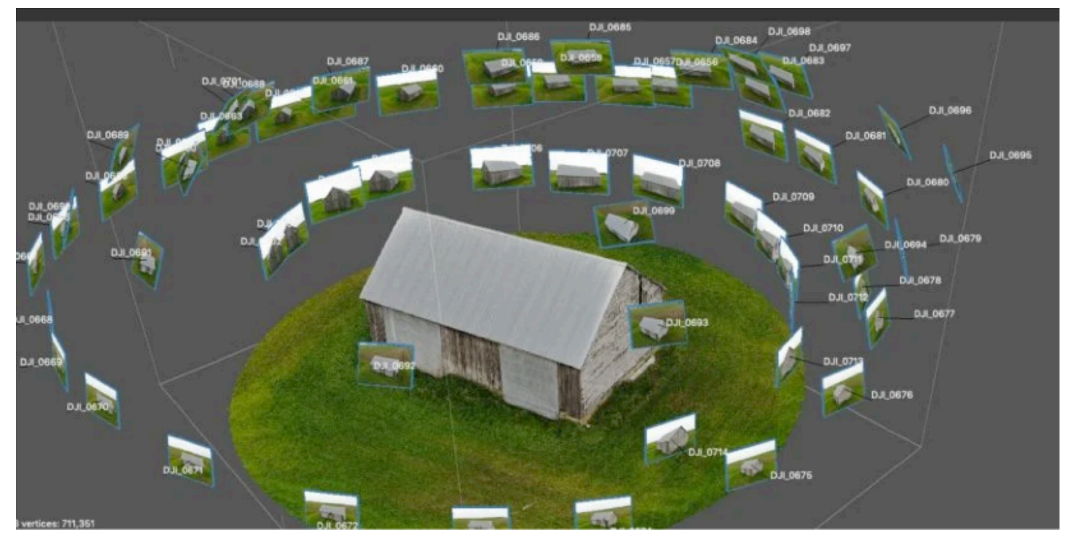
Reconstruction 3D basée sur des graphes de flots

Mica Macé

Numéro candidat : 22167

Techniques classiques de reconstruction 3D

LiDaR



Photogrammétrie

I - Graphes de flot : théorie

1. Graphe de flot : définition
2. Valeur et flot maximal
3. Coupe et coupe minimale
4. Théorème Max-Flow Min-Cut
5. Graphe résiduel et chemin augmentant
6. Méthode de Ford-Fulkerson
7. Comparatif des algorithmes de flot max

II - Segmentation d'une image 2D

1. Marquage des régions $\mathscr{O}$ et $\mathscr{B}$
2. Construction du graphe
    1. Capacités de région
    2. Capacité de frontière
    3. Définition de c
3. Résultat

III - Reconstruction 3D

1. Une nouvelle interprétation des GraphCuts
2. Ce qui change par rapport à la 2D
3. Résultat

Annexes

1a. Limites de l'intensité…
1b. … et améliorations
2. Calcul de la fonction O
3. Preuve du théorème et de l'algorithme
4. Une execution de l'algorithme
5. Programmes

Définition : Graphe de flot

Un graphe de flot est un quintuplet $(\mathscr{V}, \mathscr{E}, s, t, c)$, où :

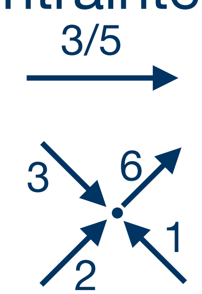
- $\mathscr{V}$: ensemble des sommets v du graphe
- $\mathscr{E} \subset \mathscr{V}^2$: ensemble des arêtes du graphe
- s : sommet source
- t : sommet puit
- $c : \mathscr{E} \to \mathbb{R}^+$: capacité des arêtes

On peut alors lui attribuer une fonction de flot :

Définition : flot

Un flot est une application $f : \mathscr{E} \to \mathbb{R}^+$ vérifiant les deux contraintes suivantes :

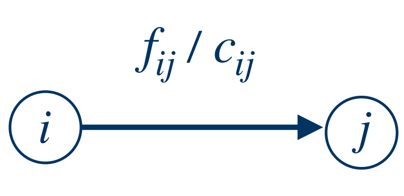
$$\forall (i,j) \in \mathscr{E} : \begin{cases} 0 \leq f_{ij} \leq c_{ij} \text{ (Contrainte de capacité)} \\ \forall v \notin \{s,t\}\ f_v^- = f_v^+ \text{ (Loi de Kirchov)} \end{cases}$$



On note

f_{ij} le flot et c_{ij} la capacité le long de l'arête (i,j)

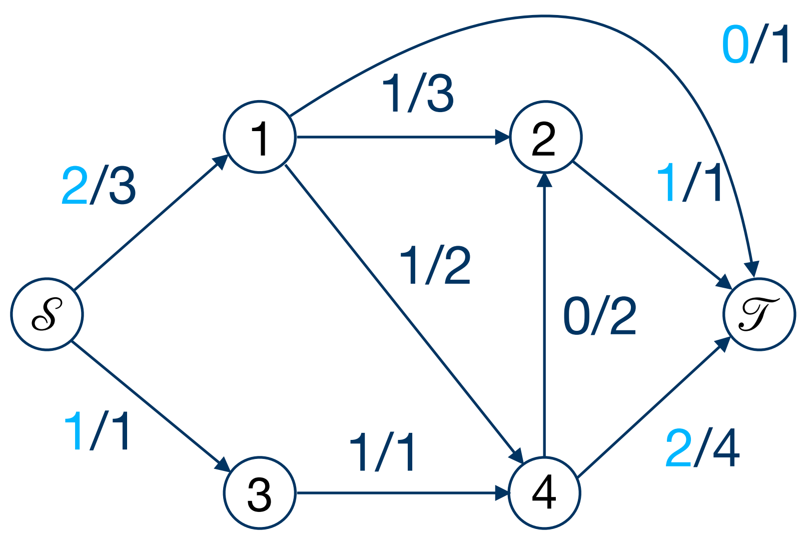
f^+, le flot entrant et f^-, le flot sortant d'un sommet.



Définition : valeur d'un flot

La valeur d'un flot f est notée $|f|$ et vaut :

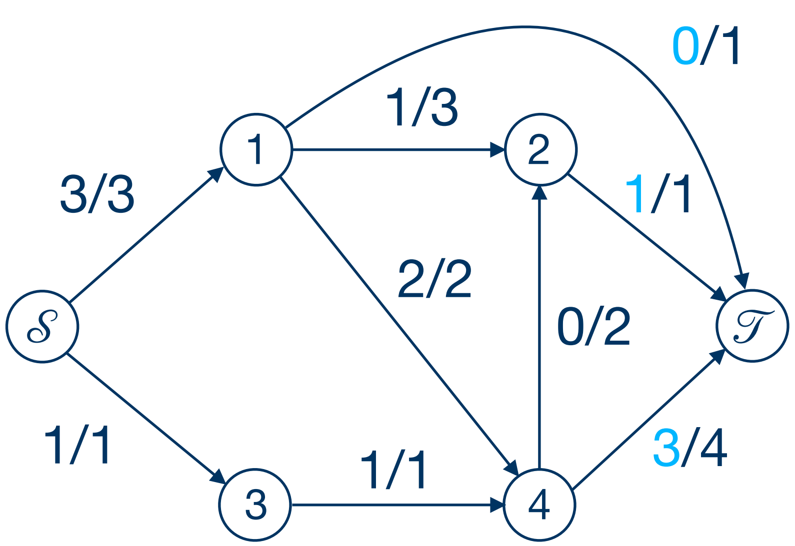
$$|f| = \sum_{i \in \mathscr{V},(i,t) \in \mathscr{E}} f_{i,t} = \sum_{i \in \mathscr{V},(s,i) \in \mathscr{E}} f_{s,i}$$



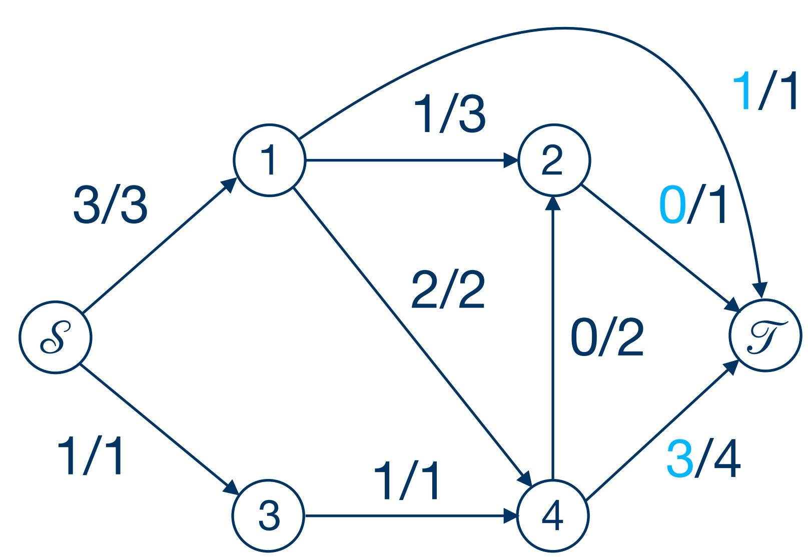
$|f| = 3$

Légende :

On note $f^\star$ un flot de valeur maximale



$|f| = |f^\star| = 4$



$|f| = |f^\star| = 4$

Définition : coupe

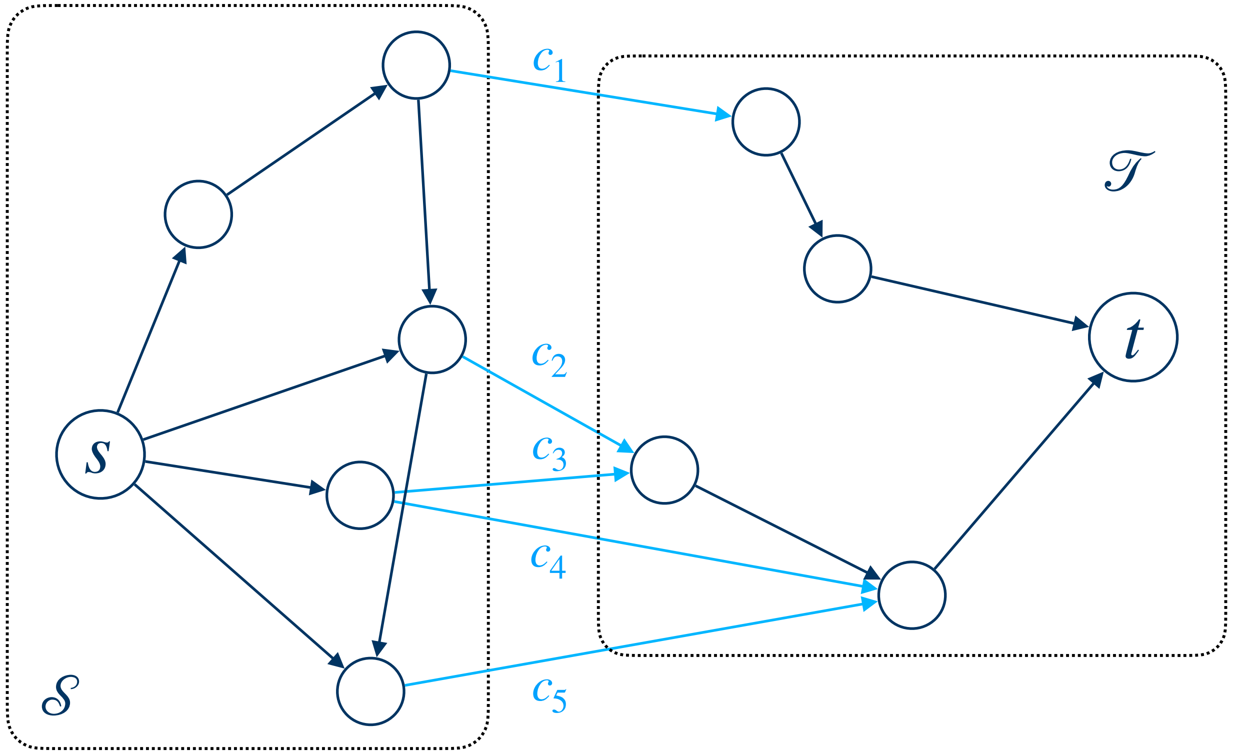
On appelle coupe d'un graphe de flot une partition $(\mathcal{S}, \mathcal{T})$ de $\mathcal{V}$ telle que :

$$s \in S, t \in T, S \cap T = \varnothing$$

La valeur de la coupe notée $\mathcal{C}(\mathcal{S}, \mathcal{T})$ est :

$$\mathcal{C}(\mathcal{S}, \mathcal{T}) = \sum_{i \in \mathcal{S}, j \in \mathcal{T}} c_{ij}$$

Une coupe minimale est une coupe de valeur minimale



$$\mathcal{C}(\mathcal{S}, \mathcal{T}) = c_1 + c_2 + c_3 + c_4 + c_5$$

Théorème (Ford - Fulkerson) : Max-flow min-cut
Pour tout graphe de flot, la valeur d'une coupe minimale
est égale à la valeur d'un flot maximal

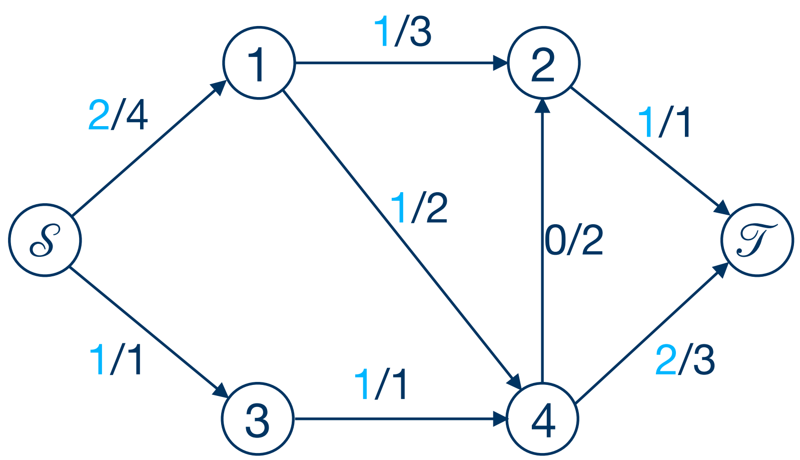
Définition : Graphe résiduel

Le graphe résiduel de G associé à f est : $R_f = (\mathscr{V}, \mathscr{E}_f, s, t, c_f)$

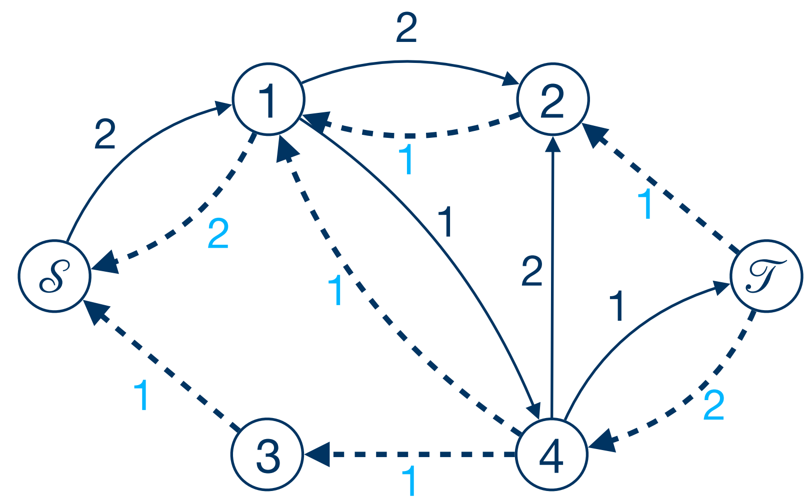
avec :

- $\mathscr{E}_f = \{(u,v) \in \mathscr{V}^2 \,|\, c_f(u,v) > 0\}$
- $c_f : \mathscr{V}^2 \longrightarrow \mathbb{R}^+$, capacité résiduelle, définie par :

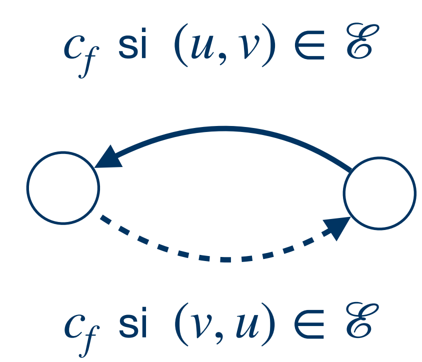
$$\forall (u,v) \in \mathscr{V}^2, c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{si } (u,v) \in \mathscr{E} \\ f(u,v) & \text{si } (v,u) \in \mathscr{E} \\ 0 & \text{sinon} \end{cases}$$



graphe G parcouru par un flot f



graphe résiduel R_f



Définition : chemin augmentant

Un chemin $s \rightarrow t$ dans le graphe résiduel R_f est appelé chemin augmentant

Entrée : graphe $(\mathscr{V}, \mathscr{E}, s, t, c)$

$f \leftarrow$ flot nul

Tant que il existe un chemin simple γ de s vers t dans R_f **:**

    $\Delta = \min\{r_f(u,v) \mid (u,v) \in \gamma\}$

    Pour $(u,v) \in \gamma$ **:**

        Si $(u,v) \in A$ **:**

            $f(u,v) \leftarrow f(u,v) + \Delta$

        Sinon :

            $f(u,v) \leftarrow f(u,v) - \Delta$

Renvoyer f

Algorithme	**Complexité temporelle**
Floyd-Fulkerson	$O(E\,.\,\|f^\star\|)$
Edmonds-Karp	$O(V\,.\,E^2)$
Dinitz	$O(V^2\,.\,E)$
Shortest-augmenting path	$O(V^2\,.\,E)$
Boykov-Kolmogorov	$O(V^2\,.\,E\,.\,\|f^\star\|)$
Pre-flow push	$O(V^2\,.\,\sqrt{E})$

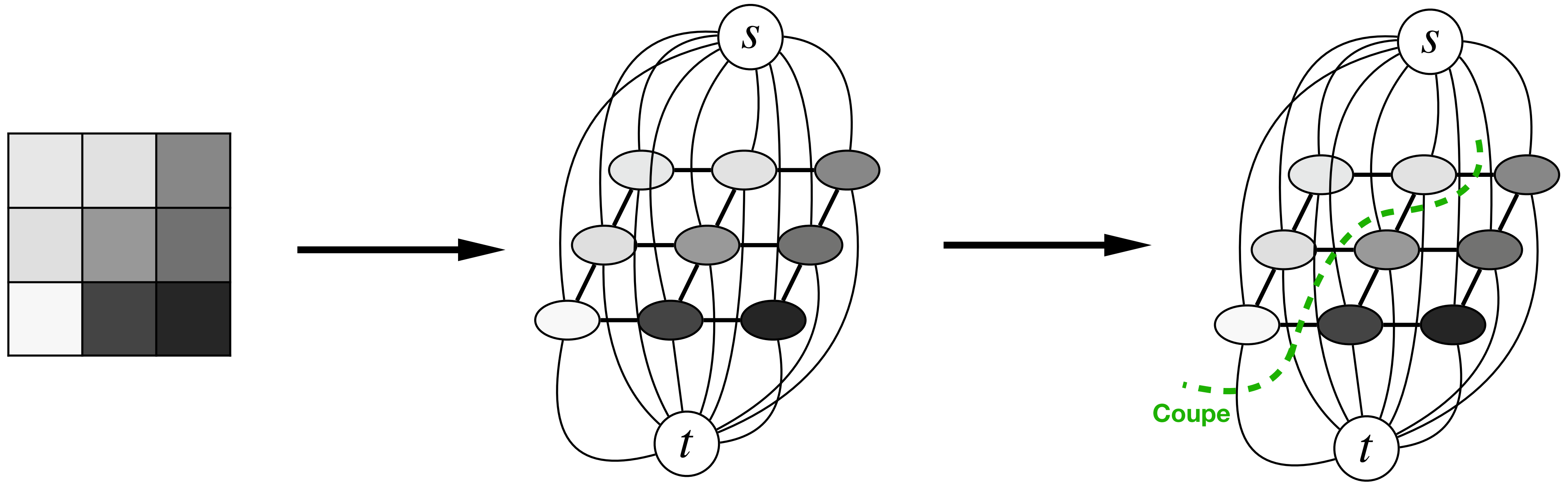
II - Segmentation d'image

Objectif

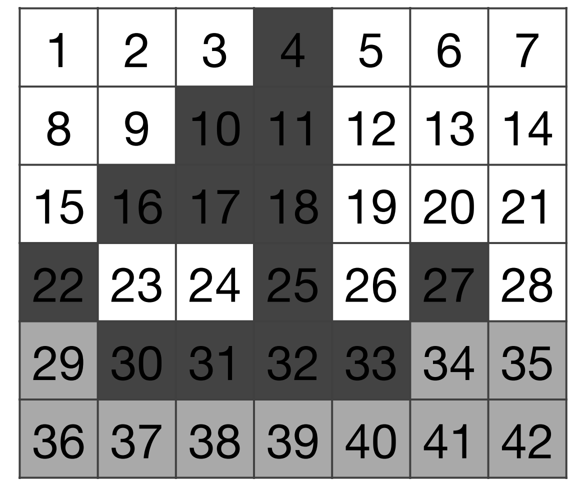
Détourer la silhouette d'un objet du fond d'une image après la sélection de quelques pixels appartenant respectivement à l'objet et au fond de l'image.

Technique utilisée

Transformer le problème du détourage d'image en un **problème de recherche de coupe minimale dans un graphe de flots.** (d'apres Boykov et Jolly)

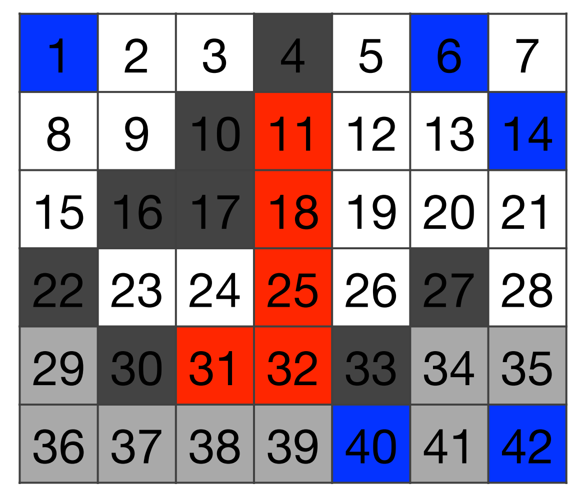


L'ensemble des pixels $\mathscr{P}$ d'une image de dimensions $w \times h$ est identifié à $[\![1, w \times h]\!]$ en allant de la gauche vers la droite puis de haut en bas.



Selection des pixels graines dans chacune des « régions » de l'image

$\begin{cases} \mathscr{O} \text{ la région de l'objet, en rouge} \\ \mathscr{B} \text{ la région du background, en bleu} \end{cases}$



Pour construire le graphe $G = (\mathcal{V}, \mathcal{E}, s, t, c)$:

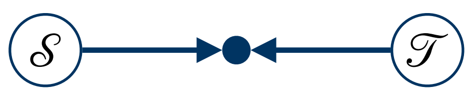
Définissons l'ensemble des sommets $\mathcal{V}$ par $\mathcal{V} = \mathcal{P} \cup \{0, wh+1\}$

(où 0 correspond à s et $wh + 1$ correspond à t)

Définissons $\mathcal{E}$ l'ensemble des arêtes en deux temps :

Les arêtes de région reliant chaque pixel à la source et au puit par :

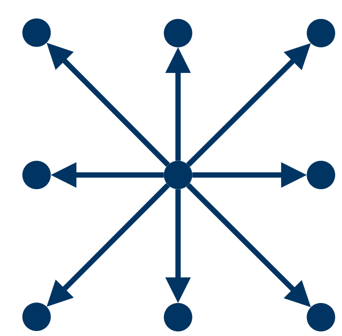
$$\mathcal{F} = \bigcup_{p \in \mathcal{P}} \{(s,p),(p,t)\}$$



Les arêtes de frontière reliant les pixels entre eux par :

$$\forall p \in \mathcal{P},\ \mathcal{N}_p = \{p' \,|\, p' \in \mathcal{P} \text{ et } d(p,p') \leq \sqrt{2}\}$$

$$\text{et : } \mathcal{N} = \{(p,p') \,|\, p \in \mathcal{P}, p' \in \mathcal{N}_p\}$$



Finalement : $\mathcal{E} = \mathcal{N} \cup \mathcal{F}$

Il reste à définir c la fonction de capacité pour chaque type d'arête

Pour les capacités de région, Boykov et Jolly posent :

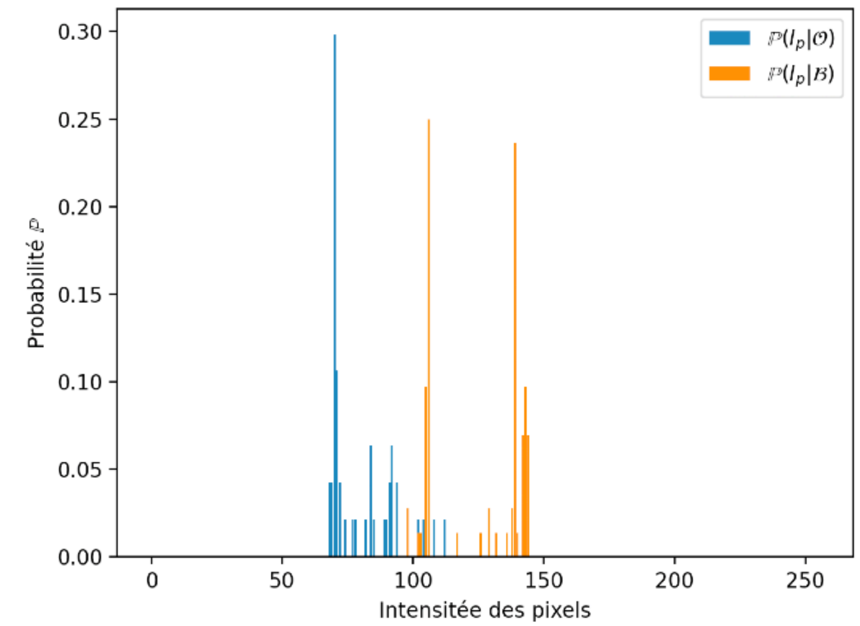
$$\forall p \in \mathcal{N}, \begin{cases} R_p(0) = -\ln(\mathbb{P}(I_p \,|\, \mathcal{O})) \\ R_p(1) = -\ln(\mathbb{P}(I_p \,|\, \mathcal{B})) \end{cases}$$

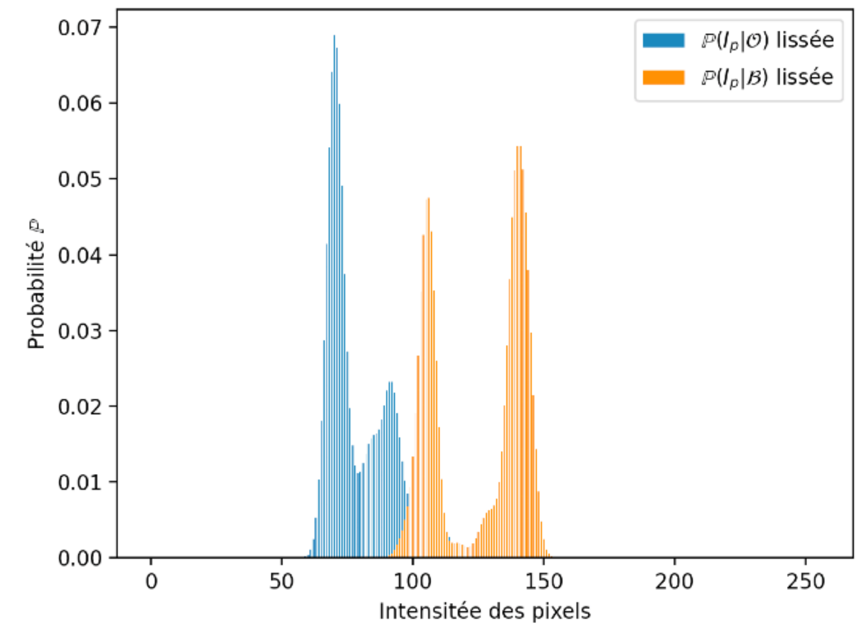
Le calcul de $\mathbb{P}$ se fait alors en trois temps :

- Calcul de l'histogramme des intensités des graines

- Normalisation pour obtenir une probabilité

- « Lissage » de la probabilité par un noyau gaussien



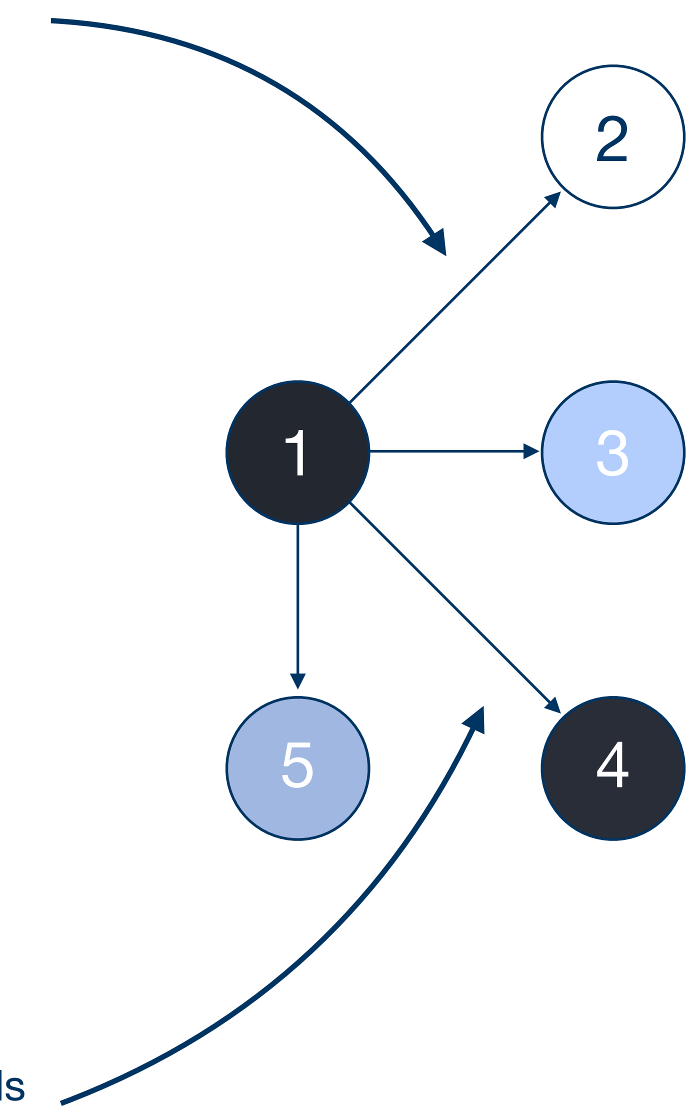


Pour les capacités de frontière, Boykov et Jolly proposent :

$$B_{\{p,q\}} = \frac{\exp\left(-\frac{(I_p - I_q)^2}{2\sigma^2}\right)}{d(p,q)}$$

$d(p,q)$: la distance euclidienne entre deux pixels

Ici, il faut une capacité très faible pour séparer ces pixels



Là, il faut une capacité très élevée pour regrouper ces pixels

Arête	Si	Alors c =
$\{p, q\}$	$\{p, q\} \in \mathcal{N}$	$B_{\{p,q\}}$
$\{p, s\}$	$p \in \mathcal{P}, p \notin \mathcal{O} \cup \mathcal{B}$	$\lambda \,.\, Rp(1)$
	$p \in \mathcal{O}$	K
	$p \in \mathcal{B}$	0
$\{p, t\}$	$p \in \mathcal{P}, p \notin \mathcal{O} \cup \mathcal{B}$	$\lambda \,.\, Rp(0)$
	$p \in \mathcal{O}$	0
	$p \in \mathcal{B}$	K

Avec

$$K = 1 + \max_{p \in \mathcal{P}} \sum_{\{p,q\} \in \mathcal{N}} B_{\{p,q\}}$$

K est utilisé pour tous les pixels déjà identifiés comme appartenant à l'objet ou au background. Il est pris très grand pour éviter que la coupe ne sépare ces pixels.

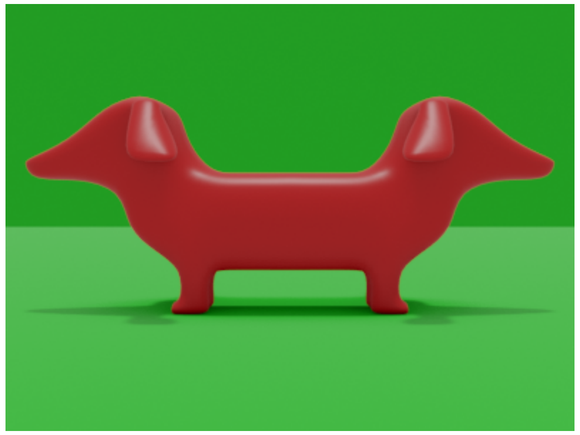
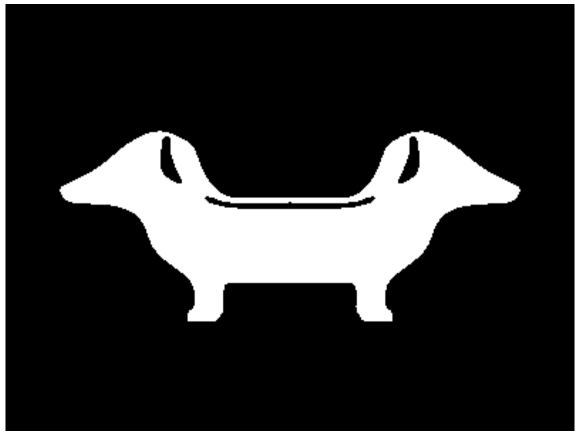


Image d'origine



Résultat de la segmentation

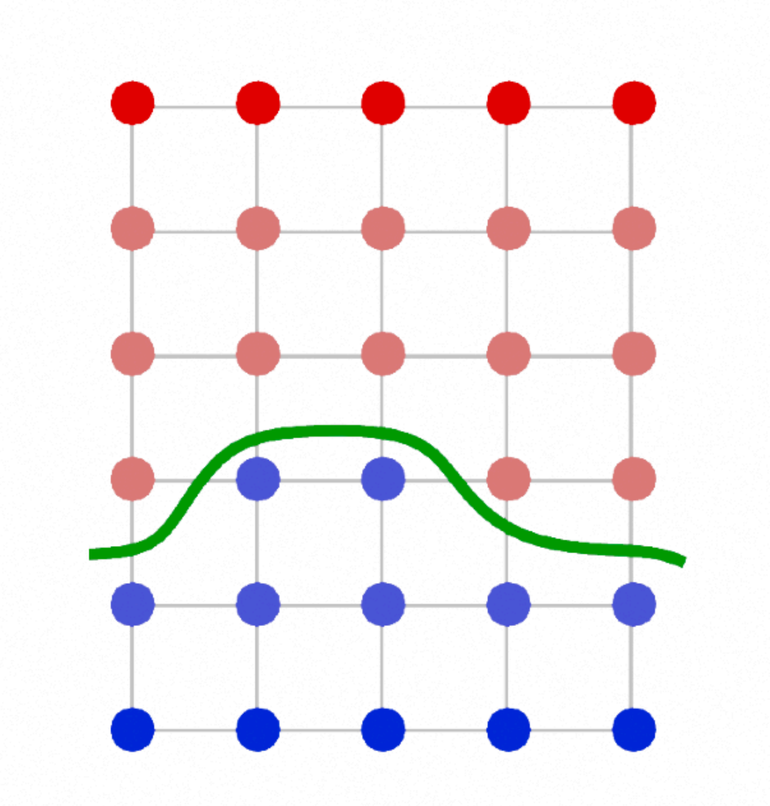
III - Reconstruction 3D

Objectif

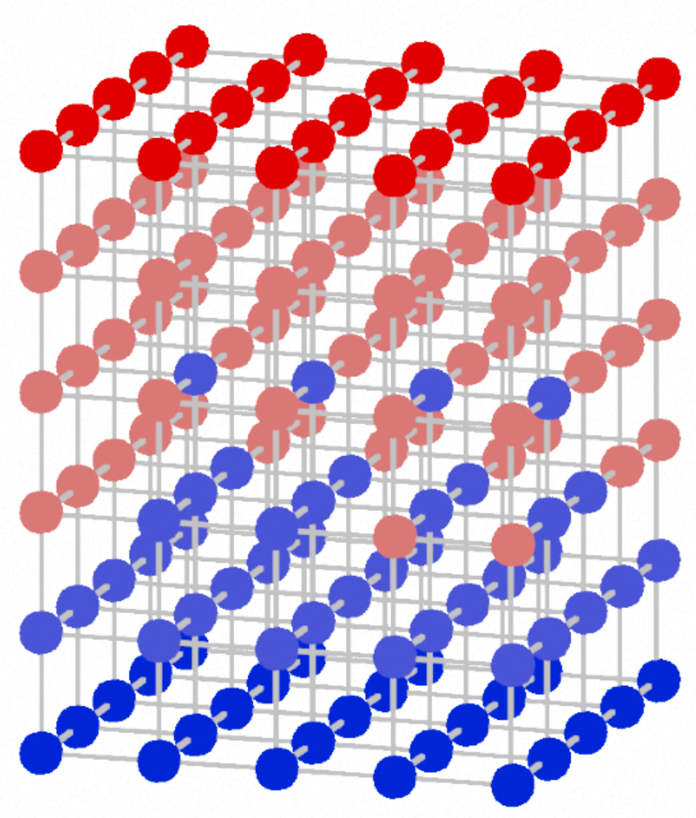
Reconstruire un nuage de points représentant le modèle 3D
d'un objet à partir d'une série de photos de l'objet.

Technique utilisée

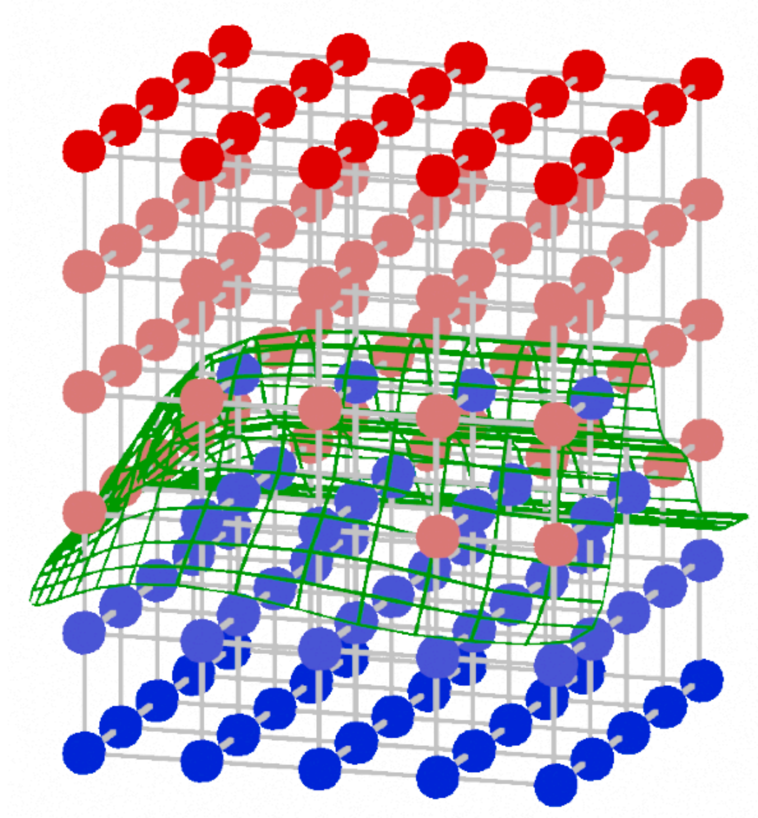
Transformer **le problème de reconstruction 3D** en un **problème de recherche de coupe minimale dans un graphe de flot** créé à partir des silhouettes des objets détourés par la méthode précédente.



Une coupe en 2D et la courbe associée

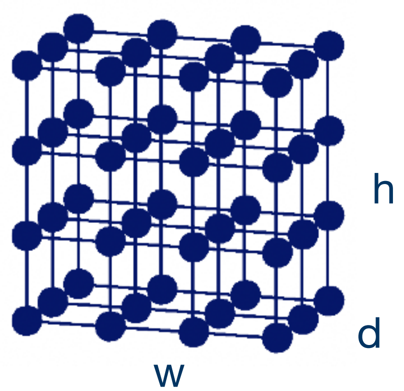


Une coupe en 3D



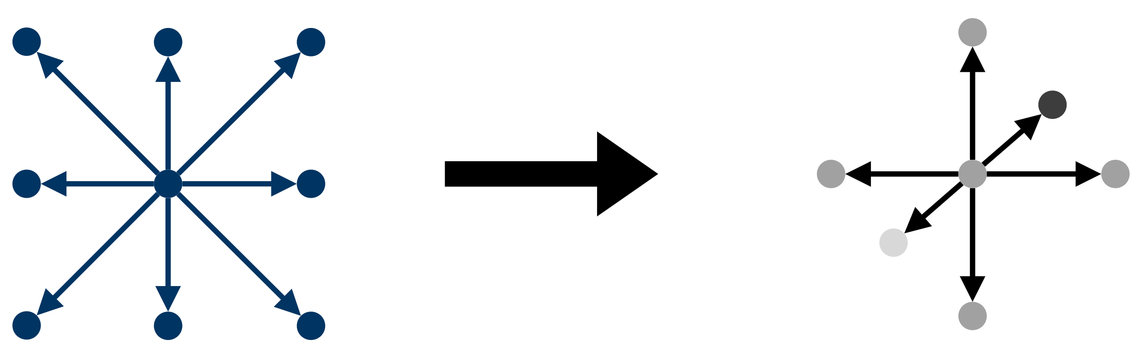
L'hypersurface de dimension 2 associée

Les pixels deviennent des <u>voxels</u> (équivalent 3D du pixel)



$$\mathcal{V}' = [\![1, w \times h \times d]\!]$$
$$\mathcal{V} = \mathcal{V}' \cup \{0, whd + 1\}$$

Les arêtes de frontière sont plus nombreuses, on <u>réduit le voisinage</u> :



$$\mathcal{F} = \bigcup_{p \in \mathcal{V} / \{s,t\}} \{(s,p),(p,t)\}$$

$\forall v \in \mathcal{V}', \mathcal{N}_v = \{v' \,|\, v' \in \mathcal{V}'$ et $d(v,v') \leq 1\}$

et : $\mathcal{N} = \{(v,v') \,|\, v \in \mathcal{V}', v' \in \mathcal{N}_v\}$

$\mathcal{E} = \mathcal{N} \cup \mathcal{F}$

<u>On ne dispose plus de l'intensité de chaque voxel</u>, car certains sont cachés par d'autre. A la place on utilise le <u>nombre de camera « observant » le voxel v, noté $O(v)$</u>

Alors on définit c par :

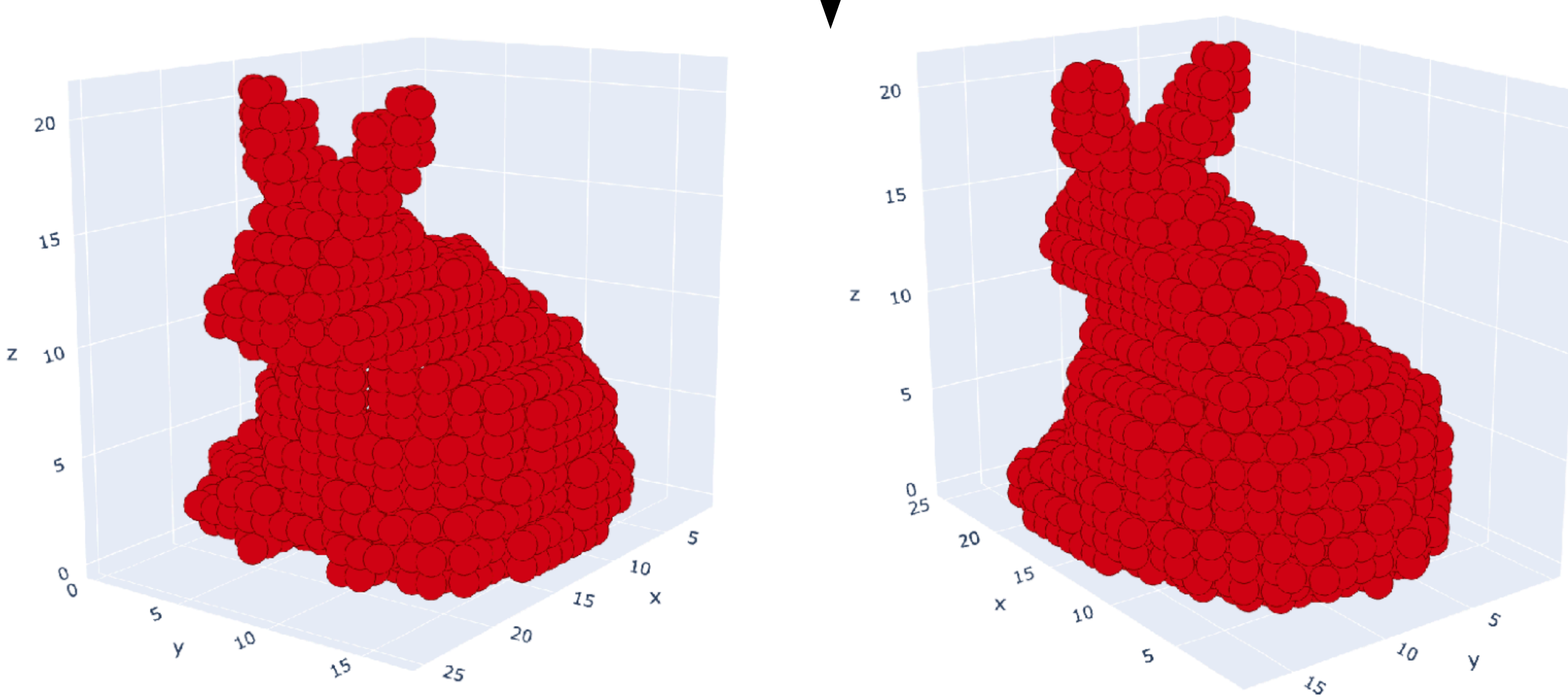
$$B_{p,q} = \frac{\alpha \cdot O(p) \cdot O(q)}{\langle O \rangle^2}$$

et : $\begin{cases} R_p(1) = \lambda(1 - \exp(\frac{O(p)}{\nu})) \\ R_p(0) = C^{ste} \end{cases}$

avec $\alpha, \lambda, \nu, C^{ste}$ à ajuster

Prises de vues fournies à l’algorithme avec leurs données spatiales

Résultat de la reconstruction 3D



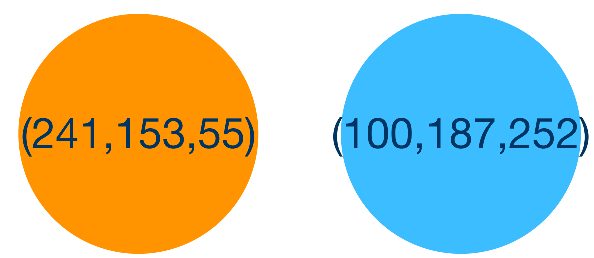
Merci pour votre attention

Soit un pixel $p = (R, G, B)$

Son intensitée I_p est définie par : $\begin{cases} I_p = \left\lfloor R\frac{299}{1000} + G\frac{587}{1000} + B\frac{114}{1000} \right\rfloor \\ I_p \in [0,255] \end{cases}$

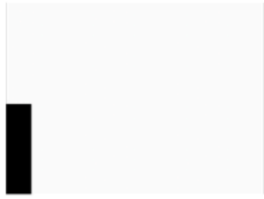
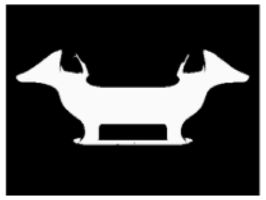
et les coefficients sont adaptés pour l'oeil humain

Problème : ces deux pixels ont même intensité et seront donc considérés comme extremement similaires



Ce problème est du à la projection de l'espace $[0,255]^3$ sur $[0,255]$ par cette fonction mais ne tient pas compte de sa géométrie.

Le tableau suivant récapitule différentes méthodes pour calculer cette différence

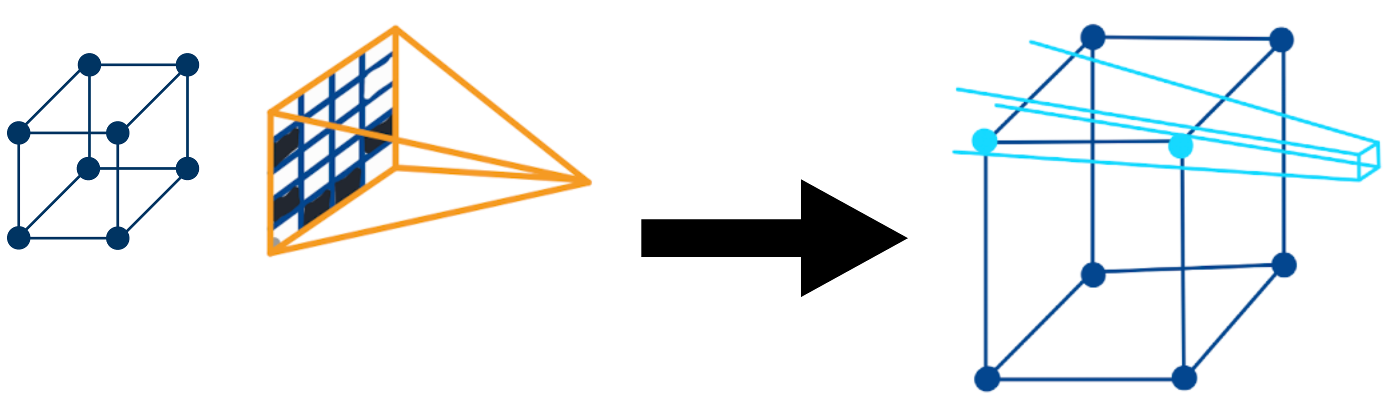
Fonction	Avantage	Inconvénient	Résultat
$\Delta_{I,pp'} = I_p - I'_p$	Formule assez intuitive	Fonctionne très mal	
$\Delta_{I,pp'} = \left\Vert \begin{pmatrix} r \\ g \\ b \end{pmatrix} - \begin{pmatrix} r' \\ g' \\ b' \end{pmatrix} \right\Vert^2$	Vectoriellement cohérente	Fonctionne mal	
$\Delta_{I,pp'} = \sum_{i=1}^{3} \frac{256}{1 + \lvert p_i - p'_i \rvert}$	Théoriquement prometteuse	Fonctionne assez mal	
$\Delta_{I,pp'} = \left\Vert \begin{pmatrix} r \\ g \\ b \end{pmatrix} - \begin{pmatrix} r' \\ g' \\ b' \end{pmatrix} \right\Vert$	Cohérente	Fonctionne mal	
$\Delta_{I,pp'} = \left\Vert \begin{pmatrix} r \\ g \\ b \end{pmatrix} + \begin{pmatrix} r' \\ g' \\ b' \end{pmatrix} \right\Vert$	Fonctionne très bien en pratique	N'a pas de raison théorique de fonctionner de manière générale, peut-être pas généralisable à d'autres images	
$\Delta_{I,pp'} = \left\Vert \begin{pmatrix} r \\ g \\ b \end{pmatrix} \wedge \begin{pmatrix} r' \\ g' \\ b' \end{pmatrix} \right\Vert$	Théoriquement très efficace sur n'importe quelle image.	Fonctionne un tout petit peu moins bien que la précédente sur cet exemple particulier.	

Soit (x_0, y_0, z_0) les coordonnées de la caméra, θ sa rotation suivant u_z, (x, y, z) les coordonnées de l'intersection d'un vecteur unitaire normal au plan de l'image et ayant pour origine (x_0, y_0, z_0), et du plan de l'image.
Enfin (x', y', z') sont les composantes du vecteur du pixel tout en haut à gauche de l'image.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 - \sin(\theta) \\ y_0 + \cos(\theta) \\ z_0 + \frac{FoV_Y}{2} \end{pmatrix} \text{ et } \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x - \cos(\theta)\,.\,\tan(\frac{FoV}{2}) \\ y - \sin(\theta)\,.\,\tan(\frac{FoV}{2}) \\ z \end{pmatrix}$$

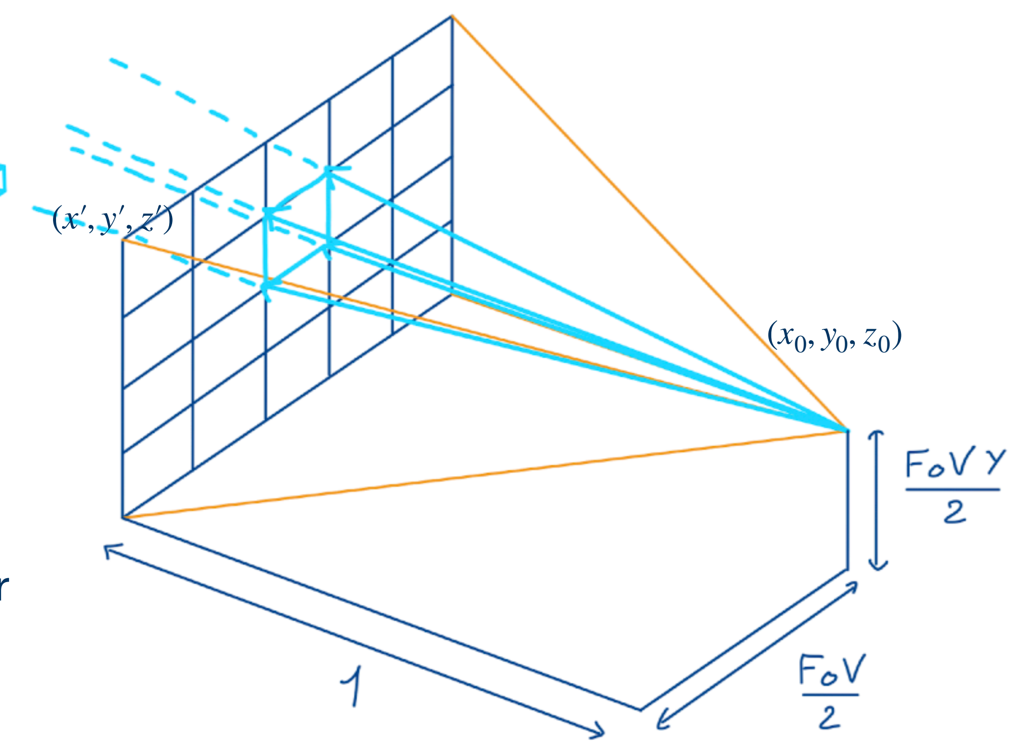
Soit :

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x_0 - \cos(\theta)\,.\,\tan(\frac{FoV}{2}) - \sin(\theta) \\ y_0 - \sin(\theta)\,.\,\tan(\frac{FoV}{2}) + \cos(\theta) \\ z_0 + \frac{FoV_Y}{2} \end{pmatrix}$$



L'image est placée à une distance (virtuelle) de 1 de la caméra

Le volume virtuel de la reconstruction (structure blanche) se fait intersecter par l'angle solide du pixel (bleu)



Lemme 1 :

Pour tout flot f et coupe $\mathscr{C}(\mathscr{S}, \mathscr{T})$, on a :

$|f| = f^{-}(\mathscr{S}) - f^{+}(\mathscr{S})$

Preuve

Par définition : $|f| = f^{-}(s) = f^{-}(s) - f^{+}(s)$

Et par la loi de Kirchov : $\forall v \neq s, t, f^{-}(v) - f^{+}(v) = 0$

Donc :

$$|f| = \sum_{v \in \mathscr{S}} f^{-}(v) - f^{+}(v) = \sum_{e \text{ sortant de } \mathscr{S}} f(e) - \sum_{e \text{ entrant dans } \mathscr{S}} f(e) = f^{-}(\mathscr{S}) - f^{+}(\mathscr{S})$$

Lemme 2 :

Pour tout flot f et coupe $\mathscr{C}(\mathscr{S}, \mathscr{T})$, on a :

$|f| \leq \mathscr{C}(\mathscr{S}, \mathscr{T})$

Preuve

$$|f| = f^{-}(\mathscr{S}) - f^{+}(\mathscr{S}) \leq f^{-}(\mathscr{S}) = \sum_{e \text{ sortant de } \mathscr{S}} f(e) \leq \sum_{e \text{ sortant de } \mathscr{S}} c(e) = \mathscr{C}(\mathscr{S}, \mathscr{T})$$

Ainsi, la valeur de toute coupe majore la valeur de tout flot.
On va alors tenter d'exhiber une coupe minimale à partir de l'algorithme de Ford et Fulkerson. Le théorème découlera de la terminaison de l'algorithme

Lemme 3 :

Soit f un flot tel qu'il n'existe aucun chemin reliant s et t dans R_f, alors il existe une coupe $\mathscr{C}(\mathscr{S}^\star, \mathscr{T}^\star)$ telle que $|f| = \mathscr{C}(\mathscr{S}^\star, \mathscr{T}^\star)$

Preuve :

Construisons alors une telle coupe. Notons $\mathscr{S}^\star = \{v \in \mathscr{V}, \exists \gamma_{s \to v} \in R_f\}$ et $\mathscr{T}^\star = \mathscr{V} / \mathscr{S}^\star$.

On vérifie alors que $\mathscr{C}(\mathscr{S}^\star, \mathscr{T}^\star)$ est bien une coupe car :

- $\mathscr{S}^\star \sqcup \mathscr{T}^\star = \mathscr{V}$
- $s \in \mathscr{S}^\star$ par définition, et $t \notin \mathscr{S}^\star$ car il n'existe pas de chemin $s \to t \in R_f$ donc $t \in \mathscr{T}^\star$

Montrons alors :

-Si $e = (u, v) \in \mathscr{E}, u \in \mathscr{S}^\star, v \in \mathscr{T}^\star$, alors $\underline{f(e) = c_e}$.
Autrement, e serait une arête directe dans R_f. Or $u \in \mathscr{S}^\star$ donc il existe un chemin $s \to u \in R_f$ et en ajoutant e à ce chemin, on disposerait d'un chemin $s \to v \in R_f$, ce qui contredit que $v \in \mathscr{T}^\star$

-Si $e = (u', v') \in \mathscr{E}, u' \in \mathscr{T}^\star, v' \in \mathscr{S}^\star$, alors $\underline{f(e) = 0}$.
Autrement, e serait une arête indirecte dans R_f. Or $v' \in \mathscr{S}^\star$ donc il existe un chemin $s \to v' \in R_f$ et en ajoutant e à ce chemin, on disposerait d'un chemin $s \to u' \in R_f$, ce qui contredit que $u' \in \mathscr{T}^\star$

Ainsi pour cette coupe, les arêtes sortantes de $\mathscr{S}^\star$ sont saturées, et celles entrantes sont vides

Par le Lemme 1, on a :

$$|f| = f^{-}(\mathscr{S}^\star) - f^{+}(\mathscr{S}^\star)$$

$$= \sum_{e \text{ sortant de } \mathscr{S}^\star} f(e) - \sum_{e \text{ entrant dans } \mathscr{S}^\star} f(e)$$

$$= \sum_{e \text{ sortant de } \mathscr{S}^\star} c_e - 0$$

$$= \mathscr{C}(\mathscr{S}^\star, \mathscr{T}^\star)$$

Corollaire 1 : Théorème max-flow min-cut

Sous les hypothèses précédentes, $|f| = f^\star$ et $\mathscr{C}(\mathscr{S}^\star, \mathscr{T}^\star)$ est une coupe minimale

Preuve :

Conséquence directe du Lemme 2

Corollaire 2 :

Le flot f renvoyé par l'algorithme de Ford et Fulkerson est un flot maximal

Preuve :

L'algorithme de Ford-Fulkerson termine lorsqu'il n'existe plus de chemin $s \to t \in R_f$,
nous venons de voir que cela est équivalent à la maximalité de f

Si on suppose que les capacités sont entières (ou rationelles, ce qui se ramène à des entier en multipliant par le pgcd), on peut prouver la terminaison de l'algorithme de Ford-Fulkerson

Lemme 3 :

La propriété : $\forall e \in \mathscr{E}, f(e) \in \mathbb{N}, \forall e' \in \mathscr{E}_f, c_f(e') \in \mathbb{N}$ est un invariant de boucle

Preuve :

La propriété est initialement vraie par hypothèse.
Supposons la vraie après j itérations de la boucle, alors par hypothèse de récurrence, $\Delta \in \mathbb{N}$ donc le flot renvoyé est à valeur entière et les nouvelles capacités du graphe résiduel également

Lemme 4 :

La propriété : Le valeur du flot augmente strictement à chaque itération est un invariant de boucle

Preuve :

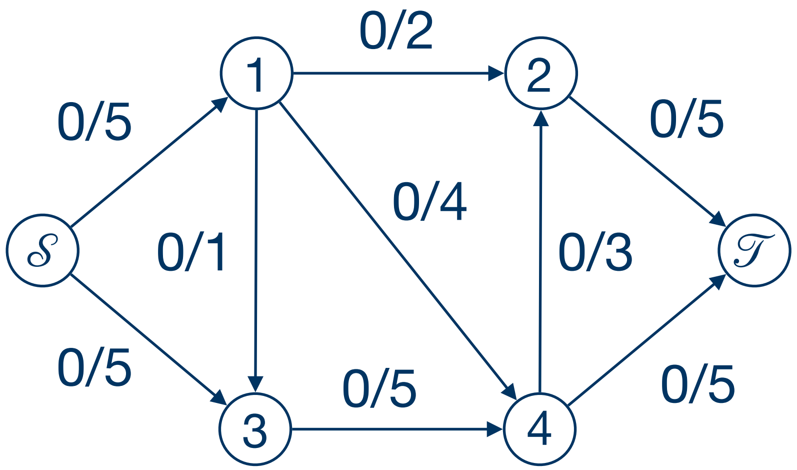
Notons f le flot avant l'itération de la boucle, et f' celui après. Alors $|f'| = |f| + \Delta$ et

$$\begin{cases} \Delta \in \mathbb{N} \text{ par le Lemme 3} \\ \Delta \neq 0 \text{ car un chemin augmentant existe} \end{cases} \text{donc } \Delta > 0$$

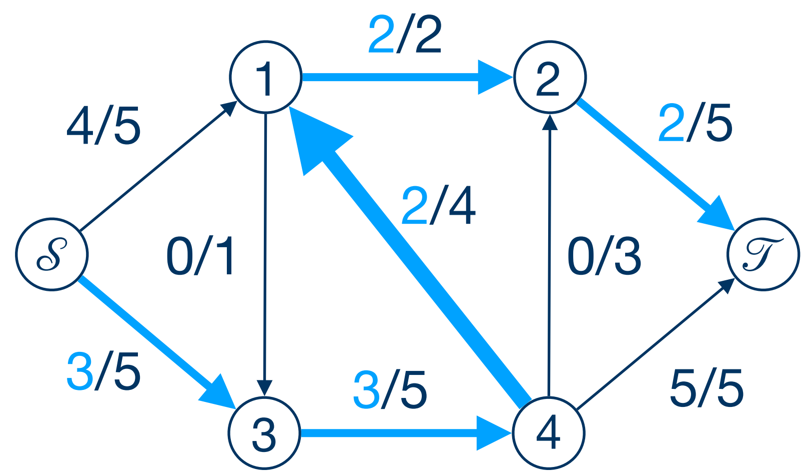
d'où $|f'| > |f|$

Preuve de la terminaison :

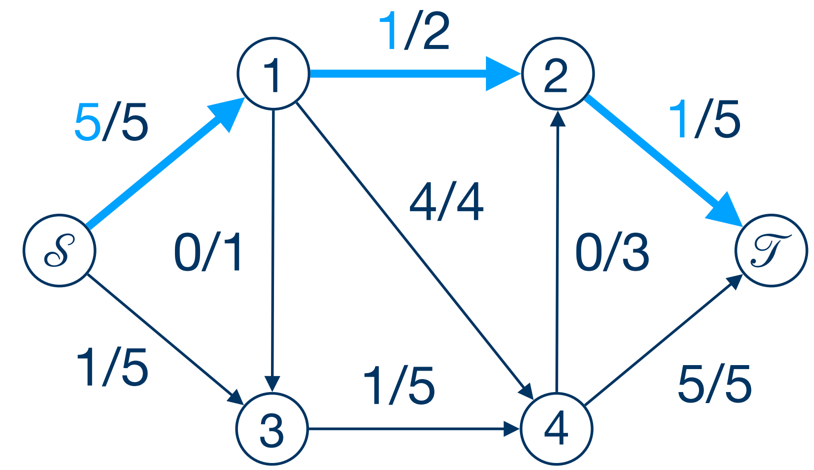
Le théorème affirme que le flot est majoré par la valeur d'une coupe minimale.
De plus la suite $(|f|_i)_{i \in \mathbb{N}}$ des valeurs du flot à l'itération i est une suite strictement croissante (Lemme 4) à valeur dans $\mathbb{N}$ (Lemme 3) et majorée, elle converge donc en un nombre fini d'étape, plus précisément en $\mathcal{O}(f^\star)$ étapes
Puisque la recherche de Δ peut se faire en $\mathcal{O}(|\mathscr{E}|)$ on a une complexité totale de l'algorithme en $\mathcal{O}(|\mathscr{E}|.f^\star)$



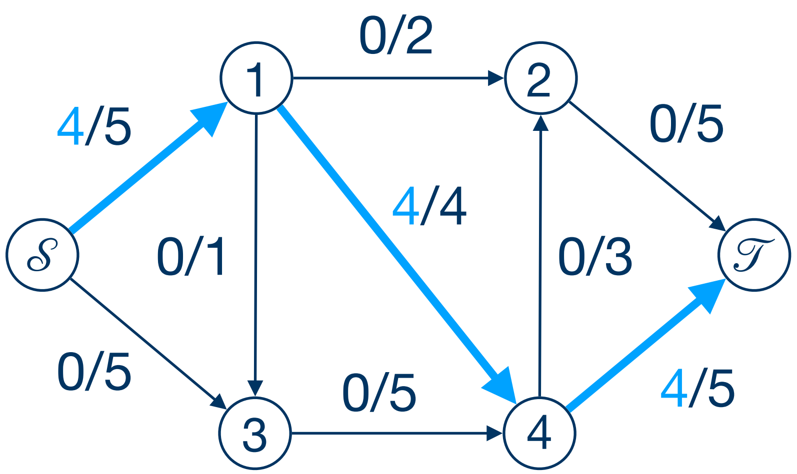
Etape 0 : $|f| = 0$



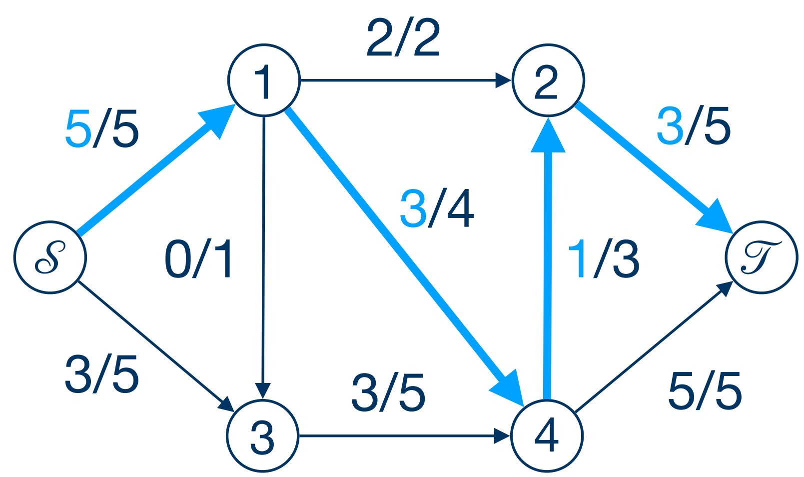
Etape 3 : $|f| = 7$



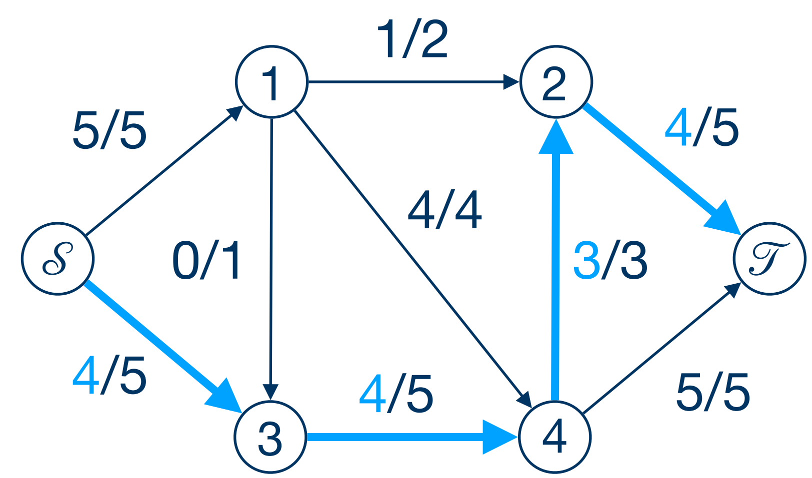
Etape 3b : $|f| = 6$



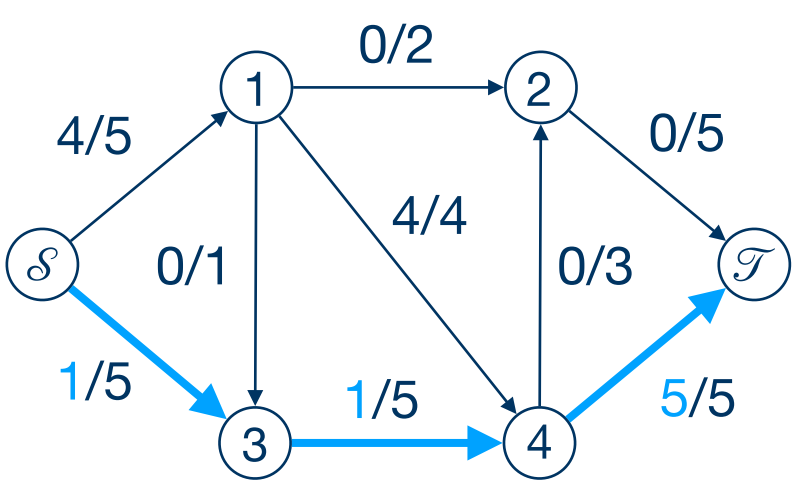
Etape 1 : $|f| = 4$



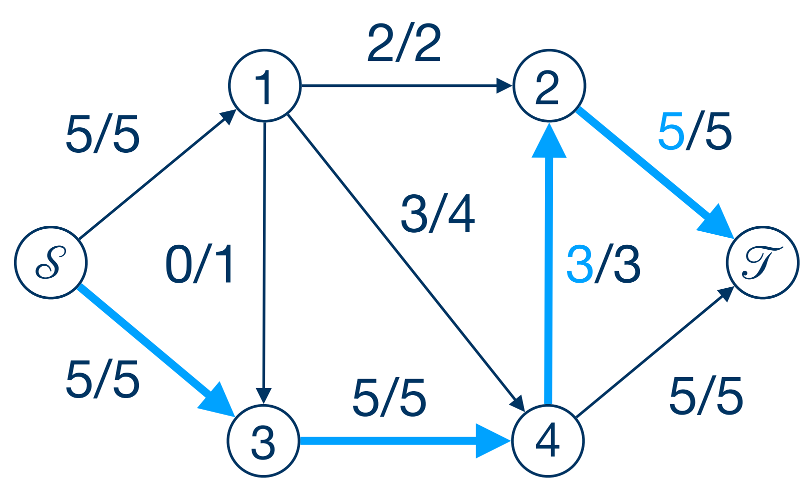
Etape 4 : $|f| = 8$



Etape 4b : $|f| = 9$



Etape 2 : $|f| = 5$



Etape 5 : $|f| = 10 = |f^\star|$

```
let dfs g =
    let n = Array.length g in
    let pere = Array.make n (-1) in
    pere.(0) <- 0;
    let rec traitement i =
        for j = 0 to n - 1 do
            if g.(i).(j) > 0 && pere.(j) = (-1) then (
                pere.(j) <- i;
                traitement j)
        done
    in
        traitement 0; pere;;

let chemin parcours_func g =
    let n = Array.length g and pere = parcours_func g in
    if pere.(n - 1) = (-1) then None
    else
        let rec aux v gamma =
            let gamma' = v::gamma in
            if v = 0 then Some gamma'
            else aux pere.(v) gamma'
        in aux (n - 1) [];;

let ford_fulkerson g parcours_func =
    let n = Array.length g in
    let f = Array.make_matrix n n 0
    and rf = Array.make_matrix n n 0 in

    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            rf.(i).(j) <- g.(i).(j)
        done
    done;

    let rec calcul_delta chemin =
        match chemin with
        | [u;v] -> rf.(u).(v)
        | u :: v :: chemin' -> min (rf.(u).(v)) (calcul_delta (v::chemin'))
        | _ -> failwith "delta : chemin trop court"

    in
    let rec update_flow delta chemin = (*Mise à jour de f et rf*)
        match chemin with
        | u :: v :: chemin' ->
            if g.(u).(v) > 0 then (            (*arête directe dans Rf*)
                f.(u).(v) <- f.(u).(v) + delta;
                rf.(u).(v) <- g.(u).(v) - f.(u).(v);
                rf.(v).(u) <- f.(u).(v))
            else (                             (*arête indirecte dans Rf*)
                f.(v).(u) <- f.(v).(u) - delta;
                rf.(u).(v) <- f.(v).(u);
                rf.(v).(u) <- g.(v).(u) - f.(v).(u))
        |_ -> ()

    in
    let rec loop () =
        match chemin parcours_func rf with
        | None -> f
        | Some chem ->
            let delta = calcul_delta chem
            in update_flow delta chem; loop ()
    in loop ();;
```

```
let bfs g =
    let n = Array.length g in
    let pere = Array.make n (-1) in
    pere.(0) <- 0;
    let rec parcours l1 =
        let l2 = ref [] in
        List.iter
        (fun i ->
            for j = 0 to n - 1 do
                if g.(i).(j) > 0 && pere.(j) = (-1) then (
                pere.(j) <- i;
                l2 := j :: !l2)
            done) l1;
        if !l2 <> [] then parcours !l2
    in
    parcours [0]; pere;;
```

```python
class SegmentedImage(object):
    def __init__(self, image_path, OBJECT, BACKGROUND):
        self.img = Image.open(image_path)
        self.img = self.img.convert("RGB")
        #self.img = ImageOps.grayscale(self.im)
        self.w, self.h = self.img.size

        # Caching the pixel values because Image.getpixel is really slow
        self.pixel_values = {p: self.img.getpixel(p) for p in self.pixels()}

        #User input
        self.obj_seeds = OBJECT
        self.back_seeds = BACKGROUND
        # Constantes
        self.LAMBDA = 0.2
        self.SIGMA = 50
        self.PX_SMOOTHING = 2

        #Utilitaire
        self.utilitaire()

        self.computeHistograms(self.obj_seeds,self.back_seeds)
        self.calculateBoundaryCosts()
        self.calculateRegionalCosts()
        self.createGraph()

    def pixels(self):
        """
        Returns :
            generator_object : contenant tous les couples (x,y) de [0,w]*[0,h]
        """
        for x in range(self.w):
            for y in range(self.h):
                yield (x, y)

    def utilitaire(self): #Conversion Image <-> Graphe
        i = 1
        self.pixel_to_vertex, self.vertex_to_pixel = {}, {}
        for p in self.pixels():
            self.pixel_to_vertex[p] = i
            self.vertex_to_pixel[i] = p
            i += 1
        return self.pixel_to_vertex, self.vertex_to_pixel


    def neighbours_8(self, x, y):
        """
        Arguments :
            x (int) : abscisse du pixel
            y (int) : ordonnée du pixel

        Returns :
            generator_object : generateur des 4 voisins du pixel pour
                               former un 8-voisinage
        """
        return ((i, j) for (i, j) in
               [(x+1, y), (x, y+1), (x+1, y+1), (x+1, y-1)]
              if 0 <= i < self.w and 0 <= j < self.h and (i != x or j != y))
```

```python
    def distance(self,p_a,p_b):
        """
        Arguments :
            p_a : pixel
            p_b : pixel

        Returns :
            float : distance euclidienne entre deux pixels
        """
        return abs(p_a[0] - p_b[0]) + abs(p_a[1] - p_b[1])

    def i_delta(self,a,b,fun):
        """
        Arguments :
            a (r,g,b)
            b (r,g,b)
            fun (int) : determine la fonction de calcul de i_delta
        Returns :
            float : i_delta
        """
        match fun:
            case 0:
                return self.L(*a) - self.L(*b)
            case 1:
                return sum([abs(a[i] - b[i]) for i in range(len(a))])
            case 2:
                return sum([255/(1+ abs(a[i] - b[i])) for i in range(len(a))])
            case 3:
                return sum([a[i] + b[i] for i in range(len(a))])
            case 4:
                return sqrt(sum([(a[i] - b[i])**2 for i in range(len(a))]))
            case 5:
                return sqrt(sum([v**2 for v in self.vectorial_product(a,b)]))


    def L(self,R,G,B):
        """
        Renvoie la couleur d'un pixel en noir et blanc perçue par l'oeil humain
        """
        return int(R * 299/1000 + G * 587/1000 + B * 114/1000)

    def vectorial_product(self,a,b):
        (xa,ya,za),(xb,yb,zb) = a,b
        return (ya*zb - yb*za, xb*za-xa*zb, xa*yb - ya*xb)

    ############### CALCUL DU COUT DE FRONTIERE ###############

    def B_pq(self, p_a, p_b):
        """
        Arguments :
            p_a : pixel (xa,ya)
            p_b : pixel (xb,yb)

        Returns :
            float : cout de frontiere entre les deux pixels
        """
        delta = self.i_delta(self.pixel_values[p_a], self.pixel_values[p_b], 3)
        return exp(- delta**2 / (2 * self.SIGMA**2)) / self.distance(p_a,p_b)
```

```python
    def calculateBoundaryCosts(self):
        self.boundary_costs = {}
        for p in self.pixels():
            self.boundary_costs[p] = {}
            for n_p in self.neighbours_8(*p):
                self.boundary_costs[p][n_p] = self.B_pq(p, n_p)

        # Calcul de K
        self.K = 1. + max(sum(self.boundary_costs[p].values()) for p in
        self.pixels())

    ################ CALCUL DU COUT DE REGION ################

    def getHistogram(self, points):
        """
        Arguments :
            points : int array in range(0,256)
        Returns :
            dict : probabilité normalisée, telle qu'aucune valeur ne soit nulle
        """
        values_count = Counter(self.L(*self.pixel_values[p]) for p in points)
        for intensite in range(256):
            if intensite not in values_count.keys():
                values_count[intensite] = 1/256
        return {value : float(count) / (len(points)) for value, count in
               values_count.items()}


    def gaussian_smoothing(self,histogram, sigma=1.0):
        """
        Lissage de l'histogramme avec un noyau gaussien

        Arguments :
            histogram (array) : Input histogram.
            sigma (float, optional): std of the Gaussian kernel

        Returns :
            array : Smoothed histogram
        """
        histogram = np.array(histogram, dtype=np.float32)
        kernel = np.exp(-0.5 * np.arange(-5 * sigma, 5 * sigma + 1)**2 /
        sigma**2)
        smoothed = convolve1d(histogram, kernel, mode='reflect') / kernel.sum()
        return smoothed

    def computeHistograms(self,obj,bkg):
        self.bkgh = self.getHistogram(bkg)
        self.objh = self.getHistogram(obj)
        intensite = [i for i in range(255)]
        self.bkg_hist = self.gaussian_smoothing([self.bkgh[i] for i in
                          intensite if i in self.bkgh.keys()], self.PX_SMOOTHING)
        self.obj_hist = self.gaussian_smoothing([self.objh[i] for i in
                          intensite if i in self.objh.keys()], self.PX_SMOOTHING)

    def R_p(self, point, hist):
        """
        Arguments :
            point : pixel (x,y)
            hist : array histogram

        Returns :
            float : cout de région du pixel de coordonnées (x,y)
        """
        Ip = self.L(*self.pixel_values[point])
        proba = hist[Ip]
        return - self.LAMBDA * log(proba)
```

```python
    def calculateRegionalCosts(self):
        self.regional_penalty_obj = {p: 0 if p in self.obj_seeds else self.K if p
         in self.back_seeds else self.R_p(p,self.obj_hist) for p in self.pixels()}
        self.regional_penalty_bkg = {p: self.K if p in self.obj_seeds else 0 if p
         in self.back_seeds else self.R_p(p,self.bkg_hist) for p in self.pixels()}


    def createGraph(self):
        self.graph = nx.Graph()
        g = self.graph

        #Creation de P
        for p in self.pixels():
            g.add_node(self.pixel_to_vertex[p])

        #Creation de la source et du puit
        self.s = 0
        self.t = self.w*self.h + 1
        g.add_node(self.s)
        g.add_node(self.t)

        #Creation des aretes de frontière:
        for x in range(self.w):
            for y in range(self.h):
                p = (x,y)
                for n_p in self.neighbours_8(x,y):
                    g.add_edge(self.pixel_to_vertex[p],
                     self.pixel_to_vertex[n_p], capacity=self.boundary_costs[p]
                     [n_p])

        #Creation des aretes de région:
        for p in self.pixels():
            g.add_edge(self.s, self.pixel_to_vertex[p],
             capacity=self.regional_penalty_obj[p])
            g.add_edge(self.pixel_to_vertex[p], self.t, capacity =
             self.regional_penalty_bkg[p])

    def cut(self):
        cut = nx.minimum_cut(self.graph, self.s, self.t,
        flow_func=shortest_augmenting_path)
        return cut[1]

    def generate_mask(self, partition):
        S,T = partition
        imarray = np.zeros((self.h,self.w), dtype=np.uint8)
        for vertex in T:
            if vertex != self.t and vertex != 0:
                (x,y) = self.vertex_to_pixel[vertex]
                imarray[y][x] = 255
        return Image.fromarray(imarray,mode='L').convert('1')
```

```python
import numpy as np
from math import sin, cos, tan, radians, pi
from PIL import Image


class solidAngle():
    def __init__(self, image, cameraPos, cameraRotation, FoV, volumeSize,
volumeOrigin, dVolume):
        #Camera
        self.cameraPos = np.array(cameraPos)
        self.cameraRx,self.cameraRy,self.cameraRz = cameraRotation
        self.image = image
        self.imageW, self.imageH = self.image.size
        self.FoV = FoV
        self.FoVY = FoV*self.imageH/self.imageW
        self.imageGlobalWidth = 2*tan(self.FoV/2)
        self.imageGlobalHeight = 2*tan(self.FoVY/2)
        self.dLocalX = self.imageGlobalWidth/self.imageW
        self.dLocalY = self.imageGlobalHeight/self.imageH
        self.dLoc = (self.dLocalX + self.dLocalY) / 2
        self.topLeftPixelVectorX =
            self.cameraPos[0]-sin(self.cameraRz)-cos(self.cameraRz)*tan(self.FoV/2)
        self.topLeftPixelVectorY =
            self.cameraPos[1]+cos(self.cameraRz)-sin(self.cameraRz)*tan(self.FoV/2)
        self.topLeftPixelVectorZ = self.cameraPos[2]+tan(self.FoVY/2)
        self.topLeftPixelVector =
             np.array([self.topLeftPixelVectorX,self.topLeftPixelVectorY,
            self.topLeftPixelVectorZ])-self.cameraPos
        self.topLeftPixelPos = self.topLeftPixelVector + self.cameraPos
        #Volume
        self.vX, self.vY, self.vZ = volumeSize
        self.x0, self.y0, self.z0 = volumeOrigin
        self.dx, self.dy, self.dz = dVolume
        self.d = (self.dx + self.dy + self.dz)/3
        #Utilitaire
        self.createConversionDict()
        self.computeGlobalCoordPixel()
        self.shapeFromImage()

    def computeGlobalCoordPixel(self):
        """
        Cree et remplis la matrice contenant les coordonnées dans R^3 des pixels
        en partant du pixel en haut à gauche et en itérant
        """
        self.globalCoordPixel = np.zeros((self.imageH+1,self.imageW+1,3))
        for i in range(self.imageH+1):
            for j in range(self.imageW+1):
                delta =
                         [j*self.dLoc*cos(self.cameraRz),
                         j*self.dLoc*sin(self.cameraRz), (-i)*self.dLoc]
                self.globalCoordPixel[i][j] =
                 np.around(self.topLeftPixelPos + np.array(delta),decimals=16)

    def createConversionDict(self):
        i = 0
        self.voxel_to_vertex, self.vertex_to_voxel = {}, {}
        for p in self.voxel():
            self.voxel_to_vertex[p] = i
            self.vertex_to_voxel[i] = p
            i += 1
        return self.voxel_to_vertex, self.vertex_to_voxel
```

```python
    def voxel(self):
        """
        Returns :
            generator_object : contenant tous les couples (x,y,z) de vertex du
             volume w*d*h
        """
        for z in range(self.vZ):
            for y in range(self.vY):
                for x in range(self.vX):
                    yield (x,y,z)


    def vertexAbovePlane(self, planVector1, planVector2, vertex):
        """
        Indique si le point est au dessus du plan engendré par v1 et v2 en
         regardant le signe du déterminant

        Arguments :
            planVector1 (np.array) : 3x1 numpy array representant v1
            planVector2 (np.array) : 3x1 numpy array representant v2
            vertex (list) : coordonnees cartésiennes du vertex [x,y,z]

        Returns :
            bool: le vertex est au dessus du plan
        """
        return np.linalg.det(
             np.array([planVector1,planVector2,np.array(vertex)-self.cameraPos])
             ) > 0

    def shapeFromImage(self):
        """
        Renvoie les coordonnées des pixels de l'image appartenant à l'objet
        L'image utilisée est en fait un masque binaire renvoyé par la
         segmentation de l'image 2D
        """
        imageArray = np.array(self.image)
        self.shapeArray = []
        for i in range(len(imageArray)):
            for j in range(len(imageArray[0])):
                if imageArray[i][j] == 255:
                    self.shapeArray.append([j,i]) #on inverse car on passe en
                                                #coordonnées locales image
```

```python
    def vertexInsideSolidAngle(self,pixel):
        """
        On regarde si le vertex est compris dans la pyramide de sommet l'origine de la caméra et de base le pixel

        Arguments :
            pixel (list) : [j,i] renvoyé par shapeFromImage(self)

        Returns :
            vertexList (list) : liste des pixels dans l'angle solide

        """
        vertexList = []

        if pixel[1] > 0 and pixel[1] < self.imageH and pixel[0] > 0 and pixel[0] < self.imageW:
            topleft = self.globalCoordPixel[pixel[1]][pixel[0]] - self.cameraPos
            topright = self.globalCoordPixel[pixel[1]][pixel[0]+1] - self.cameraPos
            botleft = self.globalCoordPixel[pixel[1]+1][pixel[0]] - self.cameraPos
            botright = self.globalCoordPixel[pixel[1]+1][pixel[0]+1] - self.cameraPos
            for (x,y,z) in self.voxel():
                point = [self.x0 + x*self.d, self.y0 + y*self.d, self.z0 + z*self.d]
                if self.vertexAbovePlane(botleft,topleft,point): #plan de gauche
                    if self.vertexAbovePlane(topleft,topright,point): #plan du haut
                        if self.vertexAbovePlane(botright,botleft,point): #plan du bas
                            if self.vertexAbovePlane(topright,botright,point): #plan de droite
                                vertexList.append(self.voxel_to_vertex[(x,y,z)])
        return vertexList

    def vertexInsideShape(self):
        """
        Pour chaque pixel de la silhouette, on regarde quels sont les vertex dans l'angle solide

        Returns :
            vertexSet (set) : ensemble des vertex dans l'angle solide de la silhouette
        """
        vertexSet = set()
        for pixel in self.shapeArray:
            inside = self.vertexInsideSolidAngle(pixel)
            for voxelId in inside:
                vertexSet.add(voxelId)
        return vertexSet
```

```python
import numpy as np
import networkx as nx
from math import sqrt, exp, pi, log, sin, cos, tan, radians
from PIL import Image, ImageOps
from solidAngle import solidAngle

class Reconstruction(object):

    def __init__(self, shapeList, cameraPos, cameraRot, cameraDist, FoV,
volumeSize, volumeOrigin, dVolume):
        """
        Parameters
        ----------

        shapeList : PIL.Image_object array
            Liste contenant les objets images de chaque photo
        cameraPos : (float array) array
            tableau contenant pour chaque camera sa position suivant les 3
             axes
        cameraRot : (float array) array
            tableau contenant pour chaque camera sa rotation en radians
             suivant les axes
        cameraDist : float
            distance fixe de la camera à l'objet sur toutes les photos
        FoV : float (radians)
            FoV de la caméra
        volumeSize : int array [width, depth, height]
            nombre de point selon chaque axe pour la reconstruction
        volumeOrigin : float array : [x0, y0, z0]
            position du premier point de la reconstruction
        dVolume : float array : [dx, dy, dz]
            espacement des points de la reconstruction suivant les 3 axes

        """
        self.radius = cameraDist
        self.cameraRot = cameraRot
        self.w, self.d, self.h = volumeSize #Garder w*d*h <= 200 000
        self.volumeSize = [self.w, self.d, self.h]
        self.volumeOrigin = volumeOrigin
        self.dVolume = dVolume
        self.nbCamera = len(shapeList)
        self.cameraPos = cameraPos
        self.shapeList = shapeList
        self.FoV = FoV

        #Constantes
        self.REGIONALPENALTYBKG = 255
        self.LAMBDA = -1.1
        self.ALPHA = 0.1
        self.NU = 1.4

        self.createConversionDict()
        self.calculate_Ov()
        self.createGraph()
    def voxel(self):
        """
        Returns :
            generator_object : contenant tous les couples (x,y,z) de vertex
             du volume w*d*h
        """
        for z in range(self.h):
            for y in range(self.d):
                for x in range(self.w):
                    yield (x,y,z)
```

```python
    def createConversionDict(self): #Conversion Volume <-> Graphe
        """
        Crée deux dictionnaires de conversion voxel(volume) en vertex(graphe)
        """
        i = 0
        self.voxel_to_vertex, self.vertex_to_voxel = {}, {}
        for p in self.voxel():
            self.voxel_to_vertex[p] = i
            self.vertex_to_voxel[i] = p
            i += 1
        return self.voxel_to_vertex, self.vertex_to_voxel

    def neighbours6(self, x, y, z): #les 6 voisins, on en calcule seulement 3.
        """
        Arguments :
            x, y, z : coordonnées du voxel

        Returns :
            (tuple comprehension) : tuple contenant toutes les coordonnées des
             voisins accessibles dans un voisinage 6 (3D)
        """
        return ((i, j, k) for (i, j, k) in [(x+1, y, z), (x, y+1, z), (x, y, z+1)]
                  if 0 <= i < self.w and 0 <= j < self.d and 0 <= k < self.h and
                    i != x or j != y or k != z))

    def regionalPenaltyObj(self, voxel):
        n = self.O[self.voxel_to_vertex[voxel]]
        return self.LAMBDA * (1 - exp(n / self.NU))
        #return 400*n**2 / self.nbCamera

    def boundaryPenalty(self, voxel1, voxel2):
        N1 = self.O[self.voxel_to_vertex[voxel1]]
        N2 = self.O[self.voxel_to_vertex[voxel2]]
        return self.ALPHA * N1 * N2 / (self.avg_O_v)**2

    def calculate_Ov(self):
        """
        Calcul du nombre de camera voyant chaque voxel

            O (dict) : dictionnaire contenant O[voxel] = NbCamera voyant le voxel
            avg_O_v (float) : valeur moyenne du nombre de caméra voyant un voxel
             quelconque
        """
        self.O = dict()
        self.avg_O_v = 0
        for vertex in range(self.w*self.h*self.w):
            self.O[vertex] = 0
        for i in range(self.nbCamera):
            s = solidAngle(self.shapeList[i], self.cameraPos[I],self.cameraRot[i],
                 self.FoV, self.volumeSize, self.volumeOrigin, self.dVolume)
            vertexSet = s.vertexInsideShape()

            for vertex in vertexSet:
                self.O[vertex] += 1
                self.avg_O_v += 1
        self.avg_O_v /= len(self.O.keys())
```

```python
def createGraph(self):

    self.graph = nx.Graph()
    g = self.graph

    #Creation du volume de reconstruction : (ensemble V)
    for v in self.voxel():
        g.add_node(self.voxel_to_vertex[v])

    self.s = self.w*self.h*self.d + 2 #le noeud s ne prend pas la valeur 0 car
    self.t = self.w*self.h*self.d + 1 #la numérotation des vertex débute à 0
    g.add_node(self.s)
    g.add_node(self.t)

    #Creation de E:
    for v in self.voxel():
        #Aretes de frontière
        vert = self.voxel_to_vertex[v]
        for n_v in self.neighbours6(*v):
                neigh = self.voxel_to_vertex[n_v]
                g.add_edge(vert, neigh, capacity =
                                              self.boundaryPenalty(v, n_v))
        #Aretes de région
        g.add_edge(self.s, vert, capacity = self.regionalPenaltyObj(v))
        g.add_edge(vert, self.t, capacity = self.REGIONALPENALTYBKG)


def cut(self):
    self.cut = nx.minimum_cut(self.graph, self.s, self.t)
    self.cut = self.cut[1]
    self.object =
         [self.vertex_to_voxel[v] for v in self.cut[0] if v != self.s]
    self.background =
         [self.vertex_to_voxel[v] for v in self.cut[1] if v != self.t]
    return self.background, self.object
```

```python
from tkinter import *
from PIL import Image, ImageTk
from tkinter.ttk import Button, Style
import pickle

def interactive_segmentation(list_url):
    global loaded_image
    loaded_image = []
    global mode
    mode = False  # True = background, False = object
    global current_image
    current_image = 0
    global photo_images  # Store PhotoImage objects in a list
    photo_images = []
    background_list = [set() for _ in range(len(list_url))]
    object_list = [set() for _ in range(len(list_url))]

    def get_x_and_y(event):
        global lasx, lasy
        lasx, lasy = event.x, event.y

    def change_mode():
        global mode
        mode = not mode
        mode_label.config(text="Mode: " + ("background" if mode else "object"))

    def draw_smth(event):
        global lasx, lasy, mode, current_image
        if mode:
            color = "blue"
            current_list = background_list[current_image]
        else:
            color = "red"
            current_list = object_list[current_image]

        canvas.create_line((lasx, lasy, event.x, event.y),
                           fill=color,
                           width=2)
        lasx, lasy = event.x, event.y
        current_list.add((lasx, lasy))

    def done():
        app.quit()

    def on_closing():
        app.quit()

    def go_left():
        global current_image
        global loaded_image
        current_image = (current_image - 1) % len(list_url)
        canvas.delete("all")  # Remove all items from the canvas
        image = loaded_image[current_image]
        canvas.create_image(0, 0, anchor="nw", image=image)
        redraw_drawings()

    def go_right():
        global current_image
        global loaded_image
        current_image = (current_image + 1) % len(list_url)
        canvas.delete("all")  # Remove all items from the canvas
        image = loaded_image[current_image]
        canvas.create_image(0, 0, anchor="nw", image=image)
        redraw_drawings()
    def redraw_drawings():
        for point in background_list[current_image]:
            x, y = point
            canvas.create_line(x-1, y-1, x+1, y+1, fill="blue", width=2)
        for point in object_list[current_image]:
            x, y = point
            canvas.create_line(x-1, y-1, x+1, y+1, fill="red", width=2)

    app = Tk()
    app.geometry("400x400")

    app.grid_rowconfigure(0, weight=1)
    app.grid_columnconfigure(0, weight=1)

    style = Style()
    style.theme_use("aqua")

    canvas = Canvas(app, bg='black')
    canvas.grid(row=0, column=0, columnspan=3, padx=10, pady=(30, 0), sticky="nsew"


    canvas.bind("<Button-1>", get_x_and_y)
    canvas.bind("<B1-Motion>", draw_smth)

    for file_path in list_url:
        image = Image.open(file_path)
        photo_image = ImageTk.PhotoImage(image)
        loaded_image.append(photo_image)
        canvas.photo_image = photo_image

    image_container = canvas.create_image(0, 0, anchor="nw", image=loaded_image[0])
    canvas.image = loaded_image[0]


    left_button = Button(app, text="<<<", command=go_left)
    right_button = Button(app, text=">>>", command=go_right)
    mode_button = Button(app, text="Mode", command=change_mode)
    done_button = Button(app, text="Done", command=done)

    mode_label = Label(app, text="Mode: " + ("background" if mode else "object"))


    left_button.grid(row=1, column=0, padx=5, pady=3)
    right_button.grid(row=1, column=1, padx=5, pady=3)
    mode_button.grid(row=1, column=2, padx=5, pady=3)
    done_button.grid(row=2, column=0, padx=5, pady=3, sticky="w")
    mode_label.grid(row=2, column=2, pady=3)


    app.protocol("WM_DELETE_WINDOW", on_closing)
    app.mainloop()

    # Save the background_list and object_list into a file
    with open('lists.pkl', 'wb') as f:
        pickle.dump((background_list, object_list), f)
```