Introduction

Ce programme est conçu pour gérer des graphes et effectuer diverses opérations sur ceux-ci, notamment l'algorithme de Prim et le calcul de la somme des poids entre deux points. Il permet aussi la création, la sauvegarde et le chargement de vos travaux.

A)Partie algorithmique

1) Structures

La structure Voisin est utilisée pour définir une arête dans le graphe, en spécifiant les nœuds qu'elle relie et le poids de cette connexion.

```
typedef struct {
    char self[50];
    char nom_lien[50];
    float poids;
} Voisin;
```

- self : Une chaîne de caractères représentant le nom du nœud source de l'arête. Ce champ identifie le nœud de départ de l'arête.
- nom_lien : Une chaîne de caractères représentant le nom du nœud destination de l'arête. Ce champ identifie le nœud d'arrivée de l'arête.
- poids : Un entier représentant le poids de l'arête. Ce champ stocke la valeur associée à la connexion entre les deux nœuds, souvent utilisé pour représenter des coûts ou des distances.

La structure Points est utilisée pour créer une liste d'adjacence, où chaque élément représente une arête sortant d'un nœud donné. Cela permet de stocker toutes les connexions d'un nœud dans une liste chaînée.

```
typedef struct Points {
    Voisin voisin;
    struct Points* next;
} Points;
```

- voisin : Une instance de la structure Voisin, représentant une arête dans le graphe.
- next : Un pointeur vers le prochain élément de type Points. Ce champ permet de créer une liste chaînée de nœuds, où chaque nœud contient une arête et un pointeur vers le nœud suivant.

La structure Graphe représente l'ensemble du graphe. Elle utilise une liste d'adjacence pour stocker les arêtes du graphe. Le pointeur head pointe vers le premier élément de cette liste, permettant ainsi de parcourir toutes les arêtes du graphe.

```
typedef struct {
    Points* head;
} Graphe;
```

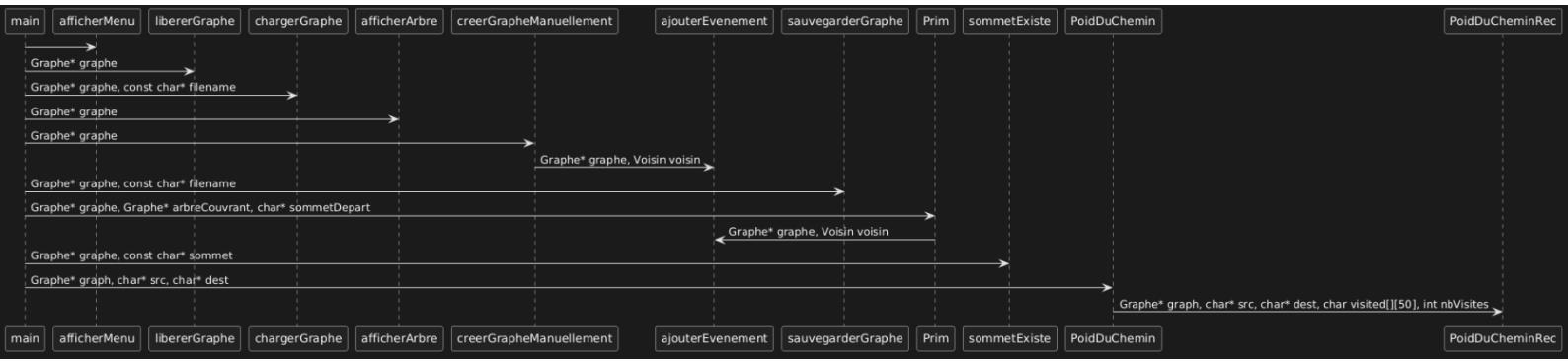
- head : Un pointeur vers le premier élément de la liste d'adjacence de type Points. Ce champ représente le début de la liste des arêtes du graphe.

2) Fonctions

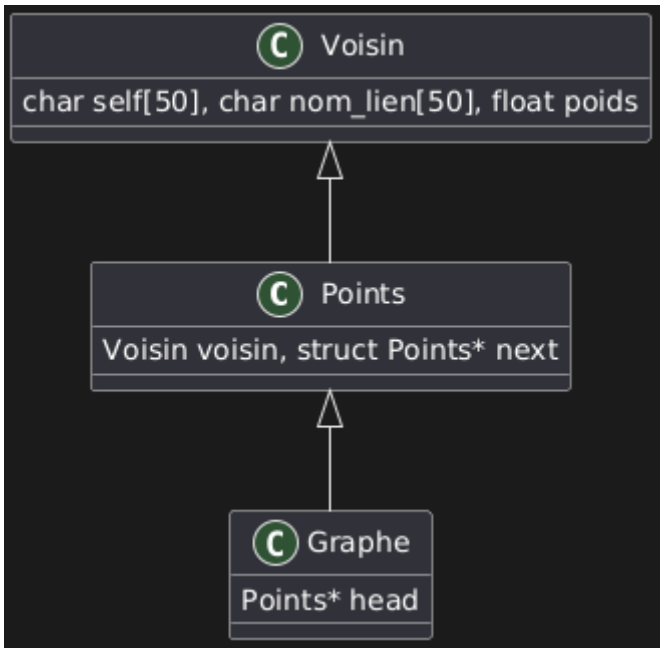
- ajouterEvenement : Ajoute une arête (représentée par un Voisin) à la liste d'adjacence d'un graphe.
- chargerGraphe : Charge un graphe à partir d'un fichier texte, en lisant chaque arête et son poids.
- sauvegarderGraphe : Sauvegarde le graphe actuel dans un fichier texte, en écrivant chaque arête et son poids.
- libererGraphe : Libère la mémoire allouée pour un graphe, en parcourant et supprimant chaque élément de la liste d'adjacence.
- afficherMenu : Affiche un menu interactif permettant à l'utilisateur de choisir parmi différentes options pour manipuler le graphe.
- afficherArbre : Affiche toutes les arêtes d'un graphe ou d'un arbre couvrant, en parcourant la liste d'adjacence.
- Prim : Implémente l'algorithme de Prim pour générer un arbre couvrant minimal à partir d'un graphe donné.
- creerGrapheManuellement : Permet à l'utilisateur de créer manuellement un graphe en ajoutant des arêtes via une interface interactive.
- PoidDuCheminRec : explore le graphe pour donner le poids entre deux points
- PoidDuChemin : initialise la fonction récursive qui calcule le poids entre deux points
- sommetExiste :parcours le graphe pour savoir si le sommet donné appartient au graphe

B)Partie UML

1) Diagramme de séquence



2) Diagramme de classe



C) Partie test

1) Charger un graphe

Le format de sauvegarde des arêtes est un format d'arrête oriente. A -> B : 8 veut dire qu'il existe un chemin en sens unique de poids 8 de A vers B. Pour l'ajout d'arête non oriente, il faut absolument ajouter le chemin contraire, c'est-à-dire le chemin de B vers A tel que : B -> A : 8. Le chargement d'un graphe supprimera le graphe en cours grâce a la fonction libererGraphe. Sauvegarder le graphe en cours avant tout nouveau chargement.

Nous devons renseigner le nom d'un graphe existant. Une erreur nous ramène au menu :

```
Entrez le nom du fichier à charger: f
Erreur lors de l'ouverture du fichier: No such file or directory
```

L'ouverture d'un fichier contenant une arrête négative aura un message d'erreur mais chargera tout de même le graphe, excepté l'arête ayant un poids négatif :

```
Entrez le nom du fichier à charger: exemple2.txt
Erreur: poids négatif détecté pour l'arête A -> B
F -> E : 5
D -> E : 6
D -> F : 3
C -> E : 5
C -> F : 3
B -> E : 4
B -> D : 2
B -> C : 3
A -> E : 3
A -> D : 3
```

Une erreur de nomenclature renviera un message d'erreur sur la ligne en question qui ne serra pas chargé, le reste du graphe sera bien chargé. La nomenclature doit être sous ce format : string -> string : float

```
Entrez le nom du fichier à charger: exemple2.txt
Erreur: format incorrect dans la ligne: A -> B 2
F -> E : 5
D -> E : 6
D -> F : 3
C -> E : 5
C -> F : 3
B -> E : 4
B -> D : 2
B -> C : 3
A -> E : 3
A -> D : 3
```

L'ouverture d'un bon fichier nous fera un affichage de notre graphe :

```
Entrez le nom du fichier à charger: g.txt
E -> F : 5
E -> D : 6
E -> C : 5
E -> B : 4
E -> A : 3
F -> E : 5
F -> C : 3
F -> D : 3
D -> E : 6
D -> F : 3
D -> A : 3
D -> B : 2
C -> E : 5
C -> F : 3
C -> B : 3
B -> E : 4
B -> D : 2
B -> C : 3
B -> A : 2
A -> E : 3
A -> D : 3
A -> B : 2
```

2) Créer des arêtes

La création d'une arête est au format orienté. L'insertion d'une seul arete permettra de créer une paire d'arête pour un graphe non orienté. Pour ne pas oublier ce détail, un message de rappel est présent.

```
Entrez le nombre d'arêtes à ajouter : (Une seul à ajouter, automatiquement en non orienté)
```

Un nombre d'arête est attendu. Une lettre ou un nombre négatif ne sont pas accepté :

```
Entrez le nombre d'arêtes à ajouter : (Une seul à ajouter, automatiquement en non orienté) -9
Erreur: veuillez entrer un nombre entier positif.
Entrez le nombre d'arêtes à ajouter: g
Erreur: veuillez entrer un nombre entier positif.
```

Lors de l'ajout d'arête, la nomenclature doit être respecte :

```
Entrez l'arête 1 (format: A -> B : poids): a
Erreur: format incorrect ou poids négatif. Veuillez réessayer.
Entrez l'arête 1 (format: A -> B : poids): a->r:3
Erreur: format incorrect ou poids négatif. Veuillez réessayer.
```

L'ajout d'un poids négatif provoquera une erreur :

```
Entrez l'arête 1 (format: A -> B : poids): A -> B : -5
Erreur: format incorrect ou poids négatif. Veuillez réessayer.
```

A la fin de l'ajout, nous retournons au menu. La nomenclature doit être sous ce format : string -> string : float. Ci-dessous un exemple d'arête compatible avec ce format :

```
Entrez l'arête 1 (format: A -> B : poids): 1 -> B : 1
Menu:
1. Charger un graphe
2. Créer des arêtes
3. Sauvegarder un graphe
4. Générer un arbre couvrant à partir du graphe chargé
5. Charger un arbre
6. Sauvegarder un arbre
7. Distance d'un point par rapport au point de départ de l'arbre
8. Quitter
```

3) Sauvegarder un graphe

Le format de sauvegarde d'un graphe est libre. Le format .txt est à privilégier. La sauvegarde d'un nouveau graphe sur un fichier déjà existant écrasera les anciennes données. Un graphe vide ne pourra pas être sauvegardé :

```
Le graphe est vide et ne peut pas être sauvegardé.
```

4) Générer un arbre couvrant

Cette fonction sert à créer l'arbre couvrant. Un sommet de départ est demandé. La saisie d'un sommet inexistant nous renvoie au menu :

```
Entrez le sommet de départ: t
Le sommet de départ n'existe pas dans le graphe.
```

```
Entrez le sommet de départ: 2
Le sommet de départ n'existe pas dans le graphe.
```

Un sommet valide construira notre arbre couvrant. Ce dernier sera alors affiché dans la console en format non orienté :

```
Entrez le sommet de départ: A
E -> A : 3
A -> E : 3
C -> F : 3
F -> C : 3
F -> D : 3
D -> F : 3
D -> B : 2
B -> D : 2
B -> A : 2
A -> B : 2
```

5) Charger un graphe

Semblable à la fonction charger graphe, nous pouvons mettre le format que l'on souhaite, le .txt étant à privilégier. Un fichier inexistant nous enverra au menu :

```
Entrez le nom du fichier à charger pour l'arbre: f
Erreur lors de l'ouverture du fichier: No such file or directory
```

Un arbre connu sera affiché :

```
Entrez le nom du fichier à charger pour l'arbre: arbre.txt
A -> B : 2
B -> A : 2
B -> D : 2
D -> B : 2
D -> F : 3
F -> D : 3
F -> C : 3
C -> F : 3
A -> E : 3
E -> A : 3
```

Attention, ouvrir un fichier qui n'est pas un arbre couvrant entrainera un échec du calcul de la distance entre deux points. Si ce fichier est modifié à la main, il devra respecter la même nomenclature que la partie Charger un graphe (graphe non orienté, string -> string : float).

6) Sauvegarder un arbre

La sauvegarde réussi d'un arbre nous ramène au menu. La sauvegarde d'un fichier avec un nom déjà existant remplacera l'ancien fichier. Sauvegarder un arbre sera accessible uniquement si un arbre a été créé :

```
L'arbre couvrant n'a pas encore été généré. Veuillez générer l'arbre couvrant d'abord.
```

7) Distance d'un point par rapport à un autre

Choisir cette option alors qu'aucun arbre couvrant n'a été créé nous enverra au menu :

```
L'arbre couvrant n'a pas encore été généré. Veuillez générer l'arbre couvrant d'abord.
```

Le choix d'un sommet de départ non présent dans notre arbre couvrant n'est pas possible :

```
Entrez le sommet de départ pour calculer les distances: p
Le sommet de départ 'p' n'existe pas dans l'arbre couvrant. Veuillez réessayer.
```

Un bon choix nous demandera un deuxième point afin de calculer la distance entre les deux :

```
Entrez le sommet de départ pour calculer les distances: A
Entrez le sommet d'arrivée pour calculer les distances:
```

Un mauvais deuxième choix est aussi bloquant :

```
Entrez le sommet d'arrivée pour calculer les distances: b
Le sommet d'arrivée 'b' n'existe pas dans l'arbre couvrant. Veuillez réessayer.
```

Une fois le deuxième choix réussi, la distance entre les deux points nous est donnée :

```
Le chemin entre A et F est : 7
```

8) Affichage

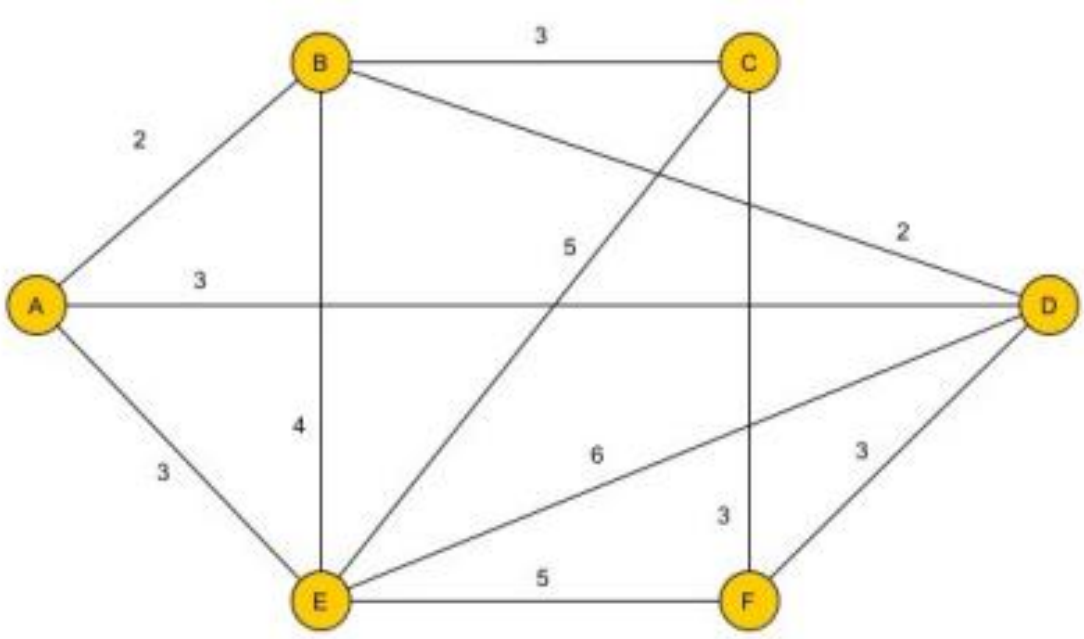
Les options 8 et 9 servent à afficher un arbre et un graphe, à condition que ces derniers existent :

```
Choisissez une option: 8
Le graphe n'a pas encore été généré.
```

```
Choisissez une option: 9
L'arbre couvrant n'a pas encore été généré. Veuillez générer l'arbre couvrant d'abord.
```

9) Exemples

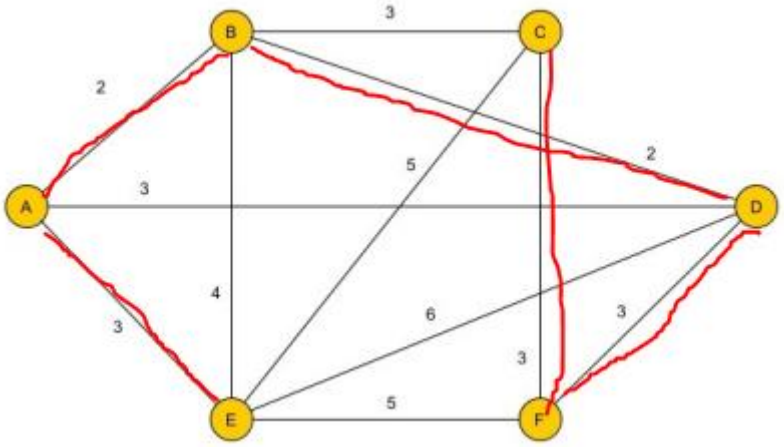
a) exemple_tp2



L’arbre couvrant créé par mon code donne :

```
Entrez le sommet de départ: A
E -> A : 3
A -> E : 3
C -> F : 3
F -> C : 3
F -> D : 3
D -> F : 3
D -> B : 2
B -> D : 2
B -> A : 2
A -> B : 2
```

En faisant l’algorithme de Prim à la main, j’ai :

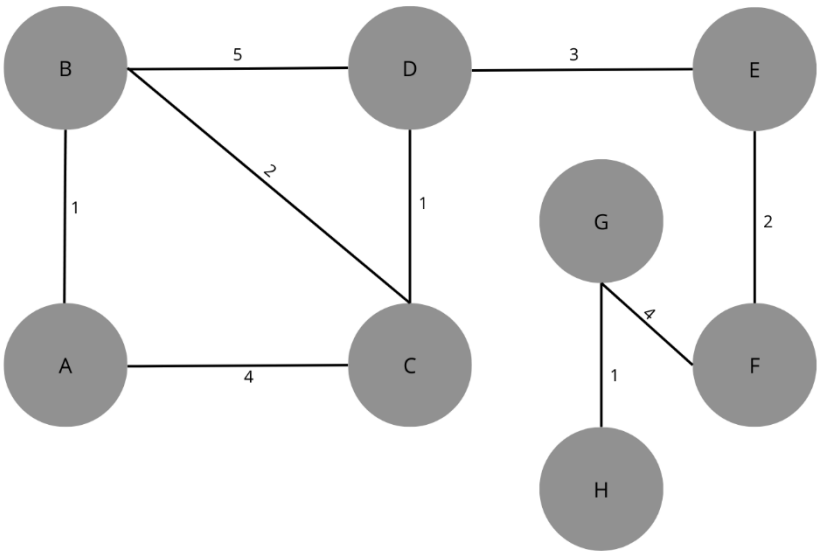


Mon code est cohérent. En plus la distance entre A et F est 7.

```
Entrez le sommet de départ pour calculer les distances: A
Entrez le sommet d'arrivée pour calculer les distances: F
Le chemin entre A et F est : 7
```

L'algorithme de Prim et la fonction de calcul fonctionnent correctement, mon code est donc cohérent.

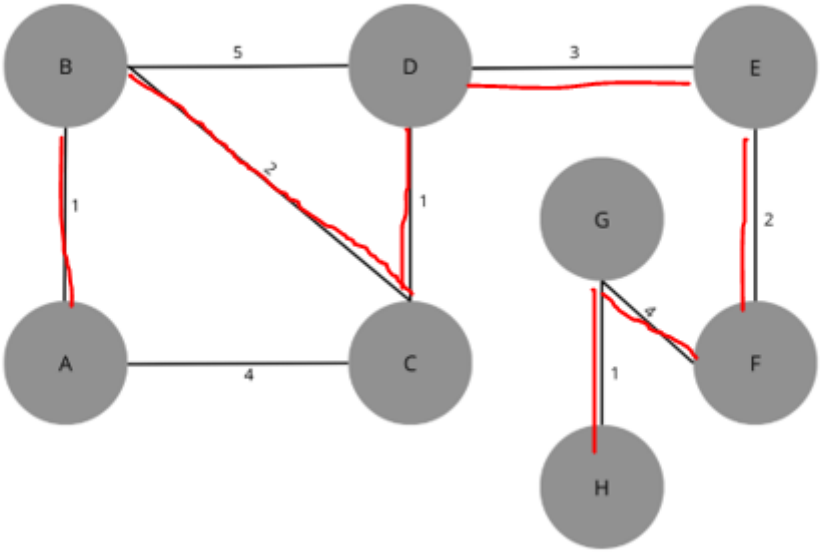
b) exemple1.txt



L'arbre couvrant créé par mon code donne :

```
Entrez le sommet de départ: A
H -> G : 1
G -> H : 1
G -> F : 4
F -> G : 4
F -> E : 2
E -> F : 2
E -> D : 3
D -> E : 3
D -> C : 1
C -> D : 1
C -> B : 2
B -> C : 2
B -> A : 1
A -> B : 1
```

En faisant l'algorithme de Prim à la main, j'ai :



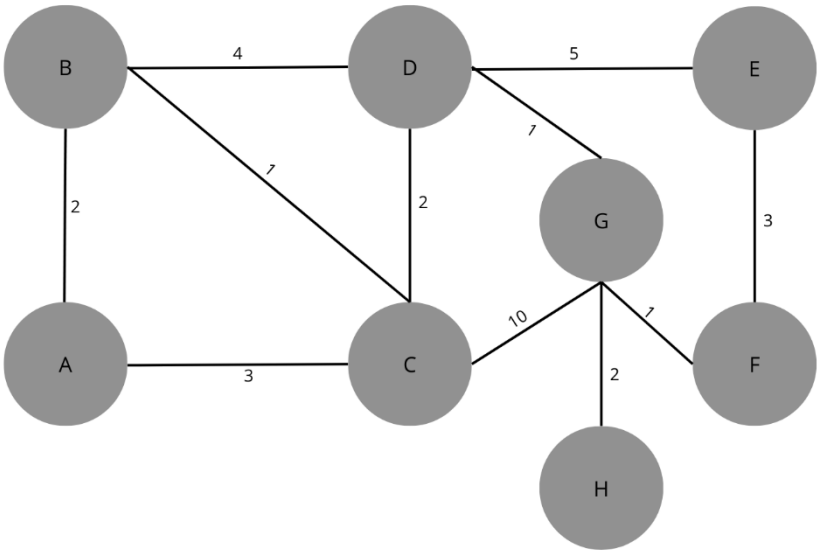
Mon code est cohérent. En plus la distance entre A et D est 4.

```
Entrez le sommet de départ pour calculer les distances: A
Entrez le sommet d'arrivée pour calculer les distances: D
Le chemin entre A et D est : 4
```

L'algorithme de Prim et la fonction de calcul fonctionnent correctement, mon code est donc cohérent.

c) exemple2.txt

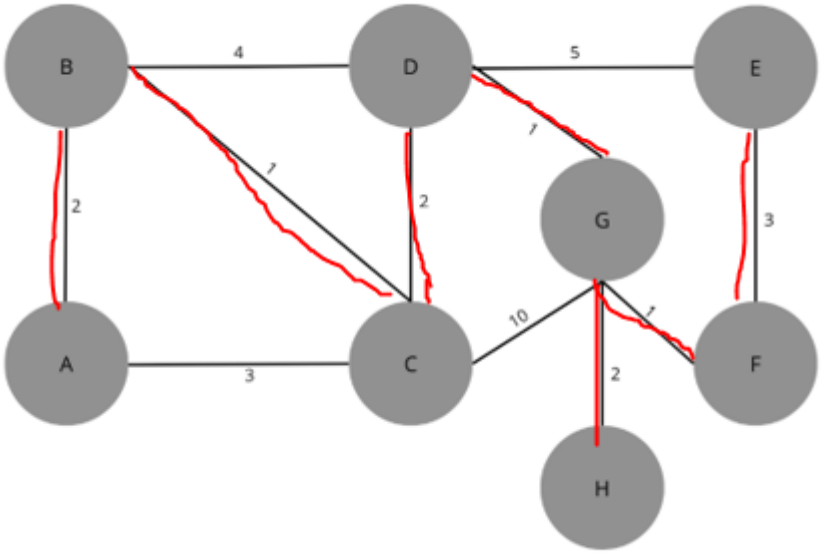
Ci-dessous un autre exemple de graphe :



L’arbre couvrant créé par mon code donne :

```
Entrez le sommet de départ: A
E -> F : 3
F -> E : 3
H -> G : 2
G -> H : 2
F -> G : 1
G -> F : 1
G -> D : 1
D -> G : 1
D -> C : 2
C -> D : 2
C -> B : 1
B -> C : 1
B -> A : 2
A -> B : 2
```

En faisant l’algorithme de Prim à la main, j’ai :

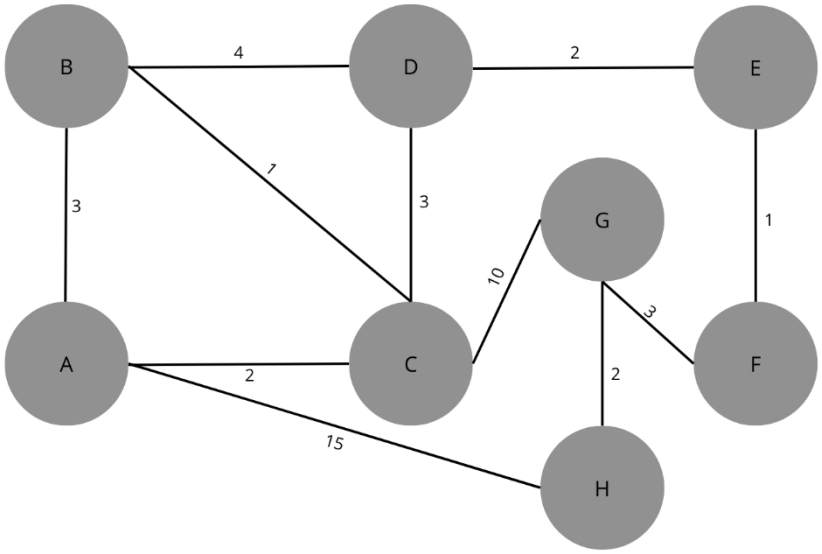


Mon code est cohérent. En plus la distance entre A et H est 8.

```
Entrez le sommet de départ pour calculer les distances: A
Entrez le sommet d'arrivée pour calculer les distances: H
Le chemin entre A et H est : 8
```

L’algorithme de Prim et la fonction de calcul fonctionnent correctement, mon code est donc cohérent.

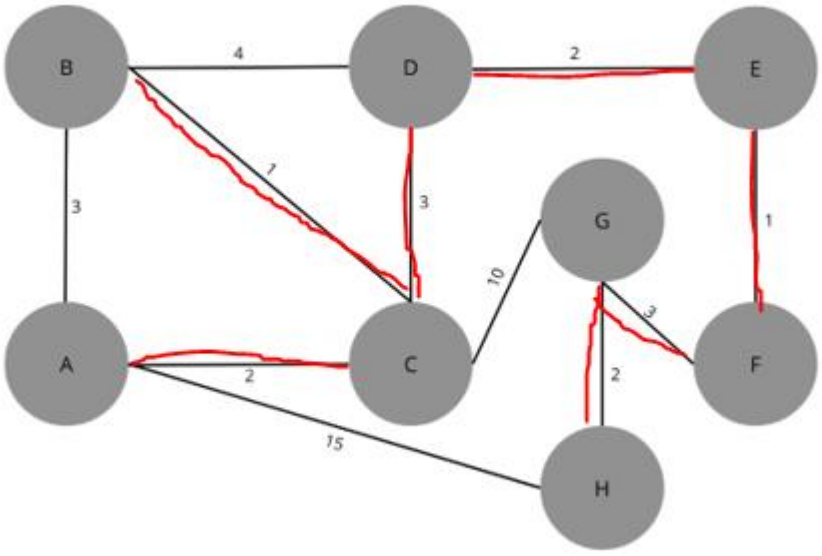
d) exemple3.txt



L'arbre couvrant créé par mon code donne :

```
Entrez le sommet de départ: A
H -> G : 2
G -> H : 2
G -> F : 3
F -> G : 3
F -> E : 1
E -> F : 1
E -> D : 2
D -> E : 2
D -> C : 3
C -> D : 3
B -> C : 1
C -> B : 1
C -> A : 2
A -> C : 2
```

En faisant l'algorithme de Prim à la main, j'ai :



Mon code est cohérent. En plus la distance entre A et H est 13.

```
Entrez le sommet de départ pour calculer les distances: A
Entrez le sommet d'arrivée pour calculer les distances: H
Le chemin entre A et H est : 13
```

L'algorithme de Prim et la fonction de calcul fonctionnent correctement, mon code est donc cohérent.

Conclusion

Le développement de ce code C m'a permis de continuer à apprendre et comprendre ce langage. Le programme est interactif via le petit menu. La mémoire est gérée et optimisée grâce aux structures et à la liste chainée. Ce TP 2 était simple en apparence mais m'a donné du fil à retordre lors de l'implémentation de Prim ainsi que pour le calcul de poids entre deux points.