Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm

Conference Paper · January 2010

DOI: 10.1109/ICCEE.2009.25 · Source: IEEE Xplore

2 authors:

Reza Abbasian
University of Regina
7 PUBLICATIONS **94** CITATIONS
SEE PROFILE

Masoud Mazloom
University of Amsterdam
30 PUBLICATIONS **745** CITATIONS
SEE PROFILE

2009 Second International Conference on Computer and Electrical Engineering

Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm

Reza Abbasian
Department of Computer Engineering
Shahid Chamran University
Ahvaz, Iran
reza.abbasian@hotmail.com

Masoud Mazloom
Department of Computer Engineering
Shahid Chamran University
Ahvaz, Iran
mmazloom@scu.ac.ir

***Abstract*—Cryptarithmetic is a class of constraint satisfaction problems which includes making mathematical relations between meaningful words using simple arithmetic operators like 'plus' in a way that the result is conceptually true, and assigning digits to the letters of these words and generating numbers in order to make correct arithmetic operations as well. A simple way to solve such problems is by depth first search (DFS) algorithm which has a big search space even for quite small problems. In this paper we proposed a solution to this problem with genetic algorithm and then optimized it by using parallelism. We also showed that the algorithm reaches a solution faster and in a smaller number of iterations than similar algorithms.**

Keywords- constraint satisfaction; cryptarithmetic; parallel genetic algorithm

I. INTRODUCTION

Cryptarithmetic is a puzzle consisting of an arithmetic problem in which the digits have been replaced by letters of the alphabet. The goal is to decipher the letters (ie. map them back onto the digits) using the constraints provided by arithmetic and the additional constraint that no two letters can have the same numerical value [1]. This type of problem was popularized during the 1930s is the *Sphinx*, a Belgian journal of recreational mathematics [2]. One of the well-known Cryptarithmetic problems which published in the July 1924 issue of Strand Magazine by Henry Dudeney [3] is shown in Fig 1.

```
  SEND
+ MORE
--------
 MONEY
```

Figure 1. Cryptarithmetic problem example.

Assigning digits to letters in the following way would be an acceptable solution which is arithmetically correct.
O=0, M=1, Y=2, E=5, N=6, D=7, R=8 and S=9.
Hence the result would be as shown in Fig. 2.

```
  9567
+ 1085
--------
 10652
```

Figure 2. An acceptable solution to problem in Fig. 1.

Constraints of the Cryptarithmetic problem are as follows:

1. The arithmetic operations are in decimal; therefore, there must be maximum ten different letters in overall strings which are being used.
2. All of the same letters should be bound to a unique digit and no two different letters could be bounded to the same digit.
3. As the words will represent numbers, the first letter of them could not be assigned to zero.
4. The resulting numbers should satisfy the problem, meaning that the result of the two first numbers (operands) under the specified arithmetic operation (plus operator) should be the third number.

Genetic Algorithms (GAs) are search algorithms inspired by genetics and natural selection [4].
These algorithms are powerful search techniques that are used to solve difficult problems in many disciplines. Unfortunately, they can be very demanding in terms of computation load and memory. Parallel Genetic Algorithms (PGAs) are parallel implementations of GAs which can provide considerable gains in terms of performance and scalability. The most important advantage of PGAs is that in many cases they provide better performance than single population-based algorithms, even when the parallelism is simulated on conventional machines [5].

This paper proposes an efficient parallel genetic algorithm to solve decimal Cryptarithmetic problems and compares the proposed algorithm with ordinary ways to solve them.
First we will review some basics of GAs then in part III and IV of the paper, the proposed algorithm will be formulated and discussed in detail. At the end we will explore some experimental results, make a conclusion of our work and

978-0-7695-3925-6/09 $26.00 © 2009 IEEE
DOI 10.1109/ICCEE.2009.25

discuss the possibilities for further improvement of the proposed algorithm.

II. Basics of Genetic Algorithms

In the beginning, there are randomly generated individuals. All those individuals create a *population*. The population in certain time is called a *generation*. According to their qualities they are chosen by operators for creation of a new generation. The quality of the population grows or decreases and give limits to some constant. Every individual is represented by its *chromosome*. Mostly chromosomes represented as a binary string. Sometimes there are more strings which are not necessarily of a binary type. The chromosome representation could be evaluated by a *fitness* function. The fitness equals to the quality of an individual and is an important pick factor for a selection process. The average fitness of a population changes gradually during the run. Operating on the population, several operators are defined. After choosing randomly a pair of individuals, *crossover* executes an exchange of the substring within the pair with some probability. There are many types of crossovers defined, but a description is beyond the scope of this report. *Mutation* is an operator for a slight change of one individual/several individuals in the population. It is random, so it is against staying in the local minimum.

Low mutation parameter means low probability of mutation. *Selection* identifies the fittest individuals. The higher the fitness, the bigger the probability to become a parent in the next generation. There are different types of selection, but the basic functionality is the same [6].

III. Formulation of the Algorithm

A. Encoding Individuals

The first part of solving a problem using GA is to define a way to encode the individuals into chromosomes. Assume that we are working with decimal numbers, then we can use an array of characters with length 10 which has indices 0 through 9; whenever we need to assign a digit to a letter we simply put that letter to the cell which has that index of the array. For example in Fig. 3 we have assigned 3 to M.

We should leave unassigned columns of the array empty and to show that they are empty cells we marked it with "-"letter as shown in Fig. 3.

E	N	O	M	Y	S	D	R	-	-
0	1	2	3	4	5	6	7	8	9

Figure 3. A sample chromosome.

Since each chromosome represents a solution to the problem, changing the place of letters with each other or with empty cells may lead us to a different solution. See Fig. 4.

E	N	-	M	S	Y	-	R	D	O
0	1	2	3	4	5	6	7	8	9

Figure 4. Different solution after changing chromosome of Fig. 3

B. Fitness Function

Fitness function evaluates the quality of an individual. The fitness function for this problem is defined with the following formula:

$$FITNESS = |R - (F + S)|$$

Assuming R as the result of the operation, F as the first operand and S as the second operand.

When the fitness of individuals gets closer to zero we will get better individuals. If the fitness of an individual is exactly zero then that individual is the correct solution. Fig. 5 shows the fitness of the chromosome illustrated in Fig. 3.

```
  S E N D
  5 0 1 6
+ M O R E
  3 2 7 0
----------
  M O N E Y
  3 2 1 0 4
```

$$FITNESS = |32104 - (5016 + 3270)| = 23818$$

Figure 5. Fitness of the chromosome illustrated in Fig. 3.

C. Mutation

Mutation operator randomly generates two numbers between 0 and 9 and exchanges the content of the cells in these two indices of the chromosome. The mutation operator should ensure that the exchange is not illegal, which means it should not allow an exchange that would result positioning any start letter of the words in the zero index of the chromosome. Furthermore, a more efficient mutation operator should not allow the exchange of empty cells.

D. Making New Generations

We have used asexual genetic algorithm in this paper, therefore, we do not have a crossover with two parents and the method to make new generations is simply to use mutation operation on selected individuals.

E. Parallelization

Parallelization is applied to the process of creation of new generations. There is a coordinator thread which is in charge of organizing, finding and distributing the fittest individuals and some of generator threads which are responsible for making new generations. Each thread has an internal population. At the beginning each thread generates a random population of a fixed size and puts them into its internal population. As the thread generates a new individual, it places it in the right place of its internal

population by using insertion sort in an increasingly manner according to individuals' fitness. This way always the better individuals have lower indices and if there is a solution, it would be in index 0. When all of the threads generated new population, the coordinator thread picks a number of best individuals plus some randomly chosen individuals from each thread's internal population and accumulates them in a list and then distributes these chosen individuals over all the threads as the individuals that the threads use to generate a new population. This process goes on until the coordinator thread finds an individual with a fitness equal to zero during the process of picking best individuals out of each thread's internal population.

There are three parameters which should be set prior to the beginning of the algorithm: number of threads, internal population size and the number of chosen individuals for each thread.

1. *Number of Threads*
   The number of threads should be chosen wisely. It mostly depends on the size of the problem. If the problem is small and we have too many threads it slows down the computation because of its overheads and if we have a big problem, an insufficient number of threads slows down the computation again, and it will look like we are solving the problem sequentially.

2. *Internal Population Size*
   The population size of each thread depends on the number of the variables in the Cryptarithmetic problem. For problems with small number of variables we need smaller population size for each thread.

3. *Number of Chosen Individuals*
   It indicates the number of the individuals that the coordinator thread should pick from each thread's internal population to create a big population of best individuals per each iteration.

The three parameters described above are problem oriented. To get close to the best performance we should make a tradeoff between these three parameters for each instance of Cryptarithmetic problem.

IV. Proposed Algorithm

A. Main Algorithm

1. Put the coordinator thread in wait state.
2. Initialize N generator threads with an inner population of size P.
3. In each generator thread do the following:
   3.1. If the inner population is empty generate a random population then go to 3.3, else go to 3.2.
   3.2. Wait for a synchronization signal from coordinator thread and once it has been received, produce new children from the list of individuals which the coordinator has passed.
   3.3. Notify the coordinator that the current job is done and ready. Go to step 3.2.
4. Wait for all generator threads to notify the coordinator thread.
5. In coordinator thread, search the inner populations' first indices for the correct solution. If the solution has been found finish the algorithm else go to 6.
6. In coordinator thread, pick R best and some randomly chosen individuals from all inner populations of threads and create a list with them.
7. Distribute the chosen population to the generator threads, signal them, put coordinator thread in wait state and go to step 4.

B. Random Population Generation Algorithm

1. Extract distinct letters from input strings and put them in a list named L.
2. Repeat the following until the desired population size is reached:
   2.1. For each letter in L do the following:
      2.1.1. Generate a random number between 0 and 9.
      2.1.2. If the random number is equal to zero and the letter is the beginning letter of the words (which should not be zero) go to 2.1.4.
      2.1.3. If the cell that is corresponding the random number is empty, put the letter into that cell, else go to step 2.1.1.
      2.1.4. If the current letter is the last letter of L and the length of L is 10 then generate a random number between 1 and 9 and exchange the place of the letter from that cell to the index of zero and the current letter to that position. Keep doing it until the second letter is not a beginning letter of the words. Else go to step 2.1.1.
   2.2. Calculate the fitness of this new individual.
   2.3. Add the individual into the right place in the inner population list.

C. Making New Generation Algorithm

1. Get the chosen parent and generate two random numbers.
2. If the two random numbers are not the same or not indicating two empty cells in the parent and will not lead us to numbers beginning with zeros, exchanges the content of the cells corresponded by the random numbers. Else go to step 1.
3. Calculate the fitness of the new individual, put it in the right place in the inner population and remove the worst individual from the population.

V. Experimental Results

The algorithm is implemented with Java language and has been applied on different Cryptarithmetic problems. All the results were gathered on a machine with Intel Core 2 Duo CPU of 2.5 GHz, 3 GB of Memory and under the same version of the JDK (1.6). Each problem was tested a hundred times and all of their statistics have been gathered. Table I through VI show the results in detail.

TABLE I. Results of a 5 Variable Problem (per 100 runs).

Problem	DAN+NAN=NORA				
Answer	921+121=1042				
Vars	5				
Min Iters	**Max Iters**	**Ave Iters**	**Min Time (ms)**	**Max Time (ms)**	**Ave Time (ms)**
2	8	4	16	172	62

In the problem illustrated in Table I, 5 generator threads and 200 individuals in each thread's population were used.

TABLE II. Results of a 6 Variable Problem (per 100 runs).

Problem	TWO+TWO=FOUR				
Answer	836+836=1672				
Vars	6				
Min Iters	**Max Iters**	**Ave Iters**	**Min Time (ms)**	**Max Time (ms)**	**Ave Time (ms)**
2	4	3	57	224	148

In the problem illustrated in Table II, 12 generator threads and 300 individuals in each thread's population were used.

TABLE III. Results of a 7 Variable Problem (per 100 runs).

Problem	BASE+BALL=GAMES				
Answer	7483+7455=14938				
Vars	7				
Min Iters	**Max Iters**	**Ave Iters**	**Min Time (ms)**	**Max Time (ms)**	**Ave Time (ms)**
2	6	4	230	960	506

In the problem illustrated in Table III, 14 generator threads and 420 individuals in each thread's population were used.

TABLE IV. Results of an 8 Variable Problem (per 100 runs).

Problem	SEND+MORE=MONEY				
Answer	9567+1085=10652				
Vars	8				
Min Iters	**Max Iters**	**Ave Iters**	**Min Time (ms)**	**Max Time (ms)**	**Ave Time (ms)**
2	7	4	180	974	680

In the problem illustrated in Table IV, 16 generator threads and 480 individuals in each thread's population were used.

TABLE V. Results of a 9 Variable Problem (per 100 runs).

Problem	BASIC+LOGIC=PASCAL				
Answer	60852+47352=108204				
Vars	9				
Min Iters	**Max Iters**	**Ave Iters**	**Min Time (ms)**	**Max Time (ms)**	**Ave Time (ms)**
4	11	8	574	3342	2533

In the problem illustrated in Table V, 25 generator threads and 500 individuals in each thread's population were used.

TABLE VI. Results of a 10 Variable Problem (per 100 runs).

Problem	BROWN+YELLOW=PURPLE				
Answer	52813+649981=702794				
Vars	10				
Min Iters	**Max Iters**	**Ave Iters**	**Min Time (ms)**	**Max Time (ms)**	**Ave Time (ms)**
3	12	7	430	3632	2421

In the problem illustrated in Table VI, 25 generator threads and 500 individuals in each thread's population were used.

Another experiment was to measure the average time for solving Cryptarithmetic problems of different sizes. Various instances of different variable numbers solved 100 times and the average time for each problem size has been measured. The results of this experiment are shown in Table VII.

TABLE VII. THE RUNTIME RESULTS OF DIFFERENT PROBLEMS SIZES.

Variable Number	Ave. Time (ms)
5	112
6	325
7	877
8	1031
9	2756
10	3511

Fig. 6 illustrates the comparison between the proposed algorithm (PGA) and the depth first search algorithm (DFS). We have taken the worst case of blind search results into account and assumed that the computer is able to check 1000 solutions per second.
It is clear from Fig. 6 that the PGA has far better results than DFS. With problems with 10 variables the PGA reaches a solution in an average time of 3.5 seconds where it takes the DFS 60 minutes to find the answer!

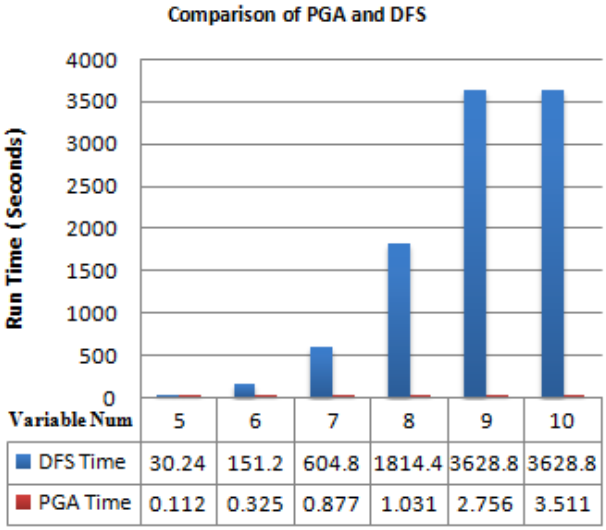

Figure 6. Comparison of PGA and DFS run times.

VI. CONCLUSION

This paper concentrated on solving Cryptarithmetic problems in an efficient way. The use of parallel genetic algorithm showed that we can even find the result of large instances of this problem within an acceptable time.

VII. FUTURE WORKS

The proposed algorithm in this paper has three initial parameters. Assigning these parameters is problem oriented and depends on the nature of the problem. Some works can be done to find a perfect mixture of these parameters for each specific Cryptarithmetic problem in order to get better results. Better mixtures can reduce the calculation time and the possible overhead of threads.

Furthermore, the operation can be extended to subtract, multiply and division and we can have more than two operands.

In this paper we only discussed Cryptarithmetic problems with decimal arithmetic operations. With a little change in the algorithm, we can even solve Cryptarithmetic problems of arbitrary bases.

REFERENCES

[1] Vinod Goel, Sketches of thought, MIT Press, 1995, pp. 87 and 88.

[2] Bonnie Averbach and Orin Chein, Problem Solving Through Recreational Mathematics, Courier Dover Publications, 2000, pp. 156.

[3] H. E. Dudeney, in *Strand Magazine* vol. 68 (July 1924), pp. 97 and 214.

[4] David Goldberg, "Genetic Algorithms in Search, Optimization *and Machine Learning,"* Addison-Wesley, Reading, MA 1989.

[5] Mariusz Nowostawski, Riccardo Poli, "Parallel Genetic Algorithm Taxonomy", KES'99.

[6] Konfrst, Z, "Parallel Genetic Algorithms: Advances, Computing Trends, Applications and Perspectives," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International 26-30 April 2004 Page(s):162.