Exemple de sujet pour l'épreuve orale d'informatique du CCMP

Assistants électroniques de navigation

Avant propos

Un sujet 0 se doit d'atteindre deux objectifs difficilement conciliables : présenter un sujet typique dans sa longueur et dans la nature des questions posées mais aussi passer en revue certaines des difficultés techniques auxquelles les candidats pourraient être exposés dans les sujets.

Dans ce sujet, il a été fait le choix de multiplier les difficultés techniques et les questions "pénibles" qui pourraient dérouter les candidats pendant les oraux, à savoir : lire ou écrire des fichiers d'assez grande taille, utiliser `scanf` et `printf` en C, manipuler des chaînes de caractères en C et des tableaux alloués dynamiquement, utiliser un module OCaml, compiler avec plusieurs fichiers et en utilisant des bibliothèques, devoir aller chercher dans la documentation le fonctionnement de fonctions (comme `sin`), etc. Le sujet essaie aussi de multiplier les différents types de questions (voir la typologie fournie dans les attendus) et d'utiliser des données "réelles" souvent plus compliquées à manipuler et relativement larges (ce qui ne sera pas forcément le cas pendant les oraux). À cause de toute cela, ce sujet ne prétend pas être "typique" de ce que verront les candidats aux oraux.

Aussi, le sujet est en format PDF et donc donne toutes les questions de façon linéaire tandis que l'épreuve est prévue pour être un véritable oral. Durant l'épreuve, les membres du jury passent très régulièrement voir les candidats et les questions seraient plutôt données une à une et certaines pourraient varier en fonction des réponses aux questions précédentes. Nombre des questions présentées ici à l'écrit seraient complétées d'instructions plus précises pour savoir où sont les fichiers à manipuler, quelles commandes il faut exécuter, etc.

Pour s'entraîner sur ce sujet en conditions réelles, il est conseillé de résoudre complètement une question avant de passer à la suivante et d'avoir accès à une personne capable de valider les questions durant l'oral ou de débloquer un problème technique. Le sujet est volontairement long et chaque exercice finit sur des questions difficiles. Pour s'entraîner en 3h30, il est recommandé de passer au plus 1h40 sur le premier exercice et 20 min sur le second.

1 Exercice 1 : programmation en C avec lecture de fichier, implémentation de listes et manipulation de données

Système de coordonnées. Dans ce sujet, on fera l'approximation que la terre est une sphère et que tout point de la sphère peut-être déterminé à l'aide de deux angles appelés latitude et longitude. Par convention les latitude et longitude sont toujours données en degré d'angle, la latitude est l'angle selon l'axe nord-sud, toujours dans l'intervalle $[-90; 90]$, -90 correspond au pôle sud, $+90$ au pôle nord. La longitude est un angle selon l'axe Est-Ouest, entre $[-180; +180]$.

Distance entre points sur une sphère. Étant donné deux points p_1 et p_2 de latitude y_1 et y_2 et de longitude x_1 et x_2, on définit la distance entre p_1 et p_2 avec la formule de haversine $d(p_1, p_2)$ définie de la façon suivante :

$$d(p_1, p_2) = 2 \times R_{\text{terre}} \times arcsin\left(\sqrt{sin\left(\frac{y_1 - y_2}{2}\right)^2 + cos(y_1)cos(y_2)sin\left(\frac{x_1 - x_2}{2}\right)^2}\right)$$

où R_{terre} est le rayon terrestre, estimé à 6371 km.

Carte fournie. Dans cet exercice nous avons à disposition deux fichiers `routes.txt` et `positions.txt` décrivant une partie des routes de l'Île-de-France. Le premier fichier décrit des segments de routes qui existent sur notre carte tandis que le second fichier donne les coordonnées (latitude et longitude) des points utilisés pour décrire les segments de routes. Plus précisément :

— le fichier `routes.txt` commence par un entier, 2 111 499, qui décrit le fait qu'il y a 2 111 499 segments de route dans notre carte. Les 2 111 499 lignes suivantes décrivent chacune un segment de route. Chaque segment de route est donné comme un quadruplet (f, t, d, n) où f est un entier qui décrit le point de départ, t est un entier qui décrit le point d'arrivée, d est un nombre flottant et donne la distance en mètres entre les points f et t tandis que n décrit le nom de la route. Ces quatre éléments seront séparés par des espaces. Voici, par exemple, les 5 premières lignes de ce fichier :

```
2111499
0 176674 864.984 Autoroute du Nord
1 11731 522.499 Autoroute du Nord
2 81556 1618.6 <ROUTE SANS NOM>
3 7624 938.6759999999998 <ROUTE SANS NOM>
```

Noter qu'une même route (comme l'autoroute du Nord) sera généralement décrite par plusieurs segments, chaque segment représentant une portion assez courte de la route.

— le fichier `positions.txt` commence par un entier, 842 089, décrivant le fait qu'il y a 842 089 points décrits. Les 842 089 lignes suivantes décrivent chacune un point, plus précisément, la i-ème ligne du fichier décrit le point $i - 2$ en donnant d'abord sa longitude puis une espace en ensuite sa latitude. Voici, par exemple, les 5 premières lignes de ce fichier :

```
1431808
2.5511375 49.0834393
2.5421644 49.0391224
2.4635493 48.8840815
2.4692871 48.8990232
```

On a donc que le point 0 a pour longitude 2.5511375, que le point 2 a pour latitude 48.8840815, etc.

Attention ! La distance donnée dans le fichier `routes.txt` ne correspond pas toujours à la distance donnée par la formule de haversine car les routes ne sont pas forcément droites.

Question 1. Proposer, sans écrire le code correspondant, des structures données (tableaux, listes, etc.) pour stocker les données contenues dans les fichiers. Discuter ce choix de représentation en fonction.

Squelette du code. Pour cette question, on vous fournit un squelette de code qu'il faudra ensuite éditer. Ce squelette contient cinq fichiers :

— `main.c` qui sera le point d'entrée de notre programme ;
— `lecture.c` qui sera un fichier édité plus tard dans le sujet pour lire la carte de l'Île-de-France ;
— `lecture.h` qui est le fichier *header* associé à `lecture.c`.

Pour compiler le programme, il faut exécuter la commande `gcc lecture.c main.c -lm`

Question 2. Éditer la fonction `lit_position` du fichier `lecture.c` pour qu'elle lise et stocke les informations du fichier `positions.txt` afin que `position[i]` contienne les coordonnées du point `i`. Éditer ensuite la fonction `nettoie` pour qu'elle libère la mémoire allouée dans `lit_position`. Vous pouvez vous inspirer de la fonction `lit_route`.

Question 3. Éditer le fichier `main.c` et compléter la fonction `ll_distance` pour qu'elle calcule la distance entre deux points selon la formule d'Haversine. Selon votre fonction, quelle est la longueur moyenne d'un segment de route parmi les segments fournis ? Quelle est la longueur maximale d'un des segments ? Vérifier les réponses obtenues en comparant avec les longueurs des segments

Question 4. Créer les structures de données nécessaires au stockage des informations du fichier `routes.txt`. Attention, pour la compilation séparée, il faudra définir normalement les tableaux et variables dans `lecture.c` mais pour pouvoir y accéder depuis `main.c`, il faudra les redéfinir précédés du mot-clef `extern` dans `lecture.h`. Il est possible de s'inspirer de ce qui a été fait pour le tableau `position` de type `geopoint` ou de la variable `nb_routes`.

Question 5. Éditer la fonction `lit_routes` du fichier `lecture.c` pour qu'elle stocke les informations du fichier `routes.txt` dans les structures que vous venez de créer.

Graphe associée à une carte. À partir des données fournies on peut créer un graphe. Dans ce graphe, les nœuds sont les différents points dont les positions sont décrites dans le fichier `positions.txt` et les arêtes de ce graphe correspondent aux segments de route décrits dans le fichier `routes.txt` : il y a une arête entre i et j de longueur d s'il existe un segment de route (i, j, d, n) ou (j, i, d, n) pour un certain nom de route n (attention de la même façon qu'il peut y avoir plusieurs routes entre deux points, il pourra y avoir plusieurs arrêtes entre deux nœuds).

Représentation en liste d'adjacence. On veut maintenant créer un fichier contenant le graphe décrit ci-haut représenté en liste d'adjacence. Ce fichier aura sur sa première ligne le nombre de nœuds (ici $842\,089$) et ensuite il y aura $842\,089$ lignes. La $i+2$ ème ligne du fichier décrit les voisins du nœud i de la façon suivante d'abord on a un entier k, le nombre de voisins du nœud i, puis une espace suivie des $2k$ nombres séparés par des espaces $n_1\ d_1\ \ldots\ n_k\ d_k$, décrivant les k voisins $n_1 \ldots n_k$ respectivement à distance $d_1 \ldots d_k$.

Question 6. Décrire quelles sont les grandes étapes à suivre pour pouvoir stocker dans un fichier la représentation en liste d'adjacence du graphe associé aux fichiers `positions.txt` et `routes.txt`.

Question 7. Écrire une fonction `sauvegarde_graphe` qui stocke dans le fichier `graphe.txt` la représentation en liste d'adjacence du graphe routier. Tester la fonction, combien de temps met-elle à s'exécuter ?

2 Exercice 2 : manipulation simple de base de données à l'aide de SQLite3

Pour cet exercice, on vous fournit un fichier `exercice_2.db` qui stocke une base de donnée SQLite3. Pour pouvoir exécuter des commandes sur ce fichier on peut lancer la commande `sqlite3 exercice_2.db` depuis le dossier où se trouve ce fichier. Cette commande lance un interpréteur SQLite3. On peut par exemple taper la commande suivante pour avoir le nombre d'enregistrements dans la table `adresse` :

```
SELECT COUNT(*) FROM adresse ;
```

Une des utilisations de cette table `adresse` est la recherche de la position géographique d'une adresse. Cette table a 7 colonnes :

- `numero` qui stocke des entiers et correspond à des numéros au sein d'une voie ;
- `rep` est un complément au numéro (par exemple si on a le numéro "2 bis", 2 ira dans la colonne numéro mais "bis" ira dans `rep` ;
- `nom_voie` stocke le nom de la voie ;
- `code_postal` stocke le code postal ;
- `nom_commune` stocke le nom de la commune ;
- enfin les communes `lon` et `lat` stockent les longitudes et latitudes associées.

La base fournie contient les adresses de l'Essonne. On suppose que deux adresses sont dans la même ville exactement quand elles partagent un nom de commune. Pour rappel, plus un point est au nord, plus sa latitude est élevée.

Question 1. Récupérer les latitude et longitude de Télécom Paris, situé au 19, place Marguerite Perey, 91120 à Palaiseau.

Question 2. Écrire une requête qui calcule pour chaque ville, la latitude la plus au nord d'une adresse de cette ville. Il est attendu une requête qui renvoie deux colonnes, une avec des noms de communes et une avec des latitudes.

Question 3. Quels sont les noms de voies qui apparaissent dans deux codes postaux différents ?

Question 4. Écrire une requête qui affiche les villes qui sont entièrement au nord de Palaiseau. C'est à dire, les villes pour lesquelles toutes les adresses de la base de données sont au nord de l'adresse la plus au nord de Palaiseau.

Remarque : il existe généralement différentes manières de résoudre les questions de SQL. Toutes les questions posées peuvent être résolues avec le fragment de SQL étudié dans le programme mais les candidats peuvent proposer des solutions qui utilisent des fonctionnalités hors programme. Le jury se réserve alors le droit de demander une autre solution au candidat (ce sera notamment le cas si la solution du candidat repose sur des fonctionnalités trop exotiques).

3 Exercice 3 : algorithmes de graphes en OCaml

Pour cet exercice, on vous fournit 4 fichiers :

— `graphe.txt` qui contient un graphe en représentation de liste d'adjacence, dans un format similaire à celui obtenu à la fin de l'exercice 1. Ce graphe est non orienté et sans arêtes multiples ;

— `oriente.ml` qui est le fichier principal à éditer dans cet exercice. Il contient déjà les fonctions `lit_graphe` et `lit_position` lisant les fichiers `graphe.txt` et `positions.txt` (voir exercice 1) ;

— enfin les deux derniers fichiers sont `PQ.ml` et `PQ.mli` et correspondent à un module appelé `PQ` qui sert de file de priorité. L'interface de ce module, donnée dans `PQ.mli`, décrit comment utiliser ce module.

Pour pouvoir utiliser le module `PQ` il faut d'abord le compiler. Pour cela on va compiler l'interface du module avec la commande `ocamlc PQ.mli` qui va créer un fichier d'interface de module compilé `PQ.cmi`. Ensuite on va compiler le module lui-même, avec la commande `ocamlc PQ.ml` qui va créer un fichier `PQ.cmo`.

Ensuite pour exécuter `oriente.ml` on peut soit le lancer dans l'interpréteur soit en le compilant mais dans les deux cas en précisant d'utiliser le module PQ dans la ligne de commande. Pour le compiler on peut faire `ocamlc PQ.cmo oriente.ml` et pour l'interpréter on peut faire `ocaml PQ.cmo oriente.ml`. Si vous voulez éditer `oriente.ml` avec interpréteur interactif, il faudra lancer l'interpréteur avec la commande `ocaml PQ.cmo`.

Question 1. Exécuter, soit dans l'interpréteur soit en compilant, `oriente.ml`.

Question 2. Le graphe lu par la fonction `lit_graphe` contient-il une ou plusieurs composantes connexes ?

Question 3. Écrire une fonction qui calcule le plus court chemin entre deux points en utilisant l'algorithme de Dijkstra. La distance d'un chemin dans le graphe est la somme des distances des arêtes composants le chemin (la distance d'une arête est fournie). Il est recommandé d'utiliser le module PQ pour cette question.

Quelle est la distance du plus court chemin entre le point 819 913 (qui est proche de Télécom Paris) et le point 282 392 (qui est proche de la station Le guichet du RER B).

Question 4. On vous garantit que la complexité algorithmique des fonctions `retire_min`, `recupere_min` et `ajoute` du module PQ est logarithmique en le nombre d'éléments de la file à priorité passée en paramètre à ces fonctions. Quelle est la complexité de votre algorithme ? Justifier.

Question 5. On suppose maintenant que, comme dans le premier exercice, le fichier `positions.txt` contient les positions des points du graphe et donc que la distance dans le graphe entre deux points de positions p_1 et p_2 est nécessairement plus grande que la distance donnée par la formule de haverstine. Expliquer comment on peut utiliser cette information à l'aide de l'algorithme A*. Cet algorithme A* a-t-il une meilleure complexité que l'algorithme Dijkstra ? A-t-il des chances d'être plus rapide en pratique sur un graphe comme celui fourni (graphe correspondant aux routes de l'Île-de-France). Expliquer.

Question 6. Écrire une fonction calculant le plus court chemin avec l'algorithme A*. Comparer le nombre de nœuds explorés dans l'algorithme A* et dans l'algorithme Dijkstra.