Projet - Segmentation d'image et arbres couvrants

Mathias Bourgoin, **Anthony** Perez, **Sophie** Robert

La *segmentation d'image* est une branche du traitement d'images dont l'objectif est de découper une image donnée en régions, correspondant à des zones de l'image ayant des intensités de pixels similaires.

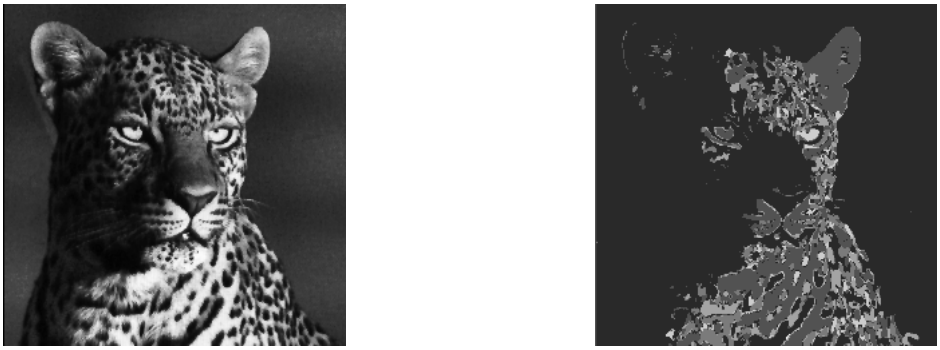

Figure 1 – Segmentation de l'image leopard.pgm obtenue avec l'algorithme présenté dans ce sujet.

De nombreuses méthodes de segmentation d'image existent ; dans ce projet vous implémenterez une méthode basée sur la notion d'*arbre couvrant de poids minimum*. Le principe de cette méthode est assez simple : en partant d'une image donnée, un graphe en forme de *grille* est construit, en prenant pour chaque sommet (*i.e.* pixel) ses huit voisins (voir Figure 4). Un algorithme **similaire** à l'algorithme de **Kruskal** (calculant un arbre couvrant de poids minimum) est alors appliqué, permettant de découper l'image en régions.

Il vous sera demandé *a minima* de :

(i) Implémenter l'algorithme de **Kruskal** en utilisant la structure de données `Union-Find`.

(ii) Valider votre algorithme sur les jeux de données proposés sur celene.

(iii) Modifier l'algorithme et la structure de données (si nécessaire) pour réaliser la segmentation d'image.

(iv) Implémenter l'algorithme de 6-coloration des régions pour obtenir une coloration propre de votre segmentation.

Le choix de(s) structure(s) utilisées pour représenter l'arbre couvrant de poids minimum est laissé libre. Un **bonus** sera cependant accordé si vous implémentez une structure graphe sous forme de **liste d'adjacence**.

Les contraintes. Par souci de simplicité, les images considérées sont en *niveau de gris* au format pgm. Une librairie de lecture et d'écriture d'images pgm vous est fournie, et il est demandé de ne pas la modifier. Ces images sont écrites en binaire, il n'est donc pas possible d'extraire des informations directement depuis les fichiers. Il vous est demandé de respecter l'arborescence contenue dans l'archive celene et de commenter votre code en Doxygen. Vous pouvez rajouter des fichiers issus de précédents TDs si besoin et **devez** travailler en binôme.

Le rendu. Vous rendrez une archive unique, obtenue en exécutant la commande `make rendu`. Le contenu attendu dans le compte-rendu est détaillé à la fin du sujet ; il vous est demandé de le respecter **à la lettre**.

L'algorithme de Kruskal

And also the trees.

Le calcul d'un *arbre couvrant de poids minimum* pour un graphe *valué*[1] $G = (V, E)$ est un problème classique en algorithmique de graphes. Il peut-être résolu très simplement en utilisant des algorithmes **gloutons**, les plus connus étant ceux de Prim [5] et de Kruskal [3]. Lors de la construction de l'arbre couvrant de poids minimum, ces algorithmes maintiennent respectivement un arbre et une *forêt* (un ensemble de plusieurs arbres non reliés entre eux).

1. Les *poids* sont situés sur les arêtes.

Dans le cadre de la segmentation d'image, il est nécessaire de se baser sur l'algorithme de Kruskal, les différents arbres de la forêt correspondant aux régions obtenues sur l'image à la fin de l'exécution de l'algorithme. L'algorithme de Kruskal est présenté (de manière informelle) dans l'Algorithme 1.

Algorithme 1 : Pseudo-code de l'algorithme de Kruskal.

Entrée : Un graphe $G = (V, E)$ et une fonction de poids $\omega : E \to \mathbb{N}$.
Sortie : Un arbre couvrant de poids minimum de G.

```
Trier les arêtes E par ordre croissant de poids;
Ajouter chaque sommet v ∈ V dans un ensemble lui étant propre;
S ← ∅;
pour (u, v) ∈ E faire
    si u et v sont dans différents ensembles alors
        S ← S ∪ {(u, v)};
        Regrouper les ensembles contenant u et v;
Retourner S ;
```

Vous trouverez ci-dessous un exemple d'utilisation de l'algorithme.

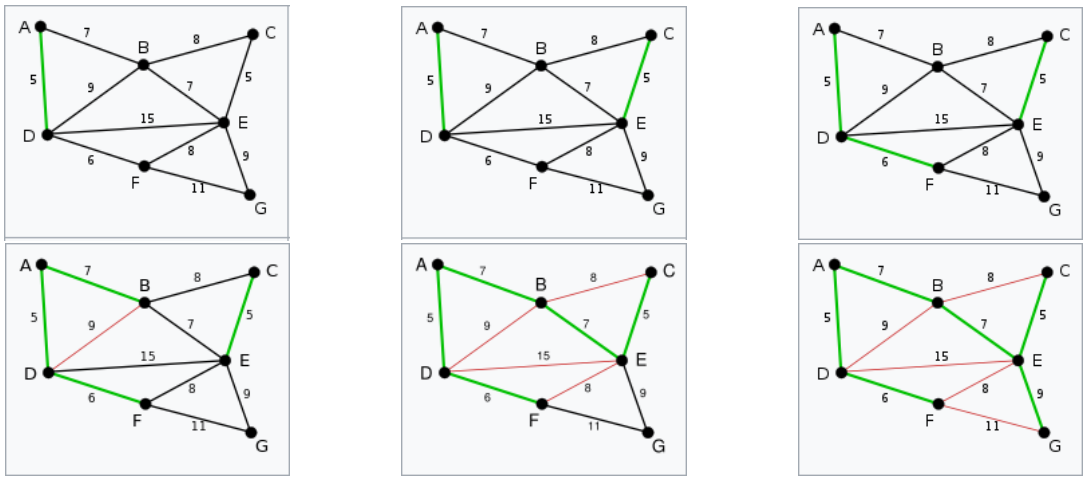

FIGURE 2 – Illustration de l'algorithme de Kruskal sur un graphe à 7 sommets. Les arêtes vertes sont prises dans l'arbre couvrant, les rouges en sont écartées. ©WIKIPEDIA

La structure de données

The way to the forest.

L'efficacité et la complexité de l'algorithme de Kruskal dépendent de l'implémentation réalisée, et donc de la structure de données utilisée. Mal codées, les opérations de recherche des ensembles contenant u et v et d'union desdits ensembles peuvent être très lentes. Pour réaliser une implémentation relativement efficace, il vous est demandé d'implémenter une structure de données de type `Union-Find`, représentant une **partition d'un ensemble**. L'utilisation de cette structure sur l'exemple de l'algorithme de Kruskal présenté Figure 2 est donné Figure 3.

Cette structure possède les primitives suivantes :

- `Trouver` permettant de déterminer dans quelle partie un élément se situe ;
- `Unifier` permettant de regrouper deux parties en une seule ;
- `Creer_ensemble` permettant de créer une partie contenant un seul élément.

Pour définir ces opérations de manière plus précise, il est nécessaire de trouver un moyen de *représenter* les parties. L'approche la plus simple consiste à choisir un *représentant* pour chaque partie, permettant d'identifier la partie entière. Ainsi, un appel à `Trouver(u)` retourne le représentant de la partie contenant u. Il vous est demandé de réaliser deux implémentations de cette structure de données :

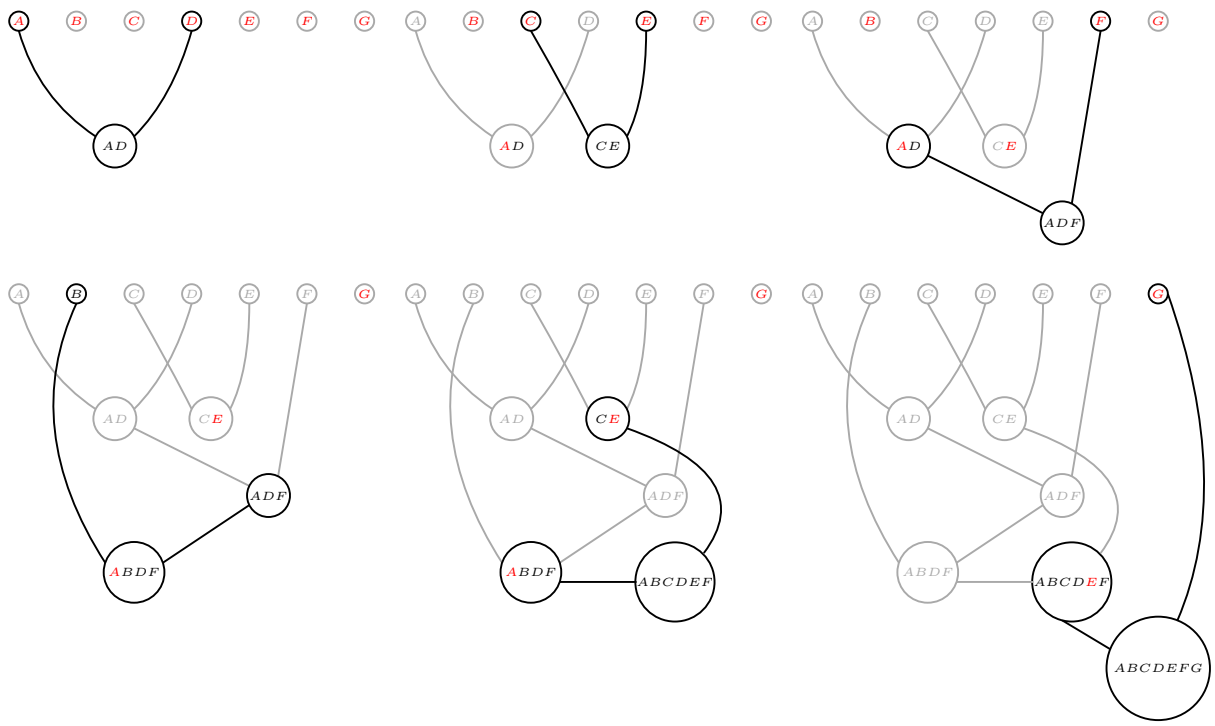

FIGURE 3 – Initialisation et maintien d'une structure de données `Union-Find` pour l'algorithme de Kruskal présenté Figure 2. Les représentants de chaque ensemble sont indiqués en rouge.

1. *liste simplement chaînée* : le représentant est alors l'élément en tête de liste.
2. *forêt* : les parties sont maintenant représentées par des arbres dans lesquels chaque noeud contient une référence vers son noeud père. Le représentant de chaque partie est alors la racine de l'arbre.

La structure de données `Union-Find` implémentée via des forêts peut être aussi mauvaise que la version liste, notamment parce que les arbres ainsi créés peuvent être très déséquilibrés. Deux précautions permettent cependant de contourner ce problème :

- *Toujours attacher le plus petit arbre au plus grand* : pour ce faire, on utilise la notion de *rang*, valant 0 pour les arbres contenant un seul élément, et augmentant de 1 chaque fois que deux arbres de même rang sont unis.
- *Compresser les chemins* : un appel de `Trouver` consiste maintenant à retourner la racine de l'arbre, qu'il faut donc parcourir... à chaque fois ! Pour éviter cela, il suffit de faire pointer *chaque* noeud de l'arbre vers la racine.

Remarques
- Les forêts sont utilisées comme moyen de *représentation*, mais peuvent *s'implémenter* sans utiliser la notion d'arbres.
- Ces deux idées peuvent également s'appliquer à l'implémentation par liste chaînée.

À faire. Vous implémenterez la structure de données `Union-Find` en utilisant *au moins* une des deux représentations données ci-dessus et vous vous en servirez pour implémenter l'algorithme de Kruskal. Implémenter les deux versions donnera lieu à un bonus. Il vous sera demandé de préciser dans le rapport les complexités des primitives en fonction de la structure de données, et les améliorations que vous avez apportées à l'implémentation par liste simplement chaînée (si elles existent). Votre implémentation doit de plus permettre de vérifier les propriétés suivantes :

$\mathcal{P}_1$ L'union des parties contenant u et v doit pouvoir dépendre d'un *critère* donné (ce sera notamment le cas lors de la segmentation). En d'autres termes, votre code doit gérer la possibilité d'ajouter une *fonction d'évaluation supplémentaire* permettant de décider si deux parties doivent être unies ou non.

$\mathcal{P}_2$ Lors du tri des arêtes par poids croissant [2], deux arêtes de même poids devront être triées selon les numéros des sommets qui la composent. En d'autres termes, pour deux arêtes (u, v) et (u', v') de même poids, on a :

$$(u, v) \leqslant (u', v')$$

si $min(u, v) < min(u', v')$ ou $min(u, v) = min(u', v')$ et $max(u, v) \leqslant max(u', v')$.

Pour vérifier votre calcul d'arbre couvrant, vous utiliserez les jeux de données fournis sur CELENE. Votre fichier de sortie décrira l'arbre en écrivant une arête par ligne sous la forme $u\ v\ \omega(u, v)$ avec $u \leqslant v$.

L'algorithme de segmentation

Photoshop boys.

Vous allez maintenant (ré-)utiliser et/ou adapter les implémentations précédentes pour implémenter l'algorithme de **segmentation d'image** [2], se présentant comme suit :

Algorithme 2 : Pseudo-code de l'algorithme de segmentation d'image.

Entrée : Un graphe $G = (V, E)$, une fonction de poids $\omega : E \to \mathbb{N}$ et $k \in \mathbb{N}$.
Sortie : Une segmentation de V en *régions* $\mathcal{S} = \{C_1, \ldots, C_r\}$.

```
Trier E par ordre croissant de poids;
Initialiser S^0 où chaque sommet v_i est seul dans sa propre composante;
pour (v_i, v_j) ∈ E faire
    Soient C_i^{q-1} et C_j^{q-1} les ensembles contenant v_i et v_j, respectivement;
    si C_i^{q-1} ≠ C_j^{q-1} et ω(v_i, v_j) ⩽ MInt(C_i^{q-1}, C_j^{q-1}) alors
        S^q ← Unification de C_i^{q-1} et C_j^{q-1};
    sinon
        S^q ← S^{q-1};
Retourner S^m;
```

Dans l'algorithme, pour deux régions C et C', on a $MInt(C, C') = min(Int(C) + \gamma(C), Int(C') + \gamma(C'))$ où $Int(C)$ correspond à l'arête de poids maximal dans la composante C et $\gamma(C) = \frac{k}{|C|}$, k étant un paramètre de l'algorithme.

Quelques indices. La valeur de $Int(C)$ peut-être très facilement obtenue au moment de l'unification de deux parties, puisqu'il s'agit simplement du poids de l'arête provoquant cette unification. De plus, il vous sera probablement utile de stocker la *taille* de chacune de vos parties durant l'exécution de votre algorithme.

À faire. Vous devez implémenter l'algorithme de segmentation d'image en adaptant l'algorithme de Kruskal précédemment codé. Pour ce faire, vous devrez générer le graphe correspondant à l'image, contenant `largeur` × `hauteur` sommets ayant le voisinage présenté Figure 4, la valuation des arêtes correspondant à la valeur absolue de la différence des intensités des pixels [3].

Pour vérifier votre algorithme de segmentation, vous utiliserez les images fournies dans l'archive proposée sur CELENE. Votre fichier de sortie décrira la segmentation en écrivant une composante par ligne, sans précaution particulière sur l'ordre dans lequel seront affichés les sommets.

FIGURE 4 – Le voisinage d'un pixel donné dans le graphe généré.

2. Vous pouvez utiliser ici une fonction standard.
3. Pour deux sommets adjacents v_i et v_j, le poids de l'arête sera donc de $|p_i - p_j|$.

La coloration de régions

Colour of fire.

Une fois la segmentation de l'image obtenue, il est maintenant nécessaire de colorer *proprement* les régions afin d'obtenir l'image segmentée. Le découpage d'une image en régions donnant lieu à un graphe *planaire*[4], il est possible de le colorer *proprement*[5] avec uniquement 4 couleurs [1, 6]. Cependant, l'algorithme de 4-coloration étant relativement compliqué, il vous est demandé d'implémenter un algorithme de 6-coloration [4], plus simple et suffisant pour les besoins de ce projet.

Algorithme 3 : Algorithme de 6-coloration d'un graphe planaire [4].

Entrée : Un graphe planaire $G = (V, E)$.
Sortie : Une 6-coloration propre de G.

pour i *de* 0 *à* $|V| - 1$ **faire**
  $Deg[j] \leftarrow$ liste doublement chaînée contenant tous les sommets de degré j;
pour i *de* $|V| - 1$ *à* 0 **faire**
  $v[i] \leftarrow$ le premier sommet de la liste $Deg[j]$, avec j plus petit indice tel que $Deg[j]$ n'est pas vide;
  Supprimer $v[i]$ de $Deg[j]$;
  pour v' *voisin de* $v[i]$ **faire**
    Supprimer (si possible) v' de la liste $Deg[k]$ le contenant, et l'ajouter à la liste $Deg[k-1]$;
pour i *de* 0 *à* $|V| - 1$ **faire**
  Colorer le sommet $v[i]$ avec la plus petite couleur (comprise entre 1 et 6) non présente dans son voisinage;

Pour garder une cohérence sur les images obtenues, vous utiliserez l'ensemble de couleurs :

$$\{41, 82, 123, 164, 205, 254\}$$

Voici quelques exemples de coloration d'images obtenues avec cet algorithme, en fonction du paramètre k indiqué en légende.

FIGURE 5 – La segmentation du léopard a été réalisée avec $k = 50$, celle du caméraman avec $k = 40$.

4. Pouvant être dessiné dans le plan sans croisement d'arêtes.
5. Une coloration est *propre* si deux sommets reliés par une arête n'ont pas la même couleur.

À faire. Vous devez générer le graphe d'adjacence des régions retournées par l'algorithme de segmentation. Il vous est de plus demandé de générer l'image obtenue avec une telle coloration en utilisant la librairie de gestion d'images PGM fournie.

Le compte-rendu *Reportivo.*

Vous justifierez succintement le choix de vos structures de données (et de vos implémentations) dans le compte-rendu. Il vous est de plus demandé d'analyser la complexité des primitives de la structure de données `Union-Find` en fonction de l'implémentation (liste simplement chaînée ou forêt). Vous pouvez évidemment répondre à cette question même si vous n'avez implémenté qu'une seule version de la structure de données.

Vous justifierez également tous les algorithmes que vous avez utilisés (par exemple pour la génération du graphe-grille pour réaliser la segmentation de l'image).

Références

[1] Kenneth Appel and Wolfgang Haken. *Every Planar Map is Four Colorable*, volume 98 of *Contemporary Mathematics*. American Mathematical Society, 1989.

[2] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient graph-based image segmentation. *International journal of computer vision*, 59(2) :167–181, 2004.

[3] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1) :48–50, 1956.

[4] David W Matula, Yossi Shiloach, and Robert E Tarjan. Two linear-time algorithms for five-coloring a planar graph. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1980.

[5] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, 36(6) :1389–1401, 1957.

[6] Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 571–575. ACM, 1996.