KMITL Sci. J. Vol. 5 No. 1 Feb. 2005

Matrix-Chain Multiplication Problem using Genetic Algorithm

Soontharee Koompairojn* and Minh Le

School of Computer Science
University of Central Florida
Orlando, FL 32816-2362 USA
E-mail: soonthar, mle@cs.ucf.edu

ABSTRACT

Tree encoding has been studied for the genetic algorithm on artificial intelligence such as sequence induction, automatic programming, machine learning, and pattern recognition [5]. This paper also presents the tree encoding for the genetic algorithm in solving the matrix-chain multiplication, which is in the form $A_1 A_2 A_3 \ldots A_N$ where A_i is a matrix. Trees are generated as the way the matrices are fully parenthesized. Then crossover and mutation are applied. The fitness value is calculated and stored at the root of the tree.

KEYWORDS: tree encoding, matrix-chain multiplication, genetic algorithm

1. INTRODUCTION

Consider a chain of N matrices, $(A_1, A_2, A_3 \ldots A_N)$, we wish to evaluate the product

$$A_1\ A_2\ A_3 \ldots A_N \qquad (1)$$

where A_i is $P_{i-1} \times P_i$ matrix. Since matrix multiplication is associative, the final result in (1) is the same for all orders of multiplying the matrices. However, the order of multiplication greatly affects the total number of operations to evaluate (1), in [4]. Therefore to find the optimal multiplication order that minimizes the number of performed operations is interesting.

In [3], a dynamic programming algorithm is used to find an optimum order. The space complexity of the proposed algorithm is $O(N^2)$ and the time complexity is $O(N^3)$, where N is the number of matrices in a chain. The time complexity for determining a near-optimal computation order proposed by Chanda [1] and Chin [2] is $O(N)$ time. The time complexity for an algorithm to find an optimum solution by using the graph theory proposed by Hu [4] is $O(N \log N)$. In this paper, we investigate this problem using the genetic algorithm technique with the tree encoding. This technique also needs $O(N)$ time.

The preliminary considerations are described on section 2. Then we explain the detail for each component of genetic algorithm in section 3. Section 4 shows the experimental results and compares the results with the optimal solution computing by the dynamic programming approach. Finally we conclude our implementation and problems on section 5.

*Corresponding author. Tel: (407) 823-3483 Fax: (407) 823-5419 E-mail: soonthar@cs.ucf.edu

2. PRELIMINARY CONSIDERATIONS

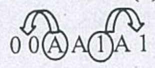
From (1), we fully parenthesized it into pairs to resolve in how the matrices are multiplied together [3]. Consider the problem of a chain (A_1, A_2, and A_3) of 3 matrices. The product $A_1A_2A_3$ can be fully parenthesized in 2 distinct ways:

$((A_1\ A_2)A_3)$
$(A_1\ (A_2\ A_3))$

When applying the genetic algorithm to this problem, the problem representation is considered. First, consider the string encoding, we can represent "(" as 0, ")" as 1, and A is a matrix ignoring the subscript since the matrices are in order. (The order of matrices in the chain is fix.) Therefore, the string representation is shown as follows:

0 0 A A 1 A 1 (2)
0 A 0 A A 1 1 (3)

We can move from (2) to (3) as follows:



However, when the number of matrices is large, there are several constraints to handle. We can show by giving an example for a chain of 4 matrices. There are 5 distinct ways of fully parenthesization. The string representations are shown below:

	Position									
string	1	2	3	4	5	6	7	8	9	10
1)	0	A	0	A	0	A	A	1	1	1
2)	0	A	0	0	A	A	1	A	1	1
3)	0	0	A	A	1	0	A	A	1	1
4)	0	0	A	0	A	A	1	1	A	1
5)	0	0	0	A	A	1	A	1	A	1

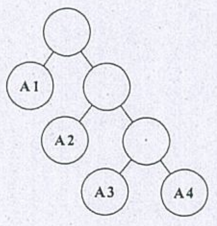
The constraints to deal with a chain of 4 matrices as mentioned above are

- the first 0 is fixed at position 0
- the last 1 is fixed at position 10
- the element 0 cannot move further than 6
- the element 1 cannot move lower than 5
- it cannot be more than two consecutive A.

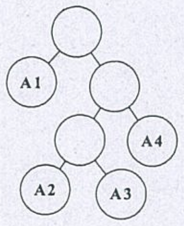
It is hard to implement a program to handle the permutation of the string representation with these constraints. Reparation is required on this representation.

Koza [5] used the tree encoding in order to generate the computation procedure (or LISP S-expression). LISP S-expression is in the form of list. We can also represent the matrix-chain multiplication by using parentheses. Therefore, The tree representation might be suitable for the matrix-chain multiplication problem. And we can easily convert a fully parenthesized matrix chain (list) into tree.

Each list has a unique tree representation. We can create tree by using the bottom up manner. For example, consider the chain of 4 matrices, $(A_1(A_2(A_3A_4)))$ can be represented as figure 1(a). And $(A_1((A_2A_3)A_4))$ can be represented as figure 1(b).



(a)
$(A_1(A_2(A_3A_4)))$



(b)
$(A_1((A_2A_3)A_4))$

Figure 1 tree representation

Notice that the leave nodes are in the same order from left to right regardless of the height of leave nodes.

3. COMPONENTS OF GENETIC ALGORITHM

In order to describe the genetic algorithm for the matrix-chain multiplication problem, the following topics are concerned.

- Initial population generation
- Fitness function
- Crossover
- Mutation

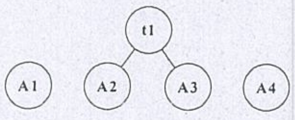
Initial population generation

Tree encoding is used for each individual of the population. In [5], the initial population of S-expression is generated randomly. Some are not good to be the population. But in this paper, all of the initial populations represent the matrix-chain multiplication.

The algorithm builds the tree in a bottom-up manner. It begins with a set of N leaves, corresponding to N matrices. Actually, these leaves are viewed as subtrees. We randomly select an arbitrary subtree from the pool, form a new subtree by using the selected subtree and its adjacency subtree, then put it back into the pool. This process is repeated until the final tree is created. For example, $N = 4$, First, we have

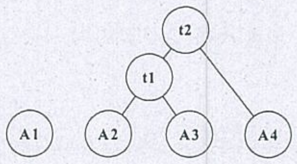
(4)

Suppose from (4), node A_2 is selected. Node A_3 is its adjacency node. They form subtree t_1 where t_1 are the parent node of A_2 and A_3, and it is the root of the new subtree.



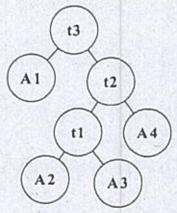
(5)

Suppose from (5), node t_1 is selected. Node A_4 is its adjacency node. The subtree t_2 is formed.



(6)

Finally, node A_1 and t_2 are formed to be the final tree.



(7)

By using the randomize-in-place algorithm, the time complexity for generating the initial population can be done in $O(N)$ time.

Fitness function

Before we discuss about the fitness calculation, consider the basic idea of matrix multiplication. We can multiply 2 matrices A and B only if the number of columns of A is equal to the number of rows of B.

Define $\langle p , q\rangle$ as p x q. Suppose A is a $\langle p , q\rangle$ matrix and B is a $\langle q , r\rangle$ matrix. The resulting matrix C is a $\langle p , r\rangle$ matrix. The time to compute C is dominated by the number of scalar multiplications, which is $p \times q \times r$.

The cost of matrix multiplication of matrix-chain (more than 2 matrices) depends on the parenthesization of a matrix product. Consider the problem of a chain $[A_1, A_2,$ and $A_3]$ of 3 matrices. The dimensions of the matrices are $\langle p , q\rangle$, $\langle q , r\rangle$,and $\langle r , s\rangle$, respectively. The costs of matrix multiplication are shown as follows: [3]

According to the parenthesization $((A_1 A_2)A_3)$,
Cost = Cost of $(A_1 A_2)$ + Cost of $((A_1 A_2) A_3)$
$= p \times q \times r \ \ + p \times r \times s$

According to the parenthesization $(A_1 (A_2 A_3))$,
Cost = Cost of $(A_2 A_3)$ + Cost of $(A_1 (A_2 A_3))$
$= q \times r \times s \ \ + p \times q \times s$

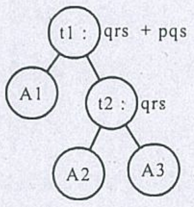


Figure 2 The fitness values for $(A_1 (A_2 A_3))$

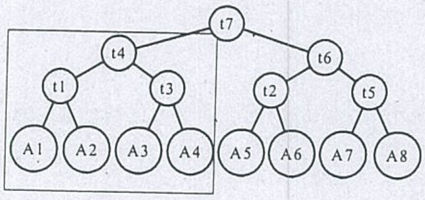
Since our problem representation is a binary tree, the cost (fitness) of matrix multiplication of matrix-chain for each pair (leave nodes) is calculated and stored at the parent node (non-leave node). For example, according to the parenthesization $(A_1(A_2A_3))$, the fitness values are shown in figure 2.

The fitness value of each individual (tree) is the fitness value at the root of tree. It can be calculated in the bottom up fashion from the leaves that have the highest height to the root of the tree. The worst case for this calculation take O(*N*).

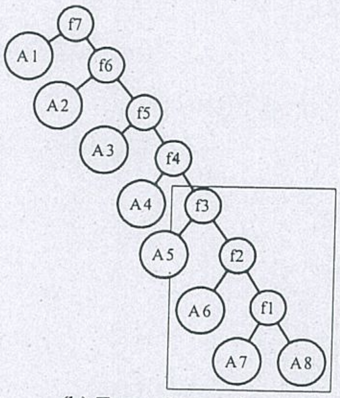
Crossover

After two parents are selected, we randomly select one subtree from the first parent. The root of each subtree keeps the information about the number of leave nodes in that subtree. From the second parent, we randomly select the subtree, which has the same number of leave nodes as the first parent. Then two subtrees are swapped. Hopefully, both have the different shape of tree.

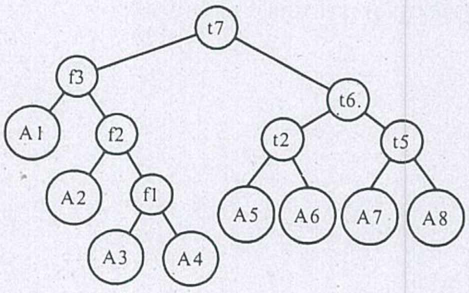
Figure 3 shows how to perform the crossover operation.
Step 1: Select 2 individuals from the population pool.
Step 2: As shown on figure 3(a), subtree t_4 is selected from the first parent.
Step3: Subtree f_3 from the second parent is selected as shown on figure 3(b). This subtree must have the same number of leave nodes as the selected subtree from the first parent
Step4: Two selected subtrees are swapped. The first child is shown on figure 3(c) and the second child is shown on figure 3(d).



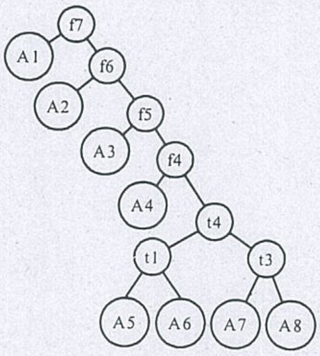
(a) Tree of first parent



(b) Tree of second parent



(c) Tree of first child



(d) Tree of second child

Figure 3 Crossover operation example

After swapping, the fitness values are recalculated as the following steps.

Step 1: The leave nodes of the new subtree are rearranged. When the new subtree is connected to the tree, the shape (structure) of tree is changed. However, the leaves of that subtree are reassigned due to the fix order of the matrices in the chain.

Step 2: The fitness values are recalculated from the bottom up to the root of that subtree.

Step 3: The fitness value of the parent of that new subtree is recalculated from the cost of its two children. This calculation is repeated for the parent node of corresponding subtrees up until it reaches the root of the tree.

The time complexity of the crossover operation depends on the leave node reordering process and the fitness calculation. The worst case of leave node reordering process is that N-1 leave nodes are rearranged. Therefore, the time complexity for this process is O(N). With the time complexity for the fitness calculation that we mentioned before, the time complexity of crossover operation is O(N).

Mutation

After the individual is selected, we randomly select two subtrees. The first subtree is selected from the non-leave node set (not include root). Then the second node is selected from all remain nodes (not include nodes in the first subtree and root). Then two subtrees are swapped.

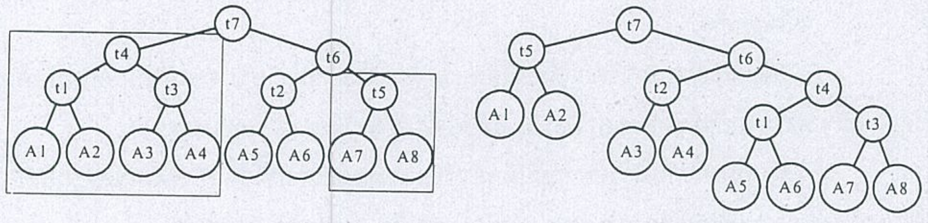
Figure 4 shows how the mutation is performed.

Step 1: Select one individual from the population pool.

Step 2: Randomly select the subtree from the selected individual.

Step 3: Randomly select the other subtree from the remain node (leaves and non-leave node except the root).
Figure 4(a) shows the result of step 2 and 3

Step4: Two selected subtrees are swapped and reordered.
The child is shown on figure 4(b)



(a) before mutation (b) after mutation

Figure 4 Mutation operation example

After swapping, the fitness values are recalculated as the fitness evaluation in crossover operation. However, there are more concerns in the mutation, for example, in figure 4, subtrees t_5 and t_4 are swapped. The leave reordering process is not applied only to subtrees t_5 and t_4 ,but it is also applied to subtree t_2. Therefore, the leave reordering process is done in the topdown fashion from the root to the leaves. The time complexity of the leave reordering process is also O(N). Then the fitness calculation is done in the bottom up manner. The time complexity for this calculation is also O(N).

4. EXPERIMENTAL RESULT

We implement the genetic algorithm with tree encoding by using JAVA. Then we run our algorithm and compare the result with the optimal solution from the dynamic programming.

The genetic algorithm parameters for our experiment are:

The population size:	in the range of 100-400
The number of generation per run:	in the range of 100-4000
The selection method:	proportional selection
Scale of the problem:	minimization
Crossover method:	tree crossover
Crossover rate:	0.6
Mutation method:	tree mutation
Mutation rate:	0.01

The first test case is from [3]. It is a chain of 6 matrices. The data is shown on table 1.

Matrix	Dimension
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

Table 1 The test case with a chain of 6 matrices

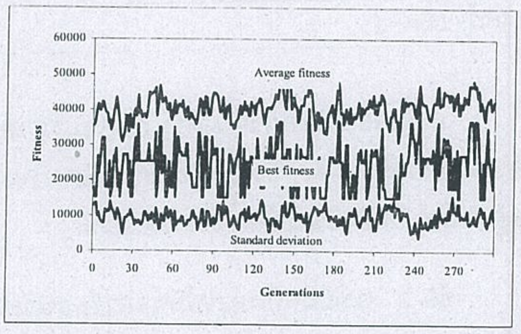


Figure 5 the GA performance of matrix-chain problem for the chain of 6 matrices with mutation only, the population size 10, and the maximum number of generation 300.

By using the dynamic programming approach, the optimal solution is $((A_1(A_2A_3))((A_4A_5)A_6))$. The minimum number of scalar multiplications to multiply the 6 matrices is 15,125.

First, we run our program with the population size 10, the number of generation 300 and mutation only. We get the optimal solution with the minimum number of scalar multiplications of 15,125. The graph is shown on figure 5.

From figure 5, we can see that the result is fluctuated. For the minimization problem, the best fitness is lower than the average fitness value. And from the result, one of our run gets seven consecutive of the optimal solution.

Run	Number of generation that have the optimal solution	
	Pop. size=10, generation=300	Pop. size=2, generation=1500
1	57	86
2	39	88
3	38	100
4	42	80
5	69	63
6	41	62
7	50	66
8	48	73
9	43	75
10	45	84

Table 2 the number of generations that have the optimal solution of the matrix-chain problem for the chain of 6 matrices

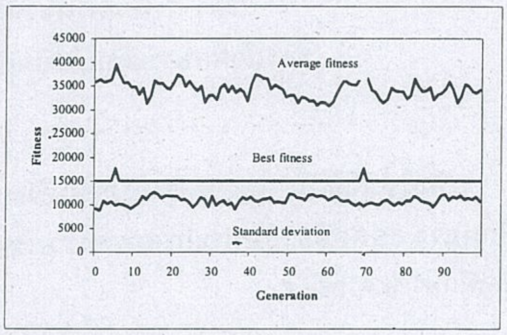


Figure 6 the GA performance of matrix-chain problem for the chain of 6 matrices with mutation and cross over, the population size 100, and the maximum number of generation 100.

After that we run the program on the chain of 6 matrices with population size of 2, the number of generation is 1500 and mutation only. Then we count the number of generation that have the optimal solution, i.e. the result of 15125, compare with the results from population size of 10 and the number of generation is 300. The results are shown on table 2.

With mutation only, the number of fitness calculation call when running with population size 10 and 300 generations is the same as with the population size 2 and 1500 generations. From table 2, it seems like the number of generations that have the optimal solution with the population size 10 and 300 generations is less than when running with the population size 2 and 1500 generations. However, the optimal solution are distribute through generations 1- 1500. Therefore, if we increase the population size and reduce the number of generation, we still get the same result.

Next, we run our program with the population size 10, the number of generation 300 and 1% mutation & 60% cross over operation. We get the optimal solution with the minimum number of scalar multiplications of 15,125. The graph is shown on figure 6. It shows that cross over has an important role in this application. With cross over, almost every generations have the optimal solution, it gives the better result than mutation only (figure 5).

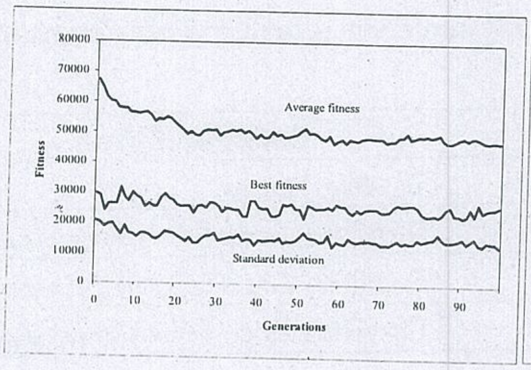
After we get the successful result on the chain of 6 matrices, we run on the chain of 7 – 40 matrices. The results are shown on Table 3. In Table 3, we show the population size (pop. Size), the number of generations per run, the chain of matrices, the best fitness of the 10 runs, the optimal solution from the dynamic programming approach, and the average of number of generation that have the optimal solution.

From Table 3, it shows that for the chain of matrices greater than 9, when the number of matrices increases, the number of optimal solution decreases.

Pop. size	Gen.	Matrices	Best fitness	Optimal solution	Generations with Optimal solution
100	100	7	14125	14125	60
100	100	8	13250	13250	80
100	100	9	16250	16250	70
100	100	10	15625	15625	15
100	100	11	15675	15675	4
100	100	12	16755	16755	0.5
100	100	15	18215	18215	0.2
500	600	19	21820	21820	0.1
500	600	20	21,755	21,505	-
400	100	40	58,309	37,287	-

Table 3 comparison of the best fitness from GA with the optimal solution from dynamic programming

We also study on the chain of 20 matrices. We run the program with the population size 400, and 100 generations and with the population size 500, and 600 generations. The results of are shown on Figure 7 and 8.



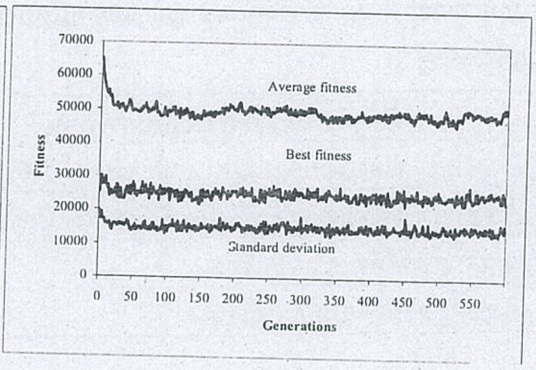


Figure 7 the GA performance of matrix-chain problem
for the chain of 20 matrices
with mutation and cross over,
the population size 400,
and the maximum number of generation 100.

Figure 8 the GA performance of matrix-chain problem
for the chain of 20 matrices
with mutation and cross over,
the population size 500,
and the maximum number of generation 600.

From figure 7 and 8, it shows that the average fitness value is dropped from the first 20 generations, and then it converges to around the fitness of 50000. We might or might not get the optimal solution if we increase the number of generations.

Table 3 shows that for the chain of matrices greater than 12, The optimal solution seldom happen. So we run the experiment for the chain of 19 and 20 matrices The best fitness values are shown on table 4 and 5. For the chain of 19 matrices, we lucky get the optimal solution. However, if we increase the number of generation and population size, we can get the better fitness value (close to the optimal solution).

From the experiment, it seems that the genetic algorithm with tree encoding might not suitable for the matrix chain multiplication problem. In order to get the optimal solution, we should run the program for many generations with large number of population size. While, running the dynamic programming give the result in a few second and certainly get the optimal solution. However, this is the good case study for representing the population with tree encoding in the genetic algorithm that have the fix number of leave node in the tree.

Run	g400p400	g500p400	g600p500
1	22220	22220	21945
2	21870	21945	21945
3	22095	22650	22300
4	22295	22270	22520
5	21995	21845	22020
6	22375	22195	**21820**
7	22295	22295	21970
8	22270	21995	22120
9	22020	22020	22020
10	22200	22145	22270
Average	22163.5	22158	22093

Table 4 The best fitness of matrix-chain problem for the chain of 19 matrices with the optimal solution 21,820

Run	g100p400	g600p500
1	22205	21855
2	22525	21730
3	22200	21880
4	22650	21855
5	22005	21755
6	21755	**21630**
7	22255	21855
8	22775	22005
9	22525	21680
10	22480	22005
Average	22337.5	21825

Table 5 The best fitness of matrix-chain problem for the chain of 20 matrices with the optimal solution 21,505

(Note g400p400 means the number of generation = 400, the population size = 400)

5. CONCLUSION

Using Genetic algorithm for solving complicated problem with a very large search space is becoming more widely used today since traditional method will require a lot of computation resource and in some problem with NP complexity, some problem seem to be unsolvable. The matrix chain problem is an interesting problem in calculating the cost for multiple matrix multiplication and we have yet found in nature any genetic algorithm method for solving them. In this paper, we present a genetic algorithm for solving the matrix-chain multiplication that use tree encoding, which is in the form of A_1, A_2, A_3 … A_n where A_i is a matrix. Individuals represented by trees, which are generated as the way the matrix, are fully parenthesized. We use cross over and mutation to generate the next generation as method of exploration, hopefully that the best solution could be reach. The fitness values then can be calculated and stored at the root of the tree. The worst case to calculate the fitness is O(N).

To apply the tree encoding with the matrix-chain multiplication is interesting. First we are dealing with the permutation of the shape of the tree. Second the element of the individual is fix because the order of the matrices must be constant. This is different from Koza's work [5] which he required the dynamic variability. It is difficult to implement the algorithm since we implement by using JAVA. We do not take the advantage of LISP programming like Koza's work. There are also several restrictions when the subtree are selected in both crossover and mutation operation. Furthermore the fitness values need to be recalculated in the bottom up fashion from leaves of the new subtree up until the root on every genetic operation.

6. ACKNOWLEDGEMENTS

We would like to thank Dr. Annie Wu for the insightful advice. We would also like to thank Dr. Nualsawat Hiransakolwong for valuable comments on early drafts of this paper.

REFERENCES

[1] Chandra, A.K. **1975** Computing matrix chain product in near optimal time. *Rep. RC-5626*, Thomas J. Watson Res. Ctr., Yorktown Heights, N.Y.

[2] Chin, F.Y. **1978** An O(n) algorithm for determining a Near-Optimal Computation Order of Matrix Chain Products. *Communications of the ACM, 21*(7).

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L. **1989** *Introduction to Algorithms.* MIT Press. McGraw-Hill.

[4] Hu, T.C., and Shing, M.T. **1982** Computation of Matrix Chain Products Part I. *SIAM J. computing, 11*(2).

[5] Koza, J.R. **1989** Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. *International Joint Conference on Artificial Intelligence.*