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

2 Correctness proof
   2.1 Miscellaneous lemmas
   2.2 Bounded closure
   2.3 Main theorem

3 Alternative formulation

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 $\text{Suc } n$ 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**

\[
\text{relation} = (\text{nat} \times \text{nat}) \text{ set}
\]

**Function**

\[
\text{fun steps :: relation } \Rightarrow \text{nat } \Rightarrow \text{relation}
\]

**Definition**

\[
\text{where}
\]

\[
\begin{align*}
\text{steps rel } 0 &= \text{rel} \\
\text{steps rel } (\text{Suc } n) &= \\
&\text{steps rel } n \cup \{ (x, y). (x, n) \in \text{steps rel } n \land (n, y) \in \text{steps rel } n \}
\end{align*}
\]

**Implementation view on the relation:**

**Definition**

\[
\text{definition preds :: relation } \Rightarrow \text{nat } \Rightarrow \text{nat set}
\]

**Definition**

\[
\text{where preds rel } y = \{ x. (x, y) \in \text{rel} \}
\]

**Definition**

\[
\text{definition succs :: relation } \Rightarrow \text{nat } \Rightarrow \text{nat set}
\]

**Definition**

\[
\text{where succs rel } x = \{ y. (x, y) \in \text{rel} \}
\]

**Lemma**

\[
\text{steps rel } (\text{Suc } n) = \\
\text{steps rel } n \cup \{ (x, y). x \in \text{preds } (\text{steps rel } n) n \land y \in \text{succs } (\text{steps rel } n) n \}
\]

**Proof**

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**

\[
\text{definition is-bound :: relation } \Rightarrow \text{nat } \Rightarrow \text{bool}
\]

**Definition**

\[
\text{where is-bound rel } n \longleftrightarrow (\forall m \in \text{Field rel}. \ m < n)
\]

**Definition**

\[
\text{definition transitive-closure rel } = \text{steps rel } (\text{SOME } n. \text{is-bound rel } n)
\]

## 2 Correctness proof

### 2.1 Miscellaneous lemmas

**Lemma**

\[
\text{finite-bound:}
\]

**Assumes**

\[
\text{finite rel}
\]

**Shows**

\[
\exists n. \text{is-bound rel } n
\]

**Using**

\[
\text{assms}
\]

**Proof**

\[
\text{induct}
\]

**Case**

\[
\text{empty}
\]

**Then Show**

\[
?\text{case by (simp add: is-bound-def)}
\]

**Next**

\[
\text{case } (\text{insert } p \text{ rel})
\]

**Then Obtain**

\[
\text{n where } n: \forall m \in \text{Field rel}. \ m < n
\]

**Unfolding**

\[
\text{is-bound-def by blast}
\]

**Obtain**

\[
x y \text{ where } p = (x, y) \text{ by } \text{(cases } p)
\]

**Then Have**

\[
\forall m \in \text{Field } (\text{insert } p \text{ rel}). \ m < \max (\text{Suc } x) (\max (\text{Suc } y) n)
\]
using n by auto
then show ?case
  unfolding is-bound-def by blast
qed

lemma steps-Suc: (x, y) ∈ steps rel (Suc n) ↔
(x, y) ∈ steps rel n ∨ (x, n) ∈ steps rel n ∧ (n, y) ∈ steps rel n
by auto

lemma steps-cases: assumes (x, y) ∈ steps rel (Suc n)
obtains (copy) (x, y) ∈ steps rel n
| (step) (x, n) ∈ steps rel n and (n, y) ∈ steps rel n
using assms by auto

lemma steps-rel: (x, y) ∈ rel ⇒ (x, y) ∈ steps 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 Clos :: relation ⇒ nat ⇒ relation
for rel :: relation and n :: nat
where
  base: (x, y) ∈ rel ⇒ (x, y) ∈ Clos rel n
| step: (x, z) ∈ Clos rel n ⇒ (z, y) ∈ Clos rel n ⇒ z < n ⇒
  (x, y) ∈ Clos rel n

theorem Clos-closure:
assumes is-bound rel n
shows (x, y) ∈ Clos rel n ↔ (x, y) ∈ rel+x
proof
  show (x, y) ∈ rel+x if (x, y) ∈ Clos rel n
    using that by induct simp-all
  show (x, y) ∈ Clos rel n if (x, y) ∈ rel+x
    using that
proof (induct rule: trancl-induct)
  case (base y)
  then show ?case by (rule Clos.base)
next
  case (step y z)
  from ⟨(y, z) ∈ rel⟩ have 1: (y, z) ∈ Clos rel n by (rule base)
  from ⟨(y, z) ∈ rel⟩ and 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
qed
lemma Clos-Suc:
  assumes \((x, y) \in \text{Clos rel } n\)
  shows \((x, y) \in \text{Clos rel } (\text{Suc } n)\)
  using assms by induct (auto intro: Clos.intros)

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

theorem steps-Clos-equiv: \((x, y) \in \text{steps rel } n \iff (x, y) \in \text{Clos rel } n\)
proof (induct n arbitrary: \(x\) \(y\))
  case 0
  show ?case
  proof
    show \((x, y) \in \text{Clos rel } 0\) if \((x, y) \in \text{steps rel } 0\)
    proof
      from that have \((x, y) \in \text{rel}\) by simp
      then show ?thesis by (rule Clos.base)
    qed
    show \((x, y) \in \text{steps rel } 0\) if \((x, y) \in \text{Clos rel } 0\)
      using that by cases simp-all
    qed
  next
  case (Suc \(n\))
  show ?case
  proof
    show \((x, y) \in \text{Clos rel } (\text{Suc } n)\) if \((x, y) \in \text{steps rel } (\text{Suc } n)\)
      using that
    proof (cases rule: steps-cases)
      case copy
      with Suc(1) have \((x, y) \in \text{Clos rel } n\) ..
      then show ?thesis by (rule Clos-Suc)
    next
    case step
    with Suc have \((x, n) \in \text{Clos rel } n\) and \((n, y) \in \text{Clos rel } n\)
      by simp-all
    then have \((x, n) \in \text{Clos rel } (\text{Suc } n)\) and \((n, y) \in \text{Clos rel } (\text{Suc } n)\)
      by (simp-all add: Clos-Suc)
    then show ?thesis by (rule Clos.step) simp
    qed
    show \((x, y) \in \text{steps rel } (\text{Suc } n)\) if \((x, y) \in \text{Clos rel } (\text{Suc } n)\)
      using that
    proof induct
      case (base \(x\) \(y\))
      then show ?case by (simp add: steps-rel)
    next
    case (step \(x\) \(z\) \(y\))
    with Suc show ?case
      by (auto simp add: steps-Suc less-Suc-eq intro: Clos.step)
    qed
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 maximum element + 1, or iterate directly over the data structures for the \texttt{preds} and \texttt{succs} implementation.

\textbf{theorem} \textit{transitive-closure-correctness}:
\begin{itemize}
  \item assumes \texttt{finite rel}
  \item shows \texttt{transitive-closure rel = rel+}
\end{itemize}
\textbf{proof} –
\begin{itemize}
  \item let \texttt{?N = SOME n, is-bound rel n}
  \item have \texttt{is-bound: is-bound rel ?N}
    \begin{itemize}
      \item by (rule someI-ex) (rule finite-bound \{OF finite rel\})
    \end{itemize}
  \item have \texttt{(x, y) ∈ steps rel ?N ⟷ (x, y) ∈ rel+ for x y}
  \item proof –
    \begin{itemize}
      \item have \texttt{(x, y) ∈ steps rel ?N ⟷ (x, y) ∈ Clos rel ?N}
        \begin{itemize}
          \item by (rule steps-Clos-equiv)
        \end{itemize}
      \item also have \texttt{... ⟷ (x, y) ∈ rel+}
        \begin{itemize}
          \item using \texttt{is-bound} by (rule Clos-closure)
        \end{itemize}
      \item finally show \texttt{?thesis}.
    \end{itemize}
  \item qed
  \item then show \texttt{?thesis unfolding transitive-closure-def by auto}
\end{itemize}
\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} \textit{Steps} :: \textit{relation ⇒ nat ⇒ nat × nat ⇒ bool}
\textbf{for} \textit{rel :: relation}
\textbf{where}
\begin{itemize}
  \item base: \texttt{(x, y) ∈ rel ⇒ Steps rel 0 (x, y)}
  \item copy: \texttt{Steps rel n (x, y) ⇒ Steps rel (Suc n) (x, y)}
  \item step: \texttt{Steps rel n (x, n) ⇒ Steps rel n (n, y) ⇒ Steps rel (Suc n) (x, y)}
\end{itemize}

\textbf{lemma} \textit{steps-equiv}: \texttt{(x, y) ∈ steps rel n ⟷ Steps rel n (x, y)}
\textbf{proof}
\begin{itemize}
  \item show \texttt{Steps rel n (x, y) if (x, y) ∈ steps rel n}
    \begin{itemize}
      \item using that
    \end{itemize}
  \item proof (induct n arbitrary: x y)
    \begin{itemize}
      \item case 0
        \begin{itemize}
          \item then have \texttt{(x, y) ∈ rel} by \texttt{simp}
        \end{itemize}
    \end{itemize}
\end{itemize}
then show ?case by (rule base)

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

show (x, y) ∈ steps rel n if Steps rel n (x, y)
using that by induct simp-all
qed

References