Global Scientific JOURNALS

GSJ: Volume 9, Issue 1, January 2021, Online: ISSN 2320-9186
www.globalscientificjournal.com

COMPLEXITY ANALYSIS OF ITERATED LOCAL SEARCH ALGORITHM FOR OPTIMIZING LATIN HYPERCUBE DESIGNS

Parimal Mridha, Lecturer, Mathematics Department,
Military Collegiate School Khulna (MCSK), Bangladesh.
parimalmridha@yahoo.com

Abstract: *Computer experiments involve a large numbers of variables, but only a few of them have no negligible influence on the response. As is recognized by several authors, the choice of the design points for computer experiments should fulfill at lest two requirements – space-filling and non-collapsing. Unfortunately, randomly generated Latin Hypercube Designs (LHDs) almost always show poor space-filling properties. On the other hand, maximin distance designs have very well space-filling properties but often show poor projection properties under the Euclidean or the Rectangular distance. To overcome this shortcoming, Morris et al. have suggested to search for maximin LHDs when looking for "optimal" designs. It is shown that the Iterated Local search(ILS) approach not only able to obtain good LHDs in the sense of space-filling property but the correlations among the factors are acceptable i.e. multi-collinearity is not high. Anyway from the point of view of computational complexity the problem is open. When numbers of factors as well as number of experimental points are large, the heuristic approaches also require a couple of hours or even more to find out a simulated optimal design. So time complexity is an important issue for a good algorithm. Specially for the need of real time solution, the time complexity of the ILS approaches is analyzed. The inner most view as well as the effect of the parameters of the algorithms have been observed and have been analyzed.*

Keywords: *ILS approach, Latin Hypercube design, Space-filling, multi-collinearity.*

1. INTRODUCTION

In general usage, **complexity** tends to be used to characterize something with many parts in intricate arrangement. Complexity Theory is concerned with the study of the *intrinsic complexity* of computational tasks. Its ``final" goals include the determination of the complexity of any well-defined task. Additional ``final" goals include obtaining an understanding of the relations between various computational phenomena (e.g., relating one fact regarding computational complexity to another). Indeed, we may say that the former type of goals is concerned with *absolute* answers regarding specific computational phenomena, whereas the latter type is concerned with questions regarding the *relation* between computational phenomena.

Interestingly, the current success of Complexity Theory in coping with the latter type of goals has been more significant. In fact, the failure to resolve questions of the ``absolute" type, led to the flourishing of methods for coping with questions of the "relative" type.

In general, Computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A computational problem is understood to be a task that is in principle amenable to being solved by a computer, which is equivalent to stating that the problem may be solved by mechanical application of mathematical steps.

Anyway there are two type of complexity regarding time and space. Time complexity is concerned with the analysis of the elapsed time of an algorithm; whereas, how much memory required is discussed in space complexity.

1.1. Some Definitions:

Some of these concept and respective are offered below:

(i) Time Complexity: A measure of the amount of time required to execute an algorithms is called time complexity. Later time complexity is discussed elaborately.

(ii) Space Complexity: The (space) complexity of a program (for a given input) is the number of elementary objects that this programs needs to store during its execution. This number is computed with respect to the size n of the input data. We thus make the assumption that each elementary object needs the same amount of space.

(**iii) Turing Machine:** Turing machines provide a model of digital computational which is more primitive, hence harder to 'program" than random access machines. However, their primitiveness becomes an advantage when they are manipulated for the purpose of proving theorical results.

A *Turing machine* is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

1.2 Time Complexity: Time Complexity comparisons are more interesting than space complexity. The programming language chosen to implement the algorithm should not affect in time complexity analysis. There are some other factors that should not affect in time complexity are-: the quality of the compiler, the speed of the computer on which the algorithm is to be executed.

The objectives of the time complexity analysis are to determine the feasibility of an algorithm by estimating an upper bound on the amount of work performed. Objectives of the time complexity analysis are also to compare different algorithms before deciding on which one to implement.

Time complexity analysis is based on the amount of work done by the algorithm. It expresses the relationship between the size of the input and the run time for the algorithm. Time complexity is usually expressed as proportionality, rather than an exact function.

To simplify analysis, we sometimes ignore work that takes a constant amount of time, independent of the problem input size. When comparing two algorithms that perform the same task, we often just concentrate on the differences between algorithms.

For time Complexity, simplified analysis can be based on:

(i) Number of arithmetic operations performed

(ii) Number of comparisons made

(iii) Number of times through a critical loop

(iv) Number of array elements accessed, etc.

1.3. Constant Time Complexity:

Algorithms whose solutions are independent of the size of the problem’s inputs are said to have constant complexity. It is denoted as 1(O).

Example:

Suppose that exponentiation is carried out using multiplications. Two ways to evaluate the polynomial

$P(x) = 4x^4 + 7x^3 - 2x^2 + 3x^1 + 6$

In Brute force method:

$P(x) = 4*x*x*x*x + 7*x*x*x - 2*x*x + 3*x + 6$

In Horner’s method:

$P(x) = (((4 * x + 7) * x - 2) * x + 3) * x + 6$

1.4 Measuring Time Complexity:

The worst-case time complexity of an algorithm is expressed as a function $T : N \rightarrow N$
Where T(n) is the maximum number of "steps" in any execution of the algorithm on inputs of "size" n. Intuitively, the amount of time an algorithm takes depends on how large is the input on which the algorithm must operate: Sorting large lists takes longer than sorting short lists; multiplying huge matrices takes longer than multiplying small ones. The dependence of the time needed to the size of the input is not necessarily linear: sorting twice the number of elements takes quite a bit more than just twice as much time; searching (using binary search) through a sorted list twice as long, takes a lot less than twice as much time. The time complexity function expresses that dependence. Note that an algorithm might take different amounts of time on inputs of the same size. We have defined the worst-case time complexity, which means that we count the maximum number of steps that any input of a particular size could take. For example, if the time complexity of an algorithm is $3n^2$, it means that on inputs of size n the algorithm requires up to $3n^2$ steps. To make this precise, we must clarify what we mean by "input size" and "step".

(i) Input Size: We can define the size of an input in a general way as the number of bits required to store the input. This definition is general but it is sometimes inconvenient because it is too low-level. More usefully we define the size of the input in a way that is problem-dependent. For example, when we are dealing with sorting algorithms, it may be more convenient to use the number of elements we want to sort as the measure of the input size. This measure ignores the size of the individual elements that are to be sorted.
Sometimes there may be several reasonable choices for the size of input. For instance, if we are dealing with algorithms for multiplying square matrices, we may express the input size as the dimension of the matrix (i.e., the number of columns or rows), or we may express the input size as the number of entries in the matrix. In this case the two measures are related to each other (the latter is the square of the former). One conclusion from this discussion is that in order to properly interpret the function that describes the time complexity of an algorithm we must be clear about how exactly we measure the size of inputs[Nicolas, (2007)].
(ii) Step: A step of the algorithm can be defined precisely if we fix a particular machine on which the algorithm is to be run. For instance, if we are using a machine with a Pentium processor, we might define a step to be one Pentium instruction. This is not the only reasonable choice: different instructions take different amounts of time, so a more refined definition might

be that a step is one cycle of the processor's clock. In general, however, we want to analyze the time complexity of an algorithm without restricting ourselves to some particular machine. We can do this by adopting a more flexible notion of what constitutes a step. In general, we will consider a step to be anything that we can reasonably expect a computer to do in a fixed amount of time. Typical examples are performing an arithmetic operation, comparing two numbers, or assigning a value to a variable.

1.5 Big-O Notation:

Big O notation (with a capital letter O, not a zero), also called **Landau's symbol**, is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions. Basically, it tells us how fast a function grows or declines.

Landau's symbol comes from the name of the German number theoretician Edmund Landau who proposed the notation. The letter O is used because the rate of growth of a function is also called its *order*.

For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by $T(n) = 4n^2 -2n +2$. If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms, we could say "$T(n)$ grows at the *order* of n^2 and write: $T(n) = O(n^2)$.

1.6. Iterated Local Search

The Latin hypercube design is a popular choice of experimental design when computer simulation is used to study a physical process. A number of methods have been proposed [Lournce et al.(2002), Martin and Otto(1996) for extending the uniform sampling to higher dimensions.

The importance of high performance algorithms for tackling difficult optimization problems cannot be understated, and in many cases the only available methods are metaheuristics. The word metaheuristics contains all heuristics methods that show evidence of achieving good quality solutions for the problem of interest within an acceptable time. Metaheuristic techniques have become more and more competitive. When designing a metaheuristic, it is preferable that it be simple, both conceptually and in practice. Naturally, it also must be effective, and if possible, general purpose. The main advantage of this approach is the ease of implementation and the quickness.

As metaheuristics have become more and more sophisticated, this ideal case has been pushed aside in the quest for greater performance. As a consequence, problem-specific knowledge (in addition to that built into the heuristic being guided) must now be incorporated into metaheuristics in order to reach the state of the art level. Unfortunately, this makes the boundary between heuristics and metaheuristics fuzzy, and we run the risk of loosing both simplicity and generality.

Here a well known metaheuristics approaches, namely general Iterated Local Search (ILS)has been discussed. Iterated Local Search is a metaheuristic designed to embed
another, problem specific, local search as if it were a black box. This allows Iterated Local Search to keep a more general structure than other metaheuristics currently in practice.

The essence of metaheuristic - the iterated local search - can be given in a nut-shell: one iteratively builds a sequence of solutions generated by the embedded heuristic, leading to far better solutions than if one were to use repeated random trials of that heuristic. This simple idea [Baxter (1981)] has a long history, and its rediscovery by many authors has lead to many different names for iterated local search like *iterated descent* [Baum (1986a), Baum (1986b)], *large-step Markov chains* [Martin et al. (1991)], *iterated Lin-Kernighan* [Johnson (1990)], *chained local optimization* [Martin and Otto (1996)], or *combinations of these* [Applegate et al. (1999)]. There are two main points that make an algorithm an iterated local search: (i) there must be a single chain that is being followed (this then excludes population-based algorithms); (ii) the search for better solutions occurs in a reduced space defined by the output of a black box heuristic. In practice, local search has been the most frequently used embedded heuristic, but in fact any optimizer can be used, be-it deterministic or not.

The purpose of this review is to give a detailed description of iterated local search and to show where it stands in terms of performance. So far, in spite of its conceptual simplicity, it has lead to a number of state-of-the art results without the use of too much problem-specific knowledge; perhaps this is because iterated local search is very malleable, many implementation choices being left to the developer. In what follows we will give a formal description of ILS and comment on its main components.

```
**Procedure** *Iterated Local Search*
    $s_0$ = Generate Initial Solution
    $s^*$ = Local Search($s_0$)
    **repeat**
    $s'$ = Perturbation($s^*$)
    $s^{*\prime}$ = Local Search($s'$)
    $s^*$ = Acceptance Criterion ($s^*$, $s^{*\prime}$)
```

```
until     termination condition met
end
```

Local search applied to the initial solution s_0 gives the starting point s^* of the walk in the set S^*. Starting with a good s^* can be important if high-quality solutions are to be reached as fast as possible. The initial solution s_0 used in the ILS is typically found one of two ways: a random starting solution is generated or a greedy construction heuristic is applied. A “random restart” approach with independent samplings is sometimes a useful strategy (in particular when all other options fail), it breaks down as the instance size grows because in that time the tail of the distribution of costs collapses. A greedy initial solution s_0 has two main advantages over random starting solutions: (i) when combined with local search, greedy initial solutions often result in better quality solutions s^*; (ii) a local search from greedy solutions takes, on average, less improvement steps and therefore the local search requires less CPU time.

The current s^*, we first apply a change or perturbation that leads to an intermediate state s' (which belongs to S where S is set of all local optimum). Then Local Search is applied to s' and we reach a solution $s^{*\prime}$ in S^*. If $s^{*\prime}$ passes an acceptance test, it becomes the next element of the walk in S^*; otherwise, one returns to s^*. The resulting walk is a case of a stochastic search in S^*, but where neighborhoods are never explicitly introduced. This iterated local search procedure should lead to good biased sampling as long as the perturbations are neither too small nor too large. If they are too small, one will often fall back to s^* and few new solutions of S^* will be explored. If on the contrary the perturbations are too large, s' will be random, there will be no bias in the sampling, and we will recover a random restart type algorithm will be recovered.

In practice, much of the potential complexity of ILS is hidden in the history dependence. If there happens to be no such dependence, the walk has no memory: the perturbation and acceptance criterion do not depend on any of the solutions visited previously during the walk, and one accepts or not $s^{*\prime}$ with a fixed rule. This leads to random walk dynamics on S^* that are “Markovian”, the probability of making a particular step from s_1^* to s_2^* depending only on s_1^* and s_2^*. Most of the wFork using ILS has been of this type, though the studies show unambiguously that incorporating memory enhances performance [Stutzle (1998)].

The main drawback of any local search algorithm is that, by definition, it gets trapped in local optima that might be significantly worse than the global optimum. The strategy employed by ILS to escape from local optima is represented by perturbations to the current local optima. The perturbation scheme takes a locally optimal solution, s^*, and produces another solution from which a local search is started at the next iteration. Hopefully, the perturbation will return a solution outside the basins of attraction of previously visited local minima. That is, it will be "near" a previously unvisited local optimum. Choice of the correct perturbation scheme is of primary importance, because it has a great influence on the intensification/diversification characteristics of the overall algorithm. Generally, the local search should not be able to undo the perturbation; otherwise one will fall back into the local optimum just visited. Perturbation schemes are commonly referred to as "strong" and "weak", depending on how much they affect the solution that they change. A perturbation scheme that is too strong has too much diversity and will reduce the ILS to an iterated random restart heuristic. A perturbation scheme that is too weak has too little diversity and will result in the ILS not searching enough of the search space.

2. **Maximin Latin Hypercube Designs:**

We will denote as follows the s-norm distance between two points x_i and x_j, $\forall\ i, j = 1, 2, \cdots, N$:

$$d_{ij} = \| x_i - x_j \|_s \qquad (1)$$

Unless otherwise mentioned, we will only consider the Euclidean distance measure ($s = 2$). In fact, we will usually consider the squared value of d_{ij} (in brief d), i.e. d^2 (saving the computation of the square root). This has a noticeable effect on the execution speed since the distances d will be evaluated many times.

2.1 Definition of LHD:

A Latin Hypercube Design (LHD) is a statistical design of experiments, which was first defined in 1979 [McKay et al. (1979)]. An LHD of k-factors (dimensions) with N design points, $x_i = (x_{i1}, x_{i2} \cdots x_{ik}) : i = 0, 1, \ldots, N-1$, is given by a $N\times k$- matrix (i.e. a matrix with N rows and k columns) X, where each column of X consists of a permutation of integers $0, 1, \cdots, N-1$ (note that each factor range is normalized to the interval $[0, N-1]$) so that for each dimension j all x_{ij} , $i = 0, 1, \cdots, N-1$ are distinct. We will refer to each row of X as a (discrete) design point and each column of X as a factor (parameter) of the design points.

We can represent $\boldsymbol{X}$ as follows

$$\boldsymbol{X} = \begin{pmatrix} x_{\mathbf{0}} \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} x_{01} & \cdots & x_{0k} \\ \vdots & \cdots & \vdots \\ x_{(N-1)1} & \cdots & x_{(N-1)k} \end{pmatrix} \qquad (2)$$

such that for each $j \in \{1, 2 \cdots, k\}$ and for all $p, q \in \{0, 1, \cdots, N-1\}$ with $p \neq q;\ x_{pj} \neq x_{qj}$ holds.

Given a LHD X and a distance d, let

$$D = \{d(x_i, x_j) : 1 \leq i < j \leq N\}.$$

Note that $|D| \leq \binom{n}{2}$. We define $D_r(X)$ as the r-th minimum distance in D, and $J_r(X)$ as the number of pairs $\{x_i, x_j\}$ having $d\ (x_i, x_j) = D_r(X)$ in X.

The maximin LHD problem aims at finding a LHD X^* such that $D_1(X)$ is as large as possible. However, a search which only takes into account the D_1 values is certainly not efficient. Indeed, the landscape defined by the D_1 values is "too flat". For this reason the search should be driven by other optimality criteria, which take into account also other values besides D_1.

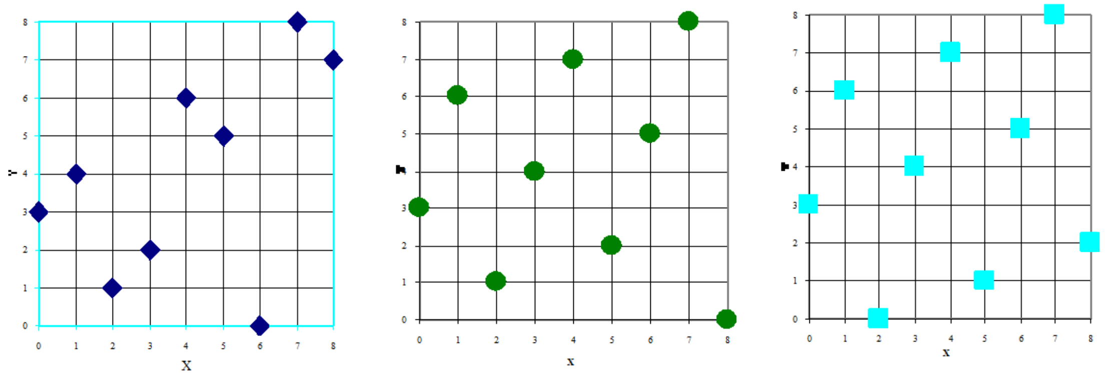

Fig: (a) $D_1(X_r)=2$, $J_1(X_r)=4$ Fig: (b) $D_1(X_{sm})=8$, $J_1(X_{sm})=4$ Fig: (c) $D_1(X_M)=8$, $J_1(X_M)=2$

Figure 2.1: Some LHDs and their corresponding (D_1, J_1) values

3.1 Optimality Criteria

In order to drive the search through LHDs we need some criterion to compare them. Below we will describe some of the criteria employed in the literature.

Opt(D_1, J_1) Optimality Criterion : Under this criterion a LHD Y can be considered better than another one X if a lexicographic ordering holds:

$$\begin{aligned} &D_1(Y) > D_1(X) \quad \text{or} \\ &D_1(Y) = D_1(X) \quad \text{and} \quad J_1(Y) < J_1(X). \end{aligned} \tag{3}$$

We illustrate this optimality criterion as follows. In Figure 2.1(a) X_r is a randomly generated LHD with $(N, k,) = (9,2)$ where $D_1(X_r) = 2$ and $J(X_r) = 4$; Figure 2.1 (b) presents an improved configuration X_{sm} where $D_1(X_{sm}) = 8$ with $J(X_{sm}) = 4$. A third LHD X_M is given in Figure 2.1 (c) where $D_1(X_M) = 8$ and $J_1(X_M) = 2$; by the Opt(D_1,J_1) criterion this is the best configuration among the three.

By generalizing this approach, we can consider the problem like a multi-objective problem with priorities: maximize the objective with highest priority D_1; within the set of optimal solutions with respect to D_1, minimize the objective with second highest priority J_1. Note that Johnson et al. [Johnson et al. (1990)] first proposed this optimality criterion.

Opt(φ) Optimality Criterion : As previously remarked, if there exist different LHDs with equal D_1 and J_1 values, i.e. in case there exist at least two LHDs X, Y such that $D_1(X) = D_1(Y) = D_1$ and $J_1(X) = J_1(Y) = J_1$, we could further consider the objective D_2 and maximize $D_2(X)$, the second smallest distance in X, and, if equality still holds, minimize $J_2(X)$, the number of occurrence of $D_2(X)$, and so on. Then an optimal design X sequentially maximizes D_{is} and minimizes J_{is} in the following order: D_1, J_1; D_2, J_2, · · · ,D_m, J_m. Morris and Mitchell [Morris and Mitchell (1995)] have used all the above measures to define a family of scalar-valued functions (to be minimized), which can be used to rank competing designs in such a way that a maximin design receives the highest ranking. This family of functions, indexed by p, is given by

$$\phi_p(x) = \sum_{r=1}^{m} \left[\frac{J_r(X)}{(D_r(X))^P} \right]^{1/P} \quad (4)$$

where p is a positive integer parameter. Under this criterion, LHD Y is better than X if

$\phi_P(Y) < \phi_R(X)$.

Note that for large enough p, each term in the sum in (3.4) dominates all subsequent terms. Through p we can control the impact of the different D_r distances: as p increases, the impact of distance D_1 becomes more and more relevant. In the form (3.4), the evaluation of ϕ_p would be computationally costly. However, it has a computationally cheaper form (see [Jin et al. (2005)]). Indeed, (3.4) can be simplified as

$$\phi_P(X) = \left[\sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{1}{d_{ij}^{P}} \right]^{\frac{1}{P}} \quad (5)$$

which can be computed without the need of detecting and ordering all the D_i values.

An apparent drawback of the Opt(ϕ) criterion, if we are interested in maximin values (maximum D_1 value), is that LHDs with smaller (better) ϕ_p can have a worse(smaller) D_1, i.e. we can have X and Y such that $\phi_P(X) < \phi_P(Y)$ and $D_1(X) < D_1(Y)$. This phenomenon has been frequently observed in our computational experiments. Nevertheless, a profitable choice is to work in order to minimize the ϕ_p function but, at the same time, keep track of the best (D_1, J_1) values observed during such minimization. This way the search in the solution space is guided by a kind of heuristic function. Such mixed approach might appear strange but, as we will demonstrate experimentally, it can be extremely effective.

While the two criteria above are strictly related to maximin values and they will be widely employed in the definition of approaches for detecting maximin solutions, for the sake of completeness, we also mention that also other optimality criteria, not necessarily related with maximin values, are available in the literature. We present a couple of them as well as the approaches for constructing the optimal Latin hypercube design in Table 3.1.

Table 3.1: Some well know approaches as well as optimal criterion for optimal experimental designs

Researchers	**Year**	**Algorithm**	**Objective functions**
Audze and Eglajs	1977	Coordinates Exchange Algorithm	Potential Energy
Park	1994	A 2-stage(exchange-and Newton-type) algorithm	Integrated mean squared error and entropy criteria
Morris and Mitchell	1995	Simulated annealing	ϕ_p criterion
Ye et al.	2000	Columnwise-pairwise	ϕ_p and entropy criteria
Fang et al.	2002	Threshold accepting algorithm	Centered L_2-discrepancy
Bates et al.	2004	Genetic algorithm	Potential energy
Jin et al.	2005	Enhanced stochastic evolutionary algorithms	ϕ_p criteria, entropy and L_2 discrepancy
Liefvendahl and Stocki	2006	Columnwise-pairwise and genetic algorithms	Minimum distance and Audze-Eglajs function
Van Dam et al.	2007	Branch-and-bound algorithm	1-norm and infinite norm distances

Grosso et al.	2008	Iterated local search and simulated annealing algorithms	ϕ_p criterion

3.2 ILS Heuristic for Maximin LHD

In Section 3.1 we have discussed a general scheme for ILS-based algorithms. Now we present the ILS based procedure for maximin Latin hypercube design. As we have stated earlier, the main components of ILS heuristic approaches are Initialization (I_S), LocalSearch (L_M), Perturbation Move (P_M), and the Stopping Rule (S_R)

The pseudo-code of the proposed ILS heuristic for maximin LHD problems is given bellow:

```
Step 1. **Initialization :** $X = I_S(\{0, 1, . . . ,N - 1\}))$
Step 2. **Local Search :** $X^* = L_M(X)$
while $S_R$ not satisfied do
Step 3. **Perturbation Move :**$X' = P_M(X)$
Step 4. **Local Search :** $X^* = L_M(X')$
Step 5. **Improvement test :** if $X^*$ is better than $X$,
set $X = X^*$
**end while**
**Return** $X$
```

Below we detail the components in order to fully specify our algorithm.

3.3 Initialization (I_S)

The initialization (I_S) procedure embedded in our algorithm is extremely simple: the first initial solution is randomly generated. In particular, the first initial solution generation is built as follows. For each component $h \in \{1, . . . , k\}$ a random permutation $v_0, . . . , v_{N-1}$ of the integers 0, 1, . . . ,$N - 1$ is generated and we set

$$x_{rh} = v_r \quad \text{for all } r \in \{0, . . . , N - 1\}.$$

Although more aggressive procedures could be designed, we chose random generation because it is fast and unbiased.

3.4 Local Search Procedure (L_S)

In order to define a local search procedure (L_S), we need to define a concept of neighborhood of a solution. Given a LHD $X = (x_1, . . . , x_N)$, its neighborhood is made of all other LHDs obtained by applying local moves to *X*. Before introducing some local moves, we first introduce the notion of critical point.

Critical point: We say that x_i is a critical point for *X*, if

$$\min_{j \neq i} d(x_i, x_j) = D_1(X),$$

i.e., the minimum distance from $\mathbf{x}_i$ to all other points is also the minimum one among all the distances in ***X***. We denote by $I(\boldsymbol{X})$ $\{1, \dots, N\}$ the set of indices of the critical points in *X*.

3.5 Local Moves (L_M): A local move is an operator that applies some form of slight perturbation to a solution *X*, in order to obtain a different solution. Different local moves define different neighborhoods for local search. In the literature two different local moves are available: Rowwise-Pairwise (RP) exchange [Park (1994)] and Columnwise-Pairwise (CP) exchange [Morris and Mitchell (1995)]. In Park's algorithm [Park (1994)] some active pairs (pairs of critical points, in our terminology) are selected. Then, for each chosen pair of two active rows, say i_1 and i_2, the RP exchange algorithm considers all the possible exchanges of corresponding elements as follows:

$$x_{i1,p} \leftrightarrow x_{i2,q} \ \forall \ p, q = 1, 2, \dots, k : p \neq q,$$

and finds the best exchange among them. The CP algorithm proposed by Morris and Mithchell [Morris and Mitchell (1995)] exchanges two randomly selected elements within a randomly chosen column. But in [Li and Wu (1997)], Li and Wu defined the CP algorithm in a bit different way: they randomly choose a column and replace it by its random permutations if a better LHD is obtained.

It is observed that the effect of CP based local search and RP based local search is not significance [Jamali (2009)]. So, here, RP based local move is considered as defined in [Jamali (2009)] which is a bit different than that of [Park (1994)]. For optimal criteria we consider Opt(ϕ) optimal criteria.

The definition of Rowwise-Pairwise Critical Local Moves (we call it LM_{RpD1}) as follows. The algorithm sequentially chooses two points (rows) such that at least one of them is a critical point, then exchanges two corresponding elements (factors) of the selected pair. If $i \in I(X)$, $r, j \in \{1, \dots, N\}$, $h, \ell \in \{1, \dots, k\}$, swapping the ℓ-th component gives the neighbor *Y* defined by

$$y_{rh} = \begin{cases} x_{rh} & \text{if } r \neq i \text{ or } h \neq \ell \\ x_{ih} & \text{if } r = j \text{ and } h = \ell \\ x_{jh} & \text{if } r = i \text{ and } h = \ell \end{cases} \tag{6}$$

It is remarked that, if Opt(D_1, J_1) be the optimality criterion, it perfectly makes sense to avoid considering pairs x_i and x_j such that $I(X) \cap \{x_i, x_j\} = \emptyset$ since any swap involving two non-critical points cannot improve the D_1 value of the current LHD.

When Opt(ϕ) is adopted as optimality criterion, any exchange can, in general, lead to an improved value of ϕ. The RP local move for Opt(ϕ) optimality criterion is denoted by $LM_{RP}\phi$ and is also defined in equation, the only difference being that we drop the requirement that at least one point must be critical.

We now illustrate the RP based local moves by considering a randomly generated initial design A : $(N,k) = (7,2)$ (see Figure 3.2(a)). Then a neighborhood solution of A, by considering points (0,2), (4,4) (here both are critical points), is LHD B, obtained after swapping the second coordinate of the points (0, 2) and (4,4) (See Figure 3.2 (b)).

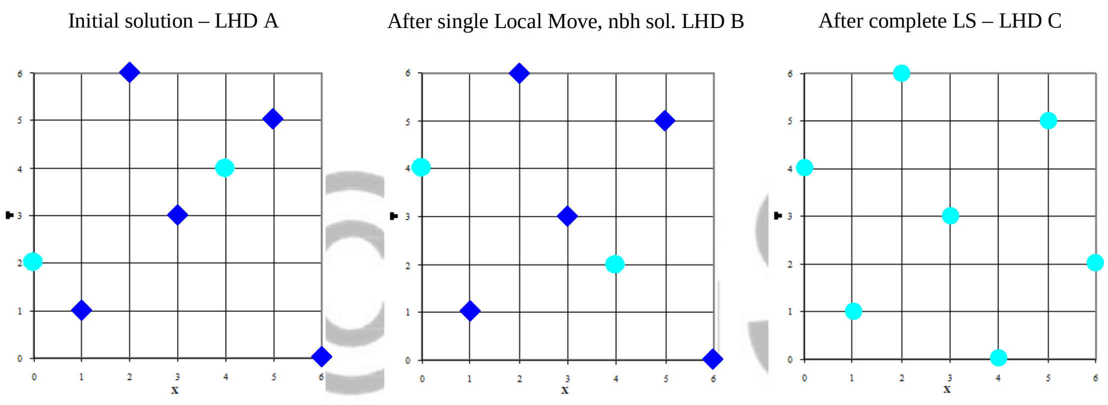

Fig: (a) $D_1(X_r)$=2, $J_1(X_r)$=3
X

Fig: (b) $D_1(X_b)$=2, $J_1(X_b)$=1
X

Fig: (c) $D_1(X_c)$=8, $J_1(X_c)$=4
X

Figure 3.2: Illustration of Neighborhood solutions for $LM_{RP}p_{D1}$ based local search (LS) procedure

Also note that LHD B is an improving neighbor of LHD A, since $(D_1, J_1)(B) = (2,1)$ whereas $(D_1, J_1)(A) = (2,3)$.

3.6 Acceptance Rule: Among the two type of local moves [Jamali (2009)], we considered Best Improve (BI) acceptance rule as there are no significant difference regarding output (see [Jamali (2009)]). For the BI acceptance rule, the whole neighborhood of the current solution is searched for the best improving neighbor. We warn again the reader that the meaning of "Y is better than X" can be defined accordingly with the Opt(D_1, J_1) or Opt(ϕ) optimality criterion. So for the Opt(D_1, J_1) optimality criterion: "Y is better than X" if

$$D_1(Y) > D_1(X) \text{ or } (D_1(X) = D_1(Y) \text{ and } J_1(X) > J_1(Y)).$$

On the other hand for Opt(ϕ) optimality criterion : "Y is better than X" if

$$\phi_p(Y) < \phi_p(X),$$

where ϕ_p is defined by (5).

3.7 Perturbation Move (P_M)

Perturbation is the key operator in ILS, allowing the algorithm to explore the search space by jumping from one local optimum to another. Basically, a perturbation is similar to a local move, but it must be somehow less local, or, more precisely, it is a move within a neighborhood larger than the one employed in the local search. Actually the perturbation operator produces the initial solutions for all the local searches after the first one. Among the two types of perturbation operators, say, (i) Cyclic Order Exchange (COE) and (ii) Pairwise Crossover (PC) proposed in [Jamali (2009)], we consider COE.

Cyclic Order Exchange (COE): Our first perturbation move procedure is Cyclic Order Exchange (COE). The operator COE produce a cyclic order exchange upon a randomly selected single component (column) of a randomly selected portion of the design points (rows). Among the three variant of COE perturbation move techniques: Single Cyclic Order Exchange (SCOE) perturbation operation, Multiple Components Cyclic Order Exchange (MCCOE), and Multiple Single Cyclic Order Exchange (MSCOE) [Jamali (2009)], we consider here only SCOE technique.

Single Cyclic Order Exchange (SCOE): For SCOE, we randomly choose two different rows (points), say x_i and x_j , such that $i < j$ and $j - i \geq 2$, in the current LHD X^*. Then, we randomly choose a column (component), say ℓ. Finally, we swap in cyclic order the value of component ℓ from point x_i to point $\mathbf{x}_j$. The pseudo-code structure for SCOE is the following.

The pseudo-code structure for SCOE is the following.

```
Step 1:  randomly select two different points $x_i$ and $x_j$
such that $i < j$ and $j - i \geq 2$
Step 2:  Randomly choose a component $\ell$
Step 3a: set temporarily $x^t_{j\ell} = x_{j\ell}$
for $t = j, j-1, \ldots, i+1$ do
    Step 3b: Replace the component $x(t)\ell$ by $x(t-1)\ell$
end for
Step 3c: and replace $x_{i\ell}$ by $x^t_{j\ell}$
```

Note that we require $j - i \geq 2$ because otherwise the perturbation would be a special case of the local move employed in the local search procedure. We illustrate the SCOE perturbation by an example. Assume we have the current LHD X^* with $N = 6$ and $k = 8$

$$X^* = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 0 & 5 & 3 & 1 & 5 & 2 & 4 & 4 \\ 1 & 0 & 4 & 2 & 4 & 3 & 3 & 5 \\ 2 & 1 & 5 & 3 & 3 & 4 & 2 & 0 \\ 3 & 2 & 0 & 4 & 2 & 5 & 1 & 1 \\ 4 & 3 & 1 & 5 & 1 & 0 & 0 & 2 \\ 5 & 4 & 2 & 0 & 0 & 1 & 5 & 3 \end{pmatrix} \qquad (7)$$

Now we randomly choose two rows (points), say $\boldsymbol{x_2}$ and $\boldsymbol{x}_5$ and we randomly choose the column (component) $\ell = 4$. Then, after the SCOE perturbation we get the following LHD X' (bold faces denote the values modified with respect to X^*),

$$X' = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 0 & 5 & 3 & 1 & 5 & 2 & 4 & 4 \\ 1 & 0 & 4 & \mathbf{5} & 4 & 3 & 3 & 5 \\ 2 & 1 & 5 & \mathbf{2} & 3 & 4 & 2 & 0 \\ 3 & 2 & 0 & \mathbf{3} & 2 & 5 & 1 & 1 \\ 4 & 3 & 1 & \mathbf{4} & 1 & 0 & 0 & 2 \\ 5 & 4 & 2 & 0 & 0 & 1 & 5 & 3 \end{pmatrix} \qquad (8)$$

Note that SCOE only slightly modifies the current LHD X^* but this exactly follows the spirit of ILS, where the perturbation should keep unchanged large portions of the current solution and should not completely disrupt it structure.

REFERENCES

Aparna D., 2012 , "Iterated Local search Approaches For Maximin Latin Hypercube Designs", M. Phil thesis paper, Department of Mathematics, Khulna university of Engineering & Tecnology, Khulna.

Audze P., and V. Eglais, 1997, "New approach to planning out of experiments, problems of dynamics and strength", Vol. 35, pp. 104-107.

Bates S. J., Sienz J. and Langley D.S., 2003, "Formulation of the Audze-Eglais Uniform Latin Hypercube design of experiments", Advanced in Engineering Software, Vol. 34, Issue 8, pp. 493-506.

Blondel V. D., Tsitsiklis J. N., 2000, "A survey of computational complexity results in systems and control", Vol. 36, pp. 1249-1274, Fang, K. T., D. K. J. Lin, P. Winkler, and Y. Zhang (2000b), “Uniform design: theory and application”, Technometrics, Vol. 42, pp. 237–248.

Felipe A.C. Viana, Venter G., 2009(Oct), “An Algorithm for Fast Optimal latin Hypercube Design of Experiments”, pp.1-4, Dol: 10.1002/nme.2750

Grosso A., Jamali A. R. J. U. and Locatelli M., 2009, “ Finding Maximin Latin Hypercube Designs by Iterated Local Search Heuristics”, *European Journal of Operations Research*, *Elsevier,* Vol. 197, pp. 541-547.

Grassberger P., 1997, “Pruned-enriched Rosenbluth method: Simulations of θ polymers of chain length up to 1000000”, Phys. Rev., Vol. 56(3), pp. 3682-3693

Helton J. C. & Davis F. J. 2000, “Sampling-based methods, in Sensitivity Analysis”, Ed.

Iman, R. L. and Conover W. J. (1982b), “Small-sample sensitivity analysis techniques for computer models, with an application to risk assessment. Communications in Statistics – Part A”, Theory and Methods 17, 1749–1842.

Jin R., W. Chen, and A. Sudjianto, 2005, “An efficient algorithm for constructing optimal design of computer experiments”, Journal of Statistical Planning and Inference, Vol. 134(1), pp. 268-287.

http://www.wikipedia.org/w/wiki.phtm?tittle=Big O notation

http://www.cs.toronto.edu/~vassos/teaching/c73/handouts/brief-com

http://www.spacefillingdesigns.nl