Calculating with Relations for Graph Algorithmics

Jesús N. Ravelo*

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Apartado Postal 89000, Caracas 1089, Venezuela

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

Abstract

Much emphasis has been placed in recent years on deriving or calculating programs rather than proving them correct. Adequate calculational frameworks are needed to support such an approach. The present work explores the use of a calculus of relations to express and reason about graph properties in an algorithmic context. We take a generic program that computes a maximal set, over some universe, satisfying some predicate P and calculate two instances of it: the computation of maximal independent sets of vertices in a graph, and the computation of maximal sets of edges without cycles (i.e. maximal forests).

1 Introduction

Interest in calculational techniques for the construction of software has only but increased in the last few years. Appropriate calculi that combine conciseness and precision permit clearer presentations of programming strategies and heuristics. Relations have found their way to programming theory both in the functional and imperative paradigms (see e.g. [3, 8]), because they provide an adequate framework to treat non-determinism. However, in the present work we calculate with relations for a different purpose.

*Research supported by Programa de Nuevas Tecnologías of CONICIT (Consejo Nacional de Investigación Científica y Tecnológica) of Venezuela, Interamerican Development Bank and Universidad Simón Bolívar, and by the Overseas Research Students Awards Scheme of the United Kingdom

Graphs are an important concept in algorithmics (see e.g. [4, 10]), if only for the variety and importance of their applications [6]. A calculational treatment of graphs, in the same spirit of [15], benefits the program construction field. This work demonstrates how to obtain, from a generic program, instances that solve graph algorithmic problems by calculation. A calculus of relations is used to specify and manipulate the relevant properties of graphs.

Section 2 introduces the graph concepts we will need, using relations as algebraic framework. The calculus of relations is only briefly introduced as it is needed; thorough introductions can be found elsewhere [1, 3, 15]. Sections 3 and 4 present, respectively, the generic program and the derivation of two graph instances of it. Section 5 provides some concluding remarks.

2 Graphs and Relations

An undirected graph is defined to be a 4-tuple $(\mathit{Vert}, \mathit{Edge}, \mathit{x1}, \mathit{x2})$ where Vert and Edge are sets, and $\mathit{x1}$ and $\mathit{x2}$ are functions of type $\mathit{Vert} \leftarrow \mathit{Edge}$ that determine the extreme vertices of an edge. We prefer to use the right-to-left type arrow both for functions and relations. First, because it is consistent with the conventional notation for functional application, in which arguments are given to functions on the right: for function $f : X \leftarrow Y$ and y of type Y, $f\,y$ is of type X. Also, because the types involved in relational composition (and therefore in functional composition) take a more natural form: for relations $R : X \leftarrow Y$ and $S : Y \leftarrow Z$, their composition is $R{\cdot}S : X \leftarrow Z$.

In an undirected graph two vertices are said to be adjacent if there exists an edge between them. That is, vertices v and w are adjacent if there is an edge e such that $v = \mathit{x1}\;e$ and $w = \mathit{x2}\;e$ or vice versa, i.e. $v = \mathit{x2}\;e$ and $w = \mathit{x1}\;e$. In a point-free style we can define the adjacency relation $\mathit{adj} : \mathit{Vert} \leftarrow \mathit{Vert}$ as $\mathit{adj} := \mathit{x1}{\cdot}\mathit{x2}^{\circ} \cup \mathit{x2}{\cdot}\mathit{x1}^{\circ}$, where relation $R^{\circ} : Y \leftarrow X$ is the converse of $R : X \leftarrow Y$. Two vertices are joinable if there exists a path of zero or more edges that traverses the graph from one to the other. The joinability relation $\mathit{join} : \mathit{Vert} \leftarrow \mathit{Vert}$ can thus be defined as the reflexive-transitive closure of the adjacency relation $\mathit{join} := \mathit{adj}^{*}$. Note that, since adj is a symmetric relation, join is also symmetric and thus an equivalence relation.

Elements and subsets of a set can be modelled by relations using the unit type $\mathsf{1}$, which is a singleton set. An element of set X is a function $x : X \leftarrow \mathsf{1}$ and a subset of X is a relation $A : X \leftarrow \mathsf{1}$; the set modelled by A is $\{\, a \mid a\, A \ast \,\}$ where $\ast$ is the element of $\mathsf{1}$. We will denote the above as $x : X$ and $A : \mathsf{Set}\, X$, and conventionally use lower-case for elements and upper-case for sets. Note the natural embedding of elements into sets, corresponding to the formation of singletons. For each type $X \leftarrow Y$ there is a largest relation Π called the universal relation of that type. In the case of $\Pi : X \leftarrow \mathsf{1}$, the universal relation models the whole set X; accordingly, we will also use X to denote such relation.

A different way of modelling subsets uses so-called coreflexive relations. For each X there is an identity function $id : X \leftarrow X$; a relation C of type $X \leftarrow X$ is coreflexive if $C \subseteq id$. Coreflexive C can be seen as representing subset $\{ c \mid c\ C\, c \}$ of X. Both representations of subsets are useful. For example, given relations $R : X \leftarrow Y$ and $S : Y \leftarrow Z$, to restrict the "middle points" in composition $R \cdot S$ to a certain subset of Y, the corresponding coreflexive C can be used thus: $R \cdot C \cdot S$. On the other hand, given $A : \mathsf{Set}\, Y$, the existential image of A under relation R can easily be expressed as $R \cdot A : \mathsf{Set}\, X$. There is an isomorphism between the two representations witnessed by the *ran*ge operator, defined for any relation R as the coreflexive $ran\, R = id \cap R \cdot R^\circ$. We will reserve the use of the term subset (or set) for their representation as relations from the unit type.

Independence

Two vertices of a graph are said to be independent if they are not adjacent. A set of vertices is independent if any pair of vertices drawn from it are independent. Formally, for graph $(\mathit{Vert}, \mathit{Edge}, \mathit{x1}, \mathit{x2})$ with adjacency relation adj, and $V : \mathsf{Set}\, \mathit{Vert}$, we define:

$$\mathit{Indep}\, V \quad := \quad V \cdot V^\circ \cap adj \ \subseteq\ \emptyset \tag{2.1}$$

Strictly speaking, this definition should be parameterised by the graph as well. We chose to omit it, here and in further definitions, since it will always be clearly determined by context.

The empty set is independent. A singleton set v is independent if no edge has both extremes equal to the vertex v. Thus, small independent sets are easy to find. A more interesting problem is that of finding largest possible (i.e. maximal) independent sets.

Forests

Given a graph, a spanning subgraph of it is a graph with the same set of vertices but only a subset of its edges. A cycle is a non-empty path that starts and ends in the same vertex without traversing any edges more than once, and a forest is a spanning subgraph with no cycles. We will now formalise these concepts using relations.

Each subset E of edges of a graph $(\mathit{Vert}, \mathit{Edge}, \mathit{x1}, \mathit{x2})$ uniquely determines a spanning subgraph of it, and vice versa. We will therefore identify such subgraphs with their sets of edges. Given $E : \mathsf{Set}\, \mathit{Edge}$, the adjacency and joinability relations of the corresponding spanning subgraph are

$adj\,E\ :=\ x1 \cdot ran\,E \cdot x2^{\circ} \cup x2 \cdot ran\,E \cdot x1^{\circ}$ and $join\,E := (adj\ E)^{*}$. Note again that $join\,E$ is an equivalence relation.

For a given partition P of the set of vertices of a graph, there is a so-called contracted graph induced by P. Its vertices are the classes of P (i.e. the elements of P), and its edges are the same ones of the original graph but now incident on the classes to which their extremes belong. We then say that a subset E of edges determines a contracted graph: the one induced by the equivalence classes of $join\,E$.

To formalise acyclicity we will use the concept of loops. An edge e is a loop if its endpoints coincide, i.e. if $adj\ e \subseteq id$. And it is a loop in the contracted graph determined by set E if its endpoints belong to the same equivalence class, i.e. $adj\ e \subseteq join\,E$. Note that if two vertices are related by $join\,E$, there must be a path between them comprising edges drawn solely from E. Hence, if edge e is not in E and is a loop in the corresponding contracted graph, we can complete a cycle starting from one of the extremes of e, travelling to the other through E, and finally returning to the starting point using e. Conversely, if E contains a cycle, we can draw an edge e from it (for cycles are non-empty) such that e is a loop in the contracted graph determined by $E-e$. We can now formalise the absence of cycles in a spanning subgraph –that is, the notion of forest– thus:

$$Cycle\,E\,e \ := \ adj\ e \subseteq join\,E \tag{2.2}$$
$$Forest\,E \ := \ (\forall e : e \subseteq E : \neg\, Cycle\,(E-e)\,e) \tag{2.3}$$

As it was the case for independent sets of vertices, it is not very challenging to find small forests. The empty subgraph (i.e. the empty set) is a forest. A singleton e is a forest if edge e is not a loop. Finding a maximal forest is another story.

3 A Generic Program

In this section we present a program that solves the generic problem of computing a maximal set, over some universe X, satisfying a certain predicate P. For the description of both the specification of the problem and the program we use Dijkstra's language of guarded commands [5] extended with Morgan's specification statement [12]. Our problem is specified as follows:

$$\begin{array}{ll} |[& \mathbf{var}\ A : \mathsf{Set}\,X; \\ & A : [\ (\forall B : A \subseteq B : P\,B \ \equiv\ B = A)\ ] \\ ]| & \end{array} \tag{3.1}$$

The block brackets |[and]| enclose a program fragment with local declarations. In (3.1) only variable A is declared, and there is a specification statement affecting A. The statement $w : [\, post\,]$ reads "Change only the variables listed in w so that the resulting final state satisfies the postcondition *post*". The provided postcondition tells us, with the *follows-from* part of the equivalence, that A satisfies P and, with the *implies*, that no other superset of A except itself satisfies P. That is, A is P-maximal.

The instance $X, P := \mathit{Vert}, \mathit{Indep}$ of (3.1) provides the problem of finding a maximal independent set of vertices, while $X, P := \mathit{Edge}, \mathit{Forest}$ corresponds to finding a maximal forest.

Some properties of generic predicate P are needed to refine (3.1) to the program in figure 3.6. First, P is assumed to be subset-closed, which means:

$$P\,\emptyset \tag{3.2}$$

$$A \subseteq B \quad \Rightarrow \quad (P\,B \Rightarrow P\,A) \tag{3.3}$$

Also, it is assumed that P is incremental with respect to predicate Q, function f and operator $\oplus$ in the following sense:

$$P\,(A \cup a) \;\equiv\; P\,A \;\wedge\; Q\,(f\,A)\,a \qquad \text{provided} \;\; a \subseteq \overline{A} \tag{3.4}$$

$$f\,(A \cup a) \;=\; f\,A \,\oplus\, a \tag{3.5}$$

$\overline{R}$ is the complement $(\Pi - R)$ of relation R, and so the proviso of equation (3.4) reads "a is an element not in set A". Provided P is subset-closed, incrementality is no restriction since such $Q, f, \oplus$ always exist: take f to be the identity function, $\oplus$ to be $\cup$, and $Q\,A\,a$ to be $P\,(A \cup a)$. However, we are not interested in any $Q, f, \oplus$ since the efficiency of the program will depend on such a selection.

Using well-known techniques for developing imperative programs (see e.g. [9, 12]) and the above properties of P, specification (3.1) can be refined to the program shown in figure 3.6. Details of the refinement process are omitted.

4 Calculating Instances

We now instantiate the generic program given above to the graph problems of computing maximal independent sets of vertices and maximal forests. The properties imposed on P must be shown to hold for predicates *Indep* and *Forest*. Following the calculational spirit of algorithmics we aim to promote, components $Q, f, \oplus$ for each instance will be calculated instead of given a priori.

Maximal Independent Sets

Recall definition (2.1) of predicate *Indep*. First, it must be shown that *Indep* is subset-closed. The empty relation $\emptyset$ is a zero of both composition and

```
|[  var A : Set X;
    |[  var C : Set X;  y : Y;
        A, C, y := ∅, X, f ∅ ;
        do C ≠ ∅ →
              |[  var x : X;
                  x :⊆ C ;  C := C − x ;
                  if    Q y x  →  A, y := A ∪ x, y ⊕ x
                  [] ¬ Q y x  →  skip
                  fi
              ]|
        od
    ]|
]|
```

Figure 3.6: Generic Program

intersection, hence, (3.2) holds. Property (3.3) follows from monotonicity of converse, composition and intersection with respect to $\subseteq$. For the incrementality property we calculate thus:

$$\begin{array}{ll}
 & \mathit{Indep}\,(V \cup v) \\
\equiv & \quad \{\ \text{unfold definition of } \mathit{Indep}\ (2.1)\ \} \\
 & (V \cup v) \cdot (V \cup v)^{\circ} \,\cap\, \mathit{adj} \ \subseteq\ \emptyset \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of converse, composition} \\ \text{and intersection over union} \end{array} \right\} \\
 & (V \cdot V^{\circ} \cap \mathit{adj}) \ \cup\ (V \cdot v^{\circ} \cap \mathit{adj}) \\
 & \ \cup\ (v \cdot V^{\circ} \cap \mathit{adj}) \ \cup\ (v \cdot v^{\circ} \cap \mathit{adj}) \ \subseteq\ \emptyset \\
\equiv & \quad \{\ \text{union}\ \} \\
 & V \cdot V^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset \ \ \wedge\ \ V \cdot v^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset \\
 & \ \wedge\ \ v \cdot V^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset \ \ \wedge\ \ v \cdot v^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset \\
\equiv & \quad \left\{ \begin{array}{l} \text{fold definition of } \mathit{Indep}\ (2.1)\text{; second and third} \\ \text{conjuncts are equivalent by symmetry of } \mathit{adj} \end{array} \right\} \\
 & \mathit{Indep}\, V \ \ \wedge\ \ v \cdot V^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset \ \ \wedge\ \ v \cdot v^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset
\end{array}$$

The middle conjunct is further manipulated. Such manipulation applies to the conjunct on the right as well.

$$\begin{array}{ll}
 & v \cdot V^{\circ} \cap \mathit{adj} \ \subseteq\ \emptyset \\
\equiv & \quad \{\ \text{shunting}\ \} \\
 & v \cdot V^{\circ} \ \subseteq\ \overline{\mathit{adj}}
\end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ Schröder's rule } \} \\
& adj\cdot V \ \subseteq \ \overline{v} \\
\equiv & \{ \text{ complementation } \} \\
& v \ \subseteq \ \overline{adj\cdot V}
\end{array}
$$

To comply with (3.4) we then take:

$$
\begin{array}{rcll}
f\,V & := & \overline{adj\cdot V} & \qquad \text{(and therefore } f\,\emptyset \ = \ \mathit{Vert}) \\
Q\,W\,v & := & v \subseteq W \ \ \wedge \ \ v \not\subseteq adj\cdot v &
\end{array}
$$

Operator $\oplus$ can also be derived, given the above definition of f, to make (3.5) hold. By distributivity and De Morgan's rule we obtain:

$$
W \oplus v \ \ := \ \ W - adj\cdot v
$$

Note that the form of Q and $\oplus$ suggest the use of adjacency lists to represent the graph. The resulting program can be further improved but we will not go into such detail.

Maximal Forests

The same process is now carried out for predicate *Forest*. The fact that *Forest* is subset-closed follows from predicate calculus. The empty range rule implies (3.2), and antimonotonicity of the universal quantifier in its range implies (3.3). We now proceed, for the incrementality, as follows:

$$
\begin{array}{ll}
& \mathit{Forest}\ (E \cup e) \\
\equiv & \{ \text{ unfold definition of } \mathit{Forest} \text{ (2.3) } \} \\
& (\forall d \ : \ d \subseteq E \cup e \ : \ \neg\ \mathit{Cycle}\ ((E \cup e) - d)\ d\,) \\
\equiv & \{ \text{ split range, one-point rule } \} \\
& (\forall d \ : \ d \subseteq E \ : \ \neg\ \mathit{Cycle}\ ((E \cup e) - d)\ d\,) \\
& \ \ \wedge \ \ \neg\ \mathit{Cycle}\ ((E \cup e) - e)\ e \\
\equiv & \left\{ \text{ assumption } e \subseteq \overline{E} \text{ implies } (E \cup e) - e \ = \ E \right\} \\
& (\forall d \ : \ d \subseteq E \ : \ \neg\ \mathit{Cycle}\ ((E \cup e) - d)\ d\,) \ \ \wedge \ \ \neg\ \mathit{Cycle}\,E\,e
\end{array}
$$

But we cannot fold *Forest* back unless we could somehow dispose of $(\cup\, e)$ in the first conjunct. The following fact suffices for our purpose:

$$
\neg\ \mathit{Cycle}\,E\,e \quad \Rightarrow \quad (\,\mathit{Cycle}\ ((E \cup e) - d)\ d \ \ \equiv \ \ \mathit{Cycle}\ (E - d)\ d\,)
$$

provided $d \subseteq E$ and $e \subseteq \overline{E}$; its proof is omitted. We then have that

$$
\mathit{Forest}\,(E \cup e) \ \ \equiv \ \ \mathit{Forest}\,E \ \ \wedge \ \neg\ \mathit{Cycle}\,E\,e \qquad \text{provided } \ e \subseteq \overline{E}.
$$

Unfold definition (2.2) of *Cycle* in the equation above, and then the following definitions suggest themselves to make property (3.4) hold:

$$
\begin{array}{rcl}
f & := & \mathit{join} \qquad \text{(and therefore } f\,\emptyset = \mathit{id}) \\
Q\,P\,e & := & \mathit{adj}\ e \not\subseteq P
\end{array}
$$

We now derive operator $\oplus$, using (3.5) as guide:

$$
\begin{array}{cl}
 & \mathit{join}\,(E \cup e) \\
= & \quad \{ \text{ unfold definition of } \mathit{join} \ \} \\
 & (\mathit{adj}\ (E \cup e))^* \\
= & \quad \left\{ \begin{array}{l} \mathit{adj} \text{ distributes over union} \\ \text{(since } \mathit{ran} \text{ and composition do)} \end{array} \right\} \\
 & (\mathit{adj}\ E \cup \mathit{adj}\ e)^* \\
= & \quad \{ \text{ property of closure: } (R \cup S)^* = (R^* \cup S)^* \ \} \\
 & (\mathit{join}\ E \cup \mathit{adj}\ e)^* \\
= & \quad \left\{ \begin{array}{l} \text{note that } \ \mathit{adj}\ e \ = \ (\mathit{x1} \cdot e) \cdot (\mathit{x2} \cdot e)^\circ \ \cup \ (\mathit{x2} \cdot e) \cdot (\mathit{x1} \cdot e)^\circ, \\ \text{property of closure:} \\ \quad \text{if } P \text{ is an equivalence and } a, b \text{ are elements, then} \\ \quad (P \cup a{\cdot}b^\circ \cup b{\cdot}a^\circ)^* \ = \ P \cdot (\mathit{id} \cup a{\cdot}b^\circ \cup b{\cdot}a^\circ) \cdot P \end{array} \right\} \\
 & \mathit{join}\ E \cdot (\mathit{id} \cup \mathit{adj}\ e) \cdot \mathit{join}\ E
\end{array}
$$

Hence, we define:

$$
P \oplus e \ := \ P \cdot (\mathit{id} \cup \mathit{adj}\ e) \cdot P
$$

Variable y of the generic program gets to hold an equivalence relation in this instance. A common way of implementing equivalence relations is through their corresponding partitions. The well-known *Union-Find* problem of manipulating partitions has been thoroughly studied (see [4] for a review) and efficient implementations are available, although a satisfactory solution for functional settings still needs to be found. Its operations can be expressed, using the related equivalence relations instead of the partitions themselves, as follows:

$$
\begin{array}{rcl}
\mathit{Same}\,P\,a\,b & := & a{\cdot}b^\circ \subseteq P \\
\mathit{Union}\,P\,a\,b & := & P \cdot (\mathit{id} \cup a{\cdot}b^\circ \cup b{\cdot}a^\circ) \cdot P
\end{array}
$$

where operation *Same* is built on, and corresponds to the use given to, *Find*. Components Q and $\oplus$ calculated above can be defined in terms of, respectively, *Same* and *Union* using the extreme vertices of edge e as parameters.

5 Conclusions

Other calculational treatments of graph algorithms include [2, 7, 11, 13, 14]. In all of them, with the exception of [7], the calculations performed on the algorithmic component of the problems can be discriminated from the calculations on the graphs, which are just data manipulated by the algorithms. In appealing to the "separation of concerns" principle, one should aim at making clear at all stages what part of the product under construction is being worked on. Such a separation is not very clear in [11, 13, 14], where the relational frameworks used aid the manipulation of both sides of the coin. In [2], where an algebra of matrices for modelling graphs is combined with conventional techniques of imperative programming, the separation is clear. Gibbons's work [7] is exceptional since it proposes an algebraic datatype of (a restricted kind of) graphs, and is based on the view that the structure of the data dictates the structure of the program.

We have presented a development in which the manipulation of graphs is clearly separated from the algorithmic principles involved –in fact, the derivation of the algorithmic component was not even presented–. Thus, the underlying proposal is the addition of relational calculi to the toolbox that already combines Morgan's refinement calculus and predicate calculus.

Acknowledgements

Thanks to Richard Bird for encouragement, a great deal of patience in supervising my work, and his endless concern with improving style of presentation. I am also grateful to the members of the Algebra of Programming Group at Oxford for much support, particularly Oege de Moor, with his amazing ability to do "a bit of everything" including to encourage the rest of us to do our bits. Financial support from Linacre College is gratefully acknowledged. Finally, Cristina Zoltan is responsible for my interest in programming as a formal activity, and for my finding a way at all through the academic jungle.

References

[1] C.J. Aarts, R.C. Backhouse, P. Hoogendijk, E. Voermans, and J. van der Woude. A relational theory of datatypes. Working document, December 1992. Available from URL `http://www.win.tue.nl/win/cs/wp/`.

[2] R.C. Backhouse, J.P.H.W. van den Eijnde, and A.J.M. van Gasteren. Calculating path algorithms. *Science of Computer Programming*, 22(1-2), April 1994.

[3] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall, 1996.

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

[5] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[6] L.R. Foulds. *Graph Theory Applications.* Springer-Verlag, 1992.

[7] J. Gibbons. An initial-algebra approach to directed acyclic graphs. In B. Möller, editor, *Mathematics of Program Construction.* Springer-Verlag LNCS 947, 1995.

[8] C.A.R. Hoare and J. He. The weakest prespecification. *Information Processing Letters*, 24(2), 1987.

[9] A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice-Hall, 1990.

[10] J.H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis.* Addison-Wesley, 1990.

[11] B. Möller and M. Russling. Shorter paths to graph algorithms. *Science of Computer Programming*, 22(1-2), April 1994.

[12] C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[13] J.N. Ravelo. A class of graph algorithms. Qualifying dissertation for transfer to DPhil status, Oxford University Computing Laboratory, January 1996. Available from URL `http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/`.

[14] M. Russling. A general scheme for breadth-first graph traversal. In B. Möller, editor, *Mathematics of Program Construction.* Springer-Verlag LNCS 947, 1995.

[15] G. Schmidt and T. Ströhlein. *Relations and Graphs.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.