Quest Journals
Journal of Research in Applied Mathematics
Volume 7 ~ Issue 6 (2021) pp: 52-54
ISSN(Online) : 2394-0743 ISSN (Print): 2394-0735
www.questjournals.org

Research Paper

An Algorithm for Analysis the Time Complexity for Iterated Local Search (ILS).

Parimal Mridha[1], Binoy Kumar Datta[2].
[1](*Lecturer in Mathematics, Military Collegiate School Khulna (MCSK), Bangladesh*)
[2](*Associate Professor in Mathematics, Military Collegiate School Khulna (MCSK), Bangladesh*)

ABSTRACT: *For the presence of combinatorial nature, the number of possible Latin Hypercube Design is very high - (N!)k (where N is number of design points and k is number of factors). Consequently, when number of factors and/ or number of design points are large then it requires hundreds of hours by the brute-force approach to find out the optimal design. So when numbers of factors as well as number of experimental points are large, the algorithm requires 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, especially when we need a real time solution. In this paper, we will introduce an algorithm for analysis the time complexity for iterated local search.*
KEYWORDS: *Latin Hypercube Design, Experimental points, iterated local search, time complexity.*

Received 15 June, 2021; Revised: 28 June, 2021; Accepted 30 June, 2021 © The author(s) 2021. Published with open access at www.questjournals.org

I. 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.

II. TYPES OF TIME COMPLEXITY

There are many different types of complexity involved in actual examples of scientific modelling. Conflation of these into a single “complexity” of scientifically modelling a certain system will generally result in confusion. There might be:

• The complexity of the data: the difficulty of encoding of a data model compactly given a coding language;

• The complexity of the informal (mental) model: the difficulty in making an informal prediction from the model given hypothetical conditions;

• The complexity of using the formal model to predict aspects of the system under study given some conditions;

• The complexity of using the formal model to explain aspects of the system under study given some conditions.

Each of these will be relative to the framework it is being considered in (although this and the type of difficulty may be implicit).

III. 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.

Finally, the analysis of time complexity for ILS is given in a tabular form along with the pseudo code of the algorithm. Table 1 represents the analysis of the time complexity for the ILS.

Table 1: Analysis of time complexity for ILS.

(i) Compute $d_{i',u}^{(s+1)} \xrightarrow{Operation} O(1)$ (comp. only swapped cord.) (ii) Step 3 (in worst case) $\xrightarrow{Operation} O(N$ (since compu. stop as soon as $d_{i',u}^{(s+1)} \leq D_1'$) (iii) Inner most for loop $\xrightarrow{Operation} O(k)$ (in BI local move) (iv) outer two for loops $\xrightarrow{Operation} O(N$ (in worse case theoretic ally $O(N^2)$) (but experiment ally swap $O(N)$) (v) WL (in local search) $\xrightarrow{Operation} O(Nk^c)$ $(0 < c < 1)$, experiment ally computed Total cost of a single LS $O(1).O(N).O(k).O(N).O(Nk^c) \approx O(k^r N^q)$ $:(1 < r < 2) \ \& \ (2 < q \leq 4)$ **Cost of a single ILS (Φ)** Perturbation (for fixed MIN)) O (log(N)) **Total Cost:** $O(k^r N^q).O(\log(N)) \approx O(k^r N^q \log(N))$ $:(1 < r < 2) \ \& \ (2 < q \leq 4)$	While do Set NonImpIteration = 0 **Whiledo** for $i = 1,\ldots\ldots, N$ do for $j = i + 1,\ldots.., N$ do Step 1= $D_1 = D_1^{(S)}$, $k' = 0$ for $l = 1,\ldots., k$ do Step 2: Swap (X_{il}, X_{jl}) Step 3: Compute $d_{i',u}^{(X+1)}$ until $d_{i',u}^{(X+1)} \geq D_1'$ $\forall\, i' = i\,.j$: u = 0………N-1; $u \neq i'$ else break Step 4 :Set $k' = k$ and $D_1' = \min\, d_{IJ}^{(X+1)}$ end for Step 5: Upload best LHD if any **else continue** end for end for Step 6: Repeat the three loops if there has been at least an improvement Otherwise STOP Return X$'$ Step 7: if X$'$ is better then set X= X$'$ and NonimpIteration = 0 Otherwise increase NonimpIteration by one Step 8: if MaxNonImp > NonimpIteration *PM* : X$'$ = ∊ (X) and Repeat the loops Otherwise STOP Return X

REFERENCES

[1]. 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.

[2]. Nicolas S., 2006-2007(Nov), "Algorithms & Complexity-Introduction", nstropa@computing.dcu.ie, CA313@Dubai City University.

[3]. Oliveto P. S., He J., Yao X., 2007, "Time Complexity of Evolutionary algorithms for Combanatories Optimization: A Decade of Results", International Journal of Automation and Computing, Dol: 10.1007/s11633-007-0281-3, Vol. 04(1), pp. 281-293.

[4]. Owen, A. B., 1994, " Controlling correlations in Latin hypercube samples", Journal of the American Statistical Association,Vol. 89, pp. 1571–1522.

[5]. Park J. S., 1994, "Optimal Latin hypercube designs for computer experiments", Journal of Statistical Planning and Inference, Vol. 39, pp. 95-111.

[6]. Santner T. J., Williams B. J., and Notz W. I., 2003, "The design and analysis of computer experiments", Springer Series in Statistics, Springer-Verlag, New York.

[7]. Steinberg G. D. M, and K. J. N. Dennis, 2006, " A construction method for orthogonal Latin hypercube designs", Biometrika, Vol. 93 (2), pp. 279-288.

[8]. Husslage B., E. R. van Dam, and D. den Hertog, 2005, "Nested maximin latin hypercube designs in two dimensions", CentER Discussion Paper No. 200579.