A Lagrangian Reconstruction of a Class of Local Search Methods

K.M.F. Choi J.H.M. Lee

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong, China
{mfchoi,jlee}@cse.cuhk.edu.hk

P.J. Stuckey

Department of Computer Science
University of Melbourne
Parkville 3052, Australia
pjs@cs.mu.oz.au

Abstract

Heuristic repair algorithms, a class of local search methods, demonstrate impressive efficiency in solving some large-scale and hard instances of constraint satisfaction problems (CSP's). In this paper, we draw a surprising connection between heuristic repair techniques and the discrete Lagrange multiplier methods by transforming CSP's into zero-one constrained optimization problems. A Lagrangian-based search scheme $\mathcal{LSDL}$ is proposed. We show how GENET, a representative heuristic repair algorithm, can be reconstructed from $\mathcal{LSDL}$. The dual viewpoint of GENET as heuristic repair method and 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 other GENET implementations reported in the literature, competing favourably with other state-of-the-art methods on a set of hard graph colouring problems. Second, our best variant, which combines techniques from heuristic repair and Lagrangian methods, is always more efficient than the reconstructed GENET, and can better it by an order of magnitude.

1 Introduction

A *constraint satisfaction problem* (CSP) [Mac77] 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. A *solution* is an assignment of values from the domains to their respective variables so that all constraints are satisfied simultaneously. CSP's 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 are based on local search, for example GSAT and Tabu Search. 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 the GDS network [AJ90], a class of local search techniques, known as heuristic repair methods and exemplified by the work reported in [MJPL92] and [DTWZ94], has been shown to be effective in solving some large-scale and some computationally hard classes of CSP's. As with other local search algorithms, heuristic repair methods can be trapped in a local minimum, a non-solution state in which no further improvement can be made. To help escape from the local minimum, Minton *et al.* proposed random restart, while Davenport *et al.* and Morris [Mor93] 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 CSP's. In this paper, we show that GENET [DTWZ94], a representative member of heuristic repair, is equivalent to a form of the Lagrange multiplier method [Sim75], a well-known technique for solving constrained optimization problems with a wealth of literature on its formal properties. This result is useful not just in establishing a formal characterization of heuristic repair methods. It also allows us to gain important insights into the various design issues of heuristic repair methods. In addition, because of the dual viewpoint of GENET as a heuristic repair method and a 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 Lagrangian search algorithms. We reconstruct GENET as an instantiation of the $\mathcal{LSDL}$ scheme and explore variations that arise from considering it as a Lagrange multiplier method. We also show how the lazy consistency optimization [ST97] developed for GENET (considered as a heuristic repair method) can be transferred to $\mathcal{LSDL}$. Thus we gain benefits from both viewpoints. Benchmarking results confirm that first, our reconstructed GENET has the same fast convergence behavior as other GENET implementations reported in the literature. Second, our best variant, which combines techniques from heuristic repair and Lagrangian methods, is always more efficient than the reconstructed GENET, and can better it by an order of magnitude.

The paper is organized as follow. The GENET network is briefly introduced in the next section, followed by a description of the discrete Lagrangian formulation of CSP's. The $\mathcal{LSDL}$ search scheme is a result of the Lagrangian formulation. We show formally that GENET is an instance of $\mathcal{LSDL}$. Experimental results of the reconstructed GENET and a number of variants are presented before related work. Finally we conclude.

2 A Brief Overview of GENET

The GENET [DTWZ94] 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 nonbinary constraints, we limit our attention to only the binary subset of GENET.

2.1 Network Architecture

Consider a CSP (U, D, C). A GENET *network* consists of a set of nodes and connections. Each variable $i \in U$ is represented in GENET by a cluster of *nodes* $\langle i, d\rangle$, one for each value $d \in D_i$. Each node $\langle i, d\rangle$ is associated with an *output* $V_{\langle i,d\rangle}$, which is 1 if value d is assigned to variable i, and 0 otherwise. A 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 nodes in clusters i_1 and i_2 respectively. Two nodes $\langle i_1, d_1\rangle$ and $\langle i_2, d_2\rangle$ are connected if $i_1 = d_1 \wedge i_2 = d_2$ violates c. Each connection has a *weight*, initially set to -1. The *input* $I_{\langle i,d\rangle}$ to a node $\langle i, d\rangle$ is:

$$I_{\langle i,d\rangle} = \sum_{\forall \langle k,e\rangle \text{ connected to } \langle i,d\rangle} W_{\langle i,d\rangle\langle k,e\rangle} V_{\langle k,e\rangle} \quad (1)$$

where $W_{\langle i,d\rangle\langle k,e\rangle}$ is the weight of the connection between $\langle i, d\rangle$ and $\langle k, e\rangle$. A *state* of network $\mathcal{N}$ is a pair $(\vec{V}, \vec{W})$, where $\vec{V}$ is a vector of outputs $(\ldots, V_{\langle i,d\rangle}, \ldots)$ for all nodes $\langle i, d\rangle$ in $\mathcal{N}$ and $\vec{W}$ is a vector of weights $(\ldots, W_{\langle i,d\rangle\langle k,e\rangle}, \ldots)$ for every pair of connected nodes $\langle i, d\rangle$ and $\langle k, e\rangle$. A state is *valid* if exactly one node in each cluster is on. A *solution state* of a network has the input of all on nodes being zero.

2.2 Convergence Procedure

The convergence procedure, outlined below, defines how a GENET network changes states and connection weights before it reaches a solution state.

```
initialize the network to a random valid state
loop
   % State update rule
   for each cluster in parallel do
      each node calculates its input
      select the node with max input to be on next
   end for
   if all nodes' output remain unchanged then
      if the input to all on nodes is zero then
         terminate and return the solution
      else
         % Heuristic learning rule
         update all connection weights by
         W^{s+1}_<i,d><k,e> = W^s_<i,d><k,e> - V^s_<i,d> V^s_<k,e>
      end if
   end if
end loop
```

Initially, a node in each cluster is selected on randomly; other nodes are off. GENET performs iterative repair by minimizing the number of constraint violation in 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 nodes have zero input. 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 the same time. In *asynchronous* update, each cluster performs input calculation and state update independently. Synchronous update can cause oscillations [DTWZ94], 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 node with the maximum input during a state update. To select

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

the next node to be on, GENET adopts the following heuristic rule. Let P be the set of nodes with maximum input. If the node currently on is in P, it remains on. Otherwise, `rand`(P) is selected to be on, where `rand` returns a random element of P. The state update rule is a direct application of the min-conflict heuristic [MJPL92].

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

$$E(\mathcal{N}, S) = \sum_{\forall(\langle i,d\rangle,\langle k,e\rangle)\in\mathcal{N}} V_{\langle i,d\rangle} W_{\langle i,d\rangle\langle k,e\rangle} V_{\langle k,e\rangle}. \quad (2)$$

$E(\mathcal{N}, S)$ is always non-positive with negative weights. The energy $E(\mathcal{N}, S_0)$ of a solution state S_0 is always 0, a global maximum value for $E(\mathcal{N}, S)$. The convergence procedure thus carries out an optimization process for the energy function $E(\mathcal{N}, S)$.

Fourth, an *iteration* constitutes one pass over the outermost loop. $W^s_{\langle i,d\rangle\langle k,e\rangle}$ denotes the weight of connection between nodes $\langle i,d\rangle$ and $\langle k,e\rangle$ and $V^s_{\langle i,d\rangle}$ denotes the output of node $\langle i,d\rangle$ in the sth iteration. Weight update aims to decrease the energy associated with the local maximum. Thus learning has the effect of filling in the local maximum in the search surface. This learning rule is similar to the breakout method [Mor93].

The CSP, where $U = \{X, Y, Z\}$, $D_X = D_Y = D_Z = \{1, 2, 3\}$ and $C = X < Y \wedge even(Y + Z)$, gives a GENET network as illustrated in Figure 1(a). There are inhibitory connections between any two nodes which violate one of the constraints. For example, there is a connection between $\langle Y, 1\rangle$ and $\langle Z, 2\rangle$ since this combination of variable assignment violates $even(Y + Z)$; hence $W_{\langle Y,1\rangle\langle Z,2\rangle} = -1$ initially. The state illustrated has the nodes $\langle X, 3\rangle$, $\langle Y, 2\rangle$, and $\langle Z, 1\rangle$ on, representing the assignment $X = 3, Y = 2, Z = 1$, and has energy -2. Updating the Y cluster of nodes proceeds by calculating $I_{\langle Y,1\rangle} = -1$, $I_{\langle Y,2\rangle} = -2$ and $I_{\langle Y,3\rangle} = -1$ so that one of $\langle Y, 1\rangle$ or $\langle Y, 3\rangle$ should be selected randomly to be on next. Suppose $\langle Y, 1\rangle$ is selected and the resulting state is shown in Figure 1(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 $W_{\langle X,3\rangle\langle Y,1\rangle}$ 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.

3 A Lagrangian Formulation of CSP's

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 two-step transformation for converting any CSP into a zero-one integer optimization problem. The first step of the transformation is inspired by the GENET network architecture and results in a Boolean satisfiability (SAT) representation of the CSP. To solve the resulting zero-one optimization problem, we use a discrete version of the Lagrange multiplier method [SW98].

3.1 CSP as Boolean Satisfiability

A GENET network of a CSP (U, D, C) can be viewed as a graphical representation of a SAT problem. We associate every node $\langle i,d\rangle$ with a Boolean variable $b_{\langle i,d\rangle}$, which is `true` if $\langle i,d\rangle$ is on and `false` otherwise. There are two types of Boolean formulae in this SAT problem: cluster formulae and connection formulae. The *cluster formula* for each variable $i \in U$:

$$C_i = \bigvee_{d\in D_i} (b_{\langle i,d\rangle} \wedge (\bigwedge_{e\in D_i, e\neq d} \bar{b}_{\langle i,e\rangle})) \quad (3)$$

enforces that exactly one node in cluster i is on. Cluster formulae enforce the valid states of the GENET network. Each connection $(\langle i,d\rangle, \langle k,e\rangle)$ has a corresponding *connection formula*

$$C_{\langle i,d\rangle\langle k,e\rangle} = \bar{b}_{\langle i,d\rangle} \vee \bar{b}_{\langle k,e\rangle} \quad (4)$$

which states that the nodes $\langle i,d\rangle$ and $\langle k,e\rangle$ cannot be both on simultaneously. Thus solving the CSP (U, D, C) is equivalent to finding a truth assignment to satisfy the following formula:

$$\bigwedge_{i\in U} C_i \quad \wedge \bigwedge_{\text{all connections}} C_{\langle i,d\rangle\langle k,e\rangle}. \quad (5)$$

3.2 CSP as Zero-One Integer Optimization

A *zero-one integer minimization* (0–1) problem consists of a set of zero-one variables $\vec{z}$, an objective function $N(\vec{z})$ 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 $N(\vec{z}^*)$ is minimized and each member $c_i(\vec{z}^*)$ of G is satisfied. In the following, we present the second step of our transformation that converts the SAT problem from last step into a 0–1 problem.

Each Boolean variable $b_{\langle i,d\rangle}$ is converted into a zero-one variable $z_{\langle i,d\rangle}$ for representing value d of variable i. The value of the variable $z_{\langle i,d\rangle}$ is 1 when $b_{\langle i,d\rangle}$ is `true`, and 0 otherwise. Each cluster formulae C_i for all $i \in U$ is transformed to the following equation:

$$\sum_{d\in D_i} z_{\langle i,d\rangle} = 1. \quad (6)$$

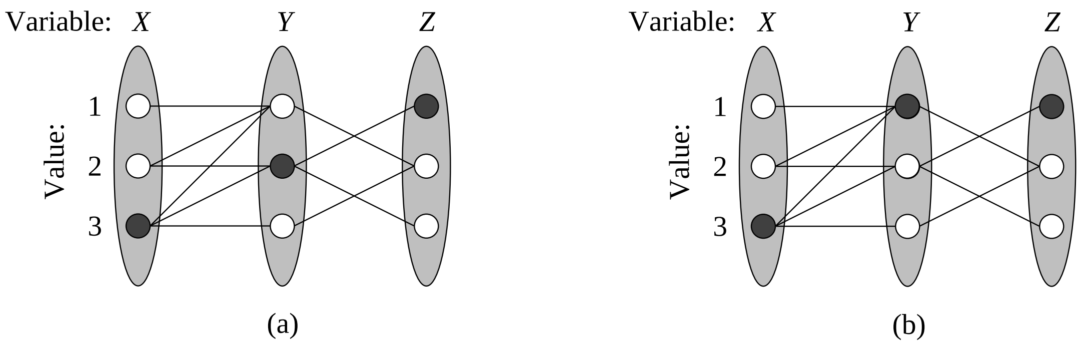

Figure 1: Example GENET network

Note that these equations define a space for proper instantiation of $\vec{z}$, which corresponds to the valid states of the GENET network.

For each connection formula $C_{\langle i,d\rangle\langle k,e\rangle}$, we define an *incompatibility function*

$$g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z}) = z_{\langle i,d\rangle} z_{\langle k,e\rangle}. \tag{7}$$

The function $g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z})$ returns 1 if both $z_{\langle i,d\rangle}$ and $z_{\langle k,e\rangle}$ are 1, and 0 otherwise. Forcing $g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z}) = 0$ has the same effect of forbidding two connected nodes $\langle i,d\rangle$ and $\langle k,e\rangle$ in the GENET network to be on at the same time. The incompatibility functions are used as indicators of constraint violations.

The resulting 0–1 problem has the following form:

$$\min \ N(\vec{z}) \tag{8}$$

subject to

$$\sum_{d \in D_i} z_{\langle i,d\rangle} = 1, \quad \text{for each variable } i \in U \tag{9}$$

$$g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z}) = 0, \quad \text{for each } (\langle i,d\rangle, \langle k,e\rangle) \in B \tag{10}$$

where $\vec{z} = (\ldots, z_{\langle i,d\rangle}, \ldots)$ is a vector of zero-one variables and B is the set of all incompatible node pairs $(\langle i,d\rangle, \langle k,e\rangle)$. To complete the 0–1 formulation of CSP, we need to define an appropriate objective function $N(\vec{z})$ so that *every solution of the CSP must correspond to a global minimum of* $N(\vec{z})$. We call this the *correspondence requirement*.

There are at least two objective functions that fulfill the correspondence requirement. The goal of solving a CSP is to find an assignment satisfying all constraints. A possible objective function, adapted from Wah and Chang [WC97], is to count the total number of constraint violations. Mathematically,

$$N(\vec{z}) = \sum_{(\langle i,d\rangle, \langle k,e\rangle) \in B} g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z}) \tag{11}$$

where $\vec{z} = (\ldots, z_{\langle i,d\rangle}, \ldots)$ is a vector of zero-one variables. The constrained minimum of the objective function defined in (11) is always zero, which corresponds to zero constraint violation. Another possibility is the trivial objective function:

$$N(\vec{z}) = 0. \tag{12}$$

The key to correspondence is that all feasible points are also global minima of the objective function. The objective function serves only to exert additional force to guide solution searching. The trivial objective function may not seem useful but it is related to the GENET convergence procedure.

3.3 Discrete Lagrange Multiplier Method

The Lagrange multiplier method is a well-known technique for solving constrained optimization problems [Sim75]. It provides a systematic approach for handling constraints, while maintaining numerical stability and solution accuracy. Since the method is applicable to only real variable problems, it cannot be applied directly to our 0–1 problem formulation. In our first attempt, we convert the resulting 0–1 problems into real variable constrained optimization problems by introducing additional constraints to restrict the real variables to hold either 0 or 1 [CW95]. Although this approach is viable, handling of the additional constraints incurs costly computation.

An alternative approach is to extend the classical Lagrange multiplier method to deal with discrete problems [SW98]. Consider the 0–1 problem (8 – 10). Similar to the real variable case, the *Lagrangian function* $L(\vec{z}, \vec{\lambda})$ is constructed as follows:

$$L(\vec{z}, \vec{\lambda}) = N(\vec{z}) + \sum_{(\langle i,d\rangle, \langle k,e\rangle) \in B} \lambda_{\langle i,d\rangle\langle k,e\rangle} g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z}) \tag{13}$$

where $\vec{z} = (\ldots, z_{\langle i,d\rangle}, \ldots)$ is a vector of zero-one variables and $\vec{\lambda} = (\ldots, \lambda_{\langle i,d\rangle\langle k,e\rangle}, \ldots)$ is a vector of *Lagrange multipliers*. Although the optimization problem is discrete, the Lagrange multipliers $\vec{\lambda}$ can take real values. Note that the constraints defined by (9) serve only to define a valid search state and are not included in the Lagrangian function. The constraints will be incorporated in the search procedure discussed below.

A global minimum of the 0–1 problem (8 – 10) can be obtained by finding a saddle point of the Lagrangian function. As in the continuous case, a *saddle point* $(\vec{z}^*, \vec{\lambda}^*)$ of the Lagrangian function $L(\vec{z}, \vec{\lambda})$ is defined by the condition

$$L(\vec{z}^*, \vec{\lambda}) \leq L(\vec{z}^*, \vec{\lambda}^*) \leq L(\vec{z}, \vec{\lambda}^*) \qquad (14)$$

for all $(\vec{z}^*, \vec{\lambda})$ and $(\vec{z}, \vec{\lambda}^*)$ sufficiently close to $(\vec{z}^*, \vec{\lambda}^*)$ [SW98]. 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 minimum of a 0–1 problem and a saddle point of its associated Lagrangian function is established using the following discrete saddle point theorem, restated as follows.

Theorem 1 (Discrete Saddle Point Theorem) [Wu98] *$\vec{z}^*$ is a constrained minimum of the optimization problem*

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

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

Since the incompatibility functions $g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z})$, for all $\langle i,d\rangle\langle k,e\rangle \in B$are always non-negative, the discrete saddle point theorem is applicable to the 0–1 problem (8 – 10).

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 $\vec{\lambda}$ [PB87]. Instead of using differential equations, the discrete Lagrange multiplier method uses difference equations [SW98]:

$$\begin{aligned} \vec{z}^{s+1} &= \vec{z}^s - \triangle_{\vec{z}} L(\vec{z}^s, \vec{\lambda}^s) && (15) \\ \vec{\lambda}^{s+1} &= \vec{\lambda}^s + \vec{g}(\vec{z}^s) && (16) \end{aligned}$$

where x^s denotes the value of x in the sth iteration, $\triangle_{\vec{z}}$ is a *discrete gradient operator* and $\vec{g}(\vec{z}) = (\ldots, g_{\langle i,d\rangle\langle k,e\rangle}(\vec{z}), \ldots)$ is a vector of incompatibility functions. In general, the discrete gradient operator is not unique. Any operator that can perform descent in the $\vec{z}$-space can be used. We give one such operator as follows.

Let m_i be the size of the domain of variable i. Given a vector $\vec{z}$, the *projection operator* π_i

$$\pi_i(\vec{z}) = (z_{\langle i,d_1\rangle}, \ldots, z_{\langle i,d_j\rangle}, \ldots, z_{\langle i,d_{m_i}\rangle}) \qquad (17)$$

gives the *ith-component* of $\vec{z}$, where $\vec{z} = (\ldots, z_{\langle i,d\rangle}, \ldots)$ is a vector of integer variables. Hence, π_i returns the vector of integer variables corresponding to variable i. The *ith partial discrete gradient operator* $\partial_i L(\vec{z}, \vec{\lambda})$ is defined as:

$$\partial_i L(\vec{z}, \vec{\lambda}) = \pi_i(\vec{z}) - \pi_i(\vec{z}') \qquad (18)$$

if the following conditions hold:

- X is a set of zero-one variable vectors such that

$$\begin{aligned} &\forall \vec{x} \in X : (\textstyle\sum_{d_k \in D_i} x_{\langle i,d_k\rangle} = 1) \wedge \\ &\qquad \forall s \neq i\ \forall d \in D_s\ (x_{\langle s,d\rangle} = z_{\langle s,d\rangle}) \wedge \\ &\forall \vec{y}\ \big([\textstyle\sum_{d_k \in D_i} y_{\langle i,d_k\rangle} = 1 \wedge \\ &\qquad \forall s \neq i\ \forall d \in D_s (y_{\langle s,d\rangle} = z_{\langle s,d\rangle})] \\ &\qquad\qquad \Rightarrow L(\vec{x}, \vec{\lambda}) \leq L(\vec{y}, \vec{\lambda})\big) \end{aligned}$$

- $\vec{z}'$ is selected from X using the following rule:
  - $\vec{z}'$ is $\vec{z}$ if $\vec{z} \in X$,
  - $\vec{z}' = \texttt{rand}(X)$ otherwise, where `rand` is a random selection function for a set.

The ith partial discrete gradient operator $\partial_i L(\vec{z}, \vec{\lambda})$ returns a *differential vector* $\vec{d}$ for the ith-component of $\vec{z}$ by (1) computing a set X of vectors $\vec{x}$ that causes the most reduction in the value of the Lagrangian and (2) selecting a vector $\vec{z}'$ from X. Note that the selection process in (2) is based on that of the GENET state update rule. The differential vector is defined in such a way that $\pi_i(\vec{z}) - \vec{d} = \pi_i(\vec{z}')$. Therefore, if $\vec{d} = \vec{0}$, there is no change in the ith-component of the vector $\vec{z}$. We are now ready to define a discrete gradient operator:

$$\pi_i(\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})) = \partial_i L(\vec{z}, \vec{\lambda}), \quad \forall i \in U. \qquad (19)$$

When $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$, either a solution is found or a local minimum is reached. Note that the constraints defined in (9) are incorporated in the definition of the partial discrete gradient operators enforcing the sum of all $z_{\langle i,d\rangle}$ for variable i to be 1.

[$I_{\vec{z}}$ **]** initialize the values of $\vec{z}$
[$I_{\vec{\lambda}}$] initialize the values of $\vec{\lambda}$
while [N] $L(\vec{z}, \vec{\lambda}) - N(\vec{z}) > 0$ (soln not found) **do**
  [$\triangle_{\vec{z}}$] update $\vec{z}$: $\vec{z} \leftarrow \vec{z} - \triangle_{\vec{z}} L(\vec{z}, \vec{\lambda})$
  if [$U_{\vec{\lambda}}$] condition for updating $\vec{\lambda}$ holds **then**
    update $\vec{\lambda}$: $\vec{\lambda} \leftarrow \vec{\lambda} + \vec{g}(\vec{z})$
  end if
end while

Figure 2: The $\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$ algorithm

The Lagrange multipliers $\vec{\lambda}$ are updated according to the incompatibility functions. If a constraint 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.

Figure 2 gives $\mathcal{LSDL}(N, \triangle_{\vec{z}}, I_{\vec{z}}, I_{\vec{\lambda}}, U_{\vec{\lambda}})$, a generic Lagrangian search algorithm for the solution of 0–1 problems transformed from CSP's. The algorithm performs local search using the discrete Lagrangian approach and has five degrees of freedom: (N) the objective function, ($\triangle_{\vec{z}}$) the discrete gradient operator, ($I_{\vec{z}}$) how the variables $\vec{z}$ are initialized, ($I_{\vec{\lambda}}$) how the Lagrange multipliers $\vec{\lambda}$ are initialized, and ($U_{\vec{\lambda}}$) when to update $\vec{\lambda}$. Where appropriate, we annotate the algorithm with the parameters of the algorithm in brackets to show where the parameters take effect.

4 GENET Reinvented

Our Lagrangian approach to constraint satisfaction, consisting of a two-step transformation and the $\mathcal{LSDL}$ (pronounced as "Lisdal") algorithm, is devised from *first principles*. While we discuss, say, some of the example objective functions and discrete gradient operators, the $\mathcal{LSDL}$ algorithm is independent of these examples. Therefore, $\mathcal{LSDL}$ defines a general scheme for a class of algorithms based on Lagrangian search. On the other hand, the GENET model is designed around the min-conflict heuristic [MJPL92] and a breakout method similar to that described in [Mor93]. In this section, we show how we can reconstruct GENET using our Lagrangian approach.

Given a CSP (U, D, C), we can check easily that the two-step transformation establishes a one-one correspondence between the GENET network of (U, D, C) and the associated 0-1 problem of (U, D, C). Next, we show that the GENET convergence procedure can be obtained by instantiating $\mathcal{LSDL}$ with proper parameters. We use $\mathcal{LSDL}$(GENET) to denote the instance of $\mathcal{LSDL}$ with the following parameters:

- N: the objective function is that of (12),
- $\triangle_{\vec{z}}$: defined by the following pseudocode
  for each variable $i \in U$ **do**
    $\pi_i(\vec{z}) \leftarrow \pi_i(\vec{z}) - \partial_i L(\vec{z}, \vec{\lambda})$ (20)
  end for
- $I_{\vec{z}}$: the values of $\vec{z}$ are initialized randomly, provided that the initial values correspond to a valid state in GENET,
- $I_{\vec{\lambda}}$: the values of $\vec{\lambda}$ are all initialized to 1, and
- $U_{\vec{\lambda}}$: $\vec{\lambda}$ are updated when a local minimum is reached.

For lack of space, we state without proof the following theorem.

Theorem 2 *Consider a CSP (U, D, C), and its corresponding GENET network and 0-1 problem. Suppose both GENET and $\mathcal{LSDL}$(GENET) use the same random selection function* `rand` *and they share the same initial state. The following hold:*

- *For all iteration s, $\vec{V}^s = \vec{z}^s$ and $\vec{W}^s = -\vec{\lambda}^s$.*
- *GENET terminates if and only if $\mathcal{LSDL}$(GENET) terminates.*
- *If both GENET and $\mathcal{LSDL}$(GENET) terminate, they return the same solution for the CSP.*

Similar results can be proven if, in $\mathcal{LSDL}$, we use instead the objective function N defined in (11) and initialize all $\vec{\lambda}$ to 0.

5 Variants

The $\mathcal{LSDL}$ viewpoint of GENET as a discrete Lagrange multiplier method explicitly makes clear many different parameters, such as the objective function, the initial assignment, the gradient operator, the strategy for updating the Lagrange multipliers $\vec{\lambda}$, which may be varied. Hence we have an opportunity to improve on GENET by exploring the space of parameters available in the $\mathcal{LSDL}$ framework. Alternatively, variants to GENET viewed as a heuristic repair method can be added to the $\mathcal{LSDL}$ framework. Hence we can directly benefit from the dual viewpoint of the GENET. We begin by examining the parameters of $\mathcal{LSDL}$, and then briefly describe lazy arc consistency [ST97], an improving technique for heuristic repair local search methods.

5.1 Objective Function

As mentioned, the objective function N is not unique. Any function related to the CSP at hand and satisfied the correspondence requirement can be used as an objective function. A good objective function, however, can help direct the search towards the solution region more efficiently. In the previous sections, two possible definitions are proposed. First, a zero function $N(\vec{z}) = 0$ is used. This is equivalent to the original GENET. Second, the objective function (11) is constructed based on the total number of constraint violations.

5.2 Initial Assignments

Similar to the classical Lagrange multiplier method, the initial assignment of the vector $\vec{z}$ can be freely generated. The simplest approach is to initialize $\vec{z}$ randomly. However, Minton *et al.* [MJPL92] suggested that a greedily generated initial assignment can boost the performance of the search.

The initial values of the Lagrange multipliers $\vec{\lambda}$ can be assigned arbitrarily. In general, any non-negative numbers can be used. The Lagrange multipliers can be initialized to 1; or following [LLW96], different values can be assigned to different Lagrange multipliers to reflect the importance of the constraints.

5.3 Gradient Operator

Any discrete gradient operator that can perform descent in the integer variable space of $\vec{z}$ can be used. The operator defined in (19) performs gradient descent in all variables in the CSP at once. The situation is similar to that of synchronous cluster update in the GENET convergence procedure. In practice, this can also lead to oscillation. The one defined in (20) corresponds more to the cluster update used in sequential implementation of GENET.

Another possible definition is to consider each original variable i independently and change the variables $\pi_i(\vec{z})$ which causes the greatest reduction in the Lagrangian function. This approach is similar to the one defined in [SW98] for solving SAT problems.

5.4 Updating Lagrange Multipliers

Unlike the continuous case, the updating frequency of the Lagrange multipliers $\vec{\lambda}$ can affect the performance of the search [SW98]. In the continuous formulation, $\vec{\lambda}$ are modified in each iteration. However, in the discrete case, two possible strategies can be used. We can either update $\vec{\lambda}$ in each iteration, or update $\vec{\lambda}$ when a local minimum is reached (i.e. when $\triangle_{\vec{z}} L(\vec{z}, \vec{\lambda}) = \vec{0}$). Note that the first strategy is just a direct adoption of the continuous case while the second strategy corresponds to Morris's breakout method [Mor93].

5.5 Lazy Arc Consistency

Arc consistency [Mac77] is an approach to simplifying a CSP to an equivalent CSP which has smaller domains for some variables. A value $e \in D_k$ for variable k is *arc inconsistent* with binary constraint c involving variables i and k if $\forall d \in D_i \; i = d \wedge k = e$ violates c. In GENET this means that for each $d \in D_i$ the node $\langle i, d\rangle$ is connected to $\langle k, e\rangle$. If $e \in D_k$ is arc inconsistent for any constraint c then it can be removed from the domain of k without removing any possible solutions. The resulting CSP is equivalent (has the same set of solutions). There are a number of arc consistency algorithms which remove all arc inconsistencies from a CSP. This improves search since the search space becomes smaller.

Lazy arc consistency [ST97] arises from noticing similarities in the calculation of arc inconsistency and (sequential) calculation of GENET inputs. Since in any GENET state $V_{\langle k,e\rangle}$ is on for at most one e, we can define v_k as the value such that $V_{\langle k,v_k\rangle}$ is on. We can modify the (sequential) input calculation for a variable i to detect some arc inconsistencies as follows:

for $k \in U, k \neq i$ **do**
  $inconsistent(k) := true$
end for
for $d \in D_i$ **do**
  $I_{\langle i,d\rangle} := 0$
  for $k \in U, k \neq i$ **do**
    if $\langle i, d\rangle$ and $\langle k, v_k\rangle$ are connected **then**
      $I_{\langle i,d\rangle} \mathrel{+}= W_{\langle i,d\rangle\langle k,v_k\rangle}$
    else
      $inconsistent(k) := false$
    end if
  end for
end for
for $k \in U, k \neq i$ **do**
  if $inconsistent(k)$ **then**
    remove v_k from D_k
  end if
end for

For example, consider the calculation of the inputs for Y for the network in Figure 1(a). Since each of $\langle Y, 1\rangle$, $\langle Y, 2\rangle$ and $\langle Y, 3\rangle$ are connected to $\langle X, 3\rangle$, we determine that it is inconsistent and should be removed from the domain of X.

The overhead of lazy arc consistency to input calculation is small, only a single statement is added to the inner loop. Lazy arc consistency improves upon full arc consistency since it is targeted at only the domain values that are selected during the search, which may be substantially less than the full search space. Lazy arc consistency has been shown [ST97] to improve GENET substantially for CSPs which are not arc consistent and does not degrade performance substantially for problems which are already arc consistent. Lazy arc consistency also allows the possibility of detecting that a CSP has no solution, which cannot be managed by GENET (and most other heuristic repair algorithms). We can incorporate lazy arc consistency in the $\mathcal{LSDL}$ framework similarly (within the calculation of the set X in the evaluation of $\partial_i L(\vec{z}, \vec{\lambda})$).

6 Experiments

In order to confirm the correctness and feasibility of our approach, we construct $\mathcal{LSDL}$(GENET) and a number of its variants. The $\mathcal{LSDL}$(GENET) implementation has two purposes. First, the $\mathcal{LSDL}$(GENET) implementation 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. Our experiments consist of two sets of problems: a set of hard graph-coloring problems from the DIMACS archive [JAMS91] and a set of randomly generated tight CSP's with arc inconsistencies.

We have conducted a set of experiments to explore the parameter space of the $\mathcal{LSDL}$ algorithm so as to identify a good combination of parameters. In the following, we report the performance of $\mathcal{LSDL}$(GENET) and our best $\mathcal{LSDL}$ variant obtained so far, denoted by $\mathcal{LSDL}$(MAX) for short, with the following parameters: (N) the one defined in (11), ($\triangle_{\vec{z}}$) the one defined in (20), ($I_{\vec{z}}$) using the greedy algorithm described in [MJPL92], ($I_{\vec{\lambda}}$) all initialized to 1, and ($U_{\vec{\lambda}}$) updated every iteration. We compare these versions against versions with the addition of lazy arc consistency, Lazy-$\mathcal{LSDL}$(GENET) and Lazy-$\mathcal{LSDL}$(MAX). Benchmark results of all $\mathcal{LSDL}$ implementations are taken on a SUN SPARCstation 10/40. In each cell, the unbracketed and the bracketed timing results represent the CPU time for the average and median of 10 runs respectively. Unless otherwise specified, all timing results are in seconds. 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 file).

6.1 Hard Graph Coloring Problem

To compare the $\mathcal{LSDL}$ implementation of GENET versus other GENET implementations and other methods we investigate its performance on a set of hard graph-coloring problems. Table 1 shows the experimental results for GENET as described in [DTWZ94], PROCLANN [LT95] an incremental formulation of GENET, DLM [SW98], and GSAT [SLM92]. We omit the lazy versions since there is no arc inconsistency in the problems and the performance is similar to that of the non-lazy versions.

The timing results of GENET represent the median of 10 runs collected on a SUN Sparc Classic, which is about 2 to 3 times slower than a SPARCstation 10/40. The results of PROCLANN represent the average of 10 runs collected on a SUN SPARCstation 10/30. The results for GSAT and DLM are averages of 10 runs on a SUN SPARCstation 10/51 and a SGI Challenge (model unknown) respectively.

Clearly $\mathcal{LSDL}$(GENET) improves substantially on GENET and PROCLANN. $\mathcal{LSDL}$(GENET) is also comparable to, if not in general better than, DLM and GSAT, which are the state of the art in solving the graph coloring benchmarks. $\mathcal{LSDL}$(MAX) gives the best timing results across all implementations. Note that GSAT and DLM solve a simpler version of the problem since the SAT versions of the graph-coloring problems in the DIMACS archive lack the cluster formula [Sel97]. Therefore, answers to these easier problems can have a vertex assigned with more than one color. Note also that the $\mathcal{LSDL}$ implementations produce solutions successfully on every run. Given a predefined maximum number of iterations (such as five million) allowed, DLM has only a 9/10 success ratio on the g250.29 problem and GSAT has a 7/10 and 9/10 success ratio on the g125.17 and g250.29 problems respectively.

6.2 Random CSPs

Table 2 shows the results of the $\mathcal{LSDL}$ implementations for a set of tight random CSP's which involve arc inconsistency, ranging from 120 to 170 variables with domains of size 10 and tightness parameters $p_1 = 0.6$ and $p_2 = 0.75$.

The table shows, besides the timing results, also the average number of iterations and the average number of values pruned. Again, the $\mathcal{LSDL}$(MAX) implementations are better than those of $\mathcal{LSDL}$(GENET) both in terms of timing and number of iterations. Clearly the addition of lazy arc consistency substantially improves $\mathcal{LSDL}$ when the problem involves a substantial amount of arc inconsistency. By reducing the search space as computation proceeds we can reduce the computation time by (almost) an order of magnitude. Note that, since the more efficient $\mathcal{LSDL}$(MAX) searches less of the space, it generally 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.

7 Related Work

DLM [SW98] is a new discrete Lagrange-multiplier-based global-search method for solving SAT problems,

Problem	GENET Median CPU Time	PROCLANN Average CPU Time	DLM Average CPU Time	GSAT Average CPU Time	$\mathcal{LSDL}$(GENET) Average (Median) CPU Time	$\mathcal{LSDL}$(MAX) Average (Median) CPU Time
g125.17	2.6 hrs	2.3 hrs	23.2 mins	4.4 mins	4.7 mins (3.7 mins)	3.2 mins (2.6 mins)
g125.18	23 s	2.5 mins	3.2 s	1.9 s	4.5 s (2.9 s)	1.1 s (0.925 s)
g250.15	4.2 s	1.1 hrs	2.8 s	4.41 s	0.418 s (0.408 s)	0.328 s (0.325 s)
g250.29	1.1 hrs	4.6 hrs	20.3 mins	20.33 mins	14.6 mins (15.7 mins)	11.3 mins (12.6 mins)

Table 1: Results on hard graph-coloring problems

Problem	$\mathcal{LSDL}$(GENET)		Lazy-$\mathcal{LSDL}$(GENET)			$\mathcal{LSDL}$(MAX)		Lazy-$\mathcal{LSDL}$(MAX)		
	Iter.	CPU Time	Iter.	Pruned	CPU Time	Iter.	CPU Time	Iter.	Pruned	CPU Time
rcsp-120	115.7	5.9 (7.1)	10.9	1018.5	2.6 (2.6)	27.8	6.0 (6.5)	10.1	923.5	2.4 (2.7)
rcsp-130	162.0	9.1 (9.1)	11.0	1099.8	3.1 (3.1)	27.4	7.0 (7.3)	8.8	884.2	2.5 (3.1)
rcsp-140	147.3	9.7 (9.7)	11.0	1183.0	3.6 (3.6)	27.1	8.2 (9.6)	8.0	831.7	2.7 (3.8)
rcsp-150	173.4	12.6 (12.7)	8.6	918.2	3.2 (4.1)	28.5	10.2 (11.4)	6.6	768.7	2.7 (4.3)
rcsp-160	167.4	14.2 (13.9)	11.0	1355.3	4.9 (4.9)	23.0	9.6 (12.7)	8.8	1101.1	4.1 (5.0)
rcsp-170	176.2	21.8 (22.2)	11.0	1433.5	7.4 (7.3)	31.9	20.1 (20.2)	11.0	1456.9	7.4 (7.4)

Table 2: Results on tight random CSP's (times in seconds)

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. Both DLM and $\mathcal{LSDL}$ handle SAT problems but $\mathcal{LSDL}$ specializes in SAT problems of a particular form, those converted from a CSP using our two-step transformation. For example, the constraints defined in (9) are embedded in our discrete gradient operator instead of being part of the Lagrangian function, resulting in reduced number of constraints. Additionally, zero-one variables are updated as independent individuals in DLM. In $\mathcal{LSDL}$, variables corresponding to the same cluster are updated as a group since searching in invalid GENET states is not fruitful.

Some might question the need of $\mathcal{LSDL}$ since SAT problems are CSP's and all CSP's can be transformed into SAT problems. DLM (and/or GSAT) may seem to suffice. DLM handles general SAT problems and $\mathcal{LSDL}$ can be considered a specialization of DLM. Our work is important since many practical CSP's have non-$\{0,1\}$ domains and not all SAT problems correspond to CSP's with non-$\{0,1\}$ domains. An important evidence supporting the effectiveness and efficiency of the $\mathcal{LSDL}$ framework is that DLM has to use different tuning parameters for different problem instances, while no special tuning is made to any of the $\mathcal{LSDL}$ implementations. In addition, the DLM variants used in tackling the graph-coloring problems employ, on top of the Lagrangian approach, an additional tabu list to remember states visited and a number of tuning heuristics. For example, the Lagrange multipliers are reset by a factor of 2/3 after every 10000 iterations. The Lagrange multipliers are incremented by, instead of 1, different constants for different graph-coloring problems. The $\mathcal{LSDL}$ results are obtained without additional machinery.

8 Concluding Remarks

The contribution of this paper is four-fold. First, we give a two-step transformation for converting any CSP into a zero-one constrained optimization problem. Second, a Lagrangian search scheme $\mathcal{LSDL}$ is proposed based on characteristics of the transformed 0–1 problem. Instantiating $\mathcal{LSDL}$ with different parameters results in specific Lagrangian search algorithm with different efficiency and behavior. Third, 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 $\mathcal{LSDL}$. Fourth, we implement $\mathcal{LSDL}$(GENET), a Lagrangian reconstruction of GENET. $\mathcal{LSDL}$(GENET) exhibits the same good convergence results as two other GENET implementations. We experiment with variants of $\mathcal{LSDL}$(GENET) obtained by considering it from a dual viewpoint: as a Lagrange multiplier method and as a heuristic repair method. By varying parameters of $\mathcal{LSDL}$ we can improvements of up to 300% 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.

Acknowledgements

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (RGC Ref. No. CUHK4302/98E) and Australian Research Council grant A49702580.

References

[AJ90] H.M. Adorf and M.D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA, 1990.

[CW95] Y. J. Chang and B. W. Wah. Lagrangian techniques for solving a class of zero-one integer linear problems. In *Proceeding of International Conference on Computer Software and Applications, IEEE*, pages 156–161, 1995.

[DTWZ94] A. Davenport, E. Tsang, C. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of the 12th National Conference on Artificial Intelligence (Seattle, WA)*, pages 325–330, 1994.

[JAMS91] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part 2 graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[LLW96] J. H. M. Lee, H. F. Leung, and H. W. Won. Towards a more efficient stochastic constraint solver. In *Second International Conference on Principles and Practice of Constraint Programming*, pages 338–352, Cambridge, Massachusetts, USA, August, 1996. Springer-Verlag, LNCS 1118.

[LT95] J. H. M. Lee and V. W. L. Tam. A framework for integrating artificial neural networks and logic programming. *International Journal of Artificial Intelligence Tools*, 4(1&2):3–32, 1995.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[MJPL92] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[Mor93] P. Morris. The breakout method for escaping from local minima. In *Proceeding of the 11th National Conference on Artificial Intelligence (Washington, DC)*, pages 40–45, 1993.

[PB87] J. Platt and A. Barr. Constrained differential optimization. In *Proceedings of Neural Information Processing System Conference*, 1987.

[Sel97] B. Selman. Private Communication, July 1997.

[Sim75] D. M Simmons. *Nonlinear Programming for Operations Research.* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[SLM92] B. Selman, H.J. Levesque, and D.G. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of AAAI-92*, pages 440–446, San Jose, CA, 1992.

[ST97] P.J. Stuckey and V. Tam. Extending EGENET with lazy constraint consistency. In *Proceedings of the Ninth IEEE International Conference on Tools with Artificial Intelligence*, pages 248 – 257, 1997.

[SW98] Y. Shang and B. W. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.

[WC97] B. W. Wah and Y. J. Chang. Trace-based methods for solving nonlinear global optimization and satisfiability problems. *Journal of Global Optimization*, 10(2):107–141, 1997.

[Wu98] 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.