Problème du voyageur de commerce

Bibi l'ibijau vient d'être embauché en tant que commercial dans une coopérative agricole. Chaque jour, il doit visiter les fermes de la région pour tenter de convaincre les kakapos exploitants de lui vendre leur production de graines. La région est grande, et il existe de nombreuses fermes. Chaque jour, il doit visiter n fermes et n'a pas de temps à perdre sur la route. Il doit donc trouver le circuit le plus court, partant de chez lui, visitant chacune des fermes, puis revenant chez lui.

1 Présentation et résolution du problème

Analysons le problème afin de connaitre son algorithme de résolution. Bibi travaille en tant que commercial : il va donc devoir prendre la voiture pour se déplacer de ferme en ferme. Son ami Jo l'a aidé : il a développé une bibliothèque complète de graphe permettant, entre autres, de chercher le plus court chemin entre deux sommets d'un graphe à l'aide de l'algorithme de Dijkstra. Mais cette simple bibliothèque est insuffisante, il faut trouver ici le circuit le plus court, qui comprend donc un ensemble de plus courts chemins.

Jo fournit également à Bibi un graphe représentant l'ensemble des routes de la région. Il a trouvé la carte originale sur Open Street Map et l'a transformé en graphe au format texte que sa bibliothèque peut lire facilement. Chaque sommet de ce graphe représente un point d'intersection entre deux routes. Le poids d'un arc correspond à la distance en mètres reliant les deux intersections voisines.

Ce graphe est cependant très volumineux et possède un très grand nombre de sommets. De plus, les routes et intersections ne sont pas toutes intéressantes. Puisque Jo voyage de ferme en ferme, il faut dans un premier temps identifier l'ensemble des plus courts chemins entre chaque ferme. Inutile en effet de chercher des routes passant par d'autres voies, Jo n'a pas le temps de visiter la région. Une fois ces routes identifiées, un sous-graphe complet formé uniquement des fermes en tant que sommets, et des routes les plus courtes entre ces fermes en tant qu'arc peut être formé. Il suffit d'appliquer un algorithme de recherche de tournée de distance minimale sur ce sous-graphe complet pour résoudre le problème de Bibi.

1.1 Le TSP

Le problème du voyageur de commerce (Travelling Salesman Problem) consiste à trouver la tournée la plus courte entre n sommets d'un graphe G. Formellement, considérons un graphe $G = (V, E)$ formé de n sommets. Sans perdre en généralité, ce graphe est considéré comme complet et non orienté. Le TSP consiste à choisir un sommet de départ, visiter chacun des $n-1$ autres sommets et revenir au point de départ. Le problème est de trouver le circuit le plus court réalisant ce parcours.

Si l'énoncé du problème est simple, sa résolution en est tout autrement. Pourtant, les applications sont nombreuses : tournées de bus dans les villes, ramassage des poubelles, mais aussi soudure de composants sur une carte électronique voir séquencage ADN avec

une version adaptée. Dans notre cas, nous travaillerons sur la problématique initiale : Bibi doit faire la tournée la plus courte entre des fermes en passant par des routes.

Comme de nombreux problèmes d'ordonnancement, la complexité naïve est en $O(n!)$, impossible donc de trouver la meilleure solution en testant toutes les combinaisons. Faisant partie de la classe des problèmes NP-complets, il n'existe pas d'algorithme générique trouvant la meilleure solution en temps polynomial. Aussi, il est nécessaire d'utiliser un algorithme de résolution approchée. Parce que nous aimons les animaux, nous allons utiliser des fourmis.

1.2 ACO

Il existe de nombreux algorithmes d'optimisation permettant de générer de bonnes solutions pour ce problème. Ces algorithmes sont appelés *méta-heuristiques*, car il s'agit d'algorithmes approchés (la solution n'est pas toujours optimale) pouvant être adaptés à un grand nombre de problèmes d'optimisation. Dans notre cas, et d'après les articles de recherche récents dans ce domaine, un des algorithmes les plus adaptés est l'ACO, pour *Ant Colony Optimization*. Cette méta-heuristique a même été développée spécifiquement pour traiter notre problème.

L'algorithme est né de l'observation de fourmis réelles recherchant de la nourriture. Chaque fourmi quitte le nid dans un but de trouver de la nourriture et retourne ensuite au nid. Lors de son déplacement, elle laisse derrière elle une trace (des *phéromones*) qui peut être trouvée et suivie par ses congénère. Cette trace disparait au cours du temps, ce qui fait que le chemin le plus court vers la nourriture sera naturellement privilégié par l'ensemble des fourmis.

Dans le cas algorithmique, le processus est itératif avec un nombre constant de fourmis à chaque itération. On initialise un graphe de phéromones, qui consiste à attribuer une probabilité de choix d'un arc sur chaque sommet. L'ensemble des fourmis réalise une tournée en fonction des phéromones présentes et de la longueur de chaque arc. Si une fourmi a trouvé un trajet plus court, ce trajet est conservé. Les phéromones sont alors mises à jour par évaporation puis par dépot pour chaque fourmi. Le trajet le plus court ayant été mis à jour à chaque itération, il suffit de le retourner en fin d'algorithme.

2 Système de coordonnées et format de données

Revenons sur les données fournies. Le graphe des routes est fourni sous la forme d'un fichier texte que l'on traitera avec la bibliothèque `Graph`. Parce que nous voulons fournir à Bibi une carte sur laquelle est dessiné l'itinéraire à suivre, il est nécessaire de connaitre les coordonnées GPS des différentes intersections. En plus du graphe des routes, un autre fichier texte est donné associant numéro de sommet de graphe et coordonnées géographiques. Étudions maintenant le fonctionnement du système de coordonnées utilisé pour ce TP et le format de sortie attendu.

2.1 Système de coordonnées

On peut repérer la position d'un point à la surface du globe terrestre à l'aide d'une paire de valeurs angulaires dont la référence de l'angle est le centre de la Terre :

- la *latitude* désigne le positionnement Nord / Sud : 0° pour l'équateur, angle positif vers le Nord, négatif vers le Sud.
- la *longitude* désigne le positionnement Est / Ouest : 0° pour le Méridien de Greenwich, angle positif pour l'est, négatif pour l'ouest.

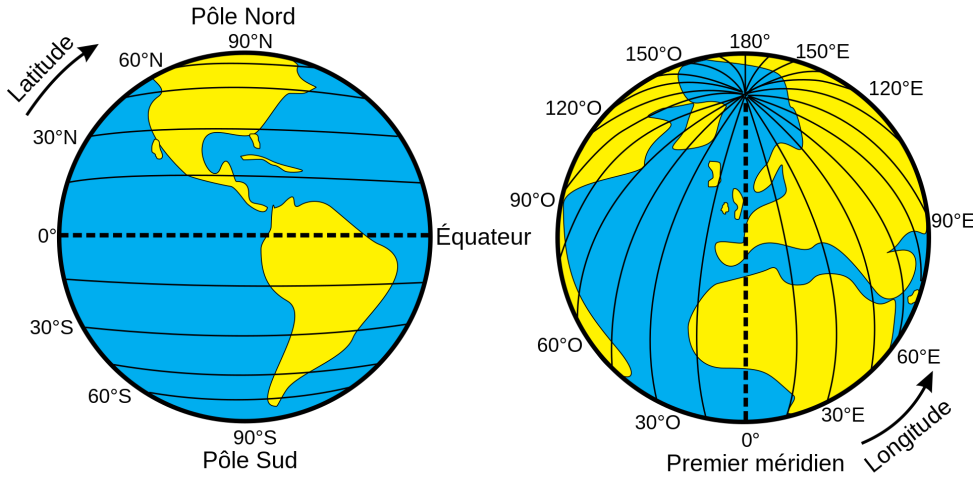

FIGURE 1 – Définition et plages de valeurs de la latitude et longitude.

Comme illustré [1] sur la figure 1, les valeurs extrêmes pour la latitude sont ±90° et celles de la longitude sont ±180°. Les sous-divisions de ces angles peuvent se faire en *minutes, secondes*, un degré étant divisé en 60 minutes et une minute divisée en 60 secondes. Une valeur d'angle intermédiaire peut également être donnée sous la forme de degré suivi d'un nombre décimal, format que nous allons utiliser.

Exemple 1. Les coordonnées du bâtiment de l'ESIEA à Laval sont 48.087658° en latitude, −0.756413° en longitude. On dit aussi 48.087658° Nord, 0.756413° Ouest. Le bâtiment voisin de l'Estaca étant plus à l'ouest, sa longitude est inférieure à celle de l'ESIEA de 0.0015°. ◢

2.2 Format de sortie

Le programme que vous allez réaliser correspond à un calcul d'itinéraires. La sortie de votre programme consiste donc en une série de coordonnées qui sont les points par lesquels passer. Il est possible de visualiser ce trajet dans des outils disponibles en ligne (par exemple le site umap) sous condition de respecter un certain format.

Le format GeoJSON sera privilégié : il permet de regrouper des portions de route simplement. Adapté du format JSON, il permet de stocker des ensembles de points ou segments selon des coordonnées géographiques. Il suffit donc d'écrire dans un fichier le bon entête, puis de rédiger la série de points qui constitue votre sortie pour créer un fichier GeoJSON valide. Les points sont regroupés au sein d'un même objet et reliés à l'aide de la clé `LineString`, comme indiqué dans l'exemple ci-après.

Exemple 2. Supposons que votre sortie soit formée de 4 coordonnées permettant de relier l'ESIEA au RU. Le fichier de sortie au format JSON devra être formé de la façon suivante.

1. Source : https://fr.wikipedia.org/wiki/Coordonnées_géographiques

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry":
      {
        "type": "LineString",
        "coordinates": [
          [-0.7563232475776288, 48.08764541472305],
          [-0.7570654336913334, 48.08727077115658],
          [-0.7569287808716526, 48.08668655157132],
          [-0.7571942777784609, 48.08664221320687]
        ]
      }
    }
  ]
}
```

Notez que chaque point est décrit d'abord selon la longitude, puis la latitude, contrairement à l'usage naturel. Il est alors possible de visualiser le trajet stocké dans ce fichier à l'aide de l'outil en ligne umap. Sur la droite, cliquez sur l'icône d'import des données, sélectionnez votre fichier et le format GeoJSON puis cliquez sur Importer.

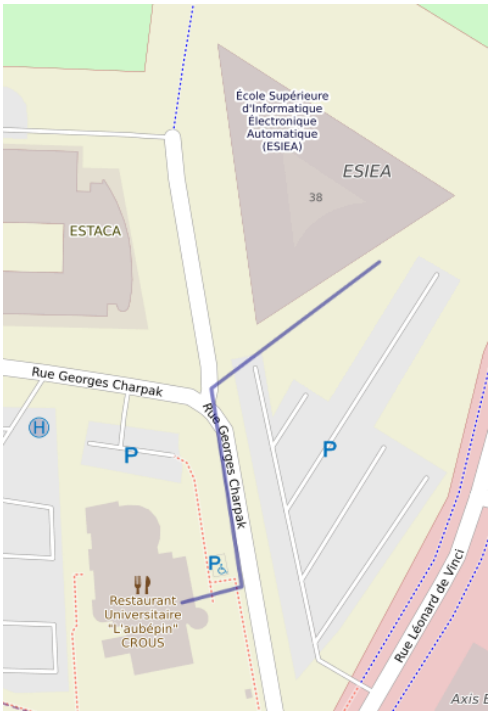

FIGURE 2 – Trajet entre l'ESIEA et le RU visualisé avec umap.

3 Extraction du sous-graphe et affichage

3.1 Tas binaire pour Dijkstra

Comme vu en cours, Dijkstra est un algorithme quadratique pour le temps d'exécution. En effet, chaque sommet du graphe est traité et le choix d'un nouveau sommet consiste à parcourir la liste des sommets non encore traités pour trouver celui de distance minimale. Cela implique une complexité linéaire pour la recherche du nœud suivant à traiter. L'implémentation d'un tas binaire pour construire une file de priorité permet de passer la recherche d'un nouveau nœud en complexité logarithmique. La complexité finale de Dijkstra serait alors en $O(n \log n)$, ce qui constitue un gain de temps primordial pour les calculs de longue distance.

L'intérêt d'une file de priorité est de pouvoir accéder à un élément ayant la plus grande (ou la plus petite) valeur en un temps inférieur au temps linéaire que prendrait une liste classique. Les temps en insertion ne doivent pas non plus dépasser la complexité linéaire. Un tas binaire est en réalité un arbre binaire parfait dont chaque nœud a une valeur inférieure ou égale à celle de ses deux enfants. L'accès à l'élément le plus petit (dans notre cas le nœud de distance minimale) est donc en temps constant puisqu'il s'agit de la racine de l'arbre.

L'implémentation de l'arbre binaire parfait est alors réalisée sous la forme d'un tableau et non d'un arbre tel que nous l'avons vu durant ce semestre. L'accès au dernier élément est donc constant puisqu'il suffit de retenir l'indice de cet élément. Cette propriété permet de construire des fonctions d'insertion et de suppression garantissant finalement une complexité logarithmique.

Todo (Tas binaire pour Dijkstra). `void Graph_dijkstra (...);`
Implémentez un tas binaire pour la gestion de la file de priorité de l'algorithme de Dijkstra en utilisant le lien Wikipédia donné plus haut. Votre algorithme doit alors pouvoir calculer un plus court chemin entre deux points quelconques des Pays de la Loire en moins d'une seconde, sans compter le temps de chargement du graphe. La fonction utilise un graphe `graph` et met à jour les tableaux `predecessors` et `distances` permettant plus tard de reconstruire le plus court chemin entre les sommets `start` et `end`.

Validation (Implémentation du tas binaire). Vérifiez le fonctionnement de votre code avec les tests `1_Dijkstra`.

3.2 Extraction de la matrice des chemins

La première étape de ce TP consiste à analyser le graphe initial au format texte pour en extraire le sous graphe complet des fermes à visiter. On dispose donc du graphe initial $G_0 = (V_0, E_0)$ dont les n_0 sommets représentent des intersection et les arcs des routes. On souhaite obtenir un graphe réduit $G = (V, E)$ formé des n sommets à visiter. Le poids des arcs reliant ces sommets représente la plus courte distance entre deux fermes. Effectuer ce traitement consiste donc à lancer l'algorithme de Dijkstra sur le graphe initial G_0 entre chaque paire de sommets du sous-graphe G.

Todo (Extraction du sous-graphe). A l'aide de la bibliothèque fournie, analysez les fichiers `graph_XXX.txt` et `coord_XXX.txt` pour extraire le sous-graphe des plus courts chemins entre les différents points à visiter. Commencez par désigner arbitrairement 3 points sur le graphe le plus petit et vérifiez votre extraction.

Todo (Affichage dans umap). En suivant l'exemple 2 donné précédemment, développez une fonction prenant en entrée un chemin sous la forme d'un `path` de coordonnées géographiques et produisant un fichier geojson interprétable par umap.

Validation (Implémentation de la matrice des chemins). Vérifiez le fonctionnement de votre code avec les tests `2_Path_Matrix`.

4 Calcul de la tournée

La section précédente nous a permis d'obtenir un graphe complet dont les sommets sont les fermes à visiter. Il est maintenant temps de savoir dans quel ordre parcourir les fermes pour minimiser la distance totale de la tournée. Rappelons que chaque ferme est reliée à une autre par un arc dont le poids est la distance minimale entre ces deux fermes. Aussi, nous disposons maintenant d'un graphe réduit complet $G = (V_E)$ formé de n sommets.

4.1 Création d'un algorithme glouton

On va commencer par un simple algorithme constructif pour établir une première tournée de qualité acceptable. On prend en entrée notre graphe réduit $G = (V, E)$, et notre point de départ s. On veut générer un chemin partant de s, passant une et une seule fois par tout les sommets de G, et revenant sur s, tout en minimisant la distance parcourue. Comme notre graphe G a été généré de manière à être complet, on peut utiliser l'algorithme suivant.

Algorithme 1 – Construction d'une tournée
Entrée. Un graphe réduit G et un sommet s
Sortie. Une tournée valide T

```
prev ← s
T ← s
Pour i allant de 1 à G.size-1 faire
    next ← argmin_{u∈V}(dist(prev, u), u ∉ T)
    T.ajout(next)
    prev ← next
Fin Pour
T.ajout(s)
Renvoyer T
```

On considère un graphe réduit à n sommets. On commence par ajouter s à notre tournée, car c'est notre point de départ. Ensuite, on ajoute les $n - 1$ autres sommets à la tournée de la manière suivante : à chaque itération, on sélectionne le sommet non exploré à distance minimum du dernier sommet inséré dans la tournée. Enfin, on ajoute de nouveau s à la tournée pour former une boucle : on revient à notre point de départ. La tournée produite sera donc une suite de sommets de la forme $(s, u_1, u_2, \ldots, u_{n-1}, s)$, traduisant l'ordre de visite des sommets du graphe.

Todo (TSP glouton). `Path* Graph_tspFromHeuristic (...);`
Développez cette fonction prenant en entrée le graphe réduit `graph` ainsi que l'identifiant de sommet de départ arrivée `station`. La fonction retourne une tournée obtenue avec l'algorithme glouton présenté précédemment.

Validation (Implémentation heuristique gloutonne). Vérifiez le fonctionnement de votre code avec les tests `3_TSP_Heuristic`.

4.2 Mise en place d'un ACO

La construction d'une tournée par une fourmi est très similaire au processus mis en place pour l'algorithme glouton. Lorsque la fourmi se trouve sur un sommet u, elle étudie tous les sommets non explorés pour choisir le suivant à atteindre. Là où l'algorithme glouton ne se basait que sur la distance pour faire son choix, la fourmi prend également en compte les **phéromones**. Cette quantité, déposée par les fourmis précédentes, va représenter l'intérêt de la colonie pour un arc du graphe.

Comme indiqué plus tôt, le processus est itératif et consiste à envoyer N fois k fourmis. Après chaque itération, on récupère le trajet le plus court parmi les k fourmis que l'on compare au meilleur trajet obtenu lors des itérations précédentes. Le graphe de phéromones est ensuite mis à jour par évaporation (les phéromones disparaissent au cours du temps) puis par dépôt (ce dernier étant plus important sur des trajets courts). L'algorithme général permettant l'optimisation du problème est donc le suivant.

Algorithme 2 – `T = ACO(`G`,`s`,`α`,`β`,`ρ`,`Q`,`n`,`k`)`
Entrées. `Un graphe réduit` G `et un sommet` s
`les paramètres` α`,` β`,` ρ `et` Q
`les paramètres` n `et` k
Sortie. `Une tournée valide` T

```
T ← ∅
P ← un graphe de phéromones
Pour i allant de 1 à n faire
    Pour j allant de 1 à k faire
        T_j ← génération d'une tournée
        Si T est vide ou T_j plus courte que T alors
            T ← T_j
        Fin Si
    Fin Pour
    Evaporation de P
    Pour j allant de 1 à k faire
        Dépôt de phéromones en fonction de T_j
    Fin Pour
Fin Pour
Renvoyer T
```

Todo (Fonction maitresse). `Path* Graph_tspFromACO (...);`
L'objectif de la section qui suit est de compléter cette fonction déroulant un ACO sur un graphe `distances` afin d'extraire une tournée de distance minimale à partir du sommet `station` du graphe. Nous détaillerons les paramètres au fur et à mesure de cette section. Vous complèterez donc cette fonction une fois que toutes les fonctions suivantes seront réalisées.

4.2.a Choix d'un arc

Lors de chaque itération, chaque fourmi doit parcourir le graphe pour générer une tournée. Sur chaque sommet, elle doit donc choisir quel sera le sommet suivant à visiter en fonction de deux variables. Plus la quantité de phéromones sur un arc est importante, plus la fourmi va être tentée de prendre ce chemin. De la même manière, plus la distance d'un arc est faible, plus la fourmi va s'y intéresser. On définira donc, pour tout arc (u, v), une probabilité $p_{u,v}$ de choisir cet arc :

$$p_{u,v} = \begin{cases} \frac{(\tau_{u,v})^\alpha \times (\frac{1}{d(u,v)})^\beta}{\sum_{w \in N} (\tau_{u,w})^\alpha \times (\frac{1}{d(u,w)})^\beta} & \text{si } v \text{ n'a pas encore été exploré} \\ 0 & \text{sinon ;} \end{cases}$$

avec $\tau_{u,v}$ et $d(u, v)$ représentant respectivement le taux de phéromones et la distance associés à l'arc (u, v), et α et β deux réels permettant de pondérer l'intérêt à l'une ou l'autre des composantes de la décision.

> **Todo** (Probabilité de choix d'un arc). `float* Graph_acoGetProbabilities (...);`
> Dans votre code, vous implémenterez la fonction demandée ayant pour paramètres :
>
> - le graphe complet réduit `Graph *distances`,
> - un graphe complet `Graph *pheromones` contenant les taux de phéromones,
> - le sommet `int start` sur lequel la fourmi se trouve,
> - un tableau `bool *inserted` permettant de savoir si un sommet a déjà été visité,
> - les paramètres `alpha` et `beta` de type `float` pour le calcul des probabilités.
>
> Cette fonction renvoie un tableau de n réels (avec un graphe à n sommets), la case i contenant la probabilité de voyager vers le i–ème sommet du graphe depuis `start`.

4.2.b Calcul d'une tournée

Ensuite, il vous faudra développer une fonction permettant de construire une tournée en fonction du graphe réduit, du graphe des phéromones, et des paramètres α et β. D'un point vue algorithmique, on est très proche de l'algorithme glouton. On commence par ajouter s à notre tournée, car c'est notre point de départ. Ensuite, on ajoute les $n-1$ autres sommets à la tournée de la manière suivante : à chaque itération, on sélectionne **aléatoirement** un sommet en prenant compte des probabilités calculés via notre précédente fonction. Enfin, on ajoute de nouveau s à la tournée pour former une boucle.

> **Todo** (Calcul d'une tournée). `Path* Graph_acoConstructPath (...);`
> Cette fonction construit une tournée dans le graphe `distances` depuis le sommet de départ `station` pour une fourmi. Elle s'appuie sur le graphe de `pheromones` et les paramètres `alpha` et `beta` pour sélectionner aléatoirement son chemin grâce à la fonction précédemment développée.

4.2.c Mise à jour du graphe de phéromones

Focalisons nous maintenant sur les phéromones. A l'initialisation de l'algorithme, chaque arc reçoit une quantité de 1. Ensuite, à chaque itération, deux mécaniques se produisent : (1) l'évaporation et (2) le dépôt. L'évaporation permet d'oublier progressivement

les chemins traversées par les fourmis, pour éviter de rester bloqué dans un optimum local. Le deuxième mécanisme consiste à marquer l'ensemble des arcs d'une tournée construite par une fourmi en fonction de la qualité du résultat, et donc donner à chaque arc traversé une vision plus globale : quel résultat final peut-on attendre en empruntant cet arc ? La mise à jour des phéromones pour un arc (u, v) se fera via la formule suivante :

$$\tau_{u,v} = (1 - \rho) \times \tau_{u,v} + \sum_{p \in P(u,v)} \frac{Q}{dist(p)},$$

avec ρ et Q deux paramètres, et $P(u, v)$ l'ensemble des tournées générées pendant l'itération courante, et contenant l'arc (u, v)

Todo (Mise à jour du graphe). Pour des raisons de clarté du code, vous devrez développer deux fonctions séparées :

```
void Graph_acoPheromoneGlobalUpdate (...);
void Graph_acoPheromoneUpdatePath (...);
```

La première se chargera de procéder à l'évaporation des phéromones présentes sur les arcs du graphe de `pheromones` en fonction du paramètre `rho`. La seconde se chargera de déposer les phéromones d'une fourmi sur le graphe de `pheromones` en fonction du chemin réalisé `path` et du paramètre `Q`, permettant de définir les arcs à altérer et la quantité de phéromones à déposer.

Validation (Implémentation de l'ACO). Vérifiez le fonctionnement de votre code avec les tests `4_TSP_ACO`.

5 Condition de rendu et barème

Attention

L'usage d'IA générative (ChatGPT, Copilot etc.) est interdite.

Votre programme doit être rendu sur Moodle dans une archive au format ZIP sous le format `NORRIS_VANDAMME.zip` (nom des binômes dans l'ordre alphabétique) avant vendredi 24 mai, 23h55. Votre programme doit être compatible Linux et être exempt de fuites ou erreurs mémoire (hors SDL). Lors de l'exécution, votre projet ne doit pas consommer plus de 8Go de mémoire vive et s'exécuter dans un temps raisonnable.

Votre archive doit obligatoirement contenir un `README.txt` détaillant l'usage de votre application et l'ensemble des fonctionnalités implémentées. Votre README doit, au minimum, comporter les informations suivantes :

- Membres du projet avec un détail de qui a fait quoi
- Fonctionnalités + temps d'exécution et mémoire consommée
- Commandes de compilation et exécution sous Linux

Respectez l'arborescence suivante pour l'exécution de votre projet :

```
Data/
NOM1_NOM2/
|_.sln
|_ProjectSRC/
|_PersonalData/
|_README
```

Afin de vérifier que vous êtes bien l'auteur du code que vous avez rendu, une soutenance par binôme aura lieu le jeudi 23 ou vendredi 24 mai. Le README doit être rendu avant la soutenance finale. Lors de cette soutenance, vous serez emmené à détailler certaines parties de votre code. Selon vos réponses, un facteur multiplicatif entre 0 et 1 s'appliquera à votre note de projet. Ce facteur multiplicatif est individuel.

La note finale N est donc calculée de la façon suivante : $c \times n - m$, avec c le coefficient individuel de soutenance, n la valeur de votre travail de TP sur 20 et m une valeur de malus pour toute consigne non respectée (entre 0 et 5).