Level-Dependent Experimental Optimization

Iliya Bluskov
Department of Mathematics
and Computer Science
University of Northern British Columbia
Prince George, B.C., Canada V2N 4Z9

February 22, 2002

Abstract

In this paper we discuss a self-adjusting and self-improving combinatorial optimization algorithm. Variations of this algorithm have been successfully applied in recent research in Design Theory. The approach is simple but general and can be applied in any instance of combinatorial optimization problem.

1 Combinatorial Optimization Problems and Local Search Algorithms

We start with a discussion on combinatorial optimization problems and the computational methods that are used for solving them. A combinatorial optimization problem deals with a set S of combinatorial objects called **solutions.** Every solution $i \in S$ possesses a number of properties. The question is to find a solution having an additional property, called **optimality**. Optimality can be defined by introducing a function $c : S \to R$ from the set of solutions to the real numbers, called the **cost function**. An **optimal solution** is a solution $i^* \in S$ such that $c(i^*) \leq c(i)$ for all $i \in S$. We formally define a **combinatorial optimization problem** (COP) as a pair (S, c) and the question of finding an optimal solution. The usual approach to solving a COP is by moving from one solution to another trying to reduce the value of the cost function until it reaches its global minimum, thereby producing an optimal solution.

In order to describe a combinatorial optimization algorithm we need some further definitions. A **move** is a function $d : S(d) \to S$, where $S(d) \subseteq S$ is the domain of d. Let D be the set of all moves of a COP. We require that the union of the domains of all the moves in D is the solution set, that is,

$$\bigcup_{d \in D} S(d) = S$$

(so that for every solution $s \in S$ there is at least one $d \in D$ so that $d(s) \in S$). A solution s' is a **neighbor** of the solution s, if $s' = d(s)$ for some $d \in D$. The **neighborhood** $N(s)$ of a solution s is the union of all neighbors of s:

$$N(s) = \bigcup_{\{d \in D | s \in S(d)\}} d(s).$$

Various optimization algorithms have been described in the literature. The differences amongst these algorithms arise in the way the algorithm moves from a current solution to another solution in the neighborhood of the current solution. Some of the widely used algorithms include random walk, simulated annealing, tabu search, steepest descent algorithm (sometimes called hill climbing), threshold accepting, great deluge algorithm, record-to-record travel, etc. ([1], [6]), generally referred to as **local search algorithms**. Each algorithm of this class can be characterized by the following common features:

1) examines one solution at a time,

2) keeps a single solution in memory, and

3) each time the current solution s is changed to s', it is true that $s' \in N(s)$.

Local search algorithms have been used to study many problems in constructive combinatorics. Amazingly, objects of quite precise structure can be found by these methods. The author's experience include some successful searches in [2], [3], [4] and [5].

In spite of the success of the known local search algorithms, there are some natural restrictions on the range of application, as well as some common problems with the performance of the algorithms, for example:

1) Possibility of getting stuck in a local minimum. This is a feature of simple algorithms (such as random walk and hill climbing, for example).

2) Necessity of adjusting the (many) parameters of the algorithm in order to achieve better convergence. This could be a problem with more complicated algorithms, such as simulated annealing, for example.

In this paper we suggest a general procedure which avoids getting stuck in a local minimum while being simple enough to require no adjustments, except in the initial (fast) stage. Needless to say, quite naturally, an algorithm with these qualities considerably extends the possible range of applications. Below are some observations that led to the creation of the new efficient algorithm.

The best optimization processes allow using sideways (cost-preserving) moves or even cost-increasing moves to avoid getting stuck in a local minimum. Most of the algorithms, however, do not have a good strategy on when or how often to allow cost-increasing moves. Decisions as to when to accept a cost-increasing move are usually based on some remotely similar natural process (simulated annealing, great deluge) or on some prespecified mathematical function. None of the known algorithms addresses the specifics of the particular *COP*.

There are not many reasons why an algorithm based on some natural process should work well in a particular *COP*. Good convergence depends considerably on the specifics of the problem. Hence the best approach to finding a good convergence algorithm would be to base the algorithm on experiments performed on the particular solutions of the problem and use the results of the experiment to measure how often an acceptance of cost-increasing moves is needed in order to obtain a good convergence.

We next describe the idea of the method in more precise terms.

2 The Idea of a Level-Dependent Experimental Optimization (LDEO)

We will use the following conventions: Solutions having the same cost will be referred to as **solutions having the same level.** Given a solution B, **a good neighbour of** B is any solution $B' \in N(B)$ with $c(B') \leq c(B)$. A cost-decreasing or cost-preserving move will be called a **good move.** In this section we give a general idea of the approach. We call it a level-dependent experimental optimization, because the strategy of moving from one solution to another is developed through a preliminary experiment performed on the solutions of the particular *COP*. The experiment is usually a rough optimization process that generally cannot find a global minimum, but can establish the average number of moves needed to find a good neighbour at any level with the exception of the lowest few levels. Let C be the set of all possible costs. By studying the data from the experiment we establish a function $f : C \to \mathcal{Z}^+$, where, for every $c \in C$, $f(c)$ is an approximation to the average number of moves needed to find a good neighbour at level c. The experiment produces values of f for all but the few lowest costs. We extend the function to the lowest costs using the results of the experiment. We call this function the **jump-down function.**

The following scheme can be used to extend the jump-down function to the lowest level. Suppose the *COP* has n possible levels and let c_i be the cost at level i, $i = 1, 2, ..., n$. Let $a_i = f(c_i)$. Suppose that the lowest level reached by the experiment is the $(k-1)$th, so we have the values $a_n, a_{n-1}, ..., a_k$ of the jump-down function. A possible extension of the jump-down function can be defined by

$$a_{k-1-i} = \left\lceil \frac{\frac{a_{n-1}}{a_n} + \frac{a_{n-2}}{a_{n-1}} + \dots + \frac{a_{k-i}}{a_{k-i+1}}}{n-k-i} a_{k-i} \right\rceil, \quad i = 0, 1, \dots, k-2.$$

Let B be a particular solution of our COP. Let $g(B, c(B))$ denote the number of good neighbours of B. Clearly, $0 \leq g(B, c(B)) \leq |N(B)|$. The probability of choosing a good move is $\frac{g(B,c(B))}{|N(B)|}$. Hence testing $\left\lceil \frac{|N(B)|}{g(B,c(B))} \right\rceil$ randomly chosen neighbours of B provides a fair chance of finding a good neighbour. Now, let $S_c \subseteq S$ be the set of all solutions of cost c. Then

$$\frac{\sum_{B \in S_c} \frac{g(B,c(B))}{|N(B)|}}{|S_c|}$$

is the average probability of finding a good neighbour at the level c. We denote this probability by p_c. Testing at most $\frac{1}{p_c}$ neighbours of B before accepting a cost-increasing solution yields fairly good convergence of the optimization process. Testing less than $\frac{1}{p_c}$ neighbours makes it difficult to get to lower levels and increases the chance that the process will never converge. A process that uses more than $\frac{1}{p_c}$ moves will reach lower levels, but it will be slower with greater chance of getting stuck in a local minimum, that is, again, there will be a possibility that the process will never converge.

We will further assume that we have established a jump-down function, that is, we know an approximation $f(c)$ to $\frac{1}{p_c}$ for every $c \in C$.

3 The Algorithm

A description of an optimization algorithm starts by defining a cost function and a move. It then continues with specifying a sequence of operations that would eventually produce an optimal solution.

Assuming that a cost function and a move have been defined and a jump-down function is known, a description of the optimization process is given by the following.

1. Start with any solution B. Set *counter* $:= 0$.
2. If [*counter* $\leq f(c(B))$ and $c(B) > 0$] then
   (a) Perform a single move d to obtain a new solution $B' = d(B) \in N(B)$. Set *counter* $:=$ *counter* $+ 1$.
   (b) If $c(B') \leq c(B)$ then set $B := B'$, set *counter* $:= 0$ and go to 2. Otherwise, go to (a).

3. If [*counter* $> f(c(B))$ and $c(B) > 0$] then set $B := B'$, set *counter* := 0 and go to 2). (Accept a cost-increasing solution and continue the process.)

4. Stop. ($c(B) = 0$, that is, a global minimum has been found.)

In other words: Try at most $f(c(B))$ neighbours of B in order to find a good one. If a good neighbour is found, accept it and continue the process. If no good neighbour is found after $f(c(B))$ moves, then accept a cost-increasing solution and continue the process.

Here we assume without loss of generality that the minimum cost is 0. Note that we can impose different stop criterion: For example, we can search for non-optimal solution.

4 Establishing of an Initial Jump-Down Function: The Preliminary Experiment

We can now take a closer look on the preliminary experiment. We use the same algorithm as above except that the stop criterion is not at a global minimum, but at some easily reachable low level and the jump-down function is more or less arbitrarily defined. In fact, the entire process of adjusting the jump-down function can also be programmed. One can take the initial jump-down function to be some general exponential function of the cost. For example, in the searches for designs in [2], we start with an initial jump-down function

$$f(c(B)) = \left\lceil \frac{|N(B)|}{2^{\frac{c(B)}{2}-1}} \right\rceil$$

for $c(B) \geq 2$. (The neibourhood in these applications has a constant size.)

Suppose the algorithm has entered a certain level L and has already accepted $d-1$ cost-preserving solutions before finding a cost-decreasing solution (the dth one). Let p_i, $i = 1, 2, ..., d$, be the number of moves made after accepting the $(i-1)$th, but before accepting the ith good neighbour. The sum $p_1 + p_2 + ... + p_d$ gives the number of moves tried in order to find a cost-decreasing solution. Hence an approximation to the average number of moves needed to find a good solution at the level L is

$$\frac{p_1 + p_2 + ... + p_{d-1} + (p_1 + p_2 + ... + p_d)}{d}.$$

Suppose the algorithm has visited n times the level L during a single run. This allow us to determine a better jump-down function. Define $s_j = 2p_1 + 2p_2 + ... + 2p_{d_j-1} + p_{d_j}$, $j = 1, 2, ..., n$, where d_j is defined as the d

above, but corresponds to the jth visit to level L. An approximation to the average number of moves needed to find a good solution at the level L would then be

$$\frac{\sum_{j=1}^{n} s_j}{\sum_{j=1}^{n} d_j}.$$

A jump-down function takes on the values which are the approximations for all visited levels during a single run of the algorithm. Once a jump-down function is accepted according to the experiment (the first run), then a new run of the algorithm can be used to possibly both "jump-down" to lower levels and determine more precise jump-down function, using the accumulated data from the two runs. Seemingly, the entire procedure can be programmed into one self-improving optimization process, where the jump-down function is periodically updated (based on the accumulated data of all runs). The self-improvability concerns the fact that the jump-down function improves with any new run (more and more data have been examined). We should mention that for the application discussed in [2] only two or three adjustments (runs) were needed for a good convergence state.

5 Conclusion

In this article we discussed a new optimization method, LDEO. The method can be applied in any *COP*, especially for problems where we are only interested in finding a global minimum. Based on the author's experience, LDEO outperforms the existing methods in both convergence and running time. The justification of the good performance of LDEO comes from the fact that it uses a self-adjusting and problem-dependent jump-down function instead of prespecified jump-down function as in the known algorithms.

Acknowledgments

This research was partially supported by NSERC grant 228274-00.

References

[1] E. H. L. Aarts and J. K. Lenstra, Introduction, in *Local Search in Combinatorial Optimization*, pages 1-17, John Wiley and Sons Ltd., 1997.

[2] R.J.R. Abel, I. Bluskov and M. Greig, Balanced Incomplete Block Designs with Block Size 8, *Journal of Combinatorial Designs* 9(2001), 233-268.

[3] I. BLUSKOV, Optimization Algorithms and Cyclic Designs, *Journal of Geometry*, 67(2000), 42-49.

[4] I. BLUSKOV AND HEIKKI HÄMÄLÄINEN, New Upper Bounds on the Minimum Size of Covering Designs, *Journal of Combinatorial Designs*, vol.6, No.1 (1998), 21-41.

[5] I. BLUSKOV AND S. MAGLIVERAS, On the Number of Mutually Disjoint Cyclic Designs and Large Sets of Designs, *Journal of Statistical Planning and Inference*, 95(2001), 133-142.

[6] K. J. NURMELA, Constructing Combinatorial Designs by Local Search, *Research Report A27* , Digital Systems Laboratory, Helsinki University of Technology, 1993.