Memoisation for constraint-based local search

Magnus Ågren

Swedish Institute of Computer Science
Box 1263, SE – 164 29 Kista, Sweden
magnus.agren@sics.se

Abstract. We present a memoisation technique for constraint-based local search based on the observation that penalties with respect to some interchangeable elements need only be calculated once. We apply the technique to constraint-based local search on set variables, and demonstrate the usefulness of the approach by significantly speeding up the penalty calculation of a commonly used set constraint.

1 Introduction and background

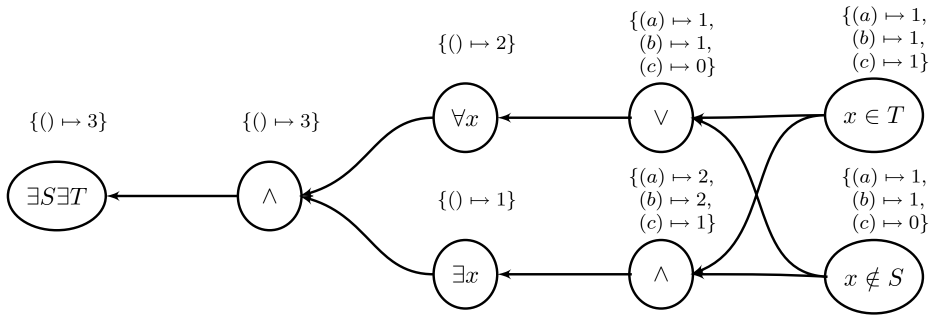
Memoisation [1] is an optimisation technique often used to speed up function calls in programming languages. By caching the calculated results for inputs to a given function, subsequent calls for already seen inputs to the function do not need to be recalculated but can be looked up and returned directly. In this paper we apply memoisation to constraint-based local search.

In constraint-based local search [2], constraint measures are used to navigate in the search space and move towards (optimal) solutions. Given a constraint, such measures include penalties and variable conflicts, which are estimations on how far the constraint currently is from being satisfied and how much each variable contributes to that distance, respectively. Since a local search algorithm may perform many moves, and each move may mean evaluating the constraint measures with respect to a large number of configurations (complete assignments), the evaluation must be done efficiently. This is often achieved by using incremental algorithms (see for example [3]).

In [4] we presented constraint measures with such incremental algorithms for using monadic existential second-order logic (∃MSO) for modelling set constraints in local search. For example, the set constraint $S \subset T$ (strict subset) can be modelled in ∃MSO by:

$$\exists S \exists T((\forall x(x \notin S \lor x \in T)) \land (\exists x(x \notin S \land x \in T))) \tag{1}$$

We call such constraint models *∃MSO constraints.* Now, given a common universe $\mathbf{U}$ for all set variables and an element u of this universe, the penalty of a primitive constraint of the form $u \in S$ or $u \notin S$ is zero if it is satisfied, and one otherwise; the penalty of a conjunction (disjunction) is the sum (minimum) of the penalties of its conjuncts (disjuncts); and the penalty of a first-order universal (existential) quantification is the sum (minimum) of the penalties of the quantified formula where the occurrences of the bound variable are replaced by each element of $\mathbf{U}$.



$\{() \mapsto 3\}$
$\exists S \exists T$
$\{() \mapsto 3\}$
$\wedge$
$\{() \mapsto 2\}$
$\forall x$
$\{() \mapsto 1\}$
$\exists x$
$\{(a) \mapsto 1, (b) \mapsto 1, (c) \mapsto 0\}$
$\vee$
$\{(a) \mapsto 2, (b) \mapsto 2, (c) \mapsto 1\}$
$\wedge$
$\{(a) \mapsto 1, (b) \mapsto 1, (c) \mapsto 1\}$
$x \in T$
$\{(a) \mapsto 1, (b) \mapsto 1, (c) \mapsto 0\}$
$x \notin S$

Fig. 1. Penalty dag of $\exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \notin S \wedge x \in T)))$

The practical relevance of using ∃MSO constraints in local search was demonstrated in [4], where a necessary built-in global constraint was assumed missing and replaced by a corresponding ∃MSO constraint, while still obtaining competitive results in terms of runtime and robustness.

In this paper we use penalty *dags* (directed acyclic graphs), namely attributed parse trees, for illustrative purposes only, and not as an implementation device for supporting incremental maintenance algorithms, as in [4]. For instance, the calculation of the penalty of (1) under the configuration $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ is illustrated by the penalty dag in Fig. 1; the map $\{() \mapsto 3\}$ above the sink $\exists S \exists T$ indicates that the penalty of (1) is 3.

Interchangeable elements (or values) must often be identified in order to solve problems efficiently. The constraint programming community has traditionally done this in the context of symmetry breaking for complete search, where the aim is to *avoid* rediscovering (symmetrically) already encountered (non)solutions (see [5] for an early reference).

On the contrary, we here take advantage of interchangeable elements, and the dag in Fig. 1 can also be used to illustrate the key idea of this paper. Consider the penalty maps above the (rightmost) $\wedge$ and $\vee$ connectives in the dag which, for the corresponding subformulas rooted at those connectives, indicate the penalties with respect to each element of the universe $\mathbf{U} = \{a, b, c\}$. Note that, in both of these penalty maps, the penalties are the same for the elements a and b. This is not by chance but because a and b are interchangeable in the sense that they are both in S and not in T under the configuration k. In fact, *any* element (of the universe) in S and not in T would be interchangeable with a and b and would also share these same penalty maps in the dag. Hence, the penalty maps need only be calculated once for all interchangeable elements, and can then later simply be returned from a cache taking interchangeability into account. We show in the following that this can lead to a significant speedup of local search algorithms with ∃MSO constraints. Note that although we only consider penalties in this paper, all results can be generalised for variable-conflicts as well.

2 Memoising ∃MSO penalties using signatures

Recall that the penalties of first-order quantifications are sums and minima of the penalties of the quantified subformula where the occurrences of the bound first-order variable are replaced by each element of the universe, and consider again (1) and $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$. *The key idea of this paper is based on the following observation:* since the elements a and b are both in S and not in T, the penalties of, for example, the quantified subformula $x \notin S \wedge x \in T$ are the same when x is replaced by a or b. Indeed, both a and b are bound to 2 in the map above the rightmost $\wedge$-node in Fig. 1. So a and b are *interchangeable* in the sense that it is only necessary to calculate the penalty of the subformula given one of the elements, and then reuse that value for the other element. We characterise such interchangeable elements by their *signatures*. The signature of an element $u \in \mathbf{U}$ with respect to a sequence of set variables $\langle S_1, \ldots, S_n \rangle$ under a configuration k is a bit string $b_1 \cdots b_n$ such that $b_i = 1$ if and only if $u \in k(S_i)$.

Example 1. The respective signatures of a, b, c with respect to $\langle S, T \rangle$ under $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ are 10, 10, 00. So a and b share the same signature.

Using signatures to reason about interchangeable elements (or values) was done also in [6], but there in the context of symmetry breaking for complete search.

We will now present a penalty maintenance algorithm for ∃MSO constraints based on memoisation and element signatures. Given an ∃MSO constraint $\Phi = \exists S_1 \cdots \exists S_n \phi$, this algorithm operates on a data structure D with the following fields:

- $D.penalty$, the penalty of Φ under the current configuration;
- $D.signature$, an array indexed by the elements of the universe such that $D.signature[u]$ is the signature of u with respect to $\langle S_1, \ldots, S_n \rangle$ under the current configuration;
- $D.cache$, an array indexed by the (first-order) quantified subformulas ϕ of Φ such that $D.cache[\phi]$ is the penalty cache of ϕ (these penalty caches correspond to the penalty maps of the quantified subformulas in Fig. 1, but are based on element signatures and not on elements);
- $D.min$, an array indexed by the first-order existential quantifications $\exists x \phi$ of Φ such that $D.min[\exists x \phi]$ is a multiset of the penalties of ϕ under the current configuration, where the occurrences of x in ϕ is replaced by each element of $\mathbf{U}$. So the minimum value of $D.min[\exists x \phi]$ is the penalty of $\exists x \phi$.

Following the ideas in [7], the penalty maintenance algorithm consists of two parts: an *initialisation* part and an *update* part. Both of these parts call a generic function *proj_penalty* which is used to traverse the ∃MSO constraint. The intuition behind this is that a call $proj\_penalty(D, \Phi, A)$ returns the penalty of Φ projected on some subset A of the universe. By initially setting A to $\mathbf{U}$, the penalty of Φ is obtained. By later setting A to $\{u\}$, for example, the penalty of Φ projected on $\{u\}$ is obtained. So given a move, for example, of the form $add(S, u)(k)$ (the result of adding u to S under k), the penalty change of Φ

Algorithm 1 Generic function for initialising and updating ∃MSO penalties.

```
1: function $proj\_penalty(D, \Phi, A)$
2:     if $\Phi$ is of the form $\forall x \phi$ then
3:         $p \leftarrow 0$
4:         for all $u \in A$ do
5:             if $D.signature[u] \in D.cache[\phi]$ then
6:                 $p \leftarrow p + D.cache[\phi][D.signature[u]]$
7:             else
8:                 $q \leftarrow proj\_penalty(D, \phi[u/x], \emptyset)$
9:                 $D.cache[\phi][D.signature[u]] \leftarrow q$
10:                $p \leftarrow p + q$
11:        return $p$
12:    if $\Phi$ is of the form $\exists x \phi$ then
13:        for all $u \in A$ do
14:            if $D.signature[u] \in D.cache[\phi]$ then
15:                $add(D.cache[\phi][D.signature[u]], D.min[\exists x \phi])$
16:            else
17:                $q \leftarrow proj\_penalty(D, \phi[u/x], \emptyset)$
18:                $D.cache[\phi][D.signature[u]] \leftarrow q$
19:                $add(q, D.min[\exists x \phi])$
20:        return $\min(D.min[\exists x \phi])$
21:    if $\Phi$ is of the form $\phi \wedge \psi$ then
22:        return $proj\_penalty(D, \phi, A) + proj\_penalty(D, \psi, A)$
23:    if $\Phi$ is of the form $\phi \vee \psi$ then
24:        return $\min(proj\_penalty(D, \phi, A), proj\_penalty(D, \psi, A))$
25:    if $\Phi$ is of the form $u \in S_i$ then return $1 - D.signature[u][i]$
26:    if $\Phi$ is of the form $u \notin S_i$ then return $D.signature[u][i]$
```

can be obtained by two calls $proj\_penalty(D, \Phi, \{u\})$ before and after the move, the penalty change being the difference of the results of these two calls. *Note that this difference is the same as the difference of the results of two such calls where* $\{u\}$ *is replaced by* $\mathbf{U}$. Also note that the current configuration would be a superfluous argument to *proj_penalty* since it is implicit from $D.signature$.

The function *proj_penalty* is shown in Algorithm 1. We here only discuss the quantifier cases on lines 2 to 20 as the other cases closely follow the penalty function described in the first section. For a call $proj\_penalty(D, \forall x\phi, A)$, the sum is calculated by looking up $D.cache[\phi]$ given the signature of each element of A. When a value is in the cache it can be directly used (line 6). Otherwise, the value is calculated by a recursive call where the occurrences of the bound variable x are replaced by the element $u \in A$, and stored in the cache for subsequent calls with the same signature (lines 8 to 10). A call $proj\_penalty(D, \exists x\phi, A)$ is similar, the only difference being that the minimum is calculated by first adding the penalty for the signature of each element of A to the multiset $D.min[\exists x\phi]$ (lines 15 and 19), of which the minimum value is then returned (line 20).

Algorithm 2 Initialise and update procedures for ∃MSO penalties.

```
1: procedure $initialise(D, \exists S_1 \cdots \exists S_n \phi, \mathbf{U})(k)$
2:     for all $u \in \mathbf{U}$ do
3:         $D.signature[u] \leftarrow$ the signature of $u$ with respect to $\langle S_1, \ldots, S_n \rangle$ under $k$
4:     $D.penalty \leftarrow proj\_penalty(D, \phi, \mathbf{U})$
5: procedure $update(D, \Phi)(k, \ell)$
6:     if $\ell$ is of the form $add(S_i, u)(k)$ or $drop(S_i, u)(k)$ then
7:         for all subformulas $\exists x\phi$ of $\Phi$ do
8:             $remove(D.cache[\phi][D.signature[u]], D.min[\exists x\phi])$
9:         $p_0 \leftarrow proj\_penalty(D, \Phi, \{u\})$
10:        for all subformulas $\exists x\phi$ of $\Phi$ do
11:            $remove(D.cache[\phi][D.signature[u]], D.min[\exists x\phi])$
12:        flip the value $D.signature[u][i]$
13:        $p_1 \leftarrow proj\_penalty(D, \Phi, \{u\})$
14:        $D.penalty \leftarrow D.penalty + (p_1 - p_0)$
15:    else
16:        failure
```

Note that $D.signature$ is used to represent the current configuration. So before a call $proj\_penalty(D, \Phi, A)$, $D.signature$ must be updated to reflect this. Also note that, before a call $proj\_penalty(D, \Phi, A)$ on an already initialised D, the penalties in any multiset $D.min[\exists x\phi]$ corresponding to the elements of A, must be removed. This is necessary since projecting the penalty of an existential quantification on $A \subseteq \mathbf{U}$ still requires taking the penalties with respect to *all* elements of $\mathbf{U}$ into account (since it is a minimum value).

The procedure *initialise* is shown in Algorithm 2. A call $initialise(D, \Phi, \mathbf{U})(k)$ initialises the signatures of $D.signature$ to reflect the configuration k (lines 2 to 3) after which a call to $proj\_penalty$ is used to initialise $D.penalty$ (line 4).

The procedure *update* is also shown in Algorithm 2. Given a move ℓ of the form $add(S_i, u)(k)$ or $drop(S_i, u)(k)$ (the results of adding or dropping u from S_i under k), a call $update(D, \Phi)(k, \ell)$ must (twice) update each multiset in $D.min$ by removing one occurrence of the value corresponding to the signature of u (lines 7 to 8 and 10 to 11). (See also the note above concerning this.) The penalty change is then obtained as the difference of the results of two calls to $proj\_penalty$ (lines 9 and 13), before and after the move ℓ has been reflected on $D.signature$ (line 12). This penalty change is then used to update $D.penalty$ (line 14).

Example 2. Let Φ denote (1) and let $\forall x\phi$ and $\exists x\psi$ denote respectively the (first-order) universal and existential quantifications of Φ. Given $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ and $\mathbf{U} = \{a, b, c\}$, the call $initialise(D, \Phi, \mathbf{U})(k)$ initialises the fields of the data structure D such that:

$$D.penalty = 3 \qquad D.signature = [a \mapsto 10, b \mapsto 10, c \mapsto 00]$$
$$D.min = [\exists x\psi \mapsto \{1, 2, 2\}] \qquad D.cache = \begin{bmatrix} \phi \mapsto [00 \mapsto 0, 10 \mapsto 1] \\ \psi \mapsto [00 \mapsto 1, 10 \mapsto 2] \end{bmatrix}$$

Considering now adding c to T, the subsequent call $\mathit{update}(D, \Phi)(k, \ell)$, where $\ell = \mathit{add}(T, c)(k)$, changes the fields of D such that:

$$D.\mathit{penalty} = 2 \qquad D.\mathit{signature} = [a \mapsto 10, b \mapsto 10, c \mapsto 01]$$
$$D.\mathit{min} = [\exists x \psi \mapsto \{0, 2, 2\}] \qquad D.\mathit{cache} = \begin{bmatrix} \phi \mapsto [00 \mapsto 0, 01 \mapsto 0, 10 \mapsto 1] \\ \psi \mapsto [00 \mapsto 1, 01 \mapsto 0, 10 \mapsto 2] \end{bmatrix}$$

The penalty is decreased to two since the constraint is now closer to being satisfied. This is calculated by:

1. removing 1 (the penalty cached for 00 in $D.\mathit{cache}[\psi]$) from $D.\mathit{min}[\exists x \psi]$, setting this multiset temporarily to $\{2, 2\}$ (lines 7 to 8 of Algorithm 2);
2. obtaining $p_0 = 1$ by the call $\mathit{proj\_penalty}(D, \Phi, \{c\})$, which also adds 1 back to $D.\mathit{min}[\exists x \psi]$ (line 9);
3. removing 1 from $D.\mathit{min}[\exists x \psi]$ again (lines 10 to 11);
4. updating the signature of c to 01 (line 12);
5. obtaining $p_1 = 0$ by the call $\mathit{proj\_penalty}(D, \Phi, \{c\})$, which also adds 0 to $D.\mathit{min}[\exists x \psi]$, setting it to $\{0, 2, 2\}$ (line 13);
6. increasing $D.\mathit{penalty}$ by the difference $p_1 - p_0 = 0 - 1 = -1$ (line 14).

3 Evaluation

The algorithms of the previous section were implemented in *Objective Caml* (`http://caml.inria.fr`) and the experiments were performed on a 2.67 GHz Intel Core i7 Linux machine (using only one processor core).

In order to evaluate the memoisation-based penalty maintenance algorithm (called *memo* below) we have compared it to the incremental penalty maintenance algorithm of [4] (called *nomemo* below). We compared these two algorithms by measuring their speed in terms of average number of iterations per second when solving two given problems *subset* and *partition*. Both problems are stated on n set variables $\mathcal{S} = \{S_1, \dots, S_n\}$ all with a common universe $\mathbf{U}$ of cardinality n such that:

– for *subset*, there is an $S_i \subset S_{i+1}$ constraint for each $1 \leq i < n$;
– for *partition*, there is a single $\mathit{Partition}(\mathcal{S})$ constraint.

While the $S_i \subset S_{i+1}$ constraints are modelled in ∃MSO as (1), the $\mathit{Partition}(\mathcal{S})$ constraint (requiring all set variables to be pairwise disjoint and their union to equal $\mathbf{U}$, where any set variable may be empty) is modelled in ∃MSO as:

$$\exists S_1 \cdots \exists S_n \left(\forall x \begin{pmatrix} (x \in S_1 \to (x \notin S_2 \land \cdots \land x \notin S_n)) \\ \land \\ (x \in S_2 \to (x \notin S_3 \land \cdots \land x \notin S_n)) \\ \land \cdots \land \\ (x \in S_{n-1} \to x \notin S_n) \\ \land \\ (x \in S_1 \lor \cdots \lor x \in S_n) \end{pmatrix} \right)$$

Algorithm 3 A simple hill climber for evaluating *memo*.

```
1: function HillClimber($\mathbf{V}, \mathbf{C}$)
2:     $k \leftarrow$ a random configuration for $\mathbf{V}$
3:     while $penalty(\mathbf{C})(k) > 0$ do
4:         choose a possible move $\ell$ of the form $add(S, u)(k)$ or $drop(S, u)(k)$
5:         minimising $penalty(\mathbf{C})(\ell)$ for
6:             $k \leftarrow \ell$
7:     return $k$
```

We chose both simple 2-ary constraints as well as a more complex n-ary constraint in order to compare the overhead versus the gain for *memo*. Intuitively, the gain should be greater for more complex constraints (that is longer ∃MSO formulas), since each saved recalculation would have been more costly for such constraints.

The local search algorithm used for the experiments is a very simple hill climber, shown in Algorithm 3. After initialising the variables $\mathbf{V}$ to a random configuration, the hill climber greedily chooses an *add* or a *drop* move minimising the penalty of all constraints $\mathbf{C}$ as the next configuration. If a configuration satisfying all constraints is found (that is, if the penalty of all constraints is zero), this solution is returned. This hill climber serves our purposes simply since none of the problems *subset* or *partition* are particularly hard. This is however irrelevant as we are here *only* interested in measuring the *speed* of a memoisation-based penalty maintenance algorithm (that is *memo*) and comparing this speed with the speed of another incremental penalty maintenance algorithm (that is *nomemo*). *Solving open instances of hard problems or making comparisons with other solving approaches are thus not purposes of this paper.*

We ran the instances where $n = |\mathbf{U}| \in \{20, 25, 30, 35, 40, 45, 50, 55\}$ for both problems and the results are shown in Table 1. For each of the problems *subset* and *partition* and with respect to a given instance n, the columns labelled *memo* and *nomemo* indicate the number of iterations per second (in Algorithm 3, higher values are better) achieved by using the respective penalty maintenance algorithms. The column labelled *speedup* indicates the speedup of running *memo* compared with *nomemo*. All values are averages over ten runs. The same random seeds were used when comparing the two different algorithms. Hence, the number of iterations (not reported here) as well as the solutions were the same for the two different algorithms.

On the one hand, using *memo* is slower for *subset* on all instances, although by a small (and seemingly constant) factor. On the other hand, using *memo* is significantly faster for *partition* on all instances. As suspected above, the overhead can be larger than the gain for simple 2-ary constraints since the cost for (re)calculating the penalty (or incrementally updating the same) is small for such constraints. This is not the case for more complex n-ary constraints, which is why the gain can be larger than the overhead for such constraints. This clearly shows the usefulness of memoisation-based penalty maintenance algorithms for local search with ∃MSO.

Table 1. Comparing *memo* with *nomemo* when solving the problems *subset* and *partition* for the instances in the column labelled n. For each problem, the columns labelled *memo* and *nomemo* indicate the number of iterations per second achieved by using the respective algorithms, and the column labelled *speedup* indicates the speedup of running *memo* compared with *nomemo*. All values are averages over ten runs.

	subset			*partition*		
n	*memo*	*nomemo*	*speedup*	*memo*	*nomemo*	*speedup*
20	665.7	893.4	0.7	1017.4	193.5	5.3
25	421.8	572.1	0.7	432.8	93.2	4.6
30	292.1	389.3	0.8	332.6	50.5	6.6
35	214.4	285.0	0.8	248.5	29.3	8.5
40	163.5	217.9	0.8	164.2	18.4	8.9
45	129.4	171.1	0.8	112.9	12.3	9.2
50	104.6	133.9	0.8	73.6	8.4	8.8
55	86.1	111.5	0.8	47.4	5.9	8.0

4 Conclusion

We have applied memoisation to the calculation of penalties for ∃MSO constraints. Our approach is based on identifying interchangeable elements in the first-order quantifications of the ∃MSO constraints, and characterising these elements by their signatures. Such interchangeable elements share penalties and need only be calculated and cached once, thereby lowering the number of necessary calculations as well as the number of cached penalties. Our results show that this can lead to a significant speedup when using ∃MSO constraints in local search.

Acknowledgements. I thank Pierre Flener and Justin Pearson for discussions and comments, as well as the anonymous referees for constructive reviews.

References

1. Michie, D.: Memo functions: a language feature with “rote-learning” properties. Research Memorandum MIP-R-29. Edinburgh: Department of Machine Intelligence & Perception (1967)
2. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
3. Van Hentenryck, P., Michel, L.: Differentiable invariants. In Benhamou, F., ed.: Proceedings of CP’06. Volume 4204 of LNCS., Springer-Verlag (2006) 604–619
4. Ågren, M., Flener, P., Pearson, J.: Generic incremental algorithms for local search. Constraints **12**(3) (September 2007) 293–324 (Collects the results of papers at *CP-AI-OR’05*, *CP’05*, and *CP’06*).
5. Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Proceedings of AAAI’91. (1991) 227–233
6. Sellmann, M., Van Hentenryck, P.: Structural symmetry breaking. In: Proceedings of IJCAI’05, Professional Book Center (2005) 298–303
7. Ågren, M.: Set Constraints for Local Search. PhD thesis, Uppsala University (2007)