A General Approach for Constraint Solving by Local Search

Philippe Galinier[1] and Jin-Kao Hao[2]

1) Ecole Polytechnique de Montréal, C.P. 6079, Montréal, (Québec) H3C 3A7

2) LERIA, Université d'Angers, 2 bd Lavoisier F-49045 Angers Cedex 01

{philippe.galinier@polymtl.ca, Jin-Kao.Hao@univ-angers.fr}

Abstract

In this paper, we present a general approach for solving constraint problems by local search. The proposed approach is based on a set of high-level constraint primitives motivated by constraint programming systems. These constraints constitute the basic bricks to formulate a given combinatorial problem. A tabu search engine ensures the resolution of the problem such formulated. Experimental results are shown to validate the proposed approach.

Keywords: Constraint solving, combinatorial optimization, tabu search

1 Introduction

An instance of the Constraint Satisfaction Problem (CSP) [19, 23] is defined by a triplet $(X, \mathcal{D}, \mathcal{C})$ where:

- $X = \{x_1, \cdots, x_n\}$ is a finite set of n variables.
- $\mathcal{D} = \{D_{x_1}, \cdots, D_{x_n}\}$ is a set of associated domains. Each domain D_{x_i} specifies the finite set of possible values of the variable x_i.
- $\mathcal{C} = \{C_1, \cdots, C_p\}$ is a finite set of p constraint. Each constraint is defined on a set of variables and specifies which combinations of values are compatible for these variables.

Given such a triplet, the problem consists in finding a complete assignment of the values to the variables that satisfies all the constraints. Such an assignment is then said consistent. Since the set of all assignments (not necessarily consistent) is defined by the Cartesian product $D_{x_1} \times \cdots \times D_{x_n}$ of the domains, solving a CSP means to determine a particular assignment among a potentially huge search space. The CSP is known to be

a NP-hard problem in the general case. Related to the CSP is the MAX-CSP problem where one seeks an assignment such that a maximum number of constraints is satisfied.

The CSP is a very general formalism able to model a large number of combinatorial search problems. The CSP can be used to formulate conveniently many well-known problems such as graph k-coloring, satisfiability (SAT) as well as many practical applications related to resource assignments, planning or timetabling. One classical approach for solving a CSP is the systematic tree search strategy combined with various domain reduction techniques [23]. This approach is largely used by constraint programming (CP) systems. In practice, complete methods based on systematic tree search may fail to solve large CSP instances, because the computing time required may become prohibitive.

Another powerful and popular strategy for solving large CSPs is the repair or Local Search approach [1]. With local search[1], an initial configuration (conflicting assignment) is first built. Then one iterates a series of moves, each move consisting in modifying the value of a variable. The goal is then to minimize gradually constraint violation until a solution is eventually found. Although the principle of local search is very simple, it has proved itself to be very effective for dealing with many hard combinatorial problems. Indeed, this approach has been used to solve several well-know problems with elementary constraints such as k-colouring, SAT, frequency assignment problems and random instances of binary CSPs. Local search has also found solutions for combinatorial search problems involving much more complex constraints like the progressive party problem.

However, local search is often applied on a case-by-case basis, leading to the situation of one problem one algorithm. Algorithms developed in such a way are rarely reusable across several different, even similar problems. In this paper, we propose a general local search approach which may be used to solve various CSPs (and MAX-CSPs). At a very high level, the proposed system can be divided into two large parts: a general formalism for problem modeling and a general search engine for problem resolution. Problem modeling is achieved by using a set of high-level constraint primitives, like in constraint programming. Problem solving is ensured by an embedded Tabu Search engine. Now solving a particular constraint problem consists simply in modeling the problem with constraint primitives. The embedded Tabu Search is then directly used to search for a solution.

The remaining of the paper is organized as follows. Section 2 reviews the related works. Section 3 presents our general approach for constraint solving by local search. Section 4 shows examples of modeling combinatorial problems with the proposed approach. Section 5 presents experimental results on selected problems. The last section gives some conclusions.

[1]Notice that traditionally the term "local search" is the synonym of the descent or the iterative improvement. In this paper, we use the term to include the wider class of neighbourhood search methods such as Tabu Search and Simulated Annealing.

2 Brief Review of Related Work

One finds in the literature several attempts of building repair-based systems for solving CSPs. Systems reported in [7, 5, 20, 6] are particularly relevant to our work. These systems differ according to the types of constraints effectively allowed, the way of handling the constraints and the resolution techniques used.

In the work of [20], each CSP variable X_i with value domain D_i is replaced by a set of $|D_i|$ *binary* variables x_{ij}: $x_{ij} = 1$ if variable X_i takes value $j \in D_j$, 0 otherwise. In this 0-1 encoding of the CSP, the possible constraints are the linear inequalities. Penalties are employed for the purpose of constraint satisfaction. The basic resolution method is based on a tabu search algorithm. Another example is *Genet* and its variants [7, 5]. This system proposes a rich language of constraints that includes both binary and non-binary constraints. The resolution methods used in this system is based on repair heuristics called Min-Conflicts (MC) and Breakout. While MC may be considered as a very simple pure LS method (*i.e.* a descent), the Breakout heuristic gives a way to adapt dynamically penalties for escaping from local optima. Very recently, a similar system is also reported in [6].

In addition to these CSP-based systems, general heuristic approaches have also been proposed for combinatorial problems based on other models. For instance, in [21], a general formalism based on the list data structure is proposed for problem formulation and a simulated annealing algorithm is used for problem resolution. Similarly, in [25], the central formulation model is the so-called over-constrained integer program (OIP) (equivalent to the general ILP model) which is solved by a repair heuristic called Wsat(OIP). One also finds in the literature studies of building reusable software components for heuristic search. Typically, these systems are defined by a library of predefined functions which are implemented with the object-oriented programming technology. One finds several examples of such systems in [8] and the recent book [24]. Finally, let us mention that local search techniques have been recently added into some commercial constraint programming systems such as Ilog optimization tools and the ECLiPSe platform. For example, ECLiPSe supports using both constraint propagation and repairs [4].

3 A General LS Approach for Constraint Solving

As indicated previously, the CSP is a very general model for formulating various combinatorial search problems. In our system, we provide a set of *constraint primitives* (or constraint types, or simply constraints by short) which may be used to model the given problem as a CSP. For resolution purpose, we use a search engine based on Tabu Search. In this section, we describe the set of constraint primitives as well as the different components of our Tabu Search engine.

3.1 Constraint Primitives and their Semantics

Constraint primitives are introduced for the purpose of problem formulation. The expression power is clearly conditioned by both the number and the nature of the available primitives. The primitives introduced in this section are chosen essentially for their generality. Although this set of primitives is far from complete to encompass all the CSPs, they are sufficient to illustrate the main concepts of the proposed approach. Each primitive involves a list of variables $[y_1..y_p]$ and possibly other parameters. We denote by $R_C \subseteq D_{y_1} \times .. \times D_{y_n}$ the set of tuples for which constraint C is satisfied.

- Constraint *tuple*($[y_1..y_p]$, $[a_1..a_p]$) forbids the tuple $[a_1..a_p]$. We have: $s \in R_C \Leftrightarrow s(y_1) \neq a_1 \vee \cdots \vee s(y_p) \neq a_p$.

- Constraint *binary*($[[x, y]]$, $[a1_x, a1_y..a_x^p, a_y^p]$) is a binary constraint defined in extension by the list $(a1_x, a1_y), \cdots, (a_x^p, a_y^p)$ of forbidden couples[2]. We have: $s \in R_C \Leftrightarrow (s(x) \neq a1_x \vee s(y) \neq a1_y) \wedge \cdots \wedge (s(x) \neq a_x^p \vee s(y) \neq a_y^p)$.

- Constraint *different*($[x, y]$): We have: $s \in R_C \Leftrightarrow s(x) \neq s(y)$.

- Constraint *distance*($[x, y], D$): We have: $s \in R_C \Leftrightarrow |s(x) - s(y)| > D$.

- Constraint *alldifferent*($[y_1..y_p]$) checks that all variables $y_1, \cdots, y_p$ receive different values. Let $\mathcal{N}_s$ represent the number of pairs of variables having the same value in the tuple s. We have: $s \in R_C \Leftrightarrow \mathcal{N}_s = 0$.

- Constraint *atmost*($[y_1..y_p], a, P$) checks that the number of variables y_i taking the value a is inferior or equal to P. Let $\mathcal{N}_s$ represent the number of variables having the value a in the tuple s: $\mathcal{N}_s = |\{i/s(y_i) = a\}|$. We have: $s \in R_C \Leftrightarrow \mathcal{N}_s \leq P$.

- Constraint *atleast*($[y_1..y_p], a, P$) checks that the number of variables y_i taking the value a is superior or equal to P. Let $\mathcal{N}_s$ represent the number of variables having the value a in the tuple s: $\mathcal{N}_s = |\{i/s(y_i) = a\}|$. We have: $s \in R_C \Leftrightarrow \mathcal{N}_s \geq P$.

- In constraint *capa*($[y_1..y_p], a, [w_1..w_p], W$), a weight $w_i > 0$ is assigned to each variable y_i and the constraint checks that the sum of the weights of the variables taking the value a is inferior or equal to W. Let $\sigma_s = \sum_{i/s(y_i)=a} w(i)$. We have: $s \in R_C \Leftrightarrow \sigma_s \leq W$.

- Constraint *nbdifferences*($[x_1, y_1, \cdots, x_p, y_p]$,$P$) involves a set of pairs of variables (x_i, y_i), $1 \leq i \leq p$ and checks that the number of pairs of variables having the same value is inferior or equal to P. Let $\mathcal{N}_s$ represent the number of pairs of variables having the same value in the tuple s: $\mathcal{N}_s = |\{i/s(x_i) = s(y_i)\}|$. We have: $s \in R_C \Leftrightarrow \mathcal{N}_s \leq P$.

[2]Although the constraint is given here by extension, it would be possible to generate the tuples with a "for"-like structure.

These constraints make it possible to represent a number of constraint problems (see Section 4). Moreover, other constraint primitives may be added if necessary allowing a higher expressiveness. However, notice that incremental algorithms must be designed for all the constraint primitives to ensure an efficient resolution by local search (see Section 3.4).

3.2 Penalty Functions Associated to Primitives

Our Tabu Search engine relies heavily on a penalty-based evaluation function to guide its search (see section 3.3). Therefore, this function must be carefully designed. The basic idea is to assign a penalty to any violated constraint, the penalty being defined according to the degree of constraint violation. For some constraints, the penalty may be simply defined as a 0/1 value depending on whether the constraint is satisfied or not. For other constraints, more subtle penalties must be devised. More formally, the penalty function $f_C : D_{y_1} \times .. \times D_{y_n} \rightarrow IR$ of constraint C associates to each tuple $s \in D_{y_1} \times .. \times D_{y_n}$ a real value $f_C(s) \geq 0$. If the constraint is satisfied then this value is 0: $s \in R_C \Leftrightarrow f_C(s) = 0$; otherwise it is a strictly positive number: $s \notin R_C \Leftrightarrow f_C(s) > 0$.

- *tuple*$([y_1..y_p])$: $f_C(s) = 0/1$
- *binary*$([x, y], [a1_x, a1_y..a^p_x, a^p_y])$: $f_C(s) = 0/1$
- *different*(x, y): $f_C(s) = 0/1$
- *distance*(x, y, D): $f_C(s) = 0/1$

For the following constraints, the penalty depends on the degree of violation of the constraint by the considered tuple:

- *alldifferent*$([y_1..y_p])$: $f_C(s) = 0/\mathcal{N}_s$
- *atmost*$(P, [y_1..y_p], a)$: $f_C(s) = 0/\mathcal{N}_s - P$
- *atleast*$(P, [y_1..y_p], a)$: $f_C(s) = 0/P - \mathcal{N}_s$
- *nbdifferences*$([x_1, y_1, \cdots, x_p, y_p]$,$P)$: $f_C(s) = 0/\mathcal{N}_s - P$
- *capa*$([y_1..y_p], a, [w_1..w_p], W)$: $f_C(s) = 0/\alpha + \beta(\sigma_s - W)$, where (α, β) are two parameters

For additional constraints, we propose to use whenever possible the following principle: to fix the penalty function $f_C(s)$ to the minimum number of variables that need to be modified to reach a consistent assignment. Note that the penalties chosen for binary constraints, *tuple*, *atmost*, *atleast* and *nbdifferences* are coherent with this principle.

Depending on the penalty function, we define the notion of *critical variable*: variable y of constraint C is critical for a tuple s if changing the value of y in s (keeping the same values for the other variables) makes it possible to reduce the penalty of the constraint: y_i is a critical variable in s if and only if

$min_{v \in Dom(y_i)} f_C(s(y_1), .., s(y_{i-1}), v, s(y_{i+1}), .., s(y_n)) < f_C(s)$. If the constraint C is satisfied, then no variable is critical in s. Otherwise, all variables or only some of them are critical, depending on the constraint. For example, for *alldifferent* constraint, a variable is critical if it takes the same value as at least one other variable.

3.3 LS Search Engine

For a given search problem (S, f) with S being a finite set of configurations and f an evaluation function $f : S \rightarrow IR$, LS needs a so-called neighbourhood function $N : S \rightarrow 2^S$ ($N(s) \subseteq S$ is called the neighbourhood of $s \in S$). A LS algorithm begins with an initial configuration $s_0 \in S$ and then generates a series of configurations $(s_i)_{i \in \{0,1,\cdots\}}$ such that $\forall i \in \{0, 1, \cdots\}$, $s_{i+1} \in N(s_i)$. Well-known examples of LS methods include various descent methods, Simulated Annealing (SA) [18] and Tabu Search (TS) [13]. The main difference among LS methods concerns the way of visiting the given neighbourhood. In this study, we have chosen TS as our resolution engine. Indeed, TS has been applied with great success to many hard combinatorial problems [13]. Numerous studies of using TS for solving CSP-like problems also suggested the interest of this method for this class of problems. We define below the components of our local search engine that are the search space, the evaluation function, the neighbourhood function and the TS meta-heuristic.

Search Space

We call configuration any complete assignment, including inconsistent assignments. Such a configuration s can be represented by the series of values taken by the variables in X:

$$s = (s(x_1), \cdots, s(x_n)).$$

The search space is the set $S = D_{x_1} \times \cdots \times D_{x_n}$ of all configurations.

Evaluation Function

The evaluation function $f(s)$ of a configuration $s \in S$ is the weighted sum of the penalty functions of all the constraints of the given problem:

$$f(s) = \sum_C p_C * f_C(s)$$

where $p_C > 0$ is the weighting associated to the constraint C. These weightings are to be fixed empirically or automatically.

Neighbourhood Function

At each iteration, the LS heuristic replaces the current configuration by a new one obtained by a local transformation called a move. Given a configuration $s \in S$, a move consists in replacing in s the current value $s(x)$ of a variable x by a new value v: such a move is denoted by the couple $< x, v >$. For each $s \in S$, the set of the configurations that can be reached by such a move constitute the neighbourhood of s.

Restricted Neighbourhood

In order to make the search more effective, we use a heuristic that consists in restricting the choice of a move to *critical variables* (see Section 3.2): in other words, a possible move $< x, v >$ will be considered only if x is critical (a variable x is critical in configuration s if x is critical for at least one constraint).

TS Algorithm

The algorithm we use is called $TabuCSP$ [10]. $TabuCSP$ is a basic adaptation of the tabu meta-heuristic to the CSP. It uses a short term memory (tabu list) and a very simple aspiration mechanism. The role of the tabu list is to avoid short term cycling and to go beyond local optima. The principle of the tabu list is the following: before a move $< x, v >$ is performed, one memorizes in the tabu list the couple $< x, s(x) >$ for a fixed number of iterations (tabu tenure). This way, the move $< x, s(x) >$ is forbidden for this period; in other words, one forbids to assign to x its previous value $s(x)$. However a tabu move can be chosen if it makes possible to reach a configuration better than the best one found so far (in TS, removing in such a way the tabu status of a particular move is called *aspiration*). Notice that the optimal tabu tenure generally depends on the instance and is difficult to obtain. However, appropriate values can be found by limited experiments or by some automatic mechanisms.

Data : *tl :* tabu tenure;
Result : the best configuration found
begin
  generate a random configuration s
  while *not Stop-Condition* **do**
    choose the best authorized move $< x, v >$
    introduce $< x, s(x) >$ in the tabu list for *tl* iterations
    assign value v to variable x in s
  return the best configuration found
end

The algorithm first builds an initial configuration s: this initial configuration is simply built by assigning to each variable any value chosen randomly in its domain. At each iteration, the $TabuCSP$ algorithm considers all authorized moves and chooses the best one (break ties randomly). Recall that a move $< x, v >$ is authorized if (1) the variable v is critical and (2) either move $< x, v >$ is not tabu, or it improves the best solution found so far. The algorithm stops if a solution has been found ($f(s) = 0$) or if a fixed limit is reached concerning the number of iterations.

3.4 Incremental Data Structures

The efficiency of TS is greatly influenced by its ability to find quickly a best move at each iteration. Therefore, an important point when implementing a TS algorithm is the design of powerful incremental data structures and algorithms. One may find, in the literature, examples for dealing with simple constraints such as binary constraints [9].

In the context of this work, we must deal with more complex non-binary constraints. In what follows, we describe the general principle we have developed for such constraints.

We use a data structure denoted by γ that associates to each possible move $< x, v >$ a positive number denoted by $\gamma(< x, v >)$ (recall a possible move is any couple $< x, v >$ such that $x \in X$ and $v \in Dom(x)$). $\gamma(< x, v >)$ memorizes the sum of the weightings of the constraints that involve variable x and that would be violated if the value v would be assigned to variable x.

Data structure γ is initialized before the first move and then updated after each move, using incremental algorithms. Given this data structure, it is now possible to compute the performance of a move $< x, v >$, i.e. the variation $\delta(< x, v >)$ of the cost function that results from this move. $\delta(< x, v >)$ can be obtained in constant time according to the equation: $\delta(< x, v >) = \gamma(< x, v >) - \gamma(< x, s(x) >)$. Note that γ also indicates if a given variable x is in conflict: it is the case if and only if $\gamma(< x, s(x) >) > 0$.

In order to implement the tabu list, we use another data structure denoted by T; each element of T corresponds also to a possible move. to know in constant time whether a move is tabu: move $< x, v >$ is tabu if and only if the current number of iterations is smaller than $T(< x, v >)$.

4 Problem Representation

In this section, we show various examples of modeling well-known problems with the help of the constraint primitives introduced in the last section. Constraints of these problems belong to different types, and are often non-binary. For each problem, we give first a brief description and present then its formulation with constraint primitives. As we will see, the formulation of a given problem can be achieved in a compact and concise way, thanks to the expressive power of the primitives.

Boolean Satisfiability

A SAT instance is defined by a set of boolean variables and a set of clauses (disjunction of literals). The problem can be represented as follows:

- Each boolean variable x_i is represented by a variable.
- All domains equal $\{0, 1\}$.
- A literal l_j is associated the value $V(l_j) = 1$ (0) if it is positive: $l_j = x_i$ (negative: $l_j = \neg x_j$). A clause $l_i \wedge \cdots \wedge l_n$ is represented by a constraint *tuple* $([l_1 \cdots l_n], [V(l_1) \cdots V(l_n)])$.

Graph k-Coloring and Graph Coloring

Let $G = (V, E)$ be an undirected graph with a vertex set V and an edge set E. A k-coloring of G is any assignment $\phi : V \rightarrow \{1 \cdots k\}$ such that no two endpoints of a

same edge receives the same value: $\{x, y\} \in E \Rightarrow \phi(x) \neq \phi(y)$. An optimal coloring of G is a k-coloring with the smallest possible k (the chromatic number $\chi(G)$ of G).

Given a couple $(G = (V, E), k)$, the k-coloring problem $P_{G,k}$ consists in finding a k-coloring of G. Problem $P_{G,k}$ is represented in the following way :

- Each vertex $v_i \in V$ is assigned a variable x_i.
- All domains equal $\{1..k\}$.
- Each edge $v_i v_j \in E$ is associated to a constraint *different*$([x_i, x_j])$.

To find an optimal coloring of G (graph coloring problem), we solve $P_{G,k}$ with decreasing values of k: given a value k_0 for which a k -coloring is known, we continue with $k = k_0 - 1, k_0 - 2, \cdots$ until we fail.

Maximum Clique

undirected graph $G = (V, E)$ is any complete sub-graph of G. Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, the Maximum Clique problem is to find a clique of maximal cardinal.

We call $P_{G,k}$ the problem to find a clique with a fixed number k of vertices. Problem $P_{G,k}$ can be represented in the following way :

- Each vertex $v_i \in V$ is assigned a variable x_i.
- All domains equal $\{0, 1\}$.
- Each non-edge $v_i v_j \notin E$ is associated to a constraint *tuple*$([x_i, x_j], [1, 1])$.
- There is a constraint *atleast*$([x_1..x_n], 1, k)$.

To find a maximum clique of G, we solve $P_{G,k}$ with increasing successive values $k = 1, 2, ...$ until we fail.

Frequency Assignment Problem

A cellular network is defined by a set $\{C_1, C_2...C_N\}$ of N cells, each cell C_i requiring T_i frequencies. The possible values for the frequencies are represented by consecutive integers in the interval $[0..NF]$. Interference occurs when two frequencies assigned to a same cell or two adjacent cells are not sufficiently separated. Therefore, there are two kinds of constraints:

- *co-cell constraint*: any pair of frequencies assigned to a radio cell must have a certain distance between them in the frequency domain.
- *adjacent-cell constraint*: any pair of frequencies assigned to two adjacent cells must be sufficiently separated in the frequency domain.

These constraints are conveniently represented by a symmetric *compatibility matrix* $M[N, N]$ where N is the number of cells in the network and each element of M is a non negative integer. Let $f_{i,k}$ denote the value of the kth frequency ($k \in \{1..T_i\}$) of C_i and let $\{1..NF\}$ denote the set of NF available frequency values, then the interference constraints are formulated as follows:

- $M[i, i]$ (i $\in \{1..N\}$) is the minimum frequency separation necessary to satisfy the co-cell constraints for the cell C_i. $\forall$ m, n $\in \{1..T_i\}$, m $\neq$ n, $|f_{i,m} - f_{i,n}| \geq M[i, i]$
- $M[i, j]$ (i, j $\in \{1..N\}$, i $\neq$ j) represents the minimum frequency separation required to satisfy the adjacent-cell constraints between two cells C_i and C_j. $M[i, j] = 0$ means there is no constraint between the cells C_i and C_j.
  $\forall$m $\in \{1..T_i\}$, $\forall$n $\in \{1..T_j\}$, $|f_{i,m} - f_{j,n}| \geq M[i, j]$

We represent the problem as follows:

- Each frequency $f_{i,l}$ is represented by a variable $x_{i,l}$
  $X = \{x_{i,l}, 1 \leq i \leq N, 1 \leq l \leq T_i\}$.
- All domains equal $[1..NF]$.
- We define the following constraints:
  - *co-cell constraint*: for each $i \in [1..N]$, for each $l, m \in [1..T_i], l < m$ we require: *distance*$([f_{i,l}, f_{i,m}], M[i, i])$.
  - *adjacent-cell constraint*: for each $i, j \in [1..N], i < j$, for each $l \in [1..T_i]$ and each $m \in [1..T_j]$ we require: *distance*$([f_{i,l}, f_{j,m}], M[i, j])$.

Progressive Party Problem

The Progressive Party Problem (PPP) appeared in a yacht club in order to organize a party that lasts for different successive time periods [3]. In the problem, there are a given number H of host boats that invite the G guest crews of other boats on their board. The size $c(g)$ of each guest crew $g \in [1..G]$ and the capacity $C(h)$ of each boat $h \in [1...H]$ are given. For each time period, each guest crew must visit a host boat respecting the following constraints:

- The capacities of the host boats must be respected.
- Each guest crew must move to a different boat for each time period.
- Two guest crews can meet at most once.

An assignment plan consists in assigning to each guest crew a boat to visit for each time period. The problem consists in finding a valid assignment plan for a maximum number of time periods. We denote by P_T the problem of finding a valid assignment

plan for a fixed number T of time periods[3]. To deal with PPP, we solve the series of problems $P_1, P_2, \cdots$.

We represent problem P_T in the following way:

- For each crew $g \in [1..G]$ and each time period $t \in [1..T]$, variable $x_{g,t}$ represents the boat visited by g at period t:
  $X = \{x_{g,t}, 1 \leq g \leq G, 1 \leq t \leq T\}$.
- All domains equal $[1..H]$.
- We define the following constraints:
  - For each time period $t \in [1...T]$ and each boat $h \in [1...H]$ we require:
    $capa([x_{1,t} \cdots x_{G,t}], [c(1) \cdots c(G)], h, C(h))$.
  - For each guest $g \in [1..G]$ we require:
    $alldifferent([x_{g,1} \cdots x_{g,T}])$.
  - For each couple (g_1, g_2) (with $g_1 < g_2$), we require:
    $nbdifferences([x_{g_1,1}, x_{g_2,1}, \cdots, x_{g_1,T}, x_{g_2,T}], 1)$.

Let us mention that we can formulate other well-known combinatorial problems including bin-packing, knapsack, unicost set covering problem, etc. In addition to progressive party problem and frequency assignment problem, we may conveniently formulate other complex real-world problems such as the problem of daily photograph scheduling of an earth observation satellite and the sports league scheduling problem.

5 Problem Resolution

In this section, we show in some detail numerical results for three of the problems introduced in the last section: the Graph Coloring Problem, the Frequency Assignment Problem and the Progressive Party Problem. Whenever possible, results are contrasted with those known in the literature. We give also performance indications for Binary Max-CSP instances solved by the proposed approach.

Graph k-coloring and graph coloring

For graph coloring, we used the following graphs from the well-known second DIMACS challenge benchmarks [17][4].

- Three *random graphs*: DSJC250.5, DSJC500.5 and DSJC1000.5. They have 250, 500 and 1000 vertices respectively and a density of 0.5 with unknown chromatic number (the smallest number of colors reported in the literature for these graphs are 28, 48 and 83 respectively).

[3] In the original problem, T=6. Larger T makes the problem harder.

[4] Available via ftp from ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/.

- Two *Leighton graphs*: le450_15c and le450_25c. They are structured graph with known chromatic number (respectively 15 and 25).

- two *flat graphs*: flat300_28 and flat1000_76. They are also structured graph with known chromatic number (respectively 38 and 76).

We are interested in these graphs because they were largely studied in the literature and constitute thus a good reference for comparisons. Moreover these graphs are difficult and represent a real challenge for graph coloring algorithms.

The tabu tenure tl used in these experiments is variable and depends on the number nb_{CFL} of conflicting vertices in the current configuration: $tl = Random(A) + \alpha * nb_{CFL}$ where A and α are two parameters and the function $Random(A)$ returns randomly a number in $\{0, \cdots, A-1\}$. Experiments of various combinations suggested that ($A = 10$, $\alpha = 0.6$) is a robust combination for the chosen graphs.

graph	k	$TabuCSP$		
DSJC250.5	28	10	2,500,000	355
	29	10	587,000	85
	30	10	97,000	15
DSJC500.5	50	10	1,495,000	402
	51	10	160,000	47
	52	10	43,000	14
DSJC1000.5	89	3(2)	4,922,000	2,099
	90	5	3,160,000	1,357
	91	5	524,000	226
	92	5	194,000	85
le450_15c	16	8 (2)	319,000	69
	17	10	18,000	5
le450_25c	26	10	107,000	38
	27	10	7,300	4
flat300_28	32	10	149,000	25
flat1000_76	87	1(4)	7,400,000	3,301
	88	2(3)	4,000,000	1,820

Table 1: Results of $TabuCSP$ for graph k-coloring

Table 1 reports results with our approach. These results are also cited in [12] as a reference for a hybrid evolutionary algorithm. Each line corresponds to particular k-coloring instance and gives the results obtained by $TabuCSP$ on this instance. These experiments consist in a series of runs, each run being limited to 10 Millions iterations. Column 3 indicates the number of successful executions and the number of fails. Column 4 and 5 display the average number of iterations and the average time for successful runs (the timing is based on an UltraSPARC-IIi 333MHz with 132 MB RAM). For each graph, the smallest value of k in the table is the smallest one for which $TabuCSP$ could

find a solution. For example, for DSJC1000.5, tabu found a solution with 89 colors but failed to find one with 88 colors.

For the k-coloring problem, $TabuCSP$ is reduced to the tabu algorithm called $TabuCOL$ of [15]. There are other known strategies for the k-coloring problem and the graph coloring problem (see [16]). Compared with these other methods, the approach used by $TabuCSP$ remains the simplest one and allows to produce competitive results with respect to other "pure" LS techniques [15, 16, 9]. Note however that, for some large graphs, $TabuCSP$ and other pure LS techniques are outperformed by hybrid strategies, notably the Hybrid Evolutionary Algorithm of [12].

Frequency Assignment

For FAP, we used a set of instances proposed by France Telecom (CNET) [14][5]. We experimented with two different ways to fix the weightings of the constraints: 1) uniform weightings, 2) weightings fixed to 1 for adjacent constraints and to infinite for co-cell constraints. Note that the option 2 is equivalent to limit the search space to the configurations that respect the co-cell constraint.

Problem	Opt/LB	$TabuCSP$			SA		CP		GCA
		NF(S)	Iter	T[sec]	NF	T[sec]	NF	T[sec]	NF
8.150.20	8	8(10)	18 923	123	8	509	9	7 200	8
8.150.30	8	8(10)	404	3	8	446	12	10 800	15
15.300.20	15	15(10)	41 484	573	15	4 788	17	3 600	20
15.300.30	15	15(10)	22 429	327	15	2 053	24	36 000	27
8.75.25.1	16	17(10)	34 414	274	17	1 382	20	1 000	19
8.75.25.3	16	16(5)	62 764	485	17	1 744	19	7 595	19
8.150.15.3	16	18(10)	46 425	668	18	3 705	22	375	22
8.150.25.6	16	16(3)	56 120	906	17	5 981	30	153	29
15.300.25.6	30	35(2)	78 266	2 940	36	5 359	47	380	47
15.300.25.9	30	35(1)	61 294	2 168	36	6 586	45	-	45

Table 2: Results of $TabuCSP$ for frequency assignment

We have used $TabuCSP$ to solve these instances. In table 5, the results of $TabuCSP$ are compared with those of the best known results obtained by approaches such as Simulated Annealing (SA), Constraint Programming (CP) and Graph Coloring Algorithms (GCA), reported in [14]. We notice first that $TabuCSP$ outperforms CP and GCA on these instances in terms of the quality of solutions. Moreover, $TabuCSP$ outperforms the SA algorithm on 4 instances and does as well as SA on the other instances (In terms of computing time, $TabuCSP$ is always faster than SA even for solutions of better quality).

The results show that our approach is very powerful for the FAP. Moreover, we notice that it is flexible enough to try different options to solve a same problem.

[5]These instances are available from the second author of the paper.

Progressive Party Problem

Recall that to solve the PPP, we solve a series of CSPs $(P_T)_{T=1,2,...}$, where P_T represents the problem of finding an assignment for T time periods. In the literature, only one instance of the Progressive Party Problem has been widely tested - that one of the original organization problem. In this instance, there are $G = 29$ crews and $H = 13$ boats, the party is organized for $T = 6$ time periods. We tried to solve the problem P_T for $T = 6$ to $T = 10$ (10 is an upper bound of the number of time periods).

After limited preliminary experiments, we decided to use weightings equal to 1 for *nbdifferences* constraints and 2 for the two other constraints.

problem	ILP	CP		$TabuCSP$	
		CP1	CP2	T[sec]	Iter
P_6	fail	27 min.	a few sec.	0.3 sec.	210
P_7	fail	28 min.	a few sec.	0.5 sec.	330
P_8	fail	fail	a few sec.	1.7 sec.	1,366
P_9	fail	fail	several hours	67.5 sec.	51,507
P_{10}	fail	fail	fail	fail	

Table 3: Results of $TabuCSP$ for the Progressive Party Problem

Table 3 presents the best known results obtained with three different methods: ILP (Integer Linear Programming) [3], CP [22, 2] and $TabuCSP$. From the table, we see that ILP fails to solve any of P_6 to P_{10}. The results of CP indicated by CP1 [22] and CP2 [2] are much more interesting. Indeed, CP1 solves P_7 and P_8 in 27 and 28 minutes respectively, but fails to solve P_9 and P_{10}. The results of CP2 are much better: indeed P_7 and P_8 are now solved in a few seconds (after several hundreds of backtracks). The problem P_9 is also solved, but using several hours (and millions of backtracks).

Using $TabuCSP$, we solve the problem up to 9 time periods. P_8 is solved very easily in two seconds (on a Sun ULTRA 1, 128 RAM, 134 MHz) with a few hundreds of iterations while P_9 is solved in about one minute with a few thousands of iterations. It is still an open question whether a solution exists for 10 time periods. However, $TabuCSP$ finds frequently configurations involving only one violated constraint.

Binary Max-CSP instances

Finally, we also used $TabuCSP$ for solving binary Max-CSP instances. The constraints in explicit binary CSPs (or Max-CSPs) are represented using the *binary* primitive. In [10], the results of $TabuCSP$ are reported and compared with those of two repair heuristics, notably the *Min-conflict Random Walk* heuristic ($MCRW$) considered to be among the most efficient AI heuristics. The comparison showed clearly that $TabuCSP$ outperforms $MCRW$ for these instances.

6 Discussion and Conclusion

In this paper, we presented a general approach for solving constraint problems by local search. This approach is based on the definition of a set of high-level constraint primitives and an advanced local search engine. The primitives provide a convenient way to model various combinatorial search problems while the LS engine ensures an efficient resolution. For an efficient handling of these primitives by LS, we introduced a set of appropriate penalty functions.

Another important issue we have addressed concerns the implementation of the TS resolution engine. Indeed, in order to make the search engine as efficient as possible, incremental data structures and incremental algorithms have been developed. Combined with a simple neighbourhood and an appropriate penalty technique for constraint handling, the resulting TS algorithm proves to be a powerful solver.

To validate the approach, we have presented examples of modeling various combinatorial problems including the progressive party problem, graph coloring problem and frequency assignment problem. Computational experiments on some well-known instances showed that the proposed approach is not only general, but also able to produce very competitive results.

In this paper, we have focused our study to the CSP model where one seeks a feasible solution for a given problem instance. It should be mentioned that there is no real difficulty to extend the system for constrained optimization problems where one is given a cost function in addition to constraints. Indeed, the penalty approach studied in this paper is naturally applicable to deal with the constraints and the cost function simultaneously.

To conclude, we think general constraint solving by LS constitutes an interesting and important alternative for tackling large and difficult combinatorial (satisfaction or optimization) problems and deserves more research efforts in the future.

References

[1] E.H.L. Aarts, J.K. Lenstra (Eds.), *Local Search in Combinatorial Optimization*, John Wiley & Sons, 1997.

[2] N. Beldiceanu, E. Bourreau, P. Chan, D. Rivreau, Partial search strategy in CHIP. *Presented at Metaheuristics International Conference*, Sophia-Antipolis, France, 1997.

[3] S.C. Brailsford, P. M. Hubbard, B. M. Smith, The progressive party problem: a difficult problem of combinatorial optimization. *Computers and Operations Research*, 23:845-8 56, 1996.

[4] A.M. Cheadle, W. Harvey, A.J. Sadler, J. Schimpf, K. Shen and M.G. Wallace, ECLiPSe: An Introduction, IC-Parc, Imperial College London, Technical Report IC-Parc-03-1, 2003.

[5] K. M. F. Choi, J. H. M. Lee, P. J. Stuckey, A Lagrangian reconstruction of GENET, *Artificial Intelligence* 123(1-2): 1-39, 2000.

[6] P. Codognet, D. Diaz, Yet another local search method for constraint solving. *Lecture Notes in Computer Science* 2264: 73-88, Springer-Verlag, 2001.

[7] A. Davenport, E.P.K. Tsang, Z. Kangmin, C. J. Wang, GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. Proc. AAAI'94, p.325-330, 1994.

[8] J.A. Ferland, A. Hertz, A. Lavoie, An object-oriented methodology for solving assignment type problems with neighborhood search techniques. *Operations Research*, 44(2):347-359, 1996.

[9] C. Fleurent, J.A. Ferland, Object-oriented implementation of heuristic search methods for graph coloring, maximum clique and satisfiability. In [17], pp619-652.

[10] P. Galinier, J.K. Hao, Tabu search for maximal constraint satisfaction problems, *Lecture Notes in Computer Science* 1330: 196-208, Springer-Verlag, 1997.

[11] P. Galinier, J.K. Hao, "Solving the progressive party problem by local search", S. Voss, S. Martello, I.H. Osman and C. Roucairol (Eds.) *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Chapitre 29: pages 419-432, Kluwer Academic Publishers, 1998.

[12] P. Galinier, J.K. Hao, Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4): 379-397, 1999.

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

[14] J.K. Hao, R. Dorne, P. Galinier, Tabu search for frequency assignment in mobile radio networks, *Journal of Heuristics*, 4(1): 47-62, 1998.

[15] A. Hertz and D. de Werra, "Using Tabu Search techniques for graph coloring". *Computing*, 39: 345-351, 1987.

[16] D.S. Johnson, C.R. Aragon L.A. McGeoch, C. Schevon, "Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning", *Operations Research*, 39(3): 378-406, 1991.

[17] D.S. Johnson and M.A. Trick, "Proceedings of the 2nd DIMACS implementation challenge", Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996.

[18] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, Optimization by Simulated Annealing, *Science*, 220:671-680, 1983.

[19] A.K. Mackworth, "Constraint satisfaction", in S.C. Shapiro (Ed.) *Encyclopedia on Artificial Intelligence*, John Wiley & Sons, NY, 1987.

[20] K. Nonobe, T. Ibaraki, A tabu search approach to the constraint satisfaction problem as a general problem solver. *European Journal of Operational Research*, 106:599-623, 1998.

[21] M. Randall, D. Abramson, A general meta-heuristic based solver for combinatorial optimisation problems. *Computational Optimisation and Applications* 20: 185-210, 2001.

[22] B. M. Smith, S. C. Brailsford, P. M. Hubbard, H. P. Williams, The Progressive party problem: integer linear programming and constraint programming compared. *Constraints*, 1(1/2):119-138, 1996.

[23] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[24] S. Voss, D.L. Woodruff (Eds.), *Optimization Software Class Libraries*, Book Series: Operations Research/Computer Science Interfaces : Volume 18, Kluwer Academic Publisher, 2002.

[25] J.P. Walser, *Integer Optimization by Local Search*, Lecture Notes in Artificial Intelligence 1637, Springer-Verlag, 1999.