

Grand Devoir 2

Structures de données et algorithmes

Graphes & arbres binaires (Graphs & Binary Trees)

Mentions générales

- Pour ce projet vous allez travailler par équipes de 2 personnes (ou seuls).
- Le projet est à rendre sur la plateforme Moodle (curs.upb.ro). S'il y a des problèmes lors du téléchargement vers ou depuis la plateforme, contactez par Teams ou par mail votre assistant des travaux pratiques.
- Le projet est à rendre **au plus tard le 27.05.2024, à 8h**. Aucun retard ne sera accepté.
- Vous recevrez des questions concernant votre solution durant la séance suivante du TP. **Les projets non présentés au laboratoire suivant ne seront pas notés ! Attention, c'est la dernière semaine du semestre, il n'y a donc pas d'autres occasions de le présenter !**
- La rendue finale du projet comportera une archive nommée Etudiant1Nom_Etudiant1Prenom_Etudiant2Nom_Etudiant2Prenom_GD2 avec :
 1. **les fichiers du code source (.cpp et .h)** et non pas des fichiers objets (i.e. *.o) ou des fichiers exécutables (i.e. *.exe) ; **SVP mettez chaque exercice dans un dossier séparé !**
 2. **un fichier de type README** où vous allez spécifier (dans quelques mots) toutes les fonctionnalités de votre projet, avec les instructions pour les employer. Au contraire, s'il y aura des exigences qui ne seront pas fonctionnelles, vous pouvez proposer des idées pour une solution possible (et peut-être obtenir des points supplémentaires).
- Pour toute question concernant le sujet du projet ou les exigences, envoyer un mail/message sur Teams à vos assistants ou utiliser le forum de la plateforme Moodle (il est recommandé de créer un nouveau post dans le groupe Teams afin que tout le monde puisse voir la question et la réponse). N'oubliez pas de mentionner @General (pour notifier tout le monde) ou @NomDuProf.
- Attention : notre équipe utilise des logiciels détectant le plagiat (Moss du Stanford). Dans le cas malheureux de plagiat, **le projet sera noté par un 0 (zéro)**.
- Attention x2 : Il est nécessaire de tester tous les exercices dans le main. Dans le cas où vous ne le faites pas, la note maximale pour chaque exercice sera de 1.

! Observation : Vous pouvez utiliser les structures de données et autres exercices que nous avons utilisés en classe. Alternativement, vous pouvez utiliser les implémentations standard de C++, MAIS pas d'autres implémentations personnalisées trouvées sur internet.

1. Vérificateur de Connectivité Réseau (Network Connectivity Checker) (1p)

Vous disposez d'un réseau de dispositifs connectés par des câbles réseau. Chaque dispositif est représenté par un identifiant entier unique. Certains dispositifs sont connectés directement par des câbles, tandis que d'autres le sont indirectement via des dispositifs intermédiaires. Votre tâche consiste à écrire une fonction pour déterminer si tous les dispositifs du réseau sont connectés, c'est-à-dire s'il existe un chemin entre chaque paire de dispositifs.

La fonction doit prendre les entrées suivantes :

1. Un entier N ($1 \leq N \leq 1000$), représentant le nombre total de dispositifs dans le réseau.

2. Une liste de tuples, chaque tuple contenant deux entiers u et v ($1 \leq u, v \leq N$, $u \neq v$), représentant une connexion directe entre le dispositif u et le dispositif v .

La fonction doit retourner une valeur booléenne : vrai si tous les dispositifs du réseau sont connectés, et faux autrement.

Observations :

- Une connexion directe entre les dispositifs u et v signifie qu'il existe un câble réseau les reliant directement.
- Les connexions indirectes se forment à travers une série de connexions directes. Par exemple, si le dispositif a est connecté au dispositif b , et que le dispositif b est connecté au dispositif c , alors le dispositif a est indirectement connecté au dispositif c .
- Le réseau peut contenir des dispositifs isolés (dispositifs non connectés à d'autres dispositifs).

Exemple :

Entrée

$N = 5$

$connexions = [(1, 2), (2, 3), (3, 4), (4, 5), (5, 1)]$

Sortie

Vrai

Dans cet exemple, tous les dispositifs sont directement ou indirectement connectés les uns aux autres, formant une boucle fermée. Par conséquent, la fonction doit retourner vrai.

Points:

0,25p lecture de l'entrée + construction du graph

0,75p algorithme & sortie correcte

-0,25p si pas de traitement des erreurs / cas particuliers

2. Arbres binaires (Binary trees) (5p)

Andrei et Sebi veulent échanger des messages textuels. Pour rendre les choses plus intéressantes, ils ont choisi d'envoyer leurs messages codés en bits et de les décoder à la réception. Au lieu d'utiliser l'encodage UTF-8, qui pourrait les amener à utiliser 32 bits pour un seul caractère, ils ont commencé à mettre en œuvre une méthode différente, utilisant le nombre d'occurrences de chaque caractère dans le texte.

L'encodage d'un message est le processus de conversion de celui-ci en une forme différente en utilisant un code. **Le code** peut être n'importe quelle autre représentation des données, tant qu'il existe une correspondance **un à un** entre l'entrée et la sortie. Regardons l'exemple suivant, où nous codons les premiers caractères de l'alphabet en utilisant 3 bits :

a	b	c	d	e	f
001	010	011	100	101	110

Les représentations en bits de chaque lettre sont appelées **codewords**.

Maintenant, le message *afbbce* devient *001101010010011101*. C'est ainsi que nous avons **codé** notre message. Pour le **décoder**, nous utilisons le même tableau, mais en inversant le processus. Nous prenons des groupes de 3 bits consécutifs et trouvons la lettre associée à chacun d'eux.

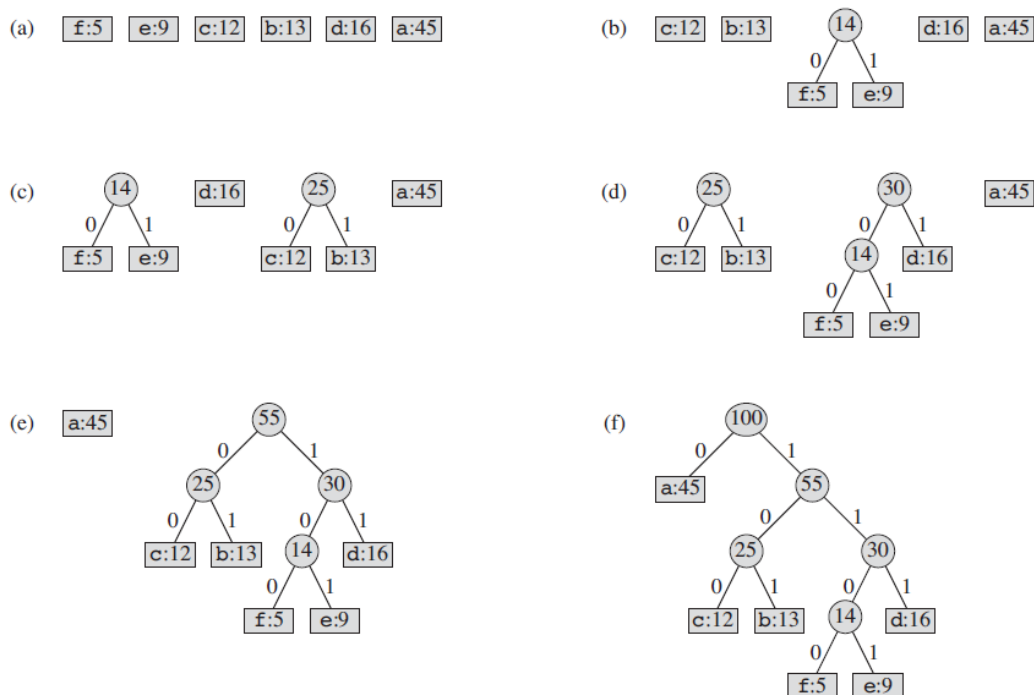
Andrei et Sebi ont pensé à quelque chose de plus intelligent et ont essayé d'utiliser un codage de longueur dynamique basé sur le nombre d'occurrences d'un caractère dans le texte. Ils veulent utiliser des **codes préfixes**, pour lesquels **aucun codeword n'est également préfixe d'un autre codeword**. Par exemple, nous pouvons coder *a* comme *0*, *b* comme *101* et *c* comme *100*. Le message *abc* devient *0101100*. Ils ont réussi à utiliser moins de bits qu'avec des codewords de longueur fixe, tout en préservant également le sens du message.

Pour utiliser cela, ils ont besoin d'un nouvel arbre binaire modifié. Ses feuilles doivent être des nœuds contenant chaque caractère et son nombre d'occurrences dans le texte, triées par nombre d'occurrences. Le codeword pour un caractère est calculé en parcourant le chemin à partir de la racine de l'arbre jusqu'à la feuille contenant ce caractère, où aller à l'enfant de *gauche* ajoute un 0 au codeword résultant, tandis qu'aller à l'enfant de *droite* ajoute un 1.

1. La première étape serait de créer un BST qui stocke, dans chaque nœud, un caractère et son nombre d'apparitions. Ils voudraient également le modifier en ajoutant une nouvelle méthode, appelée **removeMin()**, qui supprime et retourne la plus petite valeur dans ce BST.

2. La partie suivante consisterait à créer l'arbre binaire modifié (appelé **arbre de codage**). Ils commencent avec les deux caractères les moins fréquents dans le texte et les connectent comme feuilles sous le même nœud parent contenant la somme de leurs occurrences. *L'enfant à gauche* sera celui des deux ayant moins d'occurrences. Le nouveau nœud, contenant la somme des occurrences des deux caractères, sera ajouté parmi le reste des feuilles dans le BST stockant les caractères et les occurrences. Nous répétons ensuite le processus en sélectionnant, à partir du BST, les deux prochains nœuds contenant le nombre d'occurrences le plus faible.

Dans l'image suivante, nous pouvons voir ce processus pour une liste donnée de caractères et leurs occurrences dans un texte :



Comme vous pouvez le constater, les premières feuilles sélectionnées correspondent à 'f' et 'e', qui apparaissent respectivement 5 et 9 fois. Nous créons alors leur nœud parent, contenant la somme de leurs occurrences. Le processus est ensuite répété, le nœud parent remplaçant les deux feuilles précédemment sélectionnées dans la collection à partir de laquelle nous choisissons.

Dans cet exemple, le mot de code résultant pour *a* est *0*, pour *b* c'est *101*, pour *c* c'est *100*, et ainsi de suite. Nous pouvons facilement décoder le texte *111100001011100* en *dcaabf* grâce aux codes préfixes (rappelez-vous : aucun codeword n'est également préfixe d'un autre codeword).

Après avoir défini la logique, les deux amis ont commencé à planifier l'implémentation. Ils auraient besoin d'une classe appelée **EncodingTree** qui représenterait leur nouvelle structure de données. Le constructeur de cette classe reçoit en paramètre le BST mentionné dans la première étape et crée l'arbre binaire résultant (vous pouvez représenter les nœuds dans le nouvel arbre binaire de la manière que vous souhaitez).

La classe contiendrait également la méthode **char* encode(char* text)** qui reçoit en paramètre un texte et renvoie sa forme encodée en utilisant l'arbre d'encodage, et une méthode **char* decode(char* code)** qui reçoit le texte encodé et renvoie sa forme décodée.

Votre tâche est d'aider Andrei et Sebi avec leur implémentation !

Détails de la tâche:

1. Le message à envoyer avant l'encodage sera lu à partir d'un fichier appelé **message.in**, tandis que le message encodé sera écrit dans un fichier appelé **encoded.out**.
2. Le message encodé reçu sera lu à partir d'un fichier appelé **encoded.in**, tandis que le message décodé obtenu sera écrit dans un fichier appelé **message.out**.
3. Pour simplifier, les textes ne contiendront que des caractères de l'alphabet, des chiffres et le caractère *espace*.
4. En plus des structures de données mentionnées, vous pouvez utiliser d'autres structures de données dont vous avez besoin, mais vous ne pouvez pas remplacer l'une des structures de données requises par le problème par une autre (cela signifie que vous devez utiliser le BST et l'EncodingTree comme décrits)

Exigences:

1. Implémentez la méthode `removeMin()` pour BST. (0.5p)
2. Obtenez le BST dont les nœuds stockent les caractères et leurs occurrences à partir d'un texte d'entrée donné. (1p)
3. Implémentez le constructeur pour la classe `EncodingTree`. (1p)
4. Implémentez la méthode `encode()`. (1p)
5. Implémentez la méthode `decode()`. (1p)
6. Testez vos méthodes en lisant l'entrée et en écrivant la sortie dans les fichiers spécifiés (voir la section Détails de la tâche) (0.5p)

Conseils et astuces:

1. Avant de commencer l'implémentation, essayez de suivre les étapes manuellement, sur papier, et comprenez le comportement prévu. Ensuite, essayez d'appliquer l'algorithme à différents textes.
2. Suivez les étapes d'implémentation telles qu'elles sont mentionnées dans la section Exigences : commencez par l'ABR, puis continuez avec le constructeur, puis les méthodes d'encodage et de décodage.

3. Segmentation d'image (Image Segmentation) (4p)

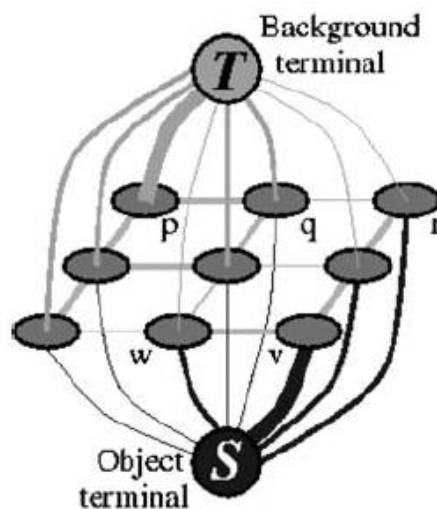
Vous travaillez dans une grande entreprise, Abode, qui crée des programmes de retouche photo. En raison de la forte demande des utilisateurs pour disposer d'un outil de segmentation automatique, votre nouvelle tâche est de créer un tel outil. La première idée qui vous vient à l'esprit est d'utiliser l'apprentissage profond, mais votre chef d'équipe vous dit que "Cela nous coûterait plus de temps et d'énergie à développer cela plutôt que d'utiliser des graphes".

Intrigué, vous commencez à rechercher sur le web et découvrez que les images peuvent être traitées comme des graphes dans lesquels **chaque pixel est un nœud**. De plus, vous avez une idée brillante : la **luminosité des pixels** peut être utilisée comme **poids** pour segmenter l'image. Pour trouver un moyen de découper l'image, vous aurez besoin d'un moyen de transmettre des informations sur la luminosité d'un nœud de départ à un nœud final, en ajoutant ainsi deux nœuds qui sont directement connectés à tous les autres nœuds.

De plus, vous constatez qu'il n'est pas nécessaire de transmettre des informations depuis des nœuds qui ne sont pas directement connectés. Vous ne connectez donc que des nœuds voisins à 4 avec un poids donné par l'idée suivante :

Si la différence entre deux pixels voisins est inférieure à un seuil, ajoutez le poids comme étant $255 - \text{différence}$, sinon 0.

En observant les valeurs transmises, vous voyez que lorsque vous voulez trouver la valeur maximale finale transmise de manière à minimiser les coupes dans le graphe et segmenter une partie de l'image (pour simplifier, nous nous concentrerons uniquement sur l'avant-plan codé comme une valeur minimale dans l'image).



Considérons la matrice suivante de 3x3 :

70 14 30

50 10 15

34 53 78

Le pixel avec la valeur = 10 a les 4 voisins suivants {14,50,53,15}.

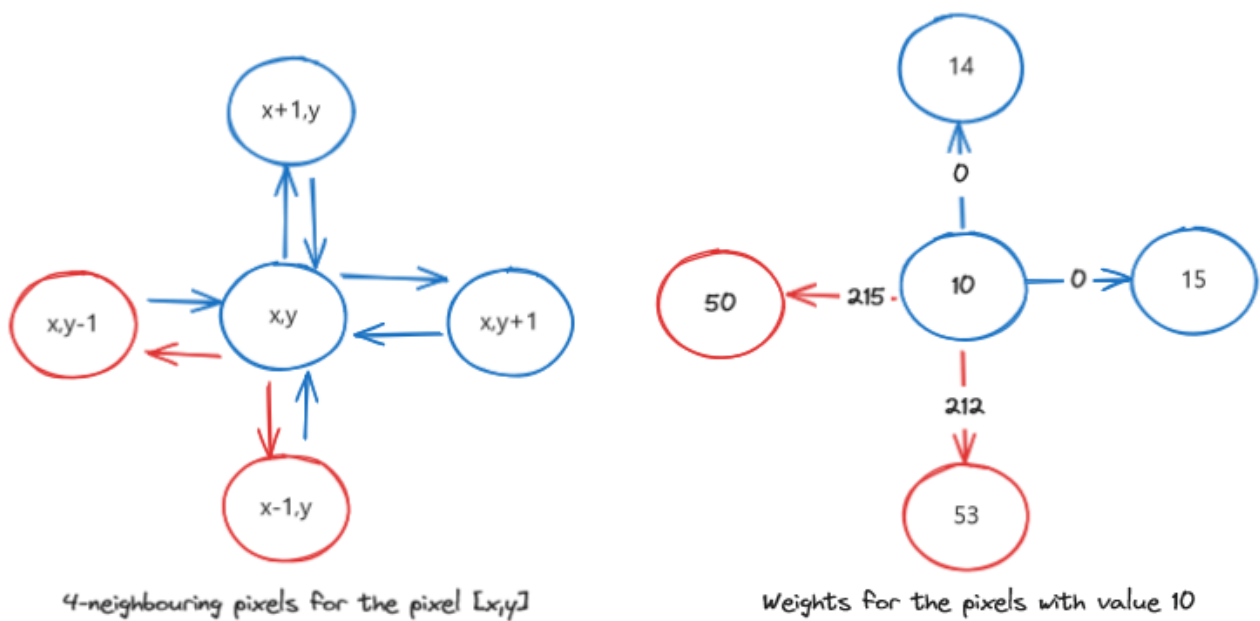
Pour attribuer les poids du pixel =10 aux autres voisins, nous utiliserons la formule donnée de sorte que pour le premier voisin nous aurons :

Pixel = 10, pixel du 1-voisin = 14, la différence entre eux est $|10-14| < 10$ donc le poids = 0 entre le sommet avec le pixel 10 et le sommet avec le pixel 14.

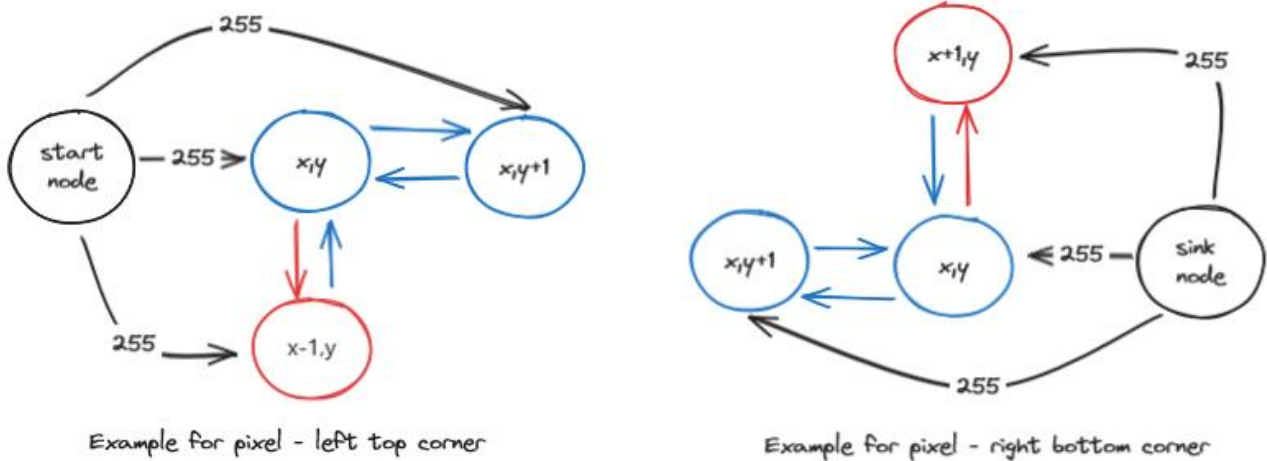
Pixel = 10, pixel du 2-voisin = 15, la différence entre eux $|10-15| < 10$ - poids = 0.

Pixel = 10, pixel du 3-voisin = 53, la différence entre eux $|10-53| > 10$ - poids = $255 - |10-53|$.

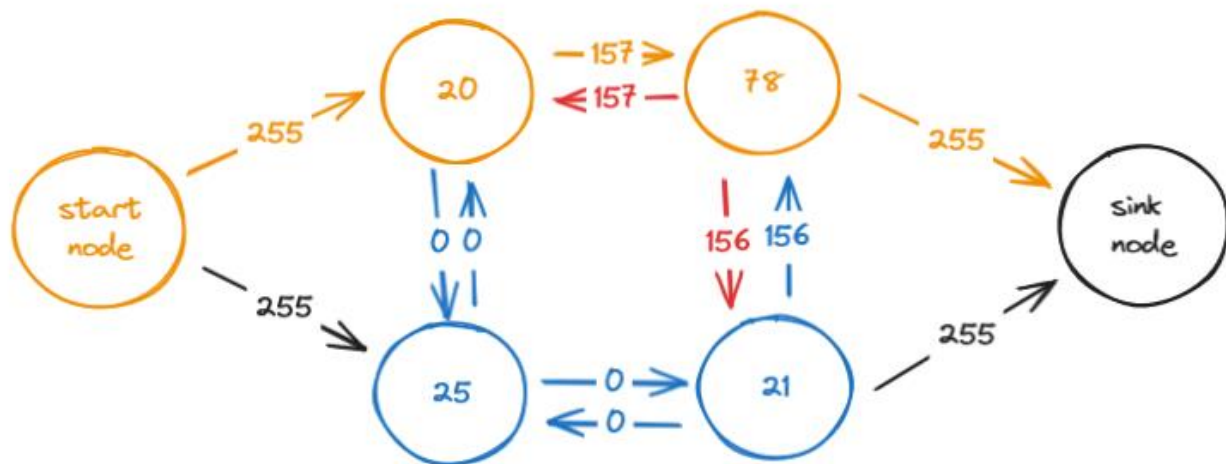
Pixel = 10, pixel du 4-voisin = 50, la différence entre eux $|10-50| > 10$ - poids = $255 - |10-50|$.



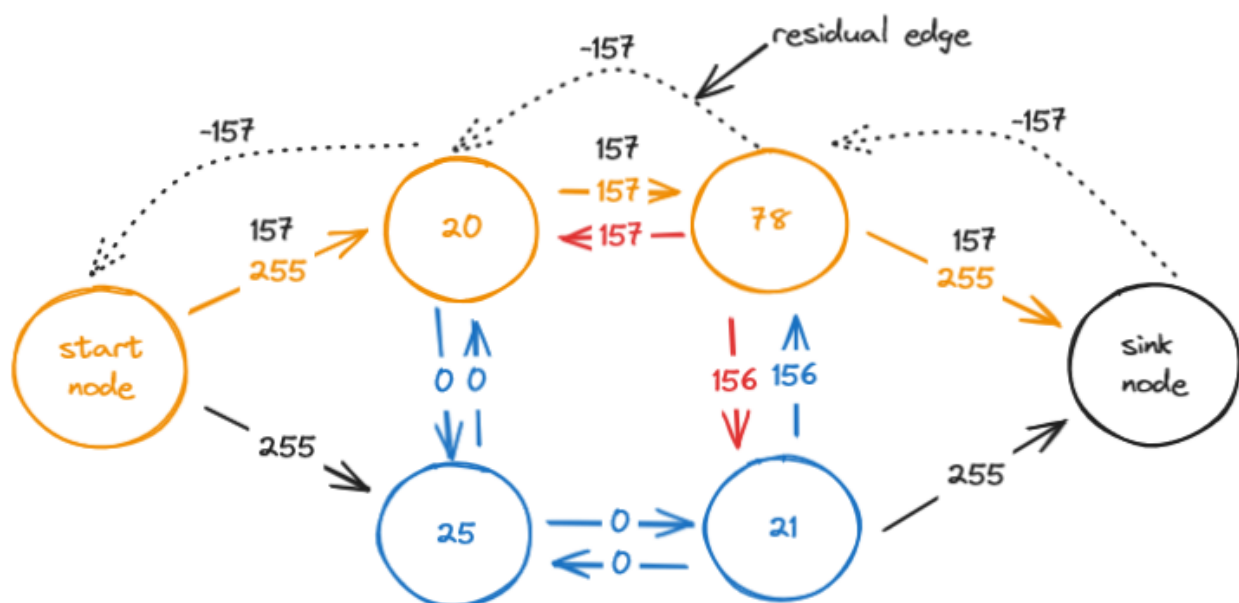
Pour les pixels situés en bordure de la photo, nous n'aurons que le nombre de voisins existants. Voici un exemple :



Pour trouver la **valeur maximale** qui peut être transférée dans le graphe (le flux maximal et, par conséquent, la coupe minimale), nous devons calculer des chemins augmentant à travers le graphe résiduel et augmenter le flux. Un chemin augmentant est un chemin d'arêtes dans le graphe résiduel (qui peut commencer à partir du graphe original) avec une capacité inutilisée supérieure à zéro allant de la source(s) à l'évier(t)/sink(t)/puite.

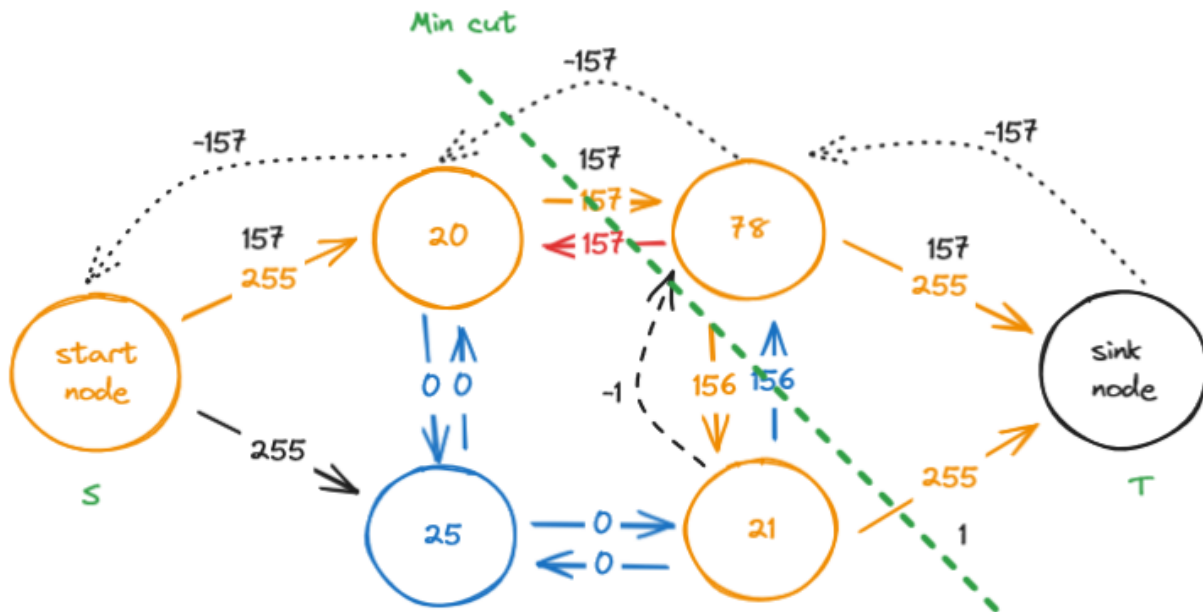


Chaque chemin augmenté possède un bottleneck/goulot d'étranglement, qui est la plus petite arête sur le chemin. Nous utilisons la valeur du goulot d'étranglement pour augmenter le flux le long du chemin (par exemple, une augmentation de 157), augmenter sur le chemin et diminuer sur les arêtes résiduelles, et elles deviennent des arêtes valides pour prendre un chemin augmenté (pour éliminer certains chemins qui ne mènent pas au flux maximal).



Le graphe résiduel est celui qui contient les arêtes résiduelles, et pas seulement les arêtes originales. Sur les arêtes ayant une capacité ou un poids de 0, il n'y aura pas de flux.

La coupe représente la partition du graphe en deux sous-ensembles disjoints (la réunion des deux ensembles de sommets est le graphe original, mais leur intersection est vide) en maintenant le nœud de départ (S) et le nœud puits/sinks (T) dans des sous-ensembles différents. Dans ce cas, nous observons que le flux est maximisé à travers les nœuds : 78 et 21. Nous devons également séparer le nœud de départ du nœud puits/sinks, donc les arêtes qui seront coupées sont de nœud 20-78, 78-21, 21-puits/sinks, incluant tous les nœuds qui font partie de l'avant-plan de l'image.



Votre chef d'équipe vous dit de commencer à travailler sur le PoC (proof of concept) mais avec certaines limitations à l'esprit :

- Les images seront de taille 6x6.
- Le seuil spécifié sera de 10.
- Le poids du nœud de départ vers les autres nœuds sera de 255.
- Le poids du nœud final vers les autres nœuds sera également de 255.

Détails des tâches :

- Implémenter le graphe pour l'image (0,75p)
- Implémenter l'algorithme pour trouver le flux maximal (1p)
- Implémenter l'algorithme de coupe du graphe (1,5p)
- Traverser le graphe (0,5p)
- Générer une image à partir du graphe restant (0,25p)

Exemple entrée/sortie

Exemple - 1

Photo initiale :

```
230 128 255 128 255 230
1 255 1 230 204 255
1 230 1 255 100 255
1 1 1 255 133 255
1 255 1 1 255 255
255 128 255 128 230 255
```

Explication :

Tout d'abord, nous créons un graphe dont le nombre de nœuds est égal au nombre de pixels dans l'image (la taille de la matrice[0] * taille[1] = 36 nœuds) + deux nœuds supplémentaires que nous utiliserons comme nœuds source et puits/sinks. Les nœuds source et puits/sinks sont connectés à tous

[illegible]

Graphe résiduel après le calcul du flux maximal :

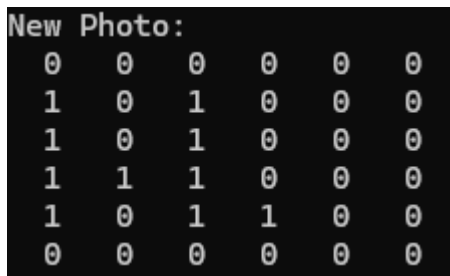
[illegible]

Nous observons qu'il y a de multiples valeurs. Pour toutes celles qui sont inférieures au maximum d'entre elles (dans ce cas 230, car nous ne comptons pas les nœuds source et puits/sinks qui ne font pas partie de l'image mais qui sont des nœuds d'aide, nous choisissons la valeur maximale pour cela car nous voulons juste un problème binaire [premier plan vs arrière-plan], mais en choisissant chaque valeur, nous pouvons générer des masques qui nous aident à segmenter chaque partie de l'image avec différentes valeurs) et qui ne sont pas 0, nous coupons les arêtes du nœud.

Ensuite, en utilisant le graphe restant, nous parcourons à partir du premier nœud qui a une connexion jusqu'au puits/sinks afin de déterminer quels nœuds sont visités et ensuite avec eux générer la nouvelle image.

Nouvelle photo:

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



```
New Photo:
0 0 0 0 0 0
1 0 1 0 0 0
1 0 1 0 0 0
1 1 1 0 0 0
1 0 1 1 0 0
0 0 0 0 0 0
```

Example – 2

Photo initiale:

```
100 107 103 101 183 145
151 151 151 5 132 100
123 32 5 5 132 100
123 5 5 123 123 154
151 5 5 151 100 123
151 5 5 100 100 123
```

Nouvelle photo:

```
0 0 0 0 0
0 0 0 5 0
0 0 5 5 0
0 5 5 0 0
0 5 5 0 0
0 5 5 0 0
```

Glossaire :

- **(Bottleneck) / Goulot d'étranglement** : la plus petite valeur qu'une arête peut avoir, limitant ainsi le flux.
- **(Capacity) / Capacité** : le poids de l'arête, la quantité d'information qui peut être transmise d'un nœud à un autre, information maximale envoyée.
- **(Flow / Max flow) Flux / Flux maximal** : Mouvement d'une certaine quantité d'information à travers un graphe connecté.
- **(Augmented path) / Chemin augmenté** : Un chemin dans le graphe qui a une capacité disponible pour l'envoi d'informations supplémentaires du nœud source au nœud puits/sinks.

- **(Residual graph) / Graphe résiduel** : lors de l'envoi de flux à travers le graphe, certaines arêtes peuvent ne pas utiliser toute leur capacité. Il comprend des arêtes directes (le flux suit la direction de l'arête) qui sont inférieures à la capacité, résultant en une capacité résiduelle disponible qui peut être utilisée pour envoyer plus de flux. Les arêtes inverses permettent un flux dans la direction opposée (par rapport à l'arête originale), signifiant qu'elles peuvent "annuler" une partie du flux qui a été précédemment envoyé. Le graphe résiduel est construit sur la base du graphe original et du flux actuel, incluant à la fois les arêtes directes et inverses, représentant le potentiel pour l'envoi de flux supplémentaire ou la suppression du flux déjà envoyé.
- **(Graph cut) / Coupe de graphe** : partitionnement du graphe en deux sous-ensembles disjoints (la réunion des deux ensembles de sommets est le graphe original, mais leur intersection est vide) en gardant le nœud de départ (S) et le nœud puits/sinks (T) dans des sous-ensembles différents.
 - **(Min-cut) / Coupe minimale** : dans notre cas, c'est pertinent (le poids total le plus faible des arêtes qui, si elles étaient supprimées, déconnecteraient la source du puits/sinks).

Liens:

<https://julie-jiang.github.io/image-segmentation/>

https://en.wikipedia.org/wiki/Graph_cuts_in_computer_vision

https://en.wikipedia.org/wiki/Maximum_flow_problem