L'image se recolle sur elle–même

Clément Picard

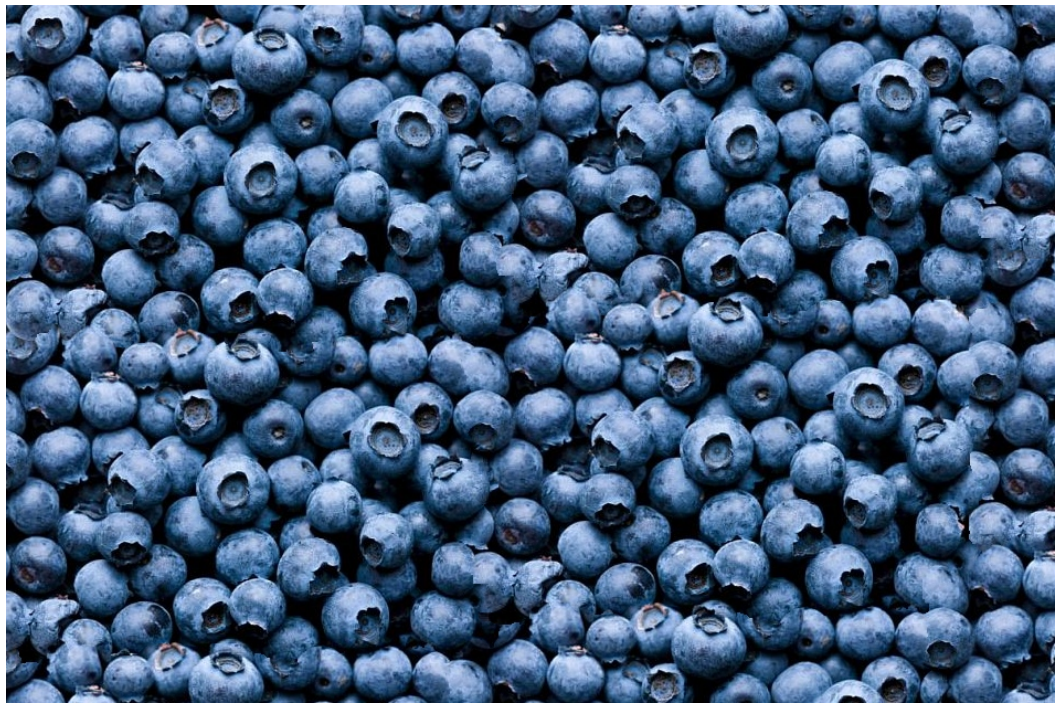
21 *avril* 2014

Plan

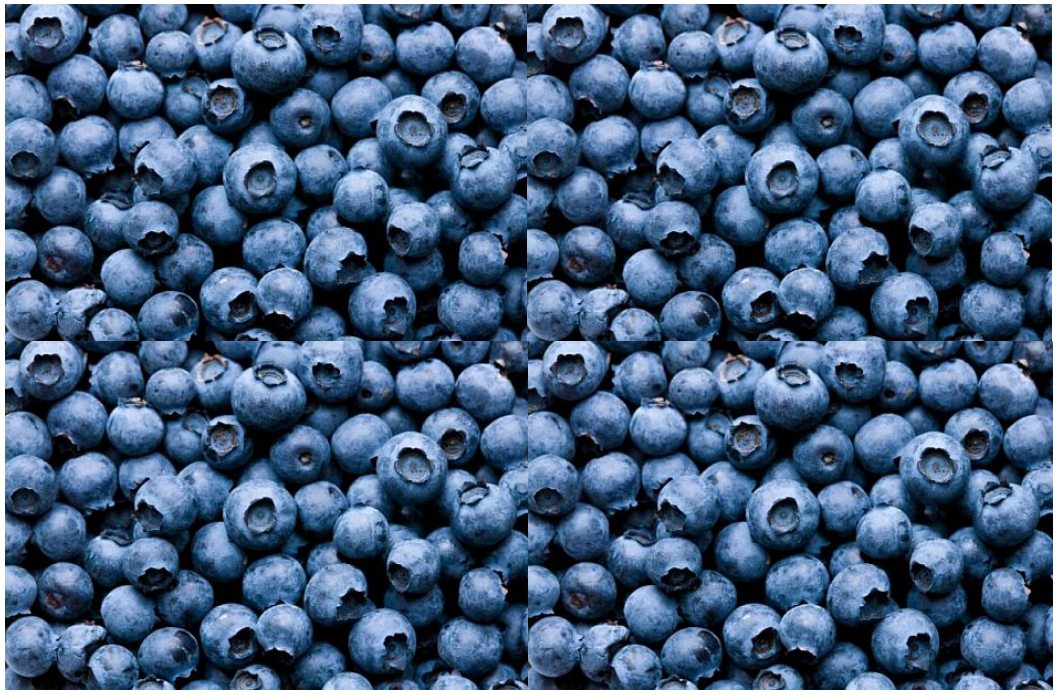
Position du problème

- Réussir à créer une grande image à partir d'une petite de façon automatique et virtuelle
- Reproduire le motif sans que des cassures n'apparaissent

$\mapsto$

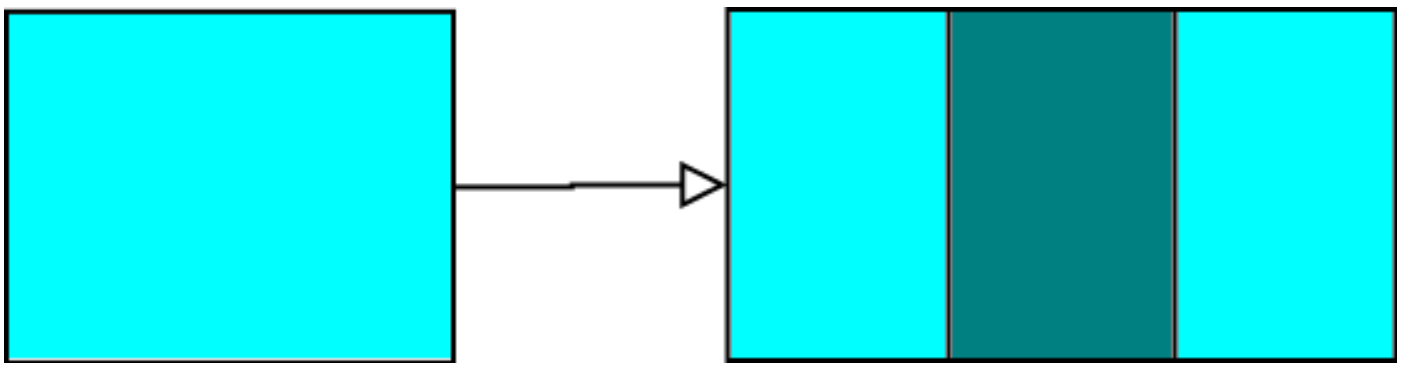


Éviter que ça se voie : ne pas juxtaposer



Le plugin texturize de Gimp

2005 : article de l'Université d'Atlanta (Georgia Institute of Technology) de Kwatra, Schödl, Essa, Turk, Bobick.
L'algorithme présenté dans cet article serait trop ambitieux pour être implémenté en informatique pour tous.
On simplifie le problème : on duplique l'image et on la recolle sur sa droite.



Un exemple

Image originale :

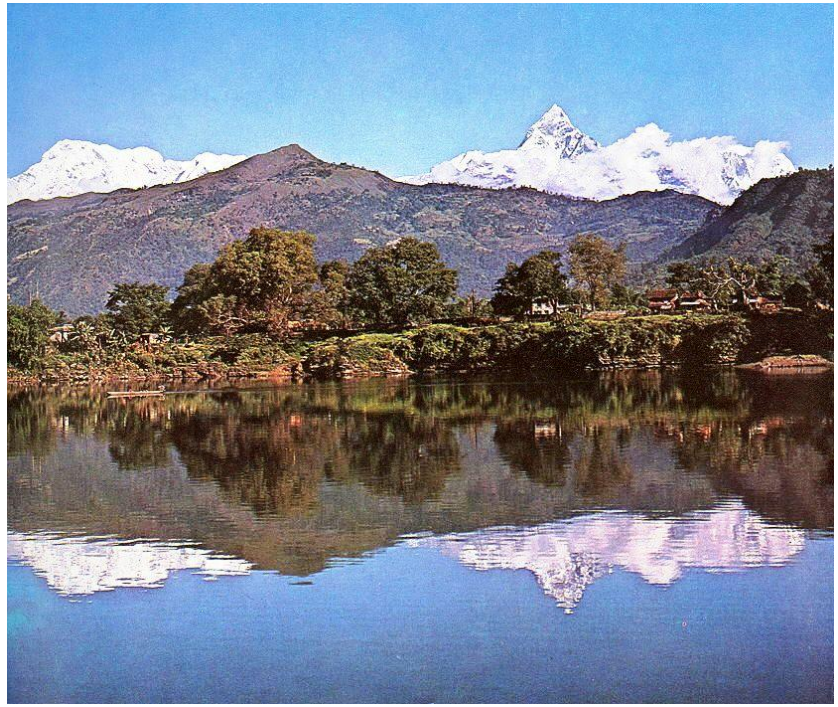
Photographie de George P. Landow aimablement mise à disposition
Adresse de son site : http://www.victorianweb.org/

Le recollement

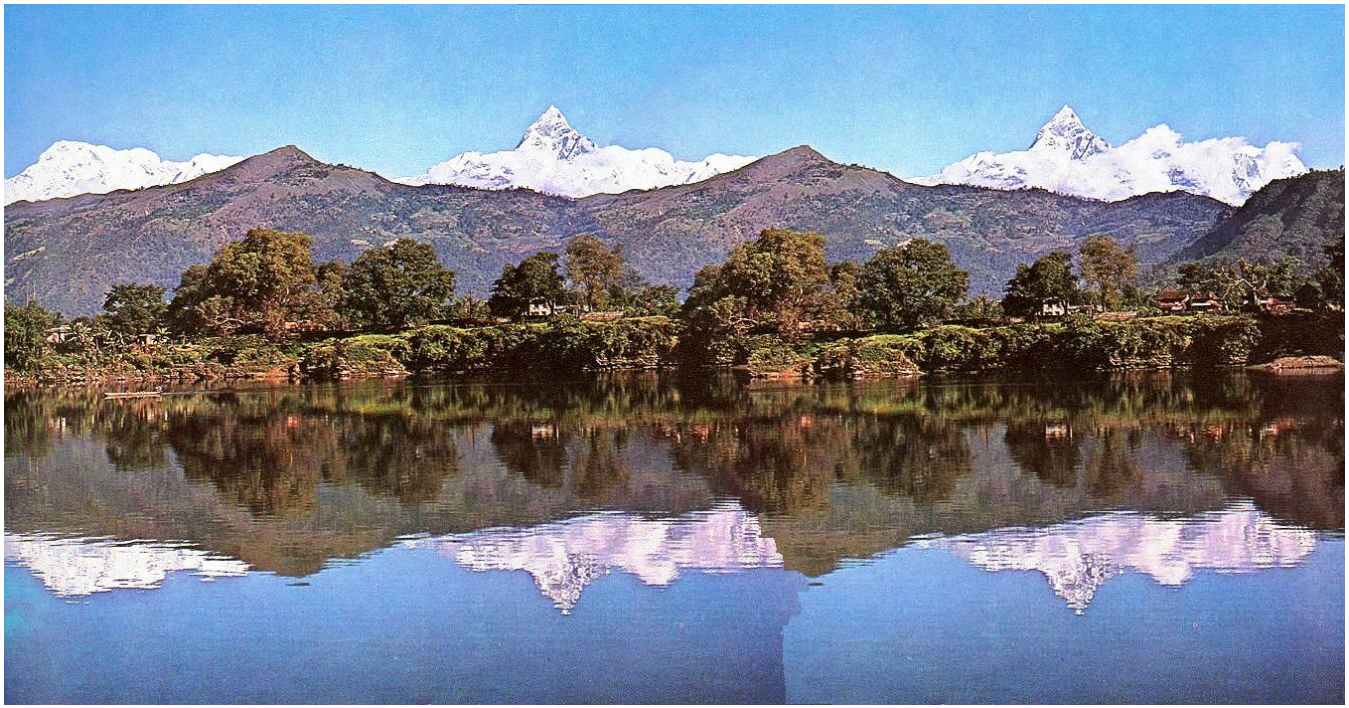
Image originale :

Photographie de George P. Landow aimablement mise à disposition
Adresse de son site : http://www.victorianweb.org/

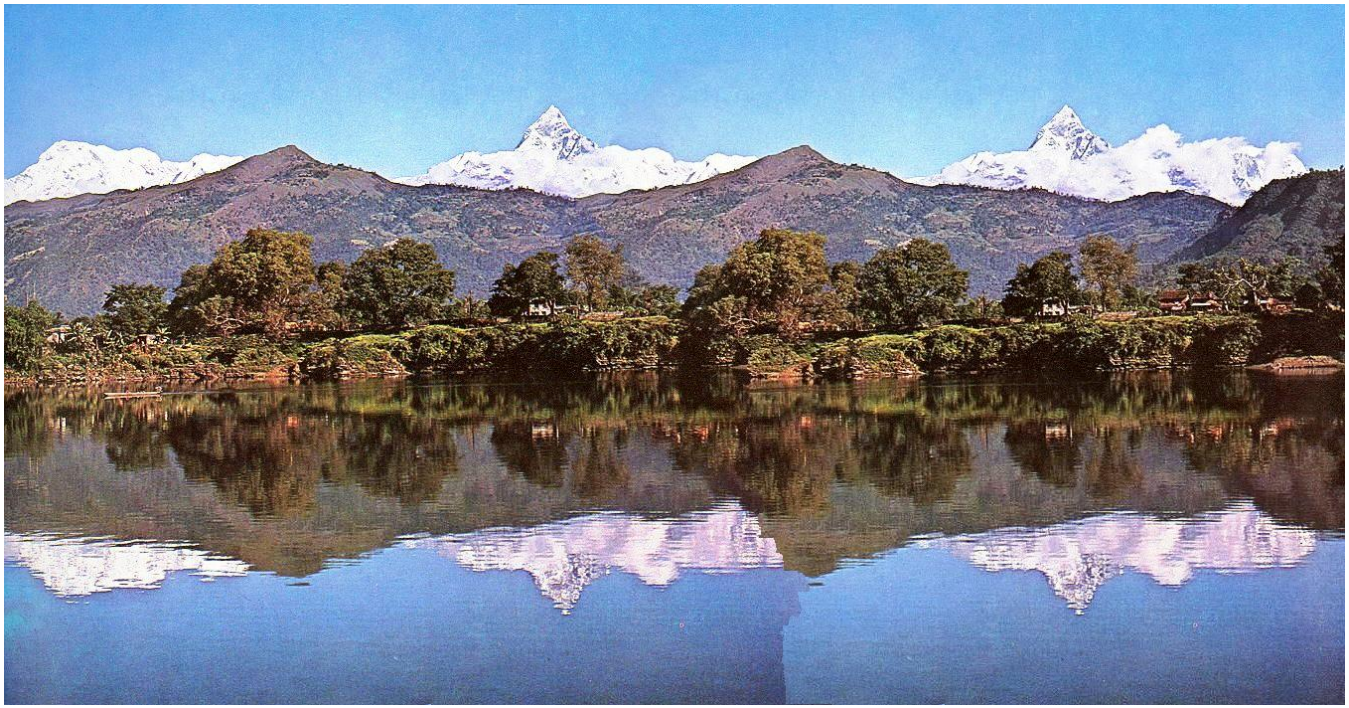
Un paysage

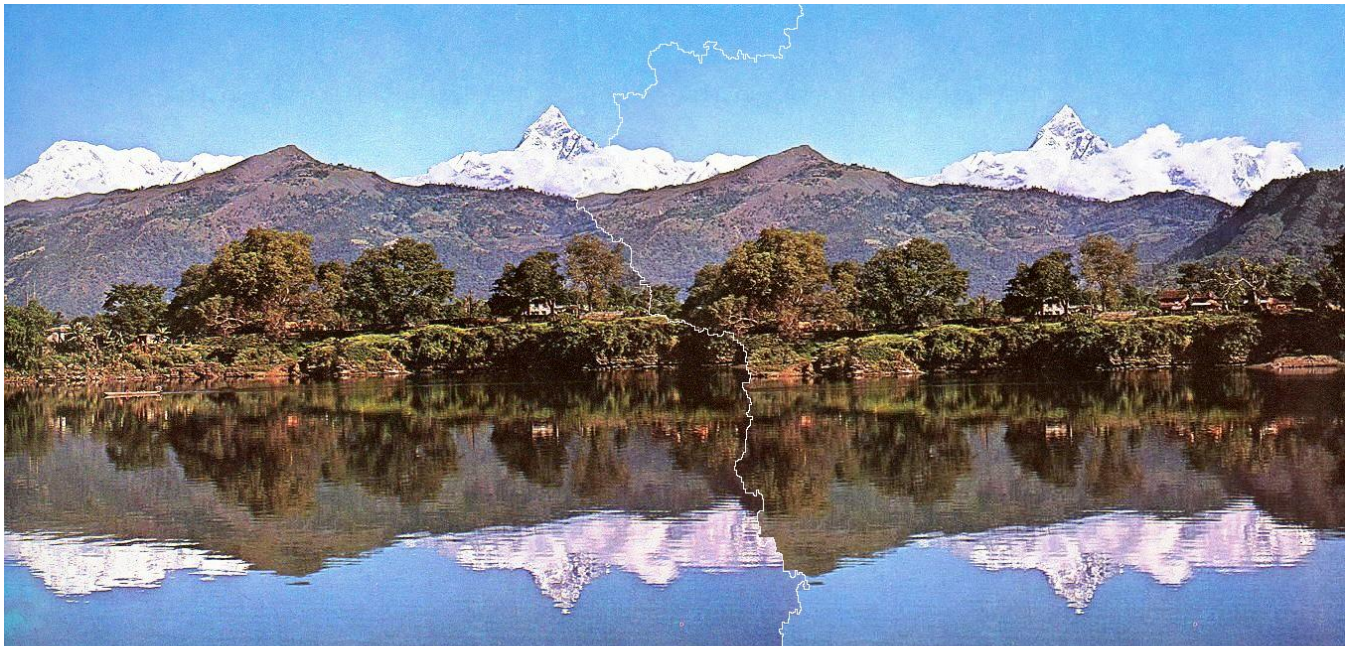


Le recollement



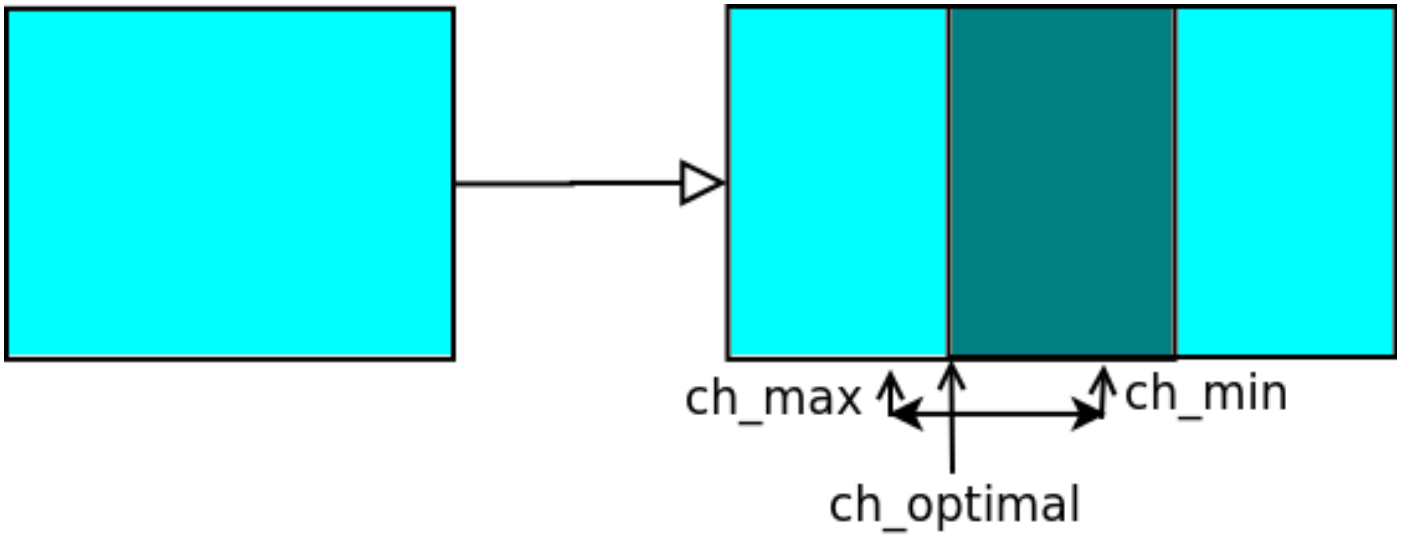
Le recollement





Les deux étapes de l'algorithme

Première étape : définir le chevauchement optimal



Seconde étape : définir la frontière du recollement

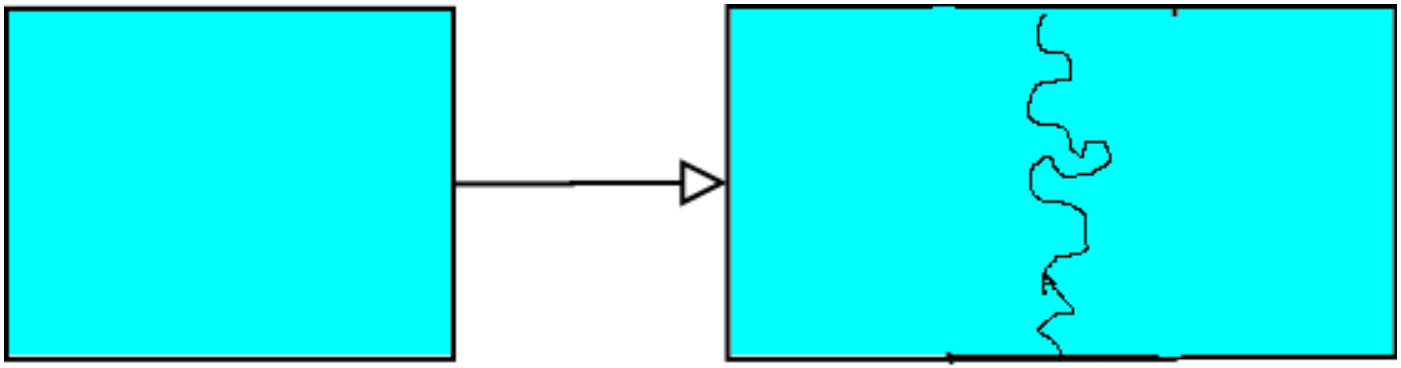


Image d'origine

Voyez–vous le recollement ?

Voyez–vous le recollement ?

Plan

Ouvrir une image en format jpg

```
import numpy as np
import matplotlib.image as mpimg
motif = mpimg.imread ("mon_image.jpg")
```

Ouvrir une image en format jpg

```
import numpy as np
import matplotlib.image as mpimg
motif = mpimg.imread ("mon_image.jpg")
```

Désormais `motif` est un tableau numpy tridimensionnel :

- `motif[i]` désigne la `i`–ième ligne de ce tableau ;
- `motif[i,j]` désigne le pixel à la ligne `i` et à la colonne `j` ;
- - `motif[i,j,0]` désigne la composante rouge,
  - `motif[i,j,1]` la composante verte,
  - `motif[i,j,2]` la composante bleue.

Si on ouvre une image en noir et blanc on obtient un tableau bidimensionnel.

Première difficulté

Composantes : `motif`[*ordonnee*, *abscisse*, *couleur*]

Seconde difficulté

```
p = motif [12, 15]
    np.array ([28 92 155])
```

Seconde difficulté

```
p = motif [12, 15]
    np.array ([28 92 155])
2*p     # multiplie chaque composante par 2
```

Seconde difficulté

```
p = motif [12, 15]
    np.array ([28 92 155])
2*p     # multiplie chaque composante par 2
    np.array ([56 184 54])
```

Seconde difficulté

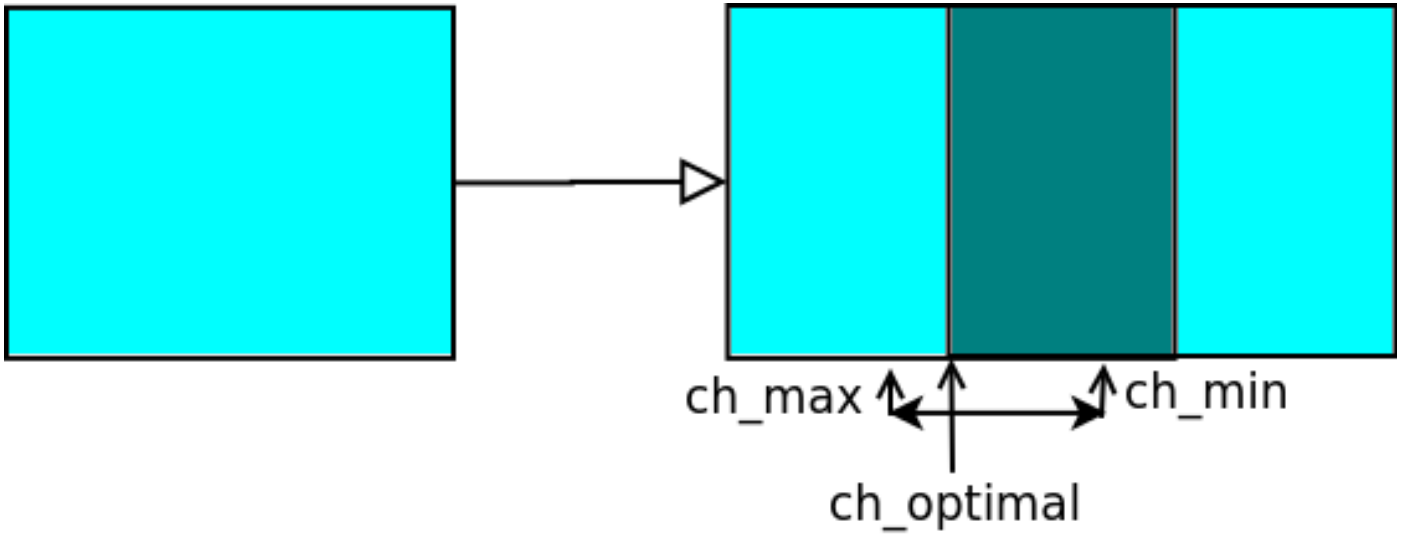
```
p = motif [12, 15]
    np.array ([28 92 155])
2*p     # multiplie chaque composante par 2
    np.array ([56 184 54])
type (p)
    np.ndarray (dtype = uint8)
```

Chaque composante est codée sur 8 bits, de 0 à 255

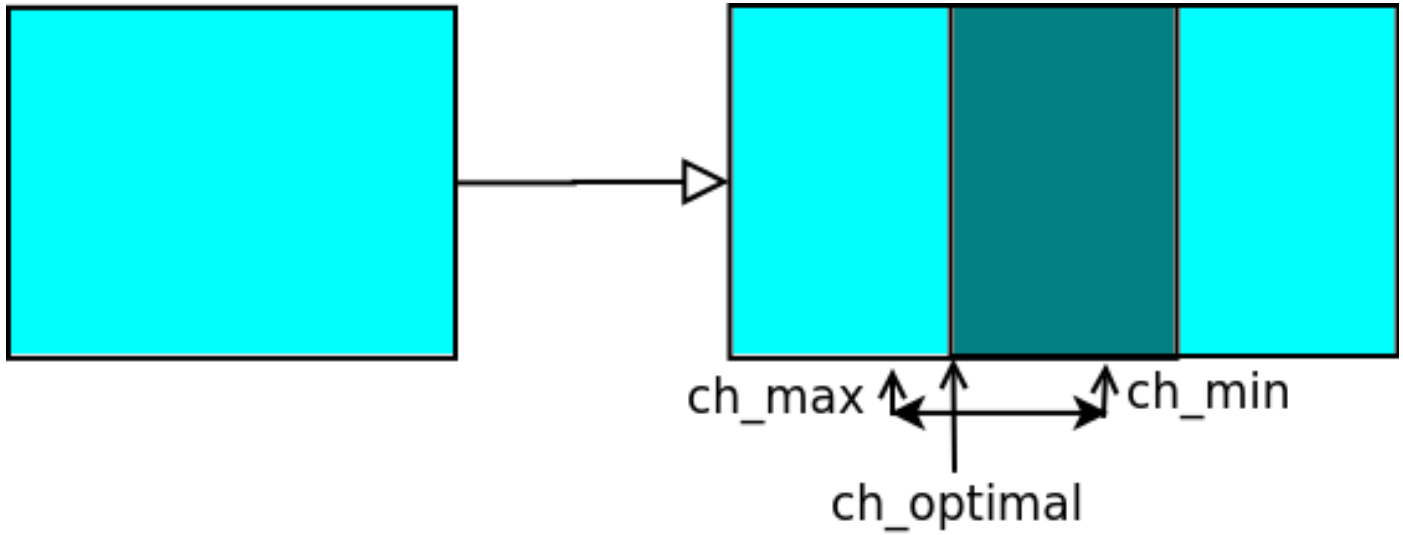
```
mc = np.array (motif, dtype = int)
```

Plan

L'importance de définir un chevauchement optimal



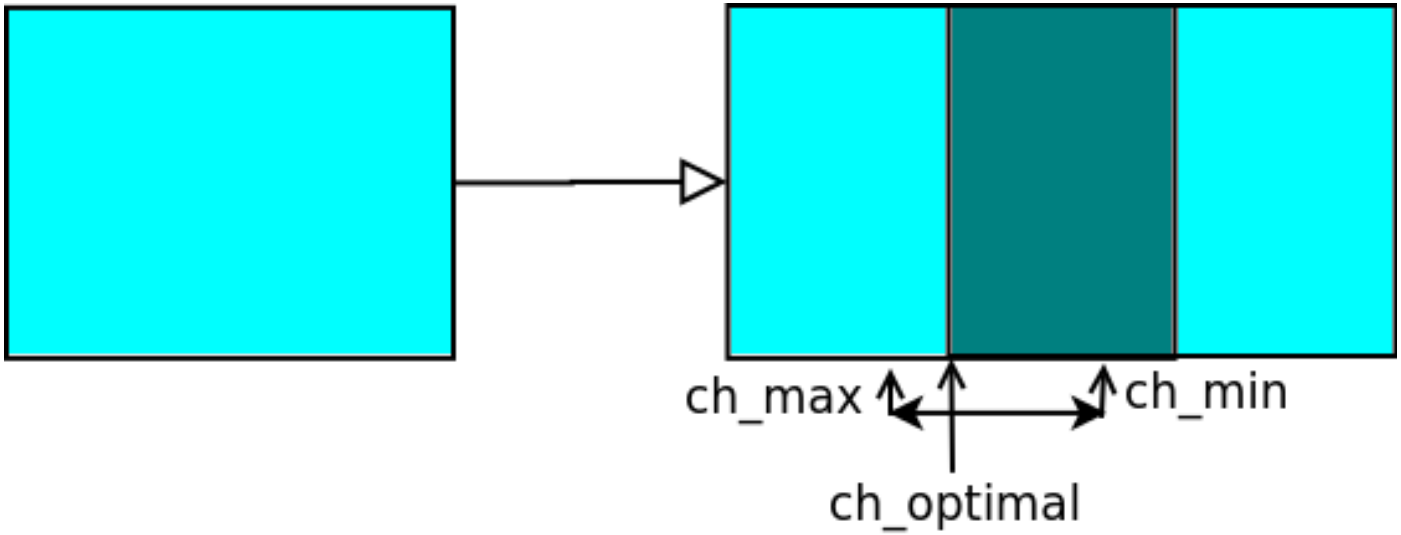
L'importance de définir un chevauchement optimal



Illustrons l'importance d'optimiser le chevauchement :

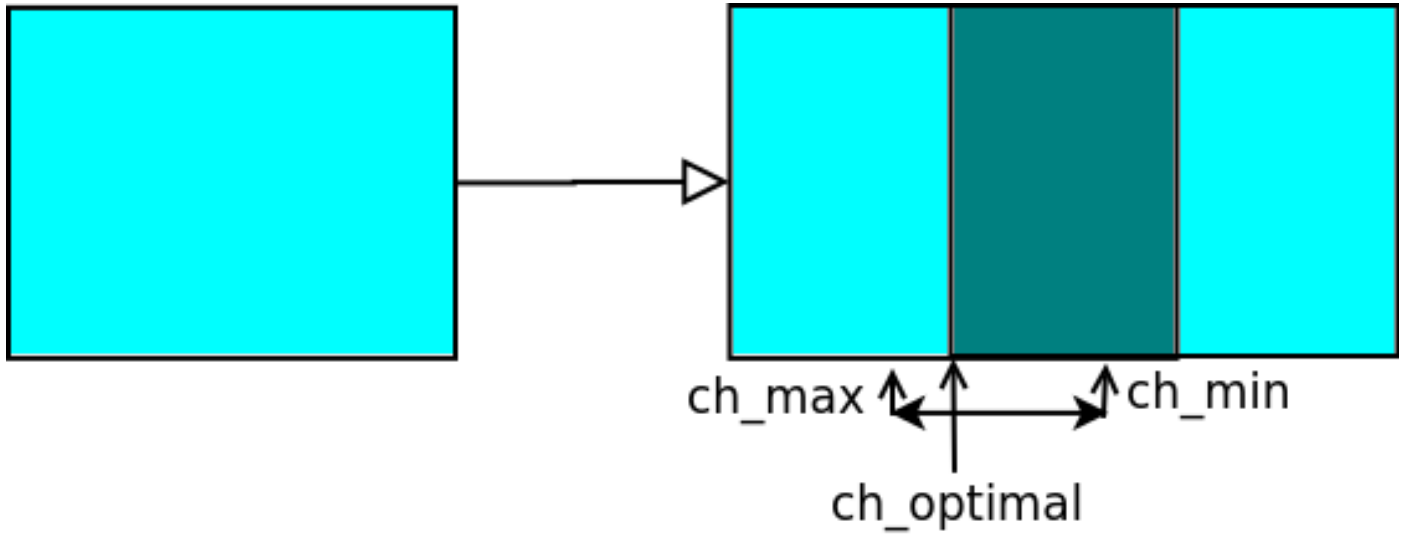
Avec un chevauchement optimal :

Avec un chevauchement quelconque :



On choisit un chevauchement $ch \in [ch_{min}, ch_{max}]$

Il y a chevauvement des pixels $mc[i, j]$ et $mc[i, largeur - ch + j]$ pour tous $i \in [\![0, hauteur[\![$ et tous $j \in [\![0, ch[\![$.



On choisit un chevauchement $ch \in [ch_{min}, ch_{max}]$

Il y a chevauvement des pixels $mc[i, j]$ et $mc[i, largeur - ch + j]$ pour tous $i \in [\![0, hauteur[\![$ et tous $j \in [\![0, ch[\![$.

Distance entre deux pixels p et p' :

$$d(p, p') = \underbrace{(p[0] - p'[0])^2}_{\text{rouge}} + \underbrace{(p[1] - p'[1])^2}_{\text{vert}} + \underbrace{(p[2] - p'[2])^2}_{\text{bleu}}$$

But : trouver le chevauchement $ch \in [\![ch_{min}, ch_{max}]\!]$ qui minimise

$$\left\{ \frac{1}{hauteur \times ch} \sum_{i=0}^{hauteur-1} \sum_{j=0}^{ch-1} d\Big(mc[i,j], mc[i, largeur - ch + j]\Big) \right\}$$

But : trouver le chevauchement $ch \in [\![ch_{min}, ch_{max}]\!]$ qui minimise

$$\left\{ \frac{1}{hauteur \times ch} \sum_{i=0}^{hauteur-1} \sum_{j=0}^{ch-1} d\Big(mc[i,j], mc[i, largeur - ch + j]\Big) \right\}$$

On suppose $ch_{min} \ll ch_{max}$

Complexité : $\mathcal{O}(hauteur \times ch_{max}^2)$
Avec $hauteur = 600$ et $ch_{max} = 400$:
$hauteur \times ch_{max}^2 = 96\,000\,000$

But : trouver le chevauchement $ch \in [\![ch_{min}, ch_{max}]\!]$ qui minimise

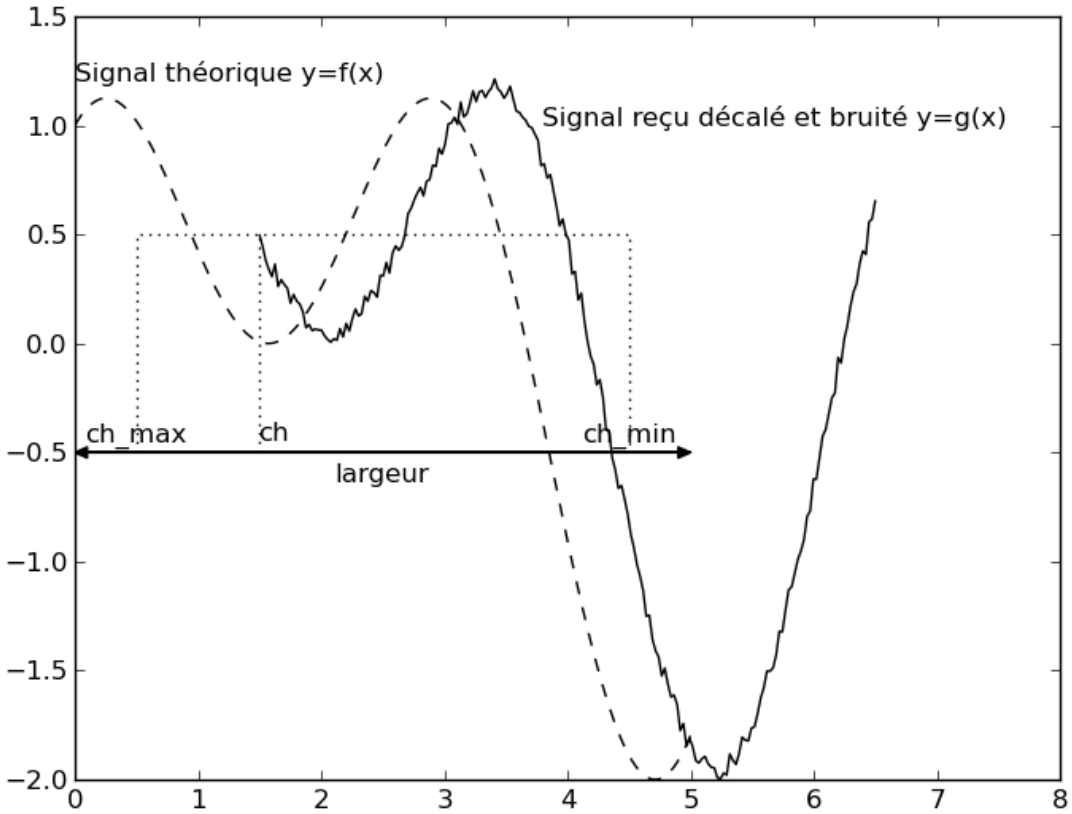
$$\left\{ \frac{1}{hauteur \times ch} \sum_{i=0}^{hauteur-1} \sum_{j=0}^{ch-1} d\Big(mc[i,j], mc[i, largeur - ch + j]\Big) \right\}$$

On suppose $ch_{min} \ll ch_{max}$

Complexité : $\mathcal{O}(hauteur \times ch_{max}^2)$
Avec $hauteur = 600$ et $ch_{max} = 400$:
$hauteur \times ch_{max}^2 = 96\,000\,000$

Il existe un algorithme de complexité
$\mathcal{O}(hauteur \times ch_{max} \times \log(ch_{max}))$ qui utilise la transformée de Fourier.

Le problème monodimensionnel



But : trouver ch qui minimise $\frac{1}{ch} \sum_{k=0}^{ch-1} (g(k) - f(largeur - ch + k))^2$

Posons $h(x) = f(\mathit{largeur} - 1 - x)$. Calculer les valeurs

$$\forall ch \in [\![ch_{min}, ch_{max}]\!], \Phi(ch) = \sum_{k=0}^{ch-1} \Big(g(k) - h(ch - 1 - k)\Big)^2$$

Complexité : $\mathcal{O}(ch_{max}^2)$

Posons $h(x) = f(\mathit{largeur} - 1 - x)$. Calculer les valeurs

$$\forall ch \in [\![ch_{min}, ch_{max}]\!], \Phi(ch) = \sum_{k=0}^{ch-1} \Big(g(k) - h(ch - 1 - k)\Big)^2$$

Complexité : $\mathcal{O}(ch_{max}^2)$

$$\Phi(ch) = \sum_{k=0}^{ch-1} g^2(k) + \sum_{k=0}^{ch-1} h^2(k) - 2\sum_{k=0}^{ch-1} g(k)h(ch - 1 - k)$$

Posons $h(x) = f(largeur - 1 - x)$. Calculer les valeurs

$$\forall ch \in [\![ch_{min}, ch_{max}]\!], \Phi(ch) = \sum_{k=0}^{ch-1} \Big(g(k) - h(ch-1-k)\Big)^2$$

Complexité : $\mathcal{O}(ch_{max}^2)$

$$\Phi(ch) = \sum_{k=0}^{ch-1} g^2(k) + \sum_{k=0}^{ch-1} h^2(k) - 2\sum_{k=0}^{ch-1} g(k)h(ch-1-k)$$

En posant $\gamma = ch - 1$:

$$\Phi(ch) = \underbrace{\sum_{k=0}^{\gamma} g^2(k) + \sum_{k=0}^{\gamma} h^2(k)}_{\mathcal{O}(ch_{max}) \text{ pour tous les } ch} \underbrace{-2\sum_{k=0}^{\gamma} g(k)h(\gamma - k)}_{\text{convolution } \mathcal{O}(ch_{max}^2)}$$

Calcul naïf de convolution

$$[g(0), g(1), \ldots, g(p), 0, \ldots, 0] \quad \text{code } G = \sum_{k=0}^{p'-1} g(k) X^k$$

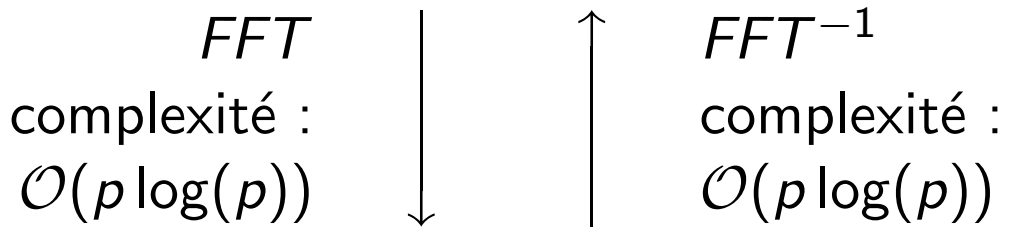
$$[h(0), h(1), \ldots, h(p), 0, \ldots, 0] \quad \text{code } H = \sum_{k=0}^{p'-1} h(k) X^k$$

$$\left[g(0)h(0), \ldots, \sum_{k=0}^{p'-1} g(k)h(p'-1-k) \right] \quad \text{code } GH$$

Coût : $\mathcal{O}(p'^2)$ avec $p' \simeq ch_{max}$

Transformée de Fourier de $G = \sum_{k=0}^{p} g(k)X^k$

$$[g(0), g(1), \ldots, g(p), 0, \ldots, 0] \text{ code } G = \sum_{k=0}^{p'-1} g(k)X^k$$



$$[G(e^{-2i0\pi/p'}), \ldots, G(e^{-2ik\pi/p'}), \ldots, G(e^{-2i(p'-1)\pi/p'})]$$

Accélération d'un calcul de convolution

$$G, H \xrightarrow[\text{convolution}]{\mathcal{O}(p^2)} GH$$

$$FFT \downarrow \text{ complexité : } \mathcal{O}(p\log(p)) \qquad FFT^{-1} \uparrow \text{ complexité : } \mathcal{O}(p\log(p))$$

$$\begin{matrix}\left[G(e^{\frac{-2ik\pi}{p'}})\right]_{0\leqslant k<p'} \\ \left[H(e^{\frac{-2ik\pi}{p'}})\right]_{0\leqslant k<p'}\end{matrix} \xrightarrow[\text{prod. terme à terme}]{\mathcal{O}(p)} \left[G(e^{\frac{-2ik\pi}{p'}})H(e^{\frac{-2ik\pi}{p'}})\right]_{0\leqslant k<p'}$$

Accélération d'un calcul de convolution

$$G, H \xrightarrow[\text{convolution}]{\mathcal{O}(p^2)} GH$$

$$FFT \downarrow \text{ complexité : } \mathcal{O}(p\log(p)) \qquad FFT^{-1} \uparrow \text{ complexité : } \mathcal{O}(p\log(p))$$

$$\left[G(e^{\frac{-2ik\pi}{p'}})\right]_{0\leqslant k<p'} \\ \left[H(e^{\frac{-2ik\pi}{p'}})\right]_{0\leqslant k<p'} \xrightarrow[\text{prod. terme à terme}]{\mathcal{O}(p)} \left[G(e^{\frac{-2ik\pi}{p'}})H(e^{\frac{-2ik\pi}{p'}})\right]_{0\leqslant k<p'}$$

Calcul de la convolution par FFT : $\mathcal{O}(p\log p)$

Une portion de code

Les tableaux sont tridimensionnels

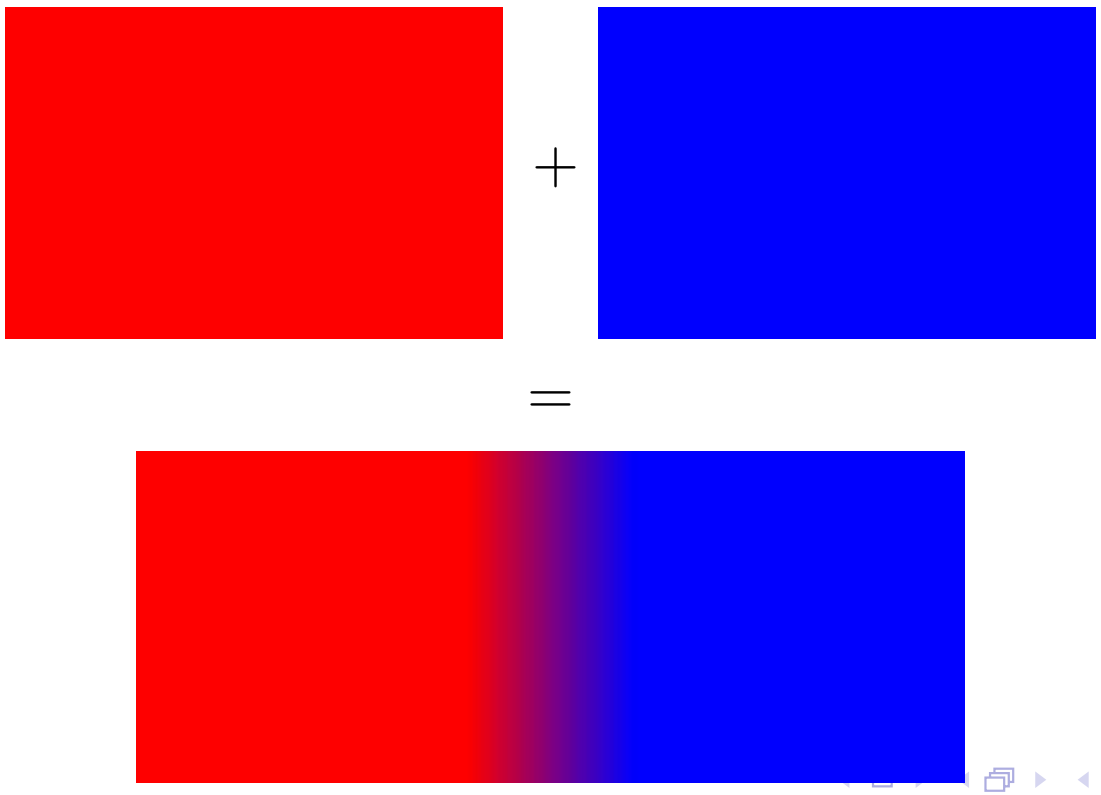
```
import numpy as np
import numpy.fft as nfft

...

fftg = nfft.fft (nzg, axis=1) # fft de la zone gauche

fftd = nfft.fft (nzd, axis=1) # fft de la zone droite

convole = np.sum (np.real (nfft.ifft (\
                fftg*fftd, axis=1)), axis = (0,2))
```

Plan

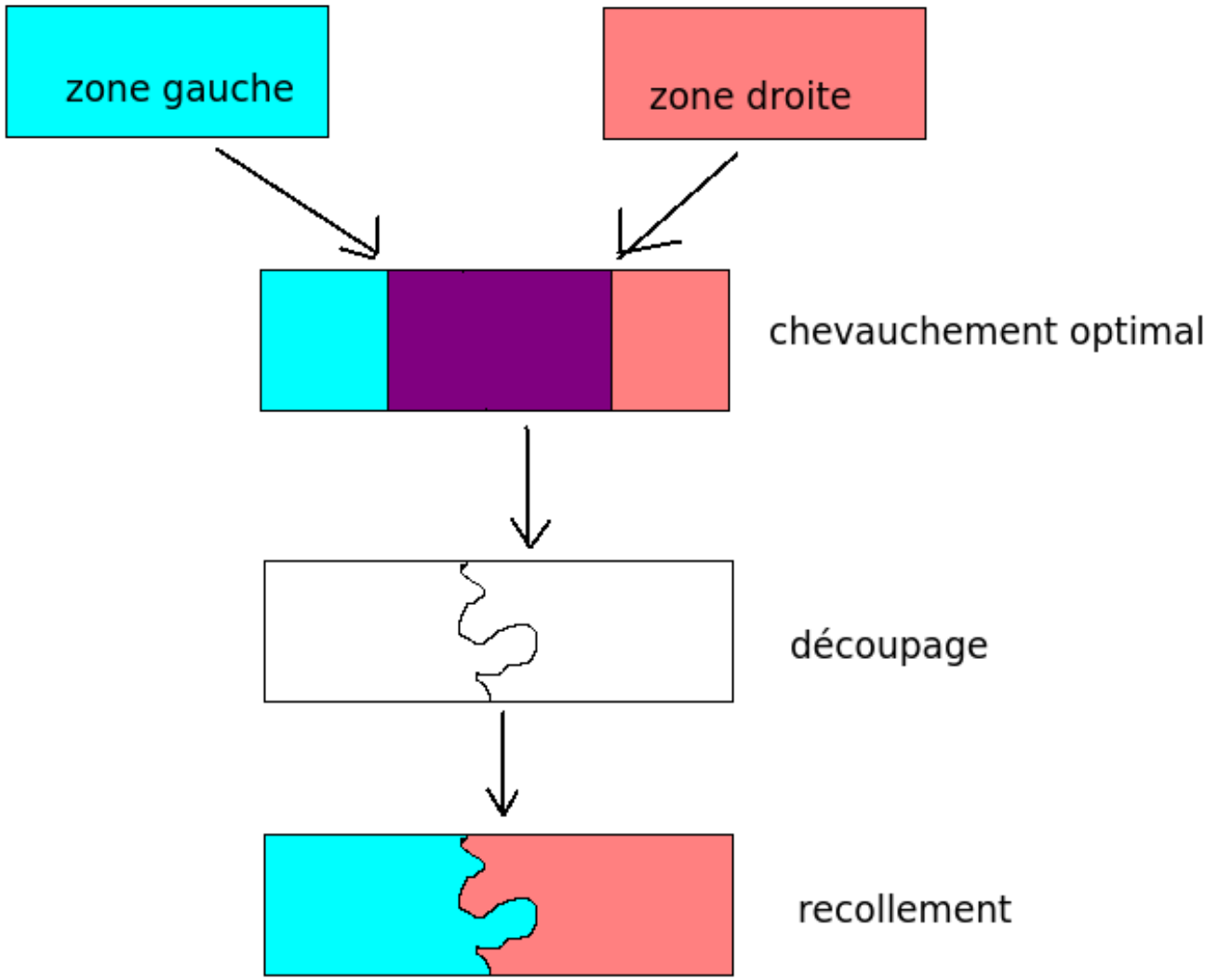
Fondu sur la zone de chevauchement

Approche simpliste : sur la zone de chevauchement, à l'abscisse $x \in [\![0, ch[\![$, on effectue une moyenne barycentique (affine par rapport à x) du pixel de la zone gauche $g[x, y]$ et du pixel de la zone droite $d[x, y]$

$$m[x, y] = g[x, y]\frac{ch - x}{ch} + d[x, y]\frac{x}{ch}$$



Les phases de l'algorithme



Coût d'un algorithme

La zone de chevauchement possède $N_p \simeq 10^5$ pixels.

Coût d'un algorithme

La zone de chevauchement possède $N_p \simeq 10^5$ pixels.

On dispose d'un algorithme A de complexité $\mathcal{O}(N_p \log(N_p))$ (quasilinéaire)
et d'un algorithme B de complexité $\mathcal{O}(N_p^2)$ (quadratique).

Coût d'un algorithme

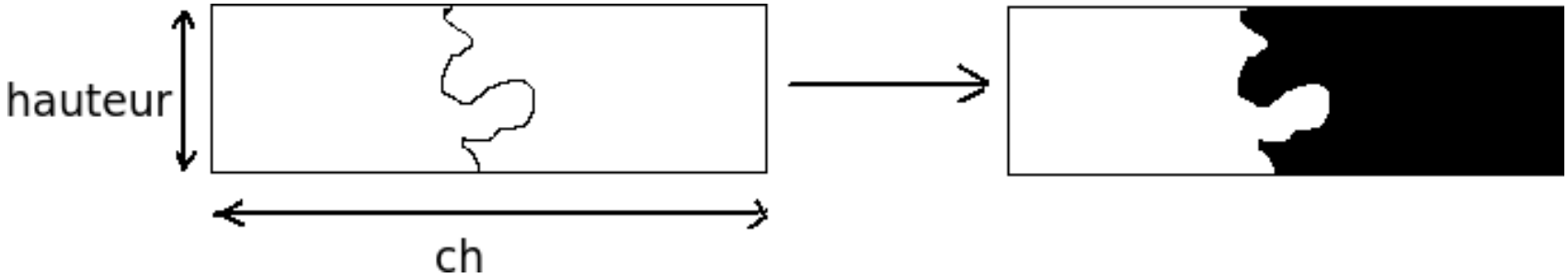
La zone de chevauchement possède $N_p \simeq 10^5$ pixels.

On dispose d'un algorithme A de complexité $\mathcal{O}(N_p \log(N_p))$ (quasilinéaire)
et d'un algorithme B de complexité $\mathcal{O}(N_p^2)$ (quadratique).

$$\frac{N_p^2}{N_p \log_2(N_p)} \simeq 6\,000$$

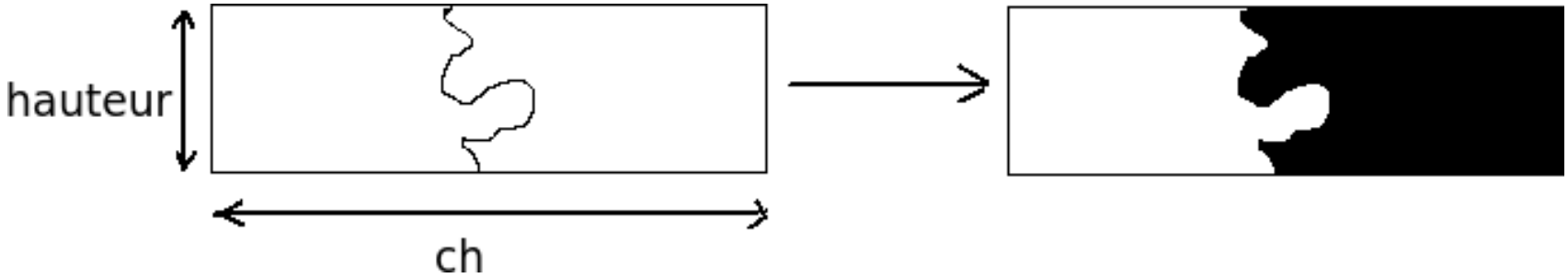
Si le temps d'exécution de l'algorithme A se mesure en secondes, celui de l'algorithme B se mesure en heures.

Remplir une figure sans dépasser une ligne noire



Algorithme récursif de remplissage sans dépasser la ligne noire tracée.

Remplir une figure sans dépasser une ligne noire



Algorithme récursif de remplissage sans dépasser la ligne noire tracée.

Appel initial : (x, y) dans la zone droite
Il n'y a pas d'appel récursif si $p(x, y)$ est déjà noir.

```
Remplir (x, y) :
      Si p(x, y) blanc :
          Noircir p(x, y)
          Si x < ch − 1 :  Remplir (x + 1, y)
          Si x > 0 :  Remplir (x − 1, y)
          Si y < hauteur − 1 :  Remplir (x, y + 1)
          Si y > 0 :  Remplir (x, y − 1)
```

Qu'est–ce qu'un "bon chemin" ?

Pour le pixel (x, y) avec chevauchement d'un pixel p_g de la zone gauche et d'un pixel p_d de la zone droite, on définit son poids $\pi(x, y) = d(p_g, p_d)$.

But : trouver un chemin entre un point d'ordonnée 0 et un point d'ordonnée $hauteur - 1$, tel que la moyenne des poids des pixels rencontrés lors du parcours soit la plus petite possible.

Chercher un chemin de longueur moyenne minimum

Le nombre N_p de pixels de la zone de chevauchement est de l'ordre de 10^5
Solution approchée : recherche avec heuristique
Coût : $\mathcal{O}(N_p \log(N_p))$ au pire

Chercher un chemin de longueur moyenne minimum

Le nombre N_p de pixels de la zone de chevauchement est de l'ordre de 10^5
Solution approchée : recherche avec heuristique
Coût : $\mathcal{O}(N_p \log(N_p))$ au pire

Recherche d'un chemin de coût moyen minimum entre deux pixels donnés en au plus N_p pixels :
Programmation dynamique, solution en $\mathcal{O}(N_p^2)$

Chercher un chemin de longueur moyenne minimum

Le nombre N_p de pixels de la zone de chevauchement est de l'ordre de 10^5
Solution approchée : recherche avec heuristique
Coût : $\mathcal{O}(N_p \log(N_p))$ au pire

Recherche d'un chemin de coût moyen minimum entre deux pixels donnés en au plus N_p pixels :
Programmation dynamique, solution en $\mathcal{O}(N_p^2)$
Problème optimal mal posé : si on exclut les cycles le problème est NP–complet

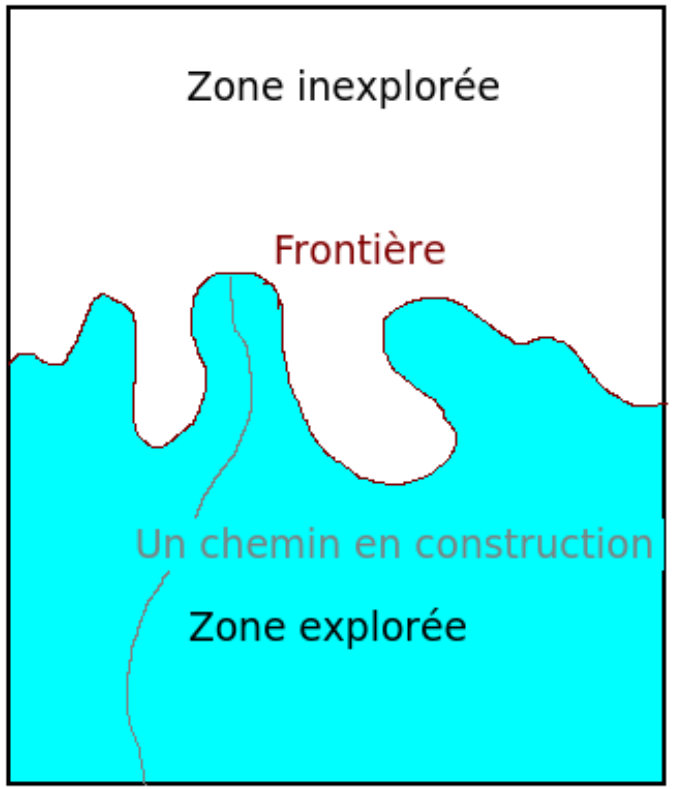
Chercher un chemin de longueur moyenne minimum

Le nombre N_p de pixels de la zone de chevauchement est de l'ordre de 10^5
Solution approchée : recherche avec heuristique
Coût : $\mathcal{O}(N_p \log(N_p))$ au pire

Recherche d'un chemin de coût moyen minimum entre deux pixels donnés en au plus N_p pixels :
Programmation dynamique, solution en $\mathcal{O}(N_p^2)$
Problème optimal mal posé : si on exclut les cycles le problème est NP–complet

Dans cet exposé on privilégie la solution approchée.

Lancer une exploration avec heuristique



Algorithme similaire à celui de Dijkstra.
Il y a trois types de pixels : les pixels déjà entièrement explorés, les pixels à la frontière de l'exploration et les pixels inexplorés.

Structures auxiliaires utilisées

- *Antecedent* : une matrice de dimensions *hauteur* $\times$ *ch* pour savoir si le pixel (i, j) est déjà exploré, et si oui, quel est le pixel voisin par lequel on l'a atteint ;

Structures auxiliaires utilisées

- *Antecedent* : une matrice de dimensions *hauteur* $\times$ *ch* pour savoir si le pixel (i, j) est déjà exploré, et si oui, quel est le pixel voisin par lequel on l'a atteint ;
- *Frontiere* : la liste des pixels $(x, y, \Sigma_{poids}, nbre, ante)$ à la frontière de l'exploration, avec pour chaque pixel le coût Σ_{poids} du chemin pour y arriver, *nbre* la longueur du chemin et *ante* son antecedent.

Structures auxiliaires utilisées

- *Antecedent* : une matrice de dimensions $hauteur \times ch$ pour savoir si le pixel (i, j) est déjà exploré, et si oui, quel est le pixel voisin par lequel on l'a atteint ;
- *Frontiere* : la liste des pixels $(x, y, \Sigma_{poids}, nbre, ante)$ à la frontière de l'exploration, avec pour chaque pixel le coût Σ_{poids} du chemin pour y arriver, *nbre* la longueur du chemin et *ante* son antecedent.
  On doit pouvoir extraire de *frontiere* le meilleur pixel ($\frac{\Sigma_{poids}}{nbre}$ minimum).

Structures auxiliaires utilisées

- *Antecedent* : une matrice de dimensions $hauteur \times ch$ pour savoir si le pixel (i, j) est déjà exploré, et si oui, quel est le pixel voisin par lequel on l'a atteint ;
- *Frontiere* : la liste des pixels $(x, y, \Sigma_{poids}, nbre, ante)$ à la frontière de l'exploration, avec pour chaque pixel le coût Σ_{poids} du chemin pour y arriver, *nbre* la longueur du chemin et *ante* son antecedent.
  On doit pouvoir extraire de *frontiere* le meilleur pixel ($\frac{\Sigma_{poids}}{nbre}$ minimum).

  IPT : utiliser une liste non triée avec calcul du minimum ; ou une liste triée avec insertion en respectant l'ordre.

  Option informatique : utilisation d'une file de priorité avec un tas.

Principe de l'algorithme

Mettre dans `Frontiere` tous les points d'ordonnée $y = 0$

$y \leftarrow 0$

Principe de l'algorithme

Mettre dans `Frontiere` tous les points d'ordonnée $y = 0$
$y \leftarrow 0$

Tant que $y < hauteur - 1$ **Faire**
    Retirer meilleur pixel $(x, y, \Sigma_p, nbre, ante)$ de `Frontiere`
    Si (x, y) inexploré :
        $Antecedent(x, y) \leftarrow ante$
        Si $x > 0$: Rajouter dans `Frontiere`
            $(x - 1, y, \Sigma_p + d_{x-1,y}, nbre + 1, (x, y))$
        Idem pour voisins droit, haut, bas
Fin tant que

Principe de l'algorithme

Mettre dans `Frontiere` tous les points d'ordonnée $y = 0$
$y \leftarrow 0$

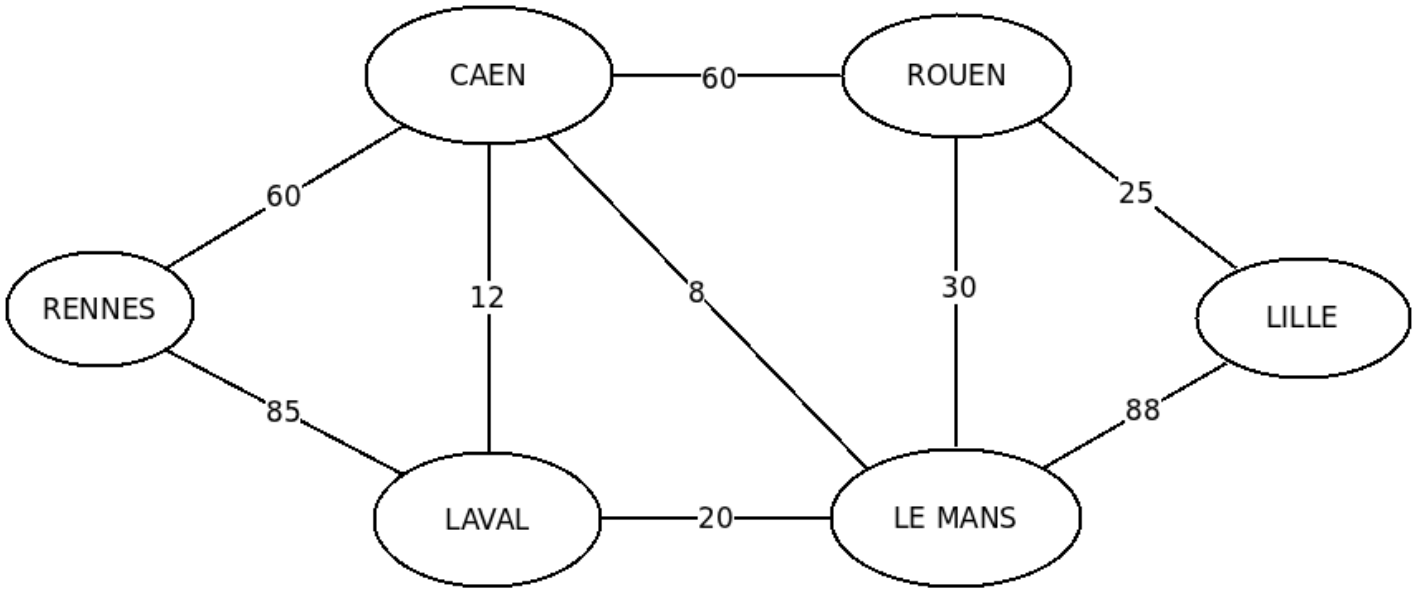
Tant que $y < hauteur - 1$ **Faire**
    Retirer meilleur pixel $(x, y, \Sigma_p, nbre, ante)$ de `Frontiere`
    Si (x, y) inexploré :
        $Antecedent(x, y) \leftarrow ante$
        Si $x > 0$: Rajouter dans `Frontiere`
            $(x - 1, y, \Sigma_p + d_{x-1,y}, nbre + 1, (x, y))$
        Idem pour voisins droit, haut, bas
Fin tant que
Tant que $y > 0$:
    Marquer (x, y)
    $(x, y) \leftarrow$ `Antecedent`(x, y)
Fin tant que

Un problème d'eau dans des tuyaux

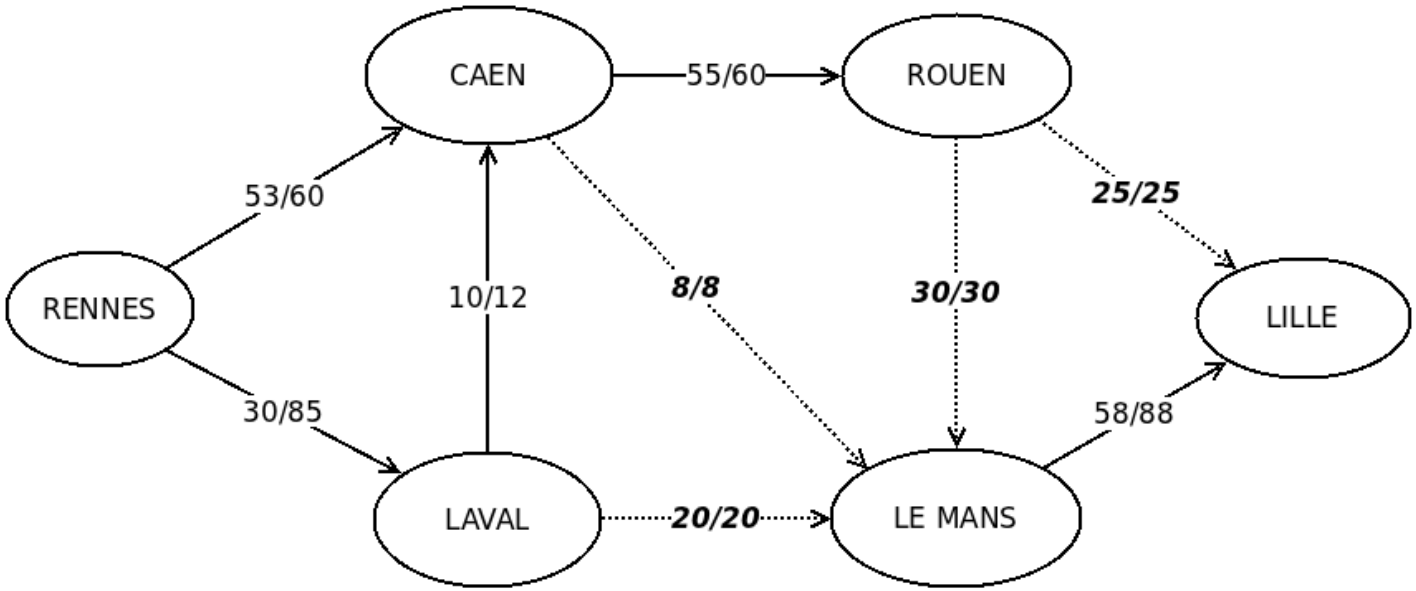
Faire parvenir un maximum d'eau de Rennes (la source) vers Lille (la destination) en passant par un réseau de villes comprenant Caen, Rouen, Laval et Le Mans sous les contraintes suivantes :

- la source ne peut qu'émettre de l'eau ;
- la destination ne peut que recevoir de l'eau ;
- La capacité des tuyaux est un entier positif ;
- le débit rentrant dans chaque ville doit être égal au flot sortant (sauf pour la source et la destination).

Graphe du réseau avec une solution

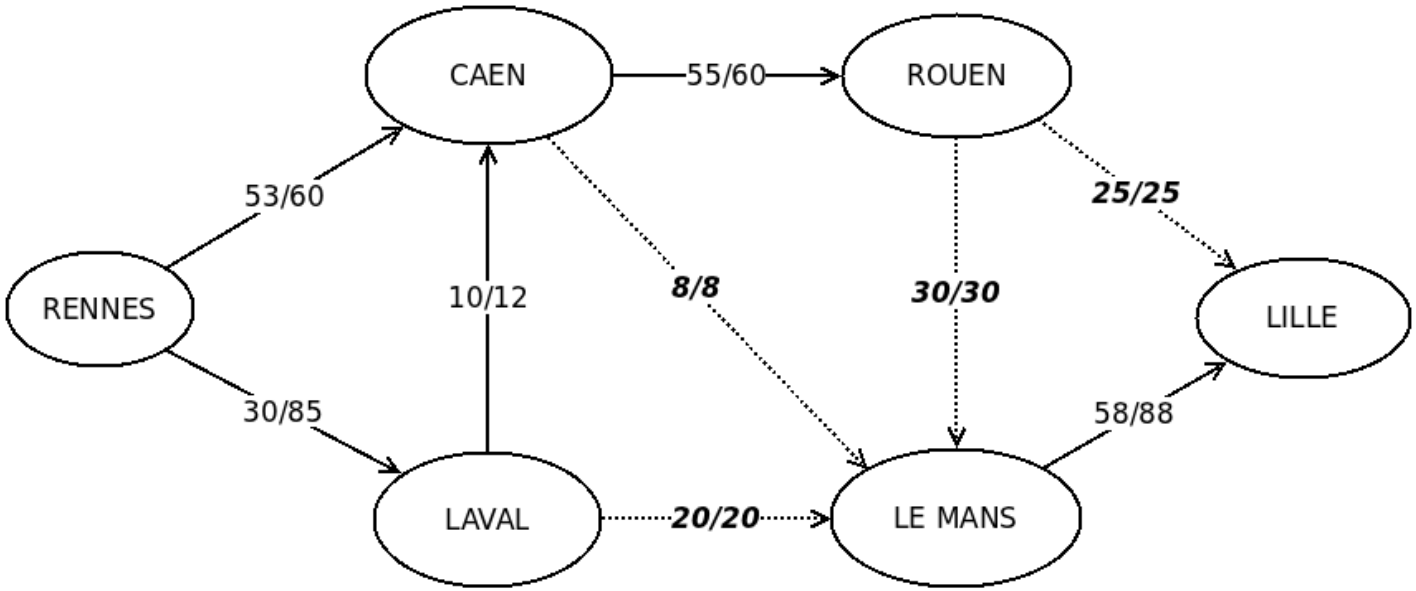


Graphe du réseau avec une solution



Le flot maximum vaut 83.

Graphe du réseau avec une solution



Le flot maximum vaut 83.

Les tuyaux saturés scindent l'ensemble des villes en deux parties.

Coupure

Dans un réseau de distribution d'eau entre plusieurs villes, avec des débits donnés dans chaque tuyau, une coupure est la donnée de deux parties A et B de l'ensemble des villes telles que :

- la source est dans A ;
- la destination est dans B ;
- A et B sont disjointes (aucune ville commune) ;
- toute ville appartient ou bien à A ou bien à B ;
- pour toutes villes $v_A \in A$ et $v_B \in B$, s'il existe un tuyau de A vers B, alors il est saturé dans le sens $A \to B$.

Un théorème

S'il existe une coupure en deux parties A et B de l'ensemble des villes, alors la somme μ des débits de tous les tuyaux liant les villes de A aux villes de B est égale à la quantité d'eau émise par la source et à la quantité d'eau reçue par la destination.

Un théorème

S'il existe une coupure en deux parties A et B de l'ensemble des villes, alors la somme μ des débits de tous les tuyaux liant les villes de A aux villes de B est égale à la quantité d'eau émise par la source et à la quantité d'eau reçue par la destination.

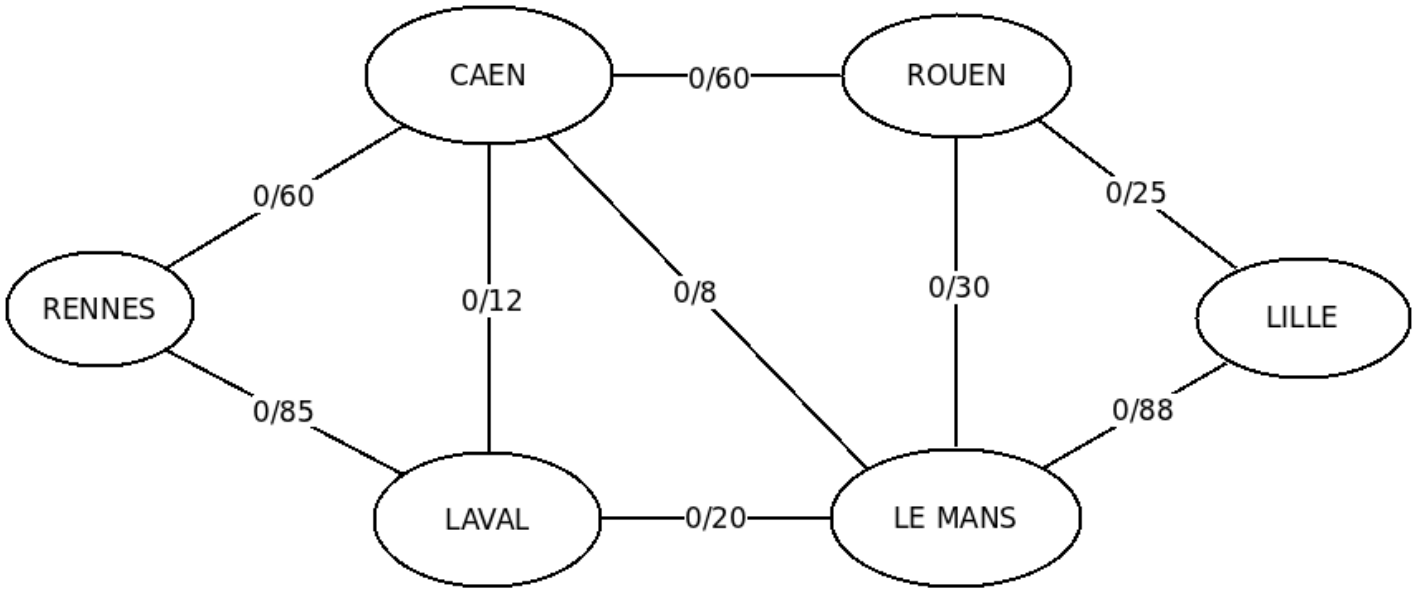
Par ailleurs μ est le débit maximum transférable de la source vers la destination : si on change le débit dans les tuyaux, la quantité reçue par la destination sera inférieure ou égale à μ.

Algorithme de Ford–Fulkerson

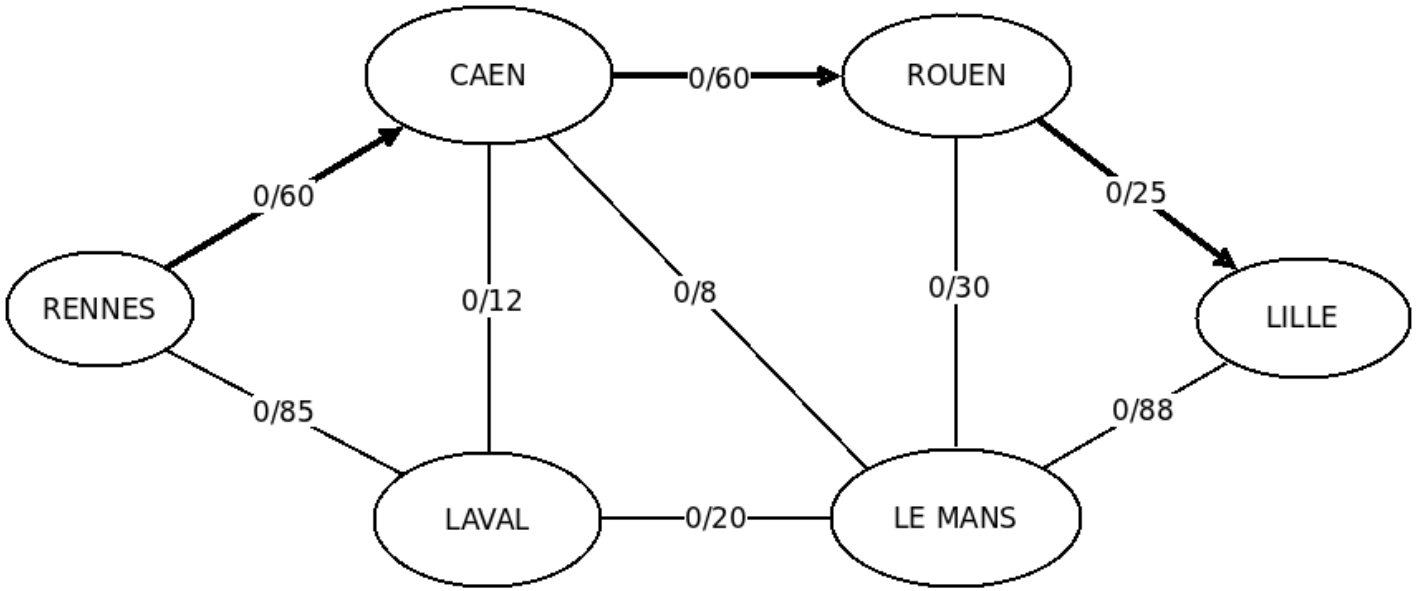
Soient deux villes v et v'. S'il y a un débit positif de v vers v' on note $d(v', v) = -d(v, v')(\leqslant 0)$

- Tant que c'est possible :
  - Trouver un chemin (sans boucle)
    $v_0 = \mathit{source} \to v_1 \to \cdots \to v_n = \mathit{destination}$ tel que pour tout $i \in \{1, 2, \ldots, n\}$, le débit $d(v_{i-1}, v_i) : v_{i-1} \to v_i$ ne soit pas égal à la capacité $c(v_{i-1}, v_i) : v_{i-1} \to v_i$.
  - Calculer $\delta = \min \left\{ c(v_{i-1}, v_i) - d(v_{i-1}, v_i) \,\middle|\, i \in \{1, \ldots, n\} \right\}$
  - Rajouter δ à tous les débits $d(v_{i-1}, v_i) : v_{i-1} \to v_i$

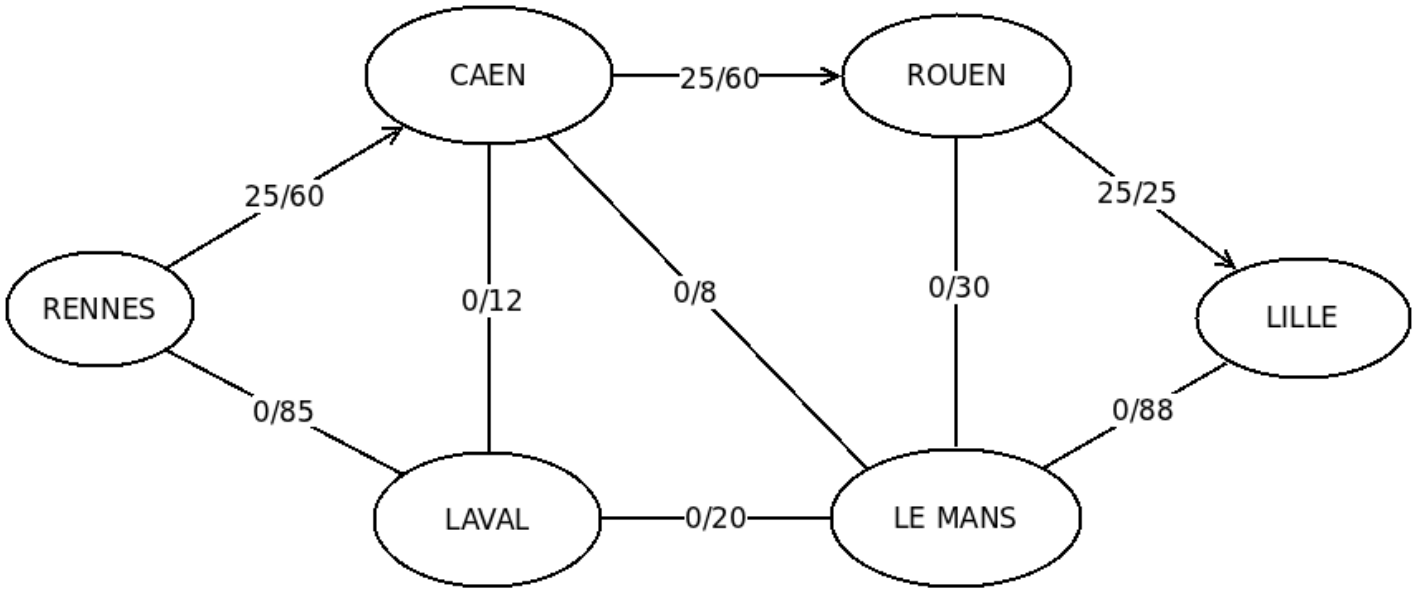
Ford–Fulkerson sur notre exemple



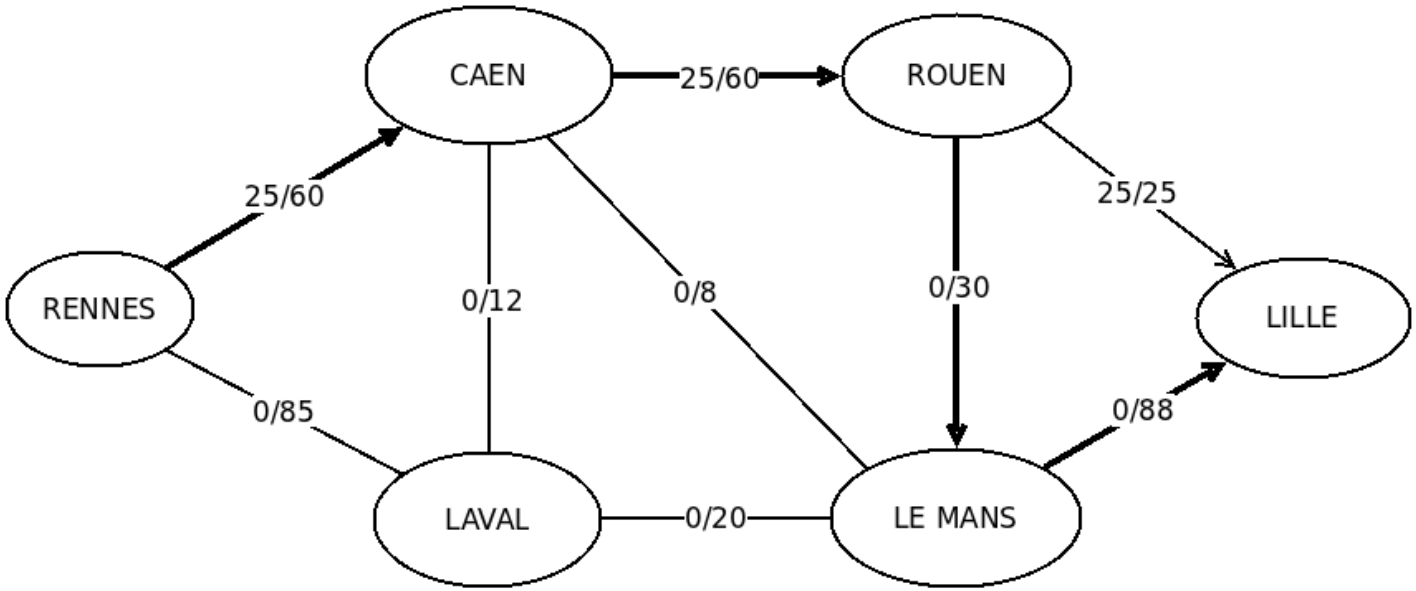
Ford–Fulkerson sur notre exemple



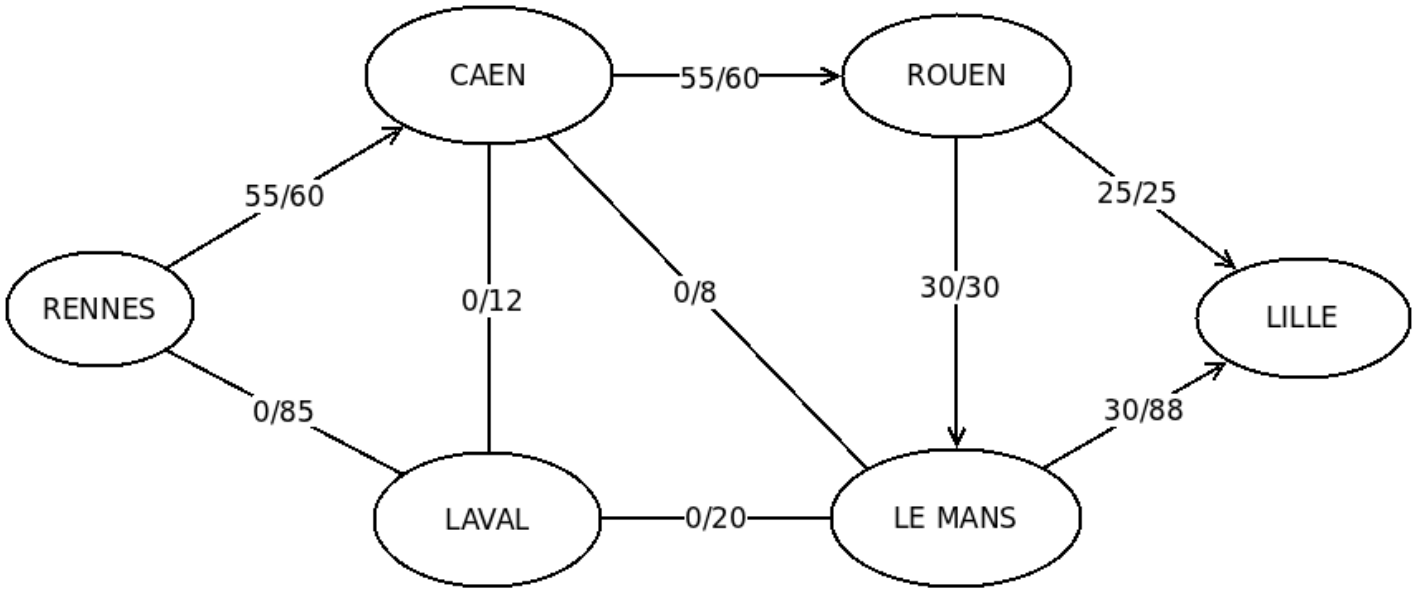
Ford–Fulkerson sur notre exemple



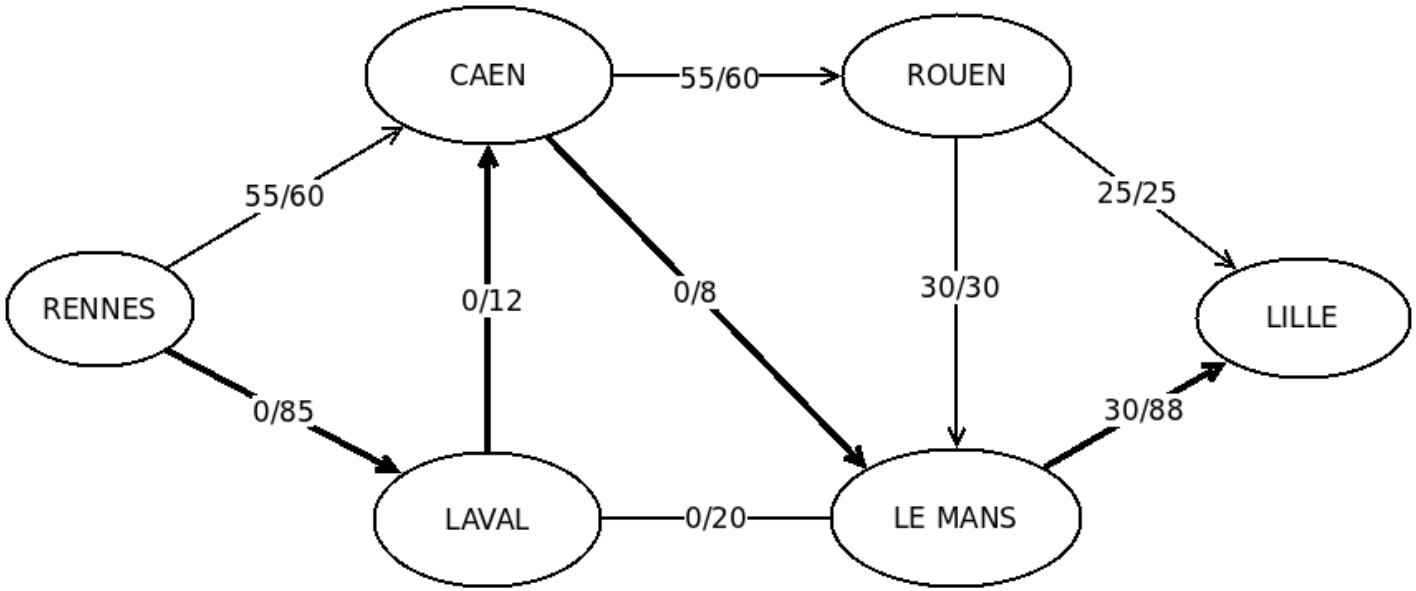
Ford–Fulkerson sur notre exemple



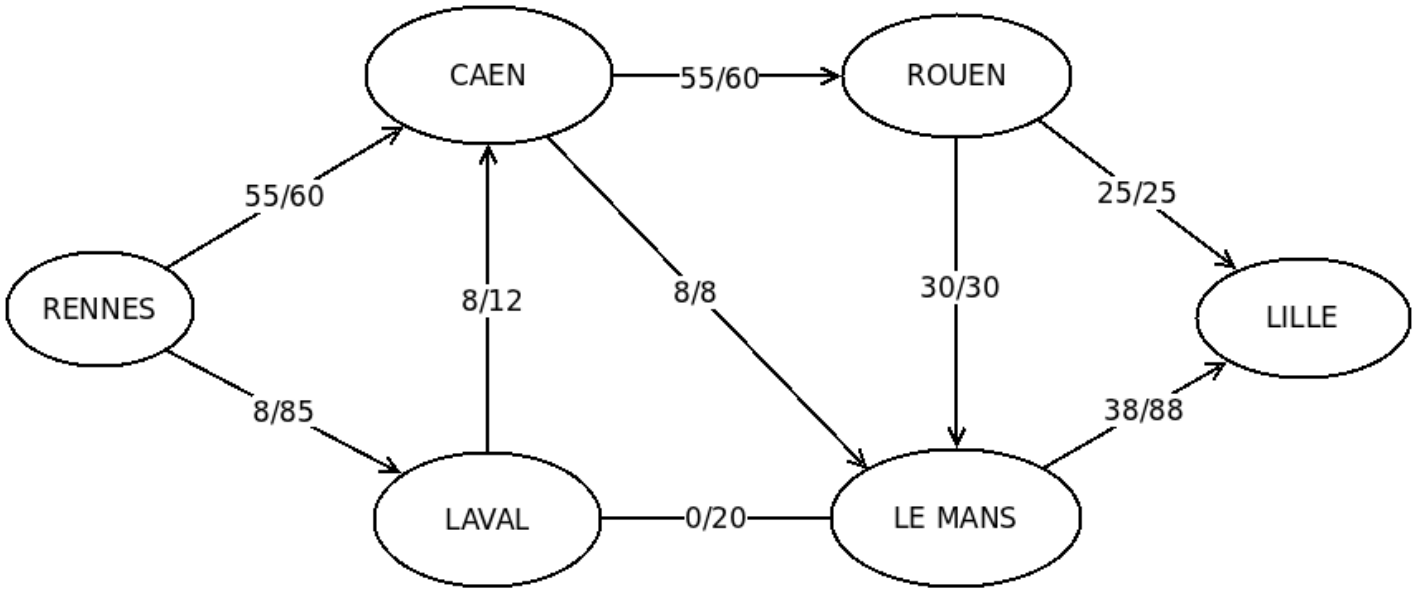
Ford–Fulkerson sur notre exemple



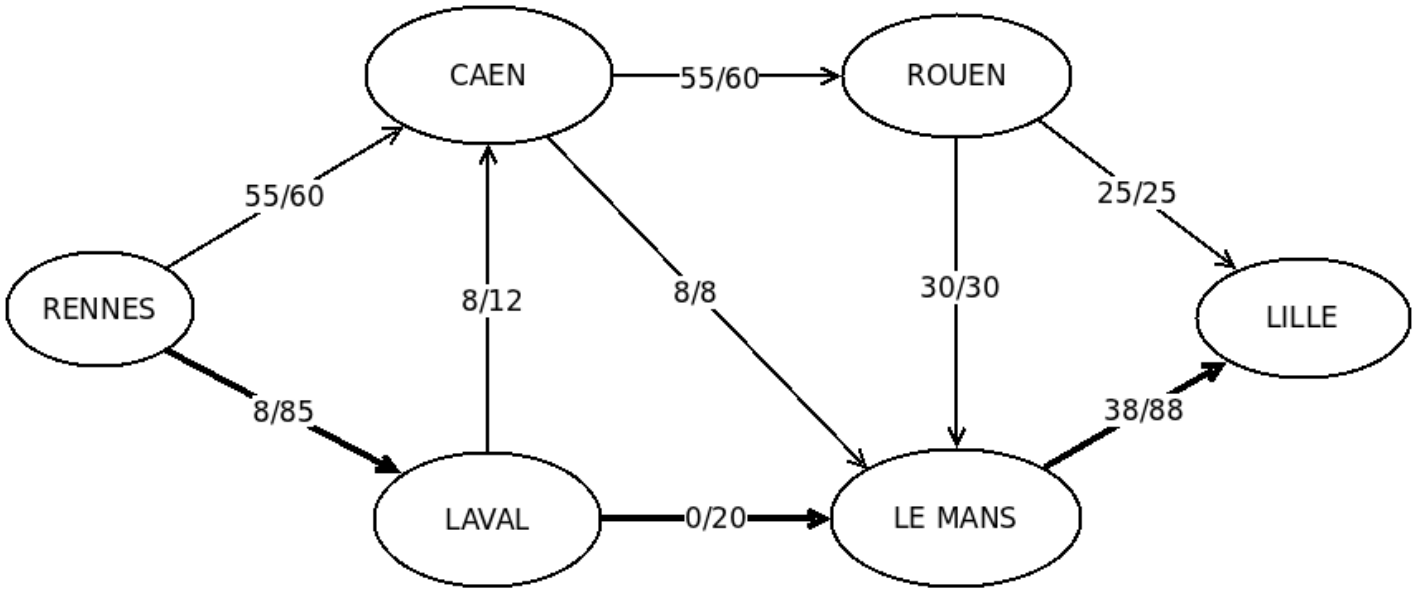
Ford–Fulkerson sur notre exemple



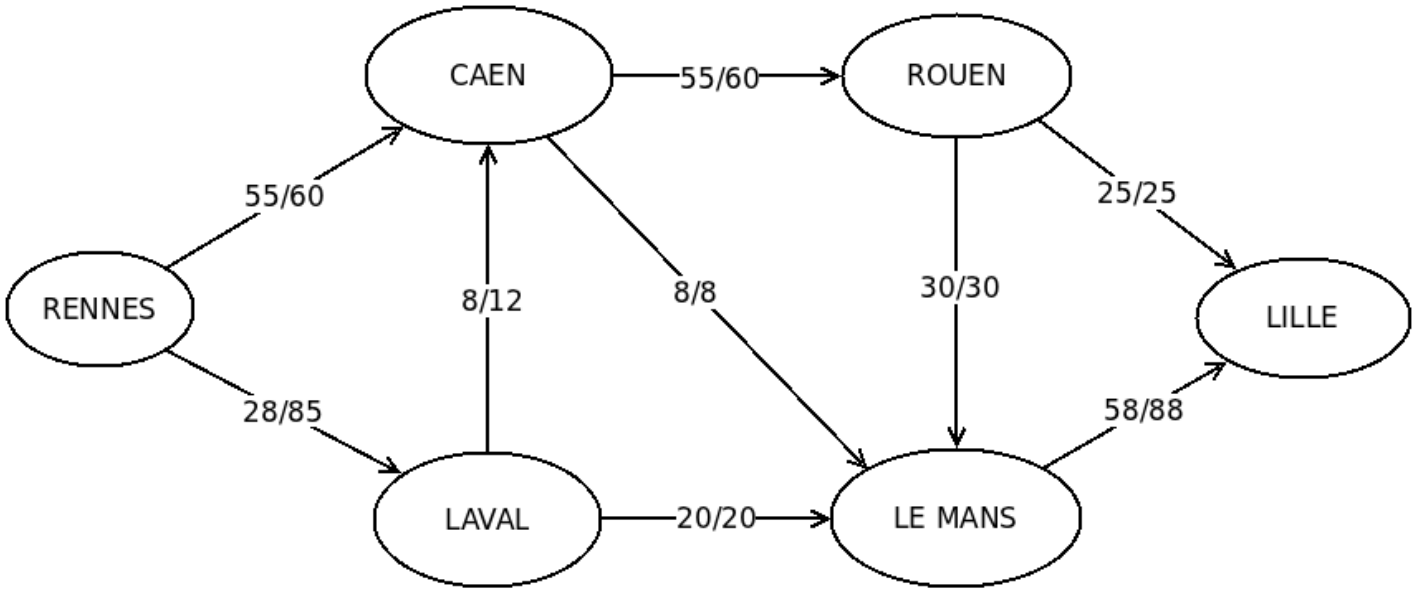
Ford–Fulkerson sur notre exemple



Ford–Fulkerson sur notre exemple

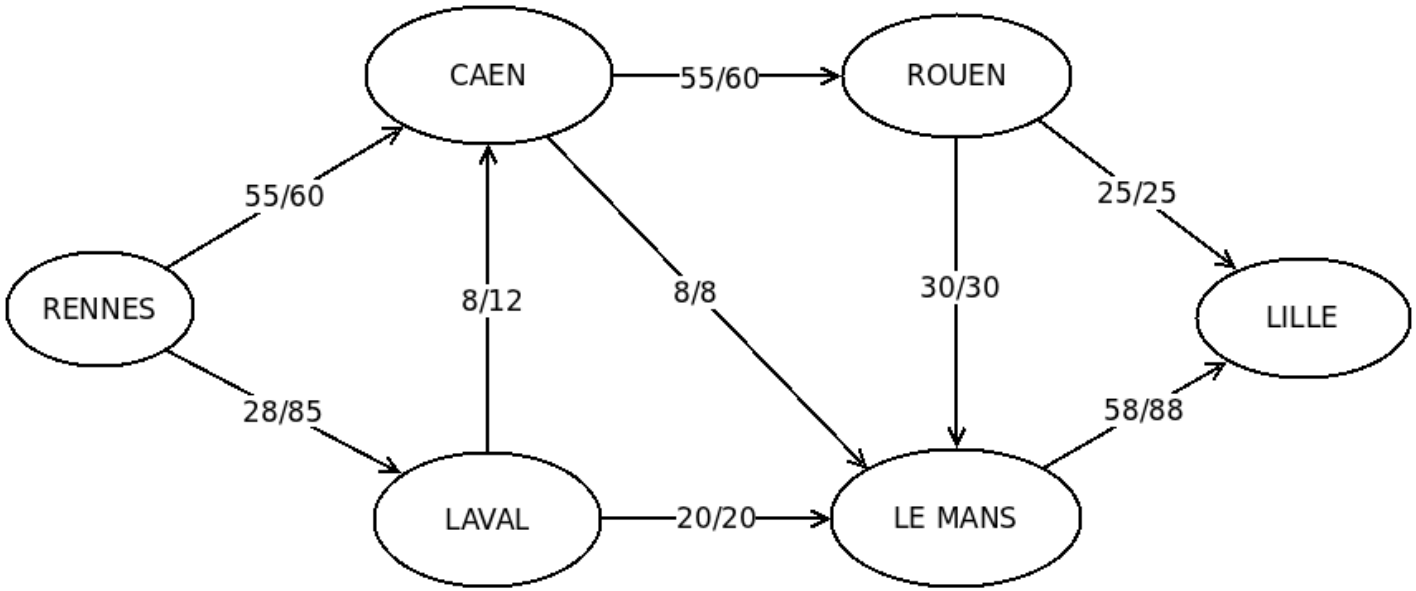


Ford–Fulkerson sur notre exemple



Ford–Fulkerson sur notre exemple

$\mu = 83$



Un second théorème

Il y a arrêt car le débit total issu de la source augmente à chaque itération, et les débits sont entiers.
Lorsque l'algorithme de Ford–Fulkerson s'arrête, une coupure est réalisée.

Un second théorème

Il y a arrêt car le débit total issu de la source augmente à chaque itération, et les débits sont entiers.
Lorsque l'algorithme de Ford–Fulkerson s'arrête, une coupure est réalisée.

Idée de la preuve : à chaque stade du programme, soit A la composante connexe contenant la source dans le graphe où on efface les tuyaux satués.

Un second théorème

Il y a arrêt car le débit total issu de la source augmente à chaque itération, et les débits sont entiers.
Lorsque l'algorithme de Ford–Fulkerson s'arrête, une coupure est réalisée.

Idée de la preuve : à chaque stade du programme, soit A la composante connexe contenant la source dans le graphe où on efface les tuyaux satués.
On peut continuer les itérations si et seulement si la destination appartient à A.

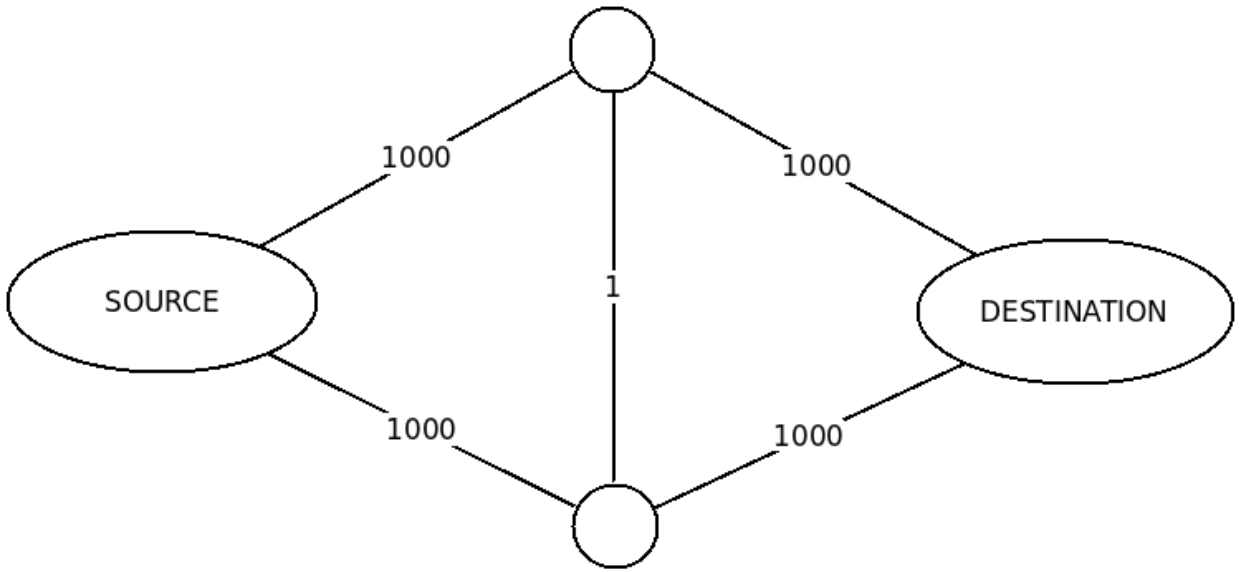
Un second théorème

Il y a arrêt car le débit total issu de la source augmente à chaque itération, et les débits sont entiers.
Lorsque l'algorithme de Ford–Fulkerson s'arrête, une coupure est réalisée.

Idée de la preuve : à chaque stade du programme, soit A la composante connexe contenant la source dans le graphe où on efface les tuyaux satués.
On peut continuer les itérations si et seulement si la destination appartient à A.
À la fin du programme, on définit B l'ensemble des villes qui n'appartiennent pas à A.

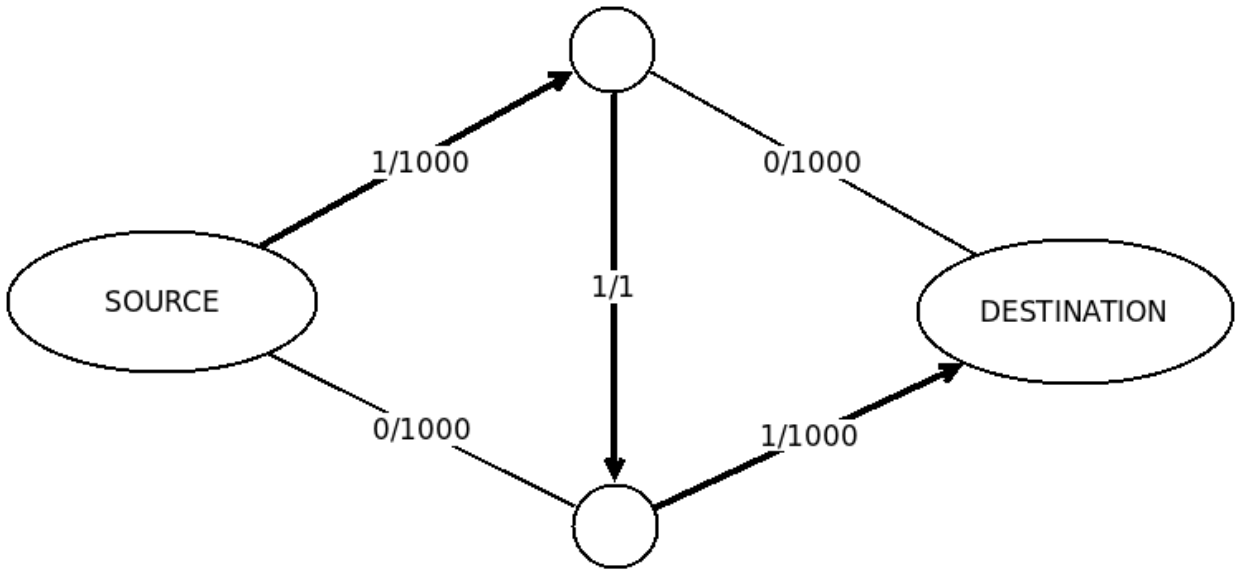
Problème de la complexité

Si on choisit les chemins de façon malheureuse, la complexité sera mauvaise.



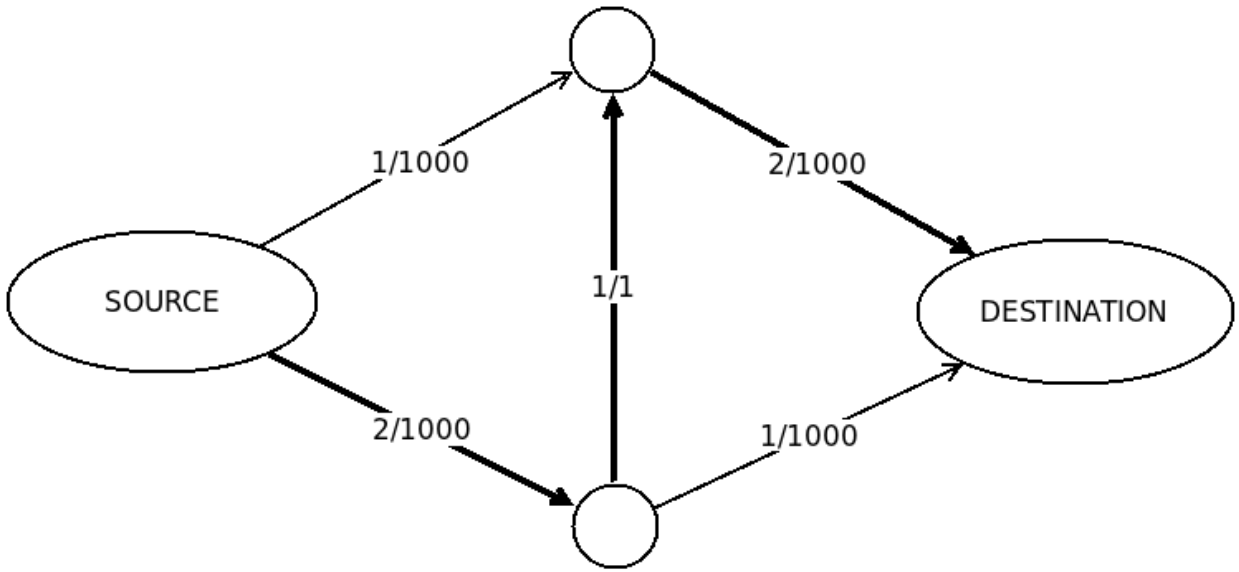
Problème de la complexité

Si on choisit les chemins de façon malheureuse, la complexité sera mauvaise.



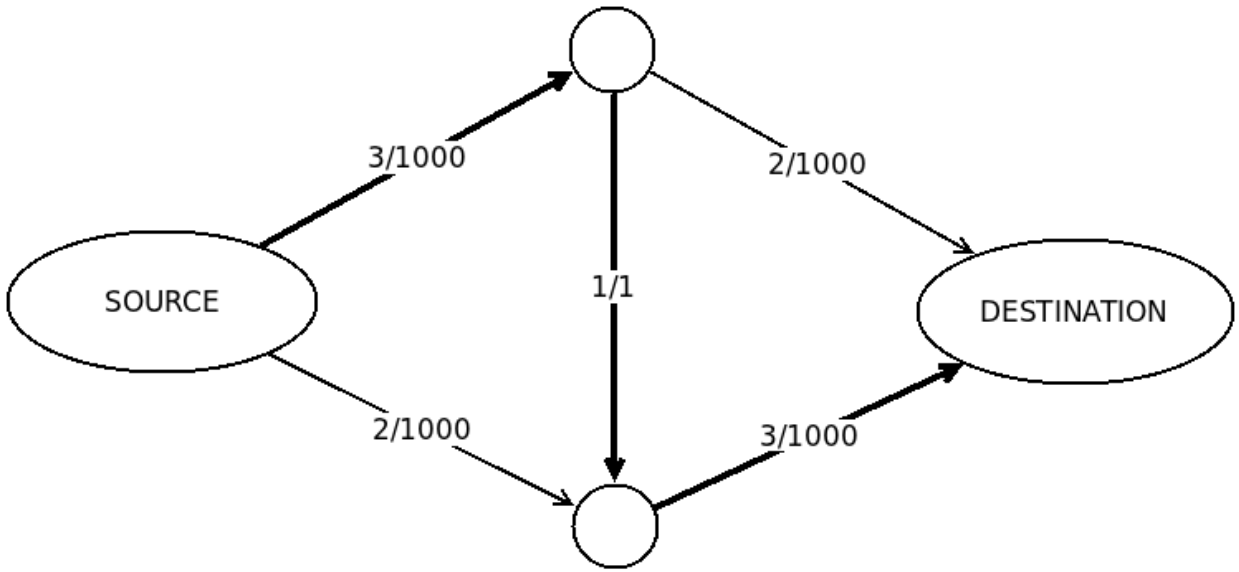
Problème de la complexité

Si on choisit les chemins de façon malheureuse, la complexité sera mauvaise.



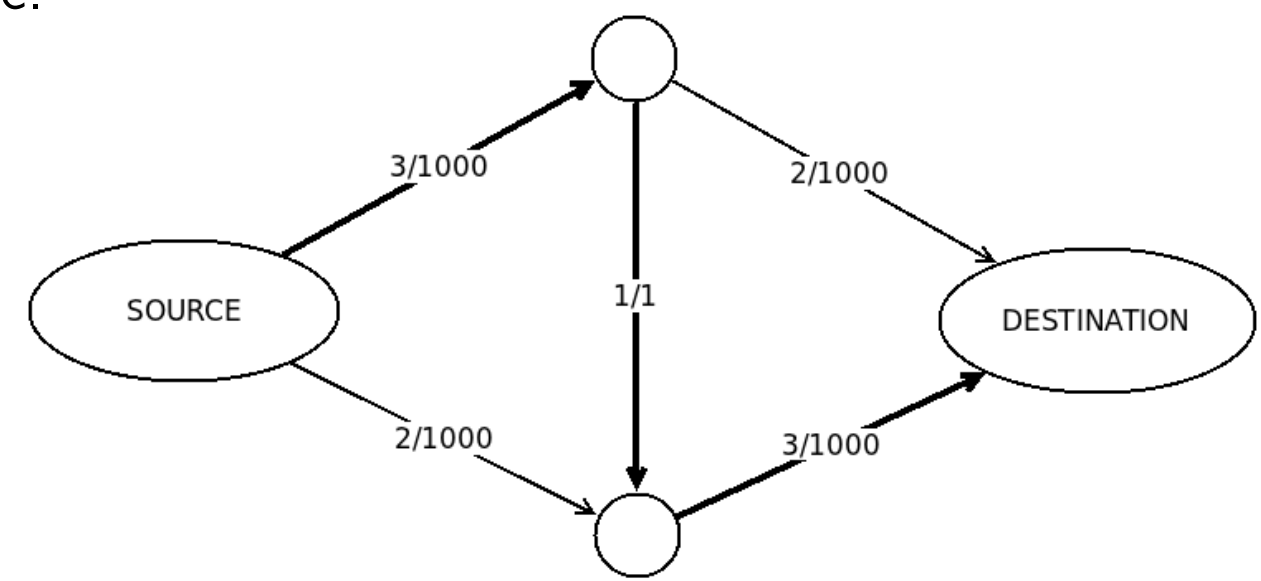
Problème de la complexité

Si on choisit les chemins de façon malheureuse, la complexité sera mauvaise.



Problème de la complexité

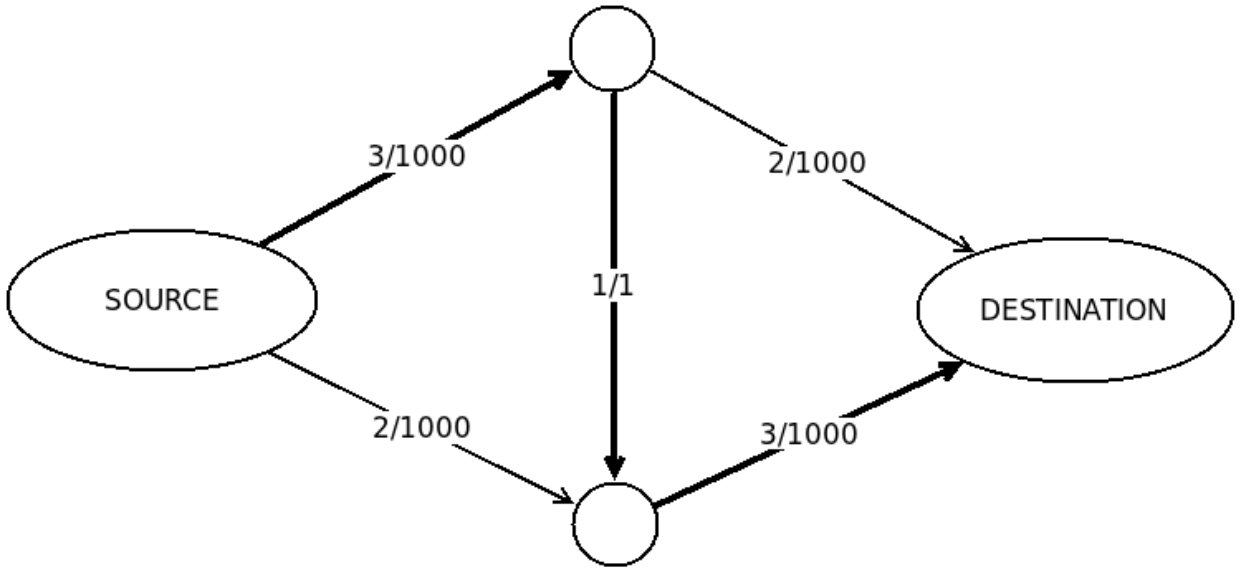
Si on choisit les chemins de façon malheureuse, la complexité sera mauvaise.



Ça fait 1001 itérations pour seulement 4 villes. On peut garantir une complexité meilleure si on impose un ordre sur les villes.

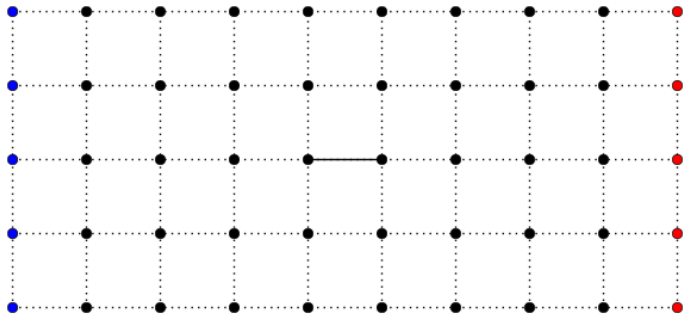
Problème de la complexité

Si on choisit les chemins de façon malheureuse, la complexité sera mauvaise.



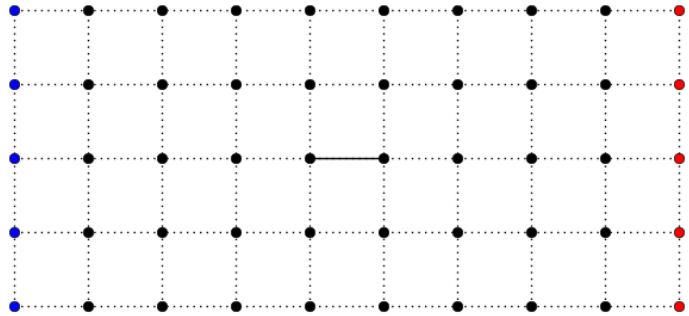
Implémentation efficace de Ford–Fulkerson : Edmonds–Karp

Le graphe des pixels

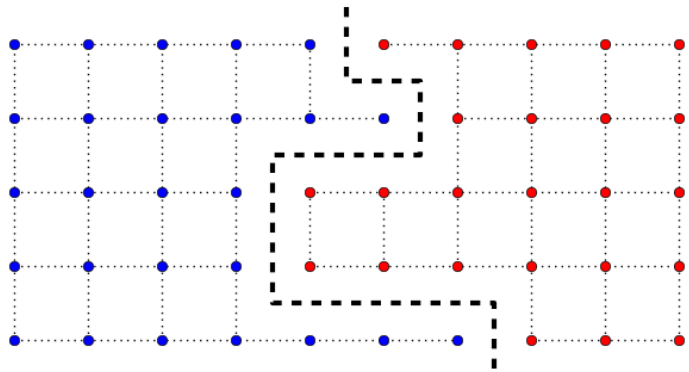


Capacité de l'arête : somme des poids des deux extrémités

Le graphe des pixels



Capacité de l'arête : somme des poids des deux extrémités



Pour aller plus loin

Réaliser une frise par recollements successifs

Pour aller plus loin

Réaliser une frise par recollements successifs

Recoller sur un cylindre pour imposer la périodicité

Pour aller plus loin

Réaliser une frise par recollements successifs

Recoller sur un cylindre pour imposer la périodicité

Passer à la 3D : recollement d'une vidéo

Pour aller plus loin

Réaliser une frise par recollements successifs

Recoller sur un cylindre pour imposer la périodicité

Passer à la 3D : recollement d'une vidéo

Passer à la 2D : recollement bidimensionnel

Table des matières