Une approche générique du logiciel pour le traitement d'images préservant les performances

Roland Levillain[1,2], Thierry Géraud[1,2], Laurent Najman[2]

[1]Laboratoire de Recherche et Développement de l'EPITA (LRDE), France
[2]Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge (LIGM), ESIEE Paris, France
{roland.levillain,thierry.geraud}@lrde.epita.fr, l.najman@esiee.fr

De la réutilisabilité des algorithmes

Problème Dans de nombreux outils de traitement d'images (TDI), les algorithmes sont écrits pour des types de données spécifiques → manque de réutilisabilité

Exemple Dilatation morphologique avec un élément structurant plat

Algorithme 1 – Implémentation **non générique** de la dilatation

```
image dilation(const image& input) {
  image output(input.nrows(), input.ncols());
  for (unsigned r = 0; r < input.nrows(); ++r)
    for (unsigned c = 0; c < input.ncols(); ++c) {
      unsigned char sup = input(r,c);
      if (r != 0 && input(r-1,c) > sup)                sup = input(r-1,c);
      if (r != input.nrows()-1 && input(r+1,c) > sup) sup = input(r+1,c);
      if (c != 0 && input(r,c-1) > sup)                sup = input(r,c-1);
      if (c != input.ncols()-1 && input(r,c+1) > sup) sup = input(r,c+1);
      output(r, c) = sup;
    }
  return output;
}
```

Observation Type des données fixes dans l'algorithme 1

- image 2D, grille régulière, valeurs compatibles avec `unsigned char`
- élément structurant 4-connexe implicitement ancré dans le code

Limitations Impossible de traiter d'autres types d'images

Solution Adopter une approche *générique* [2]

- Repartir d'une définition générale de l'algorithme :

$$\delta_B(I)(x) = \bigvee_{h \in B} I(x+h) \quad \text{avec} \begin{cases} \delta_B : \text{opérateur de dilatation} \\ I : \text{image d'entrée} \\ B : \text{élément structurant (plat)} \end{cases}$$

- Écrire une version générique de cet algorithme

Algorithme 2 – Implémentation **générique** de la dilatation

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  mln_piter(I) p(input.domain()); // Itérateur sur le domaine de 'input'
  mln_qiter(W) q(win, p); // Itérateur sur voisins de 'p' selon 'win'
  for_all(p) {
    accu::supremum<mln_value(I)> sup; // Objet calculant le sup de 'win'
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
  }
  return output;
}
```

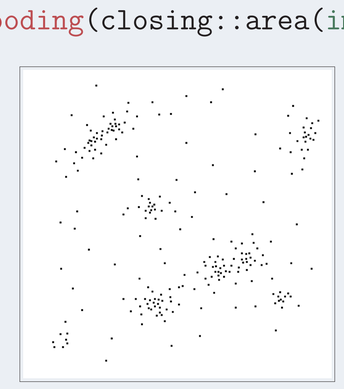
Bénéfice Pas de détails d'implémentation → réelle réutilisabilité de l'algorithme

Évolution Généraliser l'approche à l'aide d'un *framework* de TDI générique [3]

Illustration : une chaîne de traitement générique avec Milena [1]

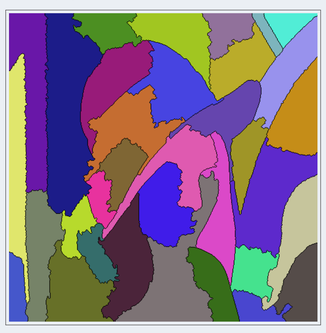
```
template <typename L, typename I, typename N>
mln_ch_value(I, L) chain(const I& ima, const N& nbh, int l, L& nb)
  { return watershed::flooding(closing::area(ima, nbh, l), nbh, nb); }
```

(a) Image 2D régulière

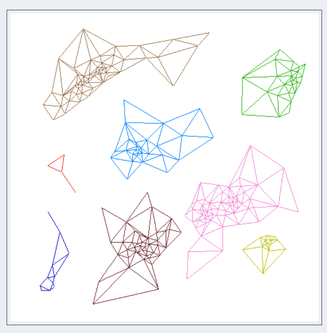


(b) Sommets d'un graphe planaire

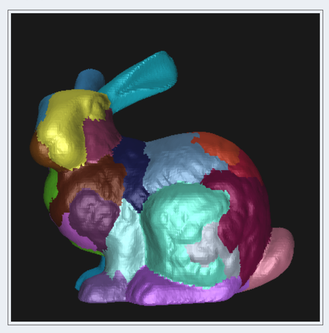
(c) Image sur surface maillée



(d) Résultat sur le gradient de (a)



(e) Résultat sur les arêtes de (b)



(f) Résultat sur la courbure de (c)

Généricité, performance et optimisations génériques

Problème Code générique potentiellement plus lent (coût de l'abstraction)

Cause L'absence de détails d'implémentation empêche le compilateur de générer du code optimisé.

Solution Tirer parti des caractéristiques des types d'entrées pour écrire des variantes rapides d'un algorithme

- Identifier une ou des caractéristique(s) permettant de produire une implémentation rapide (ex : données stockées régulièrement en mémoire)
- Déterminer le sous-ensemble des entrées de l'algorithme initial leur correspondant
- Écrire une variante tirant parti des propriétés inhérentes à ce sous-ensemble de types de données
- Code résultant : moins générique, mais pas dédié à un unique type pour autant → *optimisation générique*

Algorithme 3 – Implémentation de dilatation **optimisée, en partie générique**

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  mln_pixter(const I) pi(input);  // Itérateur sur pixels de 'input'
  mln_pixter(I) po(output);       // Itérateur sur pixels de 'output'
  mln_qixter(const I, W) q(pi, win); // Itérateur pixels voisins de 'pi'.
  for_all_2(pi, po) {  // Itération synchrone de 'pi' & 'po'.
    accu::supremum<mln_value(I)> sup;
    for_all(q)
      sup.take(q.val());
    po.val() = sup.to_result();
  }
  return output;
}
```

Résultat Réel gain à l'exécution

Observation Code de l'algorithme 3 proche de celui de l'algorithme 2 → en pratique, il est utile de commencer par écrire la version générique

Remarque Parcours des images réalisé à l'aide deux objets assimilables à des pointeurs (*pixters*) au lieu d'un unique itérateur partagé (*piter*) assimilable à point dans un espace 2D

Temps d'exécutions des algorithmes de dilatation

Implémentation	Temps pour dix invocations successives (secondes) Image (pixels) : 512^2	1024^2	2048^2
Non générique (algorithme 1)	0.10	0.39	1.53
Non générique, optimisée avec pointeurs (non montrée ici)	0.07	0.33	1.27
Générique (algorithme 2)	0.99	4.07	16.23
Optimisée, en partie générique (algorithme 3)	0.13	0.54	1.95
Algorithme 3 avec une fenêtre statique	0.06	0.28	1.03

Matériel PC avec Intel Pentium 4 @ 3,4 GHz et 2 Go RAM @ 400 MHz

Logiciel Compilateur g++ (GCC) 4.4.5, optimisation '-O3', sous Debian GNU/Linux

La plate-forme de TDI générique et performante Olena

- Implémentations basées sur la bibliothèque générique Milena, au cœur d'Olena
- Logiciel libre diffusé sous GNU General Public License (GNU GPL)
- Disponible librement sur le Web [1] (contact : olena@lrde.epita.fr)

Références

[1] EPITA Research and Developpement Laboratory (LRDE).
The Olena image processing platform.
http://olena.lrde.epita.fr.

[2] Roland Levillain, Thierry Géraud et Laurent Najman.
Milena : Write generic morphological algorithms once, run on many kinds of images.
In Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM), 2009.

[3] Roland Levillain, Thierry Géraud et Laurent Najman.
Why and how to design a generic and efficient image processing framework : The case of the Milena library.
In Proceedings of the IEEE International Conference on Image Processing (ICIP), 2010.