Problème 1 : Tournée du jardinier

Ninon Mouhat-Courvoisier
Bastien Brogliato-Martin
Romain Iavarone–Michez
Nathaël Panel-Roques
Titouan Vial

11 avril 2024

1 Introduction

Dans ce rapport, on s'intéresse à l'exercice de la tournée du jardinier. En reformulant l'énoncé, nous cherchons donc à minimiser une distance à parcourir en passant par tous les points d'un plan. Deux approches ont naturellement émergé au sein de notre groupe : l'approche géométrique, et l'étude de graphes. Une dernière approche est arrivé plus tard au cours du projet et consiste à utiliser l'aléatoire. Chacune de ces approches a donné lieu à des algorithmes très différents, que nous aborderons dans la suite de ce rapport.

2 Approche géométrique

Les programmes implémentant ces algorithmes sont situés dans les fichiers `convex_hull.py` (pour le *parcours de Graham*) et `convex_path.py` (pour la transformation de l'enveloppe en chemin couvrant).

2.1 Réflexions et heuristique

L'approche géométrique fut inspirée par une réflexion sur la relation entre le périmètre et l'aire d'une forme géométrique donnée, avec l'idée que le problème de couvrir l'ensemble des points d'un jeu de données en minimisant la distance parcourue soit similaire au problème de minimiser le périmètre d'une figure à aire fixée.

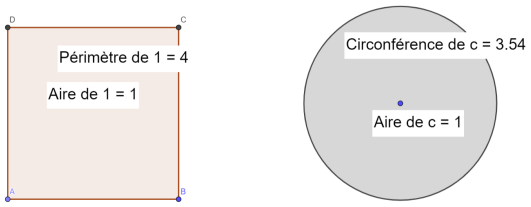


FIGURE 1 – Comparaison, à aire fixée, du périmètre d'un carré et d'un cercle

Une propriété intéressante de la géométrie euclidienne est que, à aire fixée, la figure géométrique dont le périmètre est le plus faible est le cercle (cf figure 1). De plus, si l'on souhaite se limiter au

cas d'un polygone à n côtés (avec $n \geq 3$), celui dont le périmètre est le plus faible à aire fixée est le polygone régulier, donc convexe, à n côtés. En général, pour deux figures de même aire, la plus convexe des deux aura le périmètre le plus faible.

Ainsi, nous avons fait la supposition que plus un chemin donné serait « globalement convexe », plus il serait optimal, jusqu'à une certaine limite. Formellement, cela se traduirait par l'idée que, plus le chemin en question est proche d'une enveloppe convexe de l'ensemble des points, plus il sera proche d'une solution optimale.

2.2 Détermination de l'enveloppe convexe

La première étape de l'algorithme géométrique est la détermination de l'enveloppe convexe de l'ensemble $Q = P \cup \{(0,0)\}$, avec P l'ensemble des points du problème.

Pour cela, nous utilisons l'algorithme du *parcours de Graham*, un algorithme de détermination de l'enveloppe convexe d'un ensemble de points de $\mathbb{R}^2$ de complexité $\mathcal{O}(n\log(n))$, avec $n = \text{card}(Q)$, celle-ci étant dominée par la complexité du tri des points.

Algorithme du *parcours de Graham*

Ici, Q désigne un tableau (donc indicé) plutôt qu'un ensemble.

- **Initialisation :**
  - On détermine le pivot $p_0 = (x_0, y_0) \in Q$ tel que $y_0 = \min\limits_{(x,y)\in Q}(y)$ et tel que $x_0 = \min\limits_{\substack{(x,y)\in Q \\ y=y_0}}(x)$. p_0 est placé comme premier élément de Q.
  - On trie le reste de Q selon l'angle entre le segment reliant chaque point à p_0 et la droite d'équation $y = y_0$, mesuré dans le sens trigonométrique. On note désormais $Q = [p_0, \ldots, p_{\text{card}(Q)-1}] = [(x_0, y_0), \ldots, (x_{\text{card}(Q)-1}, y_{\text{card}(Q)-1})]$.
  - On initialise E le tableau représentant les points de l'enveloppe, dans l'ordre, avec comme deux premiers éléments les deux premiers éléments de Q (le pivot et le point tel que le segment les reliant face l'angle minimal avec l'horizontale).
- **Boucle :**
  - **Pour** k variant de 2 à $\text{card}(Q) - 1$:
    - On note les deux derniers éléments de E e_{-2} et e_{-1}.
      De plus, on note $e_{-2} = (u_1, v_1)$ et $e_{-1} = (u_2, v_2)$.
    - On calcule le produit vectoriel R_T :

$$R_T := \overrightarrow{e_{-2}e_{-1}} \wedge \overrightarrow{e_{-2}p_k} = (u_2 - u_1)(y_k - v_1) - (u_2 - v_1)(x_k - u_1)$$

    - **Tant que** $R_T < 0$ (« tournant à droite »), on supprime le dernier élément de E, puis on recalcule R_T avec les nouvelles valeurs de e_{-2} et e_{-1}.
    - Dès que $R_T \geq 0$ (« tournant à gauche »), on ajoute p_k à la fin de E.

2.3 Transformation de l'enveloppe convexe en chemin

Une fois l'enveloppe convexe déterminée, on obtient le chemin final en itérant sur les éléments de Q ne faisant pas partie de E. Tant qu'il existe des éléments de Q qui ne sont pas dans E, on détermine le point p et son emplacement i_p dans le chemin qui minimise la longueur du chemin après ajout, puis on ajoute p à E à l'indice i_p (ie, on chercher la façon d'ajouter un point à E en minimisant la longueur rajoutée par cet ajout).

Une fois que tous les points de Q sont dans E, on considère le chemin complété. Sachant que Q contient tous les points de P et $(0,0)$, l'entrée, on finit le calcul du chemin en ré-indexant

E de telle sorte à ce que $(0, 0)$ soit son premier élément à l'aide d'une permutation circulaire (pour conserver l'ordre du chemin), puis on le supprime de E, le résultat demandé ne devant pas comprendre l'entrée du parc.

2.4 Complexité

L'algorithme final a donc une complexité en $\mathcal{O}(n^3)$, avec n le nombre de points de P (rigoureusement, on aurait plutôt ici du $\mathcal{O}((n+1)^3)$ car $\text{card}(Q) = n + 1$, mais ceci est équivalent à du $\mathcal{O}(n^3)$). En effet, la transformation de l'enveloppe en chemin demande au total trois boucles imbriqués toutes de tailles proportionnelles à n, induisant une complexité de cette étape en $\mathcal{O}(n^3)$. La complexité du *parcours de Graham* en $\mathcal{O}(n \log(n))$ est alors dominée asymptotiquement par la complexité du reste de l'algorithme, aboutissant finalement à une complexité théorique en $\mathcal{O}(n^3)$.

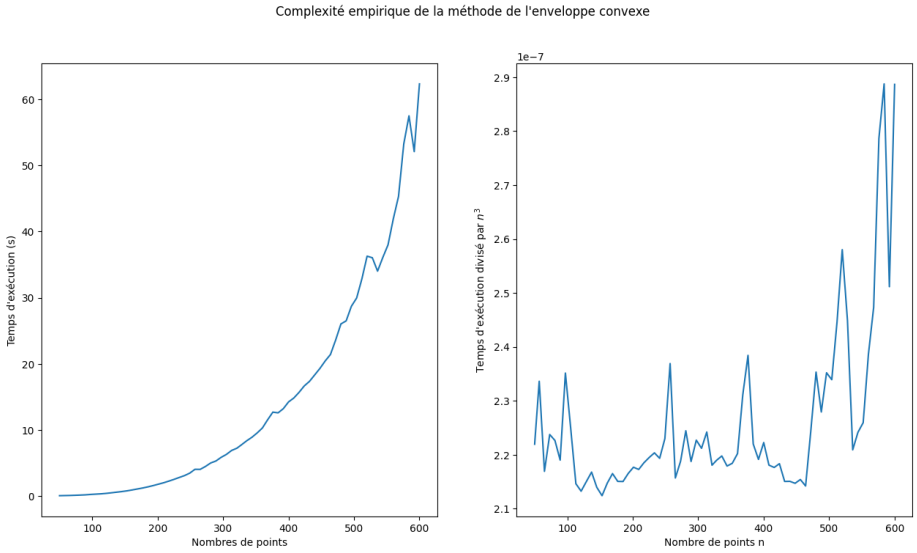


FIGURE 2 – Complexité empirique de la méthode de l'enveloppe convexe

De plus, comme visible sur la figure 2 ci-dessus, la complexité empirique semble être conforme à une complexité en $\mathcal{O}(n^3)$, malgré tout de même des irrégularités pour les grandes valeurs de n.

3 Graphes

Notons P l'ensemble des points à visiter dans le plan. On note ainsi les coordonnées : $\forall p \in P, \exists (x, y) \in \mathbb{R}$, tel que $p = (x, y)$.
Tout d'abord, les algorithmes de cette section se basent sur les propositions suivantes :
- Il est sous-optimal de tracer un schéma comportant des boucles.
- Il est sous-optimal de tracer un schéma comportant des retour sur ses pas.

On en déduit que le chemin optimal est toujours une boucle partant et revenant à l'entrée. Cela implique qu'avec une vision des graphes, chaque sommet du chemin final est de degré 2.

3.1 Algorithme Naïf

Le nom de cet algorithme est inspiré de son fonctionnement que l'on considérera comme naïf. On commence par établir un tableau de distances entre les points. Ensuite on identifie la ou les distances les plus courtes du tableau. Une fois cela fait on relie le ou les points qui vérifient la condition précédente, si et seulement si, aucun des 2 points à relier n'est de degré 2 et que cette

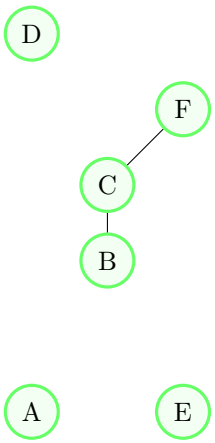
liaison ne crée pas de boucle. Puis on recommence l'opération, en enlevant du tableau la ou les distances des points qu'on a reliés, jusqu'à ce que tous les points soient de degré 2.

Nous allons maintenant nous baser sur l'exemple de l'énoncé pour montrer comment aurait fonctionner cet algorithme et quel longueur de chemin aurait-il produit.

Ici les distances les plus courtes du tableau sont les distances BC et CF. Donc on commencera par relier les points B et C et C et F.

	A	B	C	D	E	F
A	-	$\sqrt{5}$	$\sqrt{10}$	4	2	$\sqrt{13}$
B	$\sqrt{5}$	-	1	$\sqrt{5}$	$\sqrt{5}$	$\sqrt{2}$
C	$\sqrt{10}$	1	-	$\sqrt{2}$	$\sqrt{10}$	1
D	4	$\sqrt{5}$	$\sqrt{2}$	-	$\sqrt{20}$	$\sqrt{5}$
E	2	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{20}$	-	3
F	$\sqrt{13}$	$\sqrt{2}$	1	$\sqrt{5}$	3	-

(a) Distance entre les points



(b) Première étape Algorithme Naïf

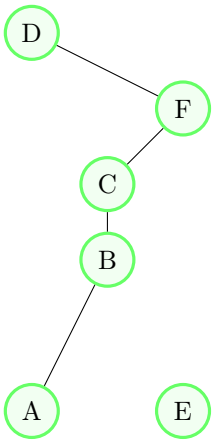
Ensuite on remarque maintenant que les nouvelles distances les plus courtes ($\sqrt{2}$) sont les distances DC et BF. Or C est de degré 2 donc on ne relie pas C à D. De plus, relier B à F formerait une boucle donc cette liaison n'est pas possible non plus.

On continue et maintenant les distances les plus courtes ($\sqrt{5}$) sont AB, BD, BE et DF. L'algorithme va fonctionner de telle sorte à en prendre une aléatoirement parmi la liste. Considérons qu'il choisit en premier DF, donc D devient de degré 2 donc il reste que AB ou BE. Choisissons AB ce qui élimine BE car B devient de degré 2.

On actualise le tableau et le graphe :

	A	B	C	D	E	F
A	-	-	-	4	2	-
B	-	-	-	-	-	-
C	-	-	-	-	-	-
D	4	-	-	-	$\sqrt{20}$	-
E	2	-	-	$\sqrt{20}$	-	-
F	-	-	-	-	-	-

(a) Distance entre les points



(b) Deuxième étape Algorithme Naïf

Ainsi la distance la plus courte devient AE. Donc à l'itération d'après il ne restera que DE à relier. Ainsi grâce à cette algorithme on obtient le chemin suivant :

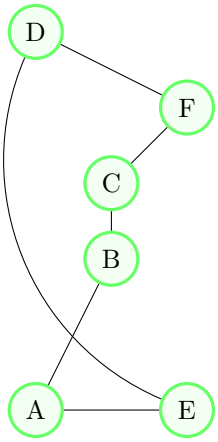


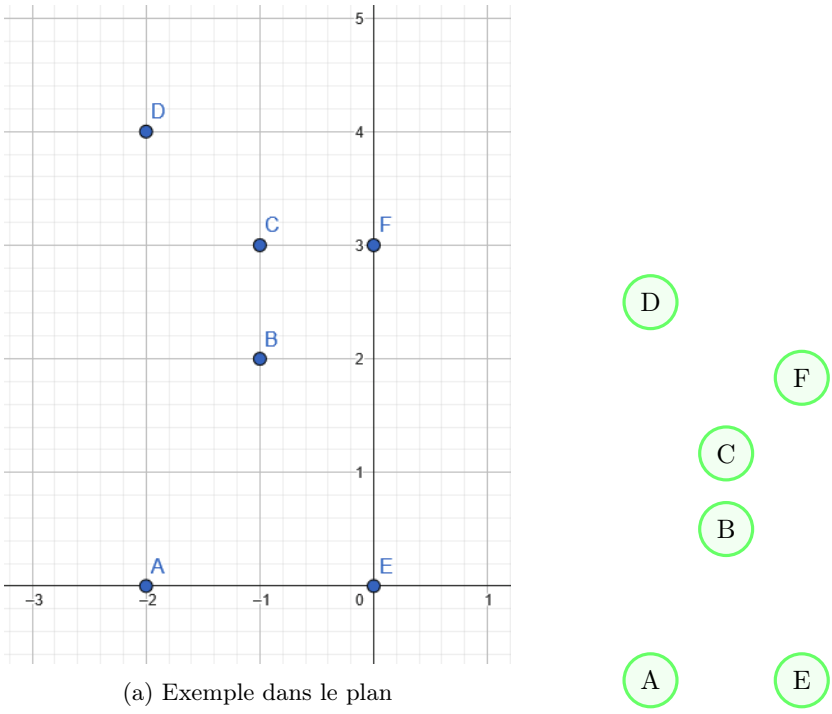
FIGURE 5 – Graphe final

Sa longueur est de 12.94 ce qui est sous-optimal puisque le prochain algorithme va nous permettre d'améliorer cette longueur.

3.2 Algorithme des Espèces en Danger

L'implémentation de cet algorithme est disponible dans le fichier `especes_en_danger.py`. Le nom de celui-ci est une invention de notre groupe.

Cet algorithme repose sur le fait que tout degré d'un sommet du chemin final est 2. On va expliquer cet algorithme à travers l'exemple suivant, illustré sous forme de graphe et dans le plan :



(a) Exemple dans le plan

FIGURE 6 – Situation de l'exemple

On va compléter ce graphe avec des arêtes pendant le déroulement de l'algorithme. Remplissons tout d'abord le tableau suivant, qui répertorie tous les poids (les distances) des arêtes encore

traçables. A toute étape, une arête traçable est une arête qui n'a pas encore été tracée, qui ne forme pas de boucle en l'ajoutant un graphe, et qui ne fait pas dépasser les degrés de ses sommets 2.

	A	B	C	D	E	F
A	-	$\sqrt{5}$	$\sqrt{10}$	4	2	$\sqrt{13}$
B	$\sqrt{5}$	-	1	$\sqrt{5}$	$\sqrt{5}$	$\sqrt{2}$
C	$\sqrt{10}$	1	-	$\sqrt{2}$	$\sqrt{10}$	1
D	4	$\sqrt{5}$	$\sqrt{2}$	-	$\sqrt{20}$	$\sqrt{5}$
E	2	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{20}$	-	3
F	$\sqrt{13}$	$\sqrt{2}$	1	$\sqrt{5}$	3	-

TABLE 1 – Distance entre les points

On va regarder, pour chaque point, quelles sont les 2 distances les plus grandes. Cela revient à regarder le pire choix de couple d'arêtes pour chaque sommet. Voici le tableau qu'on obtient :

	A	B	C	D	E	F
max1	4	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{20}$	$\sqrt{20}$	$\sqrt{13}$
max2	$\sqrt{13}$	$\sqrt{5}$	$\sqrt{10}$	4	$\sqrt{10}$	3
somme	$4+\sqrt{13}$	$2\sqrt{5}$	$2\sqrt{10}$	$4+\sqrt{20}$	$\sqrt{20}+\sqrt{10}$	$3+\sqrt{13}$

TABLE 2 – Calcul des 2 sommets les plus éloignés pour chaque sommet (itération 1)

Donc, ici le sommet le plus "en danger" (qui a la plus grande somme) est D. Pour éviter que son éloignement ne provoque une trop grande distance à parcourir dans le chemin final, on va tracer sa plus petite arête possible, c'est-à-dire : l'arête DC. On trace donc l'arête DC :

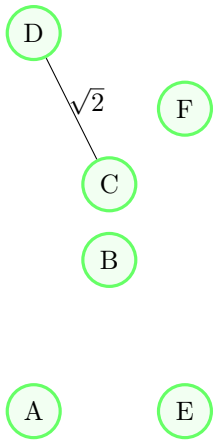


FIGURE 7 – Graphe 1ère itération

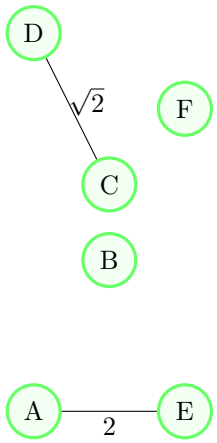
Ensuite, on met à jour notre tableau d'arêtes encore possibles à tracer. Pour l'instant, aucun sommet n'a atteint le degré 2, aucune boucle ne peut être formée, donc on enlève seulement DC. De même, le tableau des sommes maximale doit être mis à jour, puisque D et C ont maintenant une arête "imposée", de taille $\sqrt{2}$.

Puis on continue à chercher, à chaque étape, le sommet le plus "en danger". Cette fois-ci, c'est E. On trace donc l'arête EA de poids 2.

On met à jour nos deux tableaux. On retire seulement EA dans le premier. Le deuxième devient :

	A	B	C	D	E	F
max1	4	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{20}$	$\sqrt{20}$	$\sqrt{13}$
max2	$\sqrt{13}$	$\sqrt{5}$	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{10}$	3
somme	$4+\sqrt{13}$	$2\sqrt{5}$	$\sqrt{2}+\sqrt{10}$	$\sqrt{2}+\sqrt{20}$	$\sqrt{20}+\sqrt{10}$	$3+\sqrt{13}$

(a) Calcul du point le plus éloigné des autres points (itération 2)

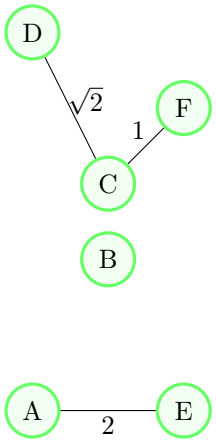


(b) Graphe 2ème itération

Le prochain sommet choisi est F. On trace donc FC. On met à jour les 2 tableaux, sachant que C a atteint le degré 2 :

	A	B	C	D	E	F
A	-	$\sqrt{5}$	-	4	-	$\sqrt{13}$
B	$\sqrt{5}$	-	-	$\sqrt{5}$	$\sqrt{5}$	$\sqrt{2}$
C	-	-	-	-	-	-
D	4	$\sqrt{5}$	-	-	$\sqrt{20}$	-
E	-	$\sqrt{5}$	-	$\sqrt{20}$	-	3
F	$\sqrt{13}$	$\sqrt{2}$	-	-	3	-

(a) Distance entre les points



(b) Graphe 3ème itération

Le prochain point choisi est E. On trace l'arête EB de poids $\sqrt{5}$. Puis on continue : on choisit A, et on trace AF. On choisit trace enfin BD.

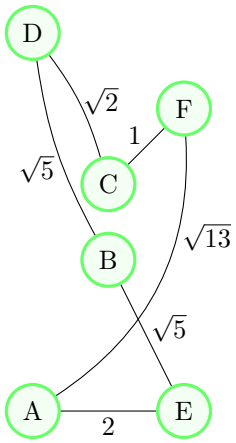


FIGURE 10 – Graphe final

Ce graphe final nous permet d'obtenir une distance totale de 12,49, ce qui est une légère amélioration par rapport à l'algorithme naïf. Dans la pratique, nous avons utilisé cet algorithme suivi de celui qui permet de "démêler" le chemin créé (décrit en section 5), qui améliore notre résultat.

Cet algorithme semble avoir une complexité qui dépend du nombre de données en $\mathcal{O}(n^3)$ et de la disposition des données. C'est pour cela qu'on obtient dans le graphique (figure 11) suivant une distribution qui ressemble à du $\mathcal{O}(n^3)$ mais qui varie aléatoirement autour de cette distribution (les points sont placés aléatoirement à chaque test).

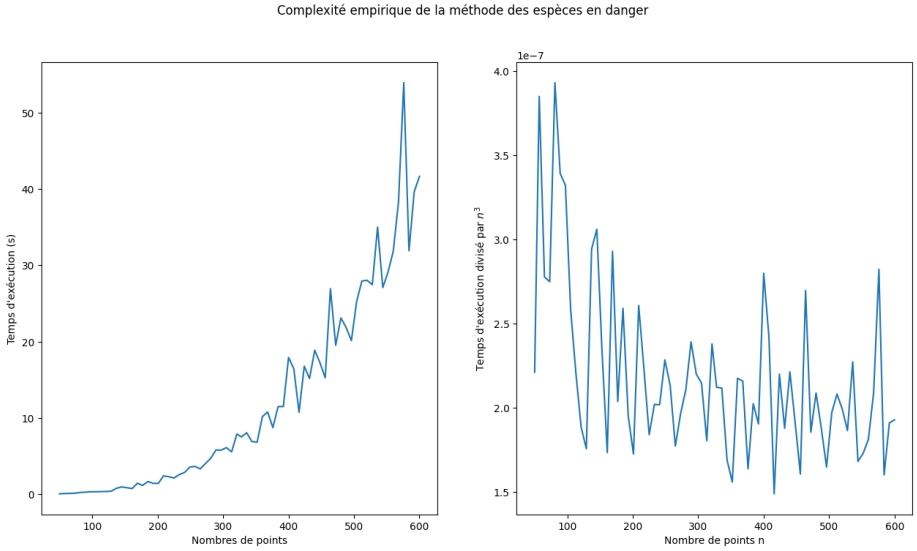


FIGURE 11 – Complexité empirique de la méthode des espèces en danger

4 Approche aléatoire

4.1 Origine des idées utilisées

L'idée derrière l'algorithme qui va être décrit dans cette partie a deux sources principales :

- le cours d'apprentissage par renforcement suivi en majeure informatique au 1er semestre de deuxième année, qui a fait émerger l'idée d'utiliser une marche aléatoire pour déterminer la direction la plus favorable à un court chemin pour un point donné
- la vidéo Coding Adventure: Ant and Slime Simulations du youtubeur Sebastian Lague, qui affine l'idée de marche aléatoire en un effet de "colonie" ressemblant au comportement simplifié d'une fourmi

4.2 Explication de l'algorithme

Cet algorithme se décompose en deux phases principales : la détermination d'un chemin aléatoire, mais influencé par des paramètres, et l'utilisation de chemins précédemment trouvés pour faire évoluer les paramètres. L'algorithme a besoin, pour fonctionner comme on l'attend, que le point $(0, 0)$ fasse partie des données à traiter.

Les étapes suivies par l'algorithme sont les suivantes :

- **Initialisation :**
  On crée au préalable une matrice des distances entre chaque point de nos données, notée `dist_mat`, on dispose alors d'une matrice de dimension (n, n) où n est le nombre de point dans nos données. On crée aussi une matrice de poids, notée `weights_mat`, de dimension (n, n), dont toutes les valeurs initiales sont 1.
- **Recherche d'un chemin :**
  1. On place tout d'abord une fourmi sur un point choisi aléatoirement parmi nos données. On rend ce point inaccessible pour le futur (la fourmi ne reviendra jamais sur un point qu'elle a déjà visité).
  2. On récupère la distance du point où on a placé la fourmi, d'indice noté i, à tous les autres points que l'on note `dist_arr`. Ces distances correspondent à la ligne i de `dist_mat`. De même, on récupère la ligne i de `weights_mat` que l'on appelle `weights_arr`.
  3. On effectue le produit terme à terme de `dist_arr` dont on a inversé les valeurs (application de la fonction $x \mapsto \frac{1}{x}$) avec `weight_arr`, en les pondérant chacune par des puissances qui seront à choisir en paramètre (on les note respectivement `dist_pow` et `weight_pow`). On obtient alors un vecteur ligne qui correspond à l'inverse de l'éloignement aux autres points de la fourmi pondéré par des poids (dont le sens n'est pas encore défini).
  4. On normalise le résultat obtenu en utilisant la norme 1 de $\mathbb{R}^n$, donc en divisant le vecteur obtenu par la somme de tous ses termes. On va utiliser ce vecteur comme une distribution de probabilité d'aller sur un des points où la fourmi n'est pas. Si la valeur obtenu pour un point est grande, la probabilité qu'on se rende sur ce point est grande (le point est alors soit proche de la fourmi, soit hautement mis en avant par les poids, voire les deux).
  5. On fait donc un choix aléatoire parmi tous les points où la fourmi n'est pas mais avec une pondération donnée par le vecteur calculé précédemment. Finalement la fourmi se déplace sur ce point, qui est rendu inaccessible pour le futur, et on réitère le processus à partir de l'étape 2 jusqu'à avoir exploré tous les points disponibles.
  6. On ferme le chemin en rajoutant comme dernière étape de revenir au premier point choisi.
- **Modification des paramètres :**
  Le paramètre variable de notre algorithme est la matrice de poids `weights_mat`. Une fois que l'on a simulé plusieurs recherches de chemin, on récupère les couples de déplacements

effectués (de la forme $(\text{point}_a, \text{point}_b)$) que l'on transforme en couple d'indice de la forme (i, j) où i est la position de point_a et j est la position de point_b dans les données initiales. On ajoute alors à `weights_mat` une valeur arbitraire à toutes les positions déterminées par les couples d'indices calculés.

Cette valeur correspond aux "phéromones" que laisse une fourmi sur son passage, et va donc mener les prochaines fourmis à emprunter les chemins où la valeur de ces phéromones est grande (ce qu'on retrouve à l'étape 4 de calcul des probabilités associées aux points des données pour le déplacement de la fourmi).

Aussi, les phéromones ne persistent pas indéfiniment, on réduit donc toutes les valeurs de `weights_mat` arbitrairement, dans notre cas en y appliquant un coefficient multiplicateur de 0.8.

— **Détermination finale du chemin :**
En répétant les étapes de détermination d'un chemin et d'ajustement des paramètres suffisamment de fois, on considère `weights_mat` converge vers un état qui permet de trouver un chemin minimisant la distance. Pour trouver le chemin final, on simule donc une dernière fois le chemin d'une fourmi et on le stocke.

On remarque en testant que le chemin peut contenir (ou non) des intersections dans les déplacements effectués, ce que l'on sait suboptimal. On applique donc l'algorithme de démêlage pour réduire la distance totale du chemin trouvé par les fourmis. On renvoie alors le chemin ainsi obtenu, ce qui conclut l'algorithme.

L'algorithme est implémenté dans le fichier `marche_aléatoire.py`.

4.3 Paramétrage de l'algorithme

Tel qu'il est décrit, l'algorithme présente plusieurs paramètres à choisir en dehors de la matrice `weights_mat` qui est directement gérée lors des étapes de recherche. On détaille donc quels sont ces paramètres :

— le nombre "d'époque" où les fourmis sont simulées, que l'on note `nb_era`, cela correspond au nombre de fois que l'on va simuler une colonie de fourmi et mettre à jour `weights_mat` avec la totalité des chemins parcourus lors de l'époque

— le nombre de fourmis simulées par époque, que l'on note `nb_sim_per_era`

— les puissances que l'on applique à `dist_arr` et `weight_arr` notées respectivement `dist_pow` et `weight_pow`

— les valeurs d'intensité et de dispersion dans le temps des phéromones, que l'on choisit de fixer respectivement à 10 et 0.8, cela permettant aux fourmis de laisser une trace sensible mais dont l'effet s'estompe suffisamment rapidement aussi (en quelques époques)

On voit donc que l'on peut fixer certains de ces paramètres, les autres seront à ajuster selon les données et les comportements que l'on veut donner aux colonies de fourmis. Pour éviter un "apprentissage" trop aléatoire on garde généralement `nb_sim_per_era` raisonnablement petit (entre 10 et 20 typiquement), et on gère la durée de l'apprentissage avec `nb_era` (dont il ne faut pas prendre une valeur trop grande, les fourmis n'apprenant plus vraiment de toute manière, et pour éviter de renforcer les phéromones sur certains chemins trop favorisés, parfois à tort. On choisit une valeur inférieure à 100 pour les jeux de données de l'énoncé ou choisis aléatoirement).

Les deux derniers paramètres, `dist_pow` et `weight_pow` sont à adapter selon la topologie des données, mais on peut essayer d'en généraliser la grandeur d'une relativement à l'autre. On veut en général que les fourmis continuent d'essayer des chemins non parcourus par leurs prédécesseurs, on veut donc `dist_pow` suffisamment important par rapport à `weight_pow`, par exemple 5 à 10 fois plus grand. Mais on veut aussi que les fourmis profitent du chemin déjà parcourus, donc on ne peut pas prendre `weight_pow` trop petit. Pour nos données, on choisit `dist_pow=11` et `weight_pow=2`, ce qui permet aux chemins de converger généralement mais d'avoir de l'exploration même après les premières époques.

On choisit en parallèle les valeurs `nb_era=65` (que l'on peut faire varier de quelques unités pour changer légèrement les rendus des colonies) et `nb_sim_per_era=20` qui semble donner les meilleurs résultats.

4.4 Complexité théorique et critique

On créé au début de l'algorithme deux matrices de taille n^2, opérations que l'on effectue en $\mathcal{O}(n^2)$.

On simule ensuite un nombre `nb_era` $\times$ `nb_sim_per_era` de parcours par une fourmi.

En considérant que les opérations arithmétiques et de choix aléatoire du module `numpy` sur des vecteurs lignes sont négligeables par rapport à nos opérations, une simulation de fourmi est en $\mathcal{O}(n)$.

En considérant cette fois que le produit d'une matrice `numpy` par un scalaire est négligeable par rapport à des opérations en $\mathcal{O}($`nb_sim_per_era` $\times n)$, la boucle du programme est finalement en $\mathcal{O}($`nb_era` $\times$ `nb_sim_per_era` $\times n)$.

Finalement, de manière générale le programme est en $\mathcal{O}(\max($`nb_era` $\times$ `nb_sim_per_era` $\times n, n^2))$. Dans nos cas c'est bien le premier argument du max qui est le plus grand, on a donc que la complexité asymptotique de la fonction est imposée par celle de la boucle.

L'expérience semble aussi valider une telle complexité à vue, augmenter chaque paramètre augmente linéairement le temps pris par une exécution de la fonction. Cette validation est clairement montrée fausse par la figure 12 qui met en évidence une relation non linéaire entre le temps d'exécution de la fonction et n (après différents tests, la complexité selon n se situe entre $n\log(n)$ et $n^{\frac{3}{2}}$)), on en déduit que nos suppositions sur les complexités des opérations `numpy` ont été trop ambitieuses et que la complexité réelle est plus compliquée à déterminer. Mais au vu des résultats de la fonction, il n'est pas vraiment pertinent de considérer cette complexité de manière isolée : il faut prendre en compte que chaque solution renvoyée par la fonction comprend une part très forte d'aléatoire et donc que les résultats varies, or on cherche une solution minimisant la distance des solutions. On va donc assez naturellement faire plusieurs appels (un grand nombre si possible) à la fonction, ce qui va largement augmenter le temps nécessaire pour trouver une bonne solution. Finalement, on voit bien qu'il n'est pas pertinent de regarder seule la complexité de la fonction, mais il faut aussi prendre en compte le nombre d'appel que l'on va y faire. Pour obtenir des bons résultats, ce nombre d'appel va probablement être au moins de l'ordre de la centaine.

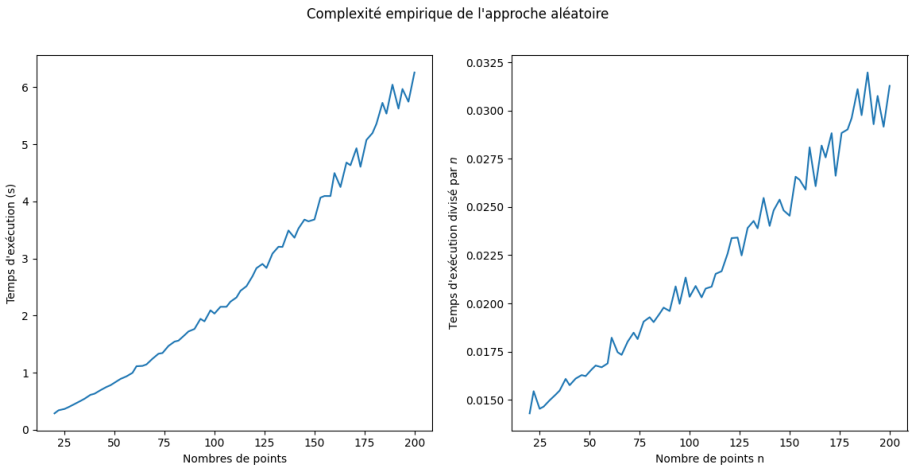


FIGURE 12 – Complexité empirique de l'approche aléatoire (comparaison par rapport à n)

5 Démêlage

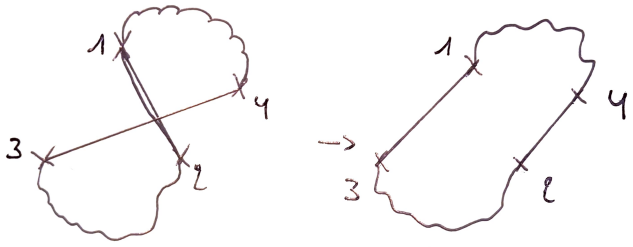


FIGURE 13 – Exemple d'un chemin comportant un croisement et de la modification à apporter pour le démêler

On remarque que beaucoup de chemins générés par des algorithmes peuvent contenir des croisements entre les segments, chose que l'on évite naturellement lorsque l'on dessine un chemin à la main puisque cela nous semble évidemment suboptimal. En effet, si deux segments (qui relient donc au total 4 points) se croisent, alors il serait plus efficace de changer les points reliés par les segments (on déplace les segments pour démêler le croisement). Cela est illustré par la transformation décrite dans la figure 13 ci-dessus : un segment relie les points 1 et 2 croise le segment qui relie les points 3 et 4, les points 1, 4 et 2, 3 étant reliés par un autre chemin. Grâce à la règle du parallélogramme, on sait que la somme des carrés des longueurs des diagonales du parallélogramme 1234 est plus grande que la somme des carrés des longueurs de ses côtés, donc la somme des longueurs des deux diagonales est plus grande que la somme des longueurs des deux côtés. Donc le chemin parcouru avec le croisement est supérieur à celui sans croisement, donc démêler ce croisement est intéressant pour réduire la distance totale d'un chemin.

Ainsi, il nous faut créer un programme qui prend en entrée un chemin, contenant ou non des segments à démêler, et qui renvoie un chemin sans croisement. L'implémentation est réalisé par la fonction `unknot_path` dans le fichier `util.py`.

Afin de trouver si deux segments se croisent, on boucle sur la totalité des segments (désignés par un couple de deux points consécutifs du chemin) et ce deux fois de manière imbriquée. Ainsi on dispose de tous les couples du produit cartésien de l'ensemble des segments avec lui-même.

Pour vérifier si deux segments se croisent, on utilise le formalisme des droites géométriques : pour un segment constitué de deux points, on trouve les paramètres $(a, b) \in \mathbb{R}^2$ de son équation paramétrique mise sous la forme $x+ay+b = 0$. On calcule ensuite les valeurs obtenues en utilisant les coordonnées des deux points de l'autre segment dans cette équation, le signe du résultat nous donne la position relative d'un point à la droite (au dessus ou en dessous). Si le produit des deux résultats est négatif, c'est que les deux points sont de part et d'autre de la droite. En répétant l'opération pour chaque segment, on sait que si tous les points sont de part et d'autre de la droite dirigé par le segment des autres points, alors il y a croisement des segments. On remarque à nouveau que c'est bien le cas dans la figure 13 : les points 3 et 4 sont de part et d'autre de la droite dirigé par le segment $(1, 2)$, et de même pour les points 1 et 2 qui sont de part et d'autre de la droite dirigé par le segment $(3, 4)$.

Si cette condition est vérifiée, on peut procéder à un démêlement. On effectue l'opération en inversant la partie du chemin complet qui se situe entre les points 2 et 3 [1], alors le démêlement est effectué et on peut recommencer l'opération de chercher un croisement. On répète l'opération jusqu'à ne plus détecter de croisement, alors le chemin est bien entièrement démêlé et on le renvoie.

1. NB : on pourrait faire de même avec les points 1 et 2

6 Comparaison des différents algorithmes

Les exemples 1 et 2 sont les exemples fournis, disponibles dans les fichiers `exemple_1.txt` et `exemple_2.txt`.

Les exemples 3 et 4 ont été générés aléatoirement et comptent respectivement 200 et 100 points. Ils sont disponibles dans les fichiers `exemple_3.txt` et `exemple_4.txt`

Algorithme	Exemple 1		Exemple 2		Exemple 3		Exemple 4	
	Distance	Temps	Distance	Temps	Distance	Temps	Distance	Temps
Espèces en danger	15,560	55,9µs	665,352	0,247s	1387,515	1,572s	954,148	0,259s
Enveloppe convexe	15,560	88,0µs	644,759	0,338s	1348,396	3,135s	909,063	0,371s
Aléatoire*	15,560	205ms	649.072	4.8s	1311.389	$\sim$ 14.9s	904.137	$\sim$ 5.0s
Force brute	15,560	320ms	-	-	-	-	-	-
Backtracking	15,560	250ms	-	-	-	-	-	-

Table 3 – Comparaison entre nos différents algorithmes

$(*)$: les temps sont donnés pour une exécution de la fonction sur les données, mais les solutions dont on garde la distance ont été trouvées en exécutant la fonction un grand nombre de fois pour trouver une bonne solution (de l'ordre de la centaine d'exécutions)

7 Conclusion

En conclusion, on peut dire que chaque algorithme a ses forces et ses faiblesses. L'algorithme par force brute est inutilisable pour des jeux de données trop grands (au delà de 10 points environ), car son temps d'exécution est trop important. Il est cependant le seul à renvoyer avec certitude une solution optimale.

Les algorithmes de l'enveloppe convexe et des espèces en dangers ont la meilleure complexité, semblant tous les deux être en $\mathcal{O}(n^3)$, donc dans un cas où le jardiner à un très grand nombre d'arbres à aller voir, il est plus intéressant d'utiliser l'un ou l'autre, l'enveloppe ayant des temps d'exécution plus stables que celui des espèces en danger.

Cependant, de manière générale, on remarque que les distances renvoyées par l'algorithme de l'enveloppe convexe sont en général plus petites que celles des espèces en danger. Cela étant dit, nous avons trouvé un contre-exemple à cette tendance, avec l'exemple « des losanges denses », dont le jeu de données est disponible dans le fichier `exemple_losange_dense.txt`. Pour ce jeu, l'algorithme des espèces trouve une distance finale de 25,521, tandis que celui de l'enveloppe trouve une distance de 25,813, contre une distance optimale réelle de 24,797, obtenue par force brute.

D'une manière très différente, l'algorithme basé sur l'aléatoire a une complexité relativement réduite selon ses paramètres. Mais comme il a été expliqué dans sa propre partie, ce n'est pas là que réside le réel coût temporel final de l'algorithme. En effet, pour obtenir une solution qui est satisfaisante, on va devoir relancer la fonction un grand nombre de fois : d'abord pour adapter les paramètres aux données précises utilisées, puis pour chercher une solution dont la distance semble proche d'un minimum des résultats de la fonction.

On remarque par contre que selon les données, cet algorithme est capable de donner des solutions proches voire meilleures que les solutions trouvées avec les autres algorithmes suboptimaux. Le coût temporel est donc à mettre en parallèle avec la valeur de la solution trouvée, et le choix d'utiliser ou non cet algorithme doit bien prendre ces facteurs en compte.

Ce travail s'est déroulé sans accroc dans notre groupe car, naturellement, des sous-groupes (stables par LCI et par passage au symétrique) se sont organiquement formés pour créer efficacement des algorithmes très différents en parallèle, tout en maintenant un lien entre tous les

membres du groupe : chacun a pu aider et donner son avis sur des décisions à prendre dans d'autres "pôles", et a pu prendre connaissance de tous les algorithmes développés, notamment grâce à la mise en place d'un dépôt `GitHub`, sur lequel chacun faisait des mises à jour régulières de ses programmes dans sa propre branche, une par heuristique.