Université catholique de Louvain

LINGI1122

Méthodes de conception des programmes

Rapport de la première partie du projet

Authors :
Van den bogaert
Alexis(37181400)
Ortegat Pierre
(19541400)
Rucquoy
Alexandre(15011400)

Professor :
Pecheur Charles

13 avril 2017

1 La théorie du problème

1.1 Problème posé

On créé une classe couverture qui est la représentation d'un ensemble de rectangle. Cet ensemble sera sous forme de tableau. Le but ici sera de fusionner les rectangles en plus grands rectangles jusqu'à obtention d'une solution localement optimale, c'est-à-dire avoir le moins de rectangles possibles.

Chaque rectangle sera représenté par un tuple de 4 valeurs : son sommet en haut à gauche (x,y) sa longueur et sa largeur. Dès lors, il nous sera très facile de déterminer la position exacte des 3 autres sommets, ce qui est indispensable afin de déterminer si 2 rectangles sont adjacents, par exemple.

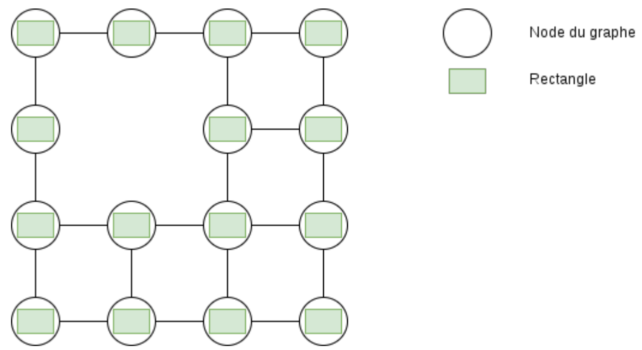
La solution que nous vous proposons permet de gérer les overlap, c'est-à-dire, le cas ou plusieurs rectangles se superposent. On considère ici que deux rectangles se superposent s'il y'a au moins un point (qui n'est PAS un sommet) de l'ensemble des points d'un rectangle A qui appartient aussi à l'ensemble des points du rectangle B.

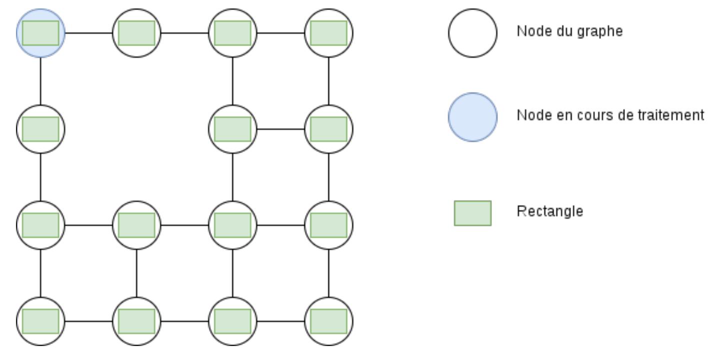
1.2 Stratégie de résolution utilisée

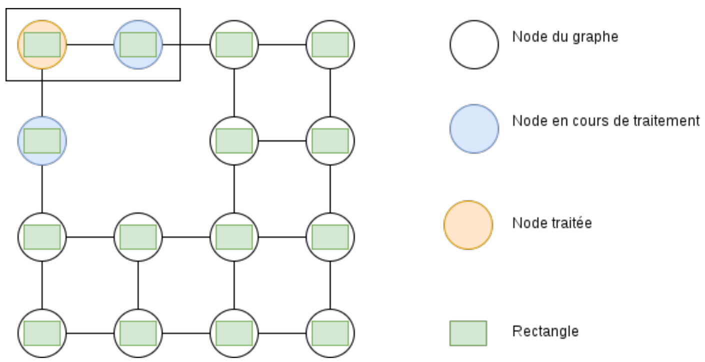
Au départ, nous avons donc une instance de la classe "couverture" qui sera un mesh de rectangles. Nous allons sur base de ce mesh créer un graphe dans lequel chaque noeud sera un rectangle. Chaque arête aura d'office un noeud de départ et un nœud d'arrivée différent. Dans notre graphe, l'existence d'une arête signifiera simplement que les deux rectangles représentés par ses nœuds de départ et arrivée sont adjacents.

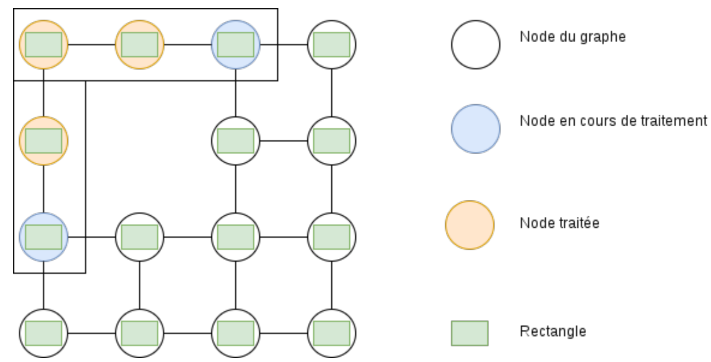
Le graphe sera créé de manière triée en fonction de ses nœuds. Comme dit plus haut, chaque rectangle est un tuple de 4 valeurs (x,y,w,h). Nous trierons donc les rectangles par ordre croissant de leur abscisse (x) puis par leur ordonnée (y) afin de construire le graphe. Notre objectif est de finir avec un graphe représentant en deux dimensions les rectangles dans le plan 2d.

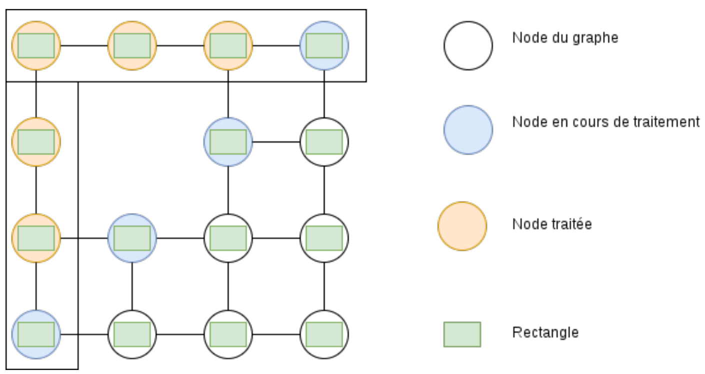
Nous parcourons ensuite le graphe créé en mode "tâche d'encre". Nous démarrons d'une node du graphe (peu importe laquelle) et nous explorons ensuite les nodes reliées à la node actuelle de la manière suivante :

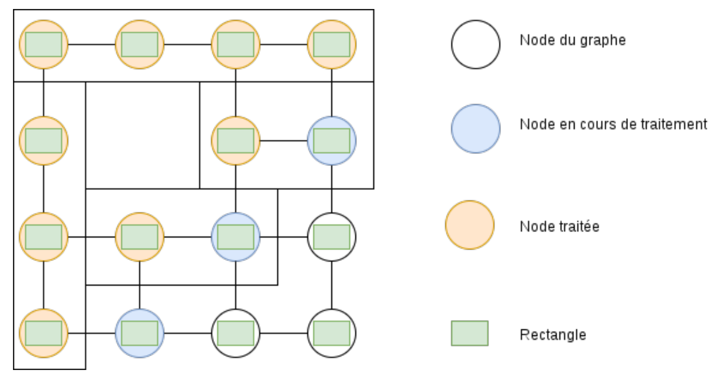

Node du graphe
Rectangle

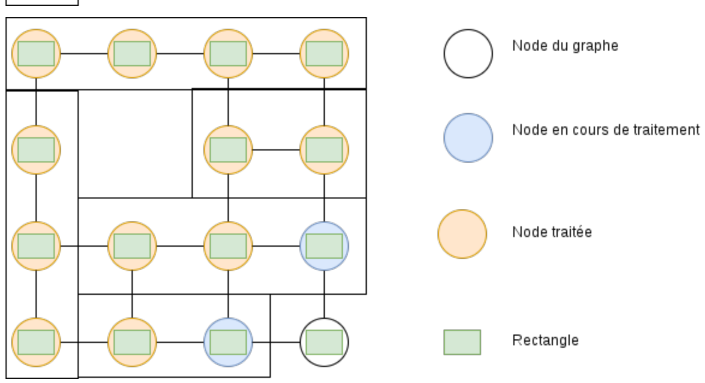

Node du graphe
Node en cours de traitement
Rectangle

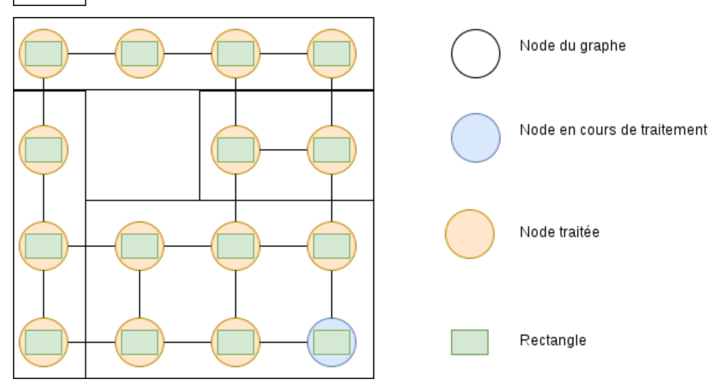

Node du graphe
Node en cours de traitement
Node traitée
Rectangle

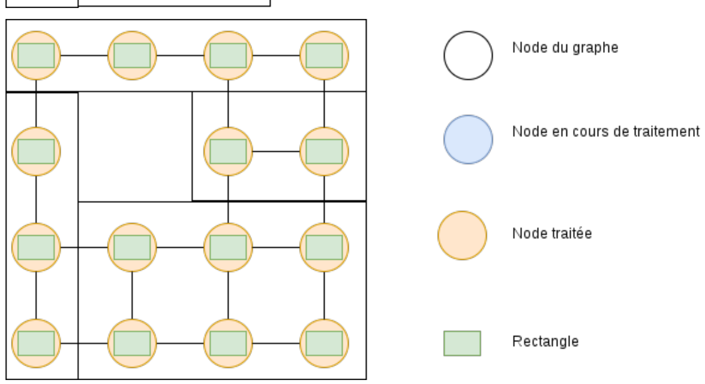

Node du graphe
Node en cours de traitement
Node traitée
Rectangle


Node du graphe
Node en cours de traitement
Node traitée
Rectangle


Node du graphe
Node en cours de traitement
Node traitée
Rectangle


Node du graphe
Node en cours de traitement
Node traitée
Rectangle


Node du graphe
Node en cours de traitement
Node traitée
Rectangle


Node du graphe
Node en cours de traitement
Node traitée
Rectangle

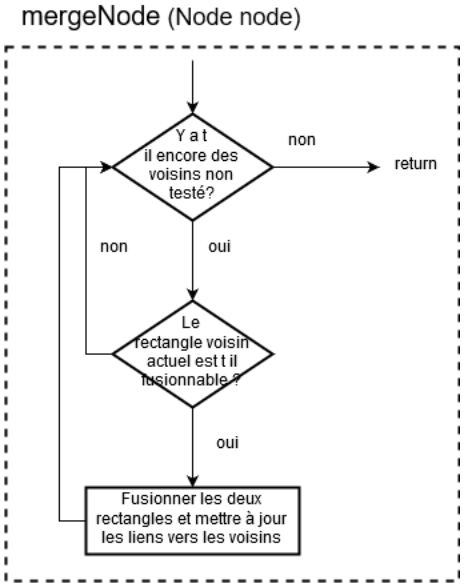
En le parcourant, nous regardons si pour chaque rectangle il existe une possibilité de le fusionner avec un autre afin de réduire le nombre total de rectangles. Nous fusionnons deux rectangles dans trois cas :

— Si les deux rectangles que l'on compare ont la même abscisse x (en d'autres termes, si l'un est au-dessus de l'autre) et w égaux (même largeur) et qu'ils sont adjacents (y1 = Y2+ h2)

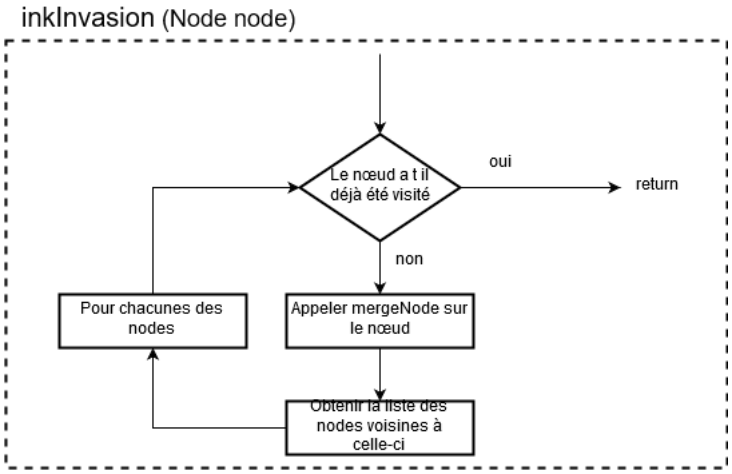
— Si les deux rectangles ont des ordonnées égales y (l'un aux côtés de l'autre) et h égaux (même hauteur) et qu'ils sont adjacents (x1 = x2+ w2)

— Si les deux rectangles sont l'un au-dessus de l'autre, complètement.

Les graphiques suivants montrent de quelle façon on compte diviser le problèmes en plus petites tâches. MergeNode fusionne une rectangle avec les rectangles voisins lorsque cela est possible.



Le fonction inkInvasion parcours le graphe (comme un tache d'encre qui s'étend) et fusionne les rectangles au passage.



1.3 Représentation des données

Ce programme utilisera principalement, deux types de structures, précisées ci-dessous :

1.3.1 La structure rectangle

Cette structure contient tout ce qui nous est nécéssaire pour définir correctement un rectangle sur un plan à deux dimension : les coordonées du point en haut à gauche du rectangle (x,y), la largeur du rectangle (w) ainsi que la heuteur du rectangle (h).

```
Rectangle ::= { nil | rectangle(int x, int y, int w, int h) }
```

le constructeurs de cette stucture est :

```
Rectangle rectangle(int x, int y, int w, int h)
```

les méthodes comprisent dans cette structure sont les suivantes :

```
getX():int - setX(int x):void
getY():int - setY(int y):void
getY():int - setW(int w):void
getH():int - setH(int h):void
```

1.3.2 La structure node

Cette structure est une node qui a pour but de faire partie d'un graphe à deux dimensions. Elle sert à positionner le rectangle qu'elle contient par rapport aux autres

rectangles du plan, contrairement à la façon classqiue qui positionne le rectangle via ses coordonées.

```
Node ::= { nil | Node(Rectangle r, Node []  neighbours,
boolean isVisited) }
```

son constructeur est :

```
couverture couverture(Rectangle r)
```

les methode de cette structure sont :

```
getRectangle():Rectangle
getNeighbours():Node[]
```

1.4 Prédicats utilisés

Overlap :

```
overlap(Rectangle r1, Rectangle r2) : bool
```

$$((r1.x + r1.l) > (r2.x) \vee (r1.y + r1.w) > (r2.y))$$

isNeighbour :

```
isNeighbour(Rectangle r1, Rectangle r2) : bool
```

$$((r1.x + r1.l) = (r2.x) \vee (r1.y + r2.w) = (r2.y))$$

2 Spécifications

2.1 Spécifications formelles

```
canMerge(Rectangle r1, Rectangle r2): bool
```

@pre : isNeighbour(r1, r2) $\wedge r1! = nill \wedge r2! = nill \wedge r1.x \geq 0 \wedge r1.y \geq 0 \wedge r1.w > 0 \wedge r1.h > 0 \wedge r2.x \geq 0 \wedge r2.y \geq 0 \wedge r2.w > 0 \wedge r2.h > 0$
@post : true

```
merge(Rectangle r1, Rectangle r2): Rectangle r
```

@pre :isNeighbour(r1,r2) $\wedge$ canMerge(r1,r2)
@post : r.w= ((r2.x+r2.w)-r1.x) $\wedge$ r.h= ((r2.y+r2.h)-r1.y)

improve(): void

@pre :$Couverture! = null$
@post : if $\exists Rectangler1, r2 canMerge(r1, r2) : merge(r1, r2)$ else locally optimized

optimize()

@pre : $\exists Couverture | Couveture! = nil \wedge Rectangle\{r1, r2\} \in Couverture | canMerge(r1, r2)$
@post : $\forall Rectangle\{r1, r2\} \in couverture, canMerge(r1, r2) : merge(r1, r2)$

contains(x:int, y:int): bool

@pre : node != null $\wedge \exists n \in node[] \wedge n.x = x \wedge n.y = y$
@post : true

2.2 Le type rectangle

Nous avons décider de représenter les rectangles par un tuple de 4 entiers (x,y,w,h) comme mentionné plus haut. Nous définissons maintenant le type concret correspondant :

Rectangle ::= { nil | rectangle(int x, int y, int w, int h) }

Avec la fonction d'abstraction correspondante suivante :

$$abs_R(R) =< x, y, w, h > \text{ si } R = rectangle(x, y, w, h)$$

Et l'invariant de représentation suivant :

$$ok_R(R) = \begin{cases} true & \text{si x,y} \geq 0 \wedge \text{w,h} > 0 \\ false & \text{sinon} \end{cases}$$

2.3 Le type Node

Nous avons pris la décision de représenter la classe Couverture par un graphe 2d composé de nodes. Les nodes sont définies concrètement de la façon suivante :

Node ::= { nil | Node(Rectangle r, Node [] neighbours, boolean isVisited) }

La fonction d'abstraction corespondante est la suivante :

$abs_N(N) = \ < r, isVisited, neighbours[0], ..., neighbourgs[n] >$ si N = Node(r, neighbours, isVisited)

L'invariant de représentation est le suivant :

$$ok_N(N) = \begin{cases} true & si\ r = \neg nil \wedge |neighbours| \geq 0 \\ true & si\ r = nil \\ false & sinon \end{cases}$$

2.4 le type Couverture

La fonction d'abstraction corespondante est la suivante :

$$abs_C(C) = \ < rect[0], rect[1], ..., rect[n-1] >$$ si R = Node[n]

L'invariant de représentation est le suivant :

$$ok_C(C) = \begin{cases} true & rect = \neg nil \\ true & si\ C = nil \\ false & sinon \end{cases}$$