Projets numériques : liste des sujets

La plupart des sujets contiennent l'objectif principal et sa description. Si vous êtes rapide, il contient aussi un bonus que vous êtes libre de faire ou non.

Partie Physique numérique

Sujet 1: équations différentielles et mécanique céleste : voyage vers Mars

Le voyage vers Mars depuis la Terre s'effectue le long d'une orbite dite de transfert ou de Hohmann. On pourra dans un premier temps se renseigner sur l'approche théorique de ce problème en mécanique du point. Le but du TP ici est de simuler les trajectoires de la Terre et de Mars avec des conditions initiales choisies arbitrairement. Pour simplifier cependant, on supposera que la Terre et Mars orbitent dans le même plan.

Étant donné cette partie réalisée, simuler la trajectoire d'une fusée sur l'orbite de transfert, et en déduire les instants des "fenêtres de tir" qui permettent effectivement de rejoindre Mars depuis la Terre. Enfin, on calcule la masse d'ergol requise pour effectuer les manœuvres permettant de quitter l'orbite terrestre avant de s'insérer en orbite autour de Mars.

1) Commencer par écrire les équations du mouvement d'une planète soumise à l'attraction gravitationnelle du Soleil.
2) Écrire une classe Orbit représentant l'orbite d'une planète autour du Soleil. Cette classe comprendra une méthode integrate() permettant d'intégrer les équations écrites précédemment. Pour réaliser l'intégration numérique, on effectuera une recherche sur les méthodes d'Euler et on utilisera un schéma d'Euler explicite.
3) Après une recherche théorique sur le transfert de Hohmann, utiliser la classe Orbit pour représenter une orbite de transfert puis calculer les fenêtres de lancement permettant de rejoindre Mars.

Pour calculer la quantité de carburant nécessaire pour effectuer le voyage entre la Terre et Mars, on commence généralement par calculer les delta V qui sont les incréments de vitesse qu'on doit donner à la sonde pour changer d'orbite. Dans le cadre du transfert de Hohmann, deux manœuvres sont nécessaires : il faut accélérer pour quitter l'orbite terrestre, puis freiner pour mettre la sonde en orbite autour de Mars. Il est donc nécessaire de calculer deux deltas V.
On supposera que la sonde a été mise en orbite autour de la Terre à l'altitude géostationnaire et que son orbite cible autour de Mars a pour altitude 400 km.
Le delta V est défini comme la différence entre la vitesse sur l'orbite circulaire de départ (ou d'arrivée) et la vitesse de l'objet à la même position mais sur une orbite hyperbolique permettant à l'objet de s'extraire (ou de s'insérer) de l'attraction de la Terre (ou de Mars).

Une fois les deltas V connus, on peut calculer la quantité de carburant nécessaire pour réaliser les manoeuvres à l'aide de : delta m = m*(1-exp(-delta V/(g*ISP))) où l'ISP (impulsion spécifique) est une grandeur caractéristique du propulseur.

Aide : la vitesse d'un objet sur une orbite hyperbolique est donnée par V = sqrt(Vinf^2+2*G*M/r) où Vinf est la vitesse à l'infini, c'est-à-dire la différence entre la vitesse de la Terre (ou de Mars) et celle de la sonde sur leur orbite héliocentrique.

4) A l'aide de ces informations, calculer la masse de carburant nécessaire au voyage sur Mars.

Bonus : Représenter plusieurs révolutions de la trajectoire de la Terre sur son orbite autour du Soleil. Zoomer éventuellement sur la trajectoire de la Terre. Que constate-t-on ?
Pour résoudre ce problème, tester d'autres schémas d'intégration numérique.

Sujet 2 : équation aux dérivées partielles : propagation de la chaleur.

On considère une tige de cuivre carré d'arête a et de longueur L, initialement en équilibre thermique avec l'air ambiant à T_a = 20°C. A l'instant initial, on place l'extrémité gauche de la barre au contact d'une source chaude T_1 = 100°C, et l'extrémité droite au contact d'une source froide T_2 = -30°C. (Vu l'équation ci-dessous on peut aussi bien utiliser le Celsius que le Kelvin). Le but du problème est d'étudier la propagation de la chaleur dans la barre, c'est-à-dire évaluer numériquement l'évolution spatio-temporelle du champ de température dans la barre en résolvant l'équation de la chaleur, donnée par

$$dT(t,x,y,z)/dt = D(Cu)*Laplacien(T)$$

où D(Cu), le coefficient de diffusion thermique du Cuivre, sera pris égal à 117 en m^2.s^(-1). La méthode consiste en plusieurs étapes. Il faut discrétiser spatialement la barre de cuivre en cellules élémentaires de taille epsilon^3, et choisir un pas de temps delta t, et résoudre pas de temps par pas de temps la propagation, c'est à dire en particulier construire un opérateur Laplacien discret, et tenir compte des conditions aux limites. NB: on considère que les sources chaudes, froides, et l'air ambiant comme des thermostats ayant une température fixe.

- On écrira donc une classe ayant pour attributs toutes les constantes du problème (T_0, T_1, T_a, a, L, etc) et initiant également un array de dimension 3 pour la modélisation sur grille du champ de température à l'instant t. A t= 0, la barre était à l'équilibre avec l'air ambiant, donc T_barre = T_a partout.
- Définir une méthode save qui permet de stocker dans un fichier l'array à l'instant t, on s'en servira à la fin pour les représentations graphiques (utiliser np.savetxt ou le module pickle)

- Définir une méthode qui prend l'array représentant le champ de température et renvoie son Laplacien discretisé. (attention aux conditions aux bords!). On prendra pour la dérivée seconde dans la direction x, par exemple, une dérivée seconde discrète d_xx T(x) == (T(x+epsilon) - 2 T(x) + T(x-epsilon))/epsilon^2. Vous pourrez utiliser la méthode numpy.diff()
- Définir une méthode simulate() qui réalise l'intégration numérique. On prendra epsilon << a. On essaiera d'anticiper quelle valeur prendre, au moins en ordre de grandeur, pour le pas de temps. Si nécessaire, on fera quelques essais différents en variant le pas de temps.
- Réaliser l'intégration numérique. On doit voir un régime permanent se mettre en place au bout d'un certain temps. Afficher des graphes 2D (en coupe) en fonction du temps de l'évolution du champ de température pour votre rapport/présentation. Utiliser plt.imshow() du module matplotlib.pyplot; ce module permet aussi de sauver des figures.

Bonus : Même problème, en deux dimensions, en coordonnées polaires. Cette fois on place une source chaude qui est un disque de rayon r au centre du système de coordonnées. Autour, il y a de l'eau à une certaine température initiale. Étudier la propagation de la chaleur dans l'eau.

Sujet 3 : Détente de Joule Gay Lussac d'un gaz de sphères dures avec collisions (1D)

On va simuler l'évolution par chocs successifs d'un gaz de particules ayant des conditions initiales très particulières (de basse entropie). On s'attend à ce que les chocs successifs amènent le système à l'équilibre dans un état de plus haute entropie et ce de façon spontanée (sans intervention extérieure).

Questions préliminaires. On considère d'abord une seule particule de 'rayon' r placée dans un segment [0, L], qui sont les "bords" fixes de la boîte. **On prendra r << L**. Pour les deux premières questions, on pourra prendre des conditions initiales arbitraires prises à la main.

- Étant donné la position du centre de masse/vecteur vitesse initiale, faire évoluer le système jusqu'au prochain instant où il se passe quelque chose (c'est-à-dire le rebond sur un bord), instant qu'il faut d'abord déterminer. (On supposera le rebond élastique, cad simple inversion de la vitesse). En déduire un code qui permet de savoir où se trouve la particule et quelle est sa vitesse à tout instant dans le futur.
- Notre système contient maintenant 2 particules identiques. Il est aussi possible que les 2 particules entrent en collision. A chaque temps t, il nous faut donc pouvoir calculer un vecteur d'instants ultérieurs (t_1, t_2) où il se passera quelque chose de significatif dans le système (soit rebond gauche, soit rebond droit, soit collision). Avancer le système jusqu'au minimum de ces instants, procéder à l'événement, et itérer. Les masses étant égales, on reverra les lois des chocs pour savoir quelle formule simple implémenter dans le cas d'une collision 1D. En déduire un code qui permet de simuler entièrement l'évolution temporelle de ce cas à deux particules. Rq : il est possible, quoique excessivement improbable, qu'il y ait un rebond gauche et un rebond droit pile au même instant. Le prendre en compte.

Déduire de ce travail préliminaire une classe permettant de simuler l'évolution d'un gaz unidimensionnel. Les attributs initiaux sont le nombre de particules n, leurs masses (toutes égales à m), les dimensions r et L. Les autres sont la liste des coordonnées des centres de masses; la liste des vitesses. (ou les arrays si on passe en numpy)

- Il faudra commencer par une méthode de condition initiale qui renvoie pour chaque particule sa position initiale et sa vitesse (positive ou négative selon l'orientation); comme on ne souhaite pas que le système soit initialement à l'équilibre, on choisira une condition initiale telle que les particules occupent seulement une partie de l'espace à gauche dans l'intervalle, par exemple dans l'intervalle [0, L/4]. Il faut s'assurer que les particules ne se chevauchent pas. Par ailleurs pour les vitesses on prendra une distribution gaussienne (ou dite aussi "normale") de moyenne nulle et de variance fixée : on utilisera le module numpy.random.normal().
- Généraliser la méthode renvoyant les instants ultérieurs où il se passe quelque chose, et pour qui (pour la particule à la ième position de gauche à droite, on calculera le temps de la prochaine collision avec son voisin gauche i-1 et son voisin droit i+1; sauf en i = 1 et i=n (rebond sur les bords); on renverra le temps minimal et la ou les collisions à effectuer cad le couple (i, i-1) ou (i, i+1)).
- Généraliser la méthode effectuant le rebond pertinent en distinguant le rebond particule-particule de particule-bord fixe.
- Généraliser la méthode simulate(t) qui permet d'avancer jusqu'à un temps t.

- Stocker aussi à chaque instant pertinent la valeur de l'énergie pour vérifier plus tard à la fin de la simulation que celle-ci est bien conservée.
- Vérifier que le système remplit toute la boîte en un temps assez court.

Bonus : partir cette fois d'un gaz à l'équilibre thermique; et le comprimer de façon quasi-statique en rapprochant lentement le bord droit L -> L(t) = L - alpha t avec alpha très petit; ou en le comprimant de façon irréversible et brutale (alpha grand). Caractériser l'état final en particulier sa température finale. On ne doit pas trouver le même état final; cf cours de thermo.

Partie percolation/graphes/arbres

Sujet 4 : percolation de l'eau dans un milieu poreux (percolation dirigée).

Soit une matrice de taille n*n emplie aléatoirement de 0 ou de 1 avec une probabilité p. On "verse de l'eau au sommet de la matrice". L'eau peut se propager dans les cases vides (0). La question est de savoir l'eau "percole", c'est-à-dire si elle atteint la dernière ligne de la matrice.

Version 1 facile : la règle est que pour la première ligne de la matrice, l'eau peut remplir une case si celle-ci est vide (contient un 0), et est bloquée sinon. Pour les lignes suivantes, une case initialement vide peut être à son tour remplie d'eau s'il y a de l'eau qui provient de la case du haut ou des cases adjacentes. On pourra faire une simulation ligne par ligne brute force.

Version 2 : En vérité l'eau peut éventuellement localement remonter avant de redescendre comme dans le graphe suivant où les X sont des cases imperméables et les O des cases remplies d'eau.

XOXXXXXX
XOXXOOOO
XOOOOXXO
XXXXXXXO

Dans ce cas, la simulation ligne par ligne n'est plus correcte puisque l'eau peut remonter de plus bas. L'adapter rendra la complexité de l'algorithme beaucoup trop coûteuse. En fait, la seule chose que l'on veut savoir, c'est l'existence ou non d'un chemin depuis le bas vers le haut. Il n'est pas nécessaire en soi de savoir exactement quelles cases seront remplies d'eau à la fin.
La question est donc assez proche de celle de trouver son chemin dans un labyrinthe. L'algorithme standard pour cela est par exemple le "DFS", algorithme de parcours en profondeur. En partant d'une des cases vides (0) de la ligne supérieure, on choisira un sens de "recherche de cases 0" fixe (bas, gauche, droite, haut, par exemple). L'algorithme continue son chemin tant qu'il le peut, mais se remémore les cases déjà visitées afin de ne pas tourner en rond. Quand il ne peut plus avancer, il doit revenir à une des cases précédentes et explorer les chemins non encore explorés à partir de cette case. Si une des cases appartient à la dernière ligne de la matrice, on a trouvé un chemin. Sinon, tous les chemins à partir de cette case auront été explorés, et on passe à une autre case vide de la première ligne. On vous encourage à vous renseigner davantage sur le DFS, la question des labyrinthes, et vous familiariser d'abord avec cet algorithme.

En réalisant un nombre suffisant de simulations, étudier la probabilité que l'eau traverse de part en part la matrice en fonction de p et de n. Ce système est connu pour exhiber une transition de phase, avec aucun chemin si $p < p_c$ une valeur critique, et des chemins sinon. (cela peut dépendre de n, mais si on prend n assez grand cela ne doit pas en dépendre). Essayer de déterminer expérimentalement p_c.

Sujet 5 : propagation des feux de forêts avec et sans vent

NB : sujet similaire à la percolation, on part donc d'une matrice similaire à celle décrite dans ce sujet. Ici les 1 représentent la présence d'un arbre, les 0 l'absence d'arbres.

En l'absence de vent, une fois qu'un arbre est en feu, ses voisins (est ouest sud nord mais aussi en diagonale) prennent également feu.

Dans une première étape, on affichera les arbres ayant brûlé à partir d'une source de feu choisie à votre guise.

Dans un deuxième temps, on dira que l'arbre (on les suppose tous identiques) brûle pendant N unités de temps, et que les arbres voisins n'ont qu'une probabilité p par unité de temps de se mettre à brûler. Effectuer à nouveau quelques simulations.

Dans un troisième temps, on inclura l'effet du vent. Celui-ci aura toujours la même direction pendant l'expérience. Donner des choix de modélisation qui vous semblent réalistes (affecter par exemple une probabilité plus grande pour que l'arbre voisin sous le vent prenne feu, et moindre pour celui contre le vent).

Les étudiants qui auront choisi ce projet pourront s'inspirer du projet 2014-2015 d'anciens étudiants de CPEI 2 (l'équivalent CPUGE 2) dont la présentation est ici

https://docs.google.com/presentation/d/e/2PACX-1vR9Az5y1yO3WP3mzr27TRLwpDu9X9CnZiZhgWezsX-DHP4HCY-TwFIeCKvOfDyqE8BjLUYGQAhlED23/pub?start=false&loop=false&delayms=5000

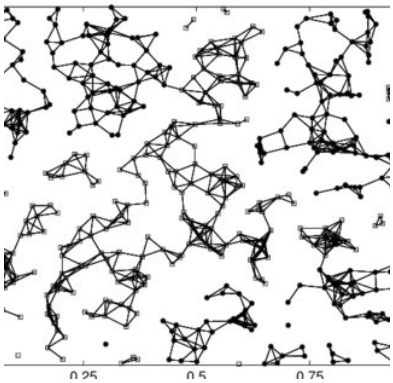
Il convient de noter ici que la modélisation du vent dans le projet ci-dessus est *optionnelle* car elle est déjà assez avancée et que les étudiants qui choisissent une telle modélisation doivent bien mesurer en accord avec le prof de cours et les enseignants en TD la difficulté de sa mise en œuvre pour le temps imparti.

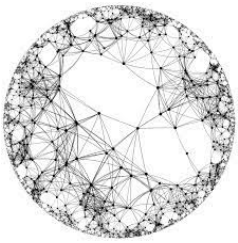
Sujet 6 : Graphes aléatoires géométriques pour la modélisation d'une épidémiologie

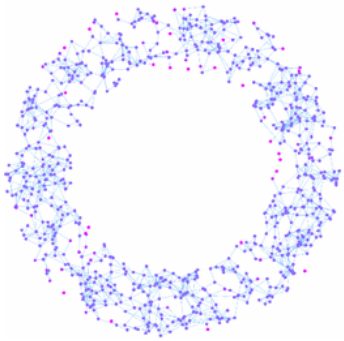
Dans ce sujet, les sommets du graphe seront les individus et les arêtes reflètent les liens existant entre individus proches géographiquement.

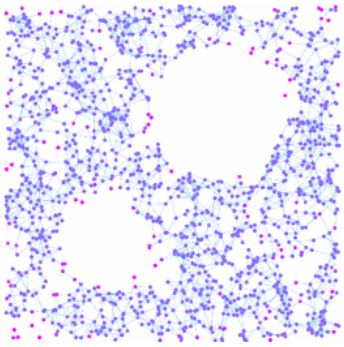
On se place donc dans un contexte où, le graphe est considéré du point de vue géométrique: deux nœuds/sommets sont adjacents s'ils sont à distance euclidienne inférieure à un rayon R (qui est un paramètre qu'on pourra varier).

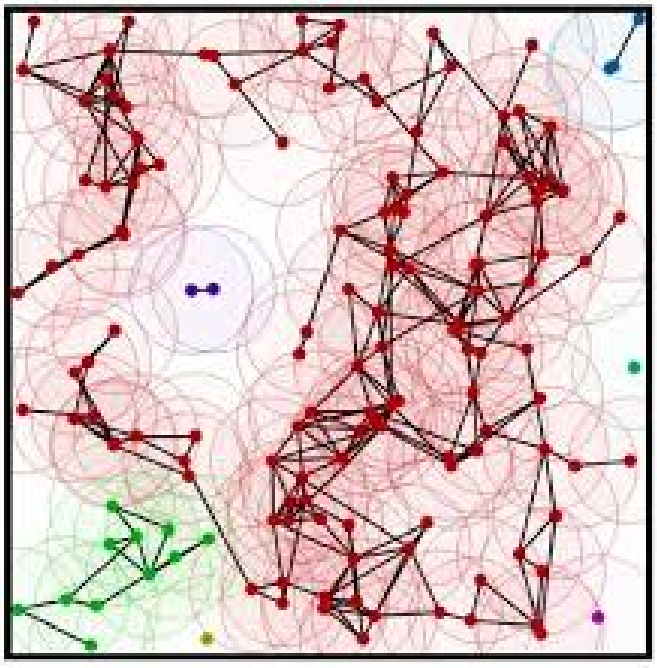
Dans une première étape, on choisira de placer uniformément aléatoirement N sommets sur une surface S (carré simple, ellipsoïde, surface torique) et selon le rayon R, on construira alors le graphe qu'on notera G(N, S, R). Ci-dessous, nous avons l'exemple de tels graphes sur des surfaces support et des paramètres différents. Notez par exemple que sur les graphes en bleu, la présence de "trous" dans les surfaces considérées influent sur les structures du graphe. Sur le dernier graphe coloré à droite, nous pouvons voir les différentes composantes connexes qui sont colorées en vert, rouge et bleu.











Dans une seconde étape, on pourra tester que le rayon R joue aussi un rôle crucial: si R=0 alors on aura N sommets isolés et si R est grand (par exemple un seul sommet couvre toute la surface) alors on aura un graphe complet (tous les sommets sont deux à deux voisins). En faisant varier R, il faut alors exhiber expérimentalement la *plus petite* valeur de R qui donnera un graphe connexe.

Dans une troisième étape, on pourra montrer expérimentalement que l'arbre couvrant de poids minimum a un poids concentré autour d'une certaine valeur M qui est fonction de N, S et R. On calcule via des simulations la moyenne M du poids de cet arbre et on peut montrer expérimentalement après ce calcul que lors de nouveau(x) tirage(s) de nouveaux graphs, les poids des nouveaux arbres couvrant de ces derniers sont proches de M.

Selon l'avancement du projet, on pourra rajouter des étapes de modélisation. Une propagation épidémique (biologie) ou une diffusion de rumeur (théorie de l'information) peut aussi se modéliser sous forme de graphes où les sommets sont les agents et les arêtes les contacts possibles. Dans ce qui suit, on va se concentrer sur le processus épidémique. Deux sommets sont adjacents si ils peuvent se contacter l'un et l'autre (voisins). Le modèle est donc celui d'un graphe avec N sommets, un rayon R de contact et une surface S. Il convient alors de rajouter de l'aléa sur des contacts possibles. Selon la résistance (voire l'immunité) et/ou le taux de contact entre deux agents (A et B) qui sont des sommets voisins, on pourra dessiner une arête entre A et B avec une certaine **probabilité**. Les questions qui se posent sont alors le nombre de *clusters* (ou composantes connexes infectés), la *taille maximale* d'un cluster (nombre de sommets/agents dans le plus grand cluster).

Selon le temps qui reste aux binômes/trinômes choisissant ce sujet, on pourra même aller beaucoup plus loin: Sachant que même si les agents bougent mais uniformément au hasard dans un petit rayon (les confinés), le graphe reste structurellement le même (admis). Si on rajoute donc une dimension temporelle et qu'on dit qu'une partie non négligeable des agents se guérissent d'eux même après un certain temps, une question cruciale qui nous aidera à comprendre le contexte actuel est par exemple: "combien d'agents immunisés pourront bloquer un processus épidémique dans une population de taille N, une surface carrée S (ou hexagonale!) et un certain rayon de contact potentiel R?"

Partie algorithmique/heuristiques

Sujet 7 et 8 : Le voyageur de commerce

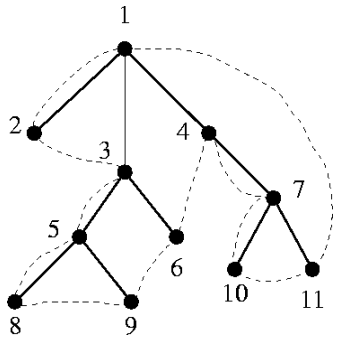
Soit n villes (A_1, … A_n) de coordonnées (xi, yi) dans le plan euclidien (limité, disons, au sous ensemble [0,L]*[0,L]. Le but est de trouver le plus court chemin qui permet de visiter chaque ville une seule fois. Par convention, on dira que l'on part de la ville A_1; on doit visiter toutes les autres puis revenir au point de départ.

Ce problème est très célèbre et malgré sa formulation simple n'a pas de solution algorithmique connue autre que le brute force (il faut alors tester toutes les permutations sur les n-1 villes restantes, c'est-à-dire une complexité en O(n-1!) ce qui est plus qu'exponentiel). Typiquement, il n'est pas possible en temps raisonnable sur un ordinateur normal de résoudre exactement pour plus que n = 15 villes. Néanmoins, il existe des solutions de complexités plus basses mais seulement approchées. On regarde ici quelques-unes d'entre elles.

Travail à faire commun aux sujets 7 et 8:
D'abord implémenter l'algorithme brute force qui renvoie la solution optimale (facile), étant donné une classe qui instancie le problème aléatoirement. A noter qu'il est trop long à évaluer dès que n est significativement plus grand que 10. On pourra aussi (mais plus tard) utiliser des librairies standard pour produire l'image des n villes et la solution optimale
(à choisir par exemple dans : https://wiki.python.org/moin/PythonGraphLibraries)

Sujet 7 : solution par arbre couvrant

Une solution approchée consiste à faire une heuristique calculant d'abord un arbre couvrant de poids minimal passant par les villes (A_1, … A_n), et offrir comme solution de visite un parcours en profondeur d'abord de l'arbre. Dans l'arbre dessiné ci-dessous en exemple dans le plan, on a alors comme parcours du voyageur de commerce 1, 2, 3, 5, 8, 9, 6, 4 7, 10, 11 pour revenir à 1.



(Rappels : si l'on trace toutes les arêtes entre chaque couple de villes (i,j) on obtient ce qu'on appelle un graphe. Il contient a priori des cycles (par exemple ville 3-4-6-3). Un arbre est un graphe dépourvu de cycles (cf. schéma ci-dessus). Un arbre de poids minimal est un donc un sous-ensemble des arêtes du graphe de départ qui atteint ("couvre") tous les points, et dont le poids est minimal. Ici le poids d'une arête entre la ville i et la ville j sera sa distance euclidienne.

Il existe plusieurs algorithmes pour déterminer l'arbre couvrant de poids minimal. Citant wikipédia, dans https://fr.wikipedia.org/wiki/Arbre_couvrant_de_poids_minimal : "Il existe de nombreux algorithmes de construction d'un arbre couvrant de poids minimal. Par exemple l'algorithme de Borůvka (le premier algorithme inventé pour ce problème), l'algorithme de Prim et l'algorithme de Kruskal." Vous choisirez l'un d'entre eux et vous l'implémenterez.

Ensuite on parcourera ce graphe en profondeur (cf TP) pour donner une solution au problème.

Enfin on étudiera l'efficacité de la méthode. L'idée est de montrer par simulation que sur une surface donnée, en générant aléatoirement le même nombre de sommets à chaque fois, la longueur du parcours ainsi obtenue est "concentrée" autour de sa moyenne. (i) On va donc générer par exemple 1000 graphes (même nombre de sommets, même surface) et calculer la moyenne M de la longueur de leur parcours. (ii) Puis, on va générer 10 graphes supplémentaires et montrer expérimentalement que sur ces 10 derniers graphes, tous les tours (longueurs du chemin du voyageur) sont proches de la moyenne M.

Sujet 8 : solution par recuit simulé

Le recuit simulé est une technique standard emprunté à la thermodynamique/physique statistique où l'on explore des solutions possibles si et seulement si c'est autorisé par "l'agitation thermique" : cela permet grosso modo de minimiser une fonction en essayant de ne pas être piégé pour toujours dans un minimum local (les "sauts thermiques" permettent de changer d'endroit de façon aléatoire" avant d'optimiser autour de ce nouvel endroit). cf. https://fr.wikipedia.org/wiki/Recuit_simulé

Plus concrètement l'algorithme est le suivant :

- On se donne une suite décroissante de températures $\{T_n\}$ qui tend vers 0.
- On choisit un chemin X_0 au hasard.
- À l'étape n+1, on choisit un chemin y voisin de X_n au sens où on a échangé 2 villes (et on a donc changé le sens du voyage entre elles).
- On tire ensuite un nombre U au hasard dans [0,1[.
- On accepte la sélection y si $U < \exp[1/T_n\ (\text{Longueur}(X_n) - \text{Longueur}(y))]$, et dans ce cas $X_{n+1} = y$, sinon on la refuse, et $X_{n+1} = X_n$.

Coder l'algorithme et explorer sa convergence en se donnant différents profils de décroissance pour la température (T_(n+1) = alpha T_n : décroissance exponentielle avec alpha < 1).

Afin d'améliorer a priori cet algorithme, on peut essayer de partir d'un chemin initial X_0 plus intelligent qu'aléatoire. On initiera le premier chemin par différentes techniques:

- Par plus proche voisin : à chaque étape i, on choisit la prochaine ville comme étant la plus proche parmi celles restantes
- Par insertion. On commence par la ville A_1 et sa plus proche voisine qu'on appelle A_k ici. On obtient un cycle A_1 -- A_k -- A_1. On cherche alors un cycle qui parcourt une ville en plus : A_1 -- A_k -- A_j --A_1, telle que parmi tous les choix possibles, cette insertion de A_j augmente le moins possible la longueur du cycle initial; puis on itère.
- Par descente locale. Etant donné un parcours de villes initial, on génère un ensemble de "solutions proches" selon une règle bien définie (par exemple, toutes les solutions obtenues par permutations de deux villes); dans cet ensemble de solutions proches, on garde la meilleure et on itère.

Coder ces trois algorithmes. On comparera ces quatre algorithmes sur N tirages de n villes (N= 1000? n= 20 ou 30? A voir selon le temps d'exécution). On regardera enfin si l'utilisation d'un de ces trois algorithmes pour déterminer un chemin initial semble augmenter ou non la performance du recuit simulé.

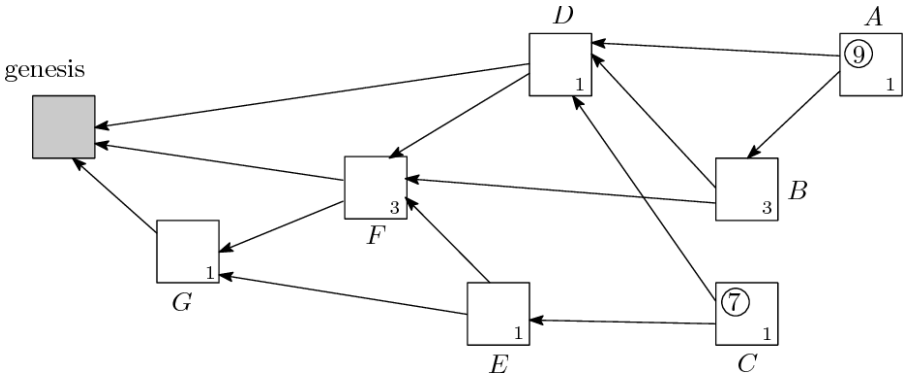
Partie algorithmique/graphes

Sujet 10 : cryptomonnaies

I - Introduction et construction du réseau

Les transactions de cryptomonnaies (Bitcoins, Ethereum, Ripple, ...) ne se font pas par l'intermédiaire d'une banque comme une monnaie classique mais en s'inscrivant dans des registres dématérialisés qui sont en fait des arbres. Les structures "classiques" contiennent deux types de participants, ceux qui créent les transactions, les clients, et ceux qui les approuvent, les miners (appelés ainsi car pour chaque

transaction approuvée, le système crée ainsi un peu de crypto-monnaie et leur donne en récompense). On étudie ici, un autre type d'arbre nommé Directed Acyclic Graph (DAG). Chaque nœud de cet arbre représente une transaction faite par un utilisateur et chaque utilisateur doit approuver k autres transactions pour enregistrer la sienne. A un instant t, toutes les transactions n'ayant pas été approuvées sont appelées *pointes* du réseau. Pour être acceptée, sa transaction doit être à son tour approuvée directement ou indirectement (i.e. par l'intermédiaire d'autres utilisateurs) par les utilisateurs suivants. Certaines transactions peuvent cependant être conflictuelles (utilisateur dépensant plus d'argent qu'il n'en ont). Elles seront alors moins approuvées et pénaliseront aussi les utilisateurs ayant approuvé ces transactions. Ainsi sont écartées les transactions litigieuses. Le registre se termine et se synchronise à l'activation d'un nonce après un certain temps.
Pour optimiser un tel arbre, nous allons créer des paramètres pertinents qui serviront de bases à l'optimisation. On considère alors les trois paramètres suivants :
❇ *Le poids d'une transaction* : Chaque transaction se verra attribué un entier n qui va définir son importance, distribué aléatoirement entre les transactions.
❇ *Le poids cumulé* : Il s'agit de la somme des poids des transactions ayant approuvé, directement ou indirectement, la transaction.
❇ *Le score* : Il s'agit de la somme des poids des transactions approuvées, directement ou indirectement, par la transaction.
Ces trois paramètres serviront à définir un algorithme de sélection de pointes. Notamment, ils serviront à affaiblir les transactions conflictuelles et renforcer les transactions honnêtes.

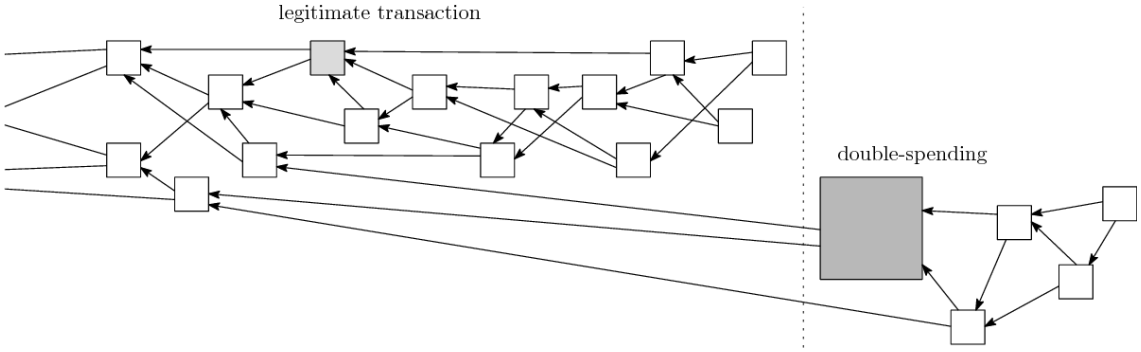


➥ **Réaliser un premier modèle d'un tel arbre pour k = 2. Réfléchissez au moyen le plus clair de faire afficher votre arbre par Python. Construisez un modèle de sélection de pointe probabiliste. Si vous les trouvez pertinents, vous êtes invité à ajouter d'autres paramètres.**

Première variante du sujet ou sujet n°1 : II - Attaque : La double transaction surpondérée.

Lors d'une attaque à double transaction, l'attaquant commence par réaliser une transaction classique avec un marchand, et le marchand décide de lui envoyer l'objet de la vente une fois que la transaction a acquis un certain poids cumulé. Ensuite, l'attaquant crée une autre transaction avec la même monnaie. Il crée ensuite plein de petites transactions qui vont approuver, directement ou indirectement, la transaction frauduleuse et donc lui donner un gros poids cumulé. La transaction frauduleuse ne doit pas approuvée la première transaction honnête. Si la transaction frauduleuse et son sous réseau se développe assez pour

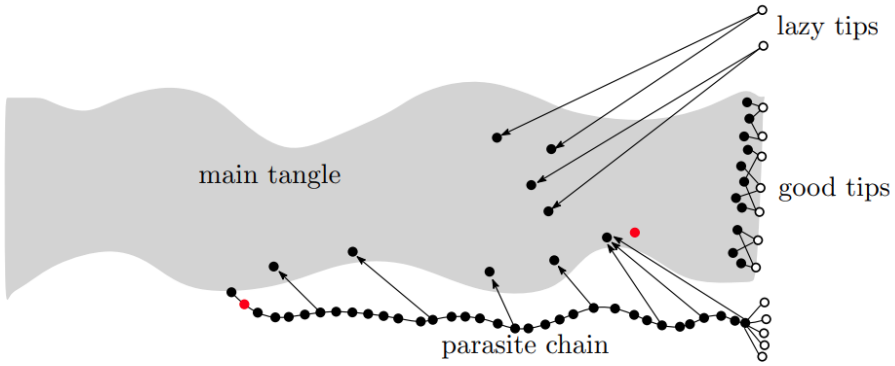
laisser la transaction honnête orpheline, celle-ci ne sera pas approuvée par la suite et l'argent n'ira pas au marchand lors de l'approbation du nonce.



➥ **Simulez une telle attaque et estimez ses chances et conditions de succès. Optimisez vos algorithmes de sélection de pointe pour diminuer ses chances.**

Deuxième variante du sujet ou sujet n°2 : II - Attaque : La double transaction par chaîne parasite.

Un attaquant construit une chaîne en secret et l'insert le long du réseau principal en référençant des transactions de manière ponctuelle. Ainsi le score des transactions de l'attaquant contient celui de la chaîne parasite en plus de celui du réseau principal. L'attaquant glisse alors de fausses transactions dans le réseau principal. A un instant t, il utilise le poids de sa chaine pour faire des transactions au score plus élevé et faire en sorte que les pointes du réseau principal ne soient plus choisies pour approbation par les nouvelles transactions. Les transactions récentes du réseau principal deviennent orphelines.



➥ **Simulez une telle attaque et estimez ses chances et conditions de succès . Optimisez vos algorithmes de sélection de pointe pour diminuer ses chances**

Partie jeux/intelligence artificielle:

Sujet 11 : joueurs optimaux au morpion 3*3 par la méthode du minimax:

On réutilisera directement les éléments pertinents de la classe codée en TP2.

Idée : lorsque c'est au tour d'un des deux joueurs, celui-ci à le choix entre plusieurs cases. Cela définit des noeuds fils dans l'arbre des parties possibles restantes. Si le joueur choisit un des fils, alors c'est au tour du second joueur, dont les actions sont elles-mêmes représentées par des noeuds fils. Chaque noeud doit contenir en valeur deux informations : l'état du jeu d'une part, et quel joueur doit jouer le prochain coup. On obtient ainsi un arbre. Dans le cas du morpion, l'arbre complet à partir du noeud racine n'est pas immense, et on peut le développer et l'explorer entièrement afin de produire un joueur qui joue optimalement. Ceci étant dit, la question demeure de comment choisir l'action optimale pour le joueur ordinateur étant donné cet arbre. L'algorithme dit "minimax" permet cela.

Le principe du minimax est de choisir pour le joueur A le fils correspond à une valeur maximale, et supposant que l'adversaire au coup d'après choisira le fils qui donnera au joueur A une valeur minimale, etc. Cette valeur ne peut s'évaluer que sur les feuilles : 1 pour A si A gagne, - 1 s'il perd, 0 si égalité. Se renseigner d'abord plus généralement sur l'algorithme minimax (https://fr.wikipedia.org/wiki/Algorithme_minimax). Il vous faudra pouvoir à chaque étape du jeu définir l'arbre restant des parties possibles, et donc avoir une classe arbre qui prend en entrée l'état du plateau et le joueur dont c'est le tour. Ensuite il faudra explorer récursivement l'arbre pour remonter les valeurs de parties finales, afin que A prenne la décision optimale (c'est-à-dire, étant donné cet algorithme, en présupposant que l'adversaire jouera lui aussi de façon optimale en minimisant votre espérance).

Vous vérifierez que dans une partie d'un joueur optimal contre un autre, la partie finit toujours par un nul.

Bonus si le temps le permet : Qu'en est il en 4 cases par 4 cases (où vous gagnez dès que trois cases alignées sont remplies de votre symbole)? On se convaincra d'abord à la main que le joueur qui commence peut systématiquement gagner en trois coups. Votre algorithme doit retrouver cela.

Sujet 12 : méthode Monte Carlo simple (Lire pour info le sujet 11).

Dans le cas de jeux plus complexes, l'arbre ne peut pas être exploré en entier dans un temps raisonnable (en particulier aux échecs ou au jeu de Go). On peut alors réaliser un minimax jusqu'à une certaine profondeur. Mais on peut aussi faire une estimation par la méthode de Monte Carlo, ce que l'on fera ici.

L'idée est que le joueur A a le choix entre des fils A_1, … A_n. Il ne connaît pas la valeur de ces choix, mais va les estimer par Monte Carlo, c'est-dire qu'étant donné un choix pris A_(i_o), il simule N parties aléatoires où les deux joueurs prennent des décisions arbitraires jusqu'à la fin de partie. Pour chaque fin de partie, on incrémente de 1 la valeur du fils A_(i_0) si A gagne, de -1 si A perd, de zéro si partie nulle. Si N est suffisamment grand, on a a priori une bonne estimation de la valeur du coup joué A_(i_0). Le joueur A décide alors de jouer le coup A_i qui a la valeur maximale.

Ce cadre marche mal avec le morpion 3*3 car ce jeu a une écrasante majorité de parties nulles par rapport à des parties gagnantes ou perdantes. On va donc passer sur un morpion 5*5, où un joueur gagne s'il y a quatre ronds ou quatre croix à la suite (en ligne, colonnes, ou diagonales). (NB : si vous voulez coder autre chose que le morpion, vous le pouvez, mais attention à ne pas prendre un jeu trop ambitieux qui

vous demanderait trop de temps! Par exemple, coder un jeu d'échecs avec toutes ses règles, ce n'est pas difficile, mais c'est trop long).

- Il faudra dans un premier temps généraliser la classe vue en TP2.
- Il faudra ensuite ajouter une méthode MonteCarlo(self, N, fils) qui renvoie la valeur moyenne du fils A_i de l'état A étant donné N simulations. Ici, il n'est pas nécessaire de coder la structure en arbre.
- Ajouter une méthode "jouer" qui permet de faire jouer un ordinateur ayant le droit à n_1 simulations par fils contre un autre ayant le droit de jouer n_2 simulations. (si n=0, l'ordinateur joue aléatoirement).
- Montrer que l'ordinateur 1, ayant n1 significativement plus grand que n2, gagne en moyenne contre l'ordinateur 2. Estimer le nombre de parties à jouer pour avoir une estimation fiable du taux de victoire de 1 contre 2. (NB : commencer est a priori un avantage, donc on fera une partie sur deux où c'est le joueur 1 qui commence, une autre où c'est le joueur 2).
- Le mieux sera à la fin de procéder à une échelle de score en ELO-scores (comme pour le classement international des joueurs d'échecs) : mettons que le joueur aléatoire a un ELO score de 1000 par définition. Un joueur qui gagne dans x % des cas contre le joueur aléatoire a alors un score 1000 + D, avec la relation x = 1/(1+10^(-D/400)). Le 400 est conventionnel : il signifie qu'un joueur ayant un score de 1400 gagne dans x = 1/1.1 = 91% des cas contre le joueur ayant un score de 1000, celui ayant 1800 gagne dans 91% des cas contre celui à 1400, etc...
  (cf wikipédia https://fr.wikipedia.org/wiki/Classement_Elo)
- Réaliser suffisamment de simulations pour pouvoir tracer la courbe ELO-score = f(n).

Bonus : afin d'améliorer les deux joueurs et la méthode en général, implémenter le fait qu'un joueur contre son adversaire s'il peut perdre au prochain coup, et qu'un joueur prend le coup gagnant s'il y en a un (dans ce cas, sans passer par la case Monte Carlo).

(Bonus 2 : se renseigner sur une version améliorée dite Monte Carlo Tree Search, et en décrire le principe dans votre présentation (mais l'implémenter est un peu trop long...).)

Sujet 13 : bataille navale.

Première partie : création d'un code qui permette de jouer une partie ordinateur contre ordinateur

Créer une bataille navale à l'aide d'une classe qui permet de faire jouer l'une contre l'autre différentes stratégies et de renvoyer le gagnant. Pour rappel le jeu est une matrice 10*10; les bateaux n'ont pas le droit d'être collés les uns aux autres mais peuvent se "joindre" en diagonale.

Ordinateur niveau 0 : le choix de la case à tirer est aléatoire
Ordinateur niveau 1 : une fois un bateau découvert, il est coulé avant de tirer de nouveau aléatoirement
Ordinateur niveau 2 : les cases adjacentes à un bateau coulé sont reconnues comme vides.
Ordinateur niveau 3 : une stratégie plus intelligente de recherche des bateaux : on vous laisse y réfléchir.

- Vérifier à chaque implémentation de stratégies que le code fonctionne bien

- Faire jouer les stratégies les unes contre les autres un grand nombre de fois pour vérifier que chaque amélioration augmente les chances de gagner

Deuxième partie : implémentation du jeu contre un humain

On vous fournit un code d'interface graphique en Python. Vous devez coder en utilisant ce que vous avez fait en première partie un jeu entre un humain et un ordinateur.