find a regular expression representing solvable configurations. TP described a neat way of finding the regular expression: think of playing backwards. Starting with $11(01)^*00(10)^*11$ (which is obviously solvable) a backwards play involves moving the 00. Moving the 00 to the left (which corresponds to doing a leftward jump backwards) gives us $0100 \rightarrow 0011$, so we get the expression $11(01)^*0011(10)^*11$. Repeating these L-moves gives us $11(01)^*00(11)^*(10)^*11$. Doing an R-move gives $01001 \rightarrow 01110$, so the

above expression becomes **ll(Ol)*llOl(ll)*(lO)*ll,** and the 00 has been eaten. Reflection of the above expressions gives more terms to the RE, and running the 00 all the way to the left or right gives some special cases.

AT thinks it is NP-compete to decide, given a 2-D board position, to decide if there is a play ending with k or less pegs. He has shown that using pegs only to the left of a vertical line, one can move at least 2 to the right of the line. Is there a limit? TP answered in the affirmative. The book *Winning ways for your mathematical plays, volume* 2 by Berlekamp and Conway describes just this problem under the name "sending a scout". They give minimum numbers of pegs needed (behind the line) to send a scout out a distance **1** to 4, and prove that a scout can't be sent out further than 4 pegs from the line.

The book *Ins and outs of peg solitaire* by John Beasley states that it is an unsolved problem whether there is a polynomial time test for solvability in 2 dimensions. TP has tried to prove AT's version of the problem (is there a reduction to $\leq k$ pegs?) NP-complete by reduction from Hamiltonian circuit, but could not construct nodes. The idea is to embed the graph in the grid with pegs, so that there is a single tracer peg that jumps around the graph visiting all the nodes. Most of the weight of the graph must be in the nodes for the reduction to work, and one must be careful to avoid generating new tracers. When the tracer reaches a node it must be able to make a choice of which edge to take out. Furthermore the tracer needs to be able to enter a node through any of the edges incident to it. It is tricky to construct a node with these properties. TP managed to prove NP-completeness if the lattice can have "concrete pegs" that can be jumped but are never moved or removed.

TP thinks it would be neat to prove the problem NP-complete, and thinks one could publish such a result. It would also be neat to show that the language of tilable configurations is regular if the board is of fixed width.

February 27

First a couple of corrections from last time: The author of "Ins and outs of Peg Solitaire" is John Beasley, and a scout cannot be sent out further than 4 pegs, as we show below.

SR gave his latest star map results. Performing the band transform as described in the last few lectures, they get 1 or 2 band transform matches in about 1.5 seconds with 100 stars using 10 bands. The program considers 700 different angles (corresponding to 700 different band transforms). If the number of bands is kept constant then their algorithm runs in $0(n \log n)$ time. They haven't implemented the matching of individual stars inside a region to get an exact position for the mystery view, and are working only with jitterless orthographic projections.

DC has been working on a program to generate an algorithm for reduction of peg positions. The idea is to find a canonical system so that no backtracking is needed. He described confluence systems. Say we have rewrite rules 011 $\rightarrow$ 100 and 110 $\rightarrow$ 001, we consider combinations of these: 0110 $\rightarrow$ 1000 and **0110** $\rightarrow$ 0001. To eliminate backtracking we add the rules 0001 $\rightarrow$ **1000** and vice versa. Unfortunately the system of rewrite rules now accepts too many peg configurations. He has been working on a two dimensional version of a confluence system.

TP gave the nifty proof that a scout cannot be sent out more than 4 pegs above a line. The idea is similar to what we saw in De Bruijn. Let $\sigma = \frac{-1+\sqrt{5}}{2} \approx 0.61 < 1$. Then $\sigma^2 + \sigma = 1$. Now give each board position a number, as shown here.

$$
\begin{array}{ccccccc}
 & & & 1 & & & \\
 & & & \sigma & & & \\
 & & & \sigma^2 & & & \\
 & & \sigma^4 & \sigma^3 & \sigma^4 & & \\
 & \sigma^6 & \sigma^5 & \sigma^4 & \sigma^5 & \sigma^6 & \\
\hline
\sigma^8 & \sigma^7 & \sigma^6 & \sigma^5 & \sigma^6 & \sigma^7 & \sigma^8 \\
\sigma^9 & \sigma^8 & \sigma^7 & \sigma^6 & \sigma^7 & \sigma^8 & \sigma^9
\end{array}
$$

The numbering below the line continues infinitely in all directions. The value of σ was chosen so that each jump reduces or preserves the sum of the numbers covered by pegs.

The sums of the first few rows below the line are

$$\frac{\sigma^5}{1 - a} + \frac{\sigma^6}{1 - u} = \sigma^3 + \sigma^4 = \sigma^2, \text{ then}$$

$$\frac{\sigma^6}{1 - 0} + \frac{\sigma^7}{1 - \sigma} = \sigma^4 + \sigma^5 = \sigma^3,$$

then σ^4 and so on. Hence the total sum below the line is $\frac{\sigma^2}{1-\sigma} = 1$, so no matter how many pegs we have below the line, we can't get a peg into the hole labelled by 1, so the 5'th row can't be reached.

The book "Winning Ways" that we mentioned in the last class contains other examples of such "pagoda functions" (so called because of the shape of the above picture). For instance, if we are given a board with certain holes empty, and wish to jump pegs till only those holes are full (this is called a reversal problem), then we need to label the board with numbers so that the sum on the inside (the original empty holes) is greater than the sum on the outside. The numbering must ensure that a peg jump can't increase the sum. Here is an example:

		-1	0	-1		
		1	1	1		
-1	1	0	1	0	1	-1
0	**1**	**1**	**2**	**1**	**1**	**0**
-1	1	0	1	0	1	-1
		1	1	1		
		-1	0	-1		

Initially the middle 4 squares are empty. The sum of their numbers is 6, while the outside sum is 4, so the reversal problem is impossible.

Note that the fastest a pagoda function can increase is like the Fibonacci sequence, since the sum of **2** adjacent values is no less than the next value.

A question was raised: are there arbitrarily large dense solvable 2-D boards? TP: yes. Arrange the pattern $10(11)^*0011$ in rows, with three empty rows, so that when each row is jumped to just one peg, these pegs line up in a column with the same pattern.

AT has done some more work on the 1 dimensional case. He gave a polynomial time algorithm for determining if a peg configuration can be reduced to *k* pegs. Say we are left with 1000001000001000001, then the sets of pegs used to get each final peg are disjoint and non-interleaved. Now consider a contiguous section $[i, j]$ of the board (i.e $j - i + 1$ adjacent positions, with some pegs as given at the start of the game). This section satisfies one of four conditions:

1. The section is not solvable (reducible to **1** peg).

2. **It** is solvable using only holes in the range $[i, j]$.

3. It is solvable using **1** extra hole to the right.

4. It is solvable using **1** extra hole to the left.

Construct a linear graph, with 2 nodes for each position on the board. The graph loos like $\mathbf{0} \rightarrow 0' \rightarrow \mathbf{1} \rightarrow 1' \rightarrow \mathbf{2} \rightarrow 2' \rightarrow \mathbf{3} \rightarrow \ldots$ Add an edge $i \rightarrow$ j if case 2 above holds. Similarly for edges $i' \rightarrow$ j', i $\rightarrow j'$ and $i' \rightarrow$ j. Now the length of the shortest path from the first node to the last node in this graph is the least number of pegs that the configuration can be reduced to.

The weighted version of the problem is NP-complete, by reduction from this version of the knapsack problem: given weights n_i, is there a set A such that $\sum_{i \in A} n_i = k$?

The reduction is as follows: for each weight n_i use two pegs, weighted 0 and n_i, next to each other and with 3 spaces on either side. Then choosing which way to jump in each pair amounts to choosing a set A.

FIRE DRILL.

March 1

JDU described a method for pruning the large search trees that a HiQ program would have to search. There may be lots of possible moves at each configuration. Give the possible moves a lexicographic ordering, say giving higher priority to moves that start higher, or are at the same height and are further to the left. Do only the most preferable move at any stage, as well as any move that the preferred move interferes with or renders impossible.

DK pointed out that the preferred move can eventually interfere with just about any position on the board, by a sequence of peg jumps. In simpler terms, there is no commutativity of moves in this game. SQ noted that the program could choose to do the preferred move first, or *never*. In other words, if the preferred move is ever done, it must be done first. You can also hash the configurations of the board that arise in the search tree, to eliminate multiple paths to the same configuration.

JDU suggested keeping track of infeasible configurations, using de Bruijn coloring, or SR's 4-coloring. We could also keep track of the possible positions for the final peg at the end of the game, using a pagoda function: color a final position 1, and color points at L_1 distance k from it with σ^k. The weight of a diamond around the final position decreases exponentially, so the pegs must be fairly dense around the position for it remain feasible. SR noted that there will generally be lots of feasible final positions given by this coloring, since the position of each peg is feasible. JDU suggested finding all solvable configurations of say 3 pegs, and constructing pagoda functions weighted high on the three pegs. DK thinks this won't eliminate many configurations that l-peg weighting doesn't eliminate. MG noted that we don't actually reduce the size of the search space much by these techniques, since we have to keep all configurations for which at least one final position is possible. DK would like to restrict the problem to having a pre-determined finish position, to allow better heuristics and search tree pruning.

SR doesn't like the idea of brute force search without good heuristics. He thinks coloring each peg one of four colors, then eliminating all but one color, and ending the game by

hopping around the lattice of the pegs of that color, should work. He also suggested using clustering (reducing clusters of pegs to one peg then eliminating the remains).

AH asked how hard the problem is in general. How far ahead does one need to think when playing the game? Most thought the problem is hard, with early decisions being important.

SJP showed that the problem "Is there a sequence of jumps leaving less than *k* pegs" is NP-complete, by reduction from Planar-3SAT (ref: David Lichtenstein *Planar formulae and* their *uses*, SIAM Journal of Computing Vol 11, no. 2, 1982).

SQ asked what hard configurations are known. (Just the few given with the HiQ game, though Beasley and Conway have more configurations.) DC suggested running the game backwards and forwards at the same time.

Said JDU, "We've had too much theory on this problem, and not enough practice."

Chapter 6

Playing God

March 1 continued

We moved on to discussing the god-game. There was some confusion on the current rules of the game, so here is a summary:

Each team will write a god program and player program, in a language of their choice. On the day of the competition, each team's god will play against all the *other* teams' players. Thus your player will have to play against the gods of all the other teams.

Each game is between one player and one god. The game starts with the ace of spades on the table. The player has the other **51** cards, and repeatedly plays a card, which is either accepted or rejected by the god. The god's rule for which cards to accept must depend only on the stack of accepted cards on the table. At early stages of the game there must be a reasonable number of acceptable cards, so that the play is not too sluggish (the **10** card rule). The play ends when the player has had all his cards accepted, or when he has had all his remaining cards rejected since the last accepted card. The score of the player is the number of plays he takes to get to either of these end states.

The player with the lowest score and the god both receive a number of points equal to the difference between the highest and lowest scores, amongst all players playing against the god (i.e. amongst the players belonging to each team except the team who wrote the god).

A good god program (one that will score highly) will use an acceptance rule that can be understood quickly be a good player program, but is not so easy that all players will understand it. A good player program will recognise lots of rules that can be fairly quickly recognised. Remember that all your player needs, is to be able to recognise all the rules that all the other players recognise, and then a few more.

March 6

MG thinks there is too wide a choice of rules available in the god game. He asked if the competition could be divided into two rounds. In the first round the programs would watch the play of other programs, so that the gods would know the level of skill of the players, and use appropriate rules. SR thought the scoring technique already provides the checks and balances to make the god rules reasonable. There was some discussion on MG's thoughts. The resolution is that if someone wants such an eavesdropping feature it could be added to the controller.

JDU described the prisoners dilemma. **2** players simultaneously choose a 0 or **1.** If they both choose **1** they each get a small minus score. and if both choose 0 they get a small plus.

However if they choose differently, the one who chose a 1 gets a large plus, and the other gets a large minus. In a contest in which programs played prisoners dilemma against each other, it turned out to be beneficial to arrange beforehand to collaborate with another program, which you would recognize by a prearranged sequence of first moves. Could we do the same in the god game? A problem is that god must use a rule that depends only on the contents of the stack of cards on the table. However, god could still determine his opponent, and hence his rule, by the first card(s) played.

JDU described a glitch in the scoring: a rule that forces longer plays, by not having many cards acceptable, would have a bigger difference between best and worst plays. Thus the scoring is biased towards hard gods, that force long games. It might be better for the points allocated to be the ratio between the excess plays of the best and worst players, where the number of excess plays is the total number of plays less 51.

AT asked if the rule a god uses can change from game to game — it could be easy for one player, and hard for the rest. JDU said try walking out of the competition room after using a rule like that.

AH described his technique for solving the easy version of problem 3 (he had missed class after thinking of this technique). Given photos of a cluster taken from Earth and Trantor, with no jitter: each star defines a plane through Earth, Trantor and the star, and the angle of elevation of this plane can be easily calculated from either view. Sorting the stars in both views by this angle and matching 1 to 1 gives the correct match. (Actually if some stars share the same plane they can be matched arbitrarily — they can't be resolved using the photos.) If there is jitter, AH hypothesised that the same technique (just sorting and matching 1 to 1) gives the best possible match by a variety of norms (L_1, L_2, L_∞) photographs. His program found matches that were by all norms better than the "correct" match.

Returning to the problem at hand, SR thinks that the essence of the problem is coming up with the correct representation for god rules. He envisions having a vast library of rules that have been used before, on which the player has been trained, and basing plays on these rules. After some discussion it was decided that the best way to use such a library would be to play the card that the most rules would accept. If the card is accepted we're happy. If the card is rejected we are still happy, since we have at least halved the number of possible rules.

SR suggested using DFA's to represent rules. If the rule is l-memory (depending only on the top card on the table) then the DFA will have at most 52 states. If it only uses the suit of cards played it will have 4 states. However the number of attributes that a god could use is large (eg. suit, rank, color, one-eyed jacks, suicide kings, . . .) and god can use an arbitrary amount of memory, so the number of potential DFA's is huge. Hence DFA's might not be the right representation. JDU noted that DFA's are more suitable for representing regular languages than regular expressions are, if there is memory.

There were various suggestions for reducing the space of rules, such as determining first which attributes god is using. SR will ask John Woodfill (a grad student who has worked on recognising regular expressions) for ideas. VG noted that there are algorithms that run in polynomial time for learning regular expressions. Unfortunately polynomial time is not good enough for us — we need to learn a rule in less than 51 plays.

AH suggested using a collection of routines, each looking for different patterns, and

suggesting plays.

JDU thinks we'll have to develop an "expert system", or more specifically a driver or controller into which one could plug concepts (such as suit and rank) that can be added as they become necessary.

MG asked if all cards must be playable in some game, for a fixed rule (yes). Thus the rule "only reds are playable" is not acceptable, though "accept only red cards after the first red card has been played" is.

DC suggested building up simple concepts using boolean connectives. SR described the version space algorithm (ref Tom Mitchell, "Generalisation as Search", Artificial Intelligence 1982). Given a space of rules that are partially ordered by specificity, we keep track of the set of rules consistent with a sequence of (positive and negative) examples of the concept. We keep only the most general concepts MG and the least general concepts LG in this set. When we see a positive example, the set MG moves down, and when we see a negative example, the set LG moves up. A problem with this technique is that MG and LG might be exponential size. Valiant proved that under certain simple conditions MG or LG will be just one rule (singleton sets).

SQ gave a progress report on HiQ. His program solves the 33-peg HiQ board in 1 second, and the 45-peg board in 6 hours. He is using no heuristics, and thinks he could easily get the run time down to 1 hour. He gets down to 2 pegs in about 15 minutes. VG said that by hand he also jumped down to 2 pegs in about 15 minutes. AH: "Congratulations, Sean! You've duplicated Vineet!"

SQ's program searches about 10^6 nodes at the 10-peg level on the 45 peg board before finding a solution with 2 pegs remaining. He caches 400 000 positions. Doubling the cache improved the run time by about 30%. JDU thinks a 32-bit address space could encode all the positions found on the way to a l-peg solution to the 45 peg board, thus eliminating repeated searches.

Dividing into 5 teams of 2 people (who haven't worked together yet) proved tricky. A greedy match failed when the last two unmatched people had worked together. Eventually the following teams were devised:

(1) DK, DC. (2) AH, SR. (3) AT, MG. (4) ET, SQ. (5) VG, AS.

March 8

JDU has written 6 gods, which can be found in /mnt/ullman/cs304 on nimbin, and phillips/CS304/public/godgame on neon. The file README describes the rules used by each. One of the rules allows suits played round robin, but if the size of the pile is divisible by 5, then any card is playable. The idea of this is so that some players would throw out the only rule that is relevant (a suit progression rule) because it has exceptions.

JDU is wary of snake oil salesman in computer science — people who think programming is easy, and hard problems can be solved by, say, neural nets: if the program doesn't work, kick it and it'll learn its errors. He thinks an effective approach is not to use fixed rules, but to determine affinities of cards with previous plays (where affinity must be a programmable and tunable notion), so the player wouldn't be thrown off by exceptions.

SQ asked if players play the same god more than once during the competition (no, SO no learning can be done during the competition).

There was some discussion on the reproducibility of god's behaviour, and it was decided that a god must always behave the same on the same set of card plays. This **is a consequence** of god's behaviour only depending on the stack of accepted cards on the table.

A note of etiquette: it is illegal for the player to check his plays before playing them, by executing his own copy of the god program, and testing the plays to see if his copy of the god will accept them.

MG has worked on constructing rules from simple attributes, using binary connectives. He hasn't written code, but would generate all rules up to a fixed complexity (about 1600 rules), and keep track of those that have made no errors on the cards so far. **He** thinks the results are promising in a limited way — he thinks a program will quickly recognise any rule that can be represented by the attributes and connectives, but will die on any rule that is not covered, such as JDU's mod-5 rule. Keeping track of rules that are correct only most of the time might fix this fragility. Even with this modification, this kind of player would break if the god switched to a different rule in the second half of the game. SQ noted that rules with lots of connectives will not be good, as no person would construct a rule like that. However, a rule that seems simple to a person could become complicated when written in terms of a small set of attributes.

A few attributes that a god might use are: color, suit, rank, picture, major, minor, high/low, letter/number, one-eyed jacks, suicide kings, rank range, n'th from last, *n*'th from start, stack length, sum of ranks, ... Note that the first two can be used to construct all other attributes.

A god could use arbitrarily obscure mathematical relations, but as MG observed, the whole point of this problem is psychology, not math. If it were math the problem would be intractable. A player can't recognise all possible attributes and rules, so one must aim to code all the likely ones. JDU thinks the teams will be able to write programs that would beat, say, a typical freshman. AH: "Hey, some of my friends are freshmen!"

SR thinks that the advantage of using neural net ideas is that a neural net is robust — each time the god disagrees with a rule that is given by a net, the net moves a little in the rule space. This makes it resistant to rules like JDU's mod-5

JDU is interested to see if the best player needs to understand the rule well, or whether one can play well without explicitly representing the rule.

SR talked about learning l-memory rules, i.e. relations between cards. He thinks a collection of feature demons will be useful (using the terminology of Rumelhart's work on recognition of blurred words: he had low level demons looking for vertical or horizontal lines, shouting their finds to the next higher level, which would be looking for letters, which would signal the demons looking for words.) JDU: "This reminds me of the joke in which a search plane flies over a desert island, and the pilot says 'Nope. On second thoughts it says HELF' "

AS asked if the kind of generalisation that neural nets do, matches the generalisation that we need in the god-game. As an example, a net that chooses cards according to the total rank mod 17 is likely to fail badly if god is choosing cards according to the total rank mod 18.

JDU: construct a string of O's and l's for a rule, with a 1 when the rule matches the god's play. We want a union of things that match the play so far, which is a kind of exact match problem. Exact match is NP-complete, but approximations would work fine for us. If

we are constructing rules with intersections and unions, then covering is the essence of the problem. SQ noted that this isn't true when the rule changes halfway through the game.

Some quotes for today:

> Just think of this as a social game played by teenage nerds. *JDU*
>
> If you look **at this not** rigidly like in neural nets, instead maybe use fuzzy logic. . . AH
>
> Oh, don't start me on fuzzy logic! *JDU*

March 13

SQ asked if there are any good heuristics for the covering problem we described in the last class. JDU suggested using a greedy approach. If the god rule just depends on the suit of the top card on the table, and the suit of the next card to be played, then we wish to cover the acceptance string (the string of 1's and O's with a 1 when the card played was accepted) by a disjoint sum of traces, for the 16 possible rules of the form: if the top card has suit1 then accept only suit2. A trace is again a string of O's and 1's, with a 1 when the rule is applicable and consistent with the play. We throw out rules that are inconsistent, then repeatedly add to the cover a remaining rule that covers the largest number of remaining 1's in the acceptance string.

This technique may be adaptable to slightly more complex rules, but might not be of any help for complex rules. If the rule depends on more attributes, we have a harder covering problem. MG is doing something similar to covering, finding the simplest rule that is correct at least 80% of the time, as well as trying to find an "exact cover" that is correct all the time. He has added a type system to his method, so that instead of having 1600 rules he now has only about 270, giving him room to add new attributes.

AH and SR found an AI article on Eleusis, a game similar to the god game, that is claimed to have been invented in the 1970's (though the god game existed earlier). The paper does some ad hoc things, and talks about similar techniques and approaches to the problem as we have in the last few classes, but doesn't seem to achieve any results.

It was decided that the competition should be on Thursday during finals week. This will give the teams time to write powerful players. Although AS will work on developing a player, he won't make the competition — "I don't have to be there for the final victory."

SQ brought up the question of how to score inside a player, in other words how to correlate the recommendations of various rules. The probability of a rule being correct when it says a card will be accepted is $\frac{tp}{tp+fp}$, where tp is the number of true positives so far, and fp is the number of false positives. We want to scale this so that we won't favor rules that hardly ever give a positive: $\frac{tp}{tp+fp} \times \frac{tp}{tp+fn}$. The reason SQ doesn't use the obvious expression $\frac{tp+tn}{tp+fp+tn+fn}$ is that the player who always answers no will be correct about 80% of the time.

JDU gave the analogy of the stock market: there are lots of rules that predict the behaviour of the stock market till yesterday, but have no correlation with its behaviour tomorrow, as many people often discover. That said, SQ noted that even with a dictionary of about 100 rules, and using his scoring technique, his player performs better than the trivial players. He keeps all the rules, and scores each of them at each play, to get the traces. He also does some evaluation to determine the next play: For each remaining card, he tries out each rule on it, scoring a yes or a no. His scoring technique, described above, estimates the

probability that a rule is correct given that it says yes, and similarly for no. He looks at $\Pi(1 - \Pr[\text{correct} \mid \text{said yes}])$ and $\Pi(1 - \Pr[\text{correct} \mid \text{said no}])$, where the products are taken over all rules' responses on a fixed card. The final decision is yes or no depending on which product is smaller. The reason for the above form of the product is that if a rule has been totally correct so far, and predicts that the card will be accepted, then the first product will be zero, so the correct rule's voice won't be drowned out.

AH and SR have also been looking at the problem of correlating the choices of a number of rules. The put a prior distribution on the rules, and each play tweaks the weight of all the rules, by Bayesian analysis. SR described the form of their rules: $S_i = V_j \rightarrow$ accept class C_k, where S_i is a state variable, such as the suit of the last card, the sum of the ranks mod 5,. . . , V_j is a value, and C_k is a card class, like hearts or picture cards.

JDU expressed surprise that some people want to use rules that have been wrong on some plays. AH explained that if the rule uses concepts that are not in the program, then the player has no choice but to use rules that err. The hope is that some rules that the player has available will approximate the correct rule.

JDU thinks that the programs we see at the competition will be able to beat people. AS noted that they will be fragile, and described how a program measuring word length, number of sentences in a paragraph, and number of semicolons, can grade school essays in close correlation with teachers' grades. However, if a student writes gibberish with long words and lots of semicolons, the teachers will pick it up while the program won't.

AT has thought up a god program that doesn't rely on its rule being recognised by only some of the players. It accepts everything until a magic card is played, then makes the player play cards only in a strange sequence. To satisfy the lo-card rule he allows other plays if the magic card is played before **10** cards have been accepted. The variation in the number of plays of a player playing this god is large, essentially just the variation of a single random variable. A similar rule is that if the first card played is red, the rule is easy, otherwise it is hard.

MG noted that this problem is an artifact of the solitaire version of the god game — such a god rule wouldn't work well in the original version, where all the players are playing round robin with the same pack of cards.

Although AT's rule throws a spanner in the works, JDU will have some proper gods on hand on the competition day, so it will still be worthwhile to write some good player programs.

March 15

JDU introduced a taxonomy of techniques for writing a good player. Firstly there is the statistical method, in which statistical analysis is used to approximate the rule without understanding it fully. The neural net and feature extraction methods fall under this heading. An advantage of these techniques is that they are malleable — they are likely to fare well even if the rule changes during a play, or if the rule uses card features that were not considered during the design of the player.

The second approach could be called generate-and-test. The player generates a large set of rules, and tests each one for consistency. This paradigm can be further divided: the player can keep track of the consistency of component subrules, which are combined to form the guess at the god's rule, or it can consider all combinations of the subrules at each step. SQ

thinks about 100,000 to 1,000,000 rules can be considered at each step, using **10** seconds on a 10 mip machine, since the rule analysis involves fast operations on short bit-strings. Thus a player considering exact rules can probably search faster than a statistical player. The price such a player pays is in brittleness — if god's rule is outside the space considered by the player, he might fail badly.

AH suggested a method for breaking a brittle program: use a rule *P* most of the time, but at random points use a different rule Q. However, the generate-and-test proponents think that their players could easily handle this.

MG thinks that the players will do better than was previously thought. He described how his player knew nothing of bridge wrap-around ordering of suits, but used a rule involving the fourth card down in the stack to play well against the god that accepted cards only in the bridge order. His program uses a generate-and-test technique, and he says it has some pretty weird rules still under consideration, at the end of a game.

There was more discussion about AT's rule. JDU suggested that the fraction of cards playable shouldn't have wild jumps during a game. The idea of a god rule is to test the players. We are really conducting an experiment, to see how well a program can learn a rule from a large ill-defined space of rules, and what type of player will be best suited to the task.

MG described how a random player, playing against any legitimate god, will average about 400 plays. The least possible number of plays is 52, so there isn't a very large range of possible scores. He is worried that the scores will be affected more by random card choices at the start of a game, than by the quality of the players. Other members of the class thought this was not likely to be a problem, and anyway we will be playing the players against lots of gods. It should be clear by the end of the competition, which are the best players.

SR introduced some new ways of looking at a player. Given a set of state variables S_i, we obtain a trace for each, with a 1 at each position that state variable is true. We wish to predict the next bit in each trace. We have a collection of models for the sequences, which predict the next bit. As before, we can keep track of the consistency of the predictions of these models.

Alternatively we can use a Fourier transform to determine periodic behaviour in the traces. We could also model a trace as a 2 state Markov process: the states are 0 and 1, and we determine the probability of the next bit being 1 if the last was 0, and vice versa. This will recognise density and persistence in the traces of the state variables. A more powerful technique is to determine for each state (0 and 1) the probability distribution on the number of steps taken before a transition is taken to the other state. If the rule is random, then the distribution will be exponentially decreasing. If the rule is "alternate colors" then the state variable "color is red" will have a peak at 1 step. We could start with the exponentially decreasing distribution, and use Bayesian inference to change the distribution as more cards are played.

MG noted that this technique will fail to recognise a pattern in a trace like the following: 01101110110111011011101101110 — a more powerful model would be needed for this.

SR described his technique as a decomposition method: he doesn't think that the table of traces needs to contain combinations of state variables. The table will have to have a huge number of state variables, though.

We can incorporate the information given by failures into this model as follows: if there

are lots of rejections before an acceptance, and S_i is true for the accepted card and false for all the rejected cards, then its trace would contain a large value (instead of 1) for the accepted card.

SR also described what he thinks is a more brittle approach: we keep a time-varying sequence of tables of states S_i against classes C_j, with an entry (i, j) in table k being Yes if the k'th play was a card in class C_j, the current state satisfied S_i, and the play was accepted; if the play was rejected the entry would be No. A pattern has the form $S_i \Rightarrow Yes(C_j)$ or $S_i \Rightarrow \neg\, Yes(C_j)$, and similarly for No.

A problem with this technique is that the number of state variables and classes will be huge, so they couldn't all be considered at the same time.

The time of the competition was set for 11am. on Thursday.

March 20

Results of the Competition

Below are brief descriptions of the god rules used to rate the players. The first few were given in advance, so were not used in the competition.

- godl: sum of the suits played, mod 4, equals rank of next card, mod 4. Suits are valued C=O; D=l; H=2; S=3.
- god2: major suit must follow minor, and vice versa.
- god3: difference in ranks no greater than 3.
- god4: letter follows red; number follows black.
- god5: suits go in round robin fashion by position mod 4, but every fifth position is "wild" (any card may be played).
- god6: card must match one of the previous 3 cards in either suit or rank
- god7: even positions: suits in bridge order; odd positions: ranks alternate >= 7, <= 7.
- god8: (KAR) card **must** have same rank or suit as the card four plays ago. First 4 cards can be anything.
- god9: (KAR) Picture cards are wild; otherwise must be in descending rank sequence mod 10, eg: AS, TD, 9H, JH, 8C, etc.
- god10: (KAR) computes rank + suit, mod 13. Current card must have rank+suit that is greater than this value, but not as much as 5 greater.
- godll: sum of last five ranks equals current rank, mod 3.
- god12: let p = square of previous rank, mod 13. Then p = 0, 1, 3, 4, 9, 10, or 12. Square of current rank, mod 13, must be equal to or successor (end-around) of p in this sequence.

- god13: rank of top, plus suit of second, plus square of rank of third plus suit of next equals 0 mod 4.
- god14: This is an illegal rule. Every other card is playable, no matter what it is.
- god15: (PMU) Add the suit to the rank of the card. The number of one's in the binary representation of the sum must alternate even/odd. (eg. AS=4=odd)
- god16: (AH) Suits in bridge order except after 3 or J in which case reverse bridge order.
- god17: (AT) 8 9 10 wild. Every 3 (neglecting 8, 9, 10) must sum to 15 with J=S Q=9 K=10.
- god18: (SR) suits in order SHDCCDHS...
- god19: play a higher red card or a lower black card.
- god20: (DK) odd positions: suit order; even positions: sum of top two card ranks must be even.
- god21: (DK) play different suit from middle card of pile. Also, card played must be opposite high/low as second card from top of pile.
- god22: color pattern is (BRRB)*.

The following table rates the various players against many of the above gods. Each entry is the number of cards played till all cards are down, or every playable card has been played. The last two columns are given for comparison: the random player chooses for his next play one of his remaining cards at random, while the circular player tries his remaining cards one by one in the natural order till one is accepted.

God	Team 1	Team 2	Team 3	Team 4	Team 5	Random	Circular
7	107	S4	104	88	161	478	126
8	78	60	78	57	69	112	162
9	97	60	245	145	54	207	212
10	127	144	168	166	163	176	153
11	115	117	114	98	127	113	118
12	69	99	S6	129	148	108	154
13	175	120	224	182	162	139	173
15	92	91	SO	111	88	70	94
16	S4	67	S5	103	98	882	149
17	56	S6	76	97	105	70	101
18	104	68	95	S7	581	519	152
19	69	68	151	104	197	103	120
20	131	S6	96	69	0	507	140
21	168	176	156	125	0	217	152

Chapter 7

CS204 Problem Statements

CS204 PROBLEM #1: Breaking the Code

Due Thursday, April 19

Let's see what we can do with simple substitution ciphers, in which a permutation π of the letters is selected, and a message is encrypted by replacing each letter x by $\pi(x)$. For example, if part of our permutation is

Letter	H	E	L	0	W	R	D
π(Letter)	A	B	C	D	E	F	G

then HELLO WORLD would be encrypted as ABCCD EDFCG.

Since there are 26!, or about 10^{25}, possible permutations, trying all does not seem like the thing to do. We can attack the problem by assuming that all the words in the original message (called *plaintext*) are found in /usr/dict/words. Thus, there may be a hundred possible words of which ABCCD is an encryption (called *ciphertest),* and perhaps a thousand of which EDFCG is an encryption (the double C in ABCCD makes it much harder to match). However, there are not 100,000 pairs of words that match ABCCD EDFCG, because the repeated C's and D's are quite restrictive. Further, we get some restriction from the fact that all the letters in the ciphertext represent distinct letters of the plaintext. With luck or good planning, we can select words of the ciphertext in a good order, and keep the number of possibilities within reason. There are some other strategies that the instructor will discuss in class.

A Simple Variation

If that seems too easy, we can observe that /usr/dict/words does not include words with suffixes and prefixes, like "nonreentering." Assume that the plaintext is chosen from words in /usr/dict/words, possibly modified by common prefixes and suffixes, such as re-, non-, -s, -ing, -er, or -ed. Be careful to use the proper formation rules, like "hate" → "hating," or "panic" → "panicked."

A Less Simple Variation

Additionally, suppose that a small fraction of the words in the plaintext may be words

that do not appear in /usr/dict/words, even after modification as above, such as "ullman" or "sparcstation."

The Enigma Variation

During World War II, Alan Turing worked, successfully, on a British project to decipher the German code, which used the *Enigma machine* for encryption and decryption. The British had captured a copy of the Enigma. machine, and the Germans knew they had, but believed the workings so complex that the British could not decode messages in a timely way, even seeing the mechanics of the machine.

The Enigma had four wheels, each of which had the effect of permuting the 26 positions that might represent letters. The wheels would be set in a special position each day; the position for the day known only to the operators of the machine. Suppose the operator types A. Depending on the position of the first, wheel, A might be in position 10. The first wheel would translate position 10 into some other position, say 22. Position 22 on the first wheel might be next to position 17 on the second wheel, and perhaps the second wheel translates position 17 to position 3. The process continues through all 4 wheels, and the final position of the last wheel is translated to a letter. Then, to make matters worse, after this letter is translated, the wheels move relative to one another, something like the wheels of an odometer, but with a more complex interrelationship. Thus, another A would not be translated to the same letter.

Suppose we are given the wiring and gearing diagram for an Enigma machine. That is, we know what the permutation of positions is for each wheel, and we know, as a function of the current positions of the wheels, what the next position will be. However, we do not know the initial positions of the wheels? and we do not know the plaintext. How can we decrypt ciphertext under these assumptions?

CS204 PROBLEM #2: Superabundant Numbers

Due Thursday, May 3

The abundance of an integer n is the sum of the divisors of n (including n itself), divided by n. Integer n is k-abundant if its abundance is at least k.

For example, the sum of the divisors of 6 is 6 + 3 + 2 + 1 = 12, and 12/6 = 2, so 6 is 2-abundant. As another example, the sum of the divisors of 120 is

$$120 + 60 + 40 + 30 + 24 + 20 + 15 + 12 + 10 + 8 + 6 + 5 + 4 + 3 + 2 + 1 = 360$$

so 120 is 3-abundant. It happens that 6 is the smallest 2-abundant number and 120 is the smallest 3-abundant number. They happen to be exactly 2- and 3-abundant, respectively, but it is generally possible that the smallest, k-abundant, number has abundance greater than k.

Your goal is to write a program that finds the α_k, the smallest k-abundant number for $k = 1, 2, \ldots$. How high can you go?

Some Suggestions

When the CS304 students worked on this problem, the concept of a Phillips number

(named for Steve Phillips, the TA) was useful. A Phillips number is a number that is more abundant than any smaller number. Clearly, α_k is a Phillips number, but there are many Phillips numbers that are not α_k for any k.

Solving this problem requires some (not very deep) mathematics, involving prime numbers. As a warmup, try answering the following questions:

1. What is the abundance of 2^i?
2. What is the abundance of $2^i 3^j$?
3. What is the abundance of $2^{i_1} 3^{i_2} 5^{i_3} \cdots p_j^{i_j} \cdots$, where p_j is the jth prime?

When you start writing code, you may wish to use the source of primes in

```
/usr/games/primes
```

This program is a generator of the primes, in order.

CS204 PROBLEM #3: Playing "God"

Due Tuesday, May 22

In the days before Dungeons and *Dragons*, one of the things nerdy teenagers did was play a card game called "God." On each round, one player was selected to be the god. The god made up a rule whereby cards could be played on a pile, which was constructed in a line, so all played cards were visible. An example of a (too simple) rule is "only a red card can be played on top of a black card, and vice versa." To begin play, all cards but one were dealt to the players, and the last was turned up to start the pile.

In turn, players offered cards for the top of the pile. If the play meets the god's rule, then the card is allowed to stay on the pile; otherwise it is withdrawn to the player's hand. Play ends when one player gets rid of his last card. That player, and the god, each score an amount equal to the sum of the cards remaining in the other players' hands.

Selection of God Rules

The scoring system suggests that the god should pick a rule that is nontrivial, but deducible with some effort. If a rule is too easy, then everyone will catch on quickly and get rid of their cards at approximately the same time. The score will tend to be low, as no one will be caught with many cards. If the rule is too hard, everyone will play as if at random, and the expected number of cards with which anyone is caught will be low as well. Ideally, from the point of view of the god, one player should figure out the rule immediately, and get rid of his cards quickly, while the other players are stumped and get rid of cards only by luck.

There are some constraints on legal rules that the god may use. First, while many good rules are "l-memory," in the sense that playability of a card depends only on the previous card played, it is permissible to use a rule in which the playability of a card depends on the entire pile. Example: "The sum of the cards on the stack, modulo 13, must be a prime." However, the following must be satisfied.

1. The rule depends only on the stack contents. No "I pass them crossed" or "Anything Sally plays is OK, but anything anybody else plays is wrong."

2. There must be at least 10 cards playable in any situation. This rule must be understood to apply on the assumption that an infinite supply of cards exists. Otherwise, with any rule and any stack of 51 cards there is only one card playable. An example of an illegal rule: "each card must be of one higher rank than the previous card on the stack, with Ace following King and Deuce following Ace." (Only 4 cards are playable in any situation.)

Even with rule (2), it is possible that the game will reach a situation where there are no legal moves for any player. Example: "A letter card must follow a number card, and vice versa."

Some Example Rules

Consider the following possibilities, for example.

1. If the rank of the card played is equal to or higher than the rank of the top card, then the two cards must be of the same color; otherwise, they must be of different colors.

2. The difference in the ranks of the cards must be no greater than 3, in the "end-around" sense (e.g., Deuce is distance 3 from Queen).

3. A letter card may not be played if either of the top two cards are letter cards.

4. Face cards (J, Q, I<) played must be of the same color as the top card. Other cards must be of a minor suit (C, D) if the top card is a major suit (H, S), and vice versa.

5. Cards must alternate odd/even ranks, but one-eyed jacks and the suicide king can be played any time.

The Problem

You should write a program to play "solitaire" God, interacting with a controller program that was written by Steve Phillips. Your program starts with 51 cards, and the Ace of Spades is assumed to be the initial card on the pile. A card is represented by two characters, the first being the rank (A, 2, 3, . . . ,9, T, J, Q, K), and the second the suit (C, D, H, S). For example, to play is the six of spades, you put 6S<newline> on the standard output. You will then receive on your standard input the character Y (yes, the god accepted your play) or N (no, the god rejects your play), followed by a <newline>.

The controller program is found in

```
portia:~ullman/controller
```

and `portia:~ullman/controller.c`.

Scoring

The controller counts the number of plays you make. It stops the game when either you run out of cards, or you have played all the cards in your hand and had each rejected. (The controller also makes a minimal attempt to detect looping situations where your program

repeats a failed attempt without first trying all your cards.) Your score will be the number of plays made minus 51. That adjustment helps avoid a situation where only god programs that reject most cards can produce large scores and therefore large variations.

Writing God Programs

There is a template god program in `portia:~ullman/god.c` that will enable you to implement god rules by following the simple comments found in that file. If you wish to write your own programs, they must expect from the controller cards in the form <rank><suit><newline> and must produce responses of the form <Y or N><newline>, on the standard input and output, respectively.

CS204 PROBLEM #4: Traffic Light Controller

Due Thursday, June 7

These days, it is quite common for traffic lights to have bumpers that sense cars approaching from any of the four directions. Also, it appears that each town has its own computer that reads bumps and follows some simple algorithm to decide whether to switch the light. A typical algorithm is to switch a light only when either

1. A bumper in the red direction is bumped but there is no bump in the green direction, or
2. There are bumps in both directions, and a time-out has occurred, e.g., the light has not switched for two minutes.

This algorithm is amazingly dumb, considering that one computer knows about bumps at all the lights in town. I think people can design much more effective algorithms.

The Map

ESC is writing a simulator for a simple road map. I suggest that trying your ideas on a 5 x 5 grid of two-way roads would be a reasonable test. There are thus 100 bumpers, four at each of the 25 intersections. Cars appear at the edges of the map at random, at a rate you select. It is possible to assume cars travel along the roads without turning or disappearing, at the speed limit set for that road, unless the car stops for a light. I am led to believe that turning, appearing, and disappearing cars will be options allowed by the simulator.

The Problem

You are to write a program that accepts a. data. structure representing all the bump/no-bump information at the intersections and modifies the structure to reflect any changes in the setting of lights. You can switch lights arbitrarily; there is no "yellow" period unless you program one in. Each call to your procedure represents one second of elapsed time. The physics of automobiles (e.g., they slow down; they don't stop on a dime) has been built into the ESC simulator. There is also, I am told, a rudimentary intelligence, in that if cars are still traveling through an intersection in one direction, and you turn the light

green in the other direction, the cars waiting in that direction will not move. Details of the structure will be provided in a "manual" ESC is preparing, but I want to get people thinking about algorithms now.

The question of what is a "good" algorithm is a bit subtle. I propose that minimizing the average time it takes a car to pass through the map is a good idea. However, we also want to avoid, say, stopping traffic in one direction and letting it flow unimpeded forever in the other, so maximum delay should also be low. Setting both directions green and hoping for the best will be regarded as an error, and may cause a crash, of both the cars and the simulator. Finally, we want to avoid failing to admit cars to the grid as they appear on the border, or "gridlock" conditions where queues at a light have grown so long that they block traffic at another intersect ion.

Chapter 8

CS204 Class Notes

CS204 — Who They Were

AW		Aaron Wallace
BJG		Bruce Goldman
BL		Ben Lai
CJ		Chris Jones
CW		Carl Witty
ER		Eric Rose
ESC		Edward Chang
ET		Eric Teller
HH		Hugh Holbrook
JD		James Drew
JDU	-	Jeff Ullman
KF	-	Kathleen Fischer
MK		Michael Killianey
MN		Mason Ng
MPF		Mike Frank
MT		Mark Torrance
PF		Perry Friedman
PKW		Peter Wagner
PS		Piyush Shah

Day 1 — April 3

Professor Ullman (JD U) welcomed everybody to Stanford's first Undergraduate Programming and Problem Solving seminar. A similar course for first year graduate students has been taught for the past two decades, with the goal of introducing them to research rnethods and to each other. Since (we expect) the course consists predominantly of seniors, JDU remarked that most of the people probably already knew each other. A quick survey, however, revealed that this was actually not the case, perhaps due to Stanford's very accessible and decentralized computing resources, and we tool; a moment to introduce ourselves.

During the quarter the class will work on four "unsolved" problems, the first one being the decryption of simple substitution ciphers from short samples of ciphertest, and the final three to be determined later. Although the class is expected to discuss the problems openly and freely, actual "solutions" to each problem will be produced in teams of three. JDU noted that, according to CS304 tradition, no person should be on the same team with any other person more than once, with the intent that people should become better acquainted with all of their colleagues. Solutions will consist of a brief (3-5 page) paper describing the problem and whatever variation(s) was considered, a proposed approach to the problem, and an evaluation of how well the chosen approach worked. Directions for future work (hypothetical, of course) should also be discussed if the solution is not complete or entirely satisfactory. JDU also remarked that the parameters of the problems were never immutable; if a problem was too difficult, it could be restricted, and if a. problem was too straightforward, it could be extended.

At this point, JDU paused to call for questions or comments, asking people to restate their names, since he had already forgotten them since the introductions. Nobody volunteered. JDU then amended, "Anybody want to speak anonymously?" ET wondered how the graduate students had faxed on the other problems. JDU answered that he was, in fact, not aware of any work on most of the problems, and that even if he was, he probably wouldn't tell. Perhaps novel and interesting approaches would be discovered that way. BL asked whether it was O.K. to call him Jeff ... "of course." The class then discovered that grading was a taboo subject; JDU simply appealed, "Skip a problem if you have to, but pour your hearts into it."

Finally, we turned to the problem of decrypting a simple substitution cipher. We consider permutations of a normal 26 character alphabet. JDU suggested a database oriented approach. Using any on-line dictionary (e.g. `/usr/dict/words` on Unix) we can generate n-ary relations consisting of words of length n. Let **word**n be the relation of words of length n, with each field corresponding to the appropriate letter. The selection operation σ can be used to select certain tuples (i.e. words) from a relation; thus $\sigma_{\$3=\$4}$(**word**5) will consist of tuples corresponding to

words with the same third and fourth letters. The projection operation π reduces the arity of a relation by projecting only specified fields.

Example:

$$\pi_{\$1,\$2,\$3,\$5}(\sigma_{\$3=\$4}(\mathbf{word5})) = \begin{array}{l} < H\ E\ L\ 0\ > \\ < F\ U\ N\ Y\ > \\ < F\ U\ R\ Y\ > \\ \quad .\quad . \end{array}$$

A third database operation is the binary join $\bowtie$. A join $\mathcal{R} \overset{\bowtie}{_{\$i=\$j}} \mathcal{S}$ would produce a relation whose tuples are generated from pairs of tuples, one from $\mathcal{R}$ and one from $\mathcal{S}$, such that the i^{th} field from $\mathcal{R}$ was equal to the j^{th} field of $\mathcal{S}$.

The database framework, therefore, can be outlined as follows.

1. Pick some ciphertext word and produce $\mathbf{word}n$, where n is of course the length of the word.
2. Select tuples based on repeated letters in the encrypted word (pattern match for equality) and project only the non-redundant fields. Let $\mathcal{R}$ be the resulting relation.
3. Repeat, 1. and 2.. naming the resulting relation $\mathcal{S}$.
4. Join $\mathcal{R}$ and $\mathcal{S}$, equating fields based on repeated letters in the encrypted words.
5. Repeat 3. and 4. for the remaining words.

Example: Given the ciphertext "ABCCD EDFCG" we generate the 7-ary relation

$$\pi_{\$1,\$2,\$3,\$4}(\sigma_{\$3=\$4}(\mathbf{word5})) \overset{\bowtie}{_{\$3=\$4,\$4=\$2}} \mathbf{word5}$$

which we expect to contain the tuple < H E L O W R D >. This would decipher the message as "hello world", but the relation may contain other tuples if the solution is not unique.

Some strategy is required in picking the words to attack. Peter suggested picking words that had the most letters in common; JDU agreed, adding that the most restricted words would have the smallest relations and would therefore be most, efficient. JDU also remarked that the ciphertest "ABBCB" was especially useful, since it had only a single possible translation i.e. "geese", "asses", or "error". "So I was wrong; it happens!" CJ suggested that very long words would also have small relations.

There is still another type of constraint that has yet to be accounted for. Since a permutation is injective, two different encrypted letters cannot represent the same decrypted letter. In the above example, for instance, the tuple < H E L O D L Y > will also be in the generated relation.

corresponding to the incorrect decryption "hello dolly". At some point,-it is necessary to select for inequalities.

A lively discussion about when to select for inequalities ensued. If we select for inequalities at the very encl, as a postprocess, we will in general produce a number of intermediate relations with incorrect tuples. This may make the join operations significantly more costly (selection and projection are linear, a. more palatable excess). On the other hand, it may be even worse to check for duplicates after (or even during) every join.

JDU suggestecl that the size of the intermediate relations may peak in the middle. If so, it may be advantageous to check for duplicates after every join only when the size of the relations are near the peak, and then post-process to eliminate the (hopefully) few incorrect_8 decryptions.

PF suggested making use of letter frequencies as an alternative approach to decryption. If the ciphertest is long enough, such a statistica. analysis would probably be significantly easier than the database oriented approach. For short samples, however, a. statistical approach will probably not work. It may be worthwhile to consider how long the ciphertext should be before switching techniques. Also, the statistical approach will not yield all possible decryptions.

As the class drew to a. close, we discussed a few of the possible variations for the problem. If one of the encrypted words could possibly come from outside the dictionary (i.e. a wildcard), the brute force approach of simply ignoring each word, one at a time, would yield potential solutions with about n times as much computation, where n was the number of words in the ciphertest. Similarly, if i words could come from outside the dictionary, the computation would be $\binom{n}{i}$ times harder.

CW suggestecl ciphertexts in which word breaks were hidden, which is certainly one of the more difficult variations if we assume the ciphertests remain short.

JDU promised to bring an article on the ENIGMA machine to class Thursday. Remember, "it's supposed to be fun; it's supposed to be interesting. If it's not, tell me about it!"

Day 2 — April 5

(An encryption program has been made available: look on portia for `~ullman/crypt` and source in `~ullman/crypt.c`.)

PKW and BJG announced that they had implemented a. solution for the simple substitution

ciphers problem, using the database techniques described earlier. A simple heuristic[1] was used to determine which words to join, and after every join, tuples that did not satisfy inequality constraints were removed.

DS asked how they handled the overhead involved in generating the basic relations (e.g. word5). PKW answered that they preprocessed the dictionary, and he described a coding and sorting scheme on `/usr/dict/words` that allows for efficient extraction of and selection on basic relations. Consider only words that are no longer than sisteen characters.[2] Obviously there are at most sixteen distinct characters in every word, and each character may be assigned a 4-bit code. The coding is then extended to entire words, producing 8-byte codes. The dictionary is sorted according to these codes, and then any ciphertest pattern can be matched in $O(\lg n)$ time by binary search of the words ordered by their codes. If an index is added, the time required for matching is constant. The tuples extracted in this manner satisfy local equality and inequality constraints, although the global problem of satisfying inequality constraints between words still remains.

One concrete result is that there are almost 1,900 solutions to the ciphertest "NON SCHMOZ KAPOP", including the cryptic "nun holdup jesus".

DS asked if the class was required to use any particular programming language. JDU said, "No, use whatever language is most suitable." PKW and BJG used C, and after some classroom discussion comparing C to LISP, BJG hypothesized that using LISP would require more memory than any equivalent C program. CJ argued that the tradeoff should include implementation time, and that, presumably, better algorithms could be found in the time saved by programming in LISP. Laker on, JDU wondered whether it would be feasible to implement a decryption program using only the Unix shell and *awk*, *sort*, and *join*. At any rate, PKW noted that, as feared, the size of intermediate relations was frequently several orders of magnitude larger than the eventual output. To alleviate this problem somewhat, all prefixes, suffixes, and proper nouns were eliminated from `/usr/dict/words`.

Rather than actually computing the intermediate relations, ET proposed a depth-first join algorithm. Only the relations generated by each individual word would actually reside in memory; the output relation would be calculated on the fly. There was some discussion as to whether this approach would yield a solution in a reasonable amount of time: it was generally agreed that finding a.11 solutions would be prohibitively time consuming. JDU commented that the question at hand was the choice between early-binding and late-binding; any two relations could be called joined, then delaying all computation until some operation was performed on the "result." JDU

[1] Number of letters in common divided by the product of the words' relation sizes.

[2] There are only seventeen words longer than sixteen characters.

turned to the possibility of 3-way joins,[3] as suggested earlier by KF. First,, the naive algorithm for a two-way join is known as the "nested loop join," i.e. match corresponding fields while iterating through all pairs of tuples. As a. result,, the join of a relation with m tuples and a relation with n tuples requires $O(mn)$ time. A better algorithm is the "sort join," in which the tuples of each relation are sorted by the field matched in the other relation. If more than one field is to be matched in the join, lexicographic ordering can be used. Following these sorts a process akin to merging will produce the joined relation. Using a sort join, $'R \bowtie \mathcal{S}$ requires $O(n \lg n$ + mlgm + $|\mathcal{R} \bowtie \mathcal{S}|)$, w1ere $n = |\mathcal{R}|$ and $m = |\mathcal{S}|$. MT pointed out that the resulting relation is already sorted, which is useful if the subsequent join must match the same fields. To generalize a two-way nested loop join to a. three-way nested loop join is obvious; what is not, so clear is how to generalize a two-way sort join. Another variant is the "hash join," which is $O(n + m + |\mathcal{R} \bowtie \mathcal{S}|)$.

PKW observed that, before the merge step of the sort, join, it would be possible to determine how large the output relation would eventually be. Assuming that there were several relations to be performed, it would be possible to order the joins such that the smallest relation would be generated at each step. JDU said that, while this was probably an example of a local optimization that might not be globally optimal, "local search techniques tend to be pretty good."

As an aside, JDU mentioned that, if a. sequence of joins had, in some sense, an acyclic structure, then the last join would be the largest. This would actually be a good thing, since it would imply that the correct output was relatively large compared to the intermediate relations. Unfortunately, because of the inequality constraints, any sequence of joins that arises in this problem would not be acyclic.

The semijoin operation, $\mathcal{R} \triangleright \mathcal{S}$, is $O(\,|\mathcal{R}| + |\mathcal{S}|\,$ j and can be used to "prune" tuples from $\mathcal{R}$. Intuitively, the operation of the semijoin is to remove from $\mathcal{R}$ a.11 tuples that do not match some tuple of $\mathcal{S}$ (in the appropriate fields). A subsequent join operation may therefore be substantially faster. $\mathcal{R} \overset{\triangleright}{\$i=\$j} \mathcal{S}$ is obtained by removing all tuples from $\mathcal{R}$ which do not match some tuple of $\pi_{\$j}(\mathcal{S})$.

Example: $\mathcal{R}$ =

A	B
1	1
2	2
3	3
4	4
5	5

$\mathcal{T}$ =

C	A
1	2
2	3
3	4
4	5

$\mathcal{R} \triangleright \mathcal{T}$ =

A	B
2	2
3	3
4	4
5	5

For any two-way join $\mathcal{R} \bowtie \mathcal{T}$, both $\mathcal{T} \triangleright \mathcal{R}$ and $\mathcal{T} \triangleright \mathcal{R}$ will generate reduced relations, i.e. no

[3] "Roughly the way I'm planning to take three of you and squash you together (if you don't form teams by Tuesday)!"

subsequent semijoin will be productive. For a series of joins, however, there is no *a priori* limit to the number of productive semijoins.

Example: $\mathcal{R}$ =

A	B
1	1
2	2
3	3
4	4
5	5

S=

B	C
1	1
2	2
3	3
4	4
5	5

$\mathcal{T}$ =

C	A
1	2
2	3
3	4
4	5

Using the relations defined above, consider ($\mathcal{R}$ ⋈ S) ⋈ 7. The sequence of semijoins

1. $\mathcal{R}$ = 'R ▷ $\mathcal{T}$
2. $\mathcal{S} = \mathcal{S} \triangleright \mathcal{R}$
3. $\mathcal{T} = \mathcal{T} \triangleright \mathcal{S}$

can be iterated productively until all three relations are empty, a rather inefficient way to obtain the empty relation. If we add the tuple < 1 5 > to $\mathcal{T}$, no semijoin operation is productive, but, the result of the joins is still empty.

The discussion then turned to the Enigma machine, used by the Nazis in World War II. JDU distributed a paper about the (successful) British efforts to decode messages from the Enigma. machine; for the remainder of the class, we tried to reach some understanding for how the machine worked.

Day 3 — April 10

The results of the voting were announced. The far and away winner was the traffic lights problem. The other two problems that we will consider are the abundant number problems and, after some debate,[4] the God game problem.

BJG turned discussion back to the decryption problem by noting that "the semijoin we discussed last time was most excellent." Two other groups have begun obtaining results, both using depth-first search algorithms as proposed earlier. MPF described his group's approach in some detail. The relations genera-ted by each ciphertest word are initially stored in memory. The smallest relation is selected, and the first tuple of the relation is used to reduce the remaining relations. Intuitively, we are assuming a particular decryption for the selected word. The smallest. remaining relation is then selected, and again its first tuple is used to constrain the other

[4]During which, BL surveyed, "How many people want to play God?"

relations. This process continues; if at any point one of the constrained relations reduces to the empty relation, we have reached a dead-encl. Otherwise, all tuples of the last remaining relation correspond to solutions. To find all solutions requires traversing the entire tree. ET announced that the full traversal was evidently not intractable; his group was able to find all solutions for a four word cryptogram in less than two minutes. JDU wondered whether depth-first search would fare as well in general, especially if all the word relations were several thousand tuples long. The claim was put forth that, in the worst case, depth-first search would perform no worse than any other technique.

PKW wondered whether these search techniques were actually different from the database oriented techniques, or whether they were simply alternative ways for computing joins. After some discussion, JDU suggested that the depth-first search was in fact similar to a nested loop join, but one in which the loop ordering is determined by relative relation sizes rather than by arbitrary tuple order. PKW pointed out that, if this were the case, depth-first search would be no more time efficient than a nested loop join, since loop ordering is immaterial when the goal is to obtain all possible solutions. The depth-first search, however, would be significantly more efficient in its use of space.

Some consideration was given to the treatment of singleton letters in the ciphertext. Consider the ciphertext word "ABC" and possible decryptions "bar" and "car." Suppose the letter "A" appears only once in the ciphertext message. Further suppose that the plaintest letters "b" and "c" are never assigned to any other ciphertest letter. The depth-first search from "car" would be redundant following the search from "bar", since all solutions to the message will be independent of the assignment to ciphertext "A". Nobody was quite sure how to take advantage of this characteristic.

The discussion turned to decrypting proper names.[5] One question which arose was whether it was possible to detect spurious solutions, i.e. those that provided incorrect decryptions for proper names. Unfortunately, it was pointed out that "there's no way to make the computer Zen the fact that there's a proper name." Instead, we can only assume that a proper name is present if no solution to the ciphertext can be found.

JDU suggested extending every relation with the "undefined" tuple, whose fields are undefined. This tuple would essentially play the role of a. wildcard. It was unclear how to extend either algorithm to account for this wilclcarcl without encountering a combinatorial esplosion.

Earlier, we observed that if at most one word from an n-word message was a wildcard, the computation would be at most $n+1$ times as hard."[6] JDU argued that the n + 1 subproblems were

[5]JDU commented that "Ullman" is already accepted by *spell* (i.e. is listed in `/usr/dict/words`), since he worked previously at Bell Labs.

[6]One decryption, assuming that there are no wildcards present. Then n more decryptions, once for each word, assuming it to be the wildcard.

closely related, and therefore the brute force technique could surely be improved. An obvious possibility is to join according to a binary tree. We expect the binary tree will require roughly lg the amount of time required by the brute force approach, although the actual time requirements depend on the relative sizes of the intermediate relations. JDU suggested that, conceptually, the tree might be "upside-down." Rather than constructing the join bottom-up from the leaves: there may be some way to construct it top-down from the root. Is there a better way to compute an *n-way* join, and subsequently the $n\ (12 - 1)$-way joins if needed?

As a final note, PKW challenged the other groups to decrypt "the quick brown fos jumped over the lazy clogs."

Day 4 — April 12

Since the Unix dictionary is really just, a word list for spell, it provides no semantic information. It is therefore not possible to determine which suffixes and prefixes are appropriate for any given word, leaving a sizable gap in our ability to decrypt actual ciphertest messages. One solution would be to arbitrarily append suffixes (and prepencl prefixes) throughout the dictionary, which seems to be how *spell* behaves. The immediate objection to this was that it would produce many spurious solutions, greatly increasing the complexity of the computation and producing further empirical evidence for the GIG 0 principle. A more promising option is for somebody with access to a NeXT machine to use its on-line Webster's dictionary to produce a more complete word list.

JDU began writing what appeared to be a ciphertest message, but which was in fact a plaintest sentence in which each letter appears exactly once. "squdgy fez blank jimp crwth vox." It seems unlikely that any of the programs yet written will solve this, especially since only one of the six words is found in the Unix dictionary.

Nobody present had solved "the quick brown fox. . . " either, although there were rumors that some missing team had. "No doubt resting on their laurels," JDU commented. The difficult?- with this sentence is the lack of equality constraints, resulting in joins that are very nearly direct products. The problem is exacerbated because the basic word relations are very large.

JDU returned to his earlier suggestion of pursuing a. top-down approach. One strategy is to partition the ciphertest words into groups such that there is very little "affinity" between groups? with affinity being a somewhat vague notion at this point. Each group could be recursively processed, if necessary and possible; the results would then be combined somehow. It was immediately pointed out that the affinity between words would depend not only on letter equalities, but also on letter inequalities, so there would be no way to partition a. ciphertext message such that each partition was entirely independent of the others.

At this point PKW suggested sampling to estimate the size of a. join operation. This imme-

diately reminded JDU of work by one of his previous Ph.D. students, Jeff Naughton. Rather than sampling some constant number of times, a more accurate technique is to sample until some (small) number of tuples were successfully joined. Consider the join of two relations with 1000 tuples each, with the result also being a relation of 1000 tuples. If we sample 1000 pairs of tuples from the original two relations, there is still a 1 in 3 chance that none of the pairs successfully join. This would suggest that the output relation was empty or estremely small. Instead we should sample the original relations until, for instance, 3 pairs' joined successfully. From the number of samples required, we can then obtain a more accurate estimate of the output relation size. Unfortunately, if the output relation is empty, this procedure will never terminate.

For our purposes, however, estimating the size of joined relations is of interest only for finding the locally optimal join order. Sampling can be used to determine this directly by organizing the possible pairs of join relations into a round-robin, and eliminating each pair after (for instance) 3 output tuples are found successfully. The final remaining pair will probably produce the smallest output relation, relative to the sizes of its input relations.

We turned again to estimating the affinity between two words. Suppose each basic word relation has n and m tuples, and each word has i and j distinct letters. Let k be the number of letters that the words have in common. The expression $nm(\frac{1}{26})^k(\frac{25}{26})^{(i-k)(j-k)}$ seems to be a reasonably intuitive estimate for affinity, although the constants should be adjusted somewhat to account for actual letter frequencies.

To get a better understanding for the affinity equation, consider a. three-letter alphabet { a, b, c } in which the respective letter frequencies are P_a = 0.6, P_b = 0.3, and P_c = *0.1*. The probability that two random letters will match is given by:

$$\sum_{L\in\{a,b,c\}} P_L^2$$

For the English alphabet this number is approximately p = 0.07; the affinity equation is then $nm(p^k)(1-p)^{(i-k)(j-k)}$. U sing these values ET calculated that the numerous inequality constraints from "QUICK BROWN" are roughly equivalent to a single equality constraint.

To estimate the potential benefit of a divide-and-conquer technique, we first analyze the cost of carrying out the straightforward computation. Recall that we are interested in a. k-way join, followed by k $(k-1)$-way joins. Assume the following:

- each relation consists of n tuples
- each 2-way join produces a relation with n tuples

[7] " '3' is an example of a small constant."

- both join and union are $O(n)$ operations

The initial k-way join requires $(k-1)O(n)$ time.
The k $(k-1)$-way joins require $(k-2)O(n)$ time each.
The union of all these relations requires (k + 1)$O(n)$ time."
Thus, the straightforward computation requires $k^2O(n)$ time.

Analysis of the divide and conquer technique is slightly more involved. Let 12. . . k be notation for the k-way join of relations $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$. $1 \ldots \bar{i} \ldots k$ is the $(k-1)$-way join, excluding the i^{th} relation. $\sum_{1<i<k} i \ldots \bar{i} \ldots k$ is then the union of the k $(k-1)$-way relations. Let C(k) be the total cost,. Then

$$C(1) = O(n)$$

$$C(k) = 2C(\frac{k}{2}) + O(n) + 2((\frac{k}{2})O(n)) + kO(n)$$

$O(n)$ is the join of the relations produced by the two $\frac{k}{2}$-way relations

$2((\frac{k}{2})O(n))$ is the cost, of the "cross" joins

$$1 \ldots \frac{k}{2} \bowtie \sum_{\frac{k}{2}+1 \leq i \leq k} (\frac{k}{2}+1) \ldots \bar{i} \ldots k$$

$$(\frac{k}{2}+1) \ldots k \bowtie \sum_{1 \leq i \leq \frac{k}{2}} 1 \ldots \bar{i} \ldots \frac{k}{2}$$

$kO(n)$ is the cost of the unions

Thus $C(k) = O(nk \lg k)$.

As we might have expected from the earlier discussion about join ordering according to a. binary tree, we can hope to replace a. factor of k with $\lg k$. This analysis is certainly suspect when our assumptions do not hold, especially if joins do not preserve relation sizes. Another caveat is that, by our problem clefinition, we do not, expect long ciphertexts; the improvement obtained over a brute force technique will be much less than an order of magnitude.

[8]Give or take a fencepost error.

Day 5 — April 17

We begin by reviewing preliminary results for the substitution cipher problem — see Table 1. Many of the programs removed proper nouns, recognized by an initial capital letter, from the dictionary. They were therefore unable to match "Christmas" and failed on sentence 2. In the third test case, the "no solution" results were blamed on the dictionary; curiously, "minicomputer" is in /usr/dict/words but "computer" is not.[9] ,411 "no solution" results were essentially immediate. The MPF/MT/CW team also solved the "quick brown fox" problem, finding approximately 1,000 solutions after several hours.

	BJG/PKW	ET/KF/JD	PS/AW		MPF/MT/CW	DS/BL/HH
1	1 min ~24K solns	1 min , 2-3000 solna.	1 min . [illegible] solns	immediate, ??? solna.	immediate, ??? solns.	out of memory
2	immediate, 18 solns	no solut ions	no solutions	no solutions	immediate, 10 solns.	no solutions
3	no solutions	no solutions	no solutions	no solutions	immediate, 4 solutions	immediate, 4 solutions

Sentence 1: My kingdom for a horse
Sentence 2: Merry Christmas and Happy New Year
Sentence 3: The Art of Computer Programming

Table 1: Preliminary Results

After we reconsidered various ways of deciding optimal join ordering, JD suggested that "Since humans are pretty good at pattern matching, a human operator should order the words of the ciphertest ." Certainly it would be interesting to see whether a person could regularly choose a more efficient join ordering than a computer, but it was agreed that some automatic heuristics should be provided in case there were a number of ciphertexts to decode.

JDU noted that the clepth-first approach that ET/KF/JD implemented used an unvarying join order throughout the search; as discussed earlier, it might be possible to reduce computation by determining join order on the fly. ET initially argued that the optima.1 join order would not change, but JDU produced an esample which seemed to contradict this. To determine locally optimal join ordering efficiently, HH suggested maintaining histograms of letter occurrences for each letter of the ciphertext; i.e. given the ciphertest "ABA" and a relation containing "bob." "bib," "did," and "dud," the histogram for 'B' would be one for 'o,' one for 'u,'and two for 'i.' He claimed that the estimates for output join relation sizes that his group had obtained were accurate to about 10%.

JDU reiterated his belief that there were ciphertexts such that the join of any subset of a certain size would be unmanageable; "SQUDGY FEZ BLANK JIMP CRWTH VOX" might be

[9] CW provided access to a more complete dictionary, estimated to be about 50% larger, located in `portia:~cwitty/class/204/ispell/words`.

one, since there are no equality constraints. Even if inequality constraints eventually reduced the number of solutions dramatically, none of the previously mentioned approaches seemed likely to overcome the huge intermediate relations. A semantic approach was considered briefly; by removing tuples whose known semantic properties didn't make sense, e.g. a preposition followed by another preposition, the intermediate relations might be reduced tremendously.

ESC suggested that if the sizes of all relakions obtained by joining different subsets of the ciphertext words could be determined, then it might be possible to avoid the worst intermediate relations. For example, a seven word cipher might behave such that any four or five words would produce unmanageably large relakions, whereas by the sixth and seventh words enough inequality constraints were created that the joins would be tractable. In that case, one could join two groups of three words, producing two large but tractable relations; these two relations could then be joined, bypassing the unmanageable intermediate relations.

To estimate the feasibility of such an approach, we considered a 5-word ciphertest consisting of 5-letter words, a.11 of whose letters are distinct. Each basic word relation is then roughly 1,000 tuples. Using the affinity expression. the join of two words will consist of approximately $1,000^2(.93)^{5^5} \approx 163,000$ solutions. Adding the third word would produce $163,000 \times 1{,}000(.93)^{10^5} \approx 4,328,000$ solutions. less than half of one percent of what would result from a straightforward 3-way Cartesian product. The justification for this estimate is that each letter of the third word is restricted by the ten letters of the first two words; we raise this restriction factor to the fifth power since there are five letters. The fourth word will generate $4.328 \times 10^6 \times 1{,}000(.93)^{15^5} \approx 18.7 \times 10^6$ solutions. After adding the fifth word, the number of solutions actually decreases somewhat, to about 13 million solutions.

PKW noted that it was very easy to find palindromes with the programs. MT mentioned that decrypting people's names often produced interesting results.

Day 6 — April 19

The *abundance* of an integer n, denoted $\alpha(n)$, is the sum of the divisors of n (including n itself) divided by n. We say n is *k-abundant* if $\alpha(n) > k$, and n is *super-abundant* if it is 3-abundant.[10] A *Phillips number* is an integer that is more abundant than any smaller integer. We distinguish a certain subset of the Phillips numbers, one consisting of the smallest k-abundant numbers, where k is an integer. We denote each of these 1,: α_k. Our goal is to find α_k, for as large a k as possible. JDU noted that α_k is doubly exponent ial — an earlier CS304 class obtained the

[10]Sometimes *abundant* is used to mean 2-abundant.

estimate $\alpha_k \approx 10^{1.7^n}$. The largest α_k which they obtained was α_{32}, which would be more than 23 million digits by the above estimate.

Before reaching such lofty levels, however, consider a. few small cases. a(6) = $\frac{1+2+3+6}{6}$ = 2, and in fact we have α_2 = 6. Similarly, we can show that α_3 = 120. KF observed that any perfect number is 2-abundant.[11] More generally,

$$\alpha(2^n) = \frac{1 + 2 + \ldots + 2^n}{2^n} = \frac{2^{n+1} - 1}{2^n}$$

. (Recall the espression for simplifying geometric series, $\sum_{i=0}^{n} x^i = \frac{x^{i+1}-1}{x-1}$.)

$$\alpha(2^n 3^m) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} 2^i 3^j}{2^n 3^m} = \frac{\sum_{i=0}^{n} 2^i \sum_{j=0}^{m} 3^j}{2^n 3^m} = (\frac{2^{n+1} - 1}{2^n})(\frac{3^{m+1} - 1}{2 \cdot 3^m})$$

It is easy to see that the abundance of any number can be easily found from its prime factorization, i.e.

$$\alpha(2^{i_1} 3^{i_2} \ldots p_n^{i_n}) = \prod_{k=1}^{n} \frac{p_k^{i_k+1} - 1}{(p_k - 1)p_k^{i_k}} \qquad \prod_{k=1}^{n} \frac{p_k}{p_k - 1} \frac{}{(p_k - 1)p_k^{i_k}} \tag{1}$$

Two other important facts are apparent from the above equation. First, multiplying a number by any other number can only increase its abundance. Be careful not to confuse this property with monotonicity. Only if we order integers by divisibility ($i \prec j$ *iff* i divides j) can we consider abundance to be monotonic. Second, there is an upper bound to the abundance that can be obtained by multiplying powers of any fixed set of primes; specifically,

$$\alpha(p_1^{i_1} p_2^{i_2} \ldots p_n^{i_n}) < \prod_{k=1}^{n} \frac{p_k}{p_k - 1}$$

From another perspective, this puts a. lower bound on the largest primes needed to reach any level of abundance.

Equation 1 allows us to consider the effect on abundance of multiplying by some number. For example,

$$4360) = \alpha(120 \cdot 3) = \alpha(120) \cdot \frac{\frac{3^3-1}{2 \cdot 3^2}}{\frac{3^2-1}{2 \cdot 3}} = \alpha(120) \cdot \frac{13}{12} = 3.25$$

$$\alpha(840) = \alpha(120 \cdot 7) = \alpha(120) \cdot \frac{7^2 - 1}{6 \cdot 7} = \alpha(120) \cdot \frac{8}{7} \approx 3.43$$

[11] A perfect number is equal to the sum of its divisors, excluding itself.

Although multiplying by 3 did not produce as large an increase in abundance[12] as multiplying by 7, JDU was quick to point out that this did not necessarily mean than multiplying by 7 was better. If we were interested in finding the smallest r-abundant number, 360 would be better than 840.

To find α_k for large k, we briefly considered an incremental approach. Starting with $\alpha_3 = 120$, for instance, there might be some algorithm for determining the smallest factor to multiply with 120 that would produce a 4-abundant number. It was quickly recognized that this would only work under the assumption that α_k divides α_{k+1}, an assumption that is not true in general.

PKW suggested that, if some suitable enumeration of integer sequences could be found, where each integer sequence specified the powers of a prime clecomposition, then binary search could be used to find large α_k. Determining such an enumeration, however, intuitively seems at least as hard as the original problem.

PF commented that it was bad to have "holes," i.e. zeroes, in the prime clecomposi t ion; JDU agreed immediately. In fact, CW recognized that the sequence of powers in the prime decomposition of a Phillips number must be monotonic (decreasing, of course), and we finished by sketching a proof of this fact.

Day 7 — April 24

Nobody had formally proven that the esponent sequence in the prime decomposition of a Phillips number must be monotonically decreasing (henceforth we will refer to this property as monotonicity, leaving implicit that it. is the exponent sequence of the prime decomposition that we are consiclering). In attempting the proof ET found that the ratio $\frac{p}{\alpha(n\cdot p)/\alpha(n)}$ seemed significant. Given a Phillips number n, the largest abundance that can be obtained by multiplying n by a single prime factor is $n \cdot p$, where p is the prime which minimizes the above ratio. By monotonicity, there are only a finite number of primes that need be considered: this leads immediately to an algorithm for finding numbers of arbitrarily large abundance.

Unfortunately, there is no easy modification of this algorithm for finding α_k. For instance, we may be tempted to find some k-abundant number using the above algorithm, then try replacing the last prime factor by some smaller prime such that the abundance remains $\geq k$. We didn't pursue this type of approach further because, intuitively, this limited form of backtracking will not, be sufficiently powerful. However, nobody was able to produce a convincing counterexample at the time.

[12]Sometimes referred to as the "abundance hit" obtained from 3.

Assuming monotonicity, very compact number representation schemes can be devised. One obvious method is to simply store the differences between subsequent exponents. JDU claimed that this representation was still inefficient, since there would be many zeroes (i.e. for the numbers that we are interested in, the exponents of neighboring primes vary very slowly). Instead, given the number $2^{i_1} \ldots p_k^{i_k}$, we can store the index of the largest prime with k factors, for k from 1 to i_1. With this representation JDU commented that the largest α_k computable was not limited by the size of the numbers, but rather by the precision with which the abundance could be computed. Even using logarithms, since it is hard to accurately compute the product of numbers close to one, we encounter problems when p_k becomes large. <<In class we obtained the estimate $\ln(\alpha(n)) \approx \sum \frac{1}{p_k^{i_k+1}}$ but I wasn't able to reproduce this. Can anybody?>

$$\begin{aligned}
\alpha(2^{i_1}3^{i_2}\cdots p_k^{i_k}) &= \prod_{k=1}^{n} \frac{p_k^{i_k+1}-1}{(p_k-1)p_k^{i_k}} \\
&= \prod_{k=1}^{n} 1 + \frac{p_k^{i_k} - 1}{p_k^{i_k}(p_k-1)} \\
\ln(\alpha(2^{i_1}3^{i_2}\cdots p_k^{i_k})) &= \sum_{k=1}^{n} \ln(1 + \frac{p_k^{i_k}-1}{p_k^{i_k}(p_k-1)})
\end{aligned}$$

If p_k is large, we will be computing $\ln(1 + \frac{p_k^i - 1}{p^i(p-1)}) \approx \ln(1 + \frac{1}{p}) \approx \frac{1}{p}$.[13]

At this point, JDU offered a geometric interpretation of the problem, viewing each prime as a rectangle with area $\frac{1}{p^{i+1}}$ and width lnp. Our goal is to find the sequence of rectangles with smallest total width, which has reached some threshold of total area; the total area corresponds to the ln of the abundance. <Unless somebody can prove $\ln(\alpha(n)) \approx \sum \frac{1}{p_k^{i_k+1}}$, I'm not sure that this interpretation is correct .≫

We decided to finish the monotonicity proof. Suppose we are given some number $n = \cdots p^i \cdots q^j$, where $p < q$ and $i < j$. Let $\hat{n} = \ldots p^{i+1} \ldots q^{j-1}$. We claim a($\hat{n}$) > $\alpha(n)$ and $\hat{n} < n$. We need to show

$$> \frac{(\frac{p^{i+1}}{(p-1)p^i})(\frac{q^{j+1}-1}{(q-1)q^j})}{(\frac{p^{i+2}-1}{(p-1)p^{i+1}})(\frac{q^j-1}{(q-1)q^{j-1}})}$$

[13]Unless $\frac{1}{p}$ begins to approach the limits of double floating point accuracy, *in my opinion* the standard math library should be sufficiently accurate (using ***loglp,*** for instance). Of course, I'm not the one writing these programs, nor do I have the experience of previous 304 students who claimed to encounter math library problems. If you really do want to implement arbitrary precision math functions, try looking at *bc* first; it may save you some work.

$$
\begin{aligned}
&> \left(\frac{p^{i+2}-p}{p^{i+2}-1}\right)\left(\frac{q^{j+1}-1}{q^{j+1}-q}\right) \\
&> \frac{1-\frac{p-1}{p^{i+2}-1}}{1-\frac{q-1}{q^{j+1}-1}} \\
\frac{q^{j+1}-1}{q-1} &> \frac{p^{i+2}-1}{p-1} \\
1+q+\cdots+q^{j} &> 1+p+\cdots+p^{i+1}
\end{aligned}
$$

which follows from our assumptions.

Let us estimate the optimal relationship between the exponents of two primes, for instance 2 and 3, if we allow non-integer exponents. We require $2^{\epsilon} = 3^{\delta}$ so that the number itself does not change, and try to maximize the abundance. Using the estimate for the abundance hit provided by the $(i+1)^{st}$ factor of p, i.e. $1 + \frac{1}{p^{i+1}}$, we want,

$$(1+\frac{1}{2^{i+1}})^{\epsilon} = (1+\frac{1}{3^{j+1}})^{\delta}$$

so $i \approx j \log_2 q$. Thus, if p_{max} is the largest prime factor, which we expect will have an exponent of 1, the largest prime with an exponent of n should be about $\sqrt[n]{p_{max}}$.

Day 8 — April 26

Earlier in the discussion JDU claimed that α_k is doubly exponential in k, and that empirically $\alpha_k \geq 10^{1.7^k}$. The argument for this is as follows. Let p_k be the largest prime factor in n, where n is a Phillips number.

$$
\begin{aligned}
\alpha(n) < &\prod_{i=1}^{k}(1+\frac{1}{p_i-1}) \\
&\lesssim \prod_{i=1}^{k}(1+\frac{1}{p_i}) \\
\ln\alpha(n) &\lesssim \sum_{i=1}^{k}\frac{1}{p_i} \\
&\lesssim \sum_{i=2}^{p_k}\frac{1}{i}\,(i \text{ is prime}) \qquad (2)
\end{aligned}
$$

From number theory we know that the number of primes less than or equal to x is approximately $\frac{x}{\ln x}$; therefore we assume that primes are, on average, ln x apart, despite great non-uniformity in the actual prime distribution. Given this, we can further simplify Equation 2 but "spreading out" the prime number singularities, i.e.

$$\begin{aligned} \ln \alpha(n) &\lesssim \sum_{i=2}^{p_k} \frac{1}{i \ln i} \\ &\lesssim \int_2^{p_k} \frac{1}{x \ln x} dx \\ &\lesssim \ln \ln p_k \end{aligned}$$

Thus we have that $\alpha(n)$ is 0(ln p_k).

Next, given the largest, prime factor p_k, we find a lower bound for n; we will find that ln **12** is $\Omega(p_k)$, which implies p_k is $O(\ln n)$. Combining this with the last result gives $\alpha(n)$ is $O(\ln \ln n)$. Let $p_j \approx \frac{p_k}{2}$. Using the prime density result stated earlier, we estimate the number of primes less than p_j to be

$$\begin{aligned} &\approx \frac{p_j}{\ln p_j} \\ &\approx \frac{p_k}{2111 \frac{p_k}{2}} \\ &\lesssim \frac{p_k}{1.5 \ln p_k} \end{aligned}$$

so the number of primes between p_j and p_k is $> \frac{p_k}{3 \ln p_k}$, each of which is greater than p_j. Thus

$$\begin{aligned} n &> (\frac{p_k}{2})^{\frac{p_k}{3 \ln p_k}} \\ \ln \ ?\mathrm{I} &> \frac{p_k}{3 \ln p_k} (\frac{p_k}{2}) \end{aligned}$$

Now we have ln n is $\Omega(p_k)$, as desired.

If we graph n *vs.* $\alpha(n)$, we see that a Phillips number has no points "northwest" of it.

Let, p_k be the largest prime factor of α_k. We can find a lower bound on p_k by using the upper bounds on the abundance that can be obtained from a single prime; we can find an upper bound on p_k by assuming that there is only a. single factor of each prime. Each possible p_k represents a family of Phillips numbers, one of which will contain α_k. For each family, we can determine upper and lower bounds on the permissible exponent sequences, thereby defining a search space in which to look for α_k.

To find some lower bound, fix p_k. MN recognized that any q such that $q^2 + q \leq p_k$ must. have an exponent of at least two; otherwise the exponent sequence wouldn't correspond to a Phillips number.

$$\begin{aligned} 1 + \frac{1}{p_k} &\leq \frac{\frac{1+q+q^2}{q^2}}{\frac{1+q}{4}} \\ &\leq \frac{1+q+q^2}{q+q^2} \\ &\leq 1 + \frac{1}{q^2+q} \end{aligned}$$

This type of reasoning can be extended to find primes which must have esponents at least three, and so on. JDU hinted that an even better lower bound can be found. Also, he suggested that the search space can be reduced dynamically, since the upper and lower bounds will tend to converge as we proceed "deeper" in the search.

Day 9 — May 1

JDU presented some of the α_k obtained previously.

$$\begin{aligned} \alpha_1 &= 1 \\ \alpha_2 &= 6 \\ \alpha_3 &= 120 \\ \alpha_4 &= 27720 \\ \alpha_5 &= 2^4 3^3 5^2 7 \cdot 11 \cdot 13 \cdot 17 \\ \alpha_6 &= 2^7 3^3 5^2 7^2 11 \cdots 29 \\ \alpha_7 &= 2^7 3^4 5^3 7^2 11 \cdots 59 \end{aligned}$$

One group (JD/ET) provided evidence that, the problem was not as "solved" as it might seem; the result obtained by the last class for α_5 was incorrect. Replacing a factor of 3 by a. factor of 2 yields a. smaller number that is still 5-abundant.

BL described the method that he and PS were using to prune the search space. To obtain abundance k, we can find upper and lower bounds for the exponent of 2. Choosing some value in this range to be the exponent of 2, we can find upper and lower bounds for the exponent of 3.

Specifically, the esponent of 3 can not be too large, or a factor of 3 can be replaced by a factor of 2 to yield a. smaller number with larger abundance. Similarly the exponent of 3 can not be too small, or a factor of 3 can replace two factors of 2. This reasoning can be repeated to find bounds for larger primes, using the largest of all smaller primes to constrain the exponent range. From the bounds on the exponents, it is possible to compute α_{max} and α_{min} for any prefix of an exponent sequence; these abundance bounds can then be used to reduce the search.

ET described the method that he and JD were using. First, they find some number that is k-abundant , presumably using the $\frac{p}{\alpha(n \cdot p)/\alpha(n)}$ ratio discussed last week. Next, find some "lower base," i.e. a number which must be a factor of α_k. This number serves as a base in the sense that, when viewing the histograms of the prime exponents, the lower base seems to support α_k. Rather than searching between min/max esponent curves, we search upward from the lower base, using the known k-abundant number as a guide. CJ argued that it might be more efficient to search left to right, as opposed to bottom-up. JD thought that this might have the same effect as multiplying the lower base by increasingly large factors, an approach they had also considered.

En route to calculating α_{20} HH and MN had found a. lower base with abundance 19.98; this required around 7400 primes, the largest of which was approximately 75,000. JDU wondered how close the lower base was to α_{20}; quick estimates show that at least 75 more abundance hits would be needed to produce the extra 0.02 in abundance. A conservative estimate is that the 19.98-abundant number is too small by a factor of at least 300 digits.

Day 10 — May 3

Several of the teams commented that they had programs which could very quickly produce $\alpha_1, \alpha_2, \ldots$ up to about α_{10}, but which would then encounter search spaces that were orders of magnitude larger than previously encountered. JDU said that this was probably to be expected, because of the great non-uniformity in the abundance. He cited an example in which an earlier team was able to reduce the search for α_{18} to only three possibilities, although for smaller k they still had to search among hundreds or thousands of possibilities. People were suitably impressed that the search space could be so tightly restricted; JDU also noted that most of the computation then involved prime generation, rather than searching and testing.

PKW and BJG found $\alpha_7 = 2^7 3^4 5^2 7^2\ 11 \ldots 53$ rather than the α_7 mentioned previously; JDU clarified that the previous values had been conjectures, not carefully obtained results.

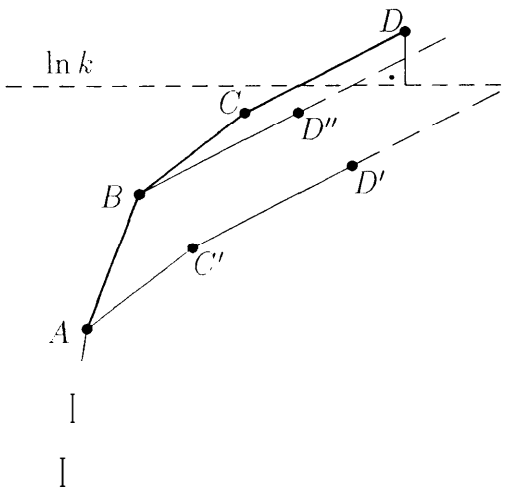
ET asked whether there might be an upper limit on the abundance. Using very rough esti-

mates, we argued that arbitrarily large abundances could be obtained, since

$$\begin{aligned}\lim_{n\to\infty} \alpha(n) &\approx \lim_{k\to\infty} \prod_{i=1}^{k} 1 + \frac{1}{p_i} \\ \ln \lim_{n\to\infty} \alpha(n) &\approx \lim_{k\to\infty} \sum_{i=1}^{k} \frac{1}{p_i} \\ &\approx \lim_{k\to\infty} \ln\ k\end{aligned}$$

MPF offered to describe the conceptual framework, developed by CW, that his team was using. Consider a logarithmic graph of abundance. Every prime factor corresponds to a line segment. The j^{th} factor of p_i corresponds to a line segment with width $\ln p_i$ and height $\ln \frac{p^{j+1}-1}{p^{j+1}-p}$; the height is the natural log of the abundance hit for the j^{th} factor of p_i. To find α_k we need to find the set of line segments with the smallest total width and height at least $\ln k$.

First we use a greedy algorithm, choosing line segments with the largest slope. Placing these segments end to end on the graph produces a convex hull of attainable abundances. Once the hull estencls beyond k, i.e. once we have found a. k-abundant number, we begin checking candidates for α_k. In the diagram below, we represent, the convex hull with thick lines. Consider the triangle



formed by $\overline{CD}$, the vertical line through D, and the line representing the abundance threshold. We claim that α_k must lie in this triangle, i.e. the line segments corresponding to the factors of

α_k must terminate in the triangle when placed end to encl. No point to the outside of $\overline{CD}$ can be reached since $\overline{CD}$ is part of the convex hull. Any point to the right of D is larger than the already known k-abundant number, and any point below the abundance threshold is insufficiently abundant.

In addition to the search constraints imposed by monotonicity, consider the following argument. The prime factor represented by $\overline{AB}$ can not be replaced, otherwise the best possible abundance must lie beneath $\overline{AC'D'}$. On the other hand, replacing the factor represented by $\overline{BC}$ could possibly lead to a smaller k-abundant number; the estension of $\overline{BD''}$ represents an upper bound on abundance without $\overline{BC}$. Since this passes through the triangle, it is possible that some sequence of line segments can be appended to D", such that the new sequence represents a smaller k-abundant number. This is possibility is shown in the diagram as a small dot just below the extension of $\overline{BD''}$. If such a sequence is found, we can reduce the triangle by "drawing" a vertical line through the point corresponding to the new number, thereby making the triangle even harder to "hit." In a. sense, we are comparing the slope and magnitude of the line segment corresponding to each factor with the "height" of the triangle, and using this comparison to limit the search.

Day 11 — May 8

Several of the groups presented the largest α_k that they had obtained. BJG and PKW were able to find

$$\alpha_{15} = 2^{15}3^85^67^411^313^317^323^329^2 \cdots 97^2101 \cdots 4583$$

CW/MT/MPF claimed

$$\begin{aligned}\alpha_{33} \;=\; & 2^{30}3^{18}5^{13}7^{10}11^813^717^719^623^629^5 \cdots 53^559^4 \cdots 139^4 \\ & 149^3 \cdots 683^3691^2 \ldots 14639^214653 \cdots 111346919\end{aligned}$$

although nobody was able to verify this in class. JDU promised to compare the α_{32} obtained this quarter with the α_{32} obtained previously, with the intent of increasing our confidence in the result if the two values agreed. MT noted that the determination of α_{32} required about 14 minutes. with most of the time consumed by the construction of the convex hull; the time required for the search was negligible.

BL presented the approach that his group had taken, with the hope that someone might be able to suggest why it didn't work; correct values up to α_7 were found, but the value produced for α_8 was not the smallest possible 8-abundant number. The basic technique was to determine the smallest number with a k-abundant idea.1 curve, i.e. in the case in which we allow non-integer

esponents. Supposedly, the actual exponent curve of α_k should be very close to the ideal curve, such that the actual exponent of every prime is less than one factor different from the ideal exponent. Simply testing the abundances at each "step" of the esponent curve should then yield α_k. JDU hypothesized that this would instead find Phillips numbers that were "distant" from other Phillips numbers, possibly missing α_k if α_k was in a neighborhood densely populated by Phillips numbers.

We turned to the God problem. Starting with the A♠ on the table, the goal is to play as many cards as possible according to some unknown God rule. The game ends when either all cards have been played, or when none of the remaining cards can be played successfully. The score of each game is determined by subtracting the number of plays that succeeded from the number of plays attempted. The God rule may be based only on the stake of the stack of successfully played cards, with each card represented as its rank and its suit. Thus, it is not possible for the rule to specify that there must be n failures before another success, where n is the rank of the top card; this would require history information that can not be obtained from the stack of successfully played cards. The rule must also be deterministic; it is not permissible for the rule to specify that cards should be randomly accepted.

JDU reviewed some earlier results. Seven computer players were written by the earlier CS304 class; all seven played better than human players. When compared with the median score from t hree trials of a player that simply played random cards, a.11 seven players beat the random with at least ten of fourteen God rules; the best computer player beat the random player thirteen out of fourteen times. For the most part, the computer players required very little computation time, the exception being the best player, which required several minutes for each play. PKW asked if a player that simply cycled through his remaining cards had ever been tried; JDU answered that, in most cases, such a strategy fails miserably. In one surprising case, however, the cycling strategy happened to be a good strategy, and the cycling player beak all the other computer players.

Although the set of possible God rules, which we refer to as the *rule space*, is, for all practical purposes, infinite, there is in fact a limit to the number of distinct, rules. We can estimate this limit as follows. Assuming that A♠ is on the table, there are 2^{51} possible subsets of the remaining cards that could be the set of successful plays. Let us assume that, each of these subsets has about twenty-five cards, i.e. that there are twenty-five cards that may be played successfully. For each of these twenty-five cards, there are 2^{50} possible subsets that could be successful third play cards, so altogether 2^{51} x $(2^{50})^{25}$ possible rules can govern the first three cards. Then

$$\#\text{ of rules} \approx 2^{51+\sum_{i=1}^{50}\frac{i^2}{2}}$$

$$\approx 2^{\frac{50^3}{12}}$$
$$\approx 2^{10000}$$

MPF commented that it would be "impossible to do better than random without assuming some bias" in the God rules; JDU agreed, saying that the problem was to try to find that bias. Presumably, he added, the God rules are "short," in the sense that they don't enumerate permissible plays in the set theoretic sense that we used above. BJG pointed out that wildcards in the rules would be difficult to account for, an example of which might be "the suits must alternate, except that black 7's may always be played."

According to JDU, the strategies used by the computer players in CS304 could be roughly divided into two classes — brittle and malleable. Brittle strategies used some reasonably large set of "features" that either applied or didn't apply, e.g. "♠ always precede cards with odd ranks." Malleable strategies were statistical in nature, ordering features according to how well they performed in the past.[14]

At the recent faculty retreat JDU listened to a. talk by John Koza, who does work with genetic algorithms. Since three people in this class are also taking Koza's class, JDU challenged them to implement a genetic algorithm. Ideally, such an algorithm would "splice" together features that worked in certain cases with features that worked in other cases. As a simple example, one feature might be reliable in predicting which cards could follow red suits, whereas another fea ture might regularly predict cards that could follow black suits. By splicing these two features, we would hopefully obtain guidelines for plays on any suit.

Day 12 — May 10

ER described a God player that he had written over the weekend. It was quite successful against the random player, winning seven out of eight trials on different God rules. The player uses a brittle strategy that only looks at the top card on the stack; the six features that the player considers are rank, suit, odd/even, face/number, color, and major/minor.[15] Once a card is played successfully, the player generates a set of generalized rules that might explain why the play was successful; each rule is a. pair of descriptors, one for the top card and one for the nest card. The set of rules is pruned after every failed play, so that all current rules axe consistent with the history of plays.

[14]JDU characterized this as being somewhat similar to the way in which baseball players were sometimes sent down to the minors, but could eventually work their way back up to the majors.

[15]Major/minor is from bridge; ♠ and ♡ are major, ♢ and ♣ are minor.

God Program	Player (2 random seeds) Seed = 34567		Seed = 54535		Random Trials Median	Range
My God 1	71	(25)	70	(25)	154	140-189
My God 2	87	(51)	104	(51)	196	126-204
Ullman's God 1	80	(51)	69	(51)	92	83-106
Ullman's God 2	74	(35)	99	(51)	113	88-126
Ullman's God 3	170	(49)	205	(46)	167	161-M
Ullman's God 4	92	(32)	98	(34)	118	101-131
Ullman's God 5	111	(48)	91	(48)	131	118-154
Ullman's God 6	74	(51)	65	(49)	87	81-107

Table 2: Performance of ER's God Player

Consider a. successful play of the 3♡ on the A♠. One generalization of this would be "odd V's may follow major suits." For this rule, the rank, odd/even, face/number, color? and major/minor features of the top card would be "don't cares," moclulo the interdependence among some of the features. The only feature of the next card that would not be a "don't care" is major/minor.

To choose the nest, card, randomly select among the rules which apply to the current top card. The rule discussed above applies to the 3♡, since V's are major. Suppose this is the rule randomly chosen. Now randomly select among the remaining cards which satisfy the rule, in this case, A♡, 5♡, . . . , 9♡. When there are no rules which apply, the player tries cards randomly.

ER noted that every successful play generated between 20-80 rules, while every failed play eliminated 0-40 rules. Two less random nest card selection techniques were also tested; choosing the nest card by letting the rules "vote" sometimes worked better, but only inconsistently. ET suggested the inverse of this, i.e. choosing the card which satisfied the fewest of the rules. ER argued that playing the most popular card was better-, since that would allow for the greatest amount of pruning. Even so, there were often more than a thousand rules left by the end of the game. The second technique was to choose the next card by using the rule with the largest set of permissible plays; this worked very poorly, presumably because such rules are too general. ER noted that, although he did not intentionally bias in favor of either more general or more specific rules, more genera.1 rules tended to be used more often since they applied to the top card more frequently.

JDU observed that "the interpretation of the rules is that they are permissive but not required," since the player doesn't prune the rule set on successful plays. In some sense. this

interpretation provides disjunction "for free." JDU digressed slightly, mentioning that very naive algorithms can usually solve Mastermind puzzles by the fourth or fifth try, because the feature set describing a Mastermind puzzle is fixed and relatively small. For the God problem, "the minute you start adding new features, you end up out of virtual memory, and that happens very quickly." In particular, this may happen if we indiscriminately estend the top card descriptor to consider the entire stack.

CJ suggested that players which considered only the top card might still do well on complex God rules, since the top card might unexpectedly manifest influences from all cards in the stack. In certain cases, there may be "top card only" God rules that are algebraically equivalent to God rules which consider the entire stack, but JDU argued that this wasn't true in general. For a God rule which only considered the card below the top card, for instance? there would be essentially two separate God games being played.

The feature set certainly doesn't have to be exhaustive; as HH noted, "You can't always hope to hit on the exact rule that the God is using, but you can try to approximate the God's behavior." JDU provided an example of this. Consider a God rule in which the parity of the sum of ranks and suits of cards on the stack must alternate. Even programs which entirely lacked the notions of suit numbering and parity were able to do reasonably well. They were able to notice that "odd" cards, i.e. those for which the sum of the rank and suit is odd, were twice as likely as "even" cards to be successful.

This notion of behavioral approximation seems to argue in favor of malleable algorithms, but JDU was quick to point out that brittle algorithms were also quite successful. Consider a God rule in which cards are divided into seven equivalence classes? by taking the square of the rank modulo 13. The rule requires that each card played must be either in the same equivalence class as the top card or in the nest equivalence class. Thus, cards of the same rank could always be played in sequence. Brittle strategies, although never actually "learning" this fact, would play all cards of a given rank after only a few false starts. Also, JDU warned against relying too heavily on statistical approaches. If you start with a large set of possible rules, it is statistically likely that some of them will do very well for the first few plays; however, there is little reason to expect that t-hey will have the ability to predict subsequent plays.

Although there is the informal notion that, the God rule should always allow at least ten cards to be playable, assuming that there is an infinite deck of cards, JDU declined to formally specify this requirement; he thought it would be too hard to do. We will all presumably follow this guideline in good faith. Every team should construct one or two God rules for the final competition; remember that the God rules should try to distinguish between players, so that some God players will perform well and others will perform poorly. ET proposed telling one of the other teams what God rules he would be using, but he reconsidered after being threatened with a. Fundamental Standards violation.

Day 13 — May 15

ET introduced an approach that JDU characterized as being "something of a neural nets approach, but instead of starting with chaos, having some built-in structure." For each feature of some moderately large feature set we can associate a histogram representing how strongly the feature recommends each remaining card. There is also a weighting function on the feature set, corresponding to the past performance of each feature; using this weighting function we combine all of the histograms to determine the nest card play.

At this stage only successful plays are used to adjust the weights; AW pointed out that we should be able to improve the technique, since "misses are as valuable as hits." ET also mentioned that he hoped negative weights might be able to represent negative correlation between features and successful card plays, although at this stage the idea was purely speculative.

An interesting heuristic that his team is developing attempts to recognize when the technique is failing, i.e. the "flail factor." Presumably, the number of failed plays between each successful play should decrease as the feature weightings more closely approsimate the God rule. If, in the middle of a game, this number suddenly increases for several successive card plays, there is reason to believe that the feature weightings are either no longer correct or, for some reason, locally inapplicable. There are several ways to attempt "recovery." First, reconstruct the weighting from the point just before the flailing, since it might have been corrupted; there is no way to know whether the rule approximated by this weighting is now entirely useless, i.e. the God rule changed entirely, or whether the weighting might again be of use, indicating that we have simply encountered some temporary special case of the God rule. Nest, start from the initial weights and construct a weighting which considers only plays since the flailing began. This weighting is probably more accurate than the combined weighting over the entire sequence of plays. ET also recommended identifying the card which immediately preceded the flailing and avoiding "similar" cards in the future; this is probably inadequate, since there is no way to know that the card which triggered the change in the God rule didn't appear much earlier. JDU suggested that, the weighting could be modified more drastically to achieve greater adaptability. Rather than subtracting some constant from the weight of every feature that wasn't a successful predictor, we might divide the feature weight by a. small constant. After only a few failures the feature would contribute very little to the combined histogram.

There are some high-scoring God rules which we agree are unfair. For instance, a God may specify that any card may be played prior to the appearance of the suicide king (K♡), after which some estremely complicated rule is applied. Then some players will, mostly by luck, not. play the suicide king until late in the game, thereby scoring very well, whereas other players will trigger the complicated rule quite early and do very poorly. Since the score of a God rule is determined by how well it differentiates between players, this God rule would score well, entirely

due to chance. We also recognized that certain God rules would be simply too difficult for any player, and should therefore be disallowed. Among these include those rules which involve an arbitrary permutation of the cards, e.g. red-black where A♠ is really 3♢, 2♠ is really 4♣, 3♠ is 9♡, and so on. BJG thought that we might be able to prove that no player could do better than random against a God rule using arbitrary permutations, where JDU characterized "better than random" as being better by at least one standard deviation, about $\sqrt{n}$ for n cards played. KF suggested that we might avoid many of these unfair rules by requiring that God rules be written without the use of lookup tables. JDU asked whether we expected our computer players to play well against simple rules; he suspected that they would tend not, at least in comparison to human players. ESC suggested that it might be interesting to see how well a computer assisted human could play. Writing a computer assist for playing God would primarily involve determining how to extract and present the data as usefully as possible.

BL pointed out that none of the approaches discussed so far seemed likely to handle exclusive-or based God rules.

Finally, there are rumors that MT is working on a genetic algorithm, breeding four generations between each successful play, but he wasn't present to confirm this.

Day 14 — May 17

Before discussing God game problem, we previewed the upcoming traffic lights controller problem. Given an n by n street grid with traffic lights at each intersection, we would like to optimize the traffic flow according to varying rates of traffic on each street. JDU said, "my intuition is that the algorithms for high rates and low rates are very different." When traffic is light, we can get good performance by just letting them "breeze through" the map. When traffic is heavy, however, we will probably want them to pass through "in waves."

ESC described the interface between the traffic simulator and the traffic lights controller. Although for this problem we are assuming a square street grid consisting of only two-way streets, the traffic simulator was designed to allow for more complex scenarios. Keeping this in mind may help explain some of the more confusing indirections used by the simulator. The main simulation loop consists of three steps. First the simulator randomly "grows" cars, either by adding new cars along the edges of the map or by "desorbing" parked cars in the interior of the map. Next the simulator models the motion of the cars, accounting for (hopefully) realistic acceleration, deceleration, and driver behavior. There is the possibility that the car will be "absorbed" somewhere in the interior of the map; this would correspond to the driver reaching her destination and parking. At each intersection the driver may choose to turn left or right or to continue forward. The simulator keeps track of the number of cars which "hit" each traffic

bumper entering an intersection. In the final step of the simulation the traffic light controller is called to update the traffic lights.

The map (`MAPmap *`) consists of streets (`MAPsid`) and crossroads (XROADid). Henceforth, we assume that streets run east/west and avenues run north/south. Each direction of a two-way street[16] is considered separately, i.e. has a distinct `MAPsid`. Given a map, we can iterate through the list of intersections on the map. The first intersection will be at the northwest corner of the map, the second intersection will be just to the east, and so on. Thus, the intersections list is ordered row-major. At each intersection, we can iterate through the streets meeting at the intersection, starting with the northbound avenue `MAPsid`, then the southbound avenue, then the eastbound street, and so on. Given an XROADid and a `MAPsid` entering the intersection, we can reference the number of cars that have hit the bumper at the intersection, along the given street, in the past second. Other physical features of the map, such as distance between intersections and speed limits, can also be accessed. See the ***man*** pages for details.

JDU suggested that the problem would still be interesting even if we disallowed turns. "Part of the problem is dealing with uncertainty, which we can still get from absorption and desorption." Turns complicate the simulator because safety checking of turning cars in the intersection is difficult. Also, left turns are somewhat unrealistic since we assume that there is an infinitely long left-turn lane; to do otherwise would require simulating multi-lane traffic, which we can avoid in general.

ET described an interesting player behavior that his team had encountered? Against an odd-even God rule, the first card played was the 2♢. ET explained that it seemed reasonable to play the card that was "most different" from the bottom A♠, although he also argued that it wouldn't matter much since the first card played would essentially be random. Given that the 2♢ succeeded, the computer player reasoned that God was accepting all minor suits. Thus it played the 3♢,...,A♢,2♣,...,A♣ in succession, inadvertently satisfying the odd-even rule at every play. Similarly, the computer played 2♡,...,A♡,...,2♠,...,K♠ to empty the deck. This serendipitous behavior is due to the tendency of computer players to induce unintentional regularity in their sequence of card plays, a phenomenon that AW/DS had also encountered. ET also mentioned that his team had decided not to pursue the "flail factor" heuristic, and that they had settled on a means for updating the histogram weights. Rather than incrementing or multiplying the weights by some constant, they adjust the weights according to the strength of the histograms' recommendations. Thus, if some rule very strongly recommended playing the 6♣, then the rule's

[16] Here we are using "street" to refer to both streets and avenues; hopefully, the context will make it clear when "street" refers to north/west roadways as opposed to roadways generically.

[17] He prefaced his description, saying that "the program was really trying, and it was completely our fault (that it wasn't really working)."

weight would be greatly decreased if the card was not accepted. Thus we would expect that the program could adjust quickly to sudden changes in the God rule.

JDU suggested that one might approach God playing from a sort of automata theoretic perspective, using states to represent "cards looked for," e.g. red face cards. Then the program would try to deduce the transitions resulting from each successful play. He also discussed another approach to feature sets. Rather than considering, for instance, rank sum modulo 2, rank sum modulo 4, rank sum modulo 13, and so on, to be primitives, we could start by saying that card rank is an "interesting" feature. Then, let the sum of any "interesting" feature over the entire stack be "interesting." Let the modulus of any "interesting" feature be "interesting." We can generate a very large feature set with this approach, one which wouldn't be dangerously biased by human intuition. Presumably, some way of assigning priorities to the feature set would have to be constructed.

Day 15 — May 22

JDU asked people to give examples of God rules for which their computer players would play perfectly. There was some uncertainty as to what was meant by playing perfectly; were we interested in God rules for which the programs would use the entire deck in exactly fifty-two plays, or were we interested in God rules which the programs would eventually obtain complete understanding of the God rule? An obvious example which will work for the first case is the God rule which allows any card to be played. JDU amended the question, motivating it from his curiosity "to see whether people understand why their programs do well or not." KF mentioned that the program she was working on would play the red-black rule perfectly, but would make a few mistakes before solving the odd-even rule. She indicated that she usually understood the program's behavior.

There was some discussion as to whether unintentional regularity was advantageous or not. Most of the teams seemed to be working towards programs which would simultaneously build as many patterns as possible, thereby inducing many unexpected regularities. On the other hand, at least one team was explicitly choosing at random. from the most likely set of unplayed cards. They hoped that the advantages gained by recognizing false patterns would outweigh losing the occasional "lucky" games. This discussion led to the question of how people were generating rules and representing them. It turns out that most of the player strategies tried thus far are brittle and require (modest) re-programming to introduce new features. JDU suggested using S-expressions to represent rules, building trees in which the nodes would stand for operators and the leaves would stand for very basic operands, i.e. features. He also noted that some of the strategies lacked the tangible notion of a "rule," relying entirely on approximating the God rule

by "fitting pieces of things together." This would seem to indicate that the God game problem ought to be well-suited to genetic algorithms, as was suggested several days earlier. For instance, there is an immediate intuitive interpretation of "survival." On the other hand, it was pointed out that genetic algorithms do not seem to hold much promise for adjusting to sudden changes in the God rule, which JDU agreed was definitely a shortcoming.

PKW introduced "parameterized rules," an example of which might be "ranksum of the top x cards modulo y." He and BJG were writing a God player that could interpret an off-line database of around 50,000 of these types of rules, comparing the results by means of histograms. They make use of "meta-rules," or rules which are used to generate other rules.

The last aspect of the God game problem that we considered was the tradeoff between "discovering" concepts or including concepts as part of the God player's vocabulary. Demonstrations of the God players will be Tuesday, May 31. JDU asked that people try to set aside an additional hour before class for the demos, so we will begin testing them at 10 a.m. Also, each team should remember to construct one or two Gods prior to the testing for the other teams to play against.

We opened discussion on the traffic lights controller by making two assumptions. First, assume that the cars do not make turns. Second, assume that the cars are never absorbed or desorbed. JDU explained that, although we will eventually want to relax the second assumption, we will probably keep the first assumption throughout the problem. "It's not so much whether you can make left turns or not, but whether unexpected events can happen [that's interesting]."

Suppose traffic is light and moving in such a way that we know there will be no conflict; is it always acceptable to simply turn all the lights green? Consider two cars approaching an intersection at right angles to each other. Even if they are staggered so that they could both pass through without affecting the other, the simulator will complain if the lights are green in both directions (since that is generally an unsafe configuration). If the light in front of one of the approaching cars is red, however, that car will have to slow down, because it has no way of knowing that the traffic lights controller will change the light to green just in time for it to pass through. Thus it is not generally true that cars which don't necessarily conflict can be passed through without delay.

Several incidental concerns arose; first, if a car cannot stop in time to avoid passing through a red light, it will simply go through the light and try to leave the intersection as quickly as possible. Should another car already be in the intersection in a manner that a collision is possible, the simulator will warn that there may have been a collision? Second, you can be certain that no cars will suddenly appear in the middle of an intersection, at least not without triggering some

[18] We don't stop the simulation, however, so presumably the simulator drivers all have incredible reflexes and manage to avoid each other.

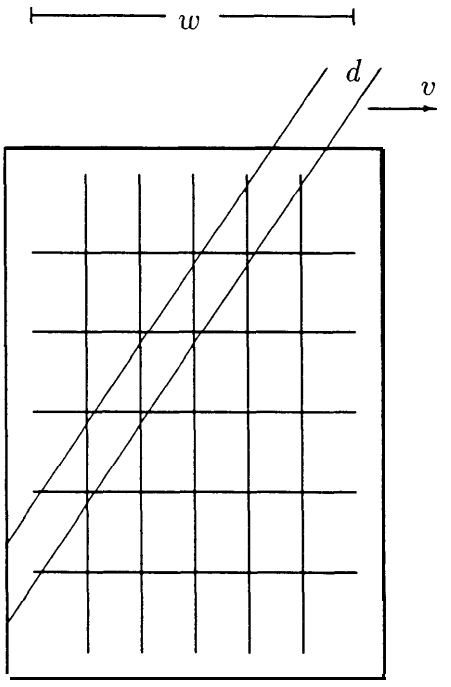
bumper in the usual fashion. In particular, cars which desorb begin with zero velocity, and so are able to come to a stop quite quickly. Finally, the speed limit is associated with the street, not a pair of intersections, so the only way to change speed limits on a straightaway is to join two streets end-to-end. It seems unlikely that we'll bother to do this.

One scheme for handling heavy traffic tries to move the cars through in waves, possibly with a diagonal rather than orthogonal wave front. JDU suggested dividing the map into quadrants, where, for instance, the upper left and lower right quadrants would favor east-west traffic, and the lower left and upper right quadrants would favor north-south traffic. Then, of course, the preferences would reverse. PKW suggested that elaborate wave schemes might not improve the throughput, as compared to simply letting all the east-west traffic flow for some relatively long time, then switching to all the north-south traffic. MK suggested that, rather than creating waves, we might let each car represent a sort of "mini-wave." Then we could set the traffic lights according to the strength of the superposition of all these waves. One consideration that arose several times was deadlock avoidance, but no definitive solution was proposed.

Day 16 — May 24

One abstraction for the traffic lights controller is to consider partitions of the map into (possibly disconnected) regions favoring either horizontal or vertical motion. The regions may alternate periodically, or sweep dynamically around the map. In this abstraction we are interested in determining the best shape and dynamic for the regions.

Consider a band favoring horizontal motion sweeping horizontally across the map.



In the diagram, the band is drawn as a diagonal. However, if we focus on a single street, it does not matter whether the band is diagonal or vertical; what matters is the size of the cross section intersecting the street in question and the speed with which the band is moving. In fact, we believe the optimum speed for the band is exactly the speed limit of the street; if it is any slower, cars will be held below the speed limit, but if it is too fast, cars will drop out from "behind" the band. Let v be the speed limit on the street, and let d be the size of the cross section. The time for a car, moving in the band, to travel across the map is $\frac{w}{v}$, where w is the width of the map. What happens to a car trying to move against the band? JDU claimed that the car should require $\frac{3w}{v}$, so that the average time to cross the map would be $\frac{2w}{v}$. He justified his intuition by arguing that "you can't get something for nothing?' We can provide a more rigorous argument for this. Suppose the car starts at the rightmost edge of a band favoring horizontal motion. Both the band and the car are moving at velocity v, but in opposite directions. The width of the band is d, and it will require $\frac{d}{2v}$ for the car to reach the leftmost edge of the band. Presumably, this will be the rightmost edge of a band favoring vertical motion, so the car will be forced to stop. Since the car is no longer moving, the vertical motion band requires $\frac{d}{v}$ to pass over the car. Thus, the car travels $\frac{d}{2}$ in time $\frac{3d}{2v}$, so the overall rate is $\frac{v}{3}$; to travel across the entire map requires $\frac{3w}{v}$.

ET and AW argued that this analysis was suspect, since "you never have to stop in the middle

[19]The *tanstaafl* principle.

of the street. If the streets are very long . . . you can synchronize so that you're driving while the vertical band passes over you." JDU agreed that edge effects might in fact prove to be important. ET suggested that we let d be equal to the length of the block; when the car reaches the vertical motion band, it will be exactly in the n-riddle of the block, so it can continue moving. Just as it reaches the end of the block, it also reaches the end of the vertical motion band, so the traffic light will turn green. AW pointed out that the throughput of traffic moving against the wave was still only half the throughput moving with the wave, since the "virtual band" in which traffic could move freely against the wave was only of length $\frac{d}{2}$.

In fact, the throughput will be even lower than this, since the lead car moving against the wave will be driving towards a red light. Since the driver has no way to know that the light will turn green just as he reaches the intersection, he will be forced to slow down. This provides a sort of "governor" mechanism, keeping the lead car just far enough behind the front edge of the virtual wave that the car won't have to slow down. With sufficiently dense traffic moving against the wave, we can expect that nothing will cross the intersection when the light turns green, but then suddenly "a bunch of cars will come flying through." To prevent this problem, AW proposed an "early warning" traffic light, so that drivers would know that the red light that they were approaching was about to turn green.[20] When we analyze the effect of these bands on traffic moving along the avenues, i.e. vertically, we find that it is possible to provide for smooth traffic flow in at least one direction. The other direction of vertical traffic, however, is forced to synchronize at every intersection, and tends to move rather slowly.

At the end, JDU noted that this was a very fragile approach. He estimated that the maximum rate of flow "is only half of what a dumber scheme can tolerate," and at high rates of traffic the map would saturate. In summary we have a fragile technique for providing reasonable traffic flow in three of four directions; we would like to provide for greater robustness and to allow for the fourth direction to move more efficiently.

Day 17 — May 29

And on the Seventeenth Day, The God Programs were Tested.

[20] JDU noted that in some European countries, yellow follows red, although probably not for the reasons that we are considering!

Day 18 — May 31

ET suggested using queuing theory to determine the number of cars waiting after a given light remained red for some period of time. If traffic in one direction is sparse, we can keep the lights red in that direction until a backlog of cars has built up, then let them all through at once. Doing this would help obtain "full utilization of the streets." JDU observed that the appearance of new cars at the edges of the map is a Poisson process, i.e. the probability of a new car is independent of past history.

Another proposed approach is to give priority to the direction of traffic that has the largest number of cars waiting. One way of giving priority, other than simply switching the lights to green, is to let the duration of the green light be longer than the red light; up to now we have been implicitly assuming that the lights stay green for the same length of time that they stay red. MPF suggested that, to avoid edge effects, the simulator should have the map "wrap-around," so that a car driving off the edge of the map would appear on the opposite edge.

The notion of the "phase angle" between adjacent traffic lights on a given street was considered again. If the traffic light controller behaves cyclically, we can represent the state of the light at 2nd St. and 3rd Ave. as sort of a pie chart with red and green time slices. For optimum traffic flow, the corresponding pie chart at 3rd St. and 3rd Ave. should be the same, modulo a phase angle rotation to account for the delay in driving one block along 3rd Ave. MT observed that, in case the phase angle was exactly 180°, traffic flow would be optimum in both directions. The same is true of a phase angle of exactly 360°, but it is not true of any other phase angle, assuming that there are only two "slices" (one red, one green) in the pie. MT suggested finding a differentiable expression for throughput, using phase angles for variables. JDU cautioned that this would probably be very hard to do, but encouraged anybody to try.

The discussion turned to handling very sparse traffic flow, where we assume that every car should be able to move across the map virtually unimpeded. In this case, the controller can try to turn the lights green predictively, so that the cars never need to stop. As traffic flow increases, it may be necessary to "reserve" green lights for each car. Then the problem reduces to resolving scheduling conflicts, i.e. when a car needs to travel north-south through an intersection that has already been reserved for east-west travel. This is exactly the problem of managing shared resources, and opens the possibility of optimization by pre-empting reservations. As an example, consider a car driving along 4th St. which has reservations at 4th and 5th Ave. If there should appear cars on 4th and 5th Ave. which need to cross the 4th St. intersections, it may be better to pre-emp t the original reservations. By doing so, only the car on 4th St. will be delayed.

MK suggested that traffic flow should be "pulled" by the traffic density at each intersection rather than "pushed" by the cars that appear on the map. Each traffic light would broadcast openings in its schedule to neighboring traffic lights, in effect requesting that cars be sent to fill

these openings. JDU seemed intrigued by this idea, commenting, "I think this would work, but I find it very unintuitive." He also noted that, in this scheme, "there would have to be somebody at the edges requesting cars." MK claimed that this would be an example of "Just In Time (JIT)" scheduling. PKW suggested looking for sequences of available time slots.

MT described a geometric representation that can be used for planning. Consider a graph, with time on both axes. Let the horizontal axis represent the starting time of some event, and let the vertical axis represent the ending time. Each event can then be represented as a point; presumably, all the points must lie above the $x = y$ line. For the traffic lights problem, there would be a graph of this sort at every intersection. Each car that needs to pass through the intersection will be represented by a point. There will be two types of points, one for cars traveling vertically and one for cars traveling horizontally. The problem is to adjust the points so that no two points of different types conflict, where we consider two points to be in conflict if the triangles obtained by projecting each point to the $x = y$ line overlap.

Day 19 — June 5

BJG complained that the traffic lights simulator ran too slowly, making it difficult to test different light controller algorithms. The simulator runs in approximately real-time on a 5 by 5 street grid with heavy traffic; since it seems necessary to simulate at least an hour of traffic flow to minimize the effects of initialization and termination transients, it takes about an hour to test even minor adjustments to a controller algorithm.

We considered two techniques for dealing with this problem. First, it may be possible to project long-term traffic flow performance from a few early observations of the traffic flow. JDU suggested that the graph of one of our principal criteria, the average delay experienced by a car crossing the grid, versus the duration of the simulation, might be hyperbolic. He predicted that the average delay would be linear for short simulations, and then would approach some "steady-state" delay as the length of simulation increased. Theoretically, it should be possible to determine the hyperbola, and hence the steady-state delay, by sampling the average delay in the early stages of the simulation. Several people objected, arguing that there would be too much noise in the samples to hope for an accurate projection of the steady-state delay. In particular' the data will vary greatly if the samples are taken during different phases of the controller cycle.

A second technique is to discard data that is obviously biased as a result of the finite length simulation. For instance, any car that passes through the grid in the first minute will probably have a lower delay on average, since the grid will not have reached its steady-state traffic density. Also, at the end of the simulation, any cars that have been on the map for only a few seconds

will probably skew the output results, so it seems reasonable to discard their statistics.

Suppose we are interested in the average length of time that it takes a car to cross the grid. For simplicity, we disregard the initialization transient and consider only the effects of the termination transient. Let n be the number of cars that have crossed the entire grid, and let t be the average length of time required for the crossing. Let n_f and t_f be the corresponding statistics for the cars still on the grid at the end of the simulation. JDU suggested that $t_f = \frac{t}{2}$ seems intuitively reasonable. Then the average time to cross the grid is $\frac{nt+nj\frac{t}{2}}{n+n_f} = t\ \frac{n_f t}{2(n+n_f)} = t(1 - \frac{n_f}{2(n_f+n)})$. If we can stop the simulation when $n = n_f$, i.e. the number of cars that have left the grid equals the number of cars still on the grid, then $t_{actual} = \frac{4}{3}t$.

MT presented an analysis of vertical traffic flow when the light controller is optimizing for horizontal traffic. Let s be the duration of the green light phase, k be the number of seconds required to traverse a single block, and d be the phase delay between adjacent traffic lights. Then $f_{down} = |[(2s-(k-d))\%2s]-s|$, and $f_{up} = |[(2s-(k+d))\%2s]-s|$. The goal is then to maximize $f_{down} + f_{up}$. In the example considered previously, $k = \frac{3}{2}s$. The proposed delay $d = \frac{3}{2}s$ yields f_{down} optimal and $f_{up} = 0$, i.e. stopping at every light, as we predicted. As another example, if $k = \frac{2}{3}s$, then $f_{up} = \frac{s}{3}$, i.e. stopping at every third light.

JDU ended class with a quote from R.W. Floyd: "Research is looking for your dime under the lamppost because that's where the light is."

Day 20 — June 7

The last day of class opened with a somewhat hectic discussion of whether constant deceleration was an accurate assumption. If so, then stopping distance should be proportional to the square of the velocity. PKW argued that deceleration should not be constant, and JDU provided a supporting rationale; specifically, that the limiting factor in braking is the rate of energy dissipation. Since there is presumably some maximum rate of energy dissipation and kinetic energy is proportional to the square of the velocity, the maximum possible deceleration should decrease as velocity increases. Consequently, stopping distance increases faster than the square of the velocity. HH pointed out that, in light of this result, there is a threshold at which increasing the speed limit will result in reduced throughput, since the lead car will be forced to travel so far behind the theoretical crest of the wave.

JDU showed a histogram, obtained by PKW and BJG, displaying the traffic flow as a function of the length of the green light cycle. "Here is an example of lying with statistics," he joked,

since the histogram accentuated the variation in traffic flow by cutting off the graph well above the zero point. A surprising result that PKW and BJG obtained was that using random phase angles, even with otherwise identical green-yellow-red cycles at every light, would lead invariably to deadlock. This result held independent of the duration of the green-yellow-red cycle.

The class toyed with the idea of randomly "blessing" certain cars. A blessed car would be given priority at every intersection, and presumably the cars near to it would benefit as well.

ET wondered whether, assuming optimization of horizontal traffic flow, it was correct to use a greedy technique to optimize vertical traffic flow. In other words, let the phase angle at the intersection of the first street be zero, and optimize the phase angle at the intersection of the second street with respect to the first. Then optimize the phase angle at the third intersection with respect to the second, and so on. Nobody was quite sure, but JDU suspected that, "As usual with greedy algorithms, the default answer is 'no.'"

BJG commented that the bumpers were not of much use, since most of the bumpers were being triggered at each cycle. Also, since the bumpers were only five yards back from the intersection, it was not possible to turn the crossing light from green to red before the car reached the intersection. AW briefly described the approach that he and DS had taken, which involved forwarding messages from one light to another warning that a car was approaching. He noted that they were able to reduce average delays, but that the throughput was roughly halved. Thus, cars would presumably back-up all around the edges of the map.[21] ET suggested that by keeping all the incoming lights red on three sides of the grid, we could really reduce the average delay. This, however, is not necessarily true, since the cars kept waiting at the three blocked sides of the grid would increase the average delay. PKW suggested measuring the average delay only of cars that successfully crossed the grid, leading JDU to remark, "So what you guys want to change isn't the controller mechanism, but the way of measuring performance. Don't tell me you haven't learned anything in this class!"

[21] "That's Mountain View's problem," JDU commented graciously. On an entirely unrelated note, JDU also provided the New York definition of a μs as "the time between the light turning green and the first person honking."

Chapter 9

Conclusions — JDU

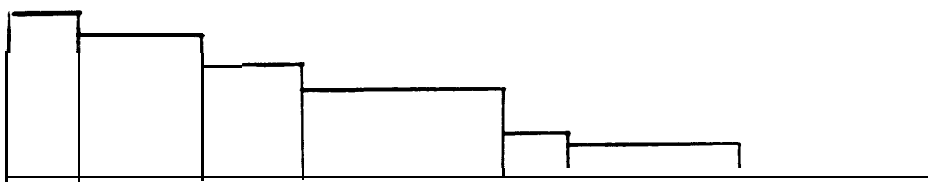
RETROSPECTIVE ON SOLUTIONS OFFERED BY THE 304 CLASS

Problem 1: Abundant Numbers

I started out with little idea how far people could go on this problem; one of my first comments was that students should think about $\alpha_{1,000,000}$. We quickly realized that α_n grows doubly exponentially in n, and that, empirically, α_n has about $(1.7)^n$ digits.

It turned out that one of the teams did far better than the others, because they discovered a way of narrowing the search that was radically different from the others, who were concerned with upper and lower limits on the number of factors for the various primes. The key idea is to think of prime factors, distinguishing different factors of the same prime. Thus, 120 is composed of "the first 2," "the second 2," "the third 2," "the first 3," and "the first 5." Each factor, say "the ith p," can be represented by a rectangle, whose width is $\log p$. The height of the rectangle is chosen so the area of the rectangle is the logarithm of the "abundance hit," which is the factor by which the abundance increases when we multiply by p a number that already has exactly $i - 1$ factors of p. The abundance hit, we easily calculated, is $1 + 1/(p + p^2 + \ldots + p)$, so its logarithm is *approximately* p^{-i}.

We can now order factors by height of their rectangles, and pick the highest ones to form a line, as suggested by



We stop as soon as the area of the rectangles reaches the logarithm of our target abundance. The width of the line is the logarithm of the number represented. Typically, we overshoot the target abundance somewhat. However, we can get a better number by replacing the last rectangle by extra powers of 2 and/or other small primes. Since these tend to be very narrow rectangles, they add less to the total area, often giving us a smaller number that

still has the target abundance. A certain amount of backtracking is necessary, in which we consider replacing several rectangles by lower rectangles that happen to get us closer to the target abundance than our "greedy" guess. However, there are usually very few changes than can be made, since all replacing rectangles will be lower than the rectangles they replace, and if we do too much replacement, we can't reach the target area (abundance) without widening the line beyond the point of our current best guess.

In practice, the team that developed this algorithm found that from a few to a few hundred improvements to the guess could be made before it was possible to prove that no further changes could get us a smaller number with a sufficiently high abundance. The limiting speed factor was how fast `/usr/games/primes` generated the primes. They were able to compute α_{31} in about 14 minutes. Beyond α_{31}, the errors inherent in the double-precision computation of logarithms and their sums was sufficiently great that it was unclear whether numbers had abundances slightly below or slightly above the target.

Problem *2:* **Tiling Halls**

My initial idea about how to solve this problem proved wrong. In fact, neither "closed semirings" nor anything like it lead to the best result. Rather, one team's careful analysis and development of the necessary theory led to the most effective solution. First, it turns out that the number of states in the automaton built from a set of tiles has a great deal to do with how quickly the instance can be solved (but we suspected that). The maximum number of states can be reduced from almost 2^{16} to 2^{12}, by filling in an entire row at a time, perhaps using a combination of tiles. Thus, all states correspond to halls with at most three rows that are partially filled or not filled at all. Note that when making a transition between states, we may "consume" from the input more than one row, but that could happen even if transitions were all based on placing a single tile.

The key theorem proved by the "winning" team is as follows. Let N be the number of states in the automaton produced; note N $\leq$ 4096. Let G be the greatest common divisor of all the numbers n such that n is the length of a tilable hall, and $n \leq$ 2N + 1. Then there is a constant *T,* which is no greater than $4N^2$ + 6N + 2, such that for all $n \geq T$, n is tilable if and only if n is a multiple of G.

That result says we only have to simulate the automaton for halls up to *T*. However, if N is large, *T* could be as large as 16 million, in principle. The actual algorithm calculates *T* "on the fly," so if the actual value of *T* is much less than N^2, we do not have to simulate out beyond *T*. Moreover, unless the target hall length turns out to be between 2N + 1 and *T,* we never have to simulate beyond 2N + 1.

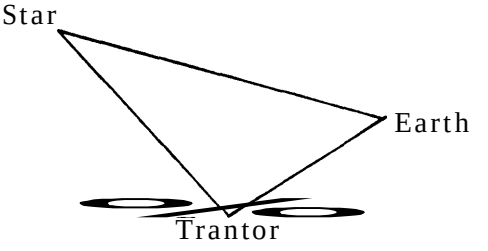
The idea is to simulate the automaton on halls of length $1, 2, \ldots$, and keep an estimate g of G and an estimate t of *T*. That is, g is the gcd of all the tilable hall lengths found so far, and t is the smallest multiple of g such that it and all higher multiples of g can be expressed as a sum of tilable hall lengths found so far. As we simulate the automaton, we call a hall length n "tilable" if n inputs can take the automaton from the initial state back to the initial state (which corresponds to three empty rows).

As soon as we find a tilable hall length, we initialize both g and t to that length. Now suppose that m is the next tilable hall length found. Let $g' = gcd(g,$ m) and let $t' = t + gm/g' - m$. Then it can be shown that g' and t' are appropriate new values of

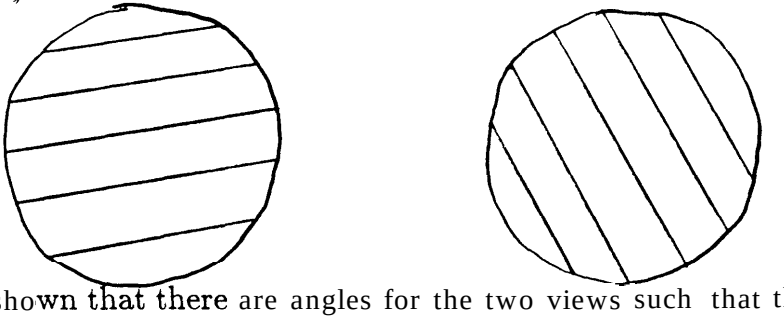
g and t. We repeat this process for each hall length up to $2N$ + 1, and the final values of t and g become T and G, respectively. If the target hall length l is greater than T, we have only to ask whether G divides 1. If $l \leq$ 2iV + 1, we already know the answer, and if $2N + 1 < l < T$, we need to simulate further, up to l.

Problem 3: Constructing Star Views

There were two interesting ideas developed here. The first dealt with the basic problem of matching stars whose two views were jittered. From the Trantor point of view, we can pass a plane through Trantor, Earth (whose location we know), and any given star as seen from Trantor. The picture below suggests this plane for one star. We may then order stars by the angle that their plane makes with the zy-plane. Similarly, we may order stars the same way from the Earth point of view. We then match star images in the two orders, assuming the stars with the smallest angles from each viewpoint match, the next smallest match, and so on. While we do not generally get the right answer (there were about 900 wrong out of 10,000, when an error of one part in 10^{-5} was used) we miss only those where "best" matches turn out to be deceptive. It can be shown that we never match stars that are too far away to be explained by the limited jitter allowed. And of course, the algorithm takes only 0(n log n) time on n stars, primarily to sort the lists of angles.



The second interesting idea involved a problem I hadn't exactly posed. It assumed views that had no jitter, but the second vantage point was unknown. The unexpected assumption was that the views were not perspective views, but orthographic projections. That is, the two vantage points were at infinity. The idea, suggested in the figure below, is to divide the image into bands (about 20 was suggested as sufficient for 100 stars). The number of stars in each band was computed, and the bands were rotated around the centers of the circles. Each time a star moves between two bands, we recalculate the numbers of stars in each band. The number of times that happens is the product of the numbers of stars and bands, so the calculation is fast.



It can be shown that there are angles for the two views such that the bands "line up," and the numbers of stars in each band will be the same for each view. From the angles

of rotation for the two views, the relative angle of the views can be computed. The only glitch is that there may, by chance, be a second pair of rotational angles such that the populations of the bands agree, yet these angles falsely indicate a solution. By taking the number of bands high enough, we reduce the probability of a "false drop" essentially to zero.

Problem 4: HiQ

We learned a great deal about the theory, but no one produced a program that made significant inroads into the problem. Sigh.

Problem 5: God

There were two solutions that performed better than the others. The best program took very much time to run; we got its results the day after the competition. It operated by a "generate and test" strategy. It had a built-in vocabulary of primitives, e.g. "top card is red." Rules were represented by Lisp s-expressions involving the primitives and logical connectives, and at each card, it would generate several thousand new expressions and check them for validity with past experience.

The second approach was statistical; it did not perform quite as well as the first, but was very fast. It had a large vocabulary of features, which it represented by characteristic vectors, e.g., the set of positions in the stack where there is a red card. The program searched for correlations between these characteristic vectors, perhaps shifted. For example, it might find that 2/3 of the time, when position i has a red card, position i + 2 has a black card. That might be the best predictor it could find of what to do when the card below the top of stack was red.

It is interesting that these programs, and the others as well, beat a random player about 75% of the time, even though the test suite consisted almost exclusively of gods that involved concepts not present in the programs. For example, one god was based on the *purity* of the rank, that is, the number of 1's in the binary representation. The rule was that the parity had to alternate even/odd. However, there is a 2/3 probability that if you change the rank by 1, e.g., play a Jack after a 10, you switch the parity of the rank. Thus, several players did surprisingly well on this rule, using the notion of odd-even, but without a notion of "parity-of-rank."

ON THE SOLUTIONS OFFERED BY THE 204 CLASS

Problem 1: Cryptograms

It turns out that my theory of how to solve cryptograms, using joins and semijoins, was not bad. However, the team with the best results used a depth-first search approach, in which they would bind the cyphertext letters for a single word to each possible plaintext word in turn, and recursively solve the partially restricted problem.

Problem 2: Abundant Numbers

Although I had seen the 304 students solve this one before, I tried very hard not to give away the best idea I knew. I was therefore quite pleased when one of the 204 teams

developed a technique, involving line slopes, that is probably identical — deep down — to the "rectangles" approach that one of the 304 teams invented.

Problem 3: God

Again, I tried not to tell what I knew, which was easier because I didn't know that much about the right approach. I was interested to discover how badly the "genetic algorithm" did on this problem, since I would have expected God to fall within the class of problems for which that approach is suitable.

The winner in the 204 group was a program with a rather simple idea. The team implemented a little programming language in which they could describe families of similar patterns very succinctly. They implemented an interpreter that considered each of the patterns described by one of the statements of their language, and then selected the next play by following the rule that best matched previous plays. By writing a program describing about 50,000 patterns, they found they got their best performance. Fewer made it too likely that the program would have nothing like the true pattern, while more caused the true pattern sometimes to be obscured by other patterns that matched the history but did not predict well. It is hard to tell whether this number of patterns would do well if we changed our assumptions about what a reasonable pattern was, and either used only very simple patterns or weirder patterns. However, the concept seems to give a lot of power for very little mechanism.

When played against the best of the 304 programs, the best 204 program was the winner, although not by the margin it achieved over the other 204 programs.

Problem 4: Traffic Lights

Unfortunately, much of the class did not get to the point of coding their solutions, I suspect because undergrads worry about finals more than grad students do. However, the class did do some interesting "theory" of traffic lights. The most interesting result is that, on Manhattan-size streets, a pattern in which traffic lights oscillate at a 10-20 second interval can provide much more throughput than obvious schemes. One day, if I have nothing to do, I'm going to devote myself to getting some rationality into the way we deal with traffic. I'm convinced that certain inventions: traffic barriers, four-way stop signs, "smart" (bumper-controlled) lights, and others will be regarded by our descendants the way we now regard blood-letting as a form of medicine.