A Generic Program for Minimal Subsets With Applications

Rudolf Berghammer

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel, Germany

Abstract. We formally develop a generic program which computes a minimal subset satisfying a certain property of a given set. To improve the efficiency of instantiations, refinements are investigated. Finally, instantiations are presented which correspond to the solution of well-known graph-theoretic problems and some further applications are sketched.

1 Introduction

Besides the computation of minimum resp. maximum subsets (with respect to size) of a given universe U the computation of minimal resp. maximal subsets (with respect to inclusion) of U is an important optimization problem, too. On the one hand, a given problem often directly demands a minimal resp. maximal subset. Examples are spanning forests and vertex bases of graphs. In the first case maximal cycle-free (or, equivalently, minimal connected) sets of edges are searched for and in the second case the task is to compute a minimal set of vertices from which all other vertices can be reached. Furthermore, minimal resp. maximal subsets are frequently used as starting points for the computation of minimum resp. maximum subsets in order to accelerate the latter. Matchings (these are sets M of edges of graphs such that for all vertices x at most one edge of M is incident on x) are a good example. To compute a maximum matching, normally a maximal matching is first computed in a very simple and efficient way. Only then the relatively large-scaled process of the iterative search for so-called augmenting alternating chains to receive a maximum matching (introduced in [5]) is started. Finally, minimal resp. maximal subsets also are used as approximations if an efficient computation of the respective minimum and maximum subsets seems to be impossible because of complexity-theoretic reasons. This might be the case for transitive reductions of graphs g, which are minimal subsets of the edges that maintain all reachability relations between vertices of g. The original interest is laid on smallest subsets of the edges which lead to the same reflexive-transitive closure as g. The subgraphs of g induced by such subsets are called minimum equivalent digraphs. Obviously, the interest is frequently limited to transitive reductions only because of the NP-completeness of the minimum equivalent digraph problem.

Since minimal and maximal subsets can be treated in a dual way, in this article we focus upon the first case. Similar to [17], first we formally develop in

Section 2 through the invariant technique a generic program which computes for a given finite set U and a specific predicate $\mathcal{P}$ on the powerset 2^U a minimal subset of U for which $\mathcal{P}$ holds. In the same section we also have a look at refinements of this program, which allow in many cases to improve the efficiency of a concrete instantiation. Then Section 3 demonstrates such instantiations to solve two graph-theoretic problems, viz. the approximation of a minimum vertex cover and the computation of a transitive reduction resp. the approximation of a minimum equivalent digraph. We have implemented these algorithms since we believe that for a final assessment of their practical use there is a definite need to supplement the theoretical worst-case analysis with experiments. Some experimental results are presented in Section 3. They also contain a comparison with two algorithms known from the literature which, finally, leads to their improvement in practice. Section 4 contains some concluding remarks.

2 Generic Computation of Minimal Subsets

Let U be a finite set and $\mathcal{P}$ a predicate on its powerset 2^U. We assume that $\mathcal{P}$ is *upwards-closed*, which means that for all sets $X, Y \in 2^U$

$$X \subseteq Y \wedge \mathcal{P}(X) \implies \mathcal{P}(Y)\,. \tag{1}$$

Note that (1) implies the negation $\neg\mathcal{P}$ of $\mathcal{P}$ to be *downwards-closed*, i.e., for all sets $X, Y \in 2^U$ if $Y \supseteq X$ and $\mathcal{P}(Y)$ is false, then also $\mathcal{P}(X)$ is false.

2.1 The Program Development

Assuming property (1), in the following we shall systematically derive a generic imperative program that computes – for U as its input – a minimal subset of U that satisfies the predicate $\mathcal{P}$. If we use the variable A of type 2^U as the program's output, then a formal specification of the requirements on A is given by the postcondition

$$\mathsf{post}(A) \;:\Longleftrightarrow\; \mathcal{P}(A) \wedge \forall\, X \in 2^A : \mathcal{P}(X) \to X = A\,.$$

For the derivation of the program we combine the invariant technique with set-theoretic and logical calculations. Doing so, for a subset $X \subseteq U$ and an element $x \in U$ we write $X \ominus x$ instead of $X \setminus \{x\}$ and $X \oplus x$ instead of $X \cup \{x\}$ to enhance readability. Heading towards a program with an initialization followed by a while-loop, the formal derivation is carried out in three stages.

First we have to develop a loop invariant. Here we follow the most commonly used technique of generalizing the postcondition (for details and many examples we refer to the textbooks [4, 6]). The corresponding calculation introduces a new variable B of type 2^U and looks as follows:

$$\begin{array}{ll} & \mathsf{post}(A) \\ \Longleftrightarrow & \mathcal{P}(A) \wedge \forall\, X \in 2^A : \mathcal{P}(X) \to X = A \\ \Longleftrightarrow & \mathcal{P}(A) \wedge \forall\, X \in 2^A : X \neq A \to \neg\mathcal{P}(X) \\ \Longleftrightarrow & \mathcal{P}(A) \wedge \forall\, x \in A : \neg\mathcal{P}(A \ominus x) \\ \Longleftarrow & \mathcal{P}(A) \wedge B = \emptyset \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\,. \end{array}$$

Only the direction "$\Longleftarrow$" of the third step is non-trivial and needs an explanation: If $X \in 2^A$ and $X \neq A$, then there exists $x \in A$ such that $X \subseteq A \ominus x$. Hence $\neg\mathcal{P}(A \ominus x)$ implies $\neg\mathcal{P}(X)$ since $\neg\mathcal{P}$ is downwards-closed due to (1). Guided by the above implication, we now define

$$\mathsf{inv}(A, B) \;:\Longleftrightarrow\; \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)$$

as loop invariant and choose $B = \emptyset$ as exit condition of the while-loop. In Pascal-like program notation, hence we have the following outline:

$$\begin{array}{l} \ldots \\ \{\ \mathsf{inv}(A, B)\ \} \\ \textbf{while}\ B \neq \emptyset\ \textbf{do}\ \cdots\ \textbf{od} \\ \{\ \mathsf{post}(A)\ \}\,. \end{array}$$

In the second stage of the program development we now have to consider the initialization of the variables A and B. We do this in combination with the choice of a suitable precondition, i.e., a requirement on the program's input U which is sufficiently general and implies additionally that the initialization establishes the loop invariant. Since we want to compute a subset of U that satisfies $\mathcal{P}$, it seems to be reasonable to demand that there exists such a subset. Due to assumption (1) this property is equivalent to $\mathcal{P}(U)$ being true. Hence we choose

$$\mathsf{pre}(U) \;:\Longleftrightarrow\; \mathcal{P}(U)$$

as precondition. This works and yields an initialization which assigns U to both variables A and B. Here is the formal justification:

$$\begin{array}{ll} & \mathsf{pre}(U) \\ \Longleftrightarrow & \mathcal{P}(U) \wedge \forall\, x \in \emptyset : \neg\mathcal{P}(U \ominus x) \\ \Longleftrightarrow & \mathcal{P}(U) \wedge U \subseteq U \wedge \forall\, x \in U \setminus U : \neg\mathcal{P}(U \ominus x) \\ \Longleftrightarrow & \mathsf{inv}(U, U)\,. \end{array}$$

Having achieved an initialization, in the last stage of the program derivation we must elaborate a loop body which maintains the invariant and ensures termination of the loop. To this end we suppose that the value of B is non-empty and b is an arbitrary element contained in it. We consider two cases. First we assume $\mathcal{P}(A \ominus b)$ to be true. Then

$$\begin{array}{ll} & \mathsf{inv}(A, B) \\ \Longleftrightarrow & \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \Longrightarrow & \mathcal{P}(A) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \Longrightarrow & \mathcal{P}(A \ominus b) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}((A \ominus b) \ominus x) \\ \Longrightarrow & \mathcal{P}(A \ominus b) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in (A \setminus B) \ominus b : \neg\mathcal{P}((A \ominus b) \ominus x) \\ \Longleftrightarrow & \mathcal{P}(A \ominus b) \wedge B \ominus b \subseteq A \ominus b \wedge \forall\, x \in (A \ominus b) \setminus (B \ominus b) : \neg\mathcal{P}((A \ominus b) \ominus x) \\ \Longleftrightarrow & \mathsf{inv}(A \ominus b, B \ominus b)\,. \end{array}$$

This calculation uses in the third step that $\mathcal{P}(A \ominus b)$ is true and the predicate $\neg\mathcal{P}$ is downwards-closed. The remaining steps apply only basic set theory and

logic and the definition of the invariant. Now we deal with the case of $\mathcal{P}(A \ominus b)$ being false. Here we obtain

$$\begin{array}{ll} & \mathsf{inv}(A,B) \\ \iff & \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \implies & \mathcal{P}(A) \wedge B \ominus b \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x) \\ \iff & \mathcal{P}(A) \wedge B \ominus b \subseteq A \wedge \forall\, x \in (A \setminus B) \oplus b : \neg\mathcal{P}(A \ominus x) \\ \iff & \mathcal{P}(A) \wedge B \ominus b \subseteq A \wedge \forall\, x \in A \setminus (B \ominus b) : \neg\mathcal{P}(A \ominus x) \\ \iff & \mathsf{inv}(A, B \ominus b)\,, \end{array}$$

where the assumption on $\mathcal{P}(A \ominus b)$ is used again in the third step and basic set theory and the definition of the invariant are applied otherwise.

In view of the just derived implications the invariant is maintained if we change in each run through the while-loop the value of B into $B \ominus b$ and, provided $\mathcal{P}(A \ominus b)$ holds, the value of A into $A \ominus b$. This can be easily achieved by a conditional and leads to the following greedy-like completion of the above program outline, where the statement $b :\in B$, known from the refinement calculus (cf. [15]), non-deterministically assigns some member of the value of B to b:

$$\begin{array}{l} \{\ \mathsf{pre}(U)\ \} \\ A, B := U, U; \\ \{\ \mathsf{inv}(A,B)\ \} \\ \textbf{while } B \neq \emptyset \textbf{ do} \\ \quad b :\in B; \\ \quad \textbf{if } \mathcal{P}(A \ominus b) \textbf{ then } A, B := A \ominus b, B \ominus b \\ \qquad\qquad\qquad\quad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\ \{\ \mathsf{post}(A)\ \}\,. \end{array} \qquad (\mathrm{MIN}_1)$$

Termination of this program is obvious since its input U is finite and, therefore, the initial value of B is finite, too. The size of the value of B, however, is strictly decreased by every run through the while-loop.

2.2 Refinements

The running time of an instantiation of the generic program (MIN_1) mainly depends on two facts, viz. the costs for the evaluation of the predicate $\mathcal{P}$ and the number of runs through the while-loop. Guided by these simple observations we now refine the program in such a way that in many cases the efficiency is – partly considerably – improved.

First we consider the evaluation of $\mathcal{P}$. From analyzing many instances of (MIN_1), e.g., those mentioned in the introduction, the additional assumption on $\mathcal{P}$ to be *decremental* arose. This property means that there exists a predicate $\mathcal{Q}$ on $2^U \times U$ such that for all non-empty sets $X \in 2^U$ and elements $x \in X$

$$\mathcal{P}(X \ominus x) \iff \mathcal{P}(X) \wedge \mathcal{Q}(X, x)\,. \qquad (2)$$

Since $\mathcal{P}(A)$ is part of the loop invariant $\mathsf{inv}(A,B)$, from (2) we get that the condition $\mathcal{P}(A \ominus b)$ of the conditional statement of (MIN_1) can be replaced by

$\mathcal{Q}(A, b)$. Doing so, we obtain the following refinement:

$$
\begin{array}{l}
\{\ \mathsf{pre}(U)\ \} \\
A, B := U, U; \\
\{\ \mathsf{inv}(A, B)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathcal{Q}(A, b) \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ \mathsf{post}(A)\ \}\,.
\end{array}
\qquad (\mathrm{MIN}_2)
$$

This program is more efficient than (MIN_1) if the evaluation of $\mathcal{Q}(X, x)$ is less expensive than that of $\mathcal{P}(X \ominus x)$, which proved to be true in all instances we have investigated.

With regard to the second observation it is obvious that the number of runs through the while-loop of (MIN_2) equals the size of the input set U. Heading towards increasing efficiency, it therefore seems to be a good idea to refine additionally the program's initialization by an efficient precomputation phase yielding a subset of U for which $\mathcal{P}$ holds and whose size is much smaller than that of U. This prompts us to introduce a new variable S of type 2^U to hold the value of the precomputation. Obviously $\mathcal{P}(S)$ and $\mathsf{inv}(S, S)$ are equivalent. Hence we can weaken the previous precondition to *true* and demand $\mathcal{P}(S)$ as postcondition of the precomputation. Altogether, we obtain the following correct refinement of the generic program (MIN_2) in which the pseudo code $\gg$ precomputation $\ll$ denotes the precomputation phase:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
\gg \text{precomputation} \ll; \\
\{\ \mathcal{P}(S)\ \} \\
A, B := S, S; \\
\{\ \mathsf{inv}(A, B)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathcal{Q}(A, b) \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ \mathsf{post}(A)\ \}\,.
\end{array}
\qquad (\mathrm{MIN}_3)
$$

If we compare this program to the first solution[1] of [17], then, besides the fact that we compute minimal subsets and [17] maximal ones, there are two main differences. First in [17] no precomputation is used which, however, is very profitable in many applications. Second in [17] an execution of the body of the while-loop enlarges the present solution but makes a second set – the candidates for an enlargement – smaller. This leads to an equality test on sets as condition

[1] In [17] data refinement is used to develop from it a further generic program, called second solution. This refinement assumes very specific properties of the used predicate and is tailored to the two applications the article deals with.

of the program's while-loop which is more costly than our simple emptiness test (or "being the universe" test in the dual program for computing maximal subsets satisfying a downwards-closed predicate).

3 Applications

In the following we instantiate the generic programs (MIN_2) and (MIN_3) to solve two graph-theoretic problems and sketch some further applications. We assume the reader to be familiar with the basic notations of graph theory; otherwise see, for instance, [3].

3.1 Vertex Cover

Let an undirected graph $g = (V, E)$ with finite sets V of vertices and E of edges be given, where each edge is a set $\{x, y\}$ of distinct vertices x and y. We suppose that in our programming language the two operations

$$\mathit{neigh}(x) = \{y \in V : \{x, y\} \in E\} \qquad \mathit{inc}(e) = \{f \in E : e \cap f \neq \emptyset\}$$

are implemented. They compute for $x \in V$ the set $\mathit{neigh}(x)$ of all neighbours resp. for $e \in E$ the set $\mathit{inc}(e)$ of all edges incident on it.

A subset C of V is called a *vertex cover* of the graph g if $C \cap e \neq \emptyset$ for all $e \in E$, i.e., if each edge of g is incident on at least one vertex of C. It is obvious that the predicate $\mathcal{P}$ on the powerset 2^V, defined by

$$\mathcal{P}(X) \;:\Longleftrightarrow\; \forall\, e \in E : X \cap e \neq \emptyset\,, \tag{3}$$

is upwards-closed and $\mathcal{P}(V)$ holds. Furthermore, from (3) we immediately obtain for all non-empty sets $X \in 2^V$ of vertices and vertices $x \in X$

$$\mathcal{P}(X \ominus x) \iff \mathcal{P}(X) \wedge \mathit{neigh}(x) \subseteq X\,. \tag{4}$$

Now we choose the inclusion $\mathit{neigh}(x) \subseteq X$ of (4) as predicate $\mathcal{Q}(X, x)$. Then $\mathcal{P}$ is decremental in the sense of (2) and we arrive at the following instantiation of (MIN_2) for computing a minimal vertex cover of g:

$$\begin{array}{l}
\{\ \mathit{true}\ \} \\
A, B := V, V; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathit{neigh}(b) \subseteq A \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\qquad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ minimal vertex cover of } g\ \}\,.
\end{array} \tag{VC_1}$$

During the execution of the program (VC_1) the inclusion $\mathit{neigh}(b) \subseteq A$ is tested exactly once for each vertex of g. Hence the total running time is $O(|E|)$ if g

is represented by an adjacency list (see e.g., [3]) and a set is represented by a Boolean array and its size.

If we want to compute a minimum vertex cover instead of a minimal one, then we are concerned with an NP-complete problem. See, for instance, [3] for a proof of this fact. In this textbook also a simple $O(|E|)$ approximation algorithm for the minimum vertex cover problem is presented and attributed to Gavril and Yannakakis. If this algorithm is expressed as an annotated while-program, we obtain the following code:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
F, S := E, \emptyset; \\
\{\ S \text{ vertex cover of } g_F = (V, E \setminus F) \wedge \mathsf{inv}'(F, S)\ \} \\
\textbf{while } F \neq \emptyset \textbf{ do} \\
\quad e :\in F; \\
\quad F, S := F \setminus \mathit{inc}(e), S \cup e\ \ \textbf{od} \\
\{\ S \text{ vertex cover of } g \wedge |S| \leq 2c_{opt}\ \}\,.
\end{array}
\qquad (\mathrm{VC}_2)
$$

Here c_{opt} denotes the *size of a minimum vertex cover* of g. The second conjunct $\mathsf{inv}'(F, S)$ of the loop invariant of (VC_2) is defined as

$$
\mathsf{inv}'(F, S) \ :\Longleftrightarrow\ \begin{cases} \exists\, M \in 2^E : \ M \text{ matching of } g\ \ \wedge \\ \qquad\qquad\quad |S| \leq 2|M|\ \ \wedge \\ \qquad\qquad\quad \forall\, f \in F, m \in M : f \cap m = \emptyset \end{cases}
$$

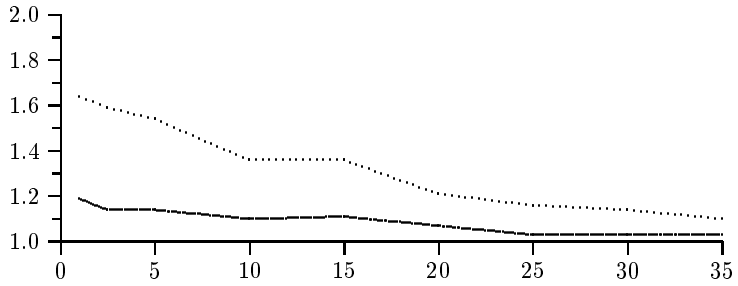
and formalizes the idea that the set of all edges that were picked by the statement $e :\in F$ during the execution of program (VC_2) forms a matching M of g such that $|S| \leq 2|M|$ and edges of F and M have no vertex in common. It is not hard to see that the loop invariant of (VC_2) is established by the initialization and maintained by the body of the while-loop. (In doing so, the third part of the body of $\mathsf{inv}'(F, S)$ is necessary to prove that for a matching M of g and an edge $e \in F$ also $M \oplus e$ is a matching of g.) Hence if the program terminates, then the first part of its postcondition trivially holds. The remaining part $|S| \leq 2c_{opt}$ follows from the fact that each minimum vertex cover C^* must include at least one vertex of any edge of any matching. This implies $|M| \leq |C^*| = c_{opt}$ from which we get the desired result $|S| \leq 2c_{opt}$ due to $|S| \leq 2|M|$.

If we compare the two programs (VC_1) and (VC_2) wrt. the usual worst case approximation bound (see, for example, [8] for details), then program (VC_2) is superior. For any input graph $g = (V, E)$ it computes a vertex cover whose size is guaranteed to be no greater than twice the minimum size c_{opt} of a vertex cover of g. In contrast with this, for a "star-like" input graph g the result of program (VC_1) may contain $(|V|-1)c_{opt}$ vertices if the non-deterministic statement $b :\in B$ unfortunately never selects the star's centre.

But if we compare the quality of the results produced for randomly generated inputs, which is a well-established mode to assess algorithms in practice, then the situation changes and program (VC_1) becomes superior. We have formulated both programs in the programming language of the relation-algebraic prototyping tool RelView (see e.g., [1, 2]) and performed numerous experiments. In

each experiment we fixed a number n of vertices, generated random graphs with n vertices (more exactly: their symmetric and irreflexive adjacency relations), where we varied the number of edges from 1% to 35%, and compared the results of (VC_1) and (VC_2) with the sizes of the minimum vertex covers. Due to the very efficient ROBDD-implementation of relations (see [12]), RelView allows treating arbitrary graphs with up to 150 vertices. For dense graphs even larger vertex sets are possible. E.g., on a Sun Fire-280R workstation with 750 MHz and 4 GByte main memory running Solaris 8 the enumeration of all minimum vertex covers for randomly generated graphs with 250 vertices takes approx. 3000 sec. (200 sec. resp. 50 sec.) in the case of 65% (75% resp. 85%) of all edges.

The following figure shows one result for $n = 100$. On its x-axis the graphs' density is listed and the ratio of the results' size and c_{opt} is listed on the y-axis. The dotted curve belongs to (VC_2) and the other one to (VC_1). Both curves have been obtained by considering 180 graphs and computing the ratios' arithmetic mean values.



All other experiments we have performed showed the same tendency. Based on these results, hence it seems to be reasonable to combine both programs, i.e., to instantiate the generic program (MIN_3) instead of (MIN_2) and to choose Gavril and Yannakakis' algorithm (VC_2) as precomputation phase. Doing so, we arrive at the following result:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
F, S := E, \emptyset; \\
\{\ S \text{ vertex cover of } g_F = (V, E \setminus F) \wedge \mathsf{inv}'(F, S)\ \} \\
\textbf{while } F \neq \emptyset \textbf{ do} \\
\quad e :\in F; \\
\quad F, S := F \setminus \mathit{inc}(e), S \cup e \ \ \textbf{od}; \\
\{\ S \text{ vertex cover of } g \wedge |S| \leq 2c_{opt}\ \} \\
A, B := S, S; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathit{neigh}(b) \subseteq A \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ minimal vertex cover of } g \wedge |S| \leq 2c_{opt}\ \}\,.
\end{array}
\qquad (\mathrm{VC}_3)
$$

This program has also $O(|E|)$ as running time (practical experiments have shown that it is only a little bit slower than each of the above single programs), the theoretical worst case approximation bound 2 of (VC_2), but also the good practical behaviour of (VC_1).

3.2 Transitive Reduction

For computing a transitive reduction we assume a *strongly connected*[2] directed graph $g = (V, R)$ with a finite set V of vertices and a relation R on V containing the graph's edges. (Now, each edge is a directed pair of vertices.) Strongly connectedness of g can be described by the equation $R^* = \mathsf{L}$, where the star denotes the reflexive-transitive-closure operator and L denotes the universal relation $V \times V$. In this section we shall also use the transposition operator on relations which we denote as $^{\mathsf{T}}$.

As already mentioned in the introduction, a transitive reduction T of g is a minimal subrelation of R such that $T^* = R^*$. Due to the strong connectedness of g and the monotonicity of the star-operator wrt. relation inclusion, the predicate $\mathcal{P}$ on the powerset 2^R, defined by

$$\mathcal{P}(X) \;:\Longleftrightarrow\; X^* = \mathsf{L}\,, \tag{5}$$

is upwards-closed and $\mathcal{P}(R)$ is valid. Furthermore, a little reflection shows that for all non-empty relations $X \in 2^R$ and edges $x \in X$

$$\mathcal{P}(X \ominus x) \iff \mathcal{P}(X) \wedge x \in (X \ominus x)^*\,. \tag{6}$$

Hence if we choose the part $x \in (X \ominus x)^*$ of (6) as predicate $\mathcal{Q}(X, x)$, then the predicate $\mathcal{P}$ of (5) is decremental in the sense of (2) and we can instantiate the generic program (MIN_2) accordingly. This yields:

$$\begin{array}{l}
\{\ \mathit{true}\ \} \\
A, B := R, R; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg\mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } b \in (A \ominus b)^* \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } \ B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ transitive reduction of } g\ \}\,.
\end{array} \tag{TR_1}$$

In this program the condition $b \in (A \ominus b)^*$ describes a simple reachability problem: Is the sink of b reachable from its source via edges from $A \ominus b$? This can be tested in time $O(|V| + |A|)$ using an adjacence list representation of A. Hence the total running time of (TR_1) is $O(|R|^2)$.

[2] The general case easily can be reduced to the case of strongly connected graphs. See, for instance, [16].

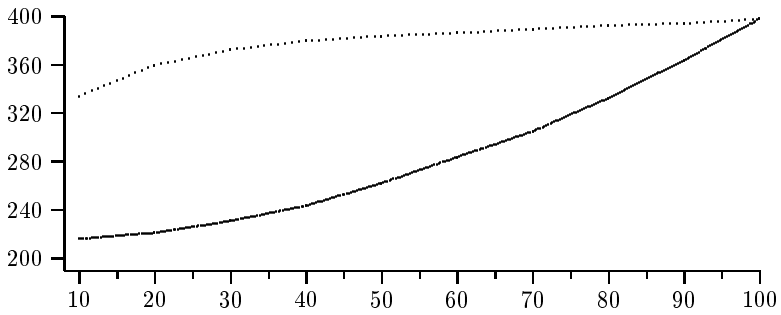
To get a more efficient program for the transitive reduction of g, we now instantiate the generic program (MIN_3) and compute the result S of the precomputation as follows: Starting from a common root, first we compute a spanning arborescence $t_1 = (V, T_1)$ of g and a spanning arborescence $t_2 = (V, T_2)$ of the transposed graph $g^\mathsf{T} = (V, R^\mathsf{T})$. Then we define S as $T_1 \cup T_2^\mathsf{T}$ and obtain by that a strongly connected subgraph $t = (V, S)$ of g such that $|S| \leq 2|V| - 2$.

Depth-first search (abbreviated as DFS) is a first possibility to compute spanning arborescences of directed graphs in edge-linear time. If we suppose a corresponding function *Dfs*, then we arrive at the following refinement of program (TR_1) the running time of which now is $O(|V|^2)$:

$$
\begin{array}{l}
\{\ \mathit{true}\ \} \\
r :\in V; \\
S := \mathit{Dfs}(R, r) \cup \mathit{Dfs}(R^\mathsf{T}, r)^\mathsf{T}; \\
A, B := S, S; \\
\{\ \mathcal{P}(A) \wedge B \subseteq A \wedge \forall\, x \in A \setminus B : \neg \mathcal{P}(A \ominus x)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } b \in (A \ominus b)^* \textbf{ then } A, B := A \ominus b, B \ominus b \\
\qquad\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ A \text{ transitive reduction of } g\ \}.
\end{array}
\qquad (\mathrm{TR}_2)
$$

Of course, one is particularly interested in transitive reductions with few edges, i.e., in approximations of minimum equivalent digraphs. Concentrating on this aspect and using implementations in the RelView system and the C programming language, we have experimented with the program (TR_2) and with a variant of it which applies breadth-first search (shortly: BFS) instead of DFS for computing the spanning arborescences of the precomputation. The program using DFS proved to be superior. But all our experiments also showed the same negative tendency, viz. the denser the input graph gets the larger gets the number of edges of the result. This is in contrast to the expectation that the number of edges of a good approximation of a minimum equivalent digraph should decrease if the density of the underlying graph increases.

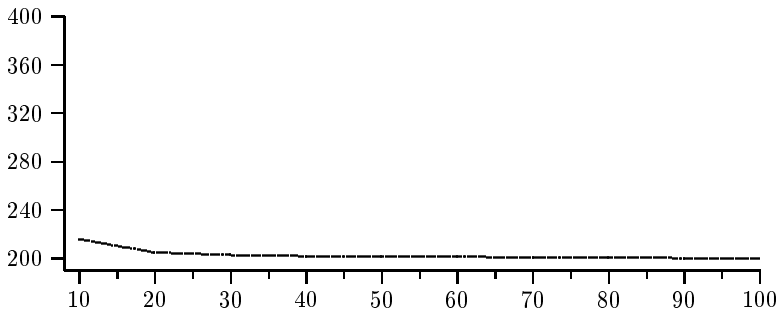
As an example, in the following figure the result of one experiment with a fixed number of 200 vertices and 200 randomly generated input graphs is depicted. (Of course, also much larger graphs have been condidered; see [10].)



As in the case of vertex covers the graphs' density is listed on the x-axis. On the y-axis the arithmetic mean-values of the sizes of the computed transitive reductions are shown, where the lower curve belongs to the DFS-precomputation and the upper one to the BFS-precomputation.

The reason for the unsatisfactory behaviour of the previous programs for computing small transitive reductions gets clear if one depicts their computed results on the graph window of RelView. They consist of a lot of short cycles and their shortness even increases if the inputs get denser. But obviously, a small transitive reduction should be composed of a few long cycles only. To obtain this goal, we have used a variant of (TR_2) in which the first arborescence $t_1 = (V, T_1)$ is computed using DFS but the second one $t_2 = (V, T_2)$ using BFS. This specific precomputation was motivated by the following idea: If DFS is applied to get t_1, then this arborescence contains long paths from its root r to every other vertex of g. Due to the use of BFS, however, the transpose of the arborescence t_2 consists of shortest paths from every vertex of g except r back to r. Hence the union of t_1 and t_2^{T} leads to a starting point of the minimalization process which is composed of long cycles only and obviously this property descends upon the process' result.

Again we have tested this approach and arrived, for example, in the case of 200 randomly generated input graphs, each with the same number of 200 vertices, at the following diagram:



The curves of all other experiments we have performed look rather similar. See [10] for details.

In the course of the diploma thesis [10] the DFS/BFS-approach to transitive reductions has been compared with the best approximation algorithm for minimum equivalent digraphs of [11]. The latter algorithm computes an approximation in nearly linear time with approximation bound between $\frac{\pi^2}{6} \approx 1.64$ and $\frac{\pi^2+1}{6} \approx 1.81$. Although the approximation bound of our program is 2, in all practical experiments it proved to be superior: If n is the number of vertices of the input graph, then the algorithm of [11] leads to results which consist in the average of $\frac{3n}{2}$ edges (i.e., $\approx$300 edges if $n = 200$ instead of the $\approx$210 edges our algorithm yields) and, furthermore, are no transitive reductions. The latter means that they contain superfluous edges. Hence it seems to be reasonable to combine it with our approach. As in the case of vertex covers, this leads to an

algorithm with the better theoretical approximation bound and good practical results.

3.3 Some Further Applications

Having presented two graph-theoretic applications in great detail, now we want to sketch some further applications of our generic minimalization programs and their maximalization variants.

First, we have applied the programs to all problems mentioned in the introduction. Furthermore, we have solved many other problems with their means including the computation of kernels of undirected graphs (which are maximal independent sets of vertices) and of minimal transversals of hypergraphs. We have used the programs also for some filter problems on sets. The latter applications are based on the fact that $\{x \in U : Q(x)\}$ is the unique maximal subset of U that satisfies the downwards-closed predicate $\mathcal{P}$ defined by

$$\mathcal{P}(X) \;:\Longleftrightarrow\; \forall\, x \in X : Q(x) \tag{7}$$

and, furthermore, $\mathcal{P}(\emptyset)$ holds and $\mathcal{P}$ is *incremental* in the sense that for all sets $X \in 2^U$ with $X \neq U$ and all elements $x \in U \setminus X$

$$\mathcal{P}(X \oplus x) \iff \mathcal{P}(X) \wedge \mathcal{Q}(x)\,. \tag{8}$$

As a consequence of (7) and (8), we can instantiate the maximalization variant of (MIN_2) which immediately leads to the following generic program:

$$\begin{array}{l}
\{\ \mathit{true}\ \} \\
A, B := \emptyset, U; \\
\{\ \mathsf{inv}(A, B)\ \} \\
\textbf{while } B \neq \emptyset \textbf{ do} \\
\quad b :\in B; \\
\quad \textbf{if } \mathcal{Q}(b) \textbf{ then } A, B := A \oplus b, B \ominus b \\
\qquad\qquad\quad \textbf{else } B := B \ominus b \textbf{ fi od} \\
\{\ A = \{x \in U : Q(x)\ \}\,.
\end{array} \tag{FIL}$$

The invariant $\mathsf{inv}(A, B)$ of the program (FIL) is given by

$$\mathsf{inv}(A, B) \;:\Longleftrightarrow\; \mathcal{P}(A) \wedge A \subseteq U \setminus B \wedge \forall\, x \in (U \setminus B) \setminus A : \neg\mathcal{P}(A \oplus x)\,.$$

A concrete instantiation of our approach is also part of RelView. This system offers some commands for the nice drawing of graphs, one of them also for orthogonal grid drawing. The underlying algorithm follows the general approach: In a first step it computes a planar subgraph with as much as possible edges. Then the subgraph is drawn on a grid (using Tamassia and Tollis' method [18] in our case). Finally, the remaining edges are inserted into this drawing to obtain an orthogonal drawing of the original graph. Finding a maximum planar subgraph is an NP-complete problem. Hence existing algorithms use heuristics. We have

decided to approximate maximum planar subgraphs by maximal ones and used the maximalization variant of our approach (to be planar is a downwards-closed property and empty graphs are pranar) in combination with the well-known Hopcroft-Tarjan planarity test (see [9]) for this task. The experiences we gained so far show that this approach is fast and achieves good drawings in practice.

4 Concluding Remarks

In this article, we have first formally developed a generic program for computing a minimal subset satisfying an upwards-closed predicate $\mathcal{P}$ from a logical specification by combining the invariant technique with set-theoretic and logical calculations. To improve the efficiency of concrete instantiations, we investigated then two refinements. The first refinement assumes $\mathcal{P}$ to be decremental and the second one applies in addition a precomputation phase which yields a good starting point for the minimalization process. Next, we have instantiated the refinements to solve two graph-theoretic problems, viz. the approximation of a minimum vertex cover and the computation of a transitive reduction as approximation of a minimum equivalent digraph. Doing so, we also have presented some experimental results and compared resp. combined our programs with two algorithms known from the literature. This leads to improvements of the practical behaviour of the latter ones. Finally, we have sketched some further applications of the programs and their variants which compute a maximal subset satisfying a downwards-closed predicate.

If we put our work in the context of other work on deriving a family of algorithms, then we see that it follows one of the key idea of generic programming, viz. the lifting of concrete algorithms to as general a level as possible without loosing efficiency. Of course, this is not the only development method. Another very prominent approach uses transformational programming and develops it step-wise from a common specification.

Presently we investigate further instantiations of our generic programs and modifications of them. In particular we seek for ways to formalize the precomputation phase of (MIN_3) in specific cases and to integrate objective functions. Using formal methods we hope that in the latter case the role of the different axioms of the set structures underlying the usual approaches (like independence structures, matroids and greedoids; see [7] for details) become more clear. We also investigate at present how to use signatures, structures, and functors in the functional language Standard ML to implement our generic programs and concrete instantiations in an easy and flexible way and we plan to do the same for an object-oriented language in the future, too. Finally we are interested in the general application of formal specification and development methods in the context of generic programs. We believe that their manifold reusability absolutely requires a well-defined functionality and a rigor mathematical verification that guarantees the specified behaviour. All this is done in combination with so-called algorithm engineering techniques (see e.g., [13, 14]) to get, besides the

advantages of formal methods, also a feeling for the programs' actual efficiency and quality and to close the gap between theory and practice.

References

1. R. Behnke et al. Applications of the RelView system. In: R. Berghammer, Y. Lakhnech (eds.): Tool support for system specification, development and verification. Springer, pages 33-47, 1999.
2. R. Berghammer, B. von Karger, and C. Ulke. Relation-algebraic analysis of Petri nets with RelView. In: T. Margaria, B. Steffen (eds.): Proc. 2nd Workshop on Tools and Applications for the Construction and Analysis of Systems, LNCS 1055, Springer, pages 49-69, 1996.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* The MIT Press, 1990.
4. E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.
5. J. Edmonds. Paths, trees, and flowers. Canadian Journal of Mathematics 17:449-467, 1965.
6. D. Gries. *The science of computer programming.* Springer, 1981.
7. P. Helman, B.M.E. Moret, H.D. Shapiro. An exact characterization of greedy structures. SIAM Journal Disc. Math. 6:274-283, 1993.
8. D. Hochbaum (ed.). *Approximation algorithms for NP-hard problems.* PWS Publishing Company, 1995.
9. J. Hopcroft and R. Tarjan. Efficient planarity testing. Journal of the ACM 21:549-568, 1974.
10. C. Kasper. *Investigating algorithms for transitive reductions and minimum equivalent digraphs (in German).* Diploma thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2001.
11. S. Khuller, B. Raghavachari, and N. Young. Approximating the minimum equivalent digraph. SIAM Journal of Computing 24:859-972, 1995.
12. B. Leoniuk. *ROBDD-based implementation of relations and relational operations with applications (in German).* Ph.D. thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2001.
13. B.M.E. Moret and H.D. Shapiro. Algorithms and experiments: The new (and old) methodology. Journal of Universal Computer Science 7:434-446, 2001.
14. B.M.E. Moret. Towards a discipline of experimental algorithmics. DIMACS Series in Discrete Mathematics and Theoretical Computer Science (to appear).
15. C. Morgan and T. Vickers (eds.). *On the refinement calculus.* Formal Approaches to Computing and Information Technology, Springer, 1992.
16. H. Noltemeier. Reduktion von Präzedenzstrukturen. Zeitschrift für Operations Research 20:151-159, 1976.
17. J. Ravelo. Two graph algorithms derived. Acta Informatica 36:489-510, 1999.
18. R. Tamassia and T.G. Tollis. Planar grid embedding in linear time. IEEE Transactions on Circuits and Systems 36:1230-1234, 1989.