Ruminations on Tarjan's Union-Find Algorithm and Connected Operators

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

ISMM 2005, Paris

Outline

Outline

Outline

Outline

Context of our work

- we develop a *generic* library for image processing
  `http://olena.lrde.epita.fr`

  Darbon, Géraud and Duret-Lutz. Generic Implementation of Morphological Image Operators; When Harry's C++ Code Met Sally's Algorithms. In proceedings of ISMM 2002.

- we were implementing algebraic openings and closings with Tarjan's Union-Find Algorithm (TUFA)

- we realized that TUFA is much more general than it appeared

Tarjan's Union-Find Algorithm (TUFA)

This algorithm:

- was designed to extract connected components on graphs
- is quasi-linear w.r.t. the number of nodes —or image points
- is extremely simple :)

- can be relevant for many mathematical morphology operators
- but... is still under-used

Some references about TUFA

Dillencourt et al.	connected components labeling
Meijster and Wilkinson	connected set opening and closing
Wilkinson and Roerdink	attribute operations
Roerdink and Meijster	watershed
Hesselink	min-tree
Najman and Couprie	component tree

About this talk

We present:

- a comprehensive and general form of Tarjan's Union-Find Algorithm (TUFA)

  $\longrightarrow$ a "ready-to-use" canvas of this algorithm

- the bridge existing between TUFA and connected operators
- how to accommodate TUFA with separated domains (filters with a domain-disjointess preservation property)

Outline

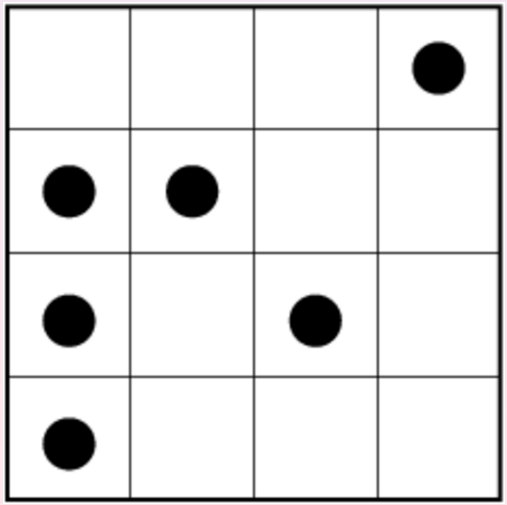
A connected component as a tree

- TUFA computes a forest of trees

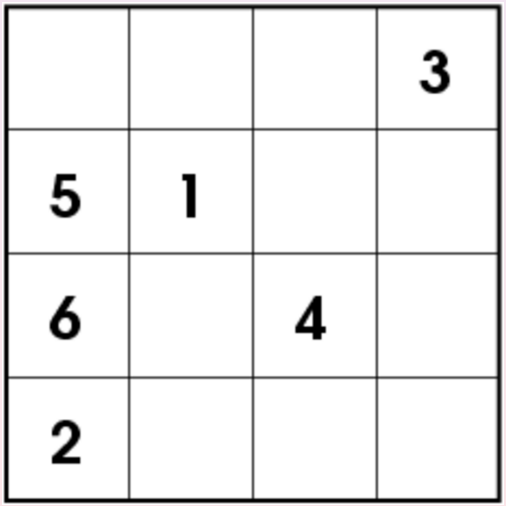
  $\longrightarrow$ it is *not* a queue-based algorithm
- a connected component $\Leftrightarrow$ a tree
- a point of a connected component $\Leftrightarrow$ a node of the tree
- a "root point" is:
  - the point that represents its connected component
  - the root node of the tree
- each node has exactly one parent
  - a root point is its own parent

Nodes (points) are ordered

D	the image domain
I	a binary set of points ($I \subseteq D$)
$\mathcal{R}_I$	a total ordering between the points of I



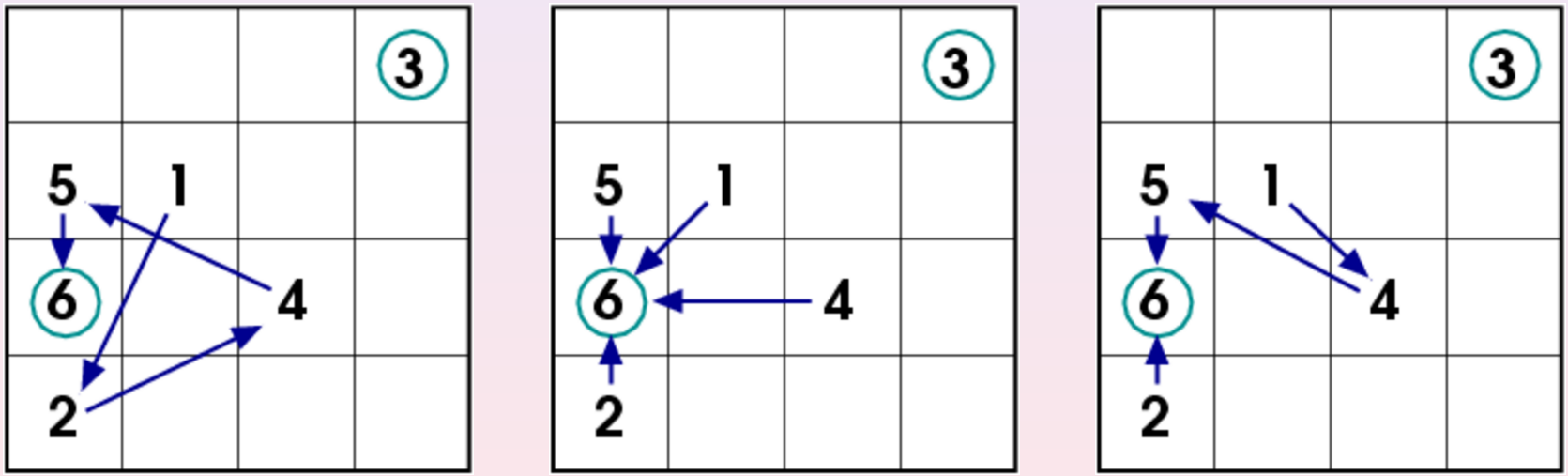
I



$\mathcal{R}_I$

Parenthood constraint

Constraint upon the forest: $\forall p \in I,\ p\ \mathcal{R}_I\ parent(p)$



three correct forests describing I

Three parts

TUFA is composed of:

- an initialization,
- a first pass,
- and a second pass.

An initialization...

```
initialization()
{
  for_all_points (p)
    is_processed[p] = false; // no point is processed yet

  current_label = 0;

  // ...extra computation...
}
```

An initialization...

```
initialization()
{
  for_all_points (p)
    is_processed[p] = false;

  current_label = 0;

  init_output(O); // default value of output points is background
}
```

Two passes

first_pass() *// browsing points of I follows* $\mathcal{R}_I$

```
{
  for_all_points (p) if ( is_in_I(p) ) {
    // body 1...
    is_processed[p] = true;
  }
}
```

second_pass() *// browsing points of I follows* $\mathcal{R}_I^{-1}$

```
{
  for_all_points (p) if ( is_in_I(p) ) {
    // body 2...
  }
}
```

First pass internals

```
// body 1:
{
   make_set(p);
   for_all_neighbors(n, p) if ( is_in_I(n) and is_processed[n] )
      do_union(n, p);
   // else ...extra computation...
}

do_union(point n, point p)
{
   point r = find_root(n);
   if ( r != p )
      set_parent(r, p);
}
```

First pass helpers

```
make_set(point p) {
  parent[p] = p;
  // ...extra computation...
}

set_parent(point r, point p) {
  parent[r] = p;
  // ...extra computation...
}

point find_root(point x) {
  if (parent[x] == x) return x;
  else return find_root(parent[x]);
}
```

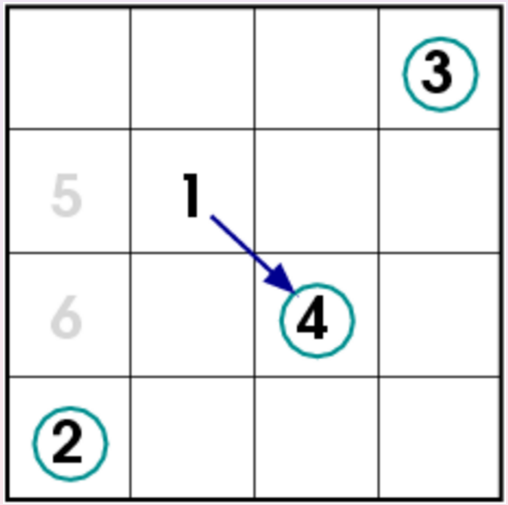
Second pass internals

```
// body 2:
{
   if ( parent[p] == p )
      O[p] = ++current_label; // new label
   else
      O[p] = O[parent[p]]; // label propagation
}
```

Outline

During the 1st pass

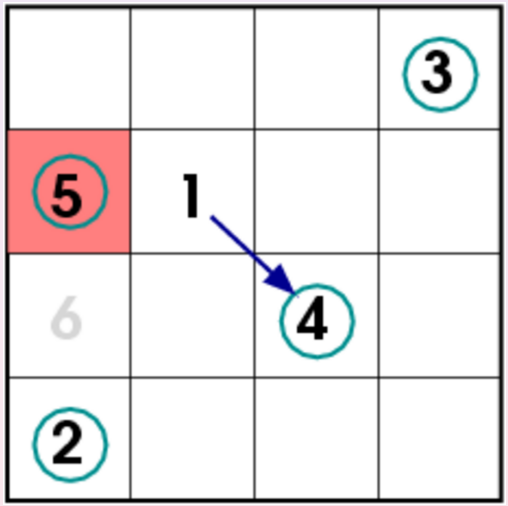
Four points have been processed.



Three trees have been constructed.

During the 1st pass

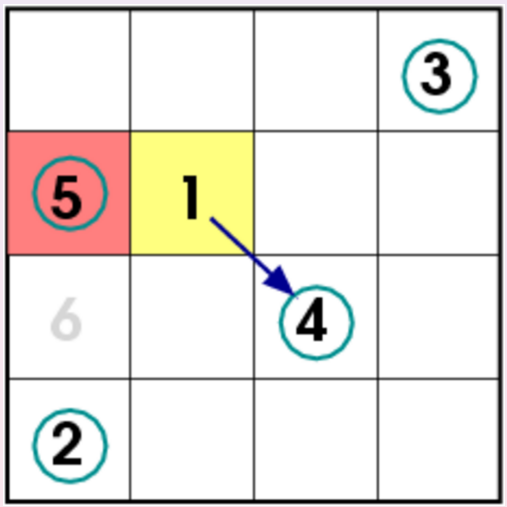
Current *p* is number **5**.



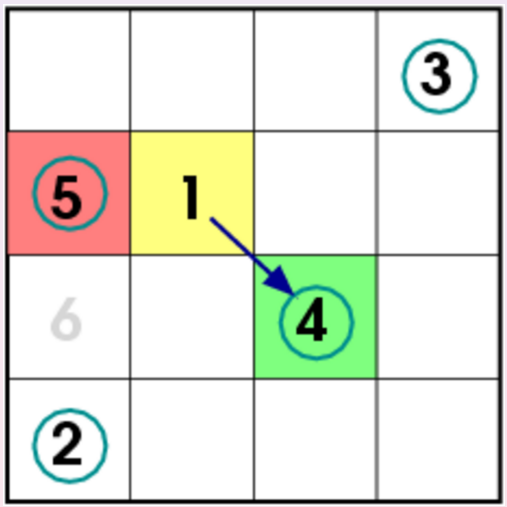
make_set(p)

During the 1st pass

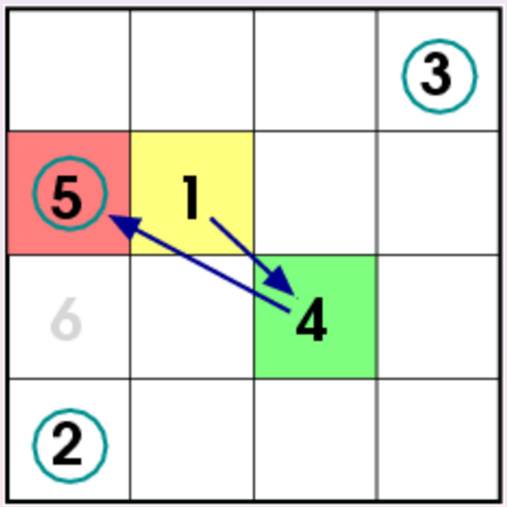
Processing the only already processed neighbor of p in I.



n = point(1)



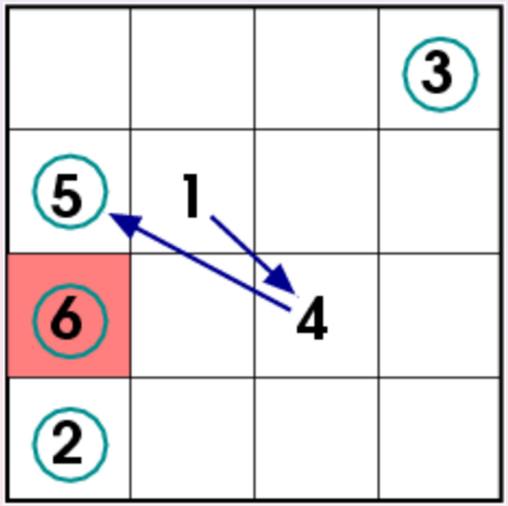
r = find_root(n)



set_parent(r, p)

During the 1st pass

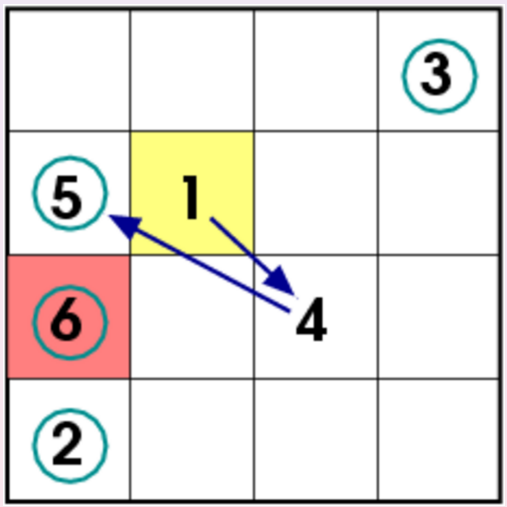
Current *p* is now number **6**.



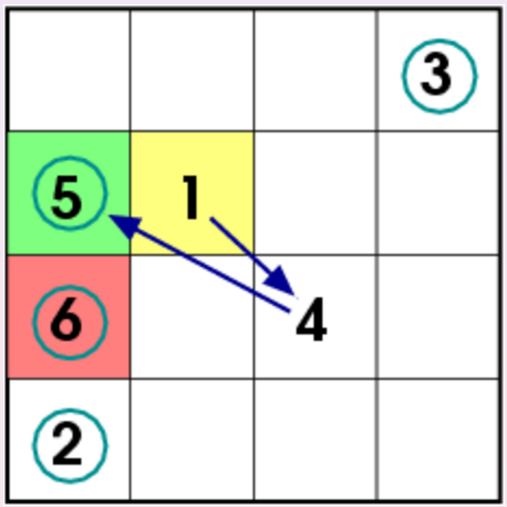
make_set(p)

During the 1st pass

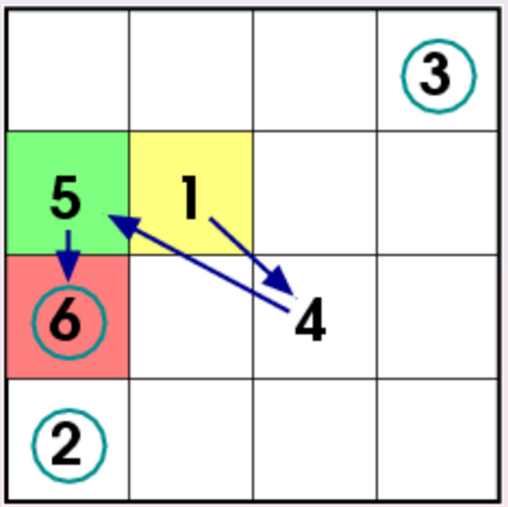
Processing a first neighbor of *p*.



n = point(1)



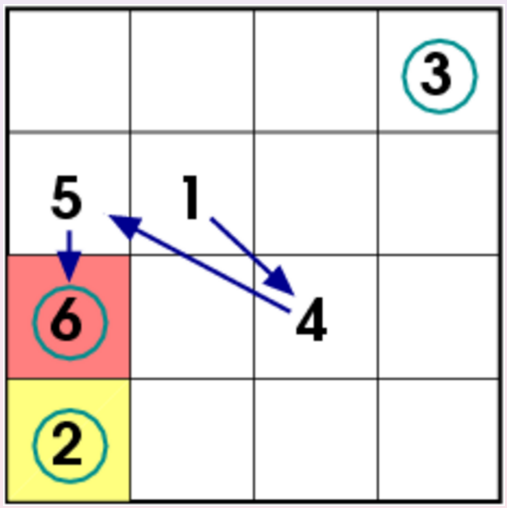
r = find_root(n)



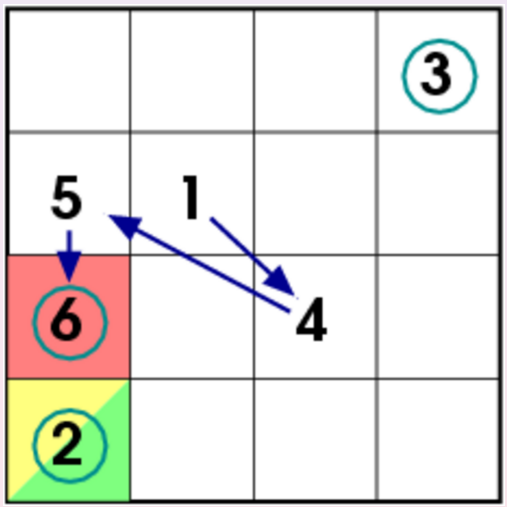
set_parent(r, p)

During the 1st pass

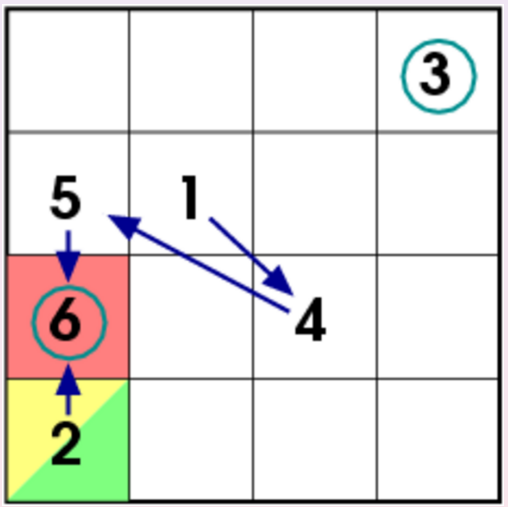
Processing a second neighbor of *p*.



n = point(2)



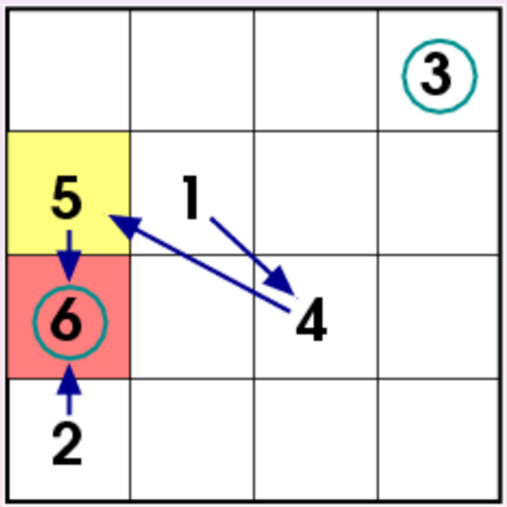
r = find_root(n)



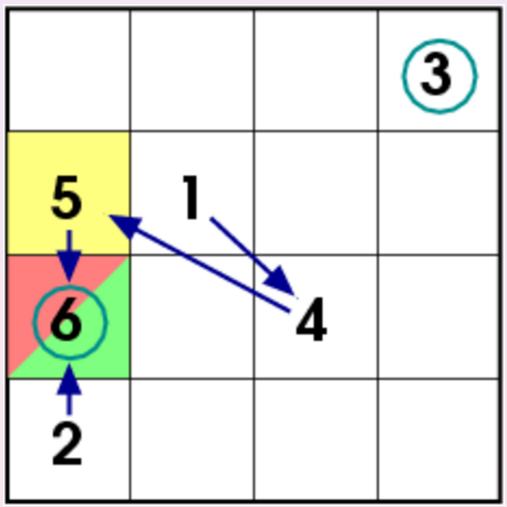
set_parent(r, p)

During the 1st pass

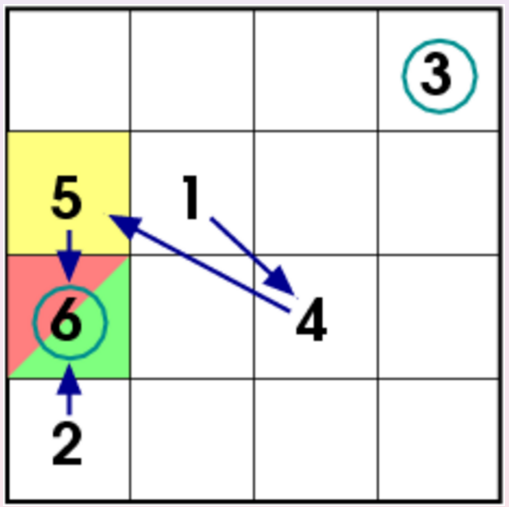
Processing the last neighbor of *p*.



n = point(5)



r = find_root(n)

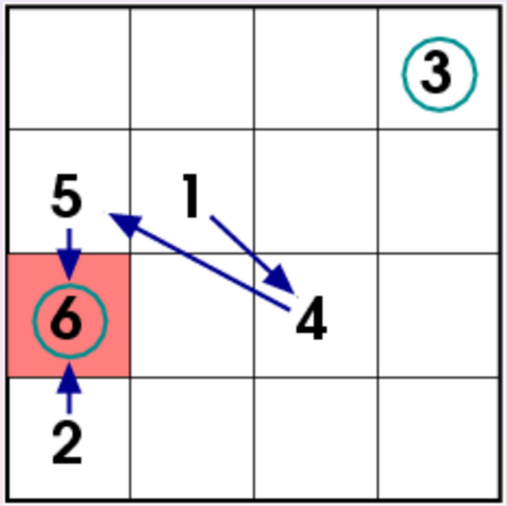


if (r != p)...

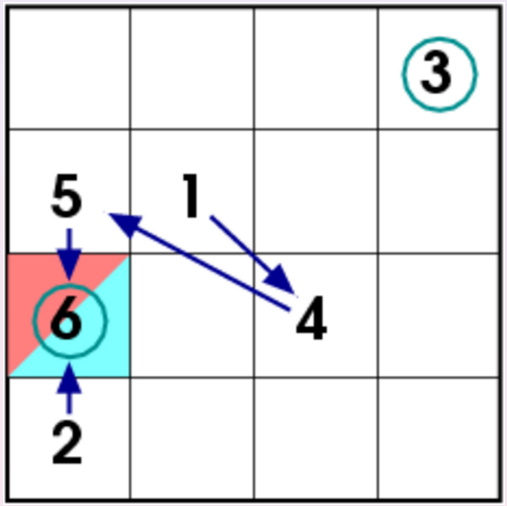
...

First pass is over; let's start the second one.

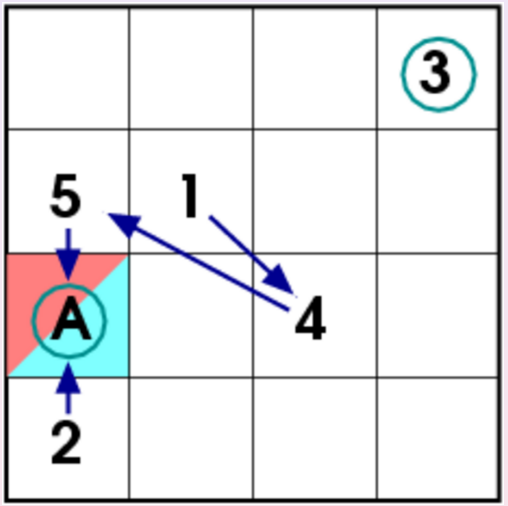
During the 2nd pass



p = point(6)

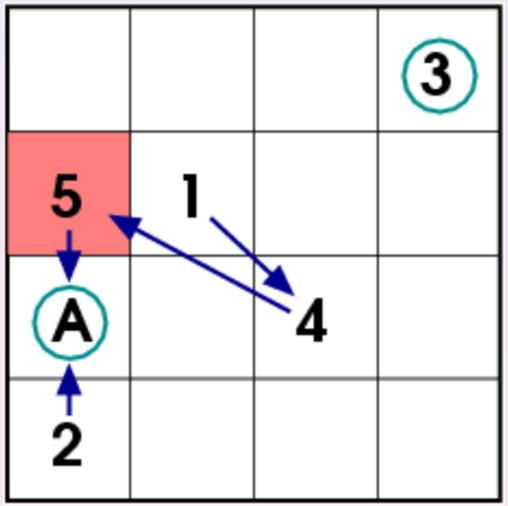


if (parent[p] == p)

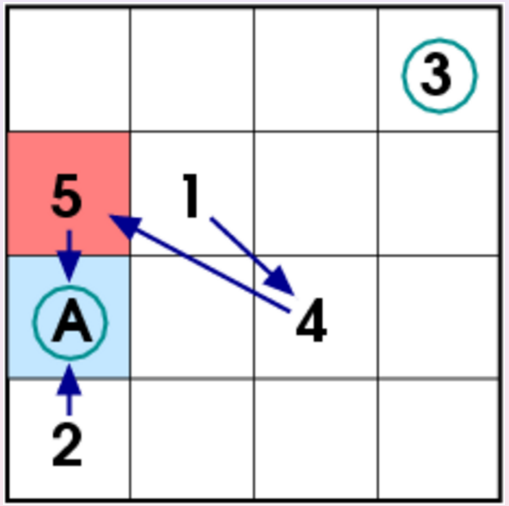


++current_label

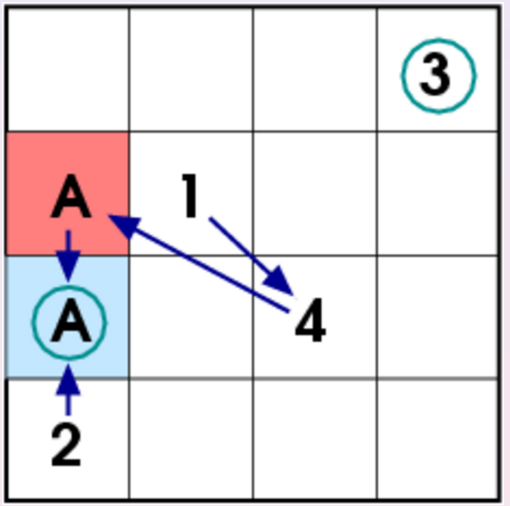
During the 2nd pass



p = point(5)

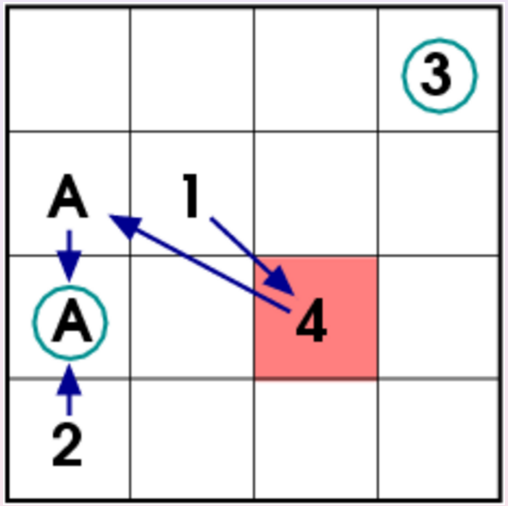


if (parent[p] == p)

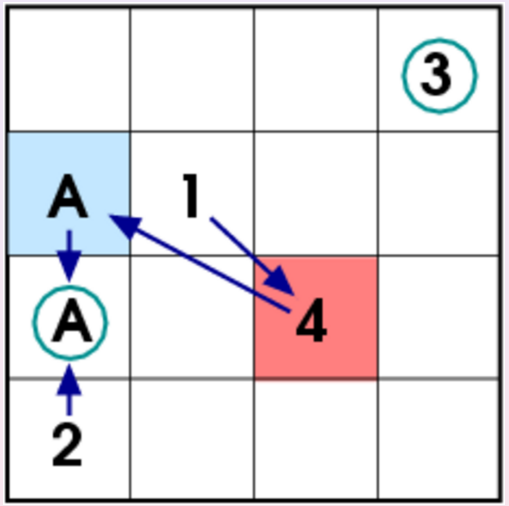


O[p] = O[parent[p]]

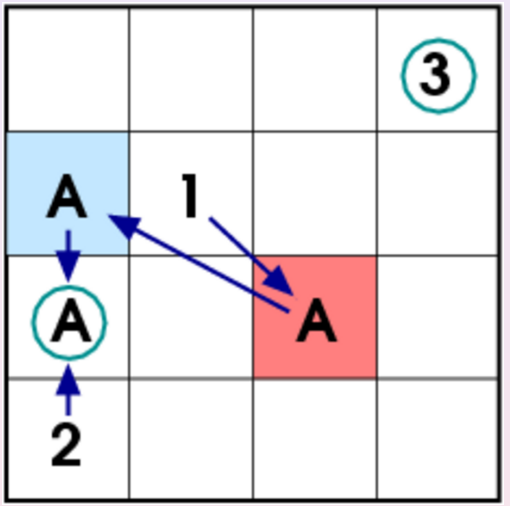
During the 2nd pass



p = point(4)

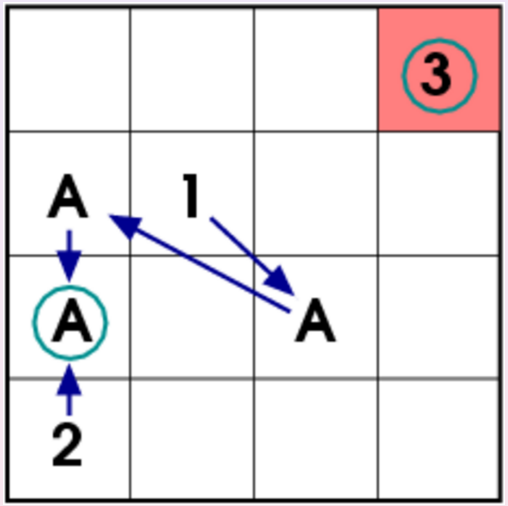


if (parent[p] == p)

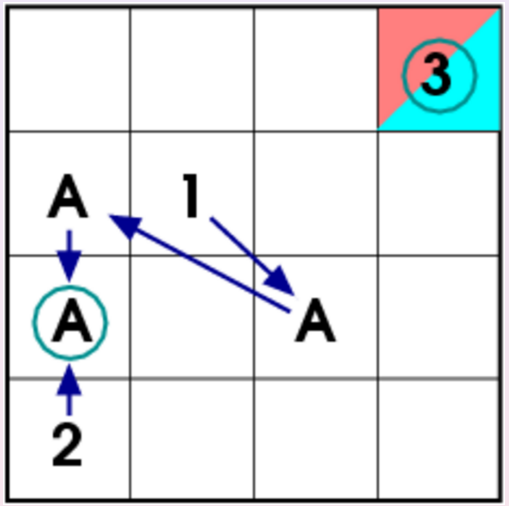


O[p] = O[parent[p]]

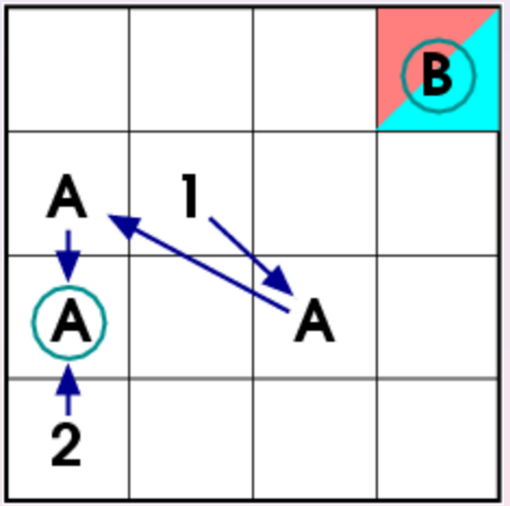
During the 2nd pass



p = point(3)



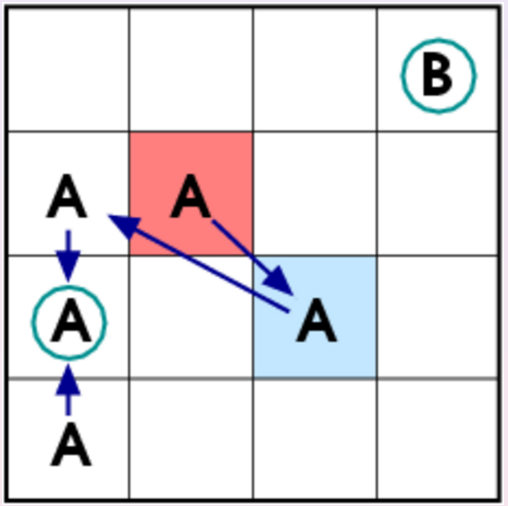
if (parent[p] == p)



++current_label

During the 2nd pass

and so one until:



the final labeling

Outline

TUFA in a nutshell

- TUFA targets a sub-domain of D (namely I).
- First pass follows $\mathcal{R}_I$ while second pass follows $\mathcal{R}_I^{-1}$.
- TUFA heavily relies upon the constraint: $p\ \mathcal{R}_I\ parent(p)$
- About root points:
  - during first pass, the current point p is a root point;
  - during second pass, the first point of a component which is reached is its root point.

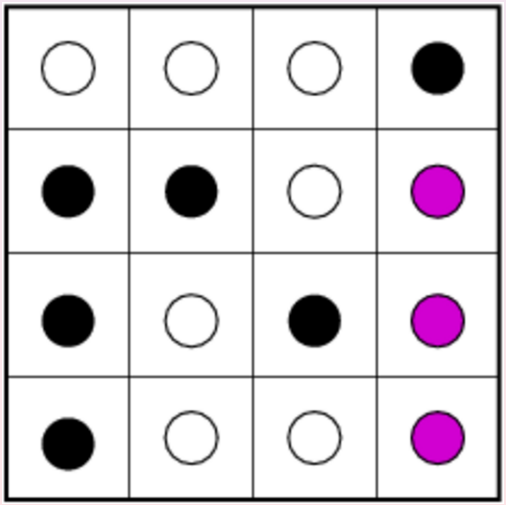
Outline

Sub-domains

Partition of D into $m + 1$ disjoint sub-domains:

- $D = D' \cup D''$
- D'' set of points not subject to forest computation
- $D' = \cup_{i=1}^{m} D_i$

  where D_i has **its own way** to compute its forest.



D_1 (black); D_2 (magenta); D'' (white).

Generalization of 1st pass

first_pass() *// browsing points of* D_i *follows* $\mathcal{R}_{D_i}$

```
{
  for_all_points (p)
  {
    if ( is_in_D1(p) )
      // body 1.1
    if ( is_in_D2(p) )
      // body 1.2
    // other domains...
    is_processed[p] = true;
  }
}
```

let us re-write this...

Re-writing of 1st pass

```
first_pass() // browsing points of D_i follows R_{D_i}
{
  for_all_points (p) if ( is_in_D1(p) ) {
    // body 1.1
    is_processed[p] = true;
  }
  for_all_points (p) if ( is_in_D2(p) ) {
    // body 1.2
    is_processed[p] = true;
  }
  // other domains...
}
```

Now sub-domains are processed one after another.

Re-writing of 2nd pass

second_pass() *// browsing points of* D_i *follows* $\mathcal{R}_{D_i}^{-1}$

```
{
  // other domains...
  for_all_points (p) if ( is_in_D2(p) )
    // body 2.2
  for_all_points (p) if ( is_in_D1(p) )
    // body 2.1
}
```

Just prefer this solution (process sub-domains in the reverse way of 1st pass).

Disjointness-preservation property

Each connected component (or flat zone) Γ created by such an operator cannot cross domains frontiers:

- $\forall \Gamma \in O,\ \exists i$ such as $\Gamma \subseteq D_i$ and $\forall j \neq i,\ \Gamma \cap D_j = \emptyset$.
- but we actually never cross frontiers of D_i:

```
for_all_points (p) if ( is_in_Di(p) ) {
  for_all_neighbors(n, p) if ( is_in_Di(n) and is_processed[n] )
    // do the job...
  is_processed[p] = true;
}
```

- so we process sub-domains in an independent way
- thus disjointness-preservation property is implicitly ensured.

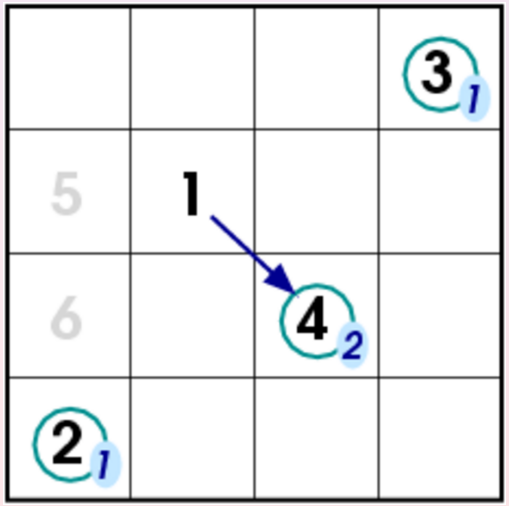
Outline

During the 1st pass

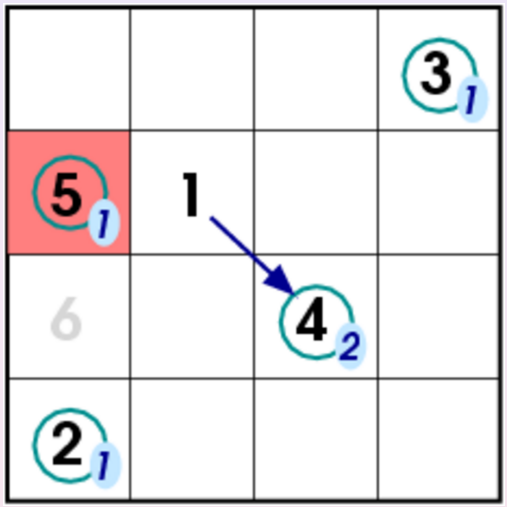
Each root point can caries data associated with its component.

For instance the number of points of its component.
(just like in the area opening)

Consider this current state of a 1st pass:

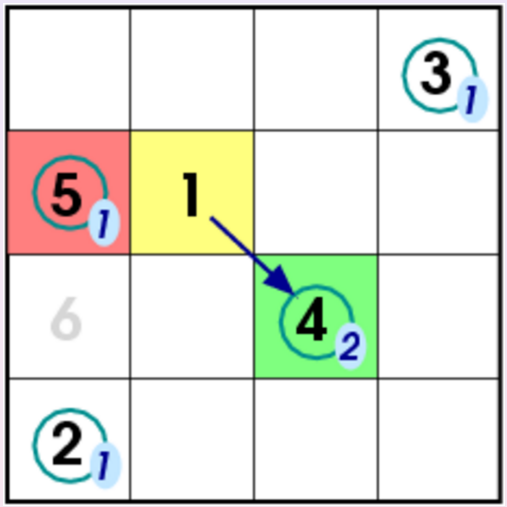


During the 1st pass



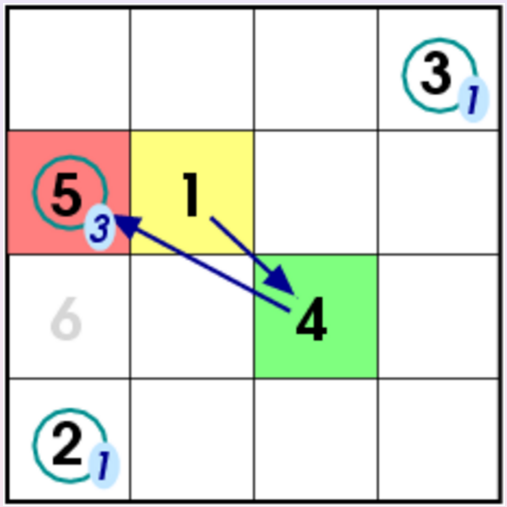
make_set(p)

also performs init_data(p)



r = find_root(n)

(as usual)



set_parent(r, p)

also performs merge_data(r, p)

Modifying the algorithm canvas

```
make_set(point p) {
  parent[p] = p;
  init_data(p);
}

set_parent(point r, point p) {
  parent[r] = p;
  merge_data(r, p);
}
```

Outline

During the 1st pass

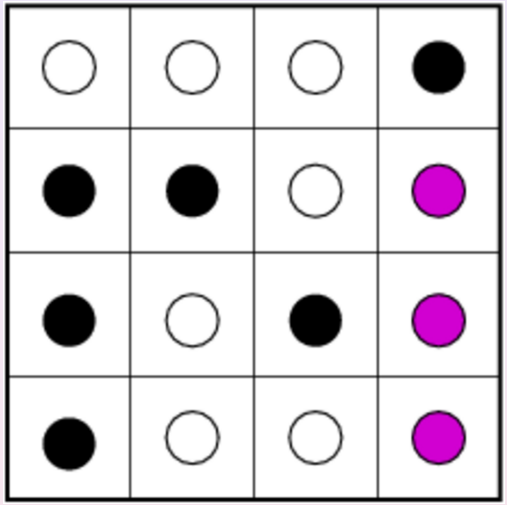
When processing D_i we access neighbors n of p.

we only `do_union` with neighbors $n \in D_i$ but...

we can also read values of $n \notin D_i$

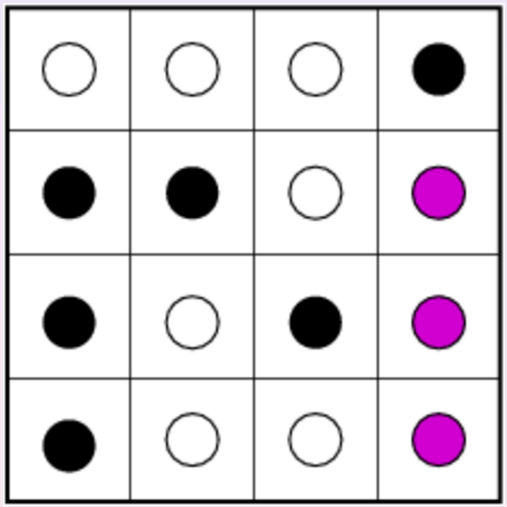
```
// body 1.i:
{
  make_set(p);
  for_all_neighbors(n, p) if ( is_in_Di(n) and is_processed[n] )
    do_union(n, p);
  else visit_external_boundary_of_Di(n, p);
}
```

During the 1st pass

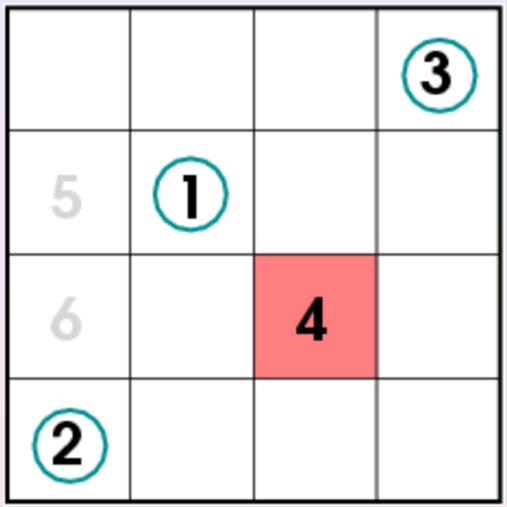


domains

During the 1st pass



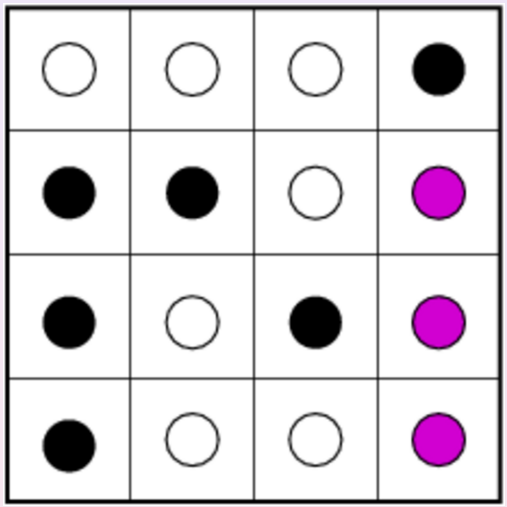
domains



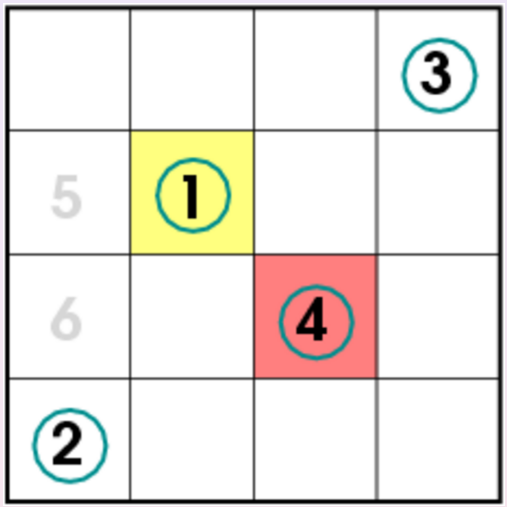
$p \in D_i$

we are going to process neighbors of p...

During the 1st pass



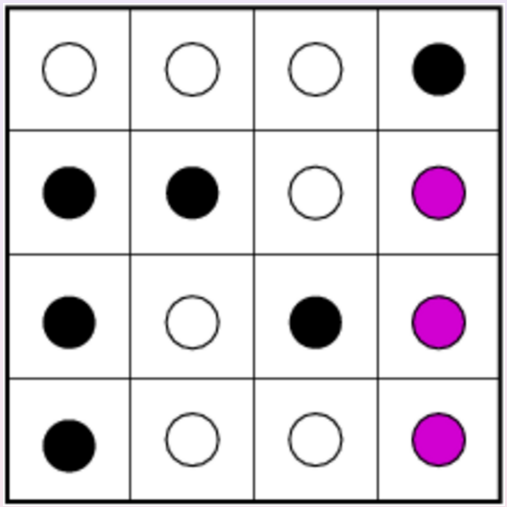
domains



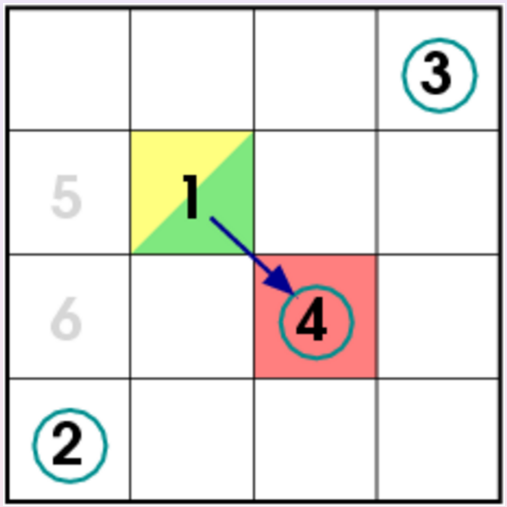
$n \in D_i$

n is in the same sub-domain as p...

During the 1st pass



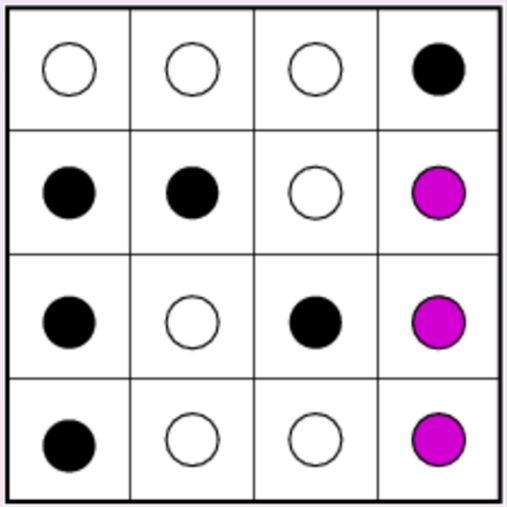
domains



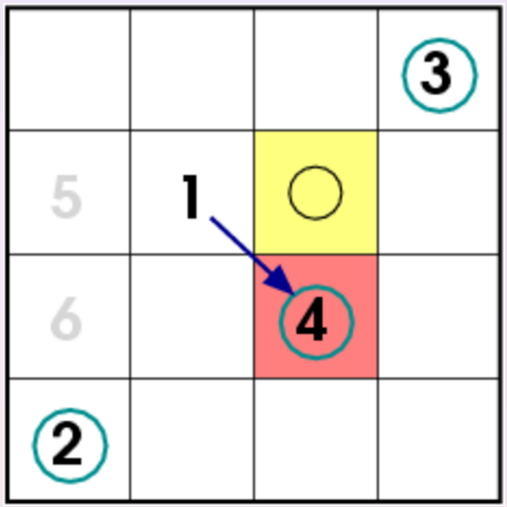
$n \in D_i$

...so we proceed as usual. Done! Now next neighbor...

During the 1st pass



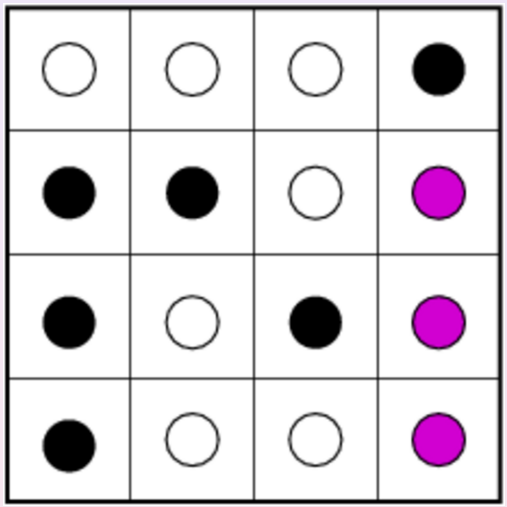
domains



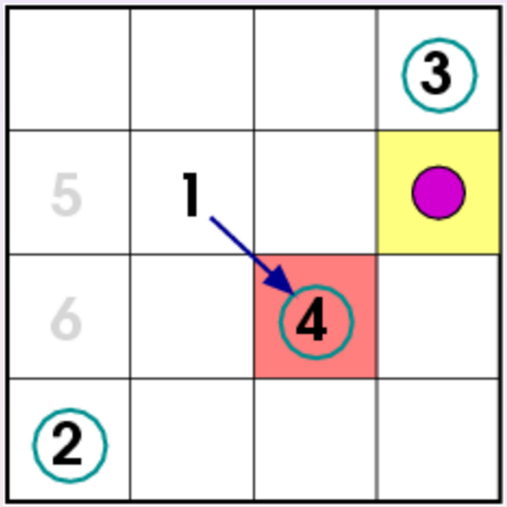
$n \in D''$

n is not in the same sub-domain as p... so ignore it or not!

During the 1st pass



domains

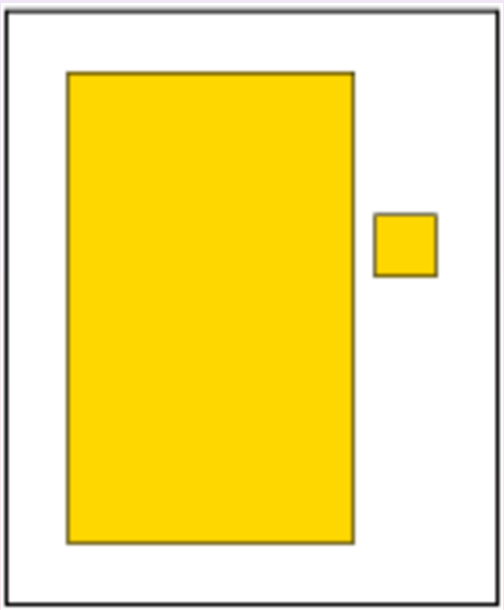


$n \in D' - D_i$

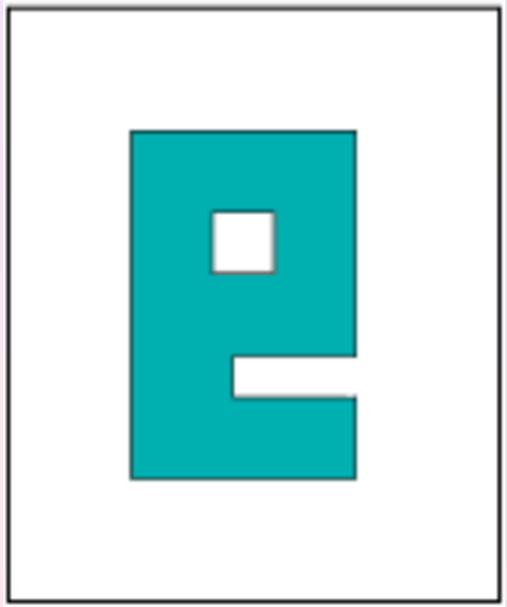
again, another domain than the one of p... so ignore it or not!

Simple example

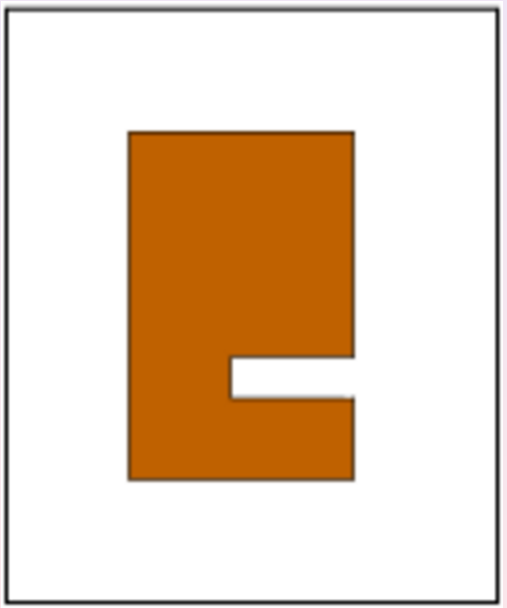
Consider this reconstruction by erosion: the result is $G \cup H$ where H is its hole (a component totally surrounded by G).



marker F



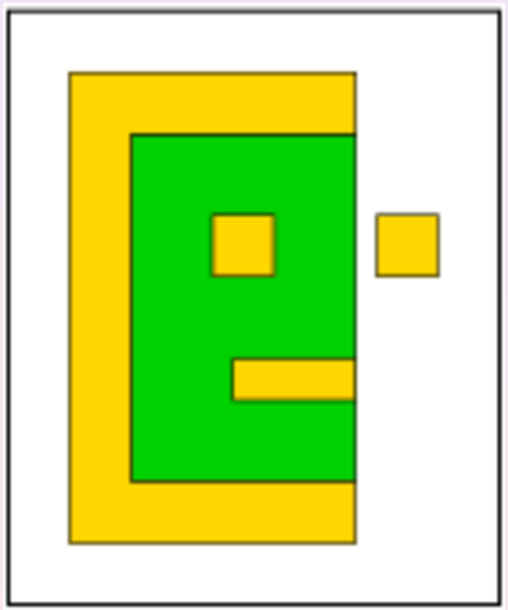
mask $G \subseteq F$



result $R^{\epsilon}_{G}(F)$

Simple example

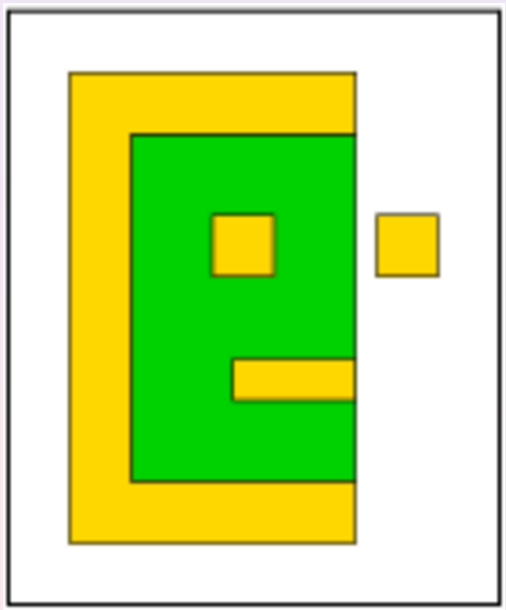
$$D = F^C \text{ (white)} \cup G \text{ (green)} \cup F \cap G^C \text{ (yellow)}$$



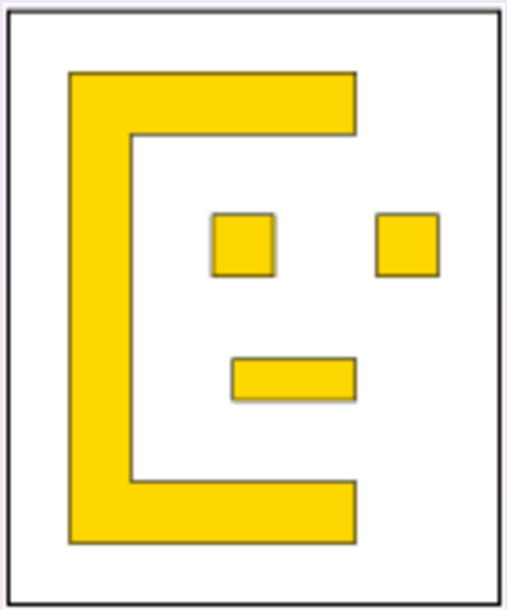
sub-domains

Simple example

the set subject to forest computation is $D' = F \cap G^C$



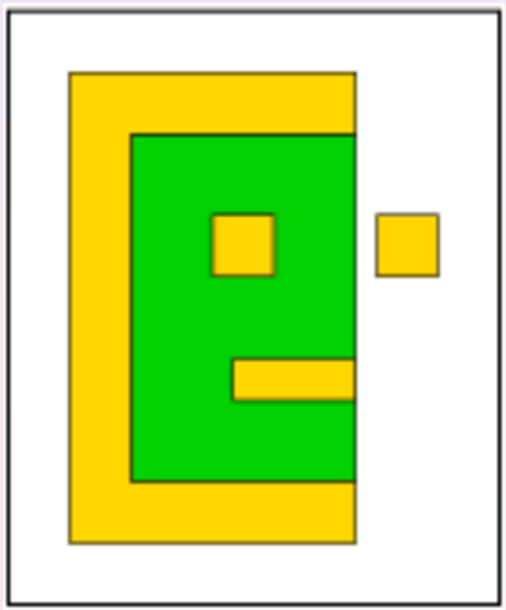
sub-domains



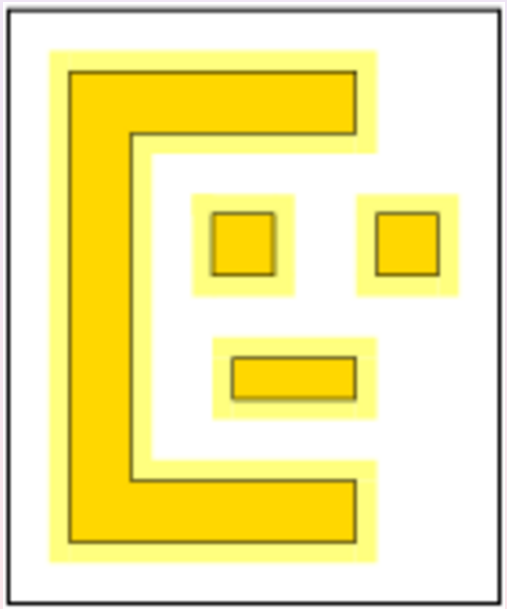
D'

Simple example

external boundary of D' is visited (pale yellow)



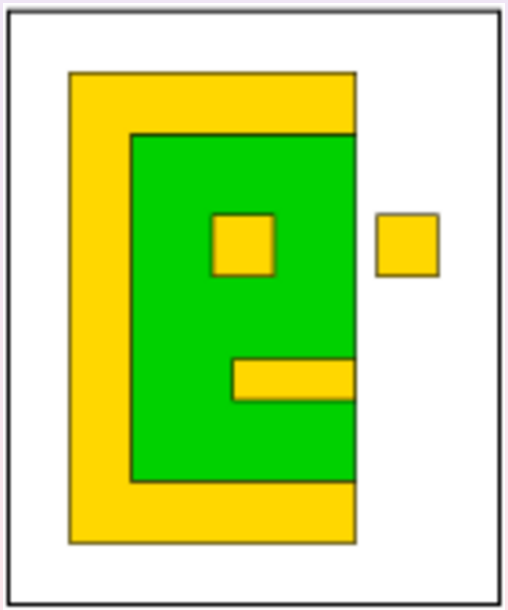
sub-domains



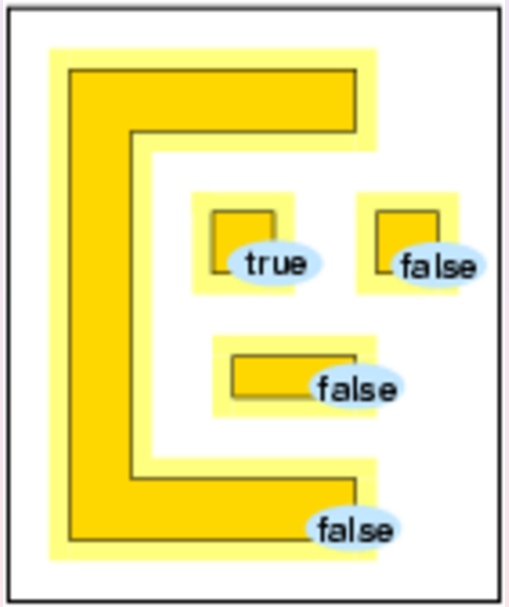
D' with $\mathcal{B}^{ext}$

Simple example

a Boolean value is attached to components



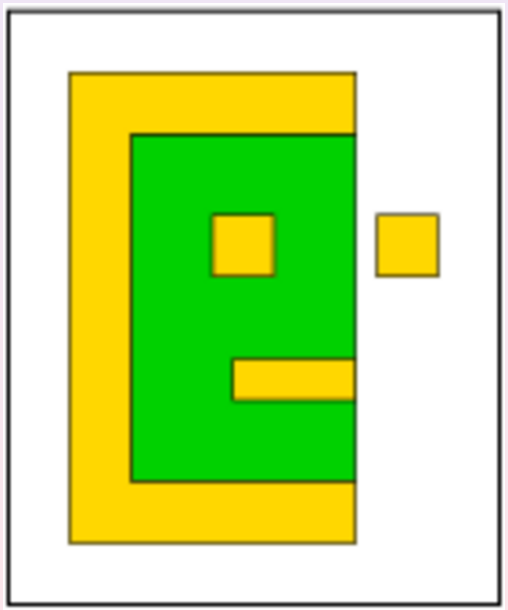
sub-domains



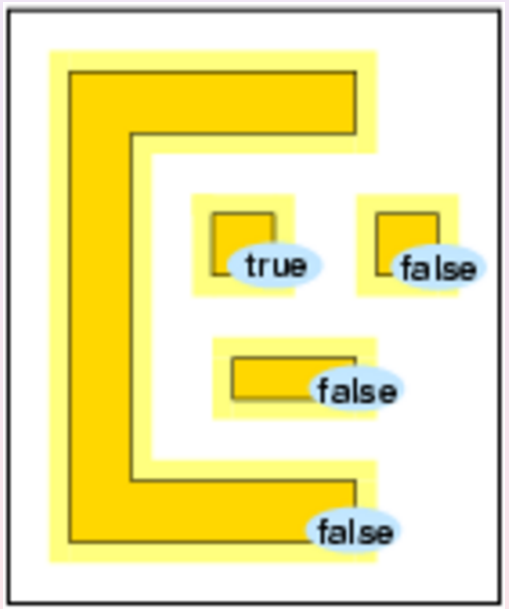
D' with data

Simple example

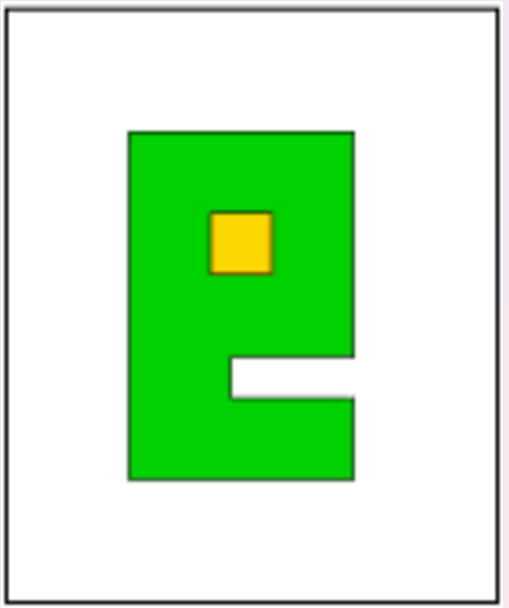
init. of TUFA: $O = G$ **and** 2nd pass of TUFA: marking “true”.



sub-domains



D' with data



$G \cup \left\{ \Gamma_{D'}^{true} \right\}$

Outline

It is fun...

original image

to easily write an efficient operator whose result is...

rectangle opening

Ordering relationships

Given:

- a function f
- $\mathcal{R}_D$ an ordering defined over D
- the ordering of the complete lattice framework

we can derive:

- $\mathcal{R}^{\uparrow}$ such as:
  $p\,\mathcal{R}^{\uparrow}p' \;\Leftrightarrow\; f(p) < f(p')$ or $(f(p) = f(p')$ and $p\,\mathcal{R}_D\,p')$
- and its reverse ordering $\mathcal{R}^{\downarrow}$.

Filtering

Our new goal is to connect flat zones

- so TUFA has to compute flat zones
- and a criterion has to tell when to merge flat zones
  - stopping to merge means stopping the filtering
  - `do_union` is just slightly modified

With $\mathcal{R}^{\uparrow}$

- the first pass of TUFA limics a flooding of *f*
- we get an extensive behavior of the connected operator.

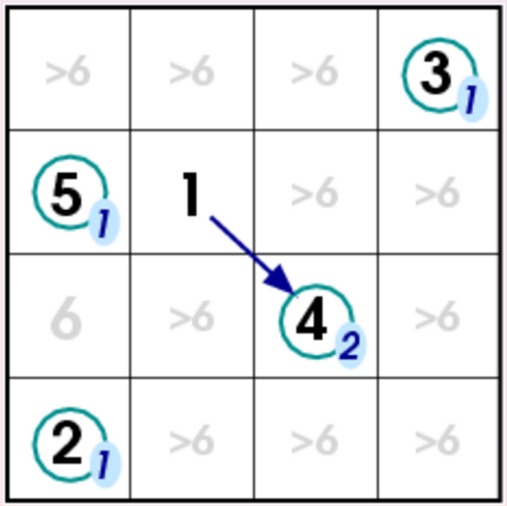
With $\mathcal{R}^{\downarrow}$ (by duality)

- we get an anti-extensive behavior.

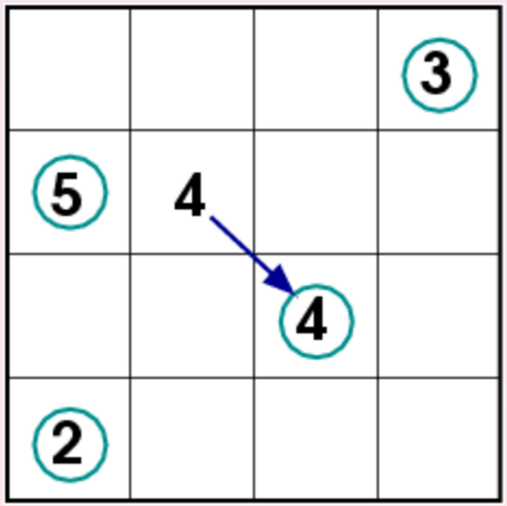
Ordering and filtering

Considering f as depicted below.

- The filter is an area closing ($\lambda = 4$) so $\mathcal{R}^{\uparrow}$ is used.
- Current point p is number **5**.
- Output will “propagate” f value of root points.



make_set(p)

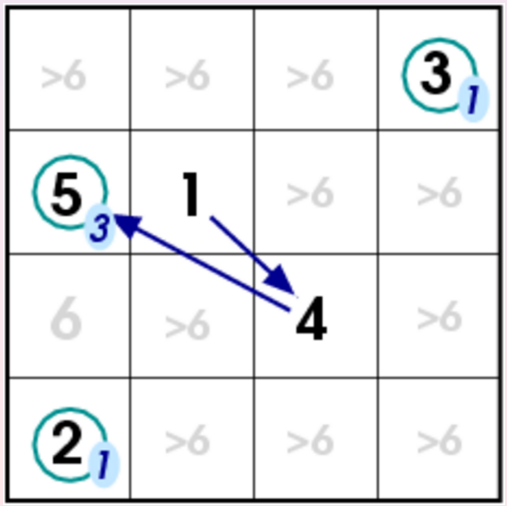


temp. output

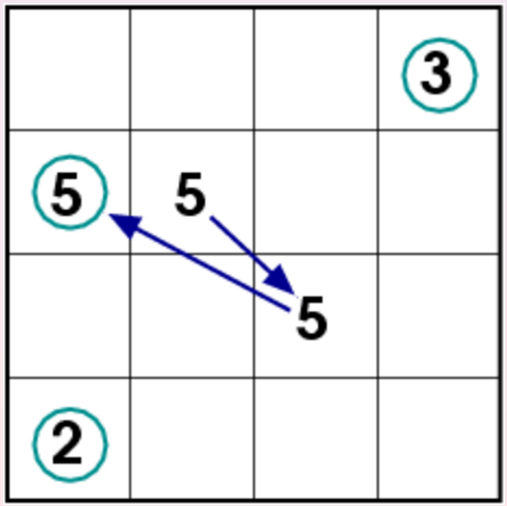
Ordering and filtering

The neighbor component of $\{p\}$ is $\{1, 4\}$.

- Its area is $2 < \lambda$ so it has to grow.
- It is thus merged with $\{p\}$.
- Output will “propagate” f value of root points.



set_parent(r, p)



output

Operators over two functions

Given

- two functions: f and g;
- two operators: φ extensive and ψ anti-extensive.

Many operators (usually self-dual) are defined by:

$$[\theta(f,g)](p) = \begin{cases} [\varphi(f,g)](p) & \text{if } p \in D^{\uparrow}(f,g) \\ [\psi(f,g)](p) & \text{if } p \in D^{\downarrow}(f,g) \\ f(p) & \text{otherwise.} \end{cases}$$

where

- θ is disjointness-preservative w.r.t. $D^{\uparrow}(f,g)$ and $D^{\downarrow}(f,g)$
- θ is expected to be extensive on $D^{\uparrow}(f,g)$ and anti-extensive on $D^{\downarrow}(f,g)$.

Domains for operators over two functions

Let us introduce:

- $D^{\circ} = \left(D^{\uparrow} \cup D^{\downarrow} \right)^{C}$
- $D^{\Updownarrow} = D^{\uparrow} \cup D^{\circ}$
- $D^{\Updownarrow} = D^{\downarrow} \cup D^{\circ}$.

For instance:

- with

$$[\theta(f,g)](p) = \begin{cases} [lev^{\uparrow}(f,g)](p) & \text{if } f(p) < g(p) \\ [lev^{\downarrow}(f,g)](p) & \text{if } f(p) > g(p) \\ f(p) & \text{if } f(p) = g(p). \end{cases}$$

- we have

$$D^{\uparrow}(f,g) = \{ p \in D,\ f(p) < g(p) \}$$
$$D^{\downarrow}(f,g) = \{ p \in D,\ f(p) > g(p) \}$$
$$D^{\circ}(f,g) = \{ p \in D,\ f(p) = g(p) \}.$$

Domains for operators over two functions

Then:

- a trivial partition of D is $D = D^{\uparrow} \cup D^{\circ} \cup D^{\downarrow}$
- but θ is not disjointness-preservative w.r.t. to $D^{\uparrow}$ and D°
  the same goes for $D^{\downarrow}$ and D°
- and we cannot simply ignore D°
  results in other sub-domains are bound to it.

Domains for operators over two functions

Now:

- let us form $D' = D_1 \cup D_2$ with $D_1 = D^{\Updownarrow}$ and $D_2 = D^{\Updownarrow}$.
- points of D° have to be processed twice
  to get proper results both in $D^{\uparrow}$ and in $D^{\downarrow}$
- until now, a point is only processed once in TUFA's first pass
- so we just have to add a "refresh" step for the points of D° between handling $D^{\Updownarrow}$ and $D^{\Updownarrow}$

First pass for operators over two functions

```
first_pass()
{
  for_all_points (p) if ( is_in_D_upper_or_equal(p) ) {
    // body 1.1
    is_processed[p] = true;
  }

  for_all_points (p) if ( is_in_D_equal(p) ) is_processed[p] = false;

  for_all_points (p) if ( is_in_D_lower_or_equal(p) ) {
    // body 1.2
    is_processed[p] = true;
  }
}
```

What have we done?

- we have started from a simple “single sub-domain” expression of TUFA
- we have extended this formulation to handle multiple sub-domains...
- ...while preserving domain-disjointness
- we have modified its form to handle the general case of $D^{\Updownarrow} \cup D^{\Updownarrow}$
- that gives an easy access to implement a lot of filters...

What's worth remembering

- TUFA has a general formulation, well-suited to morphological operators
- it is very simple

  when implementation details do not spoil algorithmic description!
- that gives an easy access to implement a lot of filters...
- a quite long catalogue of filters' implementations is provided in the paper

  and in a very concise way!

What's worth remembering

- we also present in the paper TUFA-based reconstructions
  as far as we know, this is a contribution (?)
- and about other issues...
  - we are at least as fast as the "hybrid" algorithm
    from our first experiments
  - TUFA used a fixed amount of memory
    that contrasts with queue-based algorithms
  - some natural optimizations can be performed
  - parallelization strategies on TUFA are much easier than with other algorithms.

...

Thanks for having read this material.

Remember to have a look at the olena library.

Version 1 is coming summer 2005...

This is modern generic free software.

Contributions are welcome!

olena@lrde.epita.fr