Combining Genetic Algorithms and Simulated Annealing for Constructing Lotto Designs

(Ben) Pak Ching Li
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada R3T 2N2
lipake@cs.umanitoba.ca

Abstract

An $LD(n,k,p,t;b)$ Lotto design is a set of b k-sets (blocks) of an n-set such that any p-set intersects at least one block in t or more elements. Let $L(n,k,p,t)$ denote the minimum number of blocks for any $LD(n,k,p,t;b)$ Lotto design. This paper describes an algorithm used to construct Lotto designs by combining genetic algorithms and simulated annealing and provides some experimental results.

1 Introduction

Let X be a set of n distinct elements (called an n-set). An $LD(n,k,p,t;b)$ Lotto design $(X,\mathcal{B})$ with is a collection $\mathcal{B}$ of b k-sets (blocks) from X such that any p-set intersects at least one block of $\mathcal{B}$ in t or more elements. Let $L(n,k,p,t)$ denote the minimum size of any (n,k,p,t) Lotto design. Given parameters n,k,p,t, a p-set P is said to be *represented* by a k-set K if $|K \cap B| \geq t$. A p-set P is said to be *represented by a collection* of k-sets, if there exists some set K in the collection that represents P. It is clear that any $LD(n,k,p,t;b)$ Lotto design is a collection of b k-sets that represent every p-set of X. Figure 1 shows an $LD(8,4,4,3;6)$ Lotto design and Figure 2 shows an example of a $LD(13,6,5,3;5)$ Lotto design.

The most obvious application of Lotto designs is the study of lotteries. In Canada, for example, the lottery Lotto 6/49 allows a player to buy tickets where each ticket contains 6 numbers between 1 and 49. At a later time, the government will randomly pick 6 numbers. If a player holds one

JCMCC **45** (2003), pp. 109-121

1	2	3	6
1	2	3	7
1	2	4	5
3	4	5	8
1	6	7	8
5	6	7	8

Figure 1: An $LD(8, 4, 4, 3; 6)$ Lotto Design

1	2	3	4	5	6
1	2	3	4	5	7
1	2	3	4	6	7
1	2	3	5	6	7
8	9	10	11	12	13

Figure 2: An $L(13, 6, 5, 4; 5)$ Lotto Design

or more tickets that matches the government's pick in 3 or more numbers, the player has won a prize. For example, a player owning a ticket that matches the government's pick in 3 numbers will received a prize of \$10. Lotto designs may be used to investigate how many (and which tickets) the player needs to purchase to guarantee that at least one of the purchased tickets match the government's pick in at least 3 numbers. The value, $L(49, 6, 6, 3)$ informs the player of the minimum number of tickets he/she needs to purchase to ensure matching the government's pick in 3 numbers. In order to determine what the actual tickets are, we need to be able to construct a Lotto design with $L(49, 6, 6, 3)$ blocks. A good introduction to Lotto designs can be found in [2].

We can construct Lotto designs using exhaustive or heuristic searches. Given values n, k, p and t and the number of blocks b desired, an exhaustive search will decisively determine if an $LD(n, k, p, t; b)$ Lotto design with b blocks exist. If a design with these parameters does exist, the search will be able to generate the blocks of the design and we can conclude that $L(n, k, p, t) \leq b$. Otherwise, no such design exists with these parameters and we may conclude that $L(n, k, p, t) > b$. Exhaustive searches are usually implemented as backtracking algorithms. While exhaustive searches always give you a decisive answer on whether a design with the stated parameters exist, it may not be feasible to apply this approach except for a small number of choices for the parameters, where either n or b is small. This is due to the fact that an exhaustive search has to consider every possible

combination of b blocks.

A heuristic search will try to construct a design with the given parameters. If it is able to construct such such a design, then the blocks will typically be returned by the search algorithm and we can conclude that $L(n, k, p, t) \leq b$. However, if the search fails, we cannot conclude that no such design exists with the given parameters. This is because heuristic searches do not traverse the entire search space. Instead, they look at a subset of the entire search space. Examples of heuristic search algorithms are hill climbing, tabu search, simulated annealing and genetic algorithms. Some reasons for using heuristic searches are that they finish much quicker than exhaustive searches, and they often find a solution that we desire. A good introduction to exhaustive and heuristic algorithms can be found in [5]. [1] is an excellent reference on heuristic search algorithms.

In this paper, we present another heuristic search technique that may be used to construct Lotto designs. These heuristics work with feasible solutions. In the case of Lotto designs, a *feasible solution* is simply any collection of b k-sets. This hybrid search algorithm combines the two heuristic techniques: genetic algorithms and simulated annealing. The genetic algorithm, simulated annealing and the hybrid genetic simulated annealing algorithms will be described in the next section. We will provide some experimental results for this hybrid algorithm.

2 Algorithms

In this section we will state and describe each of the following heuristic algorithms for constructing Lotto designs:

- Simulated Annealing
- Genetic Algorithm
- Genetic Simulated Annealing

2.1 Simulated Annealing

Simulated annealing is a combinatorial optimization method. Simulated annealing is based on the physical process of annealing where a crystal is cooled down from the liquid phase to the solid phase in a heat bath. By performing the cooling process in a careful manner, the energy state of the solid at the end of the cooling process is very near or at it minimum.

The idea of simulated annealing is taken from the Metropolis algorithm [9]. An important component of simulated annealing is its ability to accept a higher cost solution with some probability. This allows the algorithm to possibly escape from local minimas. For a comprehensive treatment of the simulated annealing algorithm, see , [1], [8]. Algorithm 2.1 is a simulated annealing algorithm for constructing and $LD(n, k, p, t; b)$ Lotto design. It is based on the algorithm found in [10].

Algorithm 2.1: SIMULATEDANNEALING(n, k, p, t, b)

Generate initial solution S and temperature T.
while not frozen
do
  while inner loop criterion not satisfied
  do
    Select a neighbor S^* of S.
    $\delta \leftarrow cost(S^*) - cost(S)$.
    if $\delta \leq 0$
      then $S \leftarrow S^*$
    if $\delta > 0$
      then $S \leftarrow S^*$ with probability $e^{\delta/T}$.
  Reduce temperature T.
return (Best Solution Found and its Cost)

The input parameters to Algorithm 2.1 are the values of n, k, p, t and b for the Lotto design in question. The goal of the simulated annealing algorithm is to try to construct a $LD(n, k, p, t; b)$ Lotto design. The cost function in Algorithm 2.1 is dependent on the problem being solved. For Lotto designs, we chose the cost of a solution S, denoted by $\mathbf{cost}(S)$, to be the number of p-sets that are not represented by the solution S. If all p-sets are represented by the solution, then the solution has cost 0 and the solution a Lotto design. The temperature T is used to implement a cooling schedule which is typically decreased in an exponential fashion, that is, $T_{i+1} = rT_i$ where r is a constant between 0.95 to 0.99. As the temperature decreases, the probability of accepting a worse-off solution (that is, one with higher cost) decreases. The *neighbors* for a solution is also dependent on the problem being solved. For Lotto designs, we define S^* to be a neighbor of S if and only if S and S^* differ in only one block and these two block differ in only one element. Using this neighborhood system, it is clear that each solution has exactly $bk(n-k)$ neighbors. We say that the annealing process is *frozen* if the cost function at the end of the inner **while** loop

is the same for a given number of consecutive temperatures. The inner **while** loop is typically executed a constant number of times, based on the size of the neighborhood. Finally, the selection of the initial temperature is important. Nurmela and Östergard [10] described a method of selecting an initial temperature based on the number of cost increasing moves around the initial solution. We have implemented this method for computing the initial temperature.

It is clear that Algorithm 2.1 works with a single feasible solution at a time and its ability to make "large" jumps in the solution space is limited. None the less, experiments have shown that this algorithm works quite well for finding Lotto designs.

We have implemented our simulated annealing algorithm similar to that found in [10]. There are many implementation details that we will not discuss here, but instead refer the reader to [10].

2.2 Genetic Algorithms

Genetic algorithms (GAs) are heuristic search methods that may be used to solve search and optimization problems. Genetic algorithms were introduced by Holland [4] in 1975. They are based on the process of Darwinian evolution: over many generations, the "fittest" individuals tend to dominate the population. Genetic algorithms are often more attractive than gradient search methods because they do not require complicated differential equations or a smooth search space. A GA simulates this evolutionary process by manipulating information encoded as *chromosomes*. Each chromosome is made up of pieces of information, known as *genes*. A collection of genes makes up a chromosome and a collection of chromosomes makes up a population. Feasible solutions of the problem being solved are encoded as chromosomes. There are various ways to represent a chromosome. Typically, integers are used to represent a chromosome. The representation of chromosomes is very problem-dependent, since a chromosome is used to encode a feasible solution for the problem being solved. In the case of Lotto designs, each chromosome has length b, where each gene in a chromosome contains an integer between 0 and the number of k-sets, indicating the rank (or ordering) of a k-set in the feasible solution encoded by the chromosome. A rank can be assigned to a k-set in various ways [5]. We have chosen to assign ranks to k-sets using lexicographic ordering of the k-sets [2]. Hence a chromosome encodes the collection of blocks in the feasible solution by storing the blocks' ranks (sorted in ascending order). A population of chromosomes is then a collection of feasible Lotto designs or solutions, each with b blocks.

A GA evolves a population by mixing the genes of the chromosomes in the population with some probability, using a crossover operator, and by manipulating the individual genes of each chromosome with some probability, using a mutation operator. If $g_1 = \{x_1, x_2, \ldots, x_b\}$ and $g_2 = \{y_1, y_2, \ldots, y_b\}$ are two chromosomes, then the crossover operator used in your implementation GA will generate two chromosomes $h_1 = \{x_1, x_2, \ldots, x_i, y_{i+1}, y_{i+2}, \ldots, y_b\}$ and $h_2 = \{y_1, y_2, \ldots, y_i, x_{i+1}, x_{i+2}, \ldots, x_b\}$, where i is a random number between 1 and $b-1$.This is often called a "single point crossover". The mutation operator that we used in our GA is one which moves a gene to a neighbor (if we think of the gene as a k-set), using the standard neighborhood system for a Lotto design which is described in Section 2.1.

Each chromosome is assigned an objective value and a fitness value. The *objective value* (or cost) of a chromosome g is the value returned by your objective function; it is the raw performance evaluation of a chromosome. We will use $\mathsf{cost}(g)$ to denote this value. For Lotto designs, we define the objective value of a chromosome to be the number of p-sets not represented by the solution encoded in the chromosome. This is the cost function used in Algorithm 2.1. The *fitness score*, on the other hand, is a possibly-transformed rating used by the genetic algorithm to determine the fitness of individuals for mating. The fitness value of a chromosome is typically linearly scaled over all the fitness values in the population. The GA will use the fitness value of the chromosomes in the population to determine which chromosomes will proceed to the next generation, either in its original form or in some mutated form. It should be noted that the choice of a fitness function is problem-dependent. However, chromosomes with higher fitness values are more likely evolve to the next generation while chromosomes with low fitness values tend not to evolve to the next generation. Many methods exist to determine which genes move onto the next generation. We will give a detail description of the fitness and objective functions in Section 2.3.

Several parameters may determine the outcome of a GA: the number of chromosomes in the population, the number of generations, the probability of performing a crossover operation, the probability of performing a mutation operation. During each generation, we have decided that the number of genes in the population should be the same. The probability of crossover determines how often two chosen chromosomes should exchange genetic information. The probability of mutations determines how often a chromosome should mutate or change its value. For more information about genetic algorithms, see [1],[3].

We now state a generic GA for constructing Lotto designs:

Algorithm 2.2: GA($n, k, p, t, b, pCross, pMut, popSize, maxGen$)

$\mathcal{G} \leftarrow$ initial solution with $popSize$ chromosomes
for $i \leftarrow 1$ **to** $maxGen$
do
- Compute cost and fitness values for each chromosome in $\mathcal{G}$
- Create a new empty population pool $\mathcal{G}^*$
- **while** $|\mathcal{G}^*| < |\mathcal{G}|$
  - **do**
    - Select $g_1, g_2 \in \mathcal{G}$ based on fitness values
    - Apply crossover with probability $pCross$ to get h_1, h_2
    - Mutate each gene of h_1 with probability $pMut$
    - Mutate each gene of h_2 with probability $pMut$
    - $\mathcal{G}^* \leftarrow \mathcal{G}^* \cup \{h_1, h_2\}$
- $\mathcal{G} \leftarrow \mathcal{G}^*$

return (Best solution found)

Algorithm 2.2 operates on many feasible solutions at the same time, whereas simulated annealing works with only one feasible solution at a time. It is also clear the crossover operator allow the algorithm to enforce chromosomes to make a "big" jump in the solution space, while the mutation operator allow the algorithm to enforce chromosomes to make a "small" localized jump in the solution space. Unfortunately, the genetic algorithm does not allow for local searching. The mutation operator does move a chromosome in small steps, but it is applied at most once on each gene of the chromosome. To alleviate this shortcoming, we will incorporate simulated annealing into Algorithm 2.2. This algorithm is described in the following sub-section.

2.3 Genetic Simulated Annealing

As mention in Section 2.2, genetic algorithms does not allow for detailed examination of a the search space around a feasible solution. Thus, it is possible that a feasible solution is close to a global minimum, but cannot reach it due to the inability of a genetic algorithm to perform a local search. On the other hand, simulated annealing does not have the ability to consider multiple feasible solutions or perform "large" jumps in the search space. The deficiencies simulated annealing and genetic algorithms led us to develop a hybrid algorithm that may overcome these deficiencies. [7] describes a hybrid genetic simulated annealing algorithm for non-slicing

floor-plan design. We will present a similar algorithm for constructing Lotto designs.

Algorithm 2.3 states our genetic simulated annealing algorithm for constructing Lotto designs, which is similar to the algorithm in [7]. It should be noted that it is slightly different from Algorithm 2.2. Algorithm 2.3 applies the crossover operator (with probability $pCross$) to exactly two chromosomes during each generation. In addition, the mutation process in Algorithm 2.2 has been replaced by the application of simulated annealing to the two chromosomes created by the crossover operation.

Algorithm 2.3: GSA($n, k, p, t, b, pCross, pMut, popSize, maxGen$)

$\mathcal{G} \leftarrow$ initial solution with $popSize$ chromosomes
Compute cost and fitness values for each chromosome in $\mathcal{G}$
for $i \leftarrow 1$ **to** $maxGen$
do
- $g_w \leftarrow$ chromosome in $\mathcal{G}$ with lowest fitness value
- Select $g_1, g_2 \in \mathcal{G}$ based on fitness values such that $g_1 \neq g_w$, $g_2 \neq g_w$
- Apply crossover on g_1, g_2 with probability $pCross$ to get h_1, h_2
- Apply Algorithm 2.1 with initial solution h_1
- Record best solution s_1 constructed by Algorithm 2.1
- Apply Algorithm 2.1 with initial solution h_2
- Record best solution s_2 constructed by Algorithm 2.1
- Replace g_w with either s_1 or s_2 based on fitness value

return (Best solution found)

If we let $numPSets$ denote the total number of possible p-sets, then we defined the fitness value of a chromosome g as

$$fitness(g) = \frac{1 - \frac{cost(g)}{numPSets}}{\sum_{h \in \mathcal{G}} \left(1 - \frac{cost(h)}{numPSets}\right)}.$$

where $\mathcal{G}$ is the current population pool of size $popS$. It is clear that the lower the cost (or objective value) of a chromosome g, the higher its fitness value. In addition, $0 \leq fitness(g) \leq 1$ and

$$\sum_{g \in \mathcal{G}} fitness(g) = 1.$$

Using a uniformly distributed random number generator, the choosing of the two chromosomes g_1 and g_2 in step 2b will be biased toward those chromosomes with higher fitness values.

The simulated annealing part of Algorithm 2.3 is described in Section 2.1 and has the same implementation. The details of the actual implementation of the hybrid algorithm will not be discussed here, as it could be implemented in many different ways.

3 Experimental Results and Observations

The experimental results show that our hybrid genetic simulated annealing method is actually quite effective. These tests have been performed using the set of Lotto design parameters in Table 1. The settings for the genetic simulated annealing algorithm are given in Table 2.

Test Case	n	k	p	t	b
1	15	5	4	3	24
2	12	7	5	5	59
3	15	4	4	3	52
4	14	3	4	3	144
5	19	6	2	2	15
6	14	6	5	4	31
7	12	5	3	3	29
8	13	5	3	3	34
9	13	4	3	3	78
10	13	5	5	4	49
11	12	6	6	5	38

Table 1: Designs Parameters Used For Experiments

Population size	20
Crossover probability	0.6
Crossover operator	single crossover
Maximum generations	2000

Table 2: Genetic Simulated Annealing Parameters

For comparison purposes, we report results for each test case generated by simulated annealing and genetic simulated annealing. These two

algorithms were implemented using a common set of routines and data structures. For both algorithms, we ran trials of up to 2000 generations for each test case. For simulated annealing, this meant running the algorithm up to 2000 times for each test case.

In Table 3, the first column refers to the designs tested. Column 2, list the two algorithms. Columns 3 states if the desired design was found. Column 4 gives the number of generations required to obtain the that design respectively. If the value of column 3 is "No", then the value of column 4 will be filled with the value 2000.

Case	Methods	Desired Design Found	Generations Taken
1	GSA	Yes	1514
	SA	No	2000
2	GSA	Yes	109
	SA	No	2000
3	GSA	Yes	896
	SA	No	2000
4	GSA	Yes	3
	SA	Yes	2000
5	GSA	Yes	1
	GSA	Yes	1
6	SA	No	2000
	GSA	No	2000
7	GSA	Yes	5
	SA	Yes	10
8	GSA	Yes	420
	SA	No	268
9	GSA	No	2000
	SA	No	NA
10	GSA	Yes	150
	SA	Yes	1523
11	GSA	Yes	14
	SA	Yes	234

Table 3: Results of Test Runs

For test case 1, the genetic simulated annealing algorithm discovered a new upper bound (24) for $L(15, 5, 4, 3)$. This was obtained at generation 1514. It is interesting that this Lotto design was obtain from the combining of two other chromosomes. The previously best known upper bound for $L(15, 5, 4, 3)$ was 25 blocks.

For cases 2 and 3, we see that the genetic simulated annealing algorithm was able to generate a design with the specified parameters faster than simulated annealing. In fact, simulated annealing was not able to generate a Lotto design for either cases in 2000 generations.

For case 4 and 7, both algorithms were able to generate the desired design in roughly the same number of generations.

For case 5, both simulated annealing and genetic simulated annealing was able to generate the desired Lotto design in the first generation.

For cases 6 and 9, neither algorithm was able to generate the desired design within 2000 generations.

For case 8, simulated annealing was able to generate the desired design in less generations than the hybrid algorithm.

For cases 10 and 11, both algorithms were able to generate the desired design. The hybrid algorithm was able to generate the designs in less generations than the simulated annealing algorithm. In addition, both algorithms found a better upper bound (49 blocks) for $L(13, 5, 5, 4)$ than previously known, which was 50 blocks.

From these tests, we see that the hybrid algorithm can typically find a design with cost at least as good as that found by simulated annealing alone. The hybrid algorithm also finds designs faster than simulated annealing in many cases. Another observation from our experiments is that the objective values of the chromosomes in the final generation tend to be roughly the same. This is reasonable, since as the algorithm progresses, the weaker chromosomes are thrown away and replaced with chromosomes that have better objective values.

4 Conclusions and Future Work

We have described and implemented a simple algorithm based on genetic algorithm and simulated annealing for constructing Lotto designs. This algorithm operates on a collection of feasible solutions, and promotes the sharing of information between these solutions. From running some limited experiments, we were able to see that the algorithm does as well as simulated annealing in most cases, when the performance analysis is based on the cost of the best design found and the number of generations required to found a desired design. In some cases, for example $LD(15, 5, 4, 3; 24)$, it was able to construct a Lotto design where simulated annealing was not able to construct (in our tests runs), with about the same number of iterations.

We have not studied how the sharing of information between chromosomes in a population would aid in the construction of Lotto designs. We would like to investigate this sharing of information in the near future. In addition, we would like to investigate the use of other crossover and mutation operators in the hybrid algorithm. It is known that, the choice of the crossover and mutation operators greatly affect the performance of genetic algorithms. Finally, we have only considered one way of representing a Lotto design as a chromosome. We need to investigate if there are other representations exist and if so, how does it affect the performance of the algorithm.

Acknowledgments

This research was supported by a University of Manitoba, Faculty of Science Startup Grant. I would like to thank D.L. Kreher and D.R. Stinson for the **pseudocode** LaTeX style file [6], which was used for typesetting the algorithms in this paper.

References

[1] E. Aarts, J.K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley and Sons (1997).

[2] C.J. Colbourn, J.H. Dinitz. *The CRC Handbook of Combinatorial Designs*. CRC Press (1996).

[3] K. DeJong. *Learning with Genetic Algorithms : An Overview*. Machine Learning **3** (1988), 121-138.

[4] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI (1975).

[5] D.L. Kreher, D.R.Stinson. *Combinatorial Algorithms - Generation, Enumeration and Search*. CRC Press (1999).

[6] D.L. Kreher, D.R.Stinson. *A LaTex Style File for Displaying Algorithms*. Bulletin of the ICA **30** (1999), 11-24.

[7] S. Koakutsu, M. Kang, W.W.M Dai. *Genetic simulated annealing and application to non-slicing floorplan design*. Proc. 5th ACM/SIGDA Physical Design Workshop (Virginia, USA), (1996), 134-141.

[8] P.J.M van Laarhoven. E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Reidel. Dordrecht. 1987.

[9] N. Metropolis. A. Rosenbluth. M. Rosenbluth. A. Teller. E. Teller. *Equation of State Calculations by Fast Computing Machines.* Journal of Chemical Physics **21** (1953). 1087-1092.

[10] K.J. Nurmela. P.R.J. Östergard. *Upper Bounds for Covering Designs by Simulated Annealing.* Congressus Numerantium **96** (1993). 93-111.