ing can be selective (by specifying the names of predicates to be traced). Failing predicates result in a call to a debugger which allows exhibition of the runtime stack, and a generation of a new level of PDSS environment. In tracing, calls to built in predicates can be exhibited. We find this an advantage. There are, however, no facilities for single stepping, trap setting, etc. The lack of such debugging facilities is undoubtedly a disadvantage, particularly in a commercial environment.

Source Comments.

Statement by statement comments are not allowed. All comments must be attached to the definition of a predicate; comments attached to clauses are shifted up by the PDSS save command. We view this as rather "unfriendly" restriction. Code in a large Prolog program can become a somewhat difficult to read, and allowing comments in appropriate place improves enormously the readability and understandability of the code.

Documentation.

The quality of editorship of the manuals [1,2] is pretty poor. Misprints, mistakes and errors abound. Some incorrect examples can be found, as well as erroneous and confusing references. Beginning Prolog programmers beware!

Modularity.

Modularity is one of the redeeming features of this product. Control of interfaces between various modules is very good, allowing hiding of information when wanted, exporting and importing predicates as well as making names visible or not as the application dictates. This has good effects from software engineering point of view. Another advantage accruing from this approach is the improvement of database search efficiency, by selectively making only the relevant parts of the database visible from the executing module.

In production mode, modules can be compiled (pretranslated) separately and consolidated later into one program.

Procedural Components.

MProlog provides a number of predicates which introduce procedural components into the system. We see most of these predicates as useful in generation of efficient production run software. To the same aim, MProlog allows definition of global variables. This of course is a touchy issue from the software engineering point of view. However, arguments for and against notwithstanding, the manipulation of global variables in MProlog is very tedious. (This might be viewed as either an advantage or a disadvantage!)

Inverse Predicates.

Every invertable predicate and operator provided by a language should have an inverse provided by that language. This is not the case in MProlog (as indeed it is not many other languages). For example, the MProlog predicate make_char_list begs for an inverse. This predicate takes a string and decomposes it into a list of its characters. It is almost an unavoidable conclusion that if make_char_list is used, then one would need to do the reverse sooner or later.

In Conclusion.

The development environment provided by MProlog version 1.5 leaves a lot to be desired. However, most of the disadvantages and missing features are quite easy to eradicate. We hope that the manuals will be revised soon, and that the communication of PDSS with the Operating System will be of Logicware's next priority. Debugging facilities should be expanded, to include features such as those mentioned in the text of this article.

The production facilities of MProlog are good. Modularity is important, particularly when it allows separate compilation of modules. Usage of such precompiled modules should be allowed in the Profile.MProlog file.

References.

1. MProlog Language Reference Logicware Inc., Release 1.5 Dec 1984
2. Logic Lab Reference Logicware Inc., Release 1.5 Dec 1984

Algorithms to Play Mastermind

T. M. Rao, G. Kazin, and D. O'Brien
Department of Mathematics & Computer Science
SUNY College
Brockport, NY 14420

Abstract: Several algorithms to play mastermind have been published. However, the overall complexity of the algorithm has not been discussed. In this paper, we present three algorithms and an analysis of their complexity.

1. Introduction and Terminology The problem of finding the best algorithm to play the game of mastermind is of considerable mathematical interest. This problem has attracted the attention of many researchers recently. ([1 .. 6]).

The game is played by two players, The code-maker and the code-breaker. The code-maker makes a secret code, S, of his choice. The code itself is a string of symbols from a finite alphabet C. The length of the code string N is known to both players. The code-breaker now makes a sequence of guesses to break the code. Each guess, of course, is a string of length N over C. For each guess G, the code-maker responds by giving a pair of numbers (b,c) where b is the number of positions in the secret code S for which S[i] = G[i] and c computed as follows: We first eliminate all positions considered in computing b. We then compute c by counting the number of S[i] that are also in G, thus b represents the number of right symbols in right positions and c the number of right symbols in

wrong positions. (See [1] for details). An intelligent code-breaker will exploit this information in making his next guess. The game ends when the respond is (N, 0), that is S[i] = G[i] for i = 1 to N. The object of the game is for the code-breaker to break the code in the least possible number of guesses.

The solution to this problem is of importance, because it can have applications in coding. If we treat the secret code as the message we want to transmit, then we can just transmit the sequence (b_i, c_i). At the other end the receiver can use the exact same algorithm to decode the message. Further, in Section 3, we discuss a variation of this game where only b_is are used. This would be better suited for applications in coding.

Several algorithms to play the game have been published recently. Most of these algorithms attempt to minimize the number of guesses. However, little attention has been paid to the amount of computation involved in generating the next guess. In this paper we present three algorithms and an analysis of their complexity. It should be noted that the algorithms here are presented in a way that reflects the analysis. In actual implementations, however, many other heuristics are used to reduce the number of guesses.

In what follows, C represents the finite alphabet of symbols, M = |C| represents the number of symbols in C and N denotes the length of the secret code string. We also assume that C contains a symbol, '$', that will not be a part of any secret code. We emphasize that the role of '$' is merely that of space filling. Further, for any secret code which does not contain all the symbols, a symbol not in it can be easily found. If the secret code contains all symbols, then we can use a symbol that is known to be present as the space filler and modify the algorithms accordingly.

This algorithm is a simplified version of the algorithm presented in Rao [1]. We present the algorithm in rather informal steps. For simplicity, we assume that c = {x_1, x_2, ..., x_m}.

Algorithm

Step 1: Find all the symbols, y_1, y_2, ..., y_k in S and their corresponding frequencies q_1, q_2, ..., q_k using guesses of the type:

$x_i, x_i \ldots x_i$

The response to each of these guesses will decide if the symbol is present in S, and if so its frequency of occurrence.

Step 2: A symbol, y_i, which is known to be present in S, can be solved that is its correct position determined by using guesses of the form:

$y_i\$\ \ldots\ \$\ \$y_i\$\ \ldots\ \$\ \ldots\ \ \$\$\ \ldots\ y_i\$$

where $ is the special symbol known not to be present in S. Each guess will decide if a position is right for y_i. Every occurrence of each y_i can be solved in this way. Note that once a symbol is known to be in a certain position, we retain it in that position in all future guesses.

Observe that Step 1 will take a maximum of (M - 1) guesses. Each guess, being a string of length N, will take N operations to compute it. For each symbol it takes (N - 1) guesses at worst to solve it. Thus the algorithm has an overall complexity of $O(MN + N^3)$. If one is interested in the number of guesses would be $O(M + N^2)$.

3. Algorithm 2 this algorithm works by dividing the problem into smaller subproblems. We first introduced the concept of a bag. A bag is essentially a set, except that repetitions are allowed. A bag is different from a list in that there is no ordering in a bag. Thus, if all the symbols in the secret code along with their frequencies are known these can be put in bag. The algorithm first computes the bag of these symbols and partitions it repeatedly into smaller bags until each bag has only one symbol. Each recursive call partitions the bag into two smaller bags: LBAG and RBAG. A final concatenation will produce the answer.

Algorithm

Step 1: Compute y_1, y_2, ..., y_k and q_1, q_2, ..., q_k as in step 1 Algorithm 1.

Step 2: Set LBAG = $\emptyset$ and RBAG = $\emptyset$ ($\emptyset$ is the empty bag)

Step 3: For each y_1 do the following:

Try a guess of the form $y_i y_i \ldots\ y_i\$\$\ldots\$$ where there are N/2 occurrences y_i. Let (b,c) be the response. As the number of occurrences of y_i is q_i this implies that there are b occurrences in the left half of S and $q_i - b$ on the right half. Thus we update LBAG and RBAG:

LBAG = LBAG + {y_i, ..., y_i} (b times) RBAG = RBAG + {y_i, ..., y_i} ($q_i - b$ times) where + represents bag union.

When Step 3 is completed we would have partitioned the original bag of symbols into two bags with N/2 elements each. Step 2 and Step 3 are repeated for each subbag until each smaller bag has exactly one element in it.

We observe that the complexity of making a new guess again is O(N). It takes at worst N guesses to split the original bag into two bags of size N/2. It takes another N/2 guesses to partition the left bag. Thus after another N guesses we have 4 bags of size N/4. Hence we need NlogN guesses to arrive at N bags of size one, proving that the complexity of the algorithm is $O(MN + N^2 \log N)$. The worst number of guesses would then be $O(M + N \log N)$.

Algorithm 2 is better than Algorithm 1 in both the overall complexity and the number of guesses. Further, this algorithm is better suited for coding applications. Observe that the c_i's, in this algorithm, played no role at all. This means that the algorithm is applicable in a more difficult mastermind game, the one in which the code-maker supplies only b_i's and not c_i's. From the coding point of view, we can just transmit the sequence of b_i's. In the worst case, for a message N characters we have to transmit approximately N log N integers. However, as repetition of characters in the message reduces the number of guesses, we should have to transmit far fewer than NlogN integers in an average case. (It is interesting to note that, if S is the secret code and G_i is the i^{th} guess that $N - b_i$ is the Hamming Distance between S and G_i).

4. Algorithm 3 Both the algorithms presented earlier do not exploit the information given by the code-maker completely. This algorithm makes better use of this information ensuring a fewer number of guesses. However, the amount of computation done in each guess increases sharply. We first observe that the response (b,c) can be viewed as a measure of the similarity between the guess G and the secret code S. To formalize the notion we define a simple function d as:

$d(S, G) = bN + c$

The algorithm begins with a list L of all the $(M-1)_N$ strings of length N over c-{$} ordered in some fashion.

Algorithm

Step 1: Pick any element G from L as the Next guess. Suppose (b,c) is the response. If b = N then stop else let $d_1 = d(S, G)$.

Step 2: It follows that the secret code is one of the strings X for which $d(X, G) = d_1$. Thus we eliminate all strings in L for which $d(X, G) \neq d_1$.

Step q and Step 2 are repeated until the secret code is found. Observe that the first guess will eliminate a substantial part of L and subsequent guesses will reduce L rapidly. In practice this algorithm can be implemented slightly differently.

We use C - {$} = {0, 1, 2, ..., M-2} to illustrate the implementation. The first guess will be 000 ... 0. Each guess will be treated as an N digit number in base M-1. Every guess will be numerically larger than the previous guess. Suppose G_1, ..., G_i are the first i guesses and (b_1, c_1), ..., (b_i, c_i) the corresponding responses. Then the $(i + 1)^{th}$ guess G_{i+1} will be the smallest N digit number (with possible leading zeros) greater than G_i and such that

$d(G_{i+1}, G_k)=d(S,G_k=b_kN+C_k$ for k=1 ... i.

the process will of course, stop when we have $d(S,G_i) = N^2$ for some i.

The above algorithm is one of the best if the number of guesses is the primary criterion. However, it is one of the worst in terms of overall complexity. In the worst case the time taken to generate the next guess is exponential, thus making it unsuitable for large values of N.

5. Conclusion We have discussed three algorithms to solve the Mastermind problem. Algorithm 1 is rather elementary. Algorithm 2 which proves to be the best in terms of overall complexity is suitable for coding applications. However, these two algorithms are not very suitable for coding applications. However, these two algorithms are not very suitable when used to play the game because the guesses will be predictable thus making the game uninteresting. Algorithm 3 is best suited for a game playing program, because in this case we can assume N to be a small number like 4 or 5. For such small values the time taken to generate the next guess can be ignored because the user will probably take more time to compute his b's and c's. Further, if the alphabet is randomly shuffled at the beginning of each game the program can appear to be very intelligent. We have a LISP implementation of the Algorithm 3 which uses the alphabet (0, 1, ..., M-2). At the start of each game it randomly shuffles the alphabet there by assuming a different ordering. Further, we use some heuristics in generating the next guess such as treating each position as a cycle and deleting the impossible values from each cycle. Even if a user played with the same secret code twice, the sequence of guesses will not be the same. It is indeed impossible to figure out the strategy of the program by playing with it several times.

References: (all in SIGART)

1. Rao, T.M. No. 82, Oct. 1982, 19-23
2. Gyllenskog, J., No. 84, Apr. 1983, 34-35
3. Shapiro, E., No. 85, July 1983, 28-29
4. Koppenstein, No. 81, Apr. 1984, 11-14
5. Rada, R., No. 89, July 1984, 24-25
6. Powers, D. M. W., No. 89, July 1984, 28-32.

An Eclectic 5th Generation Architecture for Ultra High Speed Computing

Larry O. Rouse, and John F. Forbes
RDA/Logicon, Marina del Rey, CA, (213) 822-1715
Forbsco, Fresno, CA, (209) 233-0126

Jean-Luc Gaudiot
EE Systems Department
University of Southern California
Los Angeles, CA
(213) 743-0249

ABSTRACT

Ultra High Speed Computing will be addressed in the form of a new computing architecture using networks of processors, program partition/allocation techniques, and with specific design elements of the type that would be used for database acceleration. This optimum networking architecture will merge commercially available components and support coordinated algorithms and software methodology to fully exploit potential gains in computational speed through parallel computing. The network of processors to be investigated will be of value at the fine grain level for computational processes and at the large grain level for database and communication processes. The complimentary relationship of dataflow and database technology will be sued to focus the investigations with an emphasis placed on hardware implementation. A Phase I program, described in this paper, would analyze the proposed architecture, including hardware and software tradeoffs, and would define an appropriate simulation based methodology for a future Phase II program in which the candidate designs would be tested.

Technical Summary

The ability to scale the number of processors in a 5th generation architecture is essential. Our design approach retains this feature and can also pool local processors efficiently while still exhibiting low broadcast communication overhead. These performance factors are often at odds in other designs.