EINSTEIN
Is a New and Super-fast Traveler Engine for Intellectual Newbies.

Camille Raymond Jerôme Audoux Justine Ranc
Yann Pravossoudovitch

29 mai 2011

"Logic will take you from A to B. Imagination will take you everywhere."

A. Einstein

Table des matières

1 Contexte

Ce projet s'inscrit dans le cadre du cours de langage C et traite de données géo-localisées sur une carte. Le site : http ://www.world-gazetteer.com a été utilisé pour obtenir les villes de tous les pays avec leur population et leurs coordonnées GPS.

Le programme est une solution visant à manipuler des données géo-localisées afin d'effectuer certains traitements sur les villes ainsi que leurs liaisons avec d'autres selon des paramètres données, comme l'autonomie d'un avion etc.

Pour réaliser ce programme, nous avons mis à profit nos connaissances sur les graphes pour obtenir un programme fonctionnel et répondant aux besoins du projet.

2 Objectifs

Le but de ce projet est de développer un programme C qui implémente les fonctionnalités suivantes :

- Lecture des données en entrée à partir d'un fichier CSV : fichier type tableur et format plus complexe avec utilisation d'une grammaire ANTLR.
- Affichage des X plus grandes villes sur Google Maps : à partir d'un fichier donné (CSV) récupération des X plus grandes villes et retranscription dans un fichier html.
- Calcul le plus court chemin entre deux villes "en petit avion" qui possède une autonomie (spécifiée).
- Calcul de l'arbre aérien couvrant minimal, prenant en compte l'autonomie de l'avion.
- Lecture des fichiers d'entrées dans un format plus complexe.

Remarque : On entend par "autonomie de l'avion" le nombre limité de kilomètres que peut parcourir l'avion.

Des points tels que l'optimisation des fonctions et l'utilisation de GitHub (plateforme collaborative) seront évalués à titre de bonus.

Une analyse globale en termes de complexité, de mémoire et de performance de calcul devra être réalisée durant le projet.

Date de rendu : Le rapport ainsi que le code source devra être remis au responsable de TD le 30 Mai avant 8h par email selon des procédures spécifiques.

Soutenance orale : Une soutenance aura lieu une semaine après le rendu du rapport et du code source. Elle se déroulera en plusieurs étapes : Présentation du projet, présentation des choix et tests effectués et discussions.

3 Outils et méthodes

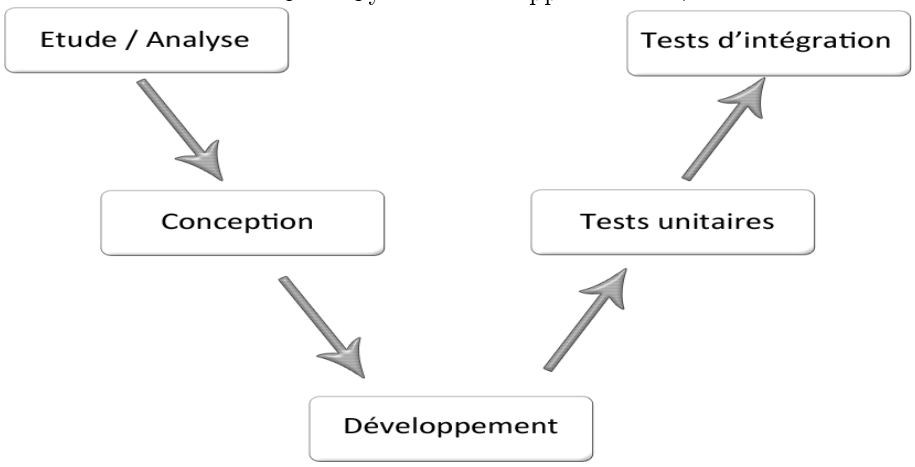
3.1 Modèle de gestion de projet

La première difficulté du projet a été l'organisation entre les membres ainsi que le déroulement du projet. Il a été décidé par le groupe de suivre, de manière globale, le "Cycle de développement en V", très utilisé dans la gestion de projet.

Il s'agit de partir d'un haut niveau d'abstraction permettant de déterminer les besoins, puis de se rapprocher des étapes concrètes pour ensuite effectuer des tests afin de livrer un programme répondant le plus pertinemment possible aux objectifs.

Le programme demandé n'étant pas un "vrai" logiciel, nous n'avons pas pu effectuer de réels tests d'intégration et de validation, notemment avec les utilisateurs mais avons imaginé des "scénarios" d'utilisation du programme (nous avons donc joué le rôle des utilisateurs).

FIG. 3.1 – Cycle de développement en V



3.2 Outils de l'organisation

3.2.1 Plateforme collaborative

Tout au long de ce projet, nous avons utilisé la plateforme GitHub pour notre code source, elle nous a été imposée mais a cependant présenté de nombreux avantages.

Elle permet avant tout de partager le code source tout au long du projet avec un système de "versioning" donnant la possibilité à chaque membre de travailler sur la version la plus récente. En cas de mauvaise manipulation, le système conserve toutes les versions et peut les fournir à tout moment.

Fonctionnalités supplémentaires :

- **Page Wiki :** Pour recenser les comptes rendus des réunions, nos recherches sur le projet ou les liens de sites utiles pour les algorithmes du projet.
- **Onglet "Issues" :** Permet de mettre à plat les problèmes (un TO-DO List) et d'assigner des travaux à chacun (à l'aide de "label") .
- **Commentaires :** De nombreuses zones de commentaires sont présentes. Elles se présentent sous plusieurs formes : commentaires sur les commits ou sur le code en lui-même et autres, etc.
- **Graphiques :** Permet l'édition de graphiques relatifs au "commit", au trafic sur la plateforme.

3.2.2 Coding Style

Après la mise en place des outils de communication, il a fallu déterminer la charte de programmation entre les membres. Le but étant d'obtenir un code clair, lisible et uniforme. Nous nous sommes donc inspirés de "Coding Style" déjà réalisés tout en y ajoutant des éléments personnels. Celui-ci se trouve dans le dossier /doc.

4 Structure Générale

4.1 Introduction

On peut voir le programme comme une succession de 3 étapes :

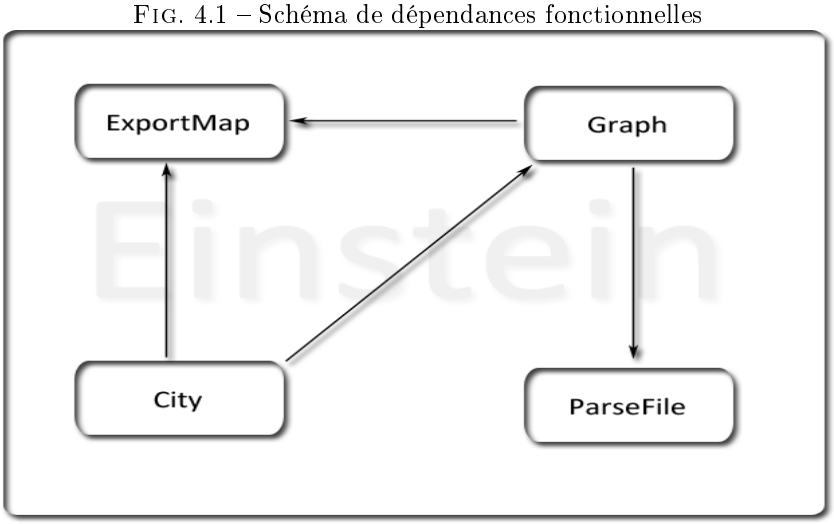
- La récupération des données à traiter.
- Le traitement de ces données (en interne).
- Le résultat, écriture dans un fichier HTML pour un affichage écran.

Ainsi, il y a 3 catégories auxquelles les fonctions peuvent être réparties (note : évidemment la partie du traitement est la partie la plus conséquente).

Remarque : Les villes sur lesquelles nous allons travailler possèdent toutes un nom et des coordonnées exprimées en degrés ou radians. Lorsque l'on utilisera le terme "**distance**", celle-ci sera exprimée en kilomètre par rapport à un point donnée.

4.2 Schéma des dépendances fonctionnelles

FIG. 4.1 – Schéma de dépendances fonctionnelles



4.3 Détails des structures et fonctionnalités du programme

4.3.1 La récupération des données à traiter

Les différentes fonctions qui vont être présentées se trouvent dans les fichiers ParseFile.(h/c) du répertoire src/headers pour le ".h" et src/source pour le ".c". Il s'agit ici de récupérer les données afin de les rendre lisibles pour les structures de traitement interne que nous avons choisies.

Compilation

Nous avons réalisé un **Makefile** qui relie l'ensemble des fichiers de notre programme.

Le menu

Le menu Einstein est UNIX-like. Il prend donc plusieurs arguments grâce au caractère '-' pour la forme courte ou les caractères '–' pour la forme longue (ex : -a ou –autonomy). Le programme affiche également une aide accessible par l'argument -h (ou –help) à chaque fois que l'utilisateur fait une erreur dans la saisie des arguments. La sécurité est alors optimisée puisque les saisies de l'utilisateur sont restreintes aux arguments définis et que tous les cas possibles sont traités et contrôlés.

Fonctions importantes associées

parseCity : extrait les informations relatives à une ville.
genFromFile : construit une structure "Graph" (explicitée dans le paragraphe suivant) à partir d'un fichier CSV qui doit contenir des villes, leur population, et les coordonnées géographiques. Elle utilise les fonctions d'extraction de villes parseAllCities.

Récupération de fichier plus complexe

Afin de mettre en pratique les théories vues en compilation, nous avons rédigé une grammaire à l'aide d'ANTLR permettant de générer un fichier CSV à partir d'un fichier de type XML.

La grammaire tient compte de la syntaxe spécifique au fichier XML fourni et permet de récupérer pour chaque ville, son nom sa population et ses coordonnées. Cette grammaire nous a permis de générer un parser XML écrit en langage C.

C'est un module complémentaire à notre projet, situé dans le dossier "Parser". Nous avons décidé de ne pas l'intégrer complètement à l'application car la présence d'ANTLR sur une machine aurait été nécessaire à la compilation de l'ensemble du projet.

Il faut donc d'abord convertir son fichier XML à l'aide du parser, puis exécuter l'application "Einstein" avec le fichier CSV généré.

4.3.2 Le traitement des données (en interne)

Les différentes fonctions qui vont être présentées se trouvent dans les fichiers Graph.(h/c) et City.(h/c) du répertoire src/headers pour les ".h" et src/source pour les ".c".

Les structures

Structure City

La structure City (ville) est le noyau du programme. Elle sera utilisée dans des matrices lors des différents traitements. C'est sur les villes que l'ensemble du programme tourne. Il traite de chemin le plus court entre deux villes, d'arbre recouvrant minimal pour un nombre précis de villes etc. Une ville est caractérisée par son nom (nameCity), sa population (populationCity) ainsi que son orientation Nord-Sud (orientationNS) et Est-Ouest (orientationEW).

Fonctions associées à la structure City

distance : permet de connaître la distance entre deux villes à partir de leurs coordonnées (on utilisera une fonction de conversion degrés/radians pour le calcul).

Fonction "bonus"

Le sujet proposait d'implémenter une fonction supplémentaire, destinée à faciliter la saisie des villes par l'utilisateur. Fonction de recherche d'une ville par son nom (searchCity) utilisée pour la complétion automatique.

Fonction affichage des X plus grandes villes

Pour afficher les X plus grandes villes, il a fallu trier le fichier par population. Nous avons utilisé le Quick Sort. La complexité de notre algorithme est la complexité de notre algorithme de tri (QuickSort) qui est en 0(n*log(n)), avec n le nombre de villes.

Structure Graph

Lors des différents algorithmes, nous avons eu besoin de définir une structure interne au programme s'approchant de la définition que l'on a d'un graphe, afin de raisonner grâce à la théorie des graphes abordée durant le semestre.

Cette structure comprend une matrice de villes (structure City) ainsi qu'une matrice des liaisons entre les villes et le nombre de villes sur lequel on travaille.

La matrice de liaison entre les villes est calculée grâce à une distance (autonomie) donnée en paramètre et détermine ainsi si oui ou non la distance réelle entre les 2 villes est prise en compte. Un "Graph" est créé automatiquement après lecture d'un fichier CSV de villes.

La matrice à été privilégiée vis à vis de la liste de villes ou de liaisons pour sa simplicité de mise en œuvre et sa rapidité d'accès.

Fonctions associées à la structure Graph

Les fonctions basiques associées à cette structure sont la création, l'initialisation et la suppression d'un graphe.

Structure Edge

Elle représente une arête (au sens de la théorie des graphes) contient deux sommets associés à des villes mais est représentée par des entiers (pour déterminer des poids affectés à chaque ville). A ces paramètres, s'ajoute la distance entre les deux sommets.

Fonctions associées à la structure Edges

Les fonctions importantes associées à cette structure sont :

- tri des arêtes par distance (Quick-Sort)
- calcul du rapport entre le nombre d'arêtes effectives d'un graphe et le nombre d'arêtes obtenues dans le graphe complet.

Les fonctions principales

Calcul du plus court chemin : Dijkstra (Fonction : shortestPath).

Il s'agit ici de calculer le plus court chemin entre deux villes. Nous avons donc mis à profit nos connaissances en graphe et avons implémenté le légendaire algorithme du néerlandais Edsger Dijkstra. Notre fonction du plus court chemin : "ShortestPath" récupère en paramètre un graphe ainsi qu'une ville de départ et une ville d'arrivée sous forme d'entier (indice de la ville dans la matrice). Elle retourne ensuite un chemin sous forme d'un tableau de villes.

Principe de l'algorithme

L'algorithme possède une boucle principale qui ne s'arrête qu'à deux conditions :

1) la ville d'arrivée vient d'être atteinte.

2) il n'y a plus de nouvelle ville à visiter.

Dans le premier cas, on vient de trouver le plus court chemin entre les deux villes, on le stocke alors dans la variable path et on retourne une valeur de succès. Dans le second cas on retourne une valeur d'erreur.

A chaque tour de boucle on effectue deux phases principales.

Phase A : Calcul des distances de la ville en par rapport à ses voisins. Depuis la ville en cours on calcule pour chacun de ses voisins leurs distances à la ville de départ en prenant en compte la distance actuellement parcourue.

Phase B : Sélection de la nouvelle ville à analyser. On cherche dans la liste des villes une ville qui ne soit pas encore visitée mais une pour laquelle on a déjà calculé une distance jusqu'à la ville de départ. On sélectionne alors la ville avec la plus faible distance.

Complexité : $0(n^2)$. Avec n le nombre de villes.

Calcul de l'arbre recouvrant de poids minimal : Kruskal

Il s'agit ici de trouver l'arbre recouvrant de poids minimal c.à.d. qu'à partir d'un nombre donné de villes on cherche à trouver un graphe connexe qui passe par toutes les villes, tout en minimisant le poids des arêtes.

Voir exemple ci-dessous.

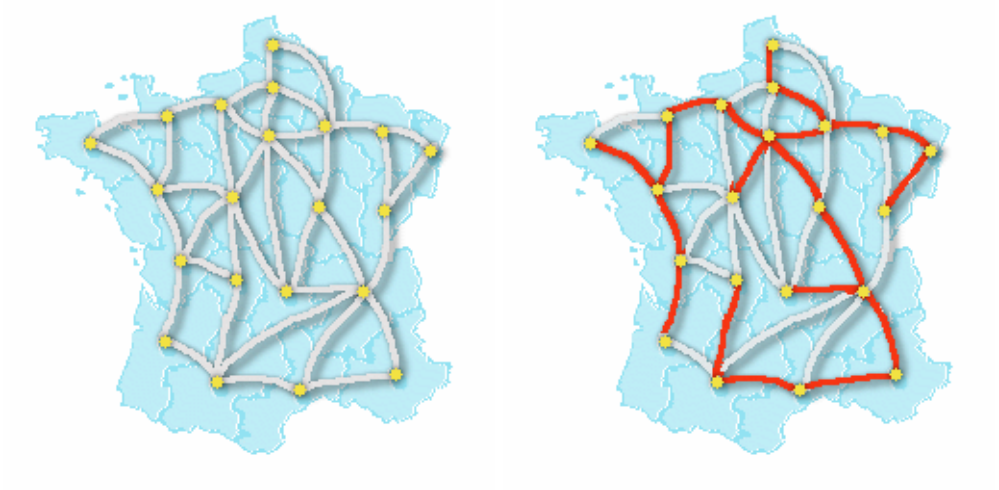
Pour l'implémentation de l'arbre aérien couvrant minimal, nous avons pensé à deux algorithmes : l'algorithme de Prim ou de Kruskal.

Il est plus optimisé d'utiliser Kruskal car la complexité de l'algorithme est en fonction du nombre d'arêtes contrairement à Prim où celui-ci se fait en fonction du nombre de sommets. Nous avons pensé que l'algorithme de Kruskal, se basant sur le nombre d'arêtes, serait plus efficace. Nous avons donc fait une fonction graphEdgesPourcentage qui calcule le pourcentage d'arêtes utilisées et l'on a vu que le graphe était rarement complet. Le choix de Kruskal s'est donc confirmé.

Le principe de l'algorithme est de ranger par ordre de poids (ici la distance) croissant les arêtes d'un graphe. Ensuite, pour chaque arête prise par ordre de poids croissant, on les ajoute à l'arbre couvrant minimal si celui-ci n'admet pas de cycle.

Complexité : $0(m.\log(n))$. Avec m le nombre d'arêtes et n le nombre de sommets.

FIG. 4.2 – Schéma arbre recouvrant de poids minimal



4.3.3 Le résultat, écriture dans un fichier HTML pour un affichage écran

Les différentes fonctions qui vont être présentées se trouvent dans les fichiers exportMap.(h/c) du répertoire src/headers pour le ".h" et src/source pour le ".c".
Il s'agit ici, après traitement, de "reconvertir" les données traitées dans un format lisible par l'utilisateur. Plus concrètement, modéliser les tracés de villes sur un fichier HTML de carte dynamique.

Fonctions associées à la sortie du programme

La fonction la plus importante est exportMap qui affiche les villes ainsi que les chemins sur un fichier HTML selon le(s) traitement(s) demandé(s) par l'utilisateur. (Arbre recouvrant minimal, plus court chemin ...).

Nous avons déclenché automatiquement l'ouverture d'un navigateur et du fichier HTML à la fin du programme en utilisant la commande System("firefox ‘pwd‘/map-output.hmtl"). Mais celle-ci ne fonctionne uniquement sous Linux et avec Mozilla Firefox.

5 Analyse et Tests

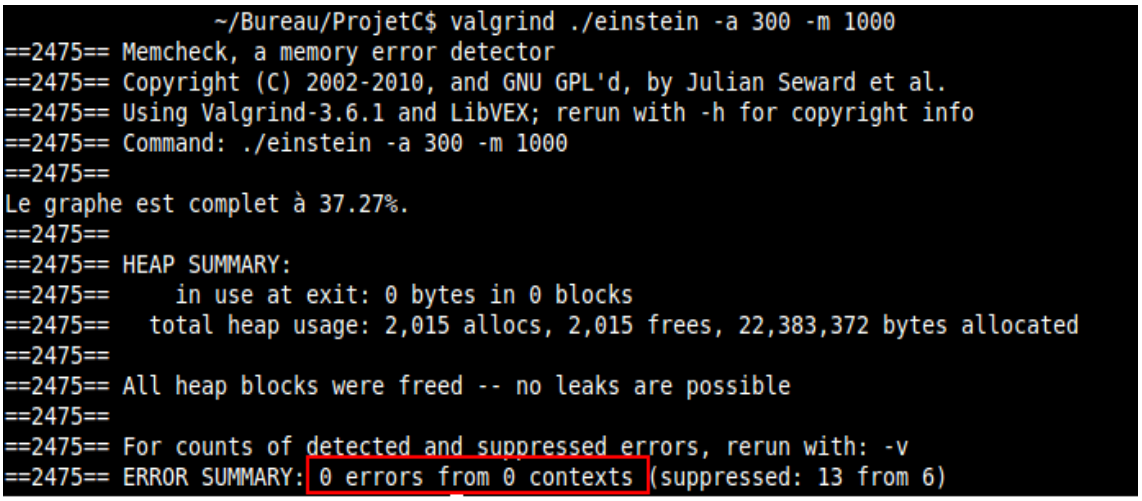
Après "débuggage" technique du programme c.à.d. problèmes de syntaxe (etc.), il faut analyser le programme en lui-même. Cela comprend les éléments non visibles, comme la mémoire. Il faut ensuite tester le programme afin d'apprécier en détail son bon fonctionnement.

5.1 Analyse de la mémoire : Valgrind

"*Valgrind est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires.*" Wikipédia

Nous avons utilisé le profiler mémoire Valgrind afin de détecter les fuites mémoires du programme. En effet, le programme est susceptible de traiter des données de taille importante, de ce fait il est nécessaire qu'il libère bien toute la mémoire allouée. Par exemple, nous avons été confrontés à des malloc et free non symétriques et avons pu les rééquilibrer grâce au programme.

FIG. 5.1 – Valgrind



5.2 Optimisation -O3

Pour optimiser le temps d'exécution de notre programme, nous avons utilisé l'option -O3 (de gcc) à la compilation.

Cette option optimise la compilation pour les fonctions les plus gloutonnes en temps et mémoire et elle réduit la taille du code sans faire les optimisations qui prennent le plus de temps. Elle s'exécute aussi sur les autres options de gcc.

5.3 Performance en temps de calcul : Flat Profile

Flat Profile détaille combien de temps processeur dure chaque fonction et combien de fois elle a été appelée. C'est un bref résumé des informations de profilage rassemblées. Ceci donne une idée des fonctions qui peuvent être réécrites ou affinées pour gagner en performance.

5.3.1 Informations techniques de Flat Profile

Flat Profile nous donne qu'une approximation des temps d'exécution pour chaque fonction appelée dans le "main". Les résultats peuvent varier d'un test à un autre (et d'une machine à une autre). La colonne %time donne le temps total passé dans la fonction main y compris les appels à d'autres fonctions. La précision des temps d'exécution est de 0.01 secondes. Ainsi, les temps d'exécution des fonctions qui sont inférieurs à 0.01 secondes afficheront 0.00 secondes par manque de précision.

Nous avons effectué plusieurs tests concernant notre programme.

5.3.2 Flat Profile sur les fonctions principales

Dijkstra

FIG. 5.2 – Flat Profile sur Dijkstra

```
288 Draguignan
163 Cannes
start: Unknown job: http://map-output.html
                ~/Bureau/ProjetC$ gprof ./einstein
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 40.00      0.04     0.04        1    40.00    70.00  createMatrix
 30.00      0.07     0.03   499501     0.00     0.00  distance
 20.00      0.09     0.02        1    20.00    20.00  shortestPath
 10.00      0.10     0.01        1    10.00    10.00  deleteGraph
  0.00      0.10     0.00     1000     0.00     0.00  createCity
  0.00      0.10     0.00     1000     0.00     0.00  parseCity
  0.00      0.10     0.00        2     0.00     0.00  searchCity
  0.00      0.10     0.00        1     0.00     0.00  exportMapPath
  0.00      0.10     0.00        1     0.00    70.00  genFromFile
  0.00      0.10     0.00        1     0.00     0.00  graphEdgesPourcentage
```

Données : Autonomie de 50km : Chemin le plus Court entre deux villes (parmi 1000 villes, distance : 1040km)

On peut voir que trois fonctions (createMatrix, shortestPath et genFromFile) représentent presque tout le temps d'exécution. Il n'y a qu'un seul appel de fonction à shortestPath et il faut compter 20.00 millisecondes pour chaque appel.

Il y a 499501 appels à la fonction distance et son exécution prend **plus de 30% du temps** total d'exécution du programme. Cependant, le temps d'exécution de cette fonction est très rapide (<0.01ms).

La fonction createMatrix n'est invoquée qu'une fois, mais elle prend **plus de 40% du temps** d'exécution du programme. Il est possible que cette fonction fasse beaucoup de choses et qu'elle soit un bon candidat à un découpage en plusieurs fonctions.

Il est également à noter que chaque appel aux fonctions genFromFile et createMatrix prend 70 millisecondes, ce qui est assez "long" comparé aux autres fonctions.

Arbre recouvrant minimal

FIG. 5.3 – Flat Profile sur l'arbre recouvrant minimal

```
                ~/Bureau/ProjetC$ ./einstein -a 1300 -m 1000
The graph is at 100.00% full.
camille@C4MI773:~/Bureau/ProjetC$ gprof ./einstein
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 70.27      0.26     0.26                             sortByDistance
 16.22      0.32     0.06   499500     0.00     0.00  distance
  5.41      0.34     0.02        1    20.00    80.00  createMatrix
  5.41      0.36     0.02        1    20.00    20.00  minSpanningTree
  2.70      0.37     0.01        1    10.00    10.00  graphEdgesPourcentage
  0.00      0.37     0.00     1000     0.00     0.00  createCity
  0.00      0.37     0.00     1000     0.00     0.00  parseCity
  0.00      0.37     0.00        2     0.00     0.00  deleteGraph
  0.00      0.37     0.00        1     0.00     0.00  exportMap
  0.00      0.37     0.00        1     0.00    80.00  genFromFile
  0.00      0.37     0.00        1     0.00     0.00  nbCities
  0.00      0.37     0.00        1     0.00     0.00  quickSortPopulation
```

Données : Autonomie de 1300km (graphe complet à 100%) : arbre recouvrant minimal des 1000 villes sur un graphe de 1000 villes.

On peut voir que quatre fonctions (createMatrix, exportMap, minSpanningTree et genFromFile) **représentent presque tout le temps d'exécution**.

Il n'y a un seul appel de fonction à minSpanningTree et il faut compter 20.00 millisecondes pour chaque appel.

Il y a 499500 appels à la fonction distance et son exécution prend environ **16% du temps** total d'exécution du programme. **Son temps d'exécution de cette fonction est très rapide (<0.01ms)** Il est également à noter que chaque appel aux fonctions genFromFile et createMatrix prend 80 millisecondes. On peut donc faire la même remarque que précédemment et discuter à l'avenir d'un découpage de cette fonction.

Calcul des N villes les plus peuplées

FIG. 5.4 – Flat Profile sur l'arbre recouvrant minimal

```
The graph is at 100.00% full.
                ~/Bureau/ProjetC$ gprof ./einstein
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     4950     0.00     0.00  distance
  0.00      0.00     0.00     1000     0.00     0.00  createCity
  0.00      0.00     0.00     1000     0.00     0.00  parseCity
  0.00      0.00     0.00        1     0.00     0.00  createMatrix
  0.00      0.00     0.00        1     0.00     0.00  deleteGraph
  0.00      0.00     0.00        1     0.00     0.00  exportMap
  0.00      0.00     0.00        1     0.00     0.00  genFromFile
  0.00      0.00     0.00        1     0.00     0.00  graphEdgesPourcentage
  0.00      0.00     0.00        1     0.00     0.00  nbCities
  0.00      0.00     0.00        1     0.00     0.00  quickSortPopulation
```

En ce qui concerne l'option "Afficher les N plus grandes villes", quelque soit le nombre de villes à afficher, **le temps d'exécution de chaque fonction est inférieur à 0.01ms**. En conclusion les résultats confortent nos choix quant à la **rapidité du programme**.

5.4 Tests unitaires

Après résolution des fuites de mémoire nous avons effectué des tests unitaires. Ces tests permettent de s'assurer du bon fonctionnement du programme en testant l'exécution de chaque fonction.

"*On écrit un test pour confronter une réalisation à sa spécification. Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification. Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie. Le test permet de vérifier que la relation d'entrée/sortie donnée par la spécification est bel et bien réalisée.*" Wikipédia

Les tests unitaires en C sont réalisables grâce à la libraire C-unit. Cette librarie permet d'avoir accès à des fonctions telles que "assertEqualsM" qui permet de tester les fonctions une par une et si les spécifications de celles-ci – donc ce qu'elles doivent faire en théorie – sont bien vérifiées en pratique. Si ces dernières ne sont pas vérifiées, C-Unit renvoie un message d'erreur précédemment mis en argument de la fonction "assertEqualsM".

FIG. 5.5 – Flat Profile sur l'arbre recouvrant minimal

```
=================================================
|               |  failed  |  succeed  |  total  |
|-----------------------------------------------|
| assertations: |        0 |        11 |      11 |
| tests:        |        0 |         8 |       8 |
| tests suites: |        0 |         4 |       4 |
=================================================
```

6 Discussions

6.1 Résultats

Pour ce projet, nous avons choisi la structure de tableaux pour sa facilité de développement, de mise en œuvre et de maintenance. Ce choix a aussi été fait dans le but de privilégier la rapidité par rapport à la mémoire.

Nous avons conservé ce choix pour toutes nos structures afin de créer une homogénéité du code. Au niveau de la maintenance et de l'optimisation, une structure de tableau nous a semblé simple et ergonomique, et donc il sera facile d'ajouter de nouvelles fonctionnalités !

Au niveau de la sécurité, nous nous sommes concentrés sur les moyens de mettre en péril l'exécution du programme et avons conclu que l'utilisateur était le principal danger. Il doit saisir **des données valides** au bon moment. Nous avons donc limité et contrôlé les données saisies ainsi que leur validité dans un menu associé à une aide qui peut s'afficher si l'utilisateur saisit de mauvaises données ou en oublie.

6.2 Perspectives et améliorations

Des structures plus complexes mais plus optimisées auraient pu être utilisées, comme les tas pour trier les arêtes ou les listes d'adjacence pour notre structure de graphe (plutôt que la matrice), ce qui nous aurait permit un gain de mémoire.

La réalisation d'une interface graphique pour le programme aurait été une bonne chose pour la lisibilité du programme côté utilisateur.

7 Conclusion

Tout au long de notre projet, nous nous sommes attelés à programmer "propre" et à respecter les étapes de conception afin d'assurer une bonne longévité au programme et de permettre son évolution. Ainsi, nous avons suivi le principe que "n'importe quel programmeur puisse facilement reprendre le programme et le maintenir".

Nous avons compris l'intérêt de la communication au sein d'un projet et sommes conscients de la simplification des procédures que nous a fourni GitHub. La mise en place de règles (comme le Coding style), de tâches assignées et la mise en avant de problèmes ont été des outils au cœur du fonctionnement de notre projet.

Le projet, algorithmiquement parlant, a été basé sur la théorie des graphes. Rechercher des algorithmes avec des théories connues a été très formateur au sens de la compréhension et de la clarté des algorithmes. Nous ne les avons pas "inventés" mais les avons **appliqués** et **compris**, puis implémentés. Il n'y a donc pas eu de "test erreurs" et nous avons évité le **syndrome du programmeur fou** qui se jette sur son ordinateur...