

**Dijkstra's Algorithm
with Fibonacci Heaps:
An Executable Description in CHR**

Jon Sneyers, Tom Schrijvers, Bart Demoen

Report CW 429, November 2005



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Dijkstra's Algorithm with Fibonacci Heaps: An Executable Description in CHR

Jon Sneyers, Tom Schrijvers, Bart Demoen

Report CW429, November 2005

Department of Computer Science, K.U.Leuven

Abstract

We construct a readable, compact and efficient implementation of Dijkstra's shortest path algorithm and Fibonacci heaps using Constraint Handling Rules (CHR), which is increasingly used as a high-level rule-based general-purpose programming language. We measure its performance in different CHR systems, investigating both the theoretical asymptotic complexity and the constant factors realized in practice.

Keywords : Constraint Handling Rules, Shortest paths (single-source, directed, non-negative weights), Priority queues, Algorithms, Performance.

CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages; E.1 [Data Structures] Graphs; Queues; Arrays.

Dijkstra’s Algorithm with Fibonacci Heaps: An Executable Description in CHR

Jon Sneyers*, Tom Schrijvers**, Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
{jon,toms,bmd}@cs.kuleuven.be

Abstract. We construct a readable, compact and efficient implementation of Dijkstra’s shortest path algorithm and Fibonacci heaps using Constraint Handling Rules (CHR), which is increasingly used as a high-level rule-based general-purpose programming language. We measure its performance in different CHR systems, investigating both the theoretical asymptotic complexity and the constant factors realized in practice.

1 Introduction

Constraint Handling Rules (CHR) [12] is a high-level programming language extension based on multi-headed committed-choice rules. Originally designed for writing constraint solvers, it is increasingly used as a general-purpose programming language. We assume that the reader is familiar with CHR, referring to [12] for an overview.

Recently, we have shown [26] that every algorithm can be implemented in CHR with the best-known asymptotic time and space complexity. However, it remains an open problem whether classical algorithms can be implemented in CHR in an elegant and compact way. Also, the constant factor hidden behind the notion of asymptotic complexity could be so huge to be completely paralyzing in practice.

Earlier work by Schrijvers and Frühwirth [25] resulted in an elegant CHR implementation of the classical union-find algorithm which has the optimal asymptotic time complexity. In this paper, we construct a readable, compact and efficient CHR program (Section 3) which implements Dijkstra’s shortest path algorithm with Fibonacci heaps. We analyze its theoretical time complexity (Section 4) and experimentally compare its performance in different CHR systems against a low-level reference implementation in C (Section 5). Clearly, low-level implementations will always be faster, at the cost of a more painful development process and less readable and adaptable code.

In this paper, we hope to demonstrate that CHR can be used as ‘*executable pseudo-code*’ for studying and designing algorithms while constructing a real program with the desired time (and space) complexity. We also investigate the

* This work was partly supported by projects G.0144.03 and G.0160.02 funded by the Research Foundation - Flanders (F.W.O.-Vlaanderen).

** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

constant factor separating CHR from low-level languages, and propose a more efficient type of constraint stores for indexing on ground integer arguments, improving the constant factor (for our program) by 35% or more.

This technical report is an extended version of [28].

2 The single-source shortest path problem

The shortest path problem [32] is one of the most basic, and most studied, problems in algorithmic graph theory. It appears as a sub-problem in many graph related algorithms, such as network optimization algorithms.

Given a weighted directed graph $G = (V, E)$ and a source node $s \in V$, we are interested in the distances from s to all other nodes of the graph. This is called the *Single-Source Shortest Path (SSSP)* problem. In the rest of this paper we will use $n = |V|$ to denote the number of nodes and $m = |E|$ to denote the number of edges. We assume the weights to be nonnegative, but not necessarily integer. We also assume that the graph is connected, so n is $O(m)$.

The SSSP problem with nonnegative edge weights can be solved using Dijkstra's classical algorithm [8]. A naive implementation of this algorithm runs in $O(n^2)$ time, which is suboptimal for non-dense graphs.

Priority queues. Efficient implementations of Dijkstra's algorithm use a *priority queue* to hold tentative distances. A priority queue is a data structure consisting of a set of item-key pairs, subject to the following operations: *insert*, adding a new pair to the queue; *extract-min*, returning and removing the pair with the minimum key; and *decrease-key*, decreasing the key of a given item.

Fibonacci heaps implement insert and decrease-key in $O(1)$ amortized time, and extract-min in $O(\log n)$ amortized time, where n is the number of elements in the queue. They are based on binomial queues [29], which implement all operations in $O(\log n)$ time. Using Fibonacci heaps [11], Dijkstra's algorithm takes $O(m + n \log n)$ time. This combination is still the fastest known algorithm for solving the SSSP problem with non-negative real edge weights.

3 An executable description of the algorithms

In this section, we give a high-level description of Dijkstra's algorithm with Fibonacci heaps. Instead of using pseudo-code, we describe the algorithms using CHR rules, which are directly executable in any CHR system. In Section 5 we will discuss its performance in different systems.

All recent CHR systems implement the refined operational semantics [9], which implies that the rules are tried in textual order. Correctness, termination, and complexity of the program crucially depend on this execution strategy.

3.1 Dijkstra's algorithm

The input graph is given as m `edge/3` constraints: a (directed) edge from node `A` to `B` with weight W is represented as `edge(A,B,W)`. Node names are integers in $[1, n]$ and weights are non-negative numbers. The query consists of a sequence of `edge/3` constraints followed by one `dijkstra(S)` constraint, where `S` is the source node. The output of the algorithm consists of n `distance(X,D)` constraints, where `D` is the distance from node `S` to node `X`.

During the execution of Dijkstra's algorithm, each node is in one of three states: *unlabeled*, *labeled* (with a tentative distance), or *scanned* (and annotated with a correct distance). Initially, every node is unlabeled except for the source node which gets tentative (and correct) distance zero. The algorithm repeatedly picks a labeled node with the smallest tentative distance and scans it.

We use a global priority queue to store the labeled nodes; the item is the node name, the key is its tentative distance. When a node `X` is scanned, we store its distance `D` as a `distance(X,D)` constraint. The unlabeled nodes are not stored explicitly. We assume that the priority queue has a `decr_or_ins(Item,Key)` operation which inserts `(Item,Key)` into the queue if `Item` is not already in the queue; otherwise it updates the key of `Item` if the new key `Key` is smaller than the original key (if it is larger, the operation has no effect). We also assume that there is an `extract_min/2` operation which returns the item with the smallest key and removes it from the queue (and fails when the queue is empty).

The distance to the source node is zero. We start by scanning the source:

```
start_scanning @ dijkstra(A) <=> scan(A,0).
```

To scan a node, all its outgoing edges are examined to relabel the neighbors. The candidate label for a neighbor `N2` is simply the sum of the distance `L` to `N`, and the weight W of the edge connecting `N` and `N2`:

```
label_neighb @ scan(N,L), edge(N,N2,W) ==> L2 is L+W, relabel(N2,L2).
```

Then we make the node scanned and scan the labeled node with the smallest tentative distance (or stop if there are no labeled nodes left):

```
scan_next @ scan(N,L) <=> distance(N,L),  
            (extract_min(N2,L2) -> scan(N2,L2) ; true).
```

Relabeling works as follows: we do nothing if the neighbor is already scanned. If the neighbor node `N` is not scanned, it is either unlabeled or labeled. Using the `decr_or_ins` operation, `N` gets a label if it is unlabeled and gets a new label if the candidate label `L` is smaller than its original label.

```
scanned @ distance(N,_) \ relabel(N,_) <=> true.  
not_scanned @ relabel(N,L) <=> decr_or_ins(N,L).
```

The full program text is given in Figure 1.

3.2 Fibonacci heaps

We use Fibonacci heaps to implement the priority queue needed in Dijkstra's algorithm.

Data structures. The items and their keys are stored as nodes in a collection of *heap-ordered* trees. A heap-ordered tree is a rooted tree where the key of any node is no less than the key of its parent. The number of children of a node is called its *rank*. Nodes can be either *marked* or *unmarked*. Root nodes are never marked.

A node containing the item I with key K is stored as an `item(I,K,R,P,M)` constraint, where R is its rank and P is the item of its parent (or 0 if it is a root). The last argument M is `u` if the node is unmarked and `m` if it is marked. The minimum-key pair is stored as a `min/2` constraint:

```
keep_min @ min(_,A) \ min(_,B) <=> A =< B | true.
```

Insert. Inserting a new item I with key K is done by adding an unmarked isolated root node and updating the minimum:

```
insert @ insert(I,K) <=> item(I,K,0,0,u), min(I,K).
```

Extract-min. Extracting the minimum node is done as follows. First we find and remove the `min` constraint (if there is none, the heap is empty and we fail) and the corresponding `item`. Then we convert the children of the minimum node to roots, which is done by the auxiliary `ch2rt` constraint. Finally we find the new minimum (done by the `findmin` constraint) and return the (old) minimum item.

```
extr @ extract_min(X,Y), min(I,K), item(I,_,_,_,_)  
      <=> ch2rt(I), findmin, X=I, Y=K.  
extr_empty @ extract_min(_,_) <=> fail.
```

The following simpagation rule converts the children of I to roots:

```
c2r @ ch2rt(I) \ item(C,K,R,I,_) <=> item(C,K,R,0,u).  
c2r_done @ ch2rt(I) <=> true.
```

To find the new minimum, it suffices to search the root nodes:

```
findmin @ findmin, item(I,K,_,0,_) ==> min(I,K).  
foundmin @ findmin <=> true.
```

We want to make sure that the number of roots is $O(\log n)$ (where n is the number of items). The following rule links trees whose roots have the same rank, reducing the number of roots:

```
same_rank @ item(I1,K1,R,0,_) , item(I2,K2,R,0,_) <=> K1 =< K2 |  
              R1 is R+1, item(I2,K2,R,I1,u), item(I1,K1,R1,0,u).
```

Decrease-key. The decrease-key operation removes the original `item` constraint and calls the auxiliary constraint `decr/5` if the new key is smaller than the original key (and fails otherwise).

```
decr @ decr(I,K), item(I,0,R,P,M) <=> K < 0 | decr(I,K,R,P,M).
decr_nok @ decr(I,K) <=> fail.
```

The `decr_or_ins` operation calls `decr/5` if the item is on the heap and the new key is smaller than the original. If the item is on the heap but the new key is larger, it does nothing; the item is inserted if it is not on the heap.

```
doi_d @ item(I,0,R,P,M), decr_or_ins(I,K) <=> K < 0 | decr(I,K,R,P,M).
doi_nop @ item(I,0,_,_,_) \ decr_or_ins(I,K) <=> K >= 0 | true.
doi_insert @ decr_or_ins(I,K) <=> insert(I,K).
```

When a key is decreased, we may have found a new minimum:

```
d_min @ decr(I,K,_,_,_) ==> min(I,K).
```

Decreasing the key of a root cannot cause a violation of the heap order:

```
d_root @ decr(I,K,R,0,_) <=> item(I,K,R,0,u).
```

If the new key is not smaller than the parent's key, there is also no problem:

```
d_ok @ item(P,PK,_,_,_) \ decr(I,K,R,P,M) <=> K >= PK | item(I,K,R,P,M).
```

Otherwise, we cut the violating node and make it a new root. The original parent is marked to indicate that it has lost a child.

```
d_prob @ decr(I,K,R,P,M) <=> item(I,K,R,0,u), mark(P).
```

To obtain the desired time complexity, we have to make sure that when a (non-root) node loses two of its children through cuts, it gets cut as well. This is called a *cascading cut*. Nodes are marked to keep track of where to make cascading cuts. The `mark` constraint decreases the rank of a node and marks it if necessary. A root node is never marked, an unmarked node becomes marked, and an already marked node is cut and its parent is marked (cascading cut):

```
m_rt @ mark(I), item(I,K,R,0,_) <=> item(I,K,R-1,0,u).
m_u @ mark(I), item(I,K,R,P,u) <=> item(I,K,R-1,P,m).
m_m @ mark(I), item(I,K,R,P,m) <=> item(I,K,R-1,0,u), mark(P).
```

This concludes the algorithm. Figure 2 lists the full CHR program. For optimization, a `pragma passive` compiler directive was added to rule `c2r`, the `m_er` rule was added and the `same_rank` rule was reformulated without guard.

```

----- dijkstra.chr -----
:- module(dijkstra,[edge/3,dijkstra/1]).
:- use_module(library(chr)).
:- use_module(fib_heap).
:- constraints
    edge(+dense_int,+int,+number),      distance(+dense_int,+number),
    dijkstra(+int),                      scan(+int,+number),      relabel(+int,+number).

start_scanning @ dijkstra(A) <=> scan(A,0).
label_neighb @ scan(N,L), edge(N,N2,W) ==> L2 is L+W, relabel(N2,L2).
scan_next    @ scan(N,L) <=> distance(N,L),
              (extract_min(N2,L2) -> scan(N2,L2) ; true).
scanned      @ distance(N,_) \ relabel(N,_) <=> true.
not_scanned @ relabel(N,L) <=> decr_or_ins(N,L).

```

Fig. 1. Dijkstra's algorithm, implemented in CHR.

```

----- fib_heap.chr -----
:- module(fib_heap,[insert/2,extract_min/2,decr/2,decr_or_ins/2]).
:- use_module(library(chr)).
:- constraints
    insert(+int,+number),  extract_min(?int,?number),  mark(+int),
    decr(+int,+number),    decr_or_ins(+int,+number),  ch2rt(+int),
    decr(+int,+number,+int,+int,+mark),  min(+int,+number),
    item(+dense_int,+number,+int,+dense_int,+mark),  findmin.
:- chr_type mark ---> m ; u.

insert @ insert(I,K) <=> item(I,K,0,0,u), min(I,K).

keep_min @ min(_,A) \ min(_,B) <=> A =< B | true.

extr      @ extract_min(X,Y), min(I,K), item(I,_,_,_)
           <=> ch2rt(I), findmin, X=I, Y=K.
extr_empty @ extract_min(.,.) <=> fail.

c2r      @ ch2rt(I) \ item(C,K,R,I,_)#X <=> item(C,K,R,0,u) pragma passive(X).
c2r_done @ ch2rt(I) <=> true.

findmin  @ findmin, item(I,K,_,0,_) ==> min(I,K).
foundmin @ findmin <=> true.

same_rank @ item(I1,K1,R,0,_) , item(I2,K2,R,0,_)
           <=> R1 is R+1, (K1 < K2 -> item(I2,K2,R,I1,u), item(I1,K1,R1,0,u)
           ; item(I1,K1,R,I2,u), item(I2,K2,R1,0,u)).

decr     @ decr(I,K), item(I,0,R,P,M) <=> K < 0 | decr(I,K,R,P,M).
decr_nok @ decr(I,K) <=> fail.

doi_d    @ item(I,0,R,P,M), decr_or_ins(I,K) <=> K < 0 | decr(I,K,R,P,M).
doi_nop  @ item(I,0,_,_,_) \ decr_or_ins(I,K) <=> K >= 0 | true.
doi_insert @ decr_or_ins(I,K) <=> insert(I,K).

d_min    @ decr(I,K,_,_,_) ==> min(I,K).
d_root   @ decr(I,K,R,0,_) <=> item(I,K,R,0,u).
d_ok     @ item(P,PK,_,_,_) \ decr(I,K,R,P,M) <=> K >= PK | item(I,K,R,P,M).
d_prob   @ decr(I,K,R,P,M) <=> item(I,K,R,0,u), mark(P).

m_rt    @ mark(I), item(I,K,R,0,_) <=> R1 is R-1, item(I,K,R1,0,u).
m_m     @ mark(I), item(I,K,R,P,m) <=> R1 is R-1, item(I,K,R1,0,u), mark(P).
m_u     @ mark(I), item(I,K,R,P,u) <=> R1 is R-1, item(I,K,R1,P,m).
m_er    @ mark(I) <=> writeln(error_mark), fail.

```

Fig. 2. Fibonacci heap, implemented in CHR.

4 Time Complexity

To obtain the desired $O(m + n \log n)$ time complexity, we depend on some CHR implementation properties, similar to the assumptions in Section 6.2 of [25]. Specifically, we require constraint stores to allow constant time insertion, deletion, and look-up on arbitrary argument positions.

4.1 Efficiency of constraint stores.

We have added type and mode declarations to allow the use of more efficient constraint store data structures, satisfying the above assumption. Optional mode declarations were first introduced in [25]. We use the extended constraint declaration syntax proposed in [27], as listed in Figures 1 and 2. Since all stored constraints are (declared) ground, allowing constant time look-ups using hash-tables constraint stores, the K.U.Leuven CHR system [23] realizes all of the above properties.

The crucial argument positions for which constant time look-ups are necessary are the first argument of `edge/3`, the first argument of `distance/2`, and the first and fourth argument of `item/5`. Constant time look-ups on these arguments can be implemented using hash-table constraint stores. We can improve the constant factors of the constraint store operations by using plain array-based constraint stores instead: these arguments are nonnegative integers that can be directly used as array positions, avoiding collisions and hash value computations.

We have implemented an additional built-in type, called `dense_int`. Look-ups on ground arguments of this type are compiled to array constraint stores. The space usage of an array constraint store is not optimal if the array is sparse, hence the name of the type: it is intended for storing $\Omega(k)$ (asymptotic lower bound) constraints whose argument of type `dense_int` has values in $[0, k]$.

4.2 Complexity of `dijkstra.chr`.

Dijkstra’s algorithm starts with an empty heap, performs $n - 1$ insert operations, $m - n$ decrease-key operations and n extract-min operations. Because we have constant-time access to the list of outgoing edges from a given node in rule `label_neighb`, the total time complexity of the algorithm is $O(nI + mD + nE)$, where I , D and E are the amortized time complexities of one insert, decrease-key and extract-min operation, respectively.

4.3 Complexity of `fib_heap.chr`.

Under the above assumptions, the analysis of [11] remains valid. Insert and decrease-key take constant amortized time, and extract-min takes logarithmic amortized time.

In the original description of Fibonacci heaps [11], the *linking step* (corresponding to the `same_rank` rule) is only performed during an extract-min operation (corresponding to the `extr` rule), just before the finding the new minimum

(`findmin`). In the variant presented here, the `same_rank` rule is triggered each time a node becomes a root or a new root is added. This does not affect the amortized time bounds, but it does improve performance in practice.

To analyze the amortized running times of the Fibonacci heap operations, we assign a *potential* to every possible heap configuration. The amortized time of an operation is its actual running time plus the net increase it causes in the potential. Hence, the actual time of a sequence of operations is equal to the total amortized time plus the total net decrease in potential. We define the potential of a heap to be the total number of trees it contains plus twice the number of marked nodes. The initial potential is zero, and the potential is always nonnegative, so the total amortized time of a sequence of operations is an upper bound on the total actual time.

Corollary 1 in [11] states that in a Fibonacci heap, nodes of rank k have at least F_{k+2} descendants, where F_i is the i -th Fibonacci number. Because the Fibonacci numbers grow exponentially, this result implies that in a Fibonacci heap with N items, the maximal rank of any node is $O(\log N)$, and that the number of roots is also $O(\log N)$ if no two roots have the same rank. In the worst case, there can be $n - 1$ items in the heap, so we get a $O(\log n)$ bound on the rank and number of roots. Performing one link (i.e. finding a partner constraint for the `same_rank` rule and executing its body) takes constant actual time, and since one root node becomes a child node, the potential decreases by one.

Insert. The insert operation adds a new root node, increasing the potential by one minus the number of links performed. This takes actual time proportional to the number of links performed. It may then trigger the `keep_min` rule, which takes constant time. Hence the amortized time for the insert operation is $O(1)$.

Extract-min. The extract-min operation either fails in constant time, or it finds the minimum item in constant time. In the latter case, it first converts $O(\log n)$ children to roots, which takes $O(\log n)$ amortized time. The potential increases by the number of new roots minus twice the number of children that were marked. This is clearly bounded by $O(\log n)$. Then `findmin` goes through the $O(\log n)$ new roots to find the new minimum. Hence the amortized time for the extract-min operation is $O(\log n)$.

Decrease-key. If decreasing the key of an item does not violate heap order, the decrease-key operation does not affect the potential and clearly takes only constant time. In the other case, a new root is added (adding one to the potential), followed by a number of cascading cuts (rule `m_m`, which decreases the potential by one, since it adds a root (+1) and unmarks a previously marked node (-2)), and finally marking an unmarked node (`m_u`, adding two to the potential) or doing nothing (`m_rt`). So the net increase of the potential is at most three minus the number of cascading cuts. Since every cut can be done in constant time, the

actual time is proportional to the number of cascading cuts. Hence the amortized time for the decrease-key operation is $O(1)$. Clearly, the “decrease-key or insert” operation also takes constant amortized time.

From the above it follows that the total time complexity of our CHR implementation of Dijkstra’s algorithm with Fibonacci heaps is $O(m + n \log n)$.

5 Experimental results

Setup. We have tested our program on sparse graphs consisting of a Hamiltonian cycle of n edges with weight 1 from node i to node $i + 1$ (and node n to node 1) and $3n$ random weight edges, 3 from every node to some randomly chosen other node. Such graphs essentially correspond to the “Rand-4” family of [3]. All tests were performed on a Pentium 4 (1.7 GHz) machine with 512 Mb RAM running Debian GNU/Linux (kernel version 2.6.8) with a low load. They can be downloaded at the K.U.Leuven CHR system website [24].

The following CHR systems were used: the K.U.Leuven CHR system [23] in SWI-Prolog 5.5.31 [31] and hProlog 2.4.12-32 [7], the K.U.Leuven JCHR system 1.0.3 [30] (Java 1.5.0), and the reference CHR implementations [16, 15] in SICStus 3.12.2 [2] and YAP 5.0.0 [6].

In the SICStus and YAP versions we have inserted extra `pragma passive` directives to avoid redundant rule trials. They are detected and added automatically by the optimizing K.U.Leuven CHR compilers.

To get an idea of the constant factor penalty incurred in using a very high level language like CHR instead of a low-level language, we have also measured the performance of `dikf`, an efficient C implementation of the Dijkstra algorithm with Fibonacci heaps. It is part of SPLIB 1.4 [3].

In SICStus and YAP, types and modes of constraint arguments cannot be declared. In JCHR, type and mode declarations are obligatory. In the K.U.Leuven CHR system, they are optional. Three versions of the SWI-Prolog and hProlog CHR program were considered: one without any type and mode declaration; one with type and mode declarations, but without using the new `dense_int` type (“...+type/mode”); and one with the declarations as in Figures 1 and 2 (“...+type/mode+array”).

The results are tabled in Figure 3 and plotted in Figure 4.

Asymptotic behavior. Without type/mode declarations, the program exhibits a quadratic time complexity, caused by using general data structures which do not allow constant time look-ups. When type and mode information is available, the optimal $O(n \log n)$ time complexity is achieved.

Constant factors. Without mode declarations, the SWI-Prolog version is about 5 to 8 times slower than the the hProlog version, and about 3 to 4 times

n	C	hProlog CHR			SWI-Prolog CHR			JCHR	YAP	SICStus
	(SPLIB)	t/m+a	t/m	none	t/m+a	t/m	none		CHR	CHR
256		0.02	0.03	0.33	0.12	0.21	1.43	0.08	0.35	0.52
512		0.05	0.07	1.00	0.26	0.44	5.43	0.23	1.40	1.91
1k		0.10	0.17	3.33	0.57	1.03	21.61	0.60	5.94	6.42
2k	< 0.01	0.22	0.36	12.04	1.16	2.08	88.95	1.32	24.85	25.38
4k	0.01	0.47	0.76	46.61	2.45	4.27	360.88	2.55	96.77	96.89
8k	0.02	0.97	1.53	190.94	5.10	8.68	<i>time</i>	5.42	350.18	373.48
16k	0.07	1.99	3.18	780.10	10.48	18.13		<i>mem</i>	<i>time</i>	<i>time</i>
32k	0.17	4.07	6.46	<i>time</i>	21.50	37.12				
64k	0.42	8.35	13.05		43.34	76.04				
128k	0.94	17.13	26.51		89.30	152.76				
256k	2.06	35.44	54.92		<i>mem</i>	<i>mem</i>				

Fig. 3. Comparing the performance of the Dijkstra CHR program with Fibonacci heaps, on random sparse graphs, using different CHR systems. Programs were aborted after 1000 seconds (“*time*”); fatal stack or heap overflows are indicated with “*mem*”.

slower than the SICStus and YAP versions. These differences are largely explained by differences in the underlying Prolog systems (see e.g. Appendix B in [22]).

The K.U.Leuven JCHR version (which has type and mode declarations but no array constraint store) is about 1.6 times faster than the corresponding SWI-Prolog version and about 3.5 times slower than the corresponding hProlog version. However, the generated Java code already ran out of memory for modestly sized input graphs of 16k nodes, most likely because of garbage collection issues.

Using arrays instead of hash-tables (when possible) improves performance by about 35% in hProlog and about 40% in SWI-Prolog.

In this test, the fastest CHR system clearly is the K.U.Leuven CHR system in hProlog. The gap between the hProlog CHR program and the SPLIB implementation is a constant factor of less than 20. Major reasons for this gap are data structure overhead (using Prolog terms to represent CHR constraint stores) and the overhead of interpreting WAM code.

6 Conclusion

We have presented a readable, compact, and efficiently executable CHR description of Dijkstra’s algorithm with Fibonacci heaps. We have analyzed it theoretically and experimentally – in several CHR systems – to investigate its time complexity.

The Fibonacci heap data structure is quite complex, and it is rather difficult to implement correctly. Implementations in imperative languages (e.g. [3]) typically take at least some 300 lines of hard-to-understand code. Even the pseudo-code description of Fibonacci heaps given in [4] is 63 lines long. In contrast, the

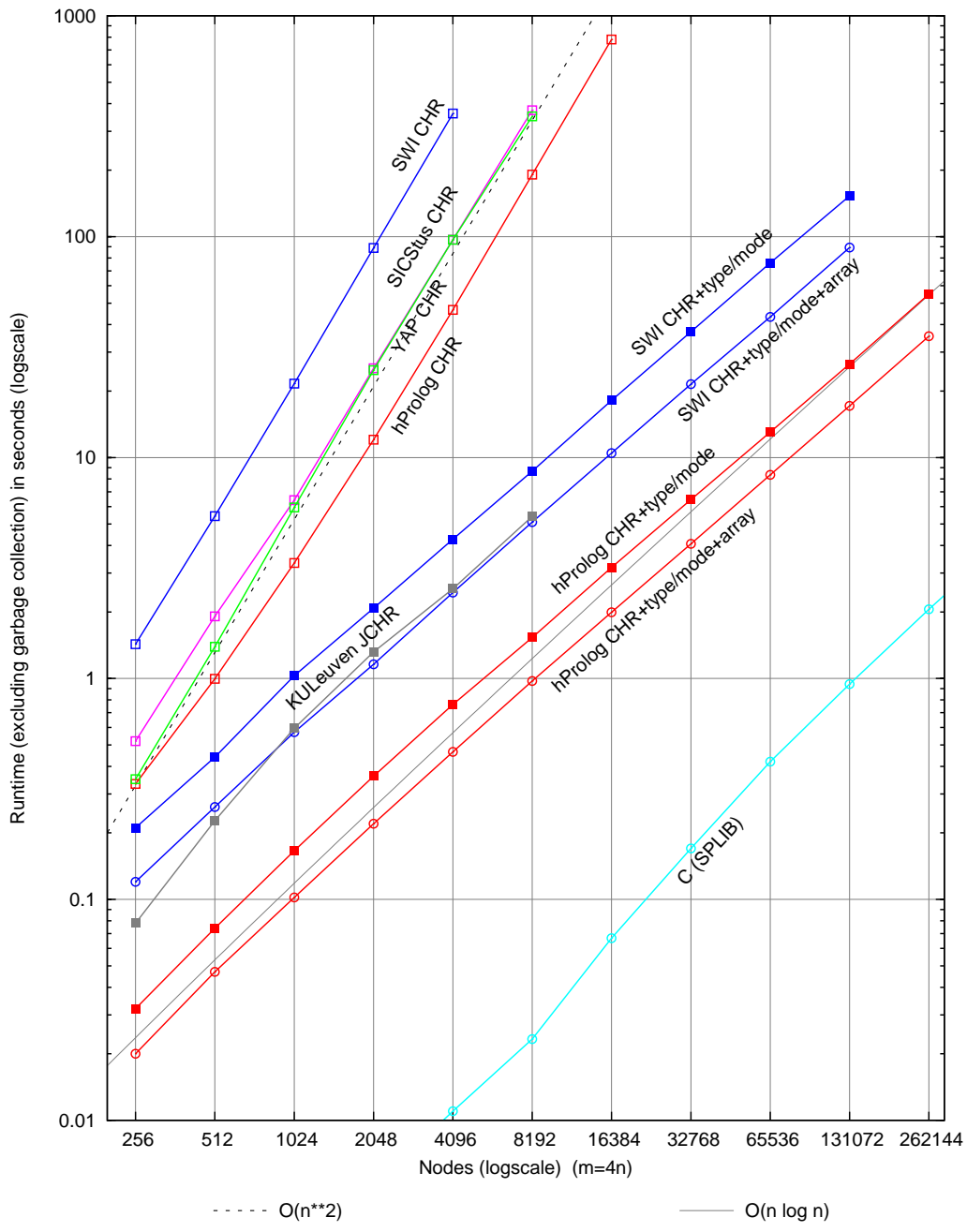


Fig. 4. Plot of the results tabled in Figure 3.

program we have constructed is directly executable and consists of just 22 CHR rules, or about 40 lines including declarations and whitespace.

6.1 Related work.

As far as we know, this is the first implementation of Fibonacci heaps in a declarative language.

King [17] constructed a functional implementation of the simpler and asymptotically slower binomial queues [29], using about 45 lines of Haskell code (for the operations needed in Dijkstra’s algorithm). Okasaki [19, 20] and Brodal [1] constructed functional implementations of many variants of priority queues, including binomial queues (in 36 lines of Standard ML code plus 17 signature lines), leftist heaps [5], and pairing heaps [10] (but not Fibonacci heaps), but they all lack the *decrease-key* operation. They conclude [1] that further research is needed to implement the *decrease-key* operation efficiently in a (purely) functional setting without increasing the bounds for the other operations.

Schoenmakers [21] uses a functional notation to aid in the analysis of a number of data structures, including Fibonacci heaps. He uses a formal functional notation extended with imperative features, such as pointer and array operators. The emphasis is on formally proving correctness and amortized complexity bounds.

King notes in [17] that Fibonacci heaps do not lend themselves to a natural functional encoding, because of their heavy usage of pointers. Imperative implementations of Fibonacci heaps usually store at least four pointers with every item in the heap: parent, left and right sibling, and one of the children. They are used for efficient access to the children of a particular node. In CHR, there is no need to store all these pointers explicitly, since the compiler automatically constructs the appropriate indexes. For example, the `c2r` rule does a look-up on the fourth argument of `item` to find the children of a node. In the code generated by the CHR compiler, a hash-table (or array) index on this argument is maintained.

McAllester introduced a pure logic programming algorithmic model [18], which was extended with rule priorities and deletion [13]. Generalizing this model to include rules with a variable priority, Ganzinger and McAllester construct a very compact implementation of Dijkstra’s algorithm in just three rules [14]. They theoretically construct an interpreter on a RAM machine for their logic programming model which can run their implementation of Dijkstra’s algorithm in $O(m \log m)$ time, which is worse than the $O(m + n \log n)$ implementation presented in this paper. However, they do not provide an implementation of Fibonacci heaps (or any other priority queue) in their logic programming formalism. The priority queue used in Dijkstra’s algorithm is not explicitly implemented: it is hidden in the variable priority of the neighbor (re-)labeling rule. In their theoretical construction of the interpreter, they suggest using Fibonacci heaps to implement variable priority rules.

6.2 Future work.

We consider CHR to be one of the most suitable languages to describe – and design – algorithms. It allows significantly more compact and readable formulations of algorithms that focus on the high-level structure, since low-level implementation issues like efficiency of look-ups are automatically handled by the CHR compiler. Using the optional type and mode declarations, the CHR program can be compiled to efficient Prolog (or Java) code which has the desired asymptotic time complexity, with a constant factor only about one order of magnitude worse than that of hand-crafted specialized low-level implementations.

We are confident that it is possible – and an interesting challenge – to further improve the constant factor by generating more specialized and inlined code. Another interesting idea would be to implement a CHR system for the host-language C, perhaps by using ideas from the Java CHR systems. The combination of both would allow high-level algorithm descriptions in CHR to be only marginally slower than direct low-level imperative implementations.

References

1. Gerth S. Brodal and Chris Okasaki. Optimal purely functional priority queues. *J. Functional Programming*, 6(6):839–857, 1996.
2. Mats Carlsson et al. The SICStus Prolog home page. <http://www.sics.se/sicstus/>.
3. Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996. SPLIB software: <http://www.avglab.com/andrew/soft.html>.
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.
5. Clark A. Crane. *Linear Lists and Priority Queues as Balanced Binary Trees*. PhD thesis, Stanford University, 1972.
6. Luís Damas, Vítor Santos Costa, Rogério Reis, and Rúben Azevedo. *YAP User's Manual*. Universidade do Porto. Home page at <http://www.ncc.up.pt/vsc/Yap/>.
7. Bart Demoen. The hProlog home page. <http://www.cs.kuleuven.ac.be/~bmd/hProlog/>.
8. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(4):269–271, 1959.
9. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Proc. 20th Intl. Conference on Logic Programming (ICLP'04)*, pages 90–104, St-Malo, France, September 2004.
10. Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
11. Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
12. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
13. Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. 1st Intl. Joint Conference on Automated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, 2001.

14. Harald Ganzinger and David A. McAllester. Logical algorithms. In *Proc. 18th Intl. Conference on Logic Programming (ICLP'02)*, pages 209–223, Copenhagen, Denmark, 2002.
15. Christian Holzbaur and Thom Frühwirth. CHR reference manual. Technical Report TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1998.
16. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue J. Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), 2000.
17. David J. King. Functional binomial queues. In *Proc. Glasgow Workshop on Functional Programming*, Ayr, Scotland, 1994.
18. David A. McAllester. The complexity analysis of static analyses. In *Proc. 6th Intl. Symposium on Static Analysis (SAS'99)*, pages 312–329, Venice, Italy, 1999.
19. Chris Okasaki. Functional data structures. In *Advanced Functional Programming*, pages 131–158, 1996.
20. Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.
21. Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis.
22. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
23. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *Selected Contributions, 1st Workshop on Constraint Handling Rules*, Ulm, Germany, May 2004.
24. Tom Schrijvers et al. The K.U.Leuven CHR system home page, 2005. <http://www.cs.kuleuven.ac.be/~toms/Research/CHR/>.
25. Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 2005. To appear.
26. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *Proc. 2nd Workshop on Constraint Handling Rules (CHR'05)*, pages 3–17, Sitges, Spain, October 2005.
27. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and continuation optimization for occurrence representations of CHR. In *Proc. 21st Intl. Conference on Logic Programming (ICLP'05)*, pages 83–97, Sitges, Spain, October 2005.
28. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming (WLP'06)*, Vienna, Austria, February 2006. Submitted.
29. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.
30. Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *Proc. 2nd Workshop on Constraint Handling Rules (CHR'05)*, pages 47–62, Sitges, Spain, October 2005.
31. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebrenik, editors, *Proc. 13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003. Home page at <http://www.swi-prolog.org>.
32. Uri Zwick. Exact and approximate distances in graphs – a survey. In *Proc. 9th European Symposium on Algorithms (ESA'01)*, pages 33–48, Århus, Denmark, 2001.