Abstract

This formulation of the Roy-Floyd-Warshall algorithm for the transitive closure bypasses matrices and arrays, but uses a more direct mathematical model with adjacency functions for immediate predecessors and successors. This can be implemented efficiently in functional programming languages and is particularly adequate for sparse relations.

Contents

1 Transitive closure algorithm 1

2 Correctness proof 2

2.1 Miscellaneous lemmas 2

2.2 Bounded closure 3

2.3 Main theorem 5

3 Alternative formulation 5

1 Transitive closure algorithm

The Roy-Floyd-Warshall algorithm takes a finite relation as input and produces its transitive closure as output. It iterates over all elements of the field of the relation and maintains a cumulative approximation of the result: step 0 starts with the original relation, and step $Suc\ n$ connects all paths over the intermediate element $n$. The final approximation coincides with the full transitive closure.

This algorithm is often named after “Floyd”, “Warshall”, or “Floyd-Warshall”, but the earliest known description is due to B. Roy [1].

Subsequently we use a direct mathematical model of the relation, bypassing matrices and arrays that are usually seen in the literature. This is more efficient for sparse relations: only the adjacency for immediate predecessors and successors needs to be maintained, not the square of all possible
combinations. Moreover we do not have to worry about mutable data structures in a multi-threaded environment. See also the graph implementation in the Isabelle sources \$ISABELLE_HOME/src/Pure/General/graph.ML and \$ISABELLE_HOME/src/Pure/General/graph.scala.

**type-synonym** relation = \((\text{nat} \times \text{nat})\) set

**fun** steps :: relation \(\Rightarrow\) nat \(\Rightarrow\) relation
**where**
steps rel 0 = rel
| steps rel (Suc n) =
  steps rel n \(\cup\) \(\{(x, y). (x, n) \in \text{steps rel n} \land (n, y) \in \text{steps rel n}\}\)

Implementation view on the relation:
**definition** preds :: relation \(\Rightarrow\) nat \(\Rightarrow\) nat set
**where** preds rel y = \(\{x. (x, y) \in \text{rel}\}\)

**definition** succs :: relation \(\Rightarrow\) nat \(\Rightarrow\) nat set
**where** succs rel x = \(\{y. (x, y) \in \text{rel}\}\)

**lemma**
steps rel (Suc n) =
  steps rel n \(\cup\) \(\{(x, y). x \in \text{preds (steps rel n)} \land y \in \text{succs (steps rel n)}\}\)
**by** (simp add: preds-def succs-def)

The main function requires an upper bound for the iteration, which is left unspecified here (via Hilbert’s choice).

**definition** is-bound :: relation \(\Rightarrow\) nat \(\Rightarrow\) bool
**where** is-bound rel n \(\leftarrow\rightarrow\) \((\forall m \in \text{Field rel}. m < n)\)

**definition** transitive-closure rel = steps rel (SOME n. is-bound rel n)

2 Correctness proof

2.1 Miscellaneous lemmas

**lemma** finite-bound:
  **assumes** finite rel
  **shows** \(\exists n. \text{is-bound rel n}\)
  **using** assms
  **proof** induct
  **case** empty
  **then show** ?case **by** (simp add: is-bound-def)
  **next**
  **case** (insert p rel)
  **then obtain** n where n: \(\forall m \in \text{Field rel}. m < n\)
  **unfolding** is-bound-def **by** blast
  **obtain** x y where p = (x, y) **by** (cases p)
  **then have** \(\forall m \in \text{Field (insert p rel)}. m < \max (\text{Suc x}) (\max (\text{Suc y}) n)\)
using \( n \) by auto
then show \(?case\)
unfolding is-bound-def by blast
qed

lemma steps-Suc: \((x, y) \in \text{steps } \text{rel } (\text{Suc } n) \iff (x, y) \in \text{steps } \text{rel } n \lor (x, n) \in \text{steps } \text{rel } n \land (n, y) \in \text{steps } \text{rel } n\)
by auto

lemma steps-cases:
assumes \((x, y) \in \text{steps } \text{rel } (\text{Suc } n)\)
obtains (copy) \((x, y) \in \text{steps } \text{rel } n\)
| \((\text{step }) (x, n) \in \text{steps } \text{rel } n \land (n, y) \in \text{steps } \text{rel } n\)
using assms by auto

lemma steps-rel: \((x, y) \in \text{rel } \Rightarrow (x, y) \in \text{steps } \text{rel } n\)
by (induct \( n \)) auto

2.2 Bounded closure

The bounded closure connects all transitive paths over elements below a given bound. For an upper bound of the relation, this coincides with the full transitive closure.

inductive-set \( \text{Clos} :: \text{relation } \Rightarrow \text{nat } \Rightarrow \text{relation} \)
for \( \text{rel} :: \text{relation} \) and \( n :: \text{nat} \)
where
\( \text{base} : (x, y) \in \text{rel } \Rightarrow (x, y) \in \text{Clos } \text{rel } n \)
| \( \text{step} : (x, z) \in \text{Clos } \text{rel } n \Rightarrow (z, y) \in \text{Clos } \text{rel } n \Rightarrow z < n \Rightarrow (x, y) \in \text{Clos } \text{rel } n \)

theorem Clos-closure:
assumes is-bound rel n
shows \((x, y) \in \text{Clos } \text{rel } n \iff (x, y) \in \text{rel }^+\)
proof
assume \((x, y) \in \text{Clos } \text{rel } n\)
then show \((x, y) \in \text{rel }^+\) by induct simp-all
next
assume \((x, y) \in \text{rel }^+\)
then show \((x, y) \in \text{Clos } \text{rel } n\)
proof (induct rule: trancl-induct)
  case (base y)
  then show \(?case\) by (rule Clos.base)
next
  case (step y z)
  from \( (y, z) \in \text{rel} \) have 1: \((y, z) \in \text{Clos } \text{rel } n\) by (rule base)
  from \( (y, z) \in \text{rel} \land \text{is-bound rel } n \) have 2: \(y < n\)
  unfolding is-bound-def Field-def by blast
  from step(3) 1 2 show \(?case\) by (rule Clos.step)
qed
lemma Clos-Suc:
  assumes \((x, y) \in \text{Clos } \text{rel } n\)
  shows \((x, y) \in \text{Clos } \text{rel } (\text{Suc } n)\)
  using \text{assms by induct } (\text{auto intro: Clos.intros})

In each step of the algorithm the approximated relation is exactly the bounded closure.

**theorem** steps-Clos-equiv: \((x, y) \in \text{steps } \text{rel } n \iff (x, y) \in \text{Clos } \text{rel } n\)

**proof** (induct \(n\) arbitrary: \(x\) \(y\))

1. **case** \(0\)
   show \}?case
   proof
   assume \((x, y) \in \text{steps } \text{rel } 0\)
   then have \((x, y) \in \text{rel} \) by simp
   then show \((x, y) \in \text{Clos } \text{rel } 0\) by (rule Clos.base)
   next
   assume \((x, y) \in \text{Clos } \text{rel } 0\)
   then show \((x, y) \in \text{steps } \text{rel } 0\) by cases simp-all
   qed

2. **next**
   **case** \((\text{Suc } n)\)
   show \}?case
   proof
   assume \((x, y) \in \text{steps } \text{rel } (\text{Suc } n)\)
   then show \((x, y) \in \text{Clos } \text{rel } (\text{Suc } n)\)
   proof (cases rule: steps-cases)
   case \text{copy}
   with \text{Suc}(1) have \((x, y) \in \text{Clos } \text{rel } n\) ..
   then show ?thesis by (rule Clos-Suc)
   next
   case \text{step}
   with \text{Suc} have \((x, n) \in \text{Clos } \text{rel } n\) \text{ and } \((n, y) \in \text{Clos } \text{rel } n\)
   by simp-all
   then have \((x, n) \in \text{Clos } \text{rel } (\text{Suc } n)\) \text{ and } \((n, y) \in \text{Clos } \text{rel } (\text{Suc } n)\)
   by (simp-all add: Clos-Suc)
   then show ?thesis by (rule Clos.step) simp
   qed

3. **next**
   assume \((x, y) \in \text{Clos } \text{rel } (\text{Suc } n)\)
   then show \((x, y) \in \text{steps } \text{rel } (\text{Suc } n)\)
   proof induct
   case \(\text{base } x\ y\)
   then show \}?case by (simp add: steps-rel)
   next
   case \(\text{step } x\ z\ y\)
   with \text{Suc} show \}?case
   by (auto simp add: steps-Suc less-Suc-eq intro: Clos.step)
2.3 Main theorem

The main theorem follows immediately from the key observations above. Note that the assumption of finiteness gives a bound for the iteration, although the details are left unspecified. A concrete implementation could choose the the maximum element + 1, or iterate directly over the data structures for the \texttt{preds} and \texttt{succs} implementation.

\textbf{theorem} \texttt{transitive-closure-correctness}:  
\textbf{assumes} finite \texttt{rel}  
\textbf{shows} \texttt{transitive-closure rel} = \texttt{rel}^+  
\textbf{proof} –  
\hspace{1em} let \(?N = \text{SOME} \ n. \ \text{is-bound} \ \texttt{rel} \ n\)  
\hspace{1em} have \text{is-bound:} is-bound \ \texttt{rel} \ ?N  
\hspace{1.5em} by (rule \texttt{someI-ex}) (rule \texttt{finite-bound} [OF \texttt{finite rel}])  
\hspace{1em} \{  
\hspace{2em} fix \(x \ y\)  
\hspace{2em} have \((x, y) \in \texttt{steps rel} \ ?N \longleftrightarrow (x, y) \in \texttt{Clos rel} \ ?N\)  
\hspace{3em} by (rule \texttt{steps-Clos-equiv})  
\hspace{2em} also have \ldots \longleftrightarrow (x, y) \in \texttt{rel}^+  
\hspace{3em} using \text{is-bound by} (rule \texttt{Clos-closure})  
\hspace{2em} finally have \((x, y) \in \texttt{steps rel} \ ?N \longleftrightarrow (x, y) \in \texttt{rel}^+\).  
\hspace{1em} \}  
\hspace{1em} then show \(?thesis\) unfolding \texttt{transitive-closure-def} by \texttt{auto}\n
\textbf{qed}

3 Alternative formulation

The core of the algorithm may be expressed more declaratively as follows, using an inductive definition to imitate a logic-program. This is equivalent to the function specification \texttt{steps} from above.

\textbf{inductive} \texttt{Steps :: relation \Rightarrow nat \Rightarrow nat \times nat \Rightarrow bool}  
\textbf{for} \texttt{rel :: relation}  
\textbf{where}  
\hspace{1em} \texttt{base:} \((x, y) \in \texttt{rel} \Longrightarrow \texttt{Steps rel 0} (x, y)\)  
\hspace{1em} | \texttt{copy:} \texttt{Steps rel n} \((x, y) \Longrightarrow \texttt{Steps rel} \ (\texttt{Suc n}) (x, y)\)  
\hspace{1em} | \texttt{step:} \texttt{Steps rel n} \((x, n) \Longrightarrow \texttt{Steps rel n} \ (n, y) \Longrightarrow \texttt{Steps rel} \ (\texttt{Suc n}) \ (x, y)\)

\textbf{lemma} \texttt{steps-equiv:} \((x, y) \in \texttt{steps rel} n \longleftrightarrow \texttt{Steps rel} n (x, y)\)  
\textbf{proof}  
\hspace{1em} assume \((x, y) \in \texttt{steps rel} n\)  
\hspace{1em} then show \texttt{Steps rel n} \((x, y)\)  
\hspace{1em} proof (induct \textit{n} arbitrary: \textit{x y})  
\hspace{1.5em} case 0

\textbf{qed}
then have \((x, y) \in \text{rel}\) by simp
then show \(?\text{case}\) by (rule base)

next
case \((\text{Suc} \ n)\)
from \(\text{Suc}(2)\) show \(?\text{case}\)
proof (cases rule: steps-cases)
case copy
with \(\text{Suc}(1)\) have Steps rel \(n\) \((x, y)\).
then show \(?\text{thesis}\) by (rule Steps.copy)
next
case step
with \(\text{Suc}(1)\) have Steps rel \(n\) \((x, n)\) and Steps rel \(n\) \((n, y)\)
by simp-all
then show \(?\text{thesis}\) by (rule Steps.step)
qed

next
assume Steps rel \(n\) \((x, y)\)
then show \((x, y) \in \text{steps rel } n\)
by induct simp-all
qed

References