Connected Graphs and Spanning Trees

GAINA, Daniel

Japan Advanced Institute of Science and Technology

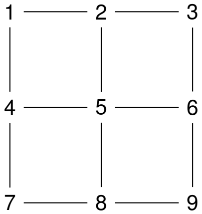
January 22, 2010

Describing the problem I

$G = (V, E)$ - graph

1. V - set of vertices
2. E - (multi)set of edges

Example:



$V = \{1, \ldots, 9\}$
$E = \{< 1, 2 >; < 1, 4 >; < 2, 3 >; < 2, 5 >; < 3, 6 >; < 4, 5 >; < 4, 7 >; < 5, 6 >; < 5, 8 >;$
$< 6, 9 >; < 7, 8 >; < 8, 9 >\}$

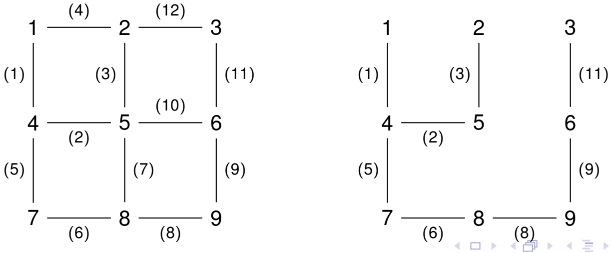
Describing the problem II

$G = (V, E)$ connected;
$T = (V, E')$ spanning tree of G when

1. T tree,
2. $E' \subseteq E$.

Theorem

Every connected graph has a spanning tree.



Towards formalization

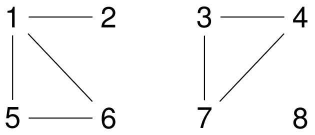
1. $\texttt{connected(G)} \Rightarrow \exists \texttt{G}' \subseteq \texttt{G.tree(G}'\texttt{)}$
2. $\texttt{connected(G)} \Rightarrow \texttt{connected(mktree(G))} \wedge \texttt{nocycle(mktree(G))}$

To do :

1. data representations for mathematical objects (graphs);
2. define
   - `connected`
   - `nocycle`
   - `mktree`

Functions on graphs

$G = (\{1,2,3,4,5,6,7,8\}, < 1,2 >; < 1,6 >; < 1,5 >; < 3,4 >; < 3,7 >; < 4,7 >; < 5,6 >)$



- `mcc(A,G)` = max. connected comp. of `A` in `G`.
  `mcc(6,G)` = $\{1,2,5,6\}$, `mcc(8,G)` = $\{8\}$
- `#cc(G)` = no. of max. connected components
  `#cc(G) = 3`
- `nocycle(G) = false`

Spanning forests I

in the attempt of proving the desired properties we realized is much easier to prove a more general result:

Every graph has a spanning forest!

Definition

A **spanning forest** of a graph is a subgraph that consists of a set of spanning trees, one for each maximal connected component of the initial graph.

We define the function `mktree` which returns the spanning forest of a graph.

Spanning forests II

G=({1,2,3,4,5,6,7,8}, <1,2>;<1,6>;<1,5>;<3,4>;<3,7>;<4,7>;<5,6>)

mktree(G)=({1,2,3,4,5,6,7,8},<1,2>;<1,6>;<1,5>;<3,4>;<3,7>)

Remark

The value is relative to the order chosen for the edges.

Properties to be proved

Assuming that we have defined

- `mcc, #cc, nocycle, mktree`

we need to prove

1. `mcc(A,G)=mcc(A,mktree(G))`
2. `#cc(G)=#cc(mktree(G))`
3. `nocycle(mktree(G))`

Then we define `connected(G):= (#cc(G)=1)` which implies

- $\texttt{connected(G)} \Rightarrow \texttt{connected(mktree(G))} \land \texttt{nocycle(mktree(G))}$

Set I

```
mod* ID {
[Id]
- equality on Id
op _=_ : Id Id -> Bool {comm}
vars I J : Id
eq [i1] : (I = I) = true .
ceq [i2] : I = J if (I = J) .
}

mod* SET(I :: ID) {
[Id < Set]
op empty : -> Set {constr}
op (_U_) : Set Set -> Set {constr assoc comm}
eq (S:Set U S) = S .
vars I I' : Id
vars S S' : Set
```

Set II

```
- (I in S) indicates whether I is an element of S or not
op _in_ :  Id Set -> Bool .
eq I in empty = false .
eq I in I' = if I = I' then true else false fi .
eq I in I' U S = if I = I' then true else I in S fi .

- (S <s S') indicates whether S is subset of S'
op _<s_ :  Set Set -> Bool .
eq empty <s S = true .
eq I <s S = if I in S then true else false fi .
eq (I U S) <s S' = if I in S' then (S <s S')
 else false fi .
```

Set III

```
- equality on Set
op _=_ :  Set Set -> Bool {comm}
eq [s1]:  (S = S) = true .
eq [s2]:  (S = S') = (S <s S') and (S' <s S) .
ceq [s3]:  S = S' if (S = S') .
}
```

GRAPH I

```
mod* VERTEX {
[Vertex]
op _=_ :  Vertex Vertex -> Bool {comm}
vars A B : Vertex
eq [v1] :  (A = A) = true .
ceq [v2] :  A = B if (A = B) .
}

mod* GRAPH(V ::  VERTEX){
[Edge]
[Graph]
op <_,_> :  Vertex Vertex -> Edge {constr}
op nil :  -> Graph {constr}
op _;_ :  Edge Graph -> Graph {constr}
}
```

GRAPH II

Remark

- `Edge` and `Graph` are constrained.
- Models consist of interpretations of terms formed with constructor and elements of sort `Vertex`.

SFOREST I

```
mod* SFOREST (V :: VERTEX){
inc(INT)
inc(SET(V{sort Id -> Vertex})*{sort Set -> VtxSet})
inc(GRAPH(V))

vars A B C : Vertex
var G : Graph

- mcc(A,G) = max. connected component of A in G
op mcc : Vertex Graph -> VtxSet
eq mcc(A,nil) = A .
eq mcc(A, < B,C > ; G) =
 if mcc(A,G) = mcc(B,G) or mcc(A,G) = mcc(C,G)
 then (mcc(B,G) U mcc(C,G)) else mcc(A,G) fi .
```

SFOREST II

```
op nocycle : Graph -> Bool
eq nocycle(nil) = true .
eq nocycle(< A,B > ; G) =
  if mcc(A,G) = mcc(B,G) then false
  else nocycle(G) fi .

- #cc(G) = no. of max. connected comp. of G
op #cc : Graph -> Int .
op #vertices : -> Nat .
eq #cc(nil) = #vertices . - no. of vertices
eq #cc(< A,B > ; G) = if mcc(A,G) = mcc(B,G) then
  #cc(G) else #cc(G) - 1 fi .
```

SFOREST III

```
- mktree(G) returns the spanning forest of G
op mktree :  Graph -> Graph
eq mktree(nil) = nil .
eq mktree(< A,B > ; G) =
 if mcc(A,G) = mcc(B,G) then mktree(G)
 else < A,B > ; mktree(G) fi .
}
```

Properties to be proved

Theorem

`mktree(G)` *is a spanning forest of G.*

1. `∀G.∀A.mcc(A,mktree(G))=mcc(A,G)`
2. `∀G.#cc(mktree(G))=#cc(G)`
3. `∀G.nocycle(mktree(G))`

First Theorem I

Lemma

Max. connected comp. of `G` *are the same as max.connected comp. of* `mktree(G)` *i.e.*
`∀G.∀A.mcc(A,mktree(G))=mcc(A,G)`

Proof by induction on the structure of `G`.

- **IB** `∀A.mcc(A,mktree(nil))=mcc(A,nil)`
- **IS** `∀G.∀A'.mcc(A',mktree(G)) = mcc(A',G)` $\Rightarrow$
  `∀A.∀B.∀C.mcc(A,mktree(<B,C>;G)) = mcc(A,<B,C>;G)`

First Theorem II

For the induction base

```
open SFOREST
op a :  -> Vertex .
red mcc(a,mktree(nil)) = mcc(a,nil) .
close
```

For the induction step

```
ops a b c :  -> Vertex .
op g :  -> Graph .
eq [IH] : mcc(A:Vertex,mktree(g)) = mcc(A,g) .
- equations corresponding to each subcase ...
red mcc(a,mktree(< b,c > ; g)) = mcc(a, < b,c > ; g) .
```

First Theorem III

Equations corresponding to each subcase

1. ```
   eq mcc(b,g) = mcc(c,g)
   eq mcc(a,g) = mcc(c,g)
   ```
2. ```
   eq mcc(b,g) = mcc(c,g) .
   eq (mcc(a,g) = mcc(c,g)) = false .
   ```
3. ```
   eq (mcc(b,g) = mcc(c,g)) = false .
   eq mcc(a,g) = mcc(b,g) .
   ```
4. ```
   eq (mcc(b,g) = mcc(c,g)) = false .
   eq mcc(a,g) = mcc(c,g) .
   ```
5. ```
   eq (mcc(b,g) = mcc(c,g)) = false .
   eq (mcc(a,g) = mcc(b,g)) = false .
   eq (mcc(a,g) = mcc(c,g)) = false .
   ```
   ```

Second Theorem I

Theorem

mktree *preserves the number of maximal connected components, i.e.* ∀G.#cc(mktree(G))=#cc(G).

Proof by induction on the structure of G.

- **IB** #cc(mktree(nil))=#cc(nil)
- **IS** ∀G.#cc(mktree(G)) = #cc(G) ⇒

  ∀B.∀C.#cc(mktree(<B,C>;G)) = #cc(<B,C>;G)

Second Theorem II

For the induction base

```
open SFOREST + EQL
red #cc(mktree(nil)) = #cc(nil) .
close
```

For the induction step

```
open SFOREST + EQL
ops a b :  -> Vertex .
op g :  -> Graph .
eq [IH] : #cc(mktree(g)) = #cc(g) .
```

1. `eq mcc(a,g) = mcc(b,g) .`
2. `eq (mcc(a,g) = mcc(b,g)) = false .`

```
red #cc(mktree(< a,b > ; g)) = #cc(< a,b > ; g) .
```

Third theorem I

Theorem

`mktree(G)` *has no cycles, i.e.*
`∀G.nocycle(mktree(G))=true`.

Proof by induction on the structure of `G`.

- **IB** `nocycle(mktree(nil))=true`
- **IS** `∀G.nocycle(mktree(G)) = true ⇒`
  `∀B.∀C.nocycle(mktree(<B,C>;G)) = true`

Third theorem II

For the induction base

```
open SFOREST
red nocycle(mktree(nil)) .
close
```

For the induction step

```
open SFOREST
ops a b :  -> Vertex .
op g :  -> Graph .
eq [IH] : nocycle(mktree(g)) = true .
```

1. `eq mcc(a,g) = mcc(b,g) .`
2. `eq (mcc(a,g) = mcc(b,g)) = false .`

```
red nocycle(mktree(< a,b > ; g)) .
```

Conclusions

- we have proved a more general property (e.g every graph has a spanning forest) in order to achieve our goal;
- we didn't use initial semantics;
- constructor-based logics sufficient for verifications;
- the data structure `VERTEX` for the set of vertices is very general and can be instantiated with natural numbers;

Exercise

1. Prove `∀G.∀A.∀B.(< A,B > in G) if (< A,B > in mktree(G))`
2.
   - A path between the vertex A and vertex B is a sequence of edges $< A_1, B_1 > \ldots < A_n, B_n >$ such that
     1. $A_1 = A$,
     2. $B_n = B$ and
     3. $A_{i+1} = B_i$ for all $i \in \{1, \ldots, n-1\}$.
   - A cycle is a path $< A_1, B_1 > \ldots < A_n, B_n >$ such that $A_1 = B_n$
   - Prove that if there exists a path between A and B then there exists a path with nocycles