Algorithmique et Génome

En biologie moléculaire, l'assemblage de fragments d'une séquence d'ADN est une opération fondamentale. En effet, les procédés de *séquençage* (identification de la séquence de bases A, C, G, T) actuels ne peuvent traiter des molécules complètes, qui ont typiquement entre 10000 et 100000 paires de bases (PBs). Ces procédés fournissent des fragments de quelques centaines de PBs, prélevés aléatoirement, qui correspondent à des facteurs de la séquence recherchée. On obtient typiquement 500 à 2000 de ces fragments, qui présentent entre eux de nombreux recouvrements.

Le problème de la plus courte sur-séquence commune (SSC) d'un ensemble de séquences est une idéalisation du problème de l'assemblage. Soit une collection F de mots sur l'alphabet {A,C,G,T}, il s'agit de trouver un mot S de longueur minimale tel que tout mot f de F soit un facteur de S. Par exemple, si F={TAG,CTA,ACT}, les mots TAGCTAACT, CTACTAG et ACTAG sont des sur-séquences communes de F, et le mot S=ACTAG est la plus courte sur-séquence commune de F.

On peut voir également ce problème comme un problème de compression, la solution est en effet une représentation compacte des données d'origine. Dans l'exemple ci-dessus, le taux de compression est de 5/9.

On supposera, sans perte de généralité, qu'aucun mot de F n'est lui-même un facteur d'un autre mot de F. En effet, si u est facteur d'un autre mot de F, alors une solution pour F - $\{u\}$ est également une solution pour F.

Question 1)

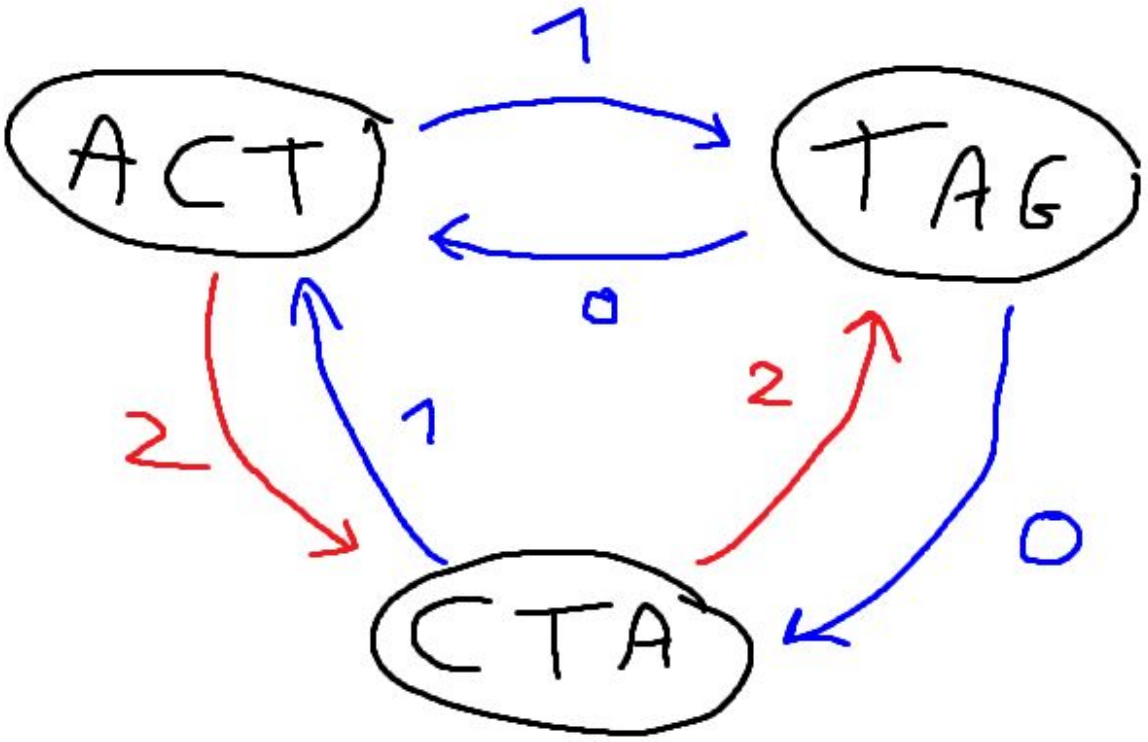
Il est possible de représenter ce problème avec une structure de graphe pondéré.

Pour obtenir la meilleure compression possible entre deux mots, le but est de mettre à la suite des mots dont le suffixe du premier et le préfixe du second ont un maximum de lettres en commun, afin de se servir des lettres du premier mot pour composer le second dans S, et donc réduire le nombre de lettres à “écrire”.

Ainsi, pour une collection donnée, il est possible d’établir un graphe pondéré où chaque mot est un sommet, chaque arc étant composé à la queue du premier mot, et à la tête du second, la pondération correspondant au nombre de lettres en commun entre suffixe du premier mot et préfixe du second, et chaque sommet est relié à tous les autres.

Ainsi, le mot SSC optimal est obtenu en empruntant le chemin passant une seule fois par tous les sommets du graphe et dont la longueur est la plus importante.

Exemple sur la collection F de l'énoncé, le chemin à prendre étant indiqué en rouge :



On peut y voir un parallèle avec le problème du voyageur de commerce, sauf qu'ici on ne cherche pas le chemin le plus court reliant tous les sommets, mais le plus long.

Question 2 :

Le problème du voyageur est connu pour ne pas avoir d'algorithme générique d'approximation pour être résolu.

La méthode naïve pour sa résolution consiste à énumérer tous les chemins possibles et de sélectionner le plus court. Dans notre cas, nous sélectionnons alors le plus long. Cette solution a une complexité en O(n!), ce qui devient vite délirant pour des grandes collections.

Il est cependant possible de réduire le temps de calcul en faisant de la programmation dynamique, comme démontré par Held et Kerp dans *A Dynamic Programming Approach to Sequencing Problems* (1962), réduisant ce temps à $O(n^2 * 2^n)$.

Nous pouvons utiliser ceci pour améliorer la solution naïve.

Ainsi, pour une collection donnée, on peut partir d'un sommet du graphe, puis pour atteindre le suivant, choisir celui dont la pondération de l'arc pour le rejoindre est la plus grande. On crée alors un autre graphe en remplaçant les deux sommets choisis par un seul sommet formant le mot obtenu en combinant les deux précédents.

Il ne reste plus qu'à recommencer sur ce même graphe, jusqu'à obtenir un graphe ne comportant qu'un seul sommet, qui sera la solution.

Ainsi, on peut écrire l'algorithme de cette manière, en supposant que :

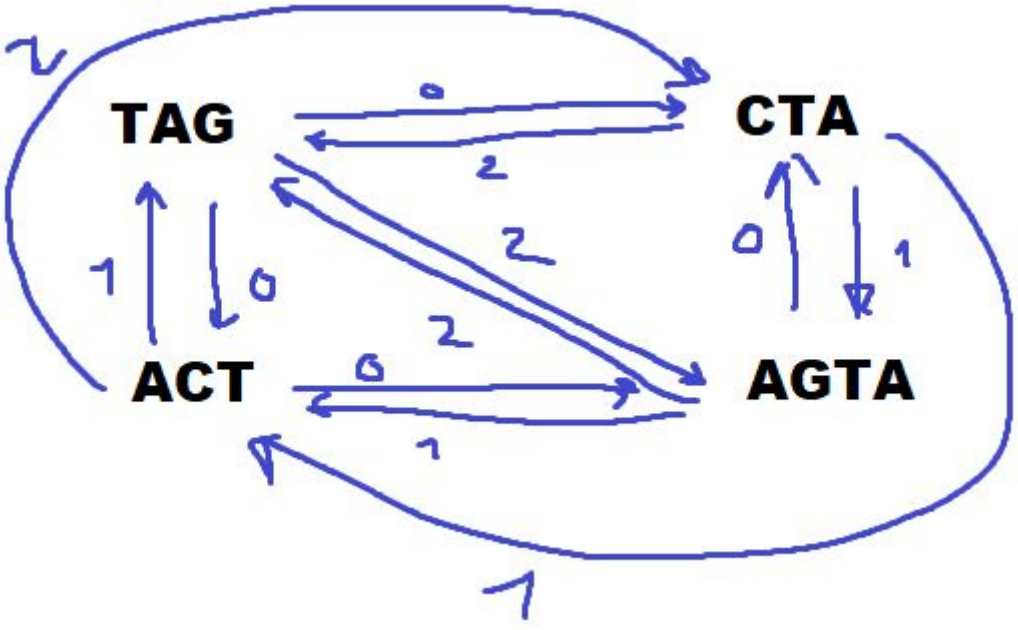
- la fonction CreeGrapheRecouvrement(Collection F) permet de créer le graphe des recouvrements de F comme expliqué dans la question 1 (sera détaillé en Question 4) .
- la fonction MeilleurArc(Sommet S) trouve parmi les différents arcs de S celui dont la pondération est la plus forte.
- la fonction Recouvrement(Mot u, Mot v) réalise le recouvrement entre les mots u et v.

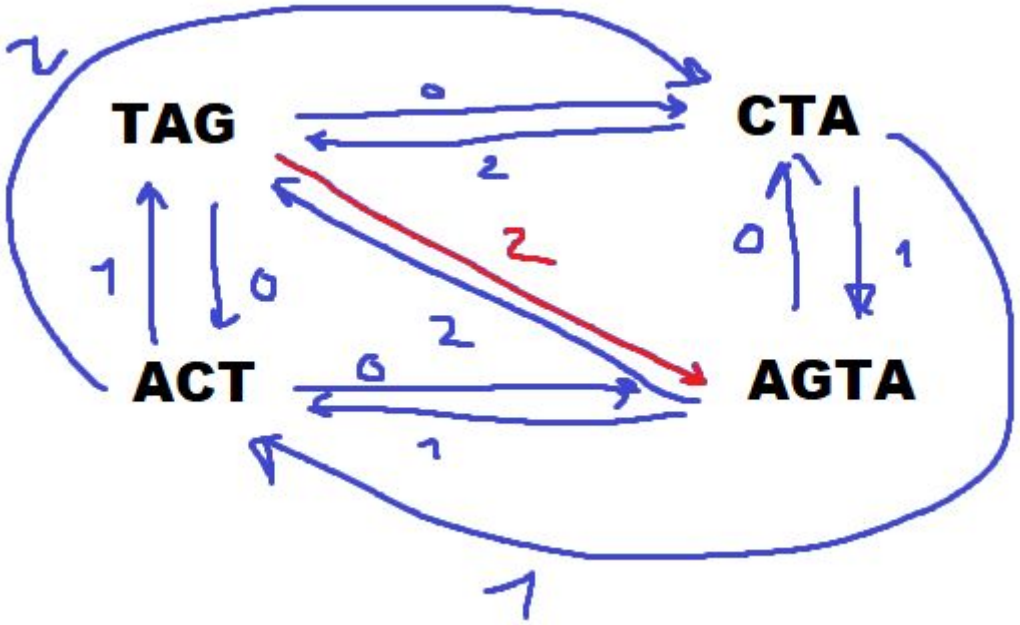
```
RechercheSCC ( Collection F) :
    Si Card(F) == 1 :
        Retourner le seul mot de F;
    Sinon :
        Graphe G = CreeGrapheRecouvrement(F);
        Sommet S = Un sommet aléatoire de G;
        Arc A = MeilleurArc(S);
        F = F \ {Tete(A), Queue(A)};
        Mot R = Recouvrement(Queue(A), Tete(A));
        F = F + {R};
        Retourner RechercheSCC (F);
```

Exemple avec F = {TAG, CTA, ACT, AGTA}

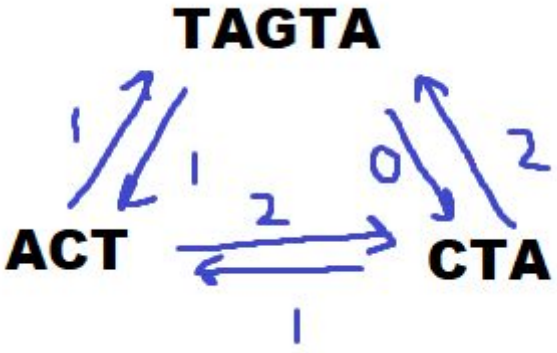
ACTAGTA est une SSC (de même que TAGTACT) si on suit le cheminement suivant :

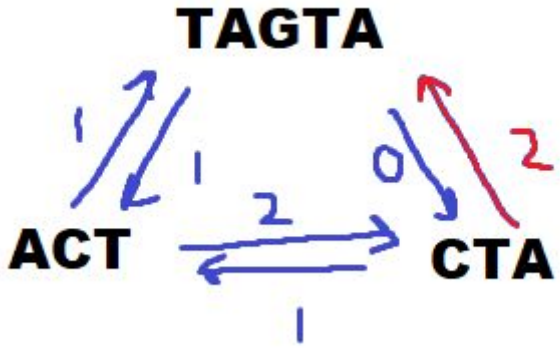
<u>Etape 1 :</u>



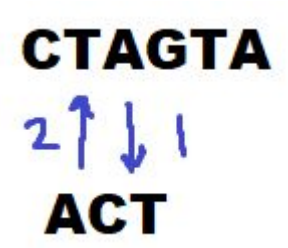


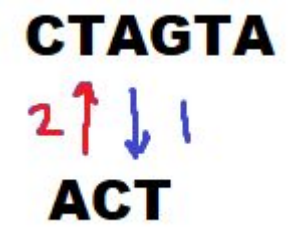
Etape 2 :





Etape 3 :





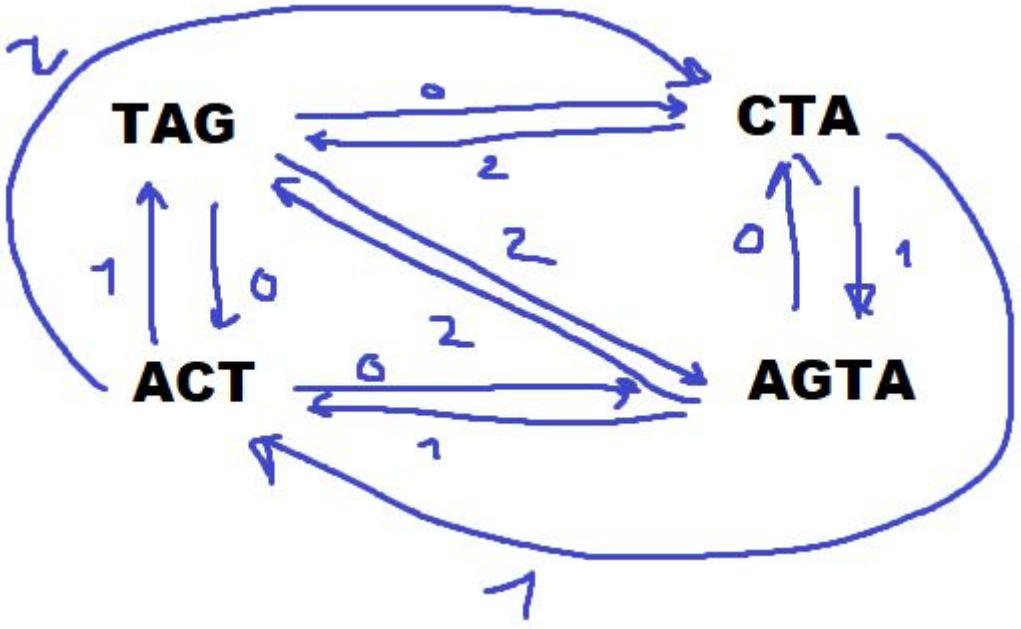
Résultat :

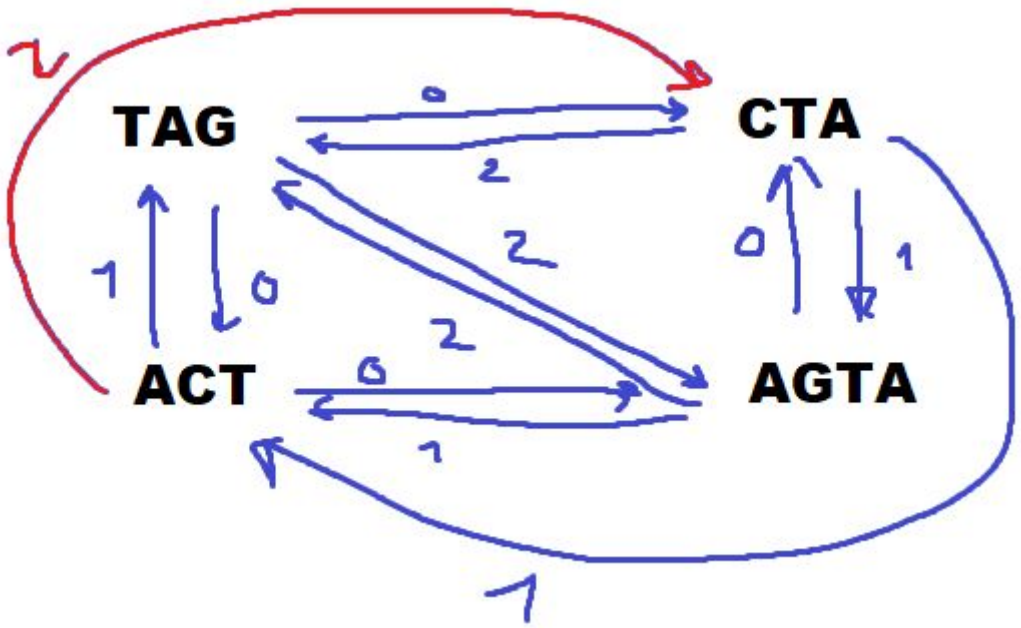
ACTAGTA

Notons cependant que le choix du sommet à chaque étape est aléatoire, et que l'algorithme peut aboutir à un autre résultat en choisissant un autre sommet.

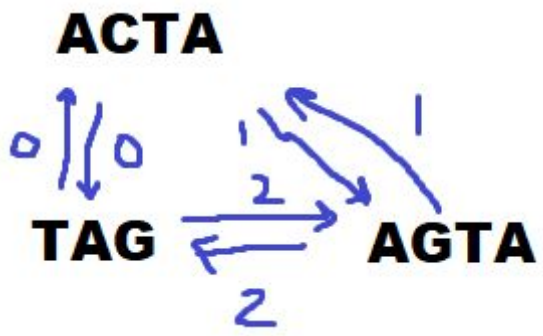
Nous pouvons voir qu'en choisissant un autre sommet à l'étape 1, on aboutit à un résultat non seulement différent, mais qui n'est même pas une SSC :

Etape 1 :

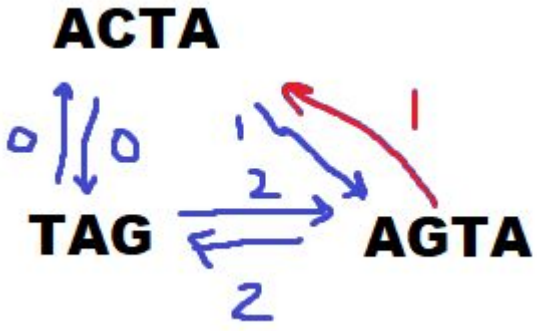




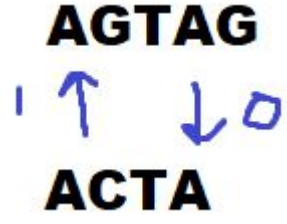
Etape 2 :

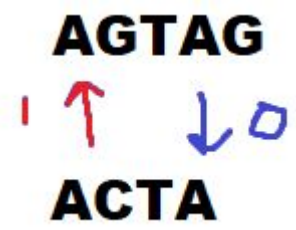


Ici, si l'algorithme a choisi le sommet ACTA, il n'a pas d'autre choix que de choisir le sommet AGTA.



Etape 3 :





Résultat :

ACTAGTAG

On a alors dans cette solution un caractère en plus par rapport à la solution précédente. Notre algorithme ne donne donc pas systématiquement la bonne solution.

Question 3 :

L'efficacité de notre précédent algorithme est liée à celle de l'opération élémentaire qui consiste à calculer la longueur maximale d'un recouvrement entre deux mots *u* et *v*.

Pour déterminer un recouvrement de taille k, il faut comparer les k derniers caractères du premier mot avec les k premiers caractères du second. Un recouvrement est de taille k si ces k caractères sont identiques.

Autrement dit, si les deux sous-chaînes contenant ces k caractères sont identiques, alors il y a recouvrement.

Pour trouver le plus grand recouvrement possible, il faut alors commencer par comparer les chaînes entières, puis retirer un caractère dans chaque, au début de la chaîne pour le premier mot, à la fin pour le second, et ce jusqu'à obtenir un recouvrement, qui sera donc automatiquement le plus grand.

Ainsi, si le mot le plus petit est de taille n, on réalise dans le pire des cas (pas de recouvrement possible entre les deux chaînes) n comparaisons, d'où la linéarité.

On peut voir un certain parallèle avec l'algorithme KMR, qui sert à retrouver les plus longs facteurs répétés entre deux séquences.

Algorithme en pseudo-code :

```
int CalculRecouvrement( mot1, mot2)
        n = min(taille(mot1), taille(mot2));
        i = n;
        Trouvé = Faux;

        Tant que (Trouvé == Faux ET i>0) :
        Faire
                mot1_copy = Les i derniers caractères de mot1;
                mot2_copy = Les i premiers caractères de mot2;
                Si (mot1_copy == mot2_copy) :
                        Trouvé = Vrai;
                Sinon :
                        i--;
        Retourner i;
```

En voici une implémentation en C :

```c
/* ============================================================ */
int Q3(char * u, char * v)
/* ============================================================ */

/*
    Retourne le recouvrement maximal entre
    un premier mot u et un deuxieme mot v.
*/
{
    int n = strlen(u);
    if (n > strlen(v))
        n = strlen(v);
    bool b = false;
    int i = n;


    while( b == false && i>0)
    {
        char * vt = (char*) malloc(i*sizeof(char *));
        strncpy(vt, v, i);
        printf("coupe 1er mot : %s\ncoupe 2nd mot : %s\n", u+(n-i), vt);
        if (strcmp(vt, u+(n-i)) == 0)
        {
            b = true;
            printf("Correspondance trouvee !\n");
        }
        else
        {
            i--;
            if (i==0)
                printf("Pas de correspondance...\n");
        }
        free(vt);
    }
    return i;
}
```

Extrait du terminal lorsqu'on exécute ce code avec les chaînes "CTA" et "TAG" (note : l'affichage de la valeur du recouvrement est dû au code de la question 4).

```
coupe 1er mot : CTA
coupe 2nd mot : TAG
coupe 1er mot : TA
coupe 2nd mot : TA
Correspondance trouvee !
Recouvrement de 2
```

Question 4 :

Il s'agit ici d'implémenter l'algorithme permettant de créer le graphe des recouvrements à partir d'une collection de fragments.

Pour implémenter cet algorithme, j'ai utilisé la librairie de Graphes écrite par Michel Couprie, déjà utilisée dans des cours précédents à l'ESIEE tels que Graphes et Algorithmes.

```c
/* ================================================================ */
graphe * Q4(char ** tabseq, int n, char * filename)
/* ================================================================ */

/*
    Crée un graphe de recouvrement des séquences du tableau
    de séquences tabseq.
*/
{
    graphe * g = InitGraphe(n, n*(n+1)/2); // Initialise le graphe.
    int rec;
    pcell p;
    g->nomsommet = (char **)malloc(n * sizeof(char *)); // Alloue la mémoire pour chaque mot
dans le graphe

    for(int i = 0; i<n; i++)
    {
        g->nomsommet[i] = (char *)malloc((strlen(tabseq[i])+1) * sizeof(char)); // Alloue la
mémoire pour les caractères de chaque mot dans le graphe en fonction de la taille de chaque mot
de tabseq.
        strcpy(g->nomsommet[i], tabseq[i]); // Nomme chaque sommet avec un mot de tabseq.
    }

    for(int i = 0; i<n; i++)
    {
        g->x[i] = rand() % 300;
        g->y[i] = rand() % 300; // Place chaque sommet sur la représentation graphique de
manière aléatoire.
        for(int j = 0; j<n; j++)
        {
            if (j != i)
            {
                rec = Q3(g->nomsommet[i], g->nomsommet[j]); // calcule le recouvrement entre
deux mots, le sommet actuel i et les autres sommets j.
                if(rec != 0)
                {
                    AjouteArcValue(g, i, j, rec); // Si le recouvrement est non nul, on crée un
arc entre les deux sommets avec comme pondération la valeur du recouvrement.
                    p = g->gamma[i];
                    printf("Recouvrement de %d\n", p->v_arc); // Et on affiche la valeur du
recouvrement.
                }
            }
        }
    }
    EPSGraphe( g, filename, 3, 2, 60, 1, 0, 0, 2); // On sauvegarde dans un fichier la
représentation du graphe.
    TermineGraphe(g); // On libère la mémoire utilsée pour le graphe.
    return g;
}
```

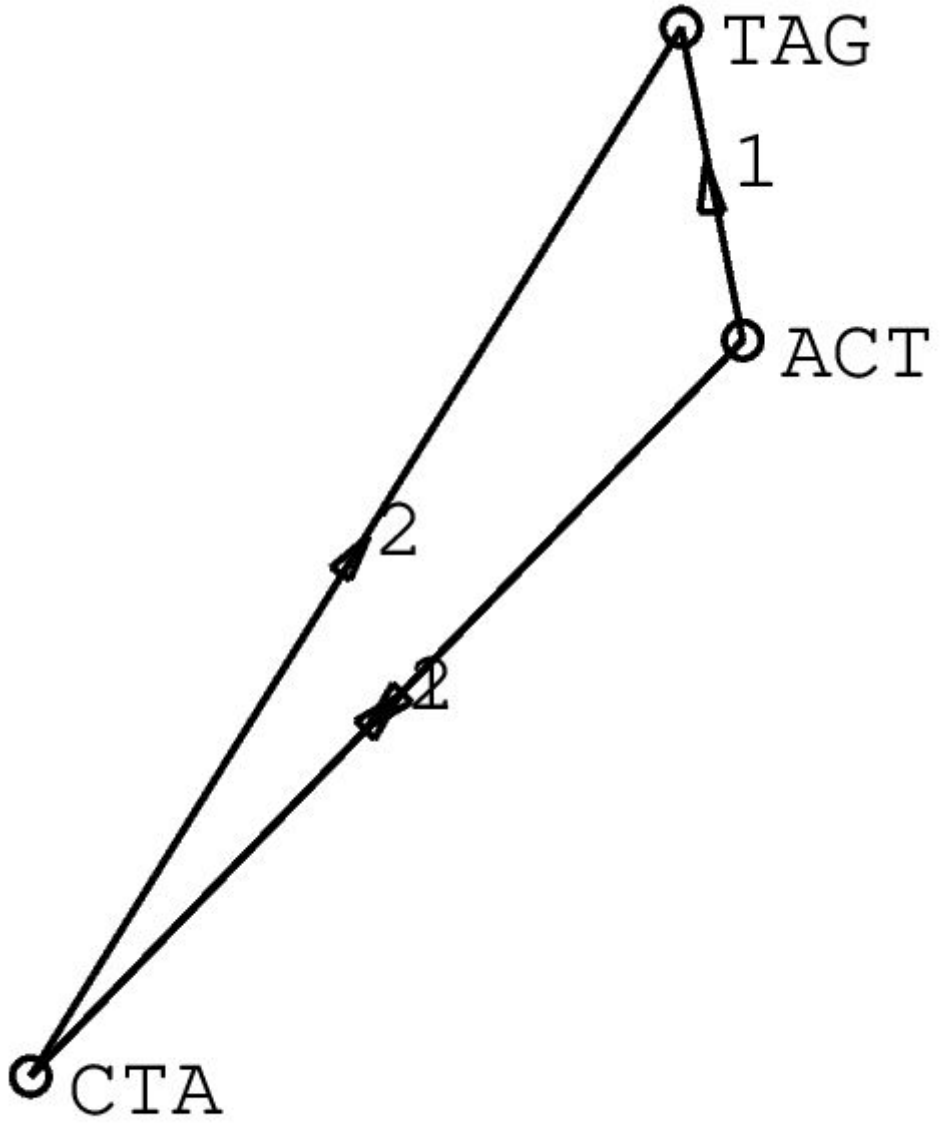
```c
/* ================================================================ */
int main(int argc, char **argv)
/* ================================================================ */
{
  char * text;
  int nbseq;
  char **tabseq;
  char **test = (char **)malloc(4 * sizeof(char *));
  for(int i = 0; i<3; i++)
    test[i] = (char *)malloc(4 * sizeof(char));
  test[0] = "TAG";
  test[1] = "CTA";
  test[2] = "ACT";

  if (argc != 2)
  {
    fprintf(stderr, "usage: %s textfile\n", argv[0]);
    exit(0);
  }
  text = readtextfile(argv[1]);
  tabseq = text2tabseq(text, CARFIN, &nbseq);
  free(text);
  Q4(test, 3, "test.eps");
  // Q4(tabseq, nbseq, "sequences.eps");
  // printtabseq(nbseq, tabseq);
  freetabseq(nbseq, tabseq);
  free(test);
}
```

Pour tester l'algorithme, nous avions à disposition un code source en C avec des fonctions utilitaires pour le traitement de séquences. J'ai directement ajouté mon code dans ce code source.

J'ai dans un premier temps testé avec l'exemple cité plus haut, avec la collection {"TAG", "CTA", "ACT"}.

Voici la représentation obtenue :



Elle correspond tout à fait à ce que nous attendions plus haut. Comme sous-entendu dans les commentaires de l'algorithme, j'ai volontairement omis de représenter les arcs où le recouvrement est nul pour alléger les dessins. Ils doivent cependant bien être présents si on veut pouvoir traiter ce graphe par la suite (ce qui n'était pas demandé dans notre cas) ! Il faut donc ajouter un else dans l'algorithme pour bien ajouter cet arc. On remarquera aussi que dans le cas de deux arcs entre deux mêmes sommets mais de sens différents, les valeurs sont superposées, ce qui rend la lecture parfois compliquée...

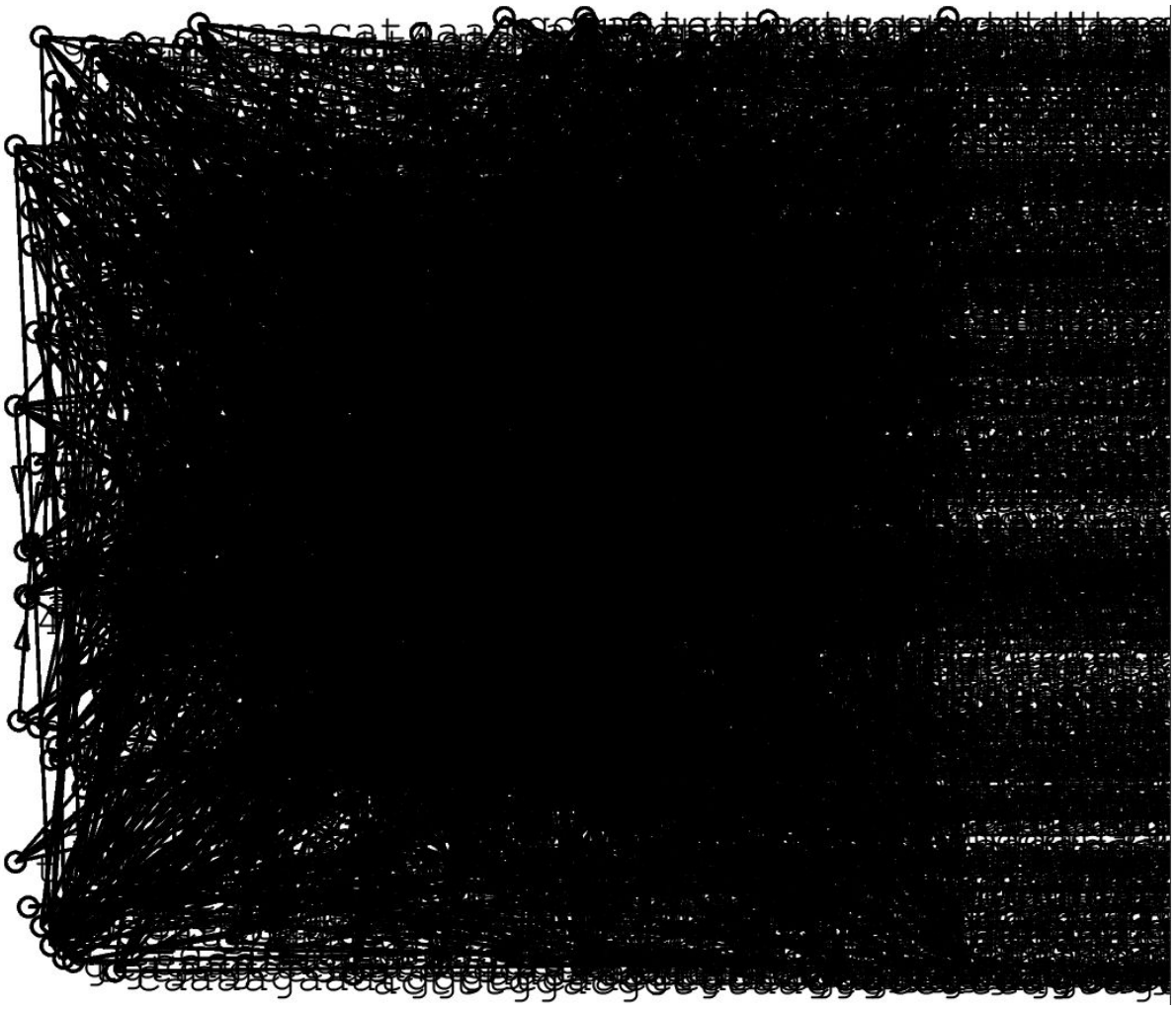
```
if(rec != 0)
{
    AjouteArcValue(g, i, j, rec); // Si le recouvrement est non nul, on crée un
ux sommets avec comme pondération la valeur du recouvrement.
    p = g->gamma[i];
    printf("Recouvrement de %d\n", p->v_arc); // Et on affiche la valeur du

}
else
    AjouteArc(g, i, j);
```

Sur le terminal, on obtient ce résultat :

```
jason@jason-Aspire-A717-72G:/media/jason/Data/ESIEE/E4/Info/INF-4202B/TP/TP1/Alg
orithmique et Génome_fichiers/TP1$ ./TP1 sequence
coupe 1er mot : TAG
coupe 2nd mot : CTA
coupe 1er mot : AG
coupe 2nd mot : CT
coupe 1er mot : G
coupe 2nd mot : C
Pas de correspondance...
coupe 1er mot : TAG
coupe 2nd mot : ACT
coupe 1er mot : AG
coupe 2nd mot : AC
coupe 1er mot : G
coupe 2nd mot : A
Pas de correspondance...
coupe 1er mot : CTA
coupe 2nd mot : TAG
coupe 1er mot : TA
coupe 2nd mot : TA
Correspondance trouvee !
Recouvrement de 2
coupe 1er mot : CTA
coupe 2nd mot : ACT
coupe 1er mot : TA
coupe 2nd mot : AC
coupe 1er mot : A
coupe 2nd mot : A
Correspondance trouvee !
Recouvrement de 1
coupe 1er mot : ACT
coupe 2nd mot : TAG
coupe 1er mot : CT
coupe 2nd mot : TA
coupe 1er mot : T
coupe 2nd mot : T
Correspondance trouvee !
Recouvrement de 1
coupe 1er mot : ACT
coupe 2nd mot : CTA
coupe 1er mot : CT
coupe 2nd mot : CT
Correspondance trouvee !
Recouvrement de 2
```

J'ai ensuite testé l'algorithme sur la séquence fournie dans l'énoncé, mais compte tenu de la taille des différentes séquences et de leur nombre, le graphe est plutôt illisible…



Par ailleurs, le temps d'exécution est également assez long (plusieurs minutes).
Cela est dû au fait qu'il y a plus de 570 séquences à traiter, et que leur longueur est également assez grande (le phénomène d'explosion combinatoire vu en Q1 et Q2 se répercute aussi à la création du graphe !).

Conclusion :

Nous avons pu montrer que le problème de recouvrement entre deux séquences pouvait être résolu avec la théorie de graphes, avec une efficacité relative tout de même. Les algorithmes développés sont également, pour la Q2 améliorable pour être plus précis sur le résultat (il n'obtient pas toujours le bon résultat). Pour celui de la Q4, mon manque d'expérience quand à la manipulation de la représentation graphique de graphiques avec de nombreux sommets rend difficile la vérification d'un résultat pour de grandes collections de séquences, mais j'ai pu montrer qu'il fonctionnait sur de plus petites collections, qui ne nécessitent pas de graphes trop grands.