A Progressive Stochastic Search Method for Solving Constraint Satisfaction Problems

Bryan Chi-ho Lam

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

©The Chinese University of Hong Kong
August, 2003

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or the whole of the materials in this thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.

香港中文大學
大學
圖書館系統
30 APR 2004
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Abstract

Constraint Satisfaction Problem (CSP) provides a powerful tool for modeling and solving many real-life problems and they are NP-complete in general. While traditional systematic search cannot find solutions within a reasonable time when solving large-scale and hard CSPs, stochastic search methods have attracted much attention of the research community. A typical stochastic search starts at a random point in a search space and moves from one point to its neighbor iteratively, provided that the new point gives a better cost value. Traditionally, a stochastic solver escapes from local optima or leaves plateaus by random restart or heuristic learning. In this thesis, we propose Progressive Stochastic Search (PSS) and its variants for solving binary CSPs. One characteristic of PSS is that we maintain a list of variables, which dictates the sequence of variables to repair. When a variable is designated to be repaired, it always has to choose a new value unless its original value does not cause any violations. Intuitively, the search can be thought to be mainly driven by a "force" so that the search is able to "rush through" the local minima and plateaus. The search paths are also slightly "marked" as the search proceeds. Random restarts are no longer necessary, and expensive heuristic learning is replaced by simple path marking. Timing results show that this approach outperforms $\mathcal{LSDL}$ implementations in N-Queens, Latin squares, random permutation generation problems and random CSPs, while it fails to win $\mathcal{LSDL}$ implementations in quasigroup completion problems and increasing permutation problems. This prompts an interesting new research direction

in the design of stochastic search schemes.

摘要

約束滿足問題 (CSP)為許多實際生活中的問題提供了一個強大的建模和求解工具。這些實際生活上的問題一般都是NP完全的。當傳統的系統搜索未能在合理的時間內對較大型和較困難的約束滿足問題求解時，隨機搜索吸引了研究人員的注意。一般的隨機搜索起始於一個隨機點，然後重覆地移向一個相鄰並具有較佳代價值的點。傳統上，隨機搜索過程在逃離局部最優位置或離開平原時，會使用隨機從新啟動或者啟發式學習法。在本論文中，我們提出一個對二元約束滿足問題求解的「進取性隨機搜索」方法(PSS) 以及它的一些變種。進取性隨機搜索的其中一個主要特點為使用一個變量表指定變量的修復次序。當一個變量被指定修復的時候，除非它原本所取的值符合所有約束，否則變量會取一個新的值。從直覺意義上來說，可以想像這種搜索主要被一種「力」驅使，「衝過」局部最小位置或平原。當搜索進行時，搜索路徑亦被輕微留下記號。我們再不需要使用隨機從新啟動，而昂貴的啟發式學習法亦被簡單的路徑記錄取代。實驗結果顯示，這種新的搜索在解決 N個皇后問題、拉丁方格、隨機排列生成問題和隨機約束滿足問題時的表現，能明顯地勝過 $\mathcal{LSDL}$。可是，它在解決准群完成問題和漸增的排列生成問題時的表現卻不及 $\mathcal{LSDL}$。這個結果在隨機搜索方案的設計上提示了一個有趣的新研究方向。

Acknowledgments

I would like to thank all those people who made this thesis possible and a memorable experience for me.

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Ho-fung Leung. Without his guidance, advices and encouragements, this thesis could not be presented. I wish to thank for his generosity and valuable discussions for the research.

I also deeply appreciate Professor Jimmy Ho Man Lee for giving me the lectures of Constraint Satisfaction. He also gave me valuable suggestions and comments for improving this work. I would like to thank him for providing the source code of $\mathcal{LSDL}$ and all benchmarks used in the experiments of this thesis.

I am grateful to the other members of constraint research group for the wonderful discussions and a pleasant working atmosphere.

Finally, I would like to take this opportunity to express my deepest appreciation to my family and Miss Tammy Pui Shan Chan for the constant care and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint Satisfaction Problem (CSP) [15] provides a powerful tool for modeling and solving many real-life problems. A CSP is conventionally defined as a problem of finding a consistent assignment of discrete values to a finite set of variables such that the assignment satisfies a finite set of given constraints over these variables.

CSPs are NP-complete in general. Many solvers for CSPs have been developed over the past three decades. The traditional approach is systematic search methods [13, 19] which are *complete* algorithms. However, they cannot find solutions within a reasonable time when solving large-scale and hard CSPs. An alternative approach, stochastic search methods [4, 5, 6, 9, 10, 30, 31] are *incomplete*, but their fast solving speed often compensates this drawback.

A typical stochastic search method is a hill-climbing algorithm, which includes a *cost function* that gives a value to every point in a search space, and a *neighborhoods function* that defines the neighbors of a particular point in the search space. The search moves from a point in the search space to a neighboring point if the latter has a better cost value than the current point. This can be interpreted as that the move is driven solely by "potential energy", though which better neighboring point to go to is usually determined randomly. The goal of the algorithm is to reach a point in the search space that has the optimal value according to the cost function, which corresponds to a solution to the original CSP. For solving CSPs, a typical cost function used is counting the number of conflicts [17]. The problem with hill-climbing algorithms is that they can be trapped in local optima, and lose direction in plateaus.

Traditionally, a stochastic solver escapes from local optima or leaves plateaus by random restart or heuristic learning. The former approach relies on the fact that there is a non-zero probability that a solution will be found after the search restarts at a randomly chosen point in the search space, if solutions really exist. The latter approach attempts to change the landscape of the search space as depicted by the cost function, until the local optimum or plateau the search is being trapped in ceases to exist.

In this thesis, we propose the Progressive Stochastic Search (PSS) and its variants for solving binary CSPs. One characteristic of PSS is that we maintain a list of variables, which dictates the sequence of variables to repair. When a variable is designated to be repaired, it always has to choose a new value even if its original value should give the best cost value. Intuitively, the search can be thought to be mainly driven by a "force" so that the search is able to "rush through" the local minima and plateaus. The search paths are also slightly "marked" as the search proceeds so as to help gathering information of the search space. Random restarts are no longer necessary, and expensive heuristic learning is replaced by simple path marking. Timing results show that this approach outperforms $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in N-Queens, Latin squares, random permutation generation problems and random CSPs, while it fails to win $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in quasigroup completion problems and increasing permutation generation problems. This prompts an interesting new research direction in the design of stochastic search schemes.

This thesis is organized as follows. In Chapter 2, we briefly introduce Constraint Satisfaction Problem and review some solving techniques published in the literatures. These solving techniques can traditionally be classified into two categories: systematic search and stochastic search. As our work can be classified into the category of stochastic search, some related work are also given. These include GENET, $\mathcal{LSDL}$ and Adaptive Search. In recent years, a hybrid approach of systematic and stochastic search has raised interest in CSP community. A brief introduction on this hybrid approach is also given in Chapter 2. The Progressive Stochastic Search scheme and its variants are described in Chapter 3. Experiments on benchmarking problems and some analysis of results are presented in Chapter 4. Chapter 5 summarizes our contributions and sheds light on future work.

Chapter 2

Background

In this chapter, we provide background information related to our research. We give a brief introduction on Constraint Satisfaction Problem (CSP). In addition, a summary of CSP solving techniques is also presented. These solving techniques can traditionally be classified into two categories: systematic search and stochastic search. As our work can be classified into the category of stochastic search, some related work is also given in the section of stochastic search. In recent years, a hybrid approach of systematic and stochastic search has raised interest in CSP community. A brief introduction on this hybrid approach is also given at the end of this chapter.

2.1 Constraint Satisfaction Problems

A CSP $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ is a tuple consisting of a set $\mathcal{V}$ of variables, a set $\mathcal{D}$ of domains and a set $\mathcal{C}$ of constraints. Each variable $v_i \in \mathcal{V}$ is associated with a domain $d(v_i) \in \mathcal{D}$ which contains the set of possible values for v_i. A constraint $c \in \mathcal{C}$ ranging over a number of variables specifies the combination(s) of values these variables can take. A binary CSP is a CSP with unary and binary constraints only.[1] A solution of a CSP is an assignment of values to all variables such that all constraints are satisfied.

[1]Note that any n-ary constraints CSP ($n > 2$) can be transformed to an equivalent binary CSP [23].

Numerous algorithms have been developed for solving CSPs. These algorithms can be typically classified into two categories: systematic search and stochastic search.

2.2 Systematic Search

The traditional search method used in solving CSPs is chronological backtracking tree search. Variables are assigned values from their domains one after another. After a variable is assigned a value, the currently partial assignment is checked for consistency. If it violates any of the constraints, the currently considered variable is assigned an alternative value. If no value is available for this variable, the most recently variable that has been assigned a value is revised. The above process is repeated until either a solution is found or all partial assignments have been checked for consistency. In the latter case, the chronological backtracking algorithm concludes that no solution exists for the CSP.

Various constraint propagation techniques can be combined with backtracking tree search to enhance the solving efficiency [13, 19]. These techniques include *node consistency* [15], *arc consistency* [15], *path consistency* [15] and *bounds consistency* [16]. The purpose of these techniques is to remove inconsistent values from the domains of variables. As a result, the search space in the search tree is reduced. These algorithms virtually explore the whole search tree systematically by depth-first search. Therefore, they are complete algorithms that guarantee to find a solution if it exists, and to report unsatisfiability otherwise. Various variable- and value-ordering heuristics [2, 11] have been investigated to improve the search speed. These heuristics aim at reducing the number of backtracks required in a search. However, systematic search generally becomes less efficient when solving large-scale and hard CSPs due to the NP-complete nature of CSP.

2.3 Stochastic Search

2.3.1 Overview

Another category of approaches to search, stochastic search has drawn much attention of the Artificial Intelligence (AI) community. This category of incomplete algorithms often solve some standard benchmarking problems, such as N-Queens and graph-coloring, in orders of magnitude better than the traditional tree search approach. Typical stochastic search algorithms first generate a complete initial variable assignment (probably random and inconsistent) and then repair the assignment by heuristic local search until a solution is found. The heuristic local search repairs the variable assignment with reference to a cost function. A possible cost function used is one that counts the number of constraint violations by the variable assignment. A variable is selected and repaired by being assigned a new value that optimizes the cost function. A drawback of this category of solving methods is that the execution can easily be trapped in local optima, *i.e.*, non-solution states in which no further improvement can be made. Two main approaches have been developed for escaping local optima. One approach is random restart [17]. Although it is simple and intuitive, information generated in a search process is completely lost. Another approach associates *weights* with the constraints and defines the cost function as a weighted sum of constraints violations [4, 6, 18, 28, 30]. When a local optimum is reached, the weights are updated. This helps not only escape from the local optima but also guides the search to solution states.

In the last decade, various stochastic search variants have been proposed, which use different cost functions, variable-orderings and escape strategies to boost the performance. In the context of satisfiability problem (SAT), GSAT [27] is a greedy local search method. Several extensions, which integrated with a random walk [25, 26], clause weight learning [7, 25], averaging in previous assignments [25] and tabu-like move restrictions [8], improve the original GSAT in some kinds of SAT problems. The Lagrange multiplier method is a well-known technique for solving constrained optimization problems. Wah *et al.* extend the classical Lagrange multiplier method to handle discrete problems [28, 34]. Their extention, called DLM (Discrete Lagrangian-based global-search method), uses the Lagrangian function as a cost function and a complicated weight update scheme to escape from local minimum.

GENET [6, 30] is a local search approach for solving binary CSPs. It uses iterative repair method to find a solution to the CSP. A heuristic learning rule is applied to escape from local minima and to help preventing the network from being trapped in the same local minima again. Several variants of GENET are developed for solving different kinds of CPS's. Fuzzy GENET [33] is proposed to solve binary fuzzy CSPs. E-GENET [14] extends GENET to handle non-binary constraints. $\mathcal{LSDL}$ [4] basically explains the behaviour of GENET as a discrete Lagrangian search algorithm and improves on GENET by choosing different parameters. Guided Local Search (GLS) [31] extends GENET to handle combinatorial optimization problems. Adaptive Search [5] introduces an error function to determine which variable is repaired at next. For each constraint, it is not associated with a weight but an error function to represent the "degree of satisfaction". Each variable is associated with an error. The error is the sum of the error function values of all constraints in which the variable is involved. The variable with the maximum error will be selected to repair in the next iteration.

In the following sections, we give details of other research work that are related to our work. These include GENET, $\mathcal{LSDL}$ and Adaptive Search. We first give a summary of GENET because our proposed method, Progressive Stochastic Search (PSS) is a heuristic search method for solving binary CSPs and the modeling of CSP in PSS is similar to that in GENET. $\mathcal{LSDL}$ is a discrete Lagrange multiplier method for solving integer constrained minimization problems. One of the variants of $\mathcal{LSDL}$, $\mathcal{LSDL}$(GENET), is a Lagrangian reconstruction of GENET. $\mathcal{LSDL}$(GENET) is the most efficient implementation of GENET that we know of. We use the performance of $\mathcal{LSDL}$(GENET) in experiments to compare the performance of PSS and its variants. A description of $\mathcal{LSDL}$ is given next. Adaptive Search is a heuristic search method for solving CSPs. The key idea of this method is using variable-based information to decide which variable should be repaired at next. This idea is closely related to the list of variables-to-be-repaired used in PSS.

2.3.2 GENET

GENET [6, 30] is a local search approach for solving CSPs with binary constraints. GENET uses iterative repair method to find a suitable assignment of variables. Once it is trapped in a local minimum, a heuristic learning rule is applied to escape from the local minimum and to avoid the network settled in the same local minimum again.

GENET first models a given binary CSP $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ as a neural network. Each node in the network represents an assignment of a value to a variable. The *state* s_i of node i is either 1 for *on* or 0 for *off*. If a node is *on*, it means the corresponding value is being assigned to the variable. A *cluster* is the set of all nodes that represents the assignments of the same variable. A *connection* between two nodes of different clusters represents an incompatible tuple of a binary constraint. Each connection contains a *weight*, which is initialized to -1. The weight of the connection between node i and j is denoted as w_{ij}. The *input* to a node is the weighted sum of all its connected nodes' states. At any time, only one node in each cluster is *on*. Therefore, every state of the network represents an assignment of values to the variables from their respective domains. A *solution* to the binary CSPs is at any network state, in which no two *on* nodes are connected to each others. For instance, the GENET network of a CSP with $\mathcal{V} = \{X, Y, Z\}$, $d(X) = d(Y) = d(Z) = \{1, 2, 3\}$ and $\mathcal{C} = \{X + Y > 3, Y + Z > 3\}$ is showed in Figure 2.1.

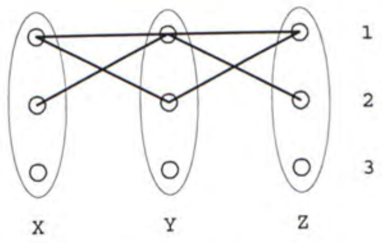


Figure 2.1: An example of GENET network

GENET starts with randomly turning *on* one node in each cluster. In each convergence cycle, every node in each cluster calculates its input. The node with maximum input in each cluster is turned *on* and the others are turned *off*. Note that the node with maximum input in each cluster represents the assignment with the fewest number of constraint violations. If there are more than one node have maximum input, a tie breaking system is run: if one of them was *on* in the previous cycle, it will remain *on*. If all the nodes were *off* in the previous cycle, a random choice is made. This is to avoid chaotic or cyclic wandering of the network states.

When the network reaches to a stable state, *i.e.*, no more changes to the *on* nodes in the network, GENET checks if that state represents a solution. A solution state is all *on* nodes have zero input. Otherwise, the network is trapped in a local minimum.

When GENET settles in a local minimum, it represents that there are some *on* label nodes that still receive negative input, *i.e.*, some constraints are still violated. The cause of this situation is the variable assignments are based on the local information received at each cluster of nodes. To escape from the local minima, a *heuristic learning rule* is used to update the weight of connections

$$w'_{ij} = w_{ij} - s_i \times s_j$$

Note that only the connections between two *on* nodes are being updated and the value of weight is decreased by one each time. Therefore, after sufficient learning cycles, the *on* node i and the *on* node j will not be the winner in its cluster. Since the weights of the connections leading to local minima has been updated, the heuristic learning rule avoids the network settling in the same local minima again.

2.3.3 $\mathcal{LSDL}$

$\mathcal{LSDL}$ [4] is a discrete Lagrange multiplier method [28] for solving integer constrained minimization problems. $\mathcal{LSDL}$ has five parameters and so it has several variants. $\mathcal{LSDL}$(GENET), one of the variants, is a Lagrangian reconstruction of GENET. Choi *et al.* [4] establish a relationship between GENET and discrete Lagrange multiplier methods. $\mathcal{LSDL}$(GENET) is shown to have the same performance as the original GENET implementation. The best variant of $\mathcal{LSDL}$ reported in [4] outperforms the reconstructed GENET by an order of magnitude. To solve a binary CSP, $\mathcal{LSDL}$ first converts the given binary CSP into an integer constrained minimization problem. Then a discrete Lagrange multiplier method is applied to solve the converted integer constrained minimization problem.

$\mathcal{LSDL}$ first uses a GENET network to model a binary CSP $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$. Then the GENET network is converted into an integer constrained minimization problem. Suppose all values in domain $d(v_i) \in \mathcal{D}$ for all $v_i \in \mathcal{V}$ are integers. Each cluster i of the network corresponds to an integer variable z_i in an integer constrained minimization problem. Each node in cluster i corresponds to one domain value of z_i. Each connection of the network is transformed into an *incompatibility function*

$$g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) = \begin{cases} 1 & \text{if } z_i = j \wedge z_k = l \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $\langle i,j\rangle\langle k,l\rangle$ is a connection between the node j in cluster i and the node l in cluster k of the network. With the incompatibility function, the integer constrained minimization problem can be defined as follows,

$$\min f(\vec{z}) \tag{2.2}$$

$$\text{subject to} \qquad z_i \in d(v_i), \qquad \forall v_i \in \mathcal{V}, \tag{2.3}$$

$$g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) = 0, \qquad \forall(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I}, \tag{2.4}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $\mathcal{I}$ is the set of all incompatible label pairs $(\langle i,j\rangle, \langle k,l\rangle)$. The objective function $f(\vec{z})$ typically used is the total number of constraint violations in an assignment [32] or just a constant, *i.e.*, $f(\vec{z}) = 0$.

With the resultant integer constrained minimization problem (2.2)-(2.4), $\mathcal{LSDL}$ solves the given binary CSP using a discrete Lagrange multiplier method. The *Lagrangian function* $L(\vec{z}, \vec{\lambda})$ is defined as

$$L(\vec{z}, \vec{\lambda}) = f(\vec{z}) + \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \lambda_{\langle i,j\rangle\langle k,l\rangle} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}), \tag{2.5}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $\vec{\lambda} = (\ldots, \lambda_{\langle i,j\rangle\langle k,l\rangle}, \ldots)$ is a vector of *Lagrange multipliers*. The goal is to obtain a global minimum of the resultant integer constrained minimization problem (2.2)-(2.4) by finding a *saddle point* [34] of the Lagrangian function $L(\vec{z}, \vec{\lambda})$. The saddle point can be found by searching descent direction in the discrete variable space of $\vec{z}$ and ascent direction in the Lagrange multiplier space of $\vec{\lambda}$.

$$\vec{z}^{s+1} = \vec{z}^{s} - GD(\Delta_{\vec{z}} L(\vec{z}^{s}, \vec{\lambda}^{s}), \vec{z}^{s}, \vec{\lambda}^{s}, s), \tag{2.6}$$

$$\vec{\lambda}^{s+1} = \vec{\lambda}^{s} + \vec{g}(\vec{z}^{s}), \tag{2.7}$$

where $\vec{x}^s$ denotes the value of $\vec{x}$ in the sth iteration, $\Delta_{\vec{z}}$ is the *discrete gradient*, GD is a *gradient descent function* and $\vec{g}(\vec{z}) = (\ldots, g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}), \ldots)$ is a vector of incompatibility functions.

$\mathcal{LSDL}$ has five parameters, namely (f) the objective function, $(I_{\vec{z}})$ initialization of the integer vector $\vec{z}$, $(I_{\vec{\lambda}})$ initialization of the Lagrange multipliers $\vec{\lambda}$, (GD) the gradient descent function, and $(U_{\vec{\lambda}})$ condition for updating the Lagrange multipliers $\vec{\lambda}$. GENET can be reconstructed as an instance of $\mathcal{LSDL}$ with a set of appropriate parameters. The details about the parameters for reconstructed GENET can be found in [4].

2.3.4 Adaptive Search

Adaptive Search [5] is a heuristic search method for solving CSPs. The key idea of this method is using variable-based information to decide which variable should be repaired at next. Then min-conflict heuristics [17] is applied to select a suitable value for the repaired variable. Adaptive Search is an iterative repair method. It terminates if either a solution is found or a pre-set limit of iterations is reached. If the search is trapped in local minima, the variable caused the trap is marked tabu [9, 10] and cannot be selected for a number of coming iterations.

The variable-based information used in Adaptive Search is obtained from the constraints during the search. For each constraint, it associates with an error function. An error function value returned by the error function represents the "degree of satisfaction" of the corresponding constraint. For instances, the error function associated to constraints $X + Y = 5$ and $Y - Z = 2$ can be defined as $X + Y - 5$ and $Y - Z - 2$ respectively. Each variable is associated with an error. The error is the sum of the error function values of all constraints in which the variable involved. The variable with the maximum error will be selected to repair in the next iteration. For example, suppose the variable assignments are $X = 5$, $Y = 4$ and $Z = 1$, the error associated to variable X, Y and Z are 4, 5 and 1 respectively. The total error of the variable assignments is then computed as the sum of the absolutes values of the errors, which is equal to 10. Therefore, variable Y will be repaired by assigning a value that minimizes the total error.

2.4 Hybrid Approach

A hybrid approach of systematic and stochastic search has raised interest in CSP community in recent years.

Yokoo proposes Weak-commitment Search [35] which employs min-conflict heuristics [17] on backtracking algorithm. All variables are given tentative initial values. The process proceeds by repeatedly constructing consistent partial solutions and extend them to include new variables one by one, until a consistent complete solution is found. If a partial solution cannot be extended, the whole partial solution is abandoned and a new partial solution is constructed from scratch, which uses the current value assignment as new tentative initial values. Richards et al. [22] propose learn-SAT algorithm that modified Weak-commitment Search with learning-by-merging [21] for SAT problem. Pesant et al. [20] use systematic branch-and-bound search to explore the set of local search neighborhoods in combinatorial optimization problems. Schaerf [24] proposes a technique that constructs a partial consistent solution incrementally. Local search is performed on the partial solution each time when the construction reaches a dead-end. Jussien et al. [12] propose Path-repair algorithm that performs local search as a basis, and uses filtering methods to prune the search space and help in selecting neighborhoods.

Chapter 3

Progressive Stochastic Search

This chapter gives an introduction to Progressive Stochastic Search (PSS). PSS is a new heuristic search method for solving binary CSPs. One characteristic of PSS is that we maintain a list of variables, which dictates the sequence of variables to repair. When a variable is designated to be repaired, it always has to choose a new value even if its original value should give the best cost value. Intuitively, the search can be thought to be mainly driven by a "force" so that the search is able to "rush through" the local minima and plateaus. The search paths are also slightly "marked" as the search proceeds. Incremental PSS (IPSS) is a variant of PSS. This variant shows an improvement over PSS on some benchmarks. Details of IPSS are also given in this chapter. Finally, we shall talk about a heuristic cluster selection strategy that integrates with PSS and IPSS to boost the performance on some benchmarks.

3.1 Progressive Stochastic Search

Our proposed method, Progressive Stochastic Search (PSS), is a novel heuristic search method for solving binary CSPs. In order to present our idea in a systematic way, we adopt the presentation of GENET [6, 30] to illustrate the idea of PSS.

3.1.1 Network Architecture

In PSS, a binary CSP $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ is represented by a network similar to GENET. A variable v_i is represented by a cluster i of *label nodes*. Each label node m_i corresponds to a value m in the domain of v_i. We assume that each domain contains more than one domain values. If a domain of a variable contains only one value, this variable can be explicitly assigned the only value and the CSP can be simplified by removing that variable and redefining the constraints. The state of a label node is either *on* or *off*. At any moment, there is exactly one label node that is in the *on* state in any cluster. Intuitively, a label node is in the *on* state means the corresponding value is being assigned to the variable.

A binary constraint $c \in \mathcal{C}$ on variables v_i and v_j is represented by weighted connections between pairs of label nodes in clusters i and j respectively. There is a connection between two label nodes m_i and n_j if $v_i = m \wedge v_j = n$ is prohibited according to c. Each connection is associated with a weight initialized to one. The weight of the connection between a label node m_i and a label node n_j is denoted as $W_{m_i n_j}$. The *output* O_{m_i} of a label node m_i is 1 if the node is *on* or 0 if *off*. The *input* I_{m_i} to a label node m_i is the weighted sum of all its connected label nodes' outputs:

$$I_{m_i} = \sum_{n_j \text{ is connecting to } m_i} W_{m_i n_j} O_{n_j} \qquad (3.1)$$

As at most one label node in each cluster is *on* at any time, a state of the network represents an assignment of values to the variables from their respective domains. A solution to the binary CSP corresponds to any network state in which no two *on* label nodes are connected to each other. For instance, the network architecture of a binary CSP with $\mathcal{V} = \{X, Y, Z\}$, $d(X) = d(Y) = d(Z) = \{1, 2, 3\}$ and $\mathcal{C} = \{X \neq Y, Y = Z\}$ is showed in Figure 3.1.

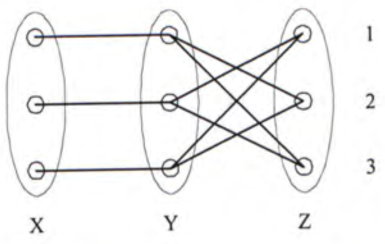


Figure 3.1: The network architecture of PSS

3.1.2 Convergence Procedure

The goal of executing the convergence procedure is to choose one label node in each cluster to turn *on* so that no two connected label nodes are turned *on* at the same time.

The network is initialized as follows. Initially, all label nodes in all clusters are in the *off* state. All weights of connections are initialized to one. Let $\mathcal{U}$ denote the set of all clusters with all label nodes in *off* state. Therefore $\mathcal{U}$ initially contains all clusters. Clusters are then removed from the set $\mathcal{U}$ one after another. When a cluster x is removed from the set $\mathcal{U}$, each label node in x calculates its input, and the label node with the minimum input is turned *on*. Ties are broken randomly. The greedy initialization [17] completes when the set $\mathcal{U}$ is empty.

We maintain a *list* $\mathcal{F}$ to be used in the convergence procedure. Immediately after the initialization, we append all clusters into the list $\mathcal{F}$ one by one, in an arbitrary order. In each convergence step, the head cluster h of the list $\mathcal{F}$ is removed from the list. Let p_h denote the *on* label node in h. If p_h has a zero input, then it remains *on* and nothing needs to be done. Otherwise, p_h is turned *off*, and the label node k_h with minimum input among all label nodes other than p_h is turned *on*. Any cluster with its *on* label node connecting to k_h is then appended to the list $\mathcal{F}$ if it is not already in the list.

In order to guide the search towards solutions, we adopt the following heuristic learning rule to update the connection weights.

$$W^{new}_{p_h n_j} \leftarrow W^{old}_{p_h n_j} + O_{n_j} \tag{3.2}$$

where p_h is the previous *on* label node in the cluster h and n_j is a label node in the cluster j connecting to p_h. This heuristic learning rule states that the weight of the connection $W_{p_h n_j}$ that exists between p_h and n_j is incremented by one if n_j is *on*, otherwise it remains unchanged.

After that, another cluster is removed from the list $\mathcal{F}$, and the above process is repeated. The convergence procedure terminates when the list $\mathcal{F}$ becomes empty.

As the input of a label node represents the number of weighted conflicts between this label node and the other *on* label nodes, the label node turned *on* by the above convergence procedure in each cluster represents a value assigned to the corresponding variable with the least number of weighted constraint violations. Clusters are appended to the list $\mathcal{F}$ if and only if its *on* label node connecting to k_h in a convergence step. Therefore, an empty list $\mathcal{F}$ at the end of a convergence step implies all *on* label nodes receive a zero input. A solution is found if all inputs of the *on* label nodes are zero. The overall PSS algorithm is shown in Figure 3.2.

Definition 3.1 A *convergence step* is one execution of the codes from line 11 to line 28 in Figure 3.2.

Lemma 3.1 Denote $I_{(p_i,\mathcal{X})} = \sum_{j \in \mathcal{X}, n_j \text{ is connected to } p_i} W_{p_i n_j} O_{n_j}$, where $\mathcal{X} \neq \emptyset$ is a set of clusters, p_i is the *on* label node in cluster i. Let $\mathcal{U}$ be the set of all clusters in the network, and $\mathcal{U}_{\mathcal{F}}$ be the set of clusters in the list $\mathcal{F}$.
At the end of a convergence step (Figure 3.2 line 28),

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}.$$

Proof. We use Mathematical Induction to prove the lemma. Let $\mathcal{U}_{\mathcal{F}}^{(n)}$ be the set of clusters in the list $\mathcal{F}$ at the nth convergence step. Before starting the convergence procedure of PSS, the list $\mathcal{F}$ is initialized by appending all clusters in an arbitrary order (Figure 3.2 line 11). At the first convergence step, the list $\mathcal{F}$ is not empty, and one cluster h must be removed from the list $\mathcal{F}$ (Figure 3.2 lines 13-15). Therefore, $\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(1)} = \{h\}$. There are three cases afterward:

1. $I_{p_h} = 0$ (The condition of line 16 in Figure 3.2 is false).
   Therefore, $I_{(p_h,\{h\})}$ is also zero as no other clusters in $\{h\}$ with their *on* label nodes connected to p_h. The lemma holds in this case.

2. $I_{p_h} \neq 0$ (The condition of line 16 in Figure 3.2 is true) and $I_{k_h} = 0$ (Figure 3.2 lines 17-18).
   Therefore, $I_{(k_h,\{h\})}$ is also zero as no other clusters in $\{h\}$ with their *on* label nodes connected to k_h. The lemma holds in this case.

3. $I_{p_h} \neq 0$ (The condition of line 16 in Figure 3.2 is true) and $I_{k_h} \neq 0$ (Figure 3.2 lines 17-18).
   Therefore, $I_{(k_h,\{\mathrm{h}\})}$ is zero as no other clusters in $\{h\}$ with their *on* label nodes connected to k_h. The lemma holds in this case.

As a result, the lemma holds for the first convergence step.
Assume that the lemma holds for the rth convergence step. At the $(r+1)$st convergence step, the list $\mathcal{F}$ is not empty, and one cluster h must be removed from the list $\mathcal{F}$ (Figure 3.2 lines 13-15). Therefore, $\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)} = \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)} \cup \{h\} - \mathcal{R}$, where $\mathcal{R}$ is a set of clusters with their *on* label nodes connected to k_h. There are three cases afterward:

1. $I_{p_h} = 0$ (The condition of line 16 in Figure 3.2 is false).
   It means that $I_{(p_h,\mathcal{U})} = 0$ and $\mathcal{R} = \emptyset$. Therefore, $I_{(p_h,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})}$ is also zero as $(\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)}) \subseteq \mathcal{U}$.

Since

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r)})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}$$

and

$$I_{(p_h,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})} = 0.$$

Therefore,

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)}$$

The lemma holds in this case.

2. $I_{p_h} \neq 0$ (The condition of line 16 in Figure 3.2 is true) and $I_{k_h} = 0$ (Figure 3.2 lines 17-18).

   It means that $I_{(k_h,\mathcal{U})} = 0$ and $\mathcal{R} = \emptyset$. Therefore, $I_{(k_h,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})}$ is also zero as $(\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)}) \subseteq \mathcal{U}$. Note that k_h is the current *on* node in cluster h.

   Since

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r)})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}$$

   and

$$I_{(k_h,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})} = 0.$$

   Therefore,

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)}$$

   The lemma holds in this case.

3. $I_{p_h} \neq 0$ (The condition of line 16 in Figure 3.2 is true) and $I_{k_h} \neq 0$ (Figure 3.2 lines 17-18).

   As all clusters with their *on* label nodes connecting to k_h are no longer in the set $(\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)})$ (Figure 3.2 lines 19-20), $I_{(k_h,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})}$ becomes zero at line 28 of Figure 3.2.

   Since

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r)})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}$$

and

$$(\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}) - \mathcal{R} \subset (\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}).$$

Therefore,

$$I_{(p_i,(\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r)})-\mathcal{R})} = 0, \forall i \in (\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}) - \mathcal{R}).$$

Since

$$I_{(p_i,(\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r)})-\mathcal{R})} = 0, \forall i \in (\mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r)}) - \mathcal{R})$$

and

$$I_{(k_h,(\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r)})-\mathcal{R})} = 0.$$

Therefore,

$$I_{(p_i,\mathcal{U}-\mathcal{U}_{\mathcal{F}}^{(r+1)})} = 0, \forall i \in \mathcal{U} - \mathcal{U}_{\mathcal{F}}^{(r+1)}$$

The lemma holds in this case.

As a result, the lemma holds for $(r+1)$st convergence step.
By the principle of Mathematical Induction, the lemma holds for all convergence steps. □

Theorem 3.1 PSS is in a solution state if the list $\mathcal{F}$ is empty at the end of a convergence step. If PSS is in a solution state, then either the list $\mathcal{F}$ is empty, or it will become empty in a finite number of convergence steps.

Proof. We first prove the statement: "PSS is in a solution state if the list $\mathcal{F}$ is empty at the end of a convergence step."

Since the list $\mathcal{F}$ is empty at the end of a convergence step, the set $\mathcal{U}_{\mathcal{F}}$ is an empty set in Lemma 1. By Lemma 1, all inputs of the *on* label nodes in $\mathcal{U}$ must be zero at the end of a convergence step. When all inputs of the *on* label nodes in all clusters are zero, no two *on* label nodes are connected to each other. This network state represents a solution state.

We then prove the statement: "If PSS is in a solution state, then either the list $\mathcal{F}$ is empty, or it will become empty in a finite number of convergence steps."

Since PSS is in a solution state, all inputs of the *on* label nodes in the network are zero. There are two cases for the state of the list $\mathcal{F}$. Suppose the list $\mathcal{F}$ is empty, then the statement is trivially true. Suppose the list $\mathcal{F}$ is not empty, a cluster h must be removed from the list $\mathcal{F}$ in each convergence step (Figure 3.2 lines 13-15). As all inputs of the *on* label nodes are zero, nothing needs to be done in h and no cluster is appended to the list $\mathcal{F}$ (Figure 3.2, the condition of line 16 is false). The list $\mathcal{F}$ will eventually become empty as one cluster is removed from it in each convergence step (Figure 3.2 lines 13-15) and the number of clusters in the list $\mathcal{F}$ is finite. □

3.1.3 An Illustrative Example

We show an example on the well-known N-Queens problem to illustrate the execution of PSS. N-Queens problem is a puzzle game, which consists of placing N queens on a $N \times N$ chessboard so that no two queens attack each other. This puzzle game can be modeled as CSP with N variables. Each variable with domain $\{1, 2, \ldots, N\}$. The $3 \times N(N-1)/2$ constraints state that no pair of queens can ever be on the same row, up-diagonal or down-diagonal. In this example, we use the 4-Queens problem as a demonstration (Figure 3.3). For clarity of presentation, we have omitted the connections between the label nodes in the figure. Figure 3.3(a) shows the initial network state. One label node in each cluster is turned *on*. The list contains all clusters with an arbitrary order initially. In the first convergence step, cluster $X1$ is removed from the list. Each label node calculates its input. As the current *on* label node 1_{X1} receives a zero input, it remains *on* in this convergence step (Figure 3.3(b)). Cluster $X2$ is removed in the next convergence step. Since the current *on* label node 3_{X2} does not receive a zero input, it must be turned *off*. The label node 1_{X2} has the minimum input, and it is turned *on* in cluster $X2$. As the *on* label node in cluster $X1$ is connected to 1_{X2}, $X1$ is appended to the list (Figure

```
/* Initialize the network */
for each $W_{m_i n_j}$ do
    $W_{m_i n_j} \leftarrow 1$
end for
Let $\mathcal{U}$ be a set and all clusters are in $\mathcal{U}$ initially
while $\mathcal{U}$ is not empty do
    select and remove a cluster $x$ from $\mathcal{U}$
    turn on a label node in $x$ with minimum input,
        breaking tie by random selection
end while
append all clusters to a list $\mathcal{F}$ in an arbitrary order
/* Convergence Step */
while list $\mathcal{F}$ is not empty do
    remove and get the head cluster $h$,
        denote $p_h$ as its on label node
    if input of $p_h \neq 0$
        turn on a label node $k_h$ ($\neq p_h$) with minimum input,
            breaking tie by random selection
        append clusters with their on label nodes connecting to $k_h$
            to the list $\mathcal{F}$
        for all clusters $j$ ($\neq h$) do
            denote $n_j$ as its on node
            if $n_j$ is connecting to $p_h$
                $W_{p_h n_j} \leftarrow W_{p_h n_j} + O_{n_j}$
            end if
        end for
    end if
end while
```

Figure 3.2: The algorithm of PSS.

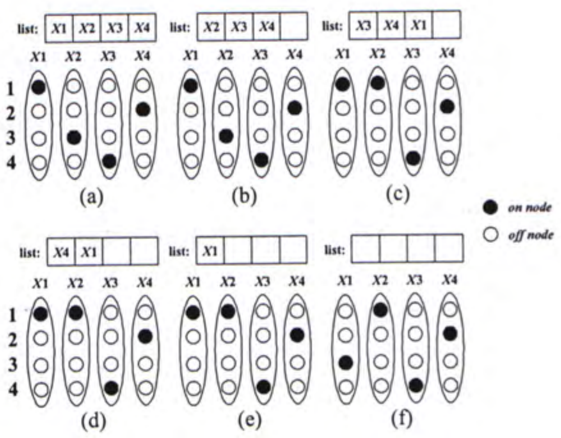


Figure 3.3: PSS: 4-Queens example.

3.3(c)). Since the previous *on* label node 3_{X2} is connected to the *on* label node 4_{X3}, the weight $W_{3_{X2}4_{X3}}$ is updated. In the next two convergence steps, both *on* label nodes of $X3$ and $X4$ receive a zero input, and no changes occur in the network (Figure 3.3(d) and (e)). In the fifth convergence step, cluster $X1$ is removed from the list again. The label node 3_{X1} receives the minimum input (zero input) and is selected to turn *on*. As there are no clusters with *on* label nodes connecting to 3_{X1}, no clusters are appended to the list. Since the previous *on* label node 1_{X1} is connected to the *on* label node 1_{X2}, the weight $W_{1_{X1}1_{X2}}$ is updated. As the list becomes empty at the end of this convergence step, a solution is found (Figure 3.3(f)).

3.2 Incremental Progressive Stochastic Search

As mentioned in the previous section, PSS works on a complete assignment and performs a heuristic search to find a solution. In this section, we introduce a variant of PSS which is called incremental PSS (IPSS). IPSS selects one cluster at a time. One label node in the selected cluster is turned *on*. The aim of the search in IPSS is to find a consistent partial assignment. This partial solution is then extended until a complete solution is obtained. The details of the network architecture and convergence procedure of IPSS are discussed in the following sections.

3.2.1 Network Architecture

The network architecture of IPSS is the same as that of PSS. However, the definition of the state of the network is refined. In PSS, a state of the network represents a complete assignment of values to the variables from their respective domains. A cluster without any *on* label node corresponds to a variable that has not been assigned a value. In IPSS, however, a state of the network represents a *partial* assignment of values to the variables from their respective domains. Therefore, any network state in which no two *on* label nodes connect to each other represents a partial solution to the CSP.

3.2.2 Convergence Procedure

The convergence procedure of IPSS is based on that of PSS. The network is initialized by setting all label nodes in every cluster to the *off* state. This network state denotes an empty assignment at the beginning.

After the network initialization, IPSS divides the set of clusters in the network into two subsets. One is a subset $\mathcal{U}_a$ of clusters, in which each cluster has one *on* label node. Another one is a subset $\mathcal{U}_u$ of clusters, in which all label nodes in these cluster are in *off* state. Initially, all clusters are in the set $\mathcal{U}_u$, and the set $\mathcal{U}_a$ is empty. Clusters are selected from $\mathcal{U}_u$ and moved to $\mathcal{U}_a$ one by one. After a cluster i is moved to the set $\mathcal{U}_a$, each of the label nodes in cluster i calculates its input, and the label node m_i in cluster i with the minimum input is turned *on*. Ties are broken randomly. We also maintain a list $\mathcal{F}$ to be used in the convergence procedure. The list $\mathcal{F}$ is initialized to be empty. Any cluster in the set $\mathcal{U}_a$ with its *on* label node connecting to m_i is appended to the list $\mathcal{F}$ if it is not already in the list. Then we apply the convergence step in PSS to the set $\mathcal{U}_a$ until the list $\mathcal{F}$ becomes empty. After the list $\mathcal{F}$ becomes empty, another cluster is selected from the set $\mathcal{U}_u$ and moved to $\mathcal{U}_a$. The convergence procedure in IPSS terminates when the set $\mathcal{U}_u$ and the list $\mathcal{F}$ are both empty. The overall IPSS algorithm is shown in Figure 3.4.

3.2.3 An Illustrative Example

We use 4-Queens problem to illustrate the execution of IPSS (Figure 3.5). For clarity of presentation, we have omitted the connections between the label nodes in the figure. Initially, all label nodes in the network are in *off* state. The set $\mathcal{U}_u$ contains all clusters. The set $\mathcal{U}_a$ and the list are both empty (Figure 3.5(a)). In Figure 3.5(b), cluster $X1$ is selected from $\mathcal{U}_u$ and moved to $\mathcal{U}_a$. As all label nodes in $X1$ receive a zero input, random selection is made to break the tie. We assume the label node 1_{X1} is turned *on*. Since no *on* label nodes connect to 1_{X1}, no clusters are appended to the list. Figure 3.5(c) shows the next network state. Cluster $X2$ is selected from $\mathcal{U}_u$ and moved to $\mathcal{U}_a$. Each of the label nodes calculates its input. The label nodes 3_{X2} and 4_{X2} both receive the minimum (zero) input, random choice is made. We assume the label node 3_{X2} is turned *on*. As a consistent partial assignment is obtained, another cluster is selected from $\mathcal{U}_u$. Suppose cluster $X3$ is selected from $\mathcal{U}_u$, the label nodes 1_{X3}, 2_{X3} and 4_{X3} receive the minimum input. We assume the label node 4_{X3} is turned *on*. At this time, the *on* label node 3_{X2} is connecting to the *on* label node 4_{X3}, and so cluster $X2$ is appended to the list (Figure 3.5(d)). A non-empty list at the end of each convergence step indicates that the network state represents an inconsistent partial assignment. Therefore, cluster $X4$ will be selected from $\mathcal{U}_u$ if the list becomes empty at the end of the convergence step. Since the list is not empty, the head cluster $X2$ is removed from the list. The current *on* label node 3_{X2} receives a non-zero input, and it

```
/* Initialize the network */
all label nodes in the clusters are in off state
$\mathcal{U}_u$ = the set of all clusters
$\mathcal{U}_a = \emptyset$
for each $W_{m_i n_j}$ do
    $W_{m_i n_j} \leftarrow 1$
end for
initialize a list $\mathcal{F}$ to be an empty list
while $\mathcal{U}_u$ is not empty do
    select a cluster $i \in \mathcal{U}_u$
    turn on a label node $m_i$ with minimum input,
        breaking tie by random selection
    append clusters in $\mathcal{U}_a$ with the on label node connecting to $m_i$
        to the list $\mathcal{F}$
    move the cluster $i$ from $\mathcal{U}_u$ to $\mathcal{U}_a$
    /* Perform PSS to the clusters in $\mathcal{U}_a$ */
    while list $\mathcal{F}$ is not empty do
        remove and get the head cluster $h$ from the list $\mathcal{F}$,
            denote $p_h$ as its on label node
        if input of $p_h \neq 0$
            turn on a label node $k_h$ ($\neq p_h$) with minimum input,
                breaking tie by random selection
            append clusters with their on label nodes connecting to $k_h$
                to the list $\mathcal{F}$
            for all clusters $j \in \mathcal{U}_a \backslash \{h\}$ do
                denote $n_j$ as its on label node
                if $n_j$ is connecting to $p_h$
                    $W_{p_h n_j} \leftarrow W_{p_h n_j} + O_{n_j}$
                end if
            end for
        end if
    end while
end while
```

Figure 3.4: The algorithm of IPSS.

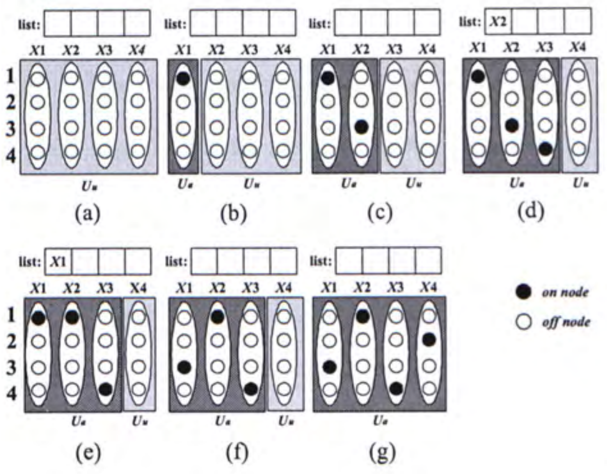


Figure 3.5: IPSS: 4-Queens example.

must be turned *off* in this convergence step. As all other label nodes receive the minimum input, random selection is made. We select 1_{X2} to be turned *on*. Unfortunately, the *on* label node 1_{X1} is connecting to 1_{X2}, and so cluster $X1$ is appended to the list (Figure 3.5(e)). Since the previous *on* label node 3_{X2} is connected to the *on* label node 4_{X3}, the weight $W_{3_{X2}4_{X3}}$ is updated. As the list is still non-empty, the head cluster $X1$ is removed from the list. The label node turned *on* this time is 3_{X1} because it is the only label node that receives a zero input in cluster $X1$ (Figure 3.5(f)). Since the previous *on* label node 1_{X1} is connected to the *on* label node 1_{X2}, the weight $W_{1_{X1}1_{X2}}$ is updated. After the above two convergence steps, the list becomes empty. Therefore, the cluster $X4$ is selected from $\mathcal{U}_u$ and moved to $\mathcal{U}_a$. The label node 2_{X4} receives a zero input and is selected to turn *on*. Since no *on* label nodes connect to 2_{X4}, no clusters are appended to the list (Figure 3.5(g)). As both $\mathcal{U}_u$ and the list are empty, a solution of 4-Queens problem is found.

3.3 Heuristic Cluster Selection Strategy

We have mentioned numerous stochastic search algorithms in Chapter 2. Different algorithms may use different *neighborhood function* to select which variable should be repaired next. GSAT [27] uses a greedy strategy. A variable will be selected next if the change of its value gives the most improvement over other variables. DLM [28] uses a descent strategy which picks any variable that has improvement.

PSS and IPSS both use a *list* $\mathcal{F}$ to store which cluster should be repaired at the next convergence step. The ordering is in a *first-in-first-out* manner. A heuristic that has been proved to improve efficiency in many cases is to integrate the idea of greedy variable ordering into PSS and IPSS. In each convergence step, a cluster with its *on* label node that has the maximum input among all *on* label nodes in other clusters is removed from the list $\mathcal{F}$. Tie is broken by random selection. We denote max-PSS and max-IPSS as variants of PSS and IPSS that use this greedy variable ordering respectively. A related heuristic called max-input ordering (MIO) for GENET or EGENET has been proposed in [29]. MIO dynamically arranges the clusters to be repaired in GENET or EGENET according to descending order of inputs for the current assignment. This approach shares the same idea with max-PSS to improve the efficiency.

We use 4-Queens problem to illustrate how the heuristic works on PSS (Figure 3.6). For clarity of presentation, we have omitted the connections between the label nodes in the figure. Figure 3.6(a) shows the initial network state. One label node in each cluster is turned *on*. The list contains all clusters with an arbitrary order initially. In each convergence step, a cluster with its *on* label node that has the maximum input among all *on* label nodes in other clusters is removed from the list. The input of *on* label node in cluster $X1, X2, X3$ and $X4$ are 0, 1, 1 and 0 respectively. As both cluster $X2$ and $X3$

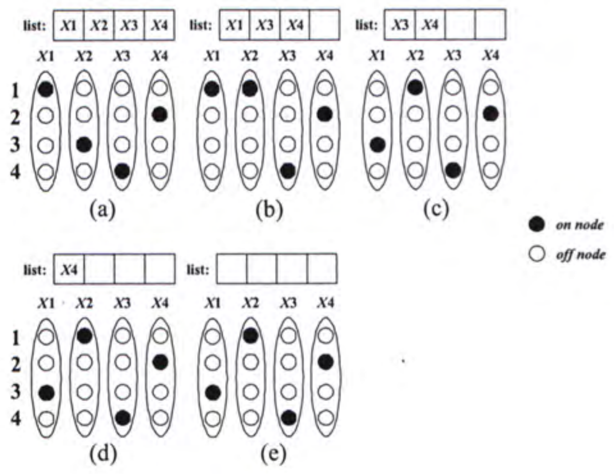


Figure 3.6: max-PSS: 4-Queens example.

have their *on* label node with maximum input, one of them will be removed from the list. In the first convergence step, we assume that cluster $X2$ is removed from the list. Each label node in cluster $X2$ calculates its input. Since the current *on* label node 3_{X2} does not receive a zero input, it must be turned *off*. The label node 1_{X2} has the minimum input, and it is turned *on* in cluster $X2$. As the *on* label node in cluster $X1$ is connected to 1_{X2}, $X1$ should appended to the list. However, cluster $X1$ is already in the list, nothing needs to be done (Figure 3.6(b)). Since the previous *on* label node 3_{X2} is connected to the *on* label node 4_{X3}, the weight $W_{3_{X2}4_{X3}}$ is updated. At the beginning of the second convergence step, the input of *on* label node in cluster $X1, X3$ and $X4$ are 1, 0 and 0 respectively. Therefore, cluster $X1$ is removed from the list. The label node 2_{X1} receives the minimum input (zero input) and is selected to turn *on* (Figure 3.6(c)). As there are no clusters with *on* label nodes connecting to 2_{X1}, no clusters are appended to the list. Since the previous *on* label node 1_{X1} is connected to the *on* label node 1_{X2}, the weight $W_{1_{X1}1_{X2}}$ is updated. In the next two convergence steps, both *on* label nodes of $X3$ and $X4$ receive a zero input, and no changes occur in the network (Figure 3.6(d) and (e)). As the list becomes empty at the end of this convergence step, a solution is found (Figure 3.6(e)).

Chapter 4

Experiments

In order to evaluate the efficiency of PSS and its variants, namely, IPSS, max-PSS and max-IPSS, experiments on four sets of problems are conducted. These include a set of N-Queens problems, a set of permutation generation problems (including increasing permutation generation and random permutation generation), a set of quasigroup completion problems (including the special cases of Latin squares) and a set of randomly generated binary CSPs (including tight CSPs and phase transition CSPs). We compare the performance of PSS and its variants with that of $\mathcal{LSDL}$(GENET) [4], the most efficient implementation of GENET that we know of, and $\mathcal{LSDL}$(IMP) [4], the most efficient variant of $\mathcal{LSDL}$.

The implementation of PSS and its variants are based on the implementation of $\mathcal{LSDL}$, which encompasses all of $\mathcal{LSDL}$(GENET), $\mathcal{LSDL}$(IMP) and the lazy variants in one implementation. Therefore, the comparison between PSS and $\mathcal{LSDL}$ is fair.

All the benchmarks are performed on a Pentium4 1.4 GHz machine with 512 Mb of memory running Linux RedHat 8.0. For each problem, 100 runs of results are recorded. The term "steps" in the tables means the number of times that the clusters are considered to select a label node to turn *on*. All the timings are measured in seconds. The timing figures without brackets are the averages of hundred runs while the figures with brackets are the medians.

All the timing results are the search time only. All problem instances used in the experiments are the same as those used in [4].

4.1 N-Queens Problems

N-Queens problem is a puzzle game, which consists of placing N queens on a $N \times N$ chessboard so that no two queens attack each other. This puzzle game can be modeled as a CSP with N variables. Each variable has a domain $\{1, 2, \ldots, N\}$. The $3 \times N(N-1)/2$ constraints state that no pair of queens can ever be on the same row, up-diagonal or down-diagonal. This set of experiments consists of 5 instances: 100-queens, 125-queens, 150-queens, 175-queens and 200-queens. Table 4.1 shows the experimental results of PSS and its variants. The results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) are presented in Table 4.2. The mean timing results are plotted in the Figure 4.1 for comparison.

Problem	PSS		IPSS	
N	Steps	**CPU time**	Steps	**CPU time**
100	119.8(117.0)	**0.0082(0.0100)**	125.4(122.0)	**0.0088(0.0100)**
125	145.0(142.5)	**0.0122(0.0100)**	149.5(147.0)	**0.0129(0.0100)**
150	166.7(165.0)	**0.0181(0.0200)**	173.2(172.0)	**0.0173(0.0200)**
175	199.8(197.0)	**0.0252(0.0200)**	198.7(195.5)	**0.0241(0.0200)**
200	221.0(218.0)	**0.0311(0.0300)**	223.2(220.0)	**0.0312(0.0300)**
	max-PSS		max-IPSS	
100	126.4(122.0)	**0.0084(0.0100)**	122.1(118.5)	**0.0071(0.0100)**
125	149.0(146.0)	**0.0135(0.0100)**	148.2(144.0)	**0.0124(0.0100)**
150	176.0(172.0)	**0.0189(0.0200)**	173.1(171.0)	**0.0175(0.0200)**
175	198.7(196.5)	**0.0245(0.0200)**	200.0(197.0)	**0.0242(0.0200)**
200	226.5(223.5)	**0.0332(0.0300)**	222.7(220.0)	**0.0310(0.0300)**

Table 4.1: PSS and its variants on N-Queens problems

As shown in Tables 4.1 and 4.2, PSS and all its variants are more efficient than $\mathcal{LSDL}$(GENET) in all cases and $\mathcal{LSDL}$(IMP) in most cases (except 100-queens). In general, the performance of PSS and its variants are almost the same, which is about 55% of the time taken by $\mathcal{LSDL}$(GENET). From the data in Table 4.2, it can be concluded that $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) do

N	$\mathcal{LSDL}$(GENET)			
	Iteration	Repairs	Learns	**CPU time**
100	42.9(30.5)	93.6(89.5)	19.6(13.5)	**0.0132(0.0100)**
125	39.5(30.0)	109.4(105.0)	18.0(13.0)	**0.0218(0.0200)**
150	37.3(30.5)	125.1(124.0)	16.7(13.5)	**0.0316(0.0300)**
175	43.1(35.0)	144.5(141.0)	19.6(16.0)	**0.0436(0.0400)**
200	44.8(36.0)	159.5(156.5)	20.3(16.0)	**0.0559(0.0600)**
N	$\mathcal{LSDL}$(IMP)			
100	23.2(17.5)	54.0(49.0)	23.2(17.5)	**0.0078(0.0100)**
125	33.3(24.5)	72.9(63.5)	33.3(24.5)	**0.0153(0.0150)**
150	27.6(19.0)	72.3(63.5)	27.6(19.0)	**0.0206(0.0200)**
175	33.5(23.0)	85.8(75.5)	33.5(23.0)	**0.0290(0.0300)**
200	34.1(24.0)	91.6(83.5)	34.1(24.0)	**0.0377(0.0400)**

Table 4.2: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on N-queens problems

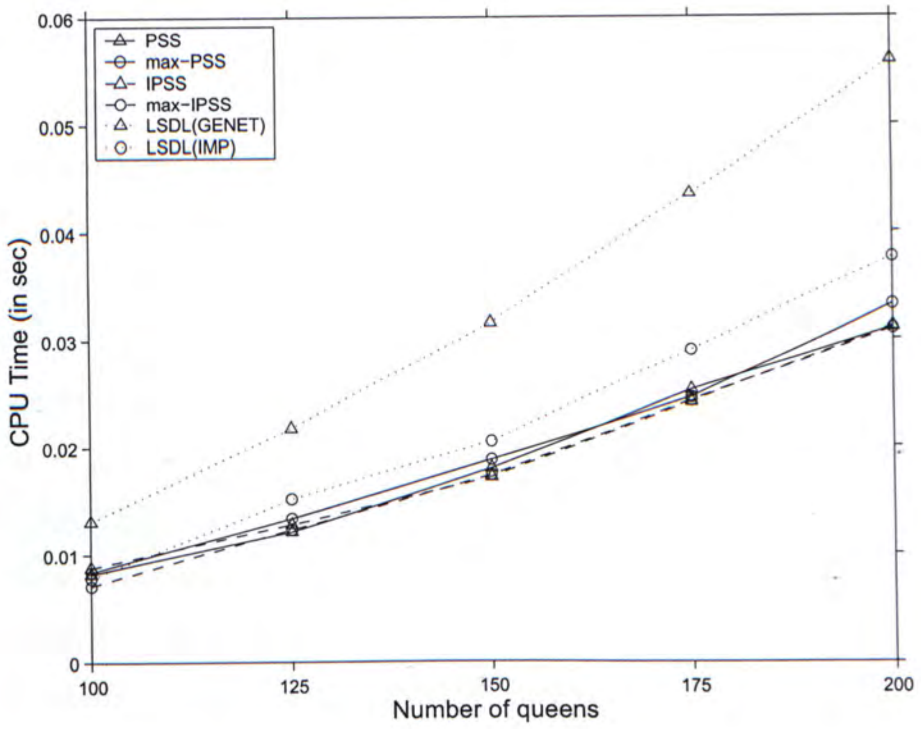
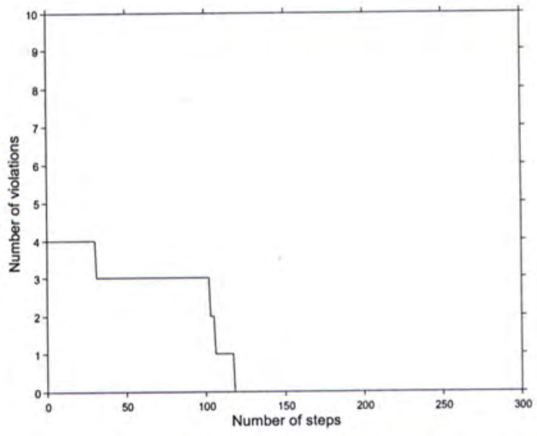


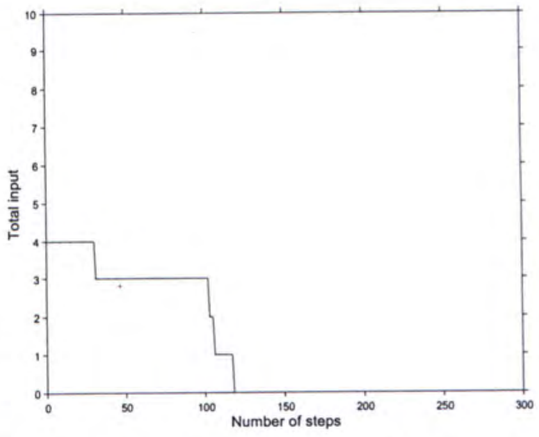
Figure 4.1: The mean time results on N-queens

learning a number of times to escape from local minima. Recall that learning is a process that updates the weights of the connections, the corresponding constraints of which are violated. We note that learning is expensive in the $\mathcal{LSDL}$ implementations. For each learning in $\mathcal{LSDL}$ implementations, the weights of several connections are updated. However, PSS and all its variants also update the weights of the connections at the end of each convergence step. If we compare the number of weights updated of $\mathcal{LSDL}$ implementations with that of PSS and its variants, we conclude that these numbers are almost the same in all problem instances. Therefore, learning is not the factor that affects the performance in this set of experiments.

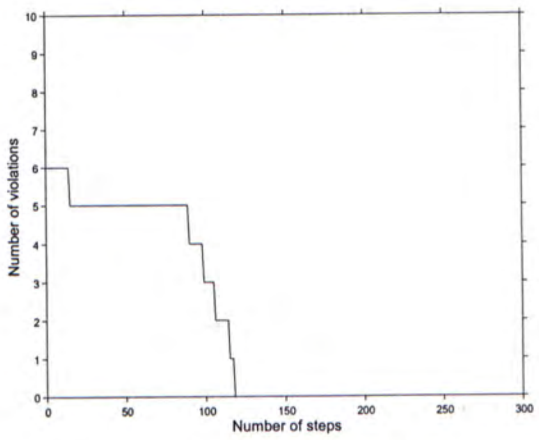
To explain why PSS and its variants have a better performance than $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP), we analyzed search processes in the experiments. From Table 4.1 and 4.2, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) use fewer repairs than PSS and all its variants. However, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) take more steps to find a solution. The number of steps taken in $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) is equal to the number of variables times the number of iterations. For example, the mean number of steps taken in $\mathcal{LSDL}$(GENET) to solve 200-queens is 8,960 (200×44.8). Among these steps, the clusters are actually repaired in only 159.5 steps and nothing really needs to be done in all other steps. Worse, these repairs have little effect on the subsequent search process. Figures 4.2 - 4.10 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 100-queens problem. Figures 4.11 - 4.19 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 200-queens problem. We can see that PSS and max-PSS quickly rush through large plateaus and the ordering to repair variables (the list $\mathcal{F}$) provides excellent direction towards solutions. For IPSS and max-IPSS, the partial solutions found can be extended easily. This is the reason why PSS and its variants have better timing results than that of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP).



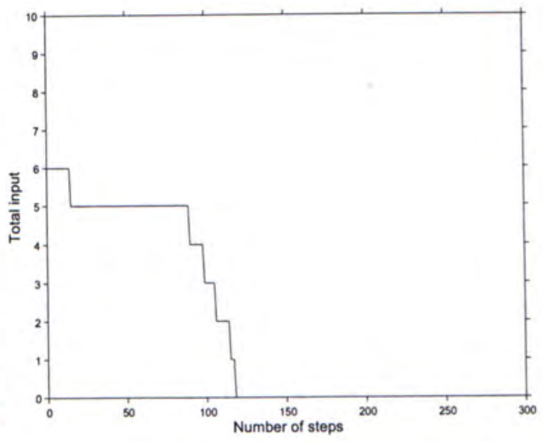
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

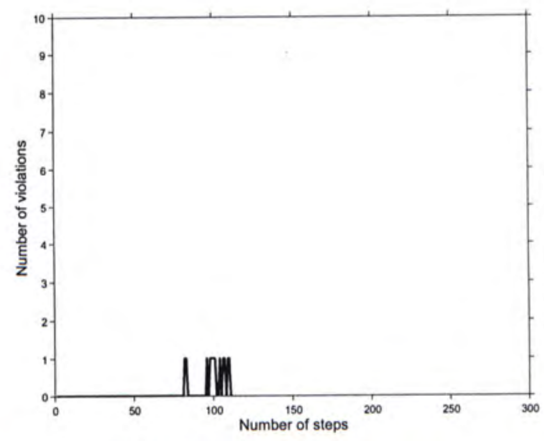


(c) max-PSS: Violation vs. Step

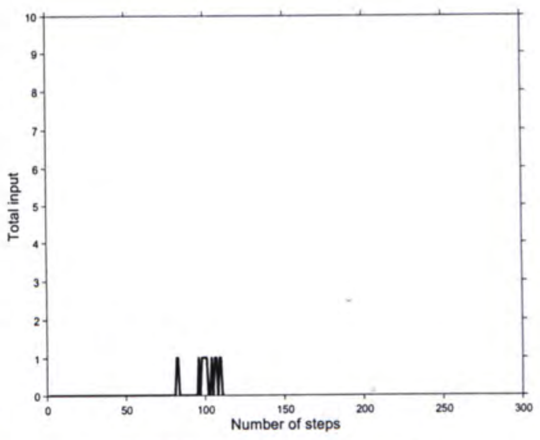


(d) max-PSS: Total input vs. Step

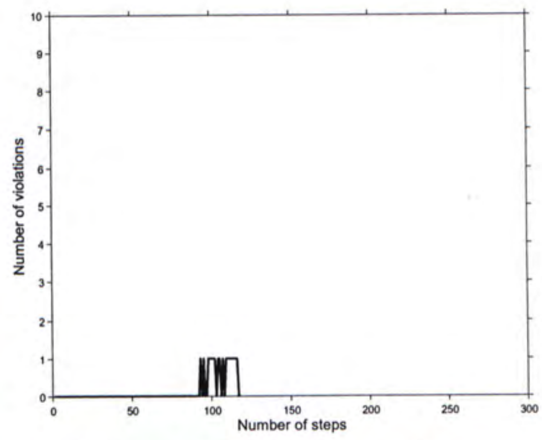
Figure 4.2: Numbers of violations and total inputs in each step of PSS and max-PSS on 100-Queens problem (average run-time case)



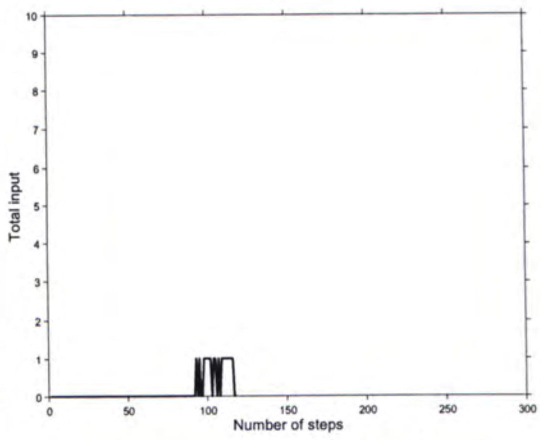
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

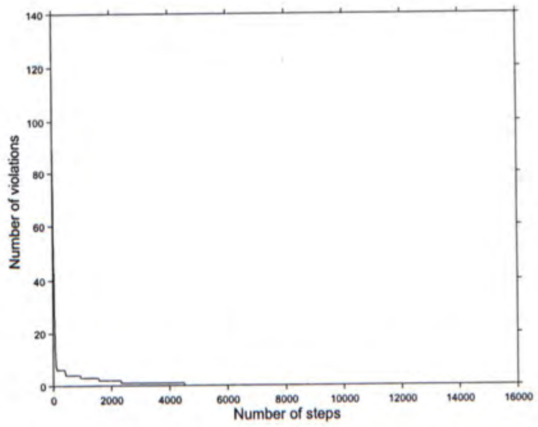


(c) max-IPSS: Violation vs. Step

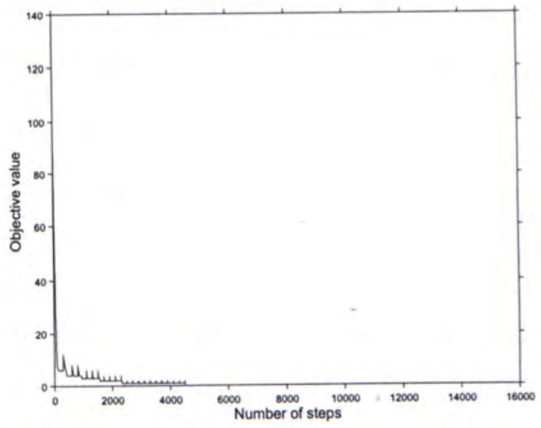


(d) max-IPSS: Total input vs. Step

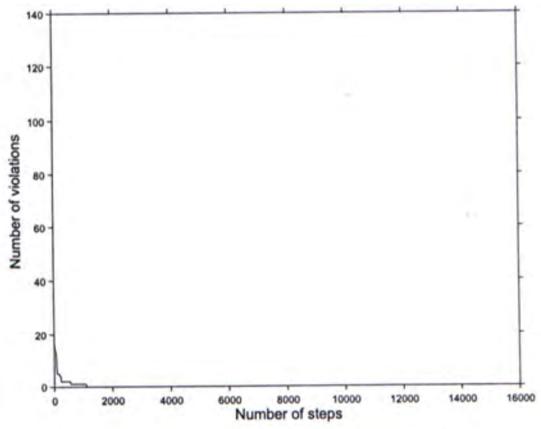
Figure 4.3: Numbers of violations and total inputs in each step of IPSS and max-IPSS on 100-Queens problem (average run-time case)



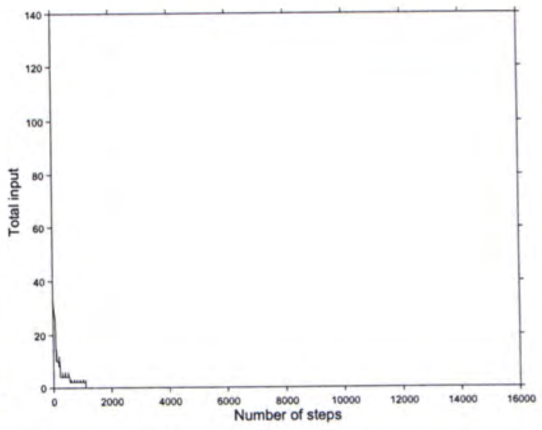
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

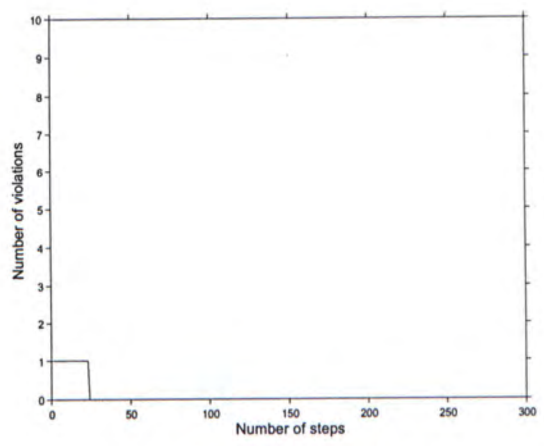


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

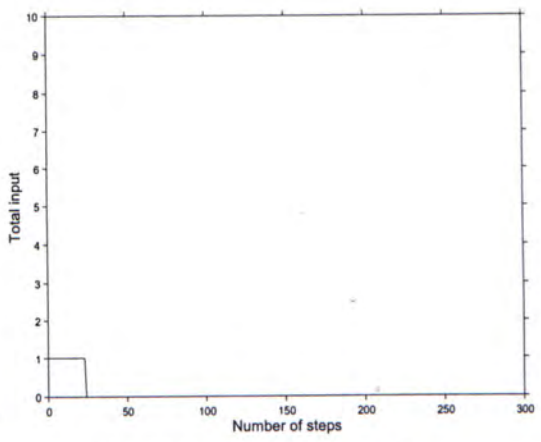


(d) $\mathcal{LSDL}$(IMP): Objective value.vs. Step

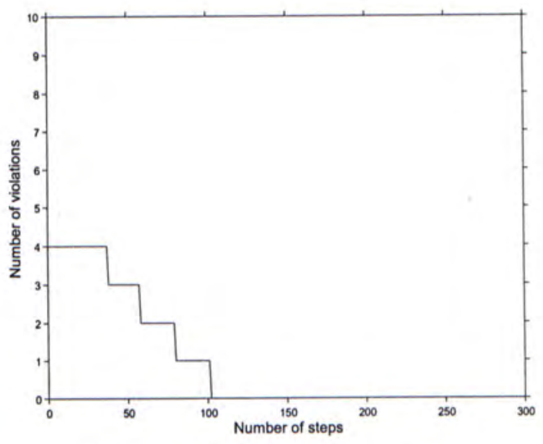
Figure 4.4: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 100-Queens problem (average run-time case)



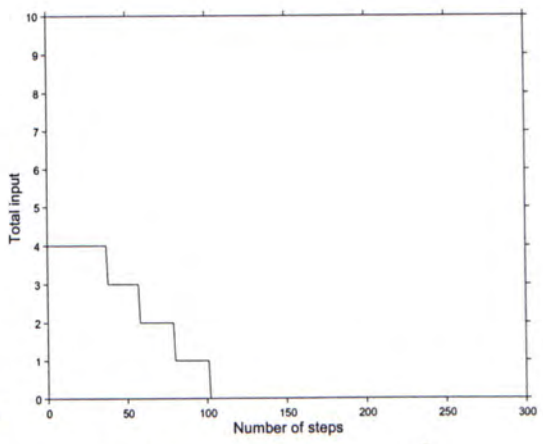
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

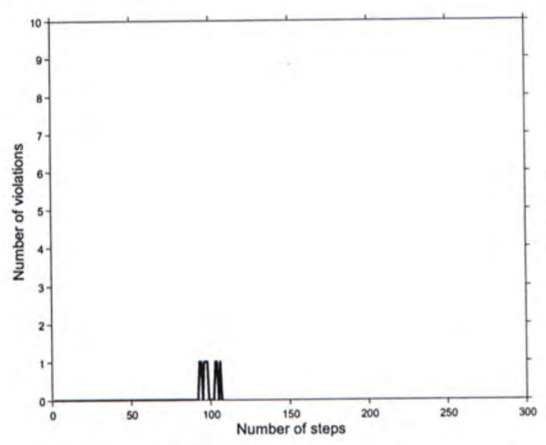


(c) max-PSS: Violation vs. Step

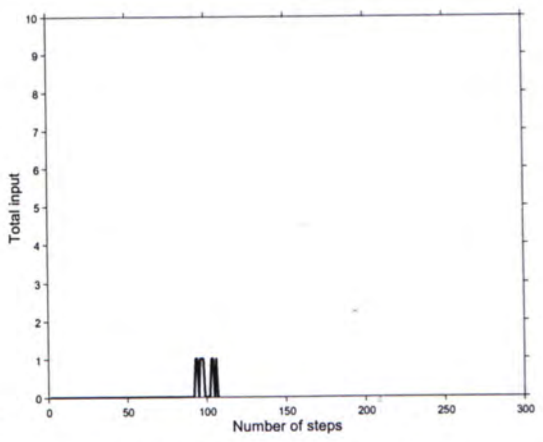


(d) IPSS: Total input vs. Step

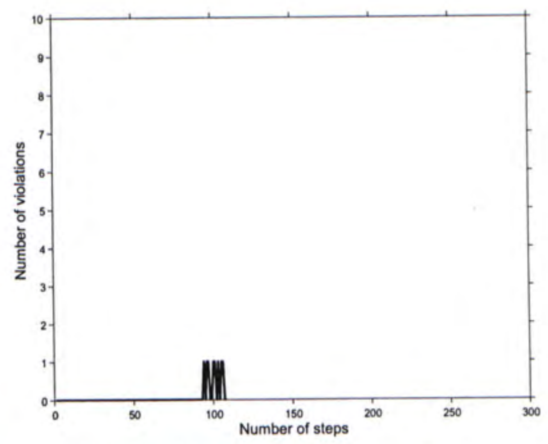
Figure 4.5: Numbers of violations and total inputs in each step of PSS and max-PSS on 100-Queens problem (short run-time case)



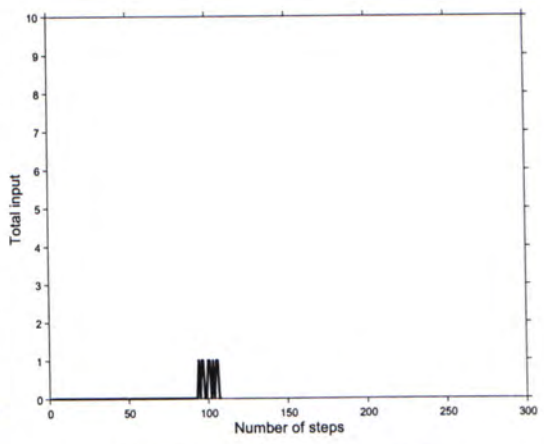
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

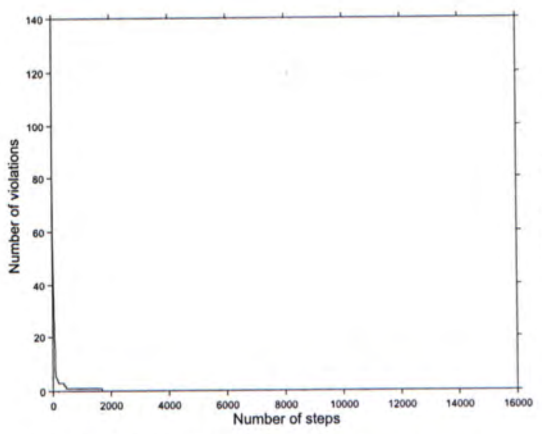


(c) max-IPSS: Violation vs. Step

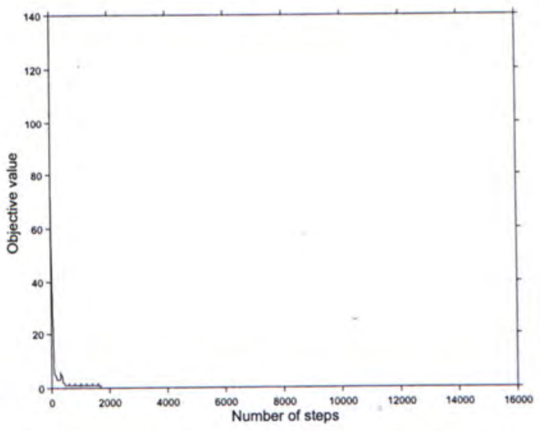


(d) PSS: Total input vs. Step

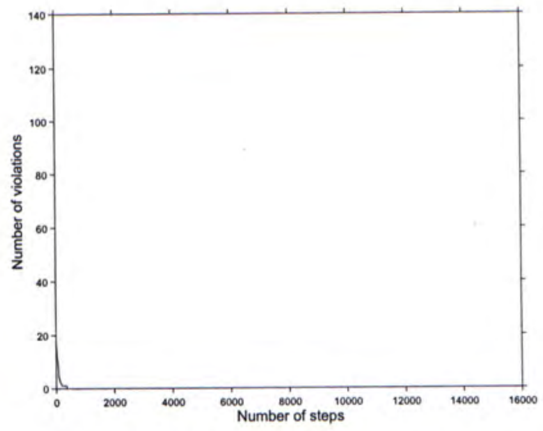
Figure 4.6: Numbers of violations and total inputs in each step of IPSS and max-IPSS on 100-Queens problem (short run-time case)



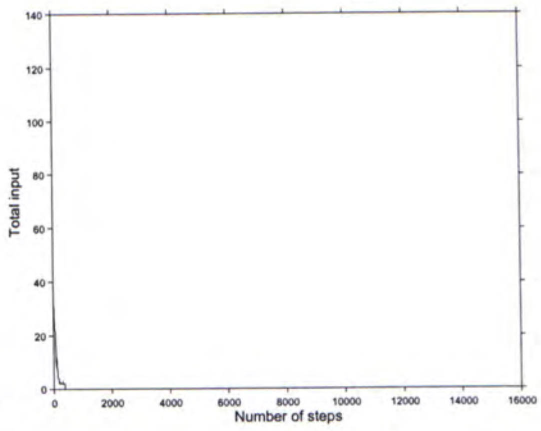
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

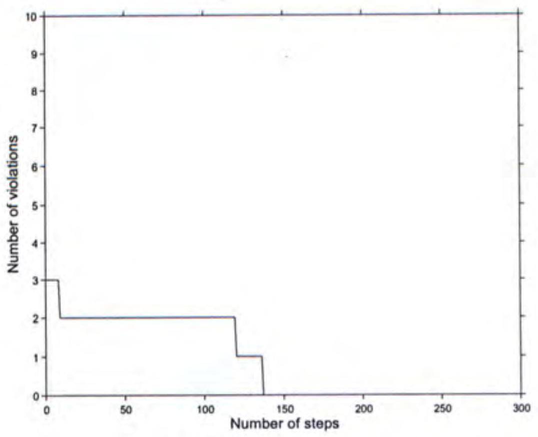


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

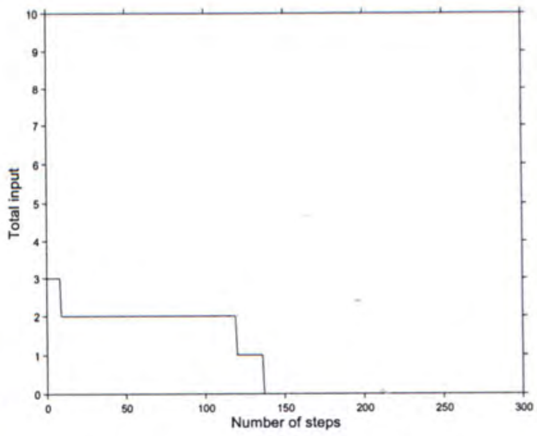


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

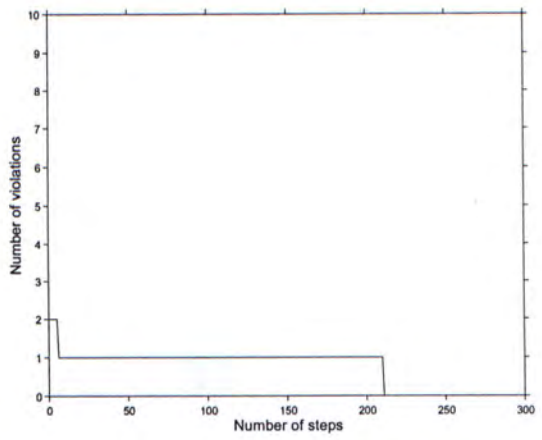
Figure 4.7: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 100-Queens problem (short run-time case)



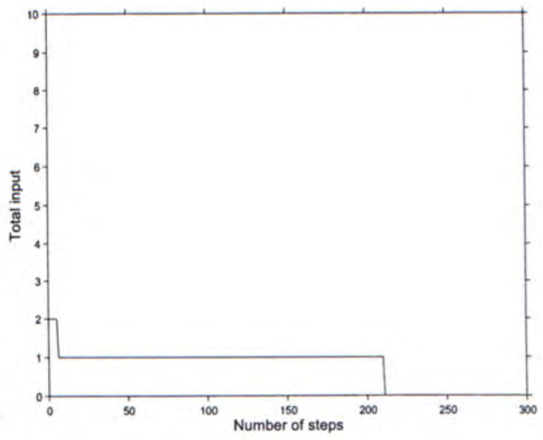
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

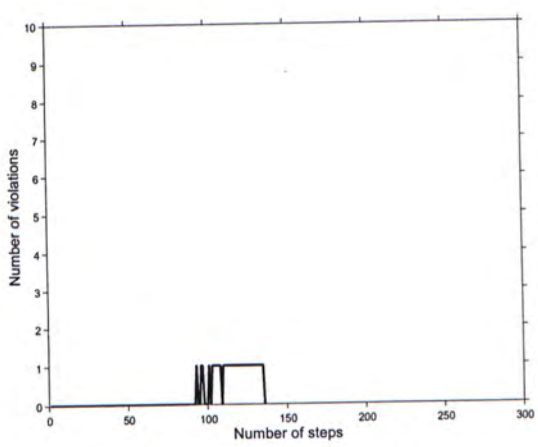


(c) max-PSS: Violation vs. Step

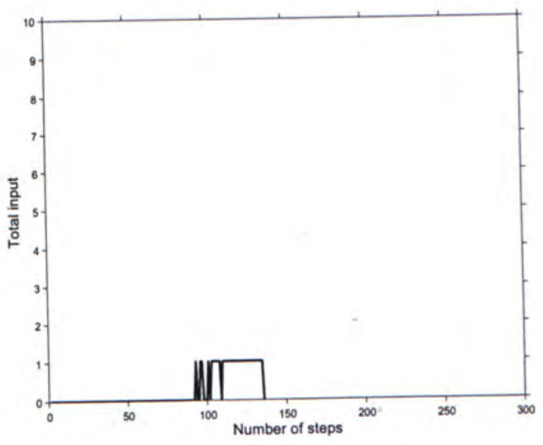


(d) max-PSS: Total input vs. Step

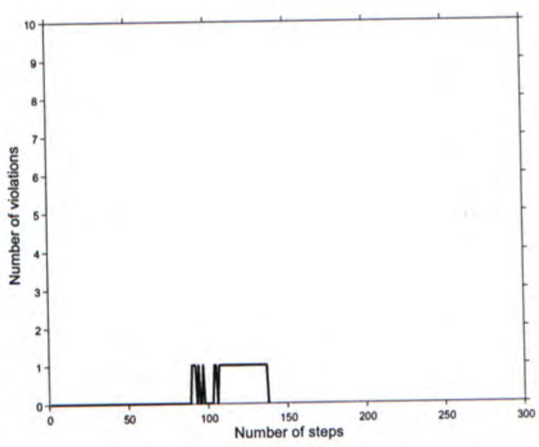
Figure 4.8: Numbers of violations and total inputs in each step of PSS and max-PSS on 100-Queens problem (long run-time case)



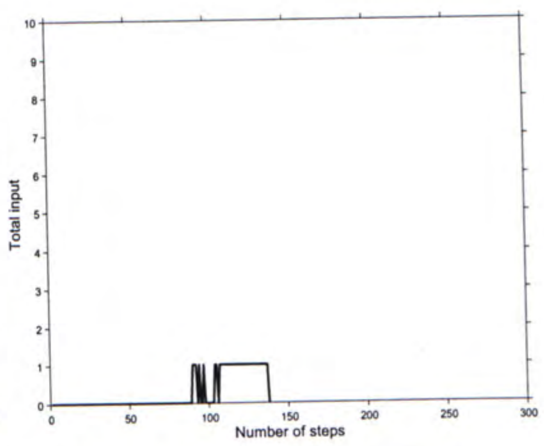
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

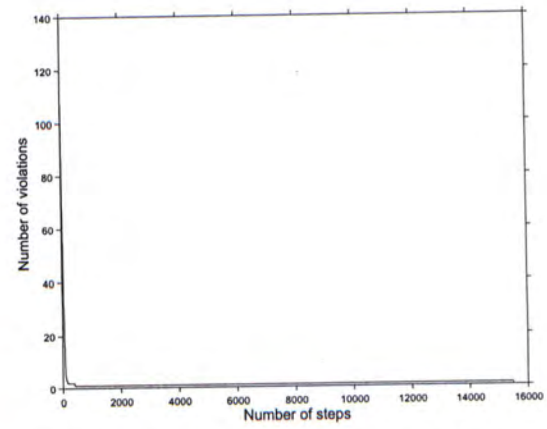


(c) max-IPSS: Violation vs. Step

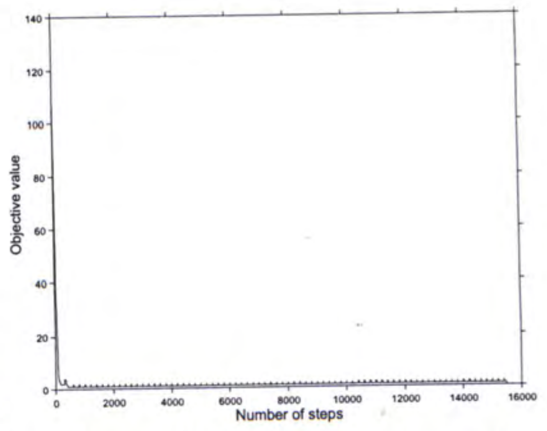


(d) max-IPSS: Total input vs. Step

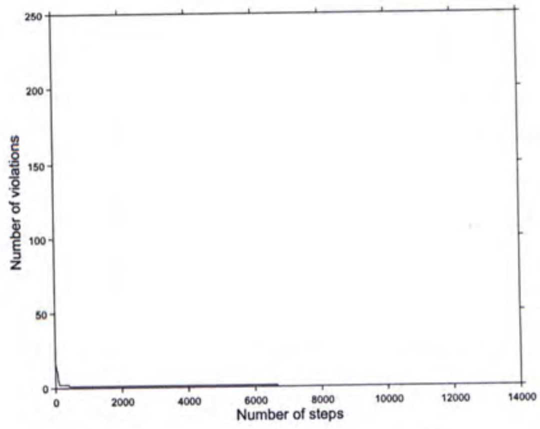
Figure 4.9: Numbers of violations and total inputs in each step of IPSS and max-IPSS on 100-Queens problem (long run-time case)



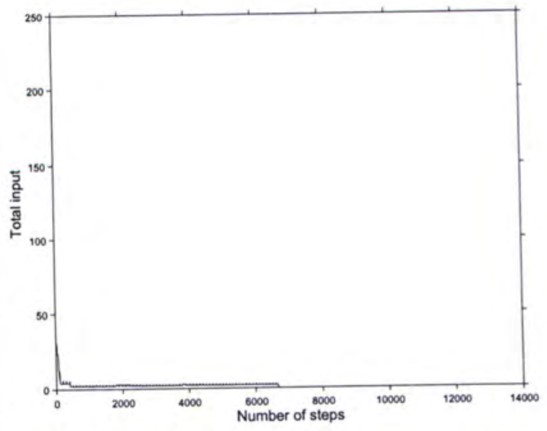
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

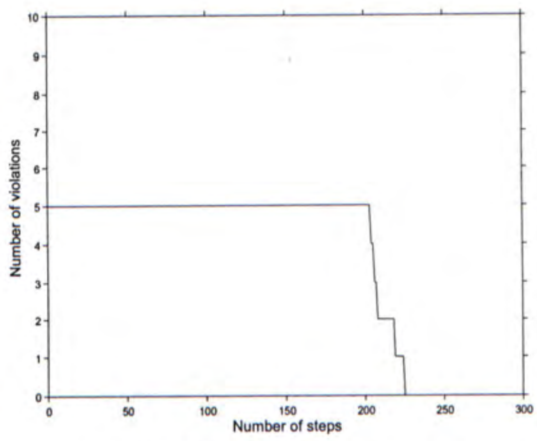


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

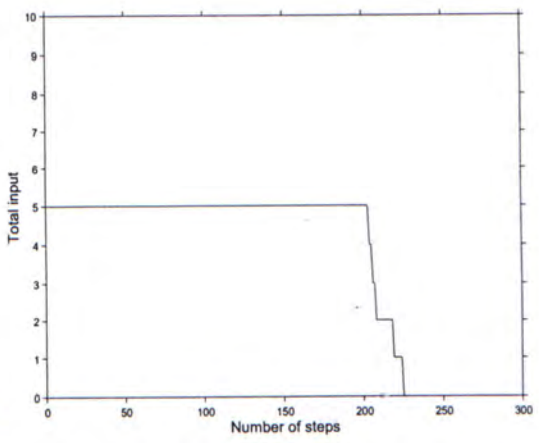


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

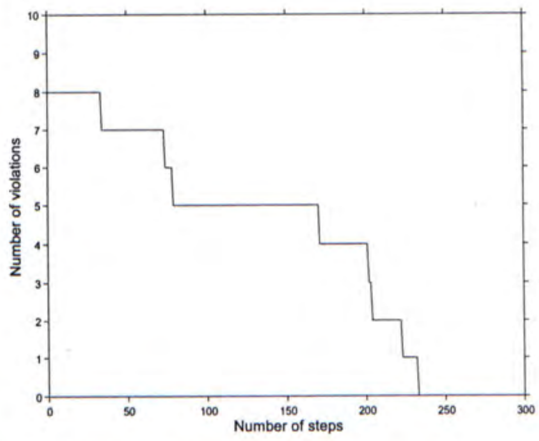
Figure 4.10: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 100-Queens problem (long run-time case)



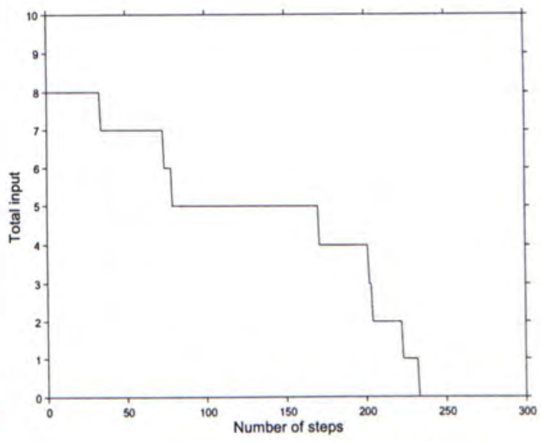
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

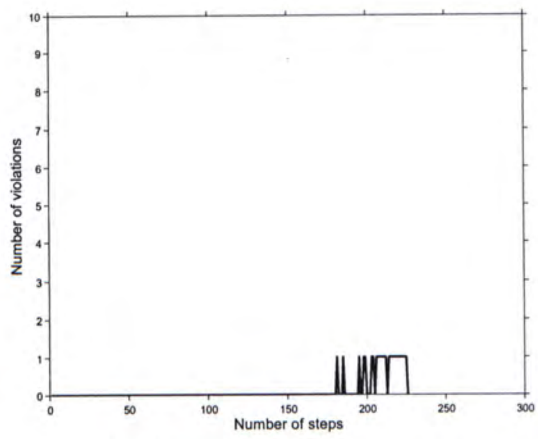


(c) max-PSS: Violation vs. Step

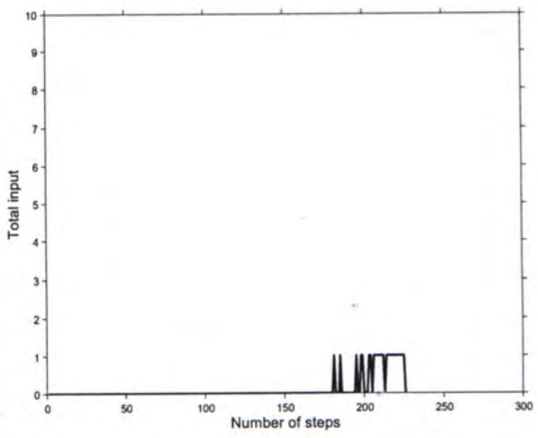


(d) max-PSS: Total input vs. Step

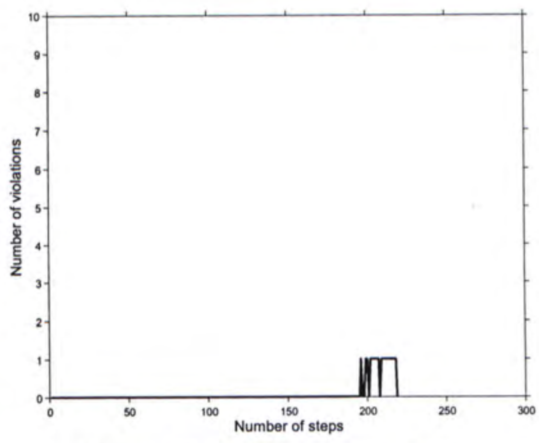
Figure 4.11: Numbers of violations and total inputs in each step of PSS and max-PSS on 200-Queens problem (average run-time case)



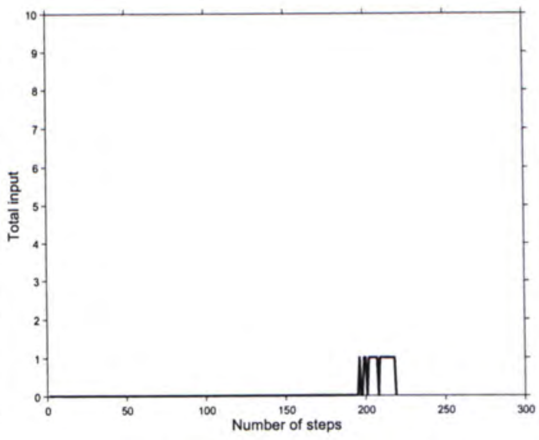
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

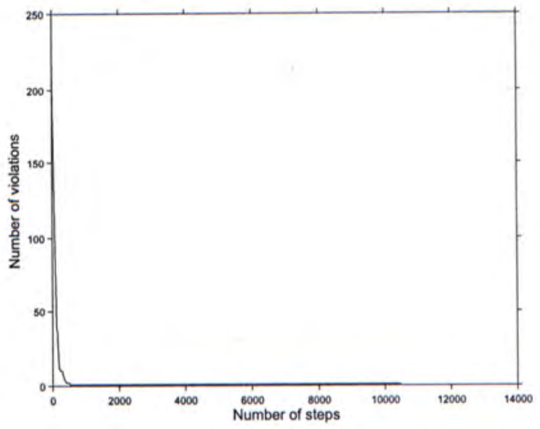


(c) max-IPSS: Violation vs. Step

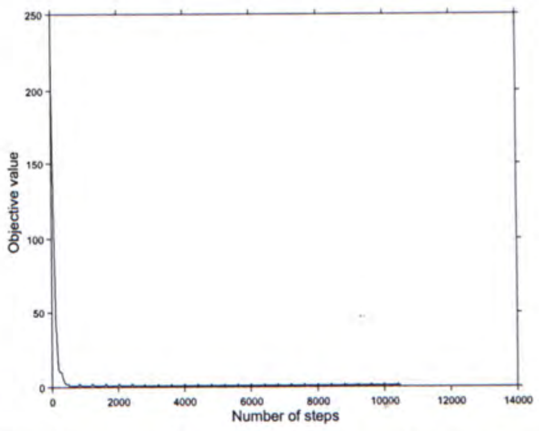


(d) max-IPSS: Total input vs. Step

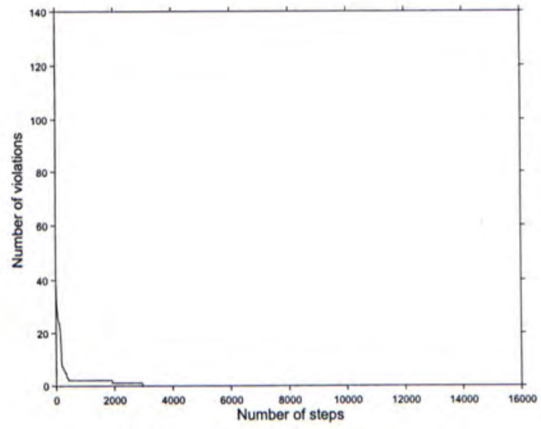
Figure 4.12: Numbers of violations and total inputs in each step of IPSS and max-IPSS on 200-Queens problem (average run-time case)



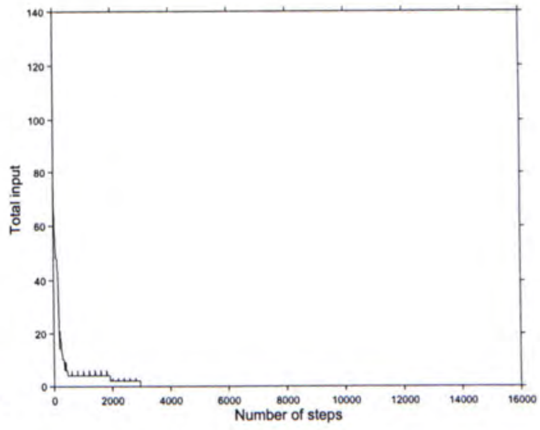
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

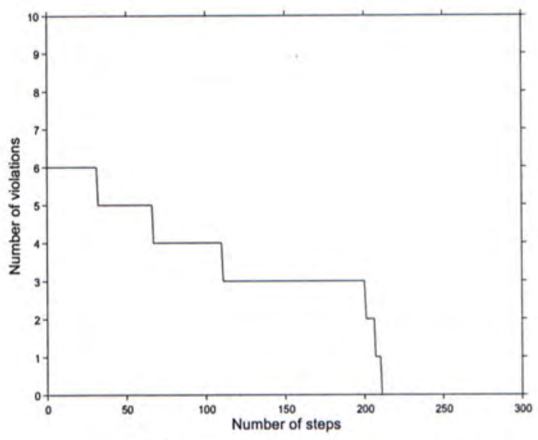


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

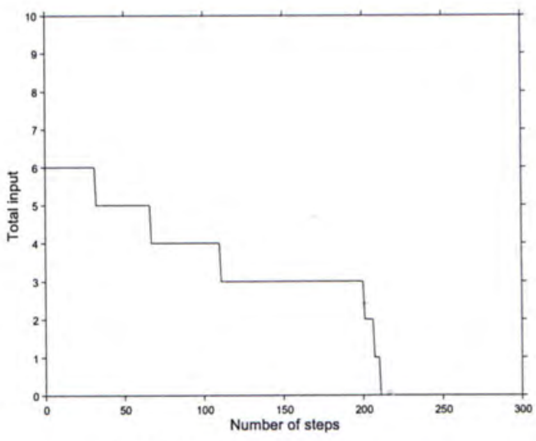


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

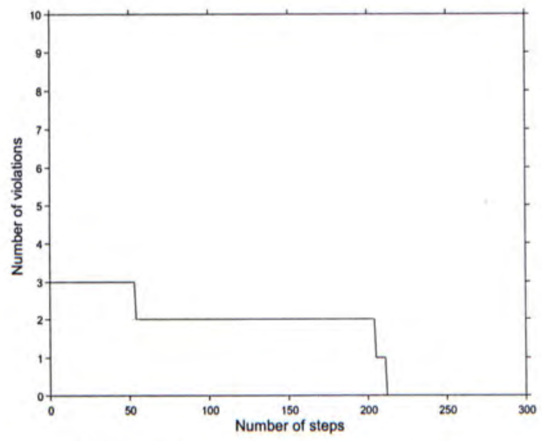
Figure 4.13: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 200-Queens problem (average run-time case)



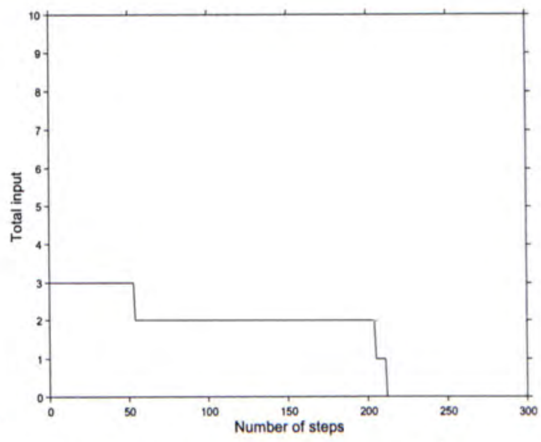
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

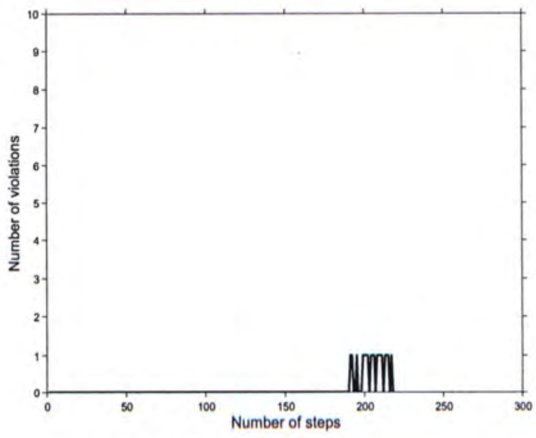


(c) max-PSS: Violation vs. Step

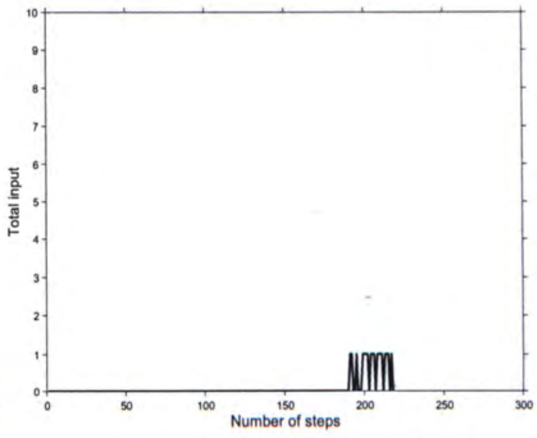


(d) max-PSS: Total input vs. Step

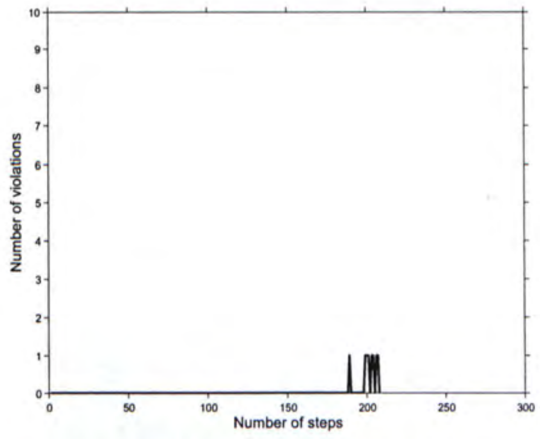
Figure 4.14: Numbers of violations and total inputs in each step of PSS and max-PSS on 200-Queens problem (short run-time case)



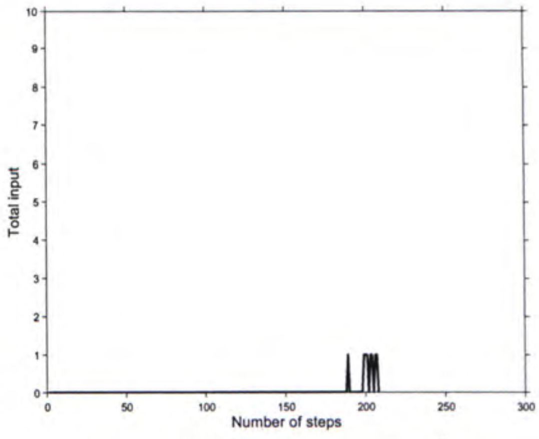
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

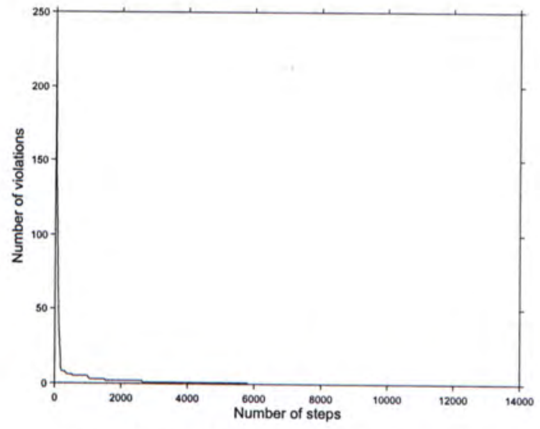


(c) max-IPSS: Violation vs. Step

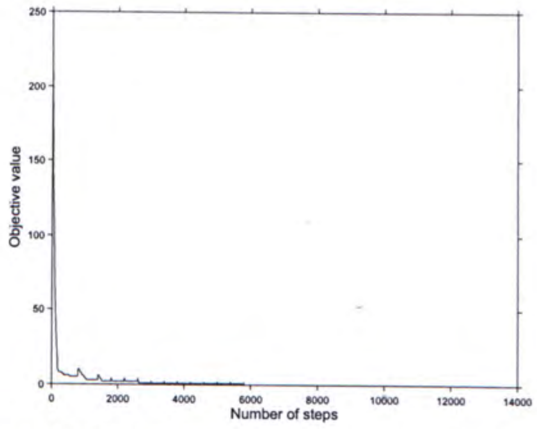


(d) max-IPSS: Total input vs. Step

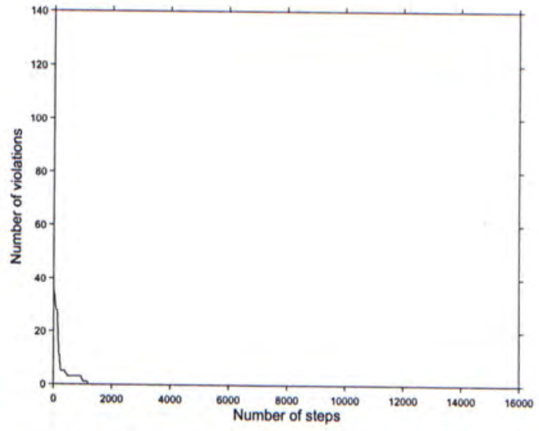
Figure 4.15: Numbers of violations and total inputs in each step of IPSS and max-IPSS on 200-Queens problem (short run-time case)



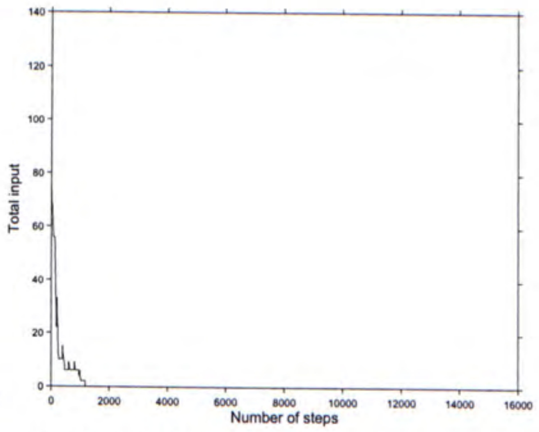
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

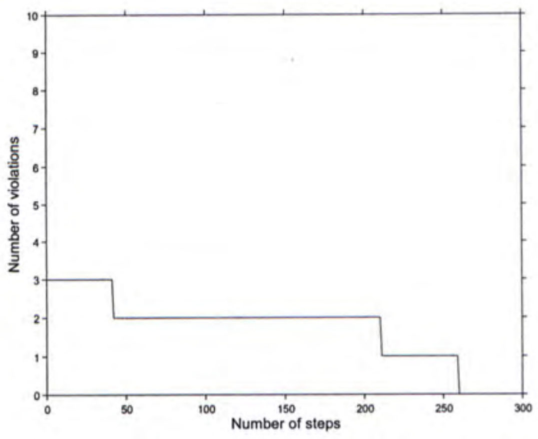


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

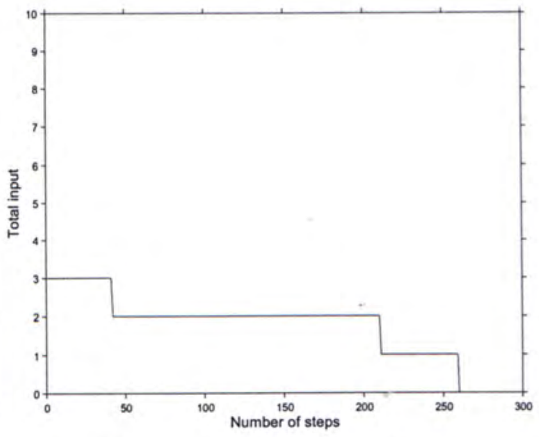


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

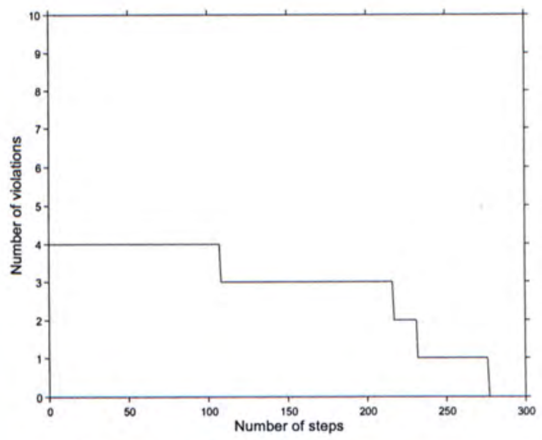
Figure 4.16: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 200-Queens problem (short run-time case)



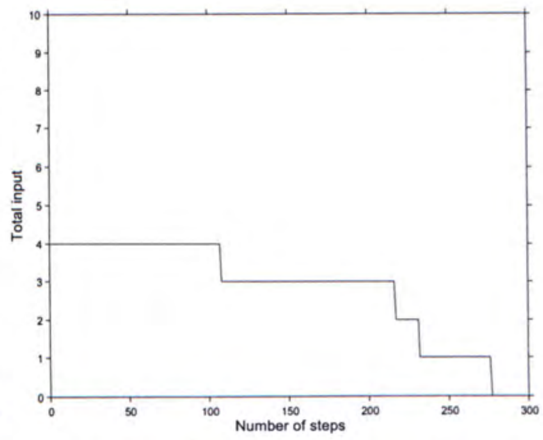
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

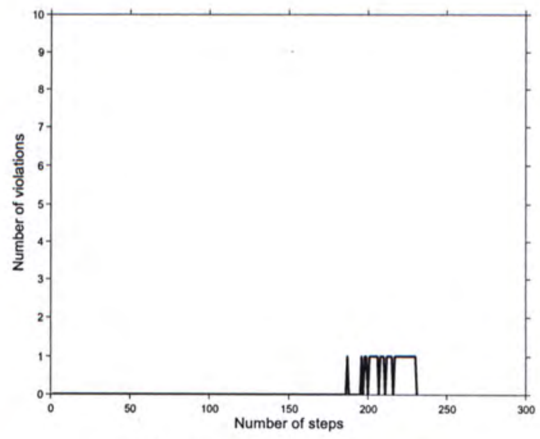


(c) max-PSS: Violation vs. Step

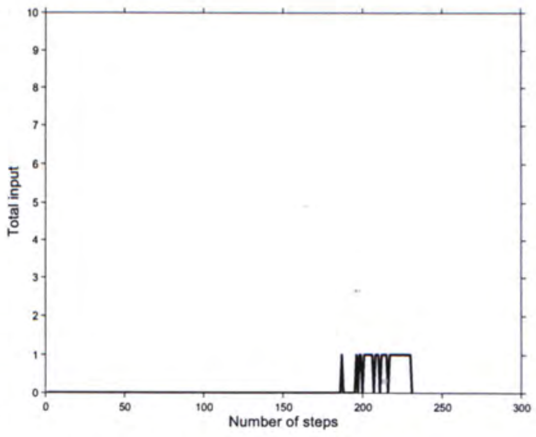


(d) max-PSS: Total input vs. Step

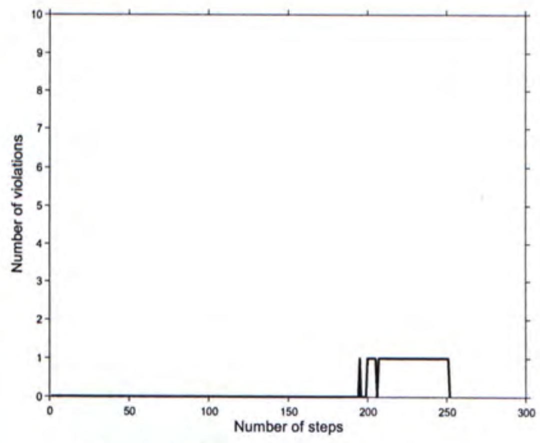
Figure 4.17: Numbers of violations and total inputs in each step of PSS and max-PSS on 200-Queens problem (long run-time case)



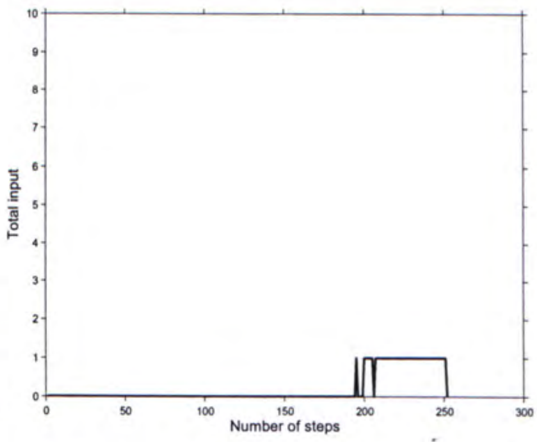
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

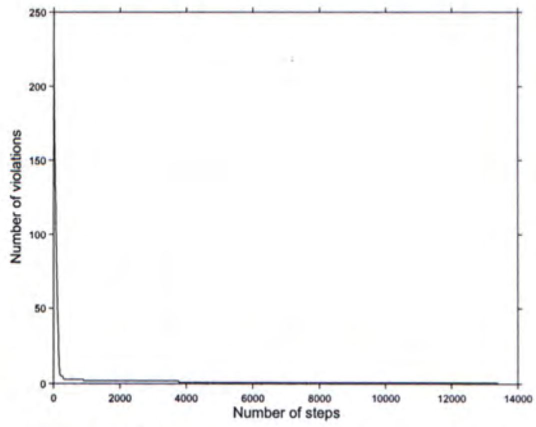


(c) max-IPSS: Violation vs. Step

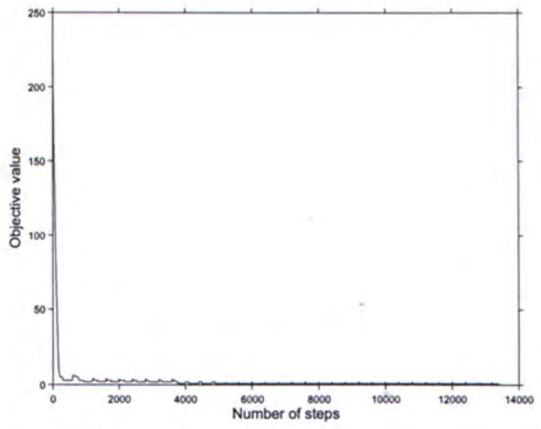


(d) max-IPSS: Total input vs. Step

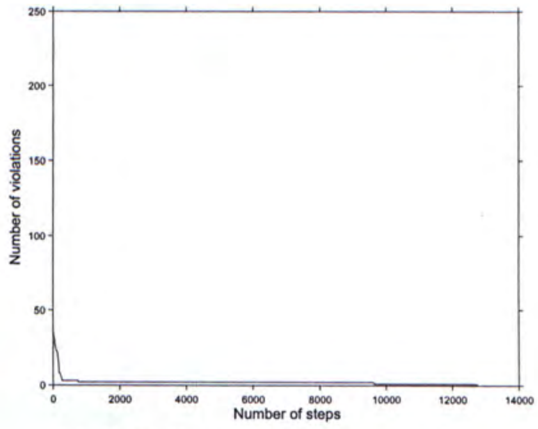
Figure 4.18: Numbers of violations and total inputs in each step of IPSS and max-IPSS on 200-Queens problem (long run-time case)



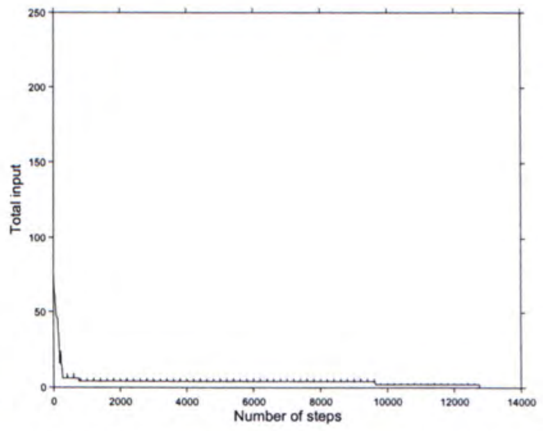
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step



(c) $\mathcal{LSDL}$(IMP): Violation vs. Step



(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

Figure 4.19: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on 200-Queens problem (long run-time case)

4.2 Permutation Generation Problems

The permutation generation problem is a combinatorial theory problem that construct a permutation p of integers 1 to n fulfilling conditions of monotonies and advances. The vector of monotonies m of size $n-1$ is defined as

$$m_i = \begin{cases} 1 & \text{if } p_{i+1} > p_i \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

for all $1 \leq i \leq n-1$. The vector of advances a of size $n-1$ is defined as

$$a_i = \begin{cases} 1 & \text{if } p_j \neq p_i + 1 \wedge p_i \neq n \text{ for all } 1 \leq j \leq i-1 \\ 0 & \text{if } p_j \neq p_i + 1 \text{ for all } i+1 \leq j \leq n \end{cases} \tag{4.2}$$

for all $1 \leq i \leq n-1$.

This problem can be modeled as a CSP with n variables. Each variable has a domain $\{1, 2, \ldots, n\}$. The constraints

$$x_i \neq x_j$$

for all $i \neq j$ and $1 \leq i, j \leq n$ restrict all variables take different values. The constraints

$$\begin{aligned} x_{i+1} &> x_i, && \text{if } m_i = 1, \\ x_{i+1} &\leq x_i, && \text{if } m_i = 0, \end{aligned}$$

for all $1 \leq i \leq n-1$ state the condition of monotonies m. The condition of advances a is stated by the constraints

$$\begin{aligned} &x_j \neq x_i + 1 \wedge x_i \neq n, && \forall 1 \leq j \leq i-1, && \text{if } a_i = 1, \\ &x_j \neq x_i + 1, && \forall i+1 \leq j \leq n, && \text{if } a_i = 0, \end{aligned}$$

for all $1 \leq i \leq n-1$.

Two types of permutation generation problems are used in this set of experiments. The first type problem is a set of increasing permutation problems. The permutation required is a sequential permutation of integers from 1 to n. The second type problem is a set of permutation problems in which the monotonies and advances are randomly generated.

4.2.1 Increasing Permutation Problems

Table 4.3 shows the experimental results of PSS and its variants on the set of increasing permutation problems, while Table 4.4 shows the experimental results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on the same set of problems. The mean timing results for increasing permutation problems are showed in the Figure 4.20 for comparison.

Problem	PSS		IPSS	
n	Steps $\times 10^3$	**CPU time**	Steps $\times 10^3$	**CPU time**
10	0.645(0.645)	**0.0000(0.0000)**	1.103(1.103)	**0.0000(0.0000)**
20	14.71(14.16)	**0.0422(0.0400)**	13.67(14.14)	**0.0405(0.0400)**
30	98.37(97.51)	**0.4417(0.4400)**	110.9(105.6)	**0.4816(0.4600)**
40	399.3(398.4)	**2.5867(2.5800)**	393.9(387.1)	**2.5104(2.4650)**
50	1123(1058)	**9.7123(9.1350)**	1164(1120)	**9.9537(9.5700)**
	max-PSS		max-IPSS	
10	0.467(0.231)	**0.0000(0.0000)**	0.572(0.572)	**0.0000(0.0000)**
20	17.23(14.32)	**0.0517(0.0500)**	24.11(25.08)	**0.0701(0.0750)**
30	140.4(147.5)	**0.6664(0.6950)**	183.6(193.7)	**0.8450(0.8900)**
40	590.0(599.2)	**4.0994(4.1550)**	812.6(821.6)	**5.5212(5.5900)**
50	1625(1648)	**15.017(15.205)**	2352(2382)	**21.524(21.785)**

Table 4.3: PSS and its variants on increasing permutation problems

Problem	$\mathcal{LSDL}$(GENET)			
n	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
10	0.361(0.365)	0.259(0.262)	0.180(0.182)	**0.0000(0.0000)**
20	11.98(11.96)	8.746(8.675)	5.997(5.980)	**0.0348(0.0300)**
30	51.42(52.68)	42.15(43.08)	25.70(26.32)	**0.2332(0.2400)**
40	160.0(153.6)	136.7(134.0)	80.03(76.81)	**1.0025(0.9900)**
50	390.6(385.3)	343.6(341.2)	195.5(192.8)	**3.1528(3.1050)**
Problem	$\mathcal{LSDL}$(IMP)			
n	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
10	0.804(0.924)	0.926(1.040)	0.804(0.924)	**0.0000(0.0000)**
20	6.671(5.447)	8.641(7.308)	6.671(5.447)	**0.0244(0.0200)**
30	24.86(24.25)	35.26(35.62)	24.86(24.25)	**0.1512(0.1600)**
40	77.19(80.63)	114.8(128.1)	77.19(80.63)	**0.6738(0.7600)**
50	196.8(199.0)	300.0(326.6)	196.8(199.0)	**2.2416(2.4450)**

Table 4.4: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problems

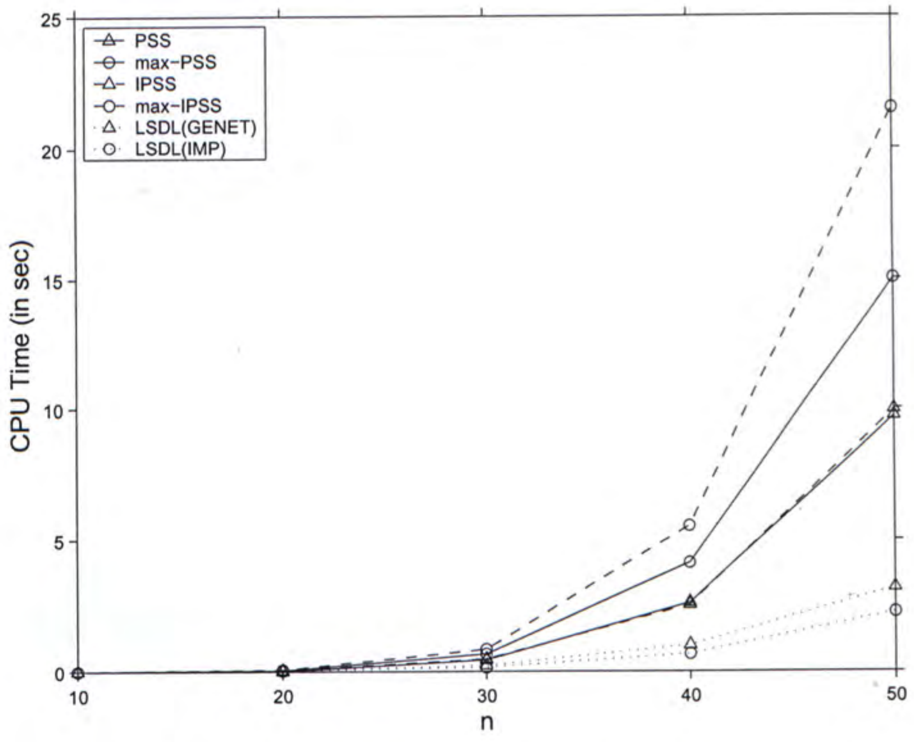
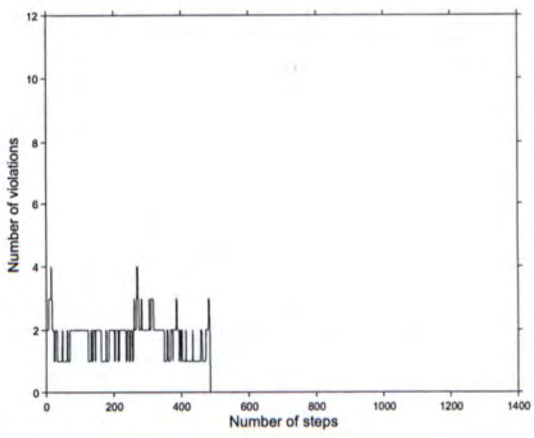


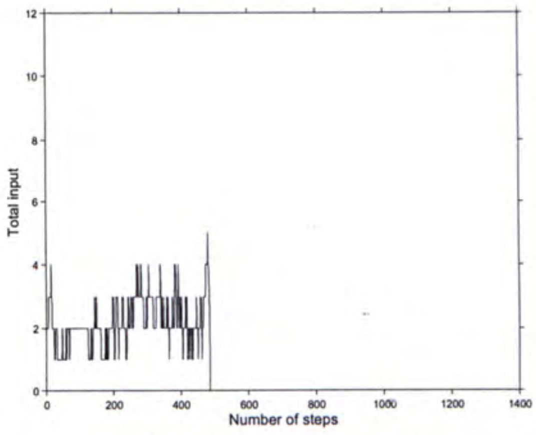
Figure 4.20: The mean time results on increasing permutation problems

The increasing permutation problem is a special case of permutation generation problem: it has only one solution. In Table 4.4, $\mathcal{LSDL}$(GENET) needs around 390,600 iterations and 195,500 learning to solve the increasing permutation problem with $n = 50$. It means that for every two iterations, one learning is required to escape from local minimum. Besides, the increasing permutation problem has another property that makes it hard for local search solvers. There exist a large number of assignments in which the number of violations equals to only 1 even though the assignment is "very wrong". We use an example to illustrate this. Assume that $n = 5$ and the variable assignment is $x_1 = 2, x_2 = 3, x_3 = 4, x_4 = 5, x_5 = 1$. All variables take the wrong values. However, only one constraint ($x_4 < x_5$) is violated.

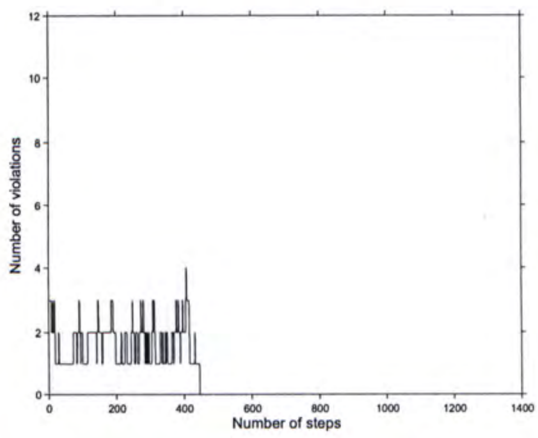
The timing results indicate that the performance of PSS and its variants are much worse than $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in this problem. We record the numbers of violations against total inputs or objective values of



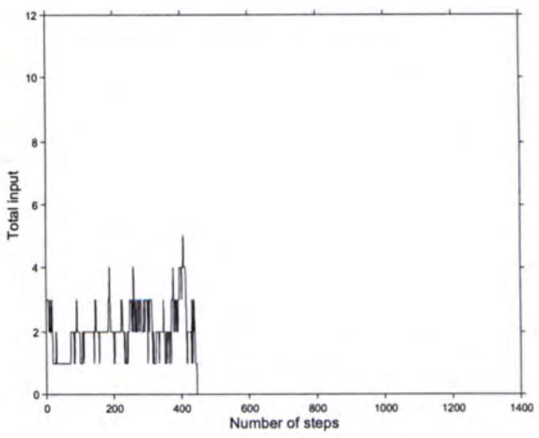
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

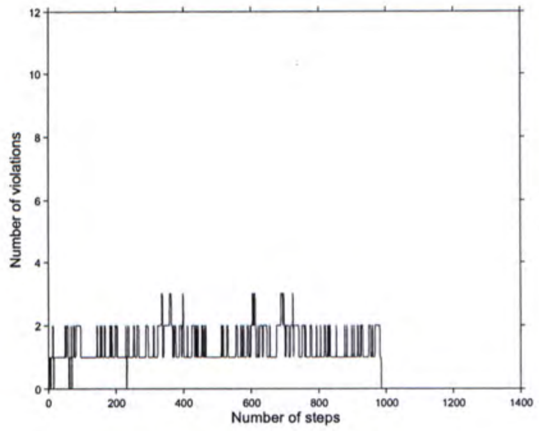


(c) max-PSS: Violation vs. Step

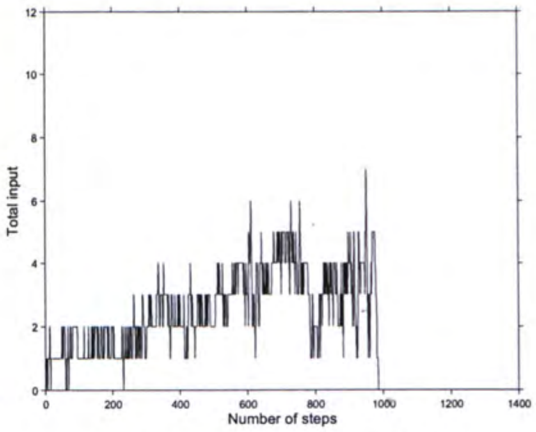


(d) max-PSS: Total input vs. Step

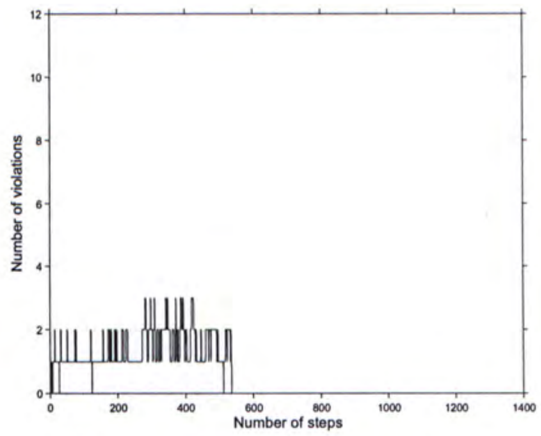
Figure 4.21: Numbers of violations and total inputs in each step of PSS and max-PSS on increasing permutation problem with $n = 10$ (average run-time case)



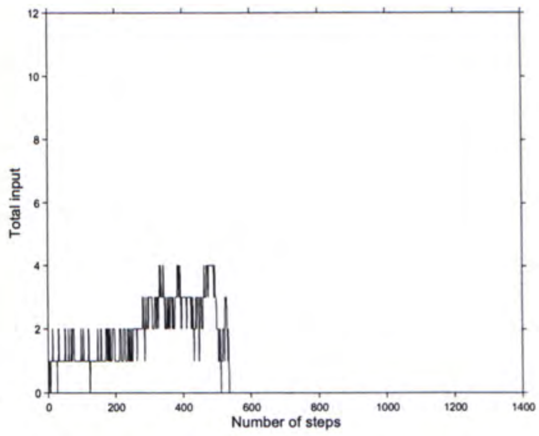
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

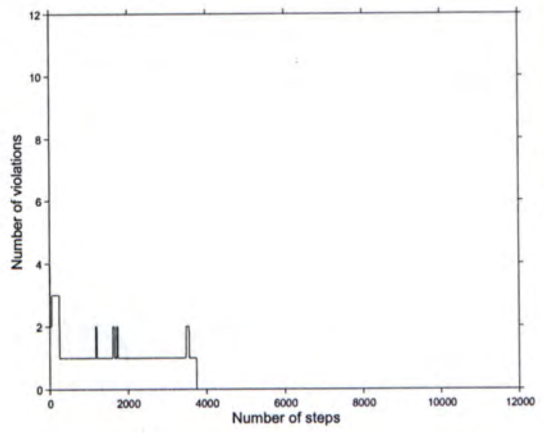


(c) max-IPSS: Violation vs. Step

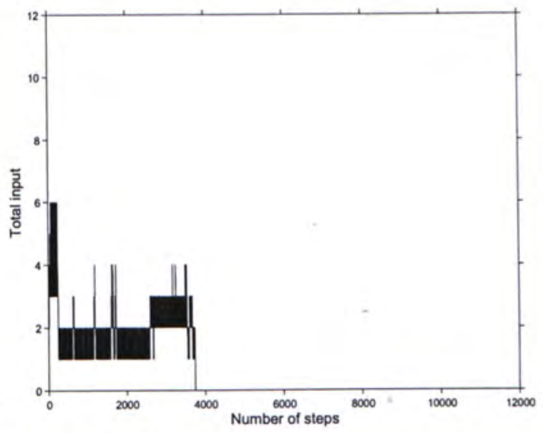


(d) max-IPSS: Total input vs. Step

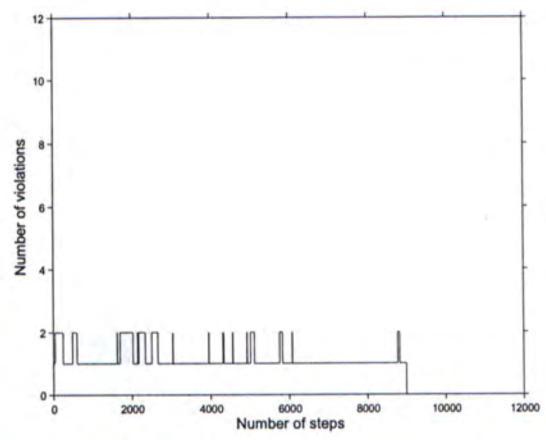
Figure 4.22: Numbers of violations and total inputs in each step of IPSS and max-IPSS on increasing permutation problem with $n = 10$ (average run-time case)



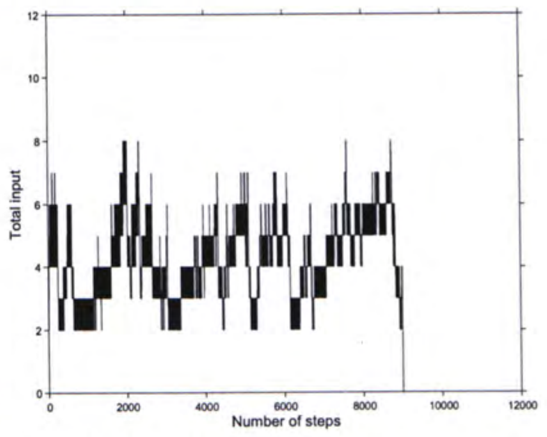
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

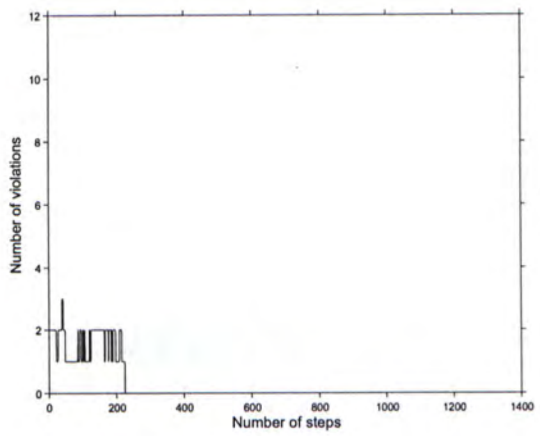


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

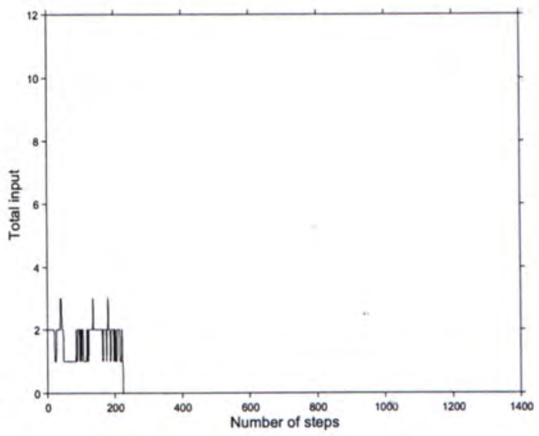


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

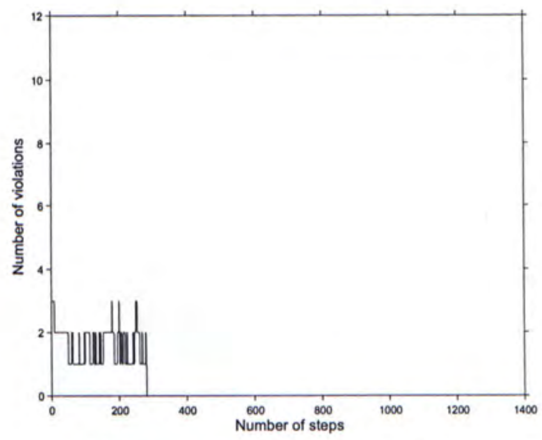
Figure 4.23: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 10$ (average run-time case)



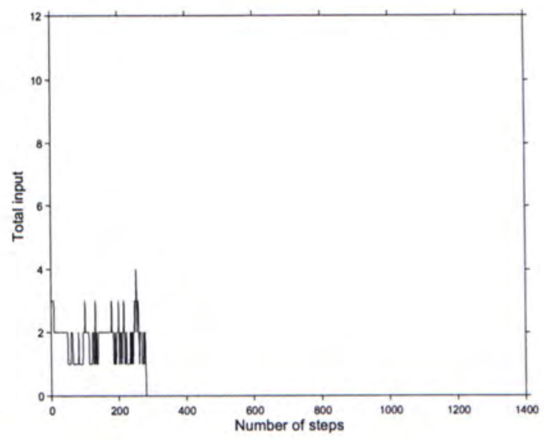
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

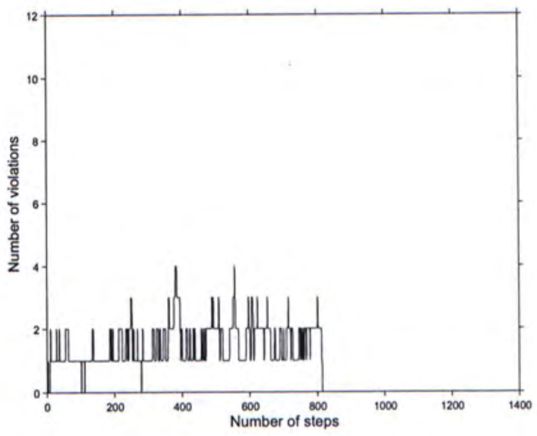


(c) max-PSS: Violation vs. Step

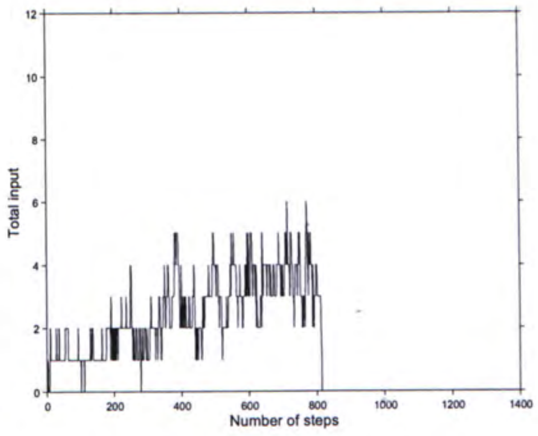


(d) max-PSS: Total input vs. Step

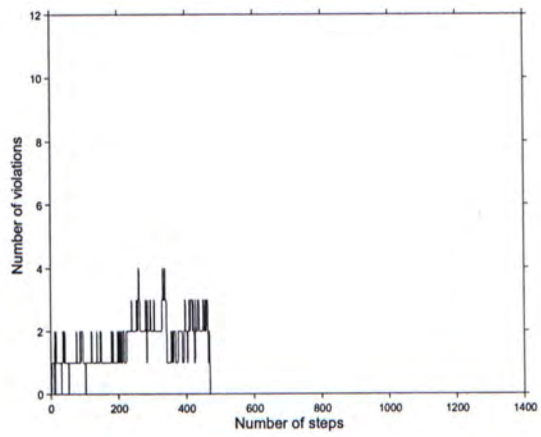
Figure 4.24: Numbers of violations and total inputs in each step of PSS and max-PSS on increasing permutation problem with $n = 10$ (short run-time case)



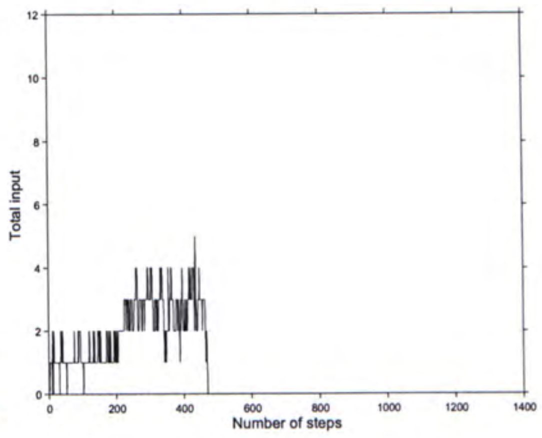
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

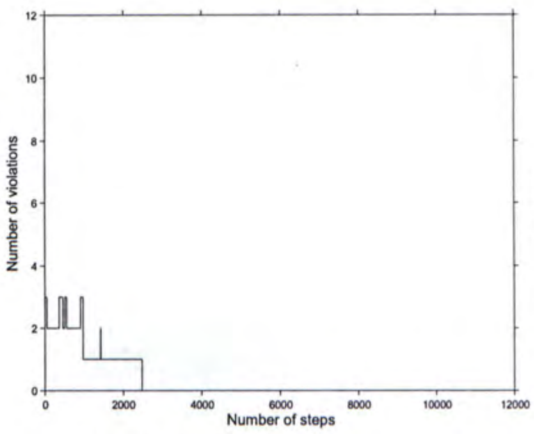


(c) max-IPSS: Violation vs. Step

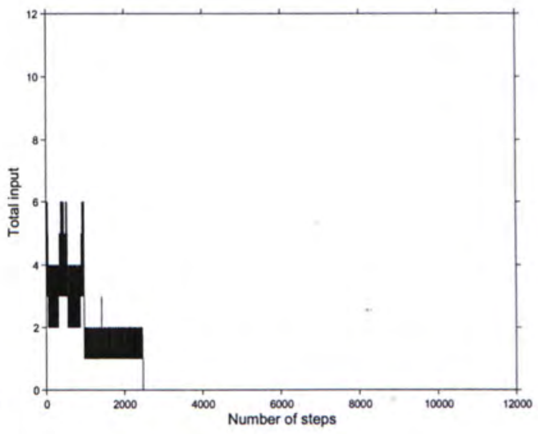


(d) max-IPSS: Total input vs. Step

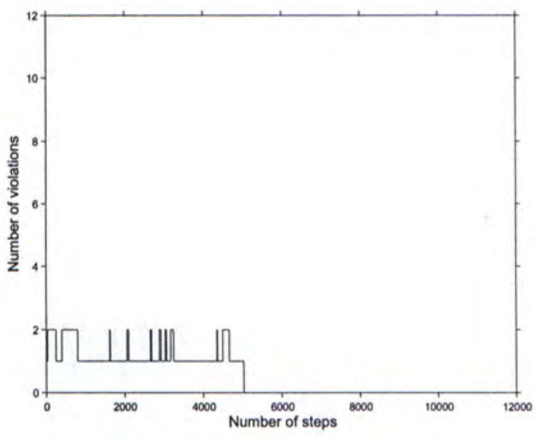
Figure 4.25: Numbers of violations and total inputs in each step of IPSS and max-IPSS on increasing permutation problem with $n = 10$ (short run-time case)



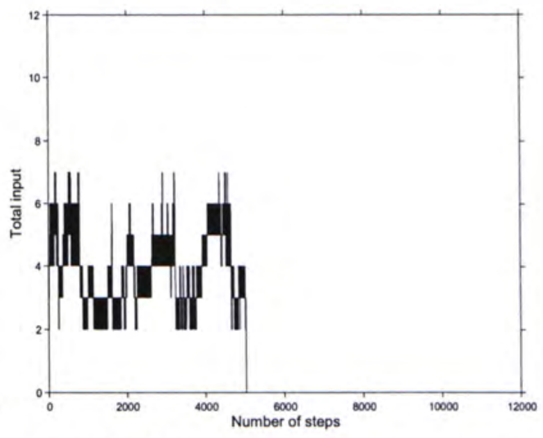
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

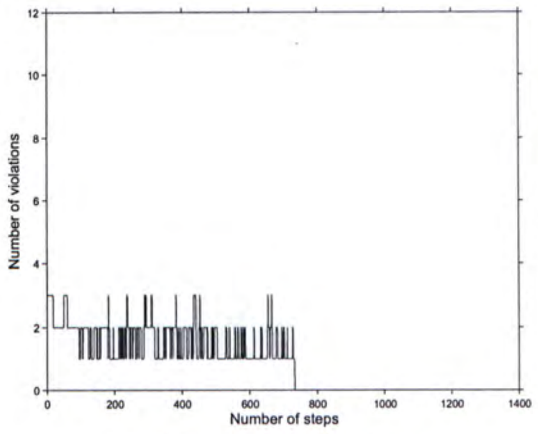


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

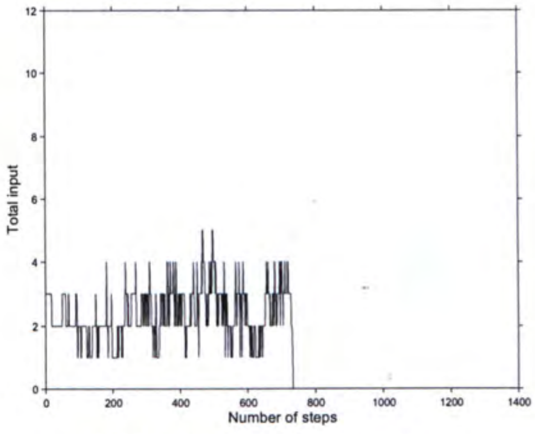


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

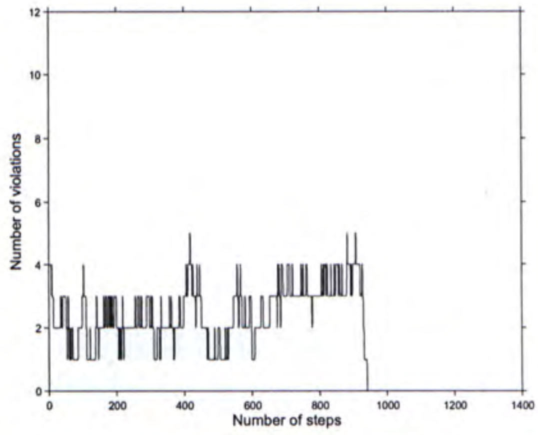
Figure 4.26: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 10$ (short run-time case)



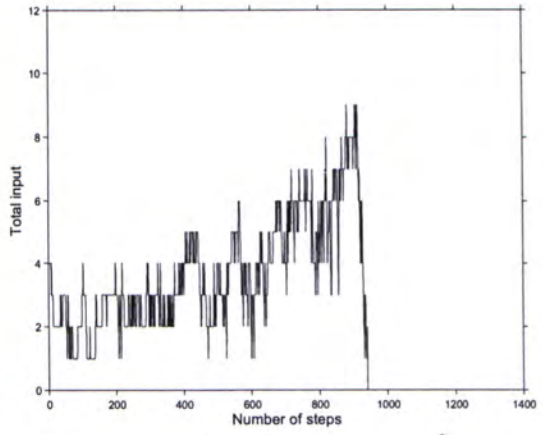
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

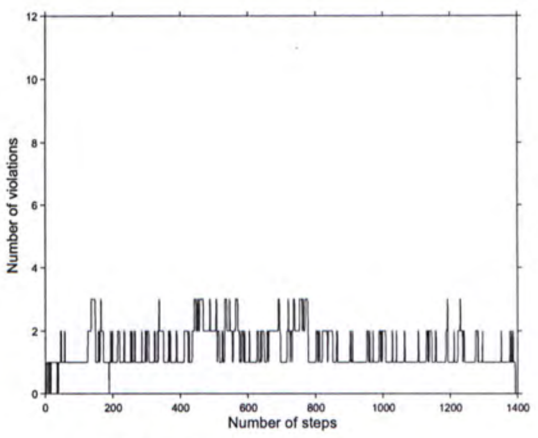


(c) max-PSS: Violation vs. Step

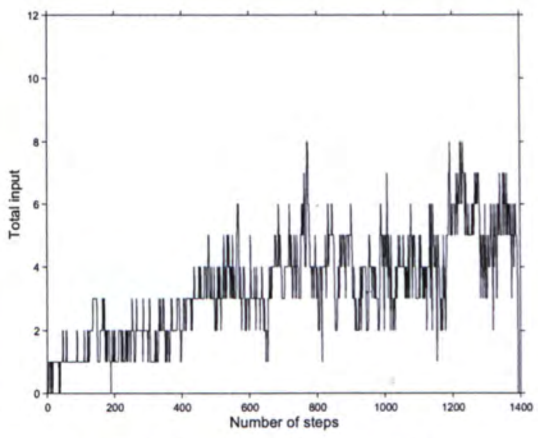


(d) max-PSS: Total input vs. Step

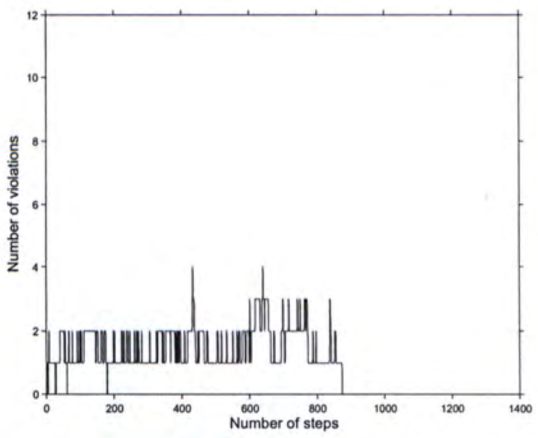
Figure 4.27: Numbers of violations and total inputs in each step of PSS and max-PSS on increasing permutation problem with $n = 10$ (long run-time case)



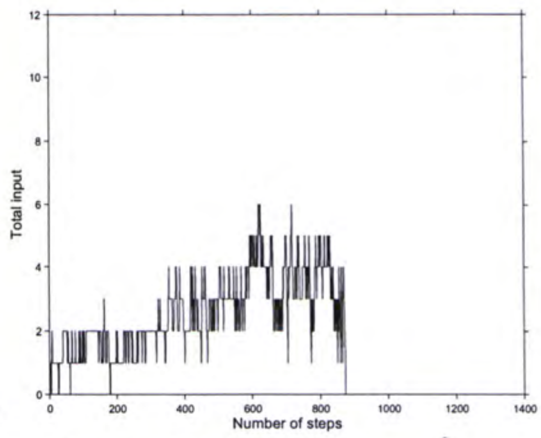
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

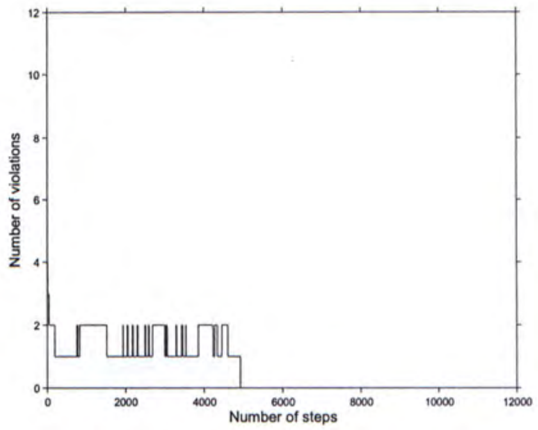


(c) max-IPSS: Violation vs. Step

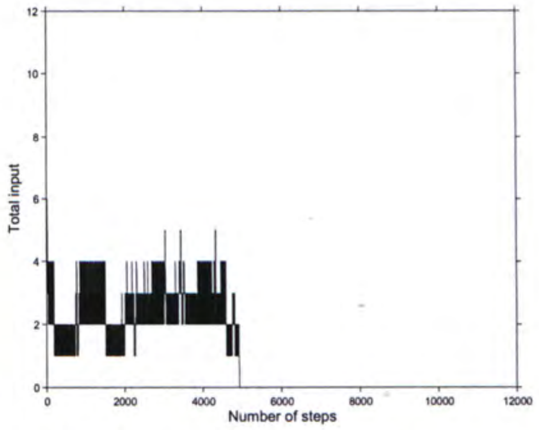


(d) max-IPSS: Total input vs. Step

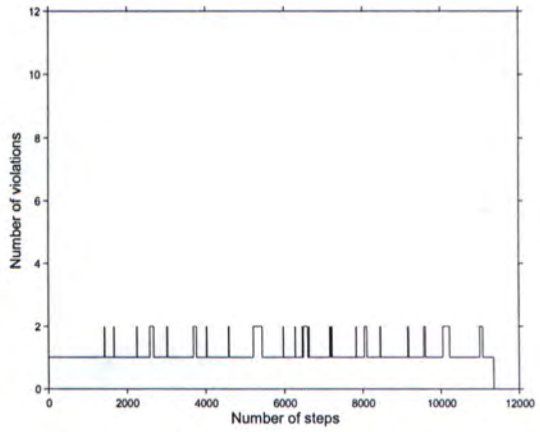
Figure 4.28: Numbers of violations and total inputs in each step of IPSS and max-IPSS on increasing permutation problem with $n = 10$ (long run-time case)



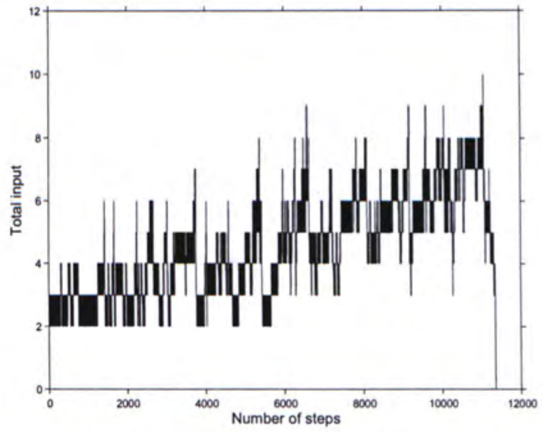
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

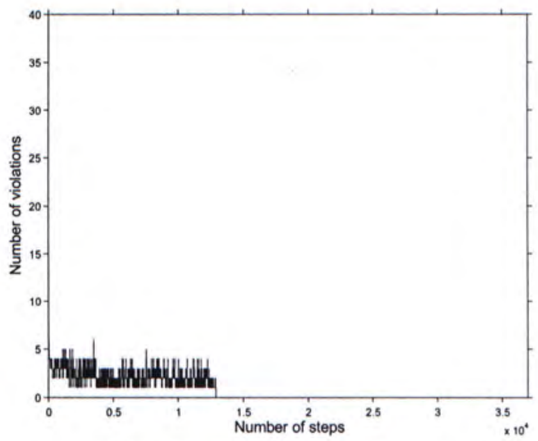


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

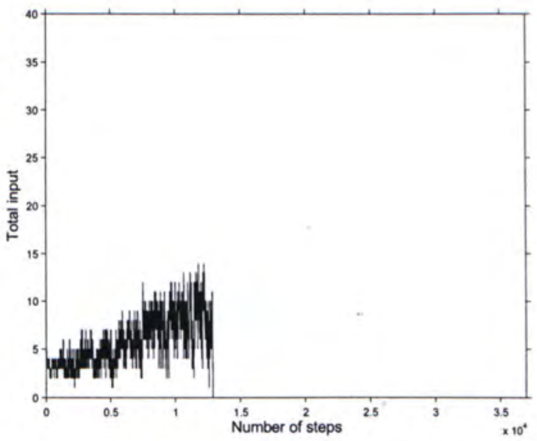


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

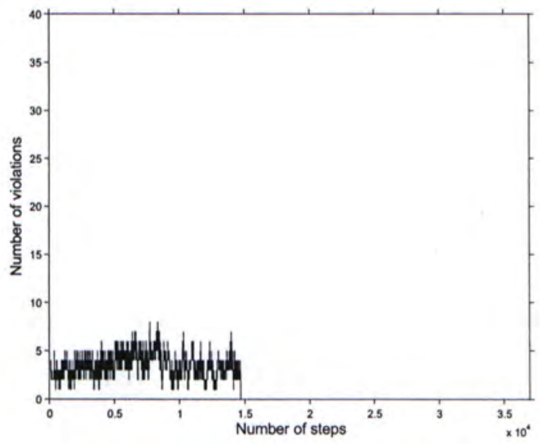
Figure 4.29: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 10$ (long run-time case)



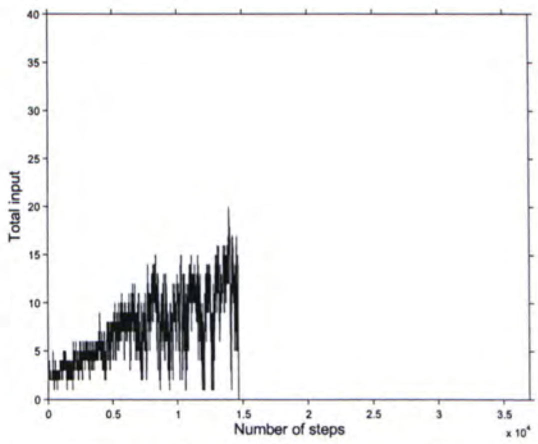
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

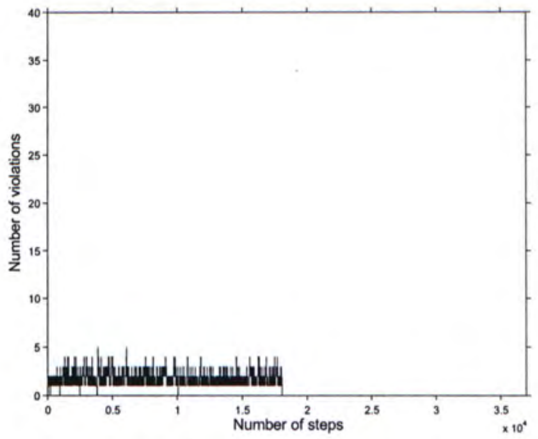


(c) max-PSS: Violation vs. Step

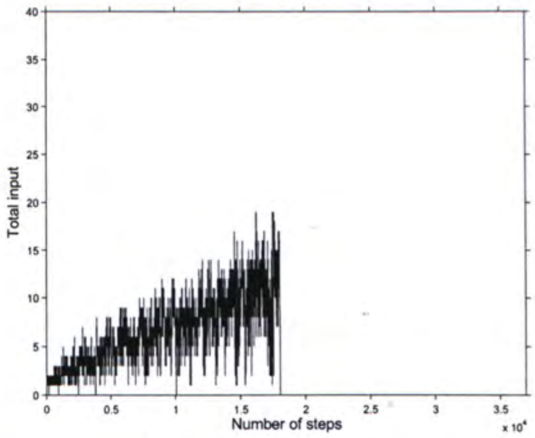


(d) max-PSS: Total input vs. Step

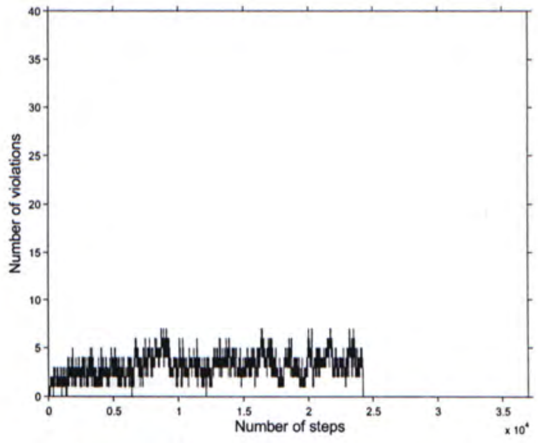
Figure 4.30: Numbers of violations and total inputs in each step of PSS and max-PSS on increasing permutation problem with $n = 20$ (average run-time case)



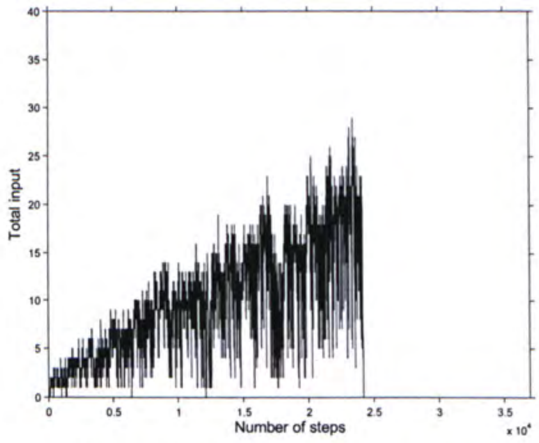
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

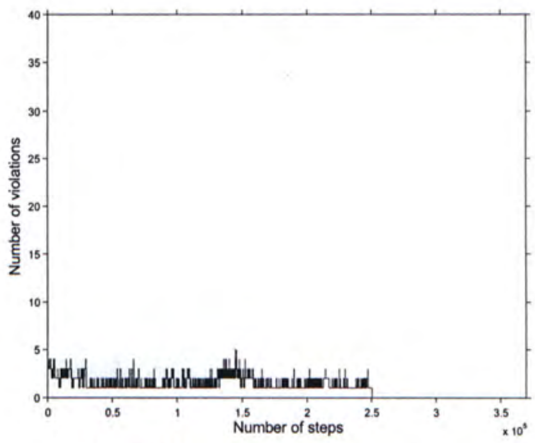


(c) max-IPSS: Violation vs. Step

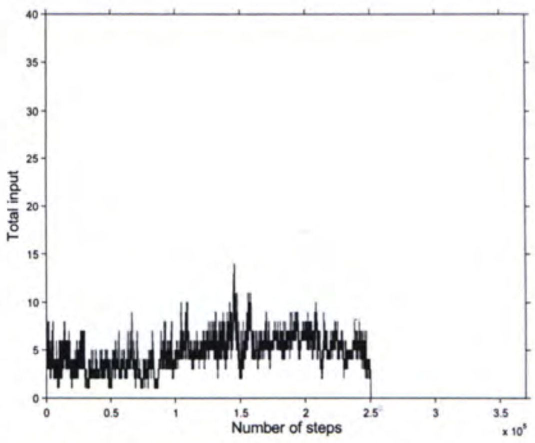


(d) max-IPSS: Total input vs. Step

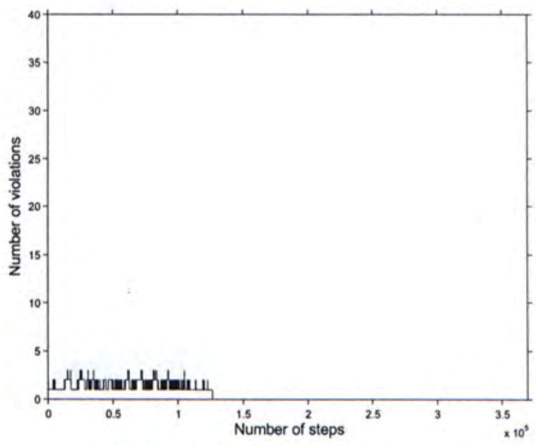
Figure 4.31: Numbers of violations and total inputs in each step of IPSS and max-IPSS on increasing permutation problem with $n = 20$ (average run-time case)



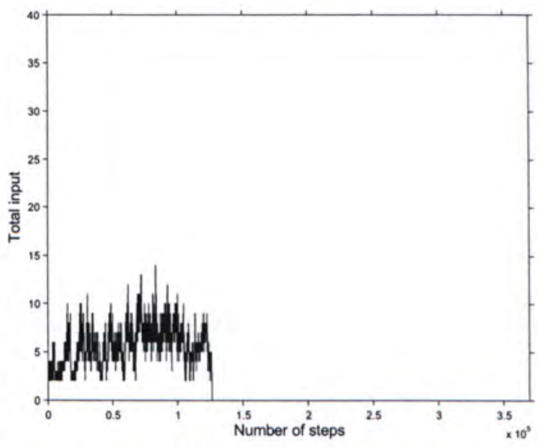
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

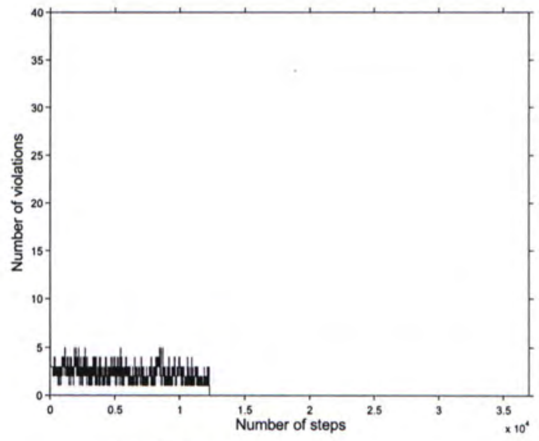


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

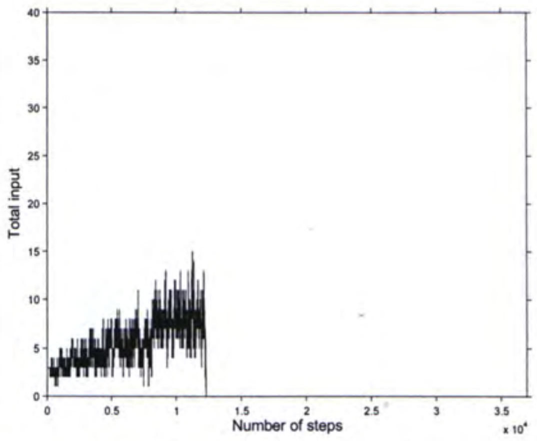


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

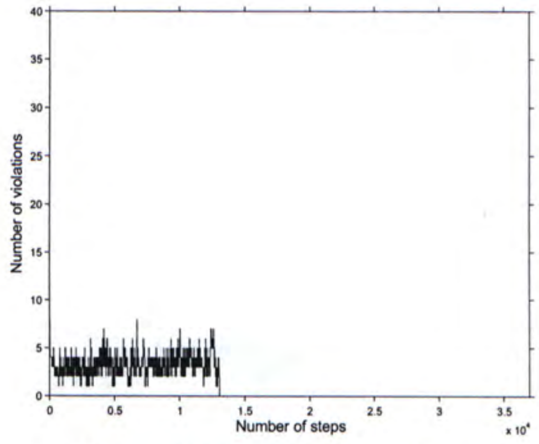
Figure 4.32: Numbers of violations and objective values in each step $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 20$ (average run-time case)



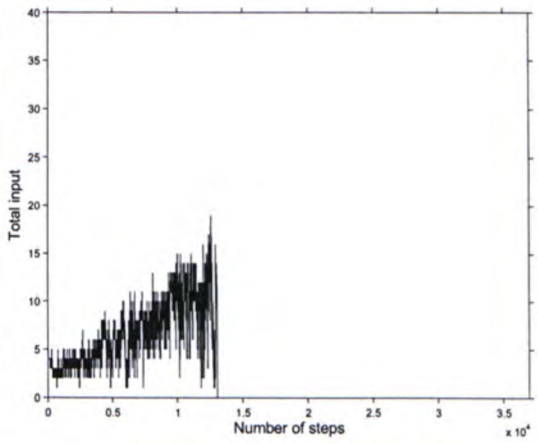
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

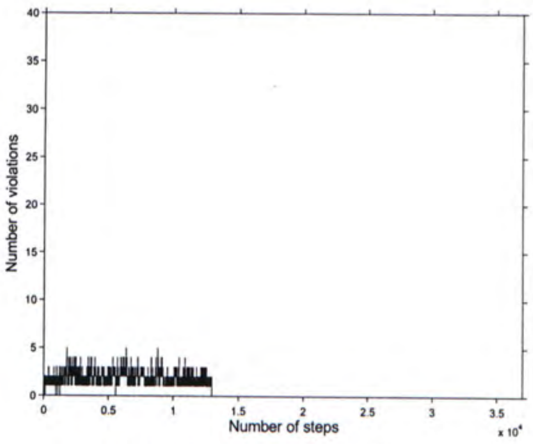


(c) max-PSS: Violation vs. Step

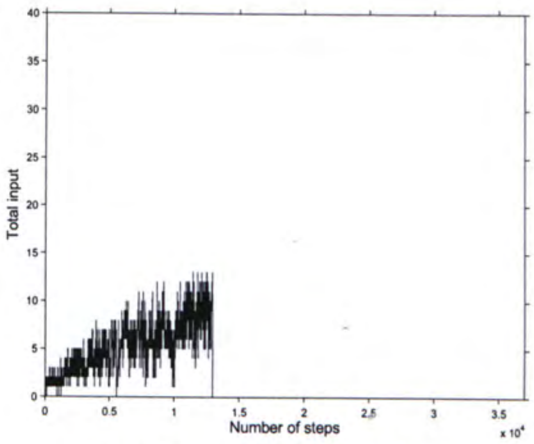


(d) max-PSS: Total input vs. Step

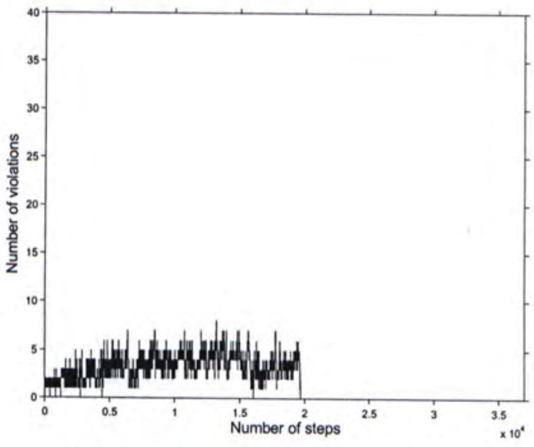
Figure 4.33: Numbers of violations and total inputs in each step of PSS and max-PSS on increasing permutation problem with $n = 20$ (short run-time case)



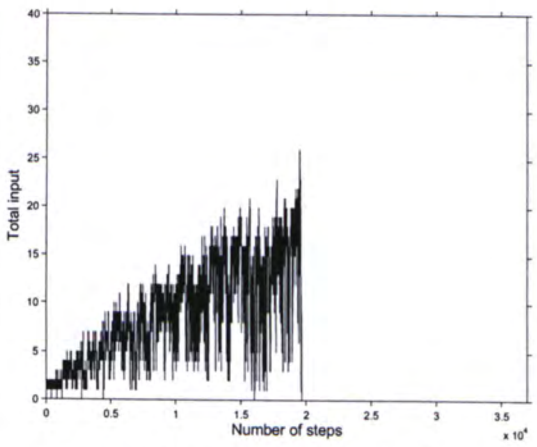
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

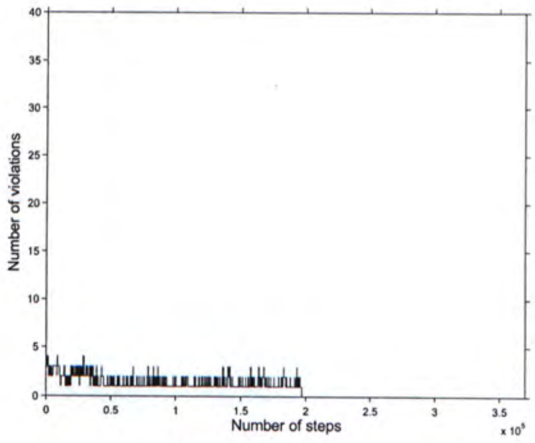


(c) max-IPSS: Violation vs. Step

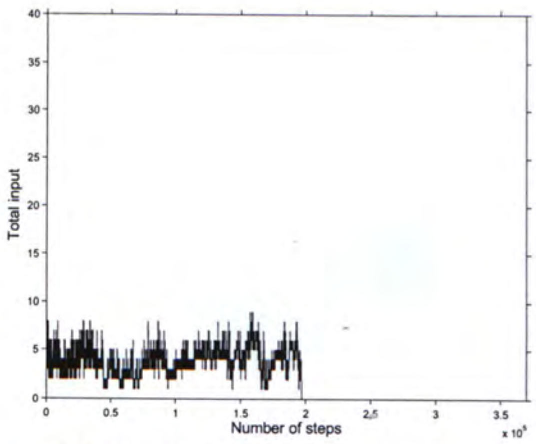


(d) max-IPSS: Total input vs. Step

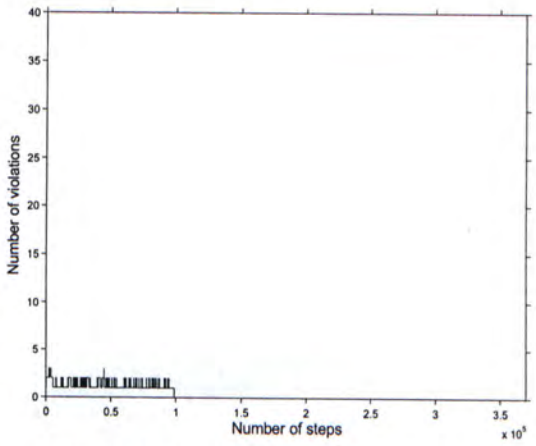
Figure 4.34: Numbers of violations and total inputs in each step of IPSS and max-IPSS on increasing permutation problem with $n = 20$ (short run-time case)



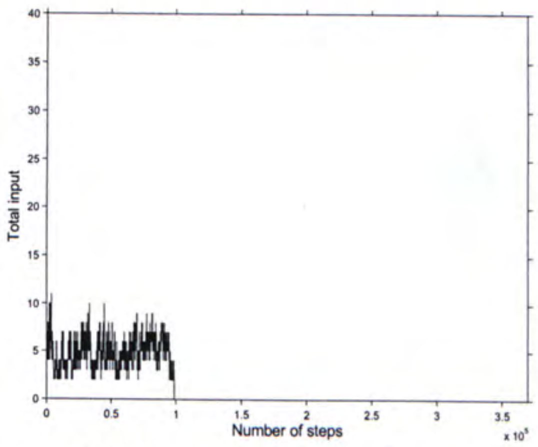
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

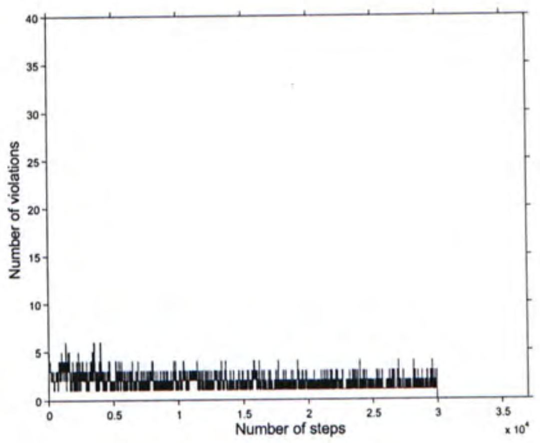


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

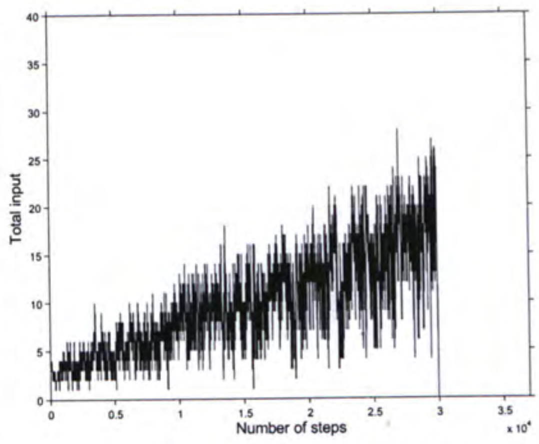


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

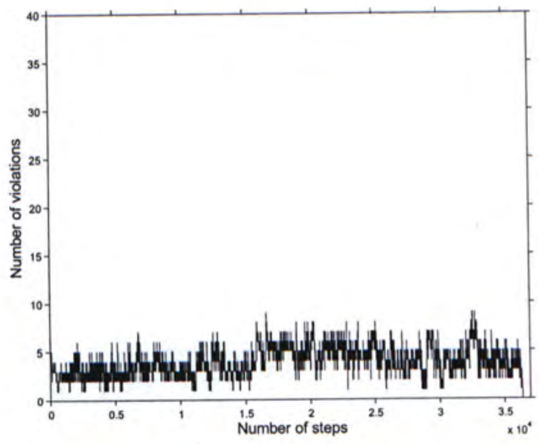
Figure 4.35: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 20$ (short run-time case)



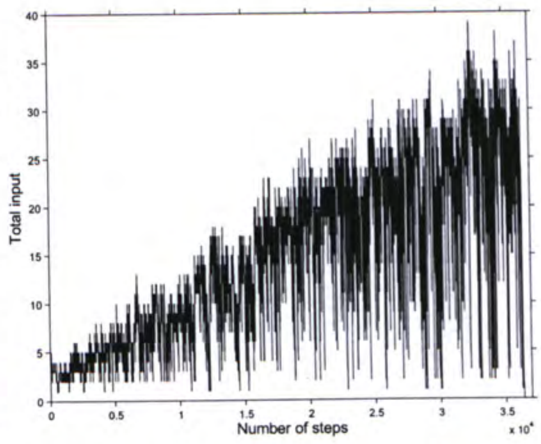
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

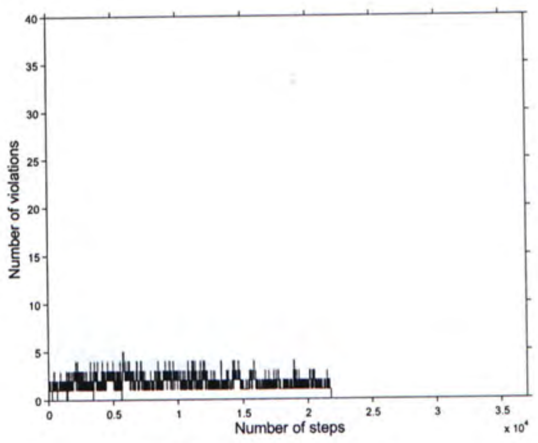


(c) max-PSS: Violation vs. Step

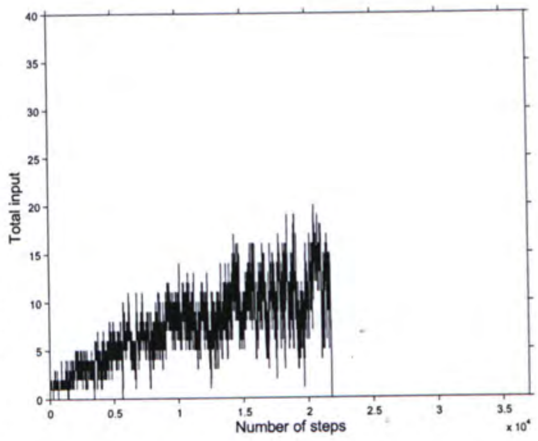


(d) max-PSS: Total input vs. Step

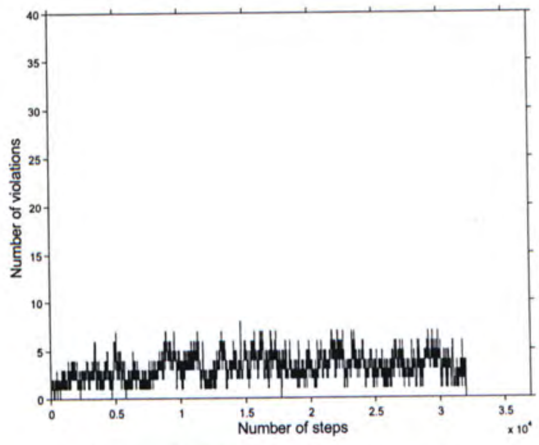
Figure 4.36: Numbers of violations and total inputs in each step of PSS and max-PSS on increasing permutation problem with $n = 20$ (long run-time case)



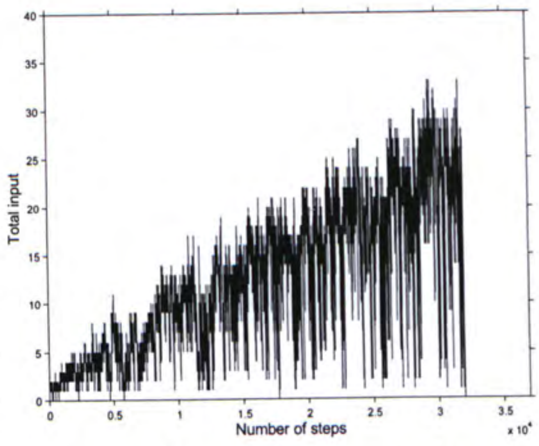
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

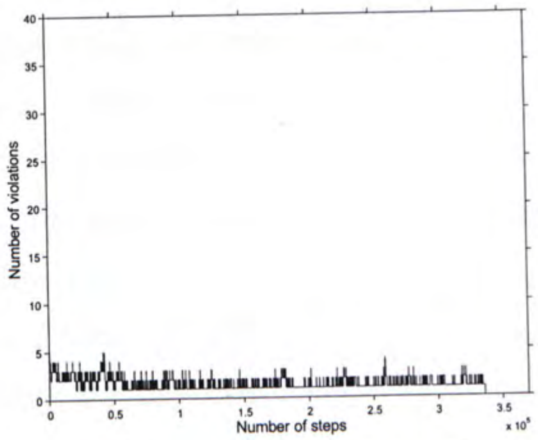


(c) max-IPSS: Violation vs. Step

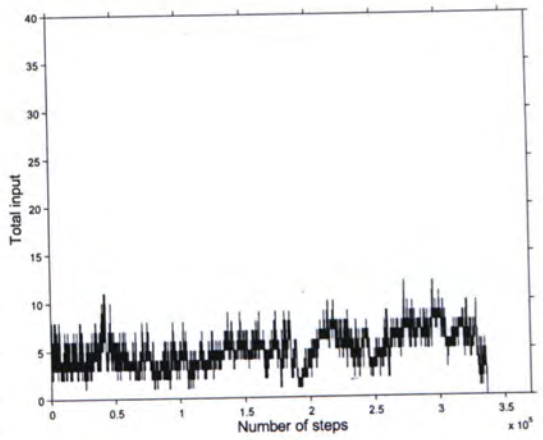


(d) max-IPSS: Total input vs. Step

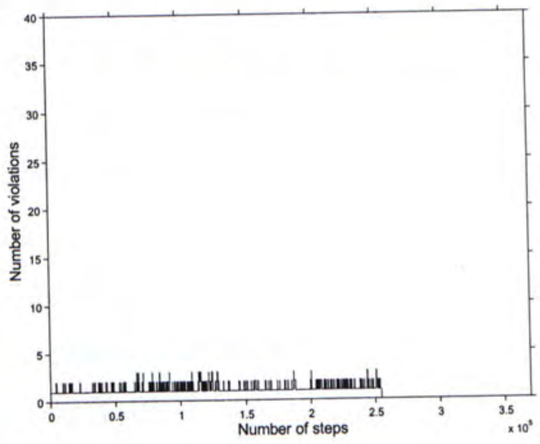
Figure 4.37: Numbers of violations and total inputs in each step of IPSS and max-IPSS on increasing permutation problem with $n = 20$ (long run-time case)



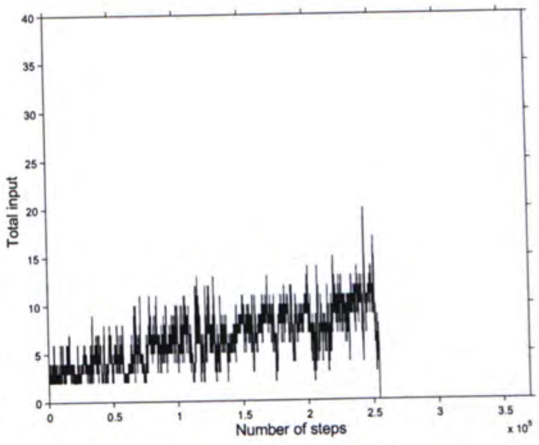
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step



(c) $\mathcal{LSDL}$(IMP): Violation vs. Step



(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

Figure 4.38: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 20$ (long run-time case)

PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 10$ in Figures 4.21 - 4.29. The numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on increasing permutation problem with $n = 20$ are shown in Figures 4.30 - 4.38. We first recall that there exists only one solution in an increasing permutation problem. Figures 4.23(a), 4.26(a) and 4.29(a) show the number of violations in each step of $\mathcal{LSDL}$(GENET) on increasing permutation problem with $n = 10$ in average run-time case, short run-time case and long run-time case respectively. Figures 4.32(a), 4.35(a) and 4.38(a) show the number of violations in each step of $\mathcal{LSDL}$(GENET) on increasing permutation problem with $n = 20$ in average run-time case, short run-time case and long run-time case respectively. From the figures, there exists a large number of valley-like plateaus in the search space. $\mathcal{LSDL}$(GENET) carefully performs learning and modifies the landscape of the search space when it traverses these valleys. This prudent approach helps it reach a solution quickly. Although $\mathcal{LSDL}$(GENET) uses more steps to solve the increasing permutation problem, it repairs fewer clusters to find a solution. On the contrary, PSS uses a lot more steps to traverse the plateaus, and it pays less attention to the landscape when it rushes through the plateaus (Figures 4.21(a), 4.24(a), 4.27(a), 4.30(a), 4.33(a) and 4.36(a)). $\mathcal{LSDL}$(IMP) performs more learning than $\mathcal{LSDL}$(GENET). This approach quickly modifies the landscape of the search space and increases the contrast between the landscape of the solutions and that of the non-solutions. Therefore, the timing results of $\mathcal{LSDL}$(IMP) outperforms $\mathcal{LSDL}$(GENET), PSS and its variants.

Figures 4.22(a), 4.25(a) and 4.28(a) show the number of violations in each step of IPSS on increasing permutation problem with $n = 10$ in average run-time case, short run-time case and long run-time case respectively. Figures 4.31(a), 4.34(a) and 4.37(a) show the number of violations in each step of

IPSS on increasing permutation problem with $n = 20$ in average run-time case, short run-time case and long run-time case respectively. For IPSS, the situation is worse as the first several hundreds steps are basically wasted: the partial solutions found are not usually a subset of the final solution. Assume that $n = 5$ and the current partial solution is $x_1 = 2, x_2 = 3, x_3 = 4$. This partial solution can be extended by assigning 5 to variable x_4. However, all existing variables take the wrong values with respect to the complete solution. The timing results of IPSS are hence worse than that of PSS because IPSS spends time on doing those futile steps. The cluster selection heuristics actually makes the situation worse, as the search is directed to rough areas.

4.2.2 Random Permutation Generation Problems

The timing results of PSS and its variants on random permutation generation problems are showed in Table 4.5. Table 4.6 shows the timing results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on the same set of problems. Figure 4.39 shows the mean time results of all implementations on random permutation generation problems. Problems in this set are easy for $\mathcal{LSDL}$ implementations and PSS implementations. All problem instances are solved almost immediately. PSS and its variants are slightly more efficient than $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) for this set of problems. The difference is more significant when the problem size grows larger and the number of solutions increases.

Figures 4.40 - 4.48 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random permutation generation problem with $n = 50$. Figures 4.49 - 4.57 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random permutation generation problem with $n = 100$. In the figures about $\mathcal{LSDL}$(GENET), we can see that $\mathcal{LSDL}$(GENET) carefully performs learning

Problem	PSS		IPSS	
n	Steps	**CPU time**	Steps	**CPU time**
50	87.8(83.0)	**0.0029(0.0000)**	85.3(83.0)	**0.0025(0.0000)**
60	127.5(111.0)	**0.0053(0.0100)**	136.5(119.0)	**0.0054(0.0100)**
70	151.7(141.0)	**0.0079(0.0100)**	156.2(143.5)	**0.0071(0.0100)**
80	134.4(126.5)	**0.0082(0.0100)**	132.7(126.0)	**0.0054(0.0100)**
90	152.6(146.5)	**0.0097(0.0100)**	173.6(162.0)	**0.0102(0.0100)**
100	144.4(142.0)	**0.0098(0.0100)**	153.0(150.0)	**0.0094(0.0100)**
	max-PSS		max-IPSS	
50	103.5(105.0)	**0.0030(0.0000)**	102.3(95.5)	**0.0027(0.0000)**
60	137.3(122.0)	**0.0054(0.0100)**	129.6(110.0)	**0.0041(0.0000)**
70	160.2(146.5)	**0.0088(0.0100)**	155.8(147.0)	**0.0064(0.0100)**
80	140.8(137.5)	**0.0073(0.0100)**	129.4(127.5)	**0.0055(0.0100)**
90	160.1(155.5)	**0.0099(0.0100)**	164.9(157.0)	**0.0090(0.0100)**
100	155.4(153.0)	**0.0101(0.0100)**	155.4(151.0)	**0.0098(0.0100)**

Table 4.5: PSS and its variants on random permutation generation problems

Problem	$\mathcal{LSDL}$(GENET)			
n	Iteration	Repairs	Learns	**CPU time**
50	35.6(23.0)	62.1(55.0)	16.1(10.0)	**0.0040(0.0000)**
60	76.1(67.5)	96.7(88.5)	36.2(32.0)	**0.0060(0.0100)**
70	122.8(63.5)	142.1(114.0)	58.6(29.0)	**0.0088(0.0100)**
80	132.7(59.0)	146.7(107.0)	64.0(27.0)	**0.0114(0.0100)**
90	107.3(57.5)	141.7(117.5)	51.4(27.0)	**0.0132(0.0100)**
100	64.4(40.0)	123.1(109.5)	29.7(18.0)	**0.0134(0.0100)**
Problem	$\mathcal{LSDL}$(IMP)			
n	Iteration	Repairs	Learns	**CPU time**
50	21.5(16.0)	50.1(48.0)	21.5(16.0)	**0.0030(0.0000)**
60	35.9(26.0)	77.3(69.0)	35.9(26.0)	**0.0040(0.0000)**
70	62.2(52.5)	155.1(147.0)	62.2(52.5)	**0.0092(0.0100)**
80	68.4(46.5)	133.9(106.5)	68.4(46.5)	**0.0092(0.0100)**
90	49.1(31.5)	130.5(114.0)	49.1(31.5)	**0.0109(0.0100)**
100	32.8(23.5)	114.7(108.0)	32.8(23.5)	**0.0142(0.0100)**

Table 4.6: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random permutation generation problems

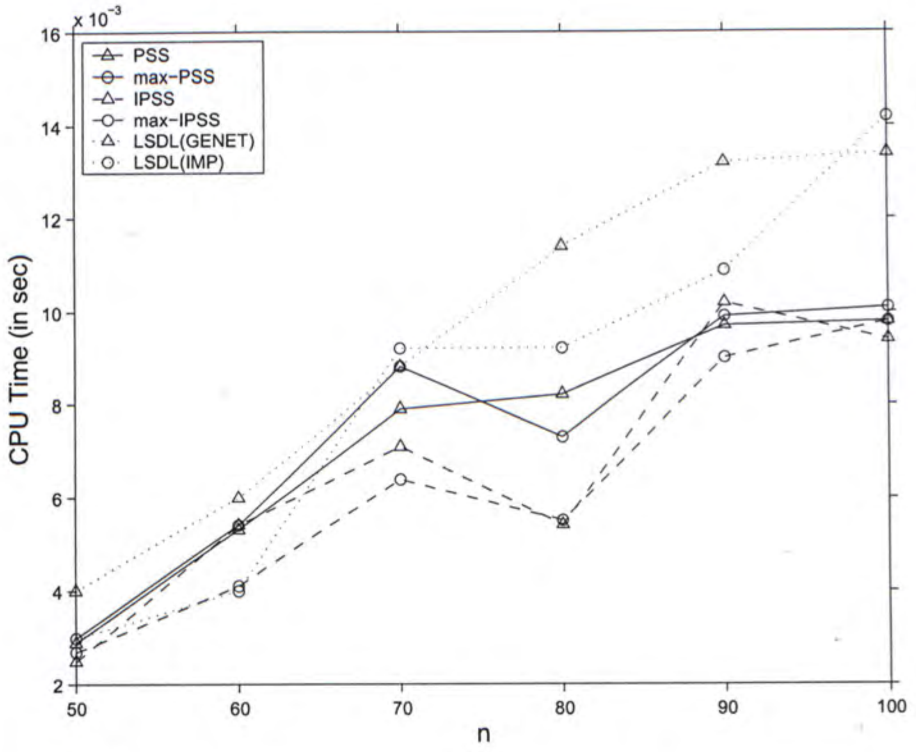
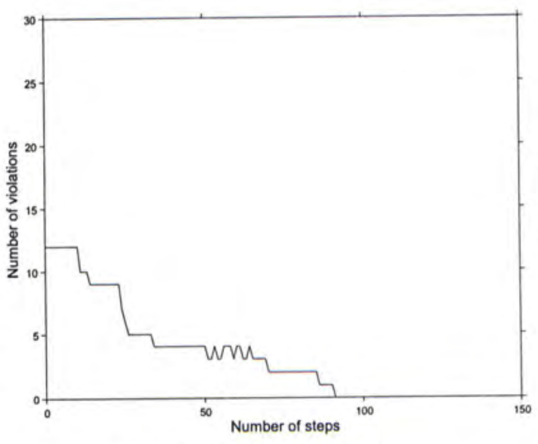
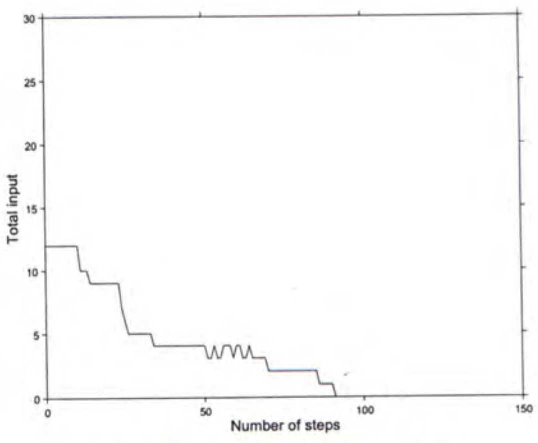


Figure 4.39: The mean time results on random permutation generation problems

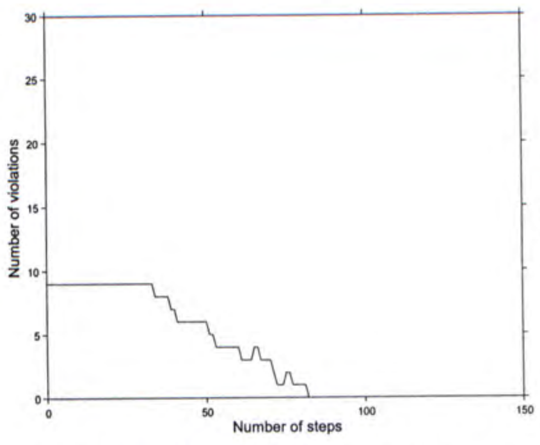
and modifies the landscape of the search space when it traverses the plateaus. This time the prudent approach reduces the search speed as there exist many solutions in the search space. The progressive approach used in PSS quickly traverses the plateaus and reaches the solution. This set of experiments illustrates the advantage of progressive approach in some benchmarking problems. The partial solutions found by IPSS can be extended easily. This further confirms that there are many solutions in this set of problem instances.



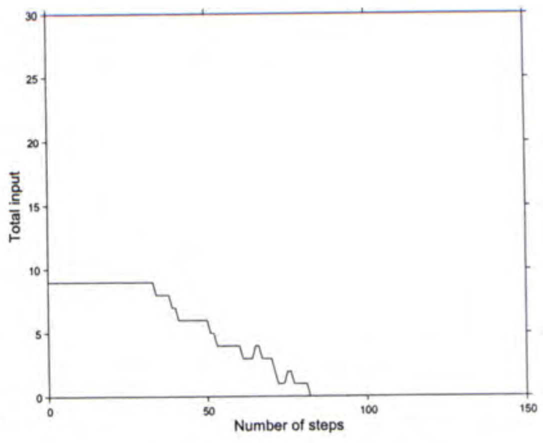
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

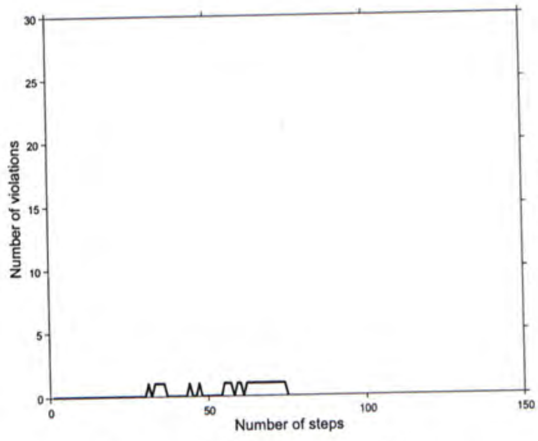


(c) max-PSS: Violation vs. Step

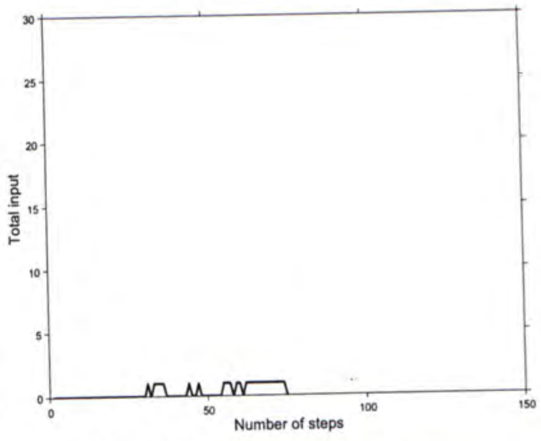


(d) max-PSS: Total input vs. Step

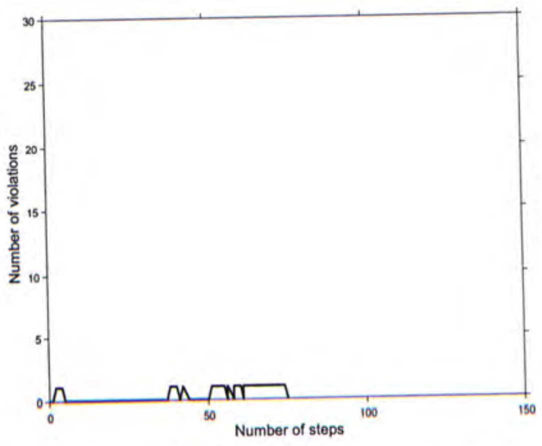
Figure 4.40: Numbers of violations and total inputs in each step of PSS and max-PSS on permutation generation problem with $n = 50$ (average run-time case)



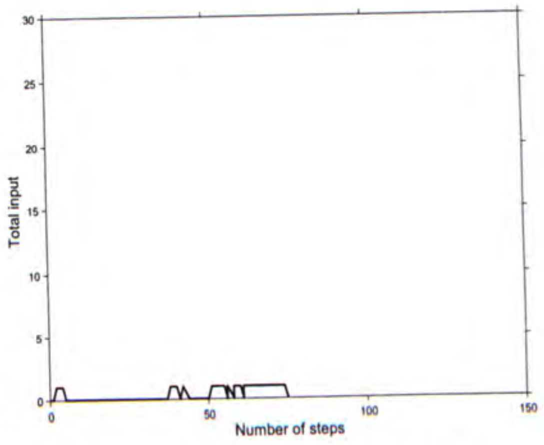
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

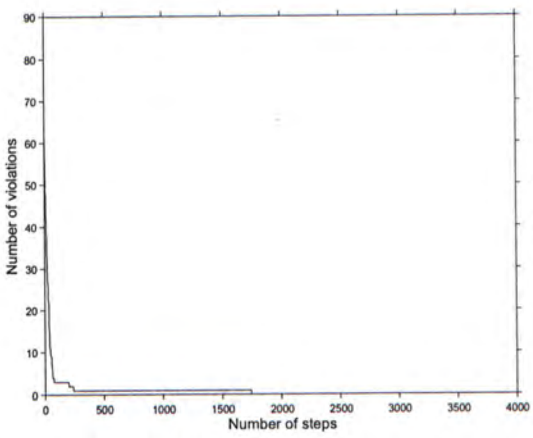


(c) max-IPSS: Violation vs. Step

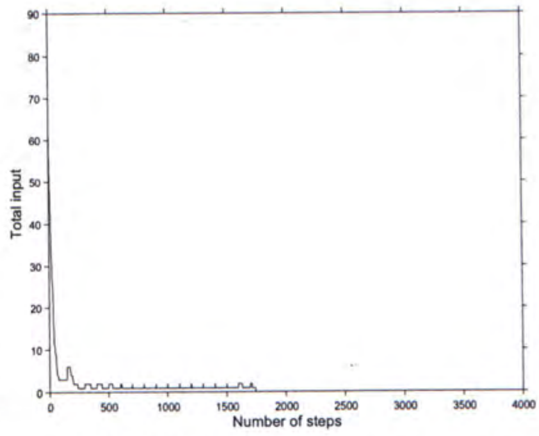


(d) max-IPSS: Total input vs. Step

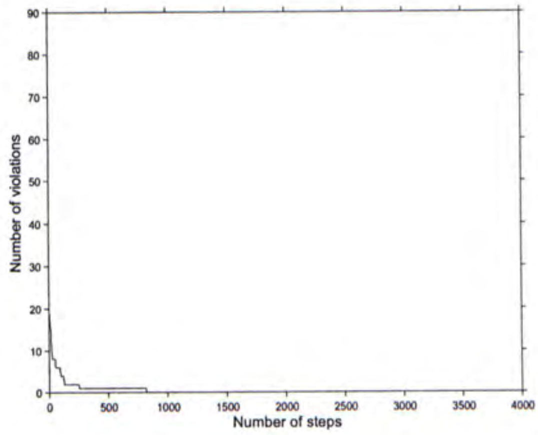
Figure 4.41: Numbers of violations and total inputs in each step of IPSS and max-IPSS on permutation generation problem with $n = 50$ (average run-time case)



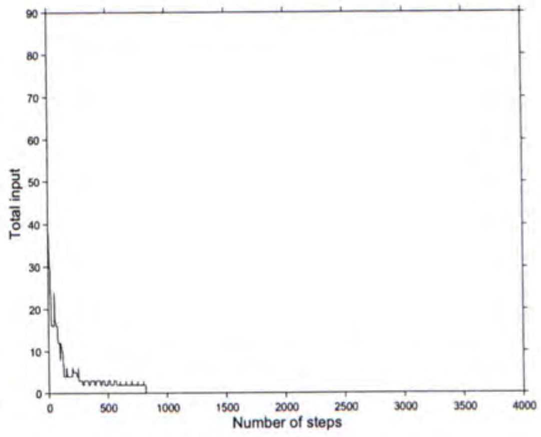
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

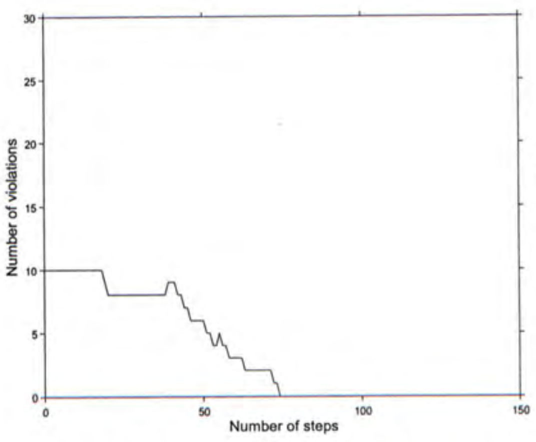


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

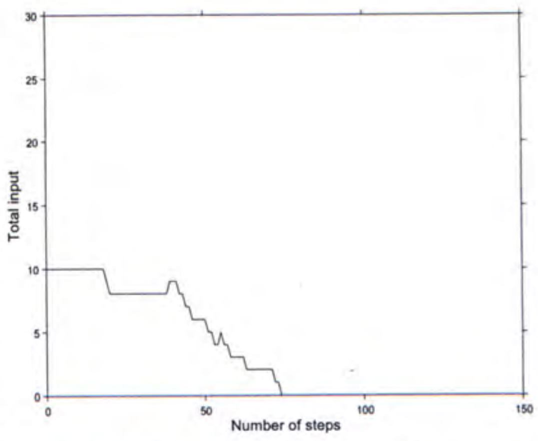


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

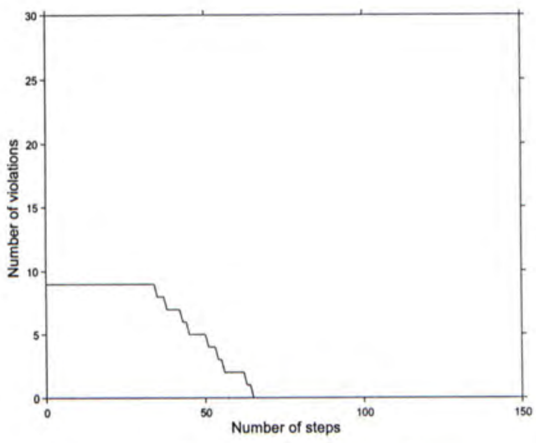
Figure 4.42: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on permutation generation problem with $n = 50$ (average run-time case)



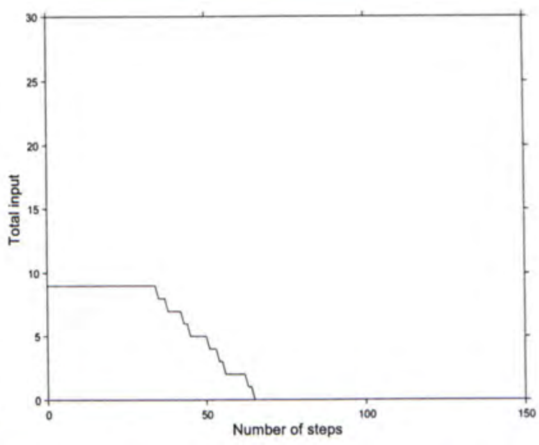
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

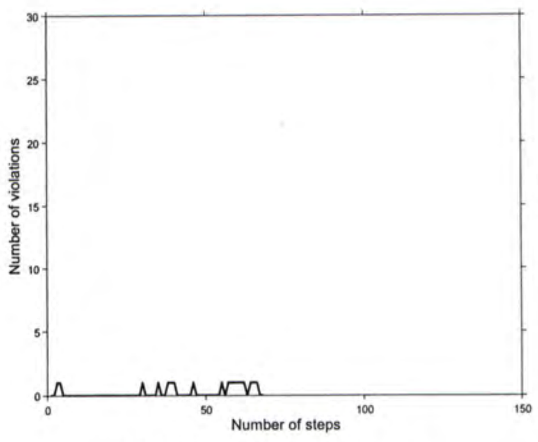


(c) max-PSS: Violation vs. Step

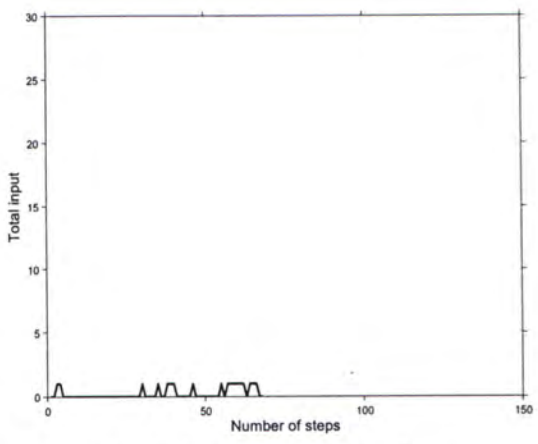


(d) max-PSS: Total input vs. Step

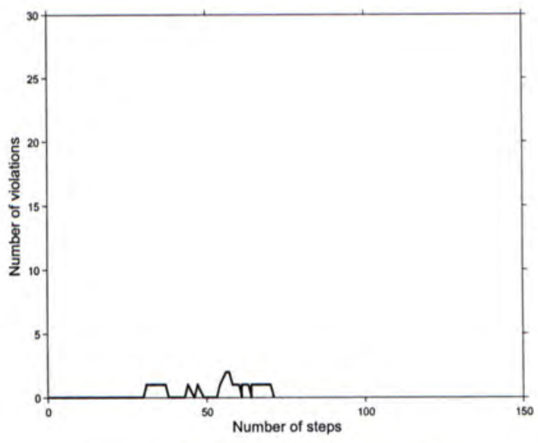
Figure 4.43: Numbers of violations and total inputs in each step of PSS and max-PSS on permutation generation problem with $n = 50$ (short run-time case)



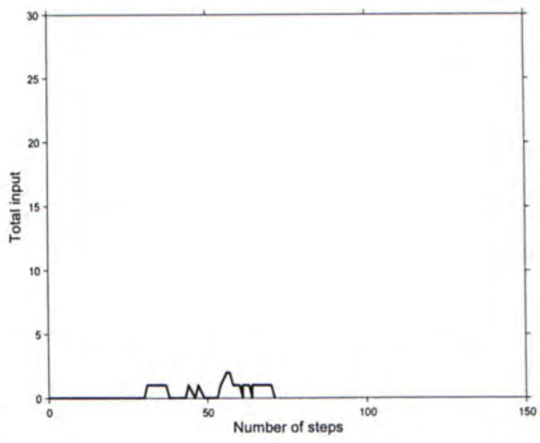
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

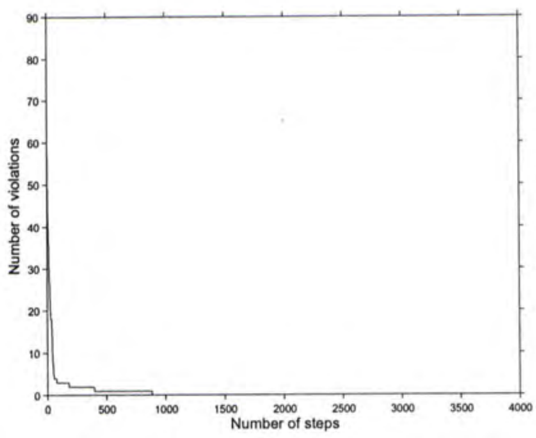


(c) max-IPSS: Violation vs. Step

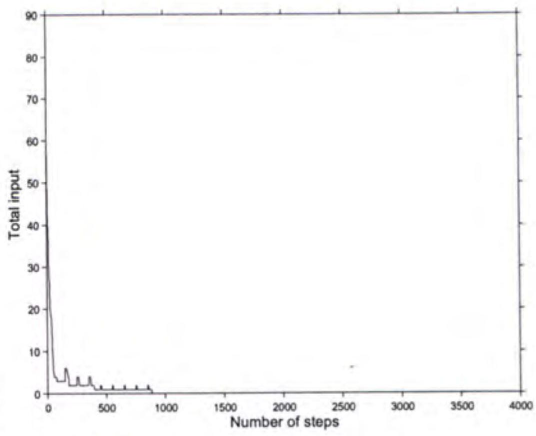


(d) max-IPSS: Total input vs. Step

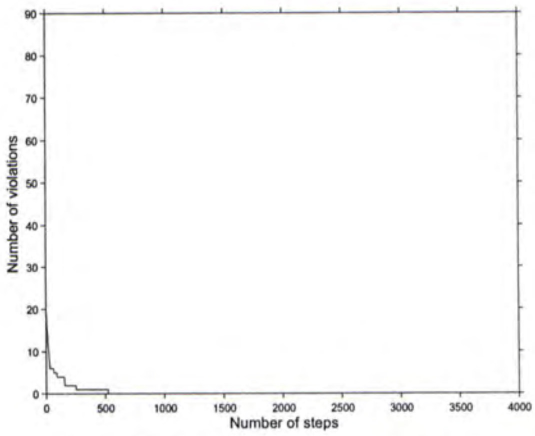
Figure 4.44: Numbers of violations and total inputs in each step of IPSS and max-IPSS on permutation generation problem with $n = 50$ (short run-time case)



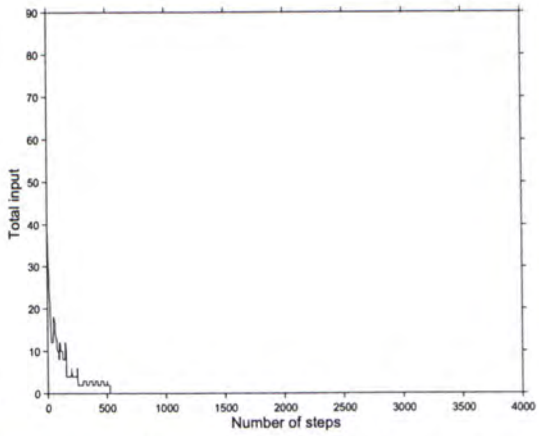
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

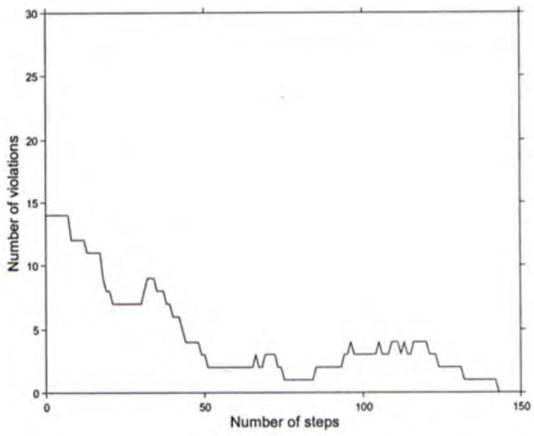


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

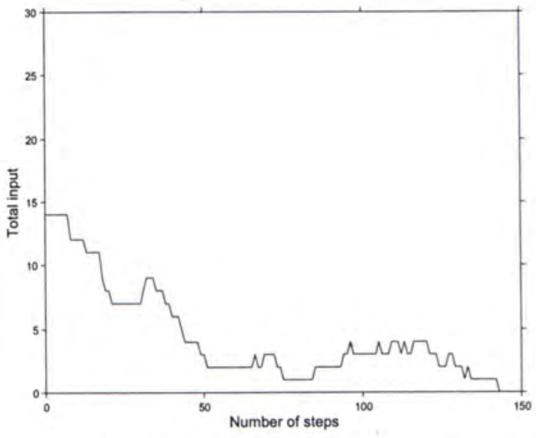


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

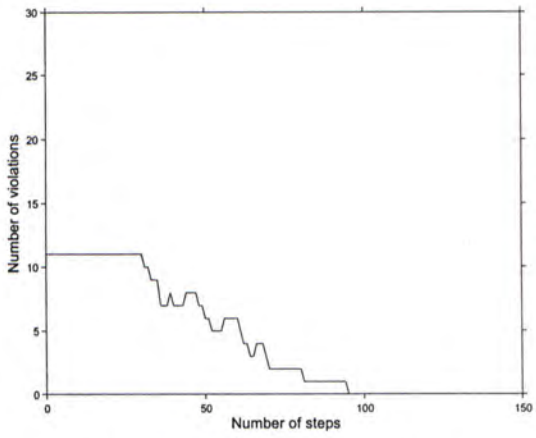
Figure 4.45: Numbers of violations and objective values in each step $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on permutation generation problem with $n = 50$ (short run-time case)



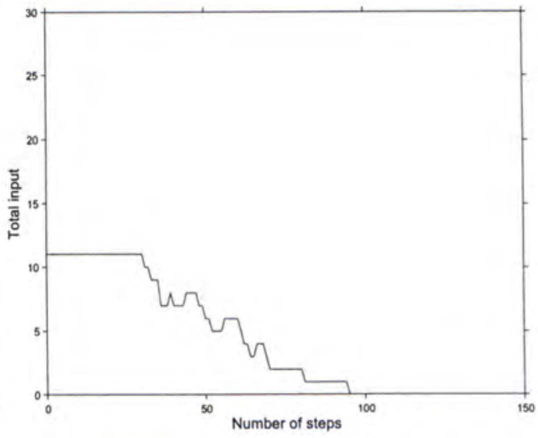
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

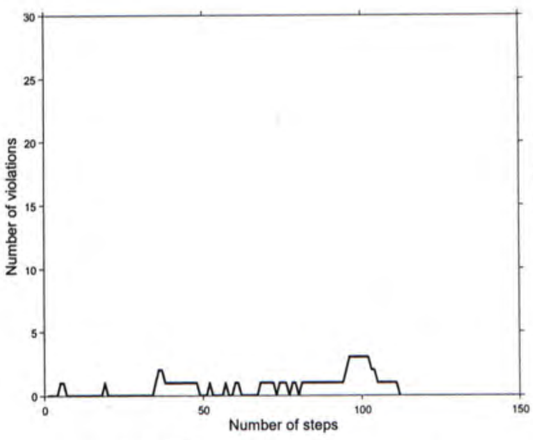


(c) max-PSS: Violation vs. Step

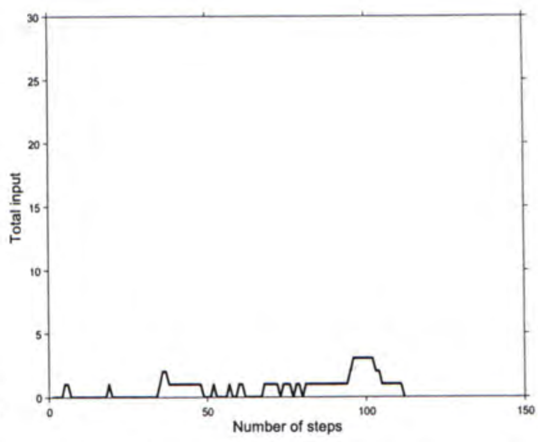


(d) max-PSS: Total input vs. Step

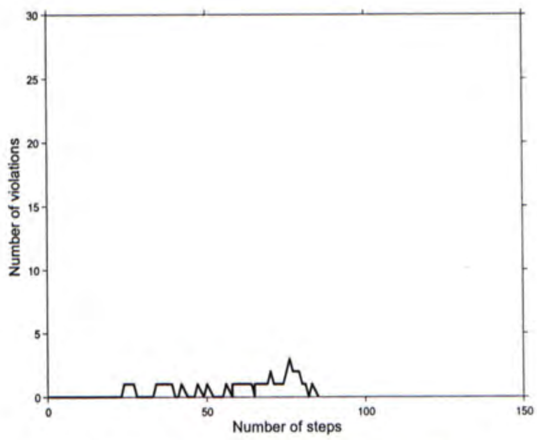
Figure 4.46: Numbers of violations and total inputs in each step of PSS and max-PSS on permutation generation problem with $n = 50$ (long run-time case)



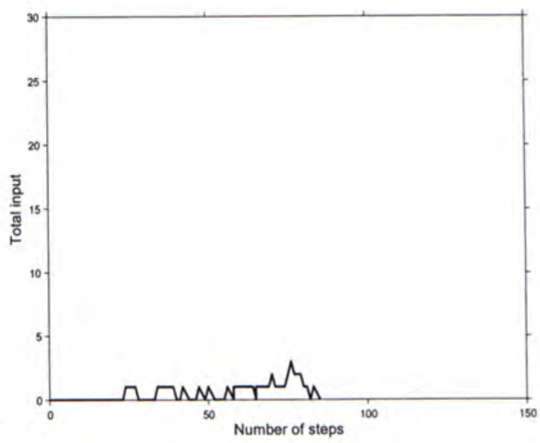
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

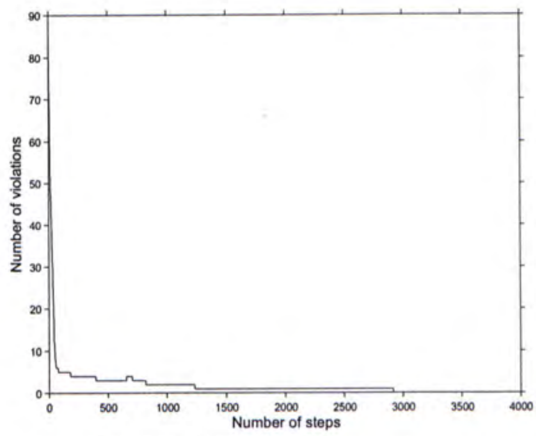


(c) max-IPSS: Violation vs. Step

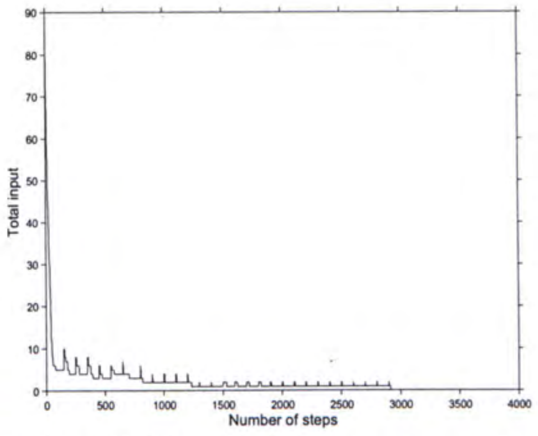


(d) max-IPSS: Total input vs. Step

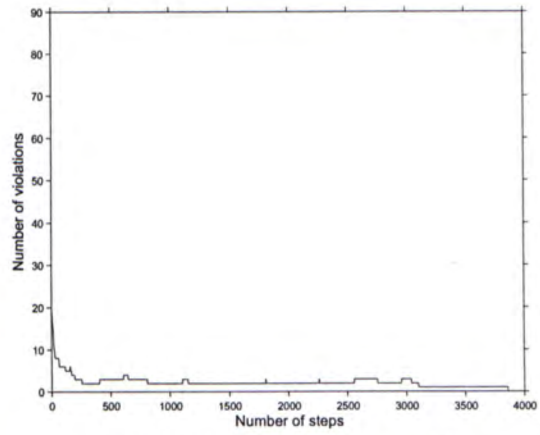
Figure 4.47: Numbers of violations and total inputs in each step of IPSS and max-IPSS on permutation generation problem with $n = 50$ (long run-time case)



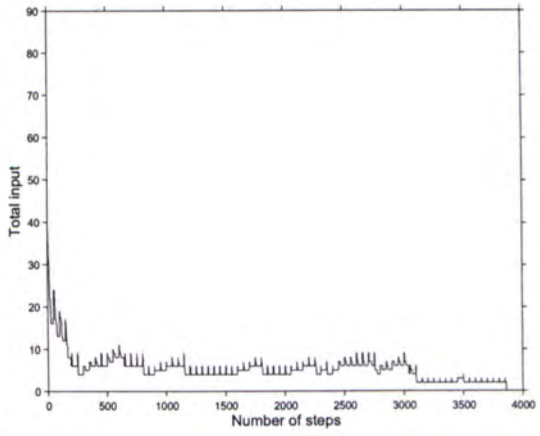
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

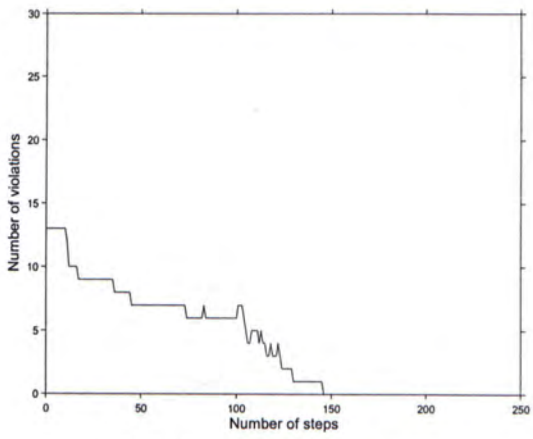


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

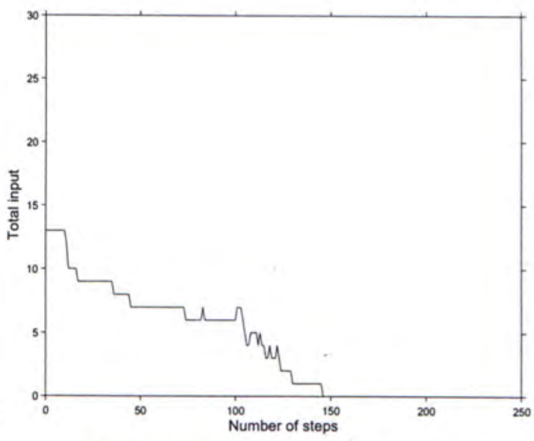


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

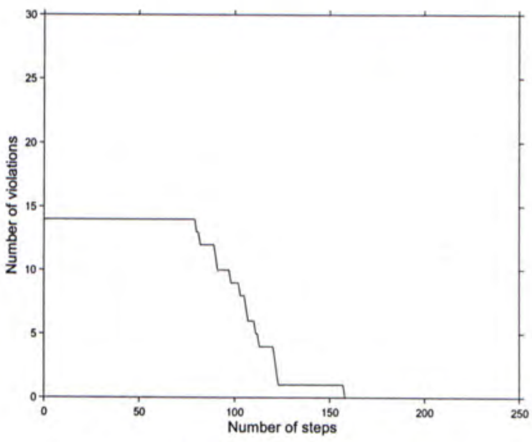
Figure 4.48: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on permutation generation problem with $n = 50$ (long run-time case)



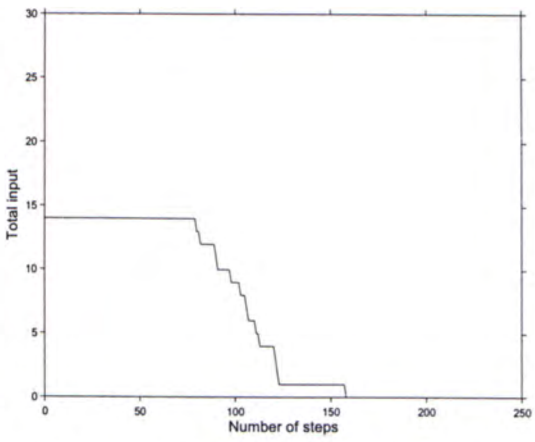
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

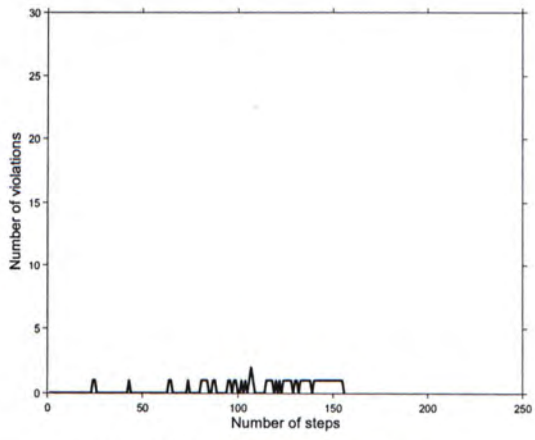


(c) max-PSS: Violation vs. Step

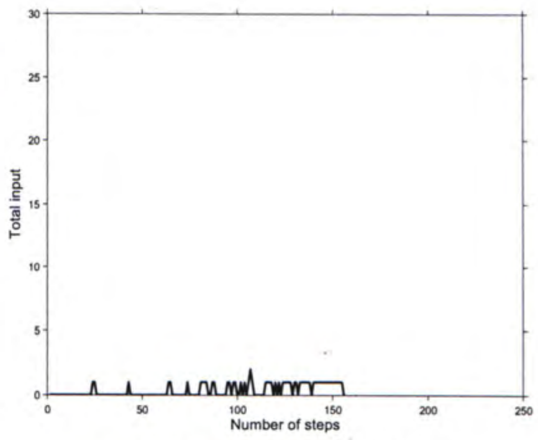


(d) max-PSS: Total input vs. Step

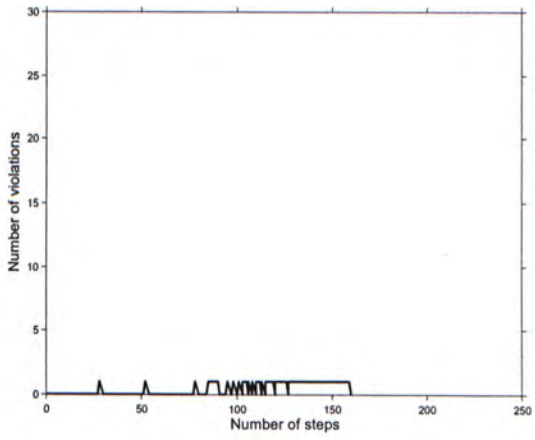
Figure 4.49: Numbers of violations and total inputs in each step of PSS and max-PSS on permutation generation problem with $n = 100$ (average run-time case)



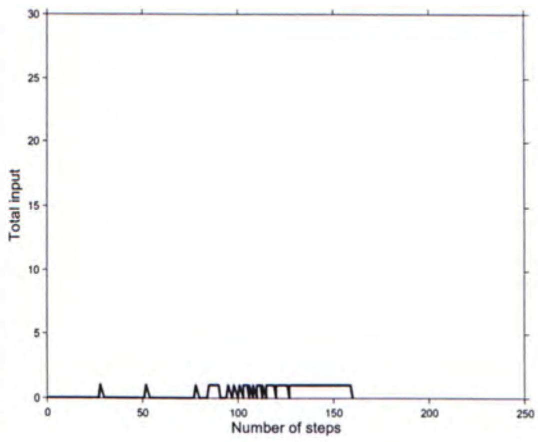
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

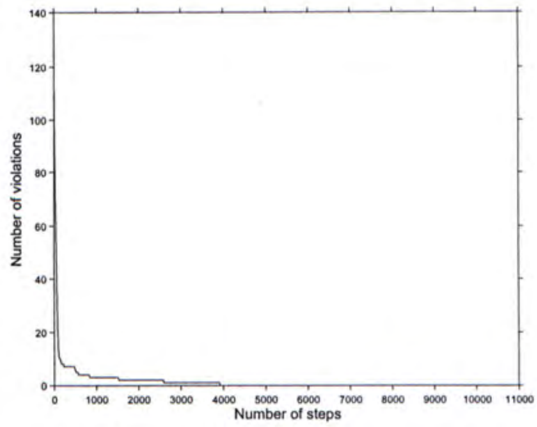


(c) max-IPSS: Violation vs. Step

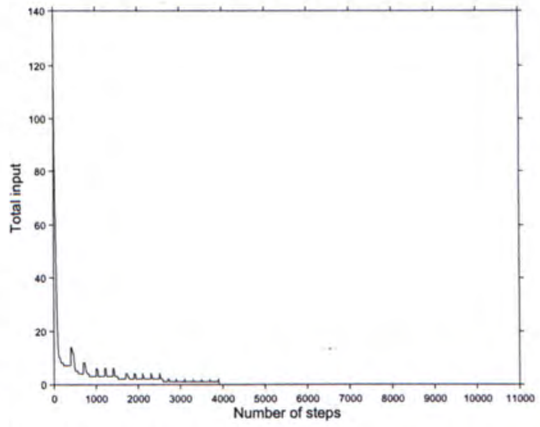


(d) max-IPSS: Total input vs. Step

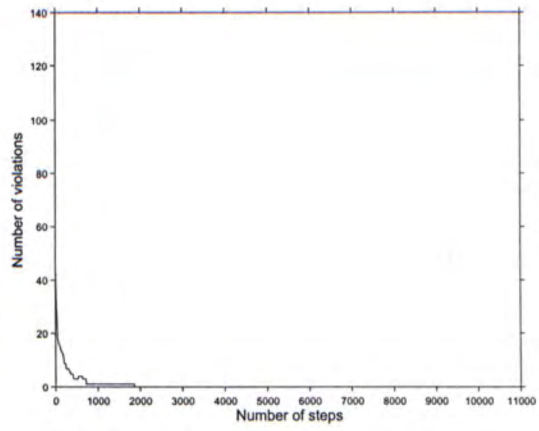
Figure 4.50: Numbers of violations and total inputs in each step of IPSS and max-IPSS on permutation generation problem with $n = 100$ (average run-time case)



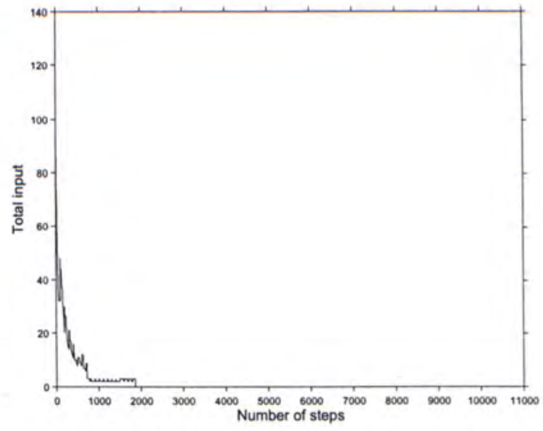
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

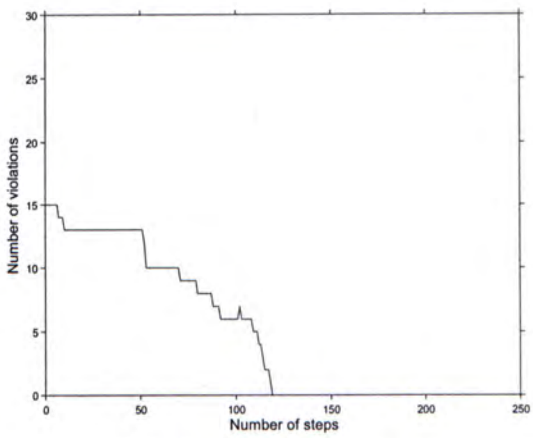


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

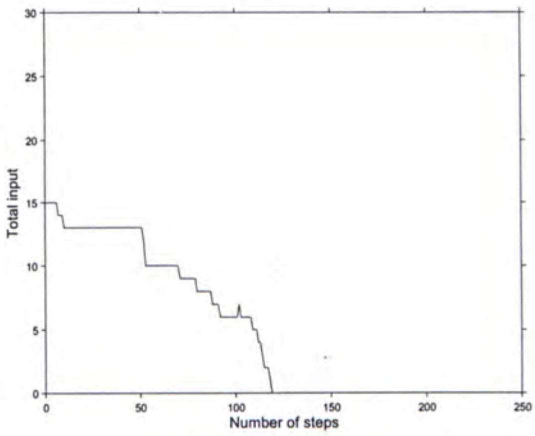


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

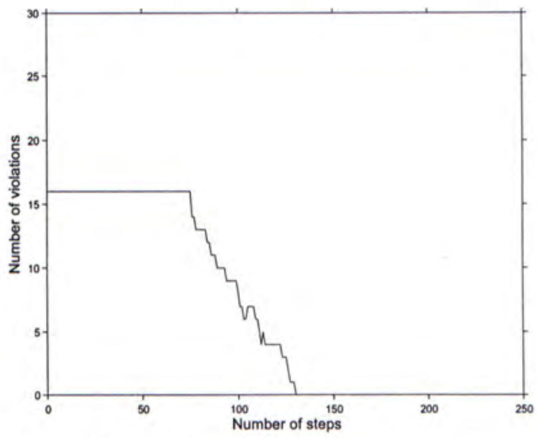
Figure 4.51: Numbers of violations and objective values/total inputs in each step $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on permutation generation problem with $n = 100$ (average run-time case)



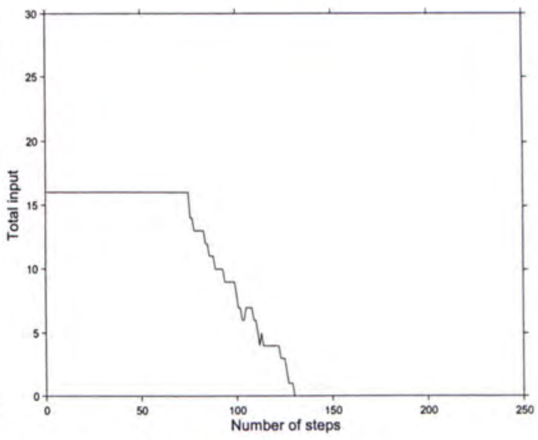
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

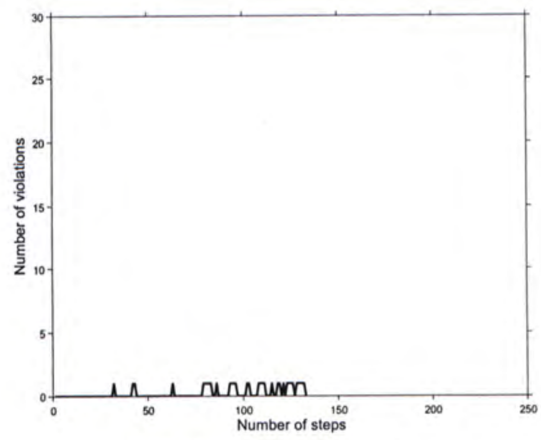


(c) max-PSS: Violation vs. Step

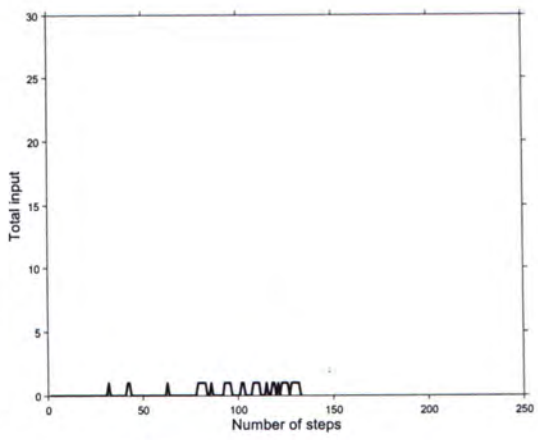


(d) IPSS: Total input vs. Step

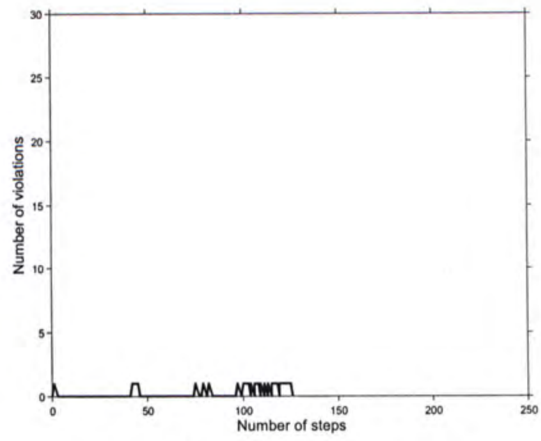
Figure 4.52: Numbers of violations and total inputs in each step of PSS and max-PSS on permutation generation problem with $n = 100$ (short run-time case)



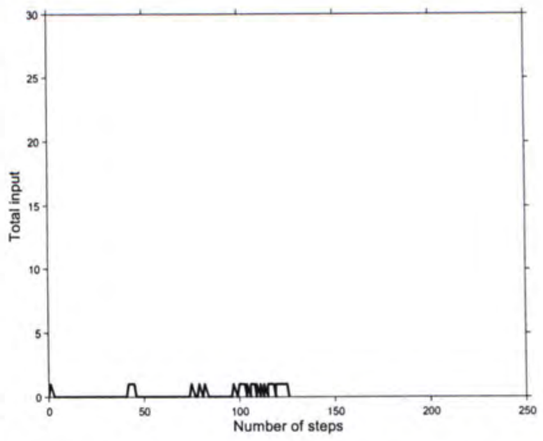
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

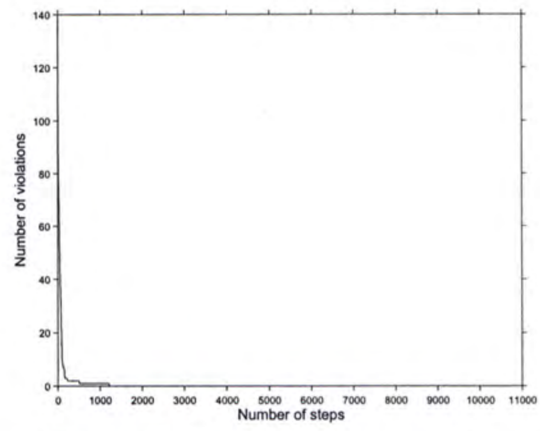


(c) max-IPSS: Violation vs. Step

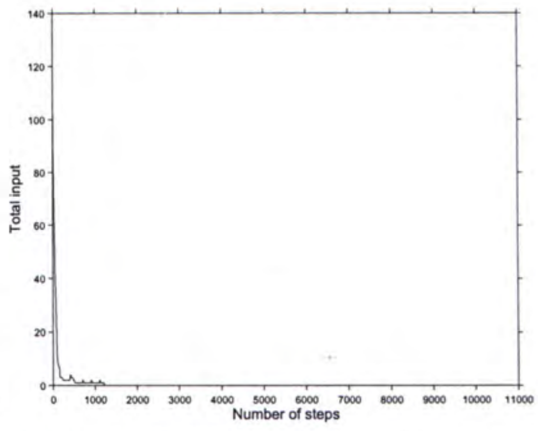


(d) max-IPSS: Total input vs. Step

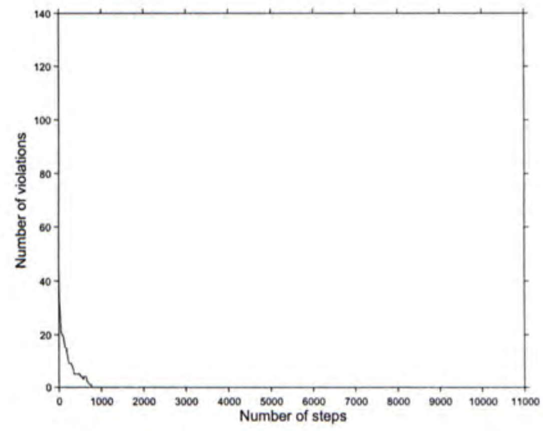
Figure 4.53: Numbers of violations and total inputs in each step of IPSS and max-IPSS on permutation generation problem with $n = 100$ (short run-time case)



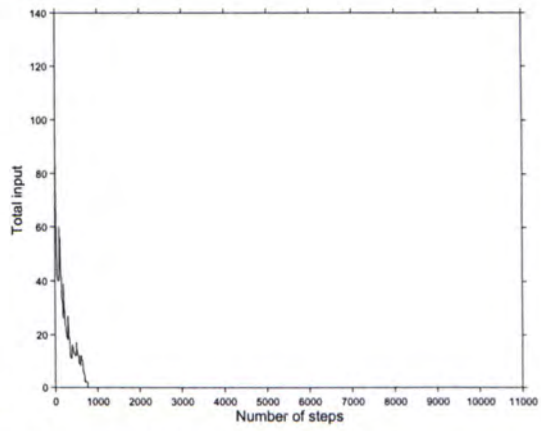
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

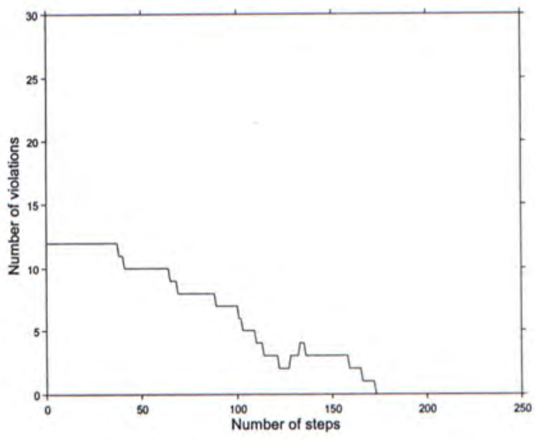


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

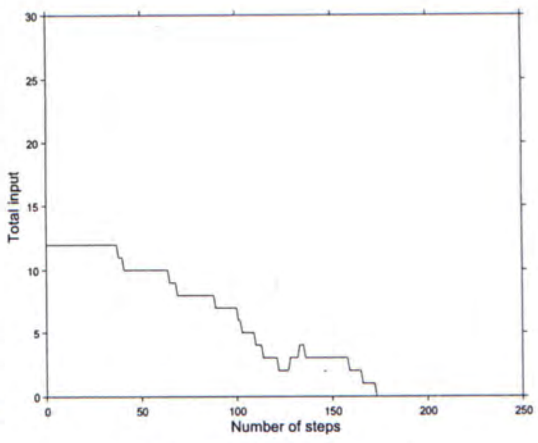


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

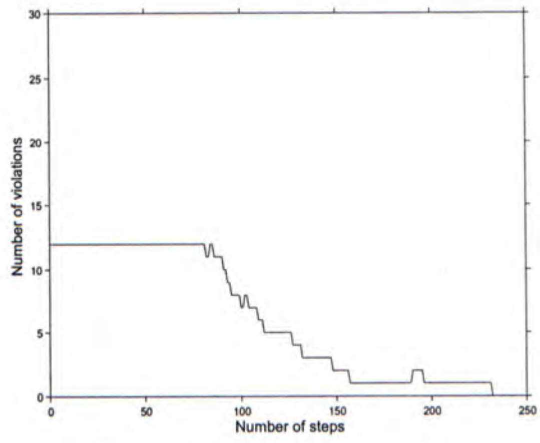
Figure 4.54: Numbers of violations and objective values in each step $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on permutation generation problem with $n = 100$ (short run-time case)



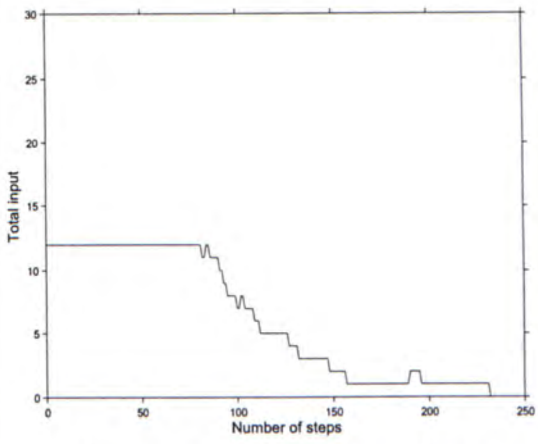
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

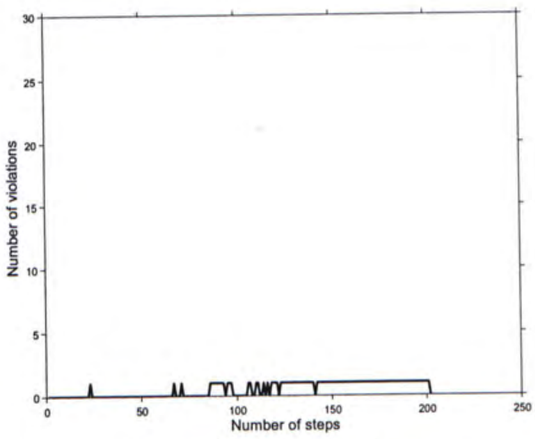


(c) max-PSS: Violation vs. Step

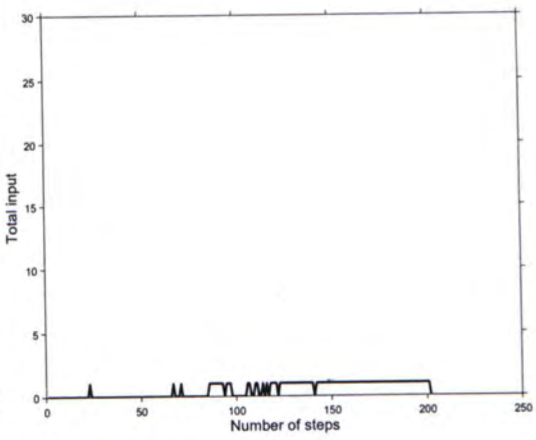


(d) max-PSS: Total input vs. Step

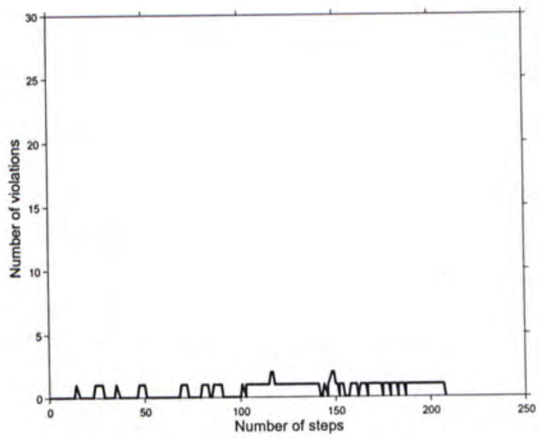
Figure 4.55: Numbers of violations and total inputs in each step of PSS and max-PSS on permutation generation problem with $n = 100$ (long run-time case)



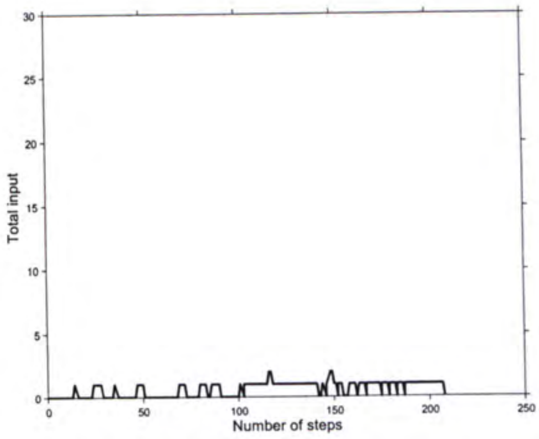
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

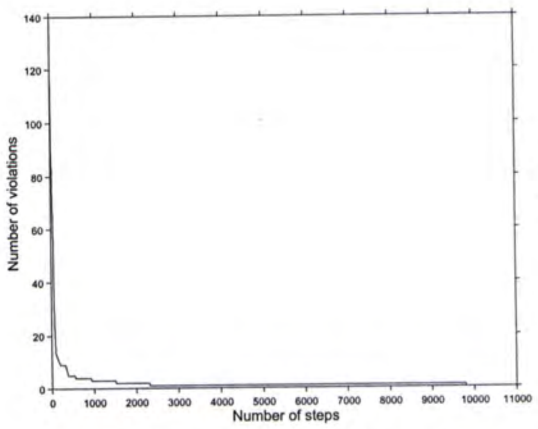


(c) max-IPSS: Violation vs. Step

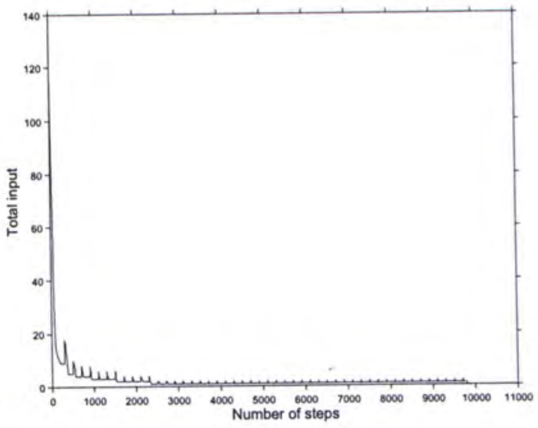


(d) max-IPSS: Total input vs. Step

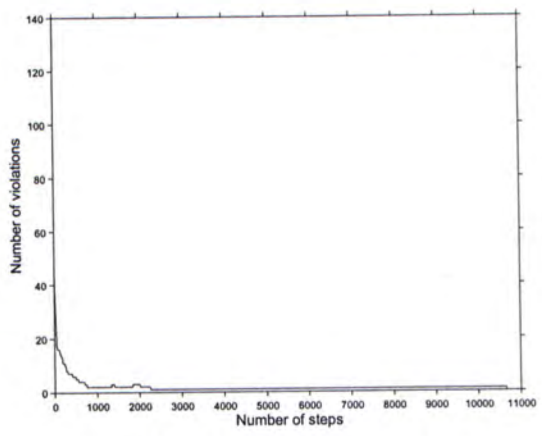
Figure 4.56: Numbers of violations and total inputs in each step of IPSS and max-IPSS on permutation generation problem with $n = 100$ (long run-time case)



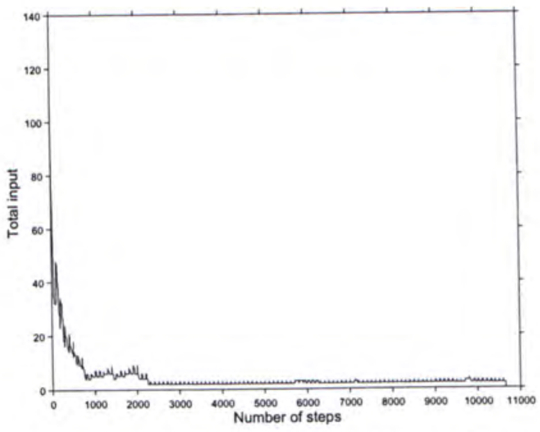
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step



(c) $\mathcal{LSDL}$(IMP): Violation vs. Step



(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

Figure 4.57: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on permutation generation problem with $n = 100$ (long run-time case)

4.3 Latin Squares and Quasigroup Completion Problems

A Latin square of order N is an $N \times N$ table of N symbols in which every symbol occurs exactly once in each row and column of the table. An incomplete Latin square of order N is a partially filled Latin square of order N. The quasigroup completion problem (QCP) [3] is a highly structured problem. The QCP is the problem that determines if the partial Latin square can be filled to be a complete Latin square.

A Latin square of order N can be modeled as a CSP with N^2 variables. Each variable represents one cell in the $N \times N$ table and has a domain $\{1, 2, \ldots, N\}$. The constraints state that no value occurs twice in a row or a column. A QCP can be modeled as a CSP that is similar to the modeling of Latin square except that the filled variables have their domains fixed to the pre-assigned value. Two sets of problems are used in this set of experiments. The first set of experiments consists of six instances of Latin square problems with orders ranging from $N = 10$ to $N = 35$ in steps of 5. The second set of experiments consists of six instances of quasigroup completion problems with orders ranging from $N = 15$ to $N = 20$.

4.3.1 Latin Square Problems

Table 4.7 shows the results of PSS and its variants on the set of Latin square problems. We give the results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) of the same problems in Table 4.8 for comparison. The mean timing results of all implementations are shown in Figure 4.58. From the timing figures, $\mathcal{LSDL}$ (GENET) has a better performance than PSS. $\mathcal{LSDL}$(IMP) has a better timing results than PSS, max-PSS and $\mathcal{LSDL}$ (GENET). IPSS and max-IPSS outperform the original PSS and also $\mathcal{LSDL}$ implementations.

Problem	PSS		IPSS	
N	Steps	**CPU time**	Steps	**CPU time**
10	149.7(120.0)	**0.0000(0.0000)**	112.6(119.0)	**0.0000(0.0000)**
15	430.3(396.0)	**0.0046(0.0000)**	270.9(274.0)	**0.0017(0.0000)**
20	1008(965.0)	**0.0134(0.0100)**	510.5(503.0)	**0.0040(0.0000)**
25	1716(1546)	**0.0377(0.0400)**	796.6(790.0)	**0.0136(0.0100)**
30	2667(2524)	**0.0728(0.0700)**	1160(1151)	**0.0227(0.0200)**
35	3706(3246)	**0.1227(0.1100)**	1586(1579)	**0.0368(0.0400)**
	max-PSS		max-IPSS	
10	121.6(138.0)	**0.0000(0.0000)**	123.7(124.0)	**0.0000(0.0000)**
15	349.6(312.0)	**0.0029(0.0000)**	273.8(281.0)	**0.0030(0.0000)**
20	649.8(641.0)	**0.0092(0.0100)**	502.1(501.0)	**0.0060(0.0100)**
25	1090(1070)	**0.0245(0.0200)**	796.6(785.0)	**0.0154(0.0200)**
30	1601(1578)	**0.0445(0.0400)**	1140(1134)	**0.0255(0.0300)**
35	2250(2189)	**0.0740(0.0700)**	1589(1587)	**0.0451(0.0400)**

Table 4.7: PSS and its variants on Latin square problems

Problem	$\mathcal{LSDL}$(GENET)			
N	Iteration	Repairs	Learns	**CPU time**
10	46.3(49.0)	134.6(133.0)	19.2(20.0)	**0.0001(0.0000)**
15	65.6(36.0)	227.5(244.0)	27.6(14.0)	**0.0042(0.0000)**
20	100.9(94.0)	535.8(520.0)	43.1(40.0)	**0.0121(0.0100)**
25	196.2(163.5)	955.0(928.5)	88.4(71.0)	**0.0324(0.0300)**
30	241.8(190.0)	1393(1332)	109.4(84.0)	**0.0587(0.0600)**
35	275.8(221.0)	1896(1838)	124.6(97.0)	**0.0957(0.0900)**
Problem	$\mathcal{LSDL}$(IMP)			
N	Iteration	Repairs	Learns	**CPU time**
10	32.3(7.00)	86.69(23.00)	32.3(7.00)	**0.0000(0.0000)**
15	27.0(25.0)	138.5(131.0)	27.0(25.0)	**0.0031(0.0000)**
20	55.5(33.0)	302.8(263.0)	55.5(33.0)	**0.0077(0.0100)**
25	71.7(55.0)	508.5(510.0)	71.7(55.0)	**0.0202(0.0200)**
30	72.6(68.5)	754.3(787.5)	72.6(68.5)	**0.0355(0.0400)**
35	116.0(96.0)	1201(1170)	116.0(96.0)	**0.0654(0.0600)**

Table 4.8: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problems

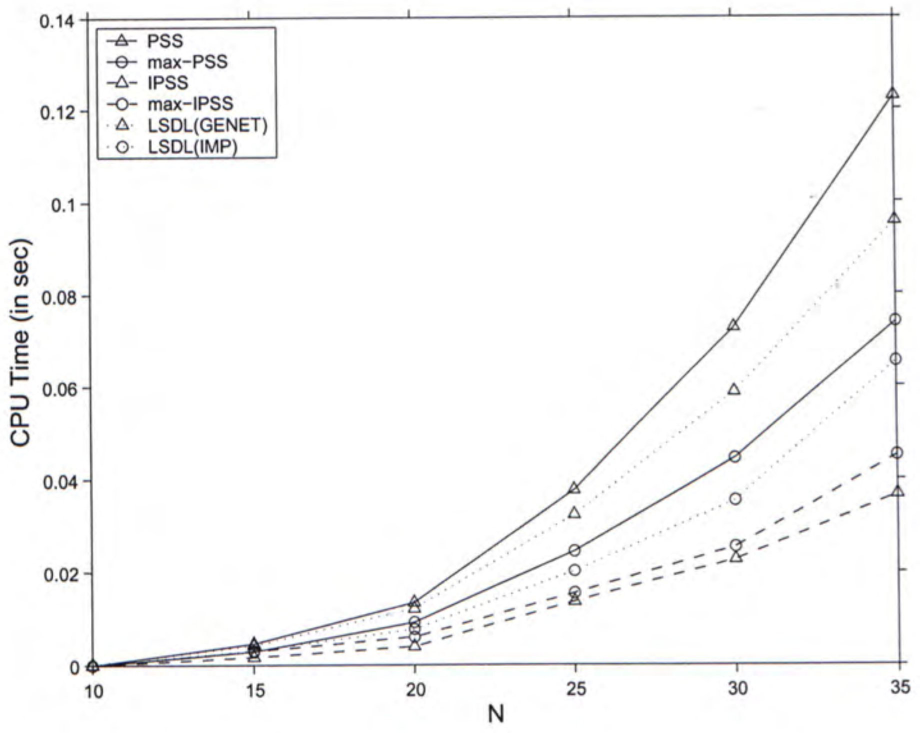
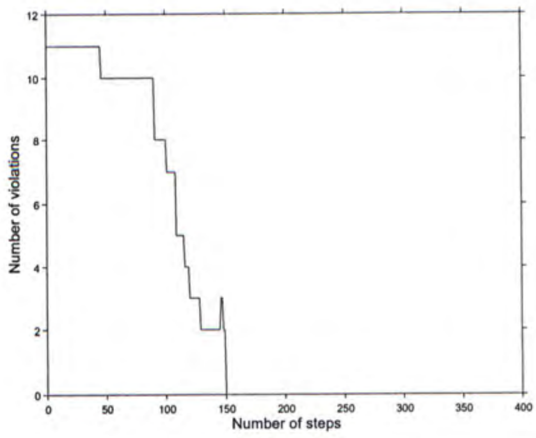


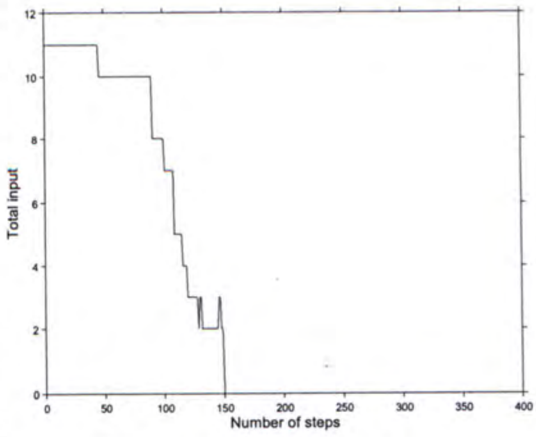
Figure 4.58: The mean time results on Latin square problems

The timing results show that IPSS has a better performance than PSS. Figures 4.59 - 4.67 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square of order 10. Figures 4.68 - 4.76 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square of order 35. In Figures 4.60(a), 4.63(a), 4.66(a), 4.69(a), 4.72(a) and 4.75(a), the partial solutions found by IPSS can be easily extended: only several steps are required to incorporate a new variable. We note that the points with zero number of violations represent the partial solutions. Moreover, IPSS always keeps the number of violations to extremely small values (typically 1), in contrast to that in the original PSS, which can be a dozen or two (Figures 4.59(a), 4.62(a), 4.65(a), 4.68(a), 4.71(a) and 4.74(a)). This experiment demonstrates the advantage of using incremental search to solve this kind of problem. On the other hand, max-PSS much improves on PSS in solving Latin square problems. Analysis of traces of execution shows that the cluster selection heuristics used helps decreasing the number of violations in a fast rate (Figures 4.59(c), 4.62(c), 4.65(c), 4.68(c), 4.71(c) and 4.74(c)). We note that max-IPSS has almost the same performance as IPSS. From Table 4.7, max-IPSS requires a little bit more time than IPSS to solve the problems. The reason is that max-IPSS needs time to select the suitable cluster in the list $\mathcal{F}$ for repairing.

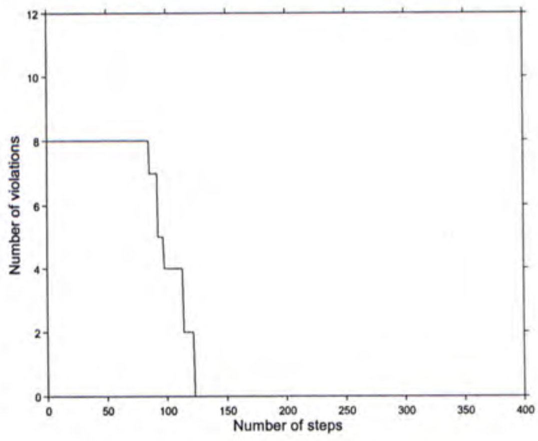
From the tables, we can see that the number of repairs in $\mathcal{LSDL}$(GENET) is nearly the same as that of IPSS, max-IPSS and max-PSS. However, the number of steps taken in $\mathcal{LSDL}$(GENET) is much more. This is the reason that all variants of PSS outperform $\mathcal{LSDL}$(GENET). Although $\mathcal{LSDL}$(IMP) uses fewer repairs than IPSS, max-IPSS and max-PSS, it takes more steps to find the solution. That makes all variants of PSS outperform $\mathcal{LSDL}$(IMP).



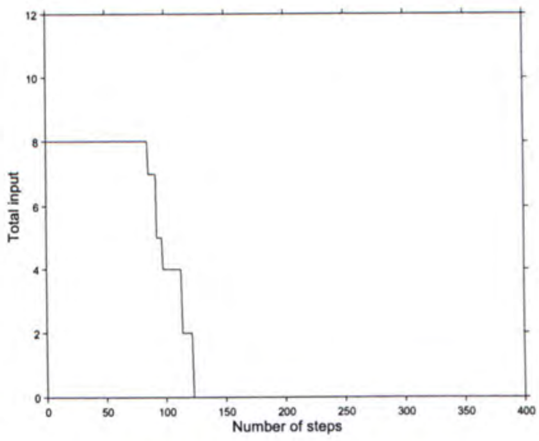
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

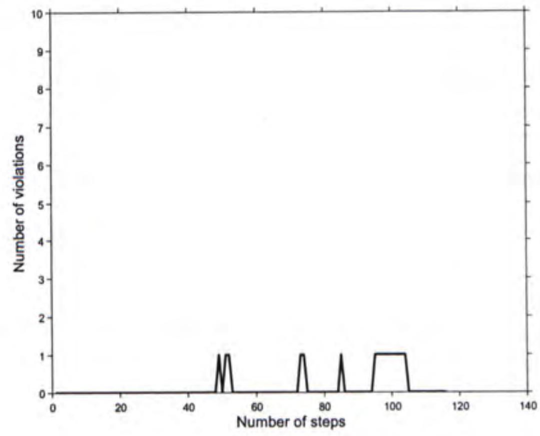


(c) max-PSS: Violation vs. Step

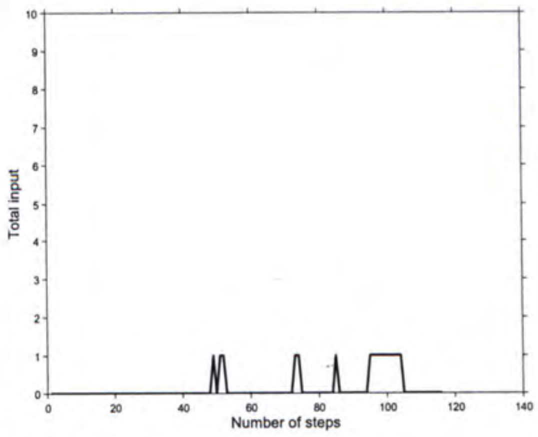


(d) max-PSS: Total input vs. Step

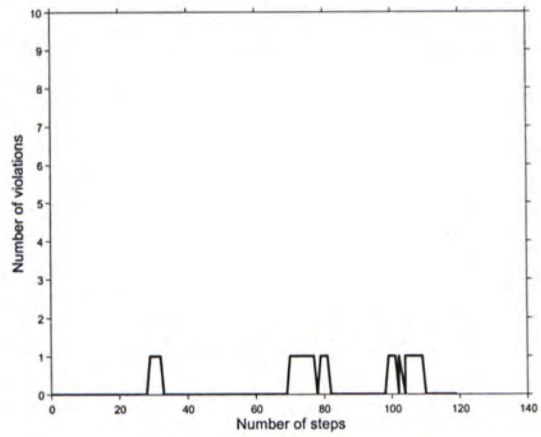
Figure 4.59: Numbers of violations and total inputs in each step of PSS and max-PSS on Latin square problem with $N = 10$ (average run-time case)



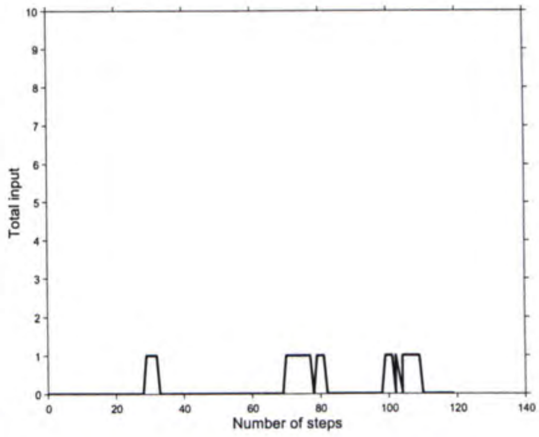
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

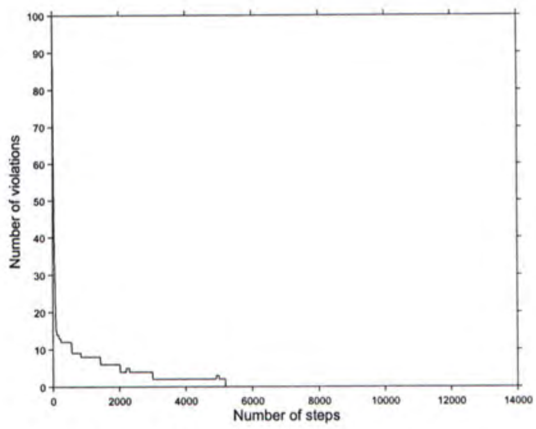


(c) max-IPSS: Violation vs. Step

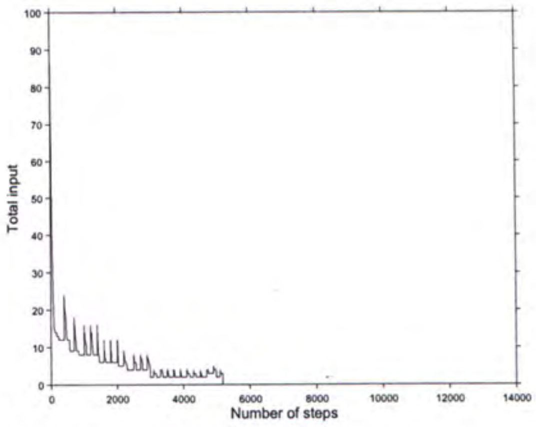


(d) max-IPSS: Total input vs. Step

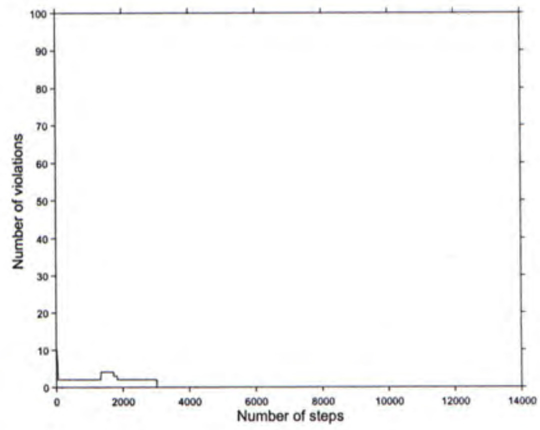
Figure 4.60: Numbers of violations and total inputs in each step of IPSS and max-IPSS on Latin square problem with $N = 10$ (average run-time case)



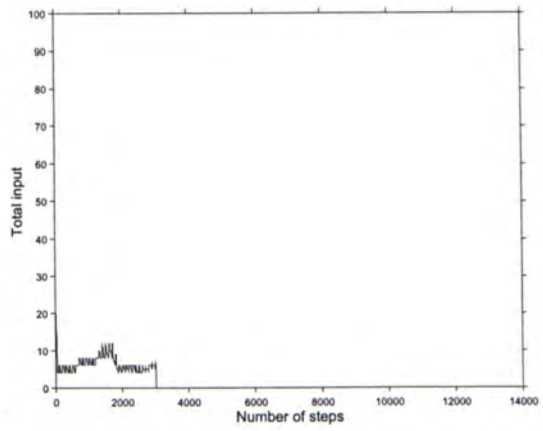
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

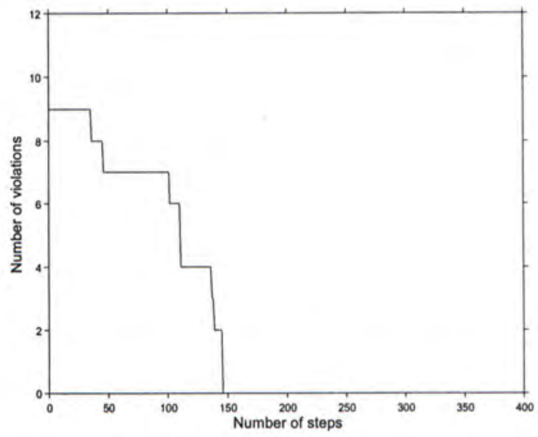


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

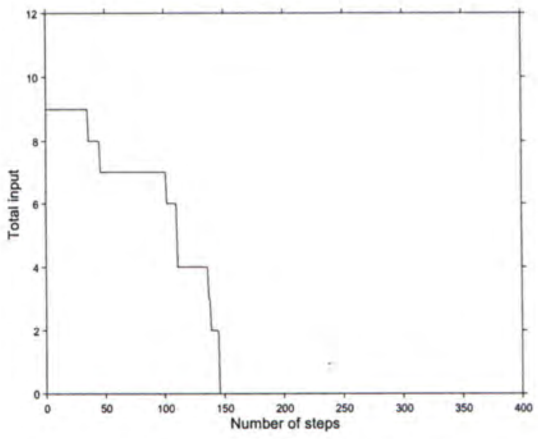


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

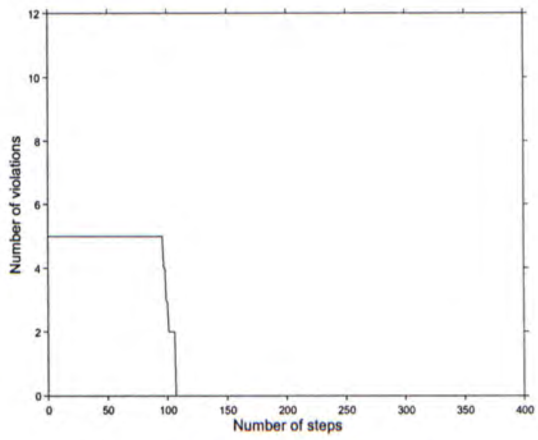
Figure 4.61: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problem with $N = 10$ (average run-time case)



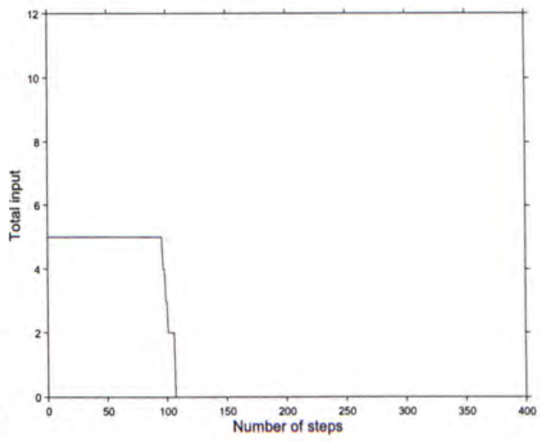
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

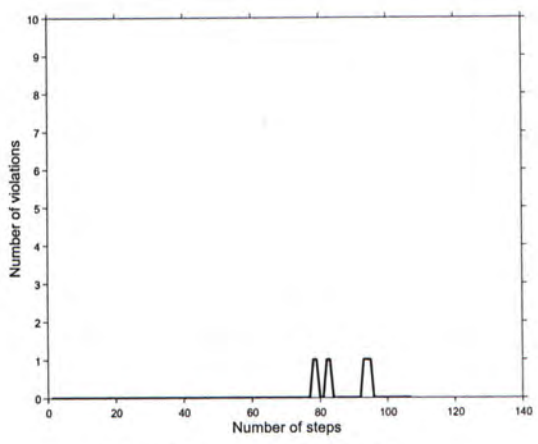


(c) max-PSS: Violation vs. Step

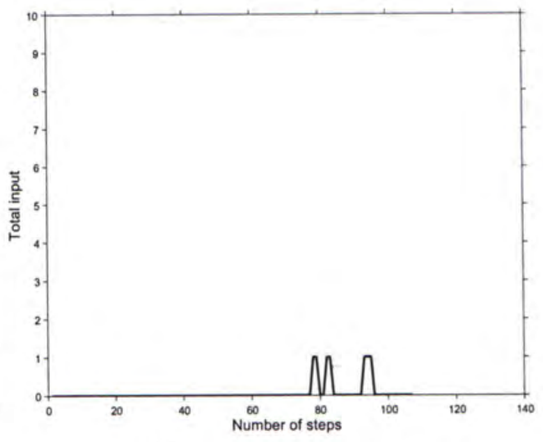


(d) max-PSS: Total input vs. Step

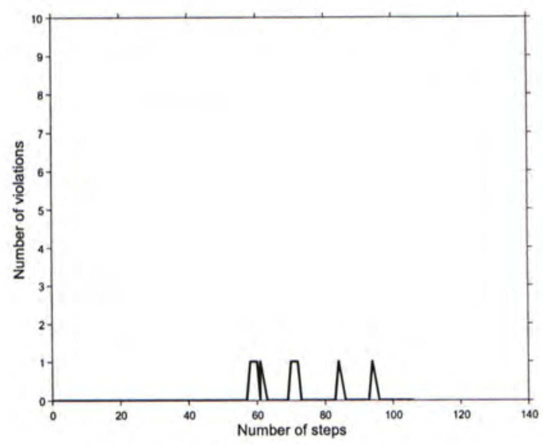
Figure 4.62: Numbers of violations and total inputs in each step of PSS and max-PSS on Latin square problem with $N = 10$ (short run-time case)



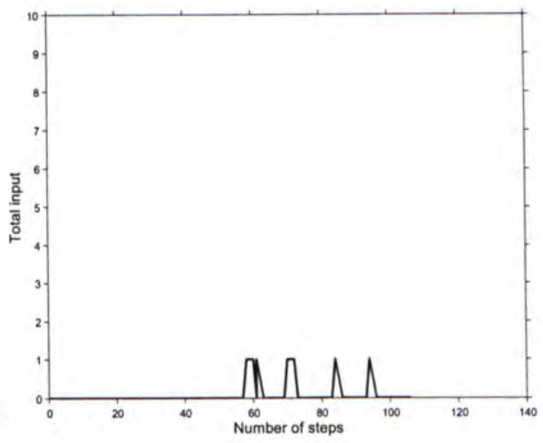
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

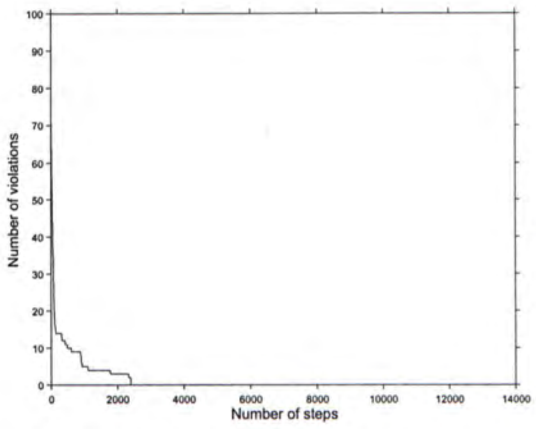


(c) max-IPSS: Violation vs. Step

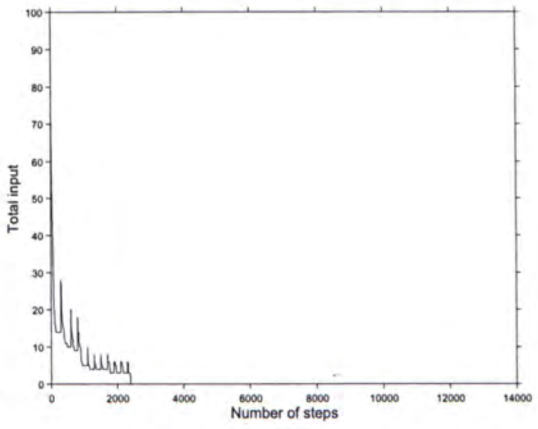


(d) max-IPSS: Total input vs. Step

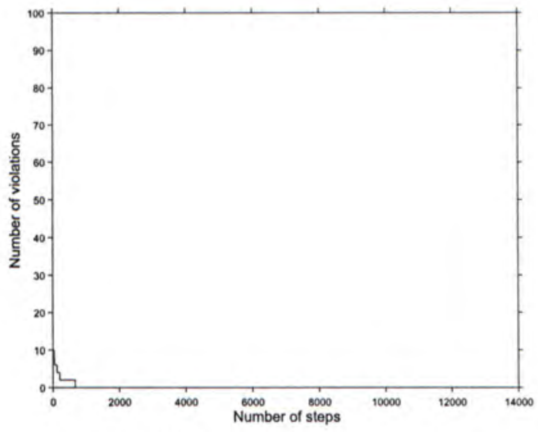
Figure 4.63: Numbers of violations and total inputs in each step of IPSS and max-IPSS on Latin square problem with $N = 10$ (short run-time case)



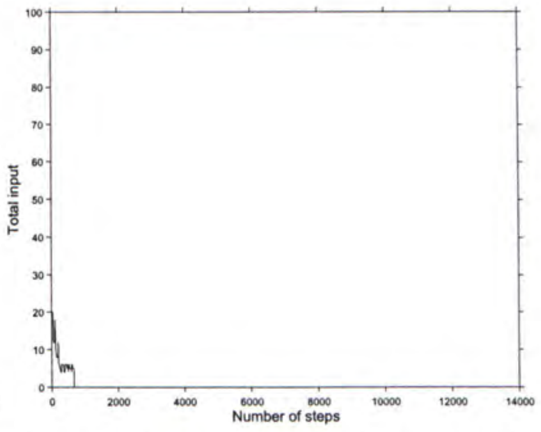
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

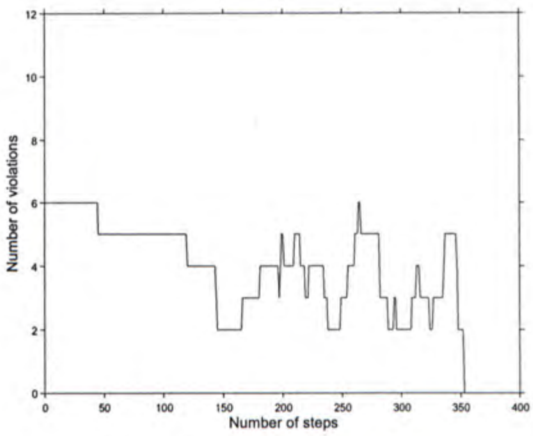


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

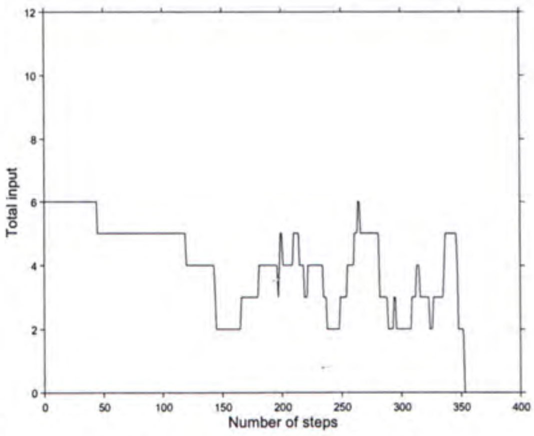


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

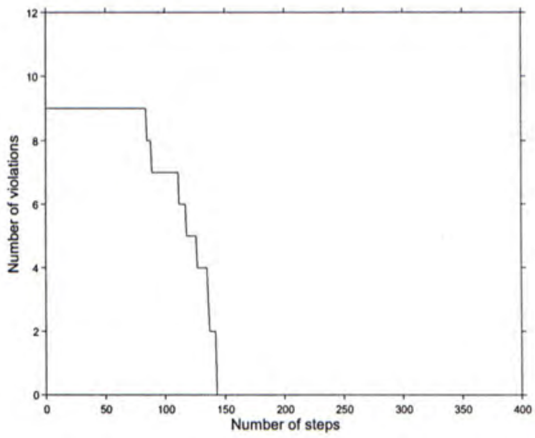
Figure 4.64: Numbers of violations and objective values in each step $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problem with $N = 10$ (short run-time case)



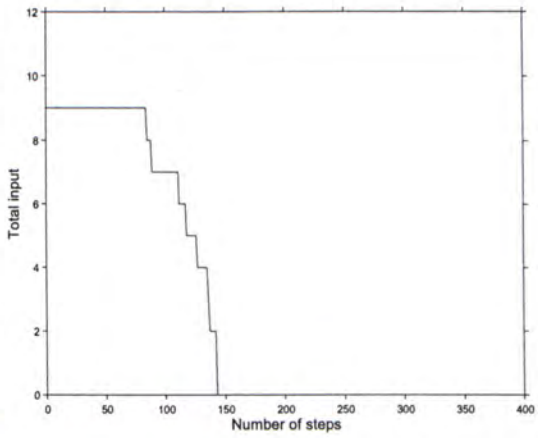
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

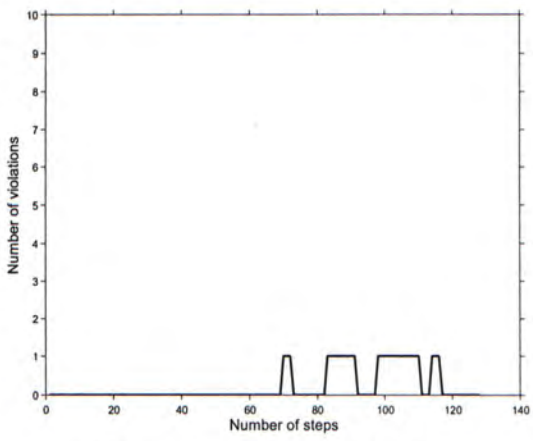


(c) max-PSS: Violation vs. Step

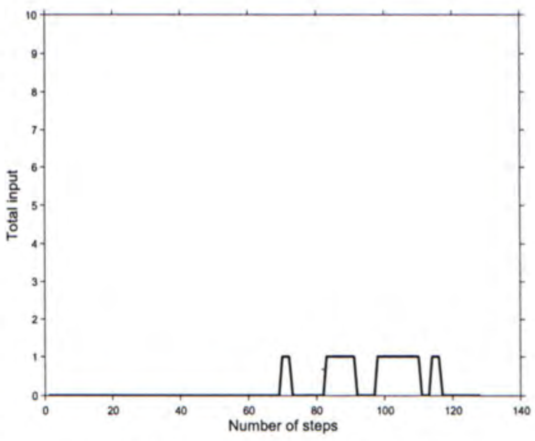


(d) max-PSS: Total input vs. Step

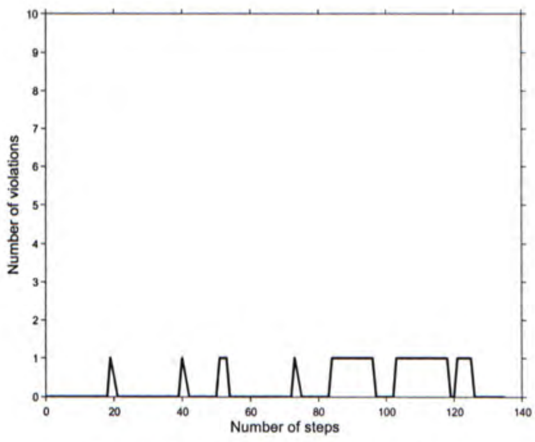
Figure 4.65: Numbers of violations and total inputs in each step of PSS and max-PSS on Latin square problem with $N = 10$ (long run-time case)



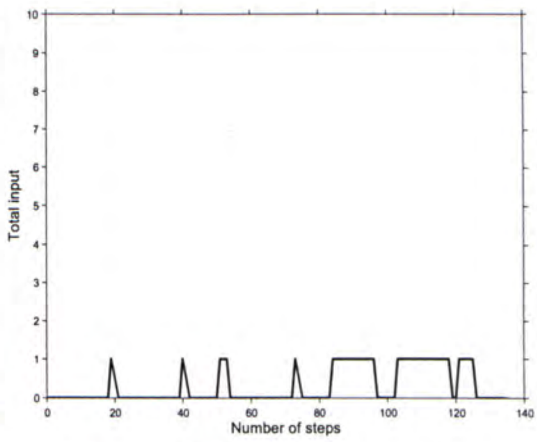
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

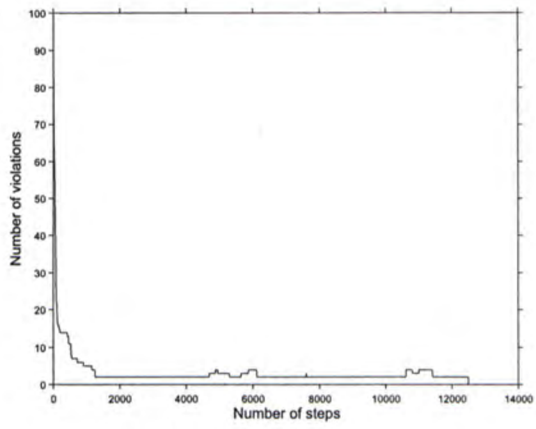


(c) max-IPSS: Violation vs. Step

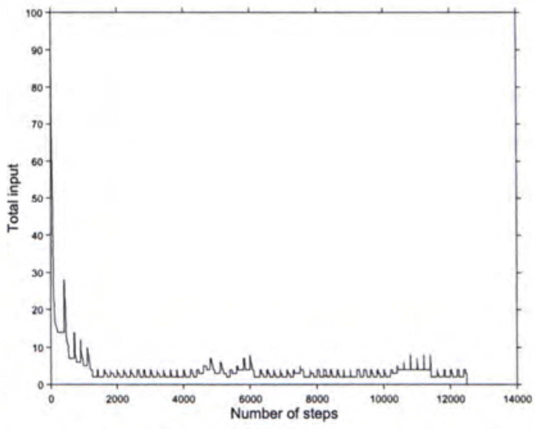


(d) max-IPSS: Total input vs. Step

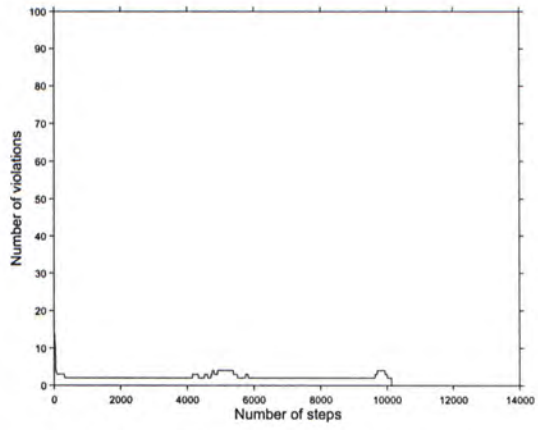
Figure 4.66: Numbers of violations and total inputs in each step of IPSS and max-IPSS on Latin square problem with $N = 10$ (long run-time case)



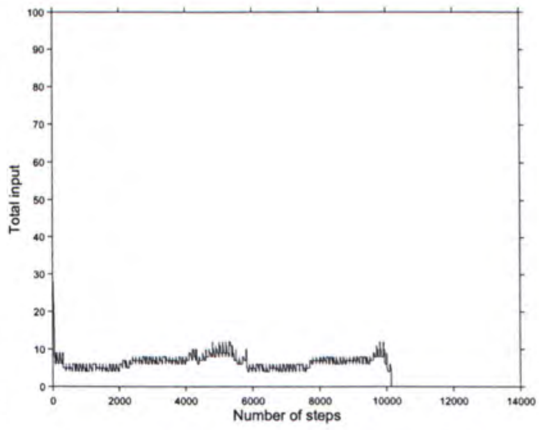
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

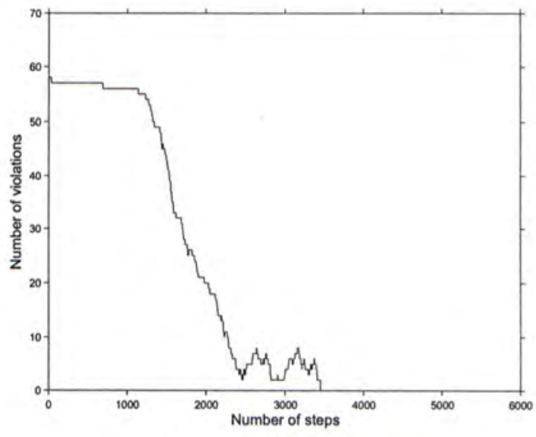


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

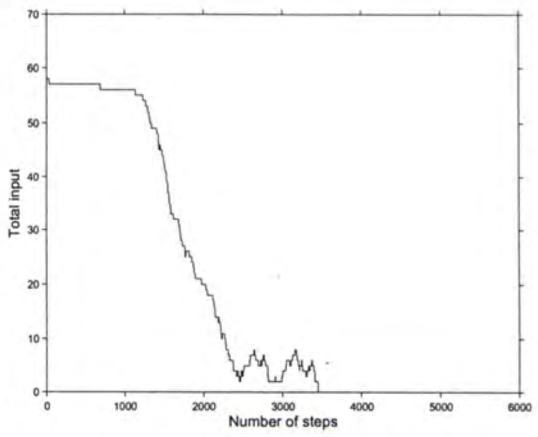


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

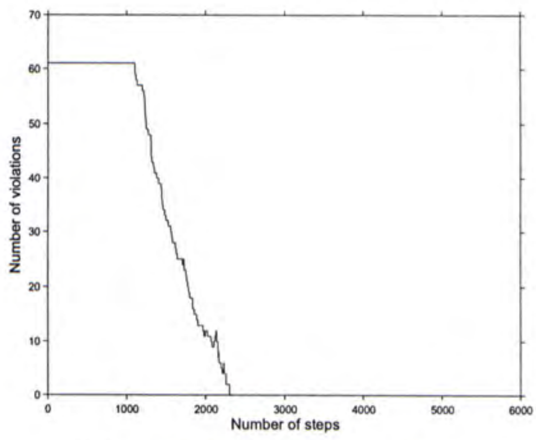
Figure 4.67: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problem with $N = 10$ (long run-time case)



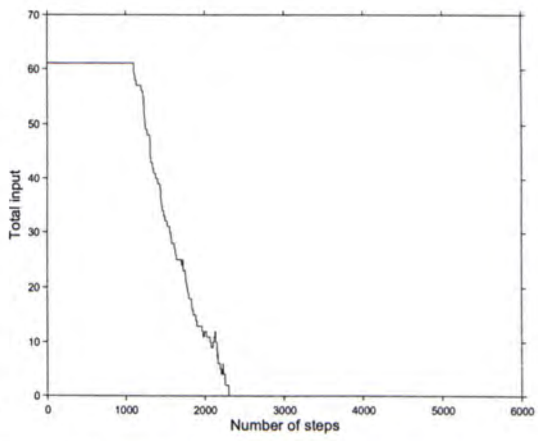
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

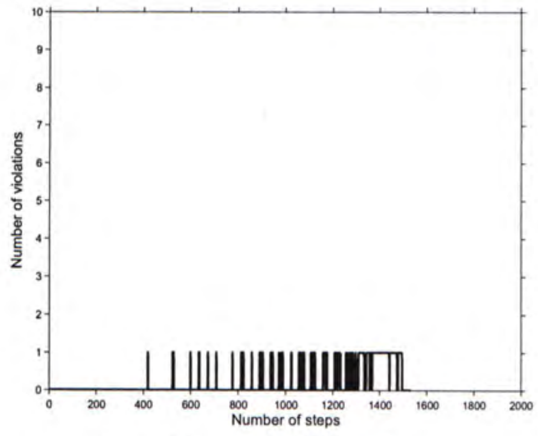


(c) max-PSS: Violation vs. Step

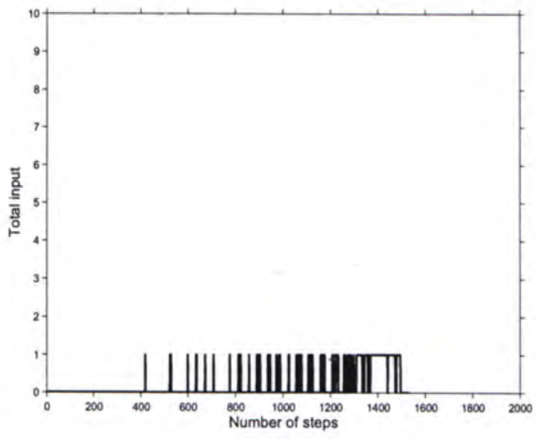


(d) max-PSS: Total input vs. Step

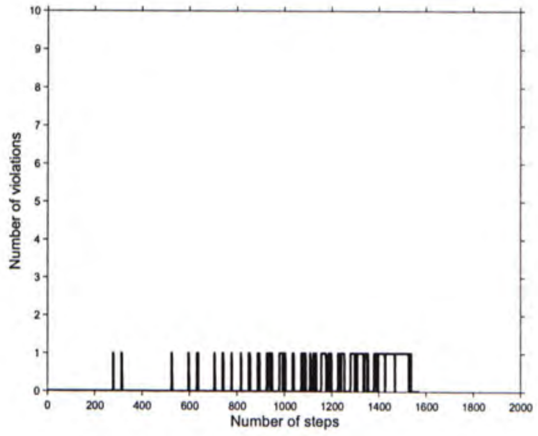
Figure 4.68: Numbers of violations and total inputs in each step of PSS and max-PSS on Latin square problem with $N = 35$ (average run-time case)



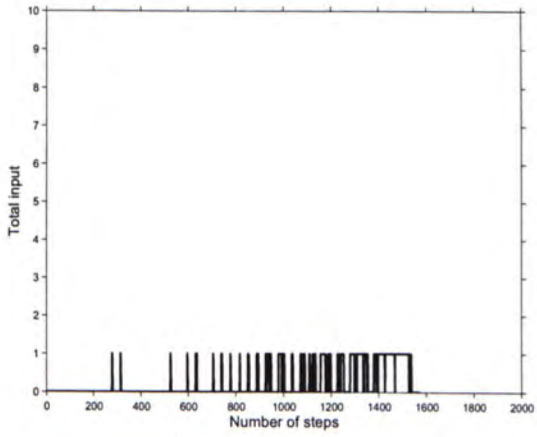
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

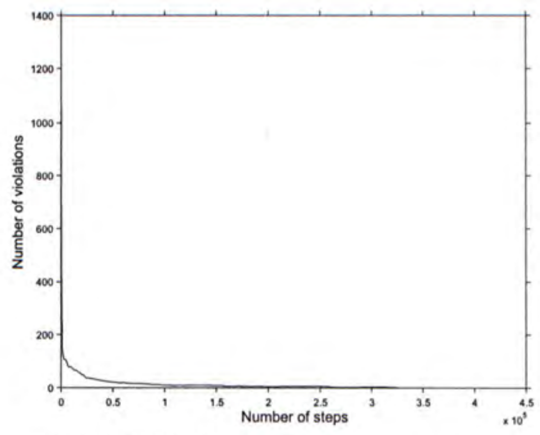


(c) max-IPSS: Violation vs. Step

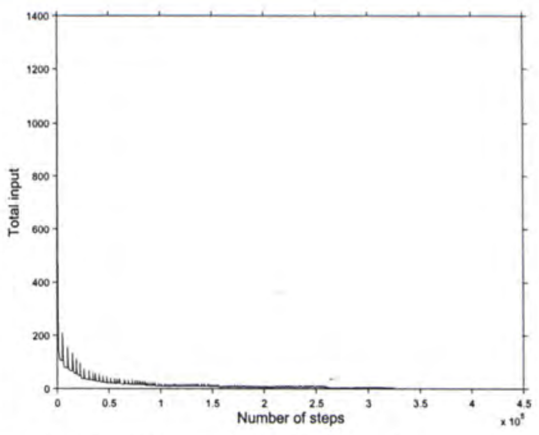


(d) max-IPSS: Total input vs. Step

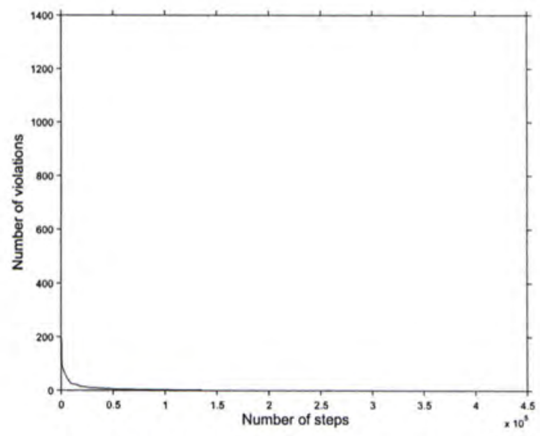
Figure 4.69: Numbers of violations and total inputs in each step of IPSS and max-IPSS on Latin square problem with $N = 35$ (average run-time case)



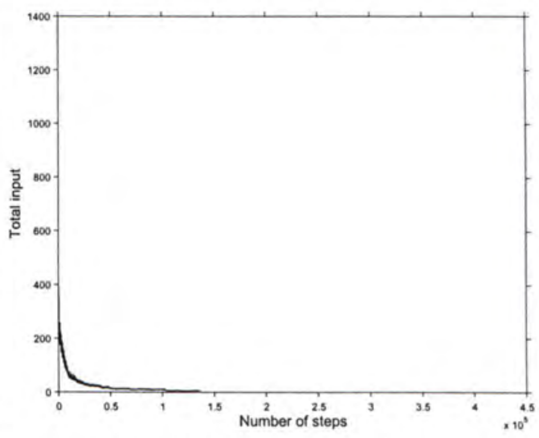
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

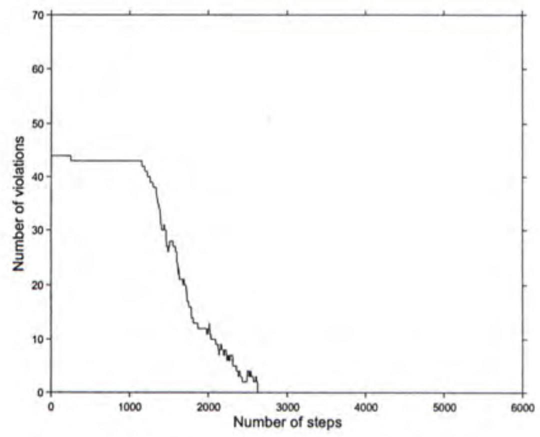


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

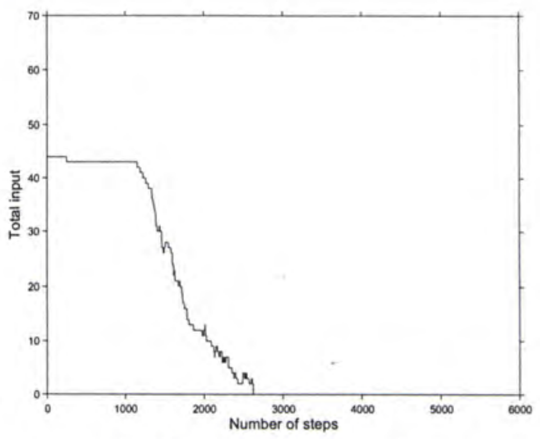


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

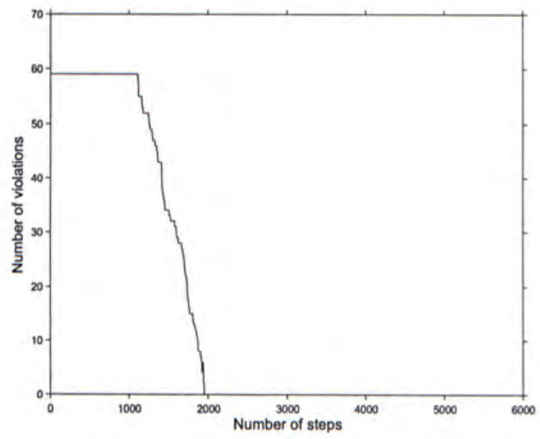
Figure 4.70: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problem with $N = 35$ (average run-time case)



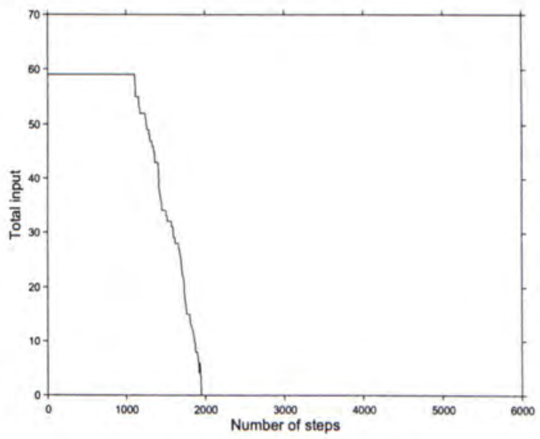
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

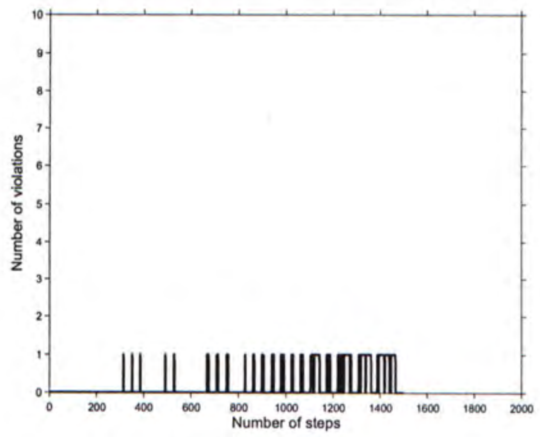


(c) max-PSS: Violation vs. Step

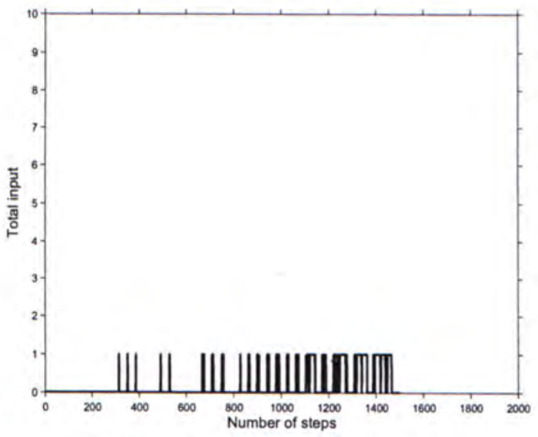


(d) max-PSS: Total input vs. Step

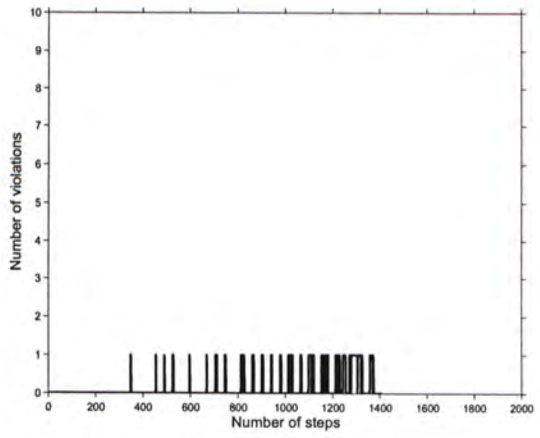
Figure 4.71: Numbers of violations and total inputs in each step of PSS and max-PSS on Latin square problem with $N = 35$ (short run-time case)



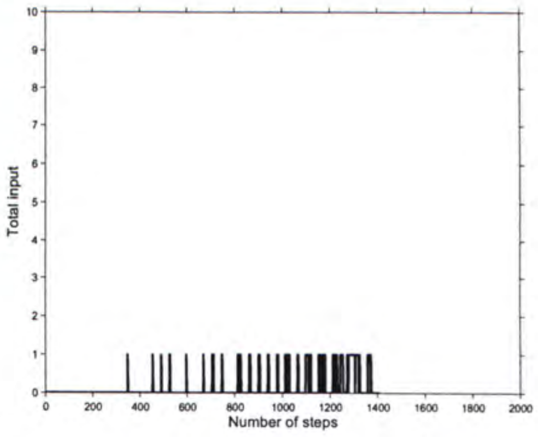
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

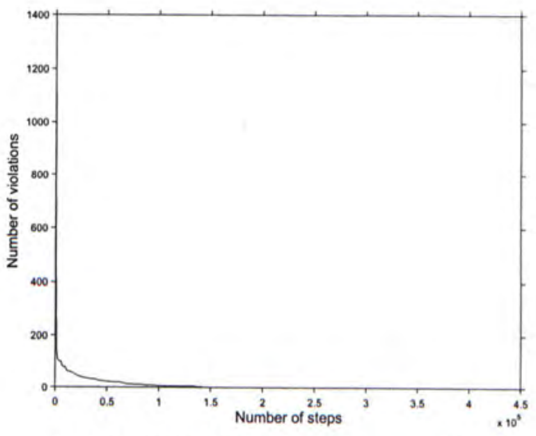


(c) max-IPSS: Violation vs. Step

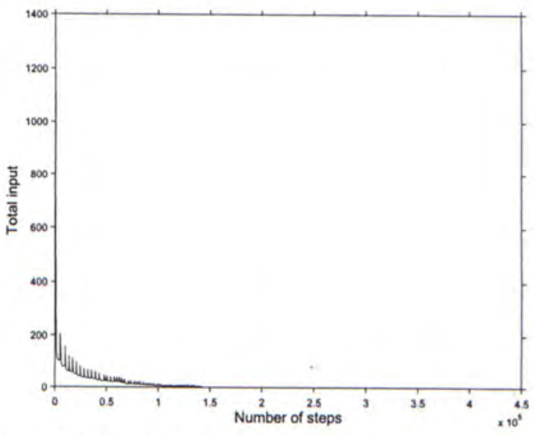


(d) max-IPSS: Total input vs, Step

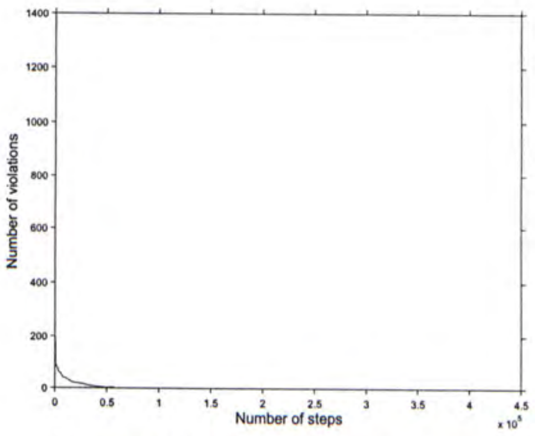
Figure 4.72: Numbers of violations and total inputs in each step of IPSS and max-IPSS on Latin square problem with $N = 35$ (short run-time case)



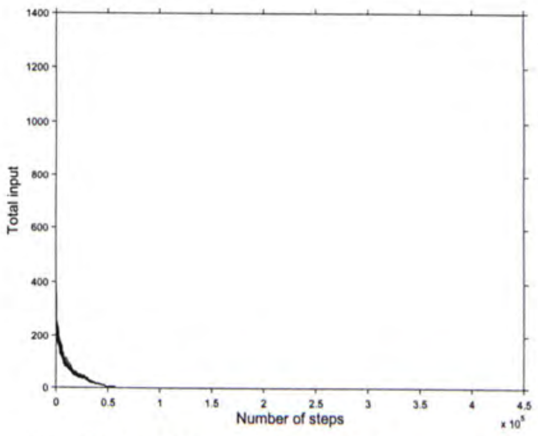
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

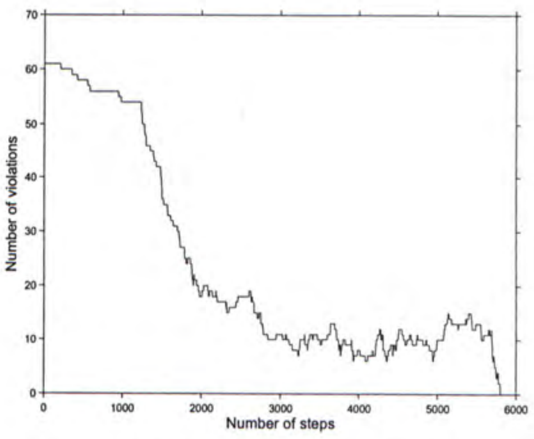


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

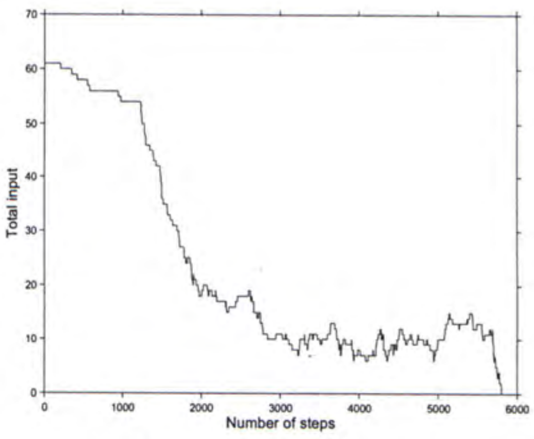


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

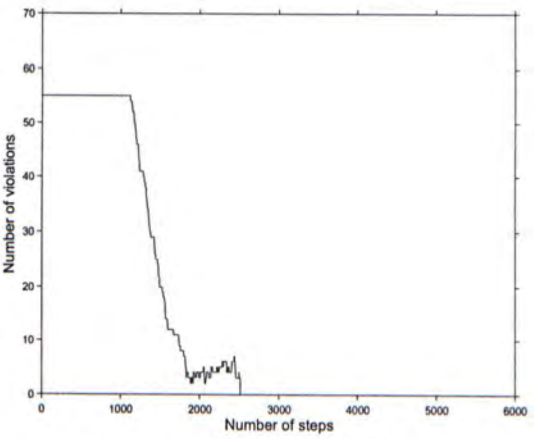
Figure 4.73: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problem with $N = 35$ (short run-time case)



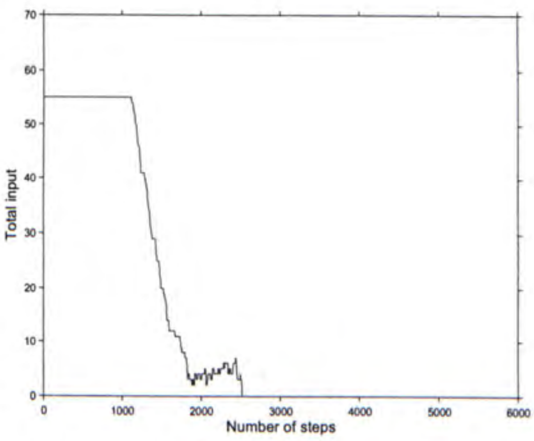
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

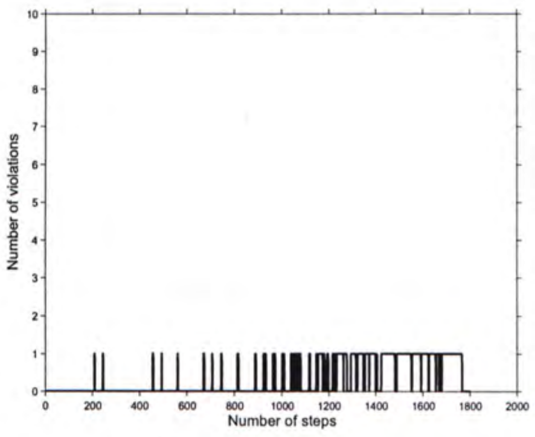


(c) max-PSS: Violation vs. Step

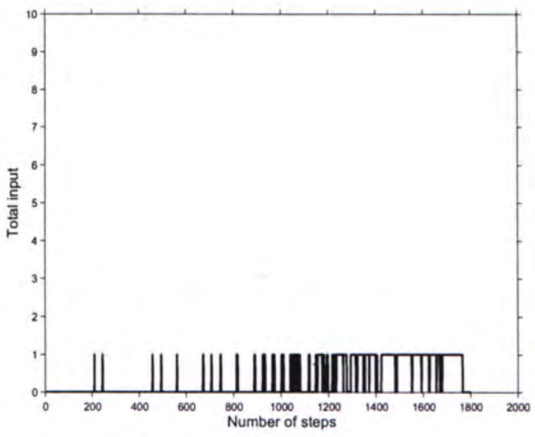


(d) max-PSS: Total input vs. Step

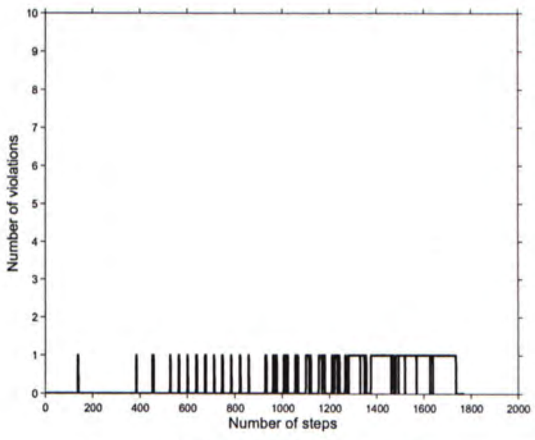
Figure 4.74: Numbers of violations and total inputs in each step of PSS and max-PSS on Latin square problem with $N = 35$ (long run-time case)



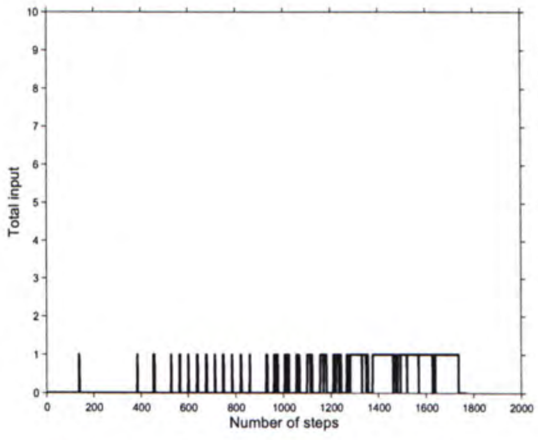
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

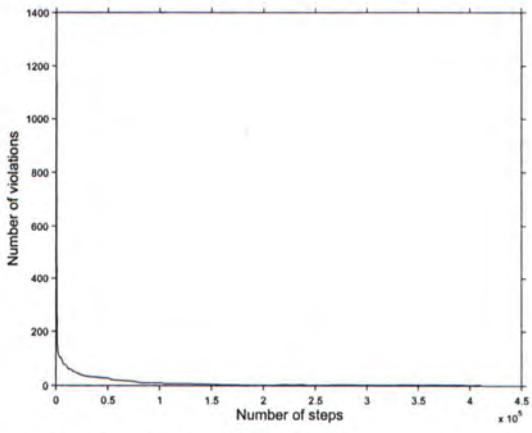


(c) max-IPSS: Violation vs. Step

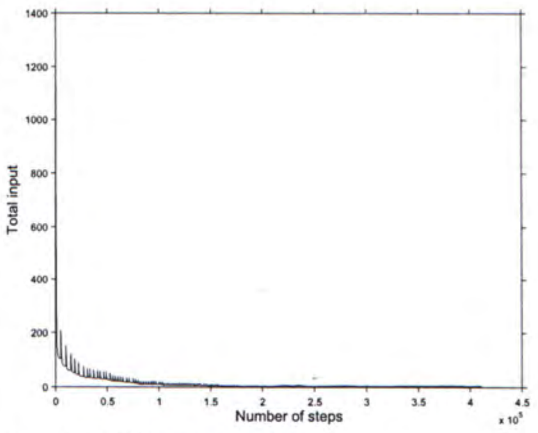


(d) max-IPSS: Total input vs. Step

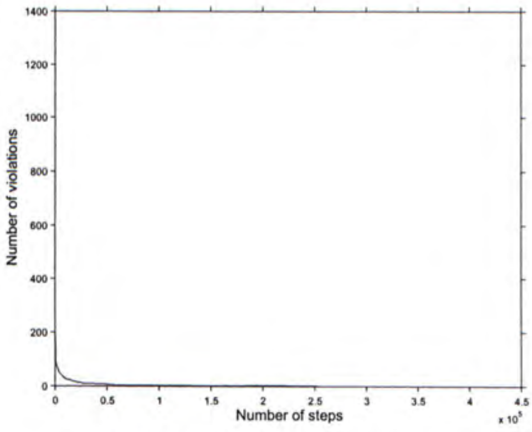
Figure 4.75: Numbers of violations and total inputs in each step of IPSS and max-IPSS on Latin square problem with $N = 35$ (long run-time case)



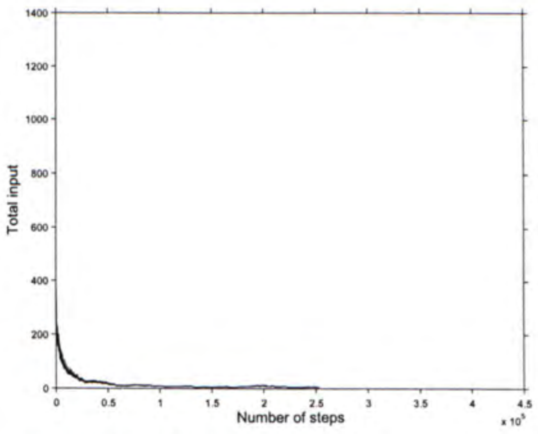
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step



(c) $\mathcal{LSDL}$(IMP): Violation vs. Step



(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

Figure 4.76: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on Latin square problem with $N = 35$ (long run-time case)

4.3.2 Quasigroup Completion Problems

The instances of QCP used in this set of experiments are randomly generated instances used in [4], which are believed to be in phase transition state, *i.e.*, roughly around 42% of the cells have pre-assigned values [3].

Table 4.9 shows the results of PSS and its variants on the set of QCPs. We give the results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) of the same problems in Table 4.10 for comparison. From the tables, the timing results indicate that the performance of PSS and its variants are not as good as $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in QCPs. The mean timing results of solving QCPs are shown in Figure 4.77.

Problem	PSS		IPSS	
N	Steps $\times 10^3$	**CPU time**	Steps $\times 10^3$	**CPU time**
15	41.07(40.19)	**0.1949(0.1900)**	66.42(68.35)	**0.2871(0.2900)**
16	68.65(67.60)	**0.3795(0.3750)**	91.07(92.92)	**0.4716(0.4800)**
17	113.6(111.9)	**0.7173(0.7100)**	178.4(184.9)	**1.0432(1.0800)**
18	166.5(166.9)	**1.1692(1.1750)**	136.5(132.4)	**0.8842(0.8600)**
19	302.1(301.2)	**2.3302(2.3400)**	507.6(486.6)	**3.6415(3.5350)**
20	426.1(431.3)	**3.7098(3.7550)**	497.8(513.6)	**4.0663(4.1900)**
	max-PSS		max-IPSS	
15	10.44(0.954)	**0.0657(0.0600)**	24.68(25.54)	**0.1040(0.1100)**
16	18.68(16.79)	**0.1225(0.1200)**	27.68(26.48)	**0.1366(0.1300)**
17	26.10(26.01)	**0.1857(0.1850)**	68.55(65.46)	**0.3436(0.3300)**
18	26.47(35.50)	**0.2787(0.2800)**	46.98(45.88)	**0.2788(0.2800)**
19	59.64(58.68)	**0.4669(0.4650)**	125.1(120.4)	**0.7454(0.7300)**
20	60.71(59.60)	**0.5447(0.5400)**	98.54(99.57)	**0.6913(0.7000)**

Table 4.9: PSS and its variants on quasigroup completion problems

Figures 4.78 - 4.86 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 15. Figures 4.87 - 4.95 show the numbers of violations against total inputs or objective values of PSS, max-PSS, IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 16. From the figures about $\mathcal{LSDL}$(GENET), we conclude that there exist many local minima in the search space. $\mathcal{LSDL}$(GENET) does learning a lot of times to

Problem	$\mathcal{LSDL}$(GENET)			
N	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
15	1.893(1.926)	5.351(5.598)	0.743(0.751)	**0.0366(0.0400)**
16	1.549(1.255)	4.714(4.162)	0.595(0.465)	**0.0366(0.0300)**
17	3.224(3.058)	9.435(9.177)	1.256(1.175)	**0.0759(0.0800)**
18	3.464(3.534)	10.62(10.59)	1.332(1.364)	**0.0955(0.0900)**
19	5.438(5.674)	17.26(18.09)	2.075(2.157)	**0.1675(0.1700)**
20	5.323(4.799)	18.26(17.56)	1.998(1.772)	**0.1979(0.1900)**
Problem	$\mathcal{LSDL}$(IMP)			
N	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
15	0.342(0.416)	2.124(2.390)	0.342(0.416)	**0.0131(0.0100)**
16	0.642(0.763)	3.287(3.711)	0.642(0.763)	**0.0199(0.0200)**
17	1.369(1.002)	7.162(6.072)	1.369(1.002)	**0.0430(0.0400)**
18	1.256(1.011)	6.896(6.780)	1.256(1.011)	**0.0459(0.0450)**
19	1.165(0.658)	7.302(4.656)	1.165(0.658)	**0.0555(0.0300)**
20	1.333(1.443)	8.516(9.166)	1.333(1.443)	**0.0736(0.0800)**

Table 4.10: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on quasigroup completion problems

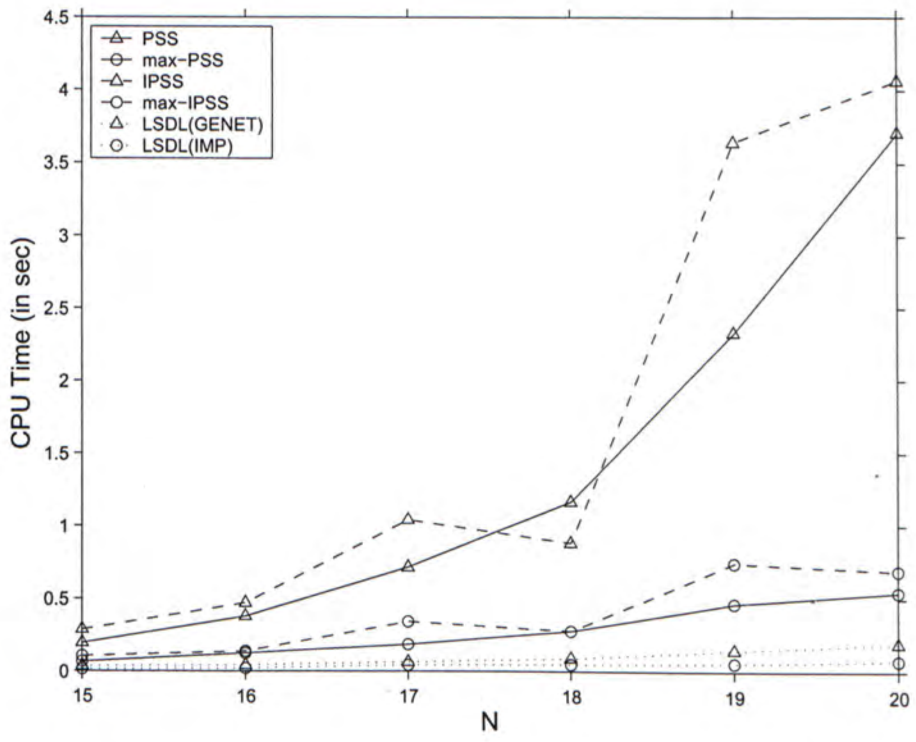


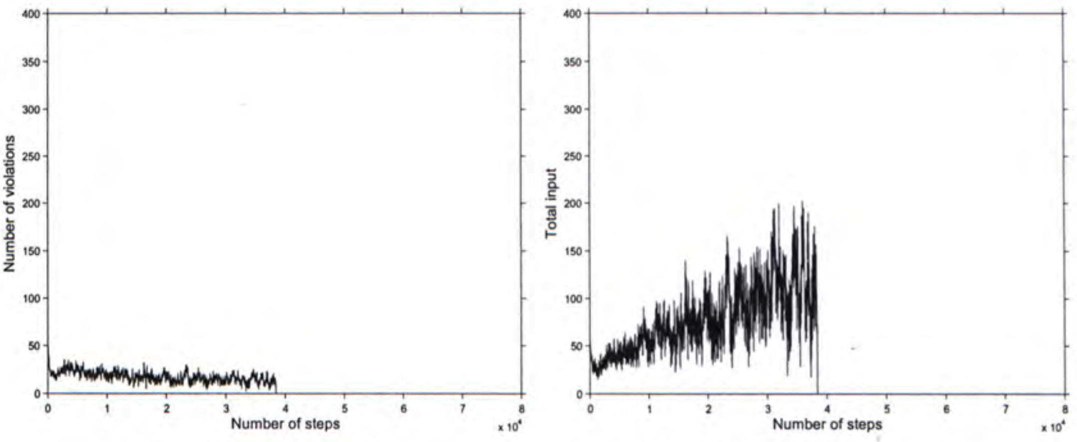
Figure 4.77: The mean time results on quasigroup completion problems

escape from local minima. From Table 4.9 and 4.10, we see that the number of repairs done and steps taken in $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) are fewer than the number of steps taken in PSS and its variants. It means that the search path of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) are shorter than those of PSS and its variants. During the search, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) select a direction that globally improves the current state, while PSS and IPSS select a direction that is dictated by the list $\mathcal{F}$. The ordering in $\mathcal{F}$ is defined by the search dynamically. Therefore, PSS, IPSS and $\mathcal{LSDL}$ implementations have totally different search paths in solving QCPs. The experimental results show that the search strategy of PSS and IPSS are not as effective as that of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in this set of experiments.

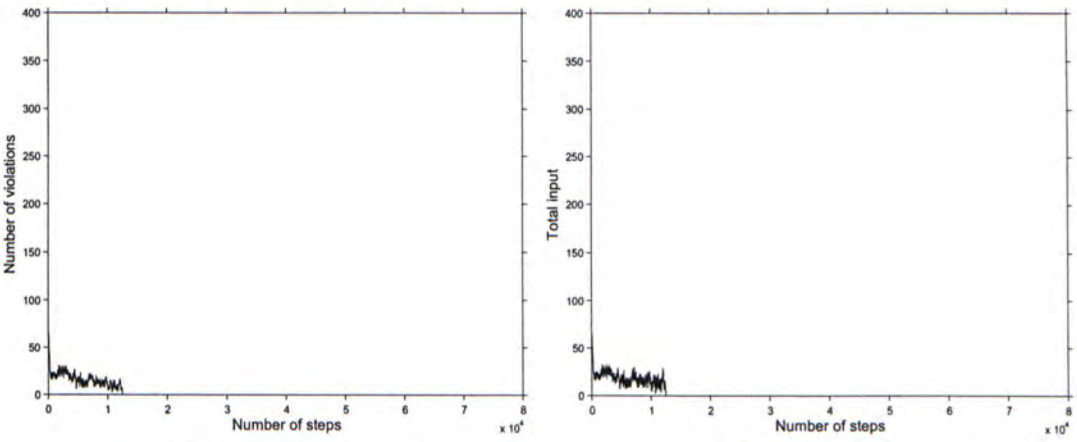
In general, PSS takes fewer steps than IPSS in solving QCPs. From Figures 4.79(a), 4.82(a), 4.85(a), 4.88(a), 4.91(a) and 4.94(a), we conclude that the partial solutions are not easy to extend. IPSS takes more steps to find the next partial solution. Therefore, PSS has a better performance of IPSS in this set of experiments. It should be noted that max-PSS and max-IPSS are shown to have a great improvement on PSS and IPSS respectively. The timing results confirm that the heuristic guides the search to select a relatively better direction in the search space.

4.4 Random CSPs

A random binary CSP is generated with four parameters (n, m, p_1, p_2), where n is the number of variables, m is the domain size of the variables, p_1 is the constraint density, and p_2 is the constraint tightness. Constraint density is the probability that a constraint exists between a pair of variables. Constraint tightness is the probability that a pair of values is incompatible with each other for a given pair of variables that is being constrained.

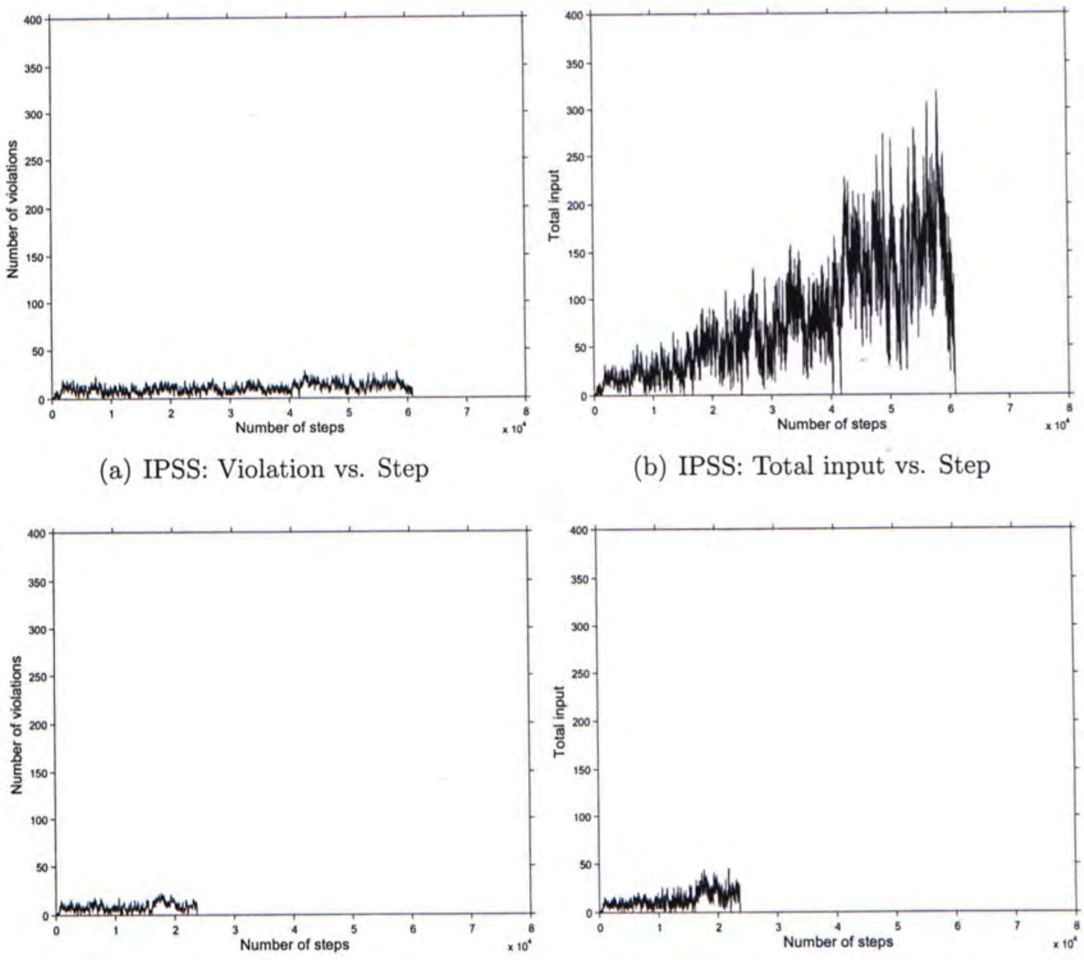


(a) PSS: Violation vs. Step (b) PSS: Total input vs. Step



(c) max-PSS: Violation vs. Step (d) max-PSS: Total input vs. Step

Figure 4.78: Numbers of violations and total inputs in each step of PSS and max-PSS on QCP of order 15 (average run-time case)



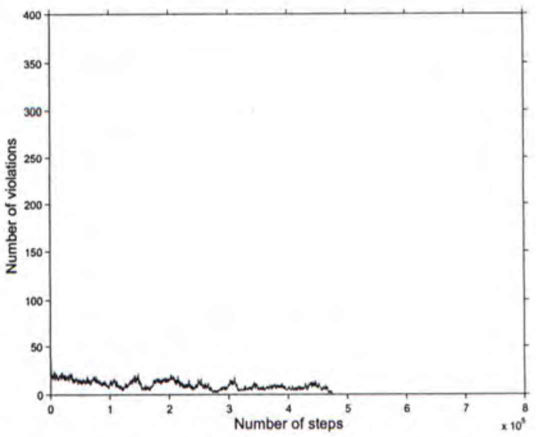
(a) IPSS: Violation vs. Step

(b) IPSS: Total input vs. Step

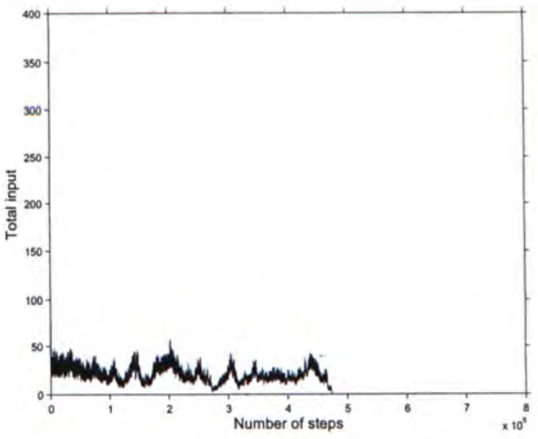
(c) max-IPSS: Violation vs. Step

(d) max-IPSS: Total input vs. Step

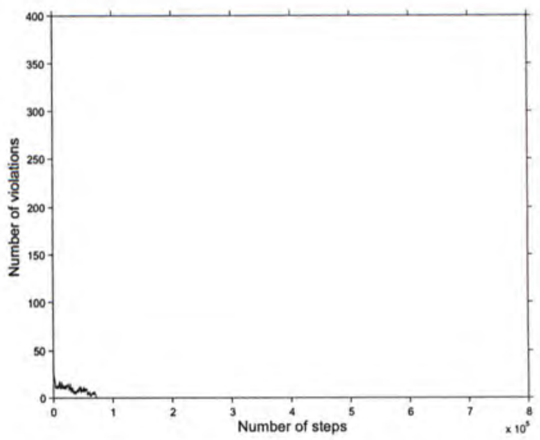
Figure 4.79: Numbers of violations and total inputs in each step of IPSS and max-IPSS on QCP of order 15 (average run-time case)



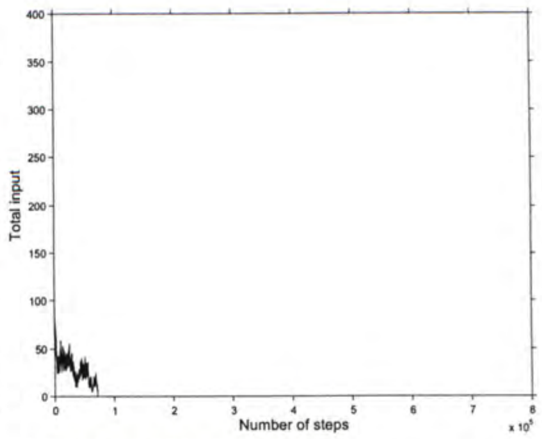
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

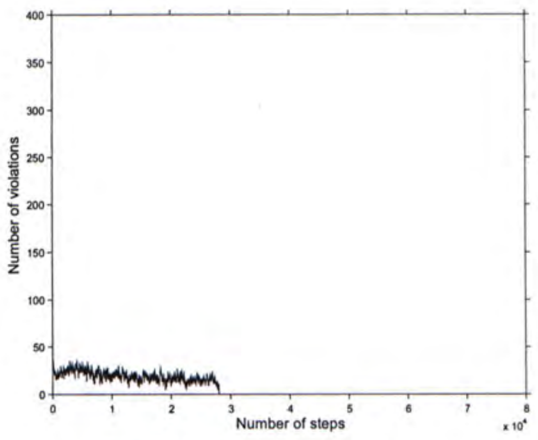


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

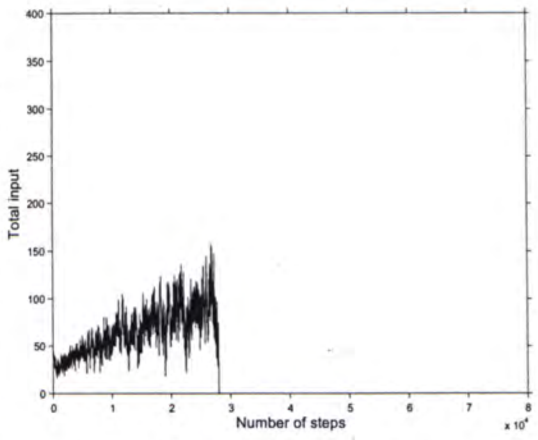


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

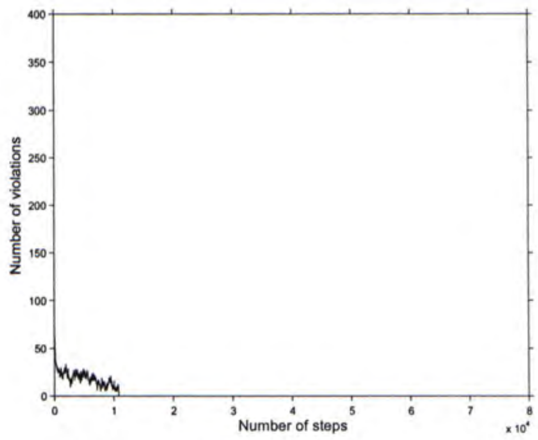
Figure 4.80: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 15 (average run-time case)



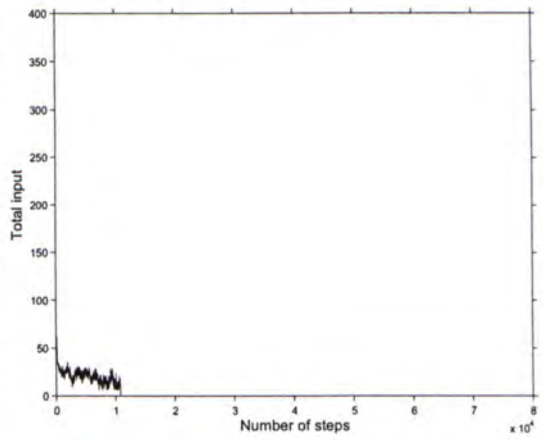
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

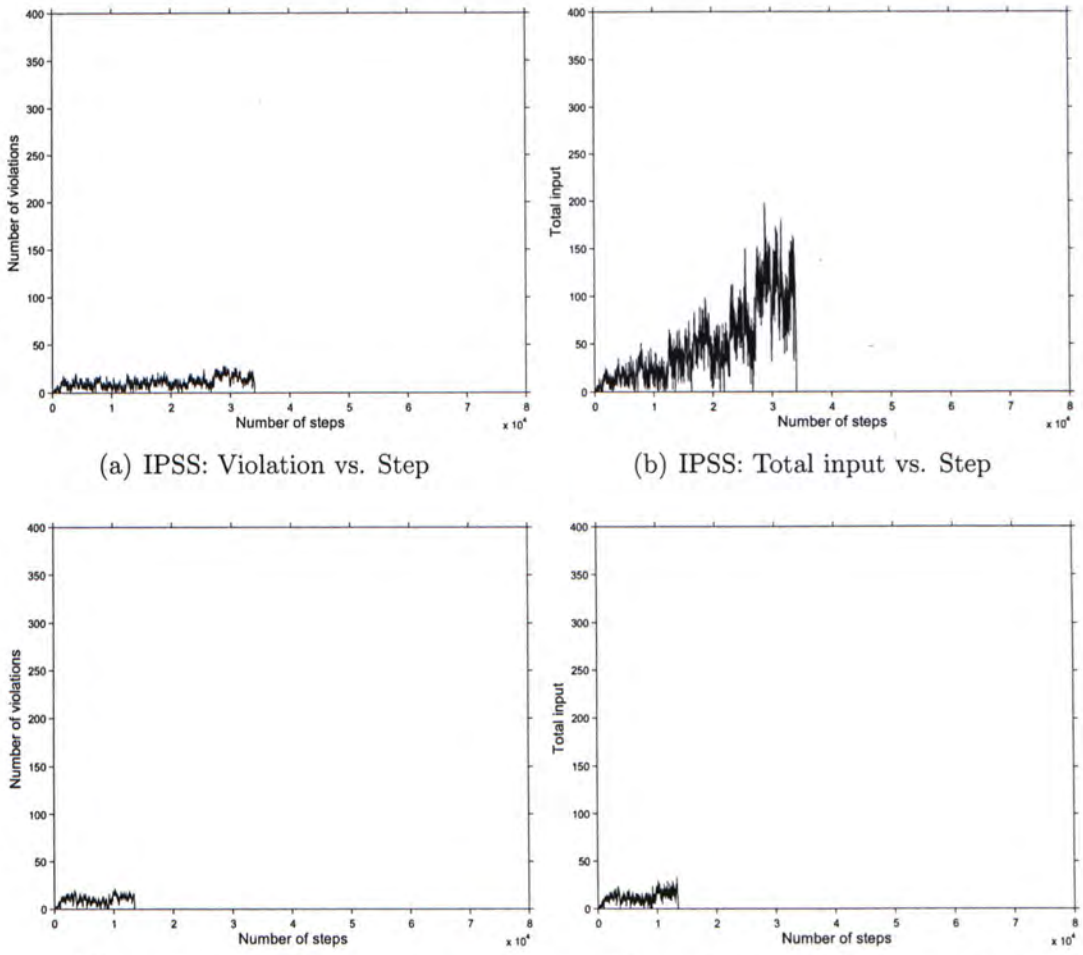


(c) max-PSS: Violation vs. Step



(d) max-PSS: Total input vs. Step

Figure 4.81: Numbers of violations and total inputs in each step of PSS and max-PSS on QCP of order 15 (short run-time case)



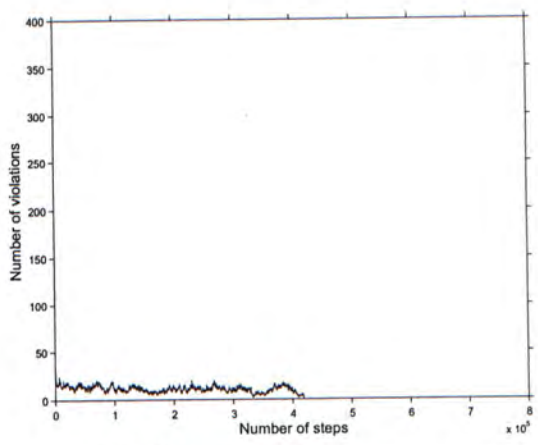
(a) IPSS: Violation vs. Step

(b) IPSS: Total input vs. Step

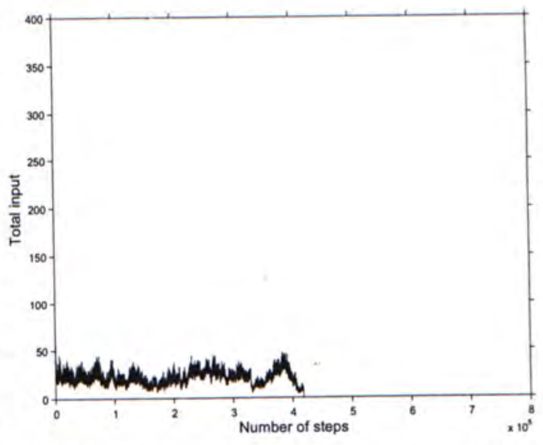
(c) max-IPSS: Violation vs. Step

(d) max-IPSS: Total input vs. Step

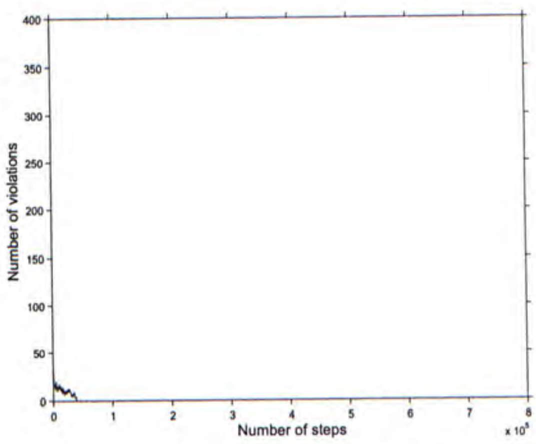
Figure 4.82: Numbers of violations and total inputs in each step of IPSS and max-IPSS on QCP of order 15 (short run-time case)



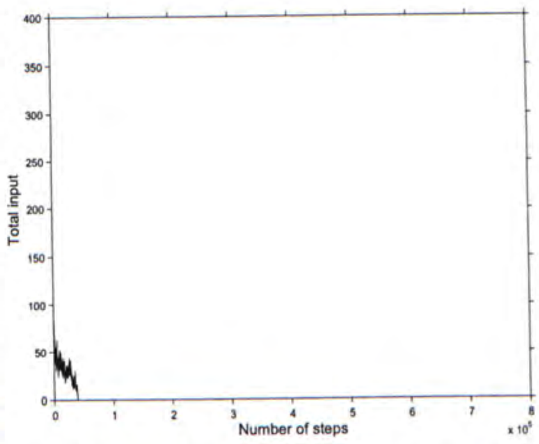
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

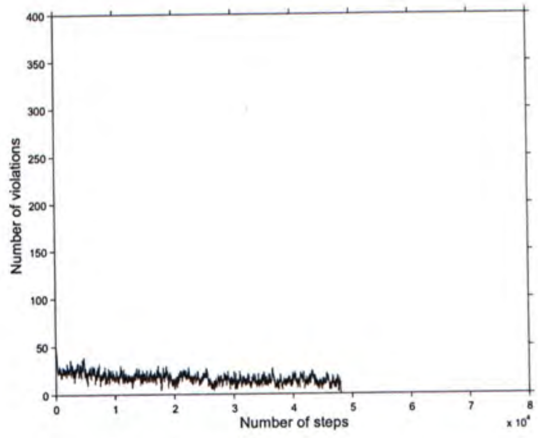


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

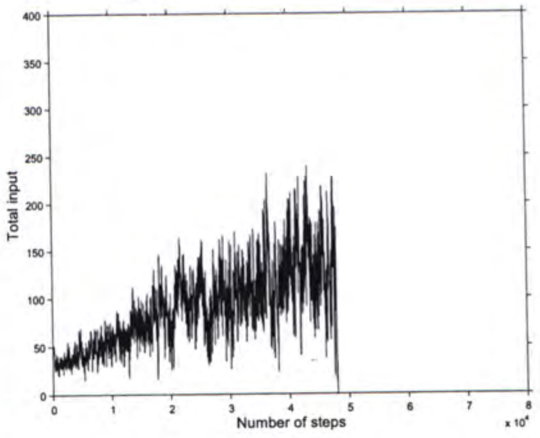


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

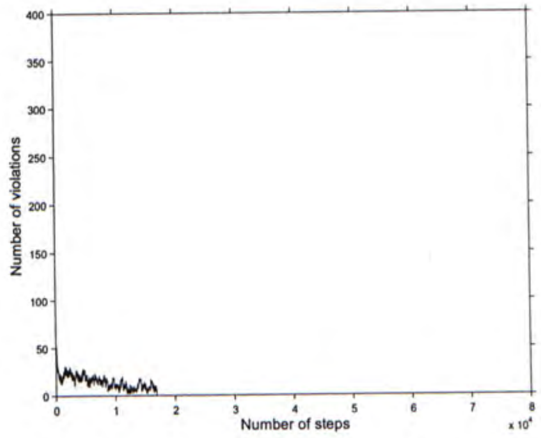
Figure 4.83: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 15 (short run-time case)



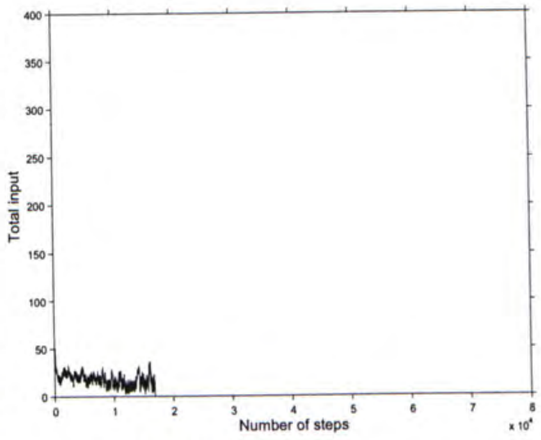
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

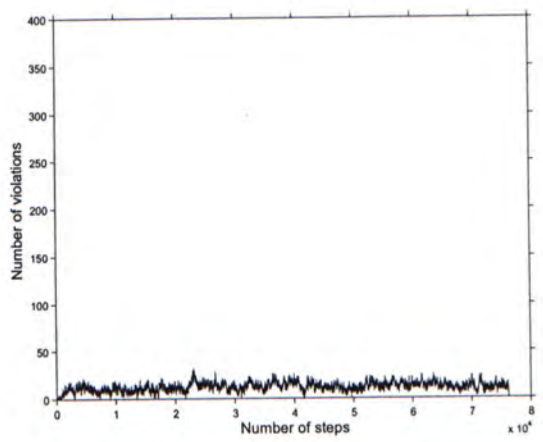


(c) max-PSS: Violation vs. Step

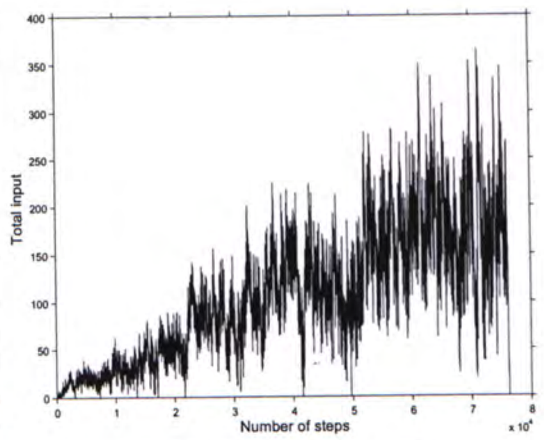


(d) IPSS: Total input vs. Step

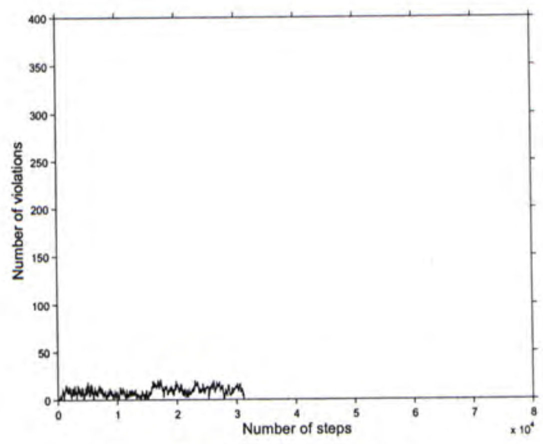
Figure 4.84: Numbers of violations and total inputs in each step of PSS and max-PSS on QCP of order 15 (long run-time case)



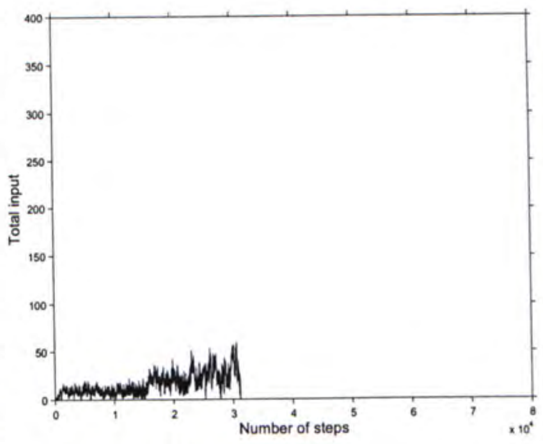
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

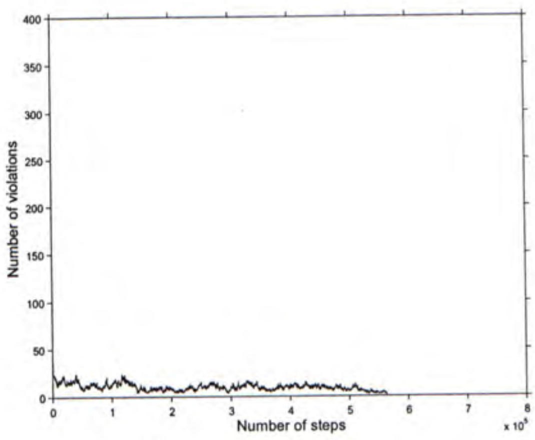


(c) max-IPSS: Violation vs. Step

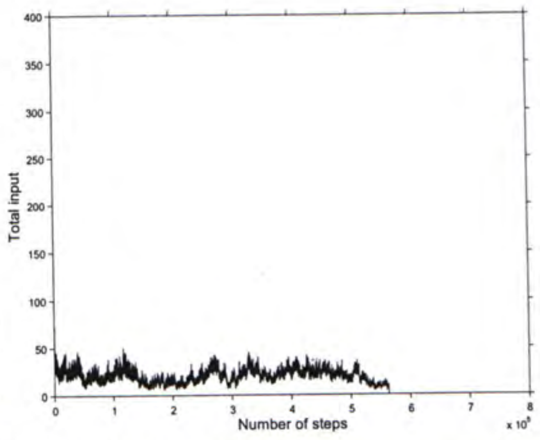


(d) max-IPSS: Total input vs. Step

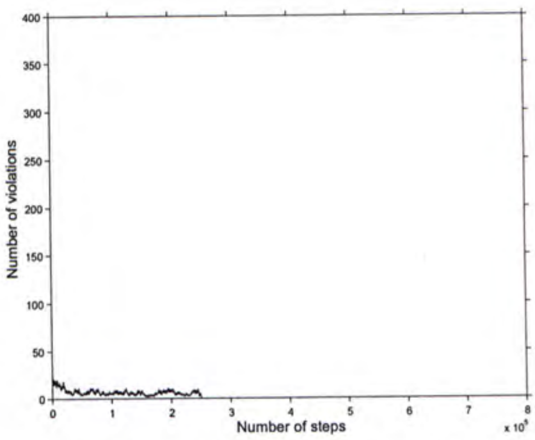
Figure 4.85: Numbers of violations and total inputs in each step of IPSS and max-IPSS on QCP of order 15 (long run-time case)



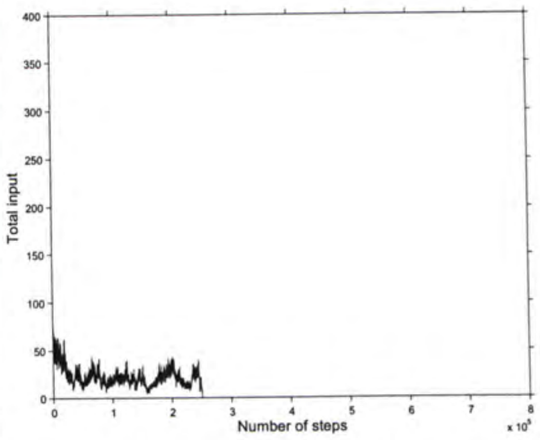
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

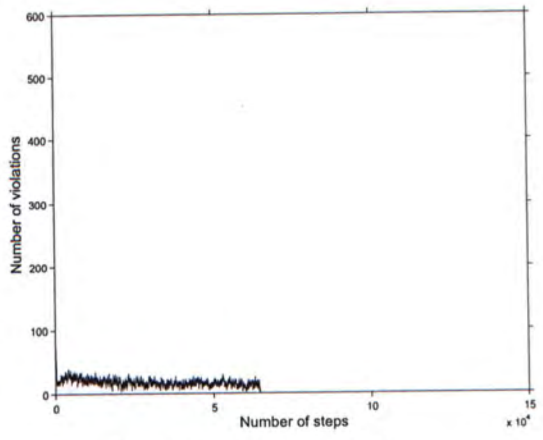


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

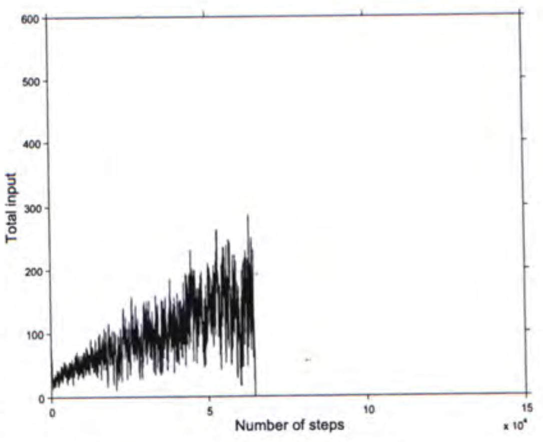


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

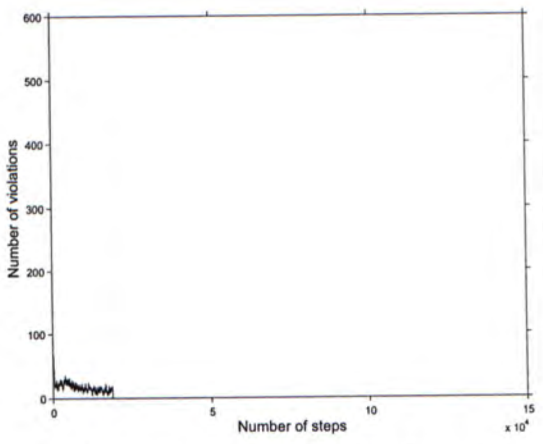
Figure 4.86: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 15 (long run-time case)



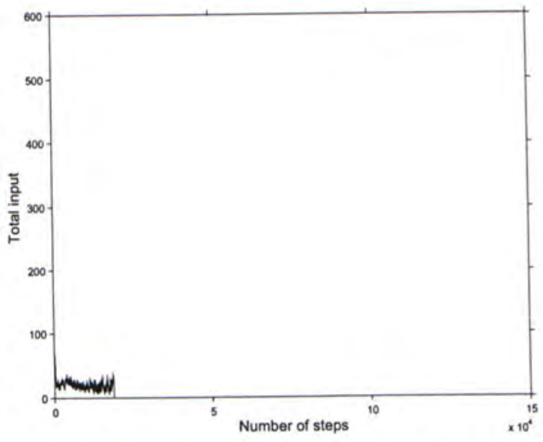
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

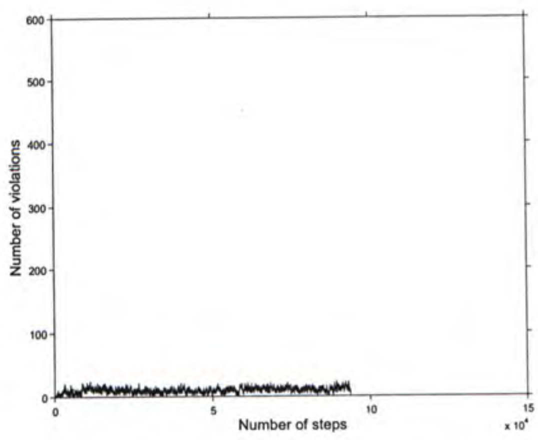


(c) max-PSS: Violation vs. Step

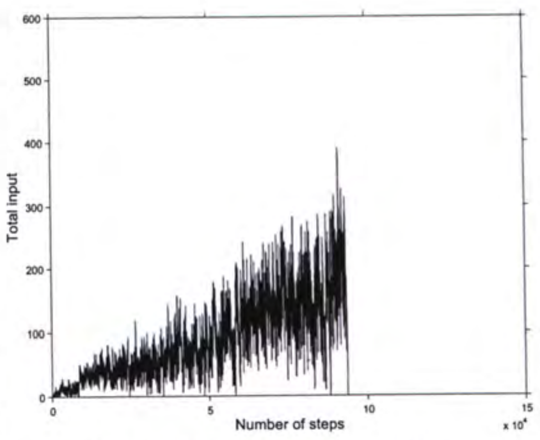


(d) max-PSS: Total input vs. Step

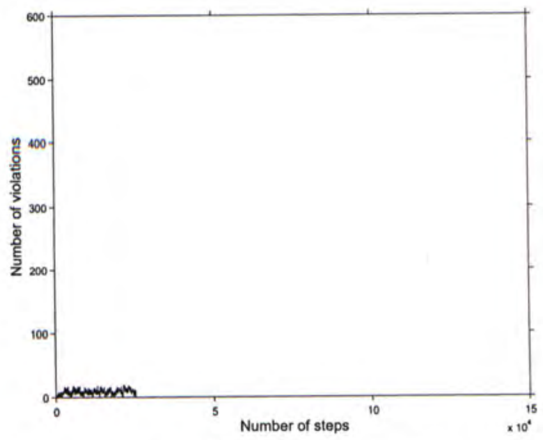
Figure 4.87: Numbers of violations and total inputs in each step of PSS and max-PSS on QCP of order 16 (average run-time case)



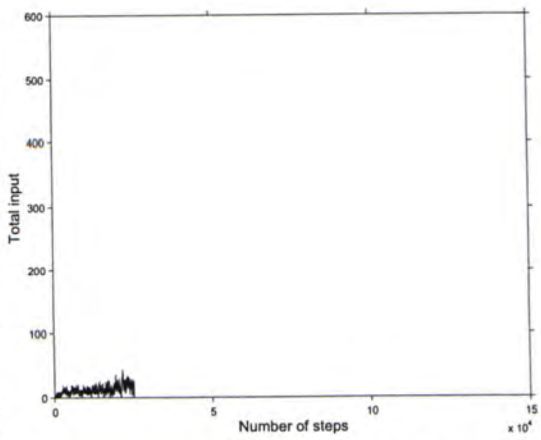
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

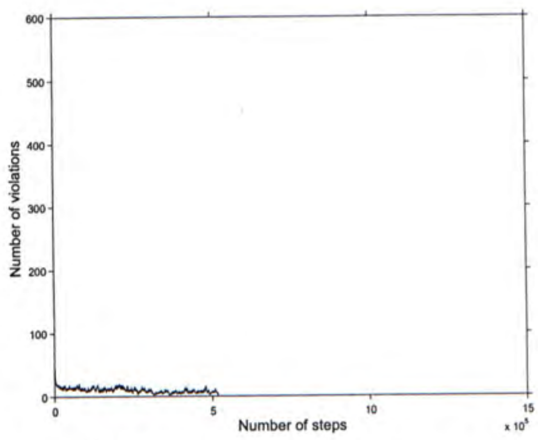


(c) max-IPSS: Violation vs. Step

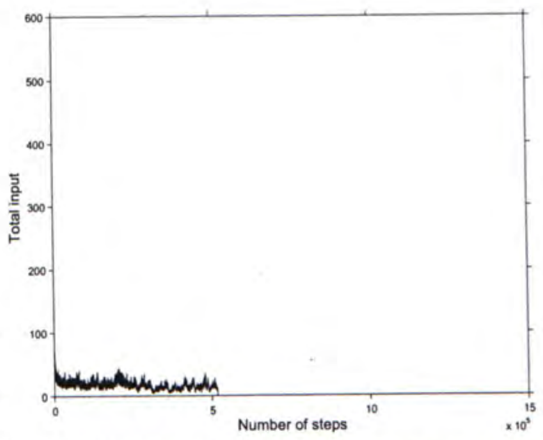


(d) max-IPSS: Total input vs. Step

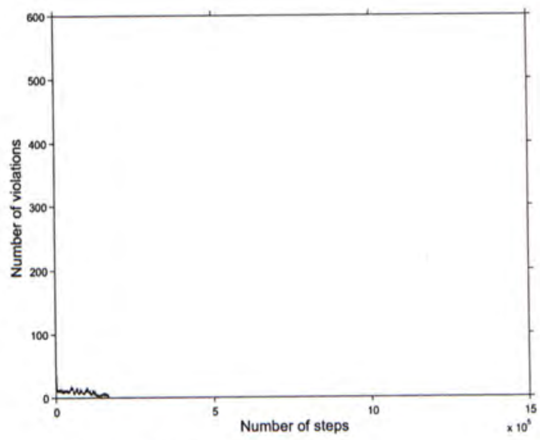
Figure 4.88: Numbers of violations and total inputs in each step of IPSS and max-IPSS on QCP of order 16 (average run-time case)



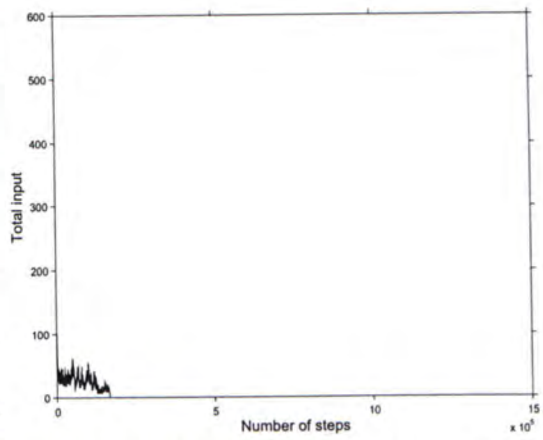
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

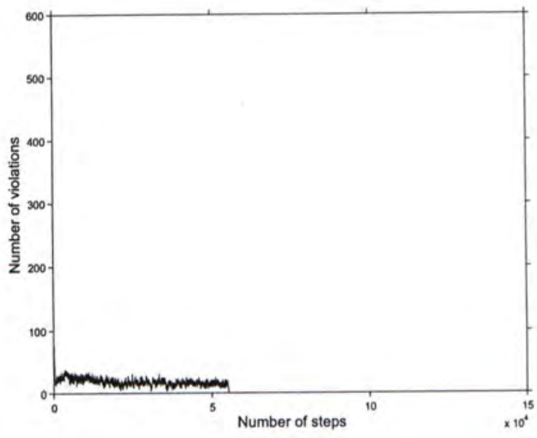


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

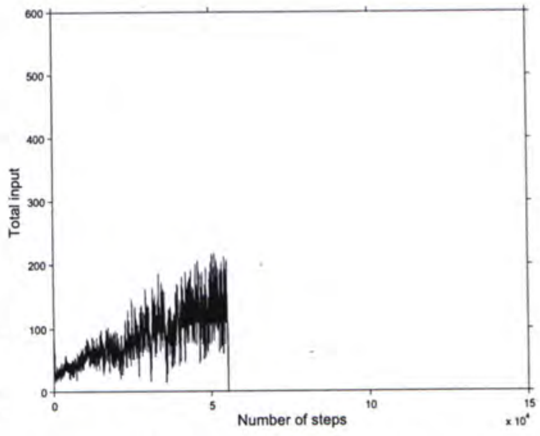


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

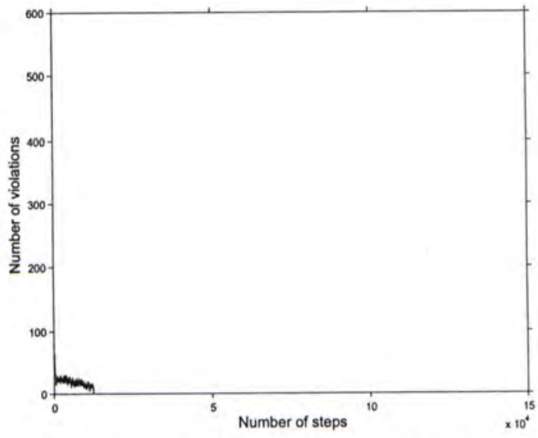
Figure 4.89: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 16 (average run-time case)



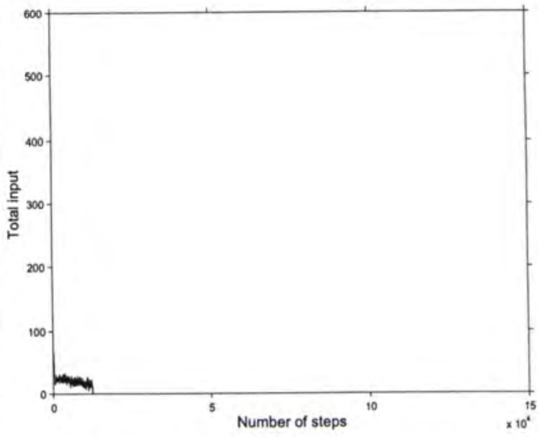
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

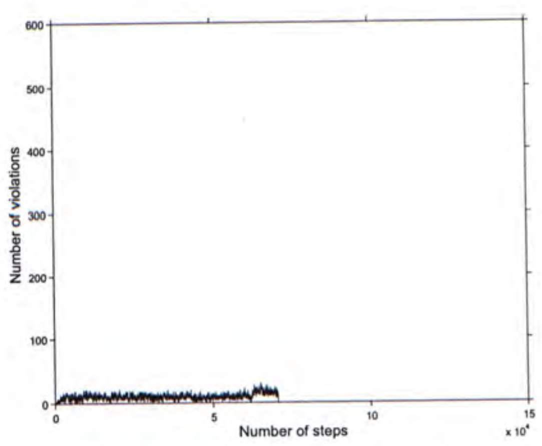


(c) max-PSS: Violation vs. Step

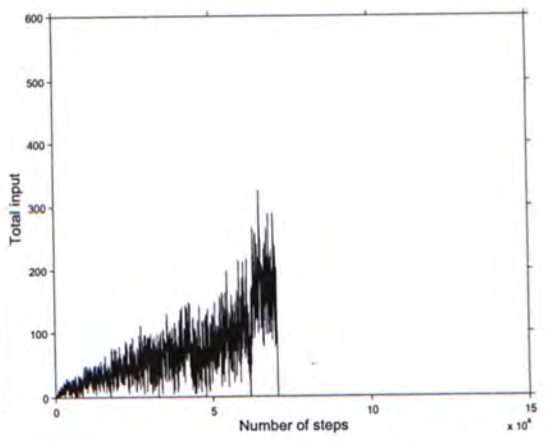


(d) max-PSS: Total input vs. Step

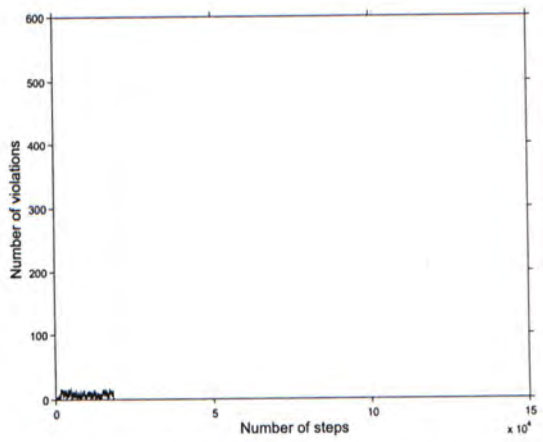
Figure 4.90: Numbers of violations and total inputs in each step of PSS and max-PSS on QCP of order 16 (short run-time case)



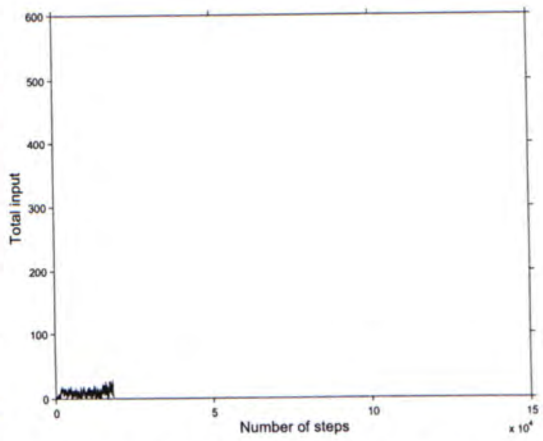
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

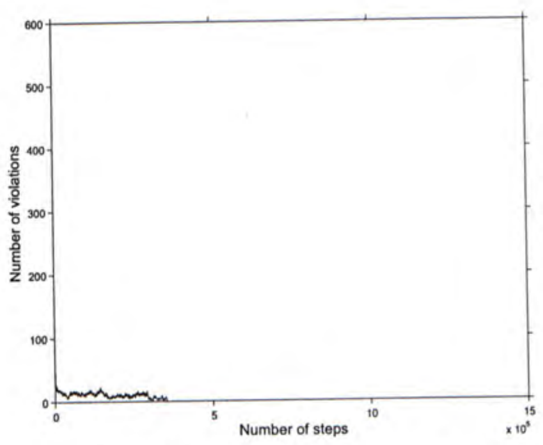


(c) max-IPSS: Violation vs. Step

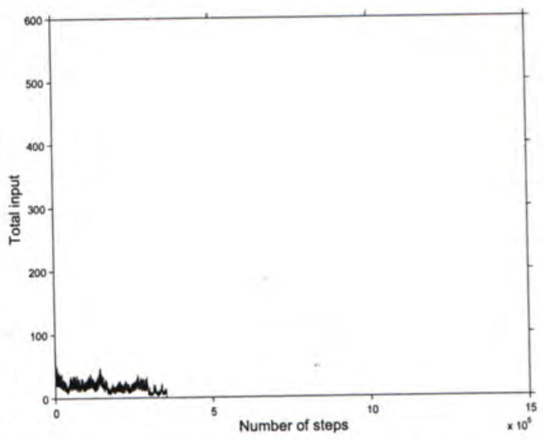


(d) max-IPSS: Total input vs. Step

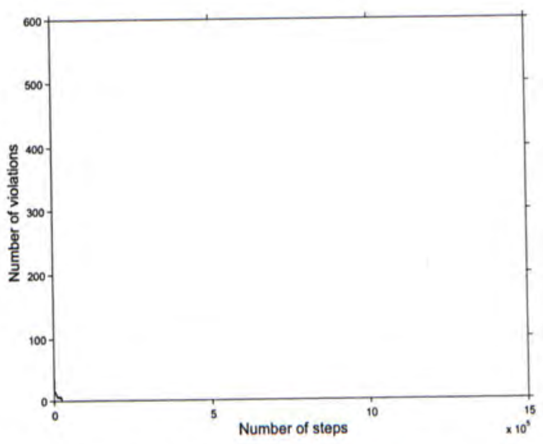
Figure 4.91: Numbers of violations and total inputs in each step of IPSS and max-IPSS on QCP of order 16 (short run-time case)



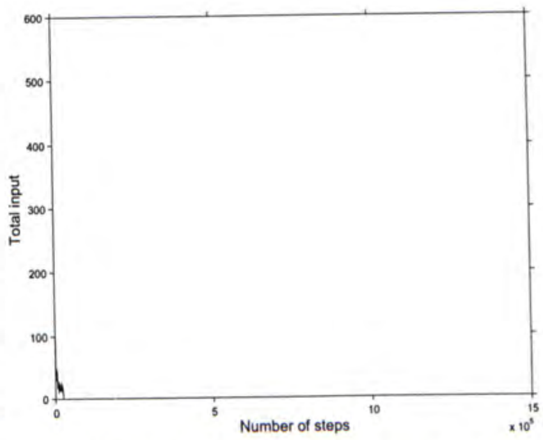
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

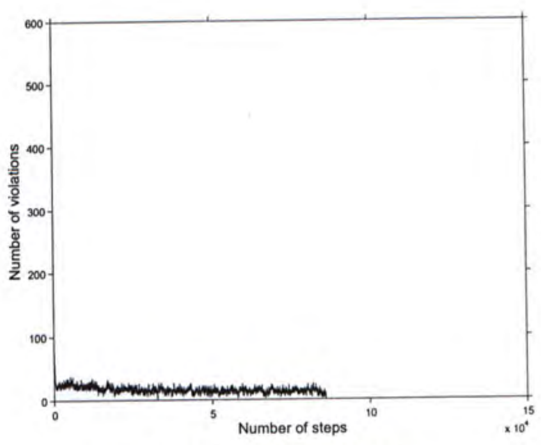


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

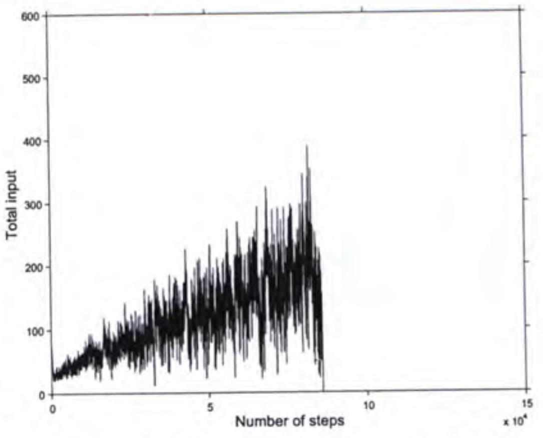


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

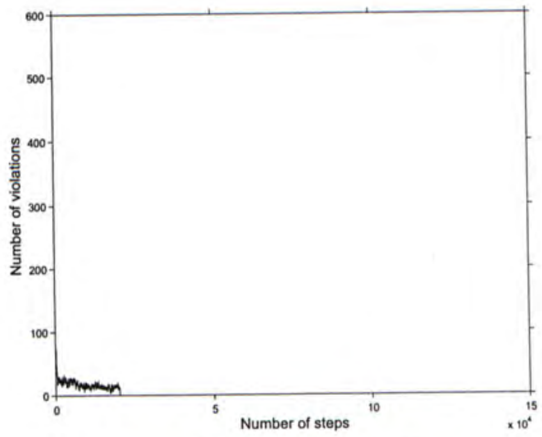
Figure 4.92: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 16 (short run-time case)



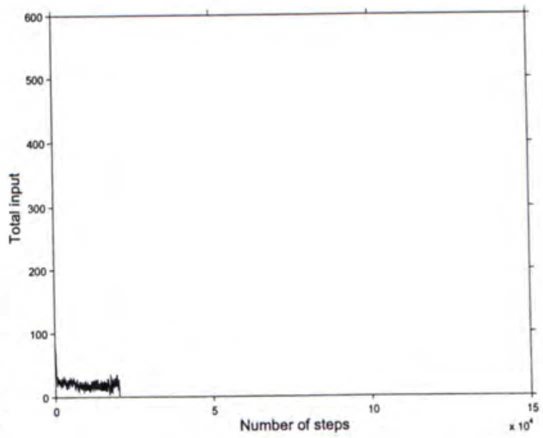
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

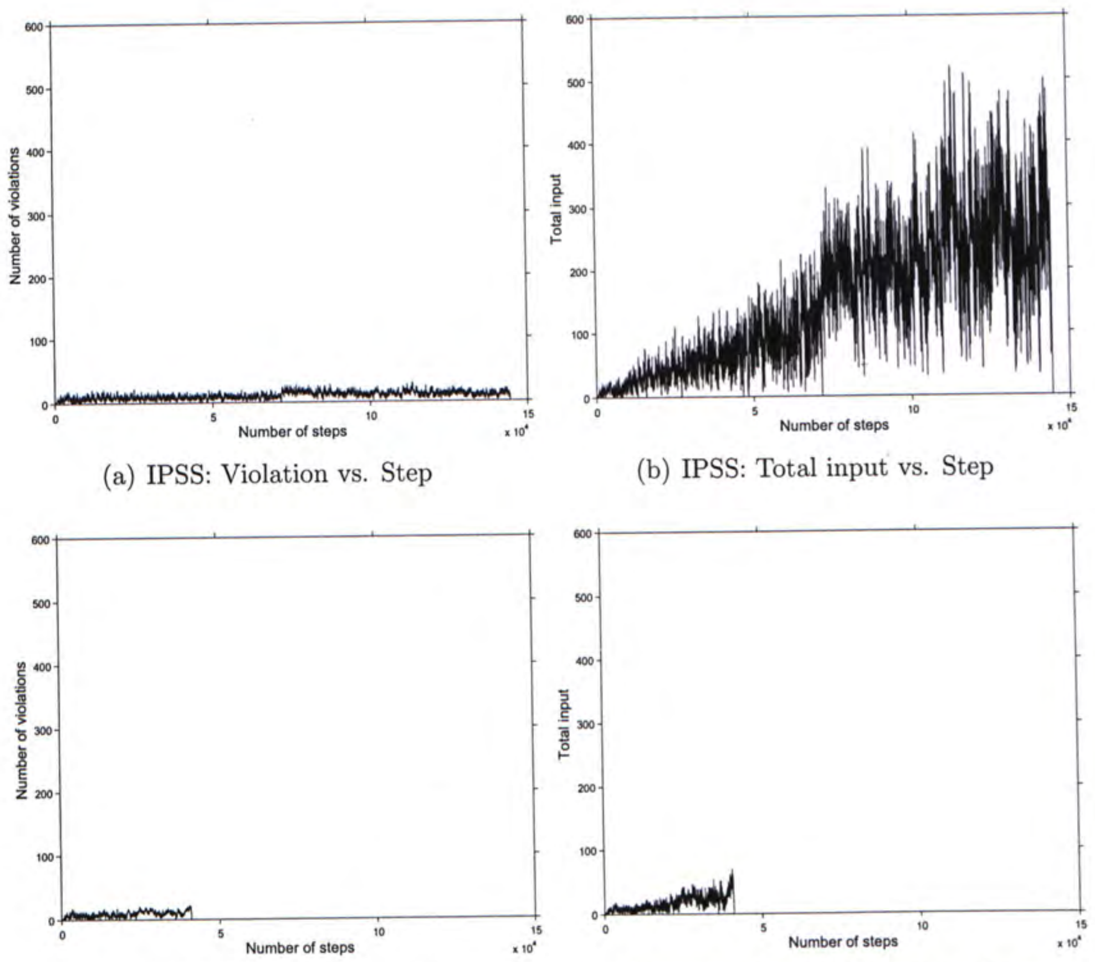


(c) max-PSS: Violation vs. Step



(d) max-PSS: Total input vs. Step

Figure 4.93: Numbers of violations and total inputs in each step of PSS and max-PSS on QCP of order 16 (long run-time case)



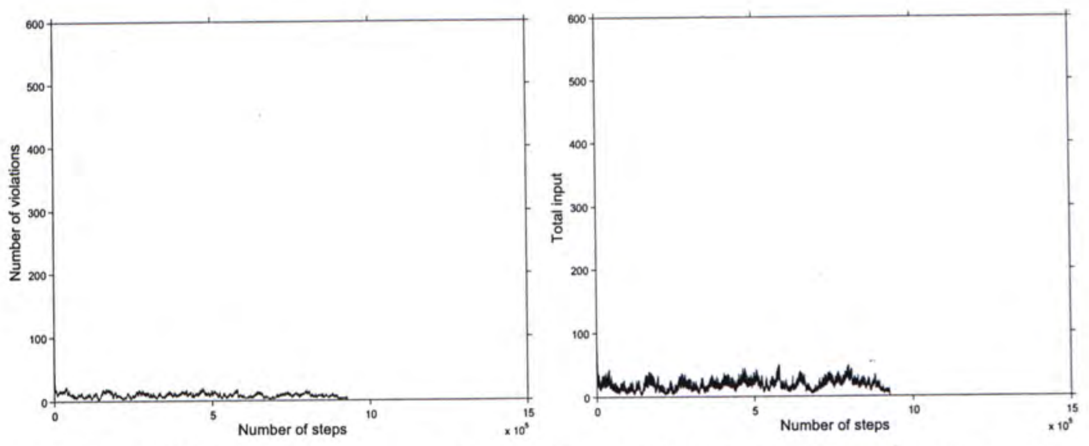
(a) IPSS: Violation vs. Step

(b) IPSS: Total input vs. Step

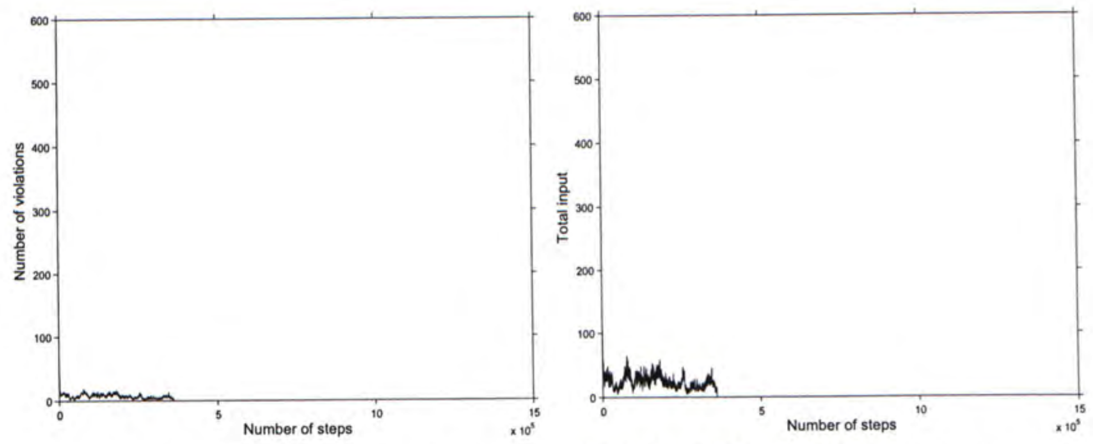
(c) max-IPSS: Violation vs. Step

(d) max-IPSS: Total input vs. Step

Figure 4.94: Numbers of violations and total inputs in each step of IPSS and max-IPSS on QCP of order 16 (long run-time case)



(a) $\mathcal{LSDL}$(GENET): Violation vs. Step (b) $\mathcal{LSDL}$(GENET): Objective value vs. Step



(c) $\mathcal{LSDL}$(IMP): Violation vs. Step (d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

Figure 4.95: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on QCP of order 16 (long run-time case)

4.4.1 Tight Random CSPs

A set of random binary CSPs with n ranging from 120 to 170, $m = 10$, $p_1 = 0.6$ and $p_2 = 0.75$ are used in this set of experiments. The execution limits of PSS and its variants in solving the problem instances are set to 5 million steps. The execution limits of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in solving the problem instances are set to 5 million iterations. We use a superscript $(x/100)$ besides the timing figures to indicate that only x out of the hundred runs are successful.

Problem	PSS		IPSS	
n	Steps	**CPU time**	Steps	**CPU time**
120	3.0(3.0)	**0.0100(0.0100)**$^{(04/100)}$	182.9(159.0)	**0.0159(0.0200)**
130	0.0(0.0)	**0.0100(0.0100)**$^{(14/100)}$	155.5(143.5)	**0.0150(0.0100)**
140	4.0(4.0)	**0.0200(0.0200)**$^{(06/100)}$	225.2(181.0)	**0.0241(0.0200)**
150	0.0(0.0)	**0.0150(0.0150)**$^{(10/100)}$	340.2(294.5)	**0.0398(0.0300)**
160	1.4(0.0)	**0.0200(0.0200)**$^{(11/100)}$	592.7(484.0)	**0.0786(0.0600)**
170	8.0(8.0)	**0.0200(0.0200)**$^{(01/100)}$	217.8(192.0)	**0.0281(0.0300)**
	max-PSS		max-IPSS	
120	80.0(76.5)	**0.0112(0.0100)**$^{(42/100)}$	185.8(168.0)	**0.0143(0.0100)**
130	70.1(89.0)	**0.0144(0.0100)**$^{(43/100)}$	160.9(154.0)	**0.0131(0.0100)**
140	97.2(93.0)	**0.0155(0.0100)**$^{(22/100)}$	213.8(182.5)	**0.0201(0.0200)**
150	73.0(89.0)	**0.0176(0.0200)**$^{(21/100)}$	211.4(186.0)	**0.0213(0.0200)**
160	85.3(88.0)	**0.0205(0.0200)**$^{(37/100)}$	340.9(351.5)	**0.0370(0.0400)**
170	110.4(103.5)	**0.0225(0.0200)**$^{(20/100)}$	205.2(192.5)	**0.0250(0.0200)**

Table 4.11: PSS and its variants on random CSPs

Table 4.11 shows results of PSS and its variants on random CSPs. The results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on the same set of problems are given for comparison in Table 4.12. From the tables, we observe that PSS and max-PSS cannot always find solutions within the pre-set limit, and IPSS and max-IPSS have a better performance than others.

The random CSPs with the above parameters are likely to have many flawed values [1]. We record the numbers of violations against total inputs or objective values of IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 120$ in Figures 4.96 - 4.101. The numbers of violations

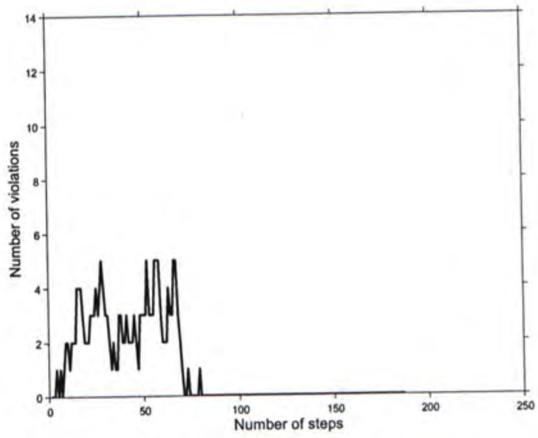
Problem	$\mathcal{LSDL}$(GENET)			
n	Iteration	Repairs	Learns	**CPU time**
120	126.6(147.0)	3084(3500)	15.8(18.0)	**0.4620(0.5200)**
130	136.8(153.0)	3513(3980)	16.4(18.5)	**0.5718(0.6500)**
140	135.8(154.0)	3672(4231)	15.9(18.0)	**0.6510(0.7500)**
150	164.7(170.0)	4653(4801)	18.7(19.0)	**0.8846(0.9100)**
160	160.4(167.0)	4773(4974)	17.8(19.0)	**0.9787(1.0200)**
170	162.7(175.0)	4998(5426)	17.5(19.0)	**1.0965(1.1900)**
Problem	$\mathcal{LSDL}$(IMP)			
n	Iteration	Repairs	Learns	**CPU time**
120	27.2(30.0)	2814(3093)	27.2(30.0)	**0.4243(0.4700)**
130	26.4(30.0)	2988(3419)	26.4(30.0)	**0.4962(0.5650)**
140	24.7(30.0)	2999(3719)	24.7(30.0)	**0.5443(0.6700)**
150	27.6(32.0)	3641(4227)	27.6(32.0)	**0.7122(0.8300)**
160	27.5(32.0)	3876(4518)	27.5(32.0)	**0.8160(0.9400)**
170	29.1(32.0)	4374(4795)	29.1(32.0)	**0.9821(1.0700)**

Table 4.12: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSPs

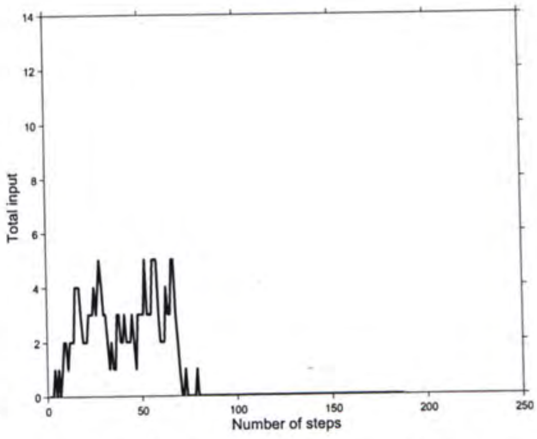
against total inputs or objective values of IPSS, max-IPSS, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 170$ are shown in Figures 4.102 - 4.107.

Figures 4.97, 4.99 and 4.101 show the number of violations in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 120$ in average run-time case, short run-time case and long-run time case respectively. Figures 4.103, 4.105 and 4.107 show the number of violations in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 170$ in average run-time case, short run-time case and long-run time case respectively. We observe that the number of violations typically maintains in a level (around several thousands), until it quickly drops to zero when a solution is found, after $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) does learning several times.

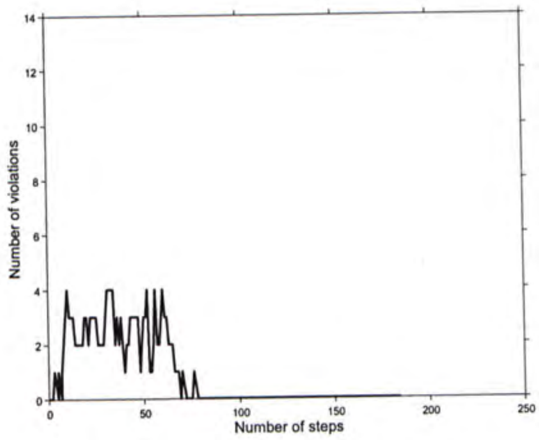
Figures 4.108(a) shows the number of violations in each step of PSS on random CSP with $n = 120$. Figures 4.109(a) shows the number of violations in each step of PSS on random CSP with $n = 170$. We see that the number of violations also typically keeps in a level. When the random CSP instance has many flawed values, PSS is not always able to find a solution like



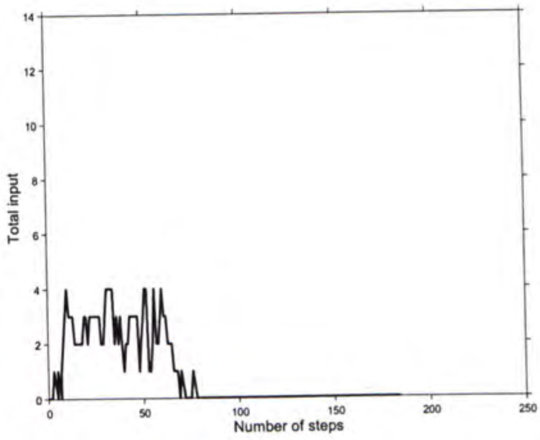
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

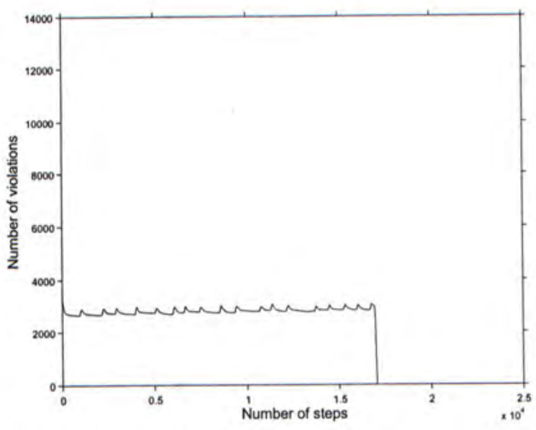


(c) max-IPSS: Violation vs. Step

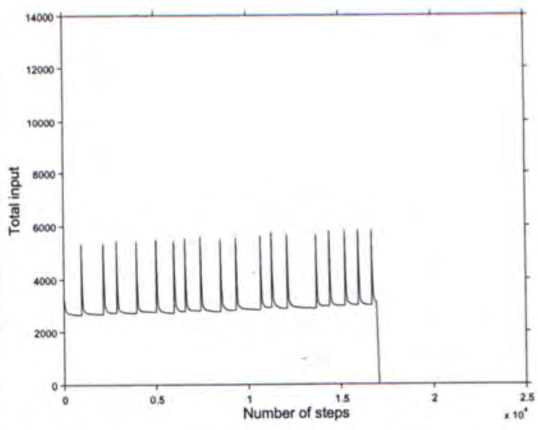


(d) max-IPSS: Total input vs. Step

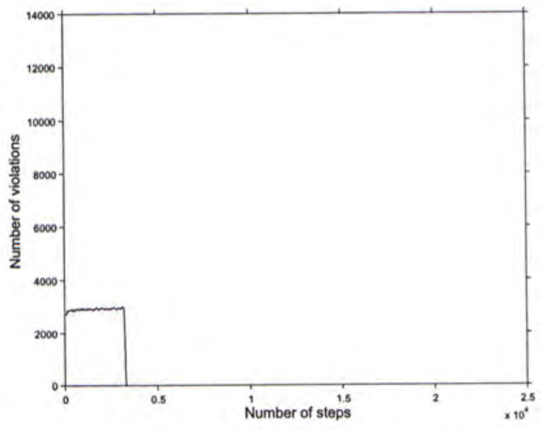
Figure 4.96: Numbers of violations and total inputs in each step of IPSS and max-IPSS on random CSP with $n = 120$ (average run-time case)



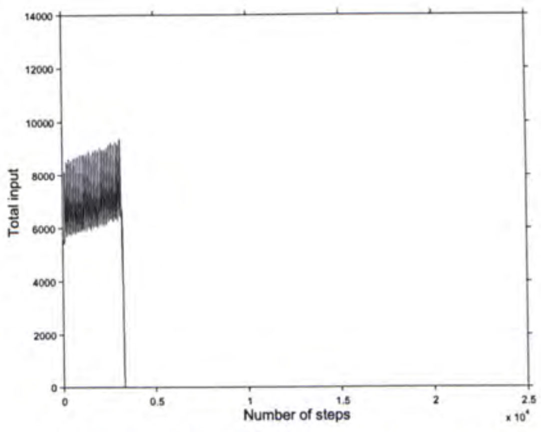
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

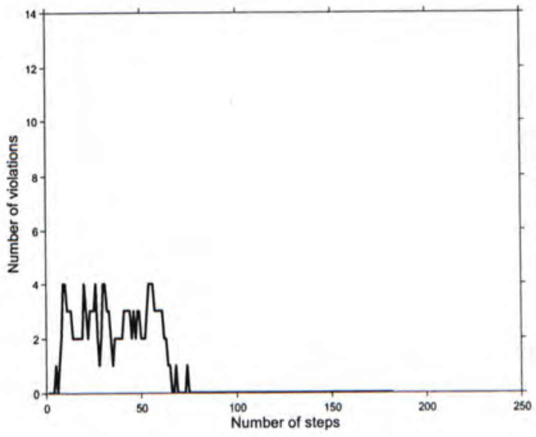


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

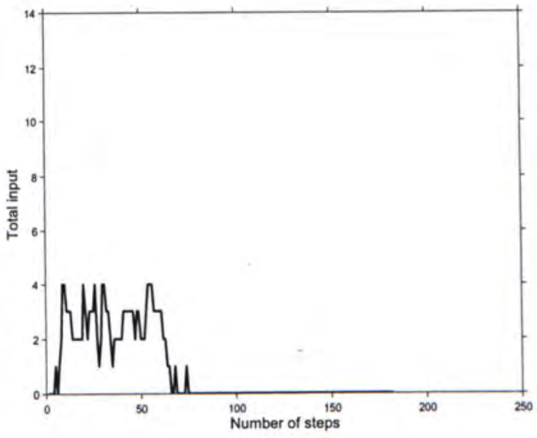


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

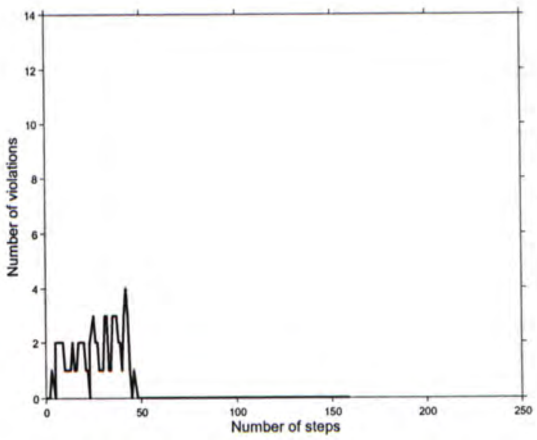
Figure 4.97: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 120$ (average run-time case)



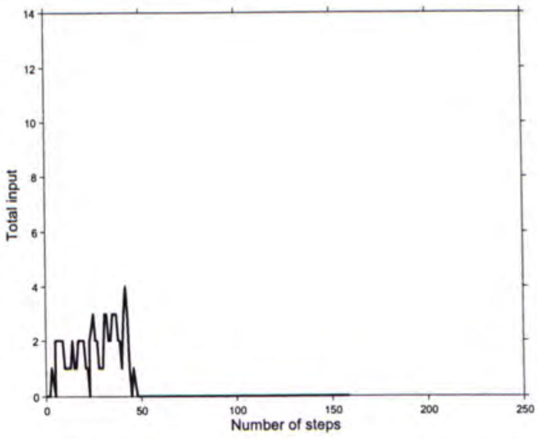
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

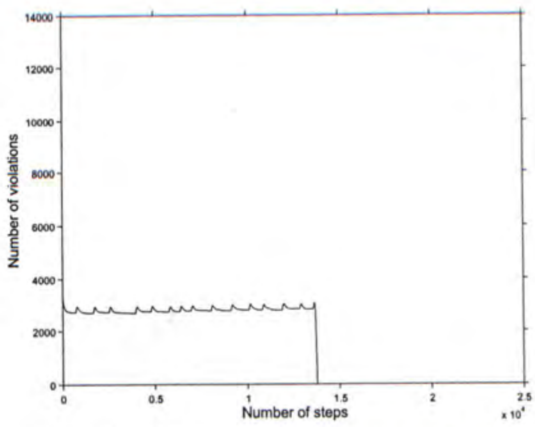


(c) max-IPSS: Violation vs. Step

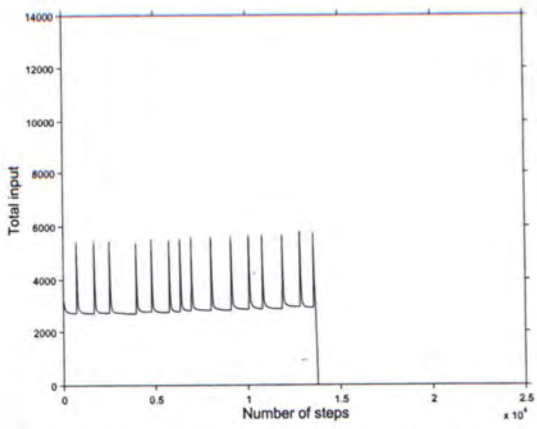


(d) max-IPSS: Total input vs. Step

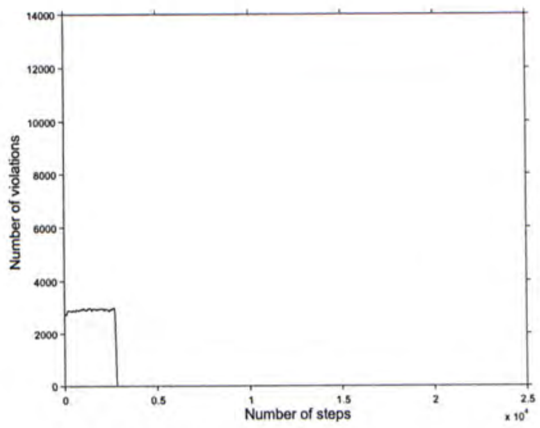
Figure 4.98: Numbers of violations and total inputs in each step of IPSS and max-IPSS on random CSP with $n = 120$ (short run-time case)



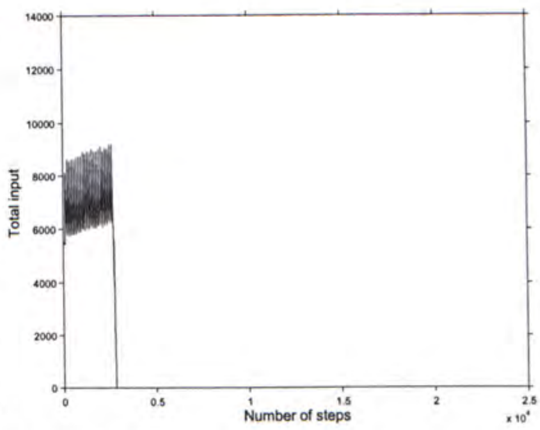
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

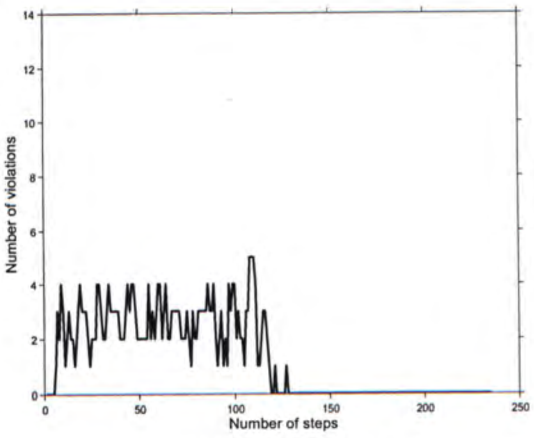


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

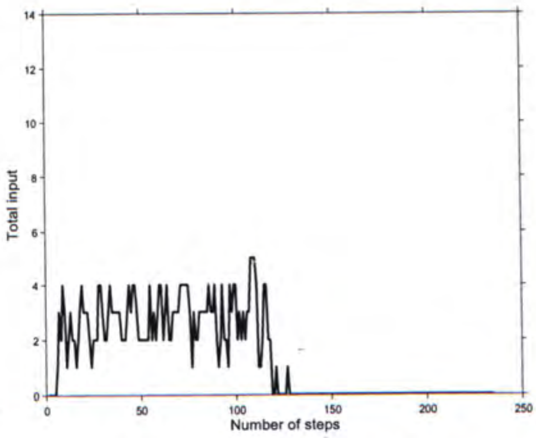


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

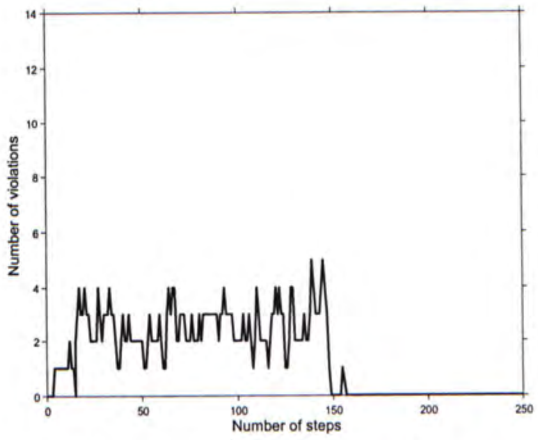
Figure 4.99: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 120$ (short run-time case)



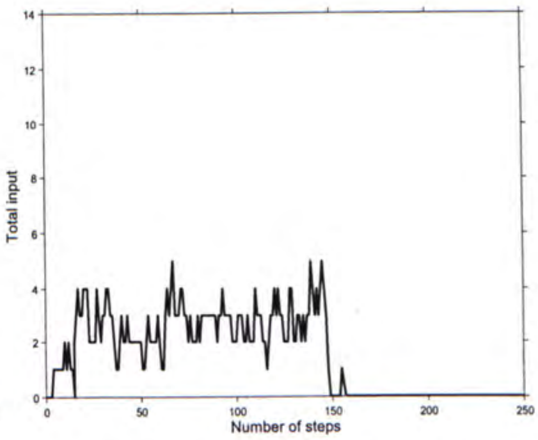
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

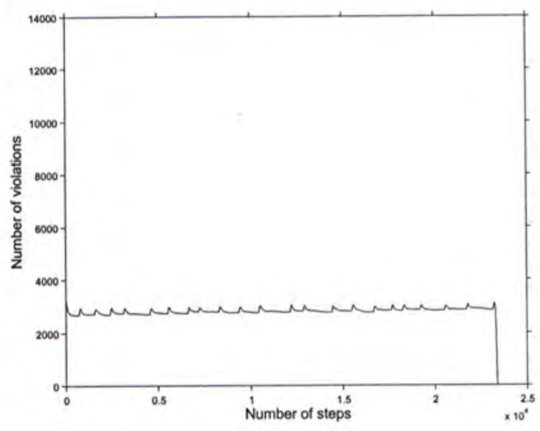


(c) max-IPSS: Violation vs. Step

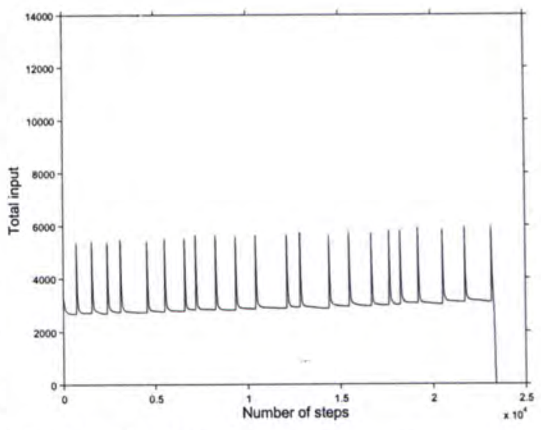


(d) max-IPSS: Total input vs. Step

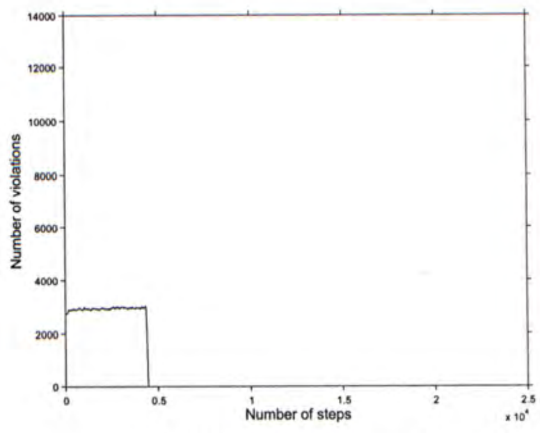
Figure 4.100: Numbers of violations and total inputs in each step of IPSS and max-IPSS on random CSP with $n = 120$ (long run-time case)



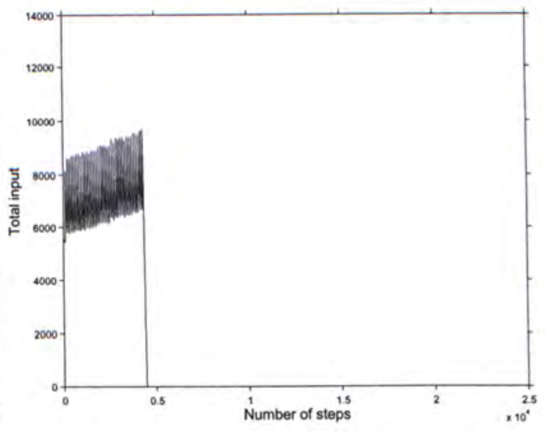
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

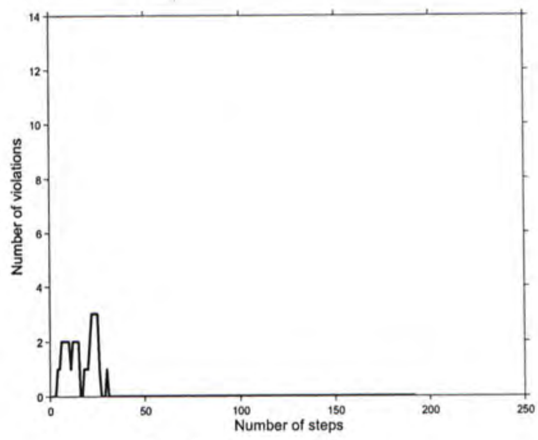


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

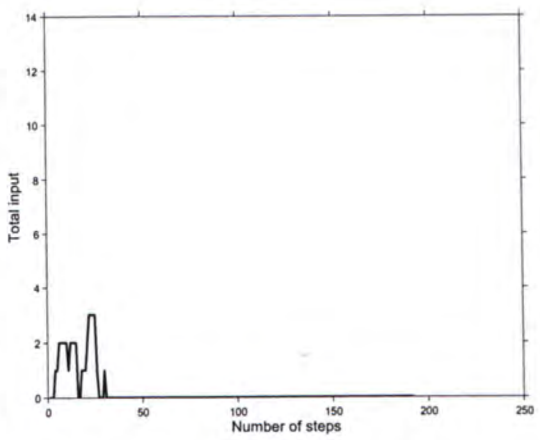


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

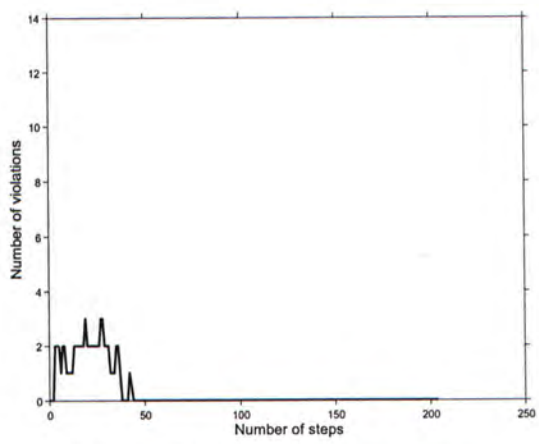
Figure 4.101: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 120$ (long run-time case)



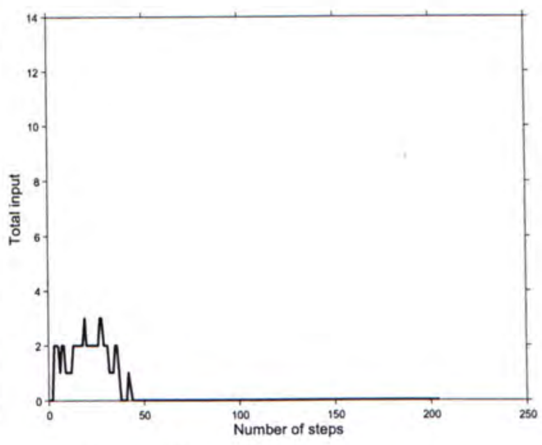
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

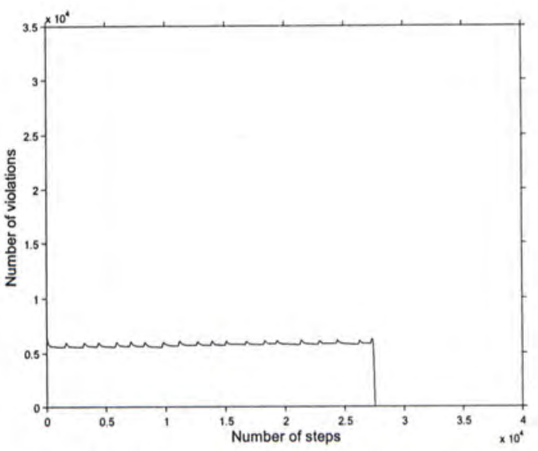


(c) max-IPSS: Violation vs. Step

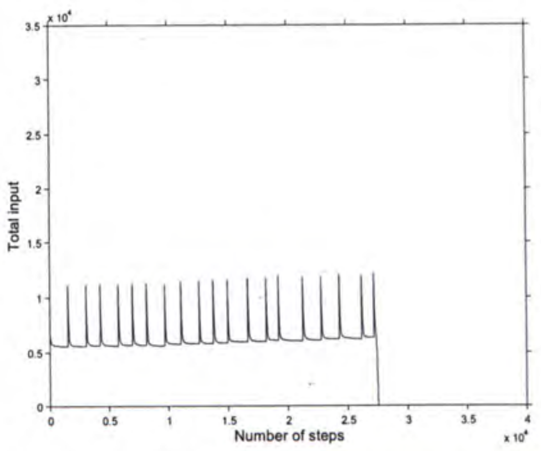


(d) max-IPSS: Total input vs. Step

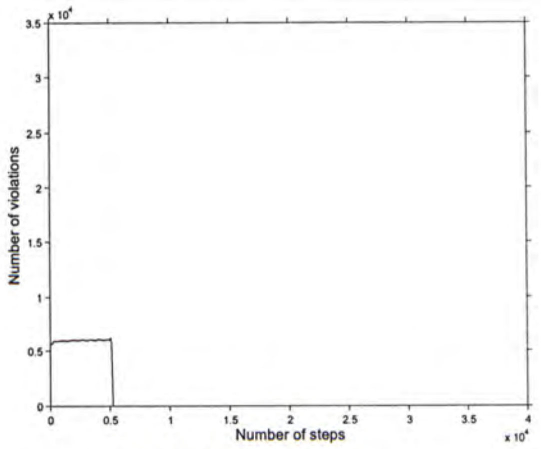
Figure 4.102: Numbers of violations and total inputs in each step of IPSS and max-IPSS on random CSP with $n = 170$ (average run-time case)



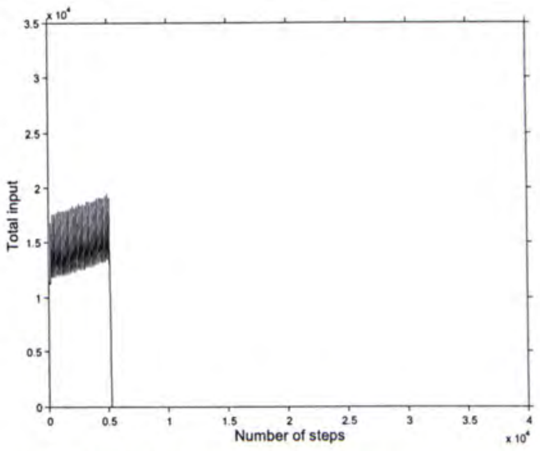
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

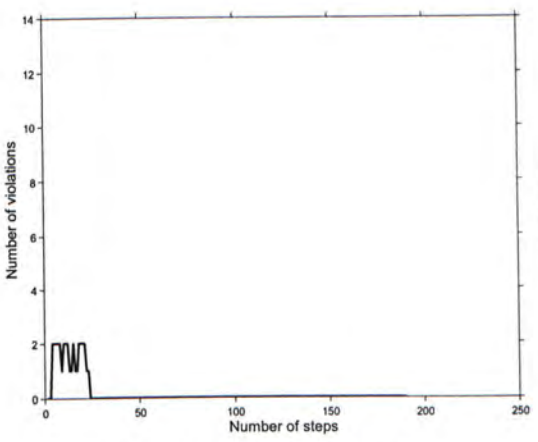


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

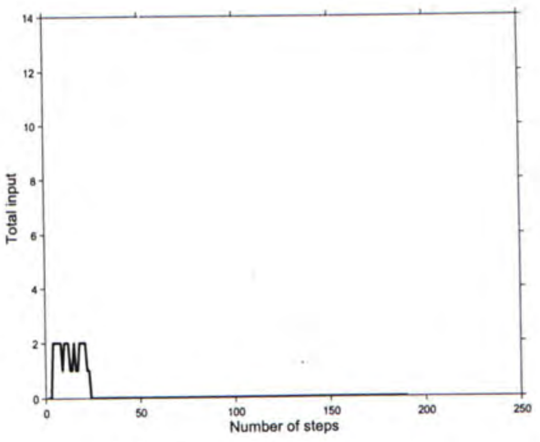


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

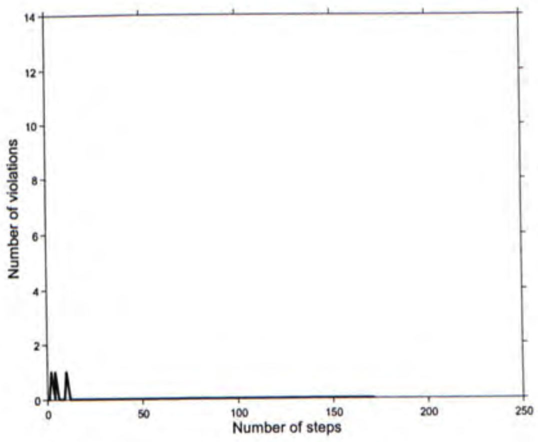
Figure 4.103: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 170$ (average run-time case)



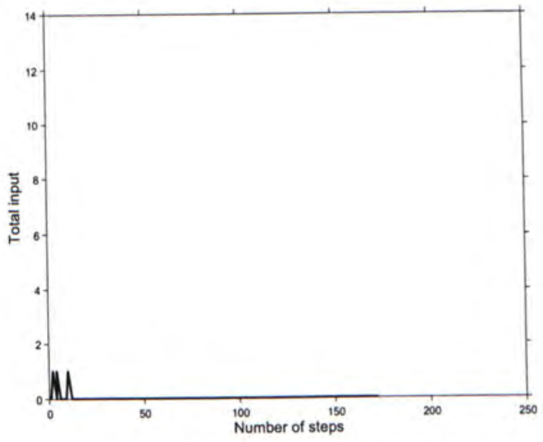
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

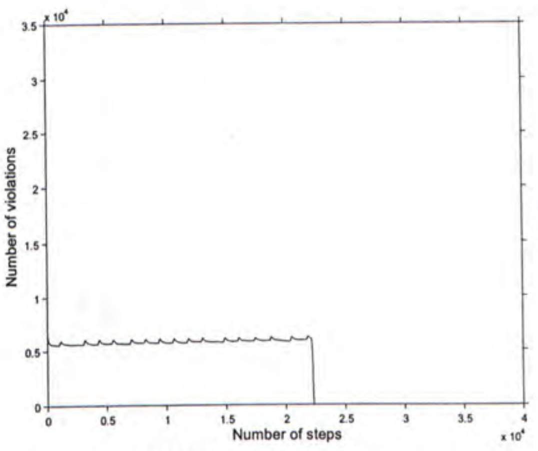


(c) max-IPSS: Violation vs. Step

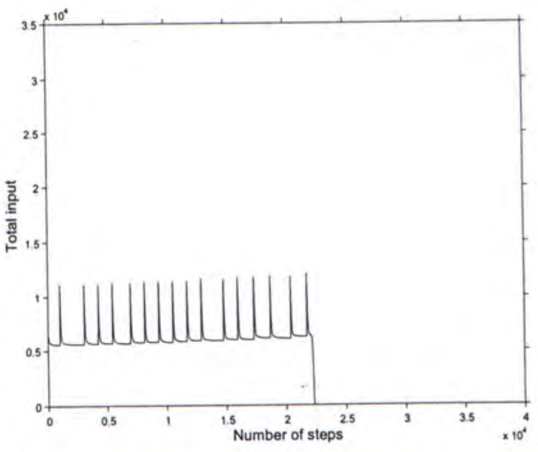


(d) max-IPSS: Total input vs. Step

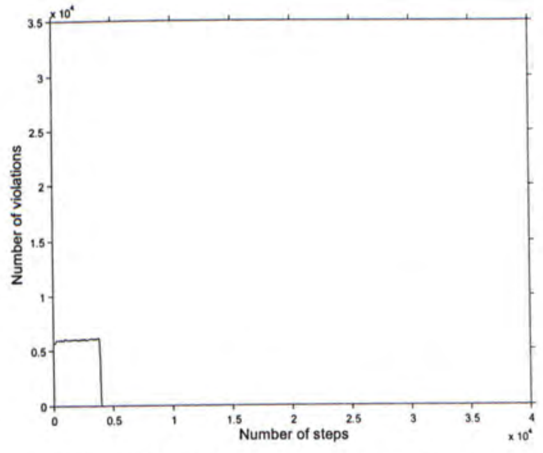
Figure 4.104: Numbers of violations and total inputs in each step of IPSS and max-IPSS on random CSP with $n = 170$ (short run-time case)



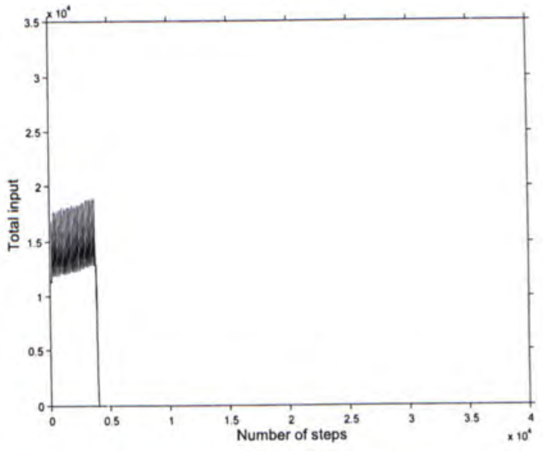
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

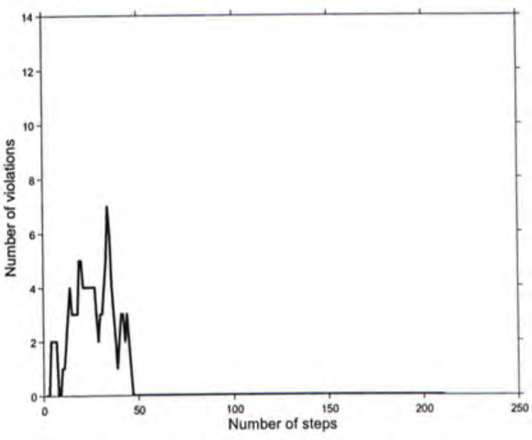


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

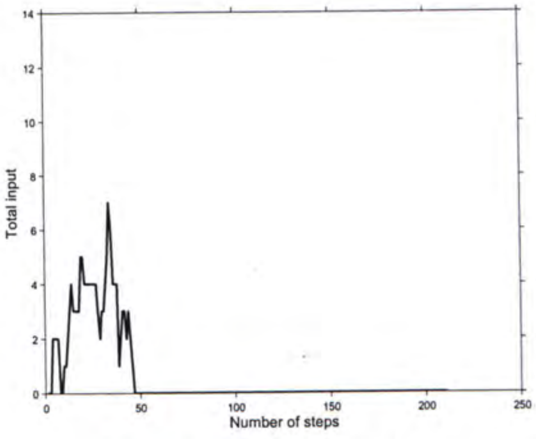


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

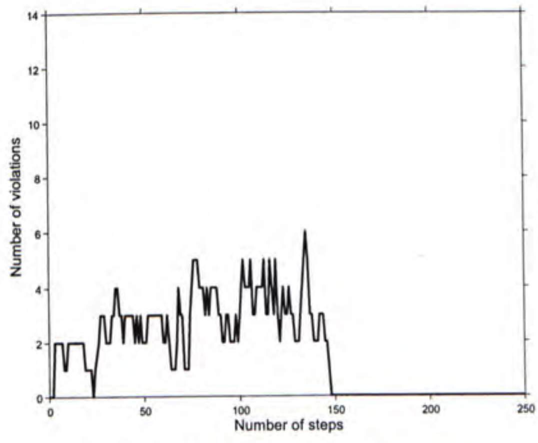
Figure 4.105: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 170$ (short run-time case)



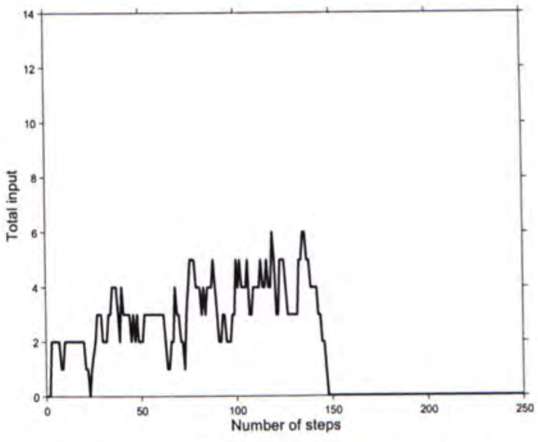
(a) IPSS: Violation vs. Step



(b) IPSS: Total input vs. Step

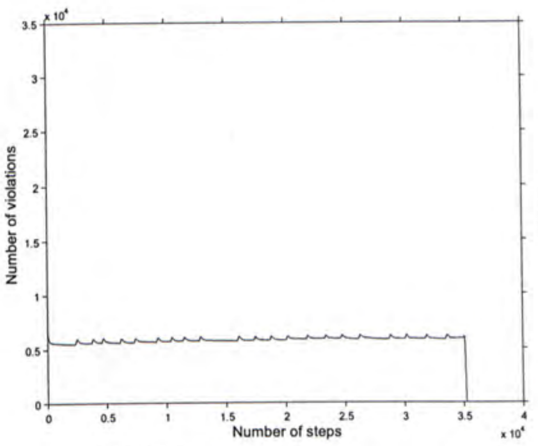


(c) max-IPSS: Violation vs. Step

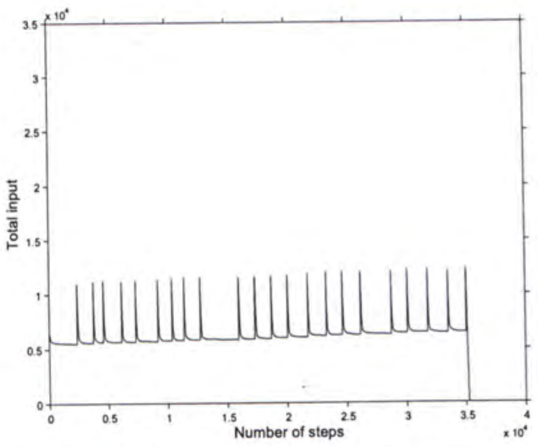


(d) max-IPSS: Total input vs. Step

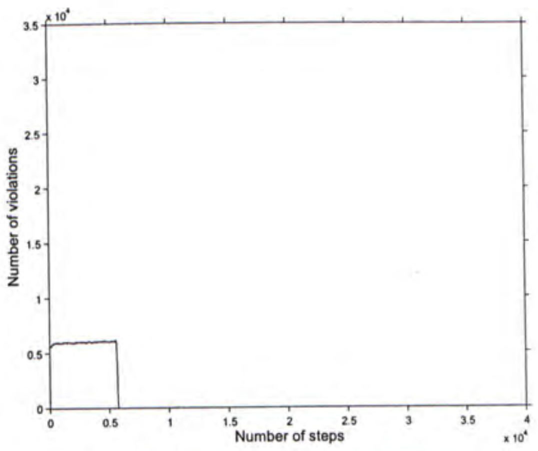
Figure 4.106: Numbers of violations and total inputs in each step of IPSS and max-IPSS on random CSP with $n = 170$ (long run-time case)



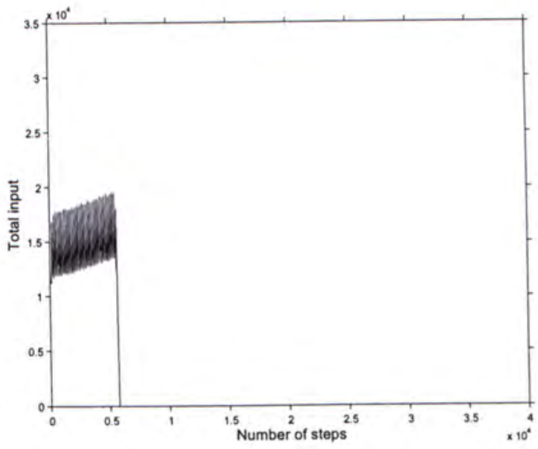
(a) $\mathcal{LSDL}$(GENET): Violation vs. Step



(b) $\mathcal{LSDL}$(GENET): Objective value vs. Step

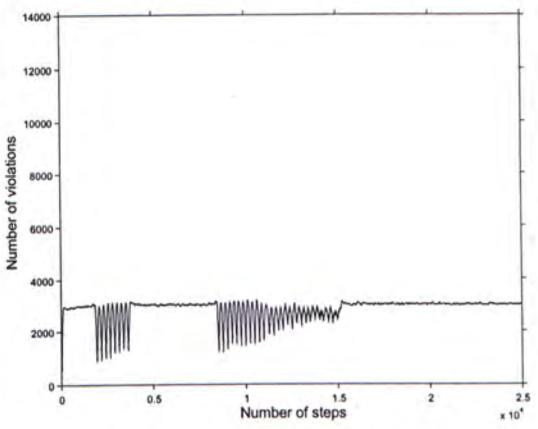


(c) $\mathcal{LSDL}$(IMP): Violation vs. Step

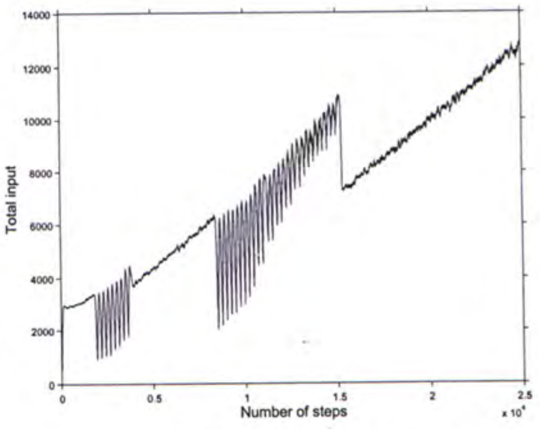


(d) $\mathcal{LSDL}$(IMP): Objective value vs. Step

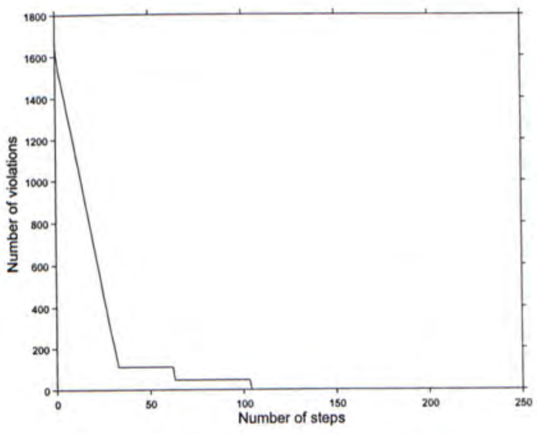
Figure 4.107: Numbers of violations and objective values in each step of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on random CSP with $n = 170$ (long run-time case)



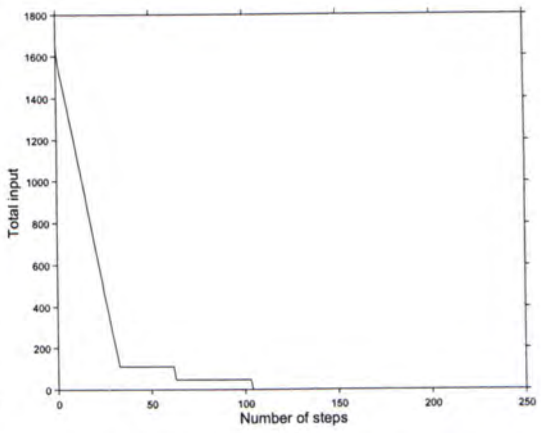
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step

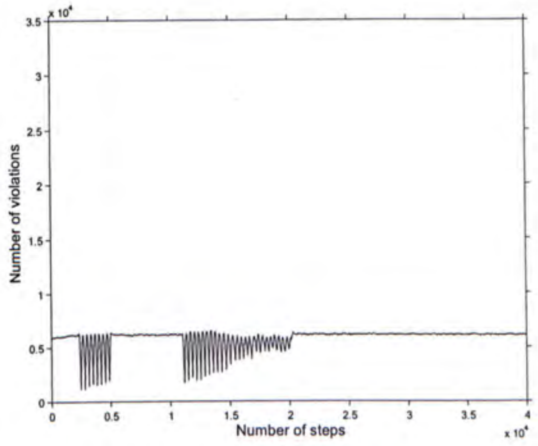


(c) max-PSS: Violation vs. Step

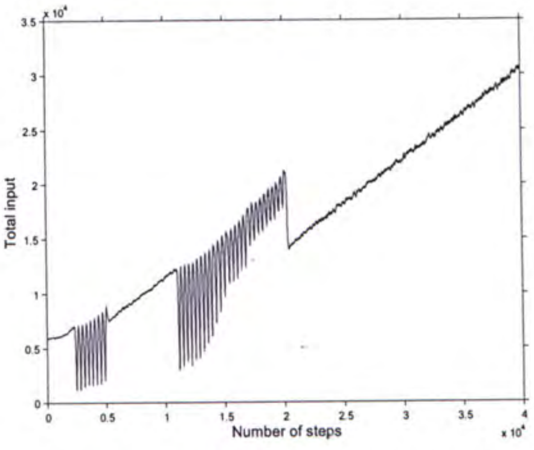


(d) max-PSS: Total input vs. Step

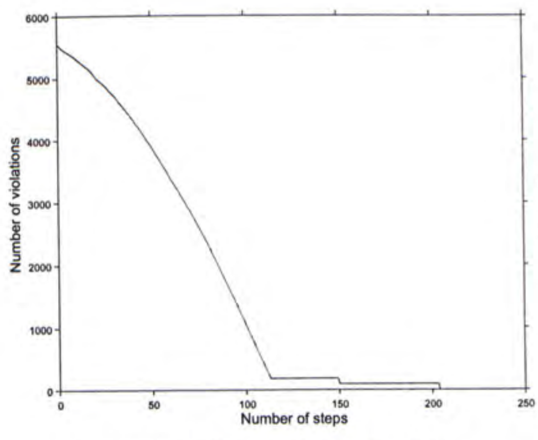
Figure 4.108: Numbers of violations and total inputs in each step of PSS and max-PSS on random CSP with $n = 120$



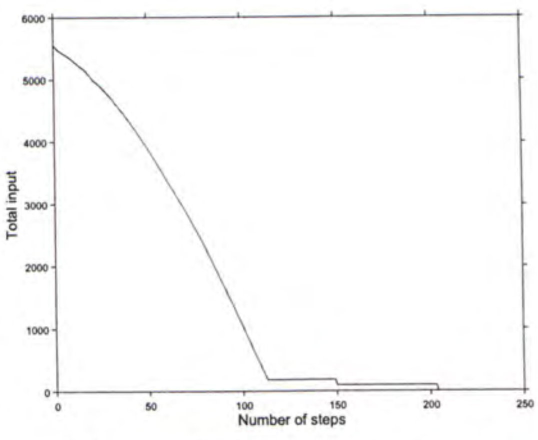
(a) PSS: Violation vs. Step



(b) PSS: Total input vs. Step



(c) max-PSS: Violation vs. Step



(d) max-PSS: Total input vs. Step

Figure 4.109: Numbers of violations and total inputs in each step of PSS and max-PSS on random CSP with $n = 170$

$\mathcal{LSDL}$(GENET) does. Although max-PSS is also not always able to find a solution, it is shown to have the best performance when we study the successful trials (refer to Table 4.11). Figures 4.108(c) and 4.109(c) show the number of violations against total inputs in each step of max-PSS on random CSP with $n = 120$ and $n = 170$ respectively. We see that the cluster selection heuristic guides the search to select an excellent direction in the search space. Interestingly, IPSS has not suffered from the problem that PSS faced. We observe that the partial solutions found can be extended to a complete solution without much difficulty (Figures 4.96(a), 4.98(a), 4.100(a), 4.102(a), 4.104(a) and 4.106(a)). Though with the help of heuristics, max-IPSS has about the same efficiency as IPSS.

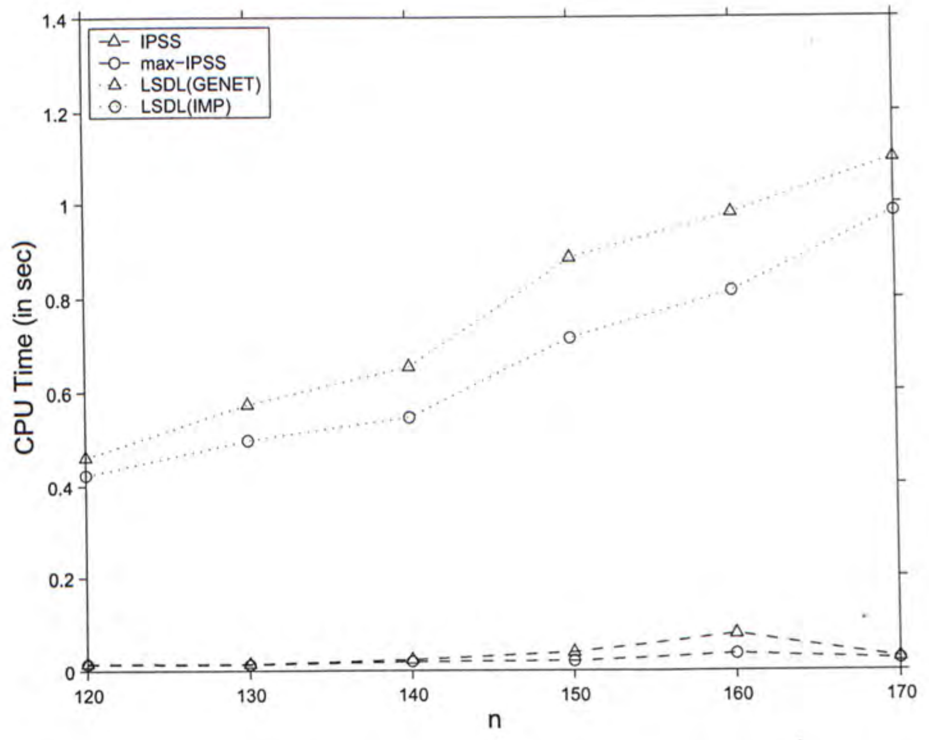


Figure 4.110: The mean time results on random CSPs

Figure 4.110 shows the mean timing results of IPSS, max-IPSS, $\mathcal{LSDL}$ (GENET) and $\mathcal{LSDL}$(IMP). As PSS and max-PSS can only solve the problem instances in some trials, we omit their timing results in the figure. It can be seen that $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) perform significantly worse than IPSS and max-IPSS.

4.4.2 Phase Transition Random CSPs

A set of random binary CSPs close to the phase transition are used in this set of experiments. All problem instances used in this experiment are the same as that in [4]. The execution limits of PSS and IPSS in solving the problem instances are set to 5 million steps, while the execution limits of max-PSS and max-IPSS are set to 10 million steps. The execution limits of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in solving the problem instances are set to 5 million iterations. We use a superscript $(x/100)$ besides the timing figures to indicate that only x out of the hundred runs are successful.

Problem	PSS		IPSS	
n	Steps $\times 10^3$	**CPU time**	Steps $\times 10^3$	**CPU time**
120	>5000	>**66.4813**$^{(0/100)}$	>5000	>**64.6240**$^{(0/100)}$
130	>5000	>**68.7239**$^{(0/100)}$	>5000	>**66.3857**$^{(0/100)}$
140	>5000	>**68.3965**$^{(0/100)}$	>5000	>**66.0671**$^{(0/100)}$
150	>5000	>**70.1852**$^{(0/100)}$	>5000	>**66.7879**$^{(0/100)}$
160	>5000	>**71.1339**$^{(0/100)}$	>5000	>**67.1439**$^{(0/100)}$
170	>5000	>**71.5648**$^{(0/100)}$	>5000	>**67.7324**$^{(0/100)}$
	max-PSS		max-IPSS	
120	>10000	>**69.6243**$^{(0/100)}$	>10000	>**71.9720**$^{(0/100)}$
130	>10000	>**72.8381**$^{(0/100)}$	>10000	>**76.4229**$^{(0/100)}$
140	>10000	>**75.9473**$^{(0/100)}$	>10000	>**79.0859**$^{(0/100)}$
150	>10000	>**79.1777**$^{(0/100)}$	>10000	>**83.0627**$^{(0/100)}$
160	>10000	>**81.8229**$^{(0/100)}$	>10000	>**85.5608**$^{(0/100)}$
170	>10000	>**84.2616**$^{(0/100)}$	>10000	>**88.5191**$^{(0/100)}$

Table 4.13: PSS and its variants on phase transition random CSPs

Table 4.13 shows results of PSS and its variants on random CSPs close to the phase transition. The results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on the same set of problems are given for comparison in Table 4.14. From Table 4.13, none of the trails can solve the problem instances. For $\mathcal{LSDL}$ implementations,

Problem	$\mathcal{LSDL}$(GENET)			
n	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
120	688.6(304.0)	597.6(366.5)	375.8(144.0)	**6.1133(3.8700)**$^{(15/100)}$
130	>5000	>3678	>2876	**>40.6479**$^{(00/100)}$
140	884.9(511.0)	811.5(583.1)	474.6(249.5)	**8.7275(6.4300)**$^{(20/100)}$
150	>5000	>4000	>2803	**>45.4193**$^{(00/100)}$
160	>5000	>4127	>2774	**>46.3004**$^{(00/100)}$
170	828.2(292.9)	831.2(450.1)	433.5(127.1)	**9.5086(5.4100)**$^{(07/100)}$
Problem	$\mathcal{LSDL}$(IMP)			
n	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
120	991.9(760.5)	1249(1047)	991.9(760.5)	**9.8117(8.4650)**$^{(12/100)}$
130	2057(2057)	2564(2564)	2057(2057)	**21.290(21.290)**$^{(01/100)}$
140	731.0(410.9)	1070(743.1)	731.0(410.9)	**8.8668(6.4500)**$^{(19/100)}$
150	1886(1886)	2667(2667)	1886(1886)	**23.150(23.150)**$^{(01/100)}$
160	383.7(197.1)	726.2(496.1)	383.7(197.1)	**6.6350(4.7450)**$^{(04/100)}$
170	2454(2473)	3615(3643)	2454(2473)	**30.863(32.410)**$^{(03/100)}$

Table 4.14: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on phase transition random CSPs

not more than 20% of the trails can solve the problem successfully. As these problem instances are hard to all solvers, it is difficult to make comparison.

In order to compare the performance of PSS implementations to that of $\mathcal{LSDL}$ implementations on random CSPs close to the phase transition, we use slightly less difficult problem instances stated in [4] to conduct another experiment.

Table 4.15 shows results of PSS and its variants on slightly easier phase transition random CSPs . The results of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on the same set of problems are given for comparison in Table 4.16. As mentioned in [4], this set of problem instances are difficult for stochastic solvers. For $\mathcal{LSDL}$ implementations, not all trails can find the solution successfully. From the tables, the performance of PSS and its variants are not as good as $\mathcal{LSDL}$ implementations in this set of experiments.

Problem	PSS		IPSS	
n	Steps $\times 10^3$	**CPU time**	Steps $\times 10^3$	**CPU time**
120	>5000	>**68.3004**$^{(0/100)}$	>5000	>**61.9739**$^{(0/100)}$
130	>5000	>**70.8367**$^{(0/100)}$	>5000	>**63.8469**$^{(0/100)}$
140	>5000	>**69.8512**$^{(0/100)}$	>5000	>**63.7774**$^{(0/100)}$
150	>5000	>**69.1622**$^{(0/100)}$	>5000	>**65.1413**$^{(0/100)}$
160	>5000	>**69.3043**$^{(0/100)}$	>5000	>**66.1019**$^{(0/100)}$
170	>5000	>**69.3696**$^{(0/100)}$	>5000	>**66.2910**$^{(0/100)}$
	max-PSS		max-IPSS	
120	>10000	>**69.5245**$^{(0/100)}$	>10000	>**72.0099**$^{(0/100)}$
130	>10000	>**72.4805**$^{(0/100)}$	>10000	>**76.0063**$^{(0/100)}$
140	>10000	>**75.9728**$^{(0/100)}$	>10000	>**78.6275**$^{(0/100)}$
150	1695.4(1695.4)	**1.4190(1.4190)**$^{(1/100)}$	>10000	>**82.0091**$^{(0/100)}$
160	>10000	>**81.5350**$^{(0/100)}$	>10000	>**85.2180**$^{(0/100)}$
170	>10000	>**83.5906**$^{(0/100)}$	>10000	>**87.1946**$^{(0/100)}$

Table 4.15: PSS and its variants slightly easier phase transition random CSPs

Problem	$\mathcal{LSDL}$(GENET)			
n	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
120	753.7(439.9)	650.8(461.1)	410.4(220.7)	**6.4147(4.6600)**$^{(34/100)}$
130	1195(322.2)	995.3(403.4)	663.8(150.5)	**10.496(4.4400)**$^{(05/100)}$
140	812.7(413.2)	740.1(474.3)	434.9(200.5)	**7.3846(4.9100)**$^{(69/100)}$
150	898.2(370.3)	848.2(467.0)	475.6(173.7)	**8.8260(5.1600)**$^{(55/100)}$
160	986.9(364.0)	953.5(492.4)	520.5(166.4)	**10.240(5.5950)**$^{(28/100)}$
170	689.9(274.9)	716.5(402.5)	354.9(122.2)	**7.6204(4.5900)**$^{(84/100)}$
Problem	$\mathcal{LSDL}$(IMP)			
n	Iteration $\times 10^3$	Repairs $\times 10^3$	Learns $\times 10^3$	**CPU time**
120	903.5(360.6)	1117(598.1)	903.5(360.6)	**8.4019(4.9000)**$^{(27/100)}$
130	3572(3572)	4248(4248)	3572(3572)	**33.480(33.480)**$^{(01/100)}$
140	625.3(222.5)	892.2(464.7)	625.3(222.5)	**6.8369(3.8900)**$^{(75/100)}$
150	667.0(191.0)	1005(439.1)	667.0(191.0)	**7.9671(3.9850)**$^{(42/100)}$
160	1717(1381)	2467(2124)	1717(1381)	**19.420(17.040)**$^{(22/100)}$
170	614.5(153.2)	994.5(406.8)	614.5(153.2)	**7.9194(3.7000)**$^{(84/100)}$

Table 4.16: $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) on slightly easier phase transition random CSPs

Chapter 5

Concluding Remarks

We end the thesis in this chapter by concluding our contributions and giving possible directions for future work.

5.1 Contributions

In this thesis we present a novel stochastic search scheme, Progressive Stochastic Search (PSS), for solving binary CSPs. A typical stochastic search method uses a *cost function* to evaluate the goodness of every point in a search space, and a *neighborhoods function* to define the neighbors of a particular point in the search space. The search starts from a random point in the search space and moves from one point to its better neighboring point until the stopping criteria are matched. This can be interpreted as that the move is driven solely by "potential energy", though the movement towards which better neighboring point is usually determined randomly. As the search only moves from one point to its neighboring point that gives an improvement in the cost, the search may stay at the current point and no other movements can be made. The search is trapped in local optima or plateaus. Random restart and heuristic learning are the methods used to escape from local optima or leave plateaus traditionally. Intuitively, this search approach can be thought to be prudent. The main novelty of PSS is that the search is able to "rush through" the local optima and plateaus with the cooperation of a new heuristic repair method and a simple search path marking method. We maintain a list of variables, which dictates the sequence of variables to repair. When a variable is being repaired, it is always assigned a new value even if its original value should give the best cost value. The search paths are slightly "marked" as the search proceeds by updating the weights of the connections at the end of each convergence step. Unlike the prudent approach used in the typical stochastic search method, the search approach of PSS is more progressive. This progressive approach shows an encouraging performance in some benchmarking problems.

We also present an incremental variant of PSS, namely IPSS. IPSS works on a partial assignment and performs PSS on that partial assignment to find a partial solution. This partial solution is then extended by adding a variable that is not involved in the partial solution until a complete solution is obtained. IPSS is found to be more efficient than PSS in some benchmarking problems that the partial solutions can be extended easily. As mentioned before, PSS and IPSS use a list of variable to dictate the sequence of variables to repair. The ordering is in a *first-in-first-out* manner. We integrate the idea of greedy variable ordering into PSS and IPSS to form other variants, namely, max-PSS and max-IPSS respectively. Experimental results show that the greedy variable ordering provides an excellent direction for the search towards the solutions in some benchmarking problems.

We perform experiments using four types of benchmarking problems, namely the N-Queens problems, the permutation generation problems, the quasigroup completion problems and Latin squares, and random constraint satisfaction problems. The results show that the PSS class of schemes can outperform $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) in N-queens problems, Latin squares, random permutation generation problems, and random CSPs. However, their performance in increasing permutation generation problems and quasigroup completion problems are worse than that of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP).

We present analysis of the search process of all these solvers in an attempt to provide an explanation to this phenomenon.

5.2 Future Work

We believe this thesis presents an interesting new approach to the design of stochastic search schemes for solving constraint satisfaction problems. As future work, we shall investigate other heuristics that can possibly improve the performance. For examples, the method that calculates the input of a label node, the learning rule that updates the connection weights, and the strategy that selects a cluster for repair at the next convergence step. With the encouraging performance of max-PSS and max-IPSS in some benchmarking problems, we believe that other suitable heuristics for the above three parts can boost up the performance of PSS.

The benchmarking problems used in the experiments of this research are almost the same as that used in $\mathcal{CSDL}$ [4] except for the hard graph-coloring problems. We have conducted an experiment for the hard graph-coloring problems. The experimental results show that the PSS class of schemes cannot find solutions within the pre-set limit. Since we are still investigating what makes this kind of problems hard to the PSS class of schemes, we extract this part from experiments and put it as future work. It is also interesting to investigate if other heuristics can help the PSS class of schemes to solve the hard graph-coloring problems.

The possibility of its integration with GENET class solvers is also another issue to be researched into. The search approach of the PSS class of schemes is progressive, while that of GENET class solvers is prudent. The experimental results show that different approaches have their advantage in different benchmarking problems. It is worthwhile to research under what situations the search should decide to use progressive approach or prudent approach, so that advantages from both side can be exploited.

Bibliography

[1] D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction: A more accurate picture. In *Proc. 3rd International Conference on Principles and Practics of Constraint Programming*, pages 107–120, 1997.

[2] J. Bitner and E.M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–655, 1985.

[3] C.Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. AAAI-97*, pages 221–226, 1997.

[4] M.F. Choi, J.H.M. Lee, and P.J. Stuckey. A lagrangian reconstruction of GENET. *Artificial Intelligence*, 123:1–39, 2000.

[5] P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *SAGA*, pages 73–90, 2001.

[6] A. Davenport, E.P.K. Tsang, C.J. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proc. AAAI-94*, pages 325–330, 1994.

[7] J. Frank, P. Cheeseman, and J. Allen. Weighting for godat: Learning heuristics for gsat. In *Proc. AAAI-96*, pages 338–343, 1996.

[8] I.P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures. In *Proc. AAAI-93*, pages 28–33, 1993.

[9] F. Glover. Tabu search part I. *Operations Research Society of America (ORSA) Journal on Computing*, 1(3):109–206, 1989.

[10] F. Glover. Tabu search part II. *Operations Research Society of America (ORSA) Journal on Computing*, 2(1):4–32, 1989.

[11] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[12] N. Jussien and O. Lhomme. The path-repair algorithm. In *CP99 Post-conference workshop on Large scale combinatorial optimisation and constraints*, volume 4, 2000.

[13] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[14] J.H.M Lee, H.F. Leung, and H.W. Won. Extending GENET for non-binary constraint satisfaction problems. In *7th International Conference on Tools with Articial Intelligence*, pages 338–342, 1995.

[15] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[16] K. Marriott and P.J. Stuckey. Programming with constraints. *The MIT Press*, 1998.

[17] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[18] P. Morris. The breakout method for escaping from local minima. In *Proc. AAAI-93*, pages 40–45, 1993.

[19] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[20] G. Pesant and M. Gendreau. A view of local search in constraint programming. In *Principles and Practice of Constraint Programming*, pages 353–366, 1996.

[21] E.T. Richards, Y. Jiang, and B. Richards. Ng-backmarking - an algorithm for constraint satisfaction. *AIP Techniques for Resource Scheduling and Planning, BT Technology Journal*, 13(1), 1995.

[22] E.T. Richards and B. Richards. Non-systematic search and learning: An empirical study. In *Proc. Conference on Principles and Practice of Constraint Programming*, pages 370–384, 1998.

[23] F. Rossi, C.J. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proc. ECAI-90*, pages 550–556, 1990.

[24] A. Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proc. IJCAI-97*, pages 1254–1259, 1997.

[25] B. Selman and H. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *Proc. IJCAI-93*, pages 290–295, 1993.

[26] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. AAAI-94*, pages 337–343, 1994.

[27] B. Selman, H. Levesque, and D.G. Mitchell. Ta new method for solving hard satisfiability problems. In *Proc. AAAI-92*, pages 440–446, 1992.

[28] Y. Shang and B.W. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal on Global Optimization*, 12:61–100, 1998.

[29] P.J. Stuckey and V.W.L. Tam. Improving GENET and EGENET by new variable ordering strategies. In *Proc. of the International Conference on Computational Intelligence and Multimedia Applications*, pages 107–112, 1998.

[30] E. Tsang and C.J. Wang. A generic neural network approach for constraint satisfaction problems. In *Neural Network Applications*, pages 12–22. SpringerVerlag, 1992.

[31] C. Voudouris and E. Tsang. Guided local search and its application to the traveling salesman problem. *Europenan Journal of Operational Research*, 113:469–499, 1999.

[32] B.W. Wah and Y.J. Chang. Trace-based methods for solving nonlinear global optimization and satisfiability problems. *Journal on Global Optimization*, 10:107–141, 1997.

[33] J.H.Y. Wong and H.F. Leung. Extending GENET to solve fuzzy constraint satisfaction problems. In *Proc. AAAI-98*, pages 380–385, 1998.

[34] Z. Wu. The discrete lagrangian theory and its application to solve nonlinear discrete constrained optimization problems. MSc Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1998.

[35] M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proc. AAAI-94*, pages 313–318, 1994.

CUHK Libraries
004077294