PROJET : LE TOUR DU MONDE LE MOINS CHER

Ce projet sera réalisé en binôme, sur les 4 dernières séances d'informatique ET en dehors des séances d'informatique.
La présence pendant les séances encadrées est obligatoire, mais la réussite de ce projet exige du travail en dehors des séances.

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.

Ce document est essentiel et doit permettre ensuite de coder rapidement votre application en vous répartissant les tâches.

1.1 Le problème du voyageur de commerce

L'objectif du projet est faire le tour du monde le moins cher. C'est le « problème du voyageur de commerce » **TSP** (*Traveling-Salesman Problem*) , qui doit visiter N villes en passant par chaque ville exactement une fois et une seule. Il commence par une ville quelconque et termine en retournant à la ville de départ. On connaît le coût (temps, distance, ou autre chose...) des trajets possibles entre chacune de ces N villes. Quel circuit faut-il choisir afin de minimiser le coût du parcours ?
Si l'on considère que toutes les villes sont reliées entre elles, on peut représenter l'ensemble des connexions possibles par un *graphe non orienté, complet, pondéré* (cf. Figure 1). Un graphe est un ensemble de sommets reliés par des arêtes. Un graphe est non orienté s'il n'y a pas de sens pour aller d'un sommet à un autre (p. ex.: on peut prendre l'avion pour aller de Paris à Rome et vice-versa). Un graphe est pondéré si le fait d'emprunter une arête a un coût.
Formellement, le problème du voyageur devient : partant d'un *sommet* donné, il s'agit de

trouver le *circuit hamiltonien* de *moindre coût* passant par tous les *sommets* d'un *graphe non orienté, complet, pondéré.* Dans un graphe, un cycle hamiltonien est un chemin d'arêtes passant par tous les sommets une fois et une seule et revenant au nœud de départ (cf. Figure 1).
Le voyageur de commerce est un problème NP-complet : la solution exacte est de complexité o(N!). Il n'existe pas d'algorithme de complexité polynomiale connu. Il est donc impossible de trouver l'optimum en un temps raisonnable lorsque le nombre de « villes » est supérieur à quelques centaine de villes! Au delà, il faut utiliser des algorithmes sous-optimaux utilisant des heuristiques, qui aboutissent à une solution « pas trop mauvaise » en un temps acceptable.

Figure 1 : le voyageur de commerce et le graphe associé

2 Résolution du TSP par un algorithme à colonie de fourmis

Parmi les méthodes de résolution du TSP, ce projet s'intéresse aux algorithmes de colonies de fourmis. Ceux ci sont inspirés du comportement de fourmis réelles dont l'action conjointe amène à trouver un chemin optimal entre un point N et un point F. Les fourmis communiquent entre elles par le dépôt de phéromones sur leur chemin, encourageant les autres fourmis à les suivre. Plus un chemin sera couvert de phéromones plus les fourmis l'emprunteront et déposeront de phéromones. La figure 2 illustre le phénomène. Au départ, les fourmis empruntent aléatoirement les chemins pour aller de N à F, déposant des phéromones sur leur passage. À la prochaine itération, les fourmis seront influencées par les phéromones déposées précédemment et emprunteront plus probablement les chemins comportant le plus de phéromones. Au bout, d'un certain temps, le chemin le plus court aura été parcouru un plus grand nombre de fois et aura donc plus de phéromones que les autres entraînant la convergence de toutes les fourmis vers celui-ci.

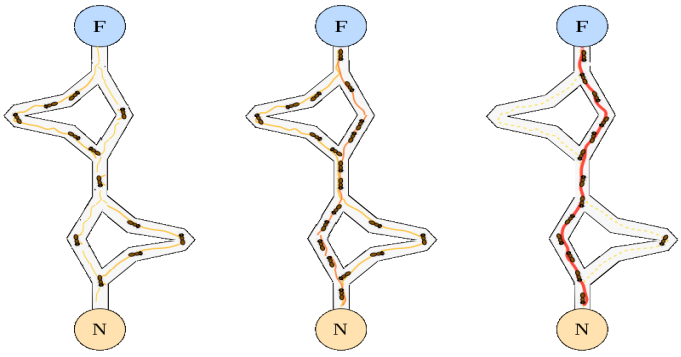


Figure 2 : évolution des chemins parcourus par les fourmis

Les algorithmes à colonie de fourmis ou ACO (*Ant Colony Optimization*) sont une famille de méta-heuristiques qui utilise plusieurs « agents » (fourmis) parcourant l'espace de recherche collectivement et partageant de l'information à travers des phéromones pour trouver une « bonne » solution.

L'algorithme réalise plusieurs itérations d'un **cycle**. Dans un cycle, toutes les fourmis parcourent un chemin hamiltonien : une fourmi passe une fois et une seule dans chaque ville. On détermine ensuite quel est le meilleur chemin pour le conserver. Les fourmis déposent leurs phéromones sur le chemin qu'elles parcourent. Ces phéromones ont une durée d'action limitée dans le temps afin de diminuer l'influence de chemins non intéressants : c'est l'**évaporation**.

2.1 Particularité des fourmis de l'ACO

Contrairement aux fourmis réelles, les fourmis virtuelles ont une **mémoire** qui leur permet de stocker le chemin parcouru à chaque cycle. Cette mémoire est « vide » au début de chaque cycle. Les fourmis virtuelles déposent une quantité d'information (phéromones) qui est proportionnelle à la **qualité** de la solution trouvée. Elles ne mettent à jour les phéromones qu'une fois leur solution (ou chemin) complètement construite et évaluée, à la fin d'un cycle. L'**évaporation** est réalisée à chaque fin de cycle, juste avant le dépôt des phéromones par les fourmis.
Lorsqu'elles sont sur une ville, les fourmis doivent décider sur quelle ville adjacente se déplacer. Alors que les fourmis réelles semblent utiliser le hasard et les phéromones, les fourmis virtuelles prennent en compte les phéromones, les villes déjà visitées et la « visibilité » des villes disponibles. La visibilité est définie comme l'inverse de la distance entre 2 villes. Au cours d'un cycle, une fourmi virtuelle ne repasse jamais par une ville déjà visitée.

3 Algorithme à colonie de fourmis

On considère un graphe **G=(X, A).** Chaque arête $\mathbf{a_i}$ est munie d'un poids $\mathbf{d_i}$. On cherche l'ensemble des arêtes Chemin**=< $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$ >** formant un circuit minimum. L'algorithme suggéré pour la résolution du TSP est le suivant :

3.1 Définition des variables utilisées

m	nombre total de fourmis de l'algorithme
n = |X|	nombre de villes dans le graphe G
$tabu_k$	Liste des villes déjà parcourues par la fourmi k
d_{ij}	Distance entre les villes i et j
τ_{ij}	Quantité de phéromones sur l'arc a_{ij}
$\eta_{ij} = 1/d_{ij}$	Visibilité de la ville j quand une fourmi se trouve dans la ville i
$\rho \in [0,1]$	Coefficient d'évaporation des phéromones
$\alpha \in R^+$	Coefficient régulant l'importance des phéromones pour le choix d'une ville
$\beta \in R^+$	Coefficient régulant l'importance de la visibilité pour le choix d'une ville
$\varepsilon > 0$	Valeur initiale non nulle de phéromones sur les arcs
$Q > 0$	Constante servant à calculer la quantité de phéromones à déposer pour chaque fourmi
MAX_CYCLE	Constante, nombre maximum de cycles autorisés.
$L_k = \Sigma_{aij \in solutionk} d_{ij}$	Longueur d'un chemin, somme des longueurs de chaque arc constituant le chemin

3.2 Algorithme

Meilleur_Chemin ← ∅
Cout_meilleur_chemin ← +∞
Initialiser tous les $\tau_{ij}(0) = \varepsilon$ (avec $\varepsilon > 0$)
Pour un nombre iter : 1 à MAX_CYCLE
Initialiser *m* fourmis sur les *n* villes.
Pour chaque fourmi *k* dans 1..*m*, initialiser la liste $tabu_k$ avec la ville de départ de la fourmi *k*
Pour chaque fourmi *k* dans *1..m*
tant que la fourmi k peut continuer son parcours
Pour la fourmi *k* dans la ville *i*, choisir la prochaine ville *j* la plus probable avec $j \notin tabu_k$.
Ajouter la ville *j* à $tabu_k$.
Mettre à jour la solution courante et la longueur à l'aide de l'arc $\mathbf{a_{ij}}$.
Fin pour
Si Meilleur_Chemin moins bon que la solution de la fourmi *k*, **alors** mettre à jour le meilleur chemin
Fin pour
Pour chaque arête $\mathbf{a_i}$ dans **G**, faire évaporer les phéromones
Pour chaque fourmi *k*, déposer les phéromones sur les arcs de son chemin
Fin Pour
Retourner Meilleur_Chemin

3.3 Choix de la prochaine ville

A chaque itération, une fourmi *k* étant à la ville *i* va choisir la ville de destination *j* en fonction de la visibilité η_{ij} de cette ville et de la quantité de phéromones τ_{ij} déjà déposée sur l'arc $\mathbf{a_{ij}}$. La probabilité d'une ville j d'être choisie par une fourmi k située à la ville i est donnée par :

$$p^k_{ij} = \tau_{ij}^{\alpha} . \eta_{ij}^{\beta} / \Sigma_{l \notin tabu_k} [\tau_{il}^{\alpha} . \eta_{il}^{\beta}] \text{ pour } j \in X \backslash tabu_k$$
$$0 \text{ si } j \in tabu_k$$

où X est l'ensemble des villes du graphe G, $tabu_k$ la liste des villes déjà visitées par la fourmi k, α et β deux coefficients contrôlant l'importance des phéromones et de la visibilité dans le choix d'une ville. Ainsi, si β = 0 seules les phéromones influenceront le choix des prochaines villes.

Cette formule permet de définir la distribution de probabilité de p^k_i (avec $\Sigma_j\ p^k_{ij} = 1$). Et c'est en utilisant cette distribution que la ville j est tirée au hasard.

3.4 Mise à jour des phéromones sur le graphe

A la fin de chaque cycle, les fourmis ont parcouru l'ensemble des n villes et les phéromones sont mises à jour sur chaque arc du graphe par la formule :

$$\tau_{ij} = \rho . \tau_{ij} + \Delta\tau_{ij}$$

où $\rho \in [0,1]$ est un coefficient d'évaporation des phéromones entre chaque cycle et où $\Delta\tau_{il}$ représente la quantité de phéromones déposée par les fourmis du cycle. La quantité de phéromones pour une fourmi k est donnée par :

$$\Delta\tau^k_{ij} = Q/L_k \text{ si } a_{ij} \in \text{solution}_k, 0 \text{ sinon}$$

Où *Q* est une constante et L_k est la somme des poids d_{ij} des arcs a_{ij} de la solution de la fourmi *k*.
On peut donc constater que, plus la solution d'une fourmi est longue, moins elle déposera de phéromones sur les arcs de sa solution. Notez que les phéromones sont déposés sur l'arc i.e. dans le sens du chemin pris par la fourmi, et non dans les deux sens.
La quantité de phéromones à déposer sur chaque arc a_{ij} est obtenue par :

$$\Delta\tau_{ij} = \Sigma_{k=1 \text{ à } m} \Delta\tau^k_{ij}$$

4 Mise en œuvre

4.1 Structures de données utilisées

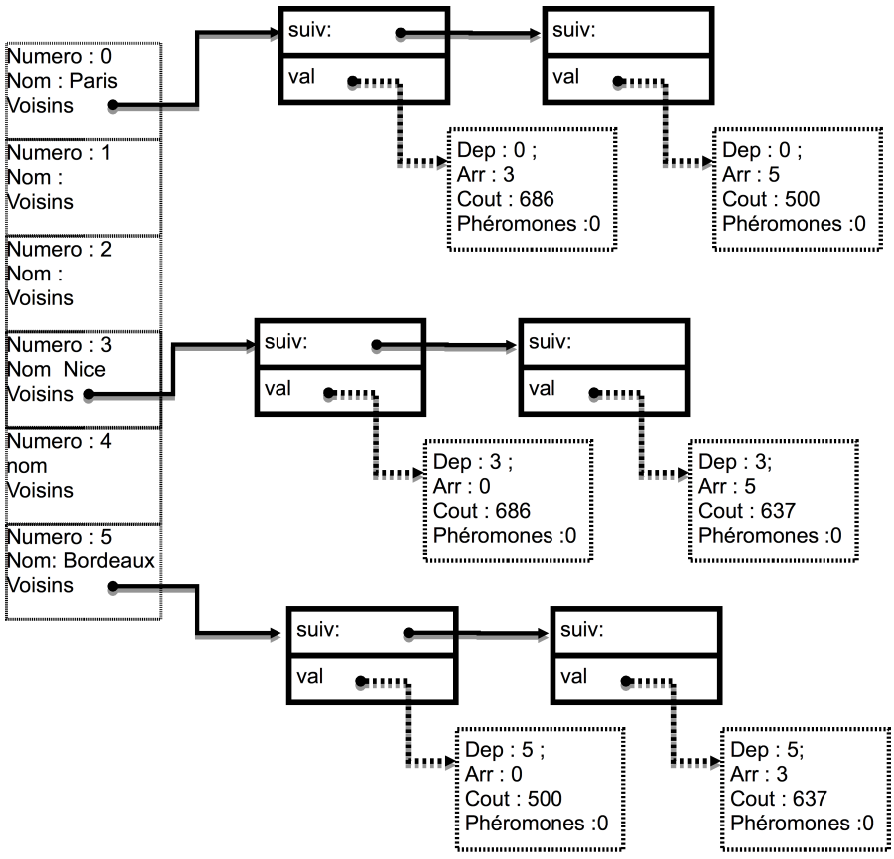
4.1.1 Graphe

Il existe plusieurs modes de représentation d'un graphe afin de gérer les liens entre sommets. Nous utiliserons une liste d'adjacence des voisins d'un sommet : c'est la liste des arcs sortant de ce sommet. Cette liste est stockée dans le champ *voisins* d'un sommet. Un arc est un quadruplet « sommet de départ », « sommet d'arrivée », « distance entre villes » et « quantité de phéromones ». Pour représenter une arête, on utilisera deux arcs orientés, un dans chaque sens.

Le graphe sera donc défini par un tableau de sommets. Chaque sommet contient ses propres caractéristiques ainsi que des informations utiles à l'algorithme :

- **numéro :** le numéro du sommet
- **nom :** le nom du sommet
- **x, y :** positions du sommet (pour la représentation graphique du graphe)
- **voisins :** la liste d'adjacence, liste de pointeurs vers les arcs sortant de ce sommet

Exemple avec 3 villes dont les distances ou couts sont de 686, 500 et 637 km



4.1.2 Fourmis

Une fourmi est représentée par une structure qui contiendra:

- **solution :** ensemble des arcs formant la solution de la fourmi. Pour stocker l'ensemble, on choisira d'utiliser une *file* de pointeurs d'arcs. Ce choix permettra de représenter non seulement la solution de la fourmi mais aussi la liste *tabu* des villes déjà visitées. Remarquez que cette file partage les arcs avec le graphe précédent à l'aide de pointeurs.
- **ville_départ et ville_courante:** indice des sommets de départ et courant de la fourmi.

4.2 Tirage de la prochaine ville

A chaque itération, les fourmis doivent choisir aléatoirement la ville suivante. Comme vu précédemment, les probabilités de choix varient en fonction de la visibilité d'une ville et de la quantité de phéromones sur chaque arc. Il n'est donc pas possible d'utiliser un générateur de nombre aléatoire de loi uniforme de type **`rand()%N`** (avec N = nombre de villes).

Pour rappel, dans le cas d'une variable aléatoire discrète X, on peut représenter la loi de probabilité par un diagramme en bâton. La figure 3 représente la loi de probabilité pour les villes : Nice, Grenoble et Lille. Comment assurer un tirage de ville qui respecte cette loi ?
Une des façons d'y parvenir est de travailler sur la fonction de répartition F(X) de la loi de probabilité. La figure 3 représente la fonction de répartition de l'exemple des trois villes. Dans cette fonction, $F(X) \leq 0.3$ correspond à l'espace de la ville de Nice, $0.3<F(X) \leq 0.8$ à celui de Grenoble et $0.8<F(X) \leq 1$ à celui de Lille. Le problème peut donc se résoudre en tirant aléatoirement un nombre entre 0 et 1 et en recherchant l'intervalle (et donc la ville) auquel ce nombre appartient. Dans l'exemple cité, si on tire 0.87, ce nombre appartient à l'intervalle correspondant à la ville de Lille. Si on tire 0.21, ce nombre appartient à l'intervalle correspondant à la ville de Nice. Tant que le tirage aléatoire suit une loi uniforme (ici donné par la fonction rand()), il y a 50 % de chance de choisir Grenoble, 30 % de chance de choisir Nice et 20 % de choisir Lille.

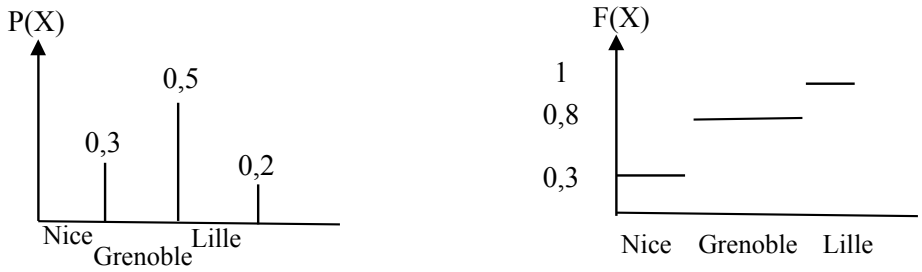


Figure 3 : les densités de probabilité et fonction de répartition

L'algorithme de calcul proposé pour la fourmi k dans la ville i devient :
tirage ← rand() /max_rand // tirage aléatoire d'un nombre réel entre 0 et 1
cumul_proba ← 0 //valeur permettant la normalisation des probabilités
fx ← 0 // valeur courante de la fonction de répartition
Pour chaque ville j $\notin tabu_k$
    cumul_proba ← cumul_proba + $\tau_{ij}^{\alpha}.H_{ij}^{\beta}$
Fin pour
Pour chaque ville j $\notin tabu_k$
    fx ← fx + $\tau_{ij}^{\alpha}.H_{ij}^{\beta}$ / cumul_proba
    si tirage ≤ fx alors retourner un pointeur sur l'arc entre les villes i et j
Fin pour
Retourne NULL // pas de choix possible

5 Choix des paramètres

L'une des difficultés de mise en œuvre de l'algorithme est le choix des paramètres ρ, α, β, ε, Q. Il n'y a pas de règles théoriques sur ce choix. Vous devrez donc les déterminer expérimentalement. Pour démarrer, nous vous conseillons d'utiliser les valeurs suivantes : ρ = 0.5, α = 1, β=2, $\varepsilon=10^{-5}$, Q=1. Par ailleurs, il a été constaté que les solutions sont meilleures quand $m \approx 2*n$ et quand les fourmis sont uniformément réparties sur l'ensemble des villes (plutôt que de partir de la même).

6 Extensions possibles

Cette version de base de l'algorithme à colonie de fourmis peut être étendue de nombreuses manières. Une extension simple de l'algorithme est la prise en compte des meilleures fourmis pour le dépôt des phéromones. Il s'agit d'une version élitiste dans laquelle les phéromones des x meilleures fourmis sont déposées plusieurs fois.
Une autre extension du programme est l'affichage graphique des villes et du chemin optimal. Les fichiers de description de graphe contiennent les coordonnées des villes, il devient donc possible de les afficher sur une fenêtre en respectant leur situation géographique relative.
Vous pouvez aussi envisager le cas où le graphe n'est pas complet (il manque par exemple une liaison entre 2 villes). Il est possible d'obtenir une solution en modifiant très légèrement l'algorithme décrit.
Le critère de convergence de l'algorithme peut aussi être étudié et amélioré pour éviter les itérations inutiles.

7 Format des fichiers de données

Plusieurs graphes vous sont fournis pour tester votre programme. Le format de ces fichiers est le suivant :

Première ligne :
deux entiers ; le nombre de sommets et le nombre d'arêtes du fichier (la moitié du nombre d'arcs)
Deuxième ligne :
une chaîne de caractères qui est : « Sommets du graphe »
X lignes :
un entier, deux réels, et une chaîne de caractères : numéro du sommet, coordonnées en x du sommet, coordonnées en y du sommet, nom du sommet
1 ligne :
une chaîne de caractères qui est : « Arêtes du graphe : noeud1 noeud2 valeur »
Y lignes :
un entier, un entier, un **réel** : sommet de départ, sommet d'arrivée, distance entre les villes

Remarque importante pour la lecture en C:

La lecture des lignes contenant les sommets du graphe (les X lignes) doit se faire en lisant d'abord un entier puis une chaîne de caractères qui peut contenir des espaces. Pour cela, on utilise :

```
char mot[512] ;
fscanf(f,"%d %lf %lf %[^\n] ",&numero,&x,&y,mot);
```

numéro contiendra alors l'entier, x et y les coordonnées et mot le nom du sommet.

7.1 Fichiers disponibles

Tous les fichiers fournis représentent des graphes dont seule la moitié des arcs sont donnés : l'arc entre les sommets i et j avec i<j est défini dans le fichier, mais pas celui entre les sommets j et i. Ces deux arcs ont en effet la même distance. Il faudra donc allouer deux arcs en mémoire pour chaque arc trouvé dans le fichier : lorsque l'on trouve un arc entre les sommets i et j dans le fichier, il faut bien sur ajouter cet arc à la liste des successeurs du sommet i. Mais il faut aussi créer un autre arc entre les sommets j et i qui sera ajouté à la liste des successeurs du sommet j.

Les fichiers graphe11.txt, graphe12.txt, graphe13.txt, graphe14.txt sont des graphes de complexité croissante (6, 10, 50, 100 sommets) générés aléatoirement. Les fichiers djibouti38.txt, berlin52.txt, KroA100.txt et Quatar194.txt correspondent à des villes ou pays réels.

8 Travail demandé

8.1 Réalisation

Faire une application qui prend un fichier de graphe en entrée et qui calcule et affiche le plus court chemin et son coût pour le problème du voyageur de commerce

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

8.2 Conseils de développement

- Commencez par définir les structures de données dont vous aurez besoin pour représenter les sommets, les arcs, les listes, files. Écrivez et testez les fonctions de base sur ces types abstraits avant de passer à la suite.
- Prévoyez un développement incrémental en testant toutes vos fonctions au fur et à mesure.
- Commencez par developper la fonction de lecture du fichier pour charger le graphe dans la structure de sommets après avoir développé les fonctions de gestion des listes de pointeurs d'arc. Vérifier ensuite le graphe chargé en développant une fonction d'affichage.
- Validez le programme réalisé sur les graphes simples avant de le tester sur des graphes plus conséquents.
- Attention : vous devez vous assurer que la compilation et vos programmes fonctionnent sur les machines de l'école.

8.3 Livrables

Chaque membre du binome copiera l'ensemble des fichiers constituant le livrable dans les répertoires : /users/phelma/phelma2017/**mon-login/**tdinfo/seance15.
Le livrable sera constitué :

- du rapport du projet, format PDF. *N'oubliez pas de faire figurer vos noms...*
- des sources de votre programme
- du Makefile
- d'un fichier README expliquant comment compiler et lancer votre (vos) programme(s)
- de vos fichiers de test

Le rapport final fera **au plus** 20 pages et ne doit pas inclure le code. Voici un plan indicatif:

1- Intro
2- Spécifications
    2.1 Données : description des structures de données
    2.3 Fonctions : prototype et rôle des fonctions essentielles
    2.4 Tests : quels sont les tests prévus
    2.5 Répartition du travail et planning prévu : qui fait quoi et quand ?
3- Implantation
    3.1 État du logiciel : ce qui fonctionne, ce qui ne fonctionne pas
    3.2 Tests effectués
    3.3 Exemple d'exécution
    3.4 Les optimisations et les extensions réalisées
4- Suivi
    4.1 Problèmes rencontrés
    4.2 Planning effectif
    4.3 Qu'avons nous appris et que faudrait il de plus?
    4.4 Suggestion d'améliorations du projet
5- Conclusion