Formally-Proven Kosaraju's algorithm

Laurent Théry

Formally-Proven Kosaraju's algorithm

Laurent Théry
Laurent.Thery@sophia.inria.fr

Abstract

This notes explains how the Kosaraju's algorithm that computes the strong-connected components of a directed graph has been formalised in the Coq prover using the SSReflect extension.

1 Introduction

Our interest in this algorithm comes from an initial quest for formalising the classic 2-Sat problem : given a set S of binary clauses, checking if S is satisfiable can be done in linear time. The standard justification is to rewrite the satisfiability problem into a graph problem. The transformation is straightforward. The vertices of the graph are the variables occurring in the clauses and their negations. Two edges are created for each clause $l_1 \vee l_2$: one from $\neg l_1$ to l_2 and the other from $\neg l_2$ to l_1. These two edges correspond to logical implications and read as "if l_1 is false then l_2 is true" and "if l_2 is false then l_1 is true" respectively. Figure 1 shows the graph that corresponds to the set $S = \{x_1 \vee x_2, x_1 \vee \neg x_3, \neg x_2 \vee x_3\}$.
Now, the claim is that the initial formula is satisfiable if and only no variable x_i and its negation $\neg x_i$ both belong to the same strongly-connected component of the resulting graph. Let us try to explain this claim informally. First, recall what a strongly-connected component is. Two vertices a and b are strongly connected if a is connected to b and b connected to a. Now, remembering the interpretation of edges as logical implications, two vertices a and b are then in the same component if the constraints given by the set of clauses force their respective truth value to be the same. In other words, if a is true, then its implication b must also be true and if b is true, then its implication a must be true. Obviously, if x_i and $\neg x_i$ are in the same

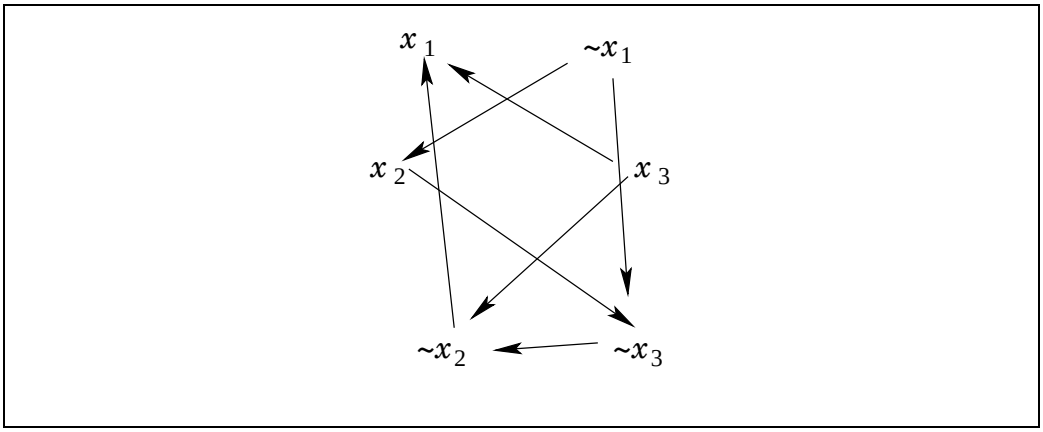

Figure 1: Implication graph for the set $S = \{x_1 \vee x_2, x_1 \vee \neg x_3, \neg x_2 \vee x_3\}$.

component, they cannot have both the same truth value, so the formula is unsatisfiable. A slightly more subtle argument needs to be used to show the converse. We will not explain it here.

Remember that our initial interest was in showing that 2-SAT can be solved in linear time. We are almost there. First, it is clear that the graph can be built in linear time. If there exists a linear algorithm that computes the strongly-connected components of a directed graph, then a single traversal of the components of the graph is sufficient to check the satisfiability of the initial set of clauses. In the literature, the algorithm for computing the strong-component of a graph is usually the one by Tarjan [4]. When trying to figure out if there were some alternatives that may lead to somewhat simpler correctness proofs, we discover the less known but beautiful Kosaraju's algorithm [3]. Its formalisation is what this note is about. We will first explain the algorithm and how it can be programmed inside the COQ system. Then, we will show how a correctness proof can be derived.

2 Kosaraju's algorithm

Figure 2 presents the graph we are going to use throughout this section. It is composed of 9 vertices and 11 edges. An arbitrary finite type T is used in SSREFLECT to represent the vertices

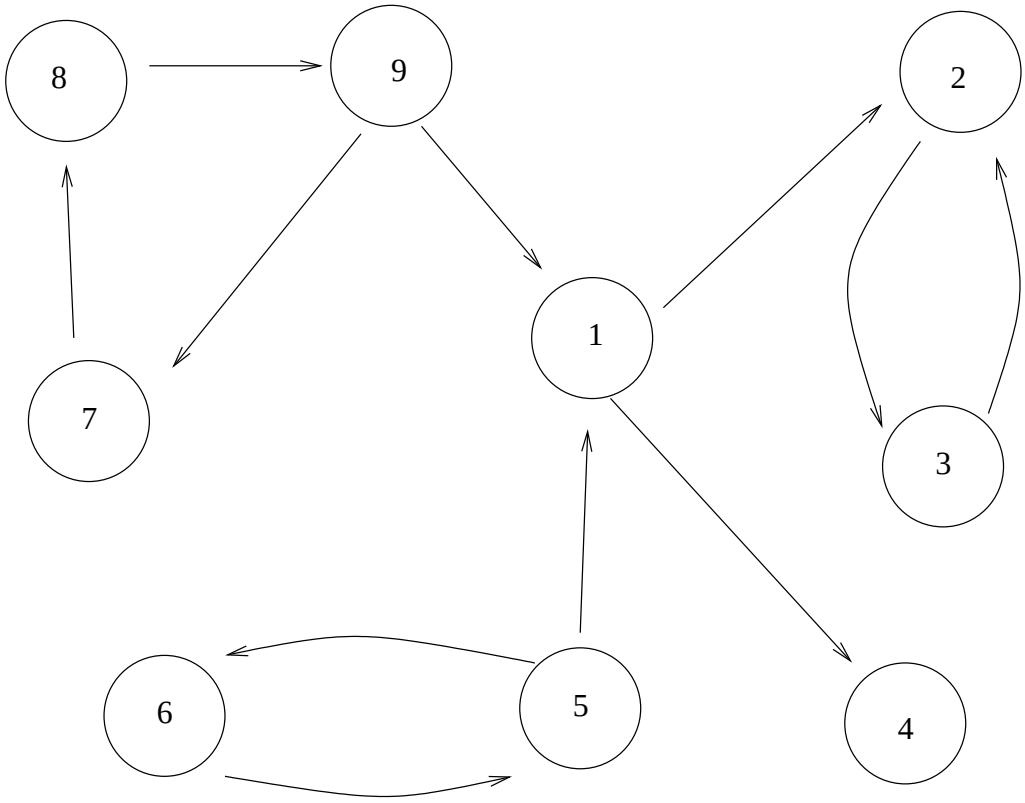

Figure 2: An example of directed graph

```
Variable T : finType.
```

This means that informally we have $T = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for our example. The adjacency relation is then represented by a relation on T

```
Variable r : rel T.
```

where $r\ x\ y$ is true if there is an edge from x to y. This means that we have $r\ 1\ 2$ but not $r\ 1\ 5$ for our example.

Now, given the relation r, in order to manipulate algorithmically the graph, we are going to use the function *rgraph* that computes the neighborhood of a vertex: *rgraph* $r\ x$ returns the sequences of vertices that are directly connected to x. We define g_r as a shortcut for our neighborhood function.

```
Definition g_r := rgraph r.
```

We have for our graph

$$\begin{array}{lcl} g_r\ 1 & = & [::\ 2;\ 4] \\ g_r\ 2 & = & [::\ 3] \\ g_r\ 3 & = & [::\ 2] \\ g_r\ 4 & = & [::] \\ g_r\ 5 & = & [::\ 1;\ 6] \\ g_r\ 6 & = & [::\ 5] \\ g_r\ 7 & = & [::\ 8] \\ g_r\ 8 & = & [::\ 9] \\ g_r\ 9 & = & [::\ 1;\ 7] \end{array}$$

Kosaraju's algorithm has two distinct phases. The first phase sorts topological the vertices. This is done with a variant of the depth-first search algorithm that is already defined in the library[1].

[1]We took some liberty with the actual SSReflect code. We have omitted systematically arguments that Coq sometimes requires in order to ensure termination. There are not essential for the correctness. Also, we have used subscript to expose explicitly the parametricity of our definition over r. The last step of the algorithm requires to take a specific r

```
Function dfs_r l x := if x ∈ l then l else foldl dfs_r (x :: l) (g_r x).
```

This definition uses the function *foldl* that iterates a function on elements of a list, i.e. $foldl\ f\ v\ [::\ x_1;\ x_2;\ \ldots\ x_n] = f\ (f\ (\ \ldots\ f\ (f\ x_1\ v)\ x_2)\ \ldots\ )\ x_n$. The call $dfs_r\ l\ x$ starts the search at x avoiding the vertices in l and returns l plus the vertices that have been encountered. For example, dfs [::] 1 returns [:: 4; 3; 2; 1], i.e. the vertices that are connected to 1.

Unfortunately, this function does not compose very well. If we call again *dfs* on an unreached vertex, let's say 5, avoiding the elements connected to 1 we get [:: 6; 5; 4; 3; 2; 1]. 5 ends up in the sequence between 1 and 6 which is problematic if we are interested by sorting topologically the graph. Of course, the problem comes from the way the vertices are collected, some kind of postfix ordering should be used. Replacing the single list l by a pair of lists fixes the problem. The first list is used to collect the vertices that need to be avoided while the other imposes a postfix order.

```
Function pdfs_r p x :=
    if x ∈ p.1 then p else
    let p' := foldl pdfs_r (x :: p.1, p.2) (g_r x) in (p'.1, x :: p'.2).
```

x is added to the first list beforehand to avoid cycling while it is added afterward for the second element to get the postfix order. Calling $pdfs_r$ ([::], [::]) 1 returns ([:: 4; 3; 2; 1], [:: 1; 4; 2; 3]), the two list represent two different ordering of the vertices connected to 1, the second one being the one we are interested in. Now, calling $pdfs_r$ ([:: 4; 3; 2; 1], [:: 1; 4; 2; 3]) 5 returns ([:: 6; 5; 4; 3; 2; 1], [:: 5; 6; 1; 4; 2; 3]). In the second list, 5 is before 1 and 6.

Getting the ordered sequence of all vertices is then done applying a *foldl* on the sequence of all the vertices given by the function *enum*

```
Definition stack_r := (foldl pdfs_r ([::], [::]) (enum T)).2.
```

If we suppose that *enum* T returns the sequence [:: 1; 2; 3; 4; 5; 6; 7; 8; 9] for our example, the value for *stack* is then [:: 9; 8; 7; 5; 6; 1; 4; 2; 3]. This ends the first phase of the algorithm.

The second phase is even simpler and takes advantage of the following two simple observations. First, the depth-first search function has been called on each element on the stack either at top level if the vertex was not reached yet, or by a recursive call. Second, if x connects to y but y occurs before x in sequence, we can conclude that y connects to x. This is because of the topological ordering. Suppose the call of the search at x was done at top-level with an avoiding sequences l_1 generating all the connected elements l_2. The sub-sequence starting at x would be the exact concatenation of l_1 and l_2. This is impossible, y should occur either in l_1 or in l_2, so also in the sub-sequence starting at x. This means that the call was not made at top-level and it is the depth-first search starting at y that has triggered the one at x. x is then connected to y.

Now if we take x the top of the stack, (in our example $x = 9$), and collect the vertices that connect to x (in our example 7, 8 and 9). What we have said before proves that this is a strongly-connected component. Two observations end the implementation of the second phase. First, computing the vertices that connect to x can be done using the same depth-first algorithm but this time applied on the graph where the edges are flipped. Second, removing the strongly-connected component of x from the stack leads to a new stack where the same process can be iterated. This iteration is implemented once again with a *foldl*.

```
Definition r' := [rel x y | r y x].
Definition kosaraju_r :=
  (foldl (fun p x →
            if x ∈ p.1 then p else
            let p' := pdfs_r' (p.1, [::]) x in (p'.1, p'.2 :: p.2))
    ([::], [::]) stack_r)).2.
```

Figure 3 collects the 11 lines of code that were necessary to implement Kosaraju's algorithm in our functional setting. Applying it on our example gives [:: [:: 2; 3]; [:: 4]; [:: 1]; [:: 5; 6]; [:: 9; 8; 7]].

3 Correctness proof

Finding the right definitions and the intermediate results is the main difficulty of this proof. This is what we are going to explain in this section. We

```
Definition g_r := rgraph r.
Function pdfs_r p x :=
    if x ∈ p.1 then p else
    let p' := foldl pdfs_r (x :: p.1, p.2) (g_r x) in (p'.1, x :: p'.2).
Definition stack_r := (foldl pdfs_r ([::], [::]) (enum T)).2.
Definition r' := [rel x y | r y x ].
Definition kosaraju_r :=
  (foldl (fun p x →
            if x ∈ p.1 then p else
            let p' := pdfs_r' (p.1, [::]) x in (p'.1, p'.2 :: p.2))
    ([::], [::]) stack_r)).2.
```

Figure 3: Kosaraju's algorithm in SSReflect

refer to the Coq source available at
http://www.sop-inria.fr/team/marelle/Laurent.Thery/Kosaraju/Kosaraju.html
for the actual proofs.

As the proof relies heavily on sequences, we first give a quick summary of the functions and predicates from the standard library we have been using in our proof.

- `uniq` l is true if the sequence l does not have duplicate;
- $\{$`subset` $l_1 \leq l_2\}$ is true if all the elements of l_1 are in l_2;
- l_1 `=`$_i$ l_2 is true if l_1 and l_2 have the same elements;
- `[disjoint` l_1 `&` l_2`]` is true if there is no common element between l_1 and l_2.
- `nth` x l n returns the n^{th} element of the sequence. It returns x if the sequence contains less than n elements;
- `[seq` $i \leftarrow l$ `|` P i `]` returns sub-sequence of l composed of the elements of l that verify the property P;
- `flatten` l returns the concatenation of all the sequences in l (l is a sequence of sequences);

- `find` P l returns the index of the first element of the sequence l that verifies the predicate P. It returns the size of the sequence if there is no element of l that verifies P;

- `index` x l returns the index of the first occurrence of x in l. It returns the size of the sequence if x does not occur in l;

In the standard library of SSReflect, there is already the notion of being connected.

```
Definition x ⟶*_r y := y ∈ dfs_r [::] x.
```

A variant of it has been defined, It is the notion of being connected but with some forbidden intermediate vertices.

```
Definition x ⟶*_{r,l} y :=
    let r' := [rel x y | r x y && y ∉ l ] in x ⟶*_{r'} y.
```

It is then possible to give a definition of what it is to be strongly connected

```
Definition x ⟷*_{r,l} y := x ⟶*_{r,l} y && y ⟶*_{r,l} y.
```

This is all we need for the graph path. The next step is to capture the post-fix order of our sequence. We first define the order in a sequence that is taken with respect to first occurrence

```
Definition x ≤_{r,l} y := index x l ≤ index y l.
```

Then, we define a function that computes the canonical vertex associated to a vertex x for a pair p of sequences (l_1, l_2). This canonical vertex is defined as the first vertex in l_2 that is strongly connected with x avoiding l_1.

```
Definition [x]_{r,p} := nth x p.2 (find (fun y → x ⟷*_{r,p.1} y) p.2).
```

The central property in our proof is the notion for a pair of sequences of vertices to be well-formed. A pair (l_1, l_2) is well-formed, if l_1 and l_2 are disjoint and every time one takes a vertex x in l_2 and a vertex y that is connected to x avoiding l_1 then y is also in l_2 and the canonical vertex of x in l_2 avoiding l_1 is before y in l_2.

```
Definition Wf_r p :=
      [disjoint p.1 & p.2]
  ∧
      ∀x y, (x ∈ p.2 ∧ x ⟶*_{r,p.1} y) ⇒ (y ∈ p.2 ∧ [x]_{r,p} ≤_{r,p.2} y)
```

The disjoint part insures that the two sequences keep separated, the "$y \in p.2$ " part that that the list is closed under connection and the "$[x]_{r,p} \leq_{r,p.2} y$" part that it is in post-fix order. Intuitively, it says that canonical elements are always on the left of the elements it connects to.

The first key property that we derive from this definition is some kind of compositionality. It indicates how well-formed sequences can be concatenated.

```
Lemma wf_cat :
    ∀l_1 l_2 l_3, Wf_r (l_1, l_2) ⇒ Wf_r (l_2 ++ l_1, l_3) ⇒ Wf_r (l_1, l_3 ++ l_2).
```

It is used to justify the computation of *foldl* inside the definition of the depth-first search where we accumulate the various sub-searches.

The second key property is related with the body of the *pdfs* and how the search starting at x can be reduced to searching at the vertices immediately connected to x.

```
Lemma wf_cons :
    ∀x l_1 l_2,
      (∀y, y ∈ l_2 ⇒ x ⟶*_{r,l_1} y) ⇒
      (∀y, r x y ⇒ y ∉ x :: l_1 ⇒ y ∈ l_2) ⇒
      x ∉ l_1 ⇒ Wf_r (x :: l_1, l_2) ⇒ Wf_r (l_1, x :: l_2).
```

This property reads like this. Let us suppose that the pair $(x :: l_1, l_2)$ is well-formed. In order to derive that $(l_1, x :: l_2)$ is well-formed it is sufficient

that l_2 is composed of vertices connected to x. Among these vertices, there must be the ones directly connected to x.

It is relatively straightforward to get the correctness of the *pdfs* function from the two previous properties. The statement is somewhat more intricate than expected since we also want to get the extra property that the result we get is without duplicate. Also, we need to clearly state that what is added to the second sequence is well-formed and contains only vertices connected with the initial vertex we start the search with.

```
Lemma pdfs_correct :        ∀p x,
      (uniq p.1 ∧ uniq p.2 ∧ {subset p.2 ≤ p.1}) ⇒
      let p' := pdfs_r p x in
      if x ∈ p.1 then p' = p else
        (uniq p'.1 ∧ uniq p'.2) ∧
        ∃l,
          (x ∈ l ∧ p'.1 =_i l ++ p.1 ∧ p'.2 = l ++ p.2) ∧
          (Wf_r (p.1, l) ∧ ∀y, y ∈ l ⇒ x ⟶*_{r,p.1} y).
```

Once *pdfs_correct* is proved, the correctness of the stack can be easily derived with a simple application of *wf_cat*. The statement of the theorem just states that the stack is well-formed and contains all the vertices.

```
Lemma stack_correct : Wf_r ([::], stack_r) ∧ ∀x, x ∈ stack_r.
```

In order to get the final correctness statement of the Kosaraju's algorithm we just need two extra properties about well-formedness. The first one states that removing the strongly-connected component of the top element of the sequence preserves well-formedness.

```
Lemma wf_inv :
    ∀x l_1 l_2,
      Wf_r (l_1, x :: l_2) ⇒ Wf_r (l_1, [seq y ← l_2 | ¬ x ⟷*_{r,l_1} y ])
```

The second one states that this strongly-connected component can be computed by finding the elements that connects to the top element.

```
Lemma wf_equiv :
  ∀x y l1 l2,
    Wfr (l1, x :: l2) ⇒ (x ⟷*r,l1 y) = ((y ∈ x :: l2) && y ⟷*r,l1 x).
```

With these two properties, it is almost direct to prove the correctness of Kosaraju's algorithm. We only use notions that are present in the standard library for its statement. The result of the algorithm is a sequence of sequence l. When l is flattened, it contains every vertex once and only once and a sequence in l represents a strongly-connected component.

```
Lemma kosaraju_correct :
  (let l := flatten kosarajur in uniq l ∧ ∀x, x ∈ l) ∧
  ∀c, c ∈ kosarajur ⇒ ∃x, ∀y, (y ∈ c) = (x ⟶*r y && y ⟶*r x)
```

Note that components are never empty since we have $x \longrightarrow_r^* x$ for all x.

4 Conclusion

Formalising Kosaraju's algorithm has been an interesting exercise. We have chosen a direct formalisation with almost no abstraction. The key part of the formalisation has been to define the notion of well-formed pairs. Thanks to the SSReflect library on sequences, we could effectively work on sequences and derive the correctness of the algorithm. Some formalisation of algorithms that computes strong components already existed [1, 2]. We believe ours is one of the more concise. If we would like to have an effective program that compute the strong component, the algorithm given in Figure 3 should be further refined. In particular, the *pdfs* function currently works on pairs of sequences of nodes. The first sequence is actually used as a set. So, it could be implemented using an array of booleans. The two operations, membership and addition of an element, could then been be implemented by a look-up and an update respectively.

References

[1] Peter Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *ITP'14*, volume 8558 of *LNCS*, pages 325–340, 2014.

[2] François Pottier. Depth-first search and strong connectivity in Coq. In *JFLA'15*. To appear.

[3] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7:67–72, 1981.

[4] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.