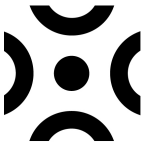
Perfectionnement à la programmation en C
Le voyageur de commerce génétique
L2 Informatique - 2024–2025
Fiche de TP 9



L'objectif de ce TP est d'implanter un programme de résolution du problème du voyageur de commerce [1]. Une approche basée sur la programmation génétique [2] est employée. Nous utilisons et approfondissons ici les notions de

- lecture d'une spécification ;
- découpage d'un projet en modules ;
- déploiement d'un projet important ;
- initiation à la programmation génétique.

Ce TP est beaucoup moins directif que les précédents. Il ressemble plus à une véritable spécification fournie par un client. Il existe ainsi plusieurs façons de mener à bien le travail demandé, mais il faut faire attention à répondre exactement à la spécification et à utiliser **tout** ce qui a été vu pour le moment dans les cours et TD/TP précédents (comme entre autres, la mise en place de pré-assertions, les fonctions à gestion d'erreurs, l'écriture de documentation, la bonne façon de pratiquer la modularisation et la mise en place de tests).

1 Spécification

La spécification du projet est divisée en deux parties : nous commençons par donner un aperçu global du problème que l'application est destinée à résoudre, puis nous nous focaliserons sur la méthode basée sur la programmation génétique à mettre en œuvre.

1.1 Le problème du voyageur de commerce

Le *problème du voyageur de commerce* décrit une situation qui prend en entrée un ensemble de villes avec leur position géographique, et qui consiste à fournir en réponse un ordre de visite de ces villes qui soit le plus court possible.

Plus précisément, l'utilitaire à concevoir, nommé `pvc`, prend en entrée un fichier texte dont chaque ligne est de la forme

```
VILLE X Y
```

où `VILLE` est un nom de ville et `X` et `Y` sont des entiers qui spécifient la position de la ville dans le plan cartésien discret (`X` est l'abscisse et `Y` est l'ordonnée). Le nom d'une ville ne contient que des caractères alphabétiques ou le caractère `'-'`. Chaque ligne du fichier contient ainsi les données nécessaires à la représentation d'une ville. On appelle *carte* la configuration de villes décrite par un tel fichier.

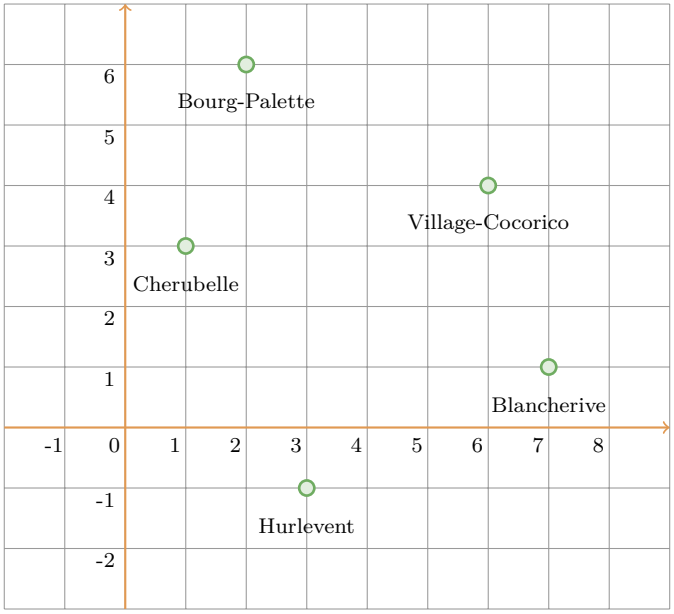
Par exemple, le fichier texte de contenu

```
Bourg-Palette 2 6
Blancherive 7 1
```

[1] Voir https://fr.wikipedia.org/wiki/Problème_du_voyageur_de_commerce.
[2] Voir https://en.wikipedia.org/wiki/Genetic_programming.

```
Hurlevent 3 -1
Cherubelle 1 3
Village-Cocorico 6 4
```

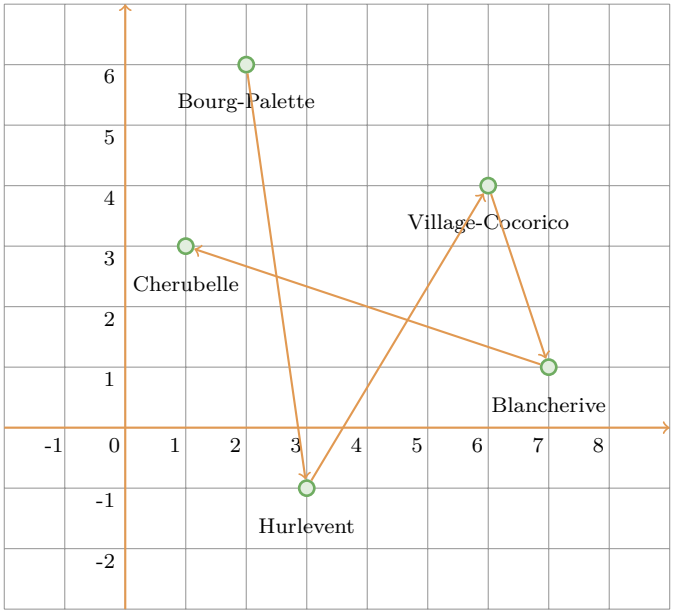
spécifie la carte



Une *visite* est un ordre de parcours des villes de la carte. La *longueur* d'une visite est le nombre total de kilomètres que demande la visite pour être effectuée[3]. Les visites sont enregistrées dans des fichiers texte sous la forme d'une ligne contenant les villes à visiter dans l'ordre, séparées par une espace. À la fin de la ligne, figure la longueur totale de la visite :

```
VILLE_1 VILLE_2 ... VILLE_N LONGUEUR
```

Par exemple, en reprenant la carte de l'exemple précédent,



illustre la visite des villes de la carte dans l'ordre Bourg-Palette, Hurlevent, Village-Cocorico, Blancherive, Cherubelle. En calculant les distances entre les villes[4], on obtient que la longueur de cette

[3] Chaque unité vaut 1 kilomètre.
[4] En utilisant le théorème de Pythagore par exemple.

visite est de

$$7.07\text{km} + 5.83\text{km} + 3.16\text{km} + 6.32\text{km} \simeq 22.39\text{km}.$$

Ce résultat est enregistré dans le fichier texte de contenu

```
Bourg-Palette Hurlevent Village-Cocorico Blancherive Cherubelle 22.39
```

On observe, bien évidemment, que toutes les visites n'ont pas la même longueur. Par exemple, on vérifie sans peine que la visite Bourg-Palette, Cherubelle, Hurlevent, Blancherive, Village-Cocorico est meilleure que la précédente car sa longueur, de 15.27km, est plus petite.

L'objectif de l'application est de fournir une visite dont la longueur est la plus petite possible en travaillant sur des cartes qui contiennent potentiellement un grand nombre de villes.

1.2 Fonctionnement de l'application

Une fois lancé, l'utilitaire `pvc` affiche la carte dans une fenêtre graphique. Lorsque la touche `Entrée` est pressée, le programme calcule la meilleure visite possible de la carte donnée selon une méthode expliquée plus loin. Dès qu'il trouve une visite meilleure que celle qu'il a précédemment calculée, il l'affiche sur la sortie standard, la dessine dans la fenêtre graphique (en effaçant au préalable la précédente) et l'écrit dans un fichier texte de nom `CARTE.pvc` où `CARTE` désigne le nom du fichier contenant la carte. Le programme poursuit ainsi ses calculs tant qu'il n'est pas interrompu. Une fois que l'utilisateur considère qu'il a laissé suffisamment de temps de calcul au programme, il peut l'interrompre et lire une solution au problème du voyageur de commerce sur la sortie standard ou bien dans le fichier ainsi créé.

1.3 Programmation génétique

Expliquons à présent une manière efficace pour calculer une visite à partir d'une carte donnée. La première chose à observer est qu'il n'est, en pratique, pas possible d'essayer toutes les visites possibles d'une carte et de ne garder que la meilleure. En effet, une carte de taille 30 demanderait

$$30! = 30 \times 29 \times \cdots \times 2 \times 1 = 265252859812191058636308480000000$$

traitements de visites, ce qui est prohibitif. Nous n'allons donc pas chercher la meilleure solution mais seulement *une bonne solution*, compromis entre temps de calcul raisonnable et longueur faible de la visite.

Appelons C la carte sur laquelle nous travaillons et n le nombre de villes qu'elle contient. La méthode de recherche d'une bonne visite sur C se décrit de la manière suivante :

1. **générer** de manière aléatoire une liste $V = [v_1, \ldots, v_k]$ de visites sur C. L'entier k est fixé (une valeur habituelle pour k est de 128) ;
2. **trier** V dans l'ordre croissant selon la longueur des visites. De cette manière, la meilleure visite de V est celle située en tête. Elle se note v_1 ;
3. **construire** une nouvelle liste V' de visites de la manière suivante.

   Placer dans V' :

   1. les visites mut(v_1), mut(v_2), …, mut(v_α), où mut est une opération de mutation appliquée sur une visite et α est un entier strictement positif. Le fonctionnement de mut est expliqué plus loin ;
   2. les visites v_1, v_2, …, v_β provenant de V, où β est un entier strictement positif ;

3. un nombre γ de visites générées aléatoirement, où γ est un entier strictement positif.

   Les entiers strictement positifs α, β et γ sont fixés pour toute la durée du calcul et vérifient $\alpha + \beta + \gamma = k$. Ainsi, V' contient k visites, tout comme V ;

4. la **nouvelle liste** de visites V est la liste V' ;

5. **tant que** le programme n'est pas interrompu, aller à l'étape 2.

On peut ainsi espérer qu'à chaque itération du calcul, la meilleure visite contenue dans V possède une longueur de plus en plus petite. Cette liste V s'appelle la *population* et les visites qu'elle contient sont ses *individus*. Le schéma global consiste donc, à partir d'une population d'individus générés aléatoirement, à améliorer les individus petit à petit et de manière aléatoire. L'espoir est d'obtenir un individu qui forme une bonne solution au problème.

Décrivons à présent l'opérateur de *mutation* mut. L'idée consiste à modifier légèrement et aléatoirement une visite pour la rendre potentiellement meilleure. Une visite est vue ici comme une liste de villes. Ainsi, si v est une visite, $v = [a_1, \dots, a_n]$ où a_1, …, a_n sont les n villes de la carte. La visite $\mathrm{mut}(v)$ est la visite calculée par le procédé suivant :

1. générer aléatoirement un entier ℓ compris entre 1 et $\lfloor \frac{n}{2} \rfloor$;
2. générer aléatoirement deux entiers i et j (vérifiant des conditions de cohérence à déterminer) ;
3. remplacer dans v le facteur $a_i, a_{i+1}, \dots, a_{i+\ell-1}$ par le facteur $a_j, a_{j+1}, \dots, a_{j+\ell-1}$. Ces deux facteurs ne doivent pas se chevaucher. Cette nouvelle visite constitue la visite calculée par $\mathrm{mut}(v)$.

Par exemple, considérons la visite symbolisée par la permutation

$$\sigma := 4\mathbf{78}219\mathbf{365}.$$

Si les entiers $\ell = 3$, $i = 2$ et $j = 7$ ont été générés, on obtient

$$\mathrm{mut}(\sigma) = 4\mathbf{365}19\mathbf{782}.$$

On observe bien en effet que le facteur rouge et le facteur bleu de σ ont été échangés pour former $\mathrm{mut}(\sigma)$. Ces facteurs sont bien spécifiés par ℓ (la longueur), i (la position du début du facteur rouge) et j (la position du début du facteur bleu).

1.4 Précisions

Les paramètres α, β et γ doivent pouvoir être choisis par l'utilisateur (des valeurs par défaut sont proposées qui peuvent dépendre du nombre n de villes de la carte). Une phase d'expérimentation est importante pour trouver de bonnes valeurs par défaut.

Une gestion efficace d'erreur doit protéger l'application. Tout ce qui peut provoquer une erreur doit être anticipé (format de carte non valide, coordonnées de villes non cohérentes, *etc.*).

1.5 Variantes et améliorations optionnelles

Un respect de toutes les consignes précédentes (y compris celles figurant dans l'en-tête de cette fiche) vallent la note maximale si tout est parfaitement mis en œuvre. Cependant, pour ceux qui veulent aller plus loin, il est possible d'inclure un ou plusieurs points parmi les suivants pour améliorer l'application :

- la possibilité de saisir une carte en cliquant dans une fenêtre graphique de sorte à créer des coordonnées (x, y) pour chaque ville ;

- la connexion de l'application à un site internet contenant une base de données de distances entre villes ;
- la prise en compte, en plus des distances entre villes, d'un coût de transport. La minimisation devra se faire par une fonction linéaire $v \mapsto \lambda_d d + \lambda_c c$ sur la distance et le coût. Elle associe à chaque visite v la quantité mentionnée, où d est la longueur de la visite, c son coût total, et λ_d et λ_c des coefficients rationnels choisis par l'utilisateur ;
- la possibilité de spécifier pour certaines villes de la carte des priorités de visite (par exemple, la ville a_3 doit être visité avant la ville a_7, qui elle même doit être visitée avant la ville a_1).

2 Écriture du programme

Utiliser à présent toutes les techniques de réflexion, d'analyse et de gestion de projet étudiées dans les séances précédentes pour mener à bien la tâche qui vous est confiée.

Voici quelques conseils donnés sous forme d'étapes à franchir :

1. obtenir une bonne compréhension du problème du voyageur de commerce (indépendamment de l'approche génétique pour trouver une solution) ;
2. lister les structures de données et les types mis en œuvre dans la représentation d'une instance de ce problème (villes, cartes, visites, *etc.*). Commencer à écrire les modules correspondants ;
3. bien tester chaque fonction qui vient d'être écrite pour capturer les erreurs de programmation le plus tôt possible ;
4. faire en sorte qu'à chaque instant, le projet compile ;
5. obtenir une compréhension raisonnable de la programmation génétique en général (et non pas encore uniquement pour la résolution du problème du voyageur de commerce) et du vocabulaire qu'elle emploie ;
6. comprendre comment cette approche peut s'appliquer au problème du voyageur de commerce en étudiant comment les solutions (des visites, ou de manière équivalentes, des permutations) permettent de mettre en place l'opération de mutation ;
7. lister les nouveaux types et structures de données spécifiques à cette approche génétique ;
8. bien décomposer chaque opération à réaliser en étapes élémentaires et en fonctions ;
9. quand une fonction modifie une donnée passée en adresse, se demander si cette donnée n'est pas partagée ailleurs, auquel cas la modification peut ne pas être souhaitée et nécessite au préalable une recopie.

3 Suggestions

Voici quelques suggestions qui peuvent être mises en place immédiatement, sans même avoir compris comment fonctionne la programmation génétique.

- Écrire une fonction qui génère une ville aléatoirement (son nom peut être laissé vide ; il s'agit seulement de tirer au hasard ses coordonnées).
- Écrire une fonction qui génère une carte au hasard (attention à ne pas générer deux villes au même endroit).

- Écrire une fonction qui calcule la distance entre deux villes. En déduire une fonction qui calcule la longueur totale d'une visite.
- Écrire une fonction qui génère une visite de manière aléatoire (pour cela, il est possible de partir de la visite visitant les villes dans un ordre arbitraire, puis de réaliser un certain nombre de transpositions au hasard consistant à échanger des paires d'éléments).
- Écrire une fonction de résolution gloutonne qui consiste à créer une visite de la façon suivante. On part d'une ville au hasard et on choisi à chaque étape la ville non visitée la plus proche. Cette fonction servira à comparer les résultats fournis par la recherche génétique (cette dernière doit faire bien mieux que la méthode gloutonne si elle est bien implantée).
- Au lieu d'implanter directement la fonction de mutation mut conformément à ce qui est expliqué, commencer par implanter une version simplifiée mut′ consistant simplement à échanger aléatoirement deux éléments de la permutation.
- Concernant la partie graphique, toute bibliothèque libre et accessible est autorisée. Ceci inclut bibliothèque ncurses[5], la bibliothèque MLV[6], ou encore la bibliothèque SDL[7].

[5]Voir https://invisible-island.net/ncurses/.
[6]Voir http://www-igm.univ-mlv.fr/~boussica/mlv/index.html.
[7]Voir https://www.libsdl.org/.