Optimal Union-Find in Constraint Handling Rules

Tom Schrijvers*
Department of Computer Science, K.U.Leuven, Belgium
(*e-mail:* `www.cs.kuleuven.ac.be/~toms/`)

Thom Frühwirth
Faculty of Computer Science, University of Ulm, Germany
(*e-mail:* `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Constraint Handling Rules (CHR) is a committed-choice rule-based language that was originally intended for writing constraint solvers. In this paper we show that it is also possible to write the classic union-find algorithm and variants in CHR. The programs neither compromise in declarativeness nor efficiency. We study the time complexity of our programs: they match the almost-linear complexity of the best known imperative implementations. This fact is illustrated with experimental results.

To appear in Theory and Practice of Logic Programming (TPLP).

KEYWORDS: declarative algorithms, time complexity analysis, disjoint-set problem, union-find algorithm, constraint handling rules

1 Introduction

When a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. For example, one often hears the argument that in Prolog some graph algorithms cannot be implemented with best known complexity because Prolog lacks destructive assignment that is needed for efficient update of the graph data structures. In particular, it is not clear whether the union-find algorithm can be implemented with optimal complexity in pure (i.e. side-effect-free) Prolog (Ganzinger and McAllester 2001).

In this programming pearl, we give a positive answer for the Constraint Handling Rule (CHR) programming language. We give a CHR implementation with the best known worst case and amortized time complexity for the classical union-find algorithm with path compression for find and union-by-rank. This is particularly remarkable, since originally CHR was intended for implementing constraint solvers only.

* Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

CHR is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae). In CHR, one distinguishes two main kinds of rules: Simplification rules replace constraints by simpler constraints, e.g. `X`$\geq$`Y` $\wedge$ `Y`$\geq$`X` $\Leftrightarrow$ `X`$=$`Y`. Propagation rules add new constraints, which may cause further simplification, e.g. `X`$\geq$`Y`$\wedge$`Y`$\geq$`Z` $\Rightarrow$ `X`$\geq$`Z`. This combination of propagation and multi-set transformation of logical formulae make CHR a unique and powerful declarative programming language.

Closest to our work is the presentation of a logical algorithm for the union-find problem in (Ganzinger and McAllester 2001). In a hypothetical bottom-up inference rule language with permanent deletions and rule priorities, a set of rules for union-find is given that is proven to run in $\mathcal{O}(M + Nlog(N))$ worst-case time for a sequence of M operations on N elements. The direct efficient implementation of these inference rule system seems infeasible. It is also not clear whether the rules given in (Ganzinger and McAllester 2001) describe the standard union-find algorithm as can be found in text books such as (Cormen et al. 1990). The authors remark that giving a rule set with optimal amortized complexity is complicated.

In contrast, we give an executable and efficient implementation that directly follows the pseudo-code presentations found in text books and that also has optimal amortized complexity. Moreover, we do not rely on rule priorities.

This paper is structured as follows. In the next Section, we review the classical union-find algorithms. Constraint Handling Rules (CHR) are briefly presented in Section 3. Then, in Section 4 we present the implementation of the classical union-find algorithm in CHR. An improved version of the implementation, featuring path compression and union-by-rank, is presented next in Section 5. In Section 6, we argue that this implementation has the same time complexity as the best known imperative implementations. This claim is experimentally evaluated in Section 7. Finally, Section 8 concludes.

2 The Union-Find Algorithm

The classical union-find (also: disjoint set union) algorithm was introduced by Tarjan in the seventies (Tarjan and van Leeuwen 1984). A classic survey on the topic is (Galil and Italiano 1991). The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations on the sets:

- `make(X)`: create a new set with the single element `X`.
- `find(X)`: return the representative of the set in which `X` is contained.
- `union(X,Y)`: join the two sets that contain `X` and `Y`, respectively (possibly destroying the old sets and changing the representative).

A new element must be introduced exactly once with `make` before being subject to `union` and `find` operations.

In the naive algorithm, these three operations are implemented as follows.

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree. Return the root as representative.
- `union(X,Y)`: find the representatives of `X` and `Y`, respectively. To join the two trees, it suffices to `link` them by making one root point to the other root.

This following imperative pseudo-code implements this algorithm:

Naive Union-Find Algorithm

```
make(x)
  p[x] ← x

union(x,y)
  link(find(x),find(y))

link(x,y)
   if x ≠ y
         then p[y] ← x

find(x)
   if x ≠ p[x]
         then  return find(p[x])
         else  return x
```

In this pseudo-code `p[x]` denotes the ancestor of `x` in the tree. If `x` is the root, then `p[x]` equals `x`.

The naive algorithm requires $\mathcal{O}(N)$ time per `find` (and `union`) in the worst case, where N is the number of elements (`make` operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve quasi-constant (i.e. almost constant) *amortized* running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root. After `find(X)` returned the root of the tree, we make every node on the path from `X` to the root point directly to the root. The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth. If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one.

For each optimization alone and for using both of them together, the worst case time complexity for a single find or union operation is $\mathcal{O}(log(N))$. For a sequence of M operations on N elements, the worst complexity is $\mathcal{O}(M+Nlog(N))$. When both optimizations are used, the amortized complexity is quasi-linear, $\mathcal{O}(M+N\alpha(N))$, where $\alpha(N)$ is an inverse of the Ackermann function and is less than 5 for all practical N.

In the naive pseudo-code, the `make`, `link` and `find` operations have to be redefined as follows, to add union-by-rank and path compression.

Union-Find with Union-by-Rank and Path Compression

```
make(x)
  p[x] ← x
  rank[x] ← 0

link(x,y)
   if x ≠ y
     if rank[x] ≥ rank[y]
       then p[y] ← x
            rank[x] ← max(rank[x],rank[y] + 1)
       else p[x] ← y

find(x)
   if x ≠ p[x]
         then p[x] ← find(p[x])
   return p[x]
```

The union-find algorithm has applications in graph theory (e.g. efficient computation of spanning trees). By definition of set operations, a union operator working on representatives of sets is an equivalence relation, i.e. we can view sets as equivalence classes. When the union-find algorithm is extended to deal with nested terms to perform congruence closure, the algorithm can be used for term unification in theorem provers and in Prolog.[1] The WAM (Aït-Kaci 1991), Prolog's traditional abstract machine, uses the basic version of union-find for variable aliasing. While *variable shunting*, a limited form of path compression, is used in some Prolog implementations (Sahlin and Carlsson 1991), we do not know of any implementation of the optimized union-find that keeps track of ranks or other weights.

3 Constraint Handling Rules (CHR)

In this section we give an overview of the syntax and operational semantics for constraint handling rules (CHR) (Frühwirth 1998; Frühwirth and Abdennadher 2003; Duck et al. 2004).

3.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

[1] It is straightforward to combine the existing CHR solvers for term unification with our union-find implementation.

Simplification rule:	$\mathit{Name}\ @\ H \Leftrightarrow C \mid B$,
Propagation rule:	$\mathit{Name}\ @\ H \Rightarrow C \mid B$,
Simpagation rule:	$\mathit{Name}\ @\ H \setminus H' \Leftrightarrow C \mid B$,

where *Name* is an optional, unique identifier of a rule, the *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (query)* is a conjunction of built-in and CHR constraints. A trivial guard expression "`true |`" can be omitted from a rule. Simpagation rules abbreviate simplification rules of the form $\mathit{Name}\ @\ H, H' \Leftrightarrow C \mid H, B$.

3.2 Operational Semantics of CHR

Given a query, the rules of the program are applied to exhaustion. A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog). When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule, when a propagation rule is applied, the body of the rule is added to the goal without removing any constraints.

This high-level description of the operational semantics of CHR leaves two main sources of non-determinism: the order in which constraints of a query are processed and the order in which rules are applied. As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program.[2] This behavior has been formalized in the so-called refined semantics that was also proven to be a concretization of the standard operational semantics (Duck et al. 2004).

In this refined semantics of actual implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written, and when it matches a head constraint of a rule, it will look for the other, *partner constraints* of the head in the *constraint store* and check the guard until an applicable rule is found. We consider such a constraint to be *active*. If the active constraint has not been removed after trying all rules, it will be put into the constraint store. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied. Obviously, ground constraints need never to be considered for waking. The discussion above will be of use in Section 6 where we derive the time complexities of our CHR programs.

[2] The nondeterminism due to the wake-up order of delayed constraints and multiple matches for the same rule are of no relevance for the programs discussed here.

4 Implementing Union-Find in CHR

The following CHR program in concrete ASCII syntax implements the operations and data structures of the naive union-find algorithm without optimizations.

- `root(X)` represents `p[X] = X`,
- `X ~> PX` represents `p[X] = PX`,
- `find(X,R)` implements `R = find(X)`,
- `make` and `union` are identical.

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the operations, so we call them *operation constraints*. The constraints `root/1` and `~>/2` represent the tree data structure and we call them *data constraints*.

ufd_basic

```
make     @ make(X) <=> root(X).

union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).

findNode @ X ~> PX \ find(X,R) <=> find(PX,R).
findRoot @ root(X) \ find(X,R) <=> R=X.

linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> Y ~> X, root(X).
```

The elements we apply `union` to are constants as usual for union-find algorithms. Hence the arguments of all constraints are constants, with the exception of the second argument of `find/2` that must be a variable that is bound to a constant in the rule `findRoot`.

Actually, the use of the built-in constraint `=` in this rule is restricted to returning the element `X` in the parameter `R`. In particular no full unification is ever performed.

This union-find program and the one in the next section are not confluent (Schrijvers and Frühwirth 2004; Schrijvers and Frühwirth 2005), i.e. results are dependent of the order in which applicable rules are applied. The main reason is that the relative order of `find` and `union` operations matters for the outcome of a `find`. This behavior is inherent in the union-find algorithm due to its update of the tree structure (see also the discussion of the logical reading of the rules in (Schrijvers and Frühwirth 2004)).

5 Optimized Union-Find

The following CHR program implements the optimized classical union-find Algorithm with path compression for find and union-by-rank (Tarjan and van Leeuwen 1984). The `union/2` constraint is implemented exactly as for the naive algorithm.

```
                                  ufd_rank
make       @ make(X) <=> root(X,0).

union      @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).

findNode   @ X ~> PX , find(X,R) <=> find(PX,R), X ~> R.
findRoot   @ root(X,_) \ find(X,R) <=> R=X.

linkEq     @ link(X,X) <=> true.
linkLeft   @ link(X,Y), root(X,RX) root(Y,RY) <=> RX >= RY |
                Y ~> X, NRX is max(RX,RY+1), root(X,NRX).
linkRight @ link(X,Y), root(Y,RY), root(X,RX) <=> RY >= RX |
                X ~> Y, NRY is max(RY,RX+1), root(Y,NRY).
```

When compared to the naive version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. The rule `findNode` has been extended for path compression already during the first pass along the path to the root of the tree. This is achieved by the help of the variable `R` that serves as a place holder for the result of the find operation. The `link` rule has been split into two rules, `linkLeft` and `linkRight`, to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is updated (both rules are applicable) and the rank is incremented by one.

6 Time Complexity

While automatic complexity analysis results for CHR exist (Frühwirth 2002), these do not take into account the refined operational semantics of CHR (Duck et al. 2004) and hence will yield only a very crude approximation of the actual complexity.

Instead we establish the time complexity of our CHR programs by first showing that they are operationally equivalent to the respective imperative algorithms. By showing next that all the individual computation steps in the CHR program have the same complexity as their imperative counterparts, we have then effectively proven that the overall time complexity properties are identical to the ones of imperative implementations.

6.1 Operational Equivalence

We start with considering the naive algorithm. Because of the refined operational semantics of CHR, the query of `make/1`, `union/2` and `find/2` constraints (and any other conjunction of constraints) is evaluated from left to right, just as is the case for equivalent calls for the imperative program.

Because of this execution order, the operation constraints behave just as their imperative counterparts. The imperative `if-then` and `if-then-else` constructs are encoded as multiple rules. The appropriate rule will be chosen because of a combination of different matchings, partner constraints and guards.

Moreover, the recursion depth for the `find/2` constraint is equal to the path from the initial node to the root just like in the imperative algorithm. The unification in the body of the `findRoot` rule cannot wake up any constraints, since the variable that is bound to a constant does not occur in any other constraint processed so far.

It is clear from the CHR program and the refined operational semantics, that there is only ever at most one operation constraint in the constraint store. Moreover, whenever a data constraint is called, the operation constraint has already been removed. Thus a data constraint will never trigger any rule, because of lack of the necessary partner constraint.

6.2 Time Complexity Equivalence

Now that we have shown the operational equivalence of the CHR program with the imperative algorithm, we still need to show that the time complexities of the different computation steps (corresponding to rule applications) are also equal.

The following time complexity assumptions of a CHR implementation are reasonable (based on the discussion of CHR operational semantics in Section 3.2). They are effectively implemented by the SICStus (Intelligent Systems Laboratory 2003), HAL (Holzbaur et al. 2005) and K.U.Leuven (Schrijvers and Demoen 2004) CHR systems. All of the following operations of the refined operational semantics take constant time:

- The `Activate` transition, excluding the cost of adding the constraint to the constraint store.
- The `Drop` transition, i.e. ending the execution of a constraint.
- The `Default` transition, i.e. switching from trying one rule to trying the next rule.
- Matching for Herbrand variables and constants, given a bounded reference chain length. This occurs in the `Simplify` and `Propagate` transitions.
- Instantiating a variable that does not occur in any constraints, i.e. an obvious optimization of the `Solve` transition.
- Checking simple arithmetic built-in constraints like `>=` and `min`.

The following complexity assumptions can be realized in practice by appropriate indexing, i.e. constraint store lookup based on shared variables.

1. The cost of finding all constraints with a particular value in a particular argument position is constant. Even if there are no such constraints. The cost of obtaining one by one all constraints from such a set is is proportional to the size of the set.
2. The CHR constraint store allows constant addition and deletion of any constraint.
3. If more than one partner constraint has to be found, an ordering of look-ups is preferred, if possible, such that the next constraint to look-up shares a variable with the previously found constraints and the active constraint.

The last item is a heuristic presented in (Holzbaur et al. 2005) and implemented in the HAL and KULeuven CHR systems. In Section 7 we will discuss appropriate constraint store data structures that fulfill the remaining assumptions.

From these assumptions it is clear that processing a data constraint takes constant time: the constraint is called, some rules are tried, some partner constraints which share a variable with the active constraint are looked for, but none are present, and finally the call ends with inserting the data constraint into the constraint store.

Hence our naive CHR implementation has the same time complexity properties as the naive imperative algorithm.

The proof of operational equivalence and equivalent complexity of the optimized algorithm and CHR program is similar. Because of this equivalence with the imperative algorithm, our CHR program also has worst-case time complexity is $\mathcal{O}(M + Nlog(N))$ and amortized time complexity $\mathcal{O}(M + N\alpha(N))$.

7 Experimental Evaluation

To experimentally validate the derived complexity derived above, we have run the CHR program in SWI-Prolog (Wielemaker) using the K.U.Leuven CHR system (Schrijvers and Demoen 2004). This CHR system will use hashtables as constraint stores for lookups on shared variables that are ground. These hashtables allow for efficient lookup, insertion and deletion of constraints. By adding the appropriate mode declarations to our program, the system establishes the groundness of shared variables.

By initializing the hashtables to the appropriate sizes and choosing the used constants appropriately, it is possible to avoid hashtable collisions. Then, the hashtables essentially behave as arrays (just as in the imperative code) and the assumptions of the previous section are effectively realized.

In contrast, the first and de facto standard CHR system, available in SICStus (Intelligent Systems Laboratory 2003), does not provide the necessary constant time operations. While it does have constant lookup time for all constraint instances of a particular constraint that contain a particular variable, it does not distinguish between argument positions. Hence, the lookup of `root(X,R)` can be done in constant time given `X`, but the lookup of `X ~> Y` is proportional to the number of `~>` constraints `X` appears in. If `X` is a node with K children, then it will be $\mathcal{O}(K)$. Moreover, while the insertion of a constraint instance is $\mathcal{O}(1)$, deletion is $\mathcal{O}(I)$, where I is the total number of instances of the constraint.

The queries we use in our experimental evaluation consist of N calls to `make/1`, to create N different elements, followed by N calls to `union/2` and N calls to `find/2`. The input arguments of the latter two are chosen at random among the elements. Even the SICStus CHR system exhibits near-linear behavior for a random set of `union` operations. So we consider instead a contrived set of `union` operations: disjoint trees of elements are unioned pairwise until all elements are part of the same tree. Figures 1(a) and 1(b) show the runtime results for SICStus and SWI-Prolog. It is clear from the figure that SICStus does not show the optimal quasi-linear behavior anymore which is still observed in SWI-Prolog.

We also compare the above two cases to the case where the hashtables are not initialized to a large enough size, but instead double in size and rehash each time their load equals their size. While individual hashtable operations no longer take constant time, on average they do (Cormen et al. 1990), which is sufficient for our complexity analysis. This is confirmed by experimental evaluation (see Figure 1(c)).

The above comparisons illustrate that it is vital for efficiency to use a CHR system with the proper constraint store data structures. To the best of our knowledge, the K.U.Leuven CHR system is currently the only system that provides hashtable-based indexing constraint stores.

8 Conclusion

We have shown in this paper that it is possible to implement the classical union-find algorithm concisely and efficiently in constraint handling rules (CHR). The implementation is easily extended with optimizations like path-compression and union-by-rank. In addition, we showed the optimal time complexity properties of our implementations. The declarative nature of CHR is no compromise for time complexity.

At `http://www.cs.kuleuven.ac.be/~toms/Research/CHR/UnionFind/` all presented programs as well as related material are available for download. The programs can be run with the proper time complexity in the latest release of SWI-Prolog. The technical report (Schrijvers and Frühwirth 2004) associated with this paper contains a detailed analysis of the confluence properties and logical semantics of our union-find implementations.

In future work we intend to investigate implementations for other variants of the union-find algorithm. For a parallel version of the union-find algorithm parallel operational semantics of CHR have to be investigated (confluence may be helpful here). A dynamic version of the algorithm, e.g. where unions can be undone, would presumably benefit from dynamic CHR constraints as defined in (Wolf 2005).

Acknowledgements. We would like to thank the participants of the first workshop on CHR in May 2004 for raising our interest in the subject. Marc Meister and the students of the constraint programming course at the University of Ulm in summer 2004 helped by implementing and discussing their versions of the union-find algorithm. Part of this work was done while Tom Schrijvers was visiting the University of Ulm in November 2004. Last but not least we would like to thank our reviewers for their suggestions and comments that greatly helped to improve the paper.

References

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* MIT Press.

Duck, G. J., Stuckey, P. J., de la Banda, M. G., and Holzbaur, C. 2004. The Refined Operational Semantics of Constraint Handling Rules. In *Proceedings of the 20th International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds.

Frühwirth, T. 1998. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 95–138.

Frühwirth, T. 2002. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. In *Electronic Notes in Theoretical Computer Science*, A. D. Pierro and H. Wiklicky, Eds. Vol. 59. Elsevier.

Frühwirth, T. and Abdennadher, S. 2003. *Essentials of Constraint Programming.* Springer.

Galil, Z. and Italiano, G. F. 1991. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comp. Surveys 23,* 3, 319ff.

Ganzinger, H. and McAllester, D. 2001. A New Meta-Complexity Theorem for Bottom-Up Logic Programs. In *International Joint Conference on Automated Reasoning.* LNCS 2083. Springer, 514–528.

Holzbaur, C., García de la Banda, M., Stuckey, P. J., and Duck, G. J. 2005. Optimizing compilation of constraint handling rules in HAL. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules*. To appear.

Intelligent Systems Laboratory. 2003. *SICStus Prolog User's Manual.* PO Box 1263, SE-164 29 Kista, Sweden.

Sahlin, D. and Carlsson, M. 1991. Variable Shunting for the WAM. Tech. Rep. SICS/R-91/9107, SICS.

Schrijvers, T. and Demoen, B. 2004. The K.U.Leuven CHR system: Implementation and Application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, T. Frühwirth and M. Meister, Eds. Number 2004-01. ISSN 0939-5091.

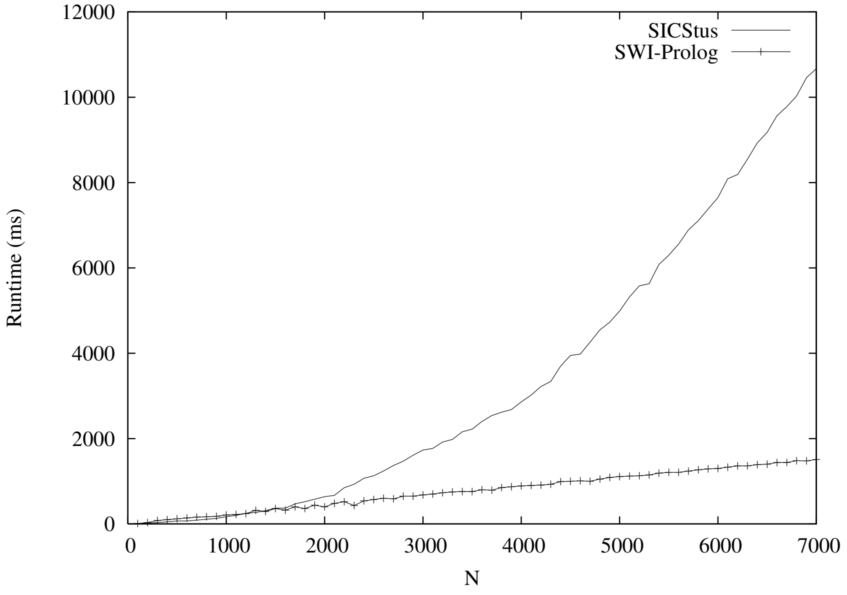
Schrijvers, T. and Frühwirth, T. 2004. Union-Find in CHR. Tech. Rep. CW389, Department of Computer Science, K.U.Leuven, Belgium. July.

Schrijvers, T. and Frühwirth, T. 2005. Analysing the CHR implementation of union-find. In *Proceedings of 19th Workshop on (Constraint) Logic Programming, Ulm, Germany.* To Appear.

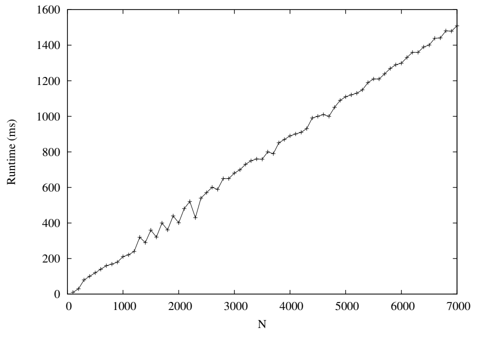
Tarjan, R. E. and van Leeuwen, J. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM 31,* 2, 245–281.

Wielemaker, J. SWI-Prolog. http://www.swi-prolog.org/.

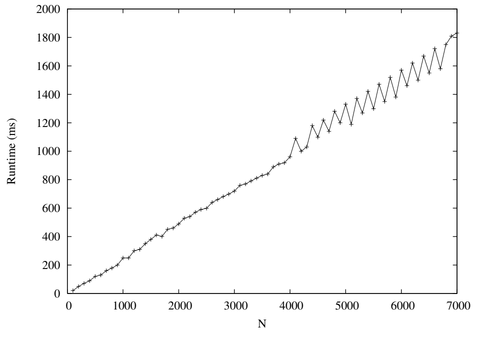
Wolf, A. 2005. Intellingent Search Strategies Based on Apdative Constraint Handling Rules. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules*. To appear.



(a) SICStus and SWI-Prolog Array



(b) Detail of Figure 1(a): SWI-Prolog Array



(c) SWI-Prolog Hashtable

Fig. 1. Observation of behavior for contrived unions.