Solution Neighbourhoods for Constraint-Directed Local Search

Jun He, Pierre Flener, and Justin Pearson
Uppsala University, Department of Information Technology, Uppsala, Sweden
Firstname.Lastname@it.uu.se

ABSTRACT

We propose *solution neighbourhoods*, which contain **only solutions** to a chosen constraint, as the solutions to a constraint capture the structure of the constraint. We save the time needed for neighbourhood evaluation of that constraint by using a solution neighbourhood. This may be useful especially for constraints for which there exists no known constant-time algorithm for neighbour evaluation. We design a solution neighbourhood for the very useful *automaton* constraint, and demonstrate the practicality of our approach on a library of nurse scheduling instances. We show the feasibility of designing solution neighbourhoods for other constraints.

1. INTRODUCTION

In constraint-based local search (CBLS, e.g., [6]), constraints are used to describe and control local search for solving a combinatorial problem. From an initial assignment of values to all decision variables, CBLS iteratively moves to another assignment, which ideally decreases the total amount of constraint violation, by exploring a neighbourhood of small changes to the current assignment, until an assignment with zero constraint violation is found or until allocated resources are exhausted. Meta-heuristics are used to escape local optima. In CBLS, search can be constraint-directed or variable-directed, and it is crucial to choose a good neighbourhood and explore it efficiently. We address how to design a neighbourhood in constraint-directed search.

In *constraint-directed search* (CDS), first a (possibly most) violated constraint is selected, and then a (possibly best) move is found by exploring a neighbourhood of that constraint, where a (possibly best) move is a move that (possibly maximally) decreases the violation of the whole constraint system. There are at least two ways to design a neighbourhood of a chosen constraint: a *variable-directed neighbourhood* selects a few decision variables of the constraint, and designs a neighbourhood by changing the assignments to those decision variables; a *constraint-directed neighbourhood* designs a neighbourhood according to the possible violation changes of the chosen constraint. With a variable-directed neighbourhood, the search focuses on a small and prospective neighbourhood, which is efficient but easily trapped at a local optimum. With a constraint-directed neighbourhood, the search exploits some of the substructure of the combinatorial problem, namely the chosen constraint, but it may be inefficient if the neighbourhood is large. Hence it is interesting to integrate the two methods, and design a hybrid neighbourhood, namely a variable-directed **and** constraint-directed neighbourhood, so that we can benefit from the advantages of both. We address how to design a hybrid neighbourhood for the very useful *automaton* constraint [3], a particular case of which is also known as the *regular* constraint [5].

When designing a hybrid neighbourhood for the *automaton* constraint, we are interested in a neighbourhood where **all** neighbours **satisfy** the constraint, called a *solution neighbourhood*, for the following reasons: first, it is easy to find a solution to an *automaton* constraint that is close to a given violating assignment by exploring the structure of the *automaton* constraint; second, as far as we know, the best algorithm for neighbour evaluation of the *automaton* constraint takes time linear in its number of decision variables [4], which is expensive, thus we can avoid such an expensive neighbour evaluation by using a solution neighbourhood.

There is of course a cost of designing a solution neighbourhood. Usually, neighbourhoods are generated in constant time per neighbour, and a neighbour evaluation takes constant time for many constraints, but it (necessarily) takes more than constant time for some constraints; e.g., the *automaton* constraint. We investigate the following question: when does it pay off to generate a small neighbourhood of **solutions** to a given constraint, in **non**-constant time per neighbour, so as to achieve evaluation in **zero time** of these neighbours for that constraint? We show that such an investment in neighbourhood generation is well amortised for such expensive constraints as *automaton*, and that even some cheap constraints such as *alldifferent* can benefit from this idea.

The contributions and organisation of this paper are as follows: we introduce the idea of using a hybrid neighbourhood for CDS in Section 2; we show how to design a solution neighbourhood for the *automaton* constraint in Section 3, where all neighbours are solutions to the chosen constraint **and** introduce useful diversification (in addition to any diversification introduced by a meta-heuristic); we present experiment results establishing the practicality of our method in Section 4; finally, in Section 5, we show the feasibility of designing solution neighbourhoods for other constraints, summarise this work, and discuss related and future work.

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

Algorithm 1 A CDS generic procedure with a hybrid neighbourhood for k variables to solve a CSP P within i_{max} steps

1: **generic procedure** $CDSearch(P = \langle X, D, C\rangle, i_{max}, k)$
2: $i \leftarrow 0$
3: $a \leftarrow$ an initial assignment of the decision variables X
4: $best \leftarrow \sum_{c\in C} Violation_c[a]$; $\bar{a} \leftarrow a$
5: **while** $best > 0$ **and** $i < i_{max}$ **do**
6: **select** a ((most) violated) constraint $c' \in C$ **do**
7: **select** a sequence V of at most k ((most) violating) decision variables of c' **do**
8: $N \leftarrow Neighbourhood_{c'}(V, a_{X_{c'}})$
9: **select** a neighbour $n \in N$ **maximising** $\sum_{c\in C}(Violation_c[a] - Violation_c[n])$ **do**
10: **update** a w.r.t. n; $v \leftarrow \sum_{c\in C} Violation_c[a]$
11: **if** $best > v$ **then** $best \leftarrow v$; $\bar{a} \leftarrow a$
12: $i \leftarrow i + 1$

2. CDS WITH A HYBRID NEIGHBOURHOOD

Let $P = \langle X, D, C\rangle$ denote a constraint satisfaction problem (CSP), where X is a finite sequence of decision variables, D is the sequence of domains of X, and C is a finite set of constraints; let a be the current assignment of X; for any constraint $c \in C$, let $X_c \subseteq X$ be the decision variables involved in c; and let a_{X_c} be the current assignment of X_c. In a CBLS system (such as the CBLS back-end of Comet [6]), two data structures are automatically maintained incrementally: $Violation_c[a]$ denotes the violation of constraint c under the assignment a; $Violation_c[x, a]$ denotes the violation of decision variable x in constraint c under the assignment a. We call $\sum_{c\in C} Violation_c[a]$ the *system violation* and $\sum_{c\in C} Violation_c[x, a]$ the *system variable violation* of decision variable x under assignment a.

The generic procedure $CDSearch(P, i_{max}, k)$ in Algorithm 1 describes how P is solved in CDS with a hybrid variable-directed (line 7) and constraint-directed (line 8) neighbourhood. The choice of parameter k is problem-specific; we will address this issue for the experiments in Section 4.2.1. It first initialises the iteration counter i to zero (line 2), a to an initial assignment (line 3), *best* (which denotes the violation of the current best assignment) to the violation of the initial assignment, and $\bar{a}$ (which denotes the current best assignment) to the initial assignment (line 4). It then checks the termination condition (either a solution has been found or the maximum number i_{max} of iterations has been reached) of the search (line 5). If the termination condition is not met, then it selects a ((most) violated) constraint $c' \in C$ (line 6) and a sequence V of at most k ((most) violating) decision variables of c' (line 7), and then designs a hybrid variable-directed (by V) and constraint-directed (by c') neighbourhood N under the current assignment a by calling the function $Neighbourhood_{c'}(V, a_{X_{c'}})$ (line 8), which returns a neighbourhood obtained by reassigning V, an instance for the *automaton* constraint being described in Algorithm 2 and discussed in Section 3.2.1. Each neighbour $n \in N$ is evaluated by the sum of the differentially computed violation changes over all constraints $c \in C$ upon the corresponding move (line 9), and a randomly chosen best move achieving the maximum violation decrease for C is made (line 10). If a new best assignment is found upon the move, then *best* and $\bar{a}$ are updated (line 11). The iteration counter i is increased after each iteration (line 12).

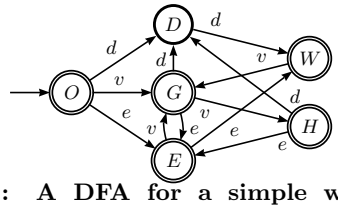

Figure 1: A DFA for a simple work scheduling constraint for one employee

Algorithm 1 lacks meta-heuristics to escape local optima, as these are orthogonal issues that can be addressed separately [6]. Diversification and intensification are not discussed here for the same reason, because we make no assumptions on the assignment a. Note that each constraint has its own neighbourhood generator based on the implementation of the function $Neighbourhood_c(V, a_{X_c})$. This can be implemented as an API extension to the constraint class of some CBLS system, such as the CBLS back-end of Comet [6] (available at `dynadec.com`).

3. SOLUTION NEIGHBOURHOODS

After an introduction to the *automaton* constraint, we show how to design a solution neighbourhood for the *automaton* constraint. We only consider the *automaton* constraint without counters here, but plan to expand on this.

3.1 The Automaton Constraint

The *automaton* constraint is defined as $automaton(X, M)$, where X is a sequence of decision variables, and M is a deterministic finite automaton (DFA). A DFA is a tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is a finite set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accepting states. The *automaton* constraint is satisfied iff M accepts the word made of the values of the sequence X of decision variables.

For example, Figure 1 gives a DFA that describes a work scheduling constraint for one employee. There are values for two work shifts, namely day (d) and evening (e), and a value for enjoying a day of vacation (v). The set of words accepted by this DFA defines the set of acceptable shift sequences.

Given an $automaton(X, M)$ constraint with $|X| = n$ decision variables and the DFA $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, the (minimised) conjunctive product of M with the DFA accepting Σ^n gives a DFA M' that only accepts n-letter words in the language of M. This construction is known in CP as the *unrolling* of M for words of length n [5]. For example, the DFA M' obtained by unrolling M in Figure 1 for words of length $n = 6$ is in Figure 2 (the reader can ignore the different line styles of the transitions and states at the moment). Note that any state q_m of M in vertical layer j is named q_m^j in M'. Any assignment made of the values labelled on the arcs of a path in M' is a solution to the *automaton* constraint, and vice versa. Note that whenever a path in M' is mentioned, we mean a path from the start state of M' to any accepting state of M'. In Figure 2, there are 49 such paths, hence the $automaton(X, M)$ constraint with $|X| = n = 6$ decision variables and the DFA M in Figure 1 has 49 solutions among the $|\Sigma|^n = 3^6 = 729$ possible assignments. By an abuse of language, we sometimes refer to an assignment as a path, such that the assignment is made of the values labelled on the arcs of the path.

As far as we know, there are two versions of violation measure [4], based on Hamming distance, for the *automaton*

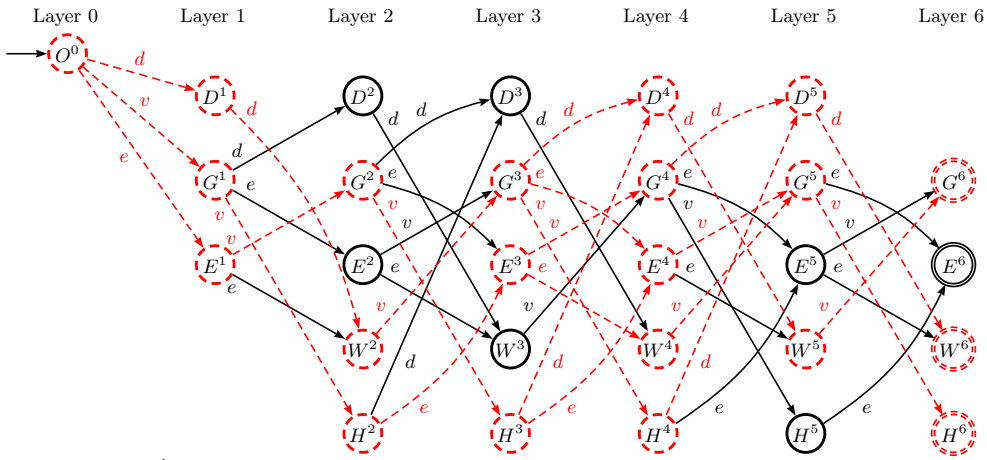

Figure 2: The DFA M' obtained by unrolling the DFA M in Figure 1 for words of length 6. State q_m of M in layer j is named q_m^j in M'. Given the current assignment $a = \langle d, v, v, e, d, v\rangle$ to the decision variables $X = \langle X_1, \ldots, X_6\rangle$, and the sequence $V = \langle X_1, X_4\rangle$ of $k = 2$ decision variables chosen to be reassigned for designing a neighbourhood, the resulting neighbourhood contains all the 7 dashed paths.

constraint in CBLS. We use the one that is shown in [4] to work both theoretically and experimentally much better.

3.2 Designing a Solution Neighbourhood

Consider an $automaton(X, M)$ constraint with a current assignment a to the decision variables X, a sequence $V \subseteq X$ of decision variables to be reassigned for designing a solution neighbourhood, and the DFA M' obtained by unrolling M for words of length $|X|$. With the structure of solutions to the *automaton* constraint as described in Figure 2, topological sorting is used, as in Algorithm 2, to design efficiently a solution neighbourhood for the *automaton* constraint.

3.2.1 Description of the Algorithm

Algorithm 2 with the unique parameter $|V| = k$ works by dashing some states and transitions in the DFA M' obtained by unrolling the DFA M for words of length $|X|$ (lines 2 to 11), so that every dashed path in M' (except the current assignment a) denotes a neighbour in the neighbourhood; and it returns the neighbourhood containing all dashed paths except a in M' (line 12). When considering decision variable X_i and visiting a dashed state q_m^{i-1} in M': if X_i is in V, then all outgoing transitions from state q_m^{i-1} and their target states in M' are dashed (lines 5 to 7), so that all feasible assignments (in the sense that a dashed path can be extended) to decision variable X_i are included in the neighbourhood; otherwise Algorithm 2 tries to extend a dashed path by preserving the assignment of X_i (lines 8 and 9), or (at failure to do so; line 10) by choosing randomly one outgoing transition from q_m^{i-1}, hence providing the chance to change also the assignment of $X_i \notin V$ (line 11; this introduces more diversity, so as to stop the search easily being trapped in some local optimum). As all transitions and states not leading to an accepting state are by construction not in M', there is a path to an accepting state of M' passing each state of M'. Hence for each dashed state (in lines 2, 7, 9, and 11), Algorithm 2 always finds such a path passing this dashed state. As any path in M' is a solution to the *automaton* constraint, Algorithm 2 designs a solution neighbourhood for the *automaton* constraint.

For example, in Figure 2 with the current assignment $a = \langle d, v, v, e, d, v\rangle$ to $|X| = 6$ decision variables and a sequence $V = \langle X_1, X_4\rangle$ of $k = 2$ decision variables, the resulting neighbourhood contains all assignments related with any dashed path in M'. There are 7 dashed paths, hence the resulting neighbourhood has 7 neighbours among the 49 solutions. Algorithm 2 first dashes the start state O^0 in layer 0 of M' (line 2). Then it dashes all outgoing transitions from O^0 and their target states (D^1, E^1, and G^1) in layer 1, as $X_1 \in V$ (lines 5 to 7). Next it visits the dashed states D^1, E^1, G^1 in this order: when visiting D^1, as there is no outgoing transition from D^1 that is labelled with v (the value assigned to X_2 under a), the only outgoing transition from D^1 and its target state W^2 in layer 2 are dashed (line 11); when visiting E^1, as there is an outgoing transition from E^1 that is labelled with v, this transition and its target state G^2 in layer 2 are dashed (lines 8 and 9); when visiting G^1, as there is an outgoing transition from G^1 that is labelled with v, this transition and its target state H^2 in layer 2 are dashed (lines 8 and 9). The process continues for a while, then Algorithm 2 will visit E^4 in layer 4. As there is no outgoing transition from E^4 that is labelled with d (the value

Algorithm 2 Design a solution neighbourhood directed by an $automaton(X, M)$ constraint and a sequence $V \subseteq X$ of decision variables under the current assignment a to the decision variables X, where the DFA M' obtained by unrolling the DFA M for words of length $|X|$ is pre-computed.

1: **function** $Neighbourhood_{automaton(X,M)}(V, a)$
2: dash the start state of the DFA M'
3: **for all** $i \leftarrow 1$ **to** $|X|$ **do**
4: **for all** dashed states q_m^{i-1} in M' **do**
5: **if** $X_i \in V$ **then**
6: **for all** outgoing transitions t from q_m^{i-1} in M' **do**
7: dash t and its target state in M'
8: **else if** exists an outgoing transition t from q_m^{i-1} so that t is labelled with the value of X_i under a **then**
9: dash t and its target state in M'
10: **else**
11: select a random outgoing transition t from q_m^{i-1} and dash t and its target state in M'
12: **return** all dashed paths except a in M'

assigned to X_5 under a), one (and only one) random transition between the two outgoing transitions ($E^4 \xrightarrow{v} G^5$ and $E^4 \xrightarrow{e} W^5$) will be chosen to be dashed (in Figure 2, the transition $E^4 \xrightarrow{v} G^5$ is chosen). The process continues until all dashed states are visited, and returns the neighbourhood of 7 neighbours (7 dashed paths on Figure 2).

3.2.2 Complexity and Theoretical Evaluation

To establish the time complexity of Algorithm 2, let $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ be the DFA of an *automaton*(X, M) constraint with $|X| = n$ decision variables, let all decision variables $x \in X$ have the same domain Σ, let M' denote the DFA obtained by unrolling M for words of length n, and let a sequence $V \subseteq X$ of k decision variables be chosen to design a solution neighbourhood. Lines 3 to 11 explore all transitions and states in M' at most once, thus take $O(n\cdot|\delta|+n\cdot|Q|)$ time (since M' has $O(n \cdot |\delta|)$ transitions and $O(n \cdot |Q|)$ states); line 12 explores all dashed paths in M' once, thus takes $O(n \cdot |N|)$ time (every path has a length of n), where $|N|$ is $O(|\Sigma|^k)$, since for each reassignment of V, at most one path will be found and inserted into the neighbourhood. Hence Algorithm 2 takes $O(n \cdot (|\delta| + |Q| + |\Sigma|^k))$ time in total.

Compared with designing a pure variable-directed neighbourhood, which takes $\Theta(|\Sigma|^k)$ time (designing a neighbourhood of size $|\Sigma|^k - 1$), our method of designing a solution neighbourhood for the *automaton* constraint takes n times more time. However we do **not** need to evaluate the violation changes for the chosen *automaton* constraint when exploring the solution neighbourhood, as the violation changes for that *automaton* constraint are **all** the same (the solution neighbourhood contains only solutions to that *automaton* constraint) and make no contribution to distinguishing the neighbours. Note that the best known algorithm for differential neighbour evaluation for the *automaton* constraint on n decision variables takes $O(n)$ time [4], which is expensive. Hence the time cost of designing the solution neighbourhood with large enough k will be paid off by the time needed for exploring the neighbourhood of the *automaton* constraint.

Consider the frequent problem pattern with at least two *automaton* constraints, each on n decision variables, so that there is no sharing of decision variables between any two *automaton* constraints. A search directed by the *automaton* constraints normally takes $O(n)$ time **per neighbour** (of the chosen *automaton* constraint) for differentially evaluating the impact on **all** the *automaton* constraints. The benefit of our proposal is that this cost can be brought down to zero time per neighbour (the time of evaluating the impact on any constraints that share decision variables with the chosen *automaton* constraint is unchanged), by investing the cost of designing a solution neighbourhood instead. Usually, the saving of $O(n)$ time per neighbour amortises that investment. We study a rostering problem of this pattern in Section 4. In Section 5, we show that even constraints with differential neighbour evaluation in $O(1)$ time can benefit from a solution neighbourhood.

4. EXPERIMENTAL EVALUATION

We now investigate experimentally the practicality of the proposed CDS with a solution neighbourhood, by comparing it to a non-hybrid neighbourhood, here called the *differ-by-one neighbourhood*, that contains **all** assignments that differ from the current assignment in **any** decision variable of a chosen constraint is explored at each iteration. Note that the differ-by-one neighbourhood is much larger than a variable-directed neighbourhood, which contains neighbours obtained by only changing the assignment of **one chosen** decision variable of a chosen constraint, and the differ-by-one neighbourhood usually works better for CDS. We implemented Algorithm 2 for the CBLS back-end of COMET [6] and ran experiments on the benchmark of NSPlib [7].

Note that the objective of our experiments is **not** to beat the state of the art of nurse rostering, but to show that using a solution neighbourhood can speed up CDS without re-tuning the meta-heuristics.

4.1 The Problem and the Model

NSPlib [7] is a very large repository of (artificially generated) instances of the *nurse scheduling problem* (NSP), which is about constructing a duty roster for nursing staff. Let N be the number of nurses, D the number of days of the scheduling horizon, and S the number of shifts. The objective is to construct an $N \times D$ matrix of values in the integer interval $[1, \ldots, S]$, with value S representing the off-duty shift.

In *instance files*, there are hard *coverage constraints* and soft preference constraints; we only use the former (as optimisation is orthogonal to our searching concerns),which can be modelled by *atLeast* constraints on the columns. There are instance files for 30×28 and 60×28 rosters.

In *case files*, there are hard constraints on the rows. We model all the constraints on a row with a single *automaton* constraint. There are 8 case files for the $N \times 28$ rosters.

COMET has the *atLeast* constraint as a built-in; however its definition of the violation of decision variables does not work well for our experiments. In the COMET (current version 2.1.1) implementation of the *atLeast* constraint, variable violations can be non-zero even if the *atLeast* constraint is satisfied; this may mislead the search if variable violations are used. We coded a revised *atLeast* constraint for our experiments, where the violations of all decision variables are zero iff the violation of the *atLeast* constraint is zero. Our experiments (not reported here for space reasons) show that our revised *atLeast* constraint works up to 5 times faster (and never slower) than the built-in one on our search procedures for the NSPlib problem.

The *automaton* constraint is not a built-in of the CBLS back-end of COMET; we implemented it as in [4].

4.2 The Search Procedures

Two search procedures were tried for the NSPlib problem.

4.2.1 CDS with a Solution Neighbourhood

We instantiate as follows the CDS generic procedure with a hybrid neighbourhood described in Algorithm 1.

When selecting a most violated constraint (line 6 of Algorithm 1), a constraint with the maximum constraint violation is usually selected; however, in our experiments, we observed that selecting a constraint that has the maximum sum of the system variable violations of the decision variables in the constraint works up to 3 times better (and never slower) than selecting a constraint with the maximum constraint violation (these experiments are not reported here for space reasons). We also observed that **always** selecting a most violated *automaton* constraint works up to 10 times faster (and never slower) than selecting a most violated constraint among all *automaton* and *atLeast* constraints.

When selecting $|V| = k$ decision variables to design a solution neighbourhood (line 7), we first select a random **violating** decision variable, and then select $k-1$ random decision variables. We observed that this always works up to 7 times faster than selecting the k most violating decision variables, and that this works up to 2 times faster than selecting k random decision variables (these experiments are not reported here for space reasons).

There are two ways to determine the value of k: (1) Choosing a *static* value. We experimented with $k = 1$ until $k = 4$ to avoid designing a too large neighbourhood. Note that every decision variable in NSPlib has the same domain of $S = 4$ values, thus a solution neighbourhood has a maximum size of $S^k = 4^4 = 256$ for $k = 4$. (2) Choosing a value *dynamically* during search. We experimented with $k = 1$ initially; if a better assignment cannot be found in $k^2 \cdot 2^{k-1}$ iterations (experimentally determined threshold; details are omitted for space reasons) and $k < 4$, then we increase the value of k by one. We never decrease the value of k, as we did not find a good way yet of doing it. We give a comparison of the two methods in Section 4.3.

Our chosen meta-heuristic is tabu search with restarts. The length of the tabu list is the maximum between 6 and the sum of the violations of all constraints. The best assignment so far is maintained. Restarting is done every $N \cdot D$ iterations. The expression for the length of the tabu list and the restart criterion were experimentally determined; details are omitted for space reasons.

4.2.2 Classical CDS

In [4], a *variable*-directed search is used to solve the NSPlib problem. However, we can achieve an improvement by a factor of up to 50 over [4] by replacing its variable-directed search with *constraint*-directed search, where a most violated *automaton* constraint is selected at each iteration (as in Section 4.2.1), the differ-by-one neighbourhood of the constraint is explored, and a best move that maximally decreases the system violation is made. Our chosen meta-heuristic is tabu search with restarts (as in Section 4.2.1).

4.3 The Results

In Table 1, we give a comparison of the two search procedures in Sections 4.2.1 and 4.2.2 on NSPlib instances [7], except for those known from [2] to be unsatisfiable (in order to accelerate the experiments). Other instances might be unsatisfiable (this information cannot be extracted from the NSPlib website), hence the percentage of solved instances might not reach 100% for that reason.

All experiments were run under COMET (version 2.1.1) and Suse Linux 11.3 on a 3.07 GHz Intel Core i7 with a 3GB RAM. All runs were allocated 30 CPU seconds, and the average performance was recorded for each instance over 25 runs. For each run of an instance, all search procedures started with the **same** randomly generated initial assignment. We did not run instances of cases 1 to 10 and cases 12 to 14, as they are very easy [2]; we also did not run instances of case 11, as the DFAs (30 or 60 DFAs with 11816 states and 41922 transitions each, **before** unrolling) for these instances are too large for handling by the *automaton* constraint of [4] under COMET (version 2.1.1) on the chosen hardware. Note that this memory problem is **independent** of search, hence both our search procedures in Sections 4.2.1 and 4.2.2 suffer from the same memory problem.

In Table 1, the average performance over 25 runs each of selected instances (namely 10 instances each for all combinations of the NSPlib complexity indicators) for each case and N is given. Each row first specifies the instances (denoted by Case, N, and Instances), and then gives the performance of each search procedure, namely the average percentage of instances solved without timing out (denoted by %S), the average runtime in seconds (denoted by Sec), and the average number of iterations (denoted by Iter). Numbers that denote the best performance in each row are boldfaced, and runtimes with a margin of 5% of the best performance in each row are also boldfaced. There are four rows for each case and N: the first one denotes the average performance over all selected instances; the following three denote the performance of three hand-picked instances, which represent three difficulty levels with the order of difficulty increasing.

Comparing the two methods mentioned in Section 4.2.1 for choosing the value of k, we observed that: (1) Considering the average percentage of solved instances and the average runtime, the method with dynamic k and the method with $k = 4$ are the winners. (2) Considering the average number of iterations, the method with $k = 4$ always wins, as it explores the largest neighbourhood. (3) Considering the three difficulty levels among the selected instances, the method with $k = 1$ always wins for solving the easy instances, but never works for the harder instances; the method with $k = 4$ and the method with dynamic k always win for solving the harder instances; the method with dynamic k works well among all the three difficulty levels.

Comparing the two search procedures in Sections 4.2.1 and 4.2.2, we observed that the search procedure with a solution neighbourhood in Section 4.2.1, except for those with static $k = 1$ and $k = 2$, always beats the search procedure in Section 4.2.2: considering the average runtime over all selected instances, our method can be up to 2 times faster; considering the runtime of one particular hard instance, our method can be up to 40 times faster.

5. CONCLUSION

In summary, we have shown that the idea of using a solution neighbourhood can be useful for constraint-directed local search. We have demonstrated the idea of designing a solution neighbourhood for the very useful *automaton* constraint. Our experiment results show that our approach not only outperforms the described improvement of [4] on the NSPlib problem, but also outperforms any other CDS procedure that we have tried. Note again that the objective of our experiments was **not** to beat the state of the art of nurse rostering; thus we do not compare with the large amount of related literature on (meta-)heuristics, because most of it is **orthogonal** to our work on neighbour evaluation [6].

Three constraint-directed neighbourhoods are introduced in [1], namely the *decreasing neighbourhood*, *preserving neighbourhood*, and *increasing neighbourhood*, which contain neighbours that decrease, preserve, and increase the constraint violation individually upon making the move. Differently from [1], we are here interested in a non-increasing neighbourhood where **all** neighbours **satisfy** the constraint. There is of course a cost of designing a solution neighbourhood for a chosen constraint, however our experiment results show that this cost for the *automaton* constraint is justified by its benefits. For some other constraints, there are ways to design a solution neighbourhood. For example, consi-

der the *alldifferent*(X) constraint, where $X = \langle X_1, \ldots, X_5 \rangle$ is a sequence of 5 decision variables with the same domain $\{1, \ldots, 5\}$. Given an assignment $a = \langle 1, 3, 5, 4, 5 \rangle$ with a sequence $V = \langle X_3 \rangle$ of $k = 1$ chosen decision variable, we can design a solution neighbourhood for the *alldifferent*(X) constraint as follows: we try each possible reassignment of V; if a solution to the constraint is obtained (e.g., with $X_3 = 2$), then the solution is added to the neighbourhood; otherwise (e.g., with $X_3 = 4$), we try to find a solution by further changing the assignment of some decision variables not in V (e.g., with $X_4 = 2$), and add the solution (if found) to the neighbourhood. We did experiments with the magic square problem, where the *alldifferent* constraint is used, and our experiment results (not reported here for space reasons) show that the cost of designing a solution neighbourhood for the *alldifferent* constraint is also justified by a runtime speed up of more than 3 times for instances of size larger than 5×5. Note that we do **not** force the use of a solution neighbourhood for **every** constraint, as each constraint has its own neighbourhood as described in the generic procedure of Algorithm 1; but we argue that using a solution neighbourhood for **some** constraints can be helpful.

Consider the connectedness of solution neighbourhoods. Given a constraint $c(X)$ over a sequence of n decision variables X and its solution neighbourhood generator, assume that if there is a solution to c extending the current assignment of decision variables $\langle X_1, \ldots, X_i \rangle (1 \leq i < n)$, then the generator can find a neighbour of the current assignment by preserving the assignment of $\langle X_1, \ldots, X_i \rangle$. This is the case for the *automaton* and *alldifferent* constraints. From an arbitrary initial assignment, any solution s to c can be reached within at most $\lceil \frac{n}{k} \rceil$ moves by moving closer to s (changing the assignment of X_1 until X_n) iteratively. As any solution to a CSP must be a solution to each constraint in the problem, we have that any solution to the problem is reachable, and solution neighbourhoods are connected.

Future work includes designing solution neighbourhoods for the *automaton* constraint with counters and other constraints.

Acknowledgements

The authors are supported by grant 2007-6445 of the Swedish Research Council (VR), and Jun He is also supported by grant 2008-611010 of China Scholarship Council and National University of Defence Technology. Many thanks to the anonymous referees.

6. REFERENCES

[1] M. Ågren, P. Flener, and J. Pearson. Revisiting constraint-directed search. *Inf. Comput.*, 207(3):438–457, 2009.

[2] N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. On matrices, automata, and double counting. In *CP- AI-OR'10*, *LNCS* 6140, pages 10–24. Springer, 2010.

[3] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *CP'04*, *LNCS* 3258, pages 107–122. Springer, 2004.

[4] J. He, P. Flener, and J. Pearson. An *automaton* constraint for local search. *Fundam. Inform.*, 107(2–3):223–248, 2011.

[5] G. Pesant. A regular language membership constraint for finite sequences of variables. In *CP'04*, *LNCS* 3258, pages 482–495. Springer, 2004.

[6] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[7] M. Vanhoucke and B. Maenhout. On the characterization and generation of nurse scheduling problem instances. *Eur. J. Oper. Res.*, 196(2):457–467, 2009.

Table 1: Benchmark results over 25 runs each on NSPlib instances, except for those known from [2] to be unsatisfiable (in order to accelerate the experiments). Other instances might be unsatisfiable (this information cannot be extracted from the NSPlib website), hence %S (the average percentage of instances solved without timing out) might not reach 100% for that reason. Note that k is the number of decision variables chosen to design a solution neighbourhood, Sec denotes the average runtime in seconds, and Iter denotes the average number of iterations. Numbers that denote the best performance in each row are boldfaced, and runtimes with a margin of 5% of the best performance in each row are also boldfaced.

			CDS with a solution neighbourhood												Classical CDS		
			$k = 1$			$k = 2$			$k = 3$			$k = 4$			dynamic k			
Case	N	Instances	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter	%S	Sec	Iter
15	30	1…120	77.5	8.26	34947	87.4	4.12	9696	91.5	3.10	4468	**93.6**	**2.57**	**2261**	93.2	**2.69**	2520	85.7	5.13	6761
15	30	2	**100.0**	**0.12**	79	**100.0**	0.18	42	**100.0**	0.19	33	**100.0**	0.24	**30**	**100.0**	0.18	363	**100.0**	0.51	267
15	30	84	0.0	30.10	127098	**100.0**	1.66	3125	**100.0**	0.42	367	**100.0**	**0.51**	**199**	**100.0**	**0.52**	411	36.0	23.56	33928
15	30	92	0.0	30.09	129315	0.0	30.17	83081	0.0	30.17	48250	**68.0**	**17.84**	**18126**	48.0	22.27	21712	0.0	30.18	42712
15	60	1…120	71.5	9.36	35456	78.5	7.06	15224	85.4	5.35	7159	**88.7**	**4.19**	**3430**	**88.7**	**4.29**	3840	75.4	8.67	10807
15	60	2	**100.0**	**0.19**	74	**100.0**	0.33	**60**	**100.0**	0.36	62	**100.0**	0.37	**60**	**100.0**	0.30	128	**100.0**	1.18	527
15	60	72	0.0	30.12	123059	0.0	30.26	69902	96.0	13.70	19920	**100.0**	**2.39**	**1933**	**100.0**	**2.34**	2170	0.0	30.16	42794
15	60	105	0.0	30.13	111310	0.0	30.15	63265	12.0	29.89	38285	**84.0**	**13.31**	**11950**	**84.0**	15.12	13044	0.0	30.25	39209
16	30	1…120	53.9	15.04	62200	91.8	2.96	7132	94.6	2.07	2914	95.3	**1.83**	**1376**	**95.4**	**1.80**	1650	92.8	3.04	4325
16	30	2	**100.0**	**0.19**	197	**100.0**	0.28	32	**100.0**	0.30	**30**	**100.0**	0.33	**30**	**100.0**	0.27	33	**100.0**	0.42	177
16	30	103	0.0	30.12	124984	0.0	30.24	78333	**100.0**	2.98	5593	**100.0**	**0.82**	**708**	**100.0**	1.59	1287	0.0	30.13	54068
16	30	107	0.0	30.12	131029	0.0	30.22	75768	4.0	29.20	46919	88.0	**9.31**	**8044**	**92.0**	**9.09**	8067	0.0	30.23	45150
16	60	1…120	57.9	14.15	53735	86.8	4.72	12238	91.8	3.09	4666	**93.3**	**2.74**	**2218**	93.0	**2.75**	2472	87.2	5.09	7754
16	60	1	**100.0**	0.26	141	**100.0**	0.44	61	**100.0**	0.46	**60**	**100.0**	0.45	**60**	**100.0**	**0.24**	61	**100.0**	1.15	352
16	60	79	0.0	30.16	111910	96.0	6.08	17215	**100.0**	1.55	2334	**100.0**	1.13	**852**	**100.0**	**0.99**	1062	0.0	30.16	52806
16	60	104	0.0	30.16	118241	0.0	30.16	90785	0.0	30.16	56593	**80.0**	**17.32**	**18216**	56.0	20.60	20575	0.0	30.15	57723