DM ALGO1

Pingouins gourmands

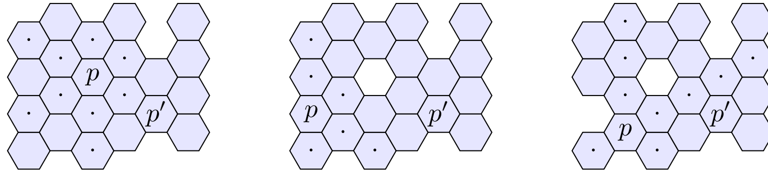
Pour le 28 novembre 2024

> Le travail en groupe est encouragé, mais vous devez rendre des copies rédigées individuellement. Pour les questions difficiles, n'hésitez pas à rédiger l'état de vos recherches.
>
> Dans une première partie, courte et indépendante du reste, on envisage une approche algorithmique naïve. Une meilleure approche est étudiée en deuxième partie. En troisième partie on construit une réduction, qui nous indique intuitivement que le problème des pingouins ne peut avoir de solution "simple". La troisième partie comporte des questions difficiles, mais il est possible de les admettre et d'avancer. C'est moins le cas dans la deuxième partie.

On considère dans ce sujet un problème issu du jeu de plateau *Pingouins* (*Hey, That's My Fish* en VO) où plusieurs pingouins jouent à tour de rôle pour essayer de manger un maximum de poissons en se déplaçant sur une banquise représentée comme un sous-ensemble de la grille hexagonale. À chaque tour, un pingouin peut se déplacer d'autant de cases qu'il le souhaite dans chacune des six directions, tant qu'il reste sur la banquise et ne croise pas d'autre pingouin. Une fois ce mouvement réalisé, la case de banquise anciennement occupée par le pingouin disparaît. Chaque case contient un certain nombre de poissons, mangés par le pingouin qui s'y rendra. La figure 1 détaille ces règles sur un exemple.

Le jeu commence sur une banquise de forme quasi-rectangulaire, mais ce terrain va évoluer au fil du temps, et ne restera même pas forcément connexe. En fin de partie, chaque pingouin va se retrouver isolé sur une composante connexe, mais malgré l'absence de compétition il ne pourra pas forcément manger tous les poissons de sa composante connexe, en fonction de la forme de celle-ci. Optimiser ce problème à un joueur est le problème qui nous intéresse ici. Par exemple, sur la banquise représentée en figure 2, le pingouin initialement placé en p ne pourra pas visiter toutes les cases.

Question 0.1: Décrire comment le pingouin peut visiter toutes les cases sauf deux sur l'exemple de la figure 2.



Les cases de banquise sont coloriées en bleu. Sur chacune des trois configurations, le pingouin p peut se déplacer en un coup sur n'importe laquelle des cases marquées d'un point, en glissant en ligne droite d'autant de cases que souhaité, mais sans jamais sortir de la banquise ni rencontrer l'autre pingouin p'. La configuration du milieu résulte de celle de gauche par l'un des mouvements de p : la case précédemment occupée par p a disparu. On passe de même de la configuration du milieu à celle de droite par encore un coup de p. À partir de cette dernière configuration, p pourra visiter la case en bas à gauche et y rester pour toujours, ou alors faire un autre mouvement et ne jamais visiter la case en question, qui sera déconnectée du reste de la banquise.

Figure 1 – Exemple de configurations successives avec deux pingouins

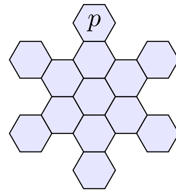


Figure 2 – Le pingouin doit visiter toutes les cases sauf deux

1 Graphe des configurations

Dans cette section et la suivante, on considère le problème d'optimisation suivant : **étant donnés** une banquise et une position de départ du pingouin sur cette banquise, **déterminer** le nombre maximal de cases que peut visiter le pingouin. Cela revient à optimiser le nombre de poissons mangés quand il y a exactement un poisson par case.

Question 1.1: Donner un algorithme permettant de calculer la longueur maximale d'un chemin à partir d'un sommet donné dans un graphe orienté acyclique. On pourra procéder par programmation dynamique.

Question 1.2: Notre problème peut être reformulé en un problème de plus long chemin, mais pas sur le graphe dont les sommets seraient les *cases* de la banquise initiale. Décrire un graphe sur les *configurations* qui permet une telle réduction, et évaluer la complexité de la méthode ainsi obtenue.

2 Exploration vers l'avant

Pour éviter le coût de la construction explicite du graphe des configurations, on cherche à développer un algorithme qui n'en construit, implicitement, que les sous-parties utiles. Notre point de départ est l'algorithme donné en Figure 3, où c_0 est la configuration initiale et $\text{next}(c)$ est l'ensemble des configurations qu'on peut obtenir en un coup à partir de c.

```
1:  todo := {(c_0, 1)}
2:  best := −∞
3:  while todo ≠ ∅ do
4:      retirer un élément (c, k) de todo
5:      best := max(best, k)
6:      for c' ∈ next(c) do
7:          ajouter (c', k + 1) à todo
8:      end for
9:  end while
10: return best
```

FIGURE 3 – Algorithme naïf pour une configuration de départ c_0

Question 2.1: Prouver la correction partielle de cet algorithme : à la fin, best est le nombre maximal de cases que le pingouin peut visiter, en comptant la case de départ.

Question 2.2: Prouver la terminaison de l'algorithme. Vous pouvez utiliser l'ordre bien fondé sur les multi-ensembles [1] en admettant sa bonne fondaison.

Notre algo naïf est encore moins bon que la construction explicite du graphe des configurations, mais on va l'améliorer. Un premier problème à régler est qu'une même configuration c peut être ajoutée plusieurs fois à todo. En particulier, on peut avoir plusieurs occurrences d'une configuration c dans todo à un moment donné – avec des valeurs de k forcément différentes si todo implémente bien un ensemble.

Question 2.3: Expliquer comment modifier l'algorithme, et sa preuve de correction, pour qu'à tout moment une configuration n'apparaisse qu'au plus une fois dans todo. (Ne pas s'inquiéter pour l'instant qu'une configuration sortie de todo ligne 4 puisse plus tard y être de nouveau ajoutée.)

Attention, les deux questions suivantes doivent être pensées ensemble, puisque la deuxième porte sur la modification effectuée dans la première.

Question 2.4: Modifier l'algorithme pour obtenir un algorithme optimal sur des configurations simples comme celles de la figure 4, au sens où les configurations considérées (ligne 4) doivent être exactement celles d'un unique chemin de longueur maximale. Pour cela :

1. https://fr.wikipedia.org/wiki/Multiensemble

FIGURE 4 – Configurations simples

- Utiliser une file de priorité pour `todo`, avec un choix judicieux de priorité.
- Raffiner la notion de configuration pour rendre compte des déconnexions.
- Éviter d'explorer des configurations sous-optimales.

Les modifications devront être précisément décrites, et expliquées, mais il n'est pas demandé de prouver l'algorithme modifié.

Question 2.5: Expliquer pourquoi, grâce à votre choix de fonction de priorité, votre algorithme ne va jamais re-introduire dans la file de priorité une configuration qui en avait été extraite (ligne 4) par le passé, sans que cela affecte la correction. Une preuve de correction complète n'est pas exigée.

Question 2.6: Décrire précisément des structures de données adéquates pour l'implémentation de cet algorithme, en prenant en compte les remarques suivantes :

- Les optimisations précédentes, évitant de traiter plusieurs fois une même configuration, ne sont pleinement effectives que si la notion de configuration ne contient pas trop d'éléments rendant deux configurations égales alors qu'elles sont équivalentes pour le problème considéré.
- Concernant la file de priorité `todo`, on constate expérimentalement un nombre nettement plus important d'opérations de modification de priorité que d'insertions.
- Une file de priorité n'est pas faite pour rechercher des élements dedans : il vous faudra mettre en place une structure auxiliaire pour associer à une configuration un noeud dans la file de priorité, s'il existe.
- Le nombre de configurations possibles est très grand même pour des configurations de départ de taille modeste. Les choix d'implémentation pourront reposer sur l'hypothèse que la banquise de départ ne fait qu'une centaine de cases au plus.

3 Les pingouins font de la logique

Les optimisations apportées à l'algorithme précédent le rendent efficace sur de nombreux exemples, mais sa complexité dans le pire cas reste exponentielle. Il semble que cela soit inévitable : nous montrons dans cette partie que le problème de décision associé à notre problème d'optimisation, dans le cas où les cases peuvent contenir zéro ou un poisson, ne peut probablement pas être résolu en temps polynomial. Plus précisément, on construit une réduction du problème NP-complet 3-SAT vers notre problème.

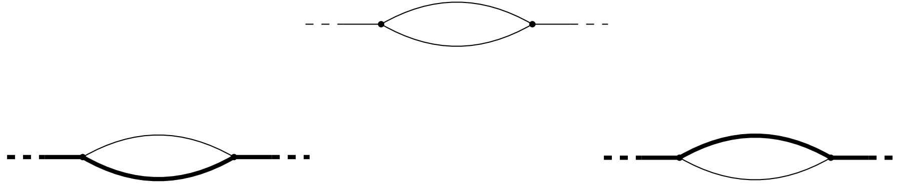


FIGURE 5 – Chemins induits par un circuit Hamiltonien dans un sous-graphe

Le problème de décision qui nous intéresse est, **étant donnés** une banquise où chaque case contient 0 ou 1 poisson, une position de départ sur cette banquise, et un objectif $k \in \mathbb{N}$, de **déterminer** si le pingouin peut manger k poissons.

Le problème 3-SAT repose sur des formules simples de logique propositionnelle. Un *littéral* est soit une variable propositionnelle P soit la négation d'une telle variable, $\neg P$. Une 3-clause est la disjonction de trois littéraux. Par exemple, $P \vee \neg Q \vee R$ est une 3-clause, de même que $P \vee \neg P \vee Q$ ou $P \vee P \vee Q$. Le problème 3-SAT est, **étant donné** un ensemble de 3-clauses, de **déterminer** si elles sont satisfiables, c'est-à-dire s'il existe une interprétation des variables rendant toutes les clauses vraies.

Pour réaliser notre réduction de 3-SAT vers les pingouins, nous allons utiliser des graphes non-orientés particuliers comme objets intermédiaires. Ces graphes seront construits à partir de sous-graphes bien choisis, de sorte que l'existence d'un circuit Hamiltonien [2] dans le graphe ne soit possible que sous certaines conditions. Par exemple, si un graphe G contient un sous-graphe tel que dessiné en haut de la figure 5, et si G contient un circuit Hamiltonien, alors ce circuit passe forcément dans le sous-graphe d'une des deux façons représentées en bas dans la même figure. Attention ici à bien interpréter les arêtes sortant du sous-graphe pour le relier au graphe englobant : une arête sortante dans le dessin correspond à une arête dans le graphe global, donc ici le sommet de gauche est de degré exactement 3 dans le graphe global.

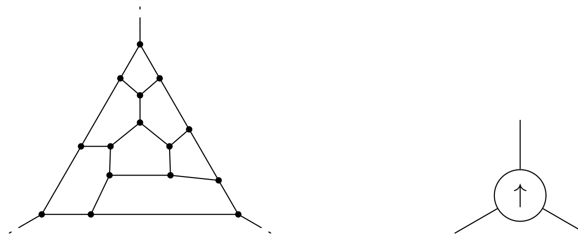


FIGURE 6 – Sous-graphe forçant un passage par l'arête supérieure

2. Un circuit Hamiltonien est un cycle élémentaire qui passe exactement une fois par chaque sommet.

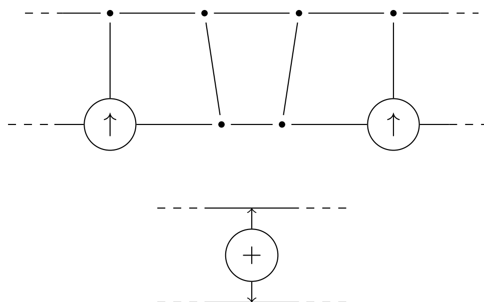


Figure 7 – Ou exclusif (xor)

Question 3.1: On considère le sous-graphe de gauche en figure 6 et l'on souhaite montrer qu'un circuit Hamiltonien dans un graphe englobant ce sous-graphe passe nécessairement par l'arête sortant en haut. Pour cela, donner tous les chemins dans le sous-graphe qui peuvent être induits par un circuit Hamiltonien dans le graphe englobant.

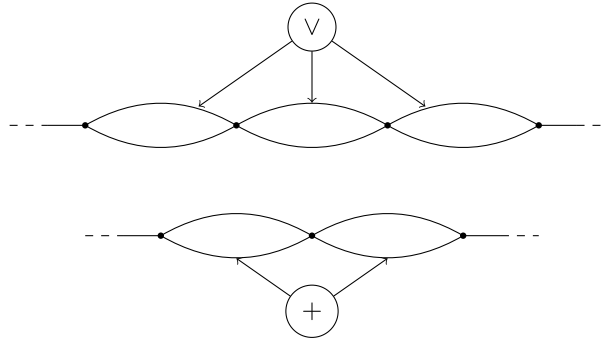
Le résultat précédent justifie de noter, dans la suite, le sous-graphe de la question précédente comme un noeud spécial décoré d'une flèche, tel que représenté à droite en figure 6. On peut considérer qu'on travaille désormais avec des graphes contenant ces noeuds spéciaux, pour lesquels les circuits Hamiltonien doivent nécessairement passer par l'arête fléchée.

Question 3.2: On considère maintenant le sous-graphe en haut de la figure 7, et sa notation raccourcie en bas de la même figure. Montrer que le sous-graphe impose une contrainte "xor" (ou exclusif) sur l'utilisation des deux successions d'arêtes horizontales, au sens où les seuls chemins induits par des circuits Hamiltoniens dans le sous-graphe sont ceux où exactement un des deux groupes d'arêtes est traversé, de sorte que la notation raccourcie peut donc se lire comme un xor entre deux arêtes.

On remarquera que les deux constructions précédentes reposent sur des graphes planaires dans lesquels chaque sommet est de degré au plus trois. On appelera de tels graphes des graphes P3. Par la suite, toutes les constructions devront satisfaire cette contrainte.

Question 3.3: Construire un sous-graphe qui permette de réaliser une contrainte "ou" entre deux arêtes : un circuit Hamiltonien doit nécessairement emprunter l'une des deux arêtes, mais il peut aussi emprunter les deux. Cette contrainte sera notée comme pour le xor mais avec un noeud $\vee$ plutôt que $+$. Votre construction devra reposer sur un sous-graphe comportant moins de seize sommets (en comptant un noeud $\uparrow$ comme un seul sommet, et en ne comptant pas les contraintes xor).

On admettra qu'on peut réaliser de la même façon une contrainte de disjonction ternaire, qu'on notera comme les contraintes "ou" mais avec trois flèches sortantes pointant les trois arêtes concernées. Cet opérateur est utilisé en figure 8 dans le codage d'une 3-clause.



Le circuit Hamiltonien doit passer dans chacun des trois blocs de la clause, en haut ou (exclusif) en bas, et la contrainte de disjonction impose qu'il passe en haut sur au moins un des trois blocs. L'idée est que le passage par le haut va signifier que le littéral correspondant est vrai, et l'on veut s'assurer que ce sera le cas pour au moins un des trois blocs pour que la clause soit satisfaite.
Pour la variable propositionnelle, le passage en haut dans l'un des blocs (resp. l'autre bloc) va signifier que la variable est vraie (resp. fausse). La contrainte xor assure qu'on n'a pas les deux à la fois.

FIGURE 8 – Codage d'une 3-clause (en haut) et d'une variable (en bas)

Question 3.4: Étant donné un ensemble de 3-clauses E, construire un graphe G_E qui a un circuit Hamiltonien ssi E est satisfaisable, en s'appuyant sur les codages de la figure 8 et des contraintes xor. Le graphe construit devra être P3 à une exception près : on s'autorise ici des croisements de contraintes xor, tel qu'illustré en figure 9. (Pour que la construction G_E soit intéressante, on veut qu'elle soit calculable par un algorithme en temps polynomial. Le sous-graphe en figure 9 est un exemple tout à fait arbitraire, qu'il ne faut pas chercher à interpréter comme une idée pour la construction de G_E.)

Question 3.5: En exploitant le fait que les arêtes verticales dans le sous-graphe codant le xor (figure 7, à gauche) sont forcément empruntées dans tout chemin Hamiltonien, expliquer comment on peut simuler des croisements de contraintes xor dans un graphe P3. Par exemple, on veut pouvoir simuler le sous-graphe de la figure 9 par un sous-graphe P3 qui induit les mêmes contraintes. (Noter que ce sous-graphe peut être inscrit dans un graphe quelconque, ce qui ne permet pas de résoudre le problème de planarité en faisant passer l'une des deux contraintes xor par l'extérieur du pentagone.)

Question 3.6: On admet qu'un graphe P3 peut se dessiner dans une grille

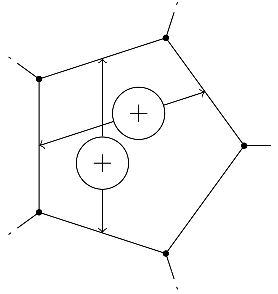


Figure 9 – Croisements de contraintes xor

hexagonale en associant une case à chaque sommet et en traçant les arêtes par des chemins dans la grille qui ne se croisent qu'aux sommets. En déduire une réduction de 3-SAT vers le problème de décision des pingouins avec 0 ou 1 poissons par cases.