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.

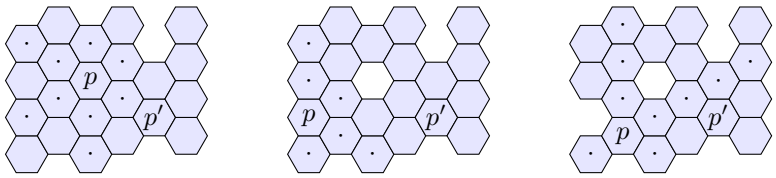
Dans une première partie, courte et indépendante du reste, on envisage une approche algorithmique naïve. Une meilleure approche est étudiée en deuxième partie. En troisième partie on construit une réduction, qui nous indique intuitivement que le problème des pingouins ne peut avoir de solution "simple". 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.

Corrigé: En fait, on peut tout manger sauf un (mais pas mieux) par exemple comme ceci :



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

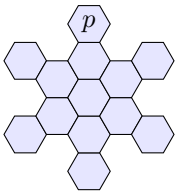
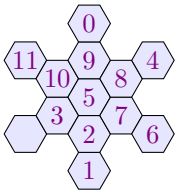


FIGURE 2 – Le pingouin doit visiter toutes les cases sauf deux



□

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.

Corrigé: Soit $G = (V, E)$ notre DAG. On commence par faire un tri topologique de V : en partant des sommets sans arcs entrants, il suffit de faire un parcours du graphe, pour obtenir une énumération $v_1, v_2, \ldots, v_n$ des sommets telle que pour tout arc $(v_i, v_j) \in E$ on aura $i < j$. On peut alors procéder par programmation dynamique : si l'on note $\ell(v)$ la longueur maximale des chemins partant de v, on a :

$$\ell(v) = 1 + \max_{(v,v') \in E} \ell(v')$$

On peut ainsi calculer $\ell(v_i)$ pour chaque v_i, en procédant itérativement pour i allant de n à 1. Il ne reste plus qu'à renvoyer la valeur obtenue pour le sommet souhaité — d'ailleurs, on peut arrêter l'itération dès qu'on atteint ce sommet, et on pourrait ne faire le tri topologique que parmi les sommets accessibles depuis le sommet qui nous intéresse, mais ces raffinements ne sont pas utiles ici. □

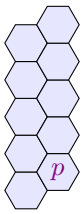
Question 1.2: Notre problème peut être reformulé en un problème de plus long chemin, mais pas sur le graphe dont les sommets seraient les *cases* de la banquise initiale. Décrire un graphe sur les *configurations* qui permet une telle réduction, et évaluer la complexité de la méthode ainsi obtenue.

Corrigé: Étant donnée une configuration de départ, on considère un graphe dont les noeuds sont toutes les configurations *incluses* dans la configuration de départ, au sens où elles sont obtenues en supprimant des cases, et en choisissant une nouvelle position du pingouin parmi les cases de la configuration de départ.

Remarque : Il est possible d'adopter deux conventions. Soit le pingouin est sur une case existante, soit le pingouin est positionné sur une case qui a déja disparu. Cela ne change essentiellement rien à ce stade : cela décale seulement de un le nombre de poissons mangés par le pingouin.

En appliquant l'algorithme précédent à ce graphe, on obtient une solution en temps linéaire en la taille du graphe — il n'est pas nécessaire de détailler cette complexité précisément en fonction des nombres de sommets et d'arêtes. Malheureusement, le graphe est de taille exponentielle en la taille de la configuration de départ, donc on a globalement un algorithme de complexité exponentielle.

Pour aller plus loin : Il n'est pas demandé de raffiner ce raisonnement, mais il est légitime de se demander si on ne peut pas éviter cette exponentielle en étant un peu plus fins dans la construction du graphe de configurations. En effet, la construction simple donnée plus haut va comporter beaucoup de configurations inaccessibles. On peut aussi remarquer qu'il est inutile de garder dans les configurations les cases devenues inaccessibles, ce qui permettrait de fusionner différentes configurations. Cependant, en partant d'une configuration de la forme suivante, on va retrouver une taille exponentielle :



En effet, on peut atteindre toutes les configurations où la colonne de gauche est intouchée et on prend un sous-ensemble quelconque de la colonne de droite. De plus, ces configurations sont connexes. Ainsi le graphe des configurations, même raffiné selon les deux idées évoquées, comporte au moins $2^{n/2}$ sommets, où n est le nombre de cases dans la configuration de départ. Cela ne veut pas dire qu'on ne peut pas raffiner avec de meilleures idées que celles-ci, mais cela semble difficile. □

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.

Corrigé: On note comme précédemment $\ell(c)$ la longueur maximale d'un chemin à partir de c. On pose de plus $\ell(\mathsf{todo}) = \max_{(c,k)\in\mathsf{todo}}(\ell(c)+k)$. On va montrer que la propriété suivante est un invariant pour la boucle de notre algorithme :

$$\max(\mathsf{best}, \ell(\mathsf{todo})) = \ell(c_0) + 1$$

Autrement dit, le membre gauche de l'équation est un invariant.

Supposons la propriété vraie sur best et todo, et montrons que leurs valeurs après un tour de boucle, notées best' et todo', satisfont la même propriété. Soit (c, k) la valeur traitée lors du tour de boucle. On a [1] :

$$\begin{aligned}
\ell(\mathsf{todo}) &= \max\big(\ell(c)+k,\ \ell(\mathsf{todo}\setminus\{(c,k)\})\big)\\
&= \max\big(\max(k, \max_{c'\in \mathrm{next}(c)} k+1+\ell(c')),\ \ell(\mathsf{todo}\setminus\{(c,k)\})\big)\\
&= \max\big(k,\ \ell(\mathsf{todo}')\big)
\end{aligned}$$

Ainsi $\max(\mathsf{best}, \ell(\mathsf{todo})) = \max(\max(\mathsf{best}, k), \ell(\mathsf{todo}')) = \max(\mathsf{best}', \ell(\mathsf{todo}'))$: on a bien invariance.

La correction partielle de l'algorithme découle de cet invariant : l'invariant est évidemment vérifié quand on atteint la boucle après exécution des lignes 1 et 2 ; en sortie de boucle, l'invariant et $\mathsf{todo} = \emptyset$ nous permettent de conclure que $\mathsf{best} = \max(\mathsf{best}, \ell(\mathsf{todo})) = \ell(c_0) + 1$. □

Question 2.2: Prouver la terminaison de l'algorithme. Vous pouvez utiliser l'ordre bien fondé sur les multi-ensembles [2] en admettant sa bonne fondaison.

Corrigé: Pour chaque configuration c on note $|c|$ le nombre de cases dans la configuration. Pour un ensemble fini $T \subseteq C \times \mathbb{N}$, où C est l'ensemble des configurations, on note $m(T) = \{\{|c| \mid (c,k) \in T_1, k \in \mathbb{N}\}\}$ le multi-ensemble des tailles des configurations apparaissant dans T : on garde ainsi la trace du nombre d'ocurrences d'une configuration c dans T, chaque occurrence associant c à une valeur de k. On définit enfin un ordre partiel sur les sous-ensembles finis de $C \times \mathbb{N}$ en posant $T_1 < T_2$ lorsque $m(T_1) \prec m(T_2)$ où $\prec$ est l'ordre multi-ensemble sur les entiers munis de l'ordre usuel $<$. Comme $(\mathbb{N}, <)$ est bien fondé, l'ordre $\prec$ est bien fondé. Autrement dit, $(C \times \mathbb{N}, <)$ est bien fondé.

L'ensemble todo nous donne alors un variant pour la boucle while de notre algorithme, selon l'ordre que nous avons défini sur $C \times \mathbb{N}$. En effet, à chaque tour de boucle on supprime un élément c de todo et on ajoute des éléments c' tels que $|c'| < |c|$. Par définition de l'ordre multi-ensemble, on a donc $\mathsf{todo}' < \mathsf{todo}$ où todo' est la valeur de todo en fin de boucle. □

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.

1. Le point délicat est le passage de la première à la deuxième ligne. Il repose sur l'identité $\ell(c) = \max(0, \max_{c'\in\mathrm{next}(c)} 1+\ell(c'))$. On a aussi $\ell(c) = \max(0, 1+\max_{c'\in\mathrm{next}(c)} \ell(c'))$ si l'on considère que le second max est pris dans $\overline{\mathbb{Z}}$, ce qui nous donne, dans le cas où $\mathrm{next}(c) = \emptyset$, $\ell(c) = \max(0, 1-\infty) = 0$.
2. `https://fr.wikipedia.org/wiki/Multiensemble`

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.)

Corrigé: On propose l'algorithme suivant :

```
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:          if ∃k'.  (c', k')  ∈ todo then
8:              remplacer (c', k') par (c', max(k', k + 1)) dans todo
9:          else
10:             ajouter (c', k + 1) à todo
11:         end if
12:     end for
13: end while
14: return best
```

Par rapport à l'algo naïf, les lignes 7 à 11 ont remplacé l'ajout de la ligne 7, fig. 3, qui est désormais fait en ligne 10. L'idée est qu'il est inutile d'avoir (c, k_1) et (c, k_2) dans todo : il suffit de garder l'item le plus prometteur, correspondant au maximum de k_1 et k_2, i.e. au plus long chemin qui a mené à c.

L'invariant n'a pas à être modifié. Pour justifier son invariance il suffit d'observer que, pour toutes valeurs initiales de best et todo, la valeur de $\ell(\mathsf{todo}')$ est la même après exécution du corps de la boucle de l'algo naïf ou de l'algo modifié. En effet, le todo' de l'algo modifié est obtenu à partir de celui de l'algo naïf en supprimant des (c', k') tel que (c', k'') est présent avec $k'' \geq k'$. □

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 :

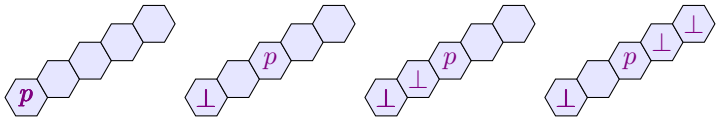
— Utiliser une file de priorité pour todo, avec un choix judicieux de priorité.
— Raffiner la notion de configuration pour rendre compte des déconnexions.
— Éviter d'explorer des configurations sous-optimales.

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

Corrigé: On suppose que les configurations sont connexes, non pas au sens des règles du jeu, mais au sens classique des graphes : le graphe formé des cases encore disponibles, avec des arêtes entre deux cases immédiatemment voisines,

FIGURE 4 – Configurations simples

doit être connexe. On considère ici que, dans une configuration, la position occupée par le pingouin n'est pas une case du graphe : sans cela, cette case pourrait être utilisée pour connecter deux composantes connexes en une seule, quand en il est déja clair que le pingouin devra choisir entre l'un et l'autre. On illustre cela sur le premier exemple de la figure 4, rappelé dans la première configuration ci-dessous :



On veut éviter que le déplacement du pingouin de deux cases vers le nord-est ne nous mène à la seconde configuration ci-dessus (où l'on représente les cases nouvellement effacées par un symbole $\perp$). Au lieu de cela, ce déplacement correspond à deux configurations possibles (les troisième et quatrièmes configurations ci-dessus) correspondant au choix de la composante connexe que l'on pourra explorer ensuite. Cela demande un peu de travail supplémentaire pour la fonction next, mais ce n'est pas une préoccupation pour cette question.

Pour une configuration c, on note $|c|$ le nombre de cases présentes dans la configuration – qui nous donne une borne supérieure sur le nombre de cases que le pingouin peut manger, moins grossière que si nos configurations n'étaient pas connexes, mais néanmoins approximative. Pour les quatre configurations montrées ci-dessus, les valeurs de $|c|$ sont 4, 3, 2 et 1 respectivement.

Nous allons modifier l'algo précédent comme suit :

- On remplace l'ensemble `todo` par une file de priorité, qui va nous permettre de traiter les items les plus prometteurs en premier. On considère que l'opération d'insertion renvoie un pointeur vers le noeud correspondant à l'élément nouvellement inséré, qui permettra un accès rapide pour une modification ultérieure de la priorité.
- La priorité associée à un élément c dans `todo` va être de la forme $(k, |c|)$ où k est la longueur du plus long chemin menant à c qu'on a trouvé jusque là. On utilise ici la lettre k volontairement : c'est la même quantité que dans les paires (c, k) qu'on mettait dans l'ensemble `todo` dans les algorithmes précédents. On favorise les explorations sans pertes en déclarant $(k, p) > (k', p')$ quand $k + p > k' + p'$, et à potentiel égal on favorise le parcours en profondeur en déclarant $(k, p) > (k', p)$ quand $k > k'$.
- Quand on extrait une configuration de la file de priorité, on l'ignore

si celle-ci ne peut plus mener à une solution plus intéressante que le maximum courant : c'est la ligne 8. Pour cela on utilise un dictionnaire qui associe à chaque configuration un noeud de la file de priorité contenant cette configuration s'il en existe un, ou une valeur spéciale $\perp$ indiquant que la configuration a été retirée de la file. Si un noeud n'est pas présent dans le dictionnaire, cela signifie qu'il n'est jamais rentré dans la file.

— La ligne 13 correspond à la ligne 8 de l'algorithme précédent : on met à jour la priorité pour ne garder trace que du chemin le plus intéressant – la différence ne peut provenir que de la composante k.

L'algorithme optimisé résultant est détaillé ci-dessous :

```
1:  todo := empty_queue()
2:  node := empty_map()
3:  best := −∞
4:  node[c_0] := insérer c_0 dans todo avec priorité (1, |c_0|)
5:  while todo ≠ ∅ do
6:      extraire de todo un élément c de priorité maximale (k, p)
7:      node[c] := ⊥
8:      sauter à la prochaine itération si k + p ≤ best
9:      best := max(best, k)
10:     for c' ∈ next(c) do
11:         if node[c'] ≠ ⊥ then                          ▷ c' est dans la queue
12:             if node[c'] a une priorité inférieure à (k + 1, |c'|) then
13:                 augmenter la priorité de node[c'] à (k + 1, |c'|)
14:             end if
15:         else
16:             if node[c'] = ⊥ then                      ▷ c' est sorti de la queue
17:                 rien à faire
18:             else                                      ▷ c' n'a encore jamais été rencontré
19:                 node[c'] := insérer c' dans todo avec priorité (k + 1, |c'|)
20:             end if
21:         end if
22:     end for
23: end while
24: return best
```

Il suffit de dérouler cet algorithme pour voir qu'il se comporte de façon optimale sur les deux exemples de la question :

— Pour le premier exemple, la configuration initiale ne mène qu'à deux configurations c' telles que $|c'| = 3$: les autres vont mises en attente dans la file et finalement ignorées. Soient c'_1 et c'_2 ces deux configurations. L'algorithme pourrait choisir l'une ou l'autre des deux configurations lors du second tour de boucle. Supposons que c'_1 est choisie. On engendre alors deux nouvelles configurations c''_1 et c''_2 satisfaisante toutes deux $|c''| = 2$, et une troisième configuration qui va être mise en attente puis ignorée. Pour le troisième tour de boucle, on aura dans la file de priorité c'_2 avec pour priorité $(2, 3)$ et les configurations c''_i avec pour priorités $(3, 2)$.

Notre ordre sur les priorités est tel qu'on choisira l'un des c_i'' : à potentiel égal, on favorise un parcours en profondeur du graphe des configurations. Ainsi de suite, l'algorithme explore un chemin sans perte de potentiel à partir de c_0. Quand il atteint ainsi une configuration sans successeurs on a $\mathsf{best} = 5$. Pour les itérations suivantes de la boucle on traitera des configurations c avec $k + p \leq 5$, qu'on ignorera une par une (ligne 8).

— Pour le second exemple, cela se passe de façon similaire. La différence est que cet exemple ne serait pas traité de façon optimale par un algorithme qui favoriserait les mouvements du pingouin à distance 1.

□

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.

Corrigé: Supposons que c a été traité et est donc sorti de la file. Il est clair, étant donné notre algorithme (lignes 7 et 16) qu'on n'insèrera plus jamais c dans la file de priorité. Il reste à justifier que cela n'affecte pas la correction : on montre ci-dessous que les nouvelles occurrences de c qui seront ignorées ne peuvent amener de meilleure solution que celles qui seront trouvées via le premier traitement de c.

Soit $(k, |c|)$ sa priorité au moment où il a été sorti. Immédiatemment après la sortie de c, la file contient des configurations c' avec des autres priorités $(k', |c'|) \leq (k, |c|)$. Supposons qu'une des configurations c', de priorité $(k', |c'|) \leq (k, |c|)$, permet d'atteindre c en p coups. Cela nous ferait retomber sur c (qui jouerait le rôle de c' dans l'algo à la ligne 10) avec pour priorité candidate $(k' + p, |c|)$. Comme c est atteint en p coups depuis c', on a $|c| + p \leq |c'|$. On a donc $k' + p + |c| \leq k' + |c'| \leq k + |c|$, d'où $k' + p \leq k$. Autrement dit on est en train de considérer un nouveau chemin vers c mais avec moins de k cases consommées pour y arriver : on ne perd rien à ignorer ce chemin étant donné qu'on a déja considéré déja celui avec k cases consommées pour arriver à c. □

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.

Corrigé: Pour les configurations, on a déja dit qu'il fallait les restreindre à des composantes connexes. Cela demande de faire un calcul de composante connexe dans l'implémentation de la fonction next. On va nécessairement se retrouver avec une implémentation en temps linéaire au pire cas, mais on peut essayer d'être en temps constant assez souvent. Pour cela il est judicieux d'implémenter next comme suit :

1. On suppose que la configuration c de départ est connexe.
2. On génère d'abord la configuration c' avec un next naïf ne se préoccupant pas de la connexité, mais supprimant juste la case correspondant à la nouvelle position du pingouin.
3. On vérifie ensuite si les six cases entourant cette nouvelle position formes plus qu'une composante connexe. Si c'est le cas, alors le graphe dans son ensemble ne peut pas avoir été déconnecté par la suppression de la nouvelle case du pingouin. Sinon, il faut faire un vrai calcul de composantes connexes, et éventuellement découper la configuration c' en autant de configurations c'_i qu'il y a de composantes connexes.

Pour avoir des opérations efficaces sur les ensembles de cases (i.e. vérifier la présence d'une case, supprimer une case, tester l'égalité de deux ensembles) une bonne idée est de représenter les ensembles par des bitmaps. Un ensemble de moins de 128 cases serait par exemple représenté sur deux entiers 64 bits, et les opérations nécessaires s'implémentent très efficacement, soit en une opéraion bit-à-bit (présence, suppression) soit en une comparaison de paires d'entiers (égalité ensembliste).

La question indique qu'on fait plus de changements de priorité que d'insertions, il semble donc judicieux d'implémenter la file de priorité par un tas de Fibonacci. Pour `node`, une table de hachage est simple et efficace. □

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.

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.

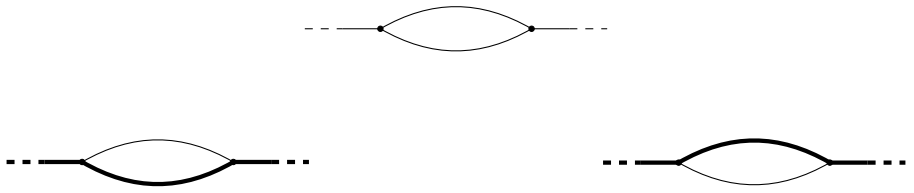


FIGURE 5 – Chemins induits par un circuit Hamiltonien dans un sous-graphe

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 [3] 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.

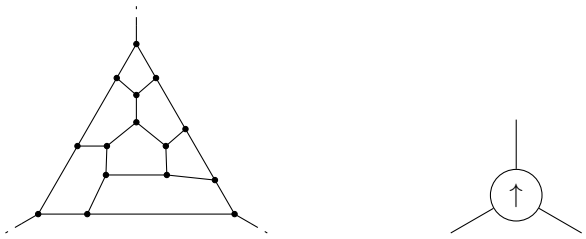


FIGURE 6 – Sous-graphe forçant un passage par l'arête supérieure

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

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

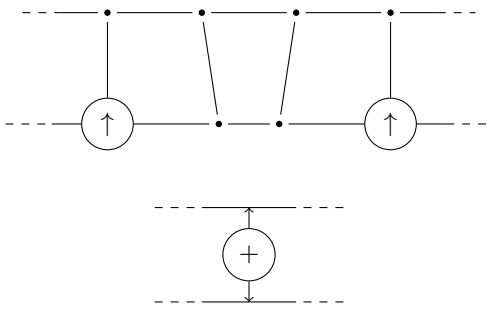


FIGURE 7 – Ou exclusif (xor)

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.

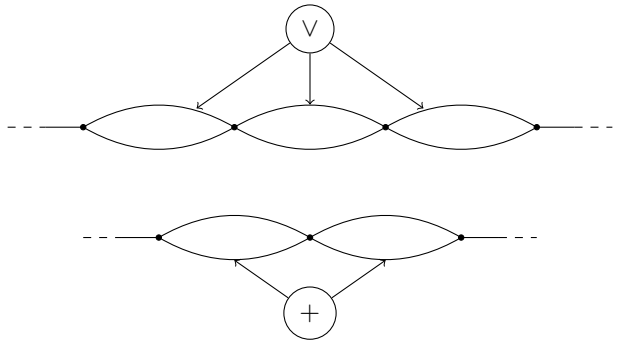
Question 3.2: On considère maintenant le sous-graphe en haut de la figure 7, et sa notation raccourcie en bas de la même figure. Montrer que le sous-graphe impose une contrainte "xor" (ou exclusif) sur l'utilisation des deux successions d'arêtes horizontales, au sens où les seuls chemins induits par 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.)

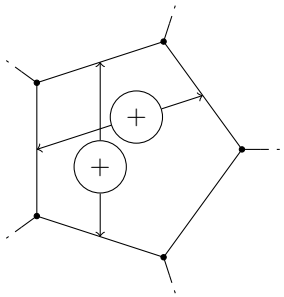


FIGURE 9 – Croisements de contraintes xor

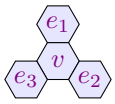
Question 3.5: En exploitant le fait que les arêtes verticales dans le sous-graphe codant le xor (figure 7, à gauche) sont forcément empruntées dans tout chemin Hamiltonien, expliquer comment on peut simuler des croisements de contraintes xor dans un graphe P3. Par exemple, on veut pouvoir simuler le 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 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.

Corrigé: Considérons une instance de 3-SAT, i.e. un ensemble E de 3-clauses. On calcule (en temps polynomial) un graphe P3 G'_E selon la méthode de la question 3.4 modifiée comme expliqué en 3.5. Il existe un circuit Hamiltonien dans G'_E ssi E est satisfaisable.

Soit $e = \{v, v'\}$ une arête quelconque de G'_E. On construit un nouveau graphe G''_E en ajoutant à G'_E trois nouveaux sommets v_1 à v_3 et en remplaçant l'arête e par quatre arêtes $\{v, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}$ et $\{v_3, v'\}$. Il existe un circuit Hamiltonien dans G'_E ssi il existe un chemin Hamiltonien dans G''_E partant de v_2.

Comme G''_E est encore P3, on peut le tracer sur une grille hexagonale : comme admis dans la question, on obtient une configuration dans laquelle tout case correspond soit à un sommet soit à une arête de G''_E, un sommet correspondant à une seule case mais une arête correspondant généralement à plusieurs cases. Quitte à modifier un peu la grille obtenue, on peut de plus exiger que les arêtes n'arrivent en une case sommet v que par l'une des trois directions suivantes (c'est déja forcé, modulo rotation, pour un sommet de degré trois, mais pour un sommet de degré deux notre contrainte interdit que les deux arêtes soient "face à face" de part et d'autre de v) :



On ne peut aller d'une case-sommet à une autre (sans passer par des cases-sommets intermédiaires) qu'en empruntant les cases correspondant à une même arête entre les deux sommets considérés. Si l'on considère des chemins de pingouins (avec glissements) plutôt que des chemins usuels dans le graphe de voisinage, cela reste vrai grâce à notre contrainte supplémentaire : la seule différence est que le pingouin pourrait ne pas visiter toutes les cases d'une arête en allant d'un sommet à l'autre via cette arête, mais cela n'a aucune importance car une fois les deux sommets visités les cases restantes de l'arête seront déconnectées

du reste du graphe.

Pour obtenir une instance I_E du problème de pingouins, il reste à placer le pingouin sur la case correspondant au sommet v_2, et à placer exactement un poisson par case correspondant à un sommet, aucun poisson sur les cases correspondant à des arêtes. Par les observations précédentes, le pingouin peut manger tous les poissons sur I_E ssi il existe un chemin Hamiltonien sur G''_E. Autrement dit on a bien construit une instance du problème de pingouins qui est positive ssi E est une instance positive de 3-SAT. □