December 1983

Report No. STAN-B-83-990

A Programming and Problem-Solving Seminar

by

John D. Hobby and Donald E. Knuth

Department of Computer Science

Stanford University
Stanford, CA 94305

Computer Science Department

A PROGRAMMING AND PROBLEM-SOLVING SEMINAR

by

John D. Hobby and Donald E. Knuth

This report contains edited transcripts of the discussions held in Stanford's course CS204, Problem Seminar, during autumn quarter 1982. Since the topics span a large range of ideas in computer science, and since most of the important research paradigms and programming paradigms were touched on during the discussions, these notes may be of interest to graduate students of computer science at other universities, as well as to their professors and to professional people in the "real world."

The present report is the fifth 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).

The production of this report was partially supported by National Science Foundation grant MCS-8300984.

Table of Contents

Index to problems in the five seminar transcripts

Cast of characters

		Professor
DEK	Donald E. Knuth	
		Teaching Assistant
JDH	John D. Hobby	
		Students
RB	Richard Beige1	
EJB	Eric J. Berglund	
WRB	William R. Burley	
SRE	Sharon R. Eisenstadt	
CGG	Cary G. Gray	
MAH	Mary A. Holstege	
EJK	Eric J. Krohn	
NMP	Nancy M. Paulikas	
PKR	Peter K. Rathmann	
SJR	Stuart J. Russell	
DS	Devika Subramanian	
scs	Steven C. Suhr	
WDW	William D. Wilson	
		Auditors
CMA	Clay M. Andres	
PKG	Pijush K. Ghosh	
KAH	Keith A. Hall	
PDK	Peter D. Karp	
		Visit or
VRP	Vaughan R. Pratt	

Notes for Thursday, September 30

At the first class meeting, DEK handed out all five problem descriptions, and made some introductory remarks about the purpose of the course and the work expected from the class. He stressed that the class will work *together* on the problems rather than in an atmosphere of competition. None of the problems have any published solutions. DEK mentioned that he feels problem three may be particularly good. Since he has -already solved this problem he will have to try to avoid forcing the solution on the class.

We are interested in problem solving methods as well as the solutions themselves. Reports should be handed in including a description of methods used and the promising approaches that failed. The purpose of this is to learn about methods used to solve problems in all areas of computer science research.

DEK had brought along some course evaluation forms from last year's CS 204 class. Comments by the students in that class stressed the heavy workload, and generally indicated that the course was a useful way to learn problem-solving skills, though not in a formal way.

Problem 1, due October 12: **Bulgarian Solitaire.**

Here's a pastime you can play with 15 cards. (And you can save money by using old punched cards that computer centers are throwing away, since the cards don't have to have any distinguishing marks.)

Deal the cards into any number of piles, with any number of cards in the individual piles. Then form a new pile by taking exactly one card from each of the existing piles. **And** keep forming new piles in this way until you get tired of the game.

Yes, it might seem silly. But the interesting thing is that, sooner or later, you are bound to reach a situation in which there are exactly five piles, containing exactly **1, 2,** 3, 4, and 5 cards, respectively. And this configuration repeats itself, so you might **as** well stop when you get to this point. Furthermore it turns out that you need at most 20 steps to reach the stopping configuration.

In general if you start with $\frac{1}{2}n(n+1)$ cards, you will always reach the configuration $\{1, 2, \ldots, n\}$ in a finite number of steps. Furthermore it has been conjectured that at most $n(n-1)$ steps are needed; this conjecture has been verified for $n \leq 5$.

The purpose of this problem is to prove or disprove the conjecture for $n \leq 10$, and to discover further information about the number of steps. Write a computer program that determines the number of initial distributions of $\frac{1}{2}n(n+1)$ cards that are at distance k from the stopping configuration, for all k, and for all $n \leq 10$.

Incidentally, when n = 10 there are 451,276 essentially different ways to deal the 55 cards, and the conjecture says that all of these will lead to the distribution $\{1, 2, \ldots, 10\}$ after at most 90 steps. Try to use time and space efficiently in your program.

Notes for Tuesday, October 5

DEK started class by asking if anyone had made any progress on the problem. PKG said we could start at $\{1, 2, \ldots, n\}$ and go backward. The predecessors form a tree, which branches where there are different ways to go back from a particular configuration; the branches can never join again. The configurations farthest from the root have no predecessors. For instance, no backward step is possible from the configuration $\{1, 1, 2, 3, \ldots, n-2, 2(n-2)\}$. This configuration seems to be a long way from $\{1, 2, \ldots, n\}$.

DEK said that this raised two points the class should investigate: Is it better to do the analysis backward starting from $\{1, 2, \ldots, n\}$? What happens in the configuration $\{1, 1, 2, 3, \ldots, n-2, 2(n-2)\}$? He added that we should also prove that any starting position eventually leads to the configuration $\{1, 2, \ldots, n\}$.

DEK asked PDK what his reaction to the problem was. PDK suggested just trying all the ways to deal the cards and playing the game from each position to see what happens.

This led to a discussion of how to enumerate the possible starting positions. PKR suggested starting from $\{55\}$ and successively taking cards out of the first pile and putting them in the second until the second pile is longer than the first. For example, we could start out $\{55\}, \{54, 1\}, \{53, 2\}, \ldots, \{28, 27\}$. DEK was afraid it would not be clear how to continue from here. EJB suggested taking all the two pile configurations and then those

with three piles, etc. DEK wanted to see how this would work on an arbitrary example.

$$\{10,8,8,7,7,7,3,2,2,1\}$$
$$\{9,9,8,7,7,7,3,2,2,1\}$$

The lower configuration clearly follows the upper one but the next step is not clear. RB suggested that we might next try $\{8,10,\ldots\}$ and seeing that that does not work, we would try taking from the second pile. The class quickly decided that one would be bound to miss some configurations this way.

SRE suggested a different approach: First try all positions of the form (54,. . .} then (53,. . .}, etc. DS said we could do this recursively and RB pointed out that the routine would need two parameters. One parameter would give the number of cards left to place and the other parameter would be a limit on the size of each pile. Suppose there are n cards altogether and each pile should contain at most m of them. DEK and KAH decided that a recursive function could be defined

$$\{k\} \cdot p_k(n-k),$$
$$\{k-1\} \cdot p_{k-1}(n+1-k),$$

$$\{1\} \cdot p_1(n-1)$$

where $k = \min(m,n)$. Here $p_j(n-j)$ is an ordered list of lists and $\{j\} \cdot p_j(n-j)$ means add a pile of j cards to the front of each list in $p_j(n-j)$.

MAH mentioned that it might take a lot of time and space to implement this recursive algorithm. DEK responded that it is possible to calculate the following pattern directly from the preceding one. Consider the previous example.

$$\{10,8,8,7,7,7,3,2,2,1\}$$
$$\{10,8,8,7,7,7,3,2,1,1,1\}$$
$$\{10,8,8,7,7,7,3,1,1,1,1,1\}$$
$$\{10,8,8,7,7,7,2,2,2,2\}$$

Scan from right to left and decrease the first possible pile, then pick up all the cards in the piles passed over and redistribute them into piles as large as possible.

WDW suggested that it may be just as fast to start from $\{1,2,\ldots,n\}$ and go backward as PKG suggested. DEK asked why it is best to consider the configurations in this order and SJR pointed out that this way we never have to consider a pattern more than once. If we start from each position and play the game until we reach $\{1,2,\ldots,n\}$ we would consider each position many times. It would take too much memory to remember which positions had already been played out. DEK opined that this is an important concept.

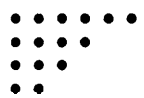
The discussion turned to how to implement this backward search. DEK asked how many predecessors the configuration $\{10,8,8,7,7,7,3,2,2,1\}$ has. Most people agreed that it has one immediate predecessor. This is correct since if 10 is the new pile the previous configuration must have been $\{9,9,8,8,8,4,3,3,2,1\}$; and 8 couldn't have been the new

pile because there must have been more piles than that. Going back one more step, we get to the unique position $\{10,9,9,9,5,4,4,3,2\}$.

WDW suggested that the positions could be stored in an array during the search but only one position really needs to be stored if we make forward steps as well as backward ones. DEK explained that a depth first search could be executed by first finding all the immediate predecessors of the starting position and searching all predecessors from one branch before going on to the next. The array would hold all the positions between the current one and $\{1, 2, \ldots, n\}$ along with the branch taken at each point. As WDW pointed out, all this can be generated from just the current position.

An alternative strategy, as DS suggested, would be a breadth first search trying all patterns that are one step from $\{1, 2, \ldots, n\}$, then all those which are two steps away, etc. However, this would take more memory than a depth first search and is not needed for this problem.

SJR suggested keeping track of the number of piles with at least one card, the number with at least two, and so forth. This would make it very easy to take one card from each pile. This brought up the idea of dual partitions. DEK explained that SJR's scheme is equivalent to keeping track of the dual of the original partition. If there are N cards all together, each configuration of piles can be though of as a partition of N. If we think of the piles as horizontal, the diagram below represents the partition $15 = 6 + 4 + 3 + 2$.



If we think of the piles as vertical, however, the partition becomes $15 = 4 + 4 + 3 + 2 + 1 + 1$. This is the dual of the original partition and it demonstrates the one to one correspondence between partitions and their duals.

DEK suggested seeing what happens to the dual partition during a forward step. Consider the previous example.

Original	Dual
$\{10,9,9,9,5,4,4,3,2\}$	$\{9,9,8,7,5,4,4,4,1\}$
$\{9,9,8,8,8,4,3,3,2,1\}$	$\{10,9,8,6,5,5,5,2\}$

In the dual representation the operation looks exactly the same except it goes in reverse. What is a forward step in the original representation is a backward step in the dual. This reversal was not noticed in class and it was erroneously concluded that a partition and its dual are the same number of steps from $\{1, 2, \ldots, n\}$.

SJR suggested that we could store for each j the number of piles with exactly j cards. Suppose the original representation is $\{a_1, a_2, \ldots, a_k\}$ where $a_1 \geq a_2 \ldots \geq a_k$ and $a_1 + a_2 + \bullet$ ♦ & ■ N. The new representation is then $(b_1, b_2, \ldots)$ where $b_1 + 2b_2 + 3b_3 + \cdots =$ N. In this representation a forward step becomes a left shift by one (discarding b_1), except that if the new pile has k cards then b_k must be incremented.

$\{a_1, \ldots, a_k\}$	$(b_1, b_2, \ldots)$
$\{10,9,9,9,5,4,4,3,2\}$	$(0,1,1,2,1,0,0,0,3,1,0,\ldots)$
$\{9,9,8,8,8,4,3,3,2,1\}$	$(1,1,2,1,0,0,0,3,2,0,\ldots)$

DEK presented the following algorithm for a backward step in this representation. Assume the data structure includes a variable s = $b_1 + b_2 + \ldots$.

Choose $k \geq s - 1$ such that $b_k > 0$;
$b_k \leftarrow b_k - 1$;
$(b_1, b_2, b_3, \ldots) \leftarrow (k - s + 1, b_1, b_2, \ldots)$;
$s \leftarrow k$.

Notes **for Thursday, October 7**

DEK started class with a remark on **asymptotics. As** a function of n, the number of configurations to consider is the number of partitions of $\frac{1}{2}n(n+1)$. This grows as *A"***for** some *A* approximately equal to five. Thus a program is likely to take about five times as long to verify the conjecture for n + 1 as it would for n. This severely limits how many small cases can be tested by computer.

Last Tuesday it was stated that a step in the game looks the same in the dual representation as it does in the original. RB and others had noticed that this is not true. RB presented a counter-example. The partition

becomes

when the piles are viewed horizontally, but

when the piles are viewed vertically.

Before class started, DEK had drawn a diagram of the entire tree resulting from taking backward steps from the initial configuration $\{1,2,3,4,5\}$ for $n = 5$. It was drawn in such a way that the left-most branch always corresponded to the largest pile being the new pile. The left-most path from the root was one of the longest. DEK showed how the number of piles in configurations along this path followed the regular pattern 655545655455654555645. He mentioned that this pattern extends to arbitrary *n* and the **configuration at the end of the path is** $\{1,1,2,3,\ldots,n-2,n-1,n-1\}$.

DEK asked if anyone had been able to prove that the game eventually results in the configuration $\{1,2,\ldots,n\}$. PKG responded that he thought he had an idea. Since there are a finite number of possible configurations the game must eventually lead to a loop.

This brought up the question of how to make sure all configurations lead to the right loop. The discussion digressed somewhat here but DEK explained what happens when an arbitrary function is used to define a sequence of elements from a finite set. The sequence is $x_1, x_2, \ldots$ where $x_{j+1} = f(x_j)$. In general at some step l the sequence enters a loop so that $x_l = x_{l+p}$ and for any j > 1, $x_{j+p} = x_j$. He went on to discuss methods for finding such cycles.

Since no one had been able to prove that the game always leads to the configuration $\{1, 2, \ldots, n\}$, DEK presented a proof of this, due to Jørgen Brandt of Aarhus University. It suffices to show that there are no other cycles besides the trivial one containing only this configuration. If there are no other cycles then the game must surely lead to this one. Conversely, if there is another cycle then configurations in that cycle never lead to $\{1, 2, \ldots, n\}$.

The proof starts by defining a function $\overline{x}$ for counting piles.

$$\overline{x} = \begin{cases} 0, & \text{if } x \leq 0; \\ 1, & \text{if } x > 0. \end{cases}$$

Suppose we are given some initial configuration $\{a_1, a_2, \ldots, a_k\}$. The number of piles initially is $\overline{a_1} + \overline{a_2} + \ldots + \overline{a_k}$. This even works if some of the u's are zero or negative. The above function says not to count them in such cases. Let's see what happens to S_i, the number of piles at step i.

$$S_1 = \overline{a_1} + \overline{a_2} + \cdots + \overline{a_k}$$

$$S_2 = \overline{a_1 - 1} + \overline{a_2 - 1} + \cdots + \overline{a_k - 1} + \overline{S_1}$$

$$\vdots$$

$$S_{j+1} = a_1 - j + a_2 - j + \cdots + \overline{a_k - j} + \overline{S_j} + \overline{S_{j-1} - 1} + \cdots + \overline{S_1 - (j-1)}$$

Eventually, $j \geq \max(a_1, a_2, \ldots, a_k)$ so we have

$$S_{j+1} = \overline{S_j} + \overline{S_{j-1} - 1} + \overline{S_{j-2} - 2} + \cdots. \tag{1}$$

Now we will assume that the initial configuration $\{a_1, a_2, \ldots, a_k\}$ is part of some cycle and see what kind of cycle it can be. Under this assumption we can assume that (1) holds for all integers j.

-First of all, suppose the period is p. There are at most p possible values for S_j, the number of piles at step j. We will let m be the maximum such value of S_j and argue that if $S_j = m$ then $S_{j-m} = m$ and $S_{j+m} = m$. Since m is the maximum pile size, only the first m terms of (1) can possibly be non-zero. Since $S_j = m$, all m of these terms in the equation for S_j must be 1. In particular the mth term $\overline{S_{j-m} - (m-1)} = 1$ so $S_{j-m} = m$.

To show that $S_{j+m} = m$ when $S_j = m$, we make use of the periodicity. By going through m cycles we can see that $S_j = S_{j+pm} = m$. The result we have just proved now shows that $S_{j+pm} = S_{j+(p-1)m} = \cdots = S_{j+m}$.

We will show that for all i, S_i is either m or $m - 1$. This will allow us to conclude that the period p divides m. First of all, if $S_i = m$ then S_{i-1} is either m or $m - 1$.

$$S_{i-1} = \overline{S_{i-2}} \; \overline{S_{i-3}} - \cdots == + \overline{S_{i-m}} - (m-2) + \overline{S_{i-m-1}} - (m-1)$$

The first $m-1$ terms are equal to 1 since because $S_i = m$ we know that $\overline{S_{i-2}-1} = \overline{S_{i-3}-2} = \cdots = \overline{S_{i-m}-(m-1)} = 1$. The last term is either 0 or 1, so we can conclude that S_{i-1} is either m or $m-1$.

Now we will prove by induction on j that, for $j \geq 1$, if $S_i = m$ then S_{i-j} is either m or $m-1$. We have already proved the basis case. In general we know that $S_i = m$ and therefore $S_{i-m} = m$. We will show that

$$S_{i-(j+1)} > 0, S_{i-(j+2)} > 1, \ldots, S_{i-(m+j-1)} > (m-2).$$

Since $S_i = m$, we already know that

$$S_{\ldots} > 0, \ldots, S_{i-(j+1)} > j, \ldots, S_{i-m} > m-1.$$

It only remains to show that the last j-l inequalities hold. But these also hold since, by the induction hypothesis and because $S_{i-m} = m$, it follows that S_{i-m-1} through $S_{i-m-(j-1)}$ are all either m or $m-1$. Taken together, these inequalities show that the first $m-1$ terms of (1) for S_{i-j} are all 1, and thus by induction $m-1 \leq S_{i-j} \leq m$ for all positive j.

Now we know that $S_i = m$ if and only if $S_{i+m} = m$. We also know that for all i, S_i is either m or $m-1$. Because of this first fact, there could not be any i for which $S_i = m$ and $S_{i-m} = m-1$ or $S_i = m-1$ and $S_{i-m} = m$. Thus, $S_i = S_{i+m}$ for all i and m must be a multiple of the period.

We now know that the period divides m and the number of piles is always either m or $m-1$. The total number of cards is of course the sum of the sizes of all the piles. Consider an arbitrary time, step 0. There are So piles in all. On the jth previous step a pile of size S_{-j} was created and its current size is $S_{-j}-(j-1)$ (if such exists). Because S_j is always either m or $m-1$, the pile size S-j $-$ (j $-$ 1) is always non-negative for $j \leq m$ and never positive for $j > m$. The total number of cards is therefore

$$N = S_{-1} + (S_{-2}-1) + \cdots + (S_{-m}-(m-1)).$$

We can define numbers δ_j to be 0 or 1 so that $S_{-j} = m - \delta_j$ for $j \leq m$. We then have

$$N = m + (m-1) + \cdots + 1 - (\delta_1 + \delta_2 + \cdots + \delta_m) = \binom{m+1}{2} - (\delta_1 + \delta_2 + \cdots + \delta_m).$$

If the total number of cards is $N = \frac{1}{2}n(n+1)$ for some n as stated in the problem, then all the δ_j must be 0 and m = n. By the above argument, the configuration of piles is always $\{m, m-1, \ldots, 1\}$ in this case. This is therefore the only cycle possible.

In general suppose the number of cards is k less than a triangular number. That is, $N = \frac{1}{2}n(n+1) - k$. The cycle must consist of $n-k$ configurations where there are n piles and k configurations where there are $n-1$. There is one cycle for each way to place $n-k$ n's and k $n-1$'s in a circle of size n. The length of the cycle will be either n or some divisor of n depending on the placement of the numbers in the circle.

For example, suppose there are 13 cards. In the context of the previous paragraph, we have N = 13, $n = 5$, and $k = 2$. The cycle consists of three configurations with five

piles and two configurations with four. There are two different cycles: one where the four pile configurations occur consecutively, and one where they don't. It is possible to deduce exactly what the configuration of piles must be at any point in either cycle. When the previous five configurations have had 5, 5, 4, 5, and 4 piles in that order, the current configuration must be $\{4, 5-1, 4-2, 5-3, 5-4\}$ or $\{4, 4, 2, 2, 1\}$.

DEK added that it would be nice to come up with an analytical proof that at most $n(n-1)$ steps are required to reach $\{1, 2, \ldots, n\}$. The best known upper bound on this number of steps is one of $O(n^3)$ due to Ron Graham. DEK said that not much progress has been made on this yet, but he did show how it is possible to analyze what happens in configurations near a given one. Consider the configuration $\{a_0, 1+a_1, 2+a_2, 3+a_3, 3+a_4\}$ if we assume that $a_0, \ldots, a_4$ are all either 0 or 1. The next four steps result in the following configurations.

$$
\begin{array}{ccccccccccc}
a_0 & 1 & +a_1 & 2+a_2 & 3+a_3 & 3+a_4 & & & & & \\
 & a_1 & 1 & +a_2 & 2 & +a_3 & 2+a_4 & 4+a_0 & & & \\
 & & a_2 & & 1+a_3 & & 1+a_4 & 3+a_0 & 4+a_1 & & \\
 & & & & a_3 & & a_4 & 2+a_0 & 3+a_1 & 4+a_2 & \\
 & & & & & & & 1+a_0 & 2+a_1 & 3+a_2 & 3+a_3+a_4
\end{array}
$$

The last configuration is exactly the original one except the numbers $(a_0, a_1, a_2, a_3, a_4)$ become $(0, a_0, a_1, a_2, a_3 + a_4)$. If $a_4 = 1$, the next step indeed results in exactly the *original configuration.

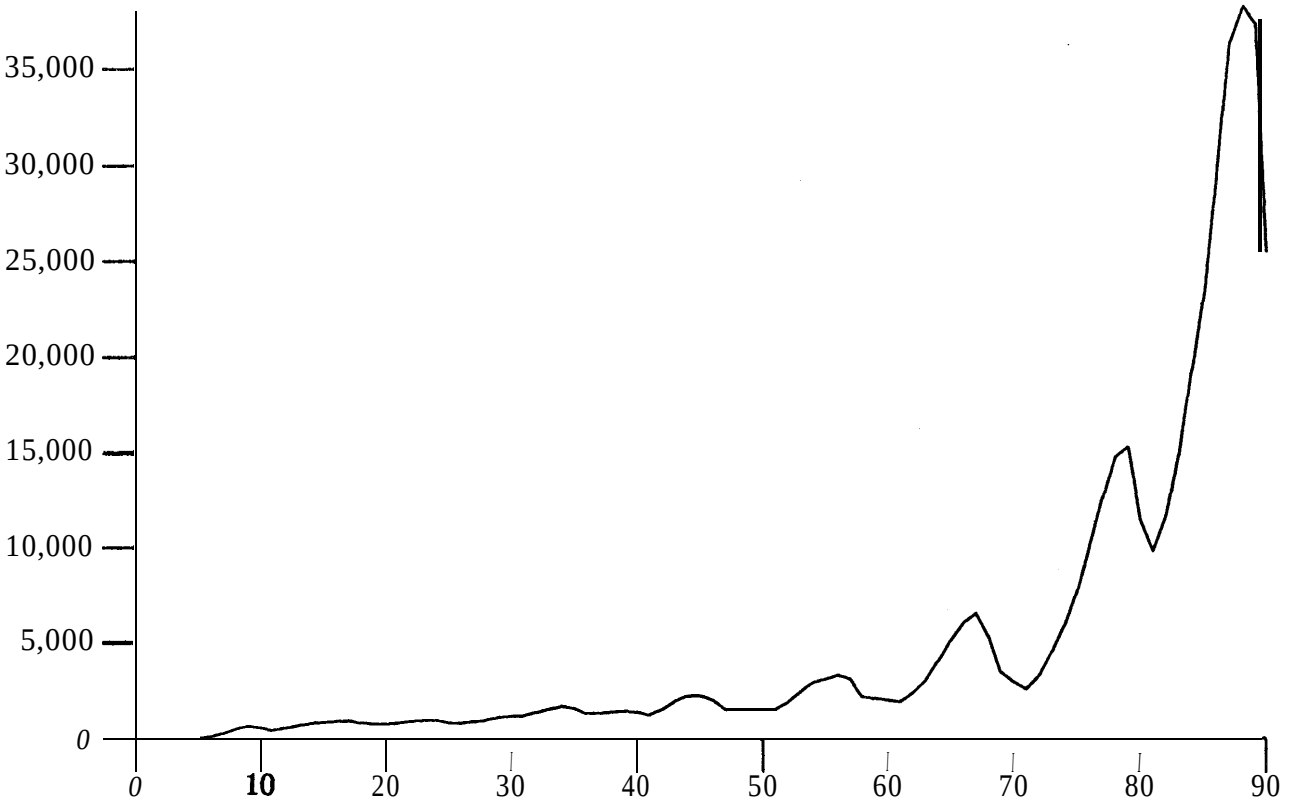
At the end of class SJR mentioned that his group had their program working **and** it had checked all cases up through $n = 11$ without finding a counter-example to the conjecture. The program took about 5 minutes for the case n = 11. He mentioned that the program consistently found that there is a distance from $\{1, 2, \ldots, n\}$ where there are relatively few configurations. This is consistent with the idea that there are many configurations at the maximum distance but there is some intermediate configuration that all of the most distant configurations must go through before getting back to $\{1, 2, \ldots, n\}$.

Solutions for problem 1

All the groups agreed that the conjecture is true for $n \leq 10$. Everybody also agreed on the number of configurations at each distance from the stopping configuration. The distribution is graphed below for $n = 10$.

Everybody used the backtracking approach that was presented in class, although most groups first considered the straight forward approach of just playing the game from each possible configuration and seeing whether they all lead to $\{1, 2, \ldots, n\}$.

All the groups represented the configurations with an array where each entry gives the number of piles having a specific number of cards. This representation appeared to be the most suitable of the three presented on the second day of class. This way, forward and backward moves involve primarily just shifting the array. Three groups, RB/SRE/SJR/SCS, WRB/PKR, and CGG/WDW used a pointer to the origin of the array so that this shifting operation could be done in constant time.



RB/SRE/SJR/SCS also used a linked list to keep track of the non-zero elements of the array. This eliminated the need to search the array when looking for a pile to break up at a backward move, although it is not clear exactly how much this saved them.

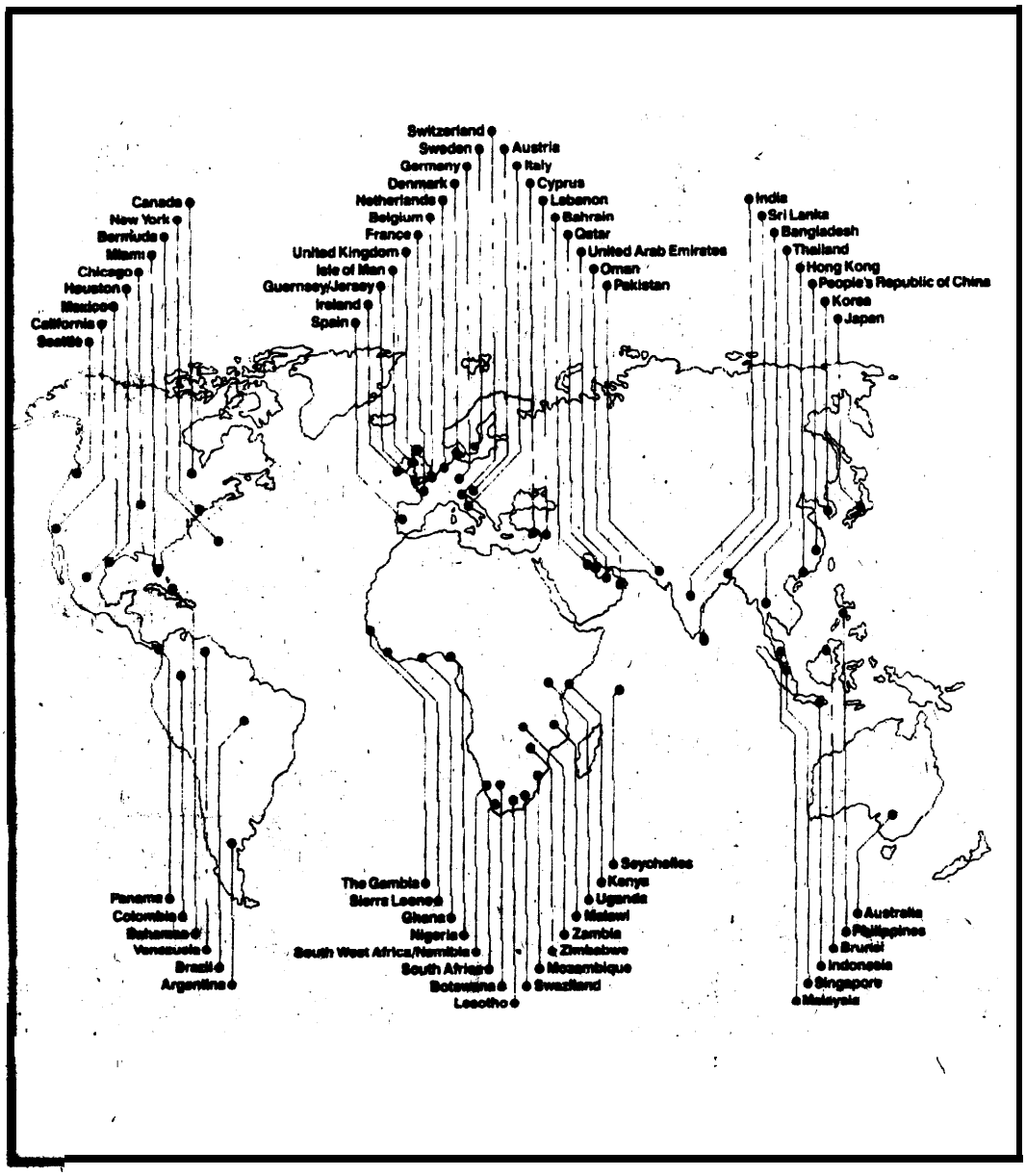
Two groups, RB/SRE, SJR/SCS and CGG/WDW used recursive routines to implement the depth first search, while the other groups did not use explicit recursion.

EJB/EJK wrote new versions of their original program to take advantage of more of the ideas raised in class. They also traced the configuration $\{1, 1, 2, 3, \ldots, n-2, n-1, n-1\}$ and verified that this always takes exactly $n(n-1)$ turns to reach the stopping configuration. It can be shown that this holds in general.

It is interesting that everyone was 30 confident of the conjecture that nobody wrote code to print a counter-example if one were found.

Problem 2, due October 26: **Layout.**

This problem was inspired by an illustration in a recent advertisement:



Given n points (xi, y_i) inside a square, develop an algorithm that connects these points to n points on some horizontal line below that square. The connecting paths should not overlap, and they should consist only of vertical and 45" segments, with at most one 45° segment per path.

Let's say that the square is $\{(x, y) \mid 0 \leq x, y \leq n + 1\}$, and that the external points are $\{(1, y_0), (2, y_0), \ldots, (n, y_0)\}$. It would be nice to find a solution that is optimum in some sense-for example, given $\delta > 0$, to find the maximum y_0 such that a solution exists with all paths at least δ apart; or, given y_0, to find the maximum possible δ; or something similar.

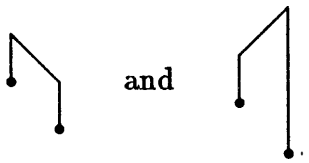
Notes for Tuesday, October 12

Class started with a discussion of the solutions to problem 1. (See the separate notes above.) DEK also suggested getting started early on problem 5 to allow more time for the competition.

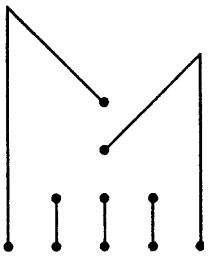
The first thing to do for problem 2 was to decide on the ground rules. DEK clarified that the problem requires choosing which points in the square should be connected to which external points. There is no need to restrict oneself to any particular mapping between internal and external points. PDK suggested also allowing the external points to be adjusted rather than requiring them to be equally spaced on a line below the square. DEK wanted to stick to the stated problem. He said that it is best to concentrate 99% of ones efforts on one particular problem while still remembering that the problem may have been chosen arbitrarily.

The problem doesn't specify exactly how optimality is to be defined. DS suggested minimizing total path length. WDW suggested that the best way to do this might be to connect the left-most internal point to (1, yo), the next to (2, y_0), and so on. In any case, the rules do not appear to leave much flexibility to optimize path length once a mapping between internal and external points is fixed.

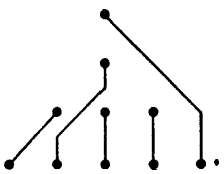
This brought up the question of whether to allow paths like



There was some doubt as to whether solutions containing such paths could be optimal under any reasonable definition. MAH suggested an example where a good solution might have to contain such paths.

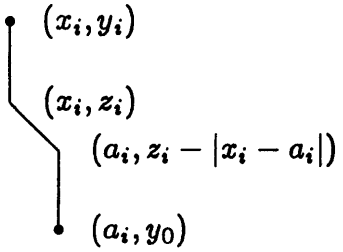


The general consensus was that such oddly shaped paths should not be allowed. Paths should be forced to always travel in the general direction of their destination. Under this restriction, one of the best solutions appears to be



PKR pointed out that the path length here is less than for the original solution MAH proposed but the minimum distance between paths is less.

DEK wanted to first examine the cases where the number of points n is very small. SJR suggested seeing what inequalities can be derived from the fact that no two paths are allowed to cross and DEK presented some notation for doing this.



Suppose we have two adjacent paths like the above, one with $i = 1$ and the other with $i = 2$. DS suggested writing equations for the paths and solving them to see if there is an intersection. DEK explained that sometimes it is necessary to attack a problem this way before getting an intuitive understanding of the situation. He came up with the following equations for the three segments of path i.

$$A: \{(x_i, y) : z_i \le y \le y_i\}$$
$$B: \{(x_i - s_i t, z_i - t) : 0 \le t \le |x_i - a_i|\}$$
$$C: \{(a_i, y) : y_0 \le y \le z_i - |x_i - a_i|\}$$

The set of points labelled A is the upper segment, the set labelled B is the diagonal segment and the set labelled C is the lowest segment. Here, s_i is the sign of $x_i - a_i$, that is $s_i = 1$ if $x_i > a_i$ and s_i = -1 if $x_i < a_i$.

Through case analysis, it is possible to derive necessary and sufficient conditions for whether the paths intersect. The first case is where the A part of path 1 intersects the A part of path 2. This case shows that for the paths to be non-intersecting, if $x_1 = x_2$ then the interval $[z_1, y_1]$ must not intersect the interval $[z_2, y_2]$. The next case is that A_1 must not intersect B_2. At the intersection point $x_1 = x_2 - s_2 t$ and $y = z_2 - t$. This implies that $t = s_2(x_2 - x_1)$ and $y = z_2 - s_2(x_2 - x_1)$. In order for this to be on both line segments $z_1 \le y \le y_1$ and $0 \le t \le |x_2 - a_2|$ must hold. Hence, if $0 \le s_2(x_2 - x_1) \le |x_2 - a_2|$ then there is an intersection unless $z_1 > z_2 - s_2(x_2 - x_1)$ or $z_2 - s_2(x_2 - x_1) > y_1$. In other words, if x_1 is between a_2 and x_2 then $z_2 - |x_2 - x_1|$ must not be in the interval $[z_1, y_1]$.

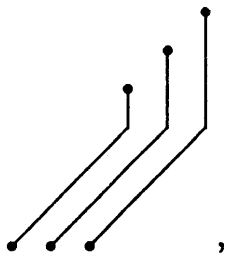
There was not time to grind out any more cases in class, but DEK explained that this could be done in principle.

Notes for Thursday, October 14

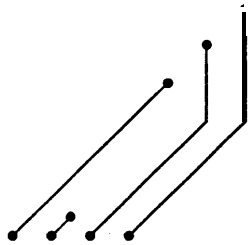
DEK brought up the question of whether the problem always has a solution. RB said yes, in fact his group had already come up with an algorithm that always finds one. The algorithm begins by connecting the right-most external point (n, y_0) to the right-most internal point. If there are any internal points to the right of (n, y_0) (that is with x > *n),* the "right-most" point is the one of these where x − y is maximum. If all internal points

have x $< n$, the one with maximal $x+y$ is taken as the right-most. In any case, if the above rules do not determine a unique point, the algorithm chooses the one with the greatest x-coordinate.

There was some concern about the apparent asymmetry of these rules, particularly in the case of ties. RB pointed out that while in the case of ties the rules usually lead to paths like



situations like



. are also possible. The point at the lower left has the largest value of x — y, but the algorithm does not choose it until the external point is to the left of it. SJR explained that the reason the algorithm works is because the first path it draws can never get in the way of any subsequent paths. To achieve this, the 45" segment of the first path should be as low as possible. That is, in the notation defined last Tuesday $z_n - |x_n - n| = y_0$.

Because of this non-interference, the algorithm can always draw the right-most path among the remaining points, leaving a simpler problem. Suppose the algorithm is at an arbitrary step and the right-most external point remaining is $(\bar{x}, y_0)$. EJB and WDW pointed out that the algorithm now chooses an internal point according to a strict right to left ordering except that points to the left of $(\bar{x}, y_0)$ are treated differently. Given two points (x_i, y_i) and (s_j, y_j), RB attempted to write down an expression for which one is farthest right in this ordering assuming no ties.

if $x_i > \bar{x}$ and $\bar{x} > x_j$ **then** i
  else if $x_i > \bar{x}$ and $x_j > \bar{x}$
    then if $x_i - y_i > x_j - y_j$ **then** i **else** j
    else if $x_j > \bar{x}$ and $\bar{x} > x_i$ **then** j
      else if $x_i + y_i > x_j + y_j$ **then** i **else** j

DEK objected that this was too complicated and EJB tried to simplify it. He said (x_i, y_i) is to the right of (x_j, y_j) when $(x_i > \bar{x} > x_j)$ or $(x_i, x_j > \bar{x}$ and $x_i - y_i > x_j - y_j)$ or $(x_i, x_j < \bar{x}$ and $x_i + y_i > x_j + y_j)$.

CGG suggested partitioning the points so that it is not necessary to worry about the position relative to $\bar{x}$. DEK had an idea for such a partition. He defined sets

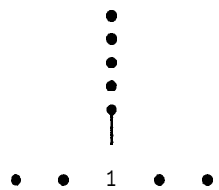
$$S_k = \{ (x_i, y_i) \mid x_i \geq k \}$$

so that if for all k, S_k has more than $n + 1 - k$ elements, then all the 45" segments can have slope $+1$. In fact, the algorithm RB presented achieves this result here. An analogous result holds for the sets

$$T_k = \{ (x_i, y_i) \mid x_i \leq k \}.$$

If $\|T_k\| \geq k$ for all k, then a dual algorithm can solve the problem with all 45" segments of slope -1. The problem can always be partitioned into pieces where one of the two conditions holds. Rename the internal points so that $x_1 \leq x_2 \leq \ldots \leq x_n$. For each i, either $x_i \leq i$ or $x_i \geq i$. Contiguous ranges where $x_i \geq i, x_{i+1} \geq i + 1, \ldots, x_j \geq$ j can be handled by the original algorithm and ranges where $x_i \leq i, x_{i+1} \leq i + 1, \ldots, x_j \leq$ j can be handled by the dual algorithm. Adjacent ranges cannot interfere because the x-coordinate varies monotonically along each path.

SCS asked what happens when many internal points are lined up above an external point. In that case, the lowest internal point must be connected directly to the external point below it.



DEK responded that one has to be careful to connect the lowest point directly in this manner. Otherwise it does not matter how the points are partitioned. In the above diagram, any two of the remaining points may be connected to the external points on the left. The other two can always be connected to the right side points.

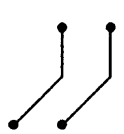
Now that two algorithms had been presented, it was clear that the points can always be connected with no crossings. DEK suggested investigating what paths should be used once it has been decided which points will be connected to which. Suppose (x_i, y_i) are to be connected to (i, y_0) for $i = 1, 2, \ldots, n$. Using last Tuesday's notation, the paths are determined by n different variables $z_1, z_2, \ldots, z_n$. It was shown that the condition that two particular paths do not intersect can be written in the form

$$(R_{11} \text{ or } R_{12} \text{ or } \ldots) \text{ and } (R_{21} \text{ or } R_{22} \text{ or } \ldots) \text{ and } \ldots$$

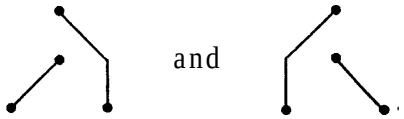
where R_{ij} are inequalities in two variables. The points can be connected as specified if and only if it is possible to simultaneously satisfy this condition for each pair of paths. DEK said that it is not too hard to check for solutions to systems of equations of this form.

This approach seemed somewhat difficult, 30 SJR suggested a more geometrical approach to the problem. For any given path and each exterior point, it is possible to define an "exclusion zone." A point is in the exclusion zone if it is not possible to connect it to the exterior point without crossing the path.

The discussion now changed to deciding which mappings between internal and external points are feasible. PDK showed that in the configuration



the connections shown are the only ones possible, while in other cases there are two possibilities, for instance



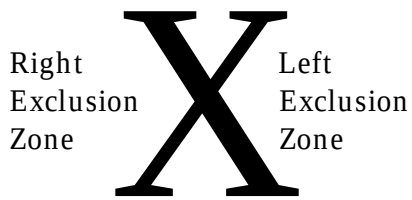
DEK brought up the problem of bipartite matchings. **A** bipartite graph is one where the nodes can be divided into two subsets in such a way that there are no edges between nodes in the same subset. The subsets can be thought of as groups of men and women. The edges would then represent possible marriages. The bipartite matching problem is to find a way to get everybody married. This problem is comparatively easy to solve because it is known that there is a matching if and only if there is no set of k men that are only connected to $k - 1$ women or vice versa.

This is possible relevant to problem 2 because the two sets of nodes could correspond to the internal and external points. The edges would then specify which connections are possible and all solutions to the layout problem would have to correspond to solutions to the matching problem.

As time was running out, DEK summarized the overall strategy: First find which mappings between internal and external points are possible and then decide which is best. Given a matching from one of the algorithms, what are the optimal connections?

Notes for Tuesday, October 19

DEK opened class by clarifying SJR's idea of an exclusion zone. An internal point (x_i, y_i) has two exclusion zones as shown below.

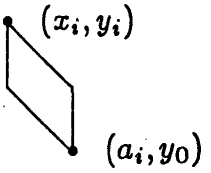


Internal points in the right exclusion zone of a point (xi, y_i) cannot be connected to external points to the right of (a_i, y_0) no matter what the path from (x_i, y_i) to (a_i, y_0) does. A similar property holds for the left exclusion zone.

DEK rephrased this idea by saying that two points (x_i, y_i) and (x_j, y_j) are visible from each other if each is in one of the other's exclusion zones. This means that if (x_i, y_i) is connected to (a_i, y_0) and (x_j, y_j) is connected to (a_j, y_0) then $a_i < a_j$ if and only if $x_i < x_j$.

SJR said there are three more exclusion zones and the class investigated some of them. If a point (x_i, y_i) is connected to an external point (a_i, y_0) with $x_i < a_i$, then no other internal point (x, y) with x < $\min(x_i, x_i + y_i - y)$ may be connected to any point to the right of (a_i, y_0). WDW pointed out that any path from (x_i, y_i) to (a_i, y_0) must lie within

a certain parallelogram.



If $a_i > x_i$ as shown, then no point above or to the right of the parallelogram can be connected to any point (a_j, y_0) with $x_i \leq a_j < a_i$.

DEK suggested trying a small example to see how these ideas can be used to help decide which points should be connected to which others. He presented the following "random" test data.

Point	Coordinates	
A	3.1	4.1
B	5.9	2.6
c	5.3	5.8
D	9.7	9.3
E	2.3	8.4
F	6.2	6.4
G	3.3	8.3
H	2.7	9.5
I	0.2	8.8

The problem is to connect *A, B,* . . . , *I* to the points **(l,O),** $(2,0)$, . . . , (9,0) in some order. DEK suggested a tree search procedure for determining what permutation of the points to use. The procedure starts by choosing arbitrarily some point to connect to *A*. Once such a connection is made, many other pairings become impossible. If *A* is connected to $(2,0)$, we can cross off all pairs of points that cannot be connected without crossing the path from *A* to **(2,0).**

	1	2	3	4	5	6	7	8	9
A	×	o	×	×	×	×	×	×	×
B	*x*	*x*	*x*						x
c	*x*	*x*							
D	*x*	*x*							
E		*x*	*x*						
F	*x*	*x*							
G		x							
H		*x*	*x*						
I		*x*	*x*						

The next decision should probably be which point to connect to $(1,0)$. This is because there are only four options remaining in the column labelled **1.**

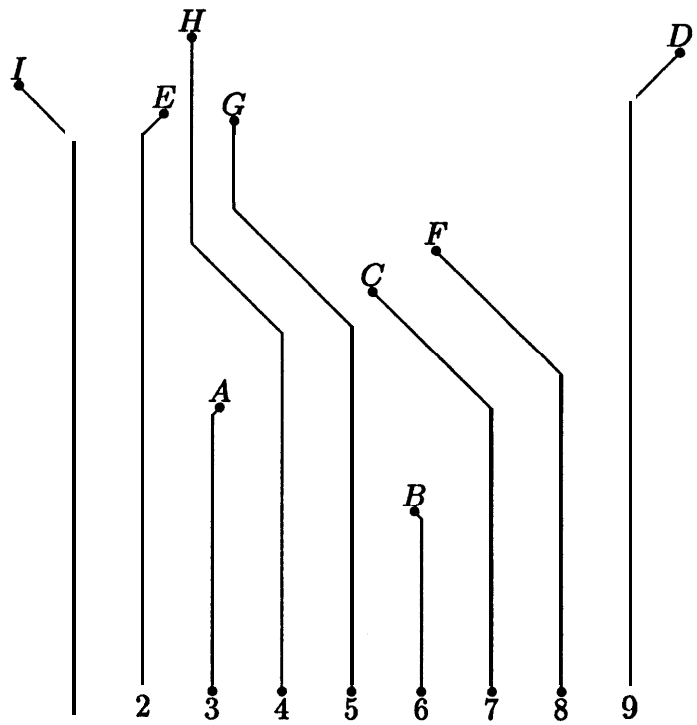
DS suggested that the algorithm would be simpler if it first decided where to connect point *A,* then where to connect *B,* etc. However, DEK explained that it.is usualy important to reduce the branching factor. He went on to point out that this strategy helps a great deal in the example. If the second decision is to connect point *H* to point **1,** there will be

no place for point I to go. This allows the algorithm to backtrack immediately and try to connect a different point to point 1.

WDW and others were concerned about the exponential behavior of this algorithm. WDW said that for n points there can be at least 2^n feasible ways to connect them. The exact bound was somewhat in doubt, but DEK suggested an example where $O(2^n/\sqrt{n})$ possibilities exist. Let all n points have z-coordinates equal to $(n+1)/2$. Then any n/2 of the n points can be connected to external points on the right side and the other $n/2$ can go to the left side.

DEK added that a Monte Carlo experiment of the type that had just been performed can do a surprisingly good job of estimating the size of a search tree. If t_i is the observed branching factor at depth i, the total size of the tree is the average value of $t_1 + t_1t_2 + t_1t_2t_3 \ldots$. Even though the variance of this expression is very large, it usually gives the right order of magnitude after a few trials.

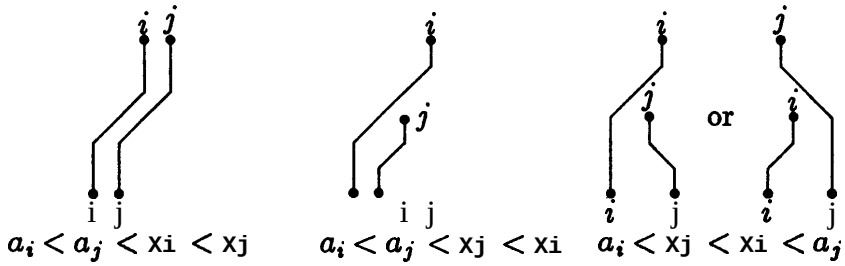
DEK brought up the question of how best to choose the exact paths once a mapping between internal and external points has been established. He had worked out the example ahead of time and come up with the following connection scheme.

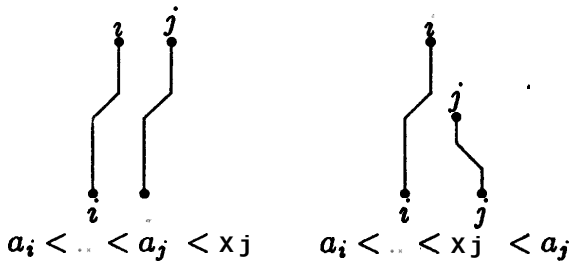


The choice of how to define the optimum paths is somewhat arbitrary, but the above paths were chosen in an attempt to maximize 6, the minimum separation between paths. DEK said it is convenient to use the Manhattan metric for δ. The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is just $|x_1 - x_2| + |y_1 - y_2|$.

DEK suggested a way to extend the exponential search algorithm to find the paths that minimize δ. In addition to checking that the paths can be drawn without crossing, it is also possible to check them for minimum separation. By considering each pair of paths, we can write down all the constraints that have to be satisfied in order to maintain a given minimal separation 6.

Consider path i from (x;, y_i) to (a_i, y_0) and path j from (x_j, y_j) to (a_j, y_0). These can be defined by the two variables z_i and z_j. There are several cases to consider depending on the relative order of x_i, x_j, a_i, and a_j.





It is safe to assume that $a_i < a_j$ and $a_i < x_i$ since the other cases are either just mirror images or they result from switching i and j. The actual constraints that apply in these cases are

$z_i \geq z_j - (\text{Xj} - \text{Xi}) + \delta$	when $a_i < a_j < x_i < x_j$
$z_i \geq y_j + (x_i - x_j) + \delta$	when $a_i < a_j < x_j < x_i$
$z_i \geq y_j + (x_i - x_j) + \delta$	when $a_i < x_j < x_i < a_j$ case **1**
$z_j \geq y_i + (x_i - x_j) + \delta$	when $a_i < x_j < x_i < a_j$ case 2
$z_i \geq z_j - (\text{Xj} - a_j) + \delta - (a_j - \text{Xi})$	when $a_i < x_i < a_j < x_j$
$z_i \geq y_j + \delta - (\text{Xj} - \text{Xi})$	when $a_i < x_i < x_j < a_j$

These equations only hold over a limitted range of δ. In particular, the last two equations are only required when $\delta > \min(a_j, x_j) - x_i$.

DEK pointed out that all the constraints except two involve only one variable, and these only occur when the 45° segments are parallel. This means that there can never be any loop in the equations. Even when all the equations are combined, no variable can appear on both sides of an inequality. This makes the equations extremely easy to solve.

DEK added that by combining this with the exponential search algorithm using a depth first search, the branching factor can be reduced even farther. Once a feasible set of paths has been constructed, the search routine need only consider branches that could lead to an improvement in 6.

Notes for Thursday, October 21

DEK promised to devote this day to discussing the progress each group had made on the problem. The first group was RB, NMP and PKR. RB explained that they had

originally planned to start with a solution such as the one obtained from the algorithm he presented last Thursday and use a hill climbing procedure to improve it. They later decided to combine this with the exhaustive search method which was presented in class.

A hill climbing procedure is based on some way of incrementally changing a potential solution and some measure of how good a solution is. One makes incremental changes until the current solution is better than any of its neighbors. At this point the current solution is at the top of a "hill".

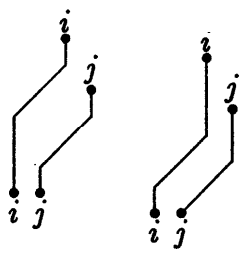
DEK explained that hill climbing is often a very successful approach to difficult combinatorial problems such as the layout problem. For example, this method has yielded some of the best practical solutions to the traveling salesman problem. (That problem is, given a set of cities and distances between them, to find the shortest round trip journey visiting all the cities. Incremental changes can be made to such a journey by reconnecting pairs of adjacent cities.) Similar changes allow hill climbing to be applied to the layout problem.

The next group was SRE, CGG, and DS. They said that they had originally planned to try all possible permutations using exclusion zones to limit the search. They also decided to use last Tuesday's tree search method, but they had a new idea for limiting the search, In addition to allowing δ to increase as better solutions are found, they planned to find the best way to connect subsets of the points and and use this to get an upper limit on 6.

DEK asked the groups to keep statistics on the size of the search tree and the number of configurations considered at each step. It would be interesting to compare the different strategies for limitting the search. To expedite this comparison, DEK suggested that each group consider the following five test cases. Take n = 9 and use the first 36 digits of each of the five constants π, e, γ, $\sqrt{2}$, and ln2. Let the digits give the coordinates of each of the internal points as the digits of π were used in last Tuesday's example. It was also suggested that a test case with n = 27 could be obtained by combining the first three of these.

. EJK was working with KAH. He said they had considered using the nine cases presented earlier for determining whether two paths cross. These were too complex so they tried exclusion zones instead, but this really boiled down to the same thing,

DEK agreed that the cases are somewhat more complicated than they appeared to be at first. He showed how some of the equations that he presented on Tuesday change as δ isincreased. When $a_i < a_j < x_i < x_j$ there are two cases depending on the size of δ.



When $\delta < x_j - x_i$, the only separation to consider is $z_i - z_j + x_j - x_i$. When $\delta > x_j - x_i$, however, $z_i - y_j + x_j - x_i$ must be greater than 6. Note that when $z_i < y_j$ as shown in the right hand diagram, the minimum separation is at most $x_j - x_i$.

EJB and WDW also planned to use a version of the tree search procedure that was presented in class. EJB said they would maintain a static table of the pairs of points that could not possibly be connected regardless of what other paths are drawn. This would avoid having to eliminate these pairs more than once.

DEK suggested that there may be other ways to save time by remembering solutions to subproblems. For instance, there may be small groups of points whose connections are largely independent of the others. By remembering the best way to connect these points it would be possible to avoid solving this same subproblem repeatedly. This is the basic idea behind dynamic programming.

WDW objected that it might take a lot of memory to remember all the subproblems. DEK agreed but he gave an example showing that this can save an enormous amount of time. It is possible to calculate Fibonacci numbers with a recursive procedure like

$$\mathbf{f(n) = if\ n \leq 1\ then\ n\ else}\ f(n-1) + f(n-2).$$

This would take an exponential amount of time although it could be done in linear time with dynamic programming.

SCS was working with WRB. He said they plan to find the best possible δ by using binary search. Their method was to pick a value for δ and try to construct a layout with that separation. If this is not possible then δ is too large, otherwise it should be increased. Their idea for constructing paths with a particular δ was to work right to left and make each path come to within δ of the preceding one.

The last group was SJR and MAH. SJR said they had written routines for recognizing exclusion zones. He showed how the zones can be enlarged as δ increases.

As time was running out, DEK compared this more geometrical approach to the more algebraic case analysis he suggested at the beginning of this problem. The geometrical approach often yields results that are very difficult to obtain algebraically, but it can lead to errors because diagrams often make things appear simpler than they really are.

Solutions for problem 2

All of the groups chose to maximize δ using some form of exhaustive search. Three groups used the Manhattan metric which was discussed in class, but SCS/WRB prefered the Euclidean metric. Everybody made some effort to choose a large y_0, although no one actually searched for a layout having the maximal y_0.

This turned out to be a difficult program to write and debug. Two of the groups definitely found suboptimal separations on some of the test cases. For the data derived from 7, the best separation turned out to be 0.6 and SCS/WRB were even able to achieve this using Euclidean distances. The data derived from e turned out to be rather uninteresting since two of the points $(2.7, 1.8)$ and $(2.8, 1.8)$ were very close together and all groups found $\delta = 0.1$. The data from π turned out to be more interesting: MAH/SJR were able to achieve $\delta = 0.5$; and SCS/WRB found $\delta = 0.4$ for Euclidean distance. The solution of MAH/SJR depended on properties of the Manhattan metric and would have a Euclidean δ of about 0.35.

One difficulty was the complexity of the code for testing geometrical separation between paths. Different groups tried to control this complexity in various ways. SCS/WRB, kept track of the maximum possible z value for each external point and the minimum possible z value for each internal point. They were able to simplify the constraints by considering these two cases separately.

MAH/SJR maintained the exclusion zone for each pair of points, one internal and one external. They considered the external points in left to right order so that it was possible to use only right-hand exclusion zones. Each time a new path was drawn, they updated all the exclusion zones so that these could be used to check future paths.

EJB/WDW used a boolean matrix as described in class on October 19th, but they considered the points. in left to right order instead of using the row or column of the matrix having the fewest possibilities remaining. All groups except RB/NMP/PKR considered paths in left to right order since this simplified the tests for separation.

RB/NMP/PKR were able to simplify some of the code by representing all the coordinates as integers.

All groups used the idea mentioned in class of increasing δ each time a better solution is found so that this can be used to control the rest of the search. This was augmented with an auxiliary search to increase δ as much as possible while maintaining the same mapping between internal and external points.

MAH/SJR also limited the search by making sure that the points that were not yet connected were never in any exclusion zones. When they found a new maximum δ and tried to improve upon it with the auxiliary search, they increased δ slightly and went back to the last point where the existing connections could achieve the new 6.

RB/NMP/PKR combined many different approaches for pruning the tree. As did EJB/WDW, they used the version of exclusion zones that DEK presented on October 19th where a left-right ordering must be preserved between points i and j whenever $|x_i - x_j| \geq |y_i - y_j|$. This also puts static constraints on which external points can be connected to which internal ones. RB/NMP/PKR first considered the internal points with the fewest such possibilities and then considered the other points at deeper levels in the search tree.

Whenever they found a complete set of connections, RB/NMP/PKR used a hill-climbing procedure in addition to the auxiliary search. The auxiliary search increased δ as much as possible without changing the mapping between internal and external points. The hill climbing procedure tried to improve upon this by swapping pairs and triples of points.

RB/NMP/PKR started their search from the solution produced by RB's algorithm and due to a bug, they were unable to improve upon it. Before giving up, their search considered 47 configurations for the data from 7, 24 for e, and 88 for π. For a random set of 20 points, their search went 11 levels deep and considered 95 configurations.

MAH/SJR gave statistics for the size of the search both with and without tree pruning. With pruning, they considered 41 positions for 7, 18 positions for e, and 27 for π. Without pruning the numbers were 265 for 7 and 331 for e. For 20 random points, the pruning reduced the tree size from 87491 to 99. In this case there were far more configurations considered at deeper levels in the tree than at more shallow levels, although in general, most of the groups found that about as many configurations were considered at all levels of the tree.

EJB and WDW were less concerned about the size of the search tree. They searched 449 configurations for 7, 320 for e, and 184 for π.

SCS and WRB did not give statistics although, like most of the other groups, their program was fast enough to produce a layout for 27 points.

Some of the groups had good ideas that they did not have time to implement. Three groups had ideas for connection schemes where in some sense all separations would be as large as possible, not just the minimum separation. SCS/WRB mentioned this but gave no details. EJB/WDW considered finding the amount of "slack" for each path in the final scheme and using this to increase the separation between some of the paths. The "slack" of a path is dependent on how much separation there is on each side in addition to the minimum 6. MAH/SJR had a similar idea for increasing the separation of some of the paths. On October 14th, DEK described how a set of connections can be broken into contiguous groups of paths, all of which slope the same way. The idea was to consider each group separately and even out the spacing by considering the heights of the parallel diagonal segments.

Problem 3, due November **9: Dynamic Huffman coding.**

Suppose that we want to transmit a file from computer to computer, compressing the information so that it isn't necessary to transmit too many bits. One way to do this is to consider the file as a long message on a 256-letter alphabet and to use a "binary prefix code." This means that each of the 256 bytes maps into a string of O's and l's, where no string is a prefix of another; such messages are easily encoded and decoded. We gain efficiency if the most common bytes are mapped into strings that require fewer than 8 bits.

If byte k occurs w_k times in the file, and if it maps into a string of length l_k, the number of bits we need to transmit is $w_0 l_0 + w_1 l_1 + \ldots + w_{255} l_{255}$. A well-known algorithm due to David Huffman can be used to find an optimum prefix encoding for a given set of weights $\{w_1, \ldots, w_n\}$ on an n-letter alphabet, i.e., a prefix code that minimizes the sum $w_1 l_1 + \ldots + w_n l_n$. It proceeds as follows: If n = 2, we simply assign the strings 0 and 1 to the two letters in the alphabet. If n > 2, Huffman's algorithm takes two letters with the smallest weights, say w_i and w_j, and replaces them by a "superletter" with weight $w_i + w_j$. This yields a problem on n − 1 letters that can be solved recursively; and if that solution assigns the string α to the superletter, we map the two low-frequency letters that were lumped together into the strings $\alpha 0$ and $\alpha 1$.

For example, suppose there are n = 4 letters {a, b, c, d}, with respective weights 2, 3, 4, 6. We combine a and b into the superletter (ab) of weight 5; now we have a problem with n = 3, and the algorithm combines c with (ab), leading to a problem with n = 2. Thus, d maps into 0 and the super-superletter $(c(ab))$ maps into 1. Unwinding, we let c ↦ 10 and (ab) ↦ 11; finally a ↦ 110 and b ↦ 111. This is an optimum code.

But there's a catch when we try to apply this idea to file transfer: The computer sending the file can compute the weights w_0 to w_{255}, but the computer receiving it won't know the code unless we fix it in advance or transmit it as a preamble to the file. Therefore it is natural to look for methods that "learn" the probabilities as they go on. Our problem will be to transmit messages $a_1 a_2 a_3 \ldots$, where each of the a_k's is a letter in a given n-letter alphabet, and where the letter a_k is transmitted according to a prefix encoding that is optimum for the frequencies of letters in the already-transmitted sequence $a_1 \ldots a_{k-1}$. We shall call this dynamic *Huffman coding,* because we continually update the code as we transmit the message.

For example, suppose that n = 4 and that we have agreed on some scheme to transmit the first letter (since the weights are all zero initially); let's say the starting code is a ↦ 00, b ↦ 01, c ↦ 10, d ↦ 11. If the first letter is b, the second letter will be transmitted with a code in which b has length 1 (e.g., a ↦ 10, b ↦ 0, c ↦ 110, d ↦ 111). If the first 15 letters involve 2 u's, 3 b's, 4 c's, and 6 d's, we will transmit the sixteenth letter with a code in which d maps to a string of length 1 and c maps to a string of length 2, as in the example above. Gradually the code converges to one that is optimum for the file as a whole. Both sender and receiver can be computing the dynamic Huffman code using the same algorithm, so the messages will be consistently encoded and decoded.

The goal of this problem is to invent an algorithm that computes a dynamic Huffman code in "real time." This means that the number of steps needed to produce the encoding of each letter is at most a constant times the length of the string used to encode that letter. It may even be desirable to use the method with an alphabet of size 65536.

Notes for Tuesday, October 26

DEK started class by explaining that, unlike the other problems, problem 3 is not completely unsolved. When he solved this problem last year, he thought it was pretty instructive, so he decided to hold off publishing it until after this class had had an opportunity to solve it.

DEK introduced the problem by explaining Huffman coding and some of its applications. One application comes from merging files. Suppose we have a set of files to be merged together two at a time. A merging scheme can be represented as a binary tree where the leaves are the files and each internal node is a file created during the merging process. The file at an internal node is formed by merging together its two children. The number of times each record is processed is given by the depth of its file in the tree. The total number of records processed is the sum of the number of records in each file times the depth of that file in the tree.

Another formulation of the problem is that of finding binary prefix codes. There is a one to one correspondence between binary trees and codes with no "wasted bits". A bit is wasted if it could be predicted from the preceding part of the code word. A wasted bit would correspond to a node in the tree with just one child.

If the code words represent letters from some alphabet and we assign probabilities to each, then the average length of a code word corresponds to the weighted path length in the tree. An optimum code is one that minimizes this quantity.

Problem 3 involves finding optimum binary prefix codes. DEK demonstrated David Huffman's algorithm for doing this. The algorithm repeatedly combines the two least frequent letters into one "superletter". To see why this works, consider the corresponding trees. For an optimum prefix code, the two least frequent letters must be at the lowest level in the tree. Otherwise, the weighted path length could be reduced by swapping one of these letters with a more common one at a greater depth in the tree. Note that there must be at least two nodes at the lowest level because nodes are not allowed to have just one child. Nodes at the same depth can be permuted arbitrarily without changing the weighted path length. It is therefore easy to make the two least common letters siblings in the tree so that they can be thought of as one superletter.

DEK explained that the purpose of this problem is to maintain the optimal code dynamically. Each letter in the file is to be encoded using a Huffman code with the letters weighted according to their frequency in the previous part of the file.

It would be possible to use the letter frequency for the whole file with the standard Huffman coding but this could take too much time and memory in some applications. There is also the problem of how to communicate the code to the receiver. RB suggested just putting the encodings of each letter at the start of the file, but this leaves the problem of how to delimit them.

The discussion turned to possible shortcomings and improvements of the scheme. SJR and RB brought up the question of how well the beginning of the file could be used to forecast the future. There was general agreement that different kinds of files would have

different byte frequencies and the beginning of a file would give some indication about what kind it is. However, WDW suggested that the file might be divided into pieces with drastically different behavior. For instance, it might consist of a-program followed by data. It might be better to base the code on only the most recent part of the file so that it would be able to adjust better when the files.

DEK said he had considered using a "window" of this sort but it would make the problem more difficult since it would involve handling deletions. He said that when thinking about a problem people tend to overemphasize the worst case. People are often surprised when they learn where a program is really spending its time.

RB pointed out that the ideal byte size depends on the type of file being encoded and SJR suggested using variable length bytes. In a text file it would be best to use the words, since a file is not likely to use a huge vocabulary.

DEK brought up the question of how to get the coding started in the original problem. Since all the byte frequencies are zero originally, the weighted path length would be zero even for degenerate trees. It would probably not be a good idea to encode the first byte with such a tree. If instead, all the weights were set to 1, then the trees with minimum weighted path length would be those with all leaves on two adjacent levels.

PKG suggested always using an optimal tree of minimal height. DEK showed how this could be done with Huffman's static algorithm. Whenever the two minimal weight letters are not uniquely determined because some of the weights are equal, always combine the two trees of the smallest height. This is known to produce the tree of minimum height among all the otherwise optimal trees.

Notes for Thursday, October 28

DEK began by presenting a general encoding scheme for the case when all the weights are equal. He demonstrated a simple function $f_n(k)$ for encoding an arbitrary symbol k from the alphabet O,l,. . . , $n-1$. The function is based on two quantities p, and q.

$$f_n(k) = \begin{cases} \text{the last } q \text{ bits of } k, & \text{if } k < p; \\ \text{the last } q+1 \text{ bits of } k+p, & \text{if } k \geq p. \end{cases}$$

The numbers p, and q should satisfy $2^q \leq n \leq 2^{q+1}$ and $n + p = 2^{q+1}$. As k increases, the code words increase lexicographically. All the code words are either q or $q+1$ bits long and p is the dividing line between these two cases. The equations for selecting p and q insure that the code will use all possible prefixes and that the last code word will be a sequence of 1 bits. When n is a power of 2, there are two ways to select p and q but they both result in the same code.

DEK presented an example from his paper using this function to determine the encoding when all the weights were zero. The alphabet consists of the letters a, b, . . . , z and a stop symbol ! . The algorithm constructs a series of trees $T_0, T_1, T_2, \ldots$ and uses T_{i-1} to encode the ith symbol in the message. The leaves of the trees consist of letters from the alphabet weighted according to how many times they have occurred in the message, except there is also a special leaf of weight 0 for the letters that have not occurred yet.

The special leaf is labelled with a sequence of letters. If we let n be the length of this sequence, then the encoding of the kth letter is $f_n(k)$ prefixed by the encoding given by the tree for that leaf. When one of these letters occurs in the message, it is removed from the sequence, given a weight of 1, and placed in the tree accordingly. Its place in the sequence is filled by the letter at the end of the sequence.

The message abracadabra ! would be encoded as follows. The first tree T_0 has the single leaf (a, b, ..., z, !). We have n = 27, p = 5, and q = 4. The first letter in the message is a and its encoding is 0000.

The root of T_1 has two leaves: on its left branch labelled 0 is the special leaf (!, b, c, ..., z); on its right branch labelled 1 is the letter a. The next letter in the message is b. This has not occurred before, so we use $f_n(k)$ where k = 1, n = 26, p = 6, and q = 4. This is prefixed by 0, so we have the encoding b $\mapsto$ 00001 as SRE pointed out.

The next tree T_2 is formed by replacing the special leaf with a two leaf subtree having (!, z, c, d, ..., y) on the left and b on the right. The next letter r is encoded using k = 17, n = 25, p = 7, and q = 4. We get 00 from the rest of the tree; $k + p$ = 24 so r $\mapsto$ 0011000.

DEK's algorithm formed T_3 by swapping the two main subtrees of T_2 and replacing the special leaf with a tree having r as its right branch. This puts leaves of weight 1 at depths 1, 2, and 3. It would be possible to have them all at depth 2, but the weighted path length would be no less. The next letter is a which has occurred before (and as NMP pointed out a $\mapsto$ 0).

The tree T_4 is the same as T_3 except a has weight 2. The example was getting too long, so DEK just wrote down the rest of the encodings:

c ~10000010
a $\mapsto$ 0
d $\mapsto$ 110000011
a $\mapsto$ 0
b $\mapsto$ 110
r $\mapsto$ 110
a $\mapsto$ 0
! $\mapsto$ 10000000.

The next problem was to devise a data structure to enable the code to be updated in real time. DEK mentioned a slight complicating factor that depends on exactly what is meant by "real time." The most natural way to generate the encoding may be to go up the tree putting bits on a stack and then read them off the stack. This would not be allowed if the real time constraint required at most a constant time between the sending of any two bits. Problem 3 is not so restrictive; a stack is allowed.

WDW asked how fast the encoding should really be. DEK said that something like 100 bits per second would be sufficient for the problem. If the algorithm is to be used in an actual application, it could be implemented in hardware and made much faster.

PKG was concerned about the feasibility of doing the encoding in real time. Sending $\Omega(n)$ symbols from an n letter alphabet seems to require sorting. It is well known that

this requires time $\Omega(n \log n)$. DEK pointed out that $\Omega(n \log n)$ bits would be sent in this case, so this much time would be allowable. He agreed, however, that Huffman coding does seem to be related to sorting since $\Omega(n \log n)$ time is required to find an optimal prefix code on n symbols, while this can be done in C(n) if the weights are sorted originally. The first result is due to Mike Paterson. It says there can be something like *n!* different optimal trees so $\Omega(\log n!)$ binary decisions would be required to select one. If the letters are given in a list sorted by weight, however, Huffman's algorithm can be modified to keep the superletters in a separate list and look at both lists when choosing the letters to merge.

PKG pointed out that Huffman's algorithm can also be generalized to minimize functions other than weighted path length. DEK clarified this by giving a set of properties that are sufficient to make a function minimizable by Huffman's algorithm.

$$w \circ x = x \circ w$$
$$w \circ x \geq w$$
$$(w \circ x) \circ (y \circ z) = (w \circ y) \circ (x \circ z)$$
$$(w \circ x) \circ y \leq w \circ (x \circ y) \quad \text{when } w \leq y$$

The superletter corresponding to two letters a_i and a_j could be given the weight $w_i \circ w_j$ instead of $w_i + w_j$. Huffman's algorithm would then find the way of combining all the letters that minimizes the total weight as calculated this way.

The ordinary version of Huffman's algorithm can by obtained by using ordered pairs for the weights. Define

$$(W, W') \circ (X, x') = (W + x, w + w' + x + x')$$

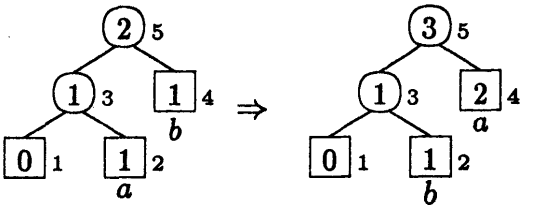
and order the pairs lexicographically. If the alphabet is $a_1, a_2, \ldots$, a, and the weights are $w_1, w_2, \ldots, w_n$, then give a_i the weight $(w_i, 0)$. The weight of the entire tree will have $w_1 + w_2 + \cdots + w_n$ as its first component, and the weighted path length as its second component. Since the first component does not depend on the tree, the second component will be minimized.

The most general continuous version of this is $(w \circ x) = k(w + x)$ where $k > 1$. The above version is the limit of this as $k \to 1$. There are papers on such generalizations by D. Stott Parker in Siam Journal *of* Computing 9 (1980), 470-489 and by Don Knuth in Journal *of* Combinatorial *Theory* **(A) 32** (1982), 216-224.

Getting back to the problem at hand, SJR suggested not having an initial alphabet but sending a representative of each letter when it first appears. This would save space if the alphabet were very large so that many of the letters would probably never appear. DEK pointed out that this would take extra space if all the letters did appear. Suppose we have a 100,000 word dictionary and **5,000** of them actually appear in the message. In the new scheme, each new word would take about lg 100,000 bits and this would amount to 5,000 lg 100,000 bits altogether. The old scheme would require lg 100,000 + lg 99,999 + $\cdots$ + lg 95,001 bits which is slightly better. Even if all the letters occurred, however, the difference would be nlgn $-$ lg *n!* bits, which is always less than $1.4n$.

Notes for. Tuesday, November 2

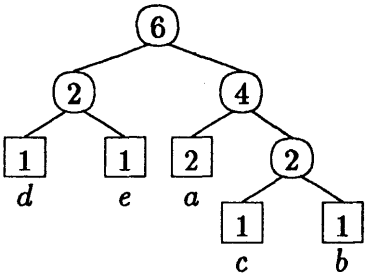
At the start of class, RB said that he and **WRB** had come up with a solution to the problem. Maintain a binary tree with weights and parent pointers **in** each node and **put** all the letters that have not occurred yet in a special leaf as **DEK** explained last time. The data structure also includes a linked list giving the nodes in order of increasing weight. Using this data structure, the tree on the left can be obtained by starting with an empty tree and inserting first a and then *b*. (The small numbers to the right of each node indicate the position in the auxiliary linked list.)



Suppose an a occurs in the file so that its weight in the tree must be increased to 2. Then the leaf labelled a is swapped with the last node of weight 1 in the linked list. In the example, this node is the leaf labelled *b* and we obtain the tree on the right.

RB explained that the general scheme is to take the node whose weight is increasing and swap it with the last node in the linked list having the same weight. In other words, the node is swapped with the head of its class. SJR was concerned that chaining through the list to find this node could take an unbounded length of time, violating the real time constraint. RB insisted, however, that he had a data structure that allows this to be done in constant time, and the class decided to put off this point for a while since the correctness of RB's algorithm had not been resolved yet.

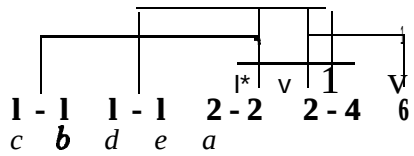
DS and **WDW** said they had come up with a similar algorithm. They presented an example that they felt would be a difficult case for RB's algorithm.



The problem here is to update the tree so that a has weight 3. This will require swapping the a node to the next higher level of the tree.

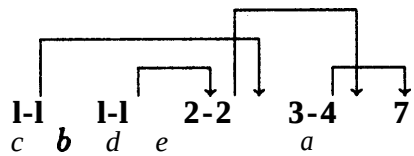
Before RB could apply his algorithm to this tree, the auxiliary list had to be added. RB explained that it would not be sufficient to just put the nodes in this list in non-decreasing order; certain orderings could not be constructed if the tree were built from scratch and the algorithm could not be expected to work with such an ordering.

WDW suggested using a bottom to top, left to right ordering. DEK showed how the nodes could be listed in this order using arrows to indicate the tree structure.



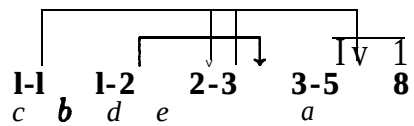
The diagram above lists both the leaves and the internal nodes according to their weight. The leaves are labelled with letters as in the tree diagram and the siblings in the tree are connected by horizontal lines with arrows connecting each pair of siblings to their parent. The arrows correspond to the parent pointers in RB's data structure.

RB's algorithm accounts for the increasing weight of a by switching it with the parent of *d* and e. The tree pointers move with the nodes and the result can be diagrammed as follows:



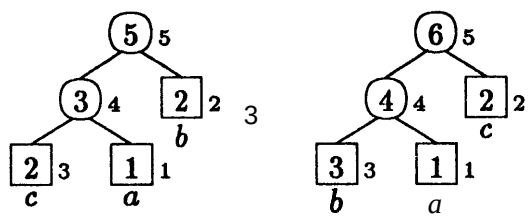
MAH pointed out that this example does not reveal the full complexity of the problem. The leaf labelled e is already at the head of its class, so there is no need to swap it with anything when its weight increases by one. However, the parent of e is not at the top of its class, so the list will get out of order if the algorithm stops immediately.

RB explained that he would solve this problem by also swapping e's parent to the head of its class and doing the same for all the ancestors of e. The result is the following:



The parent of *d* and e has been swapped with the parent of *b* and c to get it to the head of the weight 2 class, but none of e's other ancestors had to be moved.

SJR brought up another difficulty which is best understood from the standard tree representation. As before, the numbers in the nodes are the weights and the small numbers to. the right give the order in the list.



The tree on the right is what would result from using the algorithm given so far to update the tree on the left for an additional occurrence of *b*. The node *b* was swapped with *c* which was the head of its class. Since this actually increased the depth of *b*, the result is no longer a Huffman tree.

DEK pointed out that a and c were siblings but were not adjacent in the ordering. Since siblings were always adjacent in all the previous examples, the problem can probably be avoided if we can always enforce this constraint. This way, when two siblings have different weights, the lighter one will always be at the top of its class and the other one will be at the bottom of its. When the siblings are adjacent, the tree will always be one that could be obtained from Huffman's algorithm using the given weights.

The algorithm appears to be valid as long as the swapping operation preserves the condition that siblings are adjacent. DEK formalized this by giving a set of conditions that should be necessary and sufficient for a tree to be obtainable from Huffman's algorithm. There must be some ordering on the nodes so that the weights satisfy the condition

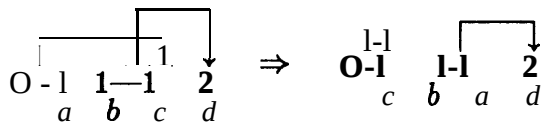
$$w_1 \le w_2 \le w_3 \le \cdots \le w_{2n-1}$$

and for every j < n, nodes 2j − 1 and 2j are siblings and their parent has weight $w_k = w_{2j-1} + w_{2j}$ with $k > j$.

SRE asked why we need the condition that parents have indexes in the ordering that are strictly greater than either of those for the children. Since the weights are additive, this condition would indeed be superfluous if none of the weights were 0.

DEK pointed out that the algorithm could be made conceptually simpler by doing all the swaps first, and then updating the weights. Since none of the weights increase by more than one at a time, the updating process will not destroy the ordering of the weights if all the nodes to be updated are first swapped to the head of their class.

DEK showed how nodes of weight 0 can mess up this scheme.



Starting from the tree on the left and swapping a with the head of its class yields the disconnected structure on the right. The problem is that we tried to swap a with his own parent. This can only happen when there are nodes of weight 0, since otherwise, parents can never have the same weight as their children.

The discussion now turned to ways of implementing the algorithm in real time. CMA suggested keeping each weight class in a linked list as well as maintaining the standard tree structure. There are pointers from each node to the class it is in, and from each class to the next heavier class.

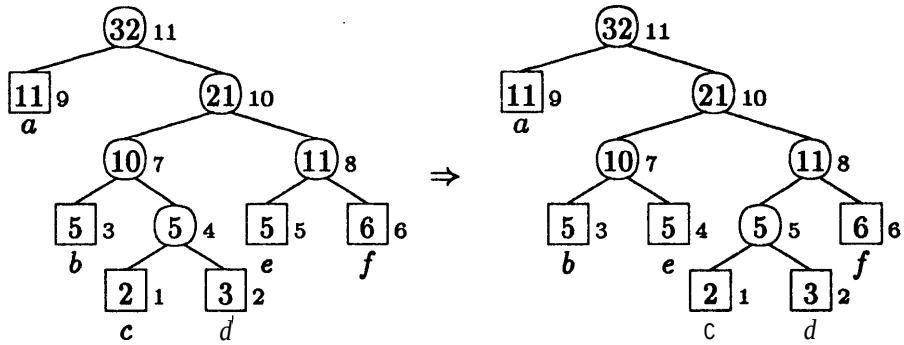
DEK explained that the challenge in this problem is probably to come up with an efficient data structure and write it up clearly. The actual program should be fairly short.

WDW pointed out that in order to show a data structure is fast enough, it will be necessary to prove an upper bound on the number of swaps that have to be performed for each update. RB explained that, since the length of the code for a letter is its number of ancestors in the tree, there can be at most this many swaps.

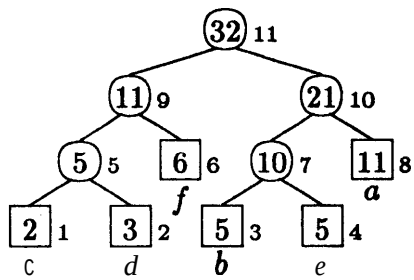
Notes for Thursday, November 4

DEK started class with an example showing how the swapping algorithm can affect the tree structure. He chose to first swap the appropriate nodes to the heads of their classes

and update the weights later. The tree on the left is being updated for an additional occurrence of *d* and the next step is shown on the right



The last node in the weight 5 class was e, so the parent of c and *d* has been swapped with that node. We are not finished yet, however, since the root of the subtree containing c, *d*, and *f* is not the last node in the weight 11 class. It must be swapped with a as shown below.



This shows that the tree structure can be changed a great deal by a single update.

DEK reiterated now that he had separated the operations of reordering the list and updating the weights in order to make the algorithm easier to understand and prove correct. It may be more efficient to perform these operations together, but it is best to get a working program first and then optimize it later.

The discussion now turned to methods for implementing the algorithm. SRE was working with MAH and EJB. She said that they had decided to use a doubly linked list of nodes with auxiliary records for each weight class. The class records would contain pointers to the first and last nodes in their class. In addition to this, of course, the standard tree structure would be maintained using pointers to parents.

WDW asked why pointers are needed to the lowest members of the classes since the swapping operation refers only to the highest. EJB said these pointers were used to determine whether a class has only one element, but it became clear that these extra pointers were not needed. DEK pointed out that by linking the class records together, it would be possible to refer to the lowest member of a class as the node after the highest member of the lower class. Anyway, as SRE pointed out, this would not be necessary because it is possible to determine whether a class has one element by just looking at the weight of the previous node.

Since it appears that the list will always contain exactly $2n - 1$ nodes, DEK suggested just putting them in an array. After all, if there are no insertions and deletions, why use

a linked list? As MAH pointed out, something will have to be done about nodes of weight **0** before this idea will work. WDW suggested letting the weights on the nodes be one more than the number of occurrences in the file so that there would never be any nodes of weight 0. However, as SRE pointed out, the resulting trees would not be Huffman trees with respect to the original weights.

DEK said that he had peeked at his paper and he had used a doubly linked list for the class records. Now, however, it is not clear why this is necessary. WDW suggested that the class records could also be put into an array. His original idea was to put the pointer to the head of class i in the ith position of the array. This had the problem that i can be arbitrarily large, so SJR suggested using an array of length 2n – 1 instead as this is the maximum number of different classes that could exist at once.

This brought of the question of how to allocate storage in such an array. DEK explained that this could be done by linking together all the unused cells in a free list. The array entries for freed cells can be used to link them together. When a new class is created, a new cell can be taken from the free list. This works as long as used cells are always inserted into the free list when they are no longer needed.

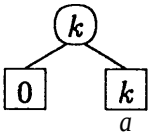
DEK summarized- the data structure defined so far. It consists of five arrays, each with 2n – **1** elements: The parent of the kth node is P_k; its left child is C_k; its weight is W_k; and G_k is a pointer to the location for its class in the T array. Note that since siblings are always adjacent, no pointer is needed for the right child.

At this point CGG made a conjecture that the ordering used by the algorithm gives the nodes in order of decreasing depth in the tree. It is certainly not possible for a heavier node to be at a deeper level than a lighter one since this could not happen in a Huffman tree. Furthermore, DEK agreed that if there are no nodes of weight **0,** then nodes in the same class can be at most one level apart.

If this condition holds in general, it may make it easier to prove that the algorithm is valid.

Solutions for problem 3

All groups implemented essentially the algorithm discussed in class except **DS/WDW** simplified their program by initializing all the weights to 1 instead of 0. The other groups use one node of weight zero for all the letters that have not occurred in the input so far. There is one problem associated with zero weight nodes that was not solved in class.



Everybody noticed that if the parent of *a* is the head of its weight class, then there is no need to swap them when updating the tree for an additional occurrence of a.

RB/WRB/EJK, DS/WDW, CGG/PKR/SCS, and NMP/SJR used 2n – 1 element. arrays of weights and pointers to the parent, the left child, and the class. EJB/SRE/MAH used a variation on this structure for their decoder, but used only records and pointers

in their encoder. For both programs, they kept the weights in class records along with the pointers to the head of each class. For their encoder, they kept the nodes in a doubly linked list with a parent pointer, a class pointer, and a bit for distinguishing left and right in each node.

DS/WDW, CGG/PKR/SCS, and NMP/SJR also gave separate programs for encoding and decoding, but most of the code was the same in both versions. NMP/SJR also had a routine for printing a picture of the final tree.

CGG/PKR/SCS tested two versions of their programs: one version used initial weights of one instead of zero. They found that the `class.tex` file required 9,450 bits (or 5.82 bits per a character) to encode with initial weights zero but only 9,310 bits (or 5.73 bits per a character) with initial weights of one. A PASCAL source file about six times longer that class. tex required 4.37 bits per a character with initial weights of zero and 4.40 bits per a character with initial weights of one. There does not appear to be any significant advantage to using initial weights of zero instead of one.

No other groups experimented with any other files besides class `.tex`, but those groups that kept track found that about 9,700 bits were required to encode it. This is somewhat more than CGG/PKR/SCS used but the groups had different end of line conventions and different ways of handling zero weight nodes.

For nodes of weight zero, EJB/SRE/MAH used regular seven-bit ASCII codes instead of the $f_n(k)$ function which DEK described on October 28. This simplified their program and it should have cost them about 100 bits, but there were so many other differences between groups that this cost is difficult to evaluate.

Since the domain of $f_n(k)$ is $\{0, 1, \ldots, n-1\}$, but the zero weight letters are actually scattered between 0 and 127, some mapping between these sets had to be maintained using arrays. When one of these letters had to be removed from the mapping, RB/WRB/EJK and CGG/PKR/SCS swapped it with the letter in position $n-1$ and decremented n. SJR/NMP solved the problem by leaving pointers from each letter with code greater than $n-1$ to its current encoding. The first n cells in their array contain the zero-weight letters and if a character no longer has weight zero, then the cell it points to no longer contains that character.

Problem 4, due November 23: List processing **with virtual memory.**

The purpose of this problem will be to design a way to test the efficiency of a proposed algorithm (rather than to develop a new algorithm). Of course, if you think of an algorithm that is better than the one to be proposed, your purpose will be to design a way to demonstrate the superiority of your own discovery. Our main emphasis will be on methods of empirical testing.

But before we get into the problem itself, it is necessary to set the scene: Some important applications of List manipulation require that each List operation be done in a bounded amount of time; one cannot afford to pause every now and then to wait for garbage to be collected and for memory to be reorganized. (For example, think of a program that is supposed to play ping pong.) Therefore some interesting algorithms have been proposed that "amortize" the time for garbage collection by doing a little of the work whenever new nodes are allocated. We shall consider the procedure of Henry G. Baker, Jr. [*CACM* 21 (1978), 280–294], which is based on previous work by dozens **of** other people including Marvin Minsky and Edsger Dijkstra. Then we shall modify Baker's algorithm (in a previously unpublished and untried way) for possible use in the hardware of a List-processing computer that someone may wish to build, by adding some features that attempt to make it perform well in a virtual memory environment. This means that we are looking for a scheme that not only does the List operations in real time, it also keeps the data organized in such a way that pointers tend to refer to nodes on the same page of memory.

We shall abstract the problem as follows: Memory consists of 2N nodes, addressed from 0 to 2N − 1. Each node p contains two pointers, called $x[p]$ and $y[p]$, where a pointer is either A or the address of a node. (We assume that A is distinguishable from 0; later this restriction will be removed.) There are also r registers, $R_1 \ldots R_r$, each of which holds a pointer. The machine needs to implement the following List instructions:

[fetch] $R_j \leftarrow l[R_k]$, where $l = x$ or y and $1 \le j, k \le r$ and $R_k \neq \Lambda$.

[store] $l[R_k] \leftarrow R_j$, where $l = x$ or y and $1 \le j, k \le r$ and $R_k \neq \Lambda$.

[allocate] $R_j \leftarrow$ **new,** where $1 \le j \le r$.

Baker's algorithm implements each of these instructions in a bounded amount of time.

Here's how it works: The memory is divided into two halves, called the "good" half G and the "condemned" half C. The good half runs from addresses Go to $G_1 - 1$ and the condemned half runs from addresses Co to $C_1 - 1$, where we always have (Go = 0, G_1 = N) and (Co = N, C_1 = 2N) or vice versa. We say "p is good" if $G_0 \le p < G_1$, and "p is condemned" if $Co \le p < C_1$; the null pointer is neither good nor condemned, it's just A.

The algorithm maintains three internal pointers to the good area of memory:

s points to the next node to be "certified";
f points to the next free node in G;
t points to the end of the free area in G.

We always have $Go \le s \le f \le t \le G_1$. Locations p in the range $f \le p < t$ are free, i.e., they contain no nodes accessible to the user. The node in location p is said to be certified

if $G_0 \le p < s$ or if $t \le p < G_1$. The basic invariant underlying Baker's algorithm is that *neither the registers* R_j nor *the link fields* $x[p]$ *or* $y[p]$ *of certified nodes p ever point to condemned* nodes. Thus, condemned nodes may be present behind the scenes, but the user's **algorithm**—which is based only on fetching, storing, and allocating, together with operations that do not affect the List structure-never gets to see them, since pointers to condemned nodes never appear in the registers.

One of the important consequences of the stated invariant is that none of the condemned nodes is accessible when $s = f$. Putting this another way, if all good nodes **have** been certified, the condemned area consists of nothing but garbage.

Baker's algorithm has three subroutines. The first one moves a node from the condemned area to the good area and leaves a forwarding address in its x field so that other pointers to this node will be able to find it in its new location. The forwarding addresses will be the only pointers from C to G.

```
procedure move (p : pointer);                { we assume that p is condemned}
  begin if f = t then overflow;              { error stop if good area is full}
  x[f] ← x[p]; y[f] ← y[p];                  { copy the node into the good area)
  x[p] ← f;                                  { leave a forwarding address }
  f ← f + 1;                                 { that node is no longer free}
  end
```

The second subroutine finds the good equivalent of a pointer that might refer to the condemned area:

```
function cleanse (p: pointer):pointer;
  begin if p is condemned then
    begin if x[p] isn't good then move(p);
    cleanse ← x[p];                          { return the good equivalent of p }
    end
  else cleanse ← p;                          {return p itself, if good or A}
  end
```

The third subroutine is used to certify an entire node:

```
procedure certify;
  begin if s < f then                        { are there are uncertified nodes? }
    begin x[s] ← cleanse (x[s]);             { wash the x field }
    y[s] ← cleanse (y[s]);                   {wash the y field }
    s ← s + 1;                               {the node is now certified}
    end;
  end
```

We could implement the fetch operation $R_j \leftarrow l[R_k]$ by simply saying $R_j \leftarrow$ *cleanse* $(l[R_k])$;. but it is slightly more efficient to fetch by doing two steps: **if** $l[R_k]$ is condemned **then** $l[R_k] \leftarrow$ *cleanse* $(l[R_k])$; $R_j \leftarrow l[R_k]$. [Exercise: Explain why this is better.]

The store operation, $l[R_k] \leftarrow R_j$, can be done without violating any of the invariants. So we are left with '$R_j \leftarrow$ **new**', which can be implemented as follows:

```
if f = t then                                  {is the good area full?}
   begin if s < f then overflow;               {error stop if uncertified nodes remain}
   f ← C0; C0 ← G0; G0 ← f; s ← f;             {switch good and condemned}
   t ← C1; C1 ← G1; G1 ← t;                    {and make the newly good area empty}
   Rj ← Λ;                                     {destroy unwanted register contents}
   for 1 ≤ k ≤ r do Rk ← cleanse (Rk);          {restore the invariant}
   end
else certify;                                  {otherwise certify another node}
if f = t then overflow;                        {error stop if good area is full}
t ← t − 1;                                     {make room for new certified node}
x[t] ← A; y[t] ← A;                            {initialize its pointers}
Rj ← t;                                        {plant a pointer to the new node}
```

Incidentally, we obtain a conventional non-real-time garbage collection scheme, if the code '**end else** *certify*' is replaced here by '**while** s < *f* **do** *certify* **end**'. The algorithm starts with Go = *s* = *f* = 0, G_1 = *t* = Co = N, $C_1 = 2N$, and all R_j = A. It can be shown that overflow will never occur if the user never needs $\frac{1}{2}N$ nodes or more at any one time.

That is Baker's algorithm. The new twist is to tie this in with the idea of virtual memory; since nodes are being moved, we might as well try to make as many links as possible point to nodes on their own page. Let's suppose that there are $2n$ pages of m nodes each, so that N = *mn;* we say that nodes p and *q* belong to the same page if and only if $\lfloor p/m \rfloor = \lfloor q/m \rfloor$. The following modified algorithm reserves the first word of each page for system use (therefore it will never appear in a register, and we could let A be represented by *0).* We use the notation ***first(p)*** for the system node on p's page; i.e., $first(p) = m\lfloor p/m \rfloor$.

The basic idea is to interrupt the order of certification when the first node is being moved to a new page, and to certify that entire page first. We maintain a stack $s_1 s_2 \ldots s_w$ of partially certified pages, and the new convention is that a good node p $\leq f$ is certified unless *p* is on the same page as some stacked node s_i and $p \geq s_i$. Variable *s* now represents the top item s_w on the stack. and $s_{i-1} = x[first(s_i)]$; an empty stack is represented by the-condition *s* = *0.*

The original algorithm is patched as follows: Insert the code

```
if f mod m = 0 then                    {is f a system node?}
  begin x[f] ← S;                      {push down the old page position}
  f ← f + 1; s ← f;                    {and switch attention to the new}
  if f = t then overflow;              {error stop if good area is full}
  end;
```

just before '$x[f] \leftarrow x[p]$' *in move.* Change '$s < f$' to '$s \neq 0$' in *certify* and in the **new** routine. Change '$s \leftarrow f$' to '$s \leftarrow 0$' in the **new** routine. And after '$s \leftarrow s + 1$' in *certify,* insert the following:

```
if s mod m = 0 then                    {did we finish certifying a page?}
  s ← x[first(s)];                     {pop stack to old page}
if s = f then                          {did we catch up to the free area?}
  begin s ← x[first(f)];               {resume old page}
  x[first(f)] ← x[first(s)];           {the present page goes on the stack}
  x[first(s)] ← f;                     {so that we can finish it later}
  end;
```

Your task is to devise some way to test how well the idea works.

Notes for Tuesday, November 9

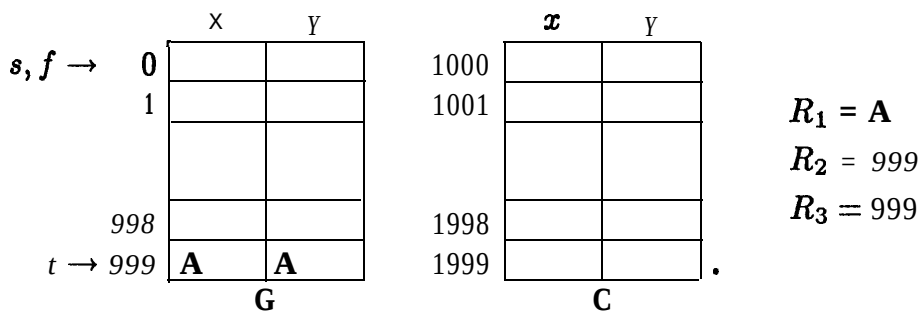
Last time, CGG had made the conjecture that the ordering used by the algorithm gives the nodes in order of decreasing depth in the tree. DEK announced that this is not always true, but the algorithm still works because if we consider only the head of each class, then these nodes are in decreasing order of depth.

DEK introduced Baker's algorithm with a long example. He explained that a List spelled with a capital "L" differs from an ordinary list in that the structure may be complex and it can even point to itself.

In the example we take N = 1000 and $r = 3$ and start with the following instructions:

$$R_3 \leftarrow \textbf{new};$$
$$R_2 \leftarrow R_3;$$

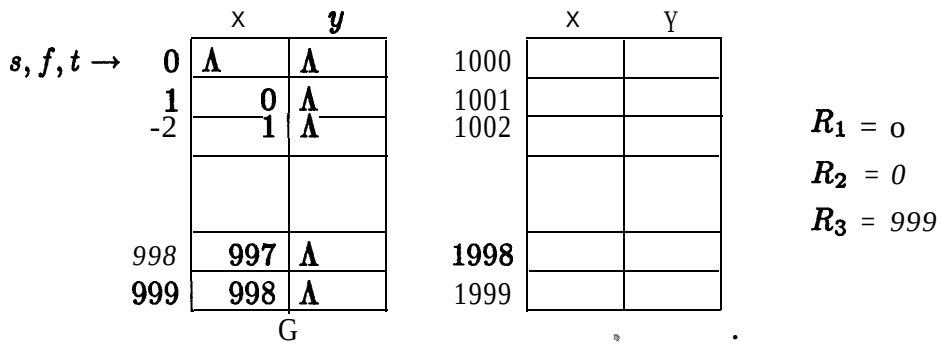
The registers originally contain the null pointer A and the **new** operation decrements t and returns the new cell that t points to. The configuration of memory is now



Next, we will repeat the following loop many times:

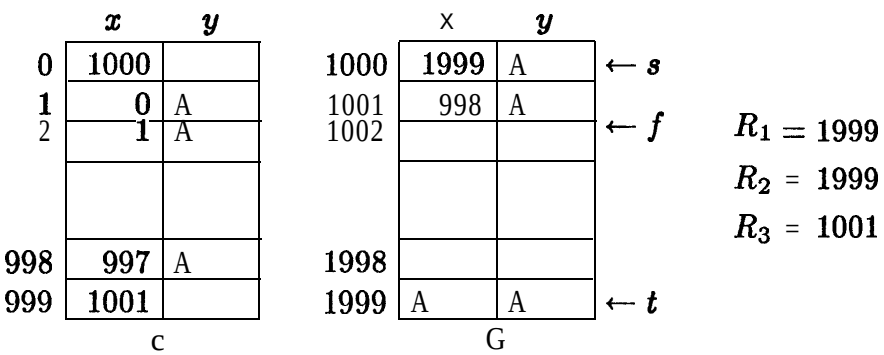
$$R_1 \leftarrow \text{new};$$
$$x[R_2] \leftarrow R_1;$$
$$R_2 \leftarrow R_1;$$

To execute the new operation, we see that $f \neq t$ *so we* call ***certify.*** Since $s = f$, we fall right through ***certify*** and t is decremented and the cell it points to becomes the new cell. The next two instructions point the x field of the previous cell at the new R_1 and get ready to repeat the cycle. Thus, the loop builds a linked list in locations 999, 998, 997, etc. The pointer t moves along with the tail of the list, and after 999 times through the loop we reach the following configuration:

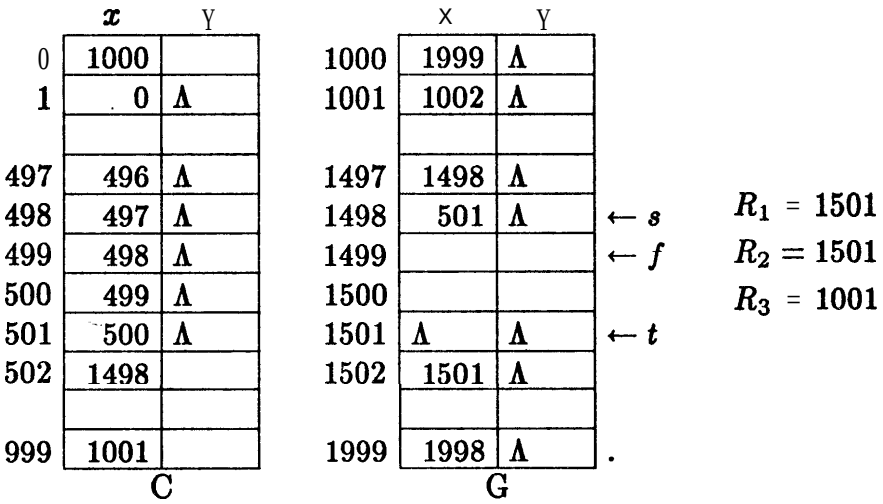


Something interesting happens when the next new instruction is executed. Since $f = t$, we avoid the call to ***certify*** and instead, we switch the good and condemned regions. If s were less than f, we would have to give up at this point because it would turn out that there might be some good data in the condemned region that could not be overwritten. Now, however, s and f are set to 1000 and t is set to 1999 indicating that the newly saved region is entirely free.

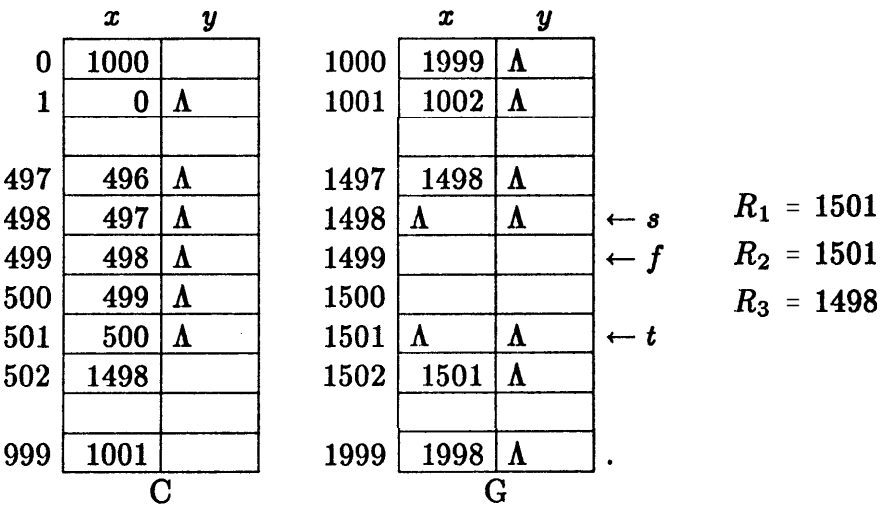
Since the invariant specifies that all the registers must point to the good region, we apply cleanse to them all. First we put the contents of cell 0 in cell 1000, make $x[0]$ point back to 1000, increment f and set R_2 to 1000. Similarly, cleansing R_3 puts the contents of cell 999 in location 1001 and points R_3 there. At this point, the invariant is restored and R_1 is assigned the new location 1999. The next two instructions are straight forward and at the end of the 1000th time through the loop, memory looks like this:



The next **new** instruction certifies location 1000, but this does nothing since that location contains no pointers to the condemned area. The new cell in the linked **list** becomes location 1998, and $x[1999]$ is pointed there. Subsequent cycles through the loop continue in locations 1997, 1996, etc. The s and f pointers increase together as one new cell is certified each time through the loop. These ***certify*** operations move the contents of locations 998, 997, etc. into locations 1002, 1003, etc. Thus, after the 1498th time through the loop, we reach the configuration



Catastrophe can be averted if we now stop building the list and carefully chop it off at exactly the right place. Suppose we now do 497 repetitions of the instruction $R_3 \leftarrow x[R_3]$, and then chop off the list with the instruction $x[R_3] \leftarrow \Lambda$. The resulting memory configuration **is**



If the next instruction is a call to **new,** the pointer f need not be increased by ***certify,*** since $x[s]$ is no longer condemned. This allows s to catch up with f and since the ***certify*** procedure does not do anything when $s = f$, neither of these pointers can change until the next memory flip. There will be no overflow, at least until after the memory flip.

However, the situation would be drastically changed if, after the 1498th time through the loop, the instructions

$R_3 \leftarrow$ **A;**
$R_2 \leftarrow$ A;
$R_1 \leftarrow$ **new;**
$R_1 \leftarrow$ A;
$R_1 \leftarrow$ **new;**

are executed. It would be hard to imagine actually wanting to clobber the registers this way, but the point is that even though no cells are accessible, the two **new** instructions are enough to cause overflow. The first **new** instruction causes s and f to be incremented by ***certify,*** and t is decremented to create the new cell. This makes $f = t$ and $s < f$, *so* that overflow occurs at the last **new** instruction.

The point is that, although it is possible to have up to about **1500** cells in use, overflow can subsequently occur when there are no cells in use at all.

DEK showed that overflow can be avoided if there are always less than N/2 cells accessible. (Actually,- we only require this at the time of a memory flip.) Suppose overflow has occurred and let n be the number of nodes that were accessible at the last memory flip. Then $f \leq$ Go + n because f is only increased by the move operation which is used for copying condemned cells that were accessible before the flip. We already saw that there can be no overflow when $s = f$. If a **new** operation is done when $s < f$, then s is incremented and t is decremented so $s + t$ says constant. Initially, s = Go and t = Go + N **– 1 so** the invariant is

$$(a - G_0) + (t - G_0) = N - 1.$$

Since overflow occurs when $f = t$, we know that $t -$ Go $\leq$ n. If n < N/2 we **have**

$$s - G_0 \geq N - 1 - n \geq n \geq f - G_0.$$

This is a contradiction because f and s are never allowed to cross each other.

Notes for Thursday, November 11

DEK brought up the question of how to show whether the modification of Baker's algorithm is beneficial in a virtual memory environment. For instance, should this algorithm be incorporated into a LISP machine?

There were many ideas about how to test the algorithm. RB suggested putting it into a LISP compiler or an operating system. This appeared rather difficult so SJR suggested simulating users working on different kinds of problems. DS suggested comparing the algorithm to other memory management algorithms.

DEK asked if anyone had access to a LISP compiler and could get data for the class. PDK said that he was involved in a project running LISP on the DOLPHIN computers and that he might be able to get some data.

DEK remarked that he had seen a study showing that the List structures actually used by a LISP program tend to be very simple. It might turn out that it is best to put lists in arrays and only use pointers when the structure is more complex. Empirical studies of FORTRAN programs have shown that expressions also tend to be very simple.

PDK suggested getting data on just the memory references made by LISP programs and applying the modified Baker's algorithm to that. WDW suggested that this could be done by writing a simple LISP interpreter and adding code to collect statistics. Such an interpreter would be fairly easy to write in LISP and SJR said it could be done in only about two pages of code.

SCS suggested using a lower level language like PASCAL because it may be easier to simulate paging that way and some people do not have much experience with LISP.

DEK explored ways of simplifying an interpreter. It might be sufficient to deal only with a fairly small subset of LISP. It would be easier to modify an existing interpreter than to write a new one from scratch. It might also be possible to use an existing version of LISP and just redefine the primitives to maintain some statistics in addition to doing their normal tasks.

The idea would be to produce a file describing all the memory accesses. Either the file could be a list of fetch, store, and allocate instructions, or another program could be used to read the file and produce such a list.

DEK pointed out that it would be necessary to figure out what the registers $R_1, \ldots, R_r$ correspond to in an actual LISP program. CGG said that there are various lists and arrays that are used by the system and are always accessible. DEK also mentioned that LISP atoms should probably correspond to the null pointers used by the algorithm.

PKR pointed out that in order to show that the algorithm is worthwhile, we must find real-time applications that use complex data structures. DEK admitted that it may be a little silly to use a real-time algorithm on a virtual memory system with page faults. However, there are still some benefits, especially when compared to having to garbage collect a large virtual memory.

DEK explained that most LISP systems maintain a free list similar to what was described last Thursday. When the list becomes empty, the system performs garbage collection by going through the memory and marking all the accessible cells. Any remaining cells are then put back into the free list.

PDK mentioned that some LISP systems can avoid pausing for garbage collection by doing this while waiting for the user to type something. However, CGG came up with an application where complex data structures are used, real time is important, and there is no need to wait for the user to type anything: robotics with vision.

WDW brought up the question again of whether it would be feasible to use PASCAL instead of LISP. DEK said that this might work, but most PASCAL programs do not maintain the kind of List structure for which the algorithm was designed. CGG said that LISP is more appropriate because the model described in the problem is very much a LISP style architecture.

RB said that the algorithm is only beneficial when the List structure contains cycles; since otherwise it would be possible to use reference counts to determine when a cell is free. CGG responded that it would be necessary to prepare for cycles since we are considering

the design of a system and it is expected to work in all cases.

Unless we can get data from actual LISP programs, it will be necessary to write test our own test programs. DEK suggested that there might be appropriate data structures in a graph algorithm such as Tarjan's algorithm for finding strongly connected components.

SCS pointed out that it would be desirable to have an application that uses large Lists because there could be a scale problem. Memory references are not likely to cluster on pages if the pages have to be very small. DEK said that there would be a trade off between the expense of having to deal with a large problem and any inaccuracies incurred by small page size. It would be possible to investigate how the effectiveness of the algorithm varies with changing page size and number of pages.

DEK suggest trying the interpreter on just a few key programs. He asked if a richer example could be achieved by having the interpreter interpret itself.

DEK also mentioned that it would also be possible to write the interpreter in **PASCAL** and have the subroutines for LISP primitives write fetch, allocate, and store instructions. The **PASCAL** routines could also just implement Baker's algorithm directly.

Even if nothing else can be achieved, DEK said that this problem should give a good idea of why there is so little good testing literature.

Notes for Tuesday, November 16

DEK began by soliciting ideas on the problem. CGG said that the only really good test for the proposed algorithm would be to implement it on a virtual machine. DEK suggested that the work done in class on this problem could be used to validate the algorithm sufficiently to justify such a large scale test.

CGG raised the question of what data structures should be used for testing the algorithm. It seems to be well suited to handling long linked lists, but a binary tree would be more difficult. DEK pointed out that a binary tree is inherently hard to deal with in virtual memory. Suppose for example, that three nodes can fit on a page. Then the best arrangement would probably be to have each page contain a parent and its two children. This would be roughly equivalent to having four way branching with one node on each page.

EJB/NMP/DS/SCS said that they were considering a simple model of computation where there *are* a set of *stack registers* and a set of *cursor registers.* The stacks are linked lists starting from the stack registers, and the cursor registers can point anywhere in any of the stacks. There are commands to push a new node or a register onto a stack, move a cursor by following either an x link or a y link, set a stack register from a cursor, or pop a stack. These commands can be implemented as follows:

push new(i)

$R_0 \leftarrow$ **new;**
$y[R_0] \leftarrow R_i;$
$R_i \leftarrow R_0;$

push register (i, j)

$R_0 \leftarrow$ **new;**
$y[R_0] \leftarrow R_i;$
$x[R_0] \leftarrow R_j;$
$R_i \leftarrow R_0;$

x move (j)

$R_j \leftarrow x[R_j];$

y move (j)

$R_j \leftarrow y[R_j];$

set stack pointer (i, j)

$R_i \leftarrow R_j;$

pop (i)

$R_i \leftarrow y[R_i];$

It would be possible to implement a subset of LISP using these primitives, but EJB said that they hoped to get by with just assigning probabilities to each command. NMP suggested dividing the program into three stages with different probabilities in each. There would be more *push* commands in the first one and more pop commands in the last.

DEK remarked that it is not possible to create circular Lists with these primitives. Circularity is one of the features that distinguish Lists from lists so it may be desirable to have it. This could be done with an x ***replace*** primitive very much like **rplaca** in LISP. The implementation would be $x[R_j] \leftarrow R_k$ where j and *k* are both cursor registers.

One of the hardest things in the analysis of algorithms is choosing a model that allows for a meaningful average case analysis. DEK said that he had never seen a good model of LISP programs so it would be interesting to see how well this one works.

Some problems such as sorting and hashing are naturally described by good models based on all possibilities being equally likely, but DEK explained that this is not the case for paging algorithms. A popular model has been the *independent reference model* which is based on the unrealistic assumption that each page has a fixed probability of being accessed by each memory reference. In **1976,** Forest Baskett and Abbas Rafii showed that the model does do a good job of predicting the frequency of page faults if the probabilities are interpreted differently. Perhaps a similar method could be used to show that our proposed model of LISP is reasonable if the probabilities are tuned properly.

EJB said he had hoped that the results would not be strongly dependent on the probabilities. This way, a clear result could be obtained without knowing how to tune them.

RB said that he and MAH were also considering implementing basic operations and assigning probabilities to each. They also considered omitting register to register assignments since these do not directly cause page faults, but DEK said that this sounded like a bad idea because future page faults would be affected.

DEK asked what memory size should be used in the simulation. MAH remarked that real programs could use widely varying amounts of memory. DEK added that the relative probabilities of *push* and *pop* are very important and CMA suggested that the memory size might depend on other probabilities in the model, although this would be a minor effect.

DEK suggested using N = **5000** and a page size of 50 with **10** pages in core. He said that for testing sorting algorithms, he had found that most of the significant effects could be observed with less than **1000** elements. The only exception was Shellsort where very large n are needed to determine whether the running time is of the form $an \log^2 n + bn \log n$ or of the form an^b with $b \approx 1.27$.

WRB pointed out that it may be desirable to gather more statistics than just the number of page faults. He suggested keeping track of the number of pointers on each page that refer to a different page and CGG extended this idea by suggesting maintaining a table giving for each pair of pages, the number of pointers from one page to the other. These data could be very valuable, but it would be very difficult to analyze them properly.

DEK closed with a few general remarks about the problem. While the general problem of optimum memory allocation is NP-complete, there is reason to hope that the modified Baker's algorithm will behave well. This will be particularly important in the future with the proliferation of memory hierarchies.

Solutions for problem 4

Most groups tested the algorithm on a sequence of randomly generated instructions based on the primitives discussed on November 16 and found that the modification to Baker's algorithm had little effect on the frequency of page faults. Despite very extensive testing, it was not possible to cover the full range of possibilities because of the large number of adjustable parameters. Only one group found differences of more than about 20% between the numbers of page faults for the original and the modified algorithm, and the modification often performed worse than the original. Everybody used the least *recently* used page replacement strategy.

Three groups generated random sequences of *push new, push register, pop, x move, y move, set stack pointer, x replace,* and y replace commands. WRB/CGG/SJR divided the simulation into two parts and gave a higher probability to commands that generate *allocate* instructions in the first part and EJB/NMP/DS/SCS did essentially the same thing except they used three parts. They carefully chose the probabilities so that there would not be more *store* instructions than *fetch* instructions as would be the case if all commands were given equal probability.

MAH/RB used a completely different strategy for generating a sequence of instructions. They took one of their programs for problem 3 and broke in into 17 blocks, each of which was hand translated into a roughly equivalent sequence of *fetch, store,* and *allocate* instructions. They modified the program to produce a sequence of code block numbers and used the translations to generate the actual instructions. This approach does seem to yield significantly different results from the random generation approach used by other groups. Data from the actual program produced about half as many page faults as a randomized list of block numbers.

WRB/CGG/SJR had a feature that allowed a single simulation to consider a range of possible choices for the number of pages in core. They kept track of the pages in order of last use and maintained a table of the number of page faults so far as a function of the number of pages allowed in core.

EJB/NMP/DS/SCS ran the original algorithm and the modification in parallel. This reduced overhead and was more convenient than doing completely separate simulations as other groups did.

One possible explanation for the ineffectiveness of the modification is that it may have been implemented incorrectly. SRE/EJK/PKR conjectured that the modification to *certify* may be referring to the wrong page when it says $s \leftarrow x[first(a)]$. They suggested using $x[first(s - 1)]$ or equivalently just $x[s - m]$. WRB/CGG/SJR actually used this corrected version of the modification, although they gave no explanation as to why they chose to diverge from the problem handout. In spite of this change, their results were roughly consistent with those of the other groups.

WRB/CGG/SJR also investigated another modification to Baker's algorithm. Their modification is "when a node is moved to a page, make sure that the next certification is done on that page". In contrast to this, the modification in the problem handout was

"... when the first node is being moved to a new page, . . . certify that entire page first." Unfortunately, neither of these modifications seemed to have a significant effect on the number of page faults.

The only group to observe significant benefits for the modified version of the algorithm was SRE/EJK/PKR. They used a random sequence of the eight primitives and a page size of 100. For various values of the probabilities for each primitive, the length of the simulation, the total number of pages, and the number resident in core, they found that the modification reduced the number of page faults by ratios ranging from **1** to about 17. The number of page faults often varied wildly for very small changes in the parameters.

Problem 5, due December 7: Microbe **computers.**

This problem concerns PAC-mites, the tiny residents of a two-dimensional rasterized universe called PAC-land.

The laws of physics/chemistry/biology/everything in PAC-land are fairly simple, **and** they can be described precisely as follows:

(1) PAC-land consists entirely of discrete sites at positions m + ni of the complex plane, where m and n are arbitrary integers.

(2) Each site is occupied by either a barrier (denoted ❑), by a blob (denoted ∘), or by a *PAC-mite* (denoted ᗧ or a variant of this symbol).

(3) A PAC-mite can exist in any of four directions ᗧ, ᗢ, Ɔ, ᗣ, which we shall call 0, 1, 2, 3; and it can have either 0, 1, 2, or 3 teeth, independent of its direction. Thus, there are sixteen possible states; those of direction 0 having various numbers of teeth are denoted by ᗧ, ᗧ, ᗧ, and ᗧ; those of direction 1 are denoted by ᗢ ᗢ ᗢ ᗢ; and we have, similarly, Ɔ Ɔ Ɔ Ɔ in direction 2, and ᗣ ᗣ ᗣ ᗣ in direction 3. It is convenient to allow the direction to be any integer and then to reduce it modulo 4, so that, e.g., direction 13 is the same as direction 1.

(4) A PAC-mite of direction d in site s = m + ni is said to be aiming at site $s + i^d$. Up to four PAC-mites can therefore be aiming simultaneously at the same site.

(5) Mites with more teeth have a more powerful bite; we say that a PAC-mite attacks site s if it aims at s and if no PAC-mite with more teeth also aims at a.

(6) Barriers stay fixed, but blobs and PAC-mites change deterministically in discrete units of time; the state of the universe at time $t + 1$ is a function of its state at time t. Moreover, this transition function is local: each site s changes in a manner that depends only on its current contents and on the PAC-mites that attack it, together with the contents of the site aimed at by its current PAC-mite, if any. (A complete definition of the transition function appears below.)

(7) The number of teeth in a PAC-mite corresponds to its age: Toothless ones have just been born, but a triple-toothed mite that exists at time t arrived at time $t - 3$.

(8) PAC-mites belong to different species, where each species is defined by a sequence of fifty "genes," denoted by the symbols u_l, v_{el}, w_k, x_k, y_{ek}, z_{ek}, for $0 \le k \le 2$ and $0 \le e, l \le 3$. Each gene is either 0, 1, 2, or 3; hence there are 4^{50} possible **species.**

(9) The genes determine the exact laws of succession, as follows:

(a) A site containing a barrier at time t contains a barrier at time $t + 1$.

(b) A site s containing a blob at time t contains a blob also at time $t + 1$, unless there is a unique PAC-mite attacking s at time t. In the latter case the site will contain a toothless PAC-mite of the same species in direction e + u_l, where the attacker has direction e and has l teeth. Thus, the attacker's u_l gene specifies the orientation of the new-born mite that consumes the blob.

(c) A site s containing a k-toothed PAC-mite of direction d at time t will normally contain a $(k+1)$-toothed PAC-mite of the same species at time $t+1$, unless $k = 3$, when it normally reverts to a blob. If $k < 3$, the direction of the $(k + 1)$-toothed mite will be

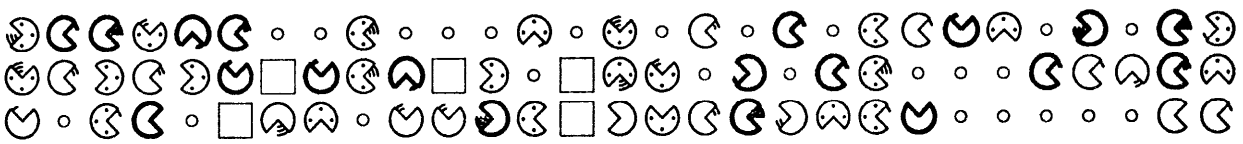
$d + w_k$, if the original mite aimed at a barrier;
$d + x_k$, if the original mite aimed at a **blob;**
$d + y_{ek}$, if the original mite aimed at a PAC-mite of the same species and of direction d + e;
$d + z_{ek}$, if the original mite aimed at a PAC-mite of a different species and of direction d + e.

(d) However, the normal aging process described in (c) is overridden if at least one PAC-mite of a different species is attacking site s. If there are two or more attackers, with at least one of a different species, site s will revert to **a blob at** time t + **1.** If there is a unique attacker, and if it belongs to a different species, suppose it has l teeth and direction $d+e$; then site s will hold a toothless **PAC-mite** of direction d + e + v_{el} at time t + 1, having the attacker's species.

These formal rules boil down to a few intuitive principles: A PAC-mite aims **at a neighboring** site, and it can see whether that site contains a barrier, a blob, or another **mite; in** the latter case, it can also perceive the direction of the neighboring mite, and it can tell whether its neighbor is friend or foe. The mite then grows another tooth, and changes **direction,** based on all this information. Changes in direction are relative; if all of PAC-land were rotated by 90°, its entire behavior would rotate by 90°. Furthermore **a PAC-mite** tries to propagate a new mite of the same species in the aimed-at site; the direction of this new mite can depend on everything the old mite knows. However, the new mite will be born only if the old mite is a unique attacker, and only when there is a change of species at the site of birth. When different attackers contend for the same site, the competition **is** so fierce that nothing but a blob remains.

Examples: (i) PAC-mites *A* and B (and no others) aim at site s, but *A* **has more** teeth. If site s contains a blob or a PAC-mite of some species different from *A*, **a new** mite of *A*'s species will be born at s. Simultaneously, *A* and B change direction and grow another tooth; or *A* dies, if it already has three teeth; or they are consumed **by some mites** attacking them. (ii) PAC-mites *A*, B, C, D all aim at site s; *A* and B have three teeth, but C and D have 2 or fewer. Then site s will not experience a birth at the next moment of time; furthermore, it will revert to a blob unless it contains a PAC-mite E of the same species as both *A* and B, where E has fewer than 3 teeth. (iii) Two adjacent **PAC-mites** *A* and B of different species, attacking each other, might cause new clones b and a to occupy the respective sites of *A* and B; the newborns might be attacking each other too.

Now you know everything about PAC-land except what you were afraid to **ask; namely,** what are you supposed to do for Problem 5? The goal is to come up with the design of a PAC-mite species that will be the most fruitful when placed in the PAC-land universe together with species designed by other members of the class. (Precise rules of the competition will be worked out by mutual agreement later. All you are supposed to do is come up with a winning sequence of 50 genes. It would also be nice to have programs that display the final struggle so that we can all watch.)



Notes for Thursday, November 18

DEK started with a few general remarks about problem 5. The problem of designing hardy PAC-mites requires understanding the global properties induced by very low-level definitions. Even if biologists knew the entire genetic code for human beings, the code would not give them a very good idea of how people behave. Similarly, there is no simple relationship between the genes of a PAC-mite and the global properties of that species.

There is reason to believe that it might be possible to construct a universal Turing machine from a family of cooperating PAC-mites. This would show that PAC-mites can perform any computation that computers are capable of. One can even speculate on whether a group of PAC-mites could ever be conscious. The book The Mind's I by Daniel Dennet and Doug Hofstadter deals with consciousness and DEK said that it was one of the things that inspired the problem.

It has proven possible to construct a Turing machine in Conway's game of Life and PAC-land is substantially more complicated than that game. The game of Life is played square grid where the cells have exactly two possible states: empty and occupied. A cell is occupied at the next generation if either it is occupied originally and two or three of its eight neighbors are occupied, or it was originally empty and it has exactly three neighbors. The Turing machine construction is based on "puffer trains" which regenerate themselves in a slightly different location every few generations, sending other Life forms off in a different direction.

The final competition in this problem will be a series of duels on a 9 by 19 board surrounded by barriers starting with one toothless mite of each species. One contestant will start at coordinates $(5,5)$ pointing to the right, and the other contestant will start at $(15,5)$ pointing left. Thus, the mites face each other at a distance of ten paces.

DEK suggested solving a few simple questions to get started and he posed the following problem: How long does it take for the contents of PAC-cell (m, n) to affect what happens at location (m', n')? As SJR pointed out, this takes at least $|m-m'|+|n-n'|$ generations. This effect can be thought of as due to the speed of light in PAC-land.

RB suggested to trying design species in the absence of competition that grow into regions of various shapes. A wavefront advancing at the speed of light would produce a diamond shape. SJR said that we might apply Huyghen's principle of optics which says that the future position of an advancing wavefront is the same as the envelope of a set of wavefronts originating from all points on the original wavefront.

DS suggested trying to design a species that annihilates itself. If all the genes are 0, then we obtain a line such as the one below that moves to the right one cell per a generation.

As SRE pointed out, this line will go until it hits a barrier and gets destroyed.

RB suggested that it might be possible to avoid having to depend on running into a barrier if we can wind up with two mites of the same age facing each other. No new mites

will be born as long as both mites are alive, since no empty cells are attacked. If one mite is older than the other, then it will die of old age first and a new mite will be born.

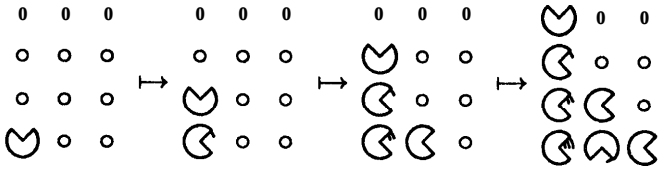
A species can be designed by playing through a few generations and deciding on the genes as they are needed to determine the directions for each new generation. The class designed a species that annihilates itself in the following manner:

On the first generation, the decisions were $u_0 = 3$ and $x_0 = 3$; on the second generation it had to be decided that $x_1 = 0$ and $u_1 = 1$ and similar decisions followed for the other generations.

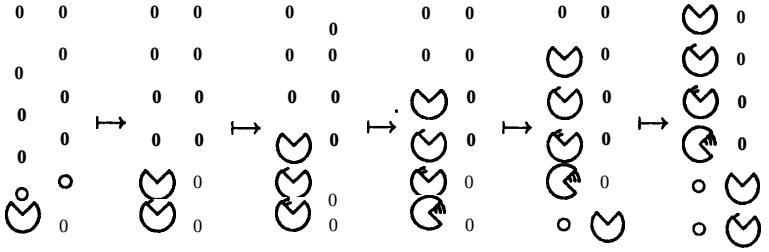
RB proposed a variation on the second generation from which exactly the same type of annihilation proved possible. The original annihilation scheme used ten genes and RB's variation used a different set of ten genes with eight of them in common. This shows that there is a probability of at least $2 \cdot 4^{-10} - 4^{-12}$ that a random species will annihilate itself.

DEK suggested a kind of "gene assembler" that would take pictorial sequences such as the one above and find the actual genes. This might be a nice way to work out new species.

SJR suggested the next exercise: construct a puffer train for PAC-land. His idea was to send out puffs that move along like the line of four mites shown above and have some of the mites turn aside. The first attempt at this was something like the following:



This did not seem to be working very well because, as WDW pointed out, the mites were turning to the side too soon. A more controlled approach results in



which works fairly well. There will be a series of puffs of four PAC-mites lined up diagonally adjacent to each other. The puffs move upward and the "train" moves to the right, except there is no train.

EJK suggested the last exercise: come up with a stable pattern that regenerates itself somehow. DEK explained that in the absence of any competition, all cycles must have a length of at least five since a PAC-mite lives four generations and its location must be unoccupied for at least one turn before a new mite can be reborn. This also shows that PAC-land can be at most $\frac{4}{5}$ full on the average.

DEK said that he had accidentally found a cycle of length five involving about 25 mites. He used a species that had been designed to self-destruct, but he started with more than one mite. He had not found any cycles of longer period, although it may even be possible to have even length cycles. For a cycle of length six, each cell would have to remain empty for two generations instead of just one.

Notes for Tuesday, November 23

DEK explained that for strategic reasons, class discussions would be limited so as not to give away too many sectrets. The current topic was how to design a fast program to simulate PAC-land.

SJR brought up parallel processing and DEK pointed out that the problem seems to be well suited to parallel hardware because it can be done with only local communication. A VLSI chip could be built with one processor for each cell in PAC-land and each processor would only have to communicate with its four neighbors.

There is one type of parallelism that applicable to the problem at hand. On a 36-bit machine like SCORE, parallel boolean operations can effectively do 36 computations at once. DEK said that this strategy has been very effective in Conway's game of Life where one bit can represent the state of a cell and the rules of succession can be written down in terms of a boolean expression.

This would not work quite as well for PAC-land where it takes more than one bit to represent a cell. If there are two different species, we need to be able to represent each species in four different ages and four different directions. With the blob and the barrier this makes 34 possibilities, so that six bits are required to represent each cell and only six cells can fit in a word.

The discussion turned to methods for accessing neighboring cells. As DEK pointed out, this is not hard to do with two dimensional, sequentially allocated arrays. SCS added that it would probably not be wise to try to take advantage of large regions containing only blobs, since PAC-land fills up rapidly and then the extra code would only add more overhead. Overhead could be further reduced by adding an extra border around the array.

CGG mentioned that for wrap-around mode, it would be ideal to use modular arithmetic. Ordinarily, the neighbors of a cell a are cells $a - N$, $a - 1$, $a + 1$, and $a + N$. For wrap-around mode, however, the right-hand neighbor of cell b on the right edge of the array is cell $b + 1 - N$ instead of cell $b + 1$. DEK pointed out that this could be done easily if N were a power of two.

To obtain the right-hand neighbor of a particular cell we could then just add one to the right-most six bits and avoid carrying into the next bit. The "brute force" way to do this would be to define

$$x' = (x \wedge \overline{m}) \text{ v } ((x + 1) \text{ A m})$$

where m is a mask having ones in the last six bits. A slightly better way to do this is

$$x' = (((x + 1) \oplus x) \text{ Am}) \oplus x.$$

(Here ⊕ denotes exclusive or.) KAH suggested yet another way, which would be good for a machine with "bit-set" and "bit-clear" primitives. It is based on a mask $b = 2^5$.

$$x' = (b \text{ A } x) \oplus ((\bar{b} \wedge x) + 1)$$

DEK mentioned that one way to get around the problem would be to store the address of cell (i, j) as Ni in one register and j in another. It would be faster, however, to just have four arrays of pointers giving the locations of the four neighbors.

PKR suggested another way to represent PAC-land that allows for boolean operations on words. Assuming again that there are two different species, he suggested having an array of bits for each of the. 34 possible ways to fill a PAC-cell. A bit in a particular array would be on if and only if the corresponding cell is occupied in the appropriate manner for that array. DEK suggested the refinement that there should be one array for each of the four ages, another four arrays for the directions, and another two for the species. Each of the previous arrays is just the bitwise and of at most three of the new arrays and the new arrays appear to be better suited to describing the rules of PAC-land.

SJR said that it would be better to first figure out how to decide which genes to use, since this should effect the choice of representation. DEK suggested starting out by considering how to recognize a unique attacker. As MAH pointed out, the obvious strategy is to scan the array sequentially and consider the four neighbors of each cell. DEK explained that this is inefficient, since it consideres each attacker four times. It would be faster to scan the attackers ahead of time, making a table that gives the attacker for each cell.

CGG said that it is sufficient to keep track of the age of the oldest attacker and the unique attacker if there is one. DEK suggested referring to the attackers found so far as timers, since younger mites are not counted as attackers if there is an older mite also aiming at the same cell. CGG summarized that the table entries contain the age of the aimer, a pointer to the first aimer, a bit indicating whether there are any other aimers, and a bit indicating whether any of them are from a different species.

DEK mentioned that it would be nice to be able to scan the aimers in order of their number of teeth. This would simplify the updating process since older aimers automatically supersede younger ones, but it did not seem worthwhile.

Suppose there is a main array *A* and an array B of aimers whose entries are records for the form *(k, d,s,* b_1, b_2) where *k, d,* and s are the age, direction and species of the first aimer and b_1 and b_2 are the bits that CGG refered to. Suppose further that we are considering cell (m, n) of the *A* array and it contains a PAC-mite of species s, direction *d,* and age *k.* DEK presented the following routine for updating *B.*

```
(m', n) ← (m, n) + i^d;
if B[m', n'] is empty or B[m', n'].age < k
  then B[m', n'] ← (k, d, s, 0, 0)
  else if B[m', n'].age = k
    then begin
      B[m', n].b1 ← 1;
      B[m', n'].b2 ← (s = B[m', n'].species);
    end;
```

Later on, non-empty entries in B can be used to determine which genes to use for updating A.

Notes for Tuesday, November 30

Nobody had any "safe" topics for class discussion, so DEK posed the problem of designing a species that can explore an arbitrary maze. Starting from a single PAC-mite, the species is supposed to eventually occupy the entire connected region of its point of origin. The region can be any contiguous set of cells surrounded by barriers.

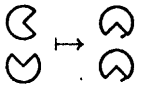
SJR suggested that each PAC-mite should turn in all four directions during its lifetime and DEK agreed that there are cases where this is certainly necessary. If the original mite is surrounded on three sides by barriers, for instance, then the species will become extinct immediately unless all four directions are tried.

SJR added that it might also be sufficient just to have each mite always try all four directions. **As** SCS pointed out, the only problem is making sure that no two mites ever attack the same cell, since the rules specify that no new mite is born in this case. SJR pointed out that the problem can be avoided if all the mites turn in unison and WDW completed the proof as follows. First, suppose some cell in the connected region is never occupied. Consider the closest such cell to the starting position. There must be an adjacent cell which at some time is occupied by a new mite. Since this mite turns in all four directions, a new mite would have to be born in the supposedly unoccupied cell.

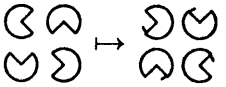
DEK pointed out that the solution uses only the following genes:

$$u_0, \ldots, u_3, x_0, x_1, x_2, y_{00}, y_{01}, y_{02}, w_0, w_1, w_2.$$

If we set all these genes to 3, then all the mites will rotate clockwise in unison. DEK asked if it is possible to choose the other 37 genes so that the species can synchronize itself when some of the mites are placed out of step. WDW suggested that if a mite sees another member of its species pointed in a different direction, it could turn to match.



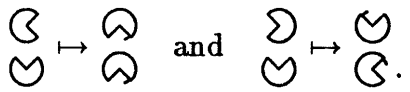
DEK pointed out that this would not work in the case



but as SJR explained that synchronization is impossible here because it would violate the 90° symmetry of PAC-land. If the universe is identical to some rotation of itself, it must remain so in subsequent generations.

The problem is easier if all the mites that are out of step are pointing in the same direction. Suppose the mites can be divided into two sets A and B, where all the members of set A are one turn ahead of the members of set B. DEK explained that it is possible for the members of the two sets to tell each other apart. The genes can specify that mites

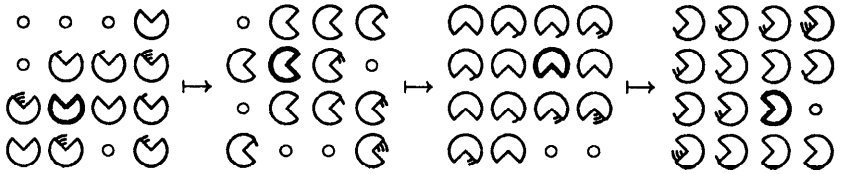
in *B* turn to match *A* when they see a member of that set, but mites in *A* continue to turn as usual when they see members of B. That is, we can have things like



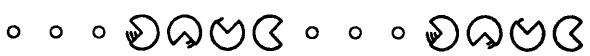
This has the effect of making *B* mites into *A* mites as soon as they see an *A* mite. Eventually, all the *B* mites will become *A* mites and if we had a non-deterministic universe, we could with probability 1, achieve synchronization from arbitrary initial configurations.

WDW said that many of the groups had already discovered that the **maze-solving** species is very powerful. When all of the genes are 3's (or when they are all 1's), they all point the same way and they do not waste many bites.

DEK said he had found that as long as four genes are set correctly, any species can survive indefinitely against the all 3's species. The single mite shown below in bold can hold out indefinitely against the all 3's species since the bold mite is always the unique attacker of the cell in front of it.



SCS said that he had found configurations where the whole board repeats itself in cycles of lengths six, seven, and eight. This was especially interesting since DEK had speculated that cycles of lengths not divisible by five would be rare. WDW said that one cycle of length seven involved rows of the form



moving across the board.

SJR brought up the idea of evolution. Every four generations a randomizer could be used to introduce mutations and after a long time, the best species would prevail. Another option would be to mix genes from pairs of species in a way somewhat related to **sexual** reproduction. SCS said that biologists have discovered high level genes that turn on groups of otherwise inactive genes. These higher level genes may be able to control useful characteristics such as the number of fingers. Something like this may be necessary in PAC-land because otherwise, a beneficial mutation would be too improbable.

DEK suggested that evolution is often too conservative. He once worked on an adaptive Tic-Tat-Toe program that basically just tried strategies and used the ones that had been most successful previously. The program was not capable of long range planning, so the most successful strategy tended to be a short term defense.

Two other possible strategies were raised at the end of class. DEK asked about ways to speed up evolution and CGG/EJE/KAH said they were developing a program that simulates PAC-land and allows genes to be left unspecified until they are needed.

Notes for Thursday, December 2

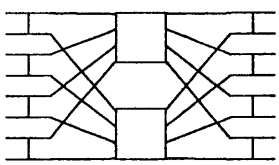
WRB started out by giving a graph of the relation a beats 6 for six different species that he and PKR were considering. No "winner" could be determined because the graph was highly cyclic. DEK said that if this happens in the final competition, the best that could be done would be to find the strongly connected components of the graph and then count indegree and outdegree.

DEK invited Vaughan Pratt to class to help lead the discussion about how to implement PAC-land in hardware. VRP said that the hardware view of a problem is to consider the communication paths first and the processors second. As RB pointed out, the problem seems to be well suited to hardware implementation because there can be one processor for each cell in PAC-land so that each processor only needs to communicate with its four neighbors.

WDW pointed out that there are two possible ways to communicate orientation information to neighboring processors: either send the absolute orientation or the relative orientation from each neighbor's point of view. CGG mentioned that the latter scheme is more natural, considering the rotational invariance of PAC-land, but as SCS pointed out, the first scheme requires only one absolute to relative conversion instead of four for each cell. VRP added that the saving is somewhat diminished because it is not possible to send the same information to each output when so much more has to be sent in the direction the PAC-mite is facing.

More information than just the direction has to be transferred between cells. RB mentioned the number of teeth and EJB added that genetic information would have to be passed when a new mite is born. WDW pointed out that the volume of this information could be reduced by only keeping short identifiers locally and looking up the actual genes from some external table.

This brought up the question of how to broadcast different information to many different destinations. VRP suggested the somewhat more abstract question of how to allow for connecting n inputs to n outputs in all n! possible ways. This can be done by divide and conquer, starting with a simple circuit with two inputs and two outputs that either swaps the inputs or passes them straight through. Using ⊐⊏ for this basic circuit, the construction is as follows:



This shows how an n input switching network can be built from two $\frac{n}{2}$ input switching networks. Here n = 8 and the two small boxes are 4 input switching networks that have been combined into a single 8 input network. In general, the network consists of $2 \lg n - 1$ rows of $\frac{n}{2}$ switches or about $n \lg n$ switches in all. Unfortunately, it takes O(n log n) time to set up the connection, so the scheme is not really applicable to PAC-land.

CGG suggested that if there are three different species, three clock cycles could be used to broadcast the genes one species at a time. VRP added that there is a trade-off between space and time: one clock cycle would suffice for 300 bit wide paths or 300 clock cycles would allow for one bit wide paths. The serial solution would allow for savings if not all the information is required every generation.

The alternative to broadcasting the genes would be to store them locally in each processor. SJR pointed out that the local solution would allow for plug-in extensibility and DEK added that it would also make it much easier to study mutations as SJR suggested last time. The local solution does not really require **100** wires to each neighbor: it would suffice to have k wires and l cycles for any k and l such that $kl = 100$.

DEK added that another saving could be achieved by just exchanging short species codes and having each processor look these up in its own 300 bit table. The 300 bit tables could be loaded serially in advance. VRP explained that this brings up a trade-off between processor costs and communication costs: broadcasting the 300 bits would eliminate the need for a big table in every processor. The choice depends on the relative cost of processors verses communication lines.

There was still some time left and the discussion turned to other PAC-land problems. VRP mentioned that it is probably undecidable whether a given configuration of PAC-land ever halts in some fashion, because PAC-land should be complicated enough to simulate a Turing machine. DEK added that it was also hard to solve the halting problem for Paterson's worms.

Notes for Tuesday, December 7

This was the day of the final competition. The five groups submitted their genes before class and the competition was held in the form of a series of duels using the pacwar program written by the T.A.

	u_l	v_{0l}	v_{1l}	v_{2l}	v_{3l}	w_k	x_k	$?/0k$	y_{1k}	y_{2k}	y_{3k}	z_{0k}	z_{1k}	z_{2k}	z_{3k}
WRB/PKR	3333	3333	3333	3333	3333	222	222	222	222	222	222	222	222	222	222
RB/SJR	0310	0010	0010	0020	0110	321	121	321	321	311	321	131	131	131	131
SRE/MAH/NMP/DS	**0310**	**0000**	**3333**	**2222**	**1111123**		**123**		**111111111111100**			**333**	**100**	**311**	
EJB/PDK/SCS, WDW	0000	0000	3333	2222	1111333	333	333	333	333	333	113	133	133	113	
CGG/EJK	**1111**	**1111**	**1111**	**3223**	**1111222**		**111111333**		**333**	**111333**		**333**	**333**	**333**	

Most of the class was spent playing duels between the five contenders. Each of the duels terminated in a finite time with a definite winner and surprisingly, the results were consistent with an ordering. The winner was WRB/PKR and RB/SJR, **SRE/MAH/NMP/DS**, **EJB/PDK/SCS/WDW**, and CGG/EJK came in second, third, forth, and fifth respectively. The following table gives the approximate length of each of the duels.

		1	**2**	**3**	**4**
WRB/PKR	1				
RB/SJR	2	200			
SRE/MAH/NMP/DS	3	228	175		
EJB/PDK/SCS/WDW	4	200	104	500	
CGG/EJK	5	295	*59*	*280*	*46*

After the duels, there was some time left for five way matches and the first of these was played on a 40 by 40 board with wrap around. Each species started with five mites in a regular pattern. RB said that his groups' species had been designed for a universe surrounded by barriers and would do poorly with wrap around. The results partially confirmed this: using the numbering from the above table, the species became extinct in the order 5, 2, 3, 4 with species number 1 outlasting the others.

The last two five way matches were played on **30** by 63 boards. The first of these **was** done in wrap around mode starting with one mite of each species roughly equally spaced. Again, the species of RB/SJR was the second one to be killed and species number **1** outlasted the others. The order of extinction was 5, 2, 4, 3.

For the final match, DEK suggested investigating the effect of partially surrounding each mite by barriers. Each of the four weaker species were surrounded this way and the whole board was also surrounded by barriers as in the duels. This time the game lasted quite a bit longer and the order of extinction was 5, 4, 2, 3 with species number 1 winning again.

Solutions for problem 5

Most groups attacked this problem by experimenting with the pacwar program which was written by the T.A. While the choice of some of the genes could be based on general strategy, the only effective way to choose an entire set of genes turned out to be by running pacwar and observing the results. Everybody maintained a small group of promising species and tested new species against those in the group, keeping the best ones. New species were selected either by modifying old ones or by exhaustively searching a small set of related species. The process involved luck and guess work.

It soon became apparent that the species where all the genes are l's or all are 3's are very powerful, and these became the species to beat. All groups came up with species that could beat one or both of these.

PKR/WRB started out by looking for fast growing species. They reasoned that the -pattern would tend to expand most rapidly when all the u genes are 0 so that newborn mites face the same direction as their parents. They also experimented with slower growing species whose babies turn 90" and to their surprise one of these appeared to be very powerful. This is the species that they eventually chose for the competition.

PKR/WRB found that their champion could beat the 3's species and could survive indefinitely against the l's, although the population gets reduced to four. They experimented with variations on the 3's species and eventually came up with one that could beat both the 3's and the 1's. The result was very similar to the species chosen by EJB/PDK/SCS/WDW, but it lost to the original champion although it beat the mirror image. (The mirror image is obtained by changing l's to 3's and visa versa.) PDK/WRB conjectured optimistically that any species that could beat their champion would lose to its mirror image.

RB/SJR also experimented with species that could beat the l's and the 3's. They modified the pacwar program so that it could play a long series of matches and just write the results in a file. They considered a large set of similar species and eventually chose one

that could beat the 1's and the 3's. They took advantage of the geometry of PAC-land by designing their species to grow only forward since the surrounding barriers prevent an attack from the rear.

SRE/MAH/NMP/DS tried variations on the 1's and the 3's, looking for something that could beat both of them. They considered modifications of the w genes, but had more success with varying the *z* genes instead. They chose the u, v, w, and x genes so that their species would propagate rapidly and attack forward, and tests showed that these choices appeared to be effective. They chose the x genes so that an isolated mite would turn in all four directions, turning backward last, and they chose the u genes so that all the babies would face forward except the one behind which would face backward. In the absence of competition, this produces a diamond shaped pattern that expands forward and backward at the speed of light. Finally, they chose the v genes so that a new mite born in place of an enemy would face the same way the enemy faced. The idea behind this is that the baby attacks the enemy's baby.

EJB/PDK/SCS/WDW also began by looking for something that could beat the 1's. After they expanded their search to include species that could distinguish friends, foes, and blobs, they obtained a promising species that could beat the 3's as well as the 1's. They considered a family of species based on this one but with different choices for the *z* genes. The u genes were all 0 and the w, x, and y genes were all 3. For the v genes, they also used strategy "our babies chase their babies". They said this weakened their front line somewhat, but tests indicated that it was worthwhile. They considered various choices for the *z* genes but made no effort to understand why some of them worked better than others. They played a small tournament among their favorites, but the results were inconclusive.

CGG/EJK did much more programming than any of the other groups did. Unfortunately, it took too much time and they speculated that they might have done better in the competition if they had spent more of their time experimenting instead. They developed a simulator that had many of the features of pacwar except that it allowed genes to be left unspecified until they were needed. This would be very useful for analyzing the effects of individual genes, however this type of analysis proved to be very difficult.

The simulator worked by building a linked list of genes that need to be specified, and for each such gene, having another linked list of locations where that gene applies. Each generation is a two pass process: the first pass builds the lists and sets the directions of the affected mites to "unspecified"; the second pass asks the user for the required genes and fixes the directions. Unfortunately the simulator turned out to be fairly slow; however, it provided the only independent confirmation of the correctness of the pacwar program.