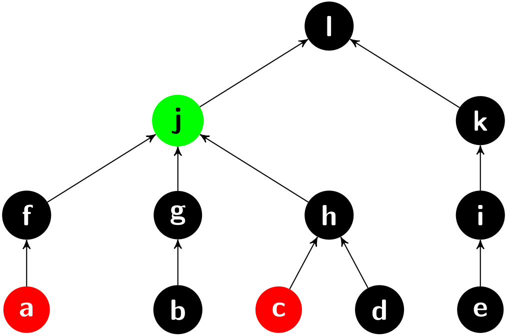
PAINFULLY SLOW: The Offline Least Common Ancestor Problem

Matthew Kilgore

May 5, 2016

Example



Definitions

Let's get some basic facts out of the way. Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$:

Definitions

Let's get some basic facts out of the way. Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$:

- We say y is a descendant of x if the unique path

$$P = (x = x_0, x_1, \ldots, x_n = y)$$

  from x to y is such that x_i is x_{i+1}'s parent.

Definitions

Let's get some basic facts out of the way. Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$:

- We say y is a descendant of x if the unique path

$$P = (x = x_0, x_1, \ldots, x_n = y)$$

  from x to y is such that x_i is x_{i+1}'s parent.
- The least common ancestor of x and y is the vertex v furthest from the root such that x and y are descendants of v.

Definitions

Let's get some basic facts out of the way. Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$:

- We say y is a descendant of x if the unique path

$$P = (x = x_0, x_1, \ldots, x_n = y)$$

  from x to y is such that x_i is x_{i+1}'s parent.
- The least common ancestor of x and y is the vertex v furthest from the root such that x and y are descendants of v.
- Alternatively, the least common ancestor of x and y is the vertex v on the unique path from x to y closest to the root.

The Question

Least Common Ancestor Problem

Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$, efficiently compute the least common ancestor of x and y.

The Question

Least Common Ancestor Problem

Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$, efficiently compute the least common ancestor of x and y.

- This arises in applications such as when examining a class inheritance hierarchy.

The Question

Least Common Ancestor Problem

Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$, efficiently compute the least common ancestor of x and y.

- This arises in applications such as when examining a class inheritance hierarchy.
- Sometimes it makes sense to find the pairs of vertices to query before actually querying.

The Question

Least Common Ancestor Problem

Given a rooted tree $T = (V, E)$ and vertices $x \neq y \in V$, efficiently compute the least common ancestor of x and y.

- This arises in applications such as when examining a class inheritance hierarchy.
- Sometimes it makes sense to find the pairs of vertices to query before actually querying.

Offline Least Common Ancestor Problem

Given a rooted tree $T = (V, E)$ and m pairs of vertices $x \neq y \in V$, efficiently compute the least common ancestor for each of the m pairs x and y.

[PAUSE FRAME]

Who?

R. E. Tarjan

Who?

R. E. Tarjan

H. N. Gabow

Initial Results

Theorem

We can solve the Offline Least Common Ancestor Problem in time $O((m+n)\alpha(m+n,n))$, where α is an inverse of the Ackermann function.

Initial Results

Theorem

We can solve the Offline Least Common Ancestor Problem in time $O((m+n)\alpha(m+n,n))$, where α is an inverse of the Ackermann function.

Here was Tarjan's original algorithm:

... problem, but it is [redacted] use the set union [redacted]y. Let T be a tree with root r and let $pairs = \{\{v_i, w_i\} \mid 1 \le i \le m\}$ be a set of m vertex pairs. We wish to compute LCA(v_i, w_i) for each pair. The following algorithm carries out the computation.

```
procedure LCA;
    begin
        for each {v, w} ∈ pairs do unmark {v, w} od;
        for each v ∈ V do create a set {v} named v od;
        SEARCH(r)
    end LCA;
```

Recursive procedure SEARCH is defined by

```
procedure SEARCH(v);
    begin
        for each w ∈ children(v) do SEARCH(w); UNION(v, w) od;
        for each {v, w} ∈ pairs do if {v, w} not marked then mark {v, w}
                                   else lca(v, w) := FIND(w) fi od
    end SEARCH;
```

Umm...

What on earth are UNION and FIND?

[FORCED DISCUSSION FRAME]

The Union-Find Data Structure

Union-Find is a data structure that provides operations on a family of disjoint sets:

The Union-Find Data Structure

Union-Find is a data structure that provides operations on a family of disjoint sets:

- $\textsc{Create}(x)$: adds $\{x\}$ to the data structure.

Union-Find is a data structure that provides operations on a family of disjoint sets:

- $\textsc{Create}(x)$: adds $\{x\}$ to the data structure.
- $\textsc{Union}(x, y)$: merges the sets in the data structure containing x and y.

The Union-Find Data Structure

Union-Find is a data structure that provides operations on a family of disjoint sets:

- $\textsc{Create}(x)$: adds $\{x\}$ to the data structure.
- $\textsc{Union}(x, y)$: merges the sets in the data structure containing x and y.
- $\textsc{Find}(x)$: returns a fixed representative element in the set in the data structure containing x.

Example

$\textsc{Create}(x)$:

$$\{\mathbf{x}\}$$

Example

$\textsc{Create}(x)$:

$$\{\mathbf{x}\}$$

$\textsc{Create}(y)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\}$$

Example

$\textsc{Create}(x)$:

$$\{\mathbf{x}\}$$

$\textsc{Create}(y)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\}$$

$\textsc{Create}(z)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\} \quad \{\mathbf{z}\}$$

Example

$\textsc{Create}(x)$:

$$\{\mathbf{x}\}$$

$\textsc{Create}(y)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\}$$

$\textsc{Create}(z)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\} \quad \{\mathbf{z}\}$$

$\textsc{Union}(x, y)$:

$$\{x, \mathbf{y}\} \quad \{\mathbf{z}\}$$

Example

$\textsc{Create}(x)$:

$$\{\mathbf{x}\}$$

$\textsc{Create}(y)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\}$$

$\textsc{Create}(z)$:

$$\{\mathbf{x}\} \quad \{\mathbf{y}\} \quad \{\mathbf{z}\}$$

$\textsc{Union}(x, y)$:

$$\{x, \mathbf{y}\} \quad \{\mathbf{z}\}$$

$\textsc{Union}(x, z)$:

$$\{\mathbf{x}, y, z\}$$

[PAUSE FRAME]

Tarjan's Union-Find I

Tarjan represents each set as a tree; the root serves as the representative element.

Tarjan's Union-Find I

Tarjan represents each set as a tree; the root serves as the representative element.

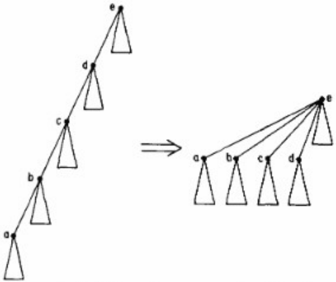


FIG 1. A *FIND* on element a, with collapsing. Triangles denote subtrees Collapsing converts tree T into tree T'

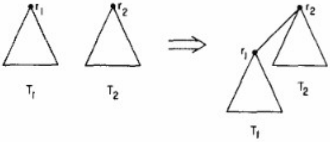


FIG 2 Union of two trees. Root r_1 of T_1 has a descendants; root r_2 of T_2 has b descendants. Root of new tree has $a + b$ descendants.

More formally, those pictures become the following:

Tarjan's Union-Find II

More formally, those pictures become the following:

Find(x)

if x.parent $\neq$ x
    x.parent $\leftarrow$ Find(x.parent)
return x.parent

Tarjan's Union-Find II

More formally, those pictures become the following:

Find(x)

if x.parent $\neq$ x
    x.parent $\leftarrow$ Find(x.parent)
return x.parent

Union(x, y)

if x $=$ y
    return
if x.rank $<$ y.rank
    x.parent $\leftarrow$ y
else if x.rank $>$ y.rank
    y.parent $\leftarrow$ x
else
    y.parent $\leftarrow$ x
    x.rank $\leftarrow$ x.rank $+$ 1

Tarjan's Union-Find II

More formally, those pictures become the following:

Find(x)

if x.parent $\neq$ x
    x.parent $\leftarrow$ Find(x.parent)
return x.parent

Union(x, y)

x $\leftarrow$ Find(x)
y $\leftarrow$ Find(y)
if x $=$ y
    return
if x.rank $<$ y.rank
    x.parent $\leftarrow$ y
else if x.rank $>$ y.rank
    y.parent $\leftarrow$ x
else
    y.parent $\leftarrow$ x
    x.rank $\leftarrow$ x.rank $+ 1$

Tarjan's Union-Find III

This data structure performs well:

Tarjan's Union-Find III

This data structure performs well:

Theorem

Given a family of disjoint sets partitioning n elements, a sequence of $m \geq n$ $\textsc{Find}$s and $n - 1$ $\textsc{Union}$s take time $\Theta(m\alpha(m, n))$, where α is an inverse of the Ackermann function.

Tarjan's Union-Find III

This data structure performs well:

Theorem

Given a family of disjoint sets partitioning n elements, a sequence of $m \geq n$ $\textsc{Find}$s and $n-1$ $\textsc{Union}$s take time $\Theta(m\alpha(m, n))$, where α is an inverse of the Ackermann function.

Proof.

Eight dense pages of black magic. $\square$

[PAUSE FRAME]

Here was Tarjan's original algorithm:

Getting Back On-Track

Here was Tarjan's original algorithm:

... problem, but it is [redacted]. Let T be a tree with root r and let $pairs = \{\{v_i, w_i\} \mid 1 \le i \le m\}$ be a set of m vertex pairs. We wish to compute LCA(v_i, w_i) for each pair. The following algorithm carries out the computation.

```
procedure LCA;
    begin
        for each {v, w} ∈ pairs do unmark {v, w} od;
        for each v ∈ V do create a set {v} named v od;
        SEARCH(r)
    end LCA;
```

Recursive procedure SEARCH is defined by

```
procedure SEARCH(v);
    begin
        for each w ∈ children(v) do SEARCH(w); UNION(v, w) od;
        for each {v, w} ∈ pairs do if {v, w} not marked then mark {v, w}
                                   else lca(v, w) := FIND(w) fi od
    end SEARCH;
```

Getting Back On-Track

Here was Tarjan's original algorithm:

problem, but it is [redacted] use the set union [redacted]y. Let T be a tree with root r and let $pairs = \{\{v_i, w_i\} \mid 1 \leq i \leq m\}$ be a set of m vertex pairs. We wish to compute LCA(v_i, w_i) for each pair. The following algorithm carries out the computation.

procedure LCA;
  begin
    for each $\{v, w\} \in pairs$ **do** unmark $\{v, w\}$ **od**;
    for each $v \in V$ **do** create a set $\{v\}$ named v **od**;
    SEARCH(r)
  end LCA;

Recursive procedure SEARCH is defined by

procedure SEARCH(v);
  begin
    for each $w \in children(v)$ **do** SEARCH(w); UNION(v, w) **od**;
    for each $\{v, w\} \in pairs$ **do if** $\{v, w\}$ not marked **then** mark $\{v, w\}$
                else $lca(v, w) :=$ FIND(w) **fi od**
  end SEARCH;

Let's parse this with an example.

Example

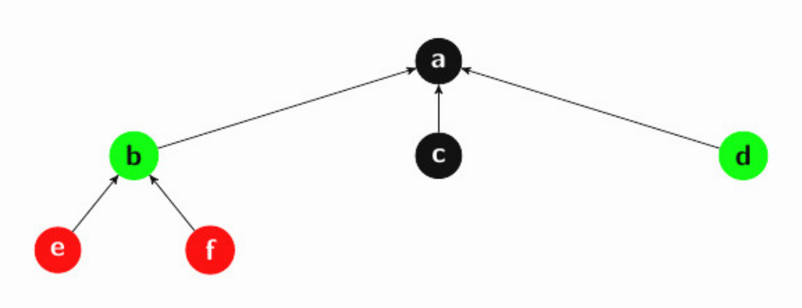
Pairs:

- b, d
- f, e

Example

Pairs:

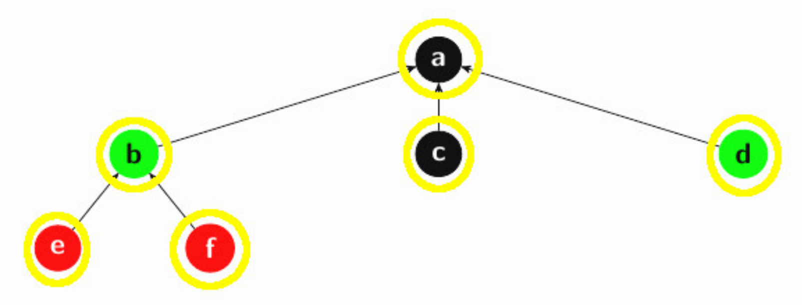
- b, d
- f, e



Example

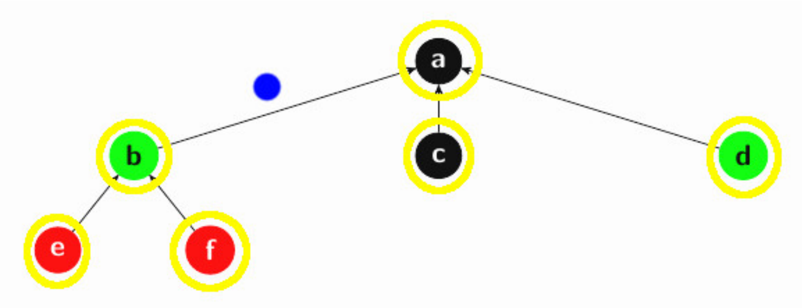
Pairs:

- b, d
- f, e



Pairs:

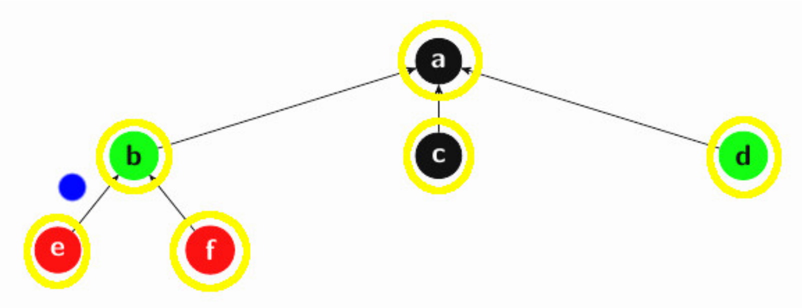
- b, d
- f, e



Example

Pairs:

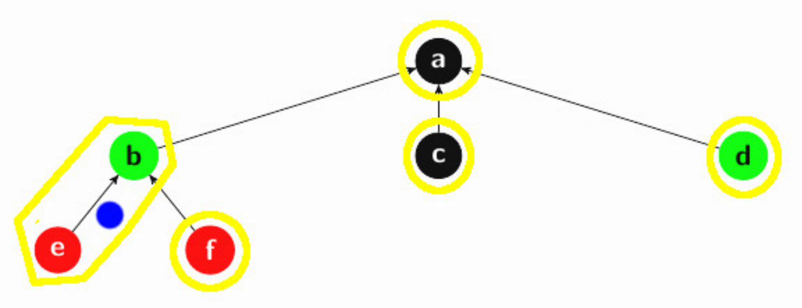
- b, d
- f, e



Example

Pairs:

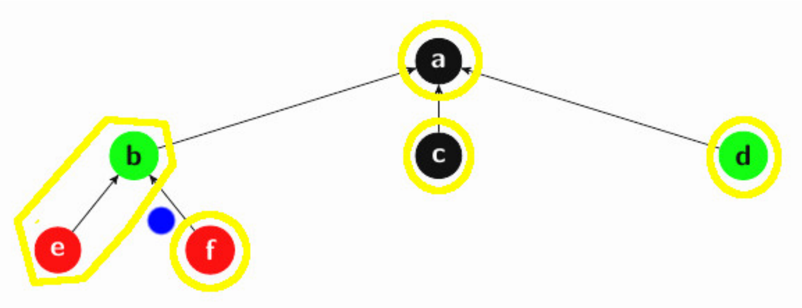
- b, d
- f, e



Example

Pairs:

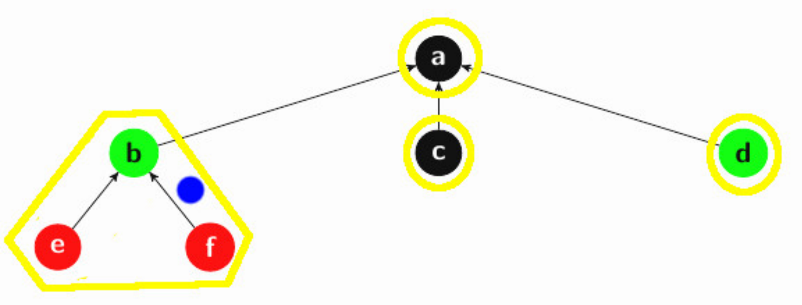
- b, d
- f, e



Example

Pairs:

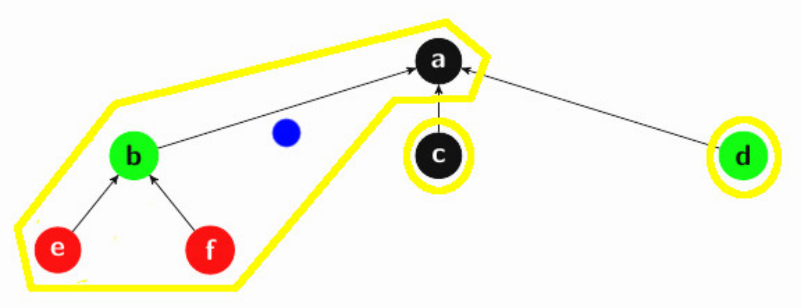
- b, d
- f, e



Example

Pairs:

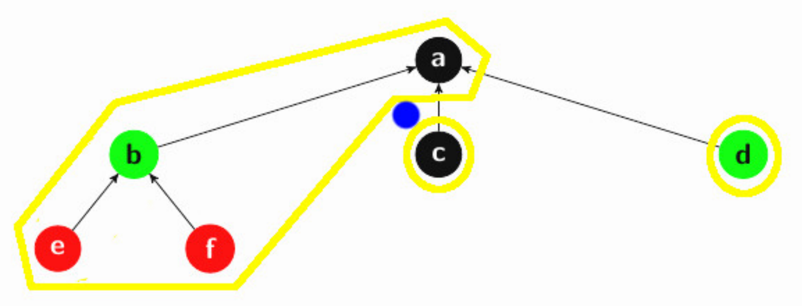
- b, d
- f, e



Example

Pairs:

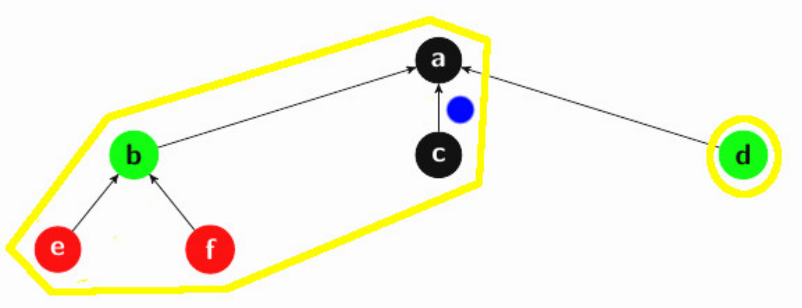
- b, d
- f, e



Example

Pairs:

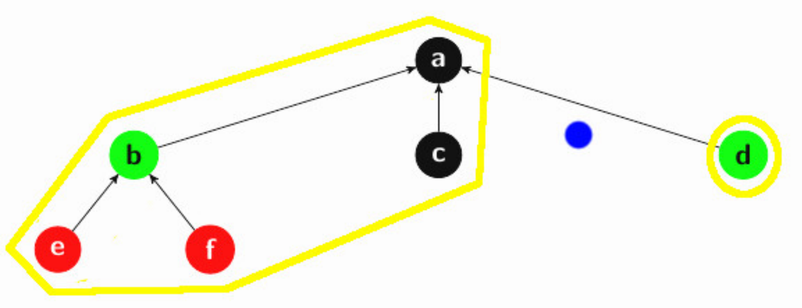
- b, d
- f, e



Example

Pairs:

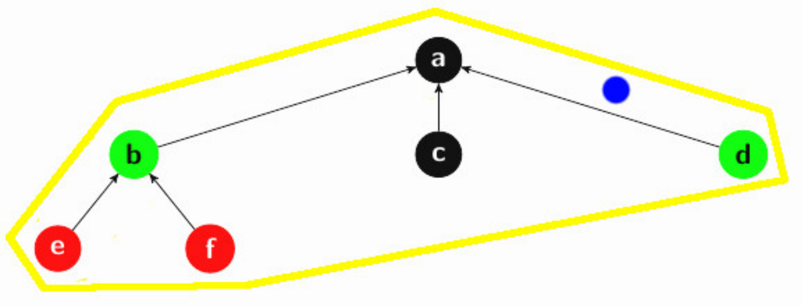
- b, d
- f, e



Example

Pairs:

- b, d
- f, e



[PAUSE FRAME]

Modernising the Code

TarjanOLCA(u)

```
u.ancestor ← u
for v ∈ u.children do
    TarjanOLCA(v)
    Union(u,v)
    Find(u).ancestor ← u
u.colour ← black
for v such that {u,v} ∈ pairs do
    if v.colour = black
        Least Common Ancestor of u and v is Find(v).ancestor
```

[FORCED DISCUSSION FRAME]

Analyzing the Code

Now that we finally understand the code, let's analyze its runtime:

Analyzing the Code

Now that we finally understand the code, let's analyze its runtime:

- The first loop implies each edge causes one Union and Find.

Analyzing the Code

Now that we finally understand the code, let's analyze its runtime:

- The first loop implies each edge causes one UNION and FIND.
- The second loop implies each pair causes one FIND.

Analyzing the Code

Now that we finally understand the code, let's analyze its runtime:

- The first loop implies each edge causes one UNION and FIND.
- The second loop implies each pair causes one FIND.
- Thus this algorithm invokes $m + n - 1$ FINDs and $n - 1$ UNIONs.

Now that we finally understand the code, let's analyze its runtime:

- The first loop implies each edge causes one UNION and FIND.
- The second loop implies each pair causes one FIND.
- Thus this algorithm invokes $m + n - 1$ FINDs and $n - 1$ UNIONs.

Theorem

We can solve the Offline Least Common Ancestor Problem in time $O((m + n)\alpha(m + n, n))$, where α is an inverse of the Ackermann function.

[END FRAME]