An Evolutionary Algorithm to Solve Cryptarithmetic Problem

Abu Sayef Md. Ishaque, Md. Bahlul Haider, Muhammad Al Mahmud Wasid,
Shah Mohammed Alaul, Md. Kamrul Hassan, Tanveer Ahsan,
Mohammed Shamsul Alam

Abstract— Cryptarithmetic problem is an interesting constraint satisfaction problem for which different algorithms have been given. But none of them can solve the problem intelligently using small space and limited time. The paper illustrates the design and implementation of an efficient evolutionary algorithm that can find the solution more quickly taking less memory.

***Keywords*—**Constraint Satisfaction Problem (CSP), Cryptarithm, Cryptarithmetic, Evolutionary Algorithm, Greedy Algorithm, Heuristic.

I. Introduction

Cryptarithm is a genre of mathematical puzzle in which the digits are replaced by letters of the alphabet or other symbols. *Cryptarithmetic* is the science and art of creating and solving *cryptarithms*.

The world's best-known *Cryptarithmetic* puzzle is undoubtedly the puzzle shown in Figure I.

```
  S E N D
+ M O R E
---------
M O N E Y
```

Figure I: Cryptarithmetic puzzle

This was first introduced by H.E. Dudeney and was first published in the July 1924 issue of Strand Magazine associated with the story of a Kidnapper's ransom demand [1].

Modernization, by introducing computers and the Internet, is making quite an impact on *Cryptarithmetic* and it has already become a standard *AI* problem because it characterizes a number of important problems in computer science arena.

Manuscript received November 29, 2004.

Abu Sayef Md. Ishaque, GrameenPhone Ltd., Telenor affiliated Cellular Phone Company, Bangladesh. (Phone: +880-171-505-583; Fax: +880-2-9882-970; e-mail: abu.sayef@grameenphone.com).

Md. Bahlul haider, International Islamic University Chittagong (IIUC), Bangladesh. E-mail: bahlul_haider@yahoo.com.

Muhammad Al Mahmud Wasid, International Islamic University Chittagong (IIUC), Bangladesh. E-mail: om_an46@yahoo.co.in.

Shah Mohammed Alaul, International Islamic University Chittagong (IIUC), Bangladesh. E-mail: smrajib2000@yahoo.com.

Md. Kamrul Hassan, International Islamic University Chittagong (IIUC), Bangladesh. E-mail: pankha_52@yahoo.com.

Tanveer Ahsan, International Islamic University Chittagong (IIUC), Bangladesh. E-mail: tanveerahsan@yahoo.com.

Mohammed Shamsul Alam, International Islamic University Chittagong (IIUC), Bangladesh. E-mail: alam_cse@yahoo.com.

II. Study Of The Problem

Cryptarithmetic is a suitable example of *Constraint Satisfaction Problem*. Instead of providing description, a *Cryptarithmetic* problem can be better described by some constraints. The constraints of defining a *Cryptarithmetic* problem are as follows

--Each letter or symbol represents only one and a unique digit throughout the problem.

--When the digits replace letters or symbols, the resultant arithmetical operation must be correct.

These two constraints lead to some other restrictions in the problem. Consider that, the base of the numbers is *10*. Then there must be at most *10* unique symbols or letters in the problem. Otherwise, it would not be possible to assign a unique digit to each unique letter or symbol in the problem. To be semantically meaningful, a number must not begin with a zero. So, the letters at the beginning of each number should not correspond to zero.

III. Finding Solution By Searching

A *blind search* can eventually find the solutions, if there is any, in a bounded time. Given that the base of the number is *10*, there may be ${}^{10}p_n$ solutions to be checked in the problem space; where *n* is the number of unique letters or symbols in the problem. As, the problem shown in the Figure 01 has eight different letters *S, E, N, D, M, O, R* and *Y, blind search* might have to search at most ${}^{10}p_8=1814400$ solutions. If the searching algorithm can check *1000* solutions per second, yet it will take about 30 minutes to get the solution in the worst case.

A *rule based searching technique* can provide the solution in minimum time. Constructing these rules is problem specific and is out of the scope of this paper.

The *toughness* of the problem can be interpreted as less number of valid solutions in relatively huge problem space. The precedence of mathematical operation in the problem indicates a basic estimation of *toughness*. Again, if there are few letters in the problem, the problem space becomes smaller which makes the problem easy. Given the number of letters in the problem, a problem can be very tough yet. This is because some of the letters or symbols can occur more than once in the problem. These critical letters, the letters that occur more than once in the problem can introduce more restrictions in the problem and reduce the number of valid solutions.

An *Evolutionary Algorithm* will search for solutions in shortest time but the performance will also reflect the *toughness* of the problem.

 © 2004 WORLD ENFORMATIKA SOCIETY

IV. Evolutionary Approach

An *Evolutionary Algorithm (EA)* is a common term for algorithms that utilize the adaptive behavior modeled after principles of nature [2].

Although definition of *Evolutionary Algorithm* differs, the more common properties of *EAs* are that collections of potential solutions to the problem at hand are maintained. These solutions are referred to as the *population* of a current *generation*. Each potential solution is called a *chromosome*. Operations are applied to the current *population* to produce a new *generation* that will hopefully contain *chromosomes* that are better solutions of the problem. This process continues until some *threshold* value or *stopping criterion* is met.

The new *population* is produced through the operators on selected *chromosomes* of current *generation*. Typically, the *chromosomes* of the current *generation,* to whom the operators are applied, are chosen based on their quality. In this way, it is more likely that the offspring chromosomes inherit desirable characteristics of its parents. Some *heuristics* or *fitness functions* are used to choose parent *chromosomes* in a *generation*.

Typically operators are developed from the genetic notation of *mutation*, *recombination* (or *crossover*) and *inversion* [3]. As an example, a *mutation* operator can choose two random numbers that may be applied to randomly change some inherited characteristics of the *chromosome*. *Crossover* operations involve combining characteristics from two or more parents and placing them in the resulting offspring.

V. Formulation Of Algorithm

A. Encoding solution to chromosome

The initial part of solving a problem in *Evolutionary Algorithm* is the encoding of solutions into *chromosomes*. The *chromosomes* should be simple enough so that operations can be executed easily and descriptive enough to characterize a unique solution. In this problem, letters should be assigned unique digits to characterize a solution. In this paper we will only discuss about the decimal solutions. So, a string of length *10* field by those letters and additional *don't care symbol(s)* is a good choice. If the base of the number is *n* then we will use an array of length *n*. The position of the letters in the string denotes their value.

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

Figure II: Chromosome

In Figure 02, '_' (Underscore) is used as a *don't care* symbol and the numbers just under the letters denote their position in the string as well as their respective values in the solution. Here, 8 and 9 are not used for any letter and the corresponding positions are occupied by *don't care* symbols. Operators may change the position of the letters and *don't care* symbol and the *chromosome* will indicate a completely different solution.

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

Figure III: Chromosome after operation(s)

So, Figure 03 indicates a different solution as the position of letters has been changed.

B. Mutation Operation

A *mutation* operator will randomly generate two different numbers from 0 to 9 and will exchange the positions of the letters in these two indexes. This will correspond to a completely different solution.

```
Y N R M _ O E D _ S
0 1 2 3 4 5 6 7 8 9
        (a)
Y N R M O _ D E _ S
0 1 2 3 4 5 6 7 8 9
        (b)
Y N E M _ O D R _ S
0 1 2 3 4 5 6 7 8 9
        (c)
```

Figure IV: Three chromosomes after Mutation on Figure 03

Here the only restriction is that the letter that comes at the starting of a number in the problem should not be the starting letter of the *chromosome*. In the problem of Figure 01 *S* and *M* cannot be *0*. This is because they are the starting numbers in the problem. A more productive mutation operator ensures that it will not exchange the position of two *don't care* symbol with each other.

Similar cross-over operation can be designed where two chromosomes exchange their attributes. This can be useful in moving faster towards the solution especially at the middle of searching when selected chromosomes adopt some of the good attributes. But for some tough problems it makes the algorithm more susceptible to *local minima deadlock*. So, cross-over operation may not be included in a general algorithm.

C. Fitness Function

A *Fitness function* usually indicates the degree of correctness of a *chromosome*. An evaluation function can be easily formulated which will calculate the error of mathematical result in the problem.

```
      S E N D
      9 7 1 6
    + M O R E
      3 5 2 7
  ------------
      M O N E Y
      3 5 1 7 0
ERROR = ABS (35170-(9716+3527)) = 21927
```

Figure V: Calculating Error in chromosome of Figure 03

The error in the *chromosome* of Figure 03 is shown in figure 05. Now this can be thought as negative fitness. *Chromosome* with minimum error is the fittest *chromosome* in a *generation*. This algorithm aims to minimize this value. When the value is zero, the solution is found. By only a few modifications in the fitness function we can find the solutions of Cryptarithmetic with more than two operands in the left hand side and with more operators like subtraction, division, multiplication. We can solve the problem `SIX + SEVEN + SEVEN = TWENTY`. Here we have three operands in the left hand side. So for this problem the fitness function will be as follows

```
ERROR= ABS( TWENTY-( SIX + SEVEN + SEVEN )).
```

D. Improving Generation through Iteration

Initially we have only one *chromosome* that is shown in Figure 02. We will first use this *chromosome* to generate the next *generation*. By *mutation* operation this *chromosome* generates a new *generation*,

 © 2004 WORLD ENFORMATIKA SOCIETY

which represents current *population*. Then we will take some fittest *chromosomes* from the current *generation* to produce the next *generation*. In iterative process, the fittest *chromosomes* of a generation get the chance to generate the offspring *generation*.

To ensure that the offspring *generation* is not worse than the current one, the fittest *chromosome* of the current *generation* can be added to the next *generation*. But, this will introduce a *deadlock* when these fittest *chromosomes* cannot lead to a solution, even though there exists a solution. So, a random *chromosome* can be given chance to contribute to the next *generation*. This modification will avoid the *deadlock*.

The number of *chromosomes* that will be kept in one generation is problem specific. If we take more *chromosomes* then we will need more computation and if we take less *chromosomes* then we might not be able to reach the solution in bounded time. After generating several generations we will find the solution when the error in the *chromosome* is *0*. The solution *chromosome* of Figure 01 is shown in the Figure 06.

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

                S E N D
                9 5 6 7
              + M O R E
                1 0 8 5
              -----------
              M O N E Y
              1 0 6 5 2
ERROR = ABS (10652-(9567+1085)) = 0
```

Figure VI: The solution chromosome

VI. Algorithm

Step 1: Scan the input strings.

Step 2: Put the letters or symbols in *ARRAY[10]*

Step 3: If number of distinct letter is less than *10* then fill rest of the indices of *ARRAY* by don't care symbols. This *ARRAY[10]* now is our current *generation*.

Step 4: For several times *generate* two random numbers *m, n* and swap the contents of index *m* and *n* of any one *chromosome* of current *generation* and copy this new *chromosome* to next *generation*.

Step 5: Evaluate the fitness of each *chromosome* of next *generation* and choose the best *chromosomes*. Now these best *chromosomes* become our current *generation*. Also include one random *chromosome* to the current *generation*. If there is no *chromosome* with error 0 in the current *generation* then go to step 4. If one of the *chromosomes* is found with error 0 then report the solution and exit.

VII. Performance

This algorithm is applied on different *Cryptarithmetic* problems and good result has been observed. For particular *SEND + MORE = MONEY* problem it checks about *1669* solutions on an average whereas blind search requires about $^{10}p_8/2=907200$ solutions to be checked on an average. The number of solutions to be checked varies because of the random property of this algorithm. If we take a closer look at the algorithm we will find out why this algorithm can find the solution quickly. In the problem of Figure 01 we see that we are adding two four digit numbers *(SEND+MORE)* and the result is a five digit number *(MONEY)*. From our mathematical experience we can say that if we add two *n* digit numbers and the result is *n+1* digit number, then the most significant digit of the result should be *1*. Our algorithm can find out that the most significant digit of the result is *1* just after the first iteration. This is because any value other than *1* in the most significant digit in the result will produce larger error values. So our algorithm will reject those *chromosomes* that have value other than *1* for the most significant digit of the result.

According to *toughness* of particular problem, few parameters like number of *chromosomes* in each *generation*, number of *chromosomes* selected for *reproduction* and random *chromosomes* taken to avoid *local minima* should be set to get solution in a minimal search.

Few common unique solution problems are analyzed with 16 *chromosomes* in each *generation* and 4 *chromosome* reproduction within which 1 was taken randomly and statistics shown in Table I have been found in 1000 runs.

Problem	*Min*	*Max*	*Avg*
SEND+MORE=MONEY	240	9248	1669
9567+1085=10652			
BASIC+LOGIC=PASCAL	240	379520	10521
60852+47352=108204			
APPLE+LEMON=BANANA	336	54288	5476
67794+94832=162626			
HACK+HACKER=REBOOT	320	112560	2326
4968+496805=501773			
EARTH+URANUS=SATURN	160	512000	49062
94583+654067=748650			
COMET+SATURN=URANUS	1232	481584	96297
61078+298354=359432			
BROWN+YELLOW=PURPLE	288	512000	87279
52813+649981=702794			

Table I: Statistics of searching some standard problems

VIII. Limitation

This is a *greedy algorithm* as it is taking the best *chromosomes* at each *iteration* and tries to converge to the nearest solution. Like other *greedy approaches*, it may stick at some *local minima*. To overcome these *local minima* we have used a random *chromosome* at each generation. This will guarantee us that we will be able to overcome the *local minima* and reach our *global minima*.

IX. Conclusion

This paper concentrated on designing an efficient *Evolutionary algorithm* to solve *cryptarithmetic Problem.* Additionally, it illustrates how to plug in techniques of *Evolutionary Approach* into *Constraint Satisfaction Problem.* This sort of design can provide efficient solution to a wide range of *Constraint Satisfaction Problem* or other generic searching problems that could be characterized as a *Constraint Satisfaction Problem* as well. So, further research should go on to design efficient solutions of different real-life searching problems.

References

[1] http: // www.geocities.com / Athens / Agora / 2160 / index.html

[2] Evolutionary Algorithm Approach for Symbolic FSM Traversals, IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), August 26-28, 2001, pp. 506-509, (with R. Drechsler).

[3] R. Drechsler, *Evolutionary Algorithm in VLSI CAD*, Kluwer Academic Publishers, Boston Hardbound, ISBN 0-7923-8168-8, May 1998.

 © 2004 WORLD ENFORMATIKA SOCIETY