INF-4102A : C++ et généricité

ESIEE PARIS

2017

Rappel

Dans le pseudo-code à droite,
— l'image d'entrée est `input`, une image binaire ;
— `D` est le domain de définition de l'image d'entrée (pour une image 2D "classique", c'est un rectangle, une boîte) ;
— la sortie est `dmap`, une image contenant des `unsigned` ;
— `max` est la valeur maximale des `unsigned` ;
— `p` et `n` sont des itérateurs sur des points (un point est un couple de coordonnées, 2 entiers donc) ;
— `N` est un voisinage ; par exemple, si $p = (2, 3)$, ses voisins forment l'ensemble $\mathcal{N}(p) = \{ (1, 3), (2, 2), (2, 4), (3, 3)\}$;
— `q` est une queue (conteneur *first in, first out*) de points.

```
D <- input.domain

for all p in D
   dmap <- max

for all p in D
   if input(p) = true
      dmap(p) <- 0
      for all n in N(p) and in D
         if input(n) = false
            q.push(p)
            break

while q is not empty
   p <- q.pop()
   for all n in N(p) and in D
      if (dmap(n) = max)
         dmap(n) <- dmap(p) + 1
         q.push(n)
```

On a les *concepts* suivants :
— un point ; un domaine / ensemble de points ; un itérateur sur un domaine ;
— une image ;
— un itérateur sur un voisinage de point.

On veut les classes concrètes correspondantes pour le cas classique :
— point2d ; box2d ; box2d_iterator ;
— image2d$< T >$;
— neighb2d_iterator.

Sans modifier l'algorithme, on voudrait :
— pouvoir calculer une carte de distance à une image d'étiquettes ;
— pouvoir calculer en même temps (que la carte de distances) une carte de la plus proche étiquette.

1 Du code

1.1 L'algorithme

Pour l'instant, votre code doit ressembler à ça :

```
using bool_t = unsigned;

image2d<unsigned> compute_dmap(const image2d<bool_t>& input)
{
   box2d D = input.domain();

   const unsigned max = unsigned(-1);
   image2d<unsigned> dmap(D);

   box2d_iterator p(D);
   for (p.start(); p.is_valid(); p.next())
      dmap(p) = max;

   std::queue<point2d> q;
   neighb2d_iterator n;

   for (p.start(); p.is_valid(); p.next())
      if (input(p) == true)
      {
         dmap(p) = 0;
         n.center_at(p);
         for (n.start(); n.is_valid(); n.next())
            if (D.has(n) and input(n) == false)
            {
               q.push(p);
               break;
            }
      }

   while (not q.empty())
   {
      point2d p = q.front();
      q.pop();
      n.center_at(p);
      for (n.start(); n.is_valid(); n.next())
         if (D.has(n) and dmap(n) == max)
         {
            dmap(n) = dmap(p) + 1;
            q.push(n);
         }
   }

   return dmap;
}
```

1.2 Les structures de données en code à *trous*

Elles doivent ressembler à ça :

```
struct point2d
{
  // ...
  int row, col;
};


// forward declaration,
// required for the lines (*) to compile
class box2d_iterator;
class neighb2d_iterator;

class box2d // is a Domain type
{
public:
  using p_iterator_type = box2d_iterator; // (*)
  using n_iterator_type = neighb2d_iterator; // (*)
  bool has(const point2d& p) const { //...
  // ...
private:
  unsigned nrows_, ncols_;
};


class box2d_iterator
// iterator over the set of points
// contained in a 2D box
{
public:
  box2d_iterator(const box2d& b) : dom_(b) {}
  void start() { // ...
  bool is_valid() const { // ...
  void next() { // ...
  // ...

  // to allow automatic coercion of objects from
  // this type to point2d:
  operator point2d() const {
    return current_p_;
  }
private:
  box2d dom_;
  point2d current_p_;
};
```

```
class neighb2d_iterator
// iterator over the set of neighbors
// of a 2D point (the attribute p_)
{
public:
  neighb2d_iterator() // ctor
  {
    delta_.push_back(point2d(-1, 0));
    // ...
  }

  void center_at(const point2d& p)
  {
    // change p_...
  }

  // as an iterator:
  void start() { // ...
  bool is_valid() const { // ...
  void next() { // ...

  // to allow automatic coercion of objects from
  // this type to point2d:
  operator point2d() const {
    point2d n;
    n.row = p_.row + delta_[i_].row;
    n.col = p_.col + delta_[i_].col;
    return n;
  }

private:
  std::vector<point2d> delta_;
  unsigned i_; // current index in delta_
  point2d p_;  // center point p_
};



template <typename T>
class image2d
{
public:
  using value_type = T;
  using point_type = point2d;
  using domain_type = box2d;

  // ctors:
  image2d(const domain_type& d) { // ...
  image2d(unsigned nrows, unsigned ncols) { // ...

  // access to pixel values:
  T& operator()(const point_type& p) { // ...
  T  operator()(const point_type& p) const { // ...

  const domain_type& domain() const { return dom_; }
  box2d bounding_box() const { return dom_; }

  template <typename U, unsigned n>
  void fill_with(U (&vals)[n]) {
    for (unsigned i = 0; i < n; ++i)
      data_[i] = vals[i];
  }
  // ...
private:
  box2d dom_;
  std::vector<T> data_; // T cannot be bool; use bool_t
};
```

Notez que la méthode `bounding_box()` d'une image renverra toujours une `box2d`, même pour des types images qui, par la suite, ne seront pas du type `image2d<T>`.

1.3 Une nouvelle version (générique !) de l'algorithme

Si l'on veut que `compute_dmap` accepte différents types d'images 2D en entrée, on peut rendre son code *générique* :

```
template <typename I>
image2d<unsigned> compute_dmap(const I& input)
{
   // these 4 new lines below express that some types depends on I
   // and on domain_type (that is, I::domain_type)
   using point_type  = typename I::point_type;
   using domain_type = typename I::domain_type;
   using p_iterator_type = typename domain_type::p_iterator_type;
   using n_iterator_type = typename domain_type::n_iterator_type;

   const domain_type& D = input.domain();
       // instead of: box2d D = input.domain();

   const unsigned max = unsigned(-1);
   image2d<unsigned> dmap(input.bounding_box());
       // instead of: image2d<unsigned> dmap(D);

   p_iterator_type p(D);
       // instead of: box2d_iterator p(D);
   for (p.start(); p.is_valid(); p.next())
      dmap(p) = max;

   std::queue<point_type> q;
   n_iterator_type n;
       // instead of: std::queue<point2d> q;
       //             neighb2d_iterator n;

   for (p.start(); p.is_valid(); p.next())
      if (input(p) == true)
      {
         dmap(p) = 0;
         n.center_at(p);
         for (n.start(); n.is_valid(); n.next())
            if (D.has(n) and input(n) == false)
            {
               q.push(p);
               break;
            }
      }

   while (not q.empty())
   {
      point2d p = q.front();
      q.pop();
      n.center_at(p);
      for (n.start(); n.is_valid(); n.next())
         if (D.has(n) and dmap(n) == max)
         {
            dmap(n) = dmap(p) + 1;
            q.push(n);
         }
   }

   return dmap;
}
```

2 Le projet

2.1 Un exemple d'un autre type d'images

Une image d'étiquettes contient des valeurs de type `unsigned`. La valeur 0 désigne le fond ; c'est une "non-étiquette" ; chaque valeur non nulle désigne un objet particulier. Une image d'étiquettes est un outil classique du traitement d'images, très souvent utilisé pour étiqueter les composantes connexes d'une image binaire. Par exemple, avec l'image binaire suivante :

```
0 0 0 0 0
0 0 0 1 0
0 0 1 1 0
0 0 0 0 0
1 0 0 1 0
0 0 0 0 0
```

l'étiquetage de ses composantes donne l'image d'étiquettes l :

```
0 0 0 0 0
0 0 0 1 0
0 0 1 1 0
0 0 0 0 0
2 0 0 3 0
0 0 0 0 0
```

Les valeurs `1`, `2` et `3` désignent ici un objet en particulier. Dans la section suivante, vous verrez que l'on va se servir d'étiquettes pour désigner, dans une image, un point de départ, un point d'arrivée et des points sur lesquels on peut marcher (un labyrinthe donc...)

Soit une image i et une fonction f. Voir les valeurs des pixels de l'image i à travers la fonction f signifie considérer une nouvelle image. Par exemple, "voir" l'image l à travers la fonction $x \mapsto x^2$ donne l'image ℓ :

```
0 0 0 0 0
0 0 0 1 0
0 0 1 1 0
0 0 0 0 0
4 0 0 9 0
0 0 0 0 0
```

Cette image n'est pas obligée d'exister en mémoire puisque les valeurs de ses pixels peuvent être calculées à la volée (remarque : on peut lire les valeurs de l'image, mais pas les modifier). En orienté-objet, cette image s'appelle un *objet léger* car son coût en mémoire est très faible.
Considérons ces types d'objets-fonctions :

```
struct fun_sqr
{
  using result_type = unsigned;
  result_type operator()(unsigned x) const {
    return x * x;
  }
};

struct fun_equal
{
  using result_type = bool;
  result_type operator()(unsigned x) const {
    return x == val_;
  }
  unsigned val_;
};
```

Ils peuvent être utilisés avec le nouveau type d'images ci-dessous :

```
template <typename I, typename F>
class image_through
{
public:
  using value_type  = typename F::result_type; // result type of f_
  using point_type  = typename I::point_type;
  using domain_type = typename I::domain_type;

  // ctor:
  image_through(const I& ima, F f) : ima_(ima), f_(f) {}

  // access to pixel values:
  value_type operator()(const point_type& p) const
  {
    return f_(ima_(p));
  }

  const domain_type& domain() const { return ima_.domain(); }
  box2d bounding_box() const { return ima_.bounding_box(); }

private:
  I ima_;
  F f_;
};


// procédure 'through' qui crée une image légère:

template <typename I, typename F>
image_through<I,F> through(const I& ima, F f)
{
  return image_through<I,F>{ima, f};
}


int main()
{
  box2d D{2, 3};
  image2d<unsigned> ima(D);
  int vals[] = { 0, 0, 0, 0, 0,
                 0, 0, 0, 1, 0,
                 0, 0, 1, 1, 0,
                 0, 0, 0, 0, 0,
                 2, 0, 0, 3, 0,
                 0, 0, 0, 0, 0 };
  ima.fill_with(vals);
  auto ima_ell = through(ima, fun_sqr{});
  auto ima_bin = through(ima, fun_equal{2});
}
```

Et ici `ima_ell` est bien l'image ℓ.
Mais à quoi ressemble l'image `ima_bin` ?

2.2 Où l'on fait le lien entre une carte de distance et un labyrinthe

L'algorithme calcule une carte (image) de distances à une image binaire :

$$\text{entrée } e \;=\; \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \longrightarrow \text{sortie } s \;=\; \begin{array}{ccccc} 4 & 3 & 2 & 1 & 2 \\ 3 & 2 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 1 & 2 & 1 & 1 & 2 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 2 & 2 & 1 & 2 \end{array}$$

On va maintenant s'intéresser aux labyrinthes...

Considérez l'image suivante E (c'est l'image dont on parle au "**Niveau 7**" plus loin) :

```
2 1 1 1 0
0 0 0 1 0
0 1 1 1 0
0 1 0 1 0
1 1 0 0 0
0 1 1 1 3
```

la valeur `0` représente un bout de mur, la valeur `2` un point de départ, la valeur `3` un point d'arrivée, et la valeur `1` une case de chemin. Pour aller du point à `2` au point à `3`, il y a une certaine distance, mais on ne peut pas marcher sur les murs. Cette distance ne se calcule que sur le chemin, c'est à dire dans ce domaine :

```
x x x x
      x
  x x x
  x   x
x x
  x x x x
```

Ce domaine n'est plus un rectangle (il nous faudra un **autre** type de domaine donc...) mais sa boîte englobante (*bounding box* in English) est encore une `box2d`. Si on sait définir des images sur des domaines quelconques (non rectangulaires, et il nous faudra donc un **nouveau** type d'images pour ça...), on peut avoir cette image binaire :

```
1 0 0 0
      0
  0 0 0
  0   0
0 0
  0 0 0 0
```

et une carte de distances peut être calculée à partir de cette image ; le résultat attendu est le suivant :

```
 0  1  2  3
          4
    7  6  5
    8     6
10  9
   10 11 12 13
```

Au final, on en déduit que le point d'arrivée, de valeur `3` dans l'image E ci-dessus, est à 13 cases de distance du point de départ. Et ça veut clairement dire que le chemin "départ → arrivée" est donné, de façon cachée (!), par l'algorithme de calcul d'une carte de distances.

3 Projet

Le projet à rendre comporte plusieurs niveaux ; plus vous passez de niveaux, mieux c'est (mais ne pas faire tous les niveaux ne sera pas si pénalisé !)

Niveau 1. On s'attend à avoir la classe `point2d`, la classe `box2d`, deux itérateurs (un sur un domaine rectangulaire, un sur le voisinage d'un point).

Niveau 2. On veut une classe d'image définie sur un domaine rectangulaire `image2d<T>` (où `T` est le type des valeurs des pixels).

Niveau 3. On veut l'algorithme de calcul d'une carte de distance (Cf. sujets précédents).

Niveau 4. On veut un nouveau type de *domaine*, la classe `partial_box2d` ; elle a un unique attribut, `ima_` de type `image2d<unsigned>` Si `ima_` est

```
2 1 1
0 0 1
1 1 1
```

le domaine est l'ensemble des points dont la valeur du pixel n'est pas nulle. On veut un nouveau type d'itérateur sur ce domaine : la classe `partial_box2d_iterator`. Itérer sur le domaine précédent donne les points suivants : $(0,0)\,(0,1)\,(0,2)\,(1,2)\,(2,0)\,(2,1)\,(2,2)$.

Niveau 5. On veut une nouvelle classe d'image, `partial_image2d<T>`, dont le domaine est de type `partial_box2d`. Une image de ce type est par exemple :

```
2  1  1  1
         1
   1  1  1
   1     1
1  1
   1  1  1  3
```

Niveau 6. Considérez l'image suivante :

```
1  0  0  0
         0
   0  0  0
   0     0
0  0
   0  0  0  0
```

Elle doit pouvoir entrer dans l'algorithme de calcul de carte de distances, et le résultat doit être :

```
 0  1  2  3
          4
    7  6  5
    8     6
10  9
   10 11 12 13
```

Niveau 7. L'image E donnée en page 7 est la donnée en entrée du projet. On veut résoudre le problème du labyrinthe et obtenir le chemin suivant :

```
1  1  1  1
         1
   1  1  1
   1
   1
   1  1  1  1
```

Comment peut-on faire ? À expliquer dans un fichier texte (pas .doc !) nommé `README`.

Niveau 8. Le faire.