Brute Force and Artificial Intelligence

David Barry Schaechter*
Lockheed Palo Alto Research Laboratory, Palo Alto, California

This paper contains a description of a brute force approach to the AIAA Artificial Intelligence Design Challenge. A systematic approach to solving this general type of problem is presented. This approach consists of first formulating the problem in a higher level language for testing and debugging purposes, and then following this step with a line-by-line translation of the higher level language into assembly code to fully exploit the speed of the computer. The final computer program is about 150 lines of code.

Background and Introduction

I HAVE been working as a control system engineer for 14 years. What is somewhat less known about me is that I have also been involved with computers, specifically, computer game playing, for an even longer segment of time. This involvement with computers has been at both the hardware and software levels. (In this case, I use the word software to mean code at the machine-code level.) I believe that this combination is crucial for one to be able, first, to understand what types of operations can be performed efficiently by a computer and, second, how to combine these operations to make the computer function at the maximum speed possible. A complete documentation of my exploits into these areas may be found in Ref. 1.

In 1978 I purchased a 6502-based SYM-1 single board computer with 4 K of memory, and a 1 MHz clock rate, in 1982 I upgraded to a Commodore 64. One of my reasons for purchasing the Commodore 64 was that I had gained quite a capability for programming the 6502, to the point that I had memorized virtually all of the op codes for the 6502 and could literally directly write machine language code. The Commodore 64 is the machine I used for this design challenge.

With the SYM-I, I had enough capability to begin programming games. The search strategy used in these game programs is almost identical to that used in the artificial intelligence contest problem. At about the same time that I purchased the SYM-I, I was given the disk-flipping stategy game known as "othello" (manufactured by CBS Toys). The game is played on an 8×8 board, and I perceived this as an easy mark for implementing on the computer. I had no background in artificial intelligence, and so my first attempt at finding good moves was a multiple depth search for what I now know is the "minimax" best move. It was relatively straightforward to produce a program that would search to a depth of four plies in less than a second of time. As is usual in these type of problems, the computation time grows exponentially with the number of plies examined. Still, after optimizing the machine code, which involves putting in special counters to find out how many times various portions of the code are executed, counting the number of machine cycles that are required by the various portions of the code, and then concentrating effort on reprogramming those portions of the code which consume the most time, I obtained a compact and fast running program. I have found that there is virtually no limit to the improvement that can be realized in rewriting specific portions of assembly code. The types of improvements that I am talking about are precalculating as many constants as possible, using internal registers for data storage instead of main memory, minimizing the use of subroutine calls (stack manipulations require more execution time), and explicitly unwrapping loops to run as straight line code. Tree searching algorithms can also be greatly accelerated by performing only those computations that are absolutely required at each node *and no more.* By the time I finished this program, I had a program that could examine moves and countermoves to a depth of six plies at the beginning of the game, and by the end of the game (when there were fewer moves from which to choose) 10–14 plies. At the conclusion of my work, I had a program that could routinely beat me (and could pass as artificial intelligence?).

Received June 18, 1987; presented as Paper 87-2337 at the AIAA 1987 Guidance, Navigation, and Control Conference, Monterey, CA, Aug. 17–19, 1987; revision received Jan. 27, 1988. Copyright © American Institute of Aeronautics and Astronautics, Inc., 1987. All rights reserved.

*Research Scientist. Associate Fellow AIAA.

The Contest Problem

The following steps were taken in coming up with a brute force procedure for solving the AIAA Artificial Intelligence Design Challenge.[2,3]

1) Write a BASIC program that will solve by direct enumeration of all possible itineraries a reduced version of the full problem. The BASIC program should contain all of the features of the final problem, and should solve the full problem by the changing of a single input parameter. In this example, the reduced problem consisted of finding the best itinerary of length n, where n was some value substantially less than the longest possible itinerary. It is relatively easy to check for the best itinerary that visits only one or two cities. If the program can be generalized to solve the higher order problem by changing a single parameter, there is almost no chance of introducing programming errors when going to the more complex problem.

2) This is the time for algorithm development. Because it is so easy to modify a BASIC program, different algorithms can be studied. Program lines are arranged in an order to minimize the total amount of computation. It is also at this stage that counters are inserted into the code to determine how often certain parts of the code are executed and to perform timing studies on the program to see if the computation time growth is consistent with the predicted exponential growth. I have also found that interpreted BASIC, the tool initially used for program development, can be accelerated by a factor of 10 by compiling, and can be accelerated another factor of 100 (i.e., a net improvement of a factor of 1000) when translated into optimized assembly language. It is at this early stage that the first (and quite accurate) estimates of the run time of the final problem can be made. It will often times permit a decision as to whether or not it will be possible to obtain a brute force solution, or if some method of shortening the search procedure will be required.

3) A compiled version of the BASIC program is run to refine further the estimates of the total run time and the number of times that various portions of the program are executed. The extra speed of the compiled version allows for more realistic estimates at a deeper level.

4) The final step is to translate the BASIC program, line by line, into optimized assembly language. It is at this stage where single byte variables are often more than enough for loop counters, intermediate results can be temporarily stored in registers rather than memory, and so forth.

Some Programming Specifics

Probability Calculations

At the outset, it should be recognized that a larger amount of the computation will be tied up in the determination of the probability that a given itinerary will exceed the budget limit. Effort must be devoted to limiting the number of times that this computation must be performed and to optimizing this calculation for the times that it must be computed. There are two ways of limiting the number of times the computation must be performed. First, if the itinerary is sufficiently inexpensive so that even in the event that all of the fares are increased by the first-class multiplier the itinerary will still be within the budget limit, then there is a zero probability that this itinerary will exceed the budget limit, and so no major computation is required. Second, if the itinerary exceeds the limit with all the coach fares in force, then again there is no reason to do a detailed computation; the probability of exceeding the budget is one.

Now let's consider the computation of the exact probability of exceeding the budget. Let the itinerary consist of n consecutive trips. Of course, two versions of a revised fare table have been precomputed (one time). The first table is the original fare table, but all of the fares to cities other than the home city have been increased by \$100 to account for the travel expense. The second table is also precomputed, but in this case all of the fares are multiplied by the first-class multiplier, and then the travel expense of \$100 is added to all the fares that do not go to the home city.

For an itinerary that consists of n trips, there will be 2^n possible fares, corresponding to whether or not coach fare is available at each leg. Two bytes are sufficient to represent all the possible fare outcomes for any itinerary of less than or equal to 16 trips. Let 0 in the ith bit represent a coach fare for the ith trip, and 1 in the ith bit represent a first-class fare. A two-byte counter that steps through the binary numbers from 0 to 2^{n-1} represents all the possible fares. A program was developed that examines each bit of the counter, and depending on whether the bit is 0 or 1, either adds (in fixed-point, two-byte arithmetic) the fare from the standard fare table, or from the table that corresponds to the first class fares. When all n bits have been examined in this manner, the resulting total fare can be compared to the budget limit. If the total exceeds the budget limit, then the element of an array that corresponds to the number of ones in the counter is incremented by one. Why? The probability of a particular fare depends only on the number of first-class fares that are encountered along a particular itinerary. For example, all itineraries that consist of two first-class fares and the rest coach fares are equally likely. Similarly, the same applies for all those itineraries with three first-class fares, etc. When the counter has proceeded through its 2^n possibilities, an array distr(i) is generated. The ith element of this array indicates how many times an itinerary of length n consisting of i first-class upgrades has exceeded the budget limit. To this point, no floating-point operations have been required, and a total of $n*2^n$ fixed-point two-byte additions have been performed (on my computer each such addition requires only 28 μs). We still have not yet computed the probability that the itinerary exceeds the budget limit. The final step in this calculation is to cumulatively sum over i the individual probabilities that there are i first-class upgrades in an itinerary times the number of times that the n trip itinerary consists of i first-class upgrades. This works, since the probability that the first leg of the trip requires a first-class upgrade is the same as the probability that the second leg of the trip requires a first-class upgrade, etc.; the probability that the first two legs of the trip require first-class upgrades is the same as the probability that the second and the fifth legs will require first-class upgrades, etc. There are a total of $n+1$ possible choices, as i ranges from 0 to n. The probability of exactly 0 first-class upgrades is just $(1-fp)^n$ the probability of one first-class upgrade anywhere along the trip is $(1-fp)^{n-1}*fp$. In general, the probability of i first-class upgrades occurring in an n trip itinerary is $(1-fp)^{n}*fp^{i}$. These values need to be computed *one time* only and stored in an array prob(i) for later use. The probability that an itinerary exceeds the budget limit is then given by the sum over i of distr (i)* prob (i). The total computational burden to perform this computation for a 10 trip itinerary is just $10*2^{10}$ fixed point additions (approximately ½ second, plus 11 floating-point multiplies).

Search Procedure

It is vital at the outset of solving this problem to get some idea of the number of possible itineraries that must be examined. At first glance, there are 11 cities to visit, so for an itinerary of length 12 (starting and ending at the home city and visiting all 10 other cities) there should be 11^{12} (i.e., $3.4*10^{12}$) possible itineraries. In actuality there may be even more itineraries than this since it is possible to visit a single city more than one time in order to make airline connections. To just enumerate all of these itineraries allowing for 100 μs per itinerary would take $3.14*10^8$ s, an amount well outside the time allotment of 20 min. Let's take a closer look at this last calculation. The home city is given. This prescribes both the first city on the itinerary and the last city. The effective length of the itinerary is now 10, not 12. Furthermore, there is no advantage to visiting the same city on consecutive stops; therefore, at each stop, there is a list of only 10 cities to choose from for the next leg of the tip. Now the number of itineraries to be examined has been reduced by a factor of 300 to 10^{10} possibilities, with no feasible itineraries being omitted. The total computation time required to search through all of these possibilities is roughly 10^6 s or about 16,000 min., still too long. Now it is quite likely that not all of these itineraries will have to be explicitly enumerated.

Consider the process of searching for the best trip. At some intermediate stage of the search, there will be a current best itinerary that consists of an itinerary of producing a certain value and costing a certain amount. While examining another itinerary, if either the budget limit is exceeded before the end of the itinerary, or the current budget exceeds the best trip budget with no hope for producing a better value, the search may be terminated. An example follows.

Consider the following itinerary that begins in DTT and then alternately hops from BOS to LAX. If after the first six trips the budget limit is already exeeded, it is pointless to examine the remaining permutations of cities for the last six slots on the itinerary. This eliminates 10^6, i.e., 1,000,000 itineraries that must be examined. This is precisely where the use of interpreted BASIC comes in handy. A program is set up to count exactly how much is saved by terminating the search at an early stage on a reduced problem. First, a single trip itinerary is examined, then a two-trip itinerary, then three, etc. Overnight runs can be tolerated at this stage of the analysis to determine the net savings. For the sample problem, another factor of 100 reduction in computation time was realized, thus reducing the total predicted computation time to 160 min. We are now in the ballpark. A brute force solution should be possible. The final heuristic that was used to make the brute force approach feasible was to assume that a good itinerary of length n would probably have the same first leg trip as the best itinerary of length $n-1$. If this hypothesis is true (as it is in many cases), then the brute force program will find the best itinerary of length n in $1/n$ times the total search time, even though the remaining time must still be used to check all the other possible itineraries to guarantee that that itinerary actually is the best. The advantages of organizing the search in this manner are that it requires no additional computation and that in most instances it will produce the best itinerary earlier in the search procedure (which has the effect of eliminating more of the itineraries examined later in the search). Consequently, if the program is interrupted before completion, there is a greater chance that it has obtained the best itinerary. If this heuristic does work, then the best itinerary is actually found in about one-tenth of the time, or about 16 min; if it does not, then nothing is lost.

Minimizing the Computation

A series of nested loops is used to generate the itineraries. All of the initial city counters are started at one. Each counter at a lower level is incremented only when all the possibilites at the next higher level have been generated. At that time, the index at the higher level is reset to one, the index at the lower level is incremented by one, and the index that governs which level of the tree is currently under consideration is decremented by one. The search is complete when an attempt is made to backup to level 0 on the tree.

The computation that must be performed in order to solve this artificial intelligence design challenge is the determination of the total value and cost for a given itinerary. The simple-minded approach to performing these calculations is to sum the values of the cities and the individual fares along the route, a total of $2n$ additions (again only fixed-point two-byte arithmetic) for each route. If there are 10^n itineraries, the total computation is $2n*^n$ additions. Only a small fraction of this computation is actually required, however. Given that the fare and value for the itinerary of length $n-1$ has already been calculated, only two more additions are needed to obtain the value and fare for any of the itineraries of length n that begin with the previously analyzed itinerary of length $n-1$. For $n=1$, 20 additions are required with either method. For $n=2$, 400 additions are needed with the first method, but only 220 for the second method. In general, the method that performs only those computations that are required at each node of the search tree will require only $2*(10+10^2+\ldots 10^n)$ additions which, for $n=10$ results in a savings of almost 90% of the computation.

The programming to implement the above procedure is quite minimal. An n dimensional value array and an n dimension cost array are defined. In order to update the cost array when traveling from city $i-1$ to city i, the following line of code is executed:

$$C(i)=C(i-1)+\text{FARE}(i,i-1)$$

where FARE $(i,i-1)$ is the fare from city $i-1$ to city i. The array is easily propagated forward along the itinerary as the itineraries are enumerated. Furthermore, when all of the cities at the end of the tree are examined, it is also easy to backup along the tree and get the current cost of the fragment of the itinerary for the start of the next branch of the tree.

In the problem statement, a city value is awarded only on the first visit to a city and not on subsequent visits. A bookkeeping method must be devised so that at each stage of the trip there is no requirement to go back and sequentially examine if a city has been previously visited. The way this was accomplished in my version of the program was to establish an additional array known as the first visit array, FV(i). The array is initialized to some large value. At the ith level of the search tree, when city j is visited, the entry FV(j) is examined. If FV$(j)<i$, then the city was previously visited and no additional value is added. If FV$(j)>=i$, then the city has not yet been visited, FV(j) is set equal to i, and the value for city j is added to the running value of the itinerary.

There is also a good deal of speed advantage that can be gained by cleverly storing data. In my version of BASIC, the default treatment of a variable is a floating point number. If a loop index needs to be incremented by one, a five byte number must be addressed and incremented. It is possible to use integer variables, but even integer variables are stored as double bytes, a needless waste of time and storage. In writing optimized machine code, of course, full advantage can be taken of the known range of all the variables. Loop indices are stored as single bytes, as is the accumulated value of the trip, while two bytes are all that are needed to store all the fare information.

Consider next how to access the fare information easily. The fare data are initially formatted as a two-dimensional 11×11 integer array. In either my interpreted BASIC or my compiled BASIC, in order to access the value of such an array (ignoring for the time being the fact that these indices are stored as two-byte integers rather than single-byte integers), the following arithmetic must be performed. The dimension of the array must be looked up, the value must be multiplied by i, the value of j must be added, and then this value is multiplied by 2 (albeit with just a shift instruction) to account for the fact that each value requires two bytes of storage. This value is then used as an index to extract the value of the array from memory. An alternative approach can be made much faster. First, let the data be organized as two 16×16 arrays. The first of these two arrays will store the most significant bytes of the fares, and the second will store the least significant bytes. Elements 17–27 will store the fares to city 1, elements 33–43 will store the fares to city 2 and, in general, element $16*i+j$ will store the fare from city j to city i. The reason for storing the array as a 16×16 array rather than in an 11×11 array should now be obvious. It is quite easy in binary arithmetic to form the quantity $16*i$ from i; it requires only four logical shifts. In fact, it is even simpler than this. An array is precalculated so that its ith entry is $16*i$. Since the lowest four bits of the product $16*i$ are guaranteed to be zero, the result can be added to j with the use of an OR instruction. To look up the fare from city i to city j, the following lines of code are used.

```
LDX I:           Load the x index with i.
LDA ARG16, X:    Get 16*i into the accumulator.
ORA J:           Form 16*i+j.
TAY:             Transfer array index to Y.
LDA FARE, Y:     Get the fare from i to j.
```

This segment of code, which must be repeatedly executed, requires only 18 μs.

Expected Fare Computation

The challenge problem contains the statement that there is a probability, FP, that the given coach fare may not be available when the trip is begun and that first-class fare will have to be paid in order to continue along the itinerary. This first-class fare carries a premium, which is FM times the original coach fare. Consider now the expected fare that will result from a single trip whose coach fare is COACH. The probability that coach fare will be paid is $(1-\text{FP})$ and the probability that first-class fare will be paid is FP. The expected fare is just $(1-\text{FP})*\text{COACH}+\text{FP}*(1+\text{FM})*\text{COACH}$ or, more simply, $(1+\text{FP}*\text{FM})*\text{COACH}$. Now, since the sum of the expected value of several trips is just the expected value of the sum of the trips, a great deal of computation can be saved by performing the expected value of the sum [i.e., multiplying the coach fare by $(1+\text{FP}*\text{FM})$] *once* at the end of the trip, rather than for each trip. Since the coach fares are all integer values, integer arithmetic can be used to accumulate the running trip cost, and only a single floating-point calculation needs to be performed when the exact expected trip value is needed. In my program, this floating-point calculation is performed once, when the output is generated.

Not Quite Brute Force

I claimed earlier that a brute force approach was applied to solving the challenge problem and I implied that my program exhaustively tried all the possibilities. This is not actually the case. Although this program can indeed find the best itinerary of a length 12 in the required time, this is not the way the problem was stated. It is possible that a city may be visited more than one time, if desired, in order to take advantage of a cheaper connecting flight. Since I was already hard up against the time constraint, I had to cheat somewhat and make use of some heuristics. The final program does exhaustively search all itineraries of length seven or less. From this point on, however, I had to tradeoff the exhaustive search for the ability to examine the longer itineraries. If a city is visited more than one time, it must be for a very good reason, since no additional value is accrued, and since an additional per diem expenditure of $100 is incurred. I translated this heuristic into the following algorithm. For itineraries of length eight or more, the program examines only those cities that have not previously been visited, along with the one city that can be visited for the cheapest cost. With logic that already existed in the program, there was no extra time cost incurred by checking if a city had been already visited. Also, the cheapest fare from a given city can be precomputed once for a given data set and stored in an array. This program change created no differences in output for the few sample data sets that I used, and this is the version of the program that I submitted to the contest.

Small Details

The Commodore 64 contains hardware to perform an interrupt 60 times/s. The main purpose of this interrupt is to scan the keyboard to determine if any keys have been pressed and to update the system clock. There is also some overhead associated with this interrupt, since the state of the system must first be saved, and later restored. After a benchmark test, I determined that the interrupt routine robs about 2% of the central processor's time. In order to reclaim this small amount of time, I disable the hardware interrupt in my assembly language routine and perform my own keyboard check to determine if the user wishes to halt the program manually.

Conclusions

The Artificial Intelligence Design Challenge was particularly intriguing to me because I happen to be an unstoppable game enthusiast. A brute force program was developed that has implemented intelligent search strategies to reduce the amount of computation time; however, the program itself is not intelligent. For instance, after spending 20 min of time to track down the best itinerary, the program will merrily spend another 20 min of time on the exact same data set. The program possesses no insight, no adaptability, and no learning capability from previous experience. Artificial intelligence is here to stay. Its study will no doubt prove fruitful in years to come. However, we should not lose sight of the fact that artificial intelligence methods will not be universally applicable; that often times, brute force methods or clear programming may go a long way in a particular application.

References

[1]Schaechter, D.B., "Brute Force and Artificial Intelligence—The AIAA Artificial Intelligence Design Challenge," AIAA Paper 87-2337, Aug. 1987.

[2]Deutsch, O.L., "The AI Design Challenge—Background, Analysis, and Relative Performance of Algorithm," AIAA Paper 87-2339, Aug. 1987.

[3]Deutsch, O.L., "The AI Design Challenge—Problem Statement," *Journal of Guidance, Control, and Dynamics,* Vol. 9, Sept.-Oct. 1986, p. 513.