**Explication du sujet du TP**

Le **TP 9** porte sur la modélisation et le traitement d’images binaires (formes noires sur fond blanc) en utilisant des **graphes**. Voici une explication détaillée du sujet et des objectifs :

**Contexte général**

Les algorithmes sur les graphes sont souvent utilisés en vision par ordinateur pour traiter des images. Dans ce TP, chaque **pixel** est vu comme un **nœud** d’un graphe, et ses relations avec les pixels voisins (haut, bas, gauche, droite, et diagonales) sont modélisées comme des **arcs** du graphe.

**Objectif principal :** À partir d’une image contenant une **forme noire sur un fond blanc** :

1. Calculer une **image de distance** où chaque pixel blanc contient la distance au pixel noir le plus proche.
2. Modéliser des opérations sur ces formes pour répondre à des questions comme :
   * Quel est le chemin le plus court entre deux pixels ?
   * Quelle est la forme obtenue en fusionnant deux images ?
   * Comment projeter un pixel sur une forme sans recalculer l’image de distance ?

**Détail des fonctionnalités attendues**

1. **Chargement et sauvegarde d'images au format PGM** :
   * Les images sont au format **PGM ASCII**, où chaque pixel est représenté par une intensité lumineuse entre 0 (noir) et 255 (blanc).
   * Exemple :

P2

5 5

255

255 255 255 255 255

255 0 0 0 255

255 0 255 0 255

255 0 0 0 255

255 255 255 255 255

1. **Calcul des plus courts chemins avec l’algorithme de Dijkstra** :
   * Contrairement à l’algorithme classique qui part d’un **seul nœud**, ici, le départ se fait depuis **tous les pixels noirs**.
   * Chaque pixel blanc reçoit sa distance minimale à un pixel noir.
   * Les coûts sont définis comme suit :
     + 2 pour les déplacements horizontaux/verticaux.
     + 3 pour les déplacements diagonaux.
2. **Sauvegarde des résultats** :
   * Sauvegarder l'image de distance en PGM.
   * Enregistrer les prédécesseurs dans un fichier texte pour tracer les chemins.
3. **Projection d’un pixel sur une forme** :
   * À partir d'un pixel donné, déterminer le pixel noir le plus proche.
   * Exploiter l’image de distance pour optimiser cette opération.
4. **Union de deux formes** :
   * Fusionner deux images de distance en prenant le minimum des distances correspondantes.
5. **Projection sur l’union des formes** :
   * Projeter un pixel sur l’union de deux formes **sans recalculer l’algorithme de Dijkstra**.

**Modèle de données utilisé**

1. **Représentation des pixels :**
   * Chaque pixel est décrit par :
     + Son intensité lumineuse (0 pour noir, une valeur entre 1 et 255 pour blanc).
     + Ses coordonnées (ligne, colonne).
2. **Représentation en mémoire :**
   * L’image est stockée dans un tableau **1D** de taille L \* C (lignes × colonnes).
   * Le pixel (i, j) se trouve à l’index i \* C + j.
3. **Relations de voisinage :**
   * Chaque pixel est relié à ses **8 voisins** (haut, bas, gauche, droite, et diagonales).
   * Les indices des voisins sont calculés dynamiquement en fonction des coordonnées.

**Exigences techniques**

1. **Modularité** :
   * Implémenter les fonctionnalités dans des modules bien séparés (.h et .cpp).
2. **Interface utilisateur** :
   * Fournir un programme avec un **menu interactif** permettant de tester chaque fonctionnalité (chargement, sauvegarde, calcul de distances, etc.).
3. **Dépôt du travail** :
   * Fournir une archive **Nom1\_Nom2.tgz** contenant :
     + Les fichiers source (.cpp, .h).
     + Un fichier Makefile pour compiler le projet.
     + Un fichier README pour expliquer son fonctionnement.
4. **Qualité du code** :
   * Pas d'erreurs ni d'avertissements à la compilation.
   * Une programmation modulaire et claire.

**Synthèse de l’objectif**

Ce TP consiste à :

1. **Modéliser une image** comme un graphe.
2. **Utiliser des algorithmes sur les graphes** pour traiter les images (calculs de distance, fusion, projection).
3. **Développer une application modulaire** permettant à l'utilisateur d'interagir avec ces données.

**Explication de pixel.cpp**

Le fichier **pixel.cpp** implémente les fonctionnalités de la classe **Pixel**, qui représente un pixel d’une image avec ses coordonnées (x, y) et son intensité lumineuse.

**Objectif du fichier pixel.cpp**

Ce fichier est responsable de la mise en œuvre des fonctionnalités de base de la classe **Pixel** :

1. **Gestion des propriétés d’un pixel** :
   * Récupérer et modifier l’intensité lumineuse.
   * Récupérer les coordonnées du pixel.
2. **Affichage des informations du pixel** :
   * Facilite le débogage et la visualisation des données.
3. **Modélisation d’un pixel** :
   * Chaque objet **Pixel** représente un point de l’image, avec ses informations lumineuses et géométriques.

Graphe.cpp

1. **Bibliothèques standard** :

* **<iostream>** : Pour afficher des messages d'erreur via cout.
* **<fstream>** : Pour gérer les flux de lecture et d'écriture de fichiers.
* **<string>**, **<vector>**, **<queue>**, **<limits>**, **<utility>**, **<functional>** : Fournissent les structures et outils nécessaires pour gérer les données.

2. **#include "graphe.h"** :

* Importe les déclarations de la classe **Graphe**.

3. **using namespace std;** :

* Évite d’écrire explicitement std:: devant chaque fonction ou type (comme cout, vector, etc.).

Constructeur par défaut

Graphe::Graphe(){}

* Ce constructeur ne fait rien d'initialement utile. Il existe simplement pour permettre de créer un objet **Graphe** sans paramètre.
* Il pourrait être utilisé dans un cas où l'initialisation est différée, par exemple lorsqu'on veut charger une image plus tard.

**Constructeur à partir d’un fichier**

Graphe::Graphe(const char \* nomFichier)

{

int valeurs = 0; //valeur du pixel qu'on va insere dans le tableau d'intensite

std::ifstream flux;

flux.open(nomFichier);

if(!flux.is\_open())

{

cout << "Impossible d'ouvrir le fichier " << nomFichier << " en lecture \n";

exit(1);

}

string caractere\_inutile;

flux >> caractere\_inutile; // Lecture du format ("P2", inutile ici)

flux >> C; // Lecture du nombre de colonnes (C).

flux >> L; // Lecture du nombre de lignes (L).

flux >> intensite\_max; // Lecture de l'intensité maximale de l'image.

image.resize(L \* C); // Redimensionne le vecteur image pour contenir L \* C pixels.

for (int i = 0; i < L \* C; i++)

{

flux >> valeurs;

image[i].setIntensiteLumineuse(valeurs); // Initialise chaque pixel avec son intensité.

}

flux.close(); // Ferme le fichier après lecture.}

**Explications détaillées :**

1. **Lecture des paramètres du fichier :**
   * Le fichier contient des informations au format **PGM ASCII** :
     + Première ligne : le format (P2, ignoré ici).
     + Deuxième ligne : le nombre de colonnes (C) et de lignes (L).
     + Troisième ligne : l’intensité maximale (intensite\_max), généralement 255.
     + Les lignes suivantes : les intensités lumineuses des pixels.

Exemple de contenu :

P2

5 5

255

255 255 255 255 255

255 0 0 0 255

255 0 255 0 255

255 0 0 0 255

255 255 255 255 255

1. **Initialisation de l’image :**

* Le constructeur utilise un tableau **1D** (image) pour représenter l'image.
* La méthode **resize(L \* C)** ajuste la taille du tableau pour contenir tous les pixels.

1. **Création des pixels :**

* Chaque pixel est initialisé en appelant **setIntensiteLumineuse(valeurs)**, où **valeurs** est lu dans le fichier.
* Les coordonnées des pixels ne sont pas explicitement stockées ici, mais peuvent être dérivées de leur position dans le tableau 1D :
  + Pour un index **i** dans le tableau :
    - **Ligne** = i / C
    - **Colonne** = i % C

1. **Gestion des erreurs :**

* Si le fichier ne peut pas être ouvert, un message est affiché, et le programme s’arrête avec **exit(1)**.

**Objectif de cette fonction**

1. Charger une image au format PGM ASCII dans un objet **Graphe**.
2. Préparer un tableau 1D où chaque élément correspond à un pixel, avec une intensité lumineuse.
3. Rendre les données disponibles pour d’autres opérations, comme le calcul des distances.

**Pourquoi cette fonction est essentielle**

* **Point d'entrée principal :** Sans elle, il serait impossible d’importer des images dans votre programme.
* **Modularité :** L’image est abstraite dans un vecteur **image**, facilitant les opérations ultérieures.
* **Gestion des fichiers :** Cette fonction permet de lire les fichiers PGM, qui sont essentiels pour travailler sur des images.

**Questions pour l’oral**

1. **Que fait cette fonction si le fichier PGM est mal formé ?**
   * Si les dimensions ne correspondent pas au nombre de pixels, cela pourrait causer un comportement imprévisible.
2. **Pourquoi utiliser un tableau 1D pour représenter une image ?**
   * Simplicité : les opérations sur les indices sont faciles à calculer avec i / C et i % C.
   * Efficacité : un tableau 1D consomme moins de mémoire qu’une structure plus complexe.
3. **Comment pourriez-vous étendre cette fonction ?**
   * Ajouter une vérification pour s’assurer que les dimensions et les pixels lus sont cohérents.
   * Supporter d’autres formats d’image.

**Explication de la fonction Graphe::Graphe(const Graphe &autre)**

Cette fonction est un **constructeur par copie**, utilisé pour créer un nouvel objet **Graphe** à partir d’un autre objet **Graphe** existant.

**Pourquoi cette fonction est importante**

* **Modularité :** Permet de dupliquer un objet Graphe sans affecter l'original.
* **Utilisation dans des algorithmes :** Certains algorithmes nécessitent de travailler sur une copie des données d’origine pour éviter de les altérer.
* **Gestion mémoire propre :** En copiant les pixels de manière explicite, cette fonction garantit que chaque objet a sa propre zone mémoire.

**Points à comprendre pour l’oral**

1. **Différence entre une copie profonde et une copie superficielle :**
   * Ici, c’est une **copie profonde** : tous les pixels de l’image sont copiés individuellement.
   * Une **copie superficielle** (par référence) aurait simplement partagé les données entre les deux objets, ce qui peut causer des conflits.
2. **Pourquoi utiliser resize avant de copier ?**
   * Pour s'assurer que le tableau **image** de la copie est correctement dimensionné avant d’y insérer les données.
3. **Est-ce que cette fonction gère correctement les objets dynamiques ?**
   * Oui, car elle redimensionne explicitement **image** et copie chaque pixel.

**Explication des fonctions operator= et ~Graphe()**

Ces fonctions concernent la gestion de l'assignation et de la destruction des objets **Graphe**. Elles sont importantes pour gérer correctement la mémoire et garantir un comportement cohérent lors de l'affectation ou de la fin de vie d'un objet.

**Objectif de l'opérateur d'assignation :**

* **Permettre une réassignation propre :** Si un objet **Graphe** est réassigné après sa création, cette fonction garantit que la nouvelle affectation est correcte.
* **Gestion mémoire sécurisée :** Avant de copier les nouvelles données, l’opérateur libère les anciennes pour éviter les fuites mémoire.
* **Indépendance des objets :** Les deux objets deviennent indépendants, et toute modification de l'un n'affecte pas l'autre.

**Pourquoi est-il vide ?**

* Dans ce programme, les ressources utilisées par la classe **Graphe** (comme **image**) sont gérées automatiquement par la bibliothèque standard **STL** (grâce à **std::vector**).
* Le destructeur de **std::vector** libère la mémoire associée au tableau dynamique. Il n’est donc pas nécessaire de gérer explicitement la mémoire dans le destructeur.

**Points importants pour l’oral**

1. **Pourquoi utiliser image.clear() avant de redimensionner ?**
   * Cela évite d’accumuler inutilement des données dans le vecteur et garantit qu’aucune mémoire n’est gaspillée.
2. **Pourquoi vérifier l’auto-assignation ?**
   * Si cette vérification était absente, des erreurs pourraient survenir, par exemple si les anciennes données sont libérées avant de copier les nouvelles.
3. **Pourquoi le destructeur est vide ?**
   * Parce que **std::vector** gère déjà la mémoire associée. Si d'autres ressources étaient utilisées (par exemple, des pointeurs), elles devraient être libérées explicitement ici.

**Explication de la fonction void Graphe::afficherImage() const**

Cette fonction affiche les propriétés et le contenu de l’image représentée par l’objet **Graphe**. Elle est utile pour visualiser les informations d'une image (dimensions, intensité maximale, pixels) directement dans la console.

**Points pour l’oral**

1. **Pourquoi afficher un espace pour les pixels noirs (intensité = 0) ?**
   * Cela rend l’image plus lisible visuellement, en séparant clairement les pixels noirs des autres.
2. **Comment cette fonction aide-t-elle au débogage ?**
   * Elle permet de vérifier que :
     + Les dimensions de l’image sont correctes.
     + Les intensités lumineuses des pixels correspondent à celles du fichier source.
3. **Que se passe-t-il si l'image n'est pas bien chargée ?**
   * La fonction affichera les valeurs par défaut du vecteur **image**, ce qui peut révéler un problème de lecture du fichier.

**Explication de la fonction void Graphe::afficherImageDistance(const std::vector<int> &distances) const**

Cette fonction affiche dans la console l’image des distances calculées à partir d’un vecteur **distances**. Chaque pixel blanc (non relié à un pixel noir) est affiché avec une valeur spéciale pour indiquer une distance "infinie".

Affichage de la distance ou d’un symbole spécial :

cout << (dist == std::numeric\_limits<int>::max() ? -1 : dist) << " ";

* Vérifie si la distance est égale à **std::numeric\_limits<int>::max()**, qui représente une distance "infinie".
* Si oui, affiche **-1** pour indiquer que ce pixel est inaccessible.
* Sinon, affiche la distance calculée.

**Objectif de la fonction**

1. **Visualisation des distances** :
   * Permet de vérifier que les distances calculées entre chaque pixel blanc et les pixels noirs sont correctes.
   * Chaque pixel affiche sa distance minimale à un pixel noir, sauf s’il est inaccessible.
2. **Débogage des calculs de distances** :
   * En affichant les distances dans une structure lisible, l’utilisateur peut vérifier les résultats obtenus par l’algorithme de Dijkstra ou tout autre algorithme utilisé.
3. **Gestion des pixels inaccessibles** :
   * Les pixels blancs sans connexion à un pixel noir sont explicitement marqués comme **-1**.

**Points pour l’oral**

1. **Pourquoi utiliser std::numeric\_limits<int>::max() pour représenter une distance infinie ?**
   * Cela permet de marquer les pixels non accessibles sans utiliser une valeur arbitraire, garantissant une bonne compatibilité avec les algorithmes de calcul de distances.
2. **Comment cette fonction peut-elle aider au débogage ?**
   * Elle permet de visualiser les distances calculées par l’algorithme. Toute incohérence dans les résultats sera facilement repérable dans la grille affichée.
3. **Pourquoi afficher -1 pour les distances infinies ?**
   * Cela rend l’affichage plus clair pour l’utilisateur, qui comprend immédiatement que ces pixels sont inaccessibles.

**Fonction getVoisin**

**Déroulement :**

1. **Initialisation :**
   * Les coordonnées actuelles du pixel sont copiées dans **ni** et **nj**.
2. **Mise à jour des coordonnées du voisin :**
   * Un switch détermine la nouvelle position (ligne et colonne) du voisin selon la direction donnée (Nord, Sud, Est, Ouest, ou diagonales).
3. **Vérification des limites :**
   * Les nouvelles coordonnées sont vérifiées pour s'assurer qu'elles restent dans les limites de l'image (0 ≤ **ni** < **L**, 0 ≤ **nj** < **C**).
4. **Conversion en index global :**
   * Si les coordonnées sont valides, elles sont converties en un **index global** à l’aide de **getCoord**.
5. **Gestion des erreurs :**
   * Si le voisin est en dehors des limites ou si la direction est invalide, la fonction retourne **-1**.

**Fonction getCoord**

int Graphe::getCoord(int i, int j) const {

return i \* C + j;

}

**Objectif :**

* Convertir les coordonnées d'un pixel (ligne, colonne) en un **index global** dans un tableau 1D.

**Objectif global des fonctions**

1. Gestion des voisins :
   * Les fonctions getVoisin et getVoisinIndiceGlobale permettent de naviguer facilement dans l’image et d’accéder aux voisins d’un pixel.
2. Facilité d’accès :
   * La fonction getCoord facilite la conversion entre coordonnées 2D (ligne, colonne) et l'index global 1D.
3. Encapsulation :
   * Les fonctions getLignes et getColonnes offrent une manière propre d’accéder aux dimensions de l’image sans exposer directement les attributs privés L et C.

**Exemple d’utilisation**

**Cas d’une image de 4 lignes et 5 colonnes :**

* Dimensions : **L = 4, C = 5**
* Index global d’un pixel à la **ligne 2, colonne 3** :

getCoord(2, 3) = 2 \* 5 + 3 = 13

Trouver le voisin Nord du pixel (ligne = 2, colonne = 3) :

int voisinNord = getVoisin(2, 3, Direction::Nord);

// ni = 2 - 1 = 1

// nj = 3

// getCoord(1, 3) = 1 \* 5 + 3 = 8

Trouver le voisin Est d’un pixel à l'index global 13 :

int voisinEst = getVoisinIndiceGlobale(13, Direction::Est);

// k = 13

// i = 13 / 5 = 2, j = 13 % 5 = 3

// nj = 3 + 1 = 4

// getCoord(2, 4) = 2 \* 5 + 4 = 14

**Points pour l’oral**

1. **Pourquoi avoir deux fonctions pour accéder aux voisins (getVoisin et getVoisinIndiceGlobale) ?**
   * **getVoisin** travaille directement avec les coordonnées 2D.
   * **getVoisinIndiceGlobale** est plus générique, car elle prend en entrée un index global.
2. **Que signifie un retour de -1 ?**
   * Cela indique un voisin invalide (hors des limites de l'image ou direction incorrecte).
3. **Avantages d’un tableau 1D pour représenter l’image :**
   * Simplifie les calculs et permet un accès rapide via des indices.

PARTIE 2 :

**Explication des fonctions de sauvegarde d'images et de données**

Ces fonctions permettent de sauvegarder différentes représentations d'une image dans des fichiers. Elles gèrent la sauvegarde au format PGM (utilisé pour les images) et sous forme de tableau pour d'autres données (comme les distances ou les prédécesseurs).

Fonction sauvegarderImageForme

**Exemple de sortie (image de 3x3 pixels) :**

Si l'image contient les pixels suivants :

255 255 255

255 0 255

255 255 255

Le fichier **PGM** généré sera :

P2

3 3

255

255 255 255

255 0 255

255 255 255

**Fonction sauvegarderImageDistance**

**Écriture des distances :**

* Parcourt le tableau **distances**.
* Si une distance est égale à **std::numeric\_limits<int>::max()**, cela signifie qu’elle est "infinie", et la valeur **255** (blanc) est écrite à la place.

**Exemple de sortie :**

Si **distances** est :

0 2 4

2 1 3

4 3 ∞

Le fichier PGM sera :

P2

3 3

255

0 2 4

2 1 3

4 3 255

Fonction sauvegarderImagePredecesseurs

**Écriture du tableau des prédécesseurs :**

* Parcourt le tableau **predecesseurs**.
* Écrit chaque prédécesseur en respectant la structure 2D de l'image.
* Si un pixel n’a pas de prédécesseur, la valeur **-1** est enregistrée.

**Exemple de sortie :**

Si **predecesseurs** est :

-1 -1 -1

-1 0 1

3 4 5

Le fichier sauvegardé sera :

Tableau des prédécesseurs :

-1 -1 -1

-1 0 1

3 4 5

**Comparaison entre les trois fonctions**

| **Fonction** | **Objectif** | **Format de sortie** |
| --- | --- | --- |
| **sauvegarderImageForme** | Sauvegarde l'image au format PGM. | Image PGM avec intensité lumineuse. |
| **sauvegarderImageDistance** | Sauvegarde l'image de distances au format PGM. | Image PGM où ∞ est remplacé par 255. |
| **sauvegarderImagePredecesseurs** | Sauvegarde le tableau des prédécesseurs. | Fichier texte avec indices des prédécesseurs. |

**Points importants pour l’oral**

1. **Pourquoi utiliser 255 pour les distances infinies ?**
   * Cela respecte le format PGM, où 255 représente le blanc (valeur maximale).
2. **Pourquoi y a-t-il trois fonctions distinctes ?**
   * Chacune traite un type de donnée différent (image brute, distances, prédécesseurs) et adapte le format de sortie en conséquence.
3. **Que faire si l’image est trop grande ?**
   * Une optimisation pourrait être d'écrire les données en plusieurs blocs pour éviter une surcharge de mémoire.

**Explication de la fonction calculerImageDistance**

Cette fonction calcule les distances les plus courtes entre tous les pixels blancs d'une image et les pixels noirs les plus proches en utilisant une version modifiée de l’**algorithme de Dijkstra**.

Signature de la fonction

void Graphe::calculerImageDistance(vector<int> &distances, vector<int> &predecesseurs)

* **Entrées :**
  + **distances** : Un vecteur où chaque index correspond à la distance minimale entre le pixel (index) et un pixel noir.
  + **predecesseurs** : Un vecteur où chaque index stocke l’index global du prédécesseur du pixel sur le chemin le plus court.
* **Sorties :**
  + Les tableaux **distances** et **predecesseurs** sont mis à jour après l'exécution.

**Initialisation des pixels noirs**

for (int i = 0; i < L \* C; ++i) {

if (image[i].getIntensiteLumineuse() == 0) {

distances[i] = 0;

file.emplace(0, i); // Ajouter (distance, index) à la file

}

}

* Tous les pixels noirs (intensité lumineuse = 0) sont initialisés à une distance de **0**.
* Ces pixels sont ajoutés à une **file de priorité** (min-heap) pour commencer la propagation des distances.

Définition des coûts de déplacement

const int costs[] = {2, 2, 2, 2, 3, 3, 3, 3};

Les coûts sont constants :

* **2** pour les déplacements orthogonaux (haut, bas, gauche, droite).
* **3** pour les déplacements diagonaux.

Boucle principale : exploration des pixels

while (!file.empty()) {

auto [distanceActuelle, index] = file.top();

file.pop();

if (distanceActuelle > distances[index]) continue;

**File de priorité :**

* Le pixel avec la distance actuelle la plus petite est extrait (priorité minimale).

**Vérification des distances :**

* Si la distance actuelle dépasse celle enregistrée dans **distances**, ce pixel est ignoré.

**Propagation aux voisins**

int y = index / C, x = index % C;

for (int d = Ouest; d <= NordEst; ++d) {

Direction direction = static\_cast<Direction>(d);

int voisin = getVoisin(y, x, direction);

if (voisin != -1) {

int nouveauCout = distanceActuelle + costs[d];

if (nouveauCout < distances[voisin]) {

distances[voisin] = nouveauCout;

predecesseurs[voisin] = index;

file.emplace(nouveauCout, voisin);

}

}

}

**Conversion de l’index en coordonnées :**

* Les indices globaux sont convertis en coordonnées **(y, x)** grâce à **index / C** et **index % C**.

**Exploration des directions :**

* Chaque pixel est relié à ses voisins dans les 8 directions.
* La fonction **getVoisin** vérifie si le voisin est dans les limites de l’image et retourne son index global.

**Mise à jour des distances :**

* Si une distance plus courte est trouvée vers un voisin, elle est mise à jour dans **distances**.
* Le prédécesseur est également mis à jour.

**Ajout à la file de priorité :**

* Le voisin est ajouté à la file avec sa nouvelle distance.

**Objectifs de la fonction**

1. **Calcul des distances minimales :**
   * Pour chaque pixel blanc, trouver la distance au pixel noir le plus proche.
2. **Stockage des prédécesseurs :**
   * Permet de retracer les chemins les plus courts.
3. **Utilisation efficace des ressources :**
   * L’utilisation d’une file de priorité garantit une complexité optimale pour l’algorithme de Dijkstra : **O((E + V) log V)**, où :
     + **V** est le nombre de pixels.
     + **E** est le nombre d’arcs (voisins).

**Exemple d’exécution**

**Données :**

* Dimensions : **3 x 3**
* Intensités lumineuses de l’image :

255 255 255

255 0 255

255 255 255

**Initialisation :**

* Les distances sont initialisées à **∞**, sauf pour les pixels noirs (intensité = 0).

distances = [∞, ∞, ∞, ∞, 0, ∞, ∞, ∞, ∞]

**Résultat final :**

* Les distances calculées (avec coûts de déplacement) :

distances = [ 2, 2, 2, 2, 0, 2, 4, 3, 4]

**Points pour l’oral**

1. **Pourquoi utiliser une file de priorité ?**
   * La file de priorité garantit que les pixels sont explorés dans l’ordre croissant de distances, ce qui est crucial pour l’algorithme de Dijkstra.
2. **Pourquoi vérifier distanceActuelle > distances[index] ?**
   * Cette vérification évite de recalculer les distances pour des pixels déjà mis à jour avec des valeurs plus courtes.
3. **Pourquoi utiliser un tableau pour les prédécesseurs ?**
   * Cela permet de retracer les chemins les plus courts pour chaque pixel, utile pour des analyses ou des visualisations ultérieures.
4. **Comment gérer les pixels inaccessibles ?**
   * Les pixels blancs sans connexion à un pixel noir conservent une distance infinie (std::numeric\_limits<int>::max()).
5. **Différence entre coûts orthogonaux et diagonaux ?**
   * Cela reflète une réalité géométrique : les déplacements diagonaux sont légèrement plus longs que les déplacements orthogonaux.

**Conclusion**

Cette fonction est centrale dans le programme. Elle combine efficacement l’algorithme de Dijkstra avec une modélisation spécifique des images en graphe, permettant de résoudre des problèmes de distances et de chemins avec précision.

**Explication de la fonction projeterPixel**

La fonction **projeterPixel** recherche le **pixel noir le plus proche** d’un pixel donné (**x, y**) dans l’image. Elle utilise la **distance de Manhattan** pour calculer cette proximité.

Si les coordonnées du pixel (**x, y**) sont en dehors des limites de l’image (définies par **L** lignes et **C** colonnes) :

* Affiche un message d’erreur dans la console.
* Retourne **{-1, -1}** pour indiquer que le pixel est invalide.

**distanceMin** : La plus petite distance calculée jusqu’ici. Initialisée à **-1** pour indiquer qu’aucune distance n’a encore été calculée.

**pixelProche** : Index global du pixel noir le plus proche.

 **Parcours de l’image :**

* Chaque pixel est analysé à l’aide d’un index global **i**.
* **Condition :** Seuls les pixels noirs (**intensité = 0**) sont pris en compte.

 **Calcul de la distance de Manhattan :**

* **dx** : Différence entre les colonnes du pixel courant (**i % C**) et du pixel cible (**x**).
* **dy** : Différence entre les lignes du pixel courant (**i / C**) et du pixel cible (**y**).
* **Distance totale :** distance = dx + dy.

 **Mise à jour de la distance minimale :**

* Si **distanceMin** est non définie (**-1**) ou si une distance plus courte est trouvée, on met à jour :
  + **distanceMin** : La nouvelle distance minimale.
  + **pixelProche** : L’index global du pixel noir le plus proche.
* Si un pixel noir a été trouvé (**pixelProche != -1**), son index global est converti en coordonnées 2D :
  + **x = pixelProche % C** : Colonne.
  + **y = pixelProche / C** : Ligne.
* Sinon, retourne **{-1, -1}** pour indiquer qu’aucun pixel noir n’a été trouvé.

**Objectif de la fonction**

1. **Trouver le pixel noir le plus proche :**
   * Recherche efficace en parcourant tous les pixels noirs de l’image.
2. **Calcul basé sur la distance de Manhattan :**
   * La distance de Manhattan est utilisée car elle reflète bien les déplacements en grille (orthogonaux).
3. **Vérification de validité :**
   * Si le pixel cible est en dehors des limites ou si aucun pixel noir n’existe, la fonction renvoie des coordonnées invalides.

**Exemple d’exécution**

**Données :**

* Dimensions : **3 x 3**.
* Image (intensités) :

255 255 255

255 0 255

255 255 255

Appel de la fonction :

std::pair<int, int> result = graphe.projeterPixel(0, 0);

**Étapes :**

1. **Vérification des limites :**
   * Pixel cible (**x = 0, y = 0**) est valide.
2. **Parcours des pixels noirs :**
   * Seul le pixel noir est à **(x = 1, y = 1)** :
     + Distance de Manhattan = abs(1 - 0) + abs(1 - 0) = 1 + 1 = 2.
3. **Résultat :**
   * Le pixel noir le plus proche est à **(x = 1, y = 1)**.
   * La fonction retourne **{1, 1}**.

 **Que signifie un retour de {-1, -1} ?**

* Le pixel cible est en dehors des limites, ou aucun pixel noir n’a été trouvé.

 **Complexité de la fonction :**

* La fonction parcourt tous les pixels de l’image : complexité **O(L × C)**.

**Explication de la fonction unionImages**

La fonction **unionImages** fusionne deux images en prenant, pour chaque pixel, la **distance minimale** entre deux images. Elle est utilisée pour combiner deux images de distances calculées indépendamment.

**Objectifs de la fonction**

1. **Fusion d'images :**
   * Combine deux images en prenant la distance minimale pour chaque pixel.
   * Cette approche est utile pour obtenir la distance minimale globale entre un pixel donné et les pixels noirs présents dans deux images.
2. **Indépendance des objets :**
   * La fonction retourne une nouvelle image sans modifier les images d’origine.
3. **Vérification de la compatibilité :**
   * Avant toute opération, elle vérifie que les dimensions des images sont identiques.

**Exemple d’exécution**

**Données :**

* Image 1 (**image**) :

5 7 9

6 2 4

8 3 10

Image 2 (**autre.image**) :

6 5 8

7 1 3

9 4 2

Graphe fusion = graphe1.unionImages(graphe2);

Fusionnée :

5 5 8

6 1 3

8 3 2

**Points pour l’oral**

1. **Pourquoi vérifier les dimensions avant la fusion ?**
   * Si les images ont des dimensions différentes, il n’est pas possible d’effectuer une correspondance pixel par pixel. Cela entraînerait des erreurs d'accès mémoire.
2. **Pourquoi copier l'image courante dans resultat ?**
   * Cela garantit que l’image fusionnée est indépendante des deux images d'origine.
3. **Que signifie "fusionner les distances" ?**
   * Pour chaque pixel, on prend la plus petite des deux distances. Cela permet de calculer la distance minimale à un pixel noir pour l’union des deux images.
4. **Complexité de la fonction :**
   * La fonction parcourt chaque pixel une fois : **O(L × C)**.
5. **Quels cas pratiques pour cette fonction ?**
   * Fusionner des images de distances pour combiner des résultats de plusieurs formes ou objets dans une même grille.

**Explication de la fonction projeterPointSurUnion**

La fonction **projeterPointSurUnion** est utilisée pour projeter un point donné (**x, y**) sur l'union de deux formes représentées par leurs distances minimales aux pixels noirs respectifs. Elle détermine la **forme la plus proche** du point et retourne les coordonnées du pixel noir correspondant.

**Objectifs de la fonction**

1. **Projection sur une union :**
   * Combine deux formes et projette un point donné sur l'union des deux.
2. **Choix optimal :**
   * Sélectionne la forme la plus proche du point parmi les deux.
3. **Gestion des pixels inaccessibles :**
   * Si le point est inaccessible dans les deux formes, la fonction retourne une erreur claire.