Proceedings of the International MultiConference of Engineers and Computer Scientists 2008 Vol I
IMECS 2008, 19-21 March, 2008, Hong Kong

Towards a Formalization of Combinatorial Local Search

Eric Monfroy, Frédéric Saubion, Broderick Crawford and Carlos Castro *

Abstract—**Although there are some frameworks to formalize constraint propagation, there are only few studies of theoretical frameworks for local search. We are here concerned with the design of a generic framework to model local search as the computation of a fixed point of functions. This work allows one to simulate standard strategies used for local search, and to easily design new strategies in a uniform framework.**

Keywords: Constraint Satisfaction Problems (CSP), Local Search, Constraint Solving

1 Introduction

Constraint Satisfaction Problems (CSP) [11] provide a general framework for the modeling of many practical applications (planning, scheduling, time tabling,...). A CSP is usually defined by a set of variables associated to domains of possible values and by a set of constraints. We only consider here CSP over finite domains. Constraints can be understood as relations over some variables and therefore, solving a CSP consists in finding tuples that belong to each constraint (an assignment of values to the variables that satisfies these constraints).

From a practical point of view, CSP can be solved by using either complete (such as constraint propagation [3]) or incomplete techniques (such as local search [1] or genetic algorithms [7]). Therefore, constraint solvers mainly rely on the implementation and combination of these techniques. Local search techniques [1] have been successfully applied to various combinatorial optimization problems. In the CSP solving context, local search algorithms are used either as the main resolution technique or in cooperation with other resolution processes (e.g., constraint propagation) [5, 9, 4]. Unfortunately, the definitions and the behaviors of these algorithms are often strongly related to specific implementations and problems.

Our purpose is to define a framework based on functions to provide uniform modeling tools which could help better understanding local search algorithms and designing new ones. The idea is similar to [10], but with a much finer grain definition of functions.

From a more conceptual and theoretical point of view, K.R. Apt has proposed a mathematical framework [2, 3] for iteration of a finite set of functions over "abstract" domains with partial ordering: this is well-suited for solving CSPs with constraint propagation. The purpose of this paper is to focus on the modeling of basic local search processes and then to improve this previous work by providing a more comprehensive definition to local search algorithms.

To obtain a finer definition of local search, we propose a computation structure (the domain of Apt's iterations) which is better suited for local search. Then, we define the basic functions that will be used iteratively on this structure to create a local search process. We identify here precisely the two basic processes used for intensification and diversification (move and neighborhood computation) and the process for jumping to other parts of the search space (restart). These three processes are abstracted at the same level by some homogeneous functions called reduction functions. The result of local search is then computed as a fixed point of this set of functions. Moreover, the theoretical framework of [3] is then extended here to fit our new function definitions.

This allows us to take into account in a single model various moves (such as for improvement, diversification, non looping, ...), neighbors (such as all neighbors, improving neighbors, ...), and restart functions that can be interleaved as needed.

2 Solving CSP with Local Search

A CSP is a tuple (X, D, C) where $X = \{x_1, \cdots, x_n\}$ is a set of variables taking their values in their respective domains $D = \{D_1, \cdots, D_n\}$. A constraint $c \in C$ is a relation $c \subseteq D_1 \times \cdots \times D_n$. Note that for sake of simplicity, we consider that each constraint is over all the variables $x_1, \ldots, x_n$. However, one can consider constraints over some of the x_i. Then, the notion of scheme [3] can be used to denote sequences of variables. In order to simplify notations, D will also denote the Cartesian product of D_i and C the union of its constraints. A tuple $d \in D$ is a solution of a CSP (X, D, C) if and only if $\forall c \in C, d \in c$. In this paper, we always consider D finite.

*The author's affiliations, respectively, are:
Universidad Técnica Federico Santa María, Valparaíso, Chile
Université de Angers, France
Pontificia Universidad Católica de Valparaíso, Chile
Universidad Técnica Federico Santa María, Valparaíso, Chile

Given an optimization problem (which can be minimizing the number of violated constraints and thus trying to find a solution to the CSP), local search techniques [1] aim at exploring the search space, moving from a sample (i.e., a representative of a tuple which is usually the tuple itself but could also be a set of tuples such as a scatter of tuples, a box of tuples, ...) to one of its neighbors.

For the resolution of a CSP (X, D, C), the search space can be often defined as the set of possible tuples of $D = D_1 \times \cdots \times D_n$ and the neighborhood is a mapping $\mathcal{N} : D \to 2^D$. This neighborhood function defines indeed the possible moves from a sample (i.e., a tuple in this case) to one of its neighbors and therefore fully defines the exploration landscape. The fitness (or evaluation) function *eval* is related to the notion of solution and can be defined as the number of constraints c that are not satisfied by the current sample, i.e., constraints such that $d \notin c$ (d being the currently visited sample, i.e., an element of D in this case). The problem to solve is then a minimization problem. Given a sample $d \in D$, two basic cases can be identified in order to continue the exploration of D:

- intensification: choose $d' \in \mathcal{N}(d)$ such that $eval(d') < eval(d)$.

- diversification: choose any other neighbor d'.

In order to integrate possible restarts (to start new paths) and to generalize this approach we will consider local search as a set of basic local searches.

3 A Computational Framework

3.1 The Computation Structure

As we have seen, local search acts usually on a structure which corresponds to points of the search space. Here, we propose a more general and abstract definition based on the notion of sample, already suggested.

Definition *(Sample)* Given a CSP (X, D, C), a sample function is a function $\varepsilon : D \to 2^D$. By extension, $\varepsilon(D)$ denotes the set $\{\varepsilon(d) | \, d \in D\}$.

Generally, $\varepsilon(d)$ is restricted to d and $\varepsilon(D) = D$, but it can also be a scatter of tuples around d, or a box of tuples covering d. Indeed, the search space D is abstracted by $\varepsilon(D)$ to be used by the local search. Note that in any case, $\varepsilon(D)$ is finite since we consider D to be finite.

The general process of local search can be abstracted by three stages:

- start or restart a search from a given starting sample,
- generate the neighborhood of the current sample,
- move from the current sample to one of the previously computed neighbor.

Definition *(Local Search Path)* A local search path p is a finite sequence $(s_1, \cdots, s_n)$ such that $\forall 1 \leq i \leq n, s_i \in \varepsilon(D)$.

A local search configuration is given by a local search path and a set of possible candidates for the next move (i.e., neighbors of the last encountered sample).

Definition *(Local Search Configuration)* A local search configuration is a pair (p, V) where $p = (s_1, \cdots, s_n)$ is a local search path and $V \subseteq 2^{\varepsilon}(D)$.

We denote by $P_{\varepsilon(D)}$ (resp. $C_{\varepsilon(D)}$) the set of all possible local search paths (respectively configurations) on $\varepsilon(D)$. Given a tuple $p = (s_1, \cdots, s_n) \in \varepsilon(D)^n$, and an element $s \in \varepsilon(D)$, we denote $p' = p \oplus s$ the tuple $(s_1, \cdots, s_n, s)$. To simplify notation, we denote $s \in p$ the fact that a sample s is a component of a path p.

Practically, a local search process aims at building a finite path whose length is either determined by reaching a solution or by having performed a maximum number of iterations. Therefore, we now define the orderings for the above structures taking into account these two main aspects of local search.

Definition *(Ordering on paths and configurations)* Given $p = (s_1, \cdots, s_n)$ and $p' = (s'_1, \cdots, s'_m)$ two paths of $P_{\varepsilon(D)}$, $p \sqsubseteq p'$ iff

- $s'_m \in Sol_{\varepsilon(D)}$
- or $s_n \notin Sol_{\varepsilon(D)}$ and $m \geq n$.

Given $c = (p, V)$ and $c' = (p', V')$ two configurations of $C_{\varepsilon(D)}$, $c \sqsubseteq c'$ iff either:

- $p \sqsubseteq p'$,
- or $p = p'$ and $V \subseteq V'$.

According to this definition, a path leading to a solution will not be extended and corresponds to a final state of computation: indeed reaching a solution is usually used as a stop criterion in local search algorithms.

Note the notion of congruence on paths: two paths $p = (s_1, \cdots, s_n)$ and $p' = (s'_1, \cdots, s'_m)$ can be "syntactically different" but equal with our ordering: $s_n \in Sol_{\varepsilon(D)}$ and $s_m \in Sol_{\varepsilon(D)}$ (as well as $s_n \notin Sol_{\varepsilon(D)}$, $s_m \notin Sol_{\varepsilon(D)}$, and $n = m$) is sufficient to have $p = p'$. This notion of congruence extends to configurations.

In order to generalize the basic local search process and to handle specific operations such as restart, we consider now a local search as a set of local search configurations that we call a *whole local search*. Therefore, we define the following structure : $\mathcal{LS}_{\varepsilon(D)} = 2^{C_{\varepsilon(D)}}$ as our general computation structure. The elements that we will handle are thus finite sets of configurations that we will often denote $\lambda = \{c_1, \ldots, c_n\}$.

Since most of the times we will be given D and we will consider only one sampling function ε on D, in the following we will often abbreviate $P_{\varepsilon(D)}$ (respectively $C_{\varepsilon(D)}$, and $\mathcal{LS}_{\varepsilon(D)}$) to P (respectively to C, and $\mathcal{LS}$).

Before extending ordering on configurations to ordering on sets of configurations, we need to define the notion of comparable couples. $|S|$ denotes the cardinality of the set S.

Definition *(Coupling)* Given $\lambda = \{c_1, \ldots, c_n\}$ and $\lambda' = \{c'_1, \ldots, c'_n\}$ in $\mathcal{LS}$, a coupling from λ and λ' is a set S such that:

- $S \subseteq \lambda \times \lambda'$,
- $\forall (c_i, c'_j) \in S,\ c'_j \sqsubseteq c_i \ \vee \ c_i \sqsubseteq c'_j$,
- and, $\forall (c_i, c'_j) \in S,\ (\nexists l \in [1..n], (c_i, c'_l) \in S) \wedge (\nexists k \in [1..n], (c_k, c'_j) \in S)$.

We say that a coupling S is maximum if there does not exist any coupling S' from λ and λ' such that $|S| \leq |S'|$.

$S_{\sqsupseteq}$ (respectively $S_{\sqsubseteq}$) is the set defined by $S_{\sqsupseteq} = \{(c_i, c'_j) \in S \,|\, c'_j \sqsubseteq c_i\}$ (respectively $S_{\sqsubseteq} = \{(c_i, c'_j) \in S \,|\, c_i \sqsubseteq c'_j\}$).

Based on this notion of coupling, we can now define an ordering on elements of $\mathcal{LS}$. Informally, let λ and λ' be two elements of $\mathcal{LS}$. Then, $\lambda \sqsubseteq \lambda'$:

- if they have the same size, and there are less elements of λ' that are smaller than elements of λ than elements of λ that are smaller than elements of λ'.
- or if λ' is bigger, and has a subset (of the same size as λ) which is also bigger than λ.

Definition *(Ordering on $\mathcal{LS}$)* Given λ and λ' in $\mathcal{LS}$, $\lambda \sqsubseteq \lambda'$ iff either:

- $|\lambda| = |\lambda'|$, and there is a maximum coupling S from λ to λ' such that $|S_{\sqsupseteq}| \leq |S_{\sqsubseteq}|$ and $|S| = n$,
- or, $|\lambda| \leq |\lambda'|$ and there exists $\lambda'' \subseteq \lambda'$ such that $|\lambda''| = |\lambda|$ and $\lambda \sqsubseteq \lambda''$

In the next section, we define the functions that will apply on these structures to perform local search.

3.2 Reduction Function Definitions

Our definition of reduction functions is based on K.R. Apts framework.

Definition *(Reduction function on a structure)* Given a partial ordering $(D, \sqsubseteq)$, a reduction function f is a function from D to D which satisfies the following properties:

- $\forall x \in D, x \sqsubseteq f(x)$ (inflationary)
- $\forall x, y \in D, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ (monotonic)

We may now distinguish two kinds of reduction functions as they apply either on a single local search configuration or on a whole local search composed of several configurations.

Definition *(Move Function)* A move function is a function:

$$\begin{array}{rl} \mu: & C \to C \\ & (p, V) \mapsto (p', \emptyset) \end{array}$$

where:

- $p' = p \oplus s$ with $s \in V$ if $p = p'' \oplus s'$ and $s' \notin Sol_{\varepsilon(D)}$ and $V \neq \emptyset$,
- $p' = p$ otherwise.

Definition *(Neighborhood Func.)* A neighborhood function is a function:

$$\begin{array}{rl} \nu: & C \to C \\ & (p, V) \mapsto (p, V \cup V') \end{array}$$

such that $V' \subseteq \varepsilon(D)$ and $V' \cap V = \emptyset$.

Move and neighborhood functions are reduction functions on C. The proof is straightforward.

We now define a function which applies on a whole local search to generate a new configuration, i.e., a starting sample from $\varepsilon(D)$ and an empty set of neighbors that will be used later to make a new search.

Definition *(Restart Function)*

$$\begin{array}{rl} restart: & \mathcal{LS} \to \mathcal{LS} \\ & \lambda \mapsto \lambda \cup \{(s, \emptyset)\} \ \text{ with } \ s \in \varepsilon(D) \end{array}$$

Restart functions are reduction functions on $\mathcal{LS}$. The proof is straightforward.

3.3 Control for Reduction Function

In order to apply the basic move and neighborhood functions on an element λ of $\mathcal{LS}$ we need to select a specific configuration c in λ using a selection function:

$$\begin{aligned} select: \quad & \mathcal{LS} \rightarrow C \\ & \lambda \mapsto c \ \text{ with } \ c \in \lambda \end{aligned}$$

We now extend move and neighborhood functions on $\mathcal{LS}$ using *select*.

Definition *(Extended Move and Neighborhood Functions)* Let f be a move or a neighborhood function. Then its extension f^{select} is defined by:

$$\begin{aligned} f^{select}: \quad & \mathcal{LS} \rightarrow \mathcal{LS} \\ & \lambda \mapsto \lambda' \end{aligned}$$

where $\lambda = [c_1, \ldots, c_i, \ldots, c_l]$ and $\lambda' = [c_1, \ldots, f(c_i), \ldots, c_l]$ with $select(\lambda) = c_i$.

Extended move and neighborhood functions are reduction functions on $\mathcal{LS}$. The proof is straightforward.

Note that this control aspects introduce non determinism in the basic reduction functions. This will be discussed in section 4. At this step, we restrict the functions in order to fit a real operational framework.

3.4 Restricting Functions to Match a Practical Framework

As mentioned before, the stop conditions are always added in local search algorithms to insure termination. These conditions are basically based on a maximum number of allowed search steps or on a notion of solution (if this notion is available). In the context of CSP solving, this notion of solution has been clearly defined and is taken into account in the definition of our computation structure. The second aspect will be introduced in the move and neighborhood functions.

Here the maximum number of operations will be defined by a maximum number σ of steps in each path (maximal length of a path) and a maximum number ρ of restarts (number of attempts to build a path leading to a solution).

Given an extended move or neighborhood function $f^{select}: \mathcal{LS} \rightarrow \mathcal{LS}$, we define its restriction $f^{(select,\sigma)}: \mathcal{LS} \rightarrow \mathcal{LS}$ as:

- $f^{(select,\sigma)}(\lambda) = f^{select}(\lambda)$ if $|select(\lambda)| \leq \sigma$
- and $f^{(select,\sigma)}(\lambda) = \lambda$ otherwise.

Concerning the restart function, its restriction is defined as:

- $restart^{\rho}(\lambda) = restart(\lambda)$ if $|\lambda| \leq \rho$
- and $restart^{\rho}(\lambda) = \lambda$ otherwise.

We must insist on the fact that, after these practical restrictions, we only consider P as the set of all possible local search paths of size σ. Therefore, this set is finite and C and $\mathcal{LS}$ are also finite. Note that only the restriction concerning σ is required to insure finiteness of the structures which is needed to fit the generic iteration framework (section 4).

3.5 Fairness

We now define fairness notions that are useful at two levels: 1) to fairly consider all samples of $\varepsilon(D)$ with restart functions, 2) to fairly consider all the configurations of a whole local search. These notions are correct w.r.t. the fact that our structures D, $\varepsilon(D)$, C and $\mathcal{LS}$ are finite.

A restart function r is *fair* if it does not neglect infinitely a sample of $\varepsilon(D)$. More formally, consider an infinite sequence of whole local searches from $\mathcal{LS}$: $\lambda_1, \lambda_2, \ldots$ Then, each element $s \in \varepsilon(D)$ appears infinitely many times in $r(\lambda_1), r(\lambda_2), \ldots$

A select function s is *fair* if it does not neglect infinitely a configuration of a whole local search. More formally, consider an infinite sequence of whole local searches from $\mathcal{LS}$: $\lambda_1, \lambda_2, \ldots$ such that $\lambda_1 \sqsubseteq \lambda_2 \sqsubseteq \ldots$. Then, each configuration $c \in C$ and $c \in \bigcup_i \lambda_i$ appears infinitely many times in the sequence $s(\lambda_1), s(\lambda_2), \ldots$

A move or neighborhood function $f^{(select,\sigma)}$ is *fair* if it relies on a fair *select* function. In the following, we consider only fair select, and restart functions, and thus, fair move and neighborhood functions.

4 Local Search as a Fixed Point of Reduction Functions

In our framework, local search will be described as a fixed point computation on the previously ordered structure.

4.1 Chaotic Iterations

K.R. Apt proposed chaotic iterations [3], a general theoretical framework for computing limits of iterations of a finite set of functions over a partially ordered set. In this paper, we do not recall all the theoretical results of K.R. Apt, but we just give the **GI** algorithm for computing fixed point of functions. Consider a finite set F of functions, and d an element of a partially ordered set $\mathcal{D}$.

GI: Generic Iteration Algorithm

$d := \bot$;
$G := F$;
While $G \neq \emptyset$ do
choose $g \in G$;
$G := G - \{g\}$;
$G := G \cup update(G, g, d)$;
$d := g(d)$;

where $\bot$ is the least element of the partial ordering $(\mathcal{D}, \sqsubseteq)$, G is the current set of functions still to be applied ($G \subseteq F$), and for all G, g, d the set of functions $update(G, g, d)$ from F is such that:

P1 $\{f \in F - G \mid f(d) = d \wedge f(g(d)) \neq g(d)\} \subseteq update(G, g, d)$.

P2 $g(d) = d$ implies that $update(G, g, d) = \emptyset$.

P3 $g(g(d)) \neq g(d)$ implies that $g \in update(G, g, d)$

Suppose that all functions in F are reduction functions as defined before and that $(\mathcal{D}, \sqsubseteq)$ is finite (note that finiteness is important as is has already been mentioned for our structure). Then, every execution of the **GI** algorithm terminates and computes in d the least common fixed point of the functions from F.

We now use the **GI** algorithm to compute the fixed point of our functions. The algorithm is thus feed with:

- a set of fair restart functions, fair move and neighborhood functions, that compose the set F,
- $\bot = \emptyset \in \mathcal{LS}$ to instantiate initial d,
- the ordering that we use is the ordering $\sqsubseteq$ on $\mathcal{LS}$.

Unfortunately, the properties P1,P2 and P3 required in [3] for the update functions (to insure the computation of the fixed point) do not fit our extended functions. Indeed, our extended functions are not deterministic and can modify a whole local search even if a previous application did not. This is due to the fact that the select function may choose a configuration which may not be modified by a move (or a neighborhood) function whereas another configuration would be modified by the same move (or neighborhood) function.

However, remember the fairness property of the select function: if one configuration can be changed by a move (or neighbor) function, then this configuration will not be neglected forever; and a sample will not be neglected forever by a restart function. Informally, the declarative definition of the update means:

P1 put in the $update(G, g, d)$ all functions not currently in G that will modify $g(d)$. This insures that all effective functions will be re-applied (correctness of the algorithm), whereas ineffective functions will not be added (efficiency reasons).

P2 to ensure termination.

P3 to add g again if it must be re-used.

Note that in practice, the update set is computed by a relaxation of the declarative definition. In our framework, the declarative definition of the update could be reformulated as follows to ensure correctness and termination. First, consider the notations: $\exists g(d) = d$ as a short cut for "there exists an element of the infinite sequence $g(d), g(d), \ldots$ which is equal to d". And similarly, $\forall g(d) = d$ is a short cut for "there does not exist an element of the infinite sequence $g(d), g(d), \ldots$ which is not equal to d". The update must satisfy the following properties:

P'1 $\{f \in F - G \mid \forall f(d) = d \wedge \exists f(g(d)) \neq g(d)\} \subseteq update(G, g, d)$.

P'2 $\forall g(d) = d$ implies that $update(G, g, d) = \emptyset$

P'3 $\exists g(g(d)) \neq g(d)$ implies that $g \in update(G, g, d)$

Basically, we need to put in the update set of functions, functions that potentially will modify d, the current whole local search.

In these conditions, the algorithm terminates and computes the least common fixed point of the functions from F, i.e., the result of the whole local search. Inspired by [3], the proof partially relies on an invariant $\forall f \in F - G, f(d) = d$ of the "while" loop in the algorithm. This invariant is preserved by our characterization of the update function (P'1, P'2 and P'3). Moreover, since we keep a finite partial ordering and a set of monotonic and inflationary functions, the results of K.R. Apt can be extended here. We characterize more precisely these fixed point results in the next section.

4.2 Characterization of the Results

We have described the basic computation structure and processes through the previous generic algorithm. We now define precisely the computed results w.r.t. the main goal of local search algorithms. Since this algorithm aims at optimizing an objective function in a CSP context, we first define an evaluation function to rate the quality of samples:

$$eval : \begin{array}{l} \varepsilon(D) \rightarrow \mathcal{E} \\ s \mapsto e \end{array}$$

where $(\mathcal{E}, >)$ is a totally ordered set, such that there exists an element $\perp$, $\forall e \in \mathcal{E}, e > \perp$. We require this evaluation to have the *solution recognition* property, i.e., $\forall s \in Sol_\varepsilon(D), eval(s) = \perp$.

We now consider a whole local search λ computed after applications of reduction functions. Its fundamental properties are related to the samples that have been explored during this computation. Therefore, we define the following function:

$$\begin{aligned}\Pi: \quad & \mathcal{LS} \rightarrow 2^{\varepsilon(D)} \\ & \lambda \mapsto \{s \in \varepsilon(D) \mid \exists (p, V) \in \lambda \text{ and } s \in p\}\end{aligned}$$

In fact this function is a projection of the computed paths to the set of samples in order to get all the samples which have been explored during search. We introduce then the ordering $\sqsubseteq_{Sol}$ to characterize quality of encountered solutions.

Definition *(Quality of solution)* Given λ and λ' in $\mathcal{LS}$, $\lambda \sqsubseteq_{Sol} \lambda'$ iff:

- either $min_{s \in \Pi(\lambda)}(eval(s)) > min_{s \in \Pi(\lambda')}(eval(s))$
- or $min_{s \in \Pi(\lambda)}(eval(s)) = min_{s \in \Pi(\lambda')}(eval(s))$ and $\lambda \sqsubseteq \lambda'$

where *min* refers to the minimum of a set of elements from $\mathcal{E}$ with respect to $>$.

Solutions are non ordered w.r.t. this ordering which can be useful to compare the results. Note that the reductions functions are compatible (i.e., monotonic and inflationnary) w.r.t. this ordering. On the other hand, the results obtained by the iteration algorithm are related to the parameters of the search.

4.3 Characterization of the Search

We discuss here the influence of the two parameters σ and ρ on local search.

$\rho \geq |\varepsilon(D)|$ In this case, thanks to the fairness property of the restart functions, all the search space can be explored. We may thus obtain a complete solver, but still practically unconceivable.

$\rho < |\varepsilon(D)|$ In this case, to obtain a complete solver, we must impose some more properties on move and neighborhood functions:

- a move function must not loop, i.e., a path must be a sequence of samples $s_1, s_2, \ldots, s_m$ with $m \leq \sigma$ s.t. there does not exist $i, j \in [1..n]$ with $s_i = s_j$.
- neighborhood functions must be total, i.e., from a sample, each sample of $\varepsilon(D)$ must be reachable by at least one of the used neighborhood functions.
- $\sigma \geq |\varepsilon(D)|$.

$\rho << |\varepsilon(D)|$ **and** $\sigma << |\varepsilon(D)|$ The previous cases, although interesting from a theoretical point of view (they help understanding why local search techniques are not considered as complete methods) are inconceivable in practice.

In this case, the search is not complete, and the quality of solution will depend on the quality of the move, neighborhood, and restart functions. This justify all the works that have conducted to improve local search by designing new algorithms: in our framework, this translate to new neighborhoods, new form of move and restart, and new strategies. Note that these consideration could be extended according to the requirements of the search : find a solution, find all solution or optimization problems. Our framework, allowing to handle several paths could be for instance interesting if one wants to extract some different solutions of equivalent quality.

5 Using the Framework

In this work, possible uses of this framework are illustrated through the description of existing strategies such as descent algorithms (WalkSat) and tabu search [8]. Concerning LS methods we focus on Tabu search (TS) [6]. Basically, this algorithm forbids moving to a sample that was visited less than l steps before. To this end, the list of the last l visited samples is memorized. On the other hand, we consider a basic descent technique with random walks RW where random moves are performed according to a certain probability p. According to our model, we only have now to design functions of the generic algorithm of Section 4.1 to model strategies. Neighborhood functions are functions $C \rightarrow C$ such that $(p, V) \mapsto (p, V \cup V')$ with different conditions:

$$\begin{aligned}FullNeighbor: \quad & V' = \{s \in D | s \notin V\} \\ TabuNeighbor: \quad & V' = \{s \in D \mid \nexists k, \\ & \quad n - l \leq k \leq n, s_k = s\} \\ DescentNeighbor: \quad & p = (s_1, \ldots, s_n) \text{ and} \\ & V' = s \subset D \text{ s.t. } \nexists s' \in V \\ & \text{s.t. } eval(s') < eval(s_n)\end{aligned}$$

Move functions are functions $C \rightarrow C$
s.t. $(p, V) \mapsto (p', \emptyset)$ with various conditions:

$$\begin{aligned}BestMove: \quad & p' = p \oplus s' \text{ and} \\ & eval(s') = min_{s'' \in V} eval(s''). \\ ImproveMove: \quad & p = p'' \oplus s_n \text{ and} \\ & p' = p \oplus s \text{ s.t. } eval(s') < eval(s_n). \\ RandomMove: \quad & p' = p \oplus s' \text{ and } s' \in V.\end{aligned}$$

We can precise here the input set of function F for algorithm GI:

$$\begin{aligned}
Tabusearch: & \{TabuNeighbor \\
& ; BestNeighBor\} \\
Randomwalk: & \{FullNeighbor \\
& ; BestNeighBor; \\
& RandomNeighbor\} \\
TabuSearch + Descent: & \{TabuNeighbor \\
& ; DescentNeighbor \\
& ; ImproveNeighBor \\
& ; BestNeighBor\} \\
Randomwalk + Descent: & \{FullNeighbor \\
& ; BestNeighBor \\
& ; RandomNeighbor \\
& ; DescentNeighbor \\
& ; ImproveNeighBor\}
\end{aligned}$$

The different algorithms correspond to different sets of input functions and different behaviours of the *choose* function in the GI algorithm. *choose* alternatively selects neighborhood and move functions. For the Random Walk algorithm, given a probability parameter p, we have to introduce a quota of p $BestMove$ functions and $1-p$ $RandomMove$ used in GI. Concerning Tabu Search, we use here a $TabuNeighbor$ with $l = 10$ and $BestMove$ functions to built our Tabu Search algorithm. At last, we combine a descent strategy by adding $DescentNeighbor$ and $ImproveMove$ to the previous sets in order to design algorithms in which a Descent is first applied in order to reach more quickly a good configuration.

Thus we design various algorithms in a single generic algorithm: this is not so easy and clear when the methods are considered from a pure algorithmic point of view. [1]

6 Conclusion

In this paper, we have proposed a framework for modeling CSP resolution with local search techniques. This framework provides a computational model as the computation of a fixed point of functions over a partial ordering, inspired the initial works of K.R. Apt [3]. It helps us to finer define the basic processes of local search at a uniform description level and to describe specific search strategies. This mathematical framework could be helpful for the design of new local search algorithms, the improvement of existing ones and their combinations.

This framework could be extended in order to include complete resolution mechanisms (constraint propagation, domain splitting) and even other metaheuristics such as evolutionary algorithms as in [10]. However in [10], granularity of functions is more coarse grain than in the framework we propose here: move and neighbor functions are integrated in one function; restart is implicit and induced by split functions. Thus [10] cannot consider several move, neighborhood, or restart and this is thus also much more restrictive in terms of strategies.

[1] Due to space limitations we did not include our experimental results on different instances of Sudoku problem.

References

[1] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[2] K. R. Apt. From chaotic iteration to constraint propagation. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 36–55. Springer, 1997.

[3] K. R. Apt. *Principles of Constraint Programming*. Cambridge Univ. Press, 2003.

[4] B. Crawford, C. Castro, and E. Monfroy. Integration of constraint programming and metaheuristics. In I. Miguel and W. Ruml, editors, *SARA*, volume 4612 of *Lecture Notes in Computer Science*, pages 397–398. Springer, 2007.

[5] F. Focacci, F. Laburthe, and A. Lodi, editors. *Local Search and Constraint Programming. In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics, volume 57 of International Series in Operations Research and Management Science*. Kluwer Academic Publishers, Norwell, MA, 2002.

[6] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

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

[8] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the satisfiability problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:457–478, 1996.

[9] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1):21–45, 2002.

[10] E. Monfroy, F. Saubion, and T. Lambert. Hybrid csp solving. In B. Gramlich, editor, *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 138–167. Springer, 2005.

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