Cet article est rédigé par des élèves. Il peut comporter des oublis ou des imperfections,
autant que possible signalés par nos relecteurs dans les notes d'édition.

Le loup, la chèvre et le chou

Année 2022-2023

Azhar Gana (classe de 3e), Charlotte Macaire, Clément Carayon,
Medhi Touhami, Adrien Tapissier (classe de 2nde)

Établissement : Lycée Français des Pays-Bas, La Haye

Enseignant·es : Stéphane Béringue, Line Boissonnet, Mathieu Buchwald, Florence Decool

Chercheur : Jordan Frecon, Télécom Saint-Étienne, Université de Lyon Laboratoire HubertCurien

Résumé

Cet article parle de la résolution du fameux problème du loup, de la chèvre et du chou. On expliquera dans cet article comment résoudre mathématiquement le premier problème rédigé de l'histoire. Ce sujet nous a été donné par Jordan Frecon, un chercheur en mathématiques, expert en intelligence artificielle. Tout d'abord, nous avons cherché à résoudre le problème avec des matrices, mais certaines matrices permettaient des transports illégaux (impossible physiquement). Nous avons alors cherché une solution algorithmique. Pour optimiser le programme Python, on l'a codé en C++. Ainsi, le problème basique résolu, nous avons cherché à approfondir notre sujet par une extension avec des variantes. On a aussi continué cela grâce aux graphes ainsi que par la création de nouveaux algorithmes qui ont été combinés avec les graphes. On a donc pu déterminer qu'il y avait plusieurs types de graphes : linéaire, circulaire et quelconque. Puis nous avons aussi trouvé un calcul pour connaître le nombre de places qu'il faudrait dans un bateau pour chaque type de variantes.

Table des matières

1 Introduction **3**
- 1.1 Présentation du problème . . . 3
- 1.2 Définitions . . . 3
- 1.3 Solution classique . . . 3

2 Présentation des variantes possibles avec des graphes **4**
- 2.1 Types de variantes . . . 4
- 2.2 Nombre de places sur le bateau requis pour chaque variantes . . . 4
  - 2.2.1 Linéaires et circulaires . . . 4
  - 2.2.2 Quelconques . . . 5

3 Résolution algorithmique des variantes linéaires grâce à des graphes **5**
- 3.1 Graphe pour 3 animaux . . . 5
- 3.2 Résolution algorithmique avec des graphes . . . 6
- 3.3 Optimisation avec C++ [1] . . . 6
  - 3.3.1 Aperçu . . . 6
  - 3.3.2 État . . . 6
  - 3.3.3 État légal . . . 7
  - 3.3.4 Transports . . . 7
  - 3.3.5 Algorithme de base . . . 7
  - 3.3.6 Liste d’incidence . . . 7

4 Tentative d’une solution analytique des variantes linéaires avec des matrices **8**
- 4.1 Modélisation du problème original . . . 8
  - 4.1.1 Vecteur d’état et matrices de transport . . . 8
  - 4.1.2 Suite de solution . . . 9
- 4.2 Résolution algorithmique du problème . . . 10
- 4.3 Tentative d’étente à des variantes . . . 10

5 Résolution analytique des variantes linéaires grâce à des ensembles **11**
- 5.1 Définitions . . . 11
- 5.2 Résolution . . . 11

A Définitions mathématiques **13**

Notes d’édition **13**

1. Introduction

1.1. Présentation du problème

Le fameux problème du loup, de la chèvre et du chou se déroule ainsi : “Un berger veut amener sa chèvre, son loup et son chou de l’autre côté d’une rivière. Son bateau est juste assez grand pour le prendre lui ainsi qu’un seul de ses animaux.” Nous allons tenter de résoudre ce problème en le modélisant, en le résolvant mathématiquement puis en étendant la solution à différentes variantes du problème.

Ce problème est le tout premier problème logique rédigé par un moine au IX ème siècle après J-C et il nous vient tout droit du folklore.

1.2. Définitions

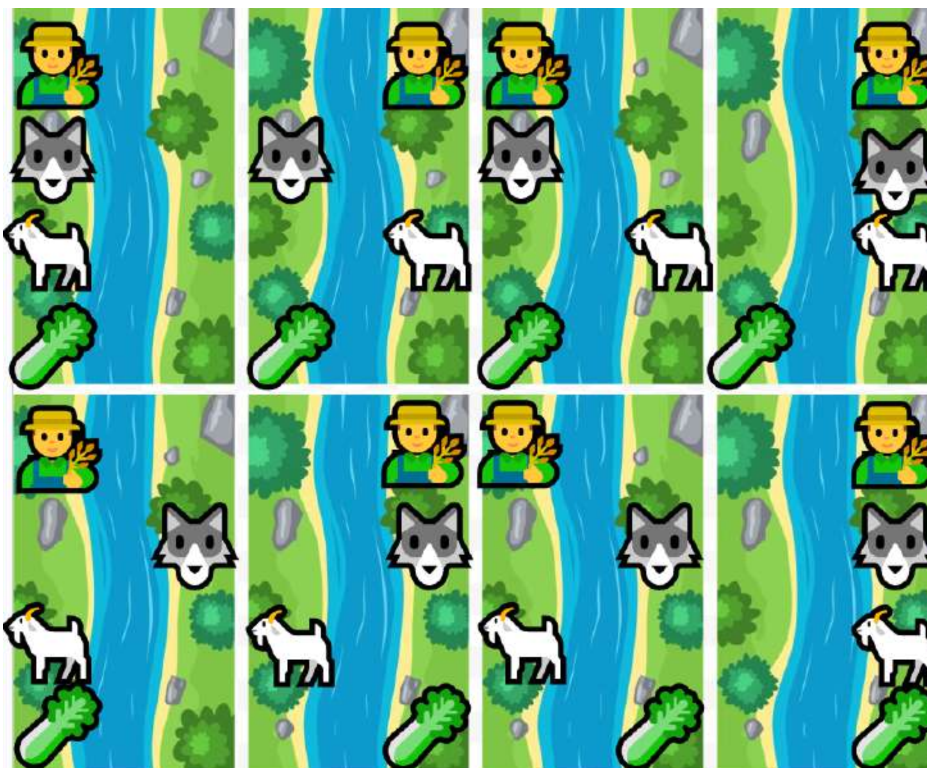
Definition 1. Un transport est ”**légal**” s’il est physiquement possible.

Exemple 1. Le berger ne peut pas transporter un animal si lui et l’animal ne sont pas sur la même rive.

Definition 2. Le mot ”**animaux**” définit les éléments présents dans le problème sauf le berger.

Definition 3. Un état de rive est ”**légal**” si aucun animal se fait manger.

1.3. Solution classique



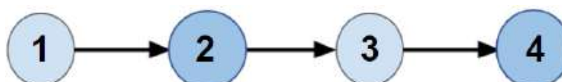
1. Le berger amène la chèvre de l’autre côté de la rivière.
2. Il revient tout seul.
3. Il prend le loup ou le chou.
4. Il revient avec la chèvre.
5. Il repart de l’autre côté de la rive avec le loup ou le chou (celui qui n’est pas déjà du bon côté).
6. Il revient tout seul sur la rive initiale.
7. Le berger traverse la rivière avec la chèvre : tout le monde est de l’autre coté.

2. Présentation des variantes possibles avec des graphes

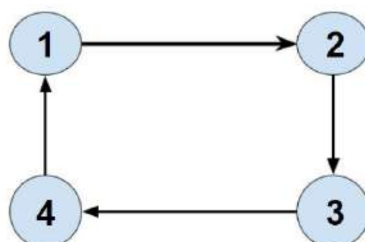
2.1. Types de variantes

NB : Dans ce sujet, nous nous sommes concentré sur les variantes linéaires car nous n'avons pas eu le temps de faire des recherches plus abouties. Nous présentons cependant les autres types de variantes que nous aurions pu étudier.

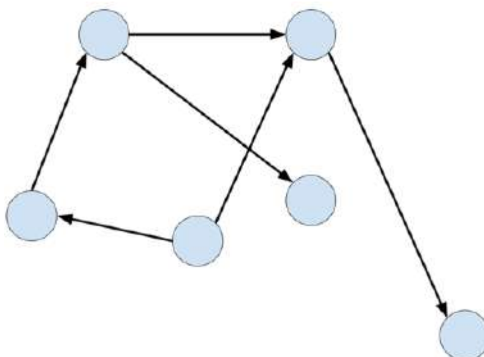
Dans ce genre de problème, il existe différents types de variantes. Il y a les variantes linéaires qui fonctionnent sous la forme d'une suite, c'est à dire 1 mange 2 qui mange 3 qui mange 4 comme on peut le voir ci-dessous.



Il y a ensuite les variantes circulaires qui fonctionnent sous la forme d'une boucle, c'est à dire 1 mange 2 qui mange 3 qui mange 4 et 4 qui mange 1 comme on peut le voir ci-dessous.



Enfin, il y a les variantes quelconques qui fonctionnent de façon aléatoire et qui n'ont pas de définition propre. Voici, ci dessous, un exemple de variantes quelconques.



2.2. Nombre de places sur le bateau requis pour chaque variantes

Voici comment trouver le nombre de places requis.

2.2.1 Linéaires et circulaires

La méthode pour connaître le nombre de places nécessaires avec les variantes circulaires et les variantes linéaires est presque la même. Pour les variantes linéaires et circulaires, si le nombre d'animaux est pair, il faut le diviser par 2. Nous obtiendrons le nombre de place minimum sur un bateau. Si le nombre d'animaux est impair, pour les variantes linéaires, il faut diviser par 2 puis enlever 0.5 et pour les variantes circulaires, il faut diviser par 2 puis rajouter 0.5. Cette dernière opération est nécessaire car il faut fermer la boucle.

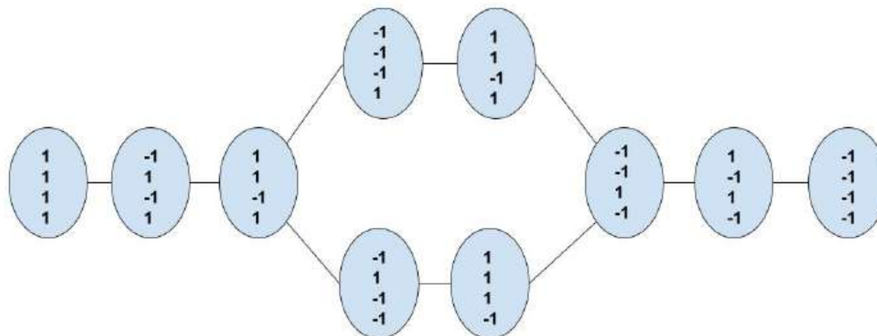
2.2.2 Quelconques

Les variantes quelconques fonctionnent de façon aléatoire donc il n'y a pas de méthode afin de trouver le nombre de places nécessaires sur le bateau. Cependant, il y a une résolution possible qui nécessite les graphes. Premièrement, il faut chercher sur le graphe les sommets qui ont le plus de connections avec les autres sommets. On comptera le nombre de sommets et on appellera cette variable n. Ensuite, il faut vérifier qu'il n'y ait plus aucune connections entre les derniers sommets. On comptera le nombre de sommets éliminés et on appellera cette variable v. Enfin, il suffit d'additionner n et v pour trouver le nombre de places nécessaires sur le bateau.

3. Résolution algorithmique des variantes linéaires grâce à des graphes

3.1. Graphe pour 3 animaux

Voici le graphe qui représente les étapes nécessaires pour 3 animaux, un passeur et une place sur le bateau : [1]



Le problème peut être résolu en utilisant les graphes. Le graphe ci-dessus utilise 1 pour la présence et -1 pour l'absence. L'ordre de la position des différents individus suit la logique : le berger, le loup, la chèvre puis le chou. On peut voir ici qu'il y a clairement deux chemins différents possibles et que lorsqu'on a 3 animaux, un passeur, et une place sur le bateau, le problème se résout en 7 étapes et il n'y a pas d'autres solutions. [2]

3.2. Résolution algorithmique avec des graphes

Algorithm 1 Algorithme avec les graphes

$A \leftarrow$ nombre d'animaux
$B \leftarrow$ nombre de places sur le bateau
$safeV \leftarrow []$
for $vertice$ **in** $\{1,-1\}^{A+1}$ **do**
  if legal($vertice$) **then**
    push $vertice$ in $safeV$
  end if
end for
$E \leftarrow []$
for (fr,to) in $safeV^2$ **do**
  if possible(fr, to) **and not** (to, fr) in E **then**
    push (fr, to) in E
  end if
end for
$graph \leftarrow$new Graph($safeV, E$)
return $graph$.BFS(FIRST, LAST)

FIGURE 1 – Programme traduit en Python [2]

```python
def solve(A: int, B: int):
  V = [i for i in range(2**(A+1))]
  safeV = [i for i in V if safe(i, A)]
  preE = []
  for fr in safeV:
    for to in safeV:
      if (to, fr) in preE:
        continue
        print(1)
      if possible(fr, to, A, B):
        preE.append((fr, to))
  E = [graph.Edge(e[0], e[1]) for e in preE]
  G = graph.UndirectedGraph(V, E)
  paths = G.path_exists(V[0],V[-1])
  return paths
```

Le programme n'est pas optimisé.

3.3. Optimisation avec C++ [1]

3.3.1 Aperçu

L'algorithme de base est un algorithme de découverte de graphes. Il est optimisé à l'aide de manipulations de bits et d'astuces dans l'algorithme de découverte.

3.3.2 État

Un état de la rive peut être représenté par une liste de 1 et de 0. 1 représentant la présence et 0 l'absence du berger et de ses achats, ainsi, l'état de la rivière est représenté d'une liste de 1 et de 0 de longueur $|A|+1$. Une liste de 1 et de 0 est une chaîne de bits, ou un nombre lorsqu'il s'agit d'ordinateurs.

Ainsi, en manipulant des nombres au niveau des bits avec des opérations binaires (opérations binaire AND, OR, NOT, etc...), nous pouvons parvenir à résoudre ce problème. Le bit le plus important est le bit représentant la présence du berger.

3.3.3 État légal

Un état légal est un état où aucun achat n'est mangé, en supposant que le n-ème achat mange le n+1-ème achat. Vérifier si un état est légal se déroule comme suit : pour chaque achat, si le prochain achat se trouve sur la même rive, et opposée a celle le berger, alors il est dangereux ; sinon il est légal. La vérification est effectuée à l'aide de la manipulation de bits.

3.3.4 Transports

Un transport est une chaîne de bits indiquant quels achats doivent être inversés, c'est-à-dire amenés de l'autre côté. Un transport ne peut s'appliquer sur un état que si tous les bits retournés sont égaux (état AND transport vaut 0 ou le transport [3]). L'application du transport consiste à effectuer un XOR sur l'état et le transport, en retournant les bits indiqués par le transport. Lorsqu'un graphe est initialisé, une liste de tous les transports possibles est générée. Ces transports sont toutes les combinaisons n-parmi-k possibles avec n=A et k allant de 0 à B. Ils sont générés à l'aide de la manipulation de bits.

3.3.5 Algorithme de base

L'algorithme de base de ce programme est un algorithme de découverte de graphes Depth First Search. Un sommet dans ce graphe représente un état légal de la rive du fleuve, et les sommets adjacents représentent les prochains états possibles de la rive du fleuve. L'algorithme de recherche DFS est récursif. Chaque étape de récursivité se déroule comme suit :

1. si le sommet actuel est le sommet de fin souhaité, retourner vrai
2. définir le sommet actuel comme visité
3. pour chaque sommet adjacent qui n'est pas encore visité, exécutez DFS pour ce sommet. Renvoyer vrai si l'un d'eux a renvoyé vrai
4. Sinon, retourner faux

3.3.6 Liste d'incidence

Auparavant, le programme comportait deux étapes : une étape de génération complète de graphes et une étape de recherche. Cependant, cette méthode était très coûteuse, car seule une fraction des sommets a été découverte lors de la recherche. Maintenant, à la place, la liste d'incidence pour un sommet donné est générée pendant la recherche DFS, donc aucune donnée n'est gaspillée. La particularité de cet algorithme DFS se trouve à l'étape 3. Les sommets adjacents se trouvent comme suit :

1. Pour chaque transport possible, vérifiez si le transport est possible [4]
2. Si oui, vérifiez si l'état transformé par ce transport a été visité ou non
3. Si oui, vérifiez si l'état transformé par ce transport est légal ou non
4. Si oui, alors l'état transformé par ce transport est un sommet adjacent.

Les conditions sont classées de la moins coûteuse à la plus coûteuse pour économiser le calcul.

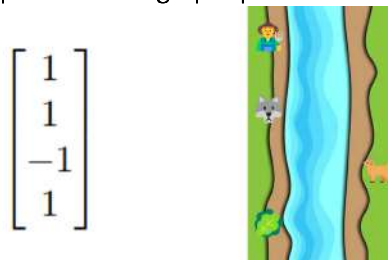
4. Tentative d'une solution analytique des variantes linéaires avec des matrices

4.1. Modélisation du problème original

4.1.1 Vecteur d'état et matrices de transport

Soit $s \in \{-1, 1\}^4$ un vecteur représentant l'état de la rive de départ, que nous nommerons vecteur d'état. Les composantes s_1, s_2, s_3, s_4 représentent respectivement la présence par 1 et l'absence par -1 du berger, du loup, de la chèvre et du chou. Ainsi, l'état de la rive opposée est représenté par $-s$.

FIGURE 2 – représentation graphique d'un vecteur d'état s



Les quatre matrices ci-dessous représentent respectivement les transports du berger, du loup, de la chèvre et du chou; nous les nommerons matrices de transports, qui forment l'ensemble T. Ainsi, l'expression $T_n \cdot s$ représente un transport du n-ème personnage.

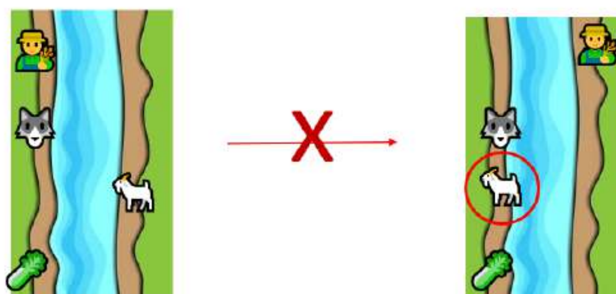
$$T_1 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_2 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_3 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_4 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Cependant, ces matrices de transport sont en réalité illégales. De tels transports se produisent lorsque nous tentons de transporter un animal alors qu'il se trouve sur la rive opposée de celle du berger.

Par exemple, étant donné que tous les membres sauf la chèvre sont présents sur une rive, le berger ne devrait pas pouvoir transporter la chèvre sans passer d'abord de l'autre côté. Cependant, en multipliant T_3 [chèvre] par le vecteur représentant l'état de la rive, on arrive à effectuer un transport illégal.

$$T_3 \cdot \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

FIGURE 3 – représentation d'un transport illégal



Cette situation peut être résolue en construisant de nouvelles matrices qui ne permettent pas de tels transports.

Afin d'éviter ces transports illégaux, on construit de nouvelles matrices. Lorsque celles-ci sont multipliées par le vecteur d'état qui aurait auparavant entraîné un transport illégal, redonne ce même vecteur d'état. Ces matrices sont trouvées en résolvant une série d'équations linéaires.

Redéfinissons les matrices T_1, T_2, T_3 et T_4 qui forment l'ensemble T :

$$T_1 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_2 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_3 = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_4 = \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}$$

Reprenons l'exemple précédente et essayons maintenant d'effectuer un transport illégal. Maintenant, quand nous effectuons la multiplication, le résultat est le vecteur facteur. [5]

$$T_3 \cdot \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

FIGURE 4 – représentation d'un transport légal corrigé

4.1.2 Suite de solution

Résoudre le problème consiste à trouver une séquence de transports qui mène tous les membres à être de l'autre côté, tout en respectant la règle principale du jeu : à aucun moment un prédateur et sa proie doivent être présents sur la même rive. Mathématiquement, cela signifie qu'il faut trouver une série de matrices $S = (S_i)_{S_i \in T}$ qui respecte la règle et le but sous forme de contraintes mathématiques.

Le but se traduit par le fait que le produit du vecteur d'état initial et de tous les matrices S_i doit être égal à l'opposé de celui-ci, ou

$$\left(\prod_i S_i\right) \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

La règle dit que le nième vecteur d'état ou son opposé, engendré par le produit du vecteur d'état initial et des n premières matrices, ne peut être jamais égal à un vecteur dit illégal.

$$c_n = \left(\prod_{i \leq n} S_i\right) \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\forall n \quad c_n, -c_n \notin \left\{ \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \right\}$$

4.2. Résolution algorithmique du problème

Bien que la motivation derrière la modélisation des transports sous forme de matrices était de formuler le problème sous forme d'une équation d'algèbre linéaire, il ne peut pas être résolu de cette façon. Ceci est dû au fait que nous voulons trouver une suite de matrices, et non pas une seule matrice. On peut donc seulement construire un algorithme basé sur les définitions mathématiques ci-dessous.

Imposons une contrainte qui interdit S d'engendrer deux fois le même vecteur d'état.

$$\forall a \neq b \quad \left(\prod_{i \leq a} S_i\right) \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \neq \left(\prod_{i \leq b} S_i\right) \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

L'algorithme itère jusqu'à ce que le vecteur d'état soit égal au vecteur final. [6] À chaque itération, l'algorithme prend successivement le produit de chaque matrice de transport et du vecteur d'état actuel, initialement le vecteur initial. Si ce produit ou son opposé n'est pas un vecteur "interdit", et n'a pas été atteint précédemment, on a alors satisfait les contraintes imposées. Dans ce cas, le vecteur d'état devient ce produit, nous ajoutons cette matrice à la solution et nous passons à l'itération suivante. Si aucun produit n'a satisfait ces conditions, alors nous avons atteint une "impasse". Nous devons donc revenir en arrière en mettant le vecteur d'état égal à l'état atteint dans l'itération précédente et en enlevant la dernière matrice de transport de la suite de solutions.

Algorithm 2 Algorithme employant les matrices

```
Solution ← [] (FILO)
Precedents ← [] (FILO)
s ← [1,1,1,1]
i ← −1
while s ≠ [−1,−1,−1,−1] do
  for M in T do
    if (s × M not in Interdits) and (−s × M not in Interdits) and (s × M not in Precedents) then
      s ← s × M
      push s to Precedents
      push M to Solution
      break
    end if
  end for
  if did not break then
    i ← i − 1
    s ← Precedents[i]
    pop Solution
  end if
end while
```

4.3. Tentative d'étente à des variantes

Étendre le problème à des variantes dont le nombre de places sur le bateau est supérieur à 1 est impossible du fait que, dans ces cas, les transports ne peuvent plus être modélisés par des transformations linéaires que représentaient les matrices [7]. En effet, trouver des matrices qui modélisent correctement les transports pour $B > 1$ est impossible. En prenant $A = 2$ et $B = 2$, nous nous rendons compte qu'il existe plus d'équations linéairement indépendantes à satisfaire qu'une matrice de transport peut satisfaire.

5. Résolution analytique des variantes linéaires grâce à des ensembles

5.1. Définitions

Soit $U = \{-1,1\}^{A+1}$ l'ensemble de tous les états possibles de la rive initiale du fleuve. Pour tout $u \in U$ et tout $i > 1$, u_1 et u_i représentent respectivement par 1 la présence et -1 l'absence du berger et de ses achats.

Soit $U_s \subset U$ l'ensemble de tous les états légals, c'est-à-dire les états où aucun achat n'est mangé. Un état est légal sauf s'il y a un prédateur et une proie sur la même rive tout en étant sur la rive opposée du berger. Cela peut se traduire par le fait que si une composante dans un état est égale à la composante suivante, elle doit être égale à la première composante. Nous notons :

$$U_s = \{u \in U : \forall i > 1 \quad u_i = u_{i+1} \implies u_i = u_1\}$$

Soit $T \subset U$ l'ensemble de tous les transports possibles. Un transport est un tuple indiquant par -1 quels composantes de l'état doivent être inversées si possible. Puisque l'agriculteur change de rive à chaque déplacement, le premier élément d'un transport doit être -1. De plus, puisque pas plus de B achats peuvent changer de rive à la fois, pas plus de B éléments autres que le premier ne peuvent être égaux à -1. Nous notons :

$$T = \{-1\} \times \{t \in \{-1,1\}^A : \#\{i : t_i = -1\} \leq B\}$$

Soit $*$ l'opération binaire désignant le transport. Un transport ne peut être opéré sur un état que s'il est possible, c'est-à-dire si tous les éléments transportés se trouvent sur la même rive du fleuve. Autrement dit, un transport est possible si pour tous les éléments égaux à -1 dans le transport, les éléments correspondants dans l'état doivent tous être égaux soit à -1 soit à 1. On note [8] :

$$\forall u \in U, t \in T \quad u * t = \begin{cases} (u_i t_i)_i & \text{si } [\forall i \quad t_i = -1 \implies u_i = 1] \text{ ou } [\forall i > 1 \quad t_i = -1 \implies u_i = -1] \\ \text{indéfini} & \text{sinon} \end{cases}$$

Résoudre le problème consiste à trouver une suite $(t_i)_{t_i \in T}$ qui vérifie deux conditions :

$$u_{\#t} = -\vec{1} \tag{1}$$

$$\forall i \quad u_i \in U_s \tag{2}$$

$$\text{où} \quad u_n = \vec{1} * t_1 * t_2 * \cdots * t_n$$

indiquant respectivement que (1) les transports successifs doivent amener tout le monde à se trouver sur l'autre rive du fleuve et que (2) les transports successifs doivent toujours générer des états légals.

5.2. Résolution

Si on tente de résoudre le problème pour $B < \lfloor \frac{A}{2} \rfloor$, on se rend rapidement compte que c'est impossible dès le premier transport, puisqu'on ne peut pas transporter suffisamment d'animaux lors de la première étape pour avoir un état légal.

***Lemma* 5.1.** *Il n'y a pas de solutions pour* $B < \lfloor \frac{A}{2} \rfloor$

Démonstration. Par la condition (1), si pour tout $B < \lfloor \frac{A}{2} \rfloor$, il y a $u_1 \notin U_s$ alors on a démontré le lemme. Voici la preuve par contradiction :

Supposons que $u_1 \in U_s$.

$$\begin{aligned} & t_1 \in T \\ & \implies (t_1)_1 = -1 \\ & \implies (\vec{1} * t_1)_1 = -1 \\ & \implies (u_1)_1 = -1 \end{aligned}$$

Puisque $(u_1)_1 = -1$, par définition de U_s nous devons avoir

$$\forall i > 1 \;\; (u_1)_i = (u_1)_{i+1} \implies (u_1)_i = -1$$

D'après cette condition, pour minimiser $\#\{i > 1 : (u_1)_i = -1\}$, il faut qu'on aie $(u_1)_i \neq (u_1)_{i+1}$. [9] Ceci ce traduit par le fait que chaque achats doit être sur la rive opposée que ses voisins. Le nombre d'achats égalant -1 doit alors être supérieur a la moitié du nombre d'achats.

$$\#\{i > 1 : (u_1)_i = -1\} \geq \lfloor \frac{A}{2} \rfloor$$

Remarquez que $t_1 = \vec{1} * t_1 = u_1$. Nous avons donc $u_1 \in T$. De cela, nous devons aussi avoir $\#\{i > 1 : u_i = -1\} \leq B$. Cependant, cela conduit à $B \geq \lfloor \frac{A}{2} \rfloor$, en contradiction avec la déclaration d'origine. □

La solution de n'importe quelle variantes linéaires est simplement une extension de la solution du problème original. Nous supposons bien légal que $B \geq \lfloor \frac{A}{2} \rfloor$. Il y a deux cas :

Cas 1 : Le nombre d'animaux est pair La solution est alors :

$$t_1 = (\underline{-1}, -1, 1, -1, 1, -1, \cdots)$$

$$t_2 = (\underline{-1}, 1, 1, 1, 1, \cdots)$$

$$t_3 = (\underline{-1}, 1, -1, 1, -1, \cdots)$$

Cas 2 : Le nombre d'animaux est impair La solution ici est :

$$t_1 = (\underline{-1}, 1, -1, 1, -1, \cdots, 1)$$

$$t_2 = (\underline{-1}, 1, 1, 1, 1, \cdots, 1)$$

$$t_3 = (\underline{-1}, -1, 1, -1, 1, \cdots, 1)$$

$$t_4 = (\underline{-1}, 1, -1, 1, -1, \cdots, 1)$$

$$t_5 = (\underline{-1}, 1, 1, 1, 1, \cdots, -1)$$

$$t_6 = (\underline{-1}, 1, 1, 1, 1, \cdots, 1)$$

$$t_7 = (\underline{-1}, 1, -1, 1, -1, \cdots, 1)$$

A. Définitions mathématiques

pour tout, il existe : Les symboles $\forall$ et $\exists$ signifient respectivement ”pour tout” et ”il existe”.

Cardinalité : Le nombre d'éléments dans un ensemble A est nommé cardinalité de A et est note $|A|$ ou $\#A$.

Produit cartésien : pour tout ensemble A et B, le produit cartésien $A \times B$ est l'ensemble de tout les tuples (a, b) tel que $a \in A$ et $b \in B$.

$$A \times B = \{(a, b) : a \in A, b \in B\}$$

Le produit d'un ensemble A par lui même n fois peut être écrit sous la forme de A^n

”put”, ”push” et types de listes : les commandes ”put” et ”pop” (mettre et enlever) indiquent respectivement d'ajouter un objet a une liste et en d'enlever un. Les listes diffèrent entre eux dans ordre on ajoute ou enlève des éléments. Par exemple, les listes ”First In Last Out” (FILO) sont des liste où les premiers éléments ajoutés sont les derniers à sortir.

la notation $liste[i]$: pour indiquer le i-ème élément d'une liste on pourra utiliser $liste[i]$. Une indice négative signifie qu'on commence à compter du dernier élément.

Références

[1] A. Gana. (2023) Wold, goat, cabbage : Bitwise in c++. MEJ LVVG. [Online]. Available : https://replit.com/@AzharGana/CLCc

[2] ——. (2023) Wold, goat, cabbage : Graphs in python. MEJ LVVG. [Online]. Available : https://replit.com/@AzharGana/LCcGraph

Notes d'édition

[1] Dans la configuration en haut à droite, il y a une erreur due certainement à une inattention : la séquence correcte est 1, -1, 1, 1.

[2] Dans ce cas particulier, on peut conclure qu'il n'existe pas d'autres solutions simplement en observant le graphique. On pouvait dire que, par la suite, des techniques sont présentées pour déterminer l'ensemble de toutes les solutions possibles dans le cas général.

[3] Cette phrase est peu claire. Il vaut mieux rendre les opérations plus explicites : (état AND transport $= 0$) ou (état AND transport $=$ transport). Il faut également ajouter la condition que le berger doit toujours être transporté.

[4] Écrite de cette manière, la condition est banalement toujours vérifiée. On voulait probablement dire qu'il faut vérifier que, pour un transport possible donné, il est possible de l'appliquer à l'état considéré.

[5] La discussion concernant les nouvelles matrices T_2, T_3 et T_4 devrait être plus approfondie. Par exemple, on pouvait expliquer pourquoi, lorsque le berger et l'animal se trouvaient sur des rives opposées, les nouvelles matrices donnent comme résultat la situation de départ.

[6] Il y a certains points qui devraient être discutés. Sommes-nous sûrs que l'état final sera atteint ? Dans le cas particulier que nous considérons, la réponse est évidente, mais en général ? Si l'état final n'est jamais atteint, comment se comporte l'algorithme ? À l'inverse, s'il existe plusieurs chemins menant à l'état final (comme dans le graphique à la p. 5), l'algorithme les trouve-t-il tous ?

[7] C'est vrai que le cas où $B > 1$ est très compliqué. Mais si nous disons 'impossible', il faudrait donner au moins un exemple de tentative de modélisation par des transformations linéaires.

[8] L'opération considérée dans la première ligne est appelée produit de Hadamard.

[9] Cette inégalité et l'observation suivante concernent les valeurs de i telles que $(u_1)_i$ ou $(u_1)_{i+1}$ sont égaux à 1.