S0004-3702(00)00053-9/FLA AID:1754 Vol.00(0)
ELSEV2s 2000/09/04 Prn:15/09/2000; 9:26

AIJ1754

P.1(1-39)
by:EL p. 1

ELSEVIER

Artificial Intelligence 00 (2000) 1–39

Artificial Intelligence

www.elsevier.com/locate/artint

A Lagrangian reconstruction of GENET

Kenneth M.F. Choi [a], Jimmy H.M. Lee [a], Peter J. Stuckey [b,*]

[a] *Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong SAR, People's Republic of China*

[b] *Department of Computer Science and Software Engineering, University of Melbourne, Parkville 3052, Australia*

Received 31 January 1999; received in revised form 11 August 2000

Abstract

GENET is a heuristic repair algorithm which demonstrates impressive efficiency in solving some large-scale and hard instances of constraint satisfaction problems (CSPs). In this paper, we draw a surprising connection between GENET and discrete Lagrange multiplier methods. Based on the work of Wah and Shang, we propose a discrete Lagrangian-based search scheme $\mathcal{LSDL}$, defining a class of search algorithms for solving CSPs. We show how GENET can be reconstructed from $\mathcal{LSDL}$. The dual viewpoint of GENET as a heuristic repair method and a discrete Lagrange multiplier method allows us to investigate variants of GENET from both perspectives. Benchmarking results confirm that first, our reconstructed GENET has the same fast convergence behavior as the original GENET implementation, and has competitive performance with other local search solvers DLM, WalkSAT, and WSAT(OIP), on a set of difficult benchmark problems. Second, our improved variant, which combines techniques from heuristic repair and discrete Lagrangian methods, is always more efficient than the reconstructed GENET, and can better it by an order of magnitude.

Keywords: Constraint satisfaction problems; Local search; Discrete Lagrangian method

1. Introduction

A *constraint satisfaction problem* (CSP) [1] is a tuple (U, D, C), where U is a finite set of variables, D defines a finite set D_x, called the *domain* of x, for each $x \in U$, and C is a finite set of constraints restricting the combination of values that the variables can take.

* Corresponding author.
E-mail address: pjs@cs.mu.oz.au (P.J. Stuckey).

0004-3702/00/$ – see front matter
PII: S0004-3702(00)00053-9

A *solution* is an assignment of values from the domains to their respective variables so that all constraints are satisfied simultaneously. CSPs are well-known to be NP-hard in general.

The traditional approach to solving CSPs is a combination of backtracking tree search and constraint propagation. Various variable and value ordering heuristics are also used to speed up the search process. Another class of solution techniques is based on local search, for example GSAT [2] and Tabu Search [3,4]. In the context of constraint satisfaction, local search first generates an initial variable assignment (or state) before making local adjustments (or repairs) to the assignment iteratively until a solution is reached. Based on a discrete stochastic neural network [5], a class of local search techniques, known as heuristic repair methods and exemplified by the work reported in [6] and [7], has been shown to be effective in solving some large-scale and some computationally hard classes of CSPs. Heuristic repair works by performing variable repairs to minimize the number of constraint violations. As with other local search algorithms, heuristic repair methods can be trapped in a *local minimum* (or *local maximum* depending on the optimization criteria), a non-solution state in which no further improvement can be made. To help escape from the local minimum, Minton et al. [6] proposed random restart, while Davenport et al. [7] and Morris [8] proposed modifying the landscape of the search surface. Following Morris, we call these *breakout methods*.

While the idea of minimizing conflicts is simple and intuitive, little is known theoretically about why and how this class of algorithms work at all and so well, although Minton et al. provide a statistical model and probabilistic analysis of the algorithms for random CSPs. In this paper, we show that GENET [7] is equivalent to a form of Lagrange multiplier method [9], a well-known technique for solving constrained optimization problems with a wealth of literature on its formal properties. We do so by first transforming a CSP into an integer constrained minimization problems and defining a Lagrangian function of the transformed problem. This result is useful not just in establishing a formal characterization of heuristic repair algorithms. It also allows us to gain important insights into the various design issues of heuristic repair methods.

Because of the dual viewpoint of GENET as a heuristic repair method and a discrete Lagrange multiplier method, we can explore variants of GENET which incorporate modifications from either viewpoint. We introduce $\mathcal{LSDL}$, a general scheme defining a class of discrete Lagrangian search algorithms for solving CSPs. We reconstruct GENET as an instantiation ($\mathcal{LSDL}$(GENET)) of the $\mathcal{LSDL}$ scheme and explore variations that arise from considering it as a discrete Lagrange multiplier method. We also show how the lazy consistency optimization [10] developed for GENET (considered as a heuristic repair method) can be transferred to $\mathcal{LSDL}$ in a straightforward manner. Thus we gain benefits from both viewpoints. Benchmarking results confirm that our reconstructed GENET has the same fast convergence behavior as the original GENET implementation. Second, by exploring the design space of $\mathcal{LSDL}$ using simple experiments, we are able to define an improved variant $\mathcal{LSDL}$(IMP), which combines techniques from heuristic repair and discrete Lagrangian methods. $\mathcal{LSDL}$(IMP) is always more efficient than the reconstructed GENET, and can better it by an order of magnitude. Third, we demonstrate that $\mathcal{LSDL}$(GENET), $\mathcal{LSDL}$(IMP), and their lazy versions, are robust across our benchmark suite, in the sense that without any tuning for the different problems, they still have acceptable performance.

Wah et al. [11,12] were the first to propose a discrete version of the Lagrangian theory but their framework and implementation has only been applied to dealing with SAT problems. Our work is based on their theory, applied to solving finite domain CSPs. A main contribution of this paper, however, is in establishing the connection between Lagrangian-based techniques and existing heuristic repair methods. We show that better algorithms can result from such a dual viewpoint. An important aim of our work is to devise a suitable local search solver for embedding in a constraint programming system. We are interested in algorithms that are good for solving at least an entire class of problems without user intervention and fine tuning. The best $\mathcal{LSDL}$ instances that we have constructed so far, while efficient, are also shown to be robust against problem variations. In other words, our method, unlike many local search solvers does not require tuning of *execution* parameters, to achieve acceptable performance for different problem classes.

The paper, a revised and enhanced version of [13], is organized as follow. The GENET network is briefly introduced in Section 2, followed by a description of the discrete Lagrangian formulation of CSPs in Section 3. In the same section, we give the $\mathcal{LSDL}$ search scheme, which is a result of the discrete Lagrangian formulation. Section 4 discusses the $\mathcal{LSDL}$ parameters in details. We show formally that GENET is an instance of $\mathcal{LSDL}$ in Section 5 and discuss how we created an improved instance of the $\mathcal{LSDL}$ scheme. In Section 6, we then briefly introduce lazy arc consistency before showing how it can be incorporated in $\mathcal{LSDL}$. Experimental results of the reconstructed GENET and an improved variant are presented in Section 7 before related work in Section 8. In Section 9, we summarize our contributions and shed light on possible future directions of research.

2. A brief overview of GENET

The GENET [7] model consists of two components: a network architecture and a convergence procedure. The former governs the network representation of a CSP, while the latter formulates how the network is updated in the solution searching process. While GENET can solve both binary and certain non-binary constraints, we limit our attention to only the binary subset of GENET.

2.1. Network architecture

Consider a binary CSP (U, D, C), a GENET *network* $\mathcal{N}$ representing this CSP consists of a set of label nodes and connections. Each variable $i \in U$ is represented in GENET by a cluster of *label nodes* $\langle i, j\rangle$, one for each value $j \in D_i$. 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 *on* if its output is 1; otherwise, it is *off*. A constraint c on variable i_1 and i_2 is represented by *weighted connections* between incompatible label nodes in clusters i_1 and i_2 respectively. Two label nodes $\langle i_1, j_1\rangle$ and $\langle i_2, j_2\rangle$ are connected if $i_1 = j_1 \wedge i_2 = j_2$ violates c. Each connection has a *weight*, initially set to -1. The *input* $I_{\langle i,j\rangle}$ to a label node $\langle i, j\rangle$ is:

$$I_{\langle i,j\rangle} = \sum_{\langle k,l\rangle \in A(\mathcal{N},\langle i,j\rangle)} W_{\langle i,j\rangle\langle k,l\rangle} V_{\langle k,l\rangle}, \tag{1}$$

```
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
```

Fig. 1. The GENET convergence procedure.

where $A(\mathcal{N}, \langle i, j\rangle)$ is the set of all label nodes connected to $\langle i, j\rangle$ and $W_{\langle i,j\rangle\langle k,l\rangle}$ is the weight of the connection between $\langle i, j\rangle$ and $\langle k, l\rangle$. A *state* $\mathcal{S}$ of network $\mathcal{N}$ is a pair $(\vec{V}, \vec{W})$, 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 every pair of connected label nodes $\langle i, j\rangle$ and $\langle k, l\rangle$. A state is *valid* if exactly one label node in each cluster is on. A valid state of a GENET network $\mathcal{N}$ induces a *valid variable assignment* to the variables of the CSP corresponding to $\mathcal{N}$. A *solution state* of a network has the input of all on label nodes being zero.

2.2. Convergence procedure

The convergence procedure shown in Fig. 1 defines how a GENET network changes states and connection weights before it reaches a solution state.

Initially, a label node in each cluster is selected on randomly; other label nodes are off. GENET performs iterative repair by minimizing the number of constraint violations using the *state update rule*. When the network is trapped in a local maximum,[1] the *heuristic learning rule* is invoked to help the network escape from the local maximum. A solution is found when all on label nodes have zero input. In the convergence procedure, a superscript s in a quantity X, as in X^s, denotes the value of X in sth iteration. There are a few points to note regarding the convergence procedure.

First, clusters can be updated in parallel either synchronously or asynchronously. In *synchronous* update, all clusters calculate their node inputs and perform state update at

[1] When any neighboring state has a total input less than or equal to the current total input.

the same time. In *asynchronous* update, each cluster performs input calculation and state update independently. Synchronous update can cause oscillations [7], while, in practice, asynchronous update leads to convergence if the network has solutions. In most sequential implementations, asynchronous update can be simulated by updating clusters in sequence in a predefined order.

Second, there could be more than one label node with the maximum input during a state update. To select the next label node to be on, GENET adopts the following heuristic rule. Let P be the set of nodes with maximum input. If the label node currently on is in P, it remains on. Otherwise, $\texttt{rand}(P)$ is selected to be on, where $\texttt{rand}(Y)$ is a function returning a random element from a set Y. The state update rule is a direct application of the min-conflict heuristic [6].

Third, we can associate an *energy* $E(\mathcal{N}, \mathcal{S})$ with every state $\mathcal{S}$ for a network $\mathcal{N}$:

$$E(\mathcal{N}, \mathcal{S}) = \sum_{(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{N}} V_{\langle i,j\rangle} W_{\langle i,j\rangle\langle k,l\rangle} V_{\langle k,l\rangle}. \tag{2}$$

$E(\mathcal{N}, \mathcal{S})$ is always non-positive with negative weights. The energy $E(\mathcal{N}, \mathcal{S}_0)$ of a solution state $\mathcal{S}_0$ is always 0, a global maximum value for $E(\mathcal{N}, \mathcal{S})$. The convergence procedure thus carries out an optimization process for the energy function $E(\mathcal{N}, \mathcal{S})$.

Fourth, an *iteration* constitutes one pass over the outermost loop. $W^s_{\langle i,j\rangle\langle k,l\rangle}$ denotes the weight of connection between label nodes $\langle i, j\rangle$ and $\langle k, l\rangle$ and $V^s_{\langle i,j\rangle}$ denotes the output of label node $\langle i, j\rangle$ in the sth iteration. Weight update aims to decrease the energy associated with the local maximum. Thus learning has the effect of pulling down the local maximum in the search surface. This learning rule is similar to the breakout method [8].

The CSP, where $U = \{u_1, u_2, u_3\}$, $D_{u_1} = D_{u_2} = D_{u_3} = \{1, 2, 3\}$ and $C = \{u_1 < u_2, \mathit{even}(u_2 + u_3)\}$, gives a GENET network as illustrated in Fig. 2(a). There are inhibitory connections between any two label nodes which violate one of the constraints. For example, there is a connection between $\langle u_2, 1\rangle$ and $\langle u_3, 2\rangle$ since this combination of variable assignment violates $\mathit{even}(u_2 + u_3)$; hence $W_{\langle u_2,1\rangle\langle u_3,2\rangle} = -1$ initially. The state illustrated has the label nodes $\langle u_1, 3\rangle$, $\langle u_2, 2\rangle$, and $\langle u_3, 1\rangle$ on, representing the assignment $u_1 = 3, u_2 = 2, u_3 = 1$, and has energy -2. Updating the u_2 cluster of label nodes proceeds by calculating $I_{\langle u_2,1\rangle} = -1$, $I_{\langle u_2,2\rangle} = -2$ and $I_{\langle u_2,3\rangle} = -1$ so that one of $\langle u_2, 1\rangle$ or $\langle u_2, 3\rangle$

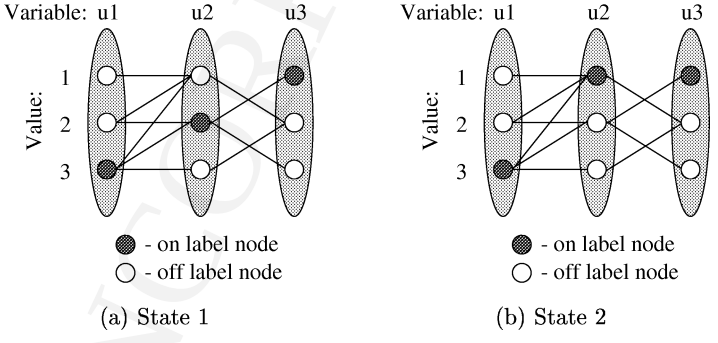

Fig. 2. Example GENET network.

should be selected randomly to be on next. Suppose $\langle u_2, 1\rangle$ is selected and the resulting state is shown in Fig. 2(b). This state has energy -1 and is a local maximum so that updating any cluster further would result in no state change. Thus the heuristic learning rule is applied, modifying $W_{\langle u_1,3\rangle\langle u_2,1\rangle}$ to be -2; hence the energy becomes -2 and the network is no longer in a local maximum. The state update rule can again be applied, trying to maximize the energy of the network.

As far as we know, the convergence of the GENET procedure is still an *open* problem.

3. A discrete Lagrangian formulation of CSPs

The energy perspective of the GENET convergence procedure suggests an optimization approach to constraint satisfaction. This approach allows us to borrow well-known optimization techniques from the literature. In this section, we show a transformation for converting any binary CSP into an integer constrained minimization problem. A discrete version of the Lagrange multiplier method [11] is used to solve the resulting minimization problem.

3.1. CSP as integer constrained minimization problem

An *integer constrained minimization* problem consists of a set of integer variables $\vec{z}$, an objective function $f(\vec{z})$ and a set G of constraints defining the *feasible space* of the problem. The goal is to find a global minimum $\vec{z}^*$ in the feasible space so that the value of $f(\vec{z}^*)$ is minimized and each constraint of G is satisfied. In the following, we present the transformation that converts a GENET network into an integer constrained minimization problem.

Given a GENET network $\mathcal{N}$ of a binary CSP (U, D, C). Suppose that each domain D_i for all $i \in U$ is a set of integers. Each cluster (variable) i of the GENET network (CSP) is represented by an integer variable z_i. The value of 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).

For each connection $(\langle i, j\rangle, \langle k, l\rangle) \in \mathcal{N}$, we define an *incompatibility function*

$$g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) = \begin{cases} 1, & \text{if } z_i = j \wedge z_k = l, \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables. The function $g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z})$ returns 1 if value j is assigned to variable i and value l is assigned to variable k, 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 integer constrained minimization problem has the form,

$$\min f(\vec{z}) \tag{4}$$

subject to

$$z_i \in D_i, \quad \forall\, i \in U, \tag{5}$$

$$g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) = 0, \quad \forall\, (\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I}, \tag{6}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $\mathcal{I}$ is the set of all incompatible label pairs $(\langle i,j\rangle, \langle k,l\rangle)$. The constraints defined in (5) are used to enforce valid assignments for the CSP. Since the solution space of a CSP is defined entirely by the constraints (5)–(6), it is equal to the feasible space of the associated integer constrained minimization problem. The objective function $f(\vec{z})$ serves only to exert additional force to guide solution searching.

The objective function $f(\vec{z})$ is defined in such a way that *every solution of the CSP must correspond to a constrained global minimum of the associated integer constrained minimization problem* (4)–(6). 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 [14], 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

$$f(\vec{z}) = \sum_{(\langle i,j\rangle, \langle k,l\rangle) \in \mathcal{I}} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}), \tag{7}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables.

Another possibility is the constant objective function

$$f(\vec{z}) = 0. \tag{8}$$

The constant 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 the binary CSP shown in Fig. 2(a). The variables $U = \{u_1, u_2, u_3\}$ are represented by a vector of integer variables $\vec{z} = (z_1, z_2, z_3)$. The domains D become constraints $z_i \in \{1,2,3\},\ 1 \leqslant i \leqslant 3$. The inhibitory connections are represented by incompatibility functions

$$g_{\langle u_1,1\rangle\langle u_2,1\rangle}(\vec{z}) = z_1 = 1 \wedge z_2 = 1,$$
$$g_{\langle u_1,2\rangle\langle u_2,1\rangle}(\vec{z}) = z_1 = 2 \wedge z_2 = 1,$$
$$g_{\langle u_1,2\rangle\langle u_2,2\rangle}(\vec{z}) = z_1 = 2 \wedge z_2 = 2,$$
$$g_{\langle u_1,3\rangle\langle u_2,1\rangle}(\vec{z}) = z_1 = 3 \wedge z_2 = 1,$$
$$g_{\langle u_1,3\rangle\langle u_2,2\rangle}(\vec{z}) = z_1 = 3 \wedge z_2 = 2,$$
$$g_{\langle u_1,3\rangle\langle u_2,3\rangle}(\vec{z}) = z_1 = 3 \wedge z_2 = 3,$$
$$g_{\langle u_2,1\rangle\langle u_3,2\rangle}(\vec{z}) = z_2 = 1 \wedge z_3 = 2,$$
$$g_{\langle u_2,2\rangle\langle u_3,1\rangle}(\vec{z}) = z_2 = 2 \wedge z_3 = 1,$$
$$g_{\langle u_2,2\rangle\langle u_3,3\rangle}(\vec{z}) = z_2 = 2 \wedge z_3 = 3,$$
$$g_{\langle u_2,3\rangle\langle u_3,2\rangle}(\vec{z}) = z_2 = 3 \wedge z_3 = 2.$$

The transformation is completed by choosing either (7) or (8) as the objective function. Hence, solving the CSP now becomes finding a constrained global minimum of the associated integer constrained minimization problem.

3.2. $\mathcal{LSDL}$: A discrete Lagrange multiplier method

The Lagrange multiplier method is a well-known technique for solving constrained optimization problems [9]. It provides a systematic approach for handling constraints, while maintaining numerical stability and solution accuracy. Until recently the method has only been applied to real variable problems. Initially we converted the resulting integer problems into real variable constrained optimization problems by introducing additional constraints to restrict the real variables to hold integer values only [15]. Although this approach is possible, handling of the additional constraints incurs costly computation making it useless in practice.

Recently Shang and Wah extended the classical Lagrange multiplier method to deal with discrete problems [11,16,17]. Consider the integer constrained minimization problem (4)–(6) transformed from the CSP (U, D, C). Similar to the classical Lagrange multiplier method [9], the *Lagrangian function* $L(\vec{z}, \vec{\lambda})$ is constructed as

$$L(\vec{z}, \vec{\lambda}) = f(\vec{z}) + \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \lambda_{\langle i,j\rangle\langle k,l\rangle} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}), \tag{9}$$

where $\vec{z} = (\ldots, z_i, \ldots)$ is a vector of integer variables and $\vec{\lambda} = (\ldots, \lambda_{\langle i,j\rangle\langle k,l\rangle}, \ldots)$ is a vector of *Lagrange multipliers*. Note that the constraints defined by (5), 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 discussed below.

A constrained minimum of the integer constrained minimization problem (4)–(6) 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* $(\vec{z}^*, \vec{\lambda}^*)$ [11,16,17] of the Lagrangian function $L(\vec{z}, \vec{\lambda})$ is defined by the condition

$$L(\vec{z}^*, \vec{\lambda}) \leqslant L(\vec{z}^*, \vec{\lambda}^*) \leqslant L(\vec{z}, \vec{\lambda}^*) \tag{10}$$

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 1 (Discrete saddle point theorem [12]). *A vector of integer variables $\vec{z}^*$ is a constrained minimum of the integer constrained minimization problem*

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

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

Note that under the conditions of the above theorem it is easy to show (see [12]) that any point $(\vec{z}^*, \vec{\lambda}')$ with $\vec{\lambda}' \geqslant \vec{\lambda}^*$ is also a saddle point of the Lagrangian function $L(\vec{z}, \vec{\lambda})$. This means that there is no requirement to decrease Lagrange multipliers during the search for a saddle point.

The construction of the constrained minimization problem (4)–(6) corresponding to a CSP ensures that each 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 problem (4)–(6) are always non-negative. Hence the discrete saddle point theorem is applicable.

Corollary 2. *For a problem of the form* (4)–(6) $\vec{z}^*$ *is a constrained minimum of the problem if and only if there exist Lagrange multipliers* $\vec{\lambda}^*$ *such that* $(\vec{z}^*, \vec{\lambda}^*)$ *is 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 $\vec{\lambda}$ [18]. Instead of using differential equations, the discrete Lagrange multiplier method uses difference equations [11,16,17]

$$\vec{z}^{s+1} = \vec{z}^s - GD\big(\triangle_{\vec{z}} L(\vec{z}^s, \vec{\lambda}^s), \vec{z}^s, \vec{\lambda}^s, s\big), \tag{11}$$

$$\vec{\lambda}^{s+1} = \vec{\lambda}^s + \vec{g}(\vec{z}^s), \tag{12}$$

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

The discrete gradient $\triangle_{\vec{z}}$ can be defined as follows. Given a vector of integer variables $\vec{z} = (\ldots, z_i, \ldots)$, we define the *projection operator* π_i, for all $i \in U$, as

$$\pi_i(\vec{z}) = z_i, \tag{13}$$

which gives the *i th-component* of $\vec{z}$. In other words, $\pi_i(\vec{z})$ returns the integer variable corresponding to variable i in U. Furthermore, let

$$N_i(\vec{z}) = \big\{\vec{z}' \mid \big(\pi_i(\vec{z}') \in D_i\big) \wedge \big(\forall j \in U\colon\ (j \neq i) \wedge \big(\pi_j(\vec{z}') = \pi_j(\vec{z})\big)\big)\big\}$$

be the *neighborhood* of a point $\vec{z}$ along the ith direction. The constraints defined in (5) are incorporated in the neighborhood $N_i(\vec{z})$, for all $i \in U$, to enforce valid assignment for each integer variable z_i. The *i th component of the discrete gradient* $\pi_i(\triangle_{\vec{z}})$, for all $i \in U$, is defined as

$$\pi_i\big(\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})\big) = L(\vec{z}, \vec{\lambda}) - L(\vec{z}', \vec{\lambda}), \tag{14}$$

where $\vec{z}' \in N_i(\vec{z})$ and $L(\vec{z}', \vec{\lambda}) \leqslant L(\vec{z}'', \vec{\lambda})$, for all $\vec{z}'' \in N_i(\vec{z})$. The ith component of the discrete gradient returns the greatest difference in the value of the Lagrangian function along the ith direction. If $\pi_i(\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})) = 0$, then $\vec{z}$ represents a minimum of $L(\vec{z}, \vec{\lambda})$ along the ith direction. When $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$, either a saddle point or a *stationary point* has been reached, at which point the update of $\vec{z}$ terminates.

```
procedure LSDL(f, I_z, I_λ, GD, U_λ)
begin
  s ← 0
  (I_z) initialize the value of z^s
  (I_λ) initialize the value of λ^s
  while (f) L(z^s, λ^s) − f(z^s) > 0 (z^s is not a solution) do
    (GD) z^{s+1} ← z^s − GD(△_z L(z^s, λ^s), z^s, λ^s, s)
    if (U_λ) condition for updating λ holds then
      λ^{s+1} ← λ^s + g(z^s)
    else
      λ^{s+1} ← λ^s
    end if
    s ← s + 1
  end while
end
```

Fig. 3. The $\mathcal{LSDL}(f, I_{\vec{z}}, I_{\vec{\lambda}}, GD, U_{\vec{\lambda}})$ procedure.

The gradient descent function *GD* returns a differential vector for updating the integer vector $\vec{z}$ according to the discrete gradient $\triangle_{\vec{z}}$. It returns $\vec{0}$ when $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$. In general, the gradient descent function *GD* is not unique. It may depend not only on the discrete gradient, but also the current position $(\vec{z}, \vec{\lambda})$ and possibly the iteration number s. We defer discussion on gradient descent functions until Section 4.4.

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 $\mathcal{LSDL}(f, I_{\vec{z}}, I_{\vec{\lambda}}, GD, U_{\vec{\lambda}})$ for solving the integer constrained minimization problems transformed from CSPs is given in Fig. 3.

The $\mathcal{LSDL}$ (pronounced as "Lisdal") procedure performs local search using the discrete Lagrange multiplier method. $\mathcal{LSDL}$ is a specialization of the generic discrete Lagrangian method described in [11]. It has five degrees of freedom, namely (f) the objective function, $(I_{\vec{z}})$ how the integer vector $\vec{z}$ is initialized, $(I_{\vec{\lambda}})$ how the Lagrange multipliers $\vec{\lambda}$ are initialized, (*GD*) the gradient descent function, and $(U_{\vec{\lambda}})$ when to update the Lagrange multipliers $\vec{\lambda}$. Where appropriate, we annotate the algorithm with the parameters in brackets to show where the parameters take effect. The role of each parameter is discussed in the next section.

4. 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 discrete Lagrangian search algorithms with different efficiency and behavior are obtained. In this section, we discuss the various parameters of $\mathcal{LSDL}$ in details.

4.1. Objective function

The objective function $f(\vec{z})$ 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 [19]. Two possible objective functions, presented in Section 3.1, 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 (7),

$$f(\vec{z}) = \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}),$$

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

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

can also be used.

4.2. Integer variable initialization

A good initial assignment of the integer variables $\vec{z}$ can speed up search. As in most local search techniques, the simplest way is to initialize the integer variables $\vec{z}$ randomly in such a way that the constraints (5) are satisfied. On the other hand, Minton et al. [6] suggest that a greedily generated initial assignment can boost the performance of the search. Morris [8] 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 integer vector $\vec{z}$, and selects the assignment which conflicts with the fewest previous selections.

4.3. Lagrange multiplier 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 [20]. If a constraint is known to be more important than the others, its associated Lagrange multipliers can be assigned a larger initial value.

4.4. Gradient descent function

The gradient descent function *GD*, which performs gradient descent in the $\vec{z}$-space, is not unique. One possible gradient descent function, GD_{sync}, can be defined as follows. Given the discrete gradient $\triangle_{\vec{z}} L(\vec{z},\vec{\lambda})$. Let

$$X_i = \left\{\vec{x} \mid \vec{x} \in N_i(\vec{z}) \wedge L(\vec{z},\vec{\lambda}) - L(\vec{x},\vec{\lambda}) = \pi_i\left(\triangle_{\vec{z}} L(\vec{z},\vec{\lambda})\right)\right\}$$

be the set of integer vectors belonging to the neighborhood $N_i(\vec{z})$ which reduce the Lagrangian function $L(\vec{z},\vec{\lambda})$ the most. The gradient descent function GD_{sync} is defined as

$$\pi_i\big(GD_{sync}(\triangle_{\vec{z}}L(\vec{z},\vec{\lambda}),\vec{z},\vec{\lambda},s)\big) = \begin{cases} 0, & \text{if } \pi_i(\triangle_{\vec{z}}L(\vec{z},\vec{\lambda}))=0, \\ \pi_i(\vec{z}) - \pi_i(\texttt{rand}(X_i)), & \text{otherwise,} \end{cases} \tag{15}$$

for all $i \in U$ (recall that U is the set of variables in the CSP and $\texttt{rand}(Y)$ is a function returning a random element from a set Y). The gradient descent function updates all variables synchronously, since each integer variable z_i will be modified to a value which minimizes $L(\vec{z},\vec{\lambda})$ in the neighbourhood $N_i(\vec{z})$. The function GD_{sync} corresponds to what occurs in GENET with synchronous variable update.

Synchronous update is known to have bad behaviour. A simple form of asynchronous gradient descent is to only update each variable one at a time in order. Then

$$GD_{async}\big(\triangle_{\vec{z}}L(\vec{z},\vec{\lambda}),\vec{z},\vec{\lambda},s\big) = e_j \cdot GD_{sync}\big(\triangle_{\vec{z}}L(\vec{z},\vec{\lambda}),\vec{z},\vec{\lambda},s\big) \quad \text{where } j = s \bmod |U| + 1, \tag{16}$$

where e_j is the unit vector in direction j. Since this gradient descent function updates each integer variable one by one, it corresponds to the updating strategy used in most sequential implementations of GENET. Note that since in each iteration only one (fixed) variable is modified, the computation of GD_{async} can be restricted to this direction.

Another possible gradient descent function GD_{dlm} is given as follows. Let

$$X = \big\{\vec{x} \mid \exists i \in U\ \vec{x} \in N_i(\vec{z}) \wedge L(\vec{z},\vec{\lambda}) - L(\vec{x},\vec{\lambda}) = \max_{j\in U} \pi_j\big(\triangle_{\vec{z}}L(\vec{z},\vec{\lambda})\big)\big\}$$

be the set of integer vectors which reduce the Lagrangian function most in some direction i. We define the gradient descent function D_{dlm} as

$$GD_{dlm}\big(\triangle_{\vec{z}}L(\vec{z},\vec{\lambda}),\vec{z},\vec{\lambda},s\big) = \begin{cases} \vec{0}, & \text{if } \triangle_{\vec{z}}L(\vec{z},\vec{\lambda}) = \vec{0}, \\ \vec{z} - \texttt{rand}(X), & \text{otherwise.} \end{cases} \tag{17}$$

Since each 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 gradient descent function. Hence, this new gradient descent function is similar to the one defined in DLM [11,16,17] for solving the SAT problems.

4.5. 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 [11,16,17]. Thus, the condition for updating the Lagrange multipliers is left unspecified in $\mathcal{LSDL}$. For example the Lagrange multipliers can be updated either

(1) at each iteration of the outermost while loop,
(2) after each $|U|$ iterations, or
(3) when $\triangle_{\vec{z}}L(\vec{z},\vec{\lambda}) = \vec{0}$.

Note that the first condition is a direct application of the strategy used in the continuous case while condition (3) corresponds to Morris's breakout method [8]. Condition (2) makes sense with asynchronous gradient descent, since in $|U|$ iterations all variables have been updated once.

5. GENET reconstructed

In this section, we show how we can reconstruct GENET using our discrete Lagrangian approach and then discuss how we improved upon the resulting $\mathcal{LSDL}$ implementation by changing design parameters.

5.1. $\mathcal{LSDL}$(GENET)

Given a binary CSP (U, D, C). The transformation described in Section 3.1 establishes a one-one correspondence between the GENET network of (U, D, C) and the associated integer constrained minimization problem of (U, D, C). The GENET convergence procedure can be obtained by instantiating $\mathcal{LSDL}$ with proper parameters. This instance of $\mathcal{LSDL}$, denoted by $\mathcal{LSDL}$(GENET), has the following parameters:

- f: the constant objective function defined in (8),
- $I_{\vec{z}}$: the 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,
- GD: the gradient descent function GD_{async} defined in (16), and
- $U_{\vec{\lambda}}$: the Lagrange multiplier $\vec{\lambda}$ are updated when $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$.

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 $\mathcal{N}$. Since, in any GENET state $\mathcal{S}$, each cluster i can have at most one on label node, we define $\vec{v} = (\ldots, v_i, \ldots)$ as the variable assignment of a GENET state $\mathcal{S}$ such that $V_{\langle i,v_i\rangle} = 1$ for all $i \in U$. Based on the state update rule of the convergence procedure of GENET and the definition of the gradient descent function (16), we derive the following lemma.

Lemma 3. *Consider a binary CSP (U, D, C), and its corresponding GENET network $\mathcal{N}$ and 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 $\vec{W}^s = -\vec{\lambda}^s$. Then*

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

Proof. In the sth iteration only a single variable $i = (s \bmod |U|) + 1$ is updated. The remaining variables are unchanged. We show that

$$v_i^{s+1} = j \Leftrightarrow z_i^{s+1} = j.$$

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 integer variable vector in the sth iteration with $z^s_i = j$ and z^s_l unchanged for all $l \neq i \in U$.

$$
\begin{aligned}
& v_i^{s+1} = j \\
\Leftrightarrow\ & V^{s+1}_{\langle i,j\rangle} = 1 \quad \text{and} \quad V^{s+1}_{\langle i,k\rangle} = 0, \forall k \neq j \in D_i \\
\Leftrightarrow\ & I^s_{\langle i,j\rangle} \geqslant I^s_{\langle i,k\rangle}, \qquad \forall k \neq j \in D_i \\
\Leftrightarrow\ & \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} \geqslant \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\ & 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) \\
& + \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} \\
\geqslant\ & 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) \\
& + \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\ & 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} 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)} -\lambda^s_{\langle i,l\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} \right) \\
& + \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} (-\lambda^s_{\langle a,b\rangle\langle c,d\rangle}) V^s_{\langle c,d\rangle} \\
\geqslant\ & 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} 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)} -\lambda^s_{\langle i,l\rangle\langle u,v\rangle} V^s_{\langle u,v\rangle} \right) \\
& + \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} (-\lambda^s_{\langle a,b\rangle\langle c,d\rangle}) V^s_{\langle c,d\rangle}, \quad \forall k \neq j \in D_i
\end{aligned}
$$

$$\Leftrightarrow L(\vec{z}^{\,s}_{i|j}, \vec{\lambda}^s) \leqslant L(\vec{z}^{\,s}_{i|k}, \vec{\lambda}^s), \quad \forall k \neq j \in D_i$$
$$\Leftrightarrow \pi_i\big(\triangle_{\vec{z}} L(\vec{z}^{\,s}, \vec{\lambda}^s)\big) = L(\vec{z}^{\,s}, \vec{\lambda}^s) - L(\vec{z}^{\,s}_{i|j}, \vec{\lambda}^s)$$
$$\Leftrightarrow \vec{z}^{\,s}_{i|j} \in X_i.$$

Since both GENET and $\mathcal{LSDL}$(GENET) use the same random selection function $\texttt{rand}(Y)$, by the gradient descent function (16), we have

$$v_i^{s+1} = j \Leftrightarrow z_i^{s+1} = j. \quad \square$$

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 4. *Consider a binary CSP* (U, D, C)*, and its corresponding GENET network* $\mathcal{N}$ *and integer constrained minimization problem. Suppose, in the* s*th 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 two 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 $(\langle i, j\rangle, \langle k, l\rangle) \in \mathcal{N}$,

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

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

Now, a simple application of Lemmas 3 and 4 results in the following theorem, which establishes the equivalence of the GENET convergence procedure and $\mathcal{LSDL}$(GENET).

Theorem 5. *Consider a binary CSP* (U, D, C)*, and its corresponding GENET network* $\mathcal{N}$ *and integer constrained minimization problem. Suppose both GENET and* $\mathcal{LSDL}$(GENET) *use the same random selection function* $\texttt{rand}(Y)$ *and they share the same initial state. For all iteration* s*,* $\vec{v}^{\,s} = \vec{z}^{\,s}$ *and* $\vec{W}^s = -\vec{\lambda}^s$*.*

Proof. The proof is by induction on iterations. 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 Lemmas 3 and 4, 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 induction, the theorem is true for all iterations s. □

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. *Consider a binary CSP* (U, D, C)*, and its corresponding GENET network* $\mathcal{N}$ *and integer constrained minimization problem. We have*

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

where $E(\mathcal{N}, \mathcal{S})$ *is the energy of GENET and* $L(\vec{z}, \vec{\lambda})$ *is the Lagrangian function of* $\mathcal{LSDL}$(GENET).

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

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

Corollary 7. *Consider a binary CSP* (U, D, C)*, and its corresponding GENET network* $\mathcal{N}$ *and integer constrained minimization problem. GENET terminates if and only if* $\mathcal{LSDL}$(GENET) *terminates.*

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

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

Similar results can be proven if, in $\mathcal{LSDL}$, we use instead the objective function $f(\vec{z})$ defined in (7) and initialize $\vec{\lambda}$ to $\vec{0}$. If, however, we use $f(\vec{z})$ defined in (7) and initialize $\vec{\lambda}$ to $\vec{1}$, the Lagrangian function becomes

$$\begin{aligned} L(\vec{z},\vec{\lambda}) &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) + \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \lambda_{\langle i,j\rangle\langle k,l\rangle} g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}) \\ &= \sum_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} \bigl(1+\lambda_{\langle i,j\rangle\langle k,l\rangle}\bigr) g_{\langle i,j\rangle\langle k,l\rangle}(\vec{z}), \end{aligned} \tag{18}$$

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

$$\vec{W} = -(\vec{1}+\vec{\lambda}). \tag{19}$$

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

5.2. *Improving on* $\mathcal{LSDL}$(GENET)

$\mathcal{LSDL}$ is a generic framework defining a class of local search algorithms based on the discrete Lagrange multiplier method. By choosing suitable parameters, different heuristic repair methods can be modeled. The design parameter space for $\mathcal{LSDL}$ is enormous, and in fact can encompass many existing local search algorithms.

In order to search for a better discrete Lagrangian search algorithm for CSPs, we have ran a number of different $\mathcal{LSDL}$ instances on a set of benchmark problems to explore the parameter space of $\mathcal{LSDL}$ [21]. In each experiment, different $\mathcal{LSDL}$ instances were constructed as follows. A single design parameter under test was varied in the $\mathcal{LSDL}$ implementation. Other design parameters remained the same as in $\mathcal{LSDL}$(GENET).

Each new variant was tested on a set of N-queens problems, a set of hard graph-coloring problems from the DIMACS archive [22], and a set of randomly generated CSPs (different from the ones we use in Section 7) are used. These substantial and comprehensive experiments, although by no means exhaustive, help us to select a good combination of $\mathcal{LSDL}$ parameters.

Collecting together all the choices for each single design parameter which led to the best performance defined our improved $\mathcal{LSDL}$ variant which we denote by $\mathcal{LSDL}$(IMP). The parameters are:

- f: the one defined in (7),
- $I_{\vec{z}}$: the integer vector $\vec{z}$ is initialized using the greedy algorithm described in [6],
- $I_{\vec{\lambda}}$: the values of Lagrange multipliers $\vec{\lambda}$ are all initialized to 1,
- GD: the gradient descent function GD_{async} defined in (16), and
- $U_{\vec{\lambda}}$: the Lagrange multiplier $\vec{\lambda}$ are updated after every $|U|$ iterations, where U is the set of variable in the CSP.

Except the hard graph-coloring instances, the problems we use for exploring the $\mathcal{LSDL}$ design parameters were different from the benchmarks used in Section 7. In this exploration, we only tested the behavior of individual parameters. In Section 7, we confirm the improved performance of $\mathcal{LSDL}$(IMP) across a different set of benchmark problems.

6. Extending $\mathcal{LSDL}$

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 for GENET can be used to extend our $\mathcal{LSDL}$ framework.

Arc consistency [1] 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 each 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 $\langle j, l\rangle$ [10,23,24]. 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 [1], 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. 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 [10,23,24] develop lazy arc consistency for the GENET model to avoid the preprocessing phase and instead only remove inconsistent values that are relevant to the GENET search.

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 $\langle j, k\rangle$ [10,23,24]. 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.

Fig. 4 gives a modified input calculation procedure for cluster i of the GENET network $\mathcal{N}$ in a state $\mathcal{S}$ [10,23,24]. The algorithm detects lazy arc inconsistency during the calculation of inputs of each cluster. When an inconsistency for the current value of variable i is detected the global variable "inconsistent(i)" is set to `true`. When the variable i is next updated its current value is removed from its domain.

For example, consider the arc inconsistent CSP and its corresponding GENET network shown in Fig. 5. When calculating the inputs of cluster u_1, we find that each of the label nodes $\langle u_1, 1\rangle$, $\langle u_1, 2\rangle$ and $\langle u_1, 3\rangle$ is connected to label node $\langle u_2, 1\rangle$, the current on label node of cluster u_2. Hence, value 1 for variable u_2 is arc inconsistent with variable u_1, and thus the node $\langle u_2, 1\rangle$ and its associated connections should be removed from the GENET network.

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

Fig. 4. Modified GENET input calculation procedure for lazy arc consistency.

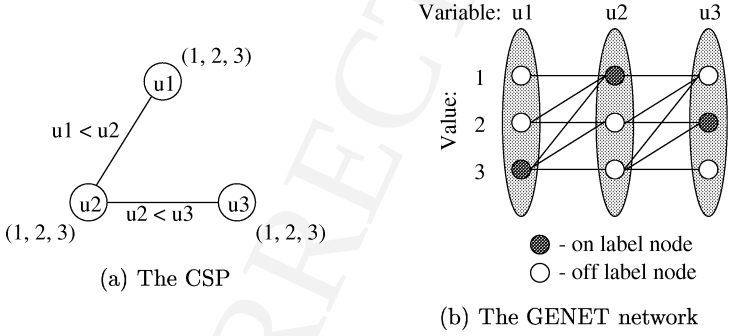

Fig. 5. An arc inconsistent CSP and its corresponding GENET network.

Since lazy arc consistency is targeted at values that are actually selected during the search, which may be much fewer than the entire search space, its overhead is much smaller than that of arc consistency. Experiments show that lazy arc consistency improves GENET substantially for CSPs which are arc inconsistent and does not degrade the performance significantly for problems which are already arc consistent [10,23,24].

Lazy arc consistency can be incorporated in $\mathcal{LSDL}$ in a similar manner. Let $\mathcal{I}$ be the set of all incompatible tuples $(\langle i, j\rangle, \langle k, l\rangle)$. The modified discrete Lagrangian search

procedure Lazy-$\mathcal{LSDL}(N, I_{\vec{z}}, I_{\vec{\lambda}}, GD, U_{\vec{\lambda}})$
begin
  $s \leftarrow 0$
  ($I_{\vec{z}}$) initialize the value of $\vec{z}^s$
  ($I_{\vec{\lambda}}$) initialize the value of $\vec{\lambda}^s$
  while (f) $L(\vec{z}^s, \vec{\lambda}^s) - f(\vec{z}^s) > 0$ ($\vec{z}^s$ 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 $(\langle i, z_i^s \rangle), \langle j, k \rangle) \notin \mathcal{I}$ **then**
        $D_i \leftarrow D_i - \{z_i^s\}$
      end if
    end for
    (GD) $\vec{z}^{s+1} \leftarrow \vec{z}^s - GD(\triangle_{\vec{z}} L(\vec{z}^s, \vec{\lambda}^s), \vec{z}^s, \vec{\lambda}^s, s)$
    if ($U_{\vec{\lambda}}$) condition for updating $\vec{\lambda}$ holds **then**
      $\vec{\lambda}^{s+1} \leftarrow \vec{\lambda}^s + \vec{g}(\vec{z}^s)$
    else
      $\vec{\lambda}^{s+1} \leftarrow \vec{\lambda}^s$
    end if
    $s \leftarrow s + 1$
  end while
end

Fig. 6. The Lazy-$\mathcal{LSDL}(N, I_{\vec{z}}, I_{\vec{\lambda}}, GD, U_{\vec{\lambda}})$ procedure.

algorithm *Lazy-$\mathcal{LSDL}$* is shown in Fig. 6. Similar to GENET, the procedure for detecting lazy arc inconsistency can be integrated in the gradient descent function *GD*. For example, lazy arc inconsistency can be detected during the evaluation of the discrete gradient $\triangle_{\vec{z}}$. We state explicitly (enclosed in the box) the detection procedure in Lazy-$\mathcal{LSDL}$ outside of *GD* to show that lazy arc consistency is independent of the gradient descent function used. In other words, any gradient descent function *GD* defined for $\mathcal{LSDL}$ could be used in Lazy-$\mathcal{LSDL}$ without any special modification.

The detection of lazy arc inconsistency, as appeared in Fig. 6, is costly. In our actual implementation, the detection procedure is performed during the evaluation of the discrete gradient $\triangle_{\vec{z}}$. When calculating the ith component of the discrete gradient $\pi_i(\triangle_{\vec{z}})$, if we find that all domain values of variable i are incompatible with the current assignment of integer variable $\pi_j(\vec{z})$, then we can remove $\pi_j(\vec{z})$ from the domain D_j of variable j.

7. Experiments

We constructed several $\mathcal{LSDL}$ instances for experimentation. They are $\mathcal{LSDL}$(GENET), $\mathcal{LSDL}$(IMP), Lazy-$\mathcal{LSDL}$(GENET), Lazy-$\mathcal{LSDL}$(IMP). In the following, we compare the efficiency of these instances on five sets of problems: a set of hard graph-coloring problems from the DIMACS archive [22], a set of permutation generation problems, a

set of quasigroup completion problems, a set of randomly generated tight binary CSPs with arc inconsistencies, and a set of randomly generated binary CSPs close to the phase transition. We aim to demonstrate the efficiency and robustness of the $\mathcal{LSDL}$ instances using these benchmarks. The $\mathcal{LSDL}$(GENET) implementation has two purposes. First, $\mathcal{LSDL}$(GENET) serves to verify if $\mathcal{LSDL}$(GENET) has the same fast convergence behavior of GENET as reported in the literature. Second, $\mathcal{LSDL}$(GENET) serves as a control in our setup to compare against its variants.

In order to have some feeling for the relative efficiency of the $\mathcal{LSDL}$ solvers with respect to other local search techniques, we also show results of DLM [11], WalkSAT (an improved version of GSAT) [25,26], WSAT(OIP) [27] solving the same problems, running on the same machine. Each of these local search techniques solve only SAT problems or over-constrained linear integer programming problems rather than CSPs directly, hence the benchmarks need to be transformed into appropriate encodings for these solvers (see Section 7.1). Additionally each of these solvers are designed to have execution parameters tuned for each problem class they are applied to in order to obtain the best possible results. For these reasons the comparison with $\mathcal{LSDL}$ is not meant to be definitive, but rather indicative that the $\mathcal{LSDL}$ solvers have competitive performance.

We use the best available implementations of DLM, WalkSAT, and WSAT(OIP) obtained from the original authors by FTP and execute the implementations on the same hardware platform (SUN SPARCstation 10/40 with 32M of memory) as the $\mathcal{LSDL}$ implementations. All implementations are executed using their *default* parameters as they are originally received, described as follows. WalkSAT usually flips a variable in a randomly selected unsatisfied clause which maximizes the total number of satisfying clauses. In every 50 out of 100 flips, however, it chooses a variable in an unsatisfied clause randomly. DLM uses a tabu list of length 50, a flat move limit of 50 and the Lagrange multipliers λ are reset to $\lambda/1.5$ in every 10000 iterations. WSAT(OIP) sets the probability of random move if no improving move is possible to 0.01, the probability of initializing a variable with zero to 0.5. It is also equipped with a history mechanism to avoid flipping the same variable in the near future, and uses a tabu memory of size 1.

Benchmark results of all $\mathcal{LSDL}$ implementations are taken on a SUN SPARCstation 10/40 with 32M of memory. We execute each problem 10 times. The execution limit of the graph-coloring problems is set to 5 million iterations (1 million iterations for WSAT(OIP) since it takes too long to run), the execution limit of the phase transition random CSPs is set to 5 million iterations, and the execution limit of the other problems is set to 1 million iterations. For some problems some solvers do not succeed in finding a solution within the preset execution limit. In such cases, we add a superscript $(x/10)$ besides the timing figures to indicate that only x out of the ten runs are successful. In each cell, the unbracketed and the bracketed timing results represent respectively the CPU time for the average and median of *only* the successful runs. We use a question mark (?) to indicate that the execution results in a memory fault. Unless otherwise specified, all timing results are in seconds and are given to an accuracy of 3 significant figures. Following the practice in the literature, the timing results represent only the search time and exclude the problem setup time (such as reading in the problem specification from a file).

7.1. Problem translation

Since none of the solvers we compare against handle CSPs directly they need to be translated to SAT, for WalkSat and DLM, and integer linear problems for WSAT(OIP).

We consider two schemes to translate CSPs into SAT. We call the scheme adopted for encoding graph-coloring problems in the DIMACS archive the *DIMACS translation*. Given a binary CSP (U, D, C) and its corresponding GENET network. We associate each label node $\langle i, j\rangle$ of GENET with a Boolean variable $b_{\langle i,j\rangle}$. A Boolean variable $b_{\langle i,j\rangle}$ is `true` if its associated label node $\langle i, j\rangle$ is on and `false` otherwise. Each connection $(\langle i, j\rangle, \langle k, l\rangle)$ of GENET is represented by a clause

$$C_{\langle i,j\rangle\langle k,l\rangle} = \neg b_{\langle i,j\rangle} \vee \neg b_{\langle k,l\rangle},$$

which states that the label nodes $\langle i, j\rangle$ and $\langle k, l\rangle$ cannot be both on simultaneously. In addition, there is a clause

$$C_i = b_{\langle i,j_1\rangle} \vee \cdots \vee b_{\langle i,j_n\rangle},$$

where $\{j_1, \ldots, j_n\} = D_i$, for each cluster i of GENET to ensure that at least one label node in each cluster i is on. The resultant SAT problem is to find a truth assignment that satisfies the Boolean formula

$$\bigwedge_{i\in U} C_i \wedge \bigwedge_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} C_{\langle i,j\rangle\langle k,l\rangle},$$

where $\mathcal{I}$ is the set of all incompatible label pairs $(\langle i, j\rangle, \langle k, l\rangle)$.

Consider the CSP shown in Fig. 2(a). According to the above translation, the CSP is transformed into the following SAT problem,

$$\begin{aligned}
&(b_{\langle u_1,1\rangle} \vee b_{\langle u_1,2\rangle} \vee b_{\langle u_1,3\rangle}) \wedge (b_{\langle u_2,1\rangle} \vee b_{\langle u_2,2\rangle} \vee b_{\langle u_2,3\rangle})\\
&\quad \wedge (b_{\langle u_3,1\rangle} \vee b_{\langle u_3,2\rangle} \vee b_{\langle u_3,3\rangle}) \wedge (\neg b_{\langle u_1,1\rangle} \vee \neg b_{\langle u_2,1\rangle}) \wedge (\neg b_{\langle u_1,2\rangle} \vee \neg b_{\langle u_2,1\rangle})\\
&\quad \wedge (\neg b_{\langle u_1,2\rangle} \vee \neg b_{\langle u_2,2\rangle}) \wedge (\neg b_{\langle u_1,3\rangle} \vee \neg b_{\langle u_2,1\rangle}) \wedge (\neg b_{\langle u_1,3\rangle} \vee \neg b_{\langle u_2,2\rangle})\\
&\quad \wedge (\neg b_{\langle u_1,3\rangle} \vee \neg b_{\langle u_2,3\rangle}) \wedge (\neg b_{\langle u_2,1\rangle} \vee \neg b_{\langle u_3,2\rangle}) \wedge (\neg b_{\langle u_2,2\rangle} \vee \neg b_{\langle u_3,1\rangle})\\
&\quad \wedge (\neg b_{\langle u_2,2\rangle} \vee \neg b_{\langle u_3,3\rangle}) \wedge (\neg b_{\langle u_2,3\rangle} \vee \neg b_{\langle u_3,2\rangle}),
\end{aligned}$$

where $b_{\langle u_1,1\rangle}$, $b_{\langle u_1,2\rangle}$, $b_{\langle u_1,3\rangle}$, $b_{\langle u_2,1\rangle}$, $b_{\langle u_2,2\rangle}$, $b_{\langle u_2,3\rangle}$, $b_{\langle u_3,1\rangle}$, $b_{\langle u_3,2\rangle}$ and $b_{\langle u_3,3\rangle}$ are the Boolean variables corresponding to the label nodes of the GENET network. Note that the DIMACS translation allows a variable of CSP to be assigned with more than one value since no clauses are used to enforce valid variable assignment. Thus, the DIMACS translation is inexact.

An *exact translation* can be obtained by augmenting the results of a DIMACS translation with the following clauses

$$C_{ijl} = \neg b_{\langle i,j\rangle} \vee \neg b_{\langle i,l\rangle},$$

for each pair $j, l \in D_i$, to enforce valid assignments for variable i of the CSP. In other words, the resultant SAT problem given by the *exact translation* is

$$\bigwedge_{i\in U} C_i \wedge \bigwedge_{i\in U}\bigwedge_{j,l\in D_i} C_{ijl} \wedge \bigwedge_{(\langle i,j\rangle,\langle k,l\rangle)\in\mathcal{I}} C_{\langle i,j\rangle\langle k,l\rangle},$$

where $\mathcal{I}$ is the set of all incompatible label pairs $(\langle i, j\rangle, \langle k, l\rangle)$. For example, the same CSP shown in Fig. 2(a) is translated to the SAT problem above with the addition of the constraints

$$(\neg b_{\langle u_1,1\rangle} \vee \neg b_{\langle u_1,2\rangle}) \wedge (\neg b_{\langle u_1,2\rangle} \vee \neg b_{\langle u_1,3\rangle}) \wedge (\neg b_{\langle u_1,3\rangle} \vee \neg b_{\langle u_1,1\rangle})$$
$$\wedge (\neg b_{\langle u_2,1\rangle} \vee \neg b_{\langle u_2,2\rangle}) \wedge (\neg b_{\langle u_2,2\rangle} \vee \neg b_{\langle u_2,3\rangle}) \wedge (\neg b_{\langle u_2,3\rangle} \vee \neg b_{\langle u_2,1\rangle})$$
$$\wedge (\neg b_{\langle u_3,1\rangle} \vee \neg b_{\langle u_3,2\rangle}) \wedge (\neg b_{\langle u_3,2\rangle} \vee \neg b_{\langle u_3,3\rangle}) \wedge (\neg b_{\langle u_3,3\rangle} \vee \neg b_{\langle u_3,1\rangle}).$$

While the relative efficiency of these two translations is outside the scope of the paper, we ran our experiments on both encodings. In the benchmarks that we use, the WalkSAT and DLM solvers always performed better on the problems obtained using the DIMACS translation.

To obtain problem specifications for WSAT(OIP), we further translate the resultant clauses into equalities and inequalities as follows. Each clause of the form

$$C_{\langle i,j\rangle\langle k,l\rangle} = \neg b_{\langle i,j\rangle} \vee \neg b_{\langle k,l\rangle}$$

is translated to inequality $E_{\langle i,j\rangle\langle k,l\rangle}$

$$E_{\langle i,j\rangle\langle k,l\rangle}\colon\ b_{\langle i,j\rangle} + b_{\langle k,l\rangle} \leqslant 1$$

for both DIMACS and exact translation. On the other hands, the clauses C_i and C_{ijk} are translated differently. For the DIMACS translation scheme, each clause of the form

$$C_i = b_{\langle i,j_1\rangle} \vee \cdots \vee b_{\langle i,j_n\rangle}$$

is translated to inequality E_i^{DIMACS}

$$E_i^{DIMACS}\colon\ b_{\langle i,j_1\rangle} + \cdots + b_{\langle i,j_n\rangle} \geqslant 1.$$

For the exact transaction, each clause $C_i = b_{\langle i,j_1\rangle} \vee \cdots \vee b_{\langle i,j_n\rangle}$ together with clauses $C_{ijl} = \neg b_{\langle i,j\rangle} \vee \neg b_{\langle i,l\rangle}$, for each pair $i, j \in D_i$ is translated to equation E_i^{exact}

$$E_i^{exact}\colon\ b_{\langle i,j_1\rangle} + \cdots + b_{\langle i,j_n\rangle} = 1.$$

Again, we applied WSAT(OIP) to both versions of the problems. As opposed to WalkSAT and DLM, WSAT(OIP) consistently performed better on problems obtained using the exact translation.

In what follows we only report the results for the faster translation: DIMACS for WalkSAT and DLM, and exact for WSAT(OIP).

7.2. Hard graph-coloring problems

To compare the $\mathcal{LSDL}$ implementation of GENET versus the original GENET implementation and other methods, we investigate its performance on a set of hard graph-coloring problems. Since this set of benchmarks is well studied we give published results for local search solvers GENET, GSAT and DLM. The importance of execution parameter tuning for DLM and WalkSAT was highlighted to us by these results since we were unable to match the published results using the default parameter settings.

Table 1 shows the experimental results for GENET as described in [7] along with those for $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP). Table 2 shows the results for DLM, GSAT,

Table 1
GENET and $\mathcal{LSDL}$ on hard graph-coloring problems

Problem	GENET	$\mathcal{LSDL}$(GENET)	$\mathcal{LSDL}$(IMP)
	Median CPU time	Average (median) CPU time	Average (median) CPU time
Platform	SPARC classic	SPARCstation 10/40	SPARCstation 10/40
SPECInt92	26.4	50.2	50.2
g125.17	2.6 hrs	4.7 (3.7) mins	3.2 (2.6) mins
g125.18	23 s	4.5 (2.9) s	1.1 (0.925) s
g250.15	4.2 s	0.418 (0.408) s	0.328 (0.325) s
g250.29	1.1 hrs	14.6 (15.7) mins	11.3 (12.6) mins

Table 2
DLM, GSAT and WSAT(OIP) on hard graph-coloring problems

Problem	DLM	GSAT	WSAT(OIP)
	Average CPU time	Average CPU time	Average (median) CPU time
Platform	SGI Challenge	SPARCstation 10/51	SPARCstation 10/40
SPECInt92	$\geqslant$ 62.4	65.2	50.2
g125.17	23.2 mins	4.4 mins$^{(7/10)}$	57.8 (57.8) mins$^{(1/10)}$
g125.18	3.2 s	1.9 s	32.8 (30.5) s
g250.15	2.8 s	4.41 s	1.2 (1.2) hrs
g250.29	20.3 mins$^{(9/10)}$	20.3 mins$^{(9/10)}$	61.4 (61.4) hrs$^{(1/10)}$

WSAT(OIP) on the same set of hard graph-coloring problems encoded using the DIMACS translation. The figures for DLM and GSAT are published results [11] (the results for GSAT are also quoted from [11] as personal communications) while those for WSAT(OIP) are obtained using the WSAT(OIP) implementation running on the same hardware as $\mathcal{LSDL}$. We omit the timing for the lazy versions of $\mathcal{LSDL}$ since there is no arc inconsistency in the problems and the performance is similar to that of the non-lazy versions.

Since the published results are obtained from different hardware platforms, we specify the platforms as well as the platforms' SPECint92 [2] rating, which is a way of estimating a machine's computing power. The timing results of GENET represent the median of 10 runs collected on a SPARC Classic with SPECint92 rating of 26.4, which is about 2 to 3 times slower than a SPARCstation 10/40 with SPECint92 rating of 50.2. The results for GSAT and DLM are averages of 10 runs on a SPARCstation 10/51 with SPECint92 rating of 65.2

[2] SPECint92 is derived from the results of a set of integer benchmarks, and can be used to estimate a machine's single-tasking performance on integer code.

and a SGI Challenge (model unknown but the SPECint92 rating of the slowest model SGI Challenge R4400 is 62.4) respectively.

Clearly $\mathcal{LSDL}$(GENET) improves substantially on the original GENET implementation. $\mathcal{LSDL}$(IMP) gives the best timing results across all implementations (normalized by SpecInt92). This experiment also demonstrates the robustness of the $\mathcal{LSDL}$ instances, which always find a solution.

7.3. Permutation generation problems

The permutation generation problem is a combinatorial theory problem suggested by J.L. Lauriere. As described in [28], given a permutation p on the integers from 1 to n, we define the vector of monotonies m of size $n-1$ as

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

for all $1 \leqslant i \leqslant n-1$. We also define a vector of advances a of size $n-1$ as

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

for all $1 \leqslant i \leqslant n-1$. The aim is to construct a permutation of integers 1 to n satisfying conditions of monotonies and advances. The problem can be modeled as a CSP with n variables, $u_1, u_2, \ldots, u_n$, each has a domain $\{1, 2, \ldots, n\}$. The constraints

$$u_i \neq u_j$$

for all $i \neq j$ and $1 \leqslant i, j \leqslant n$ specified that the variables $u_1, u_2, \ldots, u_n$ form a permutation of n. The condition of monotonies m is represented by the constraints

$$u_{i+1} > u_i, \quad \text{if } m_i = 1,$$
$$u_{i+1} \leqslant u_i, \quad \text{if } m_i = 0,$$

for all $1 \leqslant i \leqslant n-1$. Similarly, the constraints

$$\forall\, 1 \leqslant j \leqslant i-1 \quad u_j \neq u_i + 1 \wedge u_i \neq n, \quad \text{if } a_i = 1,$$
$$\forall\, i+1 \leqslant j \leqslant n \quad u_j \neq u_i + 1, \quad \text{if } a_i = 0,$$

for all $1 \leqslant i \leqslant n-1$ denote the condition of advances a. These problems involve arc inconsistency.

We experiment with two sets of permutation generation problems. The first considers the special case of generating an increasing permutation. This problem is trivial for a complete search method with arc consistency, but difficult for local search solvers. In the second set of problems, the monotonies and advances are randomly generated, and much more difficult for complete solvers. Tables 3 and 4 show the results for the two sets of problems and give the CPU times for the alternate $\mathcal{LSDL}$ implementations as well as average number of domain values deleted by the lazy arc consistency versions.

Table 3
Increasing permutations generation problems

n	$\mathcal{LSDL}$(GENET)	Lazy-$\mathcal{LSDL}$(GENET)		$\mathcal{LSDL}$(IMP)	Lazy-$\mathcal{LSDL}$(IMP)	
	CPU time	Del	CPU time	CPU time	Del	CPU time
10	0.033 (0.033)	29.1	0.008 (0.008)	0.017 (0.017)	56.3	0.008 (0.008)
20	1.07 (1.08)	233	0.213 (0.208)	0.865 (0.867)	204	0.108 (0.117)
30	8.32 (7.80)	618	1.30 (1.33)	6.97 (6.75)	447	0.440 (0.375)
40	36.3 (35.6)	1220	5.44 (5.37)	26.8 (25.9)	783	1.93 (2.29)
50	107 (106)	1996	14.8 (14.9)	85.6 (86.1)	1291	3.84 (2.90)

Table 4
Random permutation generation problems

n	$\mathcal{LSDL}$(GENET)	Lazy-$\mathcal{LSDL}$(GENET)		$\mathcal{LSDL}$(IMP)	Lazy-$\mathcal{LSDL}$(IMP)	
	CPU time	Del	CPU time	CPU time	Del	CPU time
50	0.052 (0.050)	0.6	0.065 (0.067)	0.053 (0.050)	1.7	0.063 (0.058)
60	0.098 (0.092)	1.2	0.107 (0.100)	0.075 (0.067)	2.4	0.095 (0.092)
70	0.138 (0.117)	0.2	0.157 (0.150)	0.180 (0.167)	1.8	0.215 (0.208)
80	0.398 (0.383)	0.5	0.543 (0.483)	0.408 (0.392)	2.1	0.522 (0.508)
90	0.813 (0.800)	0.6	0.902 (0.842)	0.782 (0.733)	1.1	0.800 (0.808)
100	1.19 (1.22)	1.0	1.17 (1.17)	1.04 (1.01)	2.9	1.06 (1.05)

Clearly the addition of lazy arc consistency substantially improves $\mathcal{LSDL}$ when the problems involve a large amount of arc inconsistency (the first set of problems), for both $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP). By reducing the search space as computation proceeds we can reduce the computation time by an order of magnitude. Note that, since the more efficient $\mathcal{LSDL}$(IMP) searches less of the space, it prunes less values. This illustrates the targeted nature of lazy arc inconsistency, which works best when large amount of searching covering much search space is needed.

Problems in the second set are relatively easy for $\mathcal{LSDL}$, all implementations can solve the problems almost instantly. The fast convergence also implies that little search effort is performed and few values are pruned. Thus, not much is gained from the incorporation of the lazy arc consistency technique. In this case, the number of values pruned in both lazy implementations become too insignificant to be compared meaningfully. But note that the overhead of the lazy consistency method is low, even when it provides little or no advantage.

We give the results of WalkSAT, DLM and WSAT(OIP) on the encoded versions of the same problems in Tables 5 and 6 for comparison.

Table 5
DLM, WalkSAT, and WSAT(OIP) on increasing permutations generation problems

n	WalkSAT	DLM	WSAT(OIP)
10	0.275 (0.208)	0.085 (0.067)	0.300 (0.000)
20	21.9 (21.8)$^{(8/10)}$	3.17 (3.33)	8.00 (5.50)
30	$>$83.1$^{(0/10)}$	49.7 (41.4)	185 (131)
40	$>$115$^{(0/10)}$	$>$218$^{(0/10)}$	968 (856)$^{(9/10)}$
50	$>$144$^{(0/10)}$	$>$305$^{(0/10)}$	3276 (3276)$^{(2/10)}$

Table 6
DLM, WalkSAT, WSAT(OIP) on random permutation generation problems

n	WalkSAT	DLM	WSAT(OIP)
50	1.29 (1.21)	2.38 (2.36)	170 (169)
60	2.20 (2.08)	4.81 (4.67)	3004 (2969)
70	3.89 (3.81)	8.04 (7.87)	9239 (9291)
80	5.03 (5.04)	12.4 (12.2)	20131 (19707)
90	8.11 (7.73)	20.5 (19.3)	36860 (36515)
100	24.3 (24.3)	36.4 (33.3)	?

7.4. Random CSPs

Tables 7 and 8 show the results of the $\mathcal{LSDL}$ implementations for a set of tight random CSPs which involve arc inconsistency, ranging from 120 to 170 variables with domains of size 10 and tightness parameters $p_1 = 0.6$ and $p_2 = 0.75$. As pointed out by Achlioptas et al. [29] for random CSPs of this form there are likely to be many *flawed values* (their terminology for arc inconsistent values) which may be discovered by lazy arc consistency. As in our previous experiments $\mathcal{LSDL}$(IMP) consistently improves over $\mathcal{LSDL}$(GENET). The lazy versions are always substantially better than the non-lazy counterparts on these problems with significant arc inconsistency.

Table 9 shows results of the lazy versions on insoluble random CSPs. For these problems $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) (as well as most local search methods) always terminate unsuccessfully when the iteration limit is reached, since there is no solution. Lazy arc consistency allows the detection of the insolubility of the problem (when a variable domain becomes empty) and thus quickly terminates the search.

Again, we give the results of WalkSAT, DLM and WSAT(OIP) on the encoded versions of the same problems in Table 10 for comparison.

Table 7
$\mathcal{LSDL}$(GENET) on tight random CSPs

Problem	$\mathcal{LSDL}$(GENET)	Lazy-$\mathcal{LSDL}$(GENET)	
	CPU time	Pruned	CPU time
rcsp-120-10-60-75	5.93 (7.08)	1009.1	2.88 (2.90)
rcsp-130-10-60-75	9.14 (9.14)	1097.8	3.39 (3.40)
rcsp-140-10-60-75	9.69 (9.71)	1181.7	3.96 (3.95)
rcsp-150-10-60-75	12.6 (12.7)	1267.7	4.60 (4.61)
rcsp-160-10-60-75	14.2 (13.9)	1347.8	5.48 (5.51)
rcsp-170-10-60-75	21.8 (22.2)	1443.1	8.34 (8.37)

Table 8
$\mathcal{LSDL}$(IMP) on tight random CSPs

Problem	$\mathcal{LSDL}$(IMP)	Lazy-$\mathcal{LSDL}$(IMP)	
	CPU time	Pruned	CPU time
rcsp-120-10-60-75	5.95 (6.53)	406.1	1.31 (0.208)
rcsp-130-10-60-75	6.98 (7.25)	998.8	3.19 (3.53)
rcsp-140-10-60-75	8.20 (9.62)	1066.5	3.69 (4.06)
rcsp-150-10-60-75	10.2 (11.4)	1283.2	4.78 (4.78)
rcsp-160-10-60-75	9.57 (12.7)	1242.4	5.25 (5.74)
rcsp-170-10-60-75	20.1 (20.2)	1311.1	7.71 (8.45)

Table 9
Lazy-$\mathcal{LSDL}$ on random insoluble CSPs

Problem	Lazy-$\mathcal{LSDL}$(GENET)		Lazy-$\mathcal{LSDL}$(IMP)	
	Pruned	CPU time	Pruned	CPU time
rcsp-100-10-70-90	934.6	2.35 (2.34)	907.8	2.35 (2.34)
rcsp-110-10-70-90	1025.5	2.84 (2.84)	1000.6	2.86 (2.85)
rcsp-120-10-70-90	1116.4	3.39 (3.38)	1093.0	3.43 (3.43)

7.5. Phase transition random CSPs

A set of randomly generated binary CSPs close to the phase transition is used to further verify the efficiency and robustness of our $\mathcal{LSDL}$ instances. The phase transition random CSPs are generated as follows. According to Smith and Dyer [30], the expected number of solutions of a randomly generated binary CSP is given by

$$E(N) = m^n (1 - p_2)^{n(n-1)p_1/2},$$

Table 10
DLM, WalkSAT, WSAT(OIP) on tight random CSPs

Problem	WalkSAT	DLM	WSAT(OIP)
rcsp-120-10-60-75	5.62 (5.69)	69.6 (81.7)	15772 (16187)
rcsp-130-10-60-75	6.61 (6.44)	106 (150)	19295 (13730)
rcsp-140-10-60-75	6.41 (6.07)	493 (472)	23413 (15963)
rcsp-150-10-60-75	7.00 (6.18)	1118 (700)	42035 (33858)
rcsp-160-10-60-75	7.24 (6.37)	1832 (1163)$^{(7/10)}$	54275 (46533)
rcsp-170-10-60-75	10.4 (8.48)	1742 (6.92)$^{(3/10)}$	45638 (51148)

where n is the number of variables, m is the number of values in the domain of each variable, p_1 is the constraint density and p_2 is the constraint tightness. Following Smith and Dyer [30], we set $E(N)$ to 1 to compute a predictor, $\hat{p}_2$, of the crossover point. We get

$$\hat{p}_2 = 1 - m^{-2/((n-1)p_1)},$$

which is a good prediction of the constraint tightness giving a CSP in the phase transition region. By fixing m to 10 and p_1 to 0.6, we get the following values of $\hat{p}_2$ for binary CSPs with variables ranging from 120 to 170.

n	m	p_1	p_2
120	10	0.6	0.063
130	10	0.6	0.058
140	10	0.6	0.054
150	10	0.6	0.050
160	10	0.6	0.047
170	10	0.6	0.044

We then randomly generate binary CSPs based on the above parameters and filter out the insoluble ones. Since the problem size is large, it is not practical to perform an exhaustive search on these problems. We do the insoluble problems filtering using DLM. If DLM fails to find a solution within the execution limit, we generate another problem by reducing the value of p_2 by 0.001. This process continues until a soluble problem close to phase transition is obtained.

Tables 11 and 13 show the results of different $\mathcal{LSDL}$ instances, WalkSAT, DLM and WSAT(OIP) on the phase transition random CSPs. Each problem rcsp-n-m-p_1-p_2 in the table represents a binary CSP with n variables, a uniform domain size of m, a constraint density of p_1% and a constraint tightness of p_2%. In fact these problems were so hard, even for DLM, that very few runs found a solution, making it difficult to make any meaningful comparison.

To get slightly less difficult problems, we reduced p_2 by 0.001 from the first soluble problem found, and generated random problems until DLM detected satisfiability. The

Table 11
$\mathcal{LSDL}$(GENET) on phase transition CSPs

Problem	$\mathcal{LSDL}$(GENET)	Lazy-$\mathcal{LSDL}$(GENET)
rcsp-120-10-60-5.9	28.7 (28.7)$^{(1/10)}$	287 (287)$^{(2/10)}$
rcsp-130-10-60-5.5	$>$1454$^{(0/10)}$	$>$1665$^{(0/10)}$
rcsp-140-10-60-5.0	110 (110)$^{(1/10)}$	1106 (1106)$^{(1/10)}$
rcsp-150-10-60-4.7	$>$1676$^{(0/10)}$	$>$1893$^{(0/10)}$
rcsp-160-10-60-4.4	$>$1753$^{(0/10)}$	$>$1969$^{(0/10)}$
rcsp-170-10-60-4.1	164 (164)$^{(1/10)}$	$>$2119$^{(0/10)}$

Table 12
$\mathcal{LSDL}$(IMP) on phase transition CSPs

Problem	$\mathcal{LSDL}$(IMP)	Lazy-$\mathcal{LSDL}$(IMP)
rcsp-120-10-60-5.9	$>$1677.915$^{(0/10)}$	87.6 (87.6)$^{(1/10)}$
rcsp-130-10-60-5.5	$>$1971.850$^{(0/10)}$	205 (205)$^{(1/10)}$
rcsp-140-10-60-5.0	$>$1992.902$^{(0/10)}$	2082 (2082)$^{(1/10)}$
rcsp-150-10-60-4.7	$>$2305.597$^{(0/10)}$	$>$2570$^{(0/10)}$
rcsp-160-10-60-4.4	$>$2410.570$^{(0/10)}$	1842 (1842)$^{(1/10)}$
rcsp-170-10-60-4.1	$>$2549.323$^{(0/10)}$	882 (419)$^{(3/10)}$

Table 13
WalkSAT, DLM and WSAT(OIP) on phase transition CSPs

Problem	WalkSAT	DLM	WSAT(OIP)
rcsp-120-10-60-5.9	$>$953$^{(0/10)}$	243 (243)$^{(1/10)}$	5765 (5765)$^{(1/10)}$
rcsp-130-10-60-5.5	$>$980$^{(0/10)}$	$>$1064$^{(0/10)}$	7558 (7556)$^{(1/10)}$
rcsp-140-10-60-5.0	$>$978$^{(0/10)}$	561 (730)$^{(3/10)}$	$>$3995$^{(0/10)}$
rcsp-150-10-60-4.7	$>$1001$^{(0/10)}$	$>$1097$^{(0/10)}$	4259 (4259)$^{(1/10)}$
rcsp-160-10-60-4.4	$>$1012$^{(0/10)}$	$>$1108$^{(0/10)}$	$>$7647$^{(0/10)}$
rcsp-170-10-60-4.1	$>$1018$^{(0/10)}$	921 (921)$^{(1/10)}$	$>$7626$^{(0/10)}$

results are shown in Tables 14, 15 and 16. For this set of problems $\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) are approximately equally successful in finding solutions, while $\mathcal{LSDL}$(GENET) requires less execution time to achieve this success. Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(IMP) are worse than their non-lazy counterparts, since lazy arc consistency failed to detect any inconsistencies in all our executions. As we can confirm from the results of DLM, WalkSAT, and WSAT(OIP), this set of problems are difficult for local search solvers. The $\mathcal{LSDL}$ instances are comparable with the other state of the art

Table 14
$\mathcal{LSDL}$(GENET) on slightly easier phase transition CSPs

Problem	$\mathcal{LSDL}$(GENET)	Lazy-$\mathcal{LSDL}$(GENET)
rcsp-120-10-60-5.8	133 (117)$^{(4/10)}$	504 (504)$^{(2/10)}$
rcsp-130-10-60-5.4	>1381$^{(0/10)}$	>1569$^{(0/10)}$
rcsp-140-10-60-4.9	115 (50.4)$^{(8/10)}$	313 (208)$^{(5/10)}$
rcsp-150-10-60-4.6	168 (179)$^{(4/10)}$	317 (364)$^{(7/10)}$
rcsp-160-10-60-4.3	471 (370)$^{(6/10)}$	718 (701)$^{(3/10)}$
rcsp-170-10-60-4.0	137 (98.4)$^{(10/10)}$	158 (96.4)$^{(8/10)}$

Table 15
$\mathcal{LSDL}$(IMP) on slightly easier phase transition CSPs

Problem	$\mathcal{LSDL}$(IMP)	Lazy-$\mathcal{LSDL}$(IMP)
rcsp-120-10-60-5.8	387 (387)$^{(1/10)}$	327 (39.9)$^{(3/10)}$
rcsp-130-10-60-5.4	>1896$^{(0/10)}$	>2117$^{(0/10)}$
rcsp-140-10-60-4.9	194 (48.9)$^{(5/10)}$	149 (114)$^{(7/10)}$
rcsp-150-10-60-4.6	321 (327)$^{(7/10)}$	386 (145)$^{(6/10)}$
rcsp-160-10-60-4.3	266 (266)$^{(2/10)}$	811 (811)$^{(2/10)}$
rcsp-170-10-60-4.0	400 (308)$^{(6/10)}$	467 (255)$^{(10/10)}$

Table 16
WalkSAT, DLM and WSAT(OIP) on slightly easier phase transition CSPs

Problem	WalkSAT	DLM	WSAT(OIP)
rcsp-120-10-60-5.8	>934$^{(0/10)}$	431 (331)$^{(6/10)}$	6333 (6333)$^{(1/10)}$
rcsp-130-10-60-5.4	>964$^{(0/10)}$	>1045$^{(0/10)}$	>7405$^{(0/10)}$
rcsp-140-10-60-4.9	>963$^{(0/10)}$	283 (277)$^{(10/10)}$	2595 (2619)$^{(4/10)}$
rcsp-150-10-60-4.6	>980$^{(0/10)}$	567 (782)$^{(5/10)}$	3314 (3314)$^{(2/10)}$
rcsp-160-10-60-4.3	>991$^{(0/10)}$	389 (349)$^{(4/10)}$	1457 (1457)$^{(2/10)}$
rcsp-170-10-60-4.0	>994$^{(0/10)}$	235 (231)$^{(10/10)}$	2201 (1980)$^{(8/10)}$

solvers. DLM is better able to find solutions, which is not surprising given it was used to filter the problems in the first place.

7.6. Quasigroup completion problems

The quasigroup completion problem [31] is a recently proposed CSP that combines features of both random problems and highly structured problems. A *quasigroup* is an ordered pair $(Q, \cdot)$, where Q is a set and $(\cdot)$ is a binary operation on Q such that the

equations $a \cdot x = b$ and $y \cdot a = b$ are uniquely solvable for every pair of elements a, b in Q. The constraints on a quasigroup are such that its multiplication table forms a Latin square. This means that in each row and each column of the table, each element of the set Q occurs exactly once. The *order* N of the quasigroup is the cardinality of the set Q. An *incomplete* or *partial Latin square* P is a partially filled $N \times N$ table such that no symbol occurs twice in a row or a column. The *quasigroup completion problem* (QCP) is the problem of determining whether the remaining entries of a partial Latin square P can be filled in such a way that we can obtain a complete Latin square. The pre-assigned values can be seen as a perturbation to the structure of the original problem of finding an arbitrary Latin square.

A natural formulation of a QCP as a CSP is to model each cell in the $N \times N$ multiplication table as a variable, each of which has the same domain Q. Pre-assigned cells have the domains of their corresponding variables fixed to the pre-assigned values. We use disequality constraints ($\neq$) to disallow repetition of values in the same row or column. We experiment with both Latin square problems (or QCPs with no pre-assigned cells) and difficult QCPs at phase transitions.

Tables 17 and 18 show the results of solving Latin square problems of sizes ranging from $N = 10$ to $N = 35$ in steps of 5.

$\mathcal{LSDL}$(GENET) and $\mathcal{LSDL}$(IMP) solve the problems with little difficulty. Again, results of the Lazy-$\mathcal{LSDL}$ implementations are not shown since there is no arc inconsistencies in the problems. The results for WalkSAT, DLM, and WSAT(OIP) are given for comparison.

Table 17
$\mathcal{LSDL}$ on Latin square problems

Problem	$\mathcal{LSDL}$(GENET)	$\mathcal{LSDL}$(IMP)
magic-10	0.008 (0.008)	0.008 (0.008)
magic-15	0.073 (0.067)	0.035 (0.033)
magic-20	0.195 (0.183)	0.098 (0.100)
magic-25	0.418 (0.392)	0.257 (0.250)
magic-30	1.94 (1.84)	1.32 (1.29)
magic-35	6.01 (5.48)	3.82 (3.93)

Table 18
WalkSAT, DLM and WSAT(OIP) on Latin square problems

Problem	WalkSAT	DLM	WSAT(OIP)
magic-10	0.395 (0.325)	0.125 (0.133)	0.600 (1.00)
magic-15	$66.7\ (65.7)^{(2/10)}$	0.985 (0.942)	4.00 (4.00)
magic-20	$>211^{(0/10)}$	6.26 (6.37)	201 (202)
magic-25	$>295^{(0/10)}$	29.4 (29.3)	11218 (11234)
magic-30	$>396^{(0/10)}$	103 (103)	40581 (40698)
magic-35	$>2100^{(0/10)}$	?	?

Table 19
$\mathcal{LSDL}$(GENET) on quasigroup completion problems

Problem	$\mathcal{LSDL}$(GENET)	Lazy-$\mathcal{LSDL}$(GENET)	
	CPU time	Pruned	CPU time
qcp-15	1.34 (1.18)	1108.4	0.608 (0.592)
qcp-16	1.23 (1.32)	1394.6	0.948 (0.958)
qcp-17	1.80 (1.91)	1722.8	1.50 (1.54)
qcp-18	2.29 (2.16)	2024.8	1.98 (2.10)
qcp-19	4.12 (3.95)	2503.5	2.96 (3.03)
qcp-20	5.28 (5.62)	2912.9	3.55 (3.53)

Table 20
$\mathcal{LSDL}$(IMP) on quasigroup completion problems

Problem	$\mathcal{LSDL}$(IMP)	Lazy-$\mathcal{LSDL}$(IMP)	
	CPU time	Pruned	CPU time
qcp-15	0.472 (0.483)	926.2	0.415 (0.425)
qcp-16	0.462 (0.433)	1070.7	0.583 (0.558)
qcp-17	0.862 (0.858)	1284.9	0.733 (0.842)
qcp-18	0.848 (0.708)	1206.7	0.688 (0.658)
qcp-19	2.12 (1.68)	1627.4	1.14 (0.975)
qcp-20	1.84 (2.05)	1774.3	1.22 (1.48)

Gomes and Selman [31] identifies phase transition phenomenon for QCPs with costs peak occurring roughly around 42% of pre-assignment for different values of N. A completely random pre-assignment generates problems that are either trivially soluble or trivially insoluble. We randomly choose a variable until a given percentage of variables is selected. For each selected variable, we randomly select a value from its domain. Similar to Meseguer and Walsh [32], if the selected value is incompatible with previous assignments or would wipe out the domain of some other variables using constraint propagation, we select the another random value from its domain. This process continue until a compatible assignment is obtained. Tables 19, 20 and 21 give respectively the results of $\mathcal{LSDL}$ and others in solving QCPs of orders ranging from 15 to 20 with 42% of pre-assignment. This class of problems is harder than their counterparts without pre-assignment but it is still relatively easy for all $\mathcal{LSDL}$ instances. Since pre-assignment induces arc inconsistency, we include also the results of Lazy-$\mathcal{LSDL}$ implementations which again improved the results. Again, the results for WalkSAT, DLM, and WSAT(OIP) are provided for comparison.

We note that this class of problems can be more efficiently solved by systematic search methods enforcing generalized arc consistency on the `alldifferent` global constraint [33,34]. The purpose of our experiment is two-fold. First, we show that $\mathcal{LSDL}$ instances and local search methods in general are capable of solving this class of problems

Table 21
DLM, WalkSAT, WSAT(OIP) on quasigroup completion problems

Problem	WalkSAT	DLM	WSAT(OIP)
qcp-15	$>82.5^{(0/10)}$	1.16 (1.08)	$212\ (244)^{(6/10)}$
qcp-16	$33.5\ (31.4)^{(4/10)}$	0.885 (0.758)	133 (129)
qcp-17	$>96.9^{(0/10)}$	1.66 (1.85)	$235\ (263)^{(6/10)}$
qcp-18	$>103^{(0/10)}$	1.97 (2.03)	282 (270)
qcp-19	$>110^{(0/10)}$	2.31 (2.27)	$283\ (303)^{(9/10)}$
qcp-20	$>116^{(0/10)}$	3.21 (3.17)	363 (347)

encoded using disequality constraints ($\neq$). Second, we use the problems to observe and demonstrate the scaling behaviour and robustness of our algorithms.

8. Related work

In recent years, many local search methods have been developed for solving CSPs and SAT. In the following, we briefly review some of these methods that are related to our research.

8.1. DLM

DLM [11] is a new discrete Lagrange-multiplier-based global-search method for solving SAT problems, which are first transformed into a discrete constrained optimization problem. The new method encompasses new heuristics for applying the Lagrangian methods to traverse in discrete space. Experiments confirm that the discrete Lagrange multiplier method generally outperforms the best existing methods.

The $\mathcal{LSDL}$ algorithm is closely related to DLM. Although both DLM and $\mathcal{LSDL}$ apply discrete Lagrange multiplier methods, there are substantial 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. 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. In addition, 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 CSPs, the set of constraints, which restrict valid assignments for CSPs, is incorporated in the discrete gradient. Thus, only valid assignments are searched in $\mathcal{LSDL}$. On the contrary, DLM lacks this kind of restriction. Any possible assignments, including

those which are invalid for CSPs, are considered. As a result, the efficiency of DLM is affected.

Third, the two algorithms use different gradient descent procedures to perform saddle point search. In DLM, the gradient descent procedure considers all Boolean variables of the SAT problem as a whole and modifies one Boolean variable in each update. However, in $\mathcal{LSDL}$, all integer variables can be updated at the same time. In addition, the gradient descent procedure 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 gradient descent procedure always modifies the integer variables such that there is a maximum decrease in the Lagrangian function.

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

8.2. GSAT

GSAT [2] 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 [35], 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 CSPs, such as the N-queens problems and graph-coloring problems [2]. Various extensions to the basic GSAT algorithm include mixing GSAT with a random walk strategy [25, 26], clause weight learning [25,36], averaging in previous assignments [25] and tabu-like move restrictions [37]. These modifications are shown to boost the performance of GSAT on certain kinds of problems. Latter enhanced implementations of GSAT are known as WalkSAT.

8.3. WSAT

Although local search algorithms have been successful in solving certain hard SAT problems, many combinatorial problems do not have concise propositional encoding and hence an efficient SAT problem solver, such as GSAT, cannot be applied. On the other hand, many of these problems, such as scheduling, sequencing and time-tabling, can be modeled by linear pseudo-Boolean constraints, which are linear inequalities with Boolean variables. Walser [38] extended WalkSAT, a successor of GSAT, for handling this kind of pseudo-Boolean constraint systems. Similar to WalkSAT, the resultant algorithm, called WSAT($\mathcal{PB}$), performs local search on linear pseudo-Boolean constraints. It continues to flip Boolean variables according to a randomized greedy strategy until a satisfying assignment is found or a predefined execution limit is reached. However, unlike the SAT problems, flipping a single Boolean variable is not guaranteed to satisfy a pseudo-Boolean constraint. Therefore, a score is defined for each assignment to measure its distance from the solution. In each move, WSAT($\mathcal{PB}$) tries to flip the variable which decreases the score

most. In addition, a history mechanism is implemented to avoid randomness. When there is a tie in variable selection, this history mechanism is activated to resolve it. WSAT($\mathcal{PB}$) is also equipped with a tabu memory to avoid flipping the same variable in the near future.

Various problems, such as the radar surveillance problem and the progressive party, are used to evaluate the performance of WSAT($\mathcal{PB}$). Experiments show that WSAT($\mathcal{PB}$) is more efficient than existing techniques for these domains. Furthermore, handling pseudo-Boolean constraints does not incur much overhead over the propositional case.

Walser et al. [27] also generalize WSAT($\mathcal{PB}$) from handling Boolean variables to finite domain integer variables. They introduce WSAT(OIP) for solving over-constrained integer problems. Experiments on the capacitated production planning show that WSAT(OIP) gives better performance than existing commercial mixed integer programming branch-and-bound solver.

8.4. Simulated annealing

Simulated annealing [39] 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 $\triangle E \leqslant 0$, 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. [22] investigated the feasibility of applying simulated annealing for solving graph-coloring problems. Selman and Kautz [40] 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.

9. Concluding remarks

The contribution of this paper is three-fold. First, based on the theoretical work of Wah and Shang [11], we define $\mathcal{LSDL}$, a discrete Lagrangian search scheme for CSPs. Second, we establish a surprising connection between constraint satisfaction and optimization by showing that the GENET convergence procedure, a representative repair-based local search method, is an instance of $\mathcal{LSDL}$, denoted $\mathcal{LSDL}$(GENET). Third, using the dual viewpoint of the GENET as a Lagrangian method and a heuristic repair method we construct variant of $\mathcal{LSDL}$(GENET). We empirically study these variants and show improvements of up to

75% and an average improvement of 36% over $\mathcal{LSDL}$(GENET). By adding the lazy arc consistency method to $\mathcal{LSDL}$ we can achieve additional improvements of almost *an order of magnitude* for cases with arc inconsistency, without incurring much overhead for cases without arc inconsistency. While demonstrating competitive performance with other local search solvers, the $\mathcal{LSDL}$ instances are shown to be robust across the benchmarks that we test.

Local search has always been considered just a heuristic. Results in this paper give the mathematics of local search and represent a significant step forward to the understanding of heuristic repair algorithms. The gained insight allows us to design more efficient variants of the algorithms. We conclude the paper with a few interesting directions for future research. First, on the theoretical side, at least one question remains unanswered: *under what condition(s) do the algorithms always terminate, if at all?* The importance of the question should not be underestimated although in our experience GENET has always terminated for solvable CSPs. Second, the five parameters of $\mathcal{LSDL}$ suggest ample possibilities to experiment with new and better algorithms. It is also interesting to investigate if there are other possible parameters for $\mathcal{LSDL}$. Third, it is worthwhile to investigate if $\mathcal{LSDL}$ can be extended straight-forwardly for efficient non-binary constraint-solving along the line of research of E-GENET [20,41]. Non-binary constraints are needed for modeling complex real-life applications. Although any non-binary CSP can be transformed to a binary CSP in theory, the resulting CSP is usually too large to be effectively and efficiently solved in practice. Indeed we have already obtained encouraging preliminary results in extending $\mathcal{LSDL}$ for solving non-binary CSPs [21]. Fourth, we can investigate the extension of $\mathcal{LSDL}$ to include other modifications of the GENET approach including lazy constraint consistency [10] and improved asynchronous variable orderings [42].

Acknowledgements

The work described in this paper was substantially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project No. CUHK 4302/98E). We are grateful to the fruitful discussions and comments from Benjamin Wah and Yi Shang. We would like to thank the anonymous reviewers for their comments which undoubtedly helped improve the paper. Finally, we also thank Yousef Kilani for helping with some last minute benchmarking needs.

References

[1] A.K. Mackworth, Consistency in networks of relations, AI J. 8 (1) (1977) 99–118.
[2] B. Selman, H. Levesque, D.G. Mitchell, A new method for solving hard satisfiability problems, in: Proc. AAAI-92, San Jose, CA, AAAI Press/MIT Press, 1992, pp. 440–446.
[3] F. Glover, Tabu search part I, Operations Research Society of America (ORSA) J. Comput. 1 (3) (1989) 109–206.
[4] F. Glover, Tabu search part II, Operations Research Society of America (ORSA) J. Comput. 2 (1) (1989) 4–32.
[5] H.M. Adorf, M.D. Johnston, A discrete stochastic neural network algorithm for constraint satisfaction problems, in: Proc. International Joint Conference on Neural Networks, San Diego, CA, 1990.

[6] S. Minton, M.D. Johnston, A.B. Philips, P. Laird, Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems, Artificial Intelligence 58 (1992) 161–205.
[7] A. Davenport, E. Tsang, C. Wang, K. Zhu, GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 325–330.
[8] P. Morris, The breakout method for escaping from local minima, in: Proc. AAAI-93, Washington, DC, 1993, pp. 40–45.
[9] D.M. Simmons, Nonlinear Programming for Operations Research, Prentice-Hall, Englewood Cliffs, NJ, 1975.
[10] P.J. Stuckey, V. Tam, Extending E-GENET with lazy constraint consistency, in: Proc. 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-97), 1997, pp. 248–257.
[11] Y. Shang, B.W. Wah, A discrete Lagrangian-based global-search method for solving satisfiability problems, J. Global Optimization 12 (1) (1998) 61–100.
[12] 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.
[13] K. Choi, J. Lee, P. Stuckey, A Lagrangian reconstruction of a class of local search methods, in: Proc. 10th IEEE International Conference on Tools with Artificial Intelligence, Taipei, Taiwan, ROC, 1998, pp. 166–175.
[14] B.W. Wah, Y.J. Chang, Trace-based methods for solving nonlinear global optimization and satisfiability problems, J. Global Optimization 10 (2) (1997) 107–141.
[15] Y.J. Chang, B.W. Wah, Lagrangian techniques for solving a class of zero-one integer linear problems, in: Proc. International Conference on Computer Software and Applications, IEEE, 1995, pp. 156–161.
[16] B.W. Wah, Y. Shang, A discrete Lagrangian-based global-search method for solving satisfiability problems, in: D.-Z. Du, J. Gu, P. Pardalos (Eds.), Proc. DIMACS Workshop on Satisfiability Problem: Theory and Applications, American Mathematical Society, 1997, pp. 365–392.
[17] Y. Shang, Global search methods for solving nonlinear optimization problems, PhD Thesis, Department of Computer Science, University of Illinois, 1997.
[18] J. Platt, A. Barr, Constrained differential optimization, in: Proc. Neural Information Processing System Conference, 1987.
[19] B.W. Wah, T. Wang, Y. Shang, Z. Wu, Improving the performance of weighted Lagrange-multiplier methods for nonlinear constrained optimization, in: Proc. 9th International Conference on Tools with Artificial Intelligence, IEEE, 1997, pp. 224–231.
[20] J.H.M. Lee, H.F. Leung, H.W. Won, Towards a more efficient stochastic constraint solver, in: Proc. 2nd International Conference on Principles and Practice of Constraint Programming, Cambridge, MA, Lecture Notes in Computer Science, Vol. 1118, Springer, Berlin, 1996, pp. 338–352.
[21] K.M.F. Choi, A Lagrangian reconstruction of a class of local search methods, Master's Thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1998.
[22] D. Johnson, C. Aragon, L. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; Part 2 graph coloring and number partitioning, Oper. Res. 39 (3) (1991) 378–406.
[23] P.J. Stuckey, V. Tam, Extending GENET with lazy arc consistency, Technical Report, Department of Computer Science, University of Melbourne, 1996.
[24] P.J. Stuckey, V. Tam, Extending GENET with lazy arc consistency, IEEE Trans. Systems Man Cybern. 28 (5) (1998) 698–703.
[25] B. Selman, H. Kautz, Domain-independent extensions to GSAT: Solving large structured satisfiability problems, in: Proc. IJCAI-93, Chambery, France, 1993, pp. 290–295.
[26] B. Selman, H.A. Kautz, B. Cohen, Noise strategies for improving local search, in: Proc. AAAI-94, Seattle, WA, AAAI Press/MIT Press, 1994, pp. 337–343.
[27] J.P. Walser, R. Iyer, N. Venkatasubramanyan, An integer local search method with application to capacitated production planning, in: Proc. AAAI-98, Madison, WI, AAAI Press/MIT Press, 1998, pp. 373–379.
[28] P.V. Hentenryck, Constraint Satisfaction in Logic Programming, MIT Press Cambridge, MA, 1989.
[29] D. Achliptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, C. Stamatiou, Random constraint satisfaction: A more accurate picture, in: Proc. 3rd International Conference on Principles and Practice of Constraint Programming, Springer, Berlin, 1997, pp. 107–120.
[30] B.M. Smith, M.E. Dyer, Locating the phase transition in binary constraint satisfaction problems, Artificial Intelligence 81 (1996) 155–181.

[31] C. Gomes, B. Selman, Problem structure in the presence of perturbations, in: Proc. AAAI-97, Providence, RI, 1997, pp. 221–226.
[32] P. Meseguer, T. Walsh, Interleaved and discrepancy based search, in: Proc. ECAI-98, 1998.
[33] P. Shaw, K. Stergiou, T. Walsh, Arc consistency and quasigroup completion, in: Proc. ECAI-98 Workshop on Non-Binary Constraints, 1998.
[34] K. Stergiou, T. Walsh, The difference all-difference makes, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 414–419.
[35] S. Minton, M.D. Johnston, A.B. Philips, P. Laird, Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method, in: Proc. AAAI-90, Boston, MA, AAAI Press/MIT Press, 1990, pp. 17–24.
[36] J. Frank, P. Cheeseman, J. Allen, Weighting for godat: Learning heuristics for GSAT, in: Proc. AAAI-96, Portland, OR, AAAI Press/MIT Press, 1996, pp. 338–343.
[37] I.P. Gent, T. Walsh, Towards an understanding of hill-climbing procedures, in: Proc. AAAI-93, Washington, DC, AAAI Press/MIT Press, 1993, pp. 28–33.
[38] J.P. Walser, Solving linear pseudo-boolean constraint problems with local search, in: Proc. AAAI-97, Providence, RI, AAAI Press/MIT Press, 1997, pp. 269–274.
[39] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[40] B. Selman, H. Kautz, An empirical study of greedy local search for satisfiability testing, in: Proc. AAAI-93, Washington, DC, 1993, pp. 46–51.
[41] J.H.M. Lee, H.F. Leung, H.W. Won, Extending GENET for non-binary CSP's, in: Proc. 7th IEEE International Conference on Tools with Artificial Intelligence, IEEE Computer Society Press, Washington DC, 1995, pp. 338–343.
[42] P. Stuckey, V. Tam, Improving GENET and EGENET by new variable ordering strategies, in: H. Selvaraj, B. Verna (Eds.), Proc. International Conference on Computational Intelligence and Multimedia Applications, 1998, pp. 107–112.