# Transitive closure according to Roy-Floyd-Warshall

Makarius Wenzel

April 17, 2016

**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

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 = (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) 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

```
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)
```

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

**lemma** *steps-Suc*: $(x, y) \in$ *steps rel* $(Suc\ n) \longleftrightarrow$
$(x, y) \in$ *steps rel n* $\vee$ $(x, n) \in$ *steps rel n* $\wedge$ $(n, y) \in$ *steps rel n*
**by** *auto*

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

**lemma** *steps-rel*: $(x, y) \in rel \Longrightarrow (x, y) \in$ *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*: $(x, y) \in rel \Longrightarrow (x, y) \in$ *Clos rel n*
| *step*: $(x, z) \in$ *Clos rel n* $\Longrightarrow (z, y) \in$ *Clos rel n* $\Longrightarrow z < n \Longrightarrow$
$(x, y) \in$ *Clos rel n*

**theorem** *Clos-closure*:
**assumes** *is-bound rel n*
**shows** $(x, y) \in$ *Clos rel n* $\longleftrightarrow (x, y) \in rel^+$
**proof**
**show** $(x, y) \in rel^+$ **if** $(x, y) \in$ *Clos rel n*
**using** *that* **by** *induct simp-all*
**show** $(x, y) \in$ *Clos rel n* **if** $(x, y) \in rel^+$
**using** *that*
**proof** (*induct rule*: *trancl-induct*)
**case** (*base y*)
**then show** *?case* **by** (*rule Clos.base*)
**next**
**case** (*step y z*)
**from** ‹$(y, z) \in rel$› **have** 1: $(y, z) \in$ *Clos rel n* **by** (*rule base*)
**from** ‹$(y, z) \in rel$› **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) ∈ Clos rel n
  shows (x, y) ∈ Clos rel (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) ∈ steps rel n ⟷ (x, y) ∈ Clos rel n
proof (induct n arbitrary: x y)
  case 0
  show ?case
  proof
    show (x, y) ∈ Clos rel 0 if (x, y) ∈ steps rel 0
    proof −
      from that have (x, y) ∈ rel by simp
      then show ?thesis by (rule Clos.base)
    qed
    show (x, y) ∈ steps rel 0 if (x, y) ∈ Clos rel 0
      using that by cases simp-all
  qed
next
  case (Suc n)
  show ?case
  proof
    show (x, y) ∈ Clos rel (Suc n) if (x, y) ∈ steps rel (Suc n)
      using that
    proof (cases rule: steps-cases)
      case copy
      with Suc(1) have (x, y) ∈ Clos rel n ..
      then show ?thesis by (rule Clos-Suc)
    next
      case step
      with Suc have (x, n) ∈ Clos rel n and (n, y) ∈ Clos rel n
        by simp-all
      then have (x, n) ∈ Clos rel (Suc n) and (n, y) ∈ Clos rel (Suc n)
        by (simp-all add: Clos-Suc)
      then show ?thesis by (rule Clos.step) simp
    qed
    show (x, y) ∈ steps rel (Suc n) if (x, y) ∈ Clos rel (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
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 the maximum element + 1, or iterate directly over the data structures for the *preds* and *succs* 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) ∈ steps rel ?N ⟷ (x, y) ∈ rel⁺ for x y
  proof −
    have (x, y) ∈ steps rel ?N ⟷ (x, y) ∈ Clos rel ?N
      by (rule steps-Clos-equiv)
    also have … ⟷ (x, y) ∈ rel⁺
      using is-bound by (rule Clos-closure)
    finally show ?thesis .
  qed
  then show ?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: (x, y) ∈ rel ⟹ Steps rel 0 (x, y)
| copy: Steps rel n (x, y) ⟹ Steps rel (Suc n) (x, y)
| step: Steps rel n (x, n) ⟹ Steps rel n (n, y) ⟹ Steps rel (Suc n) (x, y)

lemma steps-equiv: (x, y) ∈ steps rel n ⟷ 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


[1] B. Roy. Transitivité et connexité. In *Extrait des comptes rendus des séances de l'Académie des Sciences*, pages 216–218. Gauthier-Villars, July 1959. http://gallica.bnf.fr/ark:/12148/bpt6k3201c/f222.image.langFR.