Transitive closure according to Roy-Floyd-Warshall

Makarius Wenzel

February 23, 2021

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.

```plaintext
type-synonym relation = (nat × nat) set

fun steps :: relation ⇒ nat ⇒ relation
where
  steps rel 0 = rel
| steps rel (Suc n) =
  steps rel n ∪ {(x, y). (x, n) ∈ steps rel n ∧ (n, y) ∈ steps rel n}

Implementation view on the relation:
definition preds :: relation ⇒ nat ⇒ nat set
where
  preds rel y = {x. (x, y) ∈ rel}
definition succs :: relation ⇒ nat ⇒ nat set
where
  succs rel x = {y. (x, y) ∈ rel}

lemma
  steps rel (Suc n) =
  steps rel n ∪ {(x, y). x ∈ preds (steps rel n) ∧ y ∈ succs (steps rel n) 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 ⇒ nat ⇒ bool
where
  is-bound rel n ←→ (∀m ∈ Field rel. m < n)
definition transitive-closure rel = steps rel (SOME n. is-bound rel n)
```

2 Correctness proof

2.1 Miscellaneous lemmas

```plaintext
lemma finite-bound:
  assumes finite rel
  shows ∃n. 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: ∀m ∈ Field rel. m < n
    unfolding is-bound-def by blast
  obtain x y where p = (x, y) by (cases p)
  then have ∀m ∈ Field (insert p rel). m < max (Suc x) (max (Suc y) n)
```

2
using $n$ by auto
then show ?case
  unfolding is-bound-def by blast
qed

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

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

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

theorem Clos-closure:
  assumes is-bound rel $n$
  shows $(x, y) \in \text{Clos rel } n \iff (x, y) \in \text{rel}^+$
proof
  show $(x, y) \in \text{rel}^+$ if $(x, y) \in \text{Clos rel } n$
  using that by induct simp-all
  show $(x, y) \in \text{Clos rel } n$ if $(x, y) \in \text{rel}^+$
  using that
proof (induct rule: trancl-induct)
  case (base $y$)
  then show ?case by (rule Clos.base)
next
  case (step $y$ $z$)
  from $(y, z) \in \text{rel}$ have 1: $(y, z) \in \text{Clos rel } n$ by (rule base)
  from $(y, z) \in \text{rel}$ 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)
qued
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
qed

4
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 `preds` and `sucs` implementation.

**Theorem** transitive-closure-correctness:
assumes finite rel
shows transitive-closure rel = rel^+

**Proof** –
let \(?N = SOME n. is-bound rel n\)

have is-bound: is-bound rel \(?N\)

by (rule someI-ex) (rule finite-bound [OF (finite rel)])

have \((x, y) \in \text{steps rel } ?N \iff (x, y) \in \text{rel}^+ \text{ for } x y\)

**Proof** –

have \((x, y) \in \text{steps rel } ?N \iff (x, y) \in \text{Clos rel } ?N\)

by (rule steps-Clos-equiv)

also have \(... \iff (x, y) \in \text{rel}^+\)

using is-bound by (rule Clos-closure)

finally show \(?\text{thesis}\).

qed

then show \(?\text{thesis}\) unfolding transitive-closure-def by auto

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 `steps` from above.

**Inductive** Steps :: relation ⇒ nat ⇒ nat × nat ⇒ bool

**For** rel :: relation

where

base: Steps rel 0 (x, y) if (x, y) ∈ rel

| copy: Steps rel (Suc n) (x, y) if Steps rel n (x, y)

| step: Steps rel (Suc n) (x, y) if Steps rel n (x, n) and Steps rel n (n, y)

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

**Proof**

show Steps rel n (x, y) if (x, y) ∈ steps rel n

using that

**Proof** (induct n arbitrary: x y)

case 0

then have (x, y) ∈ rel by simp

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