Specifying Local Search Neighborhoods from a Constraint Satisfaction Problem Structure

Mateusz Ślażyński[1], Salvador Abreu[2], and Grzegorz J. Nalepa[1]

[1] AGH University of Science and Technology, Krakow, Poland
{mslaz, gjn}@agh.edu.pl
[2] University of Évora and LISP, Évora, Portugal
spa@uevora.pt

Abstract Neighborhood operators play a crucial role in defining effective Local Search solvers, allowing one to limit the explored search space and prune the fitness landscape. Still, there is no accepted formal representation of such operators: they are usually modeled as algorithms in a procedural language, lacking in compositionality and readability. In this paper we propose a new formalization capable of representing several neighborhood operators, thereby eschewing the need to code these in a full Turing complete language. The expressiveness of our proposal stems from a rich problem representation, as used in Constraint Programming models. We compare our system to competing approaches and show a clear increment in expressiveness.

1 Introduction

Local Search is a family of heuristic algorithms, typically used to find approximate solutions for hard optimization problems. The common core of those methods consists in finding an initial sub-optimal solution (called configuration in the rest of the paper) and iteratively replacing it with similar configurations, called neighbors. There is a vast body of research on the exploration methods, called metaheuristics, focused mostly on escaping local optima and finding novel configurations. While metaheuristics vary in their performance, they are generic enough to be applicable to many different problems. Various algorithmic configuration methods can be used to tune their parameters without human interaction.

Another popular method of improving Local Search performance is to define a configuration's neighborhood in a way that prunes out the not interesting (e.g. incorrect) configurations. In this paper, the function which maps a configuration to its neighborhoods is called a neighborhood operator. In contrast to metaheuristics, the most efficient neighborhood operators are problem-specific and have to be defined and tuned by hand, often directly in the form of a computer program. Such an approach suffers from poor reusability and does not generalize well to new domains. A brief revision of the state of the art on currently used methods will be presented in Section 2.

The goal of this paper is to present in full detail a novel representation of the neighborhood operators, one which exploits the structure of the problem under consideration Such formal representation will lay a theoretical foundation for the implementation of the high-level Local Search models, suitable for automated processing and synthesis. In Section 3 we will define the problem's structure in terms of constraints and variables, as in the Constraint Programming paradigm. Subsequently, the Neighborhood Definition Language (or NDL) will be presented in Section 4. To show the expressiveness of the formalism, a set of non-trivial neighborhoods will be defined in Section 5.

2 Context

Frequently, Local Search solvers are implemented in an imperative programming language without any modeling layer. To facilitate code reuse, several programming libraries were introduced, such as JAMES [1], Local++ [2] or EasyLocal++ [3]. Neighborhood operators are represented as a class or function satisfying a predefined interface. OptaPlanner [4] is a hybrid solver configurable by means of XML files. It supports several search strategies, including local search, and allows the definition of the neighborhood in a declarative manner as a composition of basic moves. There are several predefined moves applicable to common optimization problems, but to add new ones, one has to implement them in the Java language.

At the same time, the Constraint Programming and Mathematical Optimization communities have been concerned with problem modeling and human readable representations. A Mathematical Programming Language (AMPL) [5] is a modeling language closely resembling algebraic notation that can be easily translated to internal representation of various mathematical solvers. The Optimization Programming Language (OPL) [6] provides a more generic representation supporting both mathematical and constraint programming tools. Recently various Constraint Programming languages have been proposed, such as Essence [7] that incorporates high-level mathematical types or MiniZinc [8] that is, by design, solver agnostic thanks to an intermediate low level representation called FlatZinc. These tools enable the user to express a problem structure in a way as close as possible to their natural form.

Localizer [9] is a language designed to model the whole local search routine in one domain-specific modeling language. The Comet [10] language extended the Localizer approach by adding mechanisms to independently express both the problem structure and the search strategy in terms of constraint programming models. It was possible to define simple neighborhoods in an imperative manner, by selecting variables and their new values according to various heuristics. While the Comet language is not supported anymore, systems such as OscaR [11] provide very similar capabilities. Neighborhood combinators [12] extend OscaR with a declarative language used to define search strategies by combining existing neighborhood operators. The neighborhood operators themselves still have to be implemented in the Scala language. A similar, although less extendable, proposal has been also made for the MiniZinc modeling language [13].

Some authors [15,16] have proposed to define neighborhood operators as separate Constraint Programming problems with constraints corresponding to relations between the neighbors. Declarative Neighborhoods [14] implement this approach by extending MiniZinc with neighborhood operators defined in a declarative manner. Such a representation integrates well into a Constraint Programming model, allowing one to put additional requirements on the neighborhood relation, i.e. every neighbor has to satisfy a set of constraints or that the neighborhood operator requires some constraints to be satisfied before it may be applied. It has been shown [17] that a Constraint Programming solver may be effectively used to explore a local search neighborhood defined in this manner.

Recently, there has been a proposal for a system inferring possible neighborhood operators from variable types occurring in the Constraint Programming model, defined in the Essence language [18, 19]. Various variable types are connected with relevant move operators that are tried in an intelligent manner using classic multi-arm bandit strategies. The resulting solver proved to be superior to or competitive with popular generic strategies on several common combinatorial problems. As the authors focused more on exploiting basic moves in an unsupervised manner, the representation is composed of predefined operators and does not allow one to define arbitrary, well known complex neighborhood operators.

This paper expands on ideas stated briefly in the recently presented poster [20]. The new contributions compared to the past presentation consist in a formally defined semantics of the proposed language along with the selection of detailed examples, showing its expressiveness and usefulness for common optimization problems. The experimental works [21] have shown, that Genetic Programming methods combined with the NDL, are able to synthesize various Traveling Salesperson neighborhoods based only on the classical Constraint Programming model. The system used in the experiments contains a prototype runtime for NDL implemented as a Prolog Domain Specific Language, called Noodle[3].

3 Representation of the Problem Structure

A Constraint Satisfaction Problem (CSP) is classically defined [22] as a triple $P = (X, D, C)$, where X is an n-tuple of variables $X = (x_1, x_2, \dots, x_n)$, D is a n-tuple of corresponding domains $D = (D_1, D_2, \dots, D_n)$ such that $D_i \subset \mathbb{Z}$[4] and $x_i \in D_i$, C is an m-tuple of constraints $C = (C_1, C_2, \dots, C_m)$. In the general case, a k-ary constraint $C_j = \left(R_{S_j}, S_j\right)$, where S_j is called the scope of the constraint and is the result of projecting X onto a k-tuple of variables. R_{S_j} is a subset of the cartesian product of the domains of variables in S_j. In this paper the general definition will be restricted, without loss of generality, to strictly *binary scopes*, so the problem structure can be expressed as a graph.

[3] See: `https://gitlab.geist.re/pro/ndl`

[4] This is for the case of constraints over finite domains (FD).

3.1 Typing

We extend the classical representation of a CSP problem by adding *type annotations* which represent the syntactic structure of the problems, as found in modern modeling languages such as MiniZinc. Types correspond to sets of interesting entities composing the CSP problem.

There are two kinds of types in our formalization, *variable types* T and *constraint types* U. The former express indexed data structures (arrays) used in the model to group together similar variables, e.g. an array of variables corresponding to the queens' positions in the n-queens problem. Every variable type T_i is a pair (t_i, J_i) where t_i is a type identifier (name of the array) and $J_i \subseteq \mathbb{Z}^d$ is an index set addressing the corresponding d-dimensional array.

Another common abstraction used in constraint programming languages is the ability to define constraints by quantifying over the variables. Constraints defined this way share not only semantics, but also structural meaning, acting together as a kind of global constraint. To exploit this similarity, we assign a type $u_i \in U$ to every constraint, in such a way that constraints created via the same quantifier or global constraint share the same type. It is also expected of them to share types for variables falling under their scopes.

Given those definitions, a typed Constraint program is represented by a 6-tuple: $P_T = (X, D, C, T, I, U)$ where X, D and C are as previously seen, and $T = (T_1, T_2, \dots, T_n)$ is an n-tuple of types corresponding to the variables, I is an n-tuple containing addressing information about the corresponding variables $I = (i_1, i_2, \dots i_n)$ such that $i_j \in J_j$. Indexing I is correct if no variables of the same type share the same index: $\forall i, k \in 1..n, T_i = T_k \implies i_i \neq i_k$. U is an m-tuple of constraint types $U = (u_1, u_2, \dots, u_n)$.

Example 1. The 4-queens problem can be represented as $P_T = (X, D, C, T, I, U)$, where:

- $X = (x_1, x_2, x_3, x_4)$ as there are four queens and x_i represent the row taken by queen in the i-th column.
- $D = \{D_Q\}^4$ such that all variables share the same domain $D_Q = \{1, 2, 3, 4\}$.
- $C = (C_{R1}, C_{R2}, \dots, C_{R6}, C_{D1}, C_{D2}, \dots, C_{D6})$ where the C_R constraints stand for inequality (queens do not share rows) and C_D indicate queens not sharing the same diagonal on the board. We will skip enumerating scopes and allowed valuations due to space constraints.
- $T = \{T_Q\}^4$ — all the variables share the same type $T_Q = (\eta, \{1, 2, 3, 4\})$, where η is an arbitrary selected type identifier.
- $I = (1, 2, 3, 4)$.
- $U = \{u_r\}^6 * \{u_d\}^6$, as row and diagonal inequality constraints are assigned different types. The symbol $*$ here stands for concatenation of the tuples.

3.2 Typed Constraint Network

In order to represent the structure of an annotated CSP $P_T = (X, D, C, T, I, U)$, we introduce the Typed Constraint Network (TCN) — a labeled directed multigraph $G_T = (V, A, s, e, \Sigma_V, \Sigma_A, l_V, l_A)$ where:

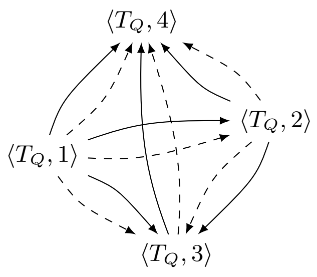


Figure 1: TCN of a 4-queens problem. Solid and dashed arcs represent constraints of type u_r and u_d respectively.

– V is a set of vertices $\{v_1, v_2, \dots, v_n\}$ representing the variables $v_j = \pi_j(X)$.
– A is a set of arcs $\{a_1, a_2, \dots, a_m\}$ representing the constraints $a_j = \pi_j(C)$.
– $s\colon A \to V$ and $e\colon A \to V$ map arcs respectively to their starting and terminal (ending) vertices. Both are projections of the corresponding constraint's scope such that $s(a_j) = \pi_1(S_j)$ and $e(a_j) = \pi_2(S_j)$. The directed nature allows arcs to mirror order in the corresponding constraint scope;
– Σ_V is a set of vertex labels $\{\Sigma_{V_1}, \Sigma_{V_2}, \dots, \Sigma_{V_n}\}$ where label is a type of the corresponding variable and its index $\Sigma_{V_j} = (T_j, i_j)$.
– $\Sigma_A = \{\sigma_{a_1}, \sigma_{a_2}, \dots, \sigma_{a_l}\}$ is set of all elements in U.
– $l_V : V \to \Sigma_V$ and $l_A : A \to \Sigma_A$ are the labeling functions.

Figure 1 shows the TCN for Example 1. We argue that such a graph represents the syntactic structure of the CP model and may be used to define useful problem-specific neighborhood operators.

4 Neighborhood Definition Language

Given an annotated CSP P_T and corresponding TCN G_T, the neighborhood specific to the problem is a relation $N \subseteq \Gamma \times \Gamma$, where Γ is the configuration space $\Gamma = D_1 \times D_2 \times \cdots \times D_n$. In order to define the operator in a fine-grained constructive manner we propose its decomposition into a set of basic operators and combinators. To ease notation, we will assume that every operator is problem specific and both P_T and G_T will not be stated explicitly in their domains.

4.1 Selectors

Selectors $\mathbb{S} = \Phi \cup \Omega \cup \Psi$ provide a way to *select* a single constraint, variable or value, based on the problem's structure and a given configuration. Every selector denotes a set of entities which meet the requirements and can be regarded as a non-deterministic choice function. Given that such a set may be empty, selectors are *not* total functions and, in the undefined cases, we will say that selector *fails.*

Constraint Selectors Pick one *arc* from the Typed Constraint Network and constitute a family of operators $\Phi = \{\phi_\Sigma, \phi_{\Sigma\times s}, \phi_{\Sigma\times e}\}$, where:

- $\phi_\Sigma \colon \Sigma_A \to A$ picks an arc with a label: $\phi_\Sigma(\sigma_{a_j}) \in \{a_k \in A \colon l_A(a_k) = \sigma_{a_j}\}$
- $\phi_{\Sigma\times s} \colon \Sigma_A \times V \to A$ picks an arc of type a_j and starting vertex v_s: $\phi_{\Sigma\times s}(\sigma_{a_j}, v_s) \in \{a_k \in A \colon l_A(a_k) = \sigma_{a_j} \wedge s(a_k) = v_s\}$
- $\phi_{\Sigma\times e} \colon \Sigma_A \times V \to A$ analogous to $\phi_{\Sigma\times s}$, but uses an ending vertex.

Variable Selectors Pick single vertex from the TCN and constitute family $\Omega = \{\omega_T, \omega_I, \omega_s, \omega_e\}$, where:

- $\omega_T \colon T \to V$ picks vertex of type matching the argument $\omega_T(T_j) \in \{v_i \in V \colon \pi_1(l_V(v_i)) = T_j\}$.
- $\omega_I \colon I \times T \to V$ picks vertex with given index and variable type matching the arguments $\omega_T(i_x, T_j) \in \{v_i \in V \colon \pi_1(l_V(v_i)) = T_j \wedge \pi_2(l_V(v_i)) = i_x\}$.
- $\omega_X \colon \Gamma \times \mathbb{Z} \times T \to V$ picks a vertex with a given value and variable type matching the arguments $\omega_T(\gamma, x, T_j) \in \{v_i \in V \colon \pi_1(l_V(v_i)) = T_j \wedge \psi_{v_i}(\gamma) = x\}$, where ψ_{v_i} is defined in the next paragraph.
- $\omega_s \colon A \to V$ and $\omega_e \colon A \to V$ pick starting and ending vertices of the arc: $\omega_s(a_j) = s(a_j)$ and $\omega_e(a_j) = e(a_j)$.

Value Selectors Pick a value based on the problem structure and configuration $\Psi = \Psi_D \cup \Psi_J \cup \Psi_V$, where:

- Ψ_D and Ψ_I are sets of choice functions $\psi_{D_j} \in D_j$ and $\psi_{J_i} \in J_i$ for every domain and index set in the given problem.
- Ψ_V is a set of functions $\psi_{v_i} \colon \Gamma \to D_i$ for every vertex v_i in the Typed Constraint Network, where D_i is domain of the corresponding variable. ψ_{v_i} maps vertices to their assigned values in the configuration $\psi_{v_i}(\gamma) = \pi_i(\gamma)$.

4.2 Selector Combinators

While selectors are designed to pick single elements, there is often the need to collect several inter-related elements, e.g. variables sharing the same value, the sum of two other values, etc. To achieve this, NDL is equipped with two types of basic logical and mathematical operators that can combine the selectors' results:

- Filters $F = \{f_=, f_\neq, f_<, f_\leq\}$ are functions that pass the first argument unchanged only if it satisfies a simple constraint, e.g. $f_= \colon \Sigma \times \Sigma \to \Sigma$ is an equality filter defined on the NDL alphabet:

$$f_=(s_1, s_2) = \begin{cases} s_1 = s_2, & s_1 \\ \text{otherwise} & \text{is undefined (fails)} \end{cases}$$

  $f_< \colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $f_\leq \colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ are analogous filters that check for inequality over their arguments.
- Arithmetic combinators $\Lambda = \{\lambda_+, \lambda_-, \lambda_\times, \ldots\}$ correspond to arithmetic functions, e.g. $\lambda_+(z_1, z_2) = z_1 + z_2$.

4.3 Modifiers

The main goal of the neighborhood operator is to modify a given configuration. *Modifiers* create a new configuration γ_N by slightly perturbing a given configuration γ_S. There are three operators in this family $M = \{\mu_s, \mu_\alpha, \mu_f\}$:

- $\mu_\alpha \colon \Gamma \times V \times \mathbb{Z} \rightarrow \Gamma$ is an partial function, that creates a new configuration with a value reassigned to the given variable. $(\forall v_i \in V, v_i \neq v_x \implies \pi_i(\mu_\alpha(\gamma_S, v_x, x)) = \pi_i(\gamma_S)) \wedge \pi_j(\mu_\alpha(\gamma_S, v_x, x)) = x$. The result is undefined if the value given does not belong to the corresponding domain.
- $\mu_s \colon \Gamma \times V \times V \rightarrow \Gamma$ swaps values of two variables. It can be defined inside NDL as $\mu_s(\gamma_s, v_i, v_j) = \mu_\alpha((\mu_\alpha(\gamma_s, v_j, \psi_{v_i}(\gamma_s)), v_i, \psi_{v_j}(\gamma_s))$.
- $\mu_f \colon \Gamma \times V \times \mathbb{Z} \times \mathbb{Z} \rightarrow \Gamma$ creates a new configuration by "flipping" a value:

$$\mu_f(\gamma, v_i, x_1, x_2) = \begin{cases} \psi_{v_i}(\gamma) = x_1, & \mu_\alpha(\gamma, v_i, x_2) \\ \psi_{v_i}(\gamma) = x_2, & \mu_\alpha(\gamma, v_i, x_1) \\ \text{otherwise} & \text{is undefined (fails)} \end{cases}$$

4.4 Move

A single move $\alpha : \Gamma \rightarrow \Gamma$ is the most basic neighborhood operator defined in the NDL language. It is a total function, defined as the composition of a fixed number of operators $O_\alpha = \{o_1, o_2, \ldots o_k\}$, where at least one of them is a modifier.

In case any of the intermediate results is undefined (one of the operator fails), we define α to be an identity function $\mathbb{I} \colon \Gamma \rightarrow \Gamma$, so it was total.

In order to avoid redundant repetitions and simplify notation, we will express composition of operations via conjunction, i.e. $\alpha(\gamma_s) = o_1 \circ o_2 \circ \ldots \circ o_k$ will be written as $\alpha(\gamma_s) = \gamma_n \Leftrightarrow r_k = o_k(\gamma_s) \wedge \ldots \wedge r_2 = o_2(\ldots) \wedge \gamma_n = o_1(r_2)$. The same applies to the selector combinators, instead of writing $x_2 = f_=(x_2, x_1)$, we will use the infix notation $x_2 = x_1$.

4.5 Move Combinators

In order to express neighborhood operators that perturb the configuration in a dynamic – possibly recursive – manner, we introduce move combinators that apply the specified move to the Typed Constraint Graph in a systematic way. Move combinators can be perceived as domain-specific recursion schemes known from category theory and functional programming [23]. We will use a functional notation to abstract the unbounded variables in the moves, so they could be applied to different parts of the graph. In terms of expressiveness, combinators belong to primitive recursive functions.

The Universal Selector Quantifier Quantifies over the results of a selector β, composing them with a given move α. $\forall_s(\gamma_s, \beta, \alpha) = \forall(\rho_i \in \beta(\gamma_s)), \alpha(\gamma_s, \rho_1) \wedge \gamma_3 = \alpha(\gamma_2, \rho_2)) \wedge \ldots \wedge \gamma_m = \alpha(\gamma_{m-1}, \rho_{m-1}))$, where γ_m is the resulting configuration. This combinator may be used to perform moves that modify many variables at once, e.g. swap rows of an array.

The Least Fixpoint Operator Performs a primitive recursion over the structure of the TCN. The fixpoint operator $\mho\colon \mathit{\Gamma} \times \alpha \times v_s \rightarrow \mathit{\Gamma}$ applies move $\alpha : \mathit{\Gamma} \times A \rightarrow \mathit{\Gamma}$ over arcs of the TCN spanning tree rooted at vertex v_s explored in a breadth first search order. If the move applied at a given arc does not change the configuration, the corresponding branch of the spanning tree is pruned. As the spanning tree is finite, such an operator is bound to terminate.

4.6 Neighborhood

As introduced in the beginning of the section, neighborhood operators in NDL are defined by means of functional composition and specific combination of basic operators: selectors and modifiers. An NDL formula explicitly defines only a single neighbor of given configuration. As some operators, e.g. selectors, are non-deterministic, in order to define the neighborhood relation, we have to quantify over all their possible results, i.e. corresponding sets of entities (this does not apply to selectors that already fall in scope of an Universal Selector Quantifier.)

The order of neighbors in such a neighborhood is not defined by NDL itself: this is left to the Local Search metaheuristic. Some of these, e.g. Simulated Annealing, may want to sample the neighborhood in a stochastic manner, while others will explore it exhaustively, or even order the neighbors according to some problem-specific heuristics.

5 Example Use Cases

In this section we will explore several neighborhood operators, and their NDL specification. The examples were chosen based on both their popularity and for their suitability to showcase the expressiveness of the language.

5.1 Kempe Chain Neighborhood

Kempe chain is an efficient neighborhood operator applicable in the graph coloring problem proposed in [24]. Given a graph $G = (V_G, E_G)$, the task is to find the minimal vertex labeling (coloring) $(\mathit{\Sigma}_C, l_c)$ that assigns different labels (colors) to the adjacent vertices $\forall\{v_1, v_2 \in E_G\}, l_c(v_1) \neq l_c(v_2)$. A coloring satisfying this property is called admissible.

The idea behind Kempe chain is to perturb an admissible coloring in such a way that admissibility is preserved — in other words, admissibility is an invariant of this neighborhood operator. Given an admissible coloring $(\mathit{\Sigma}_{c_1}, l_{c_1})$, a Kempe chain can be described in four steps:

1. Pick two vertices $v_1, v_2 \in V_G$ such that $l_{c_1}(v_1) = l_1 \wedge l_{c_1}(v_2) = l_2 \wedge l_1 \neq l_2$.
2. Create a new graph K by removing from G all vertices with coloring other than l_1 or l_2.
3. Choose a component K_{v_1} in K containing v_1.
4. Change colors of all vertices in K_{v_1} by replacing l_1 with l_2, and vice versa.

The resulting coloring is admissible and does not use more colors than $\mathit{\Sigma} c_1$.

$$\begin{tikzcd}
\langle T_C, 1\rangle \arrow[r, "u"] \arrow[d, "u"] \arrow[dr, "u"] & \langle T_C, 2\rangle \arrow[d, "u"] \arrow[dr, "u"] & \langle T_C, 3\rangle \arrow[dl, "u"'] \arrow[d, "u"] \\
\langle T_C, 6\rangle & \langle T_C, 5\rangle \arrow[l, "u"'] & \langle T_C, 4\rangle
\end{tikzcd}$$

Figure 2: TCN representing 6-nodes graph coloring problem.

CSP Model. A graph coloring problem with n vertices and m edges is represented as $P_T = (X, D, C, T, I, U)$, where:

– $X = (v_1, v_2, \ldots, v_n)$ such that every $v_i \in V_G$.
– $D = \{D_C\}^n$ such that $D_C = \{1, 2, \ldots, n\}$.
– $T = \{T_C\}^n$ such that $T_C = (\eta, \{1, 2, \ldots, n\})$.
– $I = (1, 2, \ldots, n)$.
– $C = (C_1, C_2, \ldots C_m)$ where $C_j = \big(R_{S_j}, (v_k, v_l)\big)$ such that $v_k, v_l \in E_G$.
– $U = \{u\}^m$.

A corresponding TCN for a basic 6-nodes problem is shown in Figure 2.

NDL Operator Encoding of the Kempe chain neighborhood makes heavy use of the least fixpoint operator $\mho$ and μ_f (flip) modifier:

We say that configuration γ_n is neighbor of γ_s:

$$N_C(\gamma_s) = \gamma_n \Leftrightarrow$$

when v_x and v_y are graph nodes:

$$v_x = \omega_T(T_C) \wedge v_y = \omega_T(T_C)$$

l_x and l_y are their corresponding colors:

$$\wedge\, l_x = \psi_{v_x}(\gamma_s) \wedge l_y = \psi_{v_y}(\gamma_s)$$

such that l_x and l_y are different colors:

$$\wedge\, l_1 \neq l_x$$

the color of v_x is “flipped” from l_x to l_y:

$$\wedge\, \gamma_1 = \mu_f(\gamma_s, v_x, l_x, l_y)$$

and the Kempe chain starts from node v_x:

$$\wedge\, \gamma_n = \mho(\gamma_s, \alpha, v_x)$$

α describes color changes in its adjacency:

$$\alpha(\gamma_k, a_\mho) = \gamma_{k+1} \Leftrightarrow$$

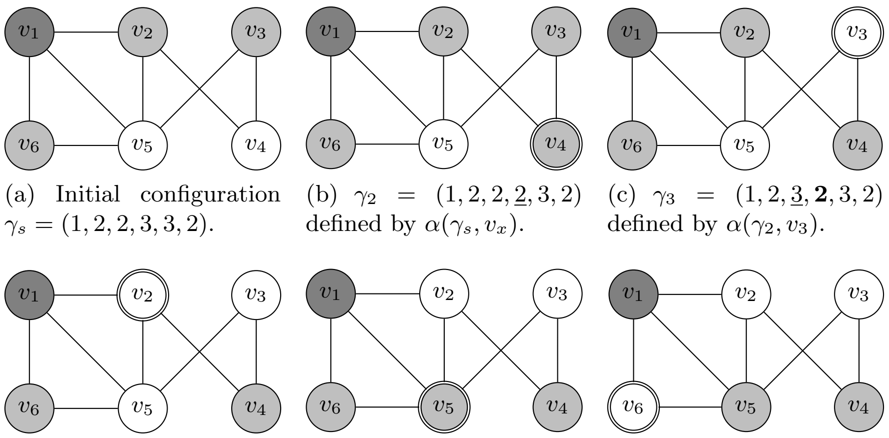
node $v_\mho$ is adjacent to the previously modified node:

$$v_\mho = \omega_e(a_\mho)$$

its color is flipped too — when the color is neither l_x nor l_y, flipping fails and we finish exploring this branch of the chain, otherwise we continue:

$$\wedge\, \gamma_{k+1} = \mu_f(\gamma_k, v_{\mho}, l_x, l_y)$$

An example of a neighborhood induced by this operator is shown in Figure 3.



(a) Initial configuration $\gamma_s = (1, 2, 2, 3, 3, 2)$.

(b) $\gamma_2 = (1, 2, 2, \underline{2}, 3, 2)$ defined by $\alpha(\gamma_s, v_x)$.

(c) $\gamma_3 = (1, 2, \underline{3}, \mathbf{2}, 3, 2)$ defined by $\alpha(\gamma_2, v_3)$.

(d) $\gamma_4 = (1, \underline{3}, \mathbf{3}, \mathbf{2}, 3, 2)$ defined by $\alpha(\gamma_3, v_2)$.

(e) $\gamma_5 = (1, \mathbf{3}, \mathbf{3}, \mathbf{2}, \underline{2}, 2)$ defined by $\alpha(\gamma_4, v_5)$.

(f) $\gamma_n = (1, \mathbf{3}, \mathbf{3}, \mathbf{2}, \mathbf{2}, \underline{3})$ defined by $\alpha(\gamma_6, v_6)$.

Figure 3: Configurations involved in the Kempe Chain move at nodes v_4, v_6.

5.2 Column Swap

A common pattern in the constraint programming is to group variables into arrays mirroring the problem structure, e.g. in job scheduling problems, column index in a two dimensional array corresponds to the moment of time and row index represents a specific machine/worker. This example will show how to exploit such structure to define neighborhoods.

The Traveling Tournament [25] problem consists in scheduling a tournament, given n teams, distances between their home towns and $n-1$ rounds. Every team plays only once per round and has to face every other team exactly once. The goal is to minimize the total distance traveled during the tournament.

There are many different neighborhood operators applicable to this problem [26], leading to efficient optimization routines. Most of them are based on local swap moves, either swapping rounds or opponents. More advanced operators swap only parts of the rounds and then fix the configuration in a way analogous to Kempe chaining.

Model. The traveling tournament problem with n teams can be represented as $P_T = (X, D, C, T, I, U)$, where:

- $X = \left(v_{1-1}, v_{1-2}, \ldots, v_{1-(n-1)}, v_{2-1}, \ldots, v_{n-(n-1)}\right)$ such that v_{i-j} represents opponent of the i-th team in the j-th round.
- $D = \{D_1\}^{n-1} * \{D_2\}^{n-1} * \cdots * \{D_n\}^{n-1}$ such that $D_i = \{1, 2, \ldots, n\} \setminus i$, so no team could play with itself.
- $T = \{T_O\}^n$ such that index set cover all indexes in the 2-d array of size $n \times n-1$: $T_O = (\tau, \{1, 2, \ldots n\} \times \{1, 2, \ldots n-1\})$.
- $I = ((1,1), (1,2), \ldots, (1, n-1), (2,1), \ldots (n, n-1))$.
- $C = \{C_R\}^{n(n-1)(n-2)} * \{C_O\}^{n(n-1)^2} * \{C_S\}^{n(n-1)^2}$, where C_R constrains variables to have different values in every round (in the column); C_O constrains variables to have different values for any team (in the array row); C_S constrains variables to be symmetrical: if team A plays with team B, then B plays with A in the same round.
- $U = \{u_R\}^{n(n-1)(n-2)} * \{u_O\}^{n(n-1)^2} * \{u_S\}^{n(n-1)^2}$.

NDL Operator. The following basic operator swaps two rounds, first by finding their columns' indexes, then by quantifying over all variables in the first column and swapping their values with corresponding variables from the second column.

$$N_S(\gamma_s) = \gamma_n \Leftrightarrow a_x = \phi_\Sigma(u_R) \wedge v_s = \omega_s(a_x) \wedge v_e = \omega_e(a_x)$$
$$\wedge (\tau, (i_{sc}, i_{sr})) = l_V(v_s) \wedge (\tau, (i_{ec}, i_{er})) = l_V(v_e) \wedge \gamma_n = \forall_s(\gamma_s, \beta, \alpha)$$

where β selects from two columns i_{sc} and i_{ec} two matches in the same row:

$$\beta \Leftrightarrow v_l = \omega_T(\tau) \wedge (\tau, (i_{lc}, i_{lr})) = l_V(v_l) \wedge i_{lc} = i_{sc} \wedge v_r = \omega_I((i_{ec}, i_{lr}), \tau)$$

and α swaps them in the configuration:

$$\alpha(\gamma_k, v_l, v_r) = \gamma_{k+1} \Leftrightarrow \gamma_{k+1} = \mu_s(\gamma_k, v_l, v_r)$$

Figure 4 presents a sequence of configurations defined by this operator.

5.3 2-opt Neighborhood

The Traveling Salesperson Problem (TSP) is a discrete optimization problem of finding a Hamiltonian Cycle with the least weight in the complete weighted graph. There is a well known family of efficient neighborhood operators applicable to this problem, named 2-opt, 3-opt, etc. corresponding to the number of changes introduced to the configuration. An m-opt operator removes m edges from the current configuration (cycle) and replaces them with m new edges. It is worth noting that for $m = 2$ there exists only one way to reassemble the cycle, but for $m > 2$, there exist $(m-1)! \times 2^{m-1}$ distinct results. Also the moves with $m > 2$ can be replaced by the repeated application of the 2-opt operator, i.e. 3-opt can be seen as at most three consecutive applications of the 2-opt.

The basic representation of the TSP configuration with n vertices is an n-tuple containing the vertices in the visiting order , i.e. $(v_1, v_4, v_3, v_5, v_2)$ represents a cycle composed of edges $\{v_1, v_4\}$, $\{v_4, v_3\}$, $\{v_3, v_5\}$, $\{v_5, v_2\}$ and $\{v_2, v_1\}$.

	1_R	2_R	3_R
1_T	2_T	3_T	4_T
2_T	1_T	4_T	3_T
3_T	4_T	1_T	2_T
4_T	3_T	2_T	1_T

(a) $\gamma_s = \begin{array}{l}(2,\ 3,\ 4,\\ 1,\ 4,\ 3,\\ 4,\ 1,\ 2,\\ 3,\ 2,\ 1)\end{array}.$

	1_R	2_R	3_R
1_T	$\mathbf{2_T}$	3_T	$\mathbf{4_T}$
2_T	1_T	4_T	3_T
3_T	4_T	1_T	2_T
4_T	3_T	2_T	1_T

(b) before $\alpha(\gamma_s, v_{1-1}, v_{1-3})$.

	1_R	2_R	3_R
1_T	4_T	3_T	2_T
2_T	$\mathbf{1_T}$	4_T	$\mathbf{3_T}$
3_T	4_T	1_T	2_T
4_T	3_T	2_T	1_T

(c) $\gamma_2 = \begin{array}{l}(\mathbf{4},\ 3,\ \mathbf{2},\\ \underline{1},\ 4,\ \underline{3},\\ 4,\ 1,\ 2,\\ 3,\ 2,\ 1)\end{array}.$

	1_R	2_R	3_R
1_T	4_T	3_T	2_T
2_T	3_T	4_T	1_T
3_T	$\mathbf{4_T}$	1_T	$\mathbf{2_T}$
4_T	3_T	2_T	1_T

(d) $\gamma_3 = \begin{array}{l}(\mathbf{4},\ 3,\ \mathbf{2},\\ \mathbf{3},\ 4,\ \mathbf{1},\\ \underline{4},\ 1,\ \underline{2},\\ 3,\ 2,\ 1)\end{array}.$

	1_R	2_R	3_R
1_T	4_T	3_T	2_T
2_T	3_T	4_T	1_T
3_T	2_T	1_T	4_T
4_T	$\mathbf{3_T}$	2_T	$\mathbf{1_T}$

(e) $\gamma_3 = \begin{array}{l}(\mathbf{4},\ 3,\ \mathbf{2},\\ \mathbf{3},\ 4,\ \mathbf{1},\\ \mathbf{2},\ 1,\ \mathbf{4},\\ \underline{3},\ 2,\ \underline{1})\end{array}.$

	1_R	2_R	3_R
1_T	4_T	3_T	2_T
2_T	3_T	4_T	1_T
3_T	2_T	1_T	4_T
4_T	1_T	2_T	3_T

(f) $\gamma_n = \begin{array}{l}(\mathbf{4},\ 3,\ \mathbf{2},\\ \mathbf{3},\ 4,\ \mathbf{1},\\ \mathbf{2},\ 1,\ \mathbf{4},\\ \mathbf{1},\ 2,\ \mathbf{3})\end{array}.$

Figure 4: Configurations involved in a column swap move. Light gray color marks the variables to be processed, dark gray highlights vertices chosen by the β selector. Subscripts T and R correspond to the team- and round-indexes accordingly.

Given this representation we can define 2-opt operator as $\gamma_n = \gamma_s[1 : i-1] * v_y * r(\gamma_s[i+1 : j-1]) * v_x * \gamma_s[j+1 : n]$. The notation $\gamma[i : j]$ here stands for the projection of the configuration γ on the vertices positioned inclusively on positions between i and j; $*$ stands for concatenation of two tuples; r reverses the tuple order. v_x and v_y are two randomly chosen, distinct vertices. Reversing is required because of the representation's directed nature.

One can see that m-opt operators require two non-trivial operations: reversing and swapping parts of the configuration. The NDL language, does not support any data structures nor procedures required for those operations. In the following paragraph we will show that enriching the problem representation allows one to overcome those limitations.

Rich Model. The issue with the basic representation is that it loses a lot of information about the problem's structure — the TSP is reduced to a permutation problem and there is no way to distinguish qualitatively between two different permutations without referring to an external criterion. A more effective representation is to use *circuit* global constraint — the configuration is now n-tuple $\gamma = (i_1, i_2, \dots i_n)$ such that k-th value represents an edge starting at vertex v_k and ending at vertex i_k. To represent *circuit* as an annotated problem, one has to decompose it into binary constraints. The resulting representation of a TSP with m vertices is $P_T = (X, D, C, T, I, U)$, such that:

- $X = (x_{n_1}, x_{n_2}, \dots x_{n_m}, x_{a_1}, x_{a_2}, \dots x_{a_m})$ where x_{n_k} corresponds to the variables falling under the *circuit* constraint and x_{a_k} are auxiliary variables corresponding to order of vertices as in the basic representation.

- $D = (D_{n_1}, D_{n_2}, \dots D_{n_m}, D_{a_1}, D_{a_2}, \dots D_{a_m})$ such that:
  - $D_{n_k} = \{1, 2, \dots, m\} \setminus \{k\}$: self-loops are forbidden.
  - $D_{a_1} = \{1\}$ (start at the first vertex) and $\forall k > 1, D_{a_k} = \{2, \dots m\}$
- There are two types of variables $T = \{T_n\}^n * \{T_a\}^n$ such that $T_n = (\tau_1, D_n)$ and $T_a = (\tau_2, \{1, 2, \dots, m\})$.
- Variables are indexed in their appearing order $I = (1, 2, \dots, m, 1, 2, \dots, m)$.
- $C = C_n * C_a * C_o$ where:
  - $C_n = \left(C_{n_1}, C_{n_2}, \dots, C_{n_{m(m-1)/2}}\right)$ expressing that every two variables x_{n_i} and x_{n_j} must not have the same value. It would mean that two edges in the cycle share an ending vertex.
  - $C_a = \left(C_{a_1}, C_{a_2}, \dots, C_{a_{m(m-1)/2}}\right)$ expressing that every two variables x_{a_i} and x_{a_j} must not have the same value. It would mean that the same vertex is visited twice in the cycle.
  - $C_o = \left(C_{o_{2-1}}, C_{2-2}, \dots C_{o_{m-m}}\right)$ — every auxiliary variable x_{a_k} with $k > 1$ has to be ordered as stated in the edges $\{x_{n_1}, \dots, x_{n_m}\}$.
- $U = \{u_1\}^{|C_n|} * \{u_2\}^{|C_a|} * \{u_3\}^{|C_o|}$ — constraints are typed as defined above.

The biggest gain from using this representation is that we explicitly handle local configuration issues such as self-loops and edges sharing the same terminal vertex. Due to the edge-based representation, swapping cycle edges is just an exchange of the terminal vertices.

It is noteworthy that the x_a variables depend on the x_n variables and do not have to be modified by the neighborhood operator. To simplify the notation, in the example below, configuration γ will cover only the x_n variables represented by the corresponding nodes in the TCN: $v_1, v_2, \dots, v_m$.

NDL Operator The 2-opt operator definition in NDL will consist of three parts: picking suitable variables, introducing change into configuration and fixing violated constraints by reversing involved edges using the least fixpoint operator.

$$\begin{aligned} N_{2_o}(\gamma_s) = \gamma_n \Leftrightarrow\ & v_x = \omega_T(T_n) \wedge v_y = \omega_T(T_n) \wedge v_x \neq v_y \wedge y = \psi_{v_y}(\gamma_s) \\ & \wedge x = \psi_{v_x}(\gamma_s) \wedge l_V(v_x) = (T_n, i_x) \wedge v_z = \omega_I(\gamma_s, y, T_n) \\ & \wedge \gamma_1 = \mu_\alpha(\gamma_s, v_y, i_x) \wedge \gamma_2 = \mu_\alpha(\gamma_s, v_z, x) \wedge \gamma_n = \mho(\gamma_2, \alpha, v_x) \end{aligned}$$

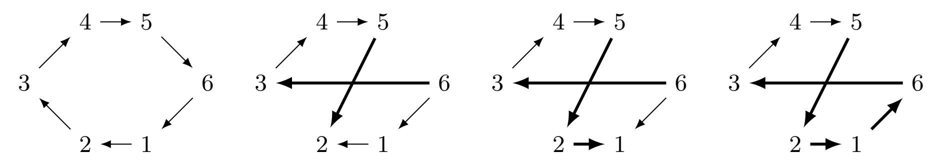
where:

$$\begin{aligned} \alpha(\gamma_k, a_\mho) = \gamma_{k+1} \Leftrightarrow\ & l_A(a_\mho) = u_1 \wedge v_s = \omega_s(a_\mho) \wedge v_s \neq v_z \\ & \wedge l_V(v_s) = (T_n, i_s) \wedge v_p = \omega_e(a_\mho) \wedge p = \psi_{v_p}(\gamma_d) \\ & \wedge p = i_s \wedge l_V(v_p) = (T_N, i_p) \wedge \gamma_{k+1} = \mu_\alpha(\gamma_k, v_s, i_p) \end{aligned}$$

Figure 5 presents a states' sequence defined by this operator in a small graph.

In order to define the 3-opt operator, one would have to extend N_{2o} by selecting an additional variable and use $f_<$ filter to ascertain their order.

It is worth of nothing that partly due to the apparent complexity of this definition, TSP has been chosen for the first experiments on automatic synthesis



(a) $(2,3,4,5,6,1)$. (b) $(2,3,4,5,\underline{2},\underline{3})$. (c) $(2,\underline{1},4,5,\mathbf{2},\mathbf{3})$, with: $a_\mho = (v_2, v_1)$. (d) $(\underline{6},\mathbf{1},4,5,\mathbf{2},\mathbf{3})$, with: $a_\mho = (v_1, v_6)$.

Figure 5: Configurations involved in a 2-opt move, replacing $(2,3)$, $(5,6)$ edges.

of the NDL operators [21]. The experiments have shown that an evolutionary algorithm is able to find an equivalent and shorter (but at the same time less obvious) definition of the 2-opt operator.

6 Related Works

As stated in Section 2, there are several existing approaches to represent neighborhood operators. In this section we will compare NDL to two of them: OptaPlanner [4] and Declarative Neighborhoods [14]. Other referenced languages either focus mostly on heuristics or combine existing neighborhoods into search strategies. Both subjects, while related, are not directly comparable to our approach in the present state of the work.

The Neighborhood Definition Language shares many similarities with OptaPlanner's XML approach. In both languages the moves are composed of variable/value selectors and modifiers ("move selectors" in the OptaPlanner terminology). Besides the basic perturbations, like a value swap/change, OptaPlanner also defines more complex domain-specific move selectors like 2-opt or group swaps/changes that can be used to implement row/column group operations. Another noticeable difference is that selectors do not exploit the constraint structure and can only refer to variable types and values. The lack of more generic combinators makes it impossible to define new complex neighborhoods such as Kempe Chain. Consequently, OptaPlanner definitions are more coarse-grained, requiring several interesting and useful operators to be directly implemented in the low-level Java programming language.

In *Declarative Neighborhoods*, neighborhood is defined as a Constraint Satisfaction problem and because of that, cannot express any kind of recursion, even one as limited as the move combinators approach used in NDL. This greatly restricts the expressiveness of the language, limiting it only to a fixed number of perturbations, similar to a single NDL move. The language may still be extended to include more complex moves (as it is done in OptaPlanner) like column or row swaps, with the restriction that they perform only a fixed number of changes. The main advantage of Declarative Neighborhoods over NDL is its ability to set specific requirements on the neighborhoods by means of Constraint Programs. This way, the neighborhood can be easily pruned and specific constraints can explicitly be made invariant. At the same time, such a pruning relies on solving a constraint satisfaction problem, which in the general case is an NP-complete procedure and may be computationally prohibitive.

7 Summary

In this paper we have presented the main elements of the Neighborhood Definition Language — a formal language capable of representing the Local Search neighborhood operators in a fine-grained manner. Compared to general-purpose programming languages, NDL has a well defined semantics and is limited to primitive recursion, effectively leading to always terminating total programs. Compared to other declarative approaches, it is much more self-contained and expressive enough to represent even complex neighborhood operators. Such results have been achieved partly with a rich problem representation borrowed from the Constraint Programming formulation, and partly with a limited set of recursion schemes, effectively exploring a problem's structure.

Our current research is focused on integration with modern modeling languages and solvers. The biggest issues include finding extraction methods capable of creating a TCN based on the CSP model, but also the injection of arbitrary NDL operators into current Local Search solvers. The natural candidates for the task are the MiniZinc language, compatible with a notion of the constraint/variable types and the Oscar/CBLS solver, which combines Local Search with the Constraint representation.

References

1. H. De Beukelaer, G. F. Davenport, G. De Meyer, and V. Fack, "JAMES: An object-oriented Java framework for discrete optimization using local search metaheuristics," *Software: Practice and Experience*, vol. 47, no. 6, pp. 921–938, Jun. 2017.
2. A. Schaerf, M. Lenzerini, and M. Cadoli, "LOCAL++: a C++ framework for local search algorithms," in *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*, Jul. 1999, pp. 152–161.
3. L. D. Gaspero and A. Schaerf, "EASYLOCAL++: an object-oriented framework for the flexible design of local-search algorithms," *Software: Practice and Experience*, vol. 33, no. 8, pp. 733–765, 2003.
4. "OptaPlanner User Guide." [Online]. Available: https://docs.optaplanner.org/7.15.0.Final/optaplanner-docs/html_single/index.html
5. R. Fourer, D. M. Gay, and B. W. Kernighan, "A Modeling Language for Mathematical Programming," *Management Science*, vol. 36, no. 5, pp. 519–554, May 1990.
6. P. Van Hentenryck, *The OPL Optimization Programming Language.* Cambridge, MA, USA: MIT Press, 1999.
7. A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel, "Essence: A constraint language for specifying combinatorial problems," *Constraints*, vol. 13, no. 3, pp. 268–306, Sep. 2008.

8. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4741, pp. 529–543.
9. L. Michel and P. Van Hentenryck, "Localizer A modeling language for local search," in *Principles and Practice of Constraint Programming-CP97*, ser. Lecture Notes in Computer Science, G. Smolka, Ed. Springer Berlin Heidelberg, 1997, pp. 237–251.
10. P. Van Hentenryck and L. Michel, *Constraint-Based Local Search.* The MIT Press, 2005.
11. OscaR Team, *OscaR: Scala in OR*, 2012.
12. R. D. Landtsheer, Y. Guyot, G. Ospina, and C. Ponsard, "Combining Neighborhoods into Local Search Strategies," in *Recent Developments in Metaheuristics*, ser. Operations Research/Computer Science Interfaces Series. Springer, Cham, 2018, pp. 43–57.
13. A. Rendl, T. Guns, P. J. Stuckey, and G. Tack, "MiniSearch: A Solver-Independent Meta-Search Language for MiniZinc," in *Principles and Practice of Constraint Programming*, G. Pesant, Ed. Cham: Springer International Publishing, 2015, vol. 9255, pp. 376–392.
14. G. Björdal, P. Flener, J. Pearson, P. J. Stuckey, and G. Tack, "Declarative Local-Search Neighbourhoods in MiniZinc," in *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, Nov. 2018, pp. 98–105.
15. G. Pesant and M. Gendreau, "A constraint programming framework for local search methods," *J. Heuristics*, vol. 5, no. 3, pp. 255–279, 1999. [Online]. Available: https://doi.org/10.1023/A:1009694016861
16. P. Shaw, B. D. Backer, and V. Furnon, "Improved local search for CP toolkits," *Ann. Oper. Res.*, vol. 115, no. 1-4, pp. 31–50, 2002. [Online]. Available: https://doi.org/10.1023/A:1021188818613
17. G. Björdal, P. Flener, J. Pearson, and P. J. Stuckey, "Exploring declarative local-search neighbourhoods with constraint programming," in *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, ser. Lecture Notes in Computer Science, T. Schiex and S. de Givry, Eds., vol. 11802. Springer, 2019, pp. 37–53. [Online]. Available: https://doi.org/10.1007/978-3-030-30048-7_3
18. Ö. Akgün, S. Attieh, I. P. Gent, C. Jefferson, I. Miguel, P. Nightingale, A. Z. Salamon, P. Spracklen, and J. Wetter, "A Framework for Constraint Based Local Search using Essence,," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018.
19. S. Attieh, N. Dang, C. Jefferson, I. Miguel, and P. Nightingale, "Athanor: High-level local search over abstract constraint specifications in essence," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, S. Kraus, Ed. ijcai.org, 2019, pp. 1056–1063. [Online]. Available: https://doi.org/10.24963/ijcai.2019/148
20. M. Ślażyński, S. Abreu, and G. J. Nalepa, "Towards a formal specification of local search neighborhoods from a constraint satisfaction problem structure," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019.*, 2019, pp. 137–138.
21. ——, "Generating local search neighborhood with synthesized logic programs," in *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019.*, 2019, pp. 168–181.

22. E. C. Freuder and A. K. Mackworth, “Chapter 2 - Constraint Satisfaction: An Emerging Paradigm,” in *Foundations of Artificial Intelligence*, ser. Handbook of Constraint Programming, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, Jan. 2006, vol. 2, pp. 13–27.
23. E. Meijer, M. Fokkinga, and R. Paterson, “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire,” in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture.* Berlin, Heidelberg: Springer-Verlag, 1991, pp. 124–144.
24. C. A. Morgenstern and H. D. Shapiro, *Chromatic Number Approximation Using Simulated Annealing.* Department of Computer Science, College of Engineering, University of New Mexico, 1986.
25. T. Walsh, *CSPLib Problem 026: Sports Tournament Scheduling*, C. Jefferson, I. Miguel, B. Hnich, T. Walsh, and I. P. Gent, Eds. [Online]. Available: http://www.csplib.org/Problems/prob026
26. L. D. Gaspero and A. Schaerf, “A composite-neighborhood tabu search approach to the travelling tournament problem,” *Journal of Heuristics*, vol. 13, pp. 189–207, 2007.