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.

**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\}
⟨proof⟩

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
⟨proof⟩

lemma steps-Suc: (x, y) ∈ steps rel (Suc n) ⟷
  (x, y) ∈ steps rel n ∨ (x, n) ∈ steps rel n ∧ (n, y) ∈ steps rel n
⟨proof⟩

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
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) ∈ Clos rel n if (x, y) ∈ rel
  | step: (x, y) ∈ Clos rel n if (x, z) ∈ Clos rel n and (z, y) ∈ Clos rel n and z < n
```

```
theorem Clos-closure:
  assumes is-bound rel n
  shows (x, y) ∈ Clos rel n ←→ (x, y) ∈ rel

lemma Clos-Suc:
  assumes (x, y) ∈ Clos rel n
  shows (x, y) ∈ Clos rel (Suc n)
```

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
```

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 succs implementation.
```
theorem transitive-closure-correctness:
  assumes finite rel
  shows transitive-closure rel = rel
```

(proof)
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.

\[
\text{inductive } \text{Steps} :: \text{relation } \Rightarrow \text{nat } \Rightarrow \text{nat } \times \text{nat } \Rightarrow \text{bool} \\
\text{for } \text{rel} :: \text{relation} \\
\text{where} \\
\text{base: } \text{Steps rel } 0 \ (x, y) \ \text{if } (x, y) \in \text{rel} \\
\text{copy: } \text{Steps rel } \text{Suc } n \ (x, y) \ \text{if } \text{Steps rel } n \ (x, y) \\
\text{step: } \text{Steps rel } \text{Suc } n \ (x, y) \ \text{if } \text{Steps rel } n \ (x, n) \ \text{and } \text{Steps rel } n \ (n, y) \\
\text{lemma steps-equiv: } (x, y) \in \text{steps rel } n \iff \text{Steps rel } n \ (x, y) \\
\langle \text{proof} \rangle
\]

References