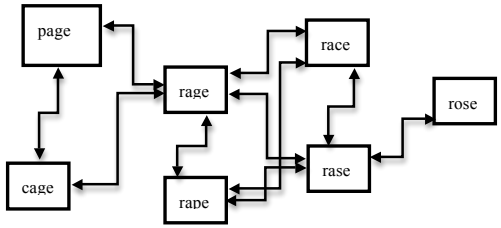
PROJET D'INFORMATIQUE 1A PET : DU COQ A L'ANE

Ce projet sera réalisé en binôme, sur les 4 dernières séances d'informatique ET en dehors des séances d'informatique. Pour la première séance, vous devez avoir lu le sujet et formé les binômes.
Attention : La réussite de ce projet exige du travail en dehors des séances.

Pour aller du coq à l'âne, passer par coq, col, cil, ail, aie, are, ure, une, âne. Ce projet propose de calculer les métagrammes les plus courts, c'est à dire la suite minimale de mots permettant de passer d'un mot initial à un mot final en en changeant qu'une seule lettre à chaque fois.

La solution la plus rapide pour résoudre ce problème se base sur les graphes et le plus court chemin dans un graphe.
Les sommets du graphe à construire contiennent chacun un mot, et un sommet est relié à un autre sommet s'il ne diffère du mot contenu dans ce deuxième sommet que d'une lettre. Les mots ont le même nombre de lettres et on ne tient pas compte de la casse (majuscules/minuscules) et des caractères accentués dans un premier temps.
Le métagramme est alors le plus court chemin partant du sommet contenant le mot de départ et arrivant au sommet contenant le mot d'arrivée.

Par exemple, le mot page est à distance unitaire des mots rage et cage. Le mot rage est à distance unitaire des mots page, cage, race, rase, rape. Et les mots rose et rase sont à distance unitaire. Il n'y a pas d'autres liens, et il est impossible de passer de rape à page directement par exemple.

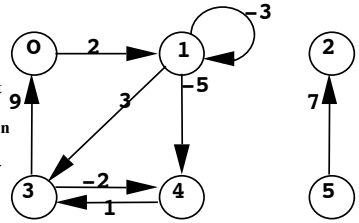


Pour aller de page à rose, il y a de nombreux chemins :
page, cage, rage, rape, rase, rose
page, rage, rape, rase, rose
page, cage, rage, rape, race, rase, rose
page, rage, rase, rose

Le chemin recherché est le dernier, qui est le plus court possible en supposant que chaque lien est de cout unitaire (1).

1. GRAPHES : DEFINITION & TERMINOLOGIE

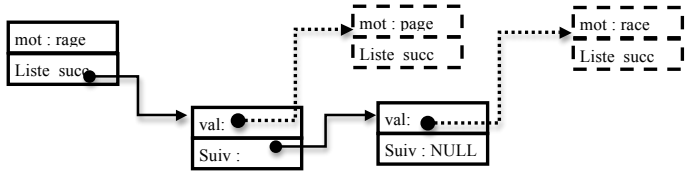
Un Graphe est défini par un couple G[X,A] où X est un ensemble de nœuds ou sommets et A est l'ensemble des paires de sommets reliés entre eux (arêtes du graphe ou « arc »)
Arc = arête orientée
chemin = séquence d'arcs menant d'un sommet i à un sommet j
circuit = chemin dont les sommets de départ et d'arrivée sont identiques
valuation, coût = valeur numérique associée à un arc ou à un sommet
degré d'un sommet = nombre d'arêtes ayant ce sommet pour extrémité
voisins : les voisins des sommets sont ceux qui sont reliés à ce sommet par un arc.



1.1. Représentation des sommets

On peut représenter un sommet par une structure contenant un mot et la liste de ses successeurs possibles, c'est à dire la liste des mots qui ne diffèrent que d'une lettre du mot contenu dans ce sommet.
Cette liste de successeurs sera donc une liste de pointeurs vers les sommets successeurs.

Graphiquement, un sommet contenant le mot rage est donc représenté par (pour simplifier, seuls les sommets successeurs page et race ont été représentés)



Les types de données utiles ressembleront à :
Les sommets : T_SOMMET est une structure

```
typedef struct {char* mot ; L_SUCC Liste_succ;} T_SOMMET ;
```

Les successeurs : un successeur est simplement un pointeur vers un sommet

```
typedef  T_SOMMET* T_SUCC ;
```

Les listes de successeurs : une liste classique avec un suivant, la valeur est un successeur défini précédemment.

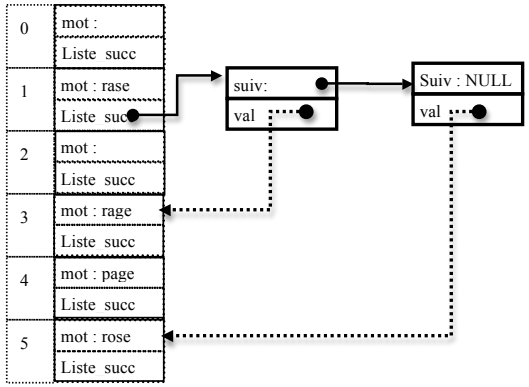
```
typedef struct lsucc { T_SUCC val; struct lsucc* suiv ; }* L_SUCC ;
```

1.2. Représentation du graphe

Le graphe sera représenté par un tableau de sommets ou une table de hachage. La fonction la plus utilisée dans les algorithmes sera la recherche d'un mot donné dans le graphe. De ce fait, la table de hachage est une structure plus adéquate, mais aussi un peu plus difficile à mettre en œuvre. Vous avez le choix entre les 2 possibilités.

1.2.1. Cas du tableau de sommets

Dans le cas d'un tableau, rechercher un mot dans le graphe sera couteux. Il sera de plus préférable de trier le tableau par ordre alphabétique des mots afin de pouvoir utiliser une recherche dichotomique.



Dans cette version, le graphe sera défini par le tableau de sommets et le nombre de sommets. Les types de données utiles ressembleront à :
Le graphe : structure contenant un tableau de sommets alloués dynamiquement et un entier représentant le nombre d'éléments de ce tableau.

```
typedef struct {T_SOMMET* graphe ; int taille;} T_GRAPHE;
```

1.2.2. Cas de la table de hachage

Utiliser une table de hachage, qui sera un tableau de liste de pointeurs de sommets assurera une recherche plus rapide.

Les types de données utiles ressembleront à :
Les listes de sommets ; une liste classique dont les valeurs sont des pointeurs vers les sommets. C'est en fait la même structure que la liste de successeurs.

```
typedef struct lsommet {
      T_SOMMET* valeur ;
      struct lsommet* suiv ; } L_SOMMET
```

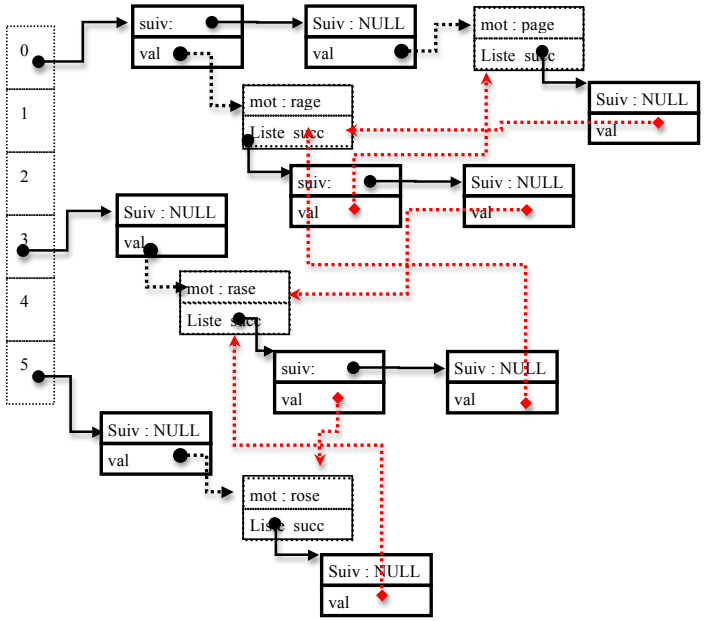
ou plus simplement

```
typedef L_SUCC  L_SOMMET ;
```

Le graphe : structure contenant la table de hachage sous forme d'un tableau de listes de pointeurs de sommets et un entier représentant le nombre de listes de cette table.

```
typedef struct {L_SOMMET* table; int taille;} T_GRAPHE;
```

La figure ci dessous représente ainsi le graphe contenant les mots rase,rage,rose,page.
On suppose que la fonction de hachage vaut 0 pour rage et page, 3 pour rase et 5 pour rose



2. CONSTRUCTION DU GRAPHE

Pour construire le graphe, vous disposez d'un fichier contenant des mots (un mot par ligne). La solution la plus rapide consiste à lire tous les mots pour construire les sommets dans un premier temps, puis à rechercher ensuite les mots à distance unité qui forment les arcs du graphe, c'est à dire les listes de successeurs.
Pour rechercher les mots à distance unité, on peut :

- Soit pour chaque mot "mot1", parcourir tous les autres mots, calculer la distance entre le mot "mot1" et les autres mots, et ajouter l'autre mot à la liste des successeurs de "mot1" si la distance est unitaire. La complexité de cette méthode est en $o(n^2)$ où n est le nombre de mots
- Soit pour chaque mot "mot1", former tous les mots possibles à distance unité (on change une seule lettre de "mot1" en explorant toutes les lettres possibles) et on regarde si le nouveau mot obtenu en changeant une lettre appartient déjà à notre dictionnaire. Pour générer tous les mots possibles à partir de "mot1", il faut modifier une seule lettre de mot1 à la fois et tester toutes les lettres de l'alphabet. En excluant les caractères accentués, il y a 26 lettres possibles ('a' à 'z'). Chaque lettre du mot doit pouvoir etre changée, il y a donc longueur de mot1 x 26 mots possibles. En français, la moyenne des longueurs des mots est 9. La complexité est donc nx26x9xcout de la recherche du mot. Cette complexité est bien inférieure à la précédente, si le cout de la recherche est faible.

Remarque importante pour la lecture de ces fichiers en C:
Les lignes contiennent une chaine de caractères qui peut contenir des espaces. Le plus simple est de lire le mot avec la fonction fgets(). Ensuite, pour assurer la compatibilité entre les fichiers des mondes Windows et Linux (le caractère \n sous windows), on supprime le dernier caractère s'il est inférieur au premier caractère affichable, ie le caractère ' ' dont le code ASCII est 32. :

```
fgets(mot,511,f);
if (mot[strlen(mot)-1]<32) mot[strlen(mot)-1]='\0';
```

3. PLUS COURT CHEMIN DANS UN GRAPHE

Dans un problème de plus court chemin, on considère un graphe orienté **G=(X, A). X** est l'ensemble des sommets, **A** celui des arcs. Chaque arc $\mathbf{a_i}$ est muni d'un coût $\mathbf{p_i}$. Un chemin $\mathbf{C=< a_1, a_2, ..., a_n >}$ possède un coût qui est la somme des coûts des arcs qui constituent le chemin. Le plus court chemin d'un sommet **d** à un sommet **a** est le chemin de coût minimum qui va de **d** à **a**. Il existe plusieurs algorithmes de calcul du plus court chemin :

- L'algorithme de parcours en largeur d'abord permet de trouver le plus court chemin dans un graphe.
- L'algorithme de Dijkstra, dans le cas d'arcs à valuation positive, est celui utilisé dans le routage des réseaux IP/OSPF.
- L'algorithme de Bellman-Ford est le seul à s'appliquer dans le cas d'arc à valeur négative. Attention cependant aux circuits de valeur négative, car il n'existe pas de solutions dans ce cas.
- L'algorithme A* ou Astar qui utilise une heuristique pour trouver rapidement une solution. Il ne s'applique qu'aux graphes dont les arcs sont à valuation positive. De plus, il nécessite qu'il soit possible d'estimer la « distance » entre deux nœuds par une fonction heuristique.

3.1. Dijkstra

L'idée de l'algorithme de Dijkstra est que lorsqu'on a trouvé un plus court chemin entre le sommet de départ **d** et un sommet intermédiaire **k**, il faut mettre à jour les successeurs possibles de **k** que l'on peut atteindre. Lorsque l'on trouve le plus court chemin entre **d** et **k**, c'est aussi le plus court chemin entre le départ et chaque sommet sur le chemin entre **d** et **k**.

Tout au long du calcul, on va donc maintenir deux ensembles :

- **C**, l'ensemble des sommets du graphe qui restent à visiter ; initialement **C=X**
- **S**, l'ensemble des sommets du graphe pour lesquels on connaît déjà leur plus cout chemin au point de départ; initialement, **S={}.**

L'algorithme se termine bien évidemment lorsqu'on atteint le sommet voulu. Pour chaque sommet **s** dans **S**, on conservera le coût actuel du chemin entre **s** et **d** dans la variable **PCC.** On conservera aussi le sommet **pere** qui le précède dans ce plus court chemin. Ainsi, pour retrouver le plus court chemin, il suffira, en partant de **a**, de remonter de prédécesseur en prédécesseur jusqu'à **d**.

3.2. Algorithme : trouver le plus court chemin entre d et a

1. **début**
2. **pour t**ous les sommets i de G[X,A] **faire** $PCC_i \leftarrow +\infty$
3. $PCC_d \leftarrow 0$
4. $S \leftarrow \{\}$
5. $C \leftarrow X$
6. **faire**
7. Sélectionner le sommet j de C de plus petite valeur PCC_j
8. $C \leftarrow C \setminus j$ // supprimer j de l'ensemble C
9. $S \leftarrow S \cup j$ // ajouter j à l'ensemble S
10. **pour** tous les sommets k adjacents à j **faire** // les successeurs de j
11. **si** $PCC_k > PCC_j + c(j,k))$ // c(j,k) est le coût pour aller de j à k
12. **alors**
13. $PCC_k \leftarrow PCC_j + c(j,k)$; // Passer par j est plus court
14. pere(k)=j ; // pour aller de a en k
15. **fin si**
16. **fin pour**
17. **tant que** $a \notin S$ et PCCj !=+∞
18. **fin**

4. FORMAT DES FICHIERS DE DONNEES

Pour tester votre algorithme, vous disposez de plusieurs fichiers de mots.

- dico1.txt contient quelques mots permettant de tester facilement la lecture et la construction du graphe (page, cage, rage, rape, race, rase, rose)
- mot4.txt contient les mots de 4 lettres du français, sans caractères accentués
- mot5.txt contient les mots de 5 lettres du français, sans caractères accentués
- mots.txt contient tous les mots du français, sans caractères accentués

5. TRAVAIL DEMANDE

Vous avez deux possibilités pour représenter le graphe : un tableau de sommets ou une table de hachage de sommets. Il n'est pas demandé d'implanter les 2 : choisissez l'une ou l'autre.
De même pour le calcul de la liste des successeurs : choisissez une des 2 méthodes proposées.

5.1. Première séance : analyse du problème.

Cette séance permet de répondre à vos questions sur les algorithmes et les structures de données. A la fin de cette séance, vous devez avoir une vision claire des grandes étapes de votre programme et vous ferez un document décrivant :

- les types de données utilisées, en explicitant le rôle de chaque élément des structures
- les modules : rôle de chaque module (couple de fichiers .c/.h)
- les prototypes de fonctions, en explicitant :
  - le rôle exact de la fonction
  - le rôle de chaque paramètre et son mode de passage (par valeur ou par adresse)
  - l'algorithme ou les grandes étapes permettant de réaliser la fonction. Précisez uniquement les points qui peuvent être délicats à comprendre et/ou programmer.
- les tests prévus
  - tests unitaires : tests des fonctions précédentes individuellement. Par exemple, il faut tester la fonction de lecture du graphe avant même d'essayer de calculer un chemin.
  - tests d'intégration : quels sont les tests que vous allez faire pour prouver que l'application fonctionne, sur quels exemples.
- la répartition du travail entre les 2 membres du binôme : quelles sont les fonctions qui seront réalisées par chaque membre du binôme et pour quelle séance.
- Le planning de réalisation du projet jusqu'à la date du rendu.

5.2. Séances suivantes

Faire une application qui demande les mots de départ et d'arrivée et qui calcule et affiche le plus court chemin pour le métagramme s'il est existe.

5.3. Rendu final :

Vous ferez un rapport court (5 à 10 pages) explicitant les points suivants :

1. Implantation
   1. État du logiciel : ce qui fonctionne, ce qui ne fonctionne pas
   2. Tests effectués
   3. Exemple d'exécution
   4. Les optimisations et les extensions réalisées
2. Suivi
   1. Problèmes rencontrés
   2. Planning effectif
   3. Qu'avons nous appris et que faudrait il de plus?
   4. Suggestion d'améliorations du projet
3. Conclusion

Prévoyez un développement incrémental en testant toutes vos fonctions au fur et à mesure.

5.4. Quelques éléments pris en compte dans la notation

Rapport : 3 points
Lecture et construction du graphe : 2+3 points
Calcul du plus court chemin: 4 points
Affichage du chemin sommet par sommet : 3 points
Tests : 4 points
Code commenté, qualité du code : 1 point
Extensions : optimisation, etc : bonus
Plagiat interne ou externe à phelma : 0/20

5.5. Quelques exemples

Avec le fichier mot5.txt
le chemin menant de homme a poule vaut 4 et suit la transformation : homme // pomme // poeme // poele // poule
le chemin menant de bonne a femme vaut 5 et suit la transformation : bonne // borne // berne // berme // ferme // femme
le chemin menant de calme a essai est impossible

6. EXTENSIONS

6.1. Caractères accentués

Les caractères accentués (bien utiles en français) doivent être gérés à l'aide des caractères larges du type **`wchar_t`** et des fonctions **`wcslen`** pour la longueur, **`wcscmp`** pour la comparaison, etc…
Dans ce cas, les codes des caractères à tester pour construire les mots (version 2 de la construction du graphe) sont les codes 97 (code du **`'a'`**) à 123 (code du **`'z'`**) et les codes 224 à 254 (codes des caractères accentués). On peut y ajouter les codes 32 (espace) et 45 (trait d'union).
Typiquement, un programme manipulant les caractères larges ressemblera à ceci :

```
main() {
      // Déclarer une chaine de caractères larges
 wchar_t chaine[512] ;
      // indispensable
 setlocale(LC_ALL,"");
      // Lecture d'une chaine large. Attention au %ls
 scanf("%ls ",chaine) ;
      // Les fonctions sur les chaines changent de nom et
      // commencent par wcsxxx
      // Les chaines constantes sont notées L"abc"
 if (wcscmp(chaine,L"fin")==0) exit(0) ;
 else { printf("La chaine lue est %ls\n ",chaine) ;
        printf("Sa longueur est %d\n",wcslen(chaine)) ;
 }
}
```

6.2. Optimisation

6.2.1. Représentation du graphe par tableau de sommets

Dans ce cas, la recherche d'un mot ou sommet peut etre longue si vous la faites de manière séquentielle. Comme le fichier n'est pas obligatoirement trié par ordre alphabétique, il faut d'abord lire tous les mots ou sommets, puis trier notre tableau de sommets. On peut alors utiliser la recherche par dichotomie pour accélérer la recherche : c'est indispensable si vous utilisez les fichiers comportant quelques centaines de milliers de mots. Dans ce cas, les fonctions suivantes de la bibliothèque C seront utiles :

- **`void qsort(void *base, size_t nmemb, size_t size,int(*compar)(const void *, const void *));`**
  La fonction qsort() trie une table contenant nmemb elements de taille size. L'argument base pointe sur le debut de la table.
- **`void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`**
  La fonction bsearch() recherche l'objet correspondant a key, dans une table de nmemb objets, commençant à l'adresse base. La taille des éléments de la table est indiquée dans size. La fonction bsearch() renvoie un pointeur sur l'élément du tableau correspondant à celui recherché, ou NULL si aucun élément ne correspond.

6.2.2. Recherche du minimum dans l'ensemble C (algo de dijsktra)

La recherche du minimum dans l'ensemble C (algorithme de Dijkstra) peut se révéler coûteuse, car elle est en o(n) dans une approche classique. Si on implante l'ensemble C sous forme d'un tas, le maintien du tas est en o(log(n)) et le minimum est au sommet de ce tas : sa recherche est donc en o(log(n)) car la suppression du minimum oblige ensuite à maintenir la structure du tas (voir le TD sur les tas). La principale difficulté ici est que le PCC des sommets déjà dans le tas peut être modifié par la suite (ligne 13 de l'algorithme). Dans ce cas, il faut aussi mettre à jour le tas, car le sommet modifié peut éventuellement ne plus respecter la condition sur les tas : le père est plus petit que les 2 fils. Il faut donc éventuellement faire remonter le sommet dont le PCC est modifié. Pour retrouver rapidement le sommet dans le tas, vous pouvez définir un champ supplémentaire dans la structure de sommet, qui contiendra l'indice du tableau tas où se trouve le sommet ou -1 si le sommet n'est pas dans le tas (ensemble C).