1 Aperçu

Pour de TP, on va travailler avec le problème du voyageur de commerce (TP1). Pour rappel on la définit de la façon suivante : soit un ensemble de n villes, ainsi que la distance entre chaque paire de villes, déterminer le plus petit tour qui visite chaque ville exactement une fois et qui revient à son point de départ. Comme pour le TP1 on va représenter le TSP avec un graphe complet non orienté. $G(V, E)$, où V est l'ensemble des villes et E l'ensemble des arrêtes. Comme G est complet il y a une arrête en toute paire de villes $:(i, j) \in E, \forall i, j \in V, i \neq j$. Le coût d'une arrête e_{ij} est $c(e_{ij})$. La symétrie assure que $c(e_{ij}) = c(e_{ij})$.

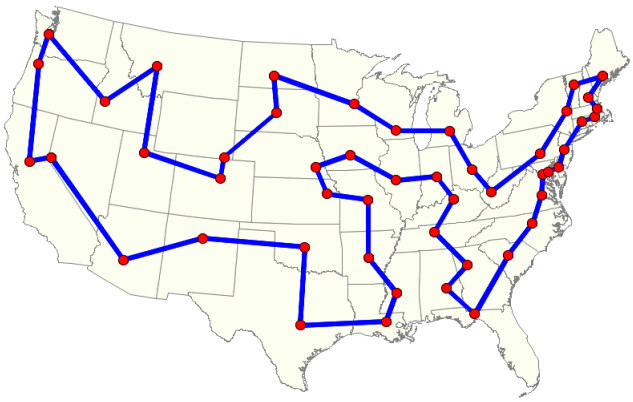

Figure 1: TSP illustration [1].

Comme vu pour le TP1, résoudre le TSP de façon exacte peut être dur même pour des petites instances. Votre travail dans ce TP sera d'utiliser une approche type colonie de fourmis pour optimiser le TSP. Vous devrez proposer un algorithme qui donne une bonne solution du TSP pour des instances de plus de 4000 villes.

2 Optimisation par colonie de fourmis [1]

L'optimisation par colonies de fourmis (ACO) est une métaheuristique inspirée par la nature. Elle a été proposée par Marco Dorigo dans sa thèse de doctorat. L'ACO a été conçue pour résoudre des problèmes combinatoires difficiles et est inspirée du processus d'optimisation mis en place par les fourmis pour trouver le plus court chemin entre la nourriture et leur nid. Les fourmis sont capables de trouver le plus court chemin sans utiliser d'indices visuels, en exploitants l'information des phéromones. En marchant, les fourmis déposent des

[1] Cette section est fortement inspirée du travail de Marco Dorigo et al (1997) [5] et Marco Dorigo et al (2006) [3], que les étudiants intéressés peuvent lire.

phéromones et suivent, avec une certaine probabilité les phéromones laissées par les autres fourmis.

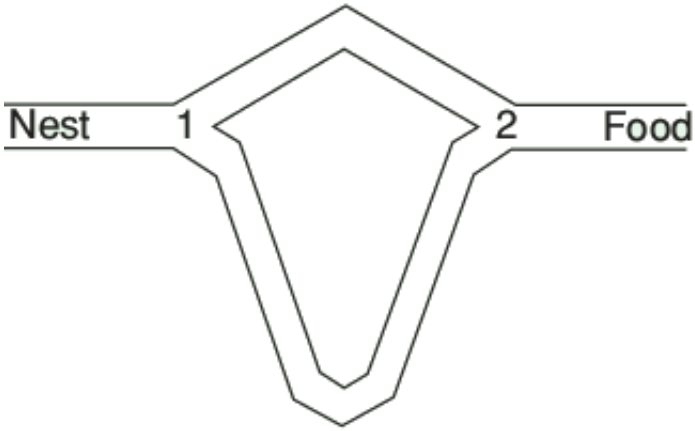

Figure 2: Illustration de la décision prise par les fourmis pour trouver le plus court chemin vers leur nourriture [3].

Considérez l'image ci-dessus, lorsque le premier groupe de fourmis arrive à la bifurcation 1, elles doivent décider si aller à droite ou à gauche. Puisqu'elles n'ont pas d'indice concernant la longueur des chemins, elles choisissent au hasard. (En moyenne la moitié des fourmis ira dans chaque branche). Considérant que toutes les fourmis vont à la même vitesse, les fourmis qui auront emprunté le chemin du haut, arriveront à la nourriture plus vite, et rentreront au nid plus vite. Lorsque les fourmis qui sont passé par en bas se retrouve au point de décision 1, le niveau de phéromone en haut sera plus élevé puisque les fourmis qui ont pris ce chemin y sont déjà passé deux fois, un fois pour aller chercher la nourriture et une fois pour revenir. Cette fois ci il est plus probable qu'elles choisissent le chemin du haut, intensifiant ainsi sa trace en phéromone. Après quelques itérations, la différence en phéromone devient suffisante pour influencer une fourmi tout juste arrivée sur le chemin. A partir de maintenant les nouvelles fourmis vont préférer choisir avec une forte probabilité le chemin du haut, puisqu'elles peuvent y percevoir une plus grande quantité de phéromone. Petit à petit la probabilité de choisir le chemin du haut augmente jusqu'à ce qu'elles passent toute par le chemin le plus court[2].

Le comportement ci-dessus a inspiré l'algorithme ACO, dans lequel un ensemble de fourmis artificielles coopèrent pour trouver la solution d'un problème par échange d'information par un dépôt de phéromone sur les arêtes du graphe. Le pseudo code si dessous présente l'idée générale de l'algorithme ACO. L'algorithme crée K fourmis, chacune responsable de créer une solution au problème, en utilisant la phéromone comme guide pour obtenir de bonnes solutions. Lorsque qu'une fourmi trouve une solution, elle peut être améliorée grâce à des heuristiques. Un fois que chaque fourmi a construit sa solution, la concentration de phéromone est mise à jour en prenant en compte quelle était la meilleure solution trouvée. La section suivante explique comment appliquer cette idée au TSP.

[2]vous pouvez voir la vidéo suivante `https://www.youtube.com/watch?v=HisgmcLaHoY` sur une colonie de fourmis dans l'insectarium de Montréal.

Algorithm 1 pseudo code de l'algorithme ACO

```
while stop condition is not met do
    Send K ants to construct K solutions.
    Apply an local search algorithm to improve the solutions found.
    Update the level of pheromone according to the best solution.
end while
```

2.1 L'ACO appliqué au TSP

Dans l'ACO, le problème est abordé en simulant un certain nombre de fourmis se déplaçant sur les arêtes du graphe, pour chacune desquelles un niveau de phéromone $\tau(e)$ est associé. Ce niveau peut être lu et modifié par les fourmis. A chaque itération, les K fourmis sont considérées, chacune construit sa solution en allant de sommet en sommet, avec la contrainte de ne pas visiter deux fois un sommet. À chaque étape la fourmi choisit le sommet à visiter à l'aider d'un phénomène aléatoire, basé sur les phéromones. Si la fourmi est au sommet i, le sommet suivant est sélectionné parmi les non visités, avec une probabilité proportionnelle à son taux de phéromone $\tau(e_{ij})$ associé au segment e_{ij}.

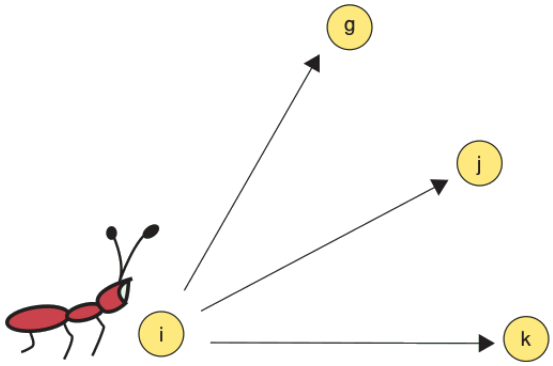

Figure 3: illustration d'une fourmi sur le sommet i choisissant le sommet suivant à visiter. [3].

A la fin de chaque itération en fonction de la qualité de la solution construite, les valeurs de phéromones sont modifiées, pour influencer les fourmis dans les itérations futures et construire des solutions proches des meilleures déjà construites.

Il y a trois étapes à l'algorithme ACO pour le TSP.

1. **construire les solutions des fourmis:** Formellement, soit S_k la solution construite par la fourmi k, avec $V(S_k) \subseteq V$ l'ensemble des sommets visités et $\overline{V}(S_k) = V \backslash V(S_k)$ l'ensemble des sommets non visités.

   Dans la construction d'une solution, les fourmis choisissent la ville à visiter par un mécanisme aléatoire. Quand la fourmi k est dans la ville i et a construit jusqu'à

maintenant la solution S_k, la ville suivante à visiter est obtenue par l'expression (1):

$$j = \begin{cases} \arg\max_{j \in \overline{V}(S_k)} \left\{ \dfrac{\tau(e_{ij})}{c(e_{ij})^\beta} \right\}, \text{si } q \leq q_0 \\ \text{Sélectionne la ville aléatoirement selon la distribution } D \\ \text{définie ci-dessous, si } q > q_0 \end{cases} \quad (1)$$

Où β est un paramètre qui détermine l'importance relative des phéromones par rapport à la distance ($\beta > 0$), q est une réalisation d'une variable aléatoire uniformément distribuée sur [0...1], q_0 est un paramètre ($0 \leq q_0 \leq 1$), et la distribution de probabilité D contient la probabilité $p_k(e_{il})$ d'aller à la ville j :

$$p_k(e_{ij}) = \begin{cases} \dfrac{\tau(e_{ij})/c(e_{ij})^\beta}{\sum_{l \in \overline{V}(S_k)} \tau(e_{il})/c(e_{il})^\beta}, \text{si } j \in \overline{V}(S_k) \\ 0, \text{sinon} \end{cases} \quad (2)$$

Pour comprendre cette construction, considérez l'exemple avec $q_0 = 0.9$. Il y a alors 90% de chance que la prochaine ville est celle qui maximise la relation entre le niveau phéromone et la longueur de l'arête (si plusieurs villes ont la même valeur alors choisir au hasard parmi celles-ci). Sinon avec 10% de chance on choisit au hasard une ville parmi celle restantes, pondérée par la phéromone et la distance à la ville selon l'équation (2).

2. **Appliquer la recherche locale.** Une fois les solutions construites, et avant de mettre à jour les phéromones, il est fréquent d'améliorer les solutions obtenues par une recherche locale. Cette étape aide l'algorithme à converger rapidement.

   Pour le TSP par exemple cet amélioration peut se faire avec l'heuristique qui échange deux arêtes si le poids total du tour s'en trouve améliorer. Cette heuristique converge vers un minimum local. C'est l'une des heuristiques 2-opt les plus connues pour le TSP [2].

   Considérez une solution S comme une séquence de sommets (suivant l'ordre de visite). Pour un sommet i, soit i' son successeur, dans S. L'heuristique 2-opt marche de la façon suivante: pour toute paire de sommets non consécutifs i, j, regarder si enlever les arêtes $e_{ii'}$ et $e_{jj'}$ est rajouter les arêtes e_{ij} et $e_{i'j'}$ diminue le coût de la solution, si oui invertir les arêtes. Répéter ce processus jusqu'à ce qu'il n'y ait plus d'arêtes échangeables. La figure suivante illustre ce mécanisme.

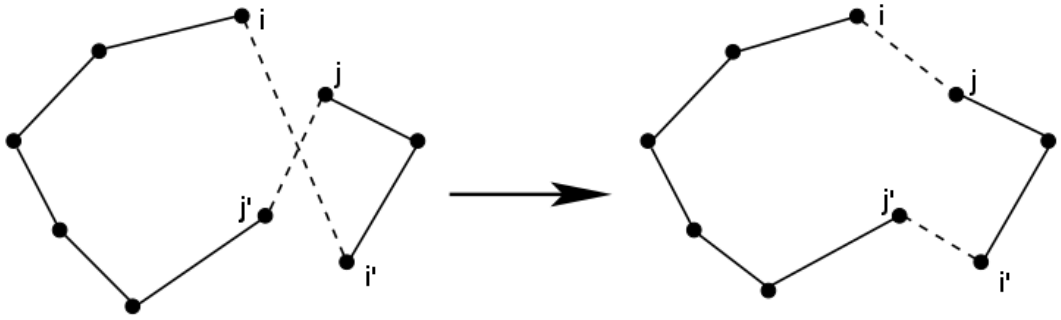

Figure 4: 2-opt heuristic illustration. Adapted from [4].

3. **mise à jour des phéromones :** Le but de l'actualisation des phéromones est d'augmenter la valeur associée aux bonnes solutions et aux solutions prometteuses, et diminuer celle des mauvaises solutions. En général cela est fait en (i) diminuant les phéromones par évaporation, et (ii) augmenter les niveaux de phéromones associé avec les meilleures solutions trouvées.

   Il y a deux types de mise à jour:

   - **mise à jour globale:** Une fois que toutes les fourmis ont construit leur chemin il faut mettre à jour les niveaux de phéromone par l'expression suivante :

   $$\tau(e_{ij}) = (1-\rho)\tau(e_{ij}) + \rho\Delta\tau(e_{ij}) \tag{3}$$

   où ρ $(0 < \rho < 1)$ est le ratio d'évaporation,

   $$\Delta\tau(e_{ij}) = \begin{cases} \dfrac{1}{L_{gb}}, \text{if } e_{ij} \in \text{meilleur tour global,} \\ 0, \text{sinon.} \end{cases} \tag{4}$$

   Si L_{gb} est la longueur du meilleur chemin du début de l'algorithme. En utilisant les expression (3) et (4), les arêtes qui appartiennent au meilleur tour voient leur niveau de phéromones augmenter, pendant que les autres diminuent.

   - **Mise à jour locale:** Le but principal de la mise à jour locale est de diversifier la recherche faite par les fourmis pendant la construction des chemins. Cela est fait en diminuant le niveau de concentration en phéromone sur les arêtes visitées, pour inciter les fourmis à prendre des arêtes non visitées et produire de nouvelles solutions. Ainsi il est moins probable que des fourmis produisent le même chemin pendant une itération.

     La recherche locale est faite par toutes les fourmis, après chaque ajout de ville à une route. Chaque fourmi applique sur la dernière arête ajoutée :

     $$\tau(e_{ij}) = (1-\varphi)\tau(e_{ij}) + \varphi\tau_0(e_{ij}). \tag{5}$$

     Où $\varphi \in (0, 1]$ est le coefficient de décroissance de la phéromone, et $\tau_0(e_{ij})$est la valeur initiale de phéromone dans l'arête e_{ij}. Pour ce TP ces valeurs sont données.

Question. Comment reliez-vous l'ACO à l'apprentissage par renforcement?

3 A vos ordinateurs

Note: Tous les fichiers source et squelettes pour ce TP sont disponibles en C++ et python sur la page moodle du cours. Vous pouvez utiliser aussi tout autre langage qui vous parait pertinent.

- **Graph.** le fichier `Graph.h/Graph.py` est un fichier prêt à être utilisé, il contient la classe graph utilisée pour représenter le problème. Un objet de cette classe a les 4 méthodes suivantes:
  - `Graph(dataset_file)` le constructeur qui prend en argument le nom du dataset (string) et construit le graphe.
  - `getN()` qui renvoie le nombre de sommets n (`.N` en python)
  - `getEdge(i,j)` qui renvoie l'objet edge correspondant à l'arête e_{ij} (`get_edge(i,j)` en python).

  Un objet **edge** e a 3 attributs :
  - `e.source:` le sommet de départ;
  - `e.destination:` le sommet d'arrivée;
  - `e.cost:` le coût du sommet.
- **Solution.** La classe Solution (définie dans *Solution.h*) a aussi quatre variables :
  - `visited:` la liste de ville visitées,
  - `not_visited:` la liste des villes non visitées,
  - `cost:` le coût de la solution actuelle. On le notera g(i).
  - `graph:` le graphe du problème (de la classe *Graph*). Cet variable est utilisée dans certaines méthodes de résolution *Solution*.

  Et quatre fonctions:
  - `Solution(g):` le constructeur de la classe Solution. Il prend en argument le *Graph* g et initialise les attributs de la solution. Cette méthode est utilisée pour créer une solution vide.
  - `Solution(sol):` une copie du constructeur, en python elle est incluse dans le constructeur. Cette méthode construit une autre solution à partir de l'instance *sol* et copie l'ancienne solution dans la nouvelle.
  - `addEdge(v,u):` ajoute l'arête (v,u) à la solution et met à jour son coût. v est un sommet déjà visité, plus précisément le dernier sommet visité. u est un sommet non visité, qu'il faut ajouter à la route.

– `print():` affiche la solution et son coût.

- **ACO** c'est la classe qui implémente l'algorithme ACO.

  Il y a plusieurs variables dans cette classe :

  – `parameter_q0:` le paramètre q_0;
  – `parameter_beta:` le paramètre β;
  – `parameter_rho:` le paramètre ρ;
  – `parameter_phi:` le paramètre φ;
  – `parameter_K:` le nombre de fourmis;
  – `parameter_init_phoromone:` les valeurs de $\tau_0(e)$ pour chaque $e \in E$;
  – `pheromone`: Le vecteur qui contient les valeurs actuelles de phéromone pour chaque arête;
  – `best`: un objet solution qui contient la meilleure solution jusqu'à maintenant.

  Complétez maintenant les méthodes suivantes de la classe ACO:

  – `getNextCity(sol):` renvoie la prochaine ville à ajouter à la solution `sol` en utilisant les expressions (1) et (2). N'hésitez pas à utiliser la librairie Numpy qui vous rendra la tache bien plus simple. Quelques liens pour vous y familiariser:

Exécution. Testez maintenant votre code en utilisant la fonction de test `testNextCity()` du fichier `Tests`

  – `heuristic2opt(sol):` applique la recherche locale 2-opt à la solution `sol`. Vous pouvez créer une fonction `inverser_ville(i,j)` dans `Solution` qui inverse deux villes et l'ordre du tour.

Exécution. Testez maintenant votre code en utilisant la fonction de test `testHeuristic2opt()` du fichier `Tests`.

  – `globalUpdate(best):` Fait la mise à jour globale en utilisant la meilleure solution trouvée, selon les formules (3) et (4).

Exécution. Testez maintenant votre code en utilisant la fonction de test `testGlobalUpdate()` du fichier `Tests`

  – `localUpdate(sol):` fait la mise à jour local en utilisant la solution `sol`, selon la formule (5).

Exécution. Testez maintenant votre code en utilisant la fonction de test `testLocalUpdate()` du fichier `Tests`

  – `runACO(numberIteration):` lance l'algorithme ACO, et renvoie la meilleure solution trouvée. Le paramètre `numberIteration` est le nombre maximum d'opérations. Vous devez implémenter ici le déroulement de l'algorithme, à savoir : la création de K fourmis, la construction de la solution de chaque fourmis, l'exécution de la

recherche locale et la mise à jour de la phéromone. Vous pouvez créer une fonction `build_sol()` qui construit une solution (une fourmi).

Exécution. Testez maintenant votre code en utilisant la fonction de test `testRunACO()` du fichier `Tests`

4 Expériences

4.1 Réglage des paramètres

Comme dans de nombreux algorithmes d'IA, il est nécessaire de régler les paramètres pour que l'algorithme puisse atteindre une performance maximale.

Soit $P = \{q_0, \beta, \rho, \varphi, K\}$ notre ensemble de paramètres, une procédure simple pour les régler un paramètre $p \in P$ est de fixer tous les autres paramètres $|P| - 1$, et de faire de multiples expériences en changeant la valeur de p, et de regarder les performances de l'algorithme (convergence, précision).

Par exemple si vous voulez régler q_0, vous pouvez mettre sa valeur à 0, lancer une expérience, observer le comportement de l'algorithme. Puis incrémentez sa valeur à 0.1 et étudiez le comportement de l'algorithme. Recommencez jusqu'à atteindre $q_0 = 1$ et identifiez quelle était la meilleure valeur. Pour vous aider vous pouvez commencer avec les valeurs suivantes pour les paramètres :

q_0	β	ρ	φ	K
0.9	2	0.1	0.1	10

Pour la suite du TP vous aurez besoin des fichier des villes, disponibles à l'adresse suivante : `https://drive.google.com/open?id=1l-g52fpl26T2D50LFbix40MP3gMB6aYe`

Utilisez maintenant les données **qatar** pour régler vos paramètre. Vous pouvez utiliser 1000 comme nombre maximum d'itérations. Pour chaque paramètre p vous devez afficher deux graphes : $p\times$coût de la meilleure solution et $p\times$temps d'exécution. Puisque l'algorithme est stochastique, lancez le 5 fois pour chaque valeur de p et reportez la valeur moyenne.

4.2 Utilisation de l'algorithme

Maintenant il temps de résoudre des plus grandes solutions de TSP. En utilisant votre ensemble de paramètres optimisés, lancez votre algorithme sur les deux instances de TSP nationales ci-dessous.

Instance	**Cities**	**Optimal cost**
Uruguay	734	79114
Canada	4663	1290319

Pour chaque instance, vous devez lancer votre algorithme 5 fois, en utilisant 1000 itérations maximum. Reportez vos résultats pour chaque exécution en remplissant la table 1. Vous devez donc reporter, toutes les 100 itérations :

gap_{best} donné par $100\% \times \frac{ACO_{best} - OPT}{OPT}$ Où ACO_{best} est le meilleur coût trouvé jusqu'à présent et OPT est le coût optimal.

iterations without improvement le nombre d'itérations faites depuis les derniers progrès de la solution.

CPU time le temps d'exécution depuis le début de l'exécution

Table 1: Exécution i for instance x

iteration	**gap_{best}**	**iterations without improvement**	**CPU time**
1			
100			
200			
300			
400			
500			
600			
700			
800			
900			
1000			

Par exemple, si pour une exécution vous trouvez, pour les données sur l'Uruguay, une solution de coût 90330 après la première itération, vous aurez une table qui commence comme ceci :

iteration	**gap_{best}**	**iterations without improvement**	**CPU time**
1	14.17%	0	0.01s
⋮	⋮	⋮	⋮

Pour conclure, vous devrez résumer vos résultats dans la table suivante.

Instance	**Best gap**	**Worst gap**	**Average gap**	**Average time**
Canada				
Uruguay				

5 Bonus

Comment pouvez-vous améliorer vos résultats? Trouvez un moyen de corriger l'algorithme ACO et reportez les améliorations.

6 Directives de remise

Le travail sera réalisé en équipe de deux ou trois. Vous remettrez un fichier .zip par personne, nommés TD1_nom_prenom_matricule.zip. Vous devez également remettre un fichier PDF contenant une explication de vos implémentations ainsi que les réponses des questions. Merci de ne pas remettre les données avec le code. Tout devra être remis avant le **2 Décembre à 23h55**. Tout travail en retard sera pénalisé d'une valeur de 10% par jour de retard.

Barème :

implémentations:	30 pts
résultats:	30 pts
rapport :	40 pts

*Ce TP a été imaginé par Daniel Aloise et développé par Rodrigo Randel

References

[1] Robert Allison. What's new in v9.4 sas/graph... [online] robslink.com. available at: http://robslink.com/sas/democd62/new_94_sas.htm [accessed in 1 nov. 2017].

[2] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[3] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[4] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages 1470–1477. IEEE, 1999.

[5] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.