A LAGRANGIAN RECONSTRUCTION OF A CLASS OF LOCAL SEARCH METHODS

By

Choi Mo Fung Kenneth

A Dissertation

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Computer Science and Engineering

The Chinese University of Hong Kong

June 1998

摘 要

近年來，愈來愈多數據顯示，局部搜尋法中的試探性修復算法能夠有效地解決一些大型及困難的約束滿足問題 (CSP's)。在這篇論文中，我們嘗試把試探性修復法與離散拉格朗日乘數法聯系起來。我們提供了一種二步的變換方法，它能夠將任何二元約束滿足問題轉換成零一整數約束極小化問題。根據變換所得的極小化問題，我們提出了一個名叫 *LSDL* 的離散拉格朗日搜尋方案。*LSDL* 包含五種不同的自由度，只要代入不同的參數，我們便能獲得不同效率及不同特性的離散拉格朗日搜尋算法。我們更證明了能夠利用 *LSDL* 重建一個試探性修復的典型算法—GENET。這結果不但對了解 GENET 提供了一些理論基礎，更指出 GENET 的雙重觀點—一方面是試探性修復法，另一方面是離散拉格朗日乘數法。此雙重觀點使我們能夠進一步分析及研究 GENET 的各種變種。實驗結果證實我們重建的 GENET 與其他文獻上記載的 GENET 有著相同的迅速收斂特性。另外，我們的最好的 GENET 變種亦比重建的 GENET 更有效率。此外，我們更擴展 *LSDL* 去解決一般的約束滿足問題。我們發現只要將一般的約束滿足問題轉換成整數約束極小化問題，我們的 *LSDL* 便能直接地運用在這些問題上。實驗指出我們擴展的 *LSDL* 在大部份的問題上都與一個擴展 GENET 的算法—E-GENET—有著相似的效率。

Abstract

In recent years, heuristic repair algorithms, a class of local search methods, have demonstrated certain success on solving some large-scale and computationally hard constraint satisfaction problems (CSP's). In this thesis, we establish a connection between heuristic repair methods and the discrete Lagrange multiplier method. We present a two-step transformation for converting binary CSP's into zero-one integer constrained minimization problems. Based on the resultant minimization problem, a generic discrete Lagrangian search scheme $\mathcal{LSDL}$ is proposed. $\mathcal{LSDL}$ has five degrees of freedom. By instantiating it with different parameters, algorithms with different efficiency and behavior can be obtained. We show that the GENET model, a representative heuristic repair algorithm, can be reconstructed by our $\mathcal{LSDL}$ framework. This result not only provides a theoretical foundation for better understanding of GENET, but also suggests a dual viewpoint of GENET: as a heuristic repair method and as a discrete Lagrange multiplier method. Variants of GENET derived from both perspectives are examined. Benchmarking results confirm that our reconstructed GENET has the same fast convergence behavior as other GENET implementations reported in the literature, and our best variant is more efficient than the reconstructed GENET. In addition, we further extend our $\mathcal{LSDL}$ framework for tackling general CSP's. By transforming any general CSP into an integer constrained minimization problem, the discrete Lagrangian search procedure $\mathcal{LSDL}$ can be applied directly. Experiments show that the performance of our extended $\mathcal{LSDL}$ is comparable with that of E-GENET, an extended GENET for efficient general CSP's solving, in most problems.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Many problems found in artificial intelligence and computer science, such as temporal reasoning, resource allocation, scheduling, time-tabling, configuration, diagnosis and satisfiability problems, can be formulated as constraint satisfaction problems (CSP's). Because of the generality of CSP's, efficient algorithms for tackling CSP's are very important. Tree search methods and local search methods are two common classes of constraint satisfaction techniques. Experience shows that local search methods are more efficient than tree search methods for a number of large-scale and computationally hard CSP's. However, local search methods are easily trapped in local minima and fail to return a solution. This thesis aims to explore a class of local search methods for solving CSP's and provide a connection between the local search methods and the Lagrange multiplier method, a well-known constrained optimization technique.

We show that the GENET model [66, 60, 7, 6], a representative member of the class of heuristic repair methods, is closely related to the saddle point search of the Lagrange multiplier method. This result not only gives us a formal characterization of the heuristic repair methods, but also allows us to gain important insights into the various design issues of heuristic repair algorithms. In addition, the dual viewpoint of GENET, as a heuristic repair method and as a discrete Lagrange multiplier method, suggests many possible modifications for the algorithm. By exploring different variants of GENET derived from both perspectives, better

heuristic repair algorithms are derived.

1.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) [37] is a tuple (U, D, C), where U is a finite set of variables, D defines a *domain* D_u which contains a set of possible values for each $u \in U$, and C is a finite set of constraints. Each constraint $c \in C$ is a relation defined over a subset of variables (i.e. $c \subseteq D_{u_1} \times D_{u_2} \times \ldots \times D_{u_k}$), restricting the combination of values that can be assigned to the variables in this subset. A *solution* is an assignment of values from the domains to their respective variables so that all constraints are satisfied simultaneously. We call such an assignment a *consistent* assignment of the CSP. In this thesis, we are concerned with an important subclass of CSP's, in which the domains are finite.

The *arity* of a constraint is the number of variables involved in the constraint. A constraint is said to be n-ary if it is on n variables. In general, a CSP may have constraints of any arity. A binary CSP is one which contains unary and binary constraints only. A *label* $\langle u, v \rangle$ [66] is a variable-value pair which represents the assignment of the value v to the variable u. Similarly, a *compound label* $(\langle u_1, v_1 \rangle, \langle u_2, v_2 \rangle, \ldots, \langle u_n, v_n \rangle)$ [66] denotes the simultaneous assignment of the values $v_1, v_2, \ldots, v_n$ to the distinct variables $u_1, u_2, \ldots, u_n$ respectively. Hence, a solution of a CSP is a compound label containing all variables and satisfying all constraints in the CSP.

1.2 Constraint Satisfaction Techniques

CSP's are, in general, NP-hard [5]. Any algorithm for solving CSP's is likely to require exponential time in problem size in the worst case. Two different kinds of algorithms, namely tree search algorithms and local search algorithms, are used to tackle CSP's. Tree search algorithms are usually based on extension of partial assignments and backtracking technique, where a *partial assignment* is an

assignment of values to a subset of variables in the CSP. Initially, a tree search algorithm starts with an empty partial assignment and extends incrementally the partial assignment by selecting an unassigned variable and assigning a value to the selected variable from its domain. If the current partial assignment cannot be extended without violating any constraints, the algorithm backtracks to one of the previous variables and instantiates the variable with another untried value from its domain. As the process continues, the algorithm will eventually either find a solution for the problem, or prove that the CSP has no solution. Since tree search algorithms are guaranteed to either return a solution if one exists or prove the insolubility of a CSP, they are sound and complete.

Many tree search algorithms have been developed for solving CSP's. Examples are simple backtracking [42, 29], backjumping [42] and backmarking [42]. In order to improve the performance, constraint propagation techniques, such as consistency algorithms [37, 29] and forward checking [21], are incorporated in tree search algorithms to reduce the search space of the algorithms. Various variable and value ordering heuristics [21, 29] are also used to further speed up the search process. Although tree search algorithms can successfully solve many real-life problems, they become inefficient when the problem size increases.

Another approach for constraint satisfaction is local search. Unlike tree search algorithms, local search algorithms are usually incomplete. In other words, they may not return a solution even if one exists. Given a CSP (U, D, C), a local search algorithm operates as follows. The algorithm first generates an initial assignment (or state), which assigns each variable $u \in U$ a value from its domain D_u, for the CSP. It then continues to make local adjustments, which depend on specific local search algorithm, to the assignment until a solution is obtained. Various local search algorithms, such as the min-conflicts heuristic [39, 40], GSAT [52, 49, 51, 13], the breakout method [41], GENET [66, 60, 7, 6], E-GENET [32, 33, 69] and DLM [62, 54, 53], are developed in recent years. They have been found to be effective in solving certain large-scale and computationally hard CSP's.

An important property of local search algorithms is that they can be trapped in

a non-solution state, called a local minimum, in which no further local adjustment can be made. Either random restart [39, 40, 52] or modification of the landscape of search surface [41] are proposed for escaping from local minima. In chapter 2, we review a number of these local search methods.

1.3 Motivation of the Research

Although a number of efficient local search algorithms are developed for solving CSP's, little work has been done on understanding why these methods have such a good performance. Minton *et al.* [40] provided a statistical model and probabilistic analysis for the min-conflicts heuristic for random CSP's. Gent and Walsh [15, 16] investigated various features of the GSAT algorithm. They evaluated the importance of greediness [15, 16], randomness [15] and hill-climbing [15, 16] of GSAT on some random SAT problems. Since different local search algorithms always have certain degrees of variation, analysis based on one method may not be applicable to the others.

Local search methods, such as the breakout method, GENET and E-GENET, rely on modifying the landscape of the search surface to escape from local minima. Although the idea is simple and intuitive, little is known theoretically about why and how they work so well. Based on the breakout method, Morris [41] gave some insights on the advantage of this approach and provided a physical analogy for the algorithm. However, a theoretical explanation does not exist.

These situations motivate us to analyze the local search methods for CSP's and to provide some theoretical foundations for these methods. In our research, we concentrate our attention on a class of local search algorithms derived from the heuristic repair method. These algorithms include the min-conflicts heuristic, GSAT, the breakout method, GENET and E-GENET. The energy perspective of GENET [35, 36], a representative of heuristic repair methods, suggest an optimization approach for constraint satisfaction. This observation motivates us to investigate the relation between heuristic repair methods and constrained optimization

techniques. By exploring the similarity between these two methods, a connection between GENET and a form of the Lagrange multiplier method [24, 55], a well-known technique for solving constrained optimization problems, is established. As a result, better understanding of the local search methods is achieved.

1.4 Overview of the Thesis

The thesis is organized as follows. Chapter 2 gives a brief review of various local search methods. The min-conflicts heuristic, GSAT, the breakout method, GENET, E-GENET, DLM, simulated annealing, genetic algorithms, tabu search and integer programming are considered. Chapter 3 provides the necessary background for the thesis. The GENET model and the Lagrange multiplier method are described.

Based on the GENET model, we present a two-step transformation for converting any binary CSP into a zero-one integer constrained minimization problem in chapter 4. The first step of the transformation gives a SAT representation of the GENET network, while the second step constructs the resultant zero-one integer constrained minimization problem from the transformed SAT problem. In chapter 5, we further transform the zero-one integer constrained minimization problem into one in the real space, and apply the Lagrange multiplier method to solve the resultant problem. Although this approach is viable, it is computationally expensive.

Chapter 6 describes the discrete Lagrange multiplier method [62, 54, 53] for solving binary CSP's. A generic scheme $\mathcal{LSDL}$, which defines a class of discrete Lagrangian search algorithms, is introduced. We show that the GENET model is equivalent to an instance of $\mathcal{LSDL}$. Variants and possible extension of $\mathcal{LSDL}$ are investigated. The performance of the variants on different CSP's are also evaluated. In chapter 7, we extend $\mathcal{LSDL}$ for tackling general CSP's. The difference between the binary and the general formulation is discussed. Various experiments

are performed to evaluate our proposed formulation. Some initial results are obtained. An inadequacy of the general formulation and a possible solution are also considered.

Chapter 8 concludes the thesis by summarizing our contributions and listing some possible directions for future research.

Chapter 2

Related Work

This chapter briefly reviews some local search methods related to our research. The min-conflicts heuristic, GSAT, the breakout method, GENET, E-GENET and DLM are well-known local search methods for solving SAT and CSP's. In the following, we describe these methods according to their chronological appearances. In addition, other optimization techniques based on local search, such as simulated annealing, genetic algorithms, tabu search and integer programming, are presented at the end of this chapter.

2.1 Min-conflicts Heuristic

Based on the Guarded Discrete Stochastic (GDS) network [1], Minton *et al.* proposed a heuristic repair method for CSP's. The method starts with an initial, possibly inconsistent, assignment and continues to repair the assignment until a solution is obtained or some terminating conditions, such as CPU time limit, are met. At each point of the search, the method repairs the assignment according to the min-conflicts heuristic [39, 40], which selects a variable currently violating some constraints and assigns it a value that minimizes the number of constraint violations with ties being broken randomly.

The min-conflicts heuristic has been found to be very successful on certain

CSP's, such as the scheduling problem for the Hubble Space Telescope, the N-queens problems and graph-coloring problems [39, 40]. Experiments show that it is much better than existing backtracking tree search algorithms. However, a potential problem of the min-conflicts heuristic is that the search can settle on an assignment in which no further repair can be made. Such an assignment is usually referred to as a local minimum of the search space. The min-conflicts heuristic does not have any special mechanism to resolve this situation. It relies on random restart to bring the search out of local minima.

2.2 GSAT

GSAT [52] is a greedy local search method for solving SAT problems. The algorithm begins with a randomly generated truth assignment. It then flips the assignment of variables to maximize the total number of satisfied clauses. The process continues until a solution is found. Similar to the min-conflicts heuristic, GSAT can be trapped in a local minimum. In order to overcome this weakness, GSAT simply restarts itself after a predefined maximum number of flips are tried.

GSAT has been found to be efficient on hard SAT problems and on some CSP's, such as the N-queens problems and graph-coloring problems [52]. Various extensions to the basic GSAT algorithm include mixing GSAT with a random walk strategy [49, 51], clause weight learning [49, 13], averaging in previous assignments [49] and tabu-like move restrictions [16]. These modifications are shown to boost the performance of GSAT on certain kinds of problems.

2.3 Breakout Method

The breakout method [41], which has mechanism for escaping from local minima, is an iterative improvement method for solving CSP's. In this method, each constraint of a CSP is considered as a set of incompatible tuples. A weight, initially set to 1, is associated with each incompatible tuple. The cost of an assignment is

the sum of the weights of violated tuples in that assignment. Similar to the min-conflicts heuristic [39, 40], the breakout method minimizes the cost of assignment until it reaches a local minimum. At that point, the weights of current violating tuples are increased to allow the search to continue.

Since the breakout method modifies the cost of an assignment, it may get trapped in infinite loops. However, experiments on SAT problems and graph-coloring problems show that breakout almost always finds a solution if one exists [41].

2.4 GENET

The GENET model is a generic neural network, first proposed by Wang and Tsang [66, 60], for solving binary CSP's. In this model, a binary CSP is represented by a network. Each possible label of CSP is denoted by a label node and each incompatible tuple of a binary constraint is represented by a weighted connection. A convergence procedure, based on the min-conflicts heuristic [39, 40], is used to search for a solution. As in the breakout method [41], GENET modifies the weight of violated connections to help escaping from local maxima. This technique is referred to as the heuristic learning rule of GENET.

Davenport *et al.* [7, 6] extended GENET for handling general constraints. Three types of general constraints, namely the `illegal` constraint, the `atmost` constraint and the `notequal` constraint, are implemented. Experimental results of the hard graph-coloring problems, the randomly generated CSP's and the car sequencing problems confirm that GENET is more efficient than existing iterative improvement methods, such as GSAT and the heuristic repair method [7, 6]. A detailed description of the binary subset of GENET is given in chapter 3.

2.5 E-GENET

E-GENET [32, 69] extends the GENET model [66, 60, 7, 6] for solving general CSP's. It uses a different network architecture for problem representation. Unlike

the GENET model, each variable of a CSP is represented by a single variable node and each constraint is represented by a constraint node. The penalty value of each incompatible tuple is stored in corresponding constraint node. A convergence procedure and a heuristic learning rule similar to that in GENET are used for solution searching.

Since E-GENET induces the problem of storing a large number of penalty value in a constraint node, Lee *et al.* [32, 69] introduced different storage schemes for different types of constraints to overcome this weakness. Several optimizations [33, 69], such as the introduction of intermediate node, the new assignment scheme of initial penalty values, the concept of contribution and the new learning heuristic, are also proposed to further improve the performance. A comprehensive constraint library [34, 69], which consists of linear arithmetic constraints, the `atmost` constraint, the disjunctive constraint and a set of global constraints from CHIP [2] are constructed. The performance of E-GENET compares favorably against that of CHIP [8], a state of the art implementation of tree search algorithms, in various benchmarks, such as the N-queens problems, the graph-coloring problems, the scheduling problems, the channel assignment problems, the Hamiltonian cycle problems and the Mystery Shopper Problem [32, 33, 34, 69].

2.6 DLM

DLM [62, 54, 53] is a discrete Lagrangian-based global search method of solving SAT problems. In this method, a SAT problem is first transformed into a discrete constrained optimization problem. The discrete Lagrange multiplier method is then applied to solve the resultant optimization problem. With the help of Lagrange multipliers, DLM can escape from local minima and continue the search without restarting the entire algorithm. DLM generally outperforms the best existing methods and can achieve an order of magnitude speedup for some SAT problems [62, 54, 53]. It also gives certain success in other problems, such as the MAX-SAT problems [63, 53] and the design problem of multiplierless QMF filter

banks [64, 53].

Wu [70] further generalized the discrete Lagrange multiplier method for solving discrete optimization problems. In this extension, the necessary conditions for saddle points, and the relation between constrained local minima and saddle points of the Lagrangian function are given. Hence, a strong mathematical foundation for the discrete Lagrange multiplier method is provided.

2.7 Simulated Annealing

Simulated annealing [28] is an optimization technique inspired by the annealing process of solids. It can escape from local minima by allowing a certain amount of worsening moves. Consider an optimization problem, every possible state of the problem is associated with an energy E. In each step of simulated annealing, the algorithm displaces from current state to a random neighboring state and computes the resulting change in energy $\triangle E$. If $\triangle E \leq 0$, the new state is accepted. Otherwise, the new state is accepted with a Boltzmann probability $e^{-\triangle E/T}$ where T is a temperature parameter of the process. At high temperature T, the Boltzmann probability approaches 1 and the algorithm searches randomly. As the temperature decreases, movements which improve the quality of the search are favored. The temperature usually decreases gradually according to an annealing schedule. If the annealing schedule cools slowly enough, the algorithm is guaranteed to find a global minimum. However, this theoretical result usually requires an infinite amount of time.

Some work has been carried out on using simulated annealing to solve CSP's. Johnson *et al.* [27] investigated the feasibility of applying simulated annealing for solving graph-coloring problems. Selman and Kautz [50] compared the performance of simulated annealing and that of GSAT on the SAT problems. Since much effort expended by simulated annealing in the initial high temperature phase is wasted, simulated annealing usually takes a longer time to reach a solution.

2.8 Genetic Algorithms

Genetic algorithms [26] are heuristic search techniques for tackling combinatorial optimization problems. They are derived from the evolution processes in nature. In genetic algorithms, a population of chromosomes, which represent states of the problem, is used to explore the search space of the problem. A fitness function is associated with the population to determine how fit a chromosome is. During each generation, new chromosomes are reproduced by crossover and mutation, and added to the population. Chromosomes are selected to survive from one generation to another by a selection function. Unfit chromosomes are discarded during this selection phase. As the process proceeds, the algorithms will eventually obtain the fittest chromosome, which corresponds to the optimal solution of the problem.

Eiben *et al.* [11, 10] evaluated the performance of genetic algorithms on some CSP's, such as the N-queens problems, the graph-coloring problems, the traffic lights problems and the Zebra problems. Riff [46] developed new fitness function and genetic operator to improve the performance of genetic algorithms for solving CSP's. Warwick and Tsang apply genetic algorithms for solving the car sequencing problems [68] and the processors configuration problems [67]. Lau and Tsang [30] also introduced a mutation-based genetic algorithm to tackle processors configuration problems. Their approach is shown to be more efficient than other published techniques.

2.9 Tabu Search

Tabu search [18, 19, 20] is a sophisticated local search method that can escape from local minima. It maintains a tabu list of prohibited search states to prevent the algorithm from visiting the same search states twice. With the help of the tabu list, non-improving moves are allowed. In general, a tabu search algorithm can be realized as follows. Initially, a state of the problem is selected randomly

as the starting point of the search. This state is regarded as the best solution obtained so far. A set of states which are in the neighborhood of the current state and are not in the tabu list is collected. The best state in this set is selected as the next state of the search. If the new state improves upon the best solution found so far, it becomes the new best solution. The tabu list is also updated according to some predefined criteria. The search continues until an acceptable solution is found.

Tabu search has been applied for solving different CSP's. Some examples are the graph-coloring problems [23], the radio links frequency assignment problems [3] and the SAT problems [38].

2.10 Integer Programming

Lagrangian relaxation [17, 12] is a well-known approach for integer programming. Consider an integer linear programming problem P,

$$Z_P = \min\ \vec{c}^T\vec{x}$$

subject to

$$\mathbf{A}\vec{x} \leq \vec{b},$$

$$\mathbf{B}\vec{x} \leq \vec{d},$$

$$\vec{x} \geq \vec{0} \text{ and integral}$$

where $\vec{b}, \vec{c}, \vec{d}, \vec{x}$ are vectors, $\mathbf{A}, \mathbf{B}$ are matrices of conformable dimensions and the constraints $\mathbf{B}\vec{x} \geq \vec{d}$ have some special structure which allow the corresponding integer linear programming problem to be solved efficiently. The Lagrangian relaxation method defines a Lagrangian problem LR,

$$Z_{LR}(\vec{u}) = \min\ \vec{c}^T\vec{x} + \vec{u}^T(\mathbf{A}\vec{x} - \vec{b})$$

subject to

$$\mathbf{B}\vec{x} \leq \vec{d},$$

$$\vec{x} \geq \vec{0} \text{ and integral}$$

where $\vec{u} \geq \vec{0}$ is a vector of Lagrange multipliers. Because of the special structure of the constraints $\mathbf{B}\vec{x} \leq \vec{d}$, the resultant Lagrangian problem LR is easier to solve than the original problem P. Since the optimal value $Z_{LR}(\vec{u})$ of the Lagrangian problem LR is guaranteed to be less than or equal to the optimal value Z_P of the original problem P, the Lagrangian relaxation method can be used to provide lower bounds in branch and bound algorithms for solving the integer linear programming problem. In addition, Lagrangian relaxation can be used as a medium for selecting branching variables and choosing the next branch to explore.

Freuder [14] pointed out that there are many possible paths to constraint satisfaction. Besides backtracking, hill climbing, neural networks and genetic algorithms, integer programming is also a possible approach for solving CSP's. Rivin and Zabih [47] developed an algebraic method for solving CSP's. In their approach, a CSP is converted into an integer linear programming problem with zero-one integer variables. The constraints of the CSP is represented by a set of linear inequalities. The transformed integer programming problem is then solved by polynomial multiplication.

Chapter 3

Background

This chapter provides the background to the thesis. A local search method, called GENET, for solving CSP's is reviewed. Furthermore, a description of optimization problem is given. The classical Lagrange multiplier method for handling constrained optimization problem is also presented.

3.1 GENET

The GENET model [66, 60, 7, 6] is a connectionist architecture for solving CSP's. It consists of two components, namely a network architecture and a convergence procedure. The network architecture gives the network representation of a CSP, while the convergence procedure is an iterative improvement algorithm for solution searching. In the following, we limit our discussion to the GENET model for solving binary CSP's.

3.1.1 Network Architecture

A GENET *network* $\mathcal{N}$ [66, 60, 7, 6] is constructed by a set of label nodes and node connections. Consider a CSP (U, D, C). Each variable $i \in U$ is represented by a cluster of *label nodes* $\langle i, j \rangle$, one for each value $j \in D_i$. Since there is a one-one correspondence between a label and a label node, we use the same notation to

denote them. Each label node $\langle i, j \rangle$ is associated with an *output* $V_{\langle i,j \rangle}$, which is 1 if value j is assigned to variable i, and 0 otherwise. A label node is said to be *on* if its output is 1; otherwise, it is *off*.

A binary constraint c on variables i_1 and i_2 is represented by *weighted connections* between incompatible label nodes in clusters i_1 and i_2. Two label nodes $\langle i_1, j_1 \rangle$ and $\langle i_2, j_2 \rangle$ are connected if $i_1 = j_1$ and $i_2 = j_2$ violates c. The *weight* $W_{\langle i,j \rangle \langle k,l \rangle}$ of each connection $(\langle i, j \rangle, \langle k, l \rangle)$, initially set to -1, is always a negative integer. The weights are modified during the learning process described later.

The *input* $I_{\langle i,j \rangle}$ to a label node $\langle i, j \rangle$ is defined as the weighted sum of output of all its connected label nodes. Let $A(\mathcal{N}, \langle i, j \rangle)$ be the set of all label nodes connected to $\langle i, j \rangle$ in network $\mathcal{N}$. The input $I_{\langle i,j \rangle}$ is

$$I_{\langle i,j \rangle} = \sum_{\langle k,l \rangle \in A(\mathcal{N}, \langle i,j \rangle)} W_{\langle i,j \rangle \langle k,l \rangle} V_{\langle k,l \rangle}. \tag{3.1}$$

A *state* $\mathcal{S}$ of a GENET network $\mathcal{N}$ is a tuple $(\vec{V}, \vec{W})$, where $\vec{V} = (\ldots, V_{\langle i,j \rangle}, \ldots)$ is a vector of outputs for all label nodes $\langle i, j \rangle$ in $\mathcal{N}$ and $\vec{W} = (\ldots, W_{\langle i,j \rangle \langle k,l \rangle}, \ldots)$ is a vector of weights for all connections $(\langle i, j \rangle, \langle k, l \rangle)$. A state is *valid* if exactly one label node in each cluster is on. A *solution state* is a valid state with the input to all on label nodes being zero.

An *energy function* of a GENET network $\mathcal{N}$ and a state $\mathcal{S}$ is defined as

$$E(\mathcal{N}, \mathcal{S}) = \sum_{(\langle i,j \rangle, \langle k,l \rangle) \in \mathcal{N}} V_{\langle i,j \rangle} W_{\langle i,j \rangle \langle k,l \rangle} V_{\langle k,l \rangle}. \tag{3.2}$$

In other words, the function returns the sum of weight of all violated connections. It also gives a measure of the "goodness" of a state $\mathcal{S}$ in terms of the total weight of violated connections. Alternatively, the energy function can also be defined as the sum of input to all on label nodes [35, 36]. Let $O(\mathcal{N}, \mathcal{S})$ be the set of all on label nodes of a network $\mathcal{N}$ and a state $\mathcal{S}$. We have

$$E'(\mathcal{N}, \mathcal{S}) = \sum_{\langle i,j \rangle \in O(\mathcal{N}, \mathcal{S})} I_{\langle i,j \rangle}. \tag{3.3}$$

Since every violated connections is summed twice in (3.3), $E'(\mathcal{N}, \mathcal{S}) = 2E(\mathcal{N}, \mathcal{S})$. In subsequent discussion, definition (3.2) is adopted.

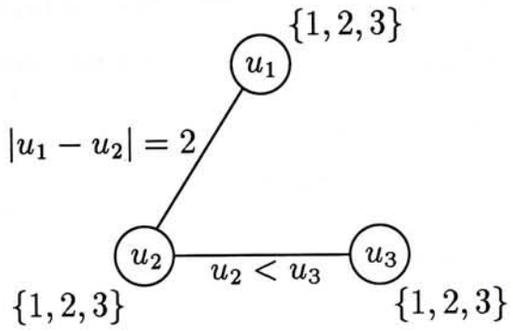


Figure 3.1: A CSP (U, D, C), where $U = \{u_1, u_2, u_3\}$, $D_{u_1} = D_{u_2} = D_{u_3} = \{1, 2, 3\}$ and $C = \{|u_1 - u_2| = 2, u_2 < u_3\}$

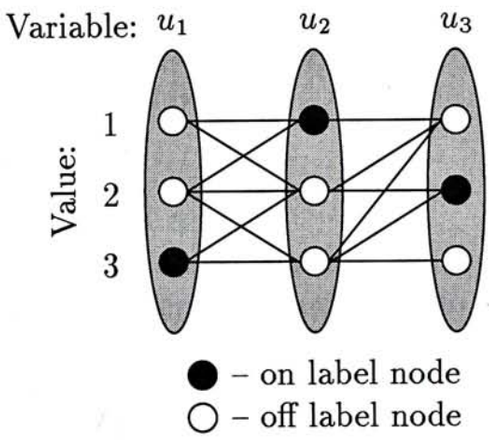


Figure 3.2: The GENET network of the CSP in Figure 3.1

For a GENET network with negative weights, $E(\mathcal{N}, \mathcal{S})$ is always non-positive. At a solution state $\mathcal{S}_0$, all constraints are satisfied. The energy $E(\mathcal{N}, \mathcal{S}_0)$ is 0, which is a global maximum value. Hence, a CSP is solved when the energy $E(\mathcal{N}, \mathcal{S})$ is equal to zero.

Figure 3.1 shows a CSP with three variables u_1, u_2, u_3, each with a domain $\{1, 2, 3\}$, and two constraints $|u_1 - u_2| = 2$ and $u_2 < u_3$. The corresponding GENET network is shown in Figure 3.2. The network consists of three clusters of label nodes, one for each of the variables u_1, u_2 and u_3. Connections are established between any two incompatible label nodes. For example, since the assignment $u_1 = 1, u_2 = 1$ violates the constraint $|u_1 - u_2| = 2$, there is a connection between

label nodes $\langle u_1, 1\rangle$ and $\langle u_2, 1\rangle$. The weights of all connections are set to -1 initially. The state illustrated, with the label nodes $\langle u_1, 3\rangle, \langle u_2, 1\rangle$ and $\langle u_3, 2\rangle$ on, represents the assignment $u_1 = 3, u_2 = 1$ and $u_3 = 2$. Since the energy of this state is zero, it is a solution state of the network.

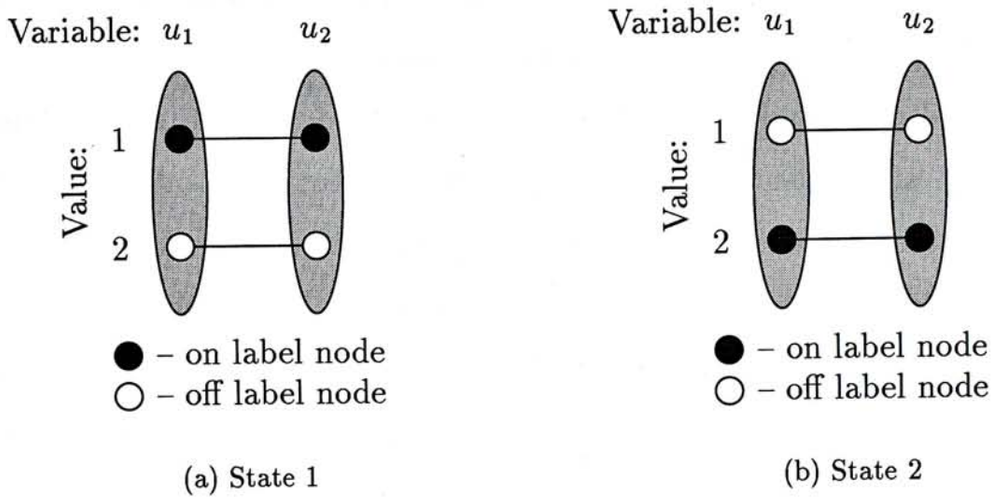
3.1.2 Convergence Procedure

The GENET *convergence procedure* [66, 60, 7, 6] outlined in Algorithm 3.1 is defined for solving CSP's. It changes the state of a GENET network continuously until a solution state is reached.

```
procedure GENET-Convergence
begin
  initialize the network to a random valid state
  loop
    {State update rule}
    for each cluster in parallel do
      calculate the input of each label nodes
      select the label node with maximum input to be on next
    end for
    if all label nodes' output remain unchanged then
      if the input to all on label nodes is zero then
        terminate and return the solution
      else
        {Heuristic learning rule}
        update all connection weights by W^{s+1}_{⟨i,j⟩⟨k,l⟩} = W^s_{⟨i,j⟩⟨k,l⟩} − V^s_{⟨i,j⟩} V^s_{⟨k,l⟩}
      end if
    end if
  end loop
end
```

Algorithm 3.1: Convergence procedure of GENET

Initially, a label node in each cluster is selected to be on randomly; others are off. This corresponds to assigning arbitrarily a value to each variable in a CSP. Next, the *state update rule* transforms the GENET network from one valid state to another by minimizing the number of constraint violations. A solution is found when all on label nodes have zero input. Effectively, the state update rule carries



(a) State 1 (b) State 2

Figure 3.3: A oscillating GENET network in synchronous update

out an optimization process for the energy function $E(\mathcal{N}, \mathcal{S})$ until a zero energy is obtained.

During the state updating process, the clusters can be updated in parallel either synchronously or asynchronously. In *synchronous* update, all clusters calculate the input and update the state of their label nodes at the same time. Alternatively, each cluster can perform input calculation and state update independently in *asynchronous* update. Synchronous update can cause oscillations between a number of states indefinitely [7, 6]. For instance, consider a GENET network with four label nodes $\langle u_1, 1\rangle$, $\langle u_1, 2\rangle$, $\langle u_2, 1\rangle$ and $\langle u_2, 2\rangle$, and two connections $(\langle u_1, 1\rangle, \langle u_2, 1\rangle)$ and $(\langle u_1, 2\rangle, \langle u_2, 2\rangle)$ shown in Figure 3.3. The weight of each connection is -1. Suppose the GENET network is in state 1 (Figure 3.3(a)), with label nodes $\langle u_1, 1\rangle$ and $\langle u_2, 1\rangle$ on, initially. In synchronous update, we calculate the inputs to each label node at the same time, and get $I_{\langle u_1,1\rangle} = -1$, $I_{\langle u_1,2\rangle} = 0$, $I_{\langle u_2,1\rangle} = -1$ and $I_{\langle u_2,2\rangle} = 0$. Hence, the network changes its state to state 2 (Figure 3.3(b)), with label nodes $\langle u_1, 2\rangle$ and $\langle u_2, 2\rangle$ on. Further state update will bring the network back to state 1 again. The whole process repeats and the GENET network oscillates between these two states indefinitely. On the other hand, in our experience, asynchronous update always find a solution if one exists. In a sequential implementation, asynchronous update can be simulated by updating

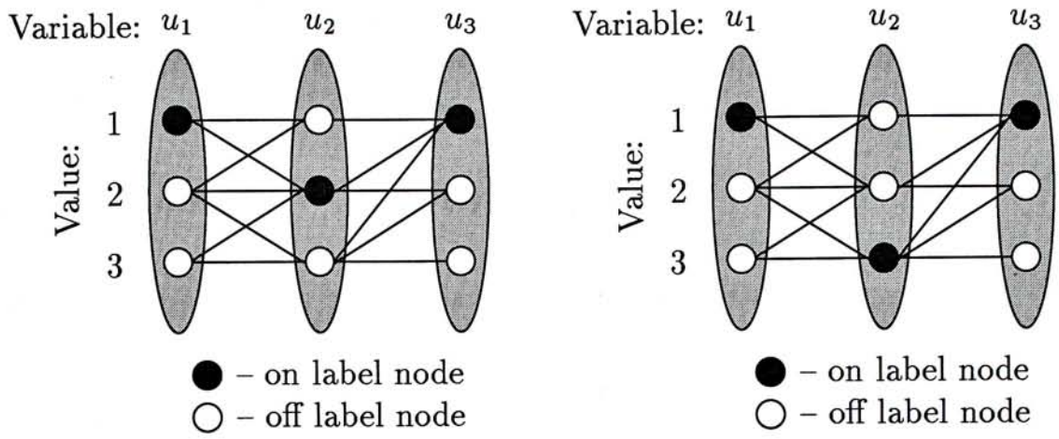
each cluster in sequence in a predefined order.

The state of the label nodes in each cluster are updated according to their inputs. Basically, the label node with the maximum input is selected to be on next. However, there could be more than one label node with the maximum input. In this case, the following heuristic rule is adopted. Let P be the set of label nodes with the maximum input. If the current on label node is in P, it remains on. Otherwise, a random label node returned by $\texttt{rand}(P)$ is selected to be on, where $\texttt{rand}(Y)$ is a function returning a random element from a set Y. Since the label node with maximum input corresponds to an assignment with fewer constraint violations, this updating strategy is a direct application of the min-conflicts heuristic [39, 40].

A GENET network can be trapped in a *local maximum*, which is a stable state in which the state updating process fails to make further improvement and yet some constraints are violated [66, 60, 7, 6]. In other words, at a local maximum $\mathcal{S}_l$, $E(\mathcal{N}, \mathcal{S}_l) < 0$ and $E(\mathcal{N}, \mathcal{S}_l) \geq E(\mathcal{N}, \mathcal{S}_n)$ for all its neighboring states $\mathcal{S}_n$. Obviously, a local maximum does not correspond to a solution of a CSP. In order to escape from a local maximum, the heuristic learning rule is used. Let an *iteration* of the convergence procedure constitutes one pass over the outermost loop of Algorithm 3.1. The *heuristic learning rule* adjusts the connection weights as follows [66, 60, 7, 6],

$$W^{s+1}_{\langle i,j\rangle\langle k,l\rangle} = W^{s}_{\langle i,j\rangle\langle k,l\rangle} - V^{s}_{\langle i,j\rangle} V^{s}_{\langle k,l\rangle} \tag{3.4}$$

where $W^{s}_{\langle i,j\rangle\langle k,l\rangle}$ is the weight of the connection $(\langle i,j\rangle, \langle k,l\rangle)$ and $V^{s}_{\langle i,j\rangle}$ is the output of the label node $\langle i,j\rangle$ in the sth iteration. This heuristic learning rule has two effects on the convergence procedure. First, weight update decreases the energy associated with the local maximum. Hence, perhaps with more than one weight updates, the local maximality is destroyed. Second, since the weights of violated connections become more negative after learning, these connections are less likely to be violated again in future state update. Note that this heuristic learning rule is similar to the breakout method [41].



(a) State 1 (b) State 2

Figure 3.4: The network convergence of GENET

Connection weight learning not only affects the local maximum, but also affects other states with constraints violated in the maximum. Hence, new local maximum may be created [7]. Furthermore, learning may block potential paths to a solution [41]. However, it is observed that this requires the constraints to interact locally in a specific "unlucky" manner and the probability of this kind of interaction for large CSP's is very small [41].

We use the GENET network of the CSP shown in Figure 3.1 to illustrate the network convergence of the GENET model. Figure 3.4 shows a state transition from state 1 to state 2 of the GENET network. State 1 (Figure 3.4(a)), with an energy -2, has the label nodes $\langle u_1, 1\rangle, \langle u_2, 2\rangle$ and $\langle u_3, 1\rangle$ on, and the weights of all connections being equal to -1. In sequential implementation, the convergence procedure, which repairs each cluster u_1, u_2 and u_3 in sequence, works as follows. Since all label nodes in cluster u_1 share the same input, label node $\langle u_1, 1\rangle$ remains on in state 2. In cluster u_2, the inputs to each label node are $I_{\langle u_2,1\rangle} = -2, I_{\langle u_2,2\rangle} = -2$ and $I_{\langle u_2,3\rangle} = -1$. Hence, label node $\langle u_2, 3\rangle$ is selected to be on next. After that, all label nodes in cluster u_3 get the same input. The state updating process ends with cluster u_3 unaltered. The resulting state shown in Figure 3.4(b) has an energy -1 and is a local maximum. Since any further state update will result

in no state change, the heuristic learning rule is activated to modify the weight $W_{\langle u_2,3\rangle\langle u_3,1\rangle}$ to -2. The stability of the GENET network is thus destroyed and the state update rule can again be applied to maximize the energy of the network.

3.2 Classical Optimization

Optimization problems belong to a class of important problems in our daily life. Many complex problems arising in business or in industry, such as decision making, resource allocation and scheduling, can be regarded as optimization problems. These problems usually require a decision maker to choose from many possible alternatives the one that yield an optimal performance. In this section, a formal description of optimization problems is given. Furthermore, the Lagrange multiplier method, a classical method for solving constrained optimization problems, is described.

3.2.1 Optimization Problems

An *optimization problem* is a problem of minimizing (or maximizing) a mathematical function of one or more variables [24, 55]. Without loss of generality, only minimization problems are discussed. In a minimization problem, the mathematical function to be minimized is known as the *objective function* of the problem. A *local minimum* is an assignment of values to the variables which gives the smallest value of the objective function among its neighborhoods. A *global minimum* is an assignment which minimizes the objective function [24, 55].

In general, any minimization problem can be classified into two different types, namely *unconstrained minimization problem* and *constrained minimization problem* [24, 55]. In unconstrained minimization problems, there are no restrictions on the values assigned to the variables. This kind of problems always have the form,

$$\min\ f(\vec{x}) \tag{3.5}$$

where $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables, "min" is the abbreviation for minimization and $f(\vec{x})$ is the objective function to be minimized.

Constrained minimization problems are represented as follows,

$$\min \ f(\vec{x}) \tag{3.6}$$

$$\text{subject to} \quad g_i(\vec{x})\{\leq, =, \geq\}0, \quad i = 1, \ldots, m \tag{3.7}$$

where $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables, "min" is the abbreviation for minimization and $f(\vec{x})$ is the objective function to be minimized. The equations and/or inequalities in (3.7) are the *constraints* of the problem, restricting the values that the variables $\vec{x}$ can take. An assignment which satisfies all constraints is said to be *feasible*; otherwise, it is *infeasible* [24, 55]. A *constrained global minimum* is a feasible assignment which minimizes the objective function of the problem. Throughout the discussion, we concentrate our attention on minimization with equality constraints (i.e. $g_i(\vec{x}) = 0, i = 1, \ldots, m$).

3.2.2 The Lagrange Multiplier Method

A minimization problem with equality constraints is formulated as,

$$\min \ f(\vec{x}) \tag{3.8}$$

$$\text{subject to} \quad g_i(\vec{x}) = 0, \quad i = 1, \ldots, m \tag{3.9}$$

where $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables. Since the equality constraints in (3.9) only reduce dimensionality, they do not establish any boundaries. A trivial way to solve the problem is to reduce it to an unconstrained minimization problem with $n - m$ variables. In other words, we first express any m variables in terms of the other $n - m$ variables by the equality constraints (3.9). Then, we substitute the result into the objective function (3.8) to eliminate the m variables. Consider an example taken from [55].

$$\min \ x_1^2 + x_2^2 + x_3^2 \tag{3.10}$$

$$\text{subject to} \quad x_1 + x_2 + x_3 - 10 = 0. \tag{3.11}$$

In this problem, we have $n = 3$ and $m = 1$. From (3.11), we get

$$x_3 = 10 - x_1 - x_2. \tag{3.12}$$

Substitute (3.12) into (3.10), we get an unconstrained minimization problem in x_1 and x_2:

$$\min\ x_1^2 + x_2^2 + (10 - x_1 - x_2)^2. \tag{3.13}$$

Subsequently, we obtain the minimum at $x_1 = x_2 = x_3 = \frac{10}{3}$.

Although the above approach is simple, the computation is very complex or impractical when the equality constraints are complicated, non-linear and the number of variables involved are large. Therefore, a more systematic and efficient method, called the *Lagrange multiplier method* [24, 55], has been developed. In the Lagrange multiplier method, the equality constraints are not considered explicitly. They are combined with the objective function to form a Lagrangian function. Consider the constrained minimization problem in (3.8 – 3.9), the *Lagrangian function* [24, 55] is defined as

$$L(\vec{x}, \vec{\lambda}) = f(\vec{x}) + \sum_{i=1}^{m} \lambda_i g_i(\vec{x}) \tag{3.14}$$

where $\vec{\lambda} = (\lambda_1, \ldots, \lambda_m)$ is a vector of *Lagrange multipliers*. The *necessary conditions* [24, 55] for constrained local minimum are

$$\nabla_{\vec{x}} L(\vec{x}, \vec{\lambda}) = 0 \tag{3.15}$$

$$\nabla_{\vec{\lambda}} L(\vec{x}, \vec{\lambda}) = 0 \tag{3.16}$$

where ∇ is the gradient operator. The conditions in (3.15 – 3.16) form a system of $n + m$ equations, linear or non-linear, with $n + m$ unknowns $\vec{x}$ and $\vec{\lambda}$. Solutions $\vec{x}$ of this system of $n + m$ equations are the constrained local minima of the original problem (3.8 – 3.9). If there is a finite number of constrained local minima, a constrained global minimum can be obtained by comparing the value of the objective function of each local minimum. Note that the set of m equality constraints (3.9) is implicitly included in condition (3.16).

Consider the previous example (3.10 – 3.11). The Lagrangian function is

$$L(x_1, x_2, x_3, \lambda) = x_1^2 + x_2^2 + x_3^2 + \lambda(x_1 + x_2 + x_3 - 10). \tag{3.17}$$

The necessary conditions are

$$\frac{\partial L(x_1, x_2, x_3, \lambda)}{\partial x_1} = 2x_1 + \lambda = 0 \tag{3.18}$$

$$\frac{\partial L(x_1, x_2, x_3, \lambda)}{\partial x_2} = 2x_2 + \lambda = 0 \tag{3.19}$$

$$\frac{\partial L(x_1, x_2, x_3, \lambda)}{\partial x_3} = 2x_3 + \lambda = 0 \tag{3.20}$$

$$\frac{\partial L(x_1, x_2, x_3, \lambda)}{\partial \lambda} = x_1 + x_2 + x_3 - 10 = 0 \tag{3.21}$$

By solving the system of equations (3.18 – 3.21), we get the constrained local minimum at $x_1 = x_2 = x_3 = \frac{10}{3}$ and the Lagrange multiplier $\lambda = -\frac{20}{3}$, which agrees with previous calculation.

3.2.3 Saddle Point of Lagrangian Function

Since the system of equations generated from the necessary conditions (3.15 – 3.16) may be very complex or highly non-linear, it can be difficult to solve them analytically. In this case, a constrained local minimum can be obtained by finding a *saddle point* $(\vec{x}^*, \vec{\lambda}^*)$ [55] of the Lagrangian function $L(\vec{x}, \vec{\lambda})$, defined by the relation,

$$L(\vec{x}^*, \vec{\lambda}) \leq L(\vec{x}^*, \vec{\lambda}^*) \leq L(\vec{x}, \vec{\lambda}^*) \tag{3.22}$$

for all $(\vec{x}^*, \vec{\lambda})$ and all $(\vec{x}, \vec{\lambda}^*)$ sufficiently close to $(\vec{x}^*, \vec{\lambda}^*)$. In other words, a saddle point is a local minimum of the Lagrangian function $L(\vec{x}, \vec{\lambda})$ in the $\vec{x}$-space and a local maximum of $L(\vec{x}, \vec{\lambda})$ in the $\vec{\lambda}$-space. The relation between a local minimum of the minimization problem with only equality constraints and a saddle point of the associated Lagrangian function is stated in the following theorem.

Theorem 3.1 (Saddle Point Theorem) [62, 54] *$\vec{x}^*$ is a local minimum of the minimization problem (3.8 – 3.9) if and only if there exists Lagrange multipliers $\vec{\lambda}^*$ such that $(\vec{x}^*, \vec{\lambda}^*)$ constitutes a saddle point of the associated Lagrangian function $L(\vec{x}, \vec{\lambda})$.*

The definition of saddle point and the saddle point theorem provide an algorithmic approach for finding a constrained local minimum. A saddle point, which corresponds to a constrained local minimum, can be identified by performing descent in the $\vec{x}$-space and ascent in the $\vec{\lambda}$-space. This method is equivalent to a dynamic system constructed with the following differential equations [43],

$$\frac{d\vec{x}}{dt} = -\nabla_{\vec{x}} L(\vec{x}, \vec{\lambda}) \tag{3.23}$$

$$\frac{d\vec{\lambda}}{dt} = \nabla_{\vec{\lambda}} L(\vec{x}, \vec{\lambda}) \tag{3.24}$$

where t is an independent time variable of the system. As the system evolves over time t, it performs gradient descent in the $\vec{x}$-space and gradient ascent in the $\vec{\lambda}$-space. At equilibrium, all gradients vanish and a saddle point of the Lagrangian function $L(\vec{x}, \vec{\lambda})$ is obtained.

Under this algorithmic point of view, the Lagrange multiplier method can be understood as follows [43]. The Lagrange multipliers $\vec{\lambda}$ are the penalties associated with the constraints and the Lagrangian function $L(\vec{x}, \vec{\lambda})$ is a penalty function. When certain constraints are violated, their corresponding Lagrange multipliers are modified to increase the penalties. These penalties will eventually force the constraints to be satisfied. At the same time, the gradient descent of $L(\vec{x}, \vec{\lambda})$ in the $\vec{x}$-space searches for a constrained local minimum.

Chapter 4

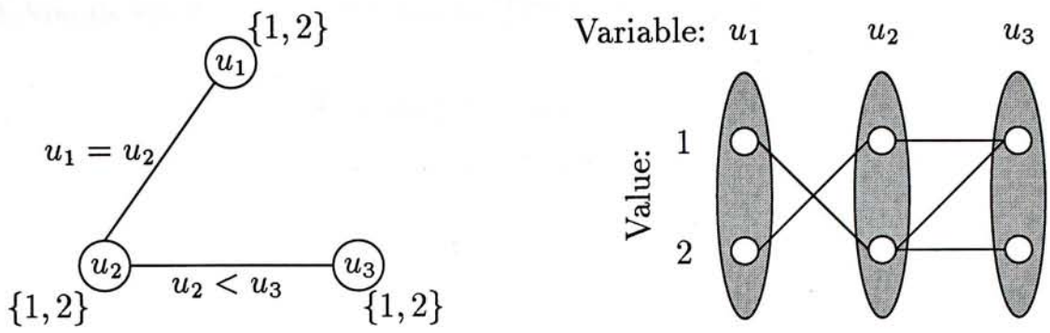
Binary CSP's as Zero-One Integer Constrained Minimization Problems

The convergence procedure of the GENET model suggests an optimization approach to constraint satisfaction. In this chapter, we present a two-step transformation for converting any CSP into a zero-one integer constrained minimization problem. The first step of the transformation converts a GENET network directly to a Boolean satisfiability (SAT) problem, while the second step derives the resultant zero-one integer constrained minimization problem from the intermediate SAT problem.

4.1 From CSP to SAT

A Boolean satisfiability (SAT) problem consists of a set of Boolean variables and a Boolean formula. Given a CSP (U, D, C). Its corresponding GENET network $\mathcal{N}$ can be viewed as a graphical representation of a SAT problem. Each label node $\langle i, j \rangle$ is associated with a Boolean variable $z_{\langle i,j \rangle}$, which is `true` if $\langle i, j \rangle$ is on and `false` otherwise.

The Boolean formula is a conjunction of two types of formulae, namely cluster



(a) The CSP

(b) The GENET network

Figure 4.1: A simple CSP and its corresponding GENET network

formulae and connection formulae. Each *cluster formula* is used to represent a cluster of label nodes, which enforces the valid states of the GENET network. The cluster formula of each cluster (variable) i

$$C_i = \bigvee_{j \in D_i} \left(z_{\langle i,j \rangle} \wedge \left(\bigwedge_{k \in D_i, k \neq j} \neg z_{\langle i,k \rangle} \right) \right) \tag{4.1}$$

ensures that exactly one label node in cluster i is on. Each connection $(\langle i,j \rangle, \langle k,l \rangle)$ in the GENET network induces a *connection formula*

$$C_{\langle i,j \rangle \langle k,l \rangle} = \neg z_{\langle i,j \rangle} \vee \neg z_{\langle k,l \rangle}, \tag{4.2}$$

which states that the label nodes $\langle i,j \rangle$ and $\langle k,l \rangle$ cannot be both on simultaneously. Hence, solving the CSP is equivalent to finding a truth assignment that satisfies the Boolean formula

$$B = \bigwedge_{i \in U} C_i \ \wedge \bigwedge_{(\langle i,j \rangle, \langle k,l \rangle) \in \mathcal{N}} C_{\langle i,j \rangle \langle k,l \rangle}. \tag{4.3}$$

Taking a simple CSP as an example, we have three variables u_1, u_2 and u_3, and two constraints. A uniform domain $D_{u_i} = \{1,2\}$ is associated with each variable u_i for all $i = 1,2,3$. The two constraints are $u_1 = u_2$ and $u_2 < u_3$. The corresponding CSP network and GENET network is shown in Figure 4.1. The induced SAT problem of the GENET network is formulated as follows. The label nodes of the GENET network is represented by the Boolean variables $z_{\langle u_1,1 \rangle}$,

$z_{\langle u_1,2\rangle}$, $z_{\langle u_2,1\rangle}$, $z_{\langle u_2,2\rangle}$, $z_{\langle u_3,1\rangle}$ and $z_{\langle u_3,2\rangle}$. The cluster formulae are

$$C_{u_1} = (z_{\langle u_1,1\rangle} \wedge \neg z_{\langle u_1,2\rangle}) \vee (\neg z_{\langle u_1,1\rangle} \wedge z_{\langle u_1,2\rangle}), \tag{4.4}$$

$$C_{u_2} = (z_{\langle u_2,1\rangle} \wedge \neg z_{\langle u_2,2\rangle}) \vee (\neg z_{\langle u_2,1\rangle} \wedge z_{\langle u_2,2\rangle}), \tag{4.5}$$

$$C_{u_3} = (z_{\langle u_3,1\rangle} \wedge \neg z_{\langle u_3,2\rangle}) \vee (\neg z_{\langle u_3,1\rangle} \wedge z_{\langle u_3,2\rangle}). \tag{4.6}$$

The connection formulae are

$$C_{\langle u_1,1\rangle\langle u_2,2\rangle} = \neg z_{\langle u_1,1\rangle} \vee \neg z_{\langle u_2,2\rangle}, \tag{4.7}$$

$$C_{\langle u_1,2\rangle\langle u_2,1\rangle} = \neg z_{\langle u_1,2\rangle} \vee \neg z_{\langle u_2,1\rangle}, \tag{4.8}$$

$$C_{\langle u_2,1\rangle\langle u_3,1\rangle} = \neg z_{\langle u_2,1\rangle} \vee \neg z_{\langle u_3,1\rangle}, \tag{4.9}$$

$$C_{\langle u_2,2\rangle\langle u_3,1\rangle} = \neg z_{\langle u_2,2\rangle} \vee \neg z_{\langle u_3,1\rangle}, \tag{4.10}$$

$$C_{\langle u_2,2\rangle\langle u_3,2\rangle} = \neg z_{\langle u_2,2\rangle} \vee \neg z_{\langle u_3,2\rangle}. \tag{4.11}$$

Combining these two types of formulae, we obtain the resultant Boolean formula of the SAT problem,

$$\begin{aligned} B = {} & C_{u_1} \wedge C_{u_2} \wedge C_{u_3} \wedge C_{\langle u_1,1\rangle\langle u_2,2\rangle} \wedge C_{\langle u_1,2\rangle\langle u_2,1\rangle} \\ & \wedge C_{\langle u_2,1\rangle\langle u_3,1\rangle} \wedge C_{\langle u_2,2\rangle\langle u_3,1\rangle} \wedge C_{\langle u_2,2\rangle\langle u_3,2\rangle}. \end{aligned} \tag{4.12}$$

4.2 From SAT to Zero-One Integer Constrained Minimization

We now complete the transformation by converting the SAT problem obtained previously to a *zero-one integer constrained minimization problem*, a constrained minimization problem with zero-one integer variables. Each Boolean variable in the SAT problem is converted to a zero-one integer variable. Since a Boolean variable can be regarded as a zero-one integer variable, we abuse notation by naming a zero-one integer variable also by its associated Boolean variable $z_{\langle i,j\rangle}$. The value of a zero-one integer variable $z_{\langle i,j\rangle}$ is 1 if value j is assigned to variable i, and 0 otherwise.

Each cluster formula C_i for all $i \in U$ is transformed to the following equation,

$$\sum_{j \in D_i} z_{\langle i,j \rangle} = 1. \tag{4.13}$$

These equations impose a space for proper instantiation of $\vec{z}$, which corresponds to valid assignments of CSP (valid state of GENET). For each connection formula $C_{\langle i,j \rangle \langle k,l \rangle}$, we define an *incompatibility function*

$$g_{\langle i,j \rangle \langle k,l \rangle}(\vec{z}) = z_{\langle i,j \rangle} z_{\langle k,l \rangle} \tag{4.14}$$

where $\vec{z} = (\ldots, z_{\langle i,j \rangle}, \ldots)$ is a vector of zero-one integer variables. The function $g_{\langle i,j \rangle \langle k,l \rangle}(\vec{z})$ returns 1 if both $z_{\langle i,j \rangle}$ and $z_{\langle k,l \rangle}$ are 1, and 0 otherwise. Hence, equating $g_{\langle i,j \rangle \langle k,l \rangle}(\vec{z})$ to 0 is equivalent to forbidding two connected label nodes $\langle i,j \rangle$ and $\langle k,l \rangle$ in the GENET network to be on at the same time. The incompatibility functions are used as indicators of constraint violations.

The resultant zero-one integer constrained minimization problem has the form,

$$\min \; N(\vec{z}) \tag{4.15}$$

subject to

$$\sum_{j \in D_i} z_{\langle i,j \rangle} = 1, \qquad \forall \, i \in U \tag{4.16}$$

$$g_{\langle i,j \rangle \langle k,l \rangle}(\vec{z}) = 0, \qquad \forall \, (\langle i,j \rangle, \langle k,l \rangle) \in \mathcal{I} \tag{4.17}$$

where $\vec{z} = (\ldots, z_{\langle i,j \rangle}, \ldots)$ is a vector of zero-one integer variables and $\mathcal{I}$ is the set of all incompatible label pairs $(\langle i,j \rangle, \langle k,l \rangle)$. Since the solution space of a CSP is defined entirely by the constraints (4.16 – 4.17), it is equal to the feasible space of the associated zero-one integer constrained minimization problem. The objective function $N(\vec{z})$ serves only to exert additional force to guide solution searching.

The objective function $N(\vec{z})$ is defined in such a way that *every solution of the CSP must correspond to a constrained global minimum of the associated zero-one integer constrained minimization problem (4.15 – 4.17).* This is called the *correspondence requirement*. In the following, we present two appropriate objective functions that fulfill the correspondence requirement. The goal of solving a

CSP is to find an assignment that satisfies all constraints. One possible objective function, adapted from Wah and Chang [61], is to count the total number of constraint violations. By measuring the total number of incompatible label pairs $(\langle i,j\rangle, \langle k,l\rangle)$ in an assignment, the objective function can be expressed as

$$\begin{aligned} N(\vec{z}) &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} z_{\langle i,j\rangle} z_{\langle k,l\rangle} \\ &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \end{aligned} \tag{4.18}$$

where $\vec{z} = (\ldots, z_{\langle i,j\rangle}, \ldots)$ is a vector of zero-one integer variables. Two properties of this objective function are stated as follows.

Property 4.1 *$\vec{z}^*$ is a constrained global minimum of the objective function $N(\vec{z})$ defined in (4.18) subjected to the constraints (4.16 – 4.17) if and only if $N(\vec{z}^*) = 0$ and all constraints (4.16 – 4.17) are satisfied.*

Proof: "$\Rightarrow$" part: If $\vec{z}^*$ is a constrained global minimum of the objective function $N(\vec{z})$ defined in (4.18) subjected to the constraints (4.16 – 4.17), all constraints (4.16 – 4.17) are satisfied. Furthermore, by the definition of the objective function $N(\vec{z})$, $N(\vec{z}^*) = 0$.

"$\Leftarrow$" part: Since the objective function $N(\vec{z})$ cannot be less than zero, $\vec{z}^*$ is a constrained global minimum if $N(\vec{z}^*) = 0$ and all constraints (4.16 – 4.17) are satisfied. □

Property 4.2 *If $\vec{z}^*$ represents a solution of the CSP, it is a constrained global minimum of the objective function $N(\vec{z})$ defined in (4.18) subjected to the constraints (4.16 – 4.17).*

Proof: If $\vec{z}^*$ represents a solution of the CSP, all constraints (4.16 – 4.17) are satisfied. In addition, by the definition of the objective function $N(\vec{z})$, $N(\vec{z}^*) = 0$. By property 4.1, $\vec{z}^*$ is a constrained global minimum of the objective function $N(\vec{z})$ subjected to the constraints (4.16 – 4.17). □

The first property suggests a condition for constrained global minima, while the second property satisfies the correspondence requirement.

Another possibility is the constant objective function

$$N(\vec{z}) = 0. \tag{4.19}$$

The objective function satisfies the correspondence requirement trivially. Basically, this trivial objective function does not help in the search of solution. We shall show later, however, that this function is related to the GENET model.

To illustrate the transformation, consider again the same CSP shown in Figure 4.1. The Boolean variables are now converted to a vector of zero-one integer variables

$$\vec{z} = (z_{\langle u_1,1\rangle}, z_{\langle u_1,2\rangle}, z_{\langle u_2,1\rangle}, z_{\langle u_2,2\rangle}, z_{\langle u_3,1\rangle}, z_{\langle u_3,2\rangle}). \tag{4.20}$$

The cluster formulae (4.4 – 4.6) become the equations

$$z_{\langle u_1,1\rangle} + z_{\langle u_1,2\rangle} = 1, \tag{4.21}$$

$$z_{\langle u_2,1\rangle} + z_{\langle u_2,2\rangle} = 1, \tag{4.22}$$

$$z_{\langle u_3,1\rangle} + z_{\langle u_3,2\rangle} = 1. \tag{4.23}$$

Similarly, the incompatibility functions

$$g_{\langle u_1,1\rangle\langle u_2,2\rangle}(\vec{z}) = z_{\langle u_1,1\rangle} z_{\langle u_2,2\rangle}, \tag{4.24}$$

$$g_{\langle u_1,2\rangle\langle u_2,1\rangle}(\vec{z}) = z_{\langle u_1,2\rangle} z_{\langle u_2,1\rangle}, \tag{4.25}$$

$$g_{\langle u_2,1\rangle\langle u_3,1\rangle}(\vec{z}) = z_{\langle u_2,1\rangle} z_{\langle u_3,1\rangle}, \tag{4.26}$$

$$g_{\langle u_2,2\rangle\langle u_3,1\rangle}(\vec{z}) = z_{\langle u_2,2\rangle} z_{\langle u_3,1\rangle}, \tag{4.27}$$

$$g_{\langle u_2,2\rangle\langle u_3,2\rangle}(\vec{z}) = z_{\langle u_2,2\rangle} z_{\langle u_3,2\rangle}, \tag{4.28}$$

are obtained from the connection formulae (4.7 – 4.11). The equations (4.21 – 4.23) and the incompatibility functions (4.24 – 4.28) are the constraints of the zero-one integer constrained minimization problem. The transformation is completed by choosing either (4.18) or (4.19) as the objective function. Hence, solving the CSP now becomes finding a constrained global minimum of the associated zero-one integer constrained minimization problem.

Chapter 5

A Continuous Lagrangian Approach for Solving Binary CSP's

In this chapter, we show how to use the Lagrange multiplier method [24, 55] to solve zero-one integer constrained minimization problems transformed from CSP's. Since the gradient of the Lagrangian function is defined over the real space only, the Lagrange multiplier method cannot be applied directly. We further transform the zero-one integer constrained minimization problem into a real constrained minimization problem, and apply the Lagrange multiplier method to the real problem [4]. A simple experiment is also presented to evaluate the feasibility of this approach.

5.1 From Integer Problems to Real Problems

The zero-one integer constrained minimization problem (4.15 – 4.17) associated with a CSP (U, D, C) can be further transformed into a real constrained minimization problem. Each integer variable $z_{\langle i,j \rangle}$ is converted to a real variable $x_{\langle i,j \rangle}$, which can take any value in the interval $(-\infty, +\infty)$. Among all possible values, only 0 and 1 are feasible for each variable $x_{\langle i,j \rangle}$. This integral restriction is imposed

by the following equality constraints [4]

$$x_{\langle i,j\rangle}(x_{\langle i,j\rangle} - 1) = 0, \quad \forall\, i \in U, j \in D_i. \tag{5.1}$$

A real variable $x_{\langle i,j\rangle}$ is 1 if and only if the corresponding zero-one integer variable $z_{\langle i,j\rangle}$ is 1. Similarly, $x_{\langle i,j\rangle}$ is 0 if and only if $z_{\langle i,j\rangle}$ equals 0.

The constraints (4.16 – 4.17) are converted to their real counterparts. The equations (4.16), which ensure valid assignment of CSP, are now replaced by

$$-\prod_{j\in D_i}(1 - x_{\langle i,j\rangle}) + \sum_{j\in D_i} x_{\langle i,j\rangle} = 1, \quad \forall\, i \in U. \tag{5.2}$$

Since, in the real space, there exist $x_{\langle i,j\rangle} \neq 0$ and $x_{\langle i,j\rangle} \neq 1$ for all $j \in D_i$ such that $\sum_{j\in D_i} x_{\langle i,j\rangle} = 1$, the extra term $-\prod_{j\in D_i}(1-x_{\langle i,j\rangle})$ is introduced to guarantee that only one value can be assigned to each variable of the CSP. Note that although the constraints (5.1) already enforce the real variable $x_{\langle i,j\rangle}$ to be either 0 or 1, this extra term can provide additional force to guide the search. Furthermore, since the extra term itself is not enough to ensure the valid assignment of CSP, it is not considered as a separate constraint. The incompatibility function $g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z})$ becomes

$$\tilde{g}_{\langle i,j\rangle\langle k,l\rangle}(\vec{x}) = x_{\langle i,j\rangle}x_{\langle k,l\rangle}, \quad \forall\, (\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I} \tag{5.3}$$

where $\mathcal{I}$ is the set of all incompatible label pairs $(\langle i,j\rangle, \langle k,l\rangle)$. Similar to its integer counterpart, $\tilde{g}_{\langle i,j\rangle\langle k,l\rangle}(\vec{x})$ returns 0 only when either $x_{\langle i,j\rangle}$ or $x_{\langle k,l\rangle}$ are 0. However, when a constraint is violated, $\tilde{g}_{\langle i,j\rangle\langle k,l\rangle}(\vec{x})$ returns a non-zero, possibly negative, value.

The resultant real constrained minimization problem is

$$\min\ \tilde{N}(\vec{x}) \tag{5.4}$$

subject to

$$x_{\langle i,j\rangle}(x_{\langle i,j\rangle} - 1) = 0, \quad \forall\, i \in U, j \in D_i \tag{5.5}$$

$$-\prod_{j\in D_i}(1 - x_{\langle i,j\rangle}) + \sum_{j\in D_i} x_{\langle i,j\rangle} = 1, \quad \forall\, i \in U \tag{5.6}$$

$$\tilde{g}_{\langle i,j\rangle\langle k,l\rangle}(\vec{x}) = 0, \quad \forall\, (\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I} \tag{5.7}$$

where $\tilde{N}(\vec{x})$ is a converted objective function such that $\vec{x}^*$ *is a constrained global minimum of the real constrained minimization problem (5.4 – 5.7) if and only if its corresponding zero-one integer variables* $\vec{z}^*$ *is a constrained global minimum of the associated zero-one integer constrained minimization problem (4.15 – 4.17).* We called this the *equivalence requirement*. Similar to the zero-one integer constrained minimization problem (4.15 – 4.17), the solution space of a CSP is equal to the feasible space of the associated real constrained minimization problem (5.4 – 5.7).

In order to fulfill the equivalence requirement, the objection function defined in (4.18) is converted to

$$\begin{aligned} \tilde{N}(\vec{x}) &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \left(x_{\langle i,j\rangle}x_{\langle k,l\rangle}\right)^2 \\ &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \left(\tilde{g}_{\langle i,j\rangle\langle k,l\rangle}(\vec{x})\right)^2 \end{aligned} \tag{5.8}$$

This transformation ensures that $\tilde{N}(\vec{x})$ is always non-negative. Hence, any $\vec{x}$ that satisfies the constraints (5.5 – 5.7) is a constrained global minimum. On the other hand, the constant objective function (4.19) does not require any modification.

Based on the equivalence requirement, the relation between a CSP and its associated real constrained minimization problem is stated in the following theorem.

Theorem 5.1 *If* $\vec{x}^*$ *is a vector of real variables represents a solution of the CSP, then* $\vec{x}^*$ *is a constrained global minimum of the associated real constrained minimization problem (5.4 – 5.7).*

Proof: Since there is a one-one correspondence between real variables $\vec{x}$ and zero-one integer variables $\vec{z}$, if the real variables $\vec{x}^*$ represents a solution of the CSP, its corresponding zero-one integer variables $\vec{z}^*$ also represents the same solution of the CSP. By the correspondence requirement, $\vec{z}^*$ is a constrained global minimum of the zero-one integer constrained minimization problem (4.15 – 4.17) transformed from the CSP. Thus, by the equivalence requirement, $\vec{x}^*$ is a constrained global minimum of the associated real constrained minimization problem (5.4 – 5.7). □

Consider the same CSP in Figure 4.1 as an example. The corresponding real

constrained minimization is

$$\min \ \tilde{N}(\vec{x}) \tag{5.9}$$

subject to

$$x_{\langle u_1,1\rangle}(x_{\langle u_1,1\rangle} - 1) = 0 \tag{5.10}$$

$$x_{\langle u_1,2\rangle}(x_{\langle u_1,2\rangle} - 1) = 0 \tag{5.11}$$

$$x_{\langle u_2,1\rangle}(x_{\langle u_2,1\rangle} - 1) = 0 \tag{5.12}$$

$$x_{\langle u_2,2\rangle}(x_{\langle u_2,2\rangle} - 1) = 0 \tag{5.13}$$

$$x_{\langle u_3,1\rangle}(x_{\langle u_3,1\rangle} - 1) = 0 \tag{5.14}$$

$$x_{\langle u_3,2\rangle}(x_{\langle u_3,2\rangle} - 1) = 0 \tag{5.15}$$

$$-(1 - x_{\langle u_1,1\rangle})(1 - x_{\langle u_1,2\rangle}) + x_{\langle u_1,1\rangle} + x_{\langle u_1,2\rangle} = 1 \tag{5.16}$$

$$-(1 - x_{\langle u_2,1\rangle})(1 - x_{\langle u_2,2\rangle}) + x_{\langle u_2,1\rangle} + x_{\langle u_2,2\rangle} = 1 \tag{5.17}$$

$$-(1 - x_{\langle u_3,1\rangle})(1 - x_{\langle u_3,2\rangle}) + x_{\langle u_3,1\rangle} + x_{\langle u_3,2\rangle} = 1 \tag{5.18}$$

$$\tilde{g}_{\langle u_1,1\rangle\langle u_2,2\rangle}(\vec{x}) = x_{\langle u_1,1\rangle}x_{\langle u_2,2\rangle} = 0 \tag{5.19}$$

$$\tilde{g}_{\langle u_1,2\rangle\langle u_2,1\rangle}(\vec{x}) = x_{\langle u_1,2\rangle}x_{\langle u_2,1\rangle} = 0 \tag{5.20}$$

$$\tilde{g}_{\langle u_2,1\rangle\langle u_3,1\rangle}(\vec{x}) = x_{\langle u_2,1\rangle}x_{\langle u_3,1\rangle} = 0 \tag{5.21}$$

$$\tilde{g}_{\langle u_2,2\rangle\langle u_3,1\rangle}(\vec{x}) = x_{\langle u_2,2\rangle}x_{\langle u_3,1\rangle} = 0 \tag{5.22}$$

$$\tilde{g}_{\langle u_2,2\rangle\langle u_3,2\rangle}(\vec{x}) = x_{\langle u_2,2\rangle}x_{\langle u_3,2\rangle} = 0 \tag{5.23}$$

where $\tilde{N}(\vec{x})$ is the objective function defined in either (5.8) or (4.19), equations (5.10 – 5.15) are the integral restrictions, equations (5.16 – 5.18) are the constraints for valid assignments and equations (5.19 – 5.23) are the constraints for the incompatibility functions.

5.2 The Lagrange Multiplier Method

As the CSP is now transformed into a real constrained minimization problem (5.4 – 5.7), the Lagrange multiplier method [24, 55] can be used to solve it. The

Lagrangian function is expressed as follows,

$$\begin{aligned} L(\vec{x},\vec{\alpha},\vec{\beta},\vec{\gamma}) &= \tilde{N}(\vec{x}) + \sum_{i\in U, j\in D_i} \alpha_{\langle i,j\rangle}\left[x_{\langle i,j\rangle}(x_{\langle i,j\rangle}-1)\right] \\ &\quad + \sum_{i\in U}\beta_i\left[-\prod_{j\in D_i}(1-x_{\langle i,j\rangle}) + \sum_{j\in D_i} x_{\langle i,j\rangle} - 1\right] \\ &\quad + \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \gamma_{\langle i,j\rangle\langle k,l\rangle}\tilde{g}_{\langle i,j\rangle\langle k,l\rangle}(\vec{x}) \end{aligned} \tag{5.24}$$

where $\vec{\alpha} = (\ldots, \alpha_{\langle i,j\rangle}, \ldots)$, $\vec{\beta} = (\ldots, \beta_i, \ldots)$ and $\vec{\gamma} = (\ldots, \gamma_{\langle i,j\rangle\langle k,l\rangle}, \ldots)$ are vectors of Lagrange multipliers.

According to the saddle point theorem, finding a constrained global minimum of the original minimization problem (5.4 – 5.7) is equivalent to finding a saddle point of the Lagrangian function (5.24). Hence, the following dynamic system is constructed:

$$\frac{d\vec{x}}{dt} = -\nabla_{\vec{x}} L(\vec{x},\vec{\alpha},\vec{\beta},\vec{\gamma}) \tag{5.25}$$

$$\frac{d\vec{\alpha}}{dt} = \nabla_{\vec{\alpha}} L(\vec{x},\vec{\alpha},\vec{\beta},\vec{\gamma}) \tag{5.26}$$

$$\frac{d\vec{\beta}}{dt} = \nabla_{\vec{\beta}} L(\vec{x},\vec{\alpha},\vec{\beta},\vec{\gamma}) \tag{5.27}$$

$$\frac{d\vec{\gamma}}{dt} = \nabla_{\vec{\gamma}} L(\vec{x},\vec{\alpha},\vec{\beta},\vec{\gamma}) \tag{5.28}$$

where t is an independent time variable. As the system evolves over time t, it descends in the $\vec{x}$-space and ascends in the space of Lagrange multipliers. When the system converges, all gradients vanish. Hence, a saddle point of $L(\vec{x},\vec{\alpha},\vec{\beta},\vec{\gamma})$, which corresponds to a constrained global minimum of (5.4 – 5.7), is obtained. Since a constrained global minimum of (5.4 – 5.7) must satisfy all constraints (5.5 – 5.7), we get a solution of the associated CSP.

5.3 Experiment

In order to verify the feasibility of this approach, an experiment on the N-queens problems is performed. The N-queens problem states that N queens are placed

N	No. of Differential Equations	CPU Time (sec)
4	88	0.49
5	165	1.50
6	278	5.70
7	434	14.84
8	640	52.19
9	903	124.05
10	1230	267.83

Table 5.1: Results of continuous Lagrangian approach on the N-queens problems

onto an $N \times N$ chessboard such that no two queens attack each other. In this experiment, the objective function defined in (5.8) is used. The resultant dynamic system (5.25 – 5.28) is solved using the *Livermore Solver for Ordinary Differential Equation (LSODE)*, a Fortran package of ODEPACK [25] for solving differential equations. Benchmark results are taken on a SUN Ultra SPARCstation.

Table 5.1 summaries the results of the experiment. In this table, the first column corresponds to the number of queens in the problem, the second column represents the number of differential equations of the dynamic system, and the third column gives the average CPU time in seconds over 5 runs. From the results, we find that the number of differential equations and the CPU time grows exponentially as the problem size increases. In addition, since an originally discrete problem is transformed into a real problem, the computation becomes more expensive [62, 54]. Hence, the performance is several order of magnitudes worse than existing constraint satisfaction techniques. In other words, the continuous Lagrangian approach is not a feasible technique for solving binary CSP's. In the next chapter, we investigate a discrete Lagrangian approach for binary CSP's.

Chapter 6

A Discrete Lagrangian Approach for Solving Binary CSP's

Shang and Wah [62, 54, 53] extended the existing Lagrange multiplier method to the discrete Lagrange multiplier method and apply it to solve SAT problems. In this chapter, we adopt this discrete Lagrange multiplier method to tackle the resultant zero-one integer constrained minimization problems obtained from the transformation of CSP's. Based on the discrete Lagrange multiplier method, we propose $\mathcal{LSDL}$, a generic discrete Lagrangian search scheme with five degrees of freedom. The GENET model is shown to be an instance of the $\mathcal{LSDL}$ framework. Dual viewpoints of GENET, as a heuristic repair method and as a discrete Lagrange multiplier method, enable us to investigate GENET variants from both perspectives. Experimental results confirm that our best variant is always more efficient than the reconstructed GENET.

6.1 The Discrete Lagrange Multiplier Method

The discrete Lagrange multiplier method [62, 54, 53] for the zero-one integer constrained minimization problem (4.15 – 4.17) transformed from the CSP (U, D, C) is described as follows. Similar to the continuous case, the Lagrangian function

$L(\vec{z}, \vec{\lambda})$ is

$$L(\vec{z}, \vec{\lambda}) = N(\vec{z}) + \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \lambda_{\langle i,j\rangle\langle k,l\rangle} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \tag{6.1}$$

where $\vec{\lambda} = (\ldots, \lambda_{\langle i,j\rangle\langle k,l\rangle}, \ldots)$ is a vector of Lagrange multipliers. Note that the constraints defined by (4.16), which serve only to define valid assignments of CSP, are not included in the Lagrangian function. The constraints will be incorporated in the discrete gradient operator discussed below.

A constrained minimum of the zero-one integer constrained minimization problem (4.15 – 4.17) can be obtained by finding a saddle point of the Lagrangian function $L(\vec{z}, \vec{\lambda})$. As in the continuous case, a *saddle point* $(\vec{z}^*, \vec{\lambda}^*)$ [62, 54, 53] of the Lagrangian function $L(\vec{z}, \vec{\lambda})$ is defined by the condition

$$L(\vec{z}^*, \vec{\lambda}) \leq L(\vec{z}^*, \vec{\lambda}^*) \leq L(\vec{z}, \vec{\lambda}^*) \tag{6.2}$$

for all $(\vec{z}^*, \vec{\lambda})$ and $(\vec{z}, \vec{\lambda}^*)$ sufficiently close to $(\vec{z}^*, \vec{\lambda}^*)$. In other words, a saddle point $(\vec{z}^*, \vec{\lambda}^*)$ of the Lagrangian function $L(\vec{z}, \vec{\lambda})$ is a minimum of $L(\vec{z}, \vec{\lambda})$ in the $\vec{z}$-space and a maximum of $L(\vec{z}, \vec{\lambda})$ in the $\vec{\lambda}$-space. The relationship between a constrained minimum of an integer constrained minimization problem and a saddle point of its associated Lagrangian function is established by the discrete saddle point theorem, which is restated as follows.

Theorem 6.1 (Discrete Saddle Point Theorem) [70] *A vector of integer variables $\vec{z}^*$ is a constrained minimum of the integer constrained minimization problem*

$$\begin{aligned} &\min\ f(\vec{z}) \\ &\textit{subject to} \quad g_i(\vec{z}) = 0, \quad i = 1, \ldots, m \end{aligned}$$

with $g_i(\vec{z})$, for all $i = 1, \ldots, m$, is non-negative for all possible value of $\vec{z}$ if and only if there exists Lagrange multipliers $\vec{\lambda}^*$ such that $(\vec{z}^*, \vec{\lambda}^*)$ constitutes a saddle point of the corresponding Lagrangian function $L(\vec{z}, \vec{\lambda}) = f(\vec{z}) + \sum_{i=1}^{m} \lambda_i g_i(\vec{z})$.

Since the incompatibility function $g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z})$, for all $(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I}$, of the zero-one integer constrained minimization problem (4.15 – 4.17) are always non-negative, the discrete saddle point theorem is applicable to the zero-one integer

constrained minimization problem (4.15 – 4.17). Note that, under this theorem, $L(\vec{z}^*, \vec{\lambda})$ is always equal to $L(\vec{z}^*, \vec{\lambda}^*)$. Hence, any point $(\vec{z}^*, \vec{\lambda}')$ with $\vec{\lambda}' \geq \vec{\lambda}^*$ is also a saddle point of the Lagrangian function $L(\vec{z}, \vec{\lambda})$.

A saddle point of the Lagrangian function $L(\vec{z}, \vec{\lambda})$ can be obtained by performing descent in the discrete variable space of $\vec{z}$ and ascent in the Lagrange multiplier space of $\vec{\lambda}$ [43]. Instead of using differential equations, the discrete Lagrange multiplier method uses difference equations [62, 54, 53]

$$\vec{z}^{s+1} = \vec{z}^s - \triangle_{\vec{z}} L(\vec{z}^s, \vec{\lambda}^s) \tag{6.3}$$

$$\vec{\lambda}^{s+1} = \vec{\lambda}^s + \vec{g}(\vec{z}^s) \tag{6.4}$$

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

In general, the discrete gradient operator is not unique. Any operator that can perform descent in the $\vec{z}$-space can be used. We give one such operator as follows. Let m_i be the size of the domain of variable i. Given a vector of zero-one integer variables $\vec{z} = (\ldots, z_{\langle i,j\rangle}, \ldots)$, we define the *zero-one projection operator* π^i for all $i \in U$,

$$\pi^i(\vec{z}) = (z_{\langle i,v_1\rangle}, \ldots, z_{\langle i,v_j\rangle}, \ldots, z_{\langle i,v_{m_i}\rangle}), \tag{6.5}$$

which gives the *ith-component* of $\vec{z}$. Hence, π^i returns the vector of zero-one integer variables corresponding to variable i in U. The *ith partial discrete gradient operator* ∂^i for all $i \in U$ is defined as

$$\partial^i L(\vec{z}, \vec{\lambda}) = \pi^i(\vec{z}) - \pi^i(\vec{z}') \tag{6.6}$$

if the following conditions hold

- X is a set of vectors of zero-one integer variables such that $\forall \vec{x} \in X$,

$$\left(\sum_{j \in D_i} x_{\langle i,j\rangle} = 1\right) \wedge \left(\forall k \neq i \in U\ \forall l \in D_k\ x_{\langle k,l\rangle} = z_{\langle k,l\rangle}\right)$$
$$\wedge\ \forall \vec{z}''\left[\left(\sum_{j \in D_i} z''_{\langle i,j\rangle} = 1 \wedge \forall k \neq i \in U\ \forall l \in D_k\ z''_{\langle k,l\rangle} = z_{\langle k,l\rangle}\right)\right.$$
$$\left.\Rightarrow L(\vec{x}, \vec{\lambda}) \leq L(\vec{z}'', \vec{\lambda})\right]$$

- $\vec{z}'$ is selected from X by

$$\vec{z}' = \begin{cases} \vec{z}, & \text{if } \vec{z} \in X \\ \texttt{rand}(X), & \text{otherwise} \end{cases}$$

where $\texttt{rand}(Y)$ returns a random element from a set Y.

The ith partial discrete gradient operator $\partial^i L(\vec{z}, \vec{\lambda})$ returns a *differential vector* $\vec{d}$ for the ith component of $\vec{z}$ by (1) computing a set X of vectors $\vec{x}$ that cause the most reduction in the value of the Lagrangian function and (2) selecting a vector $\vec{z}'$ from X. Note that the selection process in (2) is based on that of the state update rule of GENET with the same random selection function $\texttt{rand}(Y)$. Note also that the constraints defined in (4.16) are incorporated in the partial discrete gradient operators ∂^i, for all $i \in U$, enforcing the sum of all $z_{\langle i,j \rangle}$ for each variable i to be 1. If $\partial^i L(\vec{z}, \vec{\lambda}) = \vec{0}$, there is no change in the ith component of the vector $\vec{z}$. The corresponding discrete gradient operator $\triangle_{\vec{z}}$ is represented by the equations

$$\pi^i(\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})) = \partial^i L(\vec{z}, \vec{\lambda}), \qquad \forall\, i \in U. \tag{6.7}$$

When $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$, either a saddle point or a *stationary point*, at which the update of $\vec{z}$ terminates, is reached.

The Lagrange multipliers $\vec{\lambda}$ are updated according to the incompatibility functions. If an incompatible tuple is violated, its corresponding incompatibility function returns 1 and the Lagrange multiplier is incremented accordingly. In this formulation, the Lagrange multipliers $\vec{\lambda}$ are non-decreasing.

A generic discrete Lagrangian search procedure $\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$ for solving the zero-one integer constrained minimization problems transformed from CSP's is given in Algorithm 6.1. The $\mathcal{LSDL}$ (pronounced as "Lisdal") procedure performs local search using the discrete Lagrange multiplier method. It has five degrees of freedom, namely (N) the objective function, $(\triangle_{\vec{z}})$ the discrete gradient operator, $(I_{\vec{z}})$ how the integer vector $\vec{z}$ is initialized, $(I_{\vec{\lambda}})$ how the Lagrange multipliers $\vec{\lambda}$ are initialized and $(U_{\vec{\lambda}})$ when to update the Lagrange multipliers $\vec{\lambda}$. Where appropriate, we annotate the algorithm with the parameters in brackets

procedure $\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$
begin
  $(I_{\vec{z}})$ initialize the value of $\vec{z}$
  $(I_{\vec{\lambda}})$ initialize the value of $\vec{\lambda}$
  while (N) $L(\vec{z}, \vec{\lambda}) - N(\vec{z}) > 0$ ($\vec{z}$ is not a solution) **do**
    $(\triangle_{\vec{z}})$ update $\vec{z}$: $\vec{z} \leftarrow \vec{z} - \triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})$
    if $(U_{\vec{\lambda}})$ condition for updating $\vec{\lambda}$ holds **then**
      update $\vec{\lambda}$: $\vec{\lambda} \leftarrow \vec{\lambda} + \vec{g}(\vec{z})$
    end if
  end while
end

Algorithm 6.1: The $\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$ procedure

to show where the parameters take effect. The role of each parameter is discussed in the next section.

6.2 Parameters of $\mathcal{LSDL}$

$\mathcal{LSDL}$ defines a general scheme for a class of algorithms based on the discrete Lagrange multiplier method. By instantiating $\mathcal{LSDL}$ with different parameters, different Lagrangian search algorithms with different efficiency are obtained. In this section, we discuss the various parameters of $\mathcal{LSDL}$ in details.

6.2.1 Objective Function

The objective function $N(\vec{z})$ is one of the degrees of freedom of the $\mathcal{LSDL}$ algorithm. As stated before, any function that satisfies the correspondence requirement can be used. However, a good objective function can direct the search towards the solution region more efficiently [65]. Two possible objective functions, presented in chapter 4, are summarized as follows. First, since the goal of solving a CSP is to find an assignment that satisfies all constraints, the objective

function, defined in (4.18),

$$\begin{aligned} N(\vec{z}) &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} z_{\langle i,j\rangle} z_{\langle k,l\rangle} \\ &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \end{aligned}$$

where $\mathcal{I}$ is the set of incompatible tuples, reflects the total number of violated tuples. Second, the constant objective function

$$N(\vec{z}) = 0$$

can also be used.

6.2.2 Discrete Gradient Operator

The discrete gradient operator $\triangle_{\vec{z}}$, which performs gradient descent in the $\vec{z}$-space, is not unique. One possible discrete gradient operator is defined in (6.6 – 6.7). This operator performs gradient descent in all variables in the CSP at once. The situation is similar to that of synchronous update in the GENET convergence procedure. In practice, this can also lead to oscillation. We define another discrete gradient operator, the effect of which is specified by the following pseudo-code,

for each variable $i \in U$ **do**
  update $\pi^i(\vec{z})$: $\pi^i(\vec{z}) \leftarrow \pi^i(\vec{z}) - \partial^i L(\vec{z}, \vec{\lambda})$ (6.8)
end for

where ∂^i is the partial discrete gradient operator defined in (6.6). This new operator corresponds to the updating strategy used in most sequential implementations of GENET.

Another possible discrete gradient operator is defined as

$$\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{z} - \vec{z}' \tag{6.9}$$

if the following conditions are satisfied

- X is a set of vectors of zero-one integer variables such that $\forall \vec{x} \in X$,

$$\exists i \in U \Big(\sum_{j \in D_i} x_{\langle i,j \rangle} = 1 \wedge \forall k \neq i \in U \; \forall l \in D_k \; x_{\langle k,l \rangle} = z_{\langle k,l \rangle} \Big)$$
$$\wedge \quad \forall \vec{z}'' \Big[\exists m \in U \Big(\sum_{n \in D_m} z''_{\langle m,n \rangle} = 1 \wedge \forall p \neq m \in U \; \forall q \in D_p \; z''_{\langle p,q \rangle} = z_{\langle p,q \rangle} \Big)$$
$$\Rightarrow L(\vec{x}, \vec{\lambda}) \leq L(\vec{z}'', \vec{\lambda}) \Big]$$

- $\vec{z}'$ is selected from X by

$$\vec{z}' = \begin{cases} \vec{z}, & \text{if } \vec{z} \in X \\ \texttt{rand}(X), & \text{otherwise} \end{cases}$$

  where $\texttt{rand}(Y)$ returns a random element from a set Y.

The discrete gradient operator computes a set X of zero-one integer vectors $\vec{x}$ which reduce the Lagrangian function most, and returns a differential vector by selecting a vector $\vec{z}'$ from X according to the state update rule of GENET. Since each zero-one integer vector $\vec{x}$ in the set X can have at most one component $\pi^i(\vec{x})$, for some $i \in U$, being different from the current value of $\vec{z}$, only one variable of the CSP is updated by this discrete gradient operator. When $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$, there is no change in $\vec{z}$. Note that this new discrete gradient operator is similar to the one defined in DLM [62, 54, 53] for solving the SAT problems.

6.2.3 Integer Variables Initialization

A good initial assignment of the zero-one integer variables $\vec{z}$ can speed up search. As in most local search techniques, the simplest way is to initialize the zero-one integer variables $\vec{z}$ randomly in such a way that the constraints (4.16) are satisfied. On the other hand, Minton *et al.* [40] suggest that a greedily generated initial assignment can boost the performance of the search. Morris [41] points out that a greedy initialization can generally shorten the time required to reach the first local minimum. In this case, the initialization procedure iterates through each component $\pi^i(\vec{z})$ of the zero-one integer vector $\vec{z}$, and selects the assignment which conflicts with the fewest previous selections.

6.2.4 Lagrange Multipliers Initialization

Similar to the initialization of integer variables, the Lagrange multipliers $\vec{\lambda}$ can also be initialized arbitrarily. Since the update of Lagrange multipliers is non-decreasing, in general, any non-negative number can be used as the initial value. One possible way is to initialize all Lagrange multipliers to 1. In this case, all incompatible tuples have the same initial penalty. Another possibility is to initialize each Lagrange multiplier differently. For example, different initial values can be used to reflect the relative importance of constraints in the CSP [33]. If a constraint is known to be more important than the others, its associated Lagrange multipliers can be assigned a larger initial value.

6.2.5 Condition for Updating Lagrange Multipliers

Unlike the continuous case, the updating frequency of the Lagrange multipliers $\vec{\lambda}$ can affect the performance of the discrete Lagrange multiplier method [62, 54, 53]. Thus, the condition for updating the Lagrange multipliers is left unspecified in $\mathcal{LSDL}$. The Lagrange multipliers can be updated either (1) at each iteration of the outermost while loop, or (2) when $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$. Note that the former condition is a direct application of the strategy used in the continuous case while the later corresponds to Morris's breakout method [41].

6.3 A Lagrangian Reconstruction of GENET

In this section, we show how we can reconstruct GENET using our discrete Lagrangian approach. Given a CSP (U, D, C). The two-step transformation establishes a one-one correspondence between the GENET network of (U, D, C) and the associated zero-one integer constrained minimization problem of (U, D, C). The GENET convergence procedure (Algorithm 3.1) can be obtained by instantiating $\mathcal{LSDL}$ with proper parameters. This instance of $\mathcal{LSDL}$, denoted by $\mathcal{LSDL}$(GENET), has the following parameters:

- N: the constant objective function defined in (4.19),
- $\triangle_{\vec{z}}$: the discrete gradient operator defined in (6.8),
- $I_{\vec{z}}$: the zero-one integer vector $\vec{z}$ is initialized randomly, provided that the initial values correspond to a valid state in GENET,
- $I_{\vec{\lambda}}$: the values of Lagrange multipliers $\vec{\lambda}$ are all initialized to 1, and
- $U_{\vec{\lambda}}$: the Lagrange multiplier $\vec{\lambda}$ are updated when $\partial^i L(\vec{z}, \vec{\lambda}) = \vec{0}$ for all $i \in U$.

In the following, we prove the equivalence between $\mathcal{LSDL}$(GENET) and the GENET convergence procedure. Recall that a state $\mathcal{S}$ of a GENET network $\mathcal{N}$ is a tuple $(\vec{V}, \vec{W})$, where $\vec{V} = (\ldots, V_{\langle i,j \rangle}, \ldots)$ is a vector of outputs for all label nodes $\langle i,j \rangle$ in $\mathcal{N}$ and $\vec{W} = (\ldots, W_{\langle i,j \rangle \langle k,l \rangle}, \ldots)$ is a vector of weights for all connections $(\langle i,j \rangle, \langle k,l \rangle)$ in $\mathcal{N}$. Based on the state update rule of the convergence procedure of GENET and the definition of the discrete gradient operator (6.8), we derive the following lemma.

Lemma 6.1 *Consider a CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and zero-one integer constrained minimization problem. Suppose both GENET and $\mathcal{LSDL}$(GENET) use the same random selection function* `rand`(Y)*, and, in the sth iteration, $\vec{V}^s = \vec{z}^s$ and $\vec{W}^s = -\vec{\lambda}^s$, and $\pi^i(\vec{z}^{s+1}) = \pi^i(\vec{z}^s) - \partial^i L(\vec{z}^s, \vec{\lambda}^s)$. In the update of variable i from the sth to the $(s+1)$st iteration,*

$$V^{s+1}_{\langle i,j \rangle} = 1 \text{ and } V^{s+1}_{\langle i,k \rangle} = 0, \forall k \neq j \in D_i \Leftrightarrow z^{s+1}_{\langle i,j \rangle} = 1 \text{ and } z^{s+1}_{\langle i,k \rangle} = 0, \forall k \neq j \in D_i.$$

Proof: Consider updating cluster i of the GENET network $\mathcal{N}$ from the sth to the $(s+1)$st iteration. Let $A(\mathcal{N}, \langle i,j \rangle)$ be the set of all label nodes connected to $\langle i,j \rangle$ in GENET network $\mathcal{N}$, and L_i be the set of all label nodes in cluster i in GENET network $\mathcal{N}$. Furthermore, let $\vec{z}^s_{i,j}$ be the zero-one integer variable vector in the sth iteration with $z^s_{\langle i,j \rangle} = 1$, $z^s_{\langle i,k \rangle} = 0$ for all $k \neq j \in D_i$, and $z^s_{\langle u,v \rangle}$ unchanged for all $u \neq i \in U$ and $v \in D_u$.

$$V^{s+1}_{\langle i,j \rangle} = 1 \text{ and } V^{s+1}_{\langle i,k \rangle} = 0, \forall k \neq j \in D_i$$

$$
\begin{aligned}
&\Leftrightarrow \quad I^s_{\langle i,j\rangle} \geq I^s_{\langle i,k\rangle}, \qquad \forall k \neq j \in D_i \\
&\Leftrightarrow \quad \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,j\rangle)} W^s_{\langle i,j\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} \geq \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,k\rangle)} W^s_{\langle i,k\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle}, \quad \forall k \neq j \in D_i \\
&\Leftrightarrow \quad 1 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,j\rangle)} W^s_{\langle i,j\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} + \sum_{l \neq j \in D_i} \left(0 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,l\rangle)} W^s_{\langle i,l\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} \right) \\
&\qquad + \sum_{\substack{(\langle a,b\rangle,\langle c,d\rangle)\in\mathcal{N} \\ \langle a,b\rangle,\langle c,d\rangle \notin L_i}} V^s_{\langle a,b\rangle} W^s_{\langle a,b\rangle\langle c,d\rangle} V^s_{\langle c,d\rangle} \quad \geq \\
&\qquad 1 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,k\rangle)} W^s_{\langle i,k\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} + \sum_{l \neq k \in D_i} \left(0 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,l\rangle)} W^s_{\langle i,l\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} \right) \\
&\qquad + \sum_{\substack{(\langle a,b\rangle,\langle c,d\rangle)\in\mathcal{N} \\ \langle a,b\rangle,\langle c,d\rangle \notin L_i}} V^s_{\langle a,b\rangle} W^s_{\langle a,b\rangle\langle c,d\rangle} V^s_{\langle c,d\rangle}, \quad \forall k \neq j \in D_i \\
&\Leftrightarrow \quad 1 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,j\rangle)} -\lambda^s_{\langle i,j\rangle\langle u,v\rangle} z^s_{\langle u,v\rangle} + \sum_{l \neq j \in D_i} \left(0 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,l\rangle)} -\lambda^s_{\langle i,l\rangle\langle u,v\rangle} z^s_{\langle u,v\rangle} \right) \\
&\qquad + \sum_{\substack{(\langle a,b\rangle,\langle c,d\rangle)\in\mathcal{N} \\ \langle a,b\rangle,\langle c,d\rangle \notin L_i}} z^s_{\langle a,b\rangle}(-\lambda^s_{\langle a,b\rangle\langle c,d\rangle}) z^s_{\langle c,d\rangle} \quad \geq \\
&\qquad 1 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,k\rangle)} -\lambda^s_{\langle i,k\rangle\langle u,v\rangle} z^s_{\langle u,v\rangle} + \sum_{l \neq k \in D_i} \left(0 \times \sum_{\langle u,v\rangle \in A(\mathcal{N},\langle i,l\rangle)} -\lambda^s_{\langle i,l\rangle\langle u,v\rangle} z^s_{\langle u,v\rangle} \right) \\
&\qquad + \sum_{\substack{(\langle a,b\rangle,\langle c,d\rangle)\in\mathcal{N} \\ \langle a,b\rangle,\langle c,d\rangle \notin L_i}} z^s_{\langle a,b\rangle}(-\lambda^s_{\langle a,b\rangle\langle c,d\rangle}) z^s_{\langle c,d\rangle}, \quad \forall k \neq j \in D_i \\
&\Leftrightarrow \quad L(\vec{z}^s_{i,j}, \vec{\lambda}^s) \leq L(\vec{z}^s_{i,k}, \vec{\lambda}^s), \quad \forall k \neq j \in D_i \\
&\Leftrightarrow \quad \partial^i L(\vec{z}^s, \vec{\lambda}^s) = \pi^i(\vec{z}^s) - \pi^i(\vec{z}^s_{i,j})
\end{aligned}
$$

Since both GENET and $\mathcal{LSDL}$(GENET) use the same random selection function $\mathbf{rand}(Y)$, and $\pi^i(\vec{z}^{s+1}) = \pi^i(\vec{z}^s) - \partial^i L(\vec{z}^s, \vec{\lambda}^s)$, we have

$$V^{s+1}_{\langle i,j\rangle} = 1 \text{ and } V^{s+1}_{\langle i,k\rangle} = 0, \forall k \neq j \in D_i \Leftrightarrow z^{s+1}_{\langle i,j\rangle} = 1 \text{ and } z^{s+1}_{\langle i,k\rangle} = 0, \forall k \neq j \in D_i.$$

□

The lemma states that when updating variable i from the sth iteration to the $(s+1)$st iteration, the same value $j \in D_i$ will be selected by both GENET and $\mathcal{LSDL}$(GENET). By applying the lemma repeatedly to each variable $i \in U$, we get the following corollary.

Corollary 6.1 *Consider a CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and zero-one integer constrained minimization problem. Suppose both GENET and $\mathcal{LSDL}$(GENET) use the same random selection function* `rand`(Y)*, and, in the sth iteration, $\vec{V}^s = \vec{z}^s$ and $\vec{W}^s = -\vec{\lambda}^s$, and $\pi^i(\vec{z}^{s+1}) = \pi^i(\vec{z}^s) - \partial^i L(\vec{z}^s, \vec{\lambda}^s)$ for all $i \in U$. In the $(s+1)$st iteration, we have*

$$\vec{V}^{s+1} = \vec{z}^{s+1}.$$

Proof: According to lemma 6.1, for each variable $i \in U$, we have

$$V^{s+1}_{\langle i,j\rangle} = 1 \text{ and } V^{s+1}_{\langle i,k\rangle} = 0, \forall k \neq j \in D_i \Leftrightarrow z^{s+1}_{\langle i,j\rangle} = 1 \text{ and } z^{s+1}_{\langle i,k\rangle} = 0, \forall k \neq j \in D_i.$$

Hence, $\vec{V}^{s+1} = \vec{z}^{s+1}$. □

The relation between the weights $\vec{W}$ of the GENET network $\mathcal{N}$ and the Lagrange multipliers $\vec{\lambda}$ of $\mathcal{LSDL}$(GENET) is given by the following lemma.

Lemma 6.2 *Consider a CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and zero-one integer constrained minimization problem. Suppose, in the sth iteration, $\vec{V}^s = \vec{z}^s$, $\vec{W}^s = -\vec{\lambda}^s$, and, in the $(s+1)$st iteration, $\vec{V}^{s+1} = \vec{z}^{s+1}$.*

$$\vec{W}^{s+1} = -\vec{\lambda}^{s+1}.$$

Proof: We consider the lemma in two different cases. First, if $\vec{V}^{s+1} \neq \vec{V}^s$ and $\vec{z}^{s+1} \neq \vec{z}^s$, the conditions for updating the weights $\vec{W}$ and the Lagrange multiplier $\vec{\lambda}$ are false. Therefore,

$$\vec{W}^{s+1} = \vec{W}^s = -\vec{\lambda}^s = -\vec{\lambda}^{s+1}.$$

Second, if $\vec{V}^{s+1} = \vec{V}^s$ and $\vec{z}^{s+1} = \vec{z}^s$, then, for each $(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{N}$,

$$\begin{aligned}
W^{s+1}_{\langle i,j\rangle\langle k,l\rangle} &= W^s_{\langle i,j\rangle\langle k,l\rangle} - V^s_{\langle i,j\rangle} V^s_{\langle k,l\rangle} \\
&= -\lambda^s_{\langle i,j\rangle\langle k,l\rangle} - z^s_{\langle i,j\rangle} z^s_{\langle k,l\rangle} \\
&= -\lambda^s_{\langle i,j\rangle\langle k,l\rangle} - g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}^s) \\
&= -\left(\lambda^s_{\langle i,j\rangle\langle k,l\rangle} + g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}^s)\right) \\
&= -\lambda^{s+1}_{\langle i,j\rangle\langle k,l\rangle}
\end{aligned}$$

Combining these two cases, we get $\vec{W}^{s+1} = -\vec{\lambda}^{s+1}$. □

Now, a simple application of corollary 6.1 and lemma 6.2 results in the following theorem, which establishes the equivalence of the GENET convergence procedure and $\mathcal{LSDL}$(GENET).

Theorem 6.2 *Consider a CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and zero-one integer constrained minimization problem. Suppose both GENET and $\mathcal{LSDL}$(GENET) use the same random selection function* `rand`(Y) *and they share the same initial state. For all iteration s, $\vec{V}^s = \vec{z}^s$ and $\vec{W}^s = -\vec{\lambda}^s$. If they terminate, they return the same solution for the CSP.*

Proof: We prove the theorem by mathematical induction. Initially, at $s = 0$, since both GENET and $\mathcal{LSDL}$(GENET) share the same initial state,

$$\vec{V}^0 = \vec{z}^0.$$

Furthermore, since $\vec{W}^0 = -\vec{1}$ and $\vec{\lambda}^0 = \vec{1}$,

$$\vec{W}^0 = -\vec{\lambda}^0.$$

Therefore, the theorem is true at $s = 0$.

Now, suppose at $s = t$, $\vec{V}^t = \vec{z}^t$ and $\vec{W}^t = -\vec{\lambda}^t$. By corollary 6.1 and lemma 6.2, we have

$$\vec{V}^{t+1} = \vec{z}^{t+1} \quad \text{and} \quad \vec{W}^{t+1} = -\vec{\lambda}^{t+1}$$

at $s = t + 1$.

By mathematical induction, the theorem is true for all iteration s. Consequently, if both GENET and $\mathcal{LSDL}$(GENET) terminate, they return the same solution for the CSP. □

Based on this theorem, we get the following two corollaries. The first corollary states the relation between the energy of GENET and the Lagrangian function of $\mathcal{LSDL}$(GENET), while the second corollary gives the terminating properties of GENET and $\mathcal{LSDL}$(GENET).

Corollary 6.2 *Consider a CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and zero-one integer constrained minimization problem. The energy of GENET is equal to the negative of the Lagrangian function of $\mathcal{LSDL}$(GENET). Mathematically,*

$$E(\mathcal{N}, \mathcal{S}) = -L(\vec{z}, \vec{\lambda}).$$

Proof: Consider the GENET network $\mathcal{N}$ and its associated zero-one integer constrained minimization problem. Let $\mathcal{I}$ be the set of all incompatible tuples.

$$\begin{aligned} E(\mathcal{N}, \mathcal{S}) &= \sum_{(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{N}} V_{\langle i,j\rangle} W_{\langle i,j\rangle\langle k,l\rangle} V_{\langle k,l\rangle} \\ &= \sum_{(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I}} z_{\langle i,j\rangle} \left(-\lambda_{\langle i,j\rangle\langle k,l\rangle}\right) z_{\langle k,l\rangle} \\ &= -\sum_{(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I}} \lambda_{\langle i,j\rangle\langle k,l\rangle} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \\ &= -L(\vec{z}, \vec{\lambda}) \end{aligned}$$

□

Corollary 6.3 *Consider a CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and zero-one integer constrained minimization problem. GENET terminates if and only if $\mathcal{LSDL}$(GENET) terminates.*

Proof: Consider the GENET network $\mathcal{N}$ and its associated zero-one integer constrained minimization problem. Let $O(\mathcal{N}, \mathcal{S})$ be the set of all on label nodes of the GENET network $\mathcal{N}$ and a state $\mathcal{S}$.

$$\begin{aligned} \text{GENET terminates} &\Leftrightarrow I_{\langle i,j\rangle} = 0, \quad \forall \langle i,j\rangle \in O(\mathcal{N}, \mathcal{S}) \\ &\Leftrightarrow E(\mathcal{N}, \mathcal{S}) = 0 \\ &\Leftrightarrow L(\vec{z}, \vec{\lambda}) = 0 \\ &\Leftrightarrow \mathcal{LSDL}(\text{GENET}) \text{ terminates} \end{aligned}$$

□

Similar results can be proven if, in $\mathcal{LSDL}$, we use instead the objective function $N(\vec{z})$ defined in (4.18) and initialize $\vec{\lambda}$ to $\vec{0}$. If, however, we use $N(\vec{z})$ defined in (4.18) and initialize $\vec{\lambda}$ to $\vec{1}$, the Lagrangian function becomes

$$\begin{aligned} L(\vec{z},\vec{\lambda}) &= \sum_{((i,j),(k,l))\in\mathcal{I}} z_{\langle i,j\rangle} z_{\langle k,l\rangle} + \sum_{((i,j),(k,l))\in\mathcal{I}} \lambda_{\langle i,j\rangle\langle k,l\rangle} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \\ &= \sum_{((i,j),(k,l))\in\mathcal{I}} \left(1+\lambda_{\langle i,j\rangle\langle k,l\rangle}\right) g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \end{aligned} \tag{6.10}$$

where $\mathcal{I}$ is the set of all incompatible tuples. As a result, we have

$$\vec{W} = -\left(\vec{1}+\vec{\lambda}\right). \tag{6.11}$$

This version of $\mathcal{LSDL}$ is equivalent to GENET with all connection weights initialized to -2 instead of -1.

6.4 Experiments

Three experimental settings are used to evaluate our discrete Lagrangian approach. First, $\mathcal{LSDL}$(GENET) is compared with GENET to verify if it has the same fast convergence behavior as other GENET implementations. Second, several experiments are performed to evaluate the effect of different parameters of $\mathcal{LSDL}$. Parameters which give good performance in most CSP's are identified. Third, our best variant $\mathcal{LSDL}$(MAX) is tested against $\mathcal{LSDL}$(GENET).

The N-queens problems, a set of hard graph-coloring problems from the DIMACS archive [27], an instance of exceptionally hard problems (EHP's) [45], and a set of randomly generated CSP's are used in our experiments. Results of all $\mathcal{LSDL}$ implementations are taken on a SUN SPARCstation 10 model 40. Unless otherwise specified, the unbracketed and the bracketed timing results represent the CPU time in seconds for the average and the median of 10 runs respectively. Note that the results of all $\mathcal{LSDL}$ implementations are the time for finding one solution only.

N	PROCLANN Average CPU Time (sec)	$\mathcal{LSDL}$(GENET) Average (Median) CPU Time (sec)	I-$\mathcal{LSDL}$(GENET) Average (Median) CPU Time (sec)
10	0.065	0.005 (0.000)	0.027 (0.025)
20	0.218	0.003 (0.000)	0.288 (0.283)
30	0.637	0.020 (0.017)	1.005 (1.000)
40	2.145	0.028 (0.033)	2.388 (2.383)
50	4.719	0.040 (0.033)	4.627 (4.617)
60	7.711	0.068 (0.067)	8.002 (7.975)
70	13.292	0.090 (0.092)	12.818 (12.792)
80	20.629	0.178 (0.175)	19.698 (19.692)
90	33.150	0.642 (0.633)	28.283 (28.267)
100	152.795	1.078 (1.108)	39.348 (39.400)
110	261.026	1.588 (1.583)	52.585 (52.608)
120	144.709	2.033 (2.058)	68.907 (68.950)

Table 6.1: Results of $\mathcal{LSDL}$(GENET) on the N-queens problems

6.4.1 Evaluation of $\mathcal{LSDL}$(GENET)

The performance of $\mathcal{LSDL}$(GENET) is evaluated on the N-queens problems, a set of hard graph-coloring problems, and an instance of EHP's designed to defeat tree search algorithms. These experiments have two purposes. First, they serve to verify if $\mathcal{LSDL}$(GENET) exhibits the same fast convergence behavior as GENET as reported in the literature. Second, they serve as a control to compare against other variants. Wherever possible, the performance figures of two implementations of GENET are provided.

Table 6.1 shows the results of 10- to 120-queens problems for $\mathcal{LSDL}$(GENET), I-$\mathcal{LSDL}$(GENET) and PROCLANN [36], a constraint logic programming language with GENET as the constraint solver. I-$\mathcal{LSDL}$(GENET) and PROCLANN are incremental implementations of $\mathcal{LSDL}$(GENET) and GENET respectively. In these implementations, everytime new constraints are generated and posted into the constraint store, the $\mathcal{LSDL}$ procedure or the GENET solver is activated to solve the partial problem containing all constraints available in the constraint store. The benchmarking results of PROCLANN are the average of 10 runs on a

Nodes	Colors	GENET Median CPU Time	PROCLANN Average CPU Time	$\mathcal{LSDL}$(GENET) Average (Median) CPU Time
125	17	2.6 hr	2.3 hr	4.7 min (3.7 min)
125	18	23 sec	2.5 min	4.5 sec (2.9 sec)
250	15	4.2 sec	1.1 hr	0.418 sec (0.408 sec)
250	29	1.1 hr	4.6 hr	14.6 min (15.7 min)

Table 6.2: Results of $\mathcal{LSDL}$(GENET) on the hard graph-coloring problems

SUN SPARCstation 10 model 30. Since PROCLANN generates the constraints of a CSP from a program, the timing results of I-$\mathcal{LSDL}$(GENET) include the time for reading constraints to compensate the difference. The performance of $\mathcal{LSDL}$(GENET) is order of magnitudes better than that of PROCLANN. The large difference in performance is due to the fact that $\mathcal{LSDL}$(GENET) collects all constraints in the problem initialization phase and activates the Lagrangian search algorithm once. Hence, much work is saved. This effect is more prominent as the size of the problems increase. On the other hand, I-$\mathcal{LSDL}$(GENET) shows similar performance as that of PROCLANN. The difference in performance on the large problems is due to the difference in their interface.

The graph-coloring problem is to assign a color from a fixed set of colors to each vertex of the graph such that no two adjacent vertices share the same color. A set of hard graph-coloring problems from the DIMACS archive [27] are tested. Timing results of the hard graph-coloring problems for $\mathcal{LSDL}$(GENET), PROCLANN [36] and GENET described in [7] are shown in Table 6.2. Again the results of PROCLANN are the average of 10 runs taken on a SUN SPARCstation 10 model 30. The results of GENET, quoted from [7], represented the median of 10 runs collected on a SUN Sparc Classic, which is about 2 to 3 times slower than a SPARCstation 10 model 40. The hard graph-coloring problems are relatively small in size and the constraints are available to PROCLANN all at once. Therefore, most time is spent in actual searching in all implementations. Both I-$\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(GENET) have the same performance since they

are equivalent when all constraints are available and posted to the constraint store at once. $\mathcal{LSDL}$(GENET) improves substantially on both implementations of GENET. This might be related to the difference in the implementations. In $\mathcal{LSDL}$(GENET), the contribution of each incompatibility function $g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z})$ to the Lagrangian function $L(\vec{z},\vec{\lambda})$ is calculated incrementally during the update of the zero-one integer variables $\vec{z}$. When a zero-one integer variable $z_{\langle i,j\rangle}$ is updated, the contributions of previously violated incompatibility functions $g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z})$ are subtracted from the Lagrangian function $L(\vec{z},\vec{\lambda})$, while the contributions of newly violated incompatibility functions are added to the Lagrangian function. Hence, a large amount of computation is saved.

Prosser [45] designed a specific instance of EHP's to defeat forward-checking algorithm with dynamic variable ordering (fc-dvo) [44], which always chooses variables with the smallest current domain. The problem consists of 50 variables, each with a domain $\{1,2,3,4,5,6,7,8\}$. The associated constrained graph is connected. Each pair of connected variables contains 4 incompatible tuples. The tightness [60] of the problem is 0.06. Unlike tree search algorithms [44], the performance of $\mathcal{LSDL}$(GENET) seems not to be affected by this EHP. $\mathcal{LSDL}$(GENET) solves the EHP with 2.2 iterations in 0.002 seconds on average over 10 runs. It is much better than that of PROCLANN, which required 2448 convergence cycles in 3.24 seconds to solve the same problem.

In conclusion, $\mathcal{LSDL}$(GENET) exhibits similar fast convergence behavior to GENET. The difference in performance may be due to the difference in implementations.

6.4.2 Evaluation of Various Parameters

The experiments are used to evaluate the effect of various parameters of $\mathcal{LSDL}$. In each experiment, the parameter under test is varied in the $\mathcal{LSDL}$ implementation. Other parameters remains the same as that of $\mathcal{LSDL}$(GENET). The N-queens problems, the set of hard graph-coloring problems from the DIMACS archive [27], and a set of randomly generated CSP's, ranging from 100 to 150

N	$N_{\{zero\}}$ CPU Time (sec)	$N_{\{violation\}}$ CPU Time (sec)
10	0.005 (0.000)	0.000 (0.000)
20	0.003 (0.000)	0.003 (0.000)
30	0.020 (0.017)	0.013 (0.017)
40	0.028 (0.033)	0.027 (0.033)
50	0.040 (0.033)	0.045 (0.050)
60	0.068 (0.067)	0.058 (0.058)
70	0.090 (0.092)	0.090 (0.083)
80	0.178 (0.175)	0.147 (0.150)
90	0.642 (0.633)	0.657 (0.642)
100	1.078 (1.108)	1.098 (1.067)
110	1.588 (1.583)	1.522 (1.533)
120	2.033 (2.058)	2.068 (2.142)

Table 6.3: Results of $N_{\{zero\}}$ and $N_{\{violation\}}$ on the N-queens problems

variables, are used. Each tight random CSP has a uniform domain of size 10, constraint tightness [60], the proportion of pairs of values which are inconsistent in a binary constraint, 0.15, and constraint density [60], the proportion of pairs of variables which have a constraint between them, varying from 0.165 to 0.25.

Objective Function

This experiment investigates the effect of the objection function $N(\vec{z})$. Two objective functions are tested. They are

- $N_{\{zero\}}$: the objective function defined in (4.19), and
- $N_{\{violation\}}$: the objective function defined in (4.18).

Experimental results for the N-queens problems are reported in Table 6.3. The results of $N_{\{violation\}}$ is similar to that of $N_{\{zero\}}$ since only a small amount of CPU time is spent on solution searching. For the results of the hard graph-coloring problems shown in Table 6.4, except the problem with 125 nodes and 17 colors, the objective function $N_{\{violation\}}$ improves the performance of $\mathcal{LSDL}$. The poor performance for the problem with 125 nodes and 17 colors is due to an

Nodes	Colors	$N_{\{zero\}}$ CPU Time	$N_{\{violation\}}$ CPU Time
125	17	4.7 min (3.7 min)	8.0 min (3.1 min)
125	18	4.5 sec (2.9 sec)	1.2 sec (1.0 sec)
250	15	0.418 sec (0.408 sec)	0.415 sec (0.417 sec)
250	29	14.6 min (15.7 min)	11.9 min (11.8 min)

Table 6.4: Results of $N_{\{zero\}}$ and $N_{\{violation\}}$ on the hard graph-coloring problems

Problem	$N_{\{zero\}}$ CPU Time (sec)	$N_{\{violation\}}$ CPU Time (sec)
rcsp-100-10-25-15	2.742 (2.483)	1.852 (1.475)
rcsp-110-10-23-15	11.253 (10.992)	6.950 (2.108)
rcsp-120-10-21-15	7.983 (5.433)	5.375 (4.083)
rcsp-130-10-19-15	9.077 (8.308)	4.322 (3.500)
rcsp-140-10-18-15	11.000 (10.058)	9.167 (7.975)
rcsp-150-10-16.5-15	7.935 (8.692)	2.458 (2.100)

Table 6.5: Results of $N_{\{zero\}}$ and $N_{\{violation\}}$ on the tight random CSP's

exceptionally bad timing result in one of the runs. Table 6.5 shows the results of $N_{\{zero\}}$ and $N_{\{violation\}}$ on the tight random CSP's. In this set of problems, the objective function $N_{\{violation\}}$ improves the performance substantially.

Since the effect of an objective function is to exert additional force to guide the search, a good objective function can improve the overall performance of $\mathcal{LSDL}$. From the experiment, we find that $N_{\{violation\}}$ usually gives better performance than $N_{\{zero\}}$.

Discrete Gradient Operator

The efficiency of two discrete gradient operators are evaluated. The operators

- $\triangle_{\vec{z}\{many\}}$: the one defined in (6.8), and
- $\triangle_{\vec{z}\{one\}}$: the one defined in (6.9)

N	$\triangle_{\vec{z}\{many\}}$ Iter.	$\triangle_{\vec{z}\{many\}}$ CPU Time (sec)	$\triangle_{\vec{z}\{one\}}$ Iter.	$\triangle_{\vec{z}\{one\}}$ CPU Time (sec)
10	51.6	0.005 (0.000)	13.8	0.000 (0.000)
20	14.8	0.003 (0.000)	62.2	0.005 (0.000)
30	51.2	0.020 (0.017)	50.2	0.020 (0.017)
40	38.5	0.028 (0.033)	42.2	0.027 (0.025)
50	40.3	0.040 (0.033)	62.5	0.063 (0.067)
60	35.1	0.068 (0.067)	69.0	0.100 (0.100)
70	34.9	0.090 (0.092)	58.3	0.135 (0.133)
80	31.2	0.178 (0.175)	84.8	0.280 (0.275)
90	30.6	0.642 (0.633)	89.5	0.777 (0.767)
100	49.2	1.078 (1.108)	86.5	1.378 (1.375)
110	43.2	1.588 (1.583)	80.1	1.958 (1.942)
120	41.9	2.033 (2.058)	129.2	2.752 (2.683)

Table 6.6: Results of $\triangle_{\vec{z}\{many\}}$ and $\triangle_{\vec{z}\{one\}}$ on the N-queens problems

Nodes	Colors	$\triangle_{\vec{z}\{many\}}$ Iter.	$\triangle_{\vec{z}\{many\}}$ CPU Time	$\triangle_{\vec{z}\{one\}}$ Iter.	$\triangle_{\vec{z}\{one\}}$ CPU Time
125	17	708.4 k	4.7 min (3.7 min)	1737.3 k	8.1 min (5.0 min)
125	18	6125.8	4.5 sec (2.9 sec)	6119.3	2.4 sec (1.6 sec)
250	15	24.0	0.418 sec (0.408 sec)	455.1	1.122 sec (1.100 sec)
250	29	337.5 k	14.6 min (15.7 min)	1060.8 k	21.0 min (19.9 min)

Table 6.7: Results of $\triangle_{\vec{z}\{many\}}$ and $\triangle_{\vec{z}\{one\}}$ on the hard graph-coloring problems

are tested. Table 6.6 shows the CPU time and the average number of iterations of the two discrete gradient operators on the N-queens problems. The performance of $\triangle_{\vec{z}\{one\}}$ is slightly worse than that of $\triangle_{\vec{z}\{many\}}$. However, since the N-queens problems are relatively easy for $\mathcal{LSDL}$, the results are not very significant. The timing results and the average number of iterations of the hard graph-coloring problem and the tight random CSP's are shown in Tables 6.7 and 6.8 respectively. Except some problem instances, the discrete gradient operator $\triangle_{\vec{z}\{one\}}$ is not as efficient as $\triangle_{\vec{z}\{many\}}$. This difference in performance can be accounted as follows. Although both discrete gradient operators perform the same amount of work in each iteration, only one variable is updated by $\triangle_{\vec{z}\{one\}}$. On the other hand, in

Problem	$\Delta_{\vec{z}\{many\}}$ Iter.	$\Delta_{\vec{z}\{many\}}$ CPU Time (sec)	$\Delta_{\vec{z}\{one\}}$ Iter.	$\Delta_{\vec{z}\{one\}}$ CPU Time (sec)
rcsp-100-10-25-15	7245.5	2.742 (2.483)	18512.4	3.640 (3.325)
rcsp-110-10-23-15	33634.4	11.253 (10.992)	74856.4	14.545 (14.158)
rcsp-120-10-21-15	23589.6	7.983 (5.433)	38491.3	8.260 (7.433)
rcsp-130-10-19-15	22570.8	9.077 (8.308)	39774.2	8.687 (5.983)
rcsp-140-10-18-15	27130.4	11.000 (10.058)	80936.9	18.877 (21.342)
rcsp-150-10-16.5-15	17389.3	7.935 (8.692)	27473.5	6.945 (6.975)

Table 6.8: Results of $\Delta_{\vec{z}\{many\}}$ and $\Delta_{\vec{z}\{one\}}$ on the tight random CSP's

each iteration, $\Delta_{\vec{z}\{many\}}$ can update more than one variable. Hence, as reflected in the benchmarking results, $\Delta_{\vec{z}\{many\}}$ usually requires more iterations to solve a problem.

Although the performance of different discrete gradient operators seems to be problem dependent, our experiments suggest that $\Delta_{\vec{z}\{many\}}$ is likely to perform better than $\Delta_{\vec{z}\{one\}}$, which is similar to the one defined in DLM [62, 54, 53] for solving SAT problem.

Integer Variables Initialization

Two schemes for initializing the zero-one integer variables are investigated in this experiment. They are

- $I_{\vec{z}\{random\}}$: randomly initialize the zero-one integer vector $\vec{z}$, provided that the set of constraints (4.16) is satisfied, and

- $I_{\vec{z}\{greedy\}}$: greedily initialize the zero-one integer vector $\vec{z}$ according to the following procedure [40]: initialize each component $\pi^i(\vec{z})$ of the zero-one integer vector $\vec{z}$ one by one, and select the assignment which gives the fewest violations against previous selections.

The results of the N-queens problems, the hard graph-coloring problems and the tight random CSP's are given in Tables 6.9, 6.10 and 6.11 respectively. The greedy initial assignment $I_{\vec{z}\{greedy\}}$ gives us better performance in most of our test

N	$I_{\vec{z}\{random\}}$ CPU Time (sec)	$I_{\vec{z}\{greedy\}}$ CPU Time (sec)
10	0.005 (0.000)	0.002 (0.000)
20	0.003 (0.000)	0.012 (0.017)
30	0.020 (0.017)	0.013 (0.017)
40	0.028 (0.033)	0.017 (0.017)
50	0.040 (0.033)	0.032 (0.033)
60	0.068 (0.067)	0.038 (0.033)
70	0.090 (0.092)	0.065 (0.067)
80	0.178 (0.175)	0.113 (0.117)
90	0.642 (0.633)	0.483 (0.475)
100	1.078 (1.108)	0.777 (0.775)
110	1.588 (1.583)	1.055 (1.058)
120	2.033 (2.058)	1.447 (1.383)

Table 6.9: Results of $I_{\vec{z}\{random\}}$ and $I_{\vec{z}\{greedy\}}$ on the N-queens problems

Nodes	Colors	$I_{\vec{z}\{random\}}$ CPU Time	$I_{\vec{z}\{greedy\}}$ CPU Time
125	17	4.7 min (3.7 min)	6.2 min (4.4 min)
125	18	4.5 sec (2.9 sec)	2.9 sec (2.3 sec)
250	15	0.418 sec (0.408 sec)	0.307 sec (0.300 sec)
250	29	14.6 min (15.7 min)	14.2 min (13.3 min)

Table 6.10: Results of $I_{\vec{z}\{random\}}$ and $I_{\vec{z}\{greedy\}}$ on the hard graph-coloring problems

Problem	$I_{\vec{z}\{random\}}$ CPU Time (sec)	$I_{\vec{z}\{greedy\}}$ CPU Time (sec)
rcsp-100-10-25-15	2.742 (2.483)	3.050 (2.375)
rcsp-110-10-23-15	11.253 (10.992)	11.967 (10.558)
rcsp-120-10-21-15	7.983 (5.433)	4.850 (3.883)
rcsp-130-10-19-15	9.077 (8.308)	5.185 (3.975)
rcsp-140-10-18-15	11.000 (10.058)	9.335 (10.333)
rcsp-150-10-16.5-15	7.935 (8.692)	7.160 (6.508)

Table 6.11: Results of $I_{\vec{z}\{random\}}$ and $I_{\vec{z}\{greedy\}}$ on the tight random CSP's

problems. Since $I_{\vec{z}\{greedy\}}$ generates an assignment which is closer to a solution of a CSP, less effort is required for solution searching. Thus, the initial assignment scheme $I_{\vec{z}\{greedy\}}$ seems to have advantage over $I_{\vec{z}\{random\}}$.

Condition for Updating Lagrange Multipliers

As stated before, the condition for updating the Lagrange multipliers $\vec{\lambda}$ can affect the efficiency of the discrete Lagrangian search [62, 54, 53]. In this experiment, two common strategies for updating the Lagrange multipliers are tested. These two strategies are

- $U_{\vec{\lambda}\{every\}}$: update the Lagrange multipliers $\vec{\lambda}$ in every iteration, (i.e. after each update of the zero-one integer vector $\vec{z}$), and
- $U_{\vec{\lambda}\{stable\}}$: update the Lagrange multipliers $\vec{\lambda}$ when $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$.

Table 6.12 shows the results of the N-queens problems. Due to the fact that only a small amount of CPU time is used for solution searching, the effect of the different updating strategies is not significant. However, the results of the hard graph-coloring problems in Table 6.13 show that the updating strategy $U_{\vec{\lambda}\{every\}}$ gives us certain improvement. Similarly, the results in Table 6.14 show that $U_{\vec{\lambda}\{every\}}$ is more efficient than $U_{\vec{\lambda}\{stable\}}$ in most of the tight random CSP's. Recall that the Lagrange multipliers $\vec{\lambda}$ are the penalty values of the violated tuples. Therefore, updating the Lagrange multipliers will eventually guide the search to a solution. If we update the Lagrange multipliers earlier, the algorithm will search for other promising regions earlier.

In summary, the experiment suggests that the updating frequency of the Lagrange multipliers can affect the efficiency of the discrete Lagrange multiplier method. The updating strategy $U_{\vec{\lambda}\{every\}}$ is in general better than $U_{\vec{\lambda}\{stable\}}$ according to our experiments.

N	$U_{\vec{\lambda}\{stable\}}$ CPU Time (sec)	$U_{\vec{\lambda}\{every\}}$ CPU Time (sec)
10	0.005 (0.000)	0.003 (0.000)
20	0.003 (0.000)	0.007 (0.000)
30	0.020 (0.017)	0.020 (0.017)
40	0.028 (0.033)	0.023 (0.017)
50	0.040 (0.033)	0.045 (0.050)
60	0.068 (0.067)	0.065 (0.067)
70	0.090 (0.092)	0.087 (0.083)
80	0.178 (0.175)	0.180 (0.175)
90	0.642 (0.633)	0.635 (0.617)
100	1.078 (1.108)	1.185 (1.183)
110	1.588 (1.583)	1.630 (1.608)
120	2.033 (2.058)	2.175 (2.183)

Table 6.12: Results of $U_{\vec{\lambda}\{stable\}}$ and $U_{\vec{\lambda}\{every\}}$ on the N-queens problems

Nodes	Colors	$U_{\vec{\lambda}\{stable\}}$ CPU Time	$U_{\vec{\lambda}\{every\}}$ CPU Time
125	17	4.7 min (3.7 min)	3.3 min (2.7 min)
125	18	4.5 sec (2.9 sec)	4.4 sec (4.183 sec)
250	15	0.418 sec (0.408 sec)	0.522 sec (0.500 sec)
250	29	14.6 min (15.7 min)	12.3 min (12.1 min)

Table 6.13: Results of $U_{\vec{\lambda}\{stable\}}$ and $U_{\vec{\lambda}\{every\}}$ on the hard graph-coloring problems

Problem	$U_{\vec{\lambda}\{stable\}}$ CPU Time (sec)	$U_{\vec{\lambda}\{every\}}$ CPU Time (sec)
rcsp-100-10-25-15	2.742 (2.483)	3.238 (1.833)
rcsp-110-10-23-15	11.253 (10.992)	9.225 (6.467)
rcsp-120-10-21-15	7.983 (5.433)	5.955 (6.667)
rcsp-130-10-19-15	9.077 (8.308)	7.768 (6.500)
rcsp-140-10-18-15	11.000 (10.058)	11.182 (8.000)
rcsp-150-10-16.5-15	7.935 (8.692)	6.220 (5.600)

Table 6.14: Results of $U_{\vec{\lambda}\{stable\}}$ and $U_{\vec{\lambda}\{every\}}$ on the tight random CSP's

6.4.3 Evaluation of $\mathcal{LSDL}$(MAX)

In the previous experiments, different parameters of $\mathcal{LSDL}$ are evaluated. Some parameters are found to be better than the others in most problems. Combining these identified parameters, we construct our best $\mathcal{LSDL}$ variant, called $\mathcal{LSDL}$(MAX), which has:

- N: the objective function defined in (4.18),
- $\triangle_{\vec{z}}$: the discrete gradient operator defined in (6.8),
- $I_{\vec{z}}$: greedily initialize the zero-one integer vector $\vec{z}$ according to the procedure described in [40],
- $I_{\vec{\lambda}}$: initialize the Lagrange multipliers $\vec{\lambda}$ to $\vec{1}$, and
- $U_{\vec{\lambda}}$: update the Lagrange multipliers $\vec{\lambda}$ in every iteration.

We perform the experiments on the N-queens problems, a set of hard graph-coloring problem from DIMACS [27] and a set of randomly generated CSP's. The performance of $\mathcal{LSDL}$(MAX) is compared with that of $\mathcal{LSDL}$(GENET).

Table 6.15 shows the benchmarking results of the N-queens problems for $\mathcal{LSDL}$(MAX) and $\mathcal{LSDL}$(GENET). Since only a small amount of CPU time is spent on actual searching, the performance of $\mathcal{LSDL}$(MAX) is only slightly better than that of $\mathcal{LSDL}$(GENET).

The experimental results of the hard graph-coloring problems for $\mathcal{LSDL}$(MAX), $\mathcal{LSDL}$(GENET), and also DLM described in [62, 54, 53] are presented in Table 6.16. The results of DLM represent the average CPU time of 10 runs taken on a SUN SPARCstation 10 model 51. The efficiency of $\mathcal{LSDL}$(MAX) over that of $\mathcal{LSDL}$(GENET) is well demonstrated in this set of experiments. When comparing $\mathcal{LSDL}$(MAX) with DLM, $\mathcal{LSDL}$(MAX) is found to be more efficient than DLM. Besides the timing results, $\mathcal{LSDL}$(MAX) is better than DLM in the following aspects. First, given a predefined maximum number of iterations, say five million, $\mathcal{LSDL}$(MAX) produces solutions successfully on every run. On the other hand,

N	$\mathcal{LSDL}$(GENET) CPU Time (sec)	$\mathcal{LSDL}$(MAX) CPU Time (sec)
10	0.005 (0.000)	0.003 (0.000)
20	0.003 (0.000)	0.007 (0.000)
30	0.020 (0.017)	0.010 (0.017)
40	0.028 (0.033)	0.018 (0.017)
50	0.040 (0.033)	0.030 (0.033)
60	0.068 (0.067)	0.047 (0.050)
70	0.090 (0.092)	0.067 (0.058)
80	0.178 (0.175)	0.132 (0.125)
90	0.642 (0.633)	0.493 (0.475)
100	1.078 (1.108)	0.858 (0.833)
110	1.588 (1.583)	1.062 (1.033)
120	2.033 (2.058)	1.532 (1.492)

Table 6.15: Results of $\mathcal{LSDL}$(MAX) on the N-queens problems

Nodes	Colors	DLM Average CPU Time	$\mathcal{LSDL}$(GENET) Average (Median) CPU Time	$\mathcal{LSDL}$(MAX) Average (Median) CPU Time
125	17	23.2 min	4.7 min (3.7 min)	3.2 min (2.6 min)
125	18	3.2 sec	4.5 sec (2.9 sec)	1.1 sec (0.925 sec)
250	15	2.8 sec	0.418 sec (0.408 sec)	0.328 sec (0.325 sec)
250	29	20.3 min	14.6 min (15.7 min)	11.3 min (12.6 min)

Table 6.16: Results of $\mathcal{LSDL}$(MAX) on the hard graph-coloring problems

Problem	$\mathcal{LSDL}$(GENET) CPU Time (sec)	$\mathcal{LSDL}$(MAX) CPU Time (sec)
rcsp-100-10-25-15	2.742 (2.483)	2.577 (1.658)
rcsp-110-10-23-15	11.253 (10.992)	7.038 (6.192)
rcsp-120-10-21-15	7.983 (5.433)	4.248 (2.825)
rcsp-130-10-19-15	9.077 (8.308)	2.452 (1.883)
rcsp-140-10-18-15	11.000 (10.058)	5.475 (4.200)
rcsp-150-10-16.5-15	7.935 (8.692)	2.923 (1.400)

Table 6.17: Timing results of $\mathcal{LSDL}$(MAX) on the tight random CSP's

DLM gives only a 9/10 success ratio on the problem with 250 nodes and 29 colors. Second, the SAT versions of the graph-coloring problems in the DIMACS archive lack the set of constraints defined by (4.16) [48]. Therefore, answers to these easier problems can have a vertex assigned with more than one color. Third, DLM employs, on top of the discrete Lagrange multiplier method, a number of tuning heuristics and an additional tabu list to remember states that are visited [62, 54, 53]. For example, the Lagrange multipliers are reset by a factor of 2/3 after every 10000 iterations, and the Lagrange multipliers are updated by a different constants for different graph-coloring problems. However, our results are obtained by $\mathcal{LSDL}$(MAX) with no special tuning and additional machineries.

Table 6.17 shows the results of a set of tight random CSP's, ranging from 100 to 150 variables. Each random CSP has a uniform domain of size 10, constraint tightness 0.15, and constraint density varying from 0.165 to 0.25. Besides the CPU time, we also show the average number of iterations and Lagrange multiplier updates (in square bracket) in Table 6.18. The $\mathcal{LSDL}$(MAX) implementation performs about 6 – 73% better than the $\mathcal{LSDL}$(GENET) implementation in all problem instances. Furthermore, $\mathcal{LSDL}$(MAX) uses many fewer iterations to obtain a solution.

Problem	$\mathcal{LSDL}$(GENET) Iter. [λ Updates]	$\mathcal{LSDL}$(MAX) Iter. [λ Updates]
rcsp-100-10-25-15	7245.5 [2747.3]	4978.4 [4978.4]
rcsp-110-10-23-15	33634.4 [13387.4]	12150.0 [12150.0]
rcsp-120-10-21-15	23589.6 [9760.4]	6665.5 [6665.5]
rcsp-130-10-19-15	22570.8 [8681.4]	3290.3 [3290.3]
rcsp-140-10-18-15	27130.4 [10622.4]	7072.7 [7072.7]
rcsp-150-10-16.5-15	17389.3 [6508.7]	3968.7 [3968.7]

Table 6.18: Number of iterations and Lagrange multiplier updates of $\mathcal{LSDL}$(MAX) on the tight random CSP's

6.5 Extension of $\mathcal{LSDL}$

In the previous discussion, we establish a surprising connection between $\mathcal{LSDL}$ and the GENET model. This connection also suggests a dual viewpoint of GENET, as a heuristic repair method and as a discrete Lagrange multiplier method. Hence, we can improve GENET by exploring the space of parameters available in the $\mathcal{LSDL}$ framework. Alternatively, techniques developed from GENET can be used to extend our $\mathcal{LSDL}$ framework. Lazy arc consistency [56, 59, 57], a consistency method that speeds up the search of GENET, is incorporated in $\mathcal{LSDL}$. Experiments show that lazy arc consistency gives significant improvement for discrete Lagrangian search.

6.5.1 Arc Consistency

Arc consistency [37] is a well known technique for reducing the search space of a CSP. A CSP (U, D, C) is *arc consistent* if and only if for all variables $x, y \in U$ and for all value $u \in D_x$ there exists a value $v \in D_y$ such that the constraint c on variables x and y is satisfied. In the terminology of GENET, a CSP, or a GENET network $\mathcal{N}$, is arc consistent if and only if for all clusters $i, j \in U$ and for all label nodes $\langle i, k \rangle \in \mathcal{N}$ there exists a label node $\langle j, l \rangle \in \mathcal{N}$ such that there is no connection between $\langle i, k \rangle$ and $\langle j, l \rangle$ [56, 59, 57]. Obviously, values which are arc inconsistent cannot appear in any solution of CSP. Hence, we are guaranteed that

any solution of the original CSP is a solution of the corresponding arc consistent CSP. We say that the original CSP and its associated arc consistent CSP are *equivalent*.

Arc consistency gives us a way to remove useless values from the domains of variables. Algorithms, such as AC-3 [37], are usually combined with backtracking tree search to increase the efficiency. Similar algorithms can be used to preprocess a given GENET network $\mathcal{N}$ to produce an equivalent arc consistent network. The algorithms remove a label node $\langle i, j \rangle$ and its associated connections from the GENET network $\mathcal{N}$ if $\langle i, j \rangle$ is arc inconsistent. Once a label node is found to be arc inconsistent and removed from the GENET network, we need to re-check all other label nodes which may no longer be arc consistent.

6.5.2 Lazy Arc Consistency

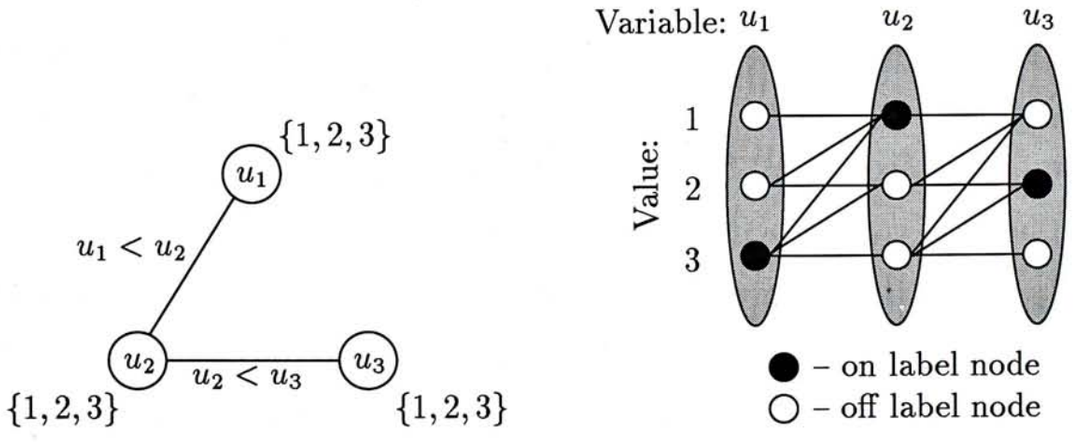
Preprocessing a GENET network with arc consistency algorithm can improve the search because of the reduction in the search space. However, since arc consistency is in general a fairly expensive operation, it is beneficial only if the improvement in efficiency is greater than the overhead of the arc consistency preprocessing phase. Stuckey and Tam [56, 59, 57] develop lazy arc consistency for the GENET model.

Let $o(\mathcal{S}, i)$ be the on label node of cluster i in state $\mathcal{S}$ of a GENET network $\mathcal{N}$. A GENET network $\mathcal{N}$ in a state $\mathcal{S}$ is *lazy arc consistent* if and only if for all clusters $i, j \in U$ there exists a label node $\langle j, k \rangle \in \mathcal{N}$ such that there is no connection between $o(\mathcal{S}, i)$ and $\langle j, k \rangle$ [56, 59, 57]. Since lazy arc consistency only enforces arc consistency for the current on label nodes, it can readily be incorporated in the convergence procedure of GENET.

Algorithm 6.2 gives a modified input calculation procedure for cluster i of the GENET network $\mathcal{N}$ in a state $\mathcal{S}$ [56, 59, 57]. The algorithm detects lazy arc inconsistency during the calculation of inputs of each cluster. For example, consider an arc inconsistent CSP and its corresponding GENET network shown in Figure 6.1. When calculating the inputs of cluster u_1, we found that each label node $\langle u_1, 1 \rangle$, $\langle u_1, 2 \rangle$ and $\langle u_1, 3 \rangle$ are connected to label node $\langle u_2, 1 \rangle$, the current on

procedure input($\mathcal{N}, \mathcal{S}, i$)
begin
  if inconsistent(i) **then**
    $\mathcal{N} \leftarrow \mathcal{N} - \{o(\mathcal{S}, i)\} - \{(o(\mathcal{S}, i), \langle u, v \rangle) \mid (o(\mathcal{S}, i), \langle u, v \rangle) \in \mathcal{N}\}$
  end if
  for each cluster $j \neq i$ **do**
    possibly_inconsistent(j) $\leftarrow$ `true`
  end for
  for each label node $\langle i, k \rangle \in \mathcal{N}$ **do**
    $I_{\langle i,k \rangle} \leftarrow 0$
    for each cluster $j \neq i$ **do**
      if $(\langle i, k \rangle, o(\mathcal{S}, j)) \in \mathcal{N}$ **then**
        $I_{\langle i,k \rangle} \leftarrow I_{\langle i,k \rangle} + W_{\langle i,k \rangle o(\mathcal{S},j)}$
      else
        possibly_inconsistent(j) $\leftarrow$ `false`
      end if
    end for
  end for
  for each cluster $j \neq i$ **do**
    inconsistent(j) $\leftarrow$ inconsistent(j) $\vee$ possibly_inconsistent(j)
  end for
end

Algorithm 6.2: A modified input calculation procedure, that can detect lazy arc consistency, for GENET



(a) The CSP

(b) The GENET network

Figure 6.1: An arc inconsistent CSP and its corresponding GENET network

procedure Lazy-$\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$
begin
  $(I_{\vec{z}})$ initialize the value of $\vec{z}$
  $(I_{\vec{\lambda}})$ initialize the value of $\vec{\lambda}$
  while (N) $L(\vec{z}, \vec{\lambda}) - N(\vec{z}) > 0$ ($\vec{z}$ is not a solution) **do**
    for each variable $i \in U$ **do**
      if $\forall j \neq i \in U \nexists k \in D_j$ such that $(\langle i, a(\vec{z}, i)\rangle, \langle j, k\rangle) \notin \mathcal{I}$ **then**
        $D_i \leftarrow D_i - \{a(\vec{z}, i)\}$
      end if
    end for
    $(\triangle_{\vec{z}})$ update $\vec{z}$: $\vec{z} \leftarrow \vec{z} - \triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})$
    if $(U_{\vec{\lambda}})$ condition for updating $\vec{\lambda}$ holds **then**
      update $\vec{\lambda}$: $\vec{\lambda} \leftarrow \vec{\lambda} + \vec{g}(\vec{z})$
    end if
  end while
end

Algorithm 6.3: The Lazy-$\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$ procedure

label node of cluster u_2. Hence, $\langle u_2, 1\rangle$ and its associated connections should be removed from the GENET network.

Since lazy arc consistency is targeted at values that are selected during the search, which may be much fewer than the entire search space, its overhead is smaller than that of arc consistency. Experiments show that lazy arc consistency improves GENET substantially for CSP's which are arc inconsistent and does not degrade the performance significantly for problems which are already arc consistent [56, 59, 57].

Lazy arc consistency can be incorporated in $\mathcal{LSDL}$ in a similar manner. Let $a(\vec{z}, i)$ be the current assignment of variable i such that $z_{\langle i, a(\vec{z}, i)\rangle} = 1$ and $z_{\langle i, j\rangle} = 0$ for all $j \neq a(\vec{z}, i) \in D_i$, and $\mathcal{I}$ be the set of all incompatible tuples $(\langle i, j\rangle, \langle k, l\rangle)$. The modified discrete Lagrangian search algorithm *Lazy-*$\mathcal{LSDL}$ is shown in Algorithm 6.3. Similar to GENET, the procedure for detecting lazy arc inconsistency can be integrated in the discrete gradient operator $\triangle_{\vec{z}}$. For example, lazy arc inconsistency can be detected during the calculation of the set X in the evaluation of the partial discrete gradient operator ∂^i. We state explicitly the detection

procedure in Lazy-$\mathcal{LSDL}$ to show that lazy arc consistency is independent of the discrete gradient operator used. In other words, any discrete gradient operator defined for $\mathcal{LSDL}$ can be used in Lazy-$\mathcal{LSDL}$ without any special modification.

6.5.3 Experiments

In order to demonstrate the efficiency of Lazy-$\mathcal{LSDL}$, we implement both Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX), which are instances of Lazy-$\mathcal{LSDL}$ with the same parameters as those of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(MAX) respectively. In both Lazy-$\mathcal{LSDL}$ implementations, the procedure for detecting lazy arc inconsistency is integrated in the discrete gradient operator to improve the efficiency. The performance of Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX) on the N-queens problems, a set of randomly generated permutation generation problems [31], a set of artificial problems [56, 59] and a set of random CSP's is compared against the non-lazy versions. Timing results of all $\mathcal{LSDL}$ and Lazy-$\mathcal{LSDL}$ implementations are taken on a SUN SPARCstation 10 model 40. The unbracketed and the bracketed results are the average and the median CPU time in second of 10 runs respectively.

The N-queens Problems

Tables 6.19 and 6.20 show the experimental results of Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX) on the 10- to 120-queens problems respectively. Since the N-queens problems are arc consistent, nothing is gained from the detection of lazy arc inconsistency. However, the overhead of the additional calculation is almost negligible.

The Permutation Generation Problems

The permutation generation problem [31] is a combinatorial problem. Its aim is to construct a permutation on integers from 1 to n satisfying the conditions of monotonies and advances. A detailed description of modeling the problem

N	$\mathcal{LSDL}$(GENET) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(GENET) CPU Time (sec)
10	0.005 (0.000)	0.002 (0.000)
20	0.003 (0.000)	0.005 (0.000)
30	0.020 (0.017)	0.023 (0.017)
40	0.028 (0.033)	0.028 (0.033)
50	0.040 (0.033)	0.048 (0.050)
60	0.068 (0.067)	0.078 (0.067)
70	0.090 (0.092)	0.098 (0.100)
80	0.178 (0.175)	0.273 (0.267)
90	0.642 (0.633)	0.787 (0.758)
100	1.078 (1.108)	1.285 (1.267)
110	1.588 (1.583)	1.748 (1.742)
120	2.033 (2.058)	2.330 (2.367)

Table 6.19: Results of Lazy-$\mathcal{LSDL}$(GENET) on the N-queens problems

N	$\mathcal{LSDL}$(MAX) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(MAX) CPU Time (sec)
10	0.003 (0.000)	0.000 (0.000)
20	0.007 (0.000)	0.003 (0.000)
30	0.010 (0.017)	0.010 (0.017)
40	0.018 (0.017)	0.018 (0.017)
50	0.030 (0.033)	0.032 (0.033)
60	0.047 (0.050)	0.047 (0.050)
70	0.067 (0.058)	0.068 (0.067)
80	0.132 (0.125)	0.242 (0.233)
90	0.493 (0.475)	0.557 (0.525)
100	0.858 (0.833)	0.920 (0.892)
110	1.062 (1.033)	1.193 (1.175)
120	1.532 (1.492)	1.563 (1.533)

Table 6.20: Results of Lazy-$\mathcal{LSDL}$(MAX) on the N-queens problems

n	$\mathcal{LSDL}$(GENET) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(GENET) CPU Time (sec)	Pruned
10	0.002 (0.000)	0.002 (0.000)	1.2
20	0.007 (0.000)	0.008 (0.008)	1.0
30	0.153 (0.083)	0.055 (0.017)	2.6
40	0.047 (0.042)	0.050 (0.033)	0.9
50	0.052 (0.050)	0.048 (0.050)	0.7
60	0.098 (0.092)	0.112 (0.108)	0.7
70	0.138 (0.117)	0.168 (0.150)	0.9
80	0.398 (0.383)	0.392 (0.367)	0.6
90	0.813 (0.800)	0.873 (0.850)	0.5
100	1.192 (1.217)	1.162 (1.192)	0.6

Table 6.21: Results of Lazy-$\mathcal{LSDL}$(GENET) on the random permutation generation problems

as a CSP can be found in [22]. In the experiment, we randomly generate the monotonies and advances for n varying from 10 to 100. These problems involve arc inconsistency. Tables 6.21 and 6.22 show the results of Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX) respectively. Both the CPU times and the average number of values pruned by the lazy arc consistency versions are presented. Since the problems are relatively easy for $\mathcal{LSDL}$, all implementations can solve the problems with little search. Therefore, few values are pruned before a solution is found. However, the experiment confirms that the overhead of lazy arc consistency is small, even when there is little advantage.

The Artificial Problems

The set of artificial problems [56, 59] is used to illustrate the advantages of Lazy-$\mathcal{LSDL}$. An artificial problem of size n is a CSP with $n+1$ variables $u_1, u_2, \ldots, u_{n+1}$ and n constraints $u_1 < u_2, u_2 < u_3, \ldots, u_n < u_{n+1}$. The domain size of each variable is either $n+1$ or $2n$. Note that the artificial problems with domain size $n+1$ are special instances of the permutation generation problem [31]. Benchmarking results of Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX) on the artificial problems are shown in Tables 6.23 and 6.24 respectively. Besides the CPU time, we also

n	$\mathcal{LSDL}$(MAX) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(MAX) CPU Time (sec)	Pruned
10	0.005 (0.000)	0.002 (0.000)	3.9
20	0.010 (0.008)	0.008 (0.008)	1.9
30	0.015 (0.017)	0.057 (0.017)	3.1
40	0.020 (0.017)	0.028 (0.025)	2.1
50	0.053 (0.050)	0.040 (0.042)	2.0
60	0.075 (0.067)	0.100 (0.075)	2.8
70	0.180 (0.167)	0.182 (0.183)	1.3
80	0.408 (0.392)	0.400 (0.350)	1.7
90	0.782 (0.733)	0.770 (0.792)	0.4
100	1.043 (1.008)	1.132 (1.092)	3.1

Table 6.22: Results of Lazy-$\mathcal{LSDL}$(MAX) on the random permutation generation problems

Prob. Size n	$\mathcal{LSDL}$(GENET) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(GENET) CPU Time (sec)	Pruned
Domain Size $= n + 1$			
10	0.033 (0.033)	0.008 (0.008)	31.8
20	1.065 (1.075)	0.260 (0.258)	261.8
30	8.318 (7.800)	1.492 (1.517)	638.5
40	36.267 (35.583)	5.552 (5.617)	1234.7
50	107.372 (105.542)	15.678 (15.675)	2007.3
Domain Size $= 2n$			
10	0.030 (0.017)	0.052 (0.050)	44.0
20	2.008 (1.792)	0.787 (0.825)	144.1
30	15.837 (13.225)	8.438 (9.008)	480.6
40	80.680 (82.925)	34.383 (35.000)	873.5
50	306.522 (314.100)	109.290 (111.625)	1482.5

Table 6.23: Results of Lazy-$\mathcal{LSDL}$(GENET) on the artificial problems

Prob. Size n	$\mathcal{LSDL}$(MAX) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(MAX) CPU Time (sec)	Pruned
Domain Size $= n + 1$			
10	0.017 (0.017)	0.005 (0.000)	24.2
20	0.865 (0.867)	0.123 (0.142)	227.4
30	6.965 (6.750)	0.713 (0.675)	520.9
40	26.775 (25.892)	1.860 (1.842)	779.7
50	85.592 (86.058)	4.507 (4.525)	1275.7
Domain Size $= 2n$			
10	0.045 (0.050)	0.000 (0.000)	0.0
20	0.330 (0.000)	0.077 (0.017)	51.8
30	6.430 (7.017)	0.407 (0.200)	142.7
40	41.175 (49.725)	3.530 (1.167)	367.0
50	97.852 (110.825)	11.507 (3.258)	677.9

Table 6.24: Results of Lazy-$\mathcal{LSDL}$(MAX) on the artificial problems

give the average number of values pruned by Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX). Both Lazy-$\mathcal{LSDL}$ implementations give significant improvement for the discrete Lagrangian search. Since Lazy-$\mathcal{LSDL}$(MAX) search a smaller space than that of Lazy-$\mathcal{LSDL}$(GENET), it prunes fewer values from domains. Similarly, as the domain size grows, the number of lazy arc inconsistencies found by both Lazy-$\mathcal{LSDL}$ is reduced since some values are never searched. These properties clearly illustrate the targeted nature of lazy arc consistency.

Random CSP's

A set of randomly generated CSP's is used to test our Lazy-$\mathcal{LSDL}$ implementations. The random CSP's, with variable size ranging from 120 to 170, have domain size 10, constraint density 0.6, and constraint tightness 0.75. A high constraint density and a high constraint tightness are chosen to ensure that each generated CSP is arc inconsistent. In order to guarantee that each random CSP is soluble, the following procedure is used. After generating a CSP with variable size n and a set of chosen parameters, we randomly generate a tuple, say

Problem	$\mathcal{LSDL}$(GENET) Iter.	$\mathcal{LSDL}$(GENET) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(GENET) Iter.	Lazy-$\mathcal{LSDL}$(GENET) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(GENET) Pruned
rcsp-120-10-60-75	152.4	7.638 (7.683)	11.0	2.602 (2.600)	1011.4
rcsp-130-10-60-75	162.0	9.135 (9.142)	11.0	3.055 (3.058)	1099.8
rcsp-140-10-60-75	147.3	9.690 (9.708)	11.0	3.607 (3.608)	1183.0
rcsp-150-10-60-75	173.4	12.645 (12.717)	8.6	3.177 (4.083)	918.2
rcsp-160-10-60-75	167.4	14.208 (13.925)	11.0	4.930 (4.908)	1355.3
rcsp-170-10-60-75	176.2	21.820 (22.183)	11.0	7.418 (7.325)	1433.5

Table 6.25: Results of Lazy-$\mathcal{LSDL}$(GENET) on the tight random CSP's with arc inconsistency

Problem	$\mathcal{LSDL}$(MAX) Iter.	$\mathcal{LSDL}$(MAX) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(MAX) Iter.	Lazy-$\mathcal{LSDL}$(MAX) CPU Time (sec)	Lazy-$\mathcal{LSDL}$(MAX) Pruned
rcsp-120-10-60-75	26.6	5.728 (7.208)	11.0	2.652 (2.650)	1018.1
rcsp-130-10-60-75	27.4	6.983 (7.250)	8.8	2.530 (3.117)	884.2
rcsp-140-10-60-75	27.1	8.198 (9.617)	8.0	2.727 (3.792)	831.7
rcsp-150-10-60-75	28.5	10.198 (11.442)	6.6	2.667 (4.250)	768.7
rcsp-160-10-60-75	23.0	9.565 (12.650)	8.8	4.100 (5.042)	1101.1
rcsp-170-10-60-75	31.9	20.058 (20.200)	11.0	7.393 (7.358)	1456.9

Table 6.26: Results of Lazy-$\mathcal{LSDL}$(MAX) on the tight random CSP's with arc inconsistency

$(\langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots, \langle u_n, v_n\rangle)$, and regard it as a solution of the CSP. If a constraint on variables u_i and u_j contains $(\langle u_i, v_i\rangle, \langle u_j, v_j\rangle)$, the one in the generated solution, as an incompatible tuple, the incompatible tuple is removed from the constraint. Two new incompatible tuples $(\langle u_i, v_i\rangle, \langle u_j, w_j\rangle)$ and $(\langle u_i, w_i\rangle, \langle u_j, v_j\rangle)$, where $w_i \neq v_i$ and $w_j \neq v_j$ are values chosen randomly from D_{u_i} and D_{u_j} respectively, are added.

Tables 6.25 and 6.26 show the timing results and the average number of iterations of various $\mathcal{LSDL}$ implementations. The average number of values removed is also recorded for Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX). The lazy versions are much more efficient than the non-lazy versions. In particular, the number of iterations is substantially reduced by the Lazy-$\mathcal{LSDL}$.

Problem	Lazy-$\mathcal{LSDL}$(GENET) Iter.	CPU Time (sec)	Pruned
rcsp-100-10-70-90	10.0	2.347 (2.342)	934.6
rcsp-110-10-70-90	10.0	2.835 (2.842)	1025.5
rcsp-120-10-70-90	10.0	3.387 (3.375)	1116.4

Table 6.27: Results of Lazy-$\mathcal{LSDL}$(GENET) on the insoluble random CSP's

Problem	Lazy-$\mathcal{LSDL}$(MAX) Iter.	CPU Time (sec)	Pruned
rcsp-100-10-70-90	10.0	2.345 (2.342)	907.8
rcsp-110-10-70-90	10.0	2.855 (2.850)	1000.6
rcsp-120-10-70-90	10.0	3.428 (3.433)	1093.0

Table 6.28: Results of Lazy-$\mathcal{LSDL}$(MAX) on the insoluble random CSP's

Table 6.27 and 6.28 show the results of each Lazy-$\mathcal{LSDL}$ implementations on some insoluble random CSP's. The insoluble CSP's, ranging from 100 to 120 variables, have a uniform domain of size 10, constraint density 0.7, constraint tightness 0.9. Since these problems have no solution, $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(MAX) execute forever. On the other hand, Lazy-$\mathcal{LSDL}$ can terminate and report insolubility when a variable domain becomes empty.

Chapter 7

Extending $\mathcal{LSDL}$ for General CSP's: Initial Results

In this chapter, we extend $\mathcal{LSDL}$ for solving general CSP's. A general CSP is transformed into an integer constrained minimization problem. New incompatibility functions are defined for different constraints. By constructing a new discrete gradient operator to accommodate the change of formulation, the discrete Lagrangian search scheme $\mathcal{LSDL}$ can be applied directly. We implement $\mathcal{LSDL}$(GENERAL), an instance of $\mathcal{LSDL}$ for general CSP's, to verify our approach. Experiments show that the performance of $\mathcal{LSDL}$(GENERAL) is comparable to that of E-GENET [32, 33, 69], an extended GENET for efficient general CSP's solving, in some problems. However, $\mathcal{LSDL}$(GENERAL) performs much worse than $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(MAX). The inadequacy of our general formulation and a possible solution are explored at the end of the chapter.

7.1 General CSP's as Integer Constrained Minimization Problems

In this section, the transformation of a general CSP into an integer constrained minimization problem is presented. The definition of the incompatibility functions

for various constraints are also given.

7.1.1 Formulation

Given a CSP (U, D, C). We assume that each domain D_i for $i \in U$ is a set of integers. The corresponding integer constrained minimization problem is formulated as follows. Each variable $i \in U$ is represented by an integer variable z_i, which can take values from a domain D_i. The integer variable z_i is equal to $j \in D_i$ if and only if value j is assigned to variable i. In other words, $\vec{z} = (\ldots, z_i, \ldots)$ corresponds to a variable assignment for (U, D, C). Each constraint $c \in C$ is denoted by an incompatibility function $g_c(\vec{z})$, which returns 0 when the constraint c is satisfied; otherwise, it returns a positive integer to represent the amount of violation of the current assignment. In general, the incompatibility function $g_c(\vec{z})$ for different constraints could be different.

Similar to the zero-one integer constrained minimization problem of a binary CSP, the resultant integer constrained minimization problem of a general CSP (U, D, C) is

$$\min \; N(\vec{z}) \tag{7.1}$$

subject to

$$z_i \in D_i, \quad \forall\, i \in U \tag{7.2}$$

$$g_c(\vec{z}) = 0, \quad \forall\, c \in C \tag{7.3}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $N(\vec{z})$ is an objective function satisfying the *correspondence requirement*, which stated that *every solution of the CSP must correspond to a constrained global minimum of the associated integer constrained minimization problem (7.1 – 7.3)*. Note that the solution space of a CSP is defined entirely by the constraints (7.2 – 7.3). The objective function is used to guide the search only.

The objective functions defined for binary CSP's can be extended directly.

The objective function (4.18) becomes

$$N(\vec{z}) = \sum_{c \in C} g_c(\vec{z}). \tag{7.4}$$

Unlike the binary counterpart, the objective function (7.4) does not measure constraint violations in terms of number of violated tuples. It simply returns the total amount of constraint violation given by the incompatibility functions. The objective function, which counts the total number of constraint violations, can be constructed as,

$$N(\vec{z}) = \sum_{c \in C} f(g_c(\vec{z})) \tag{7.5}$$

where $f(x)$ is a function which returns 1 when $x \neq 0$, and 0 otherwise. On the other hand, the constant objective function

$$N(\vec{z}) = 0 \tag{7.6}$$

can be used without any modification. Note that all these objective functions have the property that any assignment which satisfies constraints (7.2 – 7.3) is a constrained global minimum. In other words, they all satisfy the correspondence requirement.

7.1.2 Incompatibility Functions

As stated before, the incompatibility functions $g_c(\vec{z})$ are designed specifically for different constraints. In this section, we present the incompatibility functions for linear arithmetic constraints, the `illegal` constraint, the `atmost` constraint and the `among` constraint [2].

Linear Arithmetic Constraints

A linear arithmetic constraint is of the form $X \circ Y$, where X and Y are linear arithmetic expressions and $\circ \in \{=, \neq, <, \leq, >, \geq\}$. The linear expressions X and Y can be written as $A_0 + A_1u_1 + \ldots + A_ku_k$, where each A_i is an integer for $i = 0, 1, \ldots, k$ and each u_j is a variable for $j = 1, 2, \ldots, k$. A linear arithmetic

constraint is satisfied if $X \circ Y$ is satisfied. Let $\theta = \{\ldots, u_i/z_{u_i}, \ldots\}$ be a substitution, where each u_i is a variable, each z_{u_i} is an integer variable corresponding to u_i and all the u_i are distinct. An expression $E\theta$ is an instance of E obtained by simultaneously replacing each occurrence of u_i by z_{u_i}. The incompatibility functions $g_\circ(\vec{z})$, for each $\circ \in \{=, \neq, <, \leq, >, \geq\}$, are defined as follows,

$$g_=(\vec{z}) = |X\theta - Y\theta| \tag{7.7}$$

$$g_\neq(\vec{z}) = \begin{cases} 1, & \text{if } X\theta = Y\theta \\ 0, & \text{otherwise} \end{cases} \tag{7.8}$$

$$g_<(\vec{z}) = \begin{cases} X\theta - Y\theta + 1, & \text{if } X\theta \geq Y\theta \\ 0, & \text{otherwise} \end{cases} \tag{7.9}$$

$$g_\leq(\vec{z}) = \begin{cases} X\theta - Y\theta, & \text{if } X\theta > Y\theta \\ 0, & \text{otherwise} \end{cases} \tag{7.10}$$

$$g_>(\vec{z}) = \begin{cases} Y\theta - X\theta + 1, & \text{if } X\theta \leq Y\theta \\ 0, & \text{otherwise} \end{cases} \tag{7.11}$$

$$g_\geq(\vec{z}) = \begin{cases} Y\theta - X\theta, & \text{if } X\theta < Y\theta \\ 0, & \text{otherwise} \end{cases} \tag{7.12}$$

These incompatible functions simply return the amount of constraint violations based on the difference between the linear expressions $X\theta$ and $Y\theta$.

The `illegal` Constraint

The `illegal`$((u_1, u_2, \ldots, u_k), (v_1, v_2, \ldots, v_k))$ constraint disallows the simultaneous assignment of values $v_1, v_2, \ldots, v_k$ to variables $u_1, u_2, \ldots, u_k$. Since any constraint can be expressed as a set of incompatible tuples, the `illegal` constraint can be regarded as a fundamental constraint. The incompatibility function $g_{\texttt{illegal}}(\vec{z})$ is

$$g_{\texttt{illegal}}(\vec{z}) = \begin{cases} 1, & \text{if } (z_{u_1}, z_{u_2}, \ldots, z_{u_k}) = (v_1, v_2, \ldots, v_k) \\ 0, & \text{otherwise} \end{cases} \tag{7.13}$$

where, by definition, $(z_{u_1}, z_{u_2}, \ldots, z_{u_k}) = (u_1, u_2, \ldots, u_k)$.

The atmost Constraint

The atmost constraint is of the form $\texttt{atmost}(N, Var, Val)$, where N is a natural number, Var is a set of variables and Val is a set of values. It specifies that no more than N variables in Var can take values from Val. Let $n(S_{var}, S_{val})$ be the function which returns the number of variables in the set S_{var} currently assigned with values in the set S_{val}. Obviously, if $n(Var, Val) > N$, a smaller difference $n(Var, Val) - N$ would be preferred. Hence, the corresponding incompatibility function $g_{\texttt{atmost}}(\vec{z})$ is defined as

$$g_{\texttt{atmost}}(\vec{z}) = \begin{cases} n(Var\theta, Val) - N, & \text{if } n(Var\theta, Val) > N \\ 0, & \text{otherwise} \end{cases} \tag{7.14}$$

where $\theta = \{\ldots, u_i/z_{u_i}, \ldots\}$ is a substitution and $Var\theta$ is the set of integer variables obtained by simultaneously replacing each u_i in Var with z_{u_i}.

The $\texttt{atleast}(N, Var, Val)$ constraint, which specifies that no fewer than N variables taken from the variable set Var are having values in the value set Val, can be handled similarly. Thus, the incompatibility function $g_{\texttt{atleast}}(\vec{z})$ is

$$g_{\texttt{atleast}}(\vec{z}) = \begin{cases} N - n(Var\theta, Val), & \text{if } n(Var\theta, Val) < N \\ 0, & \text{otherwise} \end{cases} \tag{7.15}$$

where $\theta = \{\ldots, u_i/z_{u_i}, \ldots\}$ is a substitution and $Var\theta$ is the set of integer variables obtained by simultaneously replacing each u_i in Var with z_{u_i}.

The among Constraint

The among constraint is a global constraint introduced in CHIP [2]. It can be regarded as a combination of the atmost and the atleast constraints. Among the five variants of the among constraint, we consider the first and the second variants only. The first variant has the form

$$\texttt{among}(u_0, [u_1, u_2, \ldots, u_k], [c_1, c_2, \ldots, c_k], [v_1, v_2, \ldots, v_l]),$$

where $u_0, u_1, u_2, \ldots, u_k$ are variables, $c_1, c_2, \ldots, c_k$ are integers and $v_1, v_2, \ldots, v_l$ are domain values. It specifies that exactly u_0 terms among $u_1 + c_1, u_2 + c_2, \ldots, u_k + c_k$

having values in the list $[v_1, v_2, \ldots, v_l]$. The incompatibility function $g_{\textbf{among}}(\vec{z})$ is defined as follows. Let $n(L_t, L_v)$ be the function which returns the number of terms $u_i + c_i$ in the list L_t currently having values in the list L_v. The incompatibility function is

$$g_{\textbf{among}}(\vec{z}) = |z_{u_0} - n(L_t\theta, L_v)| \tag{7.16}$$

where $L_t = [u_1 + c_1, u_2 + c_2, \ldots, u_k + c_k], L_t = [v_1, v_2, \ldots, v_l]$, z_{u_0} is the integer variable corresponding to variable u_0, $\theta = \{\ldots, u_i/z_{u_i}, \ldots\}$ is a substitution, and $L_t\theta$ is a list of terms obtained by simultaneously replacing each u_i in L_t with z_{u_i}. Note that the incompatibility function is similar to the one defined for the $=$ constraint.

The second variant

$$\textbf{among}([N_{low}, N_{up}], [u_1, u_2, \ldots, u_k], [c_1, c_2, \ldots, c_k], [v_1, v_2, \ldots, v_l]),$$

where N_{low}, N_{up} are natural numbers, $u_1, u_2, \ldots, u_k$ are variables, $c_1, c_2, \ldots, c_k$ are integers and $v_1, v_2, \ldots, v_l$ are values, specifies that at least N_{low} and at most N_{up} terms among $u_1 + c_1, u_2 + c_2, \ldots, u_k + c_k$ can have values in $[v_1, v_2, \ldots, v_l]$. Similar to the first variant, let $n(L_t, L_v)$ be the function which returns the number of terms $u_i + c_i$ in the list L_t currently having values in the list L_v. The incompatibility function for the second variant is defined as

$$g_{\textbf{among}}(\vec{z}) = \begin{cases} N_{low} - n(L_t\theta, L_v), & \text{if } n(L_t\theta, L_v) < N_{low} \\ n(L_t\theta, L_v) - N_{up}, & \text{if } n(L_t\theta, L_v) > N_{up} \\ 0, & \text{otherwise} \end{cases} \tag{7.17}$$

where $L_t = [u_1 + c_1, u_2 + c_2, \ldots, u_k + c_k], L_t = [v_1, v_2, \ldots, v_l]$, $\theta = \{\ldots, u_i/z_{u_i}, \ldots\}$ is a substitution, and $L_t\theta$ is a list of terms obtained by simultaneously replacing each u_i in L_t with z_{u_i}. Although the `atmost` and the `atleast` constraints can be simulated by the second variant of the `among` constraint, an extra comparison is required to evaluate the incompatibility function $g_{\textbf{among}}(\vec{z})$. Therefore, we construct the `atmost` and the `atleast` constraints to improve efficiency.

The other three variants can be stated as a combination of the second variants.

They are described as follows. The third variant is

$$\textbf{among}([N_{low}, N_{up}, S], [u_1, u_2, \ldots, u_k], [c_1, c_2, \ldots, c_k], [v_1, v_2, \ldots, v_l]),$$

where $S \leq k$ is a positive integer. It is equivalent to the following set of **among** constraints

$$\begin{aligned}
&\textbf{among}([N_{low}, N_{up}], [u_1, \ldots, u_S], [c_1, \ldots, c_S], [v_1, \ldots, v_l]),\\
&\textbf{among}([N_{low}, N_{up}], [u_2, \ldots, u_{S+1}], [c_2, \ldots, c_{S+1}], [v_1, \ldots, v_l]),\\
&\qquad\qquad\vdots\\
&\textbf{among}([N_{low}, N_{up}], [u_{k-S+1}, \ldots, u_k], [c_{k-S+1}, \ldots, c_k], [v_1, \ldots, v_l]).
\end{aligned}$$

The fourth variant is of the form

$$\textbf{among}([N_{low}, N_{up}, S, N_{least}, N_{most}], [u_1, u_2, \ldots, u_k], [c_1, c_2, \ldots, c_k], [v_1, v_2, \ldots, v_l]),$$

where N_{least} and N_{most} are natural numbers. This constraint is a combination of the second and the third variants, namely

$$\begin{aligned}
&\textbf{among}([N_{low}, N_{up}, S], [u_1, \ldots, u_k], [c_1, \ldots, c_k], [v_1, \ldots, v_l]),\\
&\textbf{among}([N_{least}, N_{most}], [u_1, \ldots, u_k], [c_1, \ldots, c_k], [v_1, \ldots, v_l]).
\end{aligned}$$

Hence, it can be handled as a number of the second variants. The last variant is

$$\textbf{among}([N_{low}, N_{up}, S, I_{low}, I_{up}, I_S], [u_1, u_2, \ldots, u_k], [c_1, c_2, \ldots, c_k], [v_1, v_2, \ldots, v_l]),$$

where I_{low}, I_{up} are natural numbers and I_S is a positive integer. It is an abstraction of the following set of second variants

$$\begin{aligned}
&\textbf{among}([N_{low}, N_{up}], [u_1, \ldots, u_S], [c_1, \ldots, c_S], [v_1, \ldots, v_l]),\\
&\textbf{among}([N_{low} + I_{low}, N_{up} + I_{up}], [u_1, \ldots, u_{S+I_S}], [c_1, \ldots, c_{S+I_S}], [v_1, \ldots, v_l]),\\
&\qquad\qquad\vdots\\
&\textbf{among}([N_{low} + m \times I_{low}, N_{up} + m \times I_{up}], [u_1, \ldots, u_k], [c_1, \ldots, c_k], [v_1, \ldots, v_l]),
\end{aligned}$$

where $m = (k - S)/I_S$.

7.2 The Discrete Lagrange Multiplier Method

The discrete Lagrange multiplier method for general CSP's is similar to the one for binary CSP's. The Lagrangian function $L(\vec{z}, \vec{\lambda})$ for the integer constrained minimization problem (7.1 – 7.3) is

$$L(\vec{z}, \vec{\lambda}) = N(\vec{z}) + \sum_{c \in C} \lambda_c g_c(\vec{z}) \tag{7.18}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $\vec{\lambda} = (\ldots, \lambda_c, \ldots)$ is a vector of Lagrange multipliers, one λ_c for each constraint $c \in C$. The constraints in (7.2), which enforce valid assignments for a CSP, are not incorporated in the Lagrangian function. They are included in the discrete gradient operator for the search process.

According to the discrete saddle point theorem [70], a constrained minimum of the integer constrained minimization problem can be obtained by finding a saddle point of the Lagrangian function $L(\vec{z}, \vec{\lambda})$. Since the saddle point can be located by performing descent in the $\vec{z}$-space and ascent in the $\vec{\lambda}$-space [43], we use the same difference equations [62, 54, 53] defined for the binary case

$$\vec{z}^{s+1} = \vec{z}^s - \triangle_{\vec{z}} L(\vec{z}^s, \vec{\lambda}^s) \tag{7.19}$$

$$\vec{\lambda}^{s+1} = \vec{\lambda}^s + \vec{g}(\vec{z}^s) \tag{7.20}$$

where $\vec{g} = (\ldots, g_c(\vec{z}), \ldots)$ is a vector of incompatibility functions and $\triangle_{\vec{z}}$ is a discrete gradient operator.

Similar to the binary case, the discrete gradient operator is not unique. The discrete gradient operator (6.6 – 6.7) for binary CSP's is now redefined as follows. Given a vector of integer variables $\vec{z} = (\ldots, z_i, \ldots)$, the *projection operator* π_i

$$\pi_i(\vec{z}) = z_i \tag{7.21}$$

gives the *ith-component* of $\vec{z}$. The *ith partial discrete gradient operator* ∂_i for all $i \in U$ is given by

$$\partial_i L(\vec{z}, \vec{\lambda}) = \pi_i(\vec{z}) - \pi_i(\vec{z}') \tag{7.22}$$

when the following conditions hold:

- X is a set of integer variables vector such that $\forall \vec{x} \in X$, we have

$$x_i \in D_i \ \wedge \ \forall j \neq i \in U \ x_j = z_j$$
$$\wedge \ \forall \vec{z}'' \left[(z_i'' \in D_i \wedge \forall j \neq i \in U \ z_j'' = z_j) \Rightarrow L(\vec{x}, \vec{\lambda}) \leq L(\vec{z}'', \vec{\lambda}) \right]$$

- $\vec{z}'$ is selected from X by

$$\vec{z}' = \begin{cases} \vec{z}, & \text{if } \vec{z} \in X \\ \texttt{rand}(X), & \text{otherwise} \end{cases}$$

  where $\texttt{rand}(Y)$ returns a random element from a set Y.

Effectively, the ith partial discrete gradient operator $\partial_i L(\vec{z}, \vec{\lambda})$ returns a *differential vector* $\vec{d}$ for the ith-component of $\vec{z}$ which decreases the Lagrangian function most. Note that $\vec{d}$ is selected according to the GENET (or E-GENET) state update rule [66, 60, 7, 6, 32]. Now, the discrete gradient operator $\triangle_{\vec{z}}$ is defined by the following set of equations

$$\pi_i(\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})) = \partial_i L(\vec{z}, \vec{\lambda}), \quad \forall \, i \in U. \tag{7.23}$$

When $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$, there is no change in the vector $\vec{z}$. In this case, either a solution is found or a stationary point is reached.

The Lagrange multipliers $\vec{\lambda}$ are updated according to the incompatibility functions. Since the incompatibility function returns the amount of violation of a violated constraint, the magnitude of the update can be greater than 1. This is a major difference from the binary case.

With the above modifications, the discrete Lagrangian search procedure $\mathcal{LSDL}$ in Algorithm 6.1 can be applied directly for solving general CSP's.

7.3 A Comparison between the Binary and the General Formulation

The general formulation is also applicable to binary CSP's. In order to compare the difference between the binary and the general formulations, we consider the

same simple CSP shown in Figure 4.1. In this problem, we have three variables u_1, u_2 and u_3, each with a domain $\{1, 2\}$, and two constraints $u_1 = u_2$ and $u_2 < u_3$. As described in chapter 4, the binary formulation gives us the following zero-one integer constrained minimization problem:

$$\min\ N(\vec{z}) \tag{7.24}$$

subject to

$$z_{\langle u_1,1\rangle} + z_{\langle u_1,2\rangle} = 1, \tag{7.25}$$

$$z_{\langle u_2,1\rangle} + z_{\langle u_2,2\rangle} = 1, \tag{7.26}$$

$$z_{\langle u_3,1\rangle} + z_{\langle u_3,2\rangle} = 1, \tag{7.27}$$

$$g_{\langle u_1,1\rangle\langle u_2,2\rangle}(\vec{z}) = z_{\langle u_1,1\rangle} z_{\langle u_2,2\rangle} = 0, \tag{7.28}$$

$$g_{\langle u_1,2\rangle\langle u_2,1\rangle}(\vec{z}) = z_{\langle u_1,2\rangle} z_{\langle u_2,1\rangle} = 0, \tag{7.29}$$

$$g_{\langle u_2,1\rangle\langle u_3,1\rangle}(\vec{z}) = z_{\langle u_2,1\rangle} z_{\langle u_3,1\rangle} = 0, \tag{7.30}$$

$$g_{\langle u_2,2\rangle\langle u_3,1\rangle}(\vec{z}) = z_{\langle u_2,2\rangle} z_{\langle u_3,1\rangle} = 0, \tag{7.31}$$

$$g_{\langle u_2,2\rangle\langle u_3,2\rangle}(\vec{z}) = z_{\langle u_2,2\rangle} z_{\langle u_3,2\rangle} = 0, \tag{7.32}$$

where $\vec{z} = (z_{\langle u_1,1\rangle}, z_{\langle u_1,2\rangle}, z_{\langle u_2,1\rangle}, z_{\langle u_2,2\rangle}, z_{\langle u_3,1\rangle}, z_{\langle u_3,2\rangle})$ is a vector of zero-one integer variables, $N(\vec{z})$ is the objective function defined in either (4.18) or (4.19), equations (7.25 – 7.27) are the constraints for enforcing valid assignments for the CSP, and equations (7.28 – 7.32) are the constraints for the incompatibility functions.

In the general formulation, the same problem is represented by the following integer constrained minimization problem:

$$\min\ N(\vec{z}) \tag{7.33}$$

subject to

$$z_{u_1} \in D_{u_1}, \tag{7.34}$$

$$z_{u_2} \in D_{u_2}, \tag{7.35}$$

$$z_{u_3} \in D_{u_3}, \tag{7.36}$$

$$g_{u_1=u_2}(\vec{z}) = 0, \tag{7.37}$$

$$g_{u_2<u_3}(\vec{z}) = 0, \tag{7.38}$$

where $\vec{z} = (z_{u_1}, z_{u_2}, z_{u_3})$ is a vector of integer variables, $N(\vec{z})$ is the objective function defined in (7.4), (7.5) or (7.6), and constraints (7.34 – 7.36) and (7.37 – 7.38) are the constraints to ensure valid assignments and the constraints for the incompatibility functions respectively.

From this simple example, we find that the two formulations are different in five aspects. First, in the binary formulation, the zero-one integer variables are used to represent each possible label of a CSP. On the other hand, the general formulation denotes each variable of a CSP by an integer variable. Hence, the total number of integer variables of the resultant integer constrained minimization problem is greatly reduced. Second, because of the different representation of variables of a CSP, the constraints for restricting valid assignments for a CSP are different in the two formulations. Third, instead of breaking down every constraint of a CSP into a set of incompatible tuples and defining an incompatibility function for each incompatible tuple, the general formulation uses a single incompatibility function for each constraint of a CSP. Therefore, the storage requirement is lowered. Fourth, the discrete gradient operators are defined differently to accommodate the difference in the two formulations. Fifth, since the incompatibility functions defined in the general formulation return the amount of constraint violation, the Lagrange multipliers $\vec{\lambda}$ can be updated with a magnitude greater than 1.

Although there are quite a number of differences between the binary and the general formulation, the same discrete Lagrangian search procedure $\mathcal{LSDL}$ can be applied without any modification.

7.4 Experiments

In order to evaluate our formulation, especially our definition of the incompatibility functions, we implement an instance of $\mathcal{LSDL}$ for general CSP's. This instance, denoted by $\mathcal{LSDL}$(GENERAL), has the following parameters:

- N: since the role of an objective function is to guide the search, the objective function $N(\vec{z})$ defined in (7.5) is used.

- $\triangle_{\vec{z}}$: the discrete gradient operator is specified by

  for each variable $i \in U$ **do**

  update $\pi_i(\vec{z})$: $\pi_i(\vec{z}) \leftarrow \pi_i(\vec{z}) - \partial_i L(\vec{z}, \vec{\lambda})$

  end for

  where ∂_i is the partial gradient operator defined in (7.22).

- $I_{\vec{z}}$: unlike the binary case, the procedure for initializing the integer variables $\vec{z}$ greedily is quite computationally expensive. Hence, we choose to randomly initialize the value of $\vec{z}$ in such a way that $z_i \in D_i$, for all $i \in U$.

- $I_{\vec{\lambda}}$: the value of each λ_c, for all $c \in C$, is initialized as follows,

$$\lambda_c = \begin{cases} g_c(\vec{z}^0), & \text{if } g_c(\vec{z}^0) \neq 0 \\ 1 \quad , & \text{otherwise} \end{cases}$$

  where $\vec{z}^0$ is the initial value of the integer vector $\vec{z}$. Note that this approach is similar to the assignment scheme of initial penalty values of the optimized E-GENET [33, 69].

- $U_{\vec{\lambda}}$: because of the definition of incompatibility functions, the value of a single Lagrange multiplier may affect many possible states of the search space. Therefore, the Lagrange multipliers $\vec{\lambda}$ are updated only when $\partial_i L(\vec{z}, \vec{\lambda}) = 0$, for all $i \in U$.

Various benchmark problems, such as the N-queens problems, the graph-coloring problems and the car-sequencing problems, are used in our experiments. We compare our results with that of E-GENET [32, 33, 69], an extension of GENET for general CSP's. Whenever possible, we quote results of both original E-GENET and optimized E-GENET from [32, 33, 34, 69], results of which are average and median CPU time of 10 runs obtained on a SUN SPARCstation 10 model 30. Our experiments are performed on a SUN SPARCstation 10 model 40. Both average and median CPU time of 10 runs are presented. Unless otherwise specified, unbracketed and bracketed results represent the average and median CPU time respectively.

7.4.1 The N-queens Problems

The N-queens problems is used to verify our definition of the $\neq$ constraint, a linear arithmetic constraint, and the **among** constraint.

When the N-queens problem is expressed by the $\neq$ constraints, the problem is modeled as follows. Each row i, for $i = 1, 2, \ldots, N$, of an $N \times N$ chessboard is represented by a variable q_i with domain $D_{q_i} = \{1, 2, \ldots, N\}$. The constraints

$$q_i \neq q_j, \quad \forall\, i \neq j \text{ and } i, j = 1, 2, \ldots, N \tag{7.39}$$

$$|q_i - q_j| \neq |i - j|, \quad \forall\, i \neq j \text{ and } i, j = 1, 2, \ldots, N \tag{7.40}$$

state that no two queens can be on the same column or on the same (positive or negative) diagonal respectively. Benchmarking results are summarized in Table 7.1. From the experiment, we find that $\mathcal{LSDL}$(GENERAL) outperforms the original E-GENET. This promising results confirm the feasibility of handling a constraint as a whole, instead of breaking it down into a set of incompatible tuples, which is the case in E-GENET. Hence, the storage requirement can be greatly reduced. We do not compare $\mathcal{LSDL}$(GENERAL) against the optimized E-GENET since the results of the optimized E-GENET are obtained by modeling the N-queens problems with the **noattack** constraints, instead of the $\neq$ constraints. On the other hand, the performance of $\mathcal{LSDL}$(GENERAL) is much worse than those of $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(MAX). The great difference in efficiency may be due to the difference in the two formulations. Since each incompatibility function defined in the binary formulation represent an incompatible tuple, it can guide the search in a more refined fashion.

In order to model the N-queens problem with the **among** constraint, we use a Boolean formulation. Each square (i, j) of the chessboard is denoted by a variable q_{ij}, for all $i, j = 1, 2, \ldots, N$, with domain $\{0, 1\}$. The variable q_{ij} is 1 if a queen is placed on the square (i, j); otherwise, it is 0. The constraints are

- for each row of the chessboard,

$$\textbf{among}(1, [\text{variables of the row}], [0, \ldots, 0], [1]). \tag{7.41}$$

N	Original E-GENET Median CPU Time (sec)	$\mathcal{LSDL}$(GENERAL) Average (Median) CPU Time (sec)
10	0.046	0.025 (0.033)
20	0.165	0.155 (0.150)
30	0.510	0.457 (0.392)
40	1.222	1.060 (1.050)
50	3.582	2.095 (1.725)
60	6.840	3.513 (3.375)
70	9.902	4.122 (4.183)
80	19.752	7.180 (7.092)
90	28.467	11.663 (11.375)
100	37.582	15.145 (15.883)
110	42.211	20.945 (20.833)
120	61.672	24.657 (23.208)
130	86.083	29.430 (27.575)
140	94.377	41.405 (42.550)
150	152.001	50.047 (52.283)
160	188.033	60.047 (55.817)
170	219.317	56.083 (53.742)
180	264.543	71.040 (73.333)
190	316.562	86.517 (83.642)
200	439.952	96.715 (89.458)

Table 7.1: Results of $\mathcal{LSDL}$(GENERAL) on the N-queens problems modeled with the $\neq$ constraint

N	Optimized E-GENET CPU Time (sec)	$\mathcal{LSDL}$(GENERAL) CPU Time (sec)
10	0.009 (0.010)	0.018 (0.017)
20	0.129 (0.125)	0.083 (0.083)
30	0.255 (0.270)	0.242 (0.242)
40	0.493 (0.460)	0.550 (0.600)
50	1.580 (1.540)	0.958 (0.850)
60	1.256 (1.125)	1.888 (1.758)
70	2.792 (2.788)	1.930 (1.883)
80	2.209 (2.454)	2.952 (3.142)
90	4.136 (4.053)	4.490 (3.808)
100	4.660 (4.905)	7.357 (6.092)

Table 7.2: Results of $\mathcal{LSDL}$(GENERAL) on the N-queens problems modeled with the **among** constraint

- for each column of the chessboard,

$$\textbf{among}(1, [\text{variables of the column}], [0, \ldots, 0], [1]). \tag{7.42}$$

- for each diagonal of the chessboard,

$$\textbf{among}([0, 1], [\text{variables of the diagonal}], [0, \ldots, 0], [1]). \tag{7.43}$$

Table 7.2 shows the results for 10- to 100-queens problems. Except some problem instances, the performance of $\mathcal{LSDL}$(GENERAL) is comparable to that of the optimized E-GENET. The poor performance of $\mathcal{LSDL}$(GENERAL) on some problems can be accounted for as follows. In the optimized E-GENET, a contribution function [33, 69] is defined for the **among** constraint to speed up the search. However, $\mathcal{LSDL}$(GENERAL) does not have this kind of search information. Therefore, the optimized E-GENET is more efficient.

7.4.2 The Graph-coloring Problems

We use the graph-coloring problem to further evaluate the $\neq$ constraint. In the experiment, the set of hard graph-coloring problems from the DIMACS archive [27] is

Nodes	Colors	Original E-GENET Median CPU Time	$\mathcal{LSDL}$(GENERAL) Average (Median) CPU Time	Success Ratio
125	17	2.5 hr	–	0/10
125	18	2.6 min	1.2 min (1.0 min)	10/10
250	15	7.4 sec	14.1 sec (12.6 sec)	10/10
250	29	5 hr	–	0/10

Table 7.3: Results of $\mathcal{LSDL}$(GENERAL) on the hard graph-coloring problems

used. The execution limit of $\mathcal{LSDL}$(GENERAL) is set to one million iterations. The benchmarking results are shown in Table 7.3. Besides the CPU time, we also report the success ratio of $\mathcal{LSDL}$(GENERAL). The performance of $\mathcal{LSDL}$(GENERAL) is found to be worse than that of the original E-GENET. Among the four problem instances, $\mathcal{LSDL}$(GENERAL) fails to find any solution for the problem with 125 nodes 17 colors and the problem with 250 nodes 29 colors within the execution limit. On the other hand, $\mathcal{LSDL}$(GENERAL) outperforms the original E-GENET on the problem with 125 nodes 18 colors. Since the results of the optimized E-GENET are not available, the performance of $\mathcal{LSDL}$(GENERAL) and that of the optimized E-GENET is not compared. The performance of $\mathcal{LSDL}$(GENERAL) is also worse than that of $\mathcal{LSDL}$(MAX). Since $\mathcal{LSDL}$(GENERAL) represents a constraint with a single incompatibility function, useful information that guides the search is lost. As a result, the performance is degraded.

7.4.3 The Car-Sequencing Problems

The goal of the car-sequencing problem is to schedule cars into an assembly line so that different options can be installed on the cars and the utilization constraints are satisfied [9]. The problem is used to test the incompatibility function of the `atmost` constraint. In the experiment, a set of randomly generated problems described in [7] is used. All problems consist of 200 variables with domains varying from 17 to 28 values and approximately 1000 `atmost` constraints

Utilization %	% Succ. Runs (Median Repairs)			
	Non-binary GENET	Original E-GENET	Optimized E-GENET	$\mathcal{LSDL}$(GENERAL)
60	84 (463)	74 (223.5)	100 (282.5)	100 (301.5)
65	87 (426)	80 (223.5)	99 (262)	99 (322.0)
70	83 (456)	81 (241)	100 (280.5)	100 (348.5)
75	85 (730)	84 (339)	97 (331)	98 (424.5)
80	50 (4529)	53 (576)	73 (537)	75 (643.0)

Table 7.4: Results of $\mathcal{LSDL}$(GENERAL) on the car-sequencing problems

of various number of variables. Totally 50 problems, 10 for each utilization percentage ranging from 60% to 80%, are tested. We compare the performance of $\mathcal{LSDL}$(GENERAL) with the original E-GENET, the optimized E-GENET and the non-binary GENET [7], which is an extended GENET model for handling the `illegal` constraint, the `atmost` constraint and the `notequal` constraint. The execution limit of the non-binary GENET is set to one million repairs, while the execution limit of $\mathcal{LSDL}$(GENERAL), the original E-GENET and the optimized E-GENET is 1000 repairs.

The results are listed in Table 7.4. All successful runs of $\mathcal{LSDL}$(GENERAL) terminate in less than 15 seconds. $\mathcal{LSDL}$(GENERAL) is better than the non-binary GENET both in terms of the successful percentage and the median number of repairs. When comparing the results between $\mathcal{LSDL}$(GENERAL) and the original E-GENET, we found that $\mathcal{LSDL}$(GENERAL) always gives a higher percentage of successful runs. This performance is comparable to that of the optimized E-GENET. Therefore, we can conclude that $\mathcal{LSDL}$(GENERAL) is at least as efficient as the optimized E-GENET on handling the `atmost` constraint. On the other hand, the median number of repairs of $\mathcal{LSDL}$(GENERAL) is slightly higher than those of the original E-GENET and the optimized E-GENET.

7.5 Inadequacy of the Formulation

As confirmed by $\mathcal{LSDL}$(GENERAL), the proposed formulation shows certain success on extending $\mathcal{LSDL}$ for general CSP's. In some cases, however, the defined incompatibility function, such as the one for the $\neq$ constraint, is not sufficient to guide the search. In this section, we first point out the weaknesses of the incompatibility functions. A possible improvement is then given. Experiments show that the modification can significantly boost the search efficiency.

7.5.1 Insufficiency of the Incompatibility Functions

Given an integer constrained minimization problem, the discrete Lagrange multiplier method performs saddle point search on the *cost surface* defined by the Lagrangian function

$$L(\vec{z}, \vec{\lambda}) = N(\vec{z}) + \sum_{c \in C} \lambda_c g_c(\vec{z}).$$

Apart from the objective function $N(\vec{z})$, the incompatibility functions $g_c(\vec{z})$ also provide additional force to guide the search. However, the incompatibility functions defined for general CSP's are not sufficient.

In the general formulation, instead of decomposing a constraint into a set of incompatible tuples, we define a single incompatibility function for each constraint. Although this approach can significantly reduce the storage requirement, useful search information is lost. Since an incompatibility function of a constraint defines the cost of an assignment, a set of incompatible tuples is weighted by the same cost. Therefore, a number of large plateaus are generated in the cost surface. As a result, the search process is easily trapped in plateaus, which are difficult to escape.

Furthermore, unlike the binary case, the set of incompatible tuples of a constraint is associated with a single Lagrange multiplier. When a Lagrange multiplier is updated, instead of penalizing the current assignment, the whole set of all incompatible tuples is affected. Hence, a potential path to the solution may be blocked more easily [41]. For example, consider a CSP with 4 variables, a, b, c and

d, each with a domain $\{1,2\}$, and 15 constraints

$$c_1 : a+b \neq c+d, \quad c_2 : a+c \neq b+d, \quad c_3 : a+d \neq b+c,$$
$$c_4 : a+b \neq c, \quad c_5 : a+b \neq d, \quad c_6 : a+c \neq b,$$
$$c_7 : a+c \neq d, \quad c_8 : a+d \neq b, \quad c_9 : a+d \neq c,$$
$$c_{10} : b+c \neq a, \quad c_{11} : b+c \neq d, \quad c_{12} : b+d \neq a,$$
$$c_{13} : b+d \neq c, \quad c_{14} : c+d \neq a, \quad c_{15} : c+d \neq b.$$

The vector of integer variables $\vec{z}$ and the vector of Lagrange multipliers $\vec{\lambda}$ for the associated integer constrained minimization problem are $\vec{z} = (z_a, z_b, z_c, z_d)$ and $\vec{\lambda} = (\lambda_{c_1}, \lambda_{c_2}, \lambda_{c_3}, \lambda_{c_4}, \lambda_{c_5}, \lambda_{c_6}, \lambda_{c_7}, \lambda_{c_8}, \lambda_{c_9}, \lambda_{c_{10}}, \lambda_{c_{11}}, \lambda_{c_{12}}, \lambda_{c_{13}}, \lambda_{c_{14}}, \lambda_{c_{15}})$ respectively. Among all possible assignments of $\vec{z}$, only $(1,2,2,2), (2,1,2,2), (2,2,1,2)$ and $(2,2,2,1)$ are solutions of the problem.

Suppose the objective function (7.5) is used, and initially $\vec{\lambda} = \vec{\lambda}_0 = \vec{1}$ and $\vec{z} = (1,1,1,1)$. The values of the Lagrangian function $L(\vec{z}, \vec{\lambda}_0)$ for each $\vec{z}$ are given in the second column of Table 7.5. Since $(1,1,1,1)$ has the same Lagrangian value as its neighboring point $(2,1,1,1)$, $(1,2,1,1)$, $(1,1,2,1)$ and $(1,1,1,2)$, it is a stationary point. Therefore, the Lagrange multipliers $\vec{\lambda}$ are updated once to get $\vec{\lambda}_1 = (2,2,2,1,1,1,1,1,1,1,1,1,1,1,1)$. The new Lagrangian values $L(\vec{z}, \vec{\lambda}_1)$ for each $\vec{z}$ are shown in the third column of Table 7.5. Note that since a single Lagrange multiplier is associated with a set of incompatible tuples, $\vec{\lambda}_1$ not only affects the Lagrangian value of current assignment $(1,1,1,1)$, it also changes the Lagrangian value of $(2,2,1,1)$, $(2,1,2,1)$, $(2,1,1,2)$, $(1,2,2,1)$, $(1,2,1,2)$, $(1,1,2,2)$ and $(2,2,2,2)$. Now, $(1,1,1,1)$ is no longer a stationary point. The integer vector $\vec{z}$ can change to one of $(2,1,1,1)$, $(1,2,1,1)$, $(1,1,2,1)$ and $(1,1,1,2)$. Suppose $(2,1,1,1)$ is chosen. Again, $(2,1,1,1)$ is a stationary point. The Lagrange multipliers are updated twice to increase the penalty. The new $\vec{\lambda}$ becomes $\vec{\lambda}_2 = (2,2,2,1,1,1,1,1,1,3,1,3,1,3,1)$ and the new Lagrangian values $L(\vec{z}, \vec{\lambda}_2)$ for different $\vec{z}$ are given in the fourth column of Table 7.5. Further update changes $\vec{z}$ back to $(1,1,1,1)$. Similarly, vectors $(1,2,1,1)$, $(1,1,2,1)$ and $(1,1,1,2)$ are tried and then back to $(1,1,1,1)$ in turn. After these transitions, $\vec{\lambda}$ becomes $\vec{\lambda}_3 = (2,2,2,3,3,3,3,3,3,3,3,3,3,3,3)$ and the Lagrangian values $L(\vec{z}, \vec{\lambda}_3)$ for

$\vec{z}$	$L(\vec{z}, \vec{\lambda}_0)$	$L(\vec{z}, \vec{\lambda}_1)$	$L(\vec{z}, \vec{\lambda}_2)$	$L(\vec{z}, \vec{\lambda}_3)$
$(1,1,1,1)$	6	9	9	9
$(2,1,1,1)$	6	6	12	12
$(1,2,1,1)$	6	6	6	12
$(1,1,2,1)$	6	6	6	12
$(1,1,1,2)$	6	6	6	12
$(2,2,1,1)$	8	10	12	14
$(2,1,2,1)$	8	10	12	14
$(2,1,1,2)$	8	10	12	14
$(1,2,2,1)$	8	10	10	14
$(1,2,1,2)$	8	10	10	14
$(1,1,2,2)$	8	10	10	14
$(1,2,2,2)$	0	0	0	0
$(2,1,2,2)$	0	0	0	0
$(2,2,1,2)$	0	0	0	0
$(2,2,2,1)$	0	0	0	0
$(2,2,2,2)$	6	9	9	9

Table 7.5: The value of the Lagrangian function $L(\vec{z}, \vec{\lambda})$ for different integer variables $\vec{z}$ and Lagrange multipliers $\vec{\lambda}$ of a CSP

each $\vec{z}$ are shown in the fifth column of Table 7.5. The whole process repeats and the algorithm oscillates between $(1,1,1,1)$, $(2,1,1,1)$, $(1,2,1,1)$, $(1,1,2,1)$ and $(1,1,1,2)$ indefinitely.

In order to obtain a solution, the algorithm must pass through an integer vector with two integer variables equal to 2. However, since the Lagrangian values of $(2,2,1,1)$, $(2,1,2,1)$, $(2,1,1,2)$, $(1,2,2,1)$, $(1,2,1,2)$ and $(1,1,2,2)$ are affected by previous updates of Lagrange multipliers, they are always greater than those of $(2,1,1,1)$, $(1,2,1,1)$, $(1,1,2,1)$ and $(1,1,1,2)$. Therefore, $(2,2,1,1)$, $(2,1,2,1)$, $(2,1,1,2)$, $(1,2,2,1)$, $(1,2,1,2)$ and $(1,1,2,2)$ are never visited. In other words, potential paths to the solutions are blocked.

7.5.2 Dynamic Illegal Constraint

In order to overcome the above weaknesses, we propose the following scheme. When a constraint is violated, an `illegal` constraint of current incompatible tuple

is added to the problem. We call this newly added `illegal` constraint a *dynamic illegal constraint*. After this modification, the integer constrained minimization problem becomes

$$\min \; N(\vec{z}) \tag{7.44}$$

subject to

$$z_i \in D_i, \quad \forall\, i \in U \tag{7.45}$$

$$g_c(\vec{z}) = 0, \quad \forall\, c \in C \tag{7.46}$$

$$g_d(\vec{z}) = 0, \quad \forall\, d \in D \tag{7.47}$$

where D is the set of dynamic illegal constraints and $g_d(\vec{z})$ are the incompatibility function of the `illegal` constraint. Obviously, the new minimization problem is equivalent to the original one. However, the Lagrangian function for the new problem is reconstructed as

$$L(\vec{z}, \vec{\lambda}, \vec{\Lambda}) = N(\vec{z}) + \sum_{c \in C} \lambda_c g_c(\vec{z}) + \sum_{d \in D} \Lambda_d g_d(\vec{z}) \tag{7.48}$$

where $\vec{\Lambda} = (\ldots, \Lambda_d, \ldots)$ is a vector of Lagrange multipliers associated with the dynamic illegal constraints. The cost surface is modified by the extra term $\sum_{d \in D} \Lambda_d g_d(\vec{z})$. As a result, the dynamic illegal constraints provide an additional force to guide the search.

7.5.3 Experiments

A variant of $\mathcal{LSDL}$, called D-$\mathcal{LSDL}$(GENERAL), is implemented to verify the effectiveness of the proposed scheme. D-$\mathcal{LSDL}$(GENERAL), where "D" stands for "dynamic," has the same parameters as $\mathcal{LSDL}$(GENERAL). The only difference is the ability to generate dynamic illegal constraints. In D-$\mathcal{LSDL}$(GENERAL), when a certain constraint is violated, the corresponding dynamic illegal constraint $g_d(\vec{z})$ of the violated tuple is introduced to the system, and the initial value of Λ_d is set to 1. The N-queens problems and the hard graph-coloring problems from the DIMACS archive [27] are used to compare the performance of D-$\mathcal{LSDL}$(GENERAL)

and $\mathcal{LSDL}$(GENERAL). All experiments are performed on a SUN SPARCstation 10 model 40. Both average (unbracketed) and median (bracketed) CPU time of 10 runs are presented.

Benchmarking results for the N-queens problems are shown in Table 7.6. D-$\mathcal{LSDL}$(GENERAL) improves the performance of $\mathcal{LSDL}$(GENERAL) significantly. For the graph-coloring problems, the execution limit of both D-$\mathcal{LSDL}$(GENERAL) and $\mathcal{LSDL}$(GENERAL) is set to one million iterations. Timing results as well as the success ratios are reported in Table 7.7. D-$\mathcal{LSDL}$(GENERAL) outperforms $\mathcal{LSDL}$(GENERAL) both in terms of the CPU time and the success ratio. As a result, the usefulness of dynamic illegal constraints is confirmed.

N	$\mathcal{LSDL}$(GENERAL) CPU Time (sec)	D-$\mathcal{LSDL}$(GENERAL) CPU Time (sec)
10	0.025 (0.033)	0.007 (0.000)
20	0.155 (0.150)	0.097 (0.092)
30	0.457 (0.392)	0.250 (0.233)
40	1.060 (1.050)	0.463 (0.425)
50	2.095 (1.725)	1.332 (1.392)
60	3.513 (3.375)	1.958 (1.658)
70	4.122 (4.183)	3.228 (3.225)
80	7.180 (7.092)	4.848 (4.642)
90	11.663 (11.375)	6.588 (6.533)
100	15.145 (15.883)	9.337 (9.633)
110	20.945 (20.833)	12.830 (12.100)
120	24.657 (23.208)	16.463 (16.575)
130	29.430 (27.575)	21.563 (22.542)
140	41.405 (42.550)	24.137 (23.125)
150	50.047 (52.283)	34.307 (33.000)
160	60.047 (55.817)	37.605 (37.342)
170	56.083 (53.742)	44.308 (45.600)
180	71.040 (73.333)	53.705 (52.083)
190	86.517 (83.642)	59.192 (58.942)
200	96.715 (89.458)	67.098 (65.358)

Table 7.6: Results of D-$\mathcal{LSDL}$(GENERAL) on the N-queens problems

Nodes	Colors	$\mathcal{LSDL}$(GENERAL) Average (Median) CPU Time	$\mathcal{LSDL}$(GENERAL) Success Ratio	D-$\mathcal{LSDL}$(GENERAL) Average (Median) CPU Time	D-$\mathcal{LSDL}$(GENERAL) Success Ratio
125	17	–	0/10	4.8 hr (4.8 hr)	2/10
125	18	1.2 min (1.0 min)	10/10	1.0 min (46.9 sec)	10/10
250	15	14.1 sec (12.6 sec)	10/10	6.1 sec (6.3 sec)	10/10
250	29	–	0/10	18.9 hr (13.0 hr)	8/10

Table 7.7: Results of D-$\mathcal{LSDL}$(GENERAL) on the hard graph-coloring problems

Chapter 8

Concluding Remarks

We conclude the thesis by giving our contributions and possible directions for future research.

8.1 Contributions

The contributions of our work can be summarized as follows. We derive from the GENET model a two-step transformation for converting any binary CSP into a zero-one integer constrained minimization problem. With the help of this transformation, well-known constrained optimization techniques, such as the Lagrange multiplier method, can be applied directly for tackling CSP's. Based on the transformed zero-one integer constrained minimization problems, we propose a generic discrete Lagrangian search scheme $\mathcal{LSDL}$ for solving binary CSP's. $\mathcal{LSDL}$, which has five degrees of freedom, represents a class of discrete Lagrangian search algorithms. By instantiating $\mathcal{LSDL}$ with different parameters, algorithms with different efficiency and behavior can be generated.

We formally establish the equivalence between the GENET model, a representative of the class of heuristic repair methods, and an instance of $\mathcal{LSDL}$. This result not only provides a theoretical foundation for better understanding of GENET, but also suggests a dual viewpoint of GENET: as a heuristic repair method and as a discrete Lagrange multiplier method. As a result, the discrete

Lagrangian search scheme $\mathcal{LSDL}$ provides various important guidance for the design of better heuristic repair algorithms. In order to evaluate our approach, we implement $\mathcal{LSDL}$(GENET), a discrete Lagrangian reconstruction of GENET. Various experiments show that $\mathcal{LSDL}$(GENET) exhibits the same good convergence behavior as other GENET implementations found in the literature. Variants of $\mathcal{LSDL}$(GENET) obtained from the dual viewpoints are also examined. Our best variant $\mathcal{LSDL}$(MAX) is found to be more efficient than $\mathcal{LSDL}$(GENET). By incorporating lazy arc consistency to $\mathcal{LSDL}$, we can achieve additional order of magnitude improvements for problems with arc inconsistency, and suffer from little overhead for the problems which are already arc consistent.

We also extend $\mathcal{LSDL}$ for general CSP's. In this extension, we convert a general CSP into an integer constrained minimization problem and define a new discrete gradient operator for $\mathcal{LSDL}$. The main difference between the general and the binary formulation is that, instead of defining an incompatibility function for each incompatible tuple, we use a single incompatibility function to represent a constraint. Hence, the storage requirement is greatly reduced. With the new discrete gradient operator defined to accommodate the change of formulation, the discrete Lagrangian search scheme $\mathcal{LSDL}$ can be applied without any special modification. We implement $\mathcal{LSDL}$(GENERAL), an instance of $\mathcal{LSDL}$ for solving general CSP's, to verify our approach. The performance of $\mathcal{LSDL}$(GENERAL) is found to be comparable with that of E-GENET in most test problems. Although this straightforward generalization gives us some promising results, it does not work so well in general. In our experiments, $\mathcal{LSDL}$(GENERAL) performs much worse than $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(MAX). Therefore, much work is required in the future. In addition, since a constraint is represented by a single incompatibility function, large plateaus, which make the search more difficult, are generated. We propose dynamic illegal constraints to overcome this weakness. Experiments confirm that the addition of dynamic illegal constraints can substantially improve the performance of $\mathcal{LSDL}$(GENERAL).

8.2 Discussions

DLM [62, 54, 53] is a discrete Lagrangian search algorithm for solving SAT problems. Our $\mathcal{LSDL}$ framework is constructed according to DLM for solving CSP's. Although both DLM and $\mathcal{LSDL}$ apply the discrete Lagrange multiplier method to solve SAT problems or CSP's, there are some differences between them. First, the $\mathcal{LSDL}$ procedure consists of five degrees of freedom. For example, any objective functions that satisfy the correspondence requirement can be used, and each Lagrange multiplier can be initialized differently. Hence, different parameters can be chosen for tackling different problems. On the other hand, DLM does not emphasize this kind of freedom. It always chooses the total number of unsatisfied clauses of the SAT problem as the objective function, and always initializes the Lagrange multipliers with a fixed value. However, DLM employs, on top of the discrete Lagrangian search, a number of different tuning heuristics for different problems. For instance, it uses an additional tabu list to remember states visited, and resets the Lagrange multipliers after a number of iterations.

Second, $\mathcal{LSDL}$ searches on a smaller search space than DLM. Since $\mathcal{LSDL}$ is targeted for solving CSP's, the set of constraints, which restrict valid assignments for CSP's, is incorporated in the discrete gradient operator. Thus, only valid assignments are searched in $\mathcal{LSDL}$. On the contrary, DLM lacks this kind of restriction. Any possible assignments, including those are invalid for CSP's, are considered. As a result, the efficiency of DLM is affected.

Third, the two algorithms use different discrete gradient operators to perform saddle point search. In DLM, the discrete gradient operator considers all Boolean variables of the SAT problem as a whole and modifies one Boolean variable in each update. However, in $\mathcal{LSDL}$, the zero-one integer variables which correspond to a variable of the CSP are grouped together and updated at the same time. Hence, the discrete gradient operator used in $\mathcal{LSDL}$ is more suitable for solving CSP's. In addition, the discrete gradient operator of DLM uses the hill-climbing strategy to update the Boolean variables. In this strategy, the first assignment which

leads to a decrease in the Lagrangian function is selected to update the current assignment. In $\mathcal{LSDL}$, the discrete gradient operator always modifies the zero-one integer variables such that there is a maximum decrease in the Lagrangian function.

In summary, since the $\mathcal{LSDL}$ framework explores the structure of CSP's, it can be regarded as a specialization of DLM for solving CSP's.

8.3 Future Work

Our work represents a major step toward the understanding of heuristic repair methods. Interesting problems remain. On the theoretical side, at least one important property of $\mathcal{LSDL}$ and other heuristic repair methods is still unknown. Our experience suggests that GENET and $\mathcal{LSDL}$ terminate for solvable CSP's. However, *under what condition(s), do the algorithms always terminate, if at all?* A possible approach to tackle this question is to investigate whether the convergence properties of the continuous Lagrange multiplier method can be extended for the discrete case. Furthermore, the five degrees of freedom of $\mathcal{LSDL}$ suggest many possibilities for new heuristic repair algorithms. In our research, only a small number of parameter combinations are investigated. Other parameters and their interaction should be explored in the future. The new variable ordering heuristic developed for GENET [58] should also be included in our $\mathcal{LSDL}$ framework.

Our extension of $\mathcal{LSDL}$ for general CSP's is preliminary. Only a few general constraints are implemented. In the future, we should define new incompatibility functions for new constraints, such as the `cumulative` constraint, the `diffn` constraint and the `cycle` constraint [2]. These general constraints are useful for modeling complex real-life applications. Our proposed general formulation is straightforward. However, its performance is much worse than that of the binary formulation. We should further investigate other possible approach in the future. The idea of dynamic illegal constraints is new. Although experiments show that they can improve the performance of the search, the results are purely empirical.

Hence, we should further examine the theoretical aspect of this idea. In addition, the possibility of applying dynamic illegal constraints to other constraint satisfaction techniques should be investigated.

Last but not least, the optimizational nature of the $\mathcal{LSDL}$ framework suggests applying $\mathcal{LSDL}$ to tackle constraint satisfaction optimization problems and over-constrained systems. A *constraint satisfaction optimization problem* (CSOP) is a CSP with an objective function to be optimized. In $\mathcal{LSDL}$, since a CSP can be completely defined by the incompatibility functions, we can simply replace the objective function of $\mathcal{LSDL}$ with the one required in CSOP. Hence, when $\mathcal{LSDL}$ terminates, the solution returned will be an assignment that satisfies all constraints and minimized the objective function. In an over-constrained system, constraints are usually classified into *hard constraints* and *soft constraints*. Hard constraints are the constraints that must be satisfied by the solution, while soft constraints are those that can be violated. The goal is to find an assignment that satisfies all hard constraints and minimizes the number (or cost) of violations of soft constraints. In $\mathcal{LSDL}$, an over-constrained system can be modeled as follows. The incompatibility functions, which must be satisfied, is used to represent all hard constraints of the system, and the objective function is constructed by the soft constraints. In this way, over-constrained systems are handled in the same manner as CSOP's. Although these approaches for CSOP's and over-constrained systems are quite straightforward, the feasibility should be further confirmed.

Bibliography

[1] H. M. Adorf and M. D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA, 1990.

[2] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modeling*, 20(12):97–123, 1994.

[3] J. F. Boyce, C. H. D. Dimitropoulos, G. vom Scheidt, and J. G. Taylor. GENET and tabu search for combinatorial optimization problems. In *World Congress on Neural Networks*, Washington D. C., 1995.

[4] Y. J. Chang and B. W. Wah. Lagrangian techniques for solving a class of zero-one integer linear problems. In *Proceedings of International Conference on Computer Software and Applications, IEEE*, pages 156–161, 1995.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[6] A. Davenport. *Extensions and Evaluation of GENET in Constraint Satisfaction.* PhD thesis, Department of Computer Science, University of Essex, 1997.

[7] A. Davenport, E. Tsang, C. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Seattle, WA)*, pages 325–330, 1994.

[8] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

[9] M. Dincbas, H. Simonis, and P. V. Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of the Eighth European Conference on Artificial Intelligence*, pages 290–295, 1988.

[10] A. E. Eiben, P-E Raue, and Zs. Ruttkay. GA-easy and GA-hard constraint satisfaction problems. In *Proceedings of the ECAI'94 Workshop on Constraint Processing*, 1994.

[11] A. E. Eiben, P-E Raue, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In *Proceedings of the First IEEE Congress on Evolutionary Computing*, pages 542–547. AAAI Press/MIT Press, 1994.

[12] M. L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[13] J. Frank, P. Cheeseman, and J. Allen. Weighting for godat: Learning heuristics for GSAT. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 338–343. AAAI Press/MIT Press, 1996.

[14] E. C. Freuder. The many paths to satisfaction. In M. Meyer, editor, *Constraint Processing*, LNCS 923, pages 103–119. Springer-Verlag, 1995.

[15] I. P. Gent and T. Walsh. The enigma of SAT hill-climbing procedures. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1992.

[16] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 28–33. AAAI Press/MIT Press, 1993.

[17] A. M. Geoffrion. Lagrangian relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974.

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

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

[20] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

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

[22] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[23] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39:345–351, 1987.

[24] M. R. Hestenes. *Optimization Theory – The Finite Dimensional Case.* Wiley & Sons, NY, 1975.

[25] A. C. Hindmarsh. ODEPACK, a systematized collection of ODE solvers. In R. S. Stepleman *et al.*, editor, *Scientific Computing*, pages 55–64. North-Holland, Amsterdam, 1983.

[26] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[27] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part 2 graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[28] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

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

[30] T. L. Lau and E. P. K. Tsang. Applying a mutation-based genetic algorithm to processor configuration problems. In *Proceedings of the Eighth International Conference on Tools with Artificial Intelligence*, pages 17–24, 1996.

[31] J-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[32] J. H. M. Lee, H. F. Leung, and H. W. Won. Extending GENET for non-binary CSP's. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence*, pages 338–343, Washington D.C., USA, November 1995. IEEE Computer Society Press.

[33] J. H. M. Lee, H. F. Leung, and H. W. Won. Towards a more efficient stochastic constraint solver. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 338–352, Cambridge, Massachusetts, USA, August 1996. Springer-Verlag, LNCS 1118.

[34] J. H. M. Lee, H. F. Leung, and H. W. Won. A comprehensive and efficient constraint library using local search. In *Proceedings of the Eleventh Australian Joint Conference on Artificial Intelligence*, July 1998. (To appear).

[35] J. H. M. Lee and V. W. L. Tam. Towards the integration of artificial neural networks and constraint logic programming. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 446–452, 1994.

[36] J. H. M. Lee and V. W. L. Tam. A framework for integrating artificial neural networks and logic programming. *International Journal of Artificial Intelligence Tools*, 4(1&2):3–32, 1995.

[37] A. K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[38] B. Mazure, L. Sais, and E. Gregoire. Tabu search for SAT. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 281–285. AAAI Press/MIT Press, 1997.

[39] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 17–24. AAAI Press/The MIT Press, 1990.

[40] 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*, 58:161–205, 1992.

[41] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (Washington, DC)*, pages 40–45, 1993.

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

[43] J. Platt and A. Barr. Constrained differential optimization. In *Proceedings of Neural Information Processing System Conference*, 1987.

[44] P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 262–267, 1993.

[45] P. Prosser. Exceptionally hard problems. The comp.constraints newsgroup, September 1994.

[46] M. C. Riff. From quasi-solutions to solution: An evolutionary algorithm to solve csp. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 367–381. Springer-Verlag, 1996.

[47] I. Rivin and R. Zabih. An algebraic approach to constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 284–289, 1989.

[48] B. Selman. Private Communication, July 1997.

[49] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 290–295, 1993.

[50] B. Selman and H. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 46–51, 1993.

[51] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343. AAAI Press/MIT Press, 1994.

[52] B. Selman, H. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446. AAAI Press/MIT Press, 1992.

[53] Y. Shang. *Global Search Methods for Solving Nonlinear Optimization Problems.* PhD thesis, Department of Computer Science, University of Illinois, 1997.

[54] Y. Shang and B. W. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.

[55] D. M Simmons. *Nonlinear Programming for Operations Research.* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[56] P. J. Stuckey and V. Tam. Extending GENET with lazy arc consistency. Technical report, Department of Computer Science, University of Melbourne, 1996.

[57] P. J. Stuckey and V. Tam. Extending E-GENET with lazy constraint consistency. In *Proceedings of the Ninth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97)*, 1997.

[58] P. J. Stuckey and V. Tam. Improving GENET and E-GENET by new variable ordering strategies. In *Proceedings of the International Conference on Computational Intelligence and Multimedia Applications (ICCIMA'98)*, 1998.

[59] P. J. Stuckey and V. Tam. Extending GENET with lazy arc consistency. *IEEE Transactions on Systems, Man, and Cybernetics*, (To appear).

[60] E. P. K Tsang and C. J. Wang. A generic neural network approach for constraint satisfaction problems. In J. G. Taylor, editor, *Neural Network Applications*, pages 12–22. Springe-Verlag, 1992.

[61] B. W. Wah and Y. J. Chang. Trace-based methods for solving nonlinear global optimization and satisfiability problems. *Journal of Global Optimization*, 10(2):107–141, 1997.

[62] B. W. Wah and Y. Shang. A discrete lagrangian-based global-search method for solving satisfiability problems. In *Proceedings of DIMACS Workshop on Satisfiability Problem: Theory and Applications*, 1996.

[63] B. W. Wah and Y. Shang. Discrete lagrangian-based search for solving MAX-SAT problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 378–383, 1997.

[64] B. W. Wah, Y. Shang, and Z. Wu. Discrete lagrangian method for optimizing the design of multiplierless qmf filter banks. In *Proceedings of the Fifteenth International Conference on Application Specific Array Processors*, pages 529–538. IEEE, 1997.

[65] B. W. Wah, T. Wang, Y. Shang, and Z. Wu. Improving the performance of weighted lagrange-multiplier methods for nonlinear constrained optimization. In *Proceedings of the Ninth International Conference on Tools with Artificial Intelligence*, pages 224–231. IEEE, 1997.

[66] C. J. Wang and E. P. K. Tsang. Solving constraint satisfaction problems using neural networks. In *Proceedings of the IEE 2nd Conference on Artificial Neural Networks*, pages 295–299, 1991.

[67] T. Warwick and E. P. K. Tsang. Using a genetic algorithm to tackle the processors configuration problem. In *Proceedings of ACM Symposium on Applied Computing*, pages 217–221, 1994.

[68] T. Warwick and E. P. K. Tsang. Tackling car sequencing problems using a generic genetic algorithm. *Evolutionary Computation*, 3(3):267–298, 1995.

[69] H. W. Won. E-GENET: A stochastic constraint solver. Master's thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1997.

[70] 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, 1998.