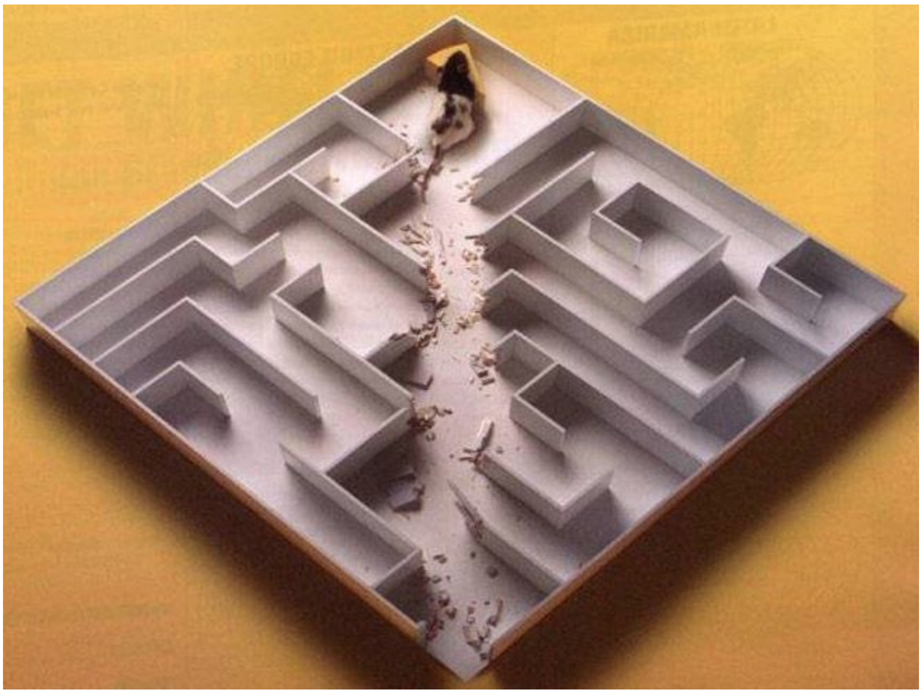
CIR2 - Algorithmique Avancée

Sujet TD&P n°5 - Séances n°7 & 8

Le problème et les algorithmes de Plus Court(s) Chemin(s)



Nick Falkner. MY CHEESE!

Partie 1 : Introduction

Résumé & Objectifs.
Deux algorithmes répondant au problème du plus court chemin, l'algorithme de **Dijkstra** et l'algorithme de **Floyd-Warshall**, sont étudiés dans ce TD&P. La partie sur l'algorithme de Dijkstra se focalise sur la compréhension de celui-ci en relevant la trace sur plusieurs exemples. L'algorithme de Floyd-Warshall sera quant à lui implémenté. Il pourra ainsi répondre à des situations bien réelles, comme l'établissement de tous les plus courts chemins entre un ensemble de villes.

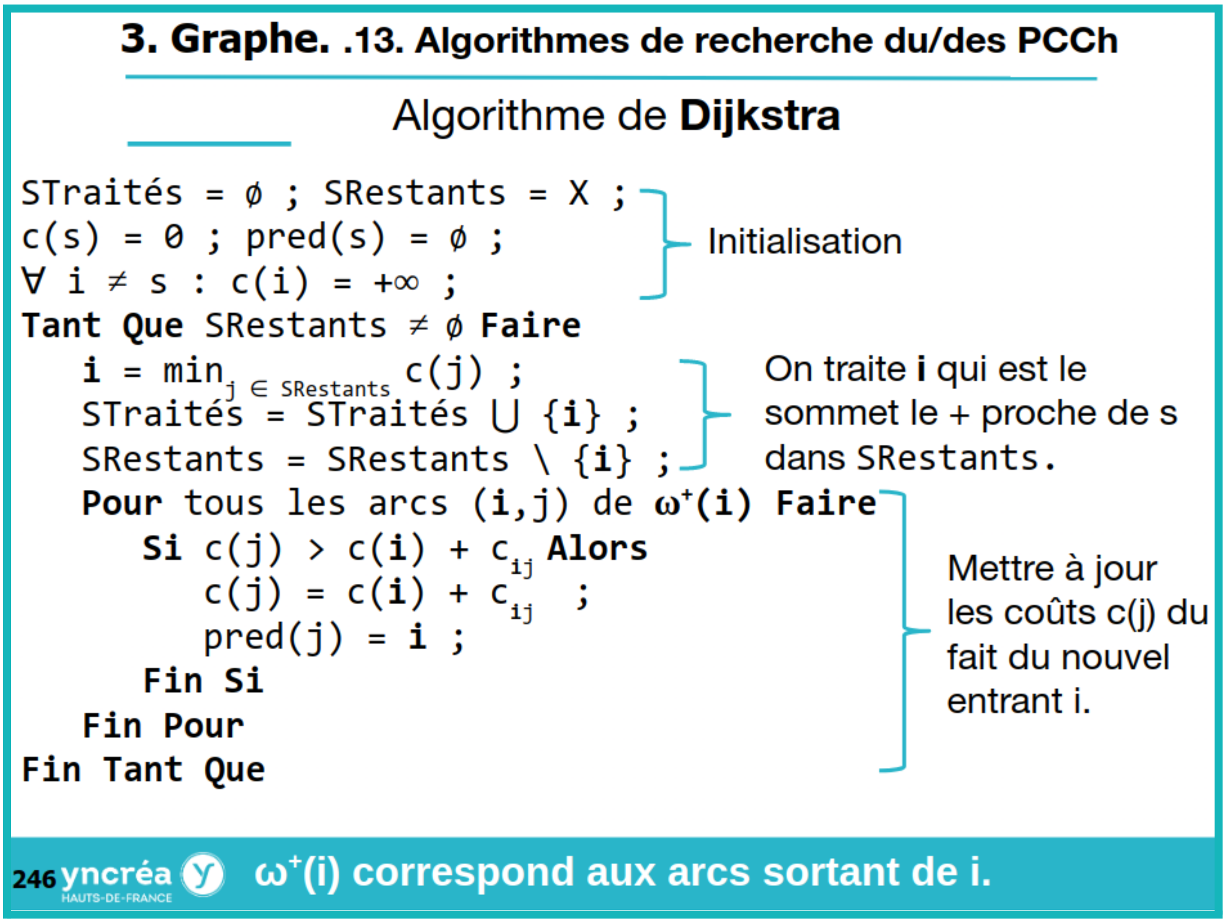
JUNIA ISEN

Samuel Deleplanque. (prénom.nom@junia.com)

Partie 2 : Algorithme de Dijkstra

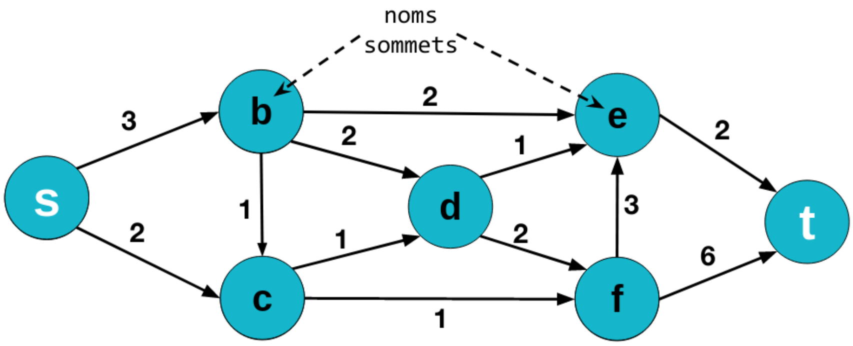
Calcul des plus courts chemins à partir d'un sommet source

La figure suivante reprend la diapositive du cours de l'algorithme de Dijkstra.



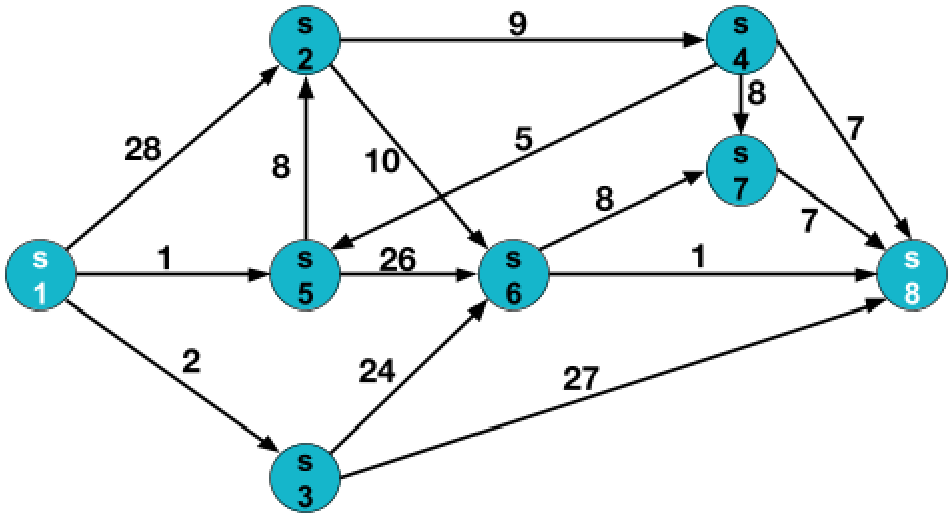
l'algorithme de Dijkstra (diapositive du cours)

- **i. Trace Algo.** Commencez sur un exemple connu : essayez d'appliquer Dijkstra sur l'exemple repris du cours et dont le graphe est copié ci-dessous. Ouvrez donc le `.pdf` de votre cours à l'endroit de cet exemple et où chaque itération de la “grande” boucle **`Tant Que`** correspond à une diapositive. Essayez alors d'anticiper à chaque fois le résultat de la diapositive suivante, cela vous entraînera à la question suivante où le graphe n'a pas encore été traité. Au préalable, vous pouvez revoir la vidéo d'une minute et déroulant l'algorithme de Dijkstra ici : https://www.youtube.com/watch?v=JPeCmKFrKio.



S’entraîner d’abord sur l’exemple du cours (diapositive du cours ≈ 254)

ii. Trace Algo. Déroulez à nouveau l’algorithme de Dijkstra mais cette fois sur le nouvel -et plus grand- exemple donné ci-dessous afin de connaître le plus court chemin entre le sommet S1 et le sommet S8. Pour cela, faites à nouveau la trace pour chaque itération de la “grande” boucle `Tant Que`.



Graphe à 8 sommets - recherche du plus court chemin entre S1 et S8.

Répliquez le tableau ci-dessous <u>pour chaque</u> itération de la “grande” boucle `Tant Que`. Dessinez ensuite le graphe final intégrant tous les coûts `c(i)` de tous les sommets `i` ainsi que la mise en “surbrillance” des arcs pour marquer le prédécesseur de chaque sommet.

La partie “Initialisation” de l’algorithme nous donne le tableau suivant :

	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**
pred	Ø							
c	0	∞	∞	∞	∞	∞	∞	∞
STraités								
SRestants	✓	✓	✓	✓	✓	✓	✓	✓

Partie 3 : l'Algorithme de Floyd-Warshall

Calcul de tous les plus courts chemins d'un graphe[1]

Résumé. Cette partie traite de l'algorithme de Floyd-Warshall et de son implémentation en C++ étape par étape. L'algorithme et une partie du code sont donnés mais c'est à vous de structurer votre programme. Nous essayerons de l'appliquer directement à un cas concret : celui d'un ensemble de villes de votre choix pour lesquelles on ne considérera plus que des routes allant vers le sud.

- **i. C++.** Déclarez les constantes ci-dessous (vous pouvez utiliser des `constexpr`) :

```
const int TAILLEDUGRAPHE = 6;
const int INFINI = 9999999;
```

- **ii. C++.** Déclarez données suivantes :

```
std::vector<string> vecSommets;
int arcs            [TAILLEDUGRAPHE][TAILLEDUGRAPHE];
int cheminCourt     [TAILLEDUGRAPHE][TAILLEDUGRAPHE];
int pointChemin     [TAILLEDUGRAPHE][TAILLEDUGRAPHE];
```

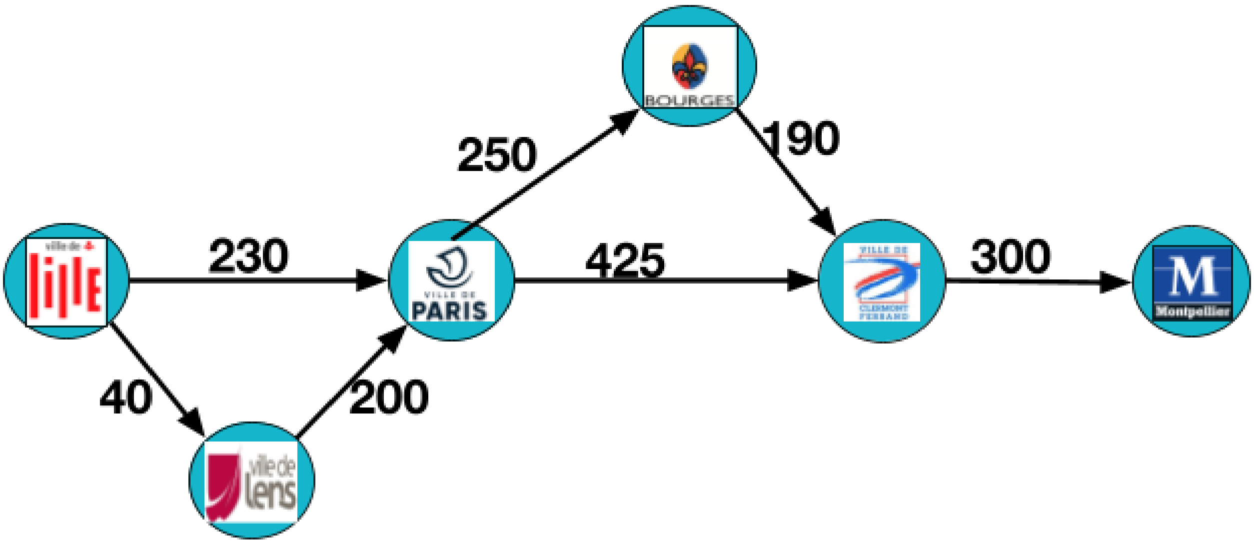
<u>Description des variables et constantes</u> :

- `vecSommets` est un vecteur qui peut prendre le nom d'une ville (`string`).
- `arcs` est un tableau 2 dimensions contenant la distance entre deux sommets/villes. Si `arcs[indiceVille1][indiceVille2] == INFINI`, cela signifie qu'il n'existe pas d'arc direct reliant `indiceVille1` à `indiceVille2`. Si vous avez des doutes sur l'emploi d'`INFINI`, imaginez ce qui se passerait dans un algorithme de plus court chemin avec l'utilisation d'un coût nul ou même négatif pour signifier la non existence d'un arc.
- `cheminCourt` est une matrice de la même taille que `arcs`, c'est elle qui va sauvegarder la distance totale du Plus Court Chemin pour chaque couple de sommets. Ses valeurs de départ sont égales à celles de `arcs`.
- `pointChemin` : À l'image de *prec* dans l'algorithme de Dijkstra, `pointChemin` va sauvegarder les nœuds intermédiaires par lesquels le chemin passe. C'est également une matrice de la même taille que `vecSommets`.
- `tailleGraphe` correspond au nombre de sommets/villes.

[1] i.e., on obtient le plus court chemin pour toutes les paires de sommets.

- **iii. Question**. A quelle structure de données vue en cours, le tableau 2D `arcs` vous fait-il penser[2] ?

- **iv. C++.** Prenez 6 villes de votre choix. Sélectionnez lesquelles doivent être reliées directement par un arc à condition que l'origine soit plus au Nord que la destination (e.g., Lille -> Paris est possible mais pas l'inverse). Vous pouvez adapter l'exemple suivant à partir de vos villes "préférées" en utilisant un outil de type *OpenStreetMap* ou *Google Maps* pour rechercher les distances entre les villes. On ne travaillera qu'avec des entiers. Tout comme l'exemple ci-dessous, considérez peu d'arcs, et donc recherchez peu de distances, afin de ne pas perdre trop de temps ; mais il est évident que le programme qui en ressortira devra marcher pour tout graphe, même les plus grands.



Exemple de graphe où les seuls chemins vont vers le sud.

- **v. C++.** Ajoutez vos 6 sommets/villes dans le vecteur associé avec des instructions de ce type (vous pouvez par exemple les ordonner en mettant la ville la plus au Nord à l'indice 0 et la ville la plus au Sud à l'indice 5) :

```
vecSommets.push_back ("Lille");
```

- **vi. C++.** Mettez toutes les distances des arcs à `INFINI` grâce à 2 boucles `for` pour parcourir la matrice `arcs` puis écrasez cette valeur par de vraies distances pour les arcs que vous avez sélectionnés. Vous pouvez reprendre l'exemple suivant où la distance entre "Clermont-Ferrand" et "Montpellier" est donnée :

```
arcs[4][5] = 300;
```

[2] Voir la partie du cours "Représentation en mémoire d'un graphe".

- **vii. C++.** Paramétrons `pointChemin` qui est propre à l'algorithme de Floyd-Warshall. Initialisez `pointChemin[indiceSommet1][indiceSommet2]` à -1 pour chaque couple (`indiceSommet1, indiceSommet2`). Pour les cas où `arcs[indiceSommet1][indiceSommet2]` est différent de `INFINI`, initialisez le sommet source tel que :

  ```
  pointChemin[indiceSommet1][indiceSommet2] = indiceSommet1;
  ```

  Il faut comprendre ici que `pointChemin` sauvegarde les noeuds intermédiaires[3] afin de pouvoir retrouver tous les nœuds d'un chemin une fois que l'algorithme est terminé. Ceci sera fait après le passage de l'algorithme de Floyd-Warshall. Ici, au début de l'algorithme, on considère que, s'il y a un arc entre `indiceSommet1` et `indiceSommet2`, alors c'est le plus court chemin "courant"[4]. Ceci est identifié par `indiceSommet1` comme ville/sommet intermédiaire. L'évolution de `pointChemin` suite au passage de l'algorithme de Floyd-Warshall aura plus de sens.
  En parallèle, il faut initialiser `cheminCourt[indiceSommet1][indiceSommet2]` tel qu'il puisse prendre le coût de l'arc associé ou alors ∞, et ceci en fonction des valeurs affectés par `pointChemin[indiceSommet1][indiceSommet2]`.

- **viii. Algo.** Prenez le temps de comprendre l'algorithme suivant en l'appliquant <u>à la main</u> sur de très petits exemples (3 ou 4 sommets) en faisant évoluer les valeurs des matrices `cheminCourt` et `pointChemin`. Plus difficile : réfléchissez à comment retrouver tous les sommets d'un plus court chemin entre deux sommets/villes.

```
Algorithme Floyd-Warshall (Partie Principale)
Pour un sommetIntermédiaire d'indice 1 à TAILLEDUGRAPHE Faire
 Pour un origine d'indice 1 à TAILLEDUGRAPHE Faire
    Pour une destination d'indice 1 à TAILLEDUGRAPHE Faire
        Si cheminCourt[origine][sommetIntermédiaire] +
           cheminCourt[sommetIntermédiaire][destination]
           < cheminCourt[origine][destination]
       Alors
           cheminCourt[origine][destination] =
           cheminCourt[origine][sommetIntermédiaire]  +
           cheminCourt[sommetIntermédiaire][destination];
           pointChemin[origine][destination] = sommetIntermédiaire;
       Fin Si
    Fin Pour
 Fin Pour
Fin Pour
```

[3] le "prec" de l'algorithme de Dijkstra a une fonction similaire.
[4] "Courant" est employé ici du fait que l'algorithme peut ensuite trouver un plus court chemin entre deux noeuds que celui formé par l'arc les reliant.

- **ix. Question.** Déterminez le nombre d'instructions élémentaires maximales à réaliser dans l'**`Algorithme`** de **`Floyd-Warshall`** en fonction de n == `TAILLEDUGRAPHE`.

- **x. C++.** Implémentez l'algorithme de Floyd-Warshall.

- **xi. Compréhension Algo et C++.**
  Objectif : récupération des données pour l'affichage d'un chemin.
  Le code C++ donné ci-dessous correspond à une méthode d'affichage **des noeuds intermédiaires**[5] d'un plus court chemin pour deux sommets/villes `sommet1` et `sommet2`. Le booléen en entrée n'est vrai qu'une fois (au début, par exemple lors d'un appel à partir du *main*) pour signifier que `sommet1` est le sommet de départ (et donc nous n'avons pas commencé d'afficher les nœuds intermédiaires). Ceci permet de débuter la liste des nœuds intermédiaires avec '`par`' et de continuer, s'il existe encore de tels nœuds, par '`et`'. Un exemple d'affichage est donné plus tard dans le sujet avec ces deux mots clés.

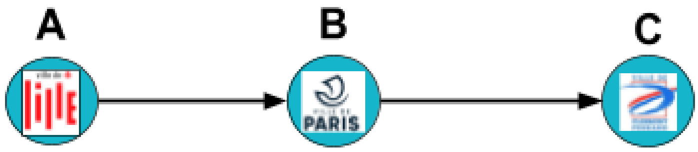
```
void afficherChemin (int sommet1, int sommet2, bool & premier)
{
   int suivant = pointChemin[sommet1][sommet2];
   if (suivant==-1 || sommet1 == sommet2 || suivant == sommet1 || suivant == sommet2)
       return;
   afficherChemin (sommet1, suivant, premier);
   if (premier)                    {
        std::cout << " \t\t par ";
        premier = false;           }
   else                            {
        std::cout << " et ";       }
   std::cout << vecSommets[suivant];
   afficherChemin (suivant, sommet2, premier);
}
```

- **xii. Question.** Quelle est la particularité de cette méthode ?

[5] Seuls les nœuds intermédiaires, s'ils existent, sont affichés par cette méthode. C'est une autre méthode qui appellera celle-ci qui affichera les nœuds sources et les nœuds destinations.

- **xiii. Algo.** Essayez de dérouler `afficherChemin` (i.e., faire la trace) avec 3 nœuds `A`, `B`, `C` (vous pouvez remplacer les 'int' par ces 3 'char' pour cet exemple afin de simplifier les choses en confondant `vecSommets[indiceNoeud] et indiceNoeud`) afin d'afficher le plus court chemin de A vers C en passant B comme dans le graphe ci-dessous. L'appel à la fonction est le suivant : `afficherChemin (A, C, true)`.
  Aide pour la trace :
  - `pointChemin[A][C]` donne `B`,
  - `pointChemin[B][C]` donne (aussi) `B`.

  A vous de comprendre ce que donnera `pointChemin[A][B]` si vous le rencontrez dans la trace.



- **xiv. C++.** Développez, et adaptez si nécessaire, `afficherChemin`. Lors de l'utilisation de cette méthode, le booléen `premier` doit être initialisé à `true`.
- **xv. C++.** Affichez tous les chemins de votre graphe. Cette fois c'est à vous de développer de A à Z une méthode **`affichagePCCh`** qui va afficher tous les Plus Courts Chemins pour chaque paire de sommets/villes de votre graphe. Appelez donc `afficherChemin` pour tous les cas plausibles, et attention aux arcs qui n'existent pas (distances `INFINI`). Voici ce que cette méthode peut produire avec le graphe de l'exemple donné plus haut :

```
Sommets intermédiaires du PCCh de 40km,(s'ils existent) de Lille vers Lens.

Sommets intermédiaires du PCCh de 230km,(s'ils existent) de Lille vers Paris.

Sommets intermédiaires du PCCh de 480km,(s'ils existent) de Lille vers Bourge.
                par Paris
Sommets intermédiaires du PCCh de 655km,(s'ils existent) de Lille vers Clermont-Ferrand.
                par Paris
Sommets intermédiaires du PCCh de 955km,(s'ils existent) de Lille vers Montpellier.
                par Paris et Clermont-Ferrand
Sommets intermédiaires du PCCh de 200km,(s'ils existent) de Lens vers Paris.

Sommets intermédiaires du PCCh de 450km,(s'ils existent) de Lens vers Bourge.
                par Paris
Sommets intermédiaires du PCCh de 625km,(s'ils existent) de Lens vers Clermont-Ferrand.
                par Paris
Sommets intermédiaires du PCCh de 925km,(s'ils existent) de Lens vers Montpellier.
                par Paris et Clermont-Ferrand
Sommets intermédiaires du PCCh de 250km,(s'ils existent) de Paris vers Bourge.

Sommets intermédiaires du PCCh de 425km,(s'ils existent) de Paris vers Clermont-Ferrand.

Sommets intermédiaires du PCCh de 725km,(s'ils existent) de Paris vers Montpellier.
                par Clermont-Ferrand
Sommets intermédiaires du PCCh de 190km,(s'ils existent) de Bourge vers Clermont-Ferrand.

Sommets intermédiaires du PCCh de 490km,(s'ils existent) de Bourge vers Montpellier.
                par Clermont-Ferrand
Sommets intermédiaires du PCCh de 300km,(s'ils existent) de Clermont-Ferrand vers Montpellier.
```

Affichage de tous les plus courts chemins et de leurs sommets intermédiaires.

Conclusion et remarques. L'algorithme de Floyd-Warshall est un algorithme de type programmation dynamique. C'est une méthode algorithmique que nous verrons en cours. Les algorithmes de ce type décomposent le problème en une série de sous-problèmes, et, si nécessaire, ces derniers seront eux même décomposés en "sous"-sous-problèmes et ainsi de suite. Les plus petits sous-problèmes sont alors résolus et leurs résultats sont remontés jusqu'à obtenir la solution au problème de base. La particularité de la programmation dynamique est de sauvegarder les résultats des sous-problèmes puisqu'ils peuvent ressurgir à d'autres moments de la résolution. Ici, on sauvegarde ces "sous-problèmes" grâce aux deux matrices `cheminCourt` et `pointChemin`. Par exemple, si nous disposons des plus courts chemins entre A et B et entre B et C, la sauvegarde de ces résultats nous permet de mettre à jour `cheminCourt[A][C]` si l'expression suivante est vraie :

```
cheminCourt[A][B] + cheminCourt[B][C]  < cheminCourt[A][C].
```

BONUS.

a. Imaginez d'autres situations où ces algorithmes sont utiles. Par exemple, les poids des arcs ne seraient plus des distances mais plutôt des coûts de stockage et de transfert de produits (resp. pour des sommets "temps et espace"). Imaginez alors un tel scénario et obtenez son optimisation grâce à l'algorithme développé dans ce TD&P.
b. Pour l'un des deux algorithmes que vous choisirez, que pensez-vous du traitement des graphes où certains poids sont négatifs ? En vous rappelant de ce qu'est un chemin absorbant, donner les conditions pour que l'on puisse appliquer l'algorithme en imaginant des pistes permettant d'arriver à un résultat.
c. (Difficile) Retrouver l'ensemble des résultats obtenus par l'algorithme de Floyd-Warshall avec Dijkstra. Essayez d'évaluer leurs performances par leur temps CPU (si besoin sur beaucoup de réplications du même calcul).
d. (Encore plus difficile). Faites quelques recherches sur la Boost Graph Library[6] afin de l'adapter à votre environnement de programmation. Vous y trouverez une implémentation de l'algorithme de Dijkstra :
    - https://www.boost.org/doc/libs/1_41_0/libs/graph/doc/dijkstra_shortest_paths.html

   Comparez votre implémentation et celle de la Boost Graph Library appliquée à chaque paire de sommets, en mesurant les temps de calculs sur les exemples vus en cours et TD&P.

Bon courage.

[6] La Boost Graph Library est généralement moins mature et moins standard que la STL mais elle reste une bonne base sur laquelle le langage C++ évolue.