Sujet TD&P n°5 - Séances 7 & 8

Réplications d'un Algorithme Glouton *Randomisé*

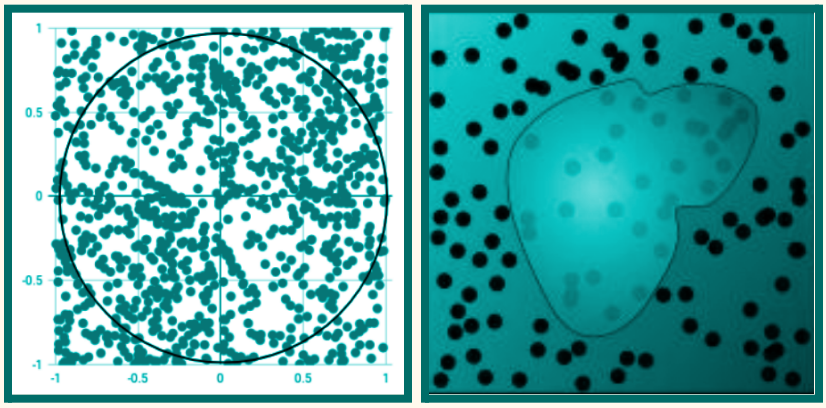

Figure 1 : 2 résultats d'algorithmes *randomisés*[1] criblés de points aléatoires pour respectivement l'approximation de π et la surface d'un lac.

Objectifs du sujet.

- Comprendre la puissance des algorithmes *randomisés* répliqués en commençant par les termes de cette phrase,
- Résoudre 2 problèmes d'optimisation dans un problème global,
- Intégrer différentes situations (entrées du problème) à partir de fichiers,
- Manipuler des graines de générateur de nombres pseudo-aléatoires pour obtenir des solutions différentes,
- Accorder une grande liberté[2] sur les différents choix (structures de données et choix algorithmiques).

Mots clés.

Problème du sac à dos ; problème du voyageur de commerce ; algorithmes randomisés ; réplications, scripts.

[1] Ici, l'algorithme *randomisé* est défini comme étant de "Monte Carlo". En effet les limites des points aléatoires sont bornées (ici : les limites des deux carrés) et le résultat est approché (on dit alors que c'est une méthode heuristique). Il n'est question ici que d'une seule réplication du programme, alors que plusieurs d'entre elles auraient donné des approximations différentes de π et de la surface d'un lac.

[2] En comparaison avec les autres sujets où les étapes sont beaucoup plus détaillées.

Samuel Deleplanque. (prénom.nom@junia.com)

Mise en situation

Imaginez que vous venez d'intégrer une société de pains au chocolat belge[3] afin d'optimiser le processus qui consiste à remplir leurs distributeurs[4] qui se trouvent aux 4 coins des Hauts-de-France. Chaque jour, vous voulez distribuer en une seule fois des produits frais locaux dans un certain nombre de distributeurs de la région Hauts-de-France. Chaque matin avant de partir, vous recevez la liste des villes avec des distributeurs et l'inventaire des produits candidats à la livraison[5]. Ainsi, chaque matin, vous n'avez pas beaucoup de temps pour réaliser les deux opérations suivantes :

- sélectionner la liste de produits à transporter en fonction de la capacité de votre véhicule électrique (opération notée comme le problème **(P1)**),
- tracer le plan de route en ordonnant les villes à visiter (opération notée comme le problème **(P2)**).

Les solutions de (P1) et (P2) suivent un objectif distinct, soit respectivement :

- maximiser le bénéfice alors que chaque produit consomme une certaine capacité de votre véhicule qui n'est pas proportionnelle à leur bénéfice[6],
- minimiser les coûts, i.e., la quantité d'électricité consommée par les véhicules et nécessaire à la tournée est à minimiser).

Si on s'en tient à la littérature scientifique[7], ces deux problèmes d'optimisation sont bien connus, il s'agit :

- du problème du sac à dos (eq. (P1)),
- du problème du voyageur de commerce (i.e. (P2), nous avons déjà traité de ce problème).

Les différentes figures de ce document devraient permettre de mieux cerner la situation.

---

[3] Si nécessaire, vous pourrez aussi considérer les couques au chocolat et même les chocolatines.
[4] Nous allons voir une version simplifiée de la problématique d'une telle entreprise qui se pose dans un grand nombre de situations, et ceci à différentes échelles : de la leur à celle d'*Amazon*, *Cdiscount*, *la Poste*, où même la livraison de courses.
[5] Les produits n'ont pas de lien avec telle ou telle adresse de livraison.
[6] On considère ici que tous les produits acheminés seront vendus et qu'importe l'adresse de livraison (hors bonus).
[7] Pour les plus curieux :
https://scholar.google.com/scholar?hl=fr&as_sdt=0%2C5&q=Knapsack+problem&btnG= (P1) et
https://scholar.google.com/scholar?hl=fr&as_sdt=0%2C5&q=traveling+salesman+problem&btnG= (P2).

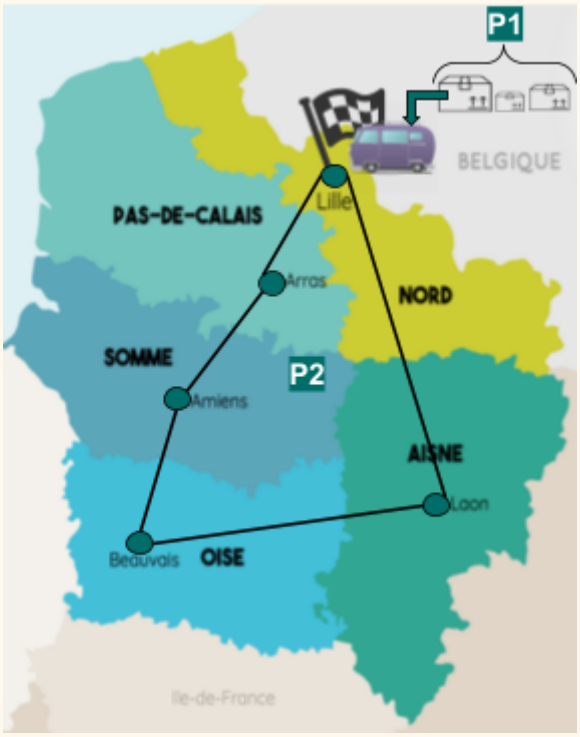

Figure 3 : Problèmes P1 et P2 : exemple de 3 produits sélectionnés et d'une tournée de 5 villes dont Lille[8] est la ville de départ.

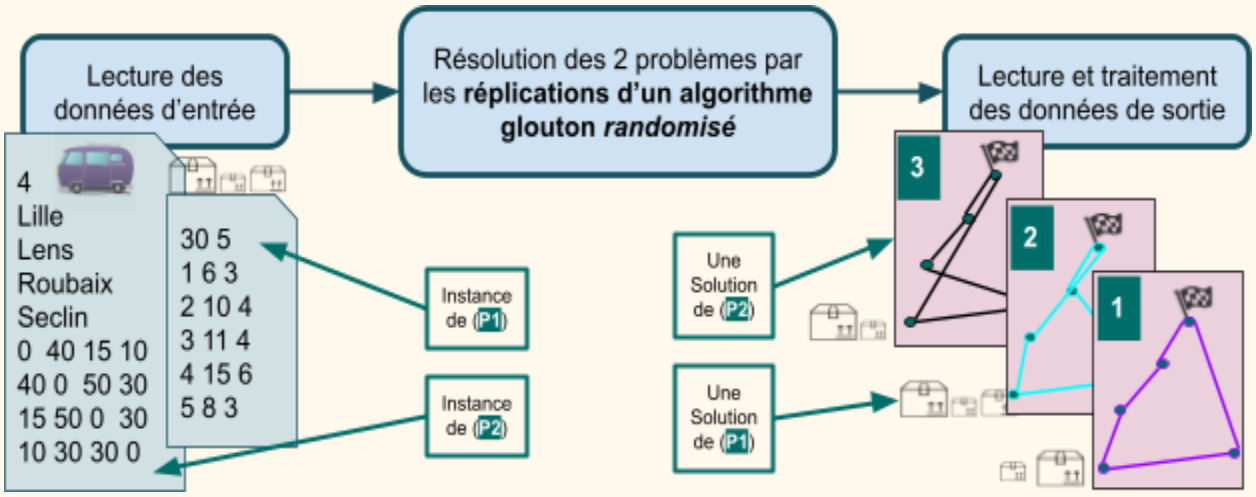

Figure 4 : Processus général du traitement des données d'entrée (INPUT) jusqu'aux données de sortie (OUTPUT). Ici, trois réplications ont été exécutées ce qui permet d'obtenir trois solutions générales (3 de (P1) et 3 de (P2)).

[8] Ref. pic.: https://www.touteleurope.eu/l-europe-en-region/l-europe-en-region-les-hauts-de-france/.

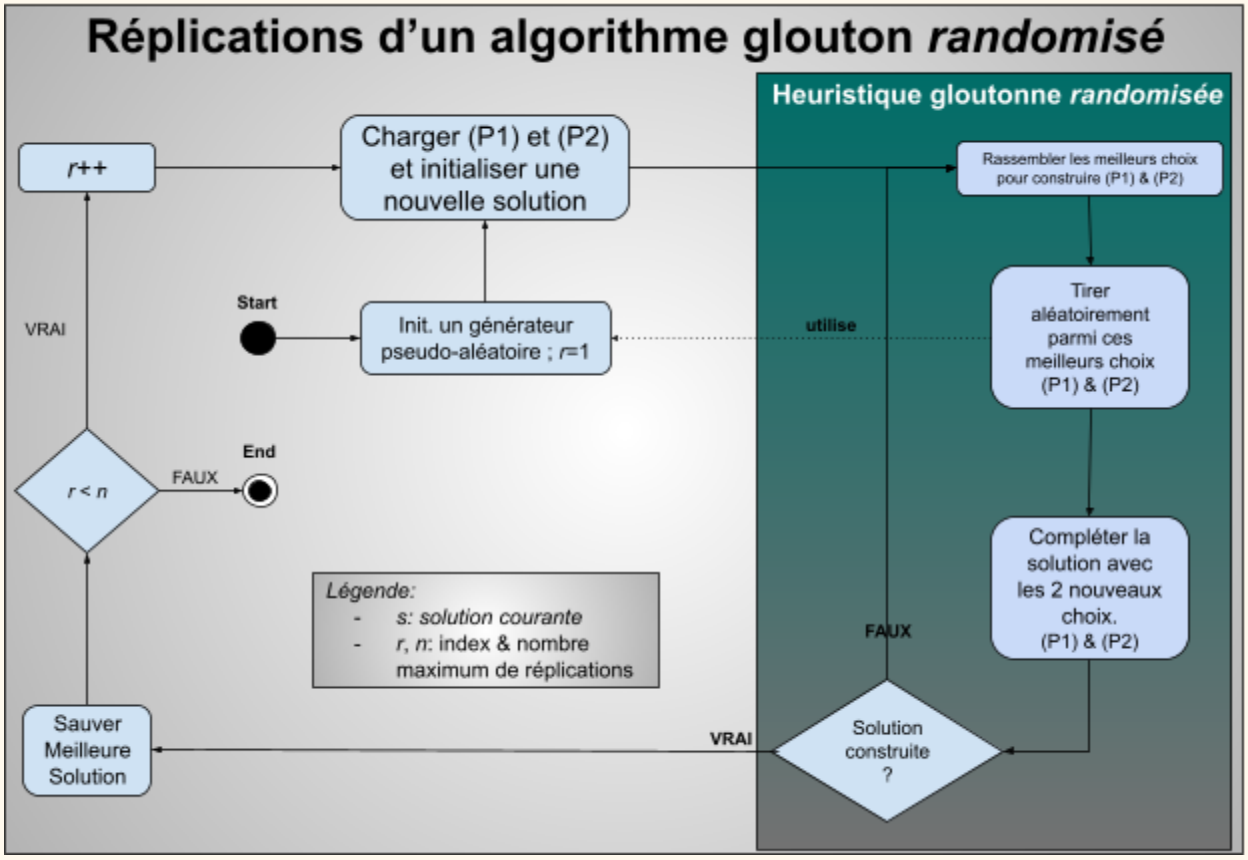

Figure 5 : Représentation des réplications de l'algorithme : choix aléatoires lors de la construction et la sauvegarde de la meilleure solution courante[9]. Nous verrons par la suite que ce schéma pourra se faire en parallélisant ou bien simplement en lançant chaque réplication comme autant d'appels à votre programme. Dans ce schéma, le même générateur de nombres pseudo-aléatoires est utilisé tout le le long des réplications. A l'inverse, si le processus articulant les réplications consiste à lancer le programme plusieurs fois (e.g., à partir d'un script) on cherchera à initialiser différemment le générateur à chaque fois (e.g., `srand(nouvelleGraine)`) . Ceci sera fonction de votre choix entre des réplications d'un même programme lancé plusieurs fois par un script ou l'articulation de ces réplications directement incluse dans un unique programme utilisant alors les *threads* du C++.

[9] "Pour la culture" : ce schéma est largement inspiré des diagrammes d'activité UML qui permet de modéliser graphiquement différents systèmes, processus et algorithmes.

Données d'entrée: villes et produits renseignés par fichiers de type texte

Dans ce travail, les situations existent et vous n'avez pas d'emblée à les générer. Vous récupérez la liste des villes et des paquets au travers de fichiers `.txt` qu'il faudra lire pour remplir vos différentes structures de données.

Lecture des fichiers de (P1).

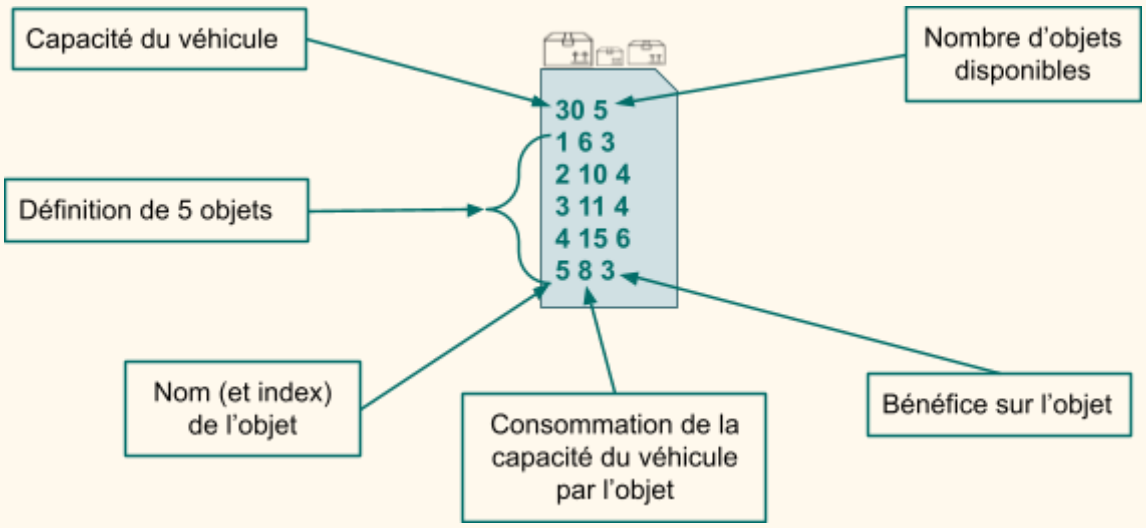

Figure 6 : Définition des éléments des fichiers de (P1). Ici on apprend de la première ligne qu'il y a 5 lignes "objet" à lire.

Lecture des fichiers de (P2).

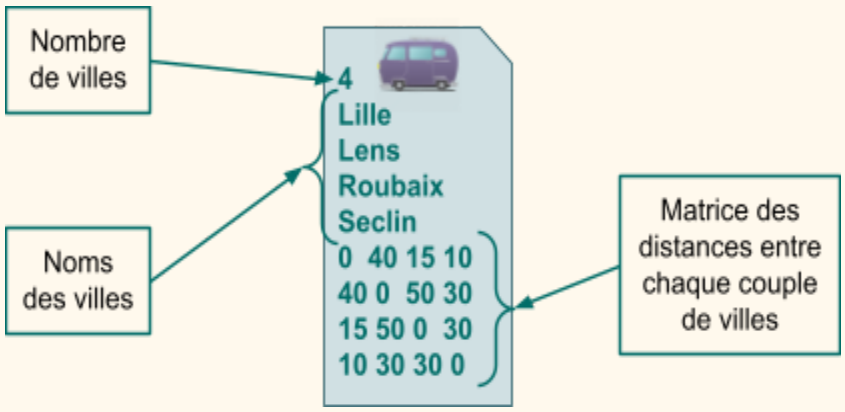

Figure 7 : Définition des éléments des fichiers de (P2). Dans la première ligne on apprend qu'il y a aura 4 villes et donc une matrice 4^2 à lire. Les "distances" sont en réalité les consommations d'énergie électrique associées aux différents déplacements possibles.

Chaque situation à traiter correspond donc à deux problèmes dont chacun trouve ses données dans son propre fichier. Pour l'exemple ci-dessus, les deux fichiers prennent les deux noms suivants :

- 5colis30capacite.txt,
- 4villes.txt.

Vous trouverez l'ensemble des fichiers ici[10]. L'idée est de fournir le nom de ces deux fichiers comme paramètres d'exécution de votre programme. Ainsi votre programme n'est pas dépendant de tel ou tel fichier.

Réalisation.

- *Q1.* **Algo**. Commencer par réfléchir aux structures de données à même d'enregistrer l'ensemble des données des deux problèmes (P1) et (P2).
- *Q2.* **c++**. Développer le code de lecture des deux fichiers de type texte et de remplissage des structures de données choisies au préalable. Le nom des fichiers sont alors deux paramètres d'exécution[11].

TECHNIQUES ALGORITHMIQUES

Vous aurez compris que (P1) et (P2) sont deux problèmes d'optimisation à part entière. Pour les résoudre, nous n'avons pas beaucoup de temps de calcul de disponible, et c'est donc une méthode dite 'approchée' (i.e., une heuristique) qui sera développée afin d'obtenir rapidement une solution tout en essayant qu'elle soit de bonne qualité. A ce propos, les algorithmes gloutons sont très généralement des heuristiques, et ce sera le cas ici.

[10] Au cas où le lien ne fonctionne pas, copier l'adresse suivante dans votre navigateur : https://yncrea-my.sharepoint.com/:f:/g/personal/samuel_deleplanque_yncrea_fr/El3r4xXK98REvLqL49PPO2AB__-9cl3UQQON_rvF-zJk7g?e=q3dsut

[11] Il faudra trouver le moyen de les ajouter si vous utilisez un IDE. Dans Visual Studio, vous pouvez essayer à partir de ce tutoriel : https://dailydotnettips.com/how-to-pass-command-line-arguments-using-visual-studio/.

1. Algorithmes *Gloutons*

Principe. Un algorithme glouton[12] va construire une solution au problème, élément après élément, et sans revenir sur le choix de ces derniers. Par exemple, lors d'une réplication de votre algorithme, les produits entrant dans le camion de livraison seront ajoutés un à un de manière définitive et il ne sera pas possible de revenir dessus (P1). Idem pour la construction de la tournée qui est réalisée ville après ville et sans possibilité d'en modifier leur ordre (P2).

Une seconde caractéristique d'un algorithme glouton est le fait d'ajouter à chaque étape un nouvel élément selon le meilleur choix possible. Ainsi, à chaque étape de la résolution de (P1), on pourra insérer dans le véhicule le produit ayant le meilleur rapport "bénéfice" / "capacité consommée" parmi les produits restants. Pour (P2), on pourra insérer, en fin de la liste de la tournée "en construction", la ville la plus proche[13] de la dernière ville qui venait d'être insérée à l'itération précédente.

Réalisation.

- *Q3.* **Algo**. Réfléchir aux structures de données constituant une solution[14], soit au travers de 2 méthodes, une pour résoudre (P1), une autre pour résoudre (P2). Pour chacun des deux problèmes, le processus devra ajouter des éléments à votre solution (resp. un nouveau produit, et une nouvelle ville) **Tant Que** la solution n'est pas entièrement construite, soit :
  - (P1) : **Tant Qu**'il reste de la capacité (suffisante) dans le véhicule,
  - (P2) : **Tant Qu**'il reste des villes à traverser[15].
- *Q4.* **c++**. Vous êtes entièrement libres du développement de votre méthode de résolution. Pensez à bien afficher les résultats obtenus pour chacun des problèmes ainsi que la composition de la solution trouvée à (P1) et de celle de (P2) mais aussi le bénéfice et l'énergie consommée.

[12] *greedy algorithm* en anglais
[13] i.e., dont le déplacement consomme le moins d'énergie.
[14] Pour le problème (P2) vous pouvez par exemple vous baser sur vos travaux précédents sur le problème du voyageur de commerce.
[15] On pourrait ici ajouter une ville à la fin de la tournée courante, en la choisissant selon celle qui fera augmenter le moins possible l'énergie totale consommée.

2. Algorithmes Gloutons *Randomisés*

Principe. L'heuristique attendue est un algorithme *randomisé*[16] où la création d'une solution se fera à l'aide de la génération de nombres aléatoires pour décider aléatoirement, parmi un ensemble de candidats, quels sont les éléments à intégrer à la solution à un moment donné de sa construction. Par exemple, en cours de l'élaboration d'une solution et pour chacun des problèmes (P1) et (P2), on choisit par un tirage aléatoire une nouvelle ville ou un nouveau produit respectivement :

- parmi les deux villes les plus proches[17] de la dernière insérée dans la solution,
- parmi les deux produits qui offrent le plus grand rapport "bénéfice"/"capacité consommée" et qui ne dépassent pas la capacité restante. Vous pourrez chercher d'autres manières de faire en les justifiant.

Réalisation.

- *Q5.* **c++**. Adaptez votre code à ces deux choix aléatoires parmi les deux paires de meilleurs candidats. Tester en changeant la graine (*eng.: seed*) du générateur afin de vérifier l'obtention de solutions différentes.

[16] Ils sont également appelés 'algorithmes probabilistes'.
[17] i.e., où l'énergie à consommer pour arriver à la seconde en partant de la première est minimale.

3. *Réplications* d'Algorithmes Gloutons Randomisés

Principe. Il est clair que l'intérêt d'avoir un tel algorithme *randomisé* apparaît dès lors qu'on l'exécute plusieurs fois. Ainsi, si un algorithme déterministe construit sa solution en faisant toujours les meilleurs choix locaux (e.g., en ajoutant la ville qui est strictement la plus proche à chaque étape), on obtiendrait toujours la même solution[18]. Avec de multiples exécutions de notre algorithme glouton *randomisé*, nous obtenons plusieurs bonnes solutions différentes[19].

Le nombre d'appels de notre algorithme *randomisé* équivaut au nombre de réplications de celui-ci. Plus on en fait, plus on obtient de solutions, et donc, plus on augmente nos chances d'en obtenir une très bonne, voire la meilleure.

Ces méthodes sont particulièrement efficaces du fait du caractère indépendant de chaque réplication **à condition d'utiliser une initialisation du générateur différente** soit d'avoir une graine différente pour chaque réplication. Ainsi, il est tout à fait envisageable de paralléliser l'ensemble de ces réplications et de trier les résultats en dynamique ou à la fin du processus.

Réalisation.

- *Q6.* **Question**. Montrer par un petit exemple[20] dessiné en 2D, en détaillant chaque étape, et en partant d'une ville particulière, qu'il n'est pas systématique d'obtenir la tournée la plus courte en prenant uniquement la meilleure ville candidate à chaque étape de la construction (i.e., la plus proche de la dernière ville insérée). On s'arrêtera pour cette question sur le problème du voyageur de commerce, mais l'équivalent pour le sac à dos peut-être fait en bonus.

- *Q7.* **Question**. Pour obtenir la meilleure solution, vaut-il mieux directement trier les solutions de chaque réplication de votre programme ou bien reconstituer une solution à partir des meilleurs résultats de (P1) et (P2) pris séparément pour chaque réplication ?

[18] Cette solution pourra être une bonne solution mais il n'est pas du tout sûr qu'elle sera la meilleure.
[19] Il y a donc une solution qui sera meilleure (ou à égalité). Selon le nombre de solutions obtenues, celle-ci devrait rapidement battre la seule obtenue par l'algorithme déterministe.
[20] A vous d'en trouver un le plus petit possible.

Remarque importante. Le travail ci-dessous indique la marche à suivre la plus simple pour réaliser plusieurs réplications de votre algorithme[21] à partir de simples scripts et afin d'obtenir plusieurs solutions différentes. Vous pouvez substituer cette partie par un travail plus complexe mais amenant à un programme plus efficace en utilisant exclusivement du C++ et la gestion du *multhreading*. Vous pouvez également utiliser du python à la place des scripts classiques présentés plus bas. Si vous décidez de partir sur une de ces 2 alternatives, vous êtes entièrement libres de la marche à suivre, mais vous devez l'expliquer clairement dans votre rapport en ajoutant les actions particulières à réaliser pour la compilation le cas échéant.

- *Q8.* **c++**. Adaptez votre code pour le faire correspondre au fonctionnement suivant :
  - En séquentiel ou en parallèle, les réplications seront exécutées de manière indépendante. Pour les différencier, une graine de générateur deviendra paramètre d'une réplication accompagnée du nom des deux fichiers des problèmes (P1) et (P2). On utilise ici un script (e.g., script Powershell sous Windows ou encore script shell bash sous Linux) dont chaque instruction correspond à l'appel d'une réplication de votre programme accompagné d'une graine[22] et de 2 noms de fichiers. Les appels seront alors consécutifs, et les sorties consoles des réplications seront enregistrées dans un fichier. Ces sorties pourront rassembler :
    - le numéro de la réplication (qui peut être déterminé par la graine),
    - le bénéfice attendu pour les produits sélectionnés (résultat 1),
    - la consommation totale de la tournée (résultat 2),
    - un calcul pondéré réalisé à partir des 2 premiers. A vous de choisir la pondération selon vos envies : vous pouvez ainsi privilégier le bénéfice ou une faible consommation d'énergie.

[21] Vous avez peut-être déjà compris ici que ce sera grâce à des graines différentes utilisées pour initialiser le générateur de nombre.
[22] Si le nombre de réplications, i.e. le nombre d'appels à votre algorithme dont chacun est accompagné d'une graine, est important, vous en aurez vite marre de faire des 'copier-coller' du nom de votre programme. Vous pouvez alors développer un tout nouveau programme qui servira exclusivement à l'écriture de tout le script grâce à une boucle Pour/For qui remplacera ces 'copier-coller'.

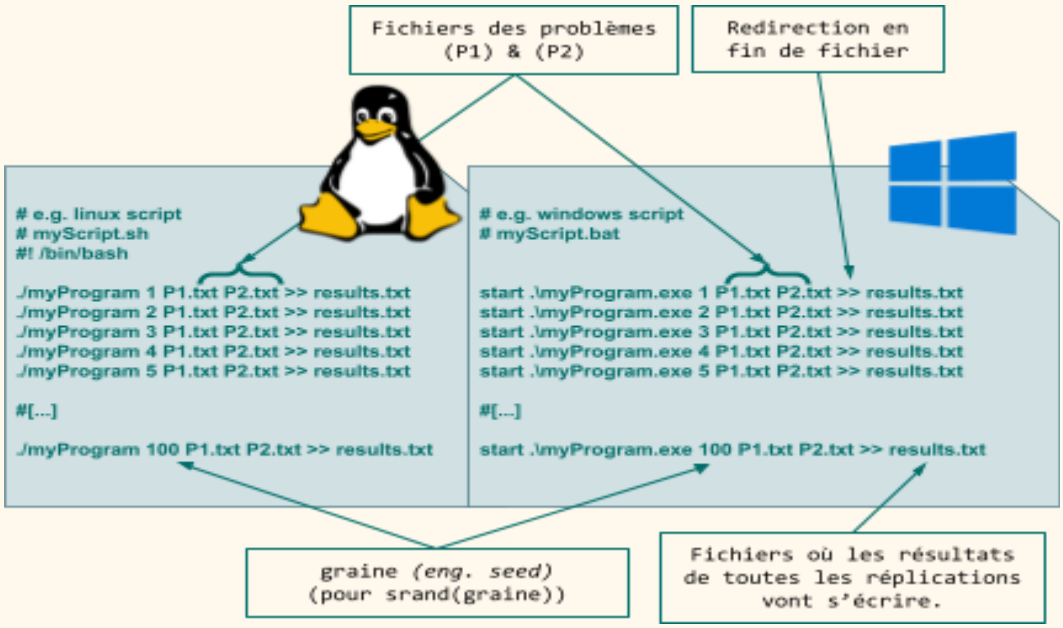

Figure 8 : Exemple de scripts Linux et Windows[23]

Pour aller plus loin[24], on trouvera un moyen de mettre ces réplications en concurrence ou en parallèle.

4. Heuristique 2 en 1

Principe. Les deux problèmes d'optimisation sont ici résolus consécutivement dans une même méthode. Mais il serait tout à fait possible de séparer les problèmes en deux méthodes puisque les solutions d'un des deux problèmes n'a pas d'impact sur l'autre puisqu'on ne considére pas l'affectation d'un paquet vers adresse[25].

Réalisation.

- *Q9.* **c++**. Vous êtes libre de procéder à la récupération des solutions de (P1) et de (P2) obtenues après toutes les réplications afin de construire une nouvelle solution. Cette dernière va donc se former par la meilleure solution obtenue de (P1) puis la meilleure solution obtenue de (P2). Vous êtes maîtres de tous les changements à réaliser dans la structure de votre système de résolution.

[23] Cet exemple est juste une ébauche, + ou - juste, libre à vous de rechercher comment réaliser un tel script sur le même type de système que celui utilisé pour compiler votre programme.
[24] Même si on y est déjà !
[25] Ceci au risque de ne plus avoir de paquet à livrer. Si cela est trop effrayant on pourra considérer qu'en plus de la livraison, la visite d'une adresse pourra être faite aussi dans le but de récupérer les pains au chocolat qui ont pris un coup de vieux.

5. Résultats

Réalisation.

- *Q10.* **c++**. A vous de générer des nouveaux cas : générez vous-même les deux fichiers d'instance des problèmes (P1) et (P2). Pour cela, à vous d'utiliser encore la génération de nombres pseudo-aléatoires et surtout réfléchissez bien aux limites à fixer pour créer des situations relativement sensées. Vous pouvez intégrer vos travaux sur le sujet précédent pour ce qui concerne le problème (P2).

- *Q11.* **c++**. En intégrant des cas du problème du voyageur de commerce du sujet précédent, essayez d'évaluer la qualité des résultats obtenus ici avec votre algorithme de résolution du problème (P2) pour des instances qui peuvent être résolues par la méthode exacte de type brute force. Reportez dans un tableau le gap, i.e., la distance en pourcentage entre la solution optimale (brute force) et ce que vous obtenez ici avec votre heuristique.

- *Q12.* **c++**. En générant des situations de taille importante (grand nombre de villes et grand nombre d'éléments potentiellement transportables), réaliser des *benchmarks* de façon à tester la puissance de votre ordinateur. Vous êtes libre de choisir la manière adéquate pour présenter vos résultats en reportant les informations qui vous paraissent importantes.

6. Rapport & Code

Votre **code**[26] commenté sera accompagné d'un rapport de 5 pages maximum au format **.pdf**. En plus d'instructions pour compiler et exécuter votre code, motivez vos choix avec par exemple des **explications** sur les structures de données et algorithmes mis en place. Toutes les réponses aux questions doivent être présentes dans le rapport même si ces justifications peuvent être un copier-coller du commentaire C++ provenant directement de votre code. Vous pouvez inclure dans votre archive différents fichiers d'instances. Un effort sur l'orthographe sera vivement apprécié.

Bonus. Exclusivement pour ce TD&P noté: les bonus seront directement ajoutés au rapport en plus du code (et non envoyés par *Teams* ou par *emails*). Les explications liées aux bonus ne sont pas limitées en nombre de pages dans votre rapport.

Le tout (code et rapport) sera *zippé* ou *targzé* dans un fichier "**NumeroEquipe.[..].CIR2.zip**" ou .**targz**, le numéro de votre équipe a été donnée avec la composition de celle-ci. La soumission du devoir se fera à travers Microsoft Teams.

[26] Ici seuls les fichiers sources sont attendus (.cpp, hpp/h).

7. Bonus[27]

N et M candidats. Alors que le nombre de candidats à la construction de la solution pour les problèmes (P1) et (P2) est fixé à 2 pour chacun d'entre eux, essayez de faire varier ce chiffre. Par exemple, si N et M sont respectivement le nombre de candidats pour le problème (P1) et le problème (P2), essayez d'évaluer la qualité des résultats pour `N = {2, 3, 4, 5}` et `M = {2, 3, 4, 5}` afin d'en faire sortir la meilleure combinaison[28].

Un sac à dos brut de décoffrage. A la manière de l'algorithme brute force du sujet précédent, implémentez l'algorithme de votre choix afin de résoudre le problème du sac à dos de manière exacte. Par comparaison des résultats, vous pourrez ainsi évaluer la qualité des solutions obtenues ici pour le problème (P1) sur de petites instances.

Solution du sac à dos déterministe. En détaillant chaque étape, montrer qu'il n'est pas systématique d'obtenir un chargement maximisant le bénéfice en prenant uniquement le meilleur candidat à chaque étape de la construction d'une solution au problème P1 (e.g., l'élément ayant le meilleure rapport bénéfice/consommation de la capacité du véhicule).

Opérateurs locaux. Vous disposez maintenant d'un programme générant plusieurs solutions au problème (P1) ∪ (P2) et encore plus si on réalise toutes les combinaisons possibles à partir de toutes les solutions des sous-problèmes (P1) et (P2). Il s'agit désormais de perturber une solution déjà construite dans le but de l'améliorer. Cette perturbation prendra le nom d'opérateur à appliquer directement sur la solution. Par exemple, à propos du sous-problème (P1), on pourra tenter d'échanger un objet sélectionné par un objet non sélectionné mais qui a un meilleur rapport bénéfice/poids tout en ayant un poids respectant la contrainte de capacité qui évolue en conséquence de l'opérateur. On peut également réfléchir à des opérateurs qui échangeraient deux objets sélectionnés contre un seul non sélectionné et inversement[29]. Pour (P2) on peut imager un opérateur échangeant l'ordre de deux villes en vérifiant l'impact sur la consommation totale. Enfin, après chaque tentative, on vérifie si la nouvelle solution est meilleure, et on la remplace par l'ancienne le cas échéant[30]. Pour plus d'information sur ce type

[27] Pour rappel : les bonus ne représentent pas le travail attendu qui est déjà important pour deux séances. Il permet d'aller plus loin sur les questions qui vous intéressent, il est d'ailleurs possible de proposer sa propre question bonus.
[28] Pour un nombre fixe de réplications.
[29] A vous de déterminer les règles qui permettent de sélectionner les bons candidats. Vous pouvez aussi ici utiliser la génération de nombres aléatoires afin de faire des choix parmi les candidats en entrée de vos opérateurs.
[30] Dans certains cas, il peut être utile de garder les anciennes puisqu'elles peuvent être à l'origine de plusieurs nouvelles et bonnes solutions sur de futurs applications d'opérateur.

de méthode, vous pouvez vous renseigner sur la métaheuristique GRASP évoquée dans le cours. A vous de mettre en place un tel processus.

Multi-véhicules. La réalité est plus compliquée et si on commence à s'en approcher pour un tel système à grande échelle, on va vite s'apercevoir que plusieurs véhicules de capacité différente devront faire les livraisons. Dans la littérature scientifique, ce problème est nommé *Capacitated Vehicle Routing Problem*. A vous d'imaginer comment intégrer cette composante après s'être aperçu que les problèmes (P1) et (P2) ne sont plus indépendants, puis à vous de développer de nouveaux algorithmes pour résoudre ce nouveau problème. Ceci est un travail conséquent, certains chercheurs travaillent même dessus depuis 40 ans, rien de tel pour passer un bel été 😉!

De meilleurs algorithmes. Il existe un grand nombre d'algorithmes capables de résoudre le problème du voyageur de commerce et celui du sac à dos. Pour le premier on pourra par exemple penser au vieil algorithme de Christofides[31] toujours classé parmi les meilleurs mais ce bonus vous laisse libre de choisir un algorithme pour résoudre l'un ou l'autre des problèmes après avoir justifié votre choix selon ses performances décrites dans la littérature.

Les bonus du passé. Que vous ayez ou pas répondu aux bonus des sujets précédents, vous pouvez les intégrer dans celui-ci. Par exemple, une documentation Doxygen serait la bienvenue, ou encore l'utilisation de coordonnées GPS et le calcul des distances selon la courbure de la terre peut-être intéressante.

[31] Ref : Christofides, Nicos. *Worst-case analysis of a new heuristic for the traveling salesman problem*. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976. Document original : https://apps.dtic.mil/sti/pdfs/ADA025602.pdf mais le wikipedia sera plus simple : https://fr.wikipedia.org/wiki/Algorithme_de_Christofides