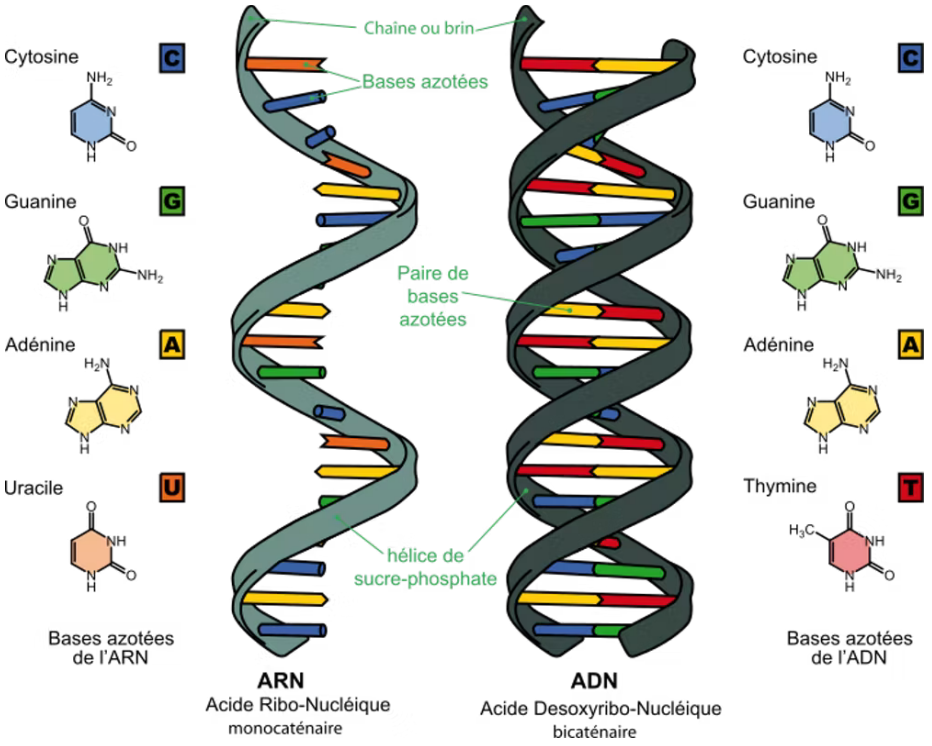
BERTAUX Richard
SANCHEZ BERMUDEZ Yaiza
PÉRIGNAC Juliette
BOUKHEMIA Clara

Responsable UE : M. BOUCELMA Omar
Tuteur : M. BENHAMOU Belaid

Projet de Licence Mathématiques et Informatique :

Modélisation de la structure secondaire de l'ARN



Année Scolaire : 2021-2022

Aix*Marseille université
Initiative d'excellence

Table des matières

1. Introduction

L'ADN, ou Acide Désoxyribonucléique, est une macromolécule présente dans toutes les cellules de notre corps et qui contient toute notre information génétique, appelée génome. L'ADN est l'initiateur du développement, du fonctionnement et de la reproduction des êtres vivants. Situé dans le noyau de nos cellules, il a une structure bien précise. En effet, les molécules d'ADN s'agencent en une double hélice, composée de deux brins enroulés l'un autour de l'autre. Un brin est une séquence de nucléotides, chacun constitué d'une des bases azotées suivantes : l'adénine (A), la cytosine (C), la guanine (G) ou la thymine (T). C'est l'ordre dans lequel se succèdent les nucléotides le long d'un brin d'ADN qui constitue la séquence qui porte l'information génétique. De plus, cette information est contenue dans les chromosomes, et plus précisément dans les différents gènes qui les composent, comme on le voit sur la figure ci-dessous.

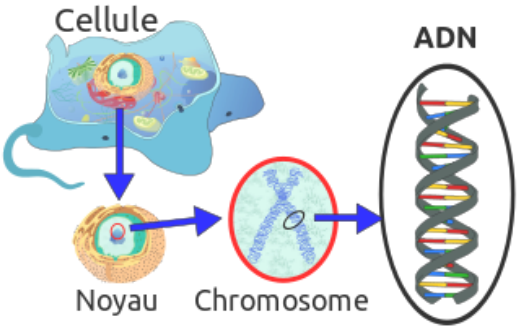


Figure 1 : L'ADN nucléaire d'une cellule d'eucaryote est situé dans des chromosomes au sein du noyau.

En revanche, pour notre projet, ce n'est pas exactement l'ADN qui nous intéresse, mais l'ARN, ou Acide Ribonucléique, qui a une structure chimique très proche de celle de l'ADN. L'ARN est généralement synthétisé dans le noyau des cellules, à partir d'un segment d'ADN dont il est une copie. En particulier, c'est grâce à l'ARN, qu'on appelle à un certain stade ARN messager, que les cellules demandent la synthèse de différentes protéines, visant à répondre à leurs besoins vitaux. Contrairement à l'ADN, l'ARN est constitué d'un seul brin, aussi représenté par une séquence de nucléotides. Les bases azotées sont les mêmes que pour l'ADN, à l'exception de la thymine (T), qui est remplacée par l'uracile (U) lors de la copie de l'ADN. Cela étant dit, on peut s'intéresser au rôle de l'ARN dans la synthèse des protéines, et plus particulièrement à une certaine phase : le repliement de l'ARN.

Lorsque l'ARN messager sort du noyau, il se tord et se replie sur lui-même. Par plusieurs processus chimiques, celui-ci va alors pouvoir passer de sa structure primaire à sa structure secondaire, qu'on appelle aussi structure stable. Cette forme est maintenue pendant seulement quelques secondes, puis l'ARN donne une protéine. Ce qui nous intéresse pour ce projet est donc la phase de repliement de l'ARN, et sa forme stable, très difficile à isoler. En effet, la connaissance de la structure stable de l'ARN a de nombreux intérêts dans le domaine médical. Le premier avantage de savoir à quoi ressemble un brin d'ARN sous sa forme stable serait de prédire la protéine qui sera créée par la suite. Il arrive aussi que l'ARN se replie mal, et que la protéine créée ne puisse pas exercer sa fonction. Dans le pire des cas, celle-ci sera amenée à exercer une autre fonction néfaste pour l'organisme, si bien qu'elle pourra entraîner des maladies. Grâce à la connaissance de la structure secondaire de l'ARN, il sera alors possible d'arrêter un tel processus, et ainsi d'empêcher le développement de maladies telles que le cancer. Enfin, au-delà du domaine médical, on pourrait tout simplement orienter l'ARN, pour qu'il puisse fabriquer une protéine souhaitée.

Notre problème consiste alors à prédire la structure secondaire d'une molécule d'ARN, compte tenu uniquement de sa séquence de nucléotides. Cet important problème biologique peut être résolu à l'aide de la programmation dynamique, mais pour ce projet nous utiliserons la programmation linéaire en entiers (ILP : Integer Linear Programming), qui sera tout aussi efficace et qui permettra d'ajouter des contraintes plus facilement.

On cherchera alors à nous rapprocher le plus possible de la forme la plus stable de l'ARN. Pour cela, nous nous efforcerons de modéliser le problème de la structure stable de l'ARN (RNA Folding Problem), d'abord avec une modélisation simplifiée. Puis, notre but étant de nous rapprocher le plus possible de la réalité, nous étendrons le modèle biologique et la formulation ILP afin d'y incorporer des contraintes de stabilité et des caractéristiques biologiques plus réalistes du problème.

2. Une première modélisation du problème

2.1. Des outils nécessaires à la modélisation

Avant de rentrer dans la modélisation du problème, regardons de plus près comment fonctionnent les outils que nous allons utiliser. Tout au long du projet, nous traiterons de nouvelles contraintes afin de nous rapprocher de la forme la plus stable possible de l'ARN. Compte tenu d'une séquence de nucléotides donnée, on pourra grâce à la programmation linéaire en entiers écrire toutes les contraintes de stabilité associées à cette séquence dans un fichier texte.

Cette programmation permet de considérer des **problèmes d'optimisation**. Elle utilise pour traiter ces problèmes une fonction de coût, qu'on appelle fonction objective, des variables entières et des contraintes linéaires sur ces variables. Le but est alors de minimiser ou de maximiser la fonction objective. On peut écrire un certain problème de plusieurs manières différentes, mais qui auront toujours la même forme globale, comme sur l'exemple ci-dessous.

$$\begin{array}{ll} \text{maximiser} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{tel que} & A\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \\ \text{et} & \mathbf{x} \in \mathbb{Z}, \end{array} \quad , \qquad \begin{array}{ll} \text{maximiser} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{tel que} & A\mathbf{x} + \mathbf{s} = \mathbf{b}, \\ & \mathbf{s} \geq \mathbf{0}, \\ & \mathbf{x} \geq \mathbf{0}, \\ \text{et} & \mathbf{x} \in \mathbb{Z}, \end{array}$$

Figure 2 : Formes canonique et standard d'un problème d'optimisation linéaire en nombres entiers (c et b étant des vecteurs et A une matrice à valeurs entières)

La programmation se fera en Python, et à chaque nouvelle contrainte ajoutée, on pourra créer un nouveau fichier incluant de nouvelles variables, afin d'affiner notre modélisation. Par la suite, on verra plus en détail comment on utilise la programmation en entiers pour transformer en contraintes claires ce qui se passe au niveau biologique et chimique lors du repliement de l'ARN.

Maintenant que nos contraintes sont stockées dans un fichier, il reste à résoudre le problème, c'est-à-dire à maximiser la fonction objective. Pour cela, nous utiliserons le solveur Gurobi, qui fonctionne via le terminal de nos ordinateurs. Une fois le fichier donné à Gurobi, celui-ci prend en compte les contraintes et propose une valeur objective, ainsi que les valeurs correspondantes associées à chaque variable du problème. On pourra grâce à cet outil faire un nombre considérable de tests sur des séquences de nucléotides générées aléatoirement et de tailles différentes. Aussi, tous ces tests seront effectués sur la même machine pour ne pas fausser les résultats obtenus.

2.2. Définitions utiles à la formulation des contraintes

Mais avant de nous lancer dans les tests, nous allons voir plusieurs définitions nécessaires à la compréhension du repliement de l'ARN.

Tout d'abord, introduisons la **notion d'appariement**. Un appariement définit un ensemble de paires disjointes des lettres qui composent une séquence. Par exemple, si on a la séquence ACGUGCCACGAU, un appariement possible est l'ensemble {(A,C), (G,U), (G,C), (C,A), (C,G), (A,U)}. Le fait que les paires soient disjointes spécifie qu'un caractère ne peut pas être dans plus d'une paire. En revanche, certains caractères peuvent ne faire partie d'aucune paire (par exemple si la séquence contient un nombre impair de caractères).

Une paire d'un appariement est dite complémentaire si les deux caractères de la paire sont (A,U) ou (C,G). Un appariement est alors complémentaire si toutes les paires le composant sont complémentaires. En fait, la

complémentarité d'un appariement est l'une des contraintes nécessaires au repliement de l'ARN. En effet, d'un point de vue biologique, les caractères A et U représentent les nucléotides qui peuvent se lier entre eux, de même que les caractères C et G.

En reprenant notre séquence de nucléotides, on peut représenter les paires complémentaires en reliant les deux nucléotides de la paire par une courbe. **Un appariement peut alors être imbriqué** si les courbes correspondant à chaque paire de l'appariement ne se croisent pas, comme on peut le voir sur la figure ci-dessous.

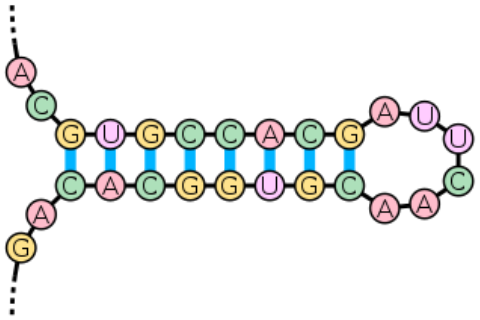


Figure 3 : Exemple d'appariement imbriqué sur un brin d'ARN

Pour commencer avec une modélisation simplifiée du problème de repliement de l'ARN, nous considérons que la stabilité d'un appariement imbriqué se mesure par le nombre de paires appariées qui le composent. Étant donnée une séquence de nucléotides, on doit donc trouver un appariement imbriqué qui maximise le nombre de liaisons entre deux nucléotides.

2.3. Formulation des contraintes en programmation ILP

Dans un premier temps, notre but est de trouver pour une séquence donnée un appariement imbriqué ne contenant que des paires complémentaires et disjointes. Nous devons donc modéliser ces contraintes dans la formulation ILP.

La programmation linéaire en entiers pour le problème du repliement de l'ARN fonctionne avec des variables binaires qu'on notera P(i,j) pour chaque paire de nucléotides dont les positions sont indicées i et j, avec $1 \leq i < j \leq n$, où n est la longueur de la séquence donnée. Ainsi, si P(i,j) = 1, alors les nucléotides i et j seront appairés, sinon ils ne seront pas appairés et P(i,j) = 0. Modélisons alors trois premières contraintes.

Tout d'abord, tout appariement doit être complémentaire, c'est-à-dire que si la paire de nucléotides placés en positions i et j ne fait pas partie des paires (A,U), (U,A), (C,G) ou (G,C), alors on interdit l'appariement et P(i,j) = 0.

Ensuite, chaque nucléotide doit être apparié au plus une fois. On aura donc la contrainte suivante pour toutes les positions j de la séquence de nucléotides : $\sum_{k<j} P(k,j) + \sum_{k>j} P(j,k) \leq 1$.

Enfin, les paires doivent être imbriquées. En d'autres mots, elles ne doivent pas se croiser. On peut donc écrire la contrainte suivante pour toutes les positions i, j, k, l telles que i < k < j < l, on aura l'inégalité : $P(i,j) + P(k,l) \leq 1$. On pourra ainsi éviter les appariements comme celui présenté sur la figure ci-dessous, pour favoriser ceux ayant une forme similaire à ce qu'on a vu précédemment sur la figure 3, et qui sont plus stables.

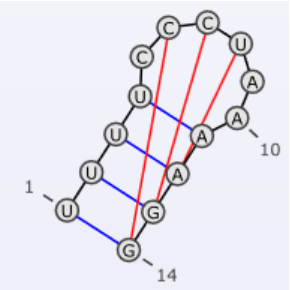


Figure 4 : Exemple d'appariement non imbriqué sur un brin d'ARN

Nous disposons maintenant de toutes les contraintes nécessaires à la première modélisation. Pour utiliser la programmation en entiers, il nous reste à écrire une fonction objective qui maximise le nombre de variables affectées à la valeur 1, c'est-à-dire maximiser le nombre de paires de l'appariement afin d'obtenir l'appariement le plus stable. Notre fonction objective sera donc : $f_1 = max \sum_{i<j} P(i,j)$.

2.4. Un premier exemple

Pour commencer, nous avons considéré la séquence de nucléotides suivante : ACUGU. Le programme first-RNA donné par notre tuteur génère le fichier suivant, qui prend en compte les contraintes de base :

\

```
Maximize
+ P(1,2)
+ P(1,3)
+ P(1,4)
+ P(1,5)
+ P(2,3)
+ P(2,4)
+ P(2,5)
+ P(3,4)
+ P(3,5)
+ P(4,5)
such that
P(1,2) = 0
P(1,4) = 0
P(2,3) = 0
P(2,5) = 0
P(3,4) = 0
P(3,5) = 0
P(4,5) = 0
+ P(1,2) + P(1,3) + P(1,4) + P(1,5) <= 1
+ P(1,2) + P(2,3) + P(2,4) + P(2,5) <= 1
+ P(1,3) + P(2,3) + P(3,4) + P(3,5) <= 1
+ P(1,4) + P(2,4) + P(3,4) + P(4,5) <= 1
+ P(1,5) + P(2,5) + P(3,5) + P(4,5) <= 1
P(1,3) + P(2,4) <= 1
P(1,3) + P(2,5) <= 1
P(1,4) + P(2,5) <= 1
P(1,4) + P(3,5) <= 1
P(2,4) + P(3,5) <= 1
binary
P(1,2)
P(1,3)
P(1,4)
P(1,5)
P(2,3)
P(2,4)
P(2,5)
P(3,4)
P(3,5)
P(4,5)
end
```

FONCTION OBJECTIVE

CONTRAINTE DE COMPLÉMENTARITÉ

CHAQUE NUCLÉOTIDE EST APPARIÉ AU PLUS UNE FOIS

L'APPARIEMENT DOIT ÊTRE IMBRIQUÉ : LES LIENS ENTRE LES NUCLÉOTIDES NE DOIVENT PAS SE CROISER

LES VARIABLES SONT BINAIRES :
$P(i,j) = 1$ SI LES NUCLÉOTIDES SONT APPARIÉS
$P(i,j) = 0$ SINON

Figure 5 : Fichier généré par le programme first-RNA pour la séquence ACUGU

En exécutant Gurobi, on obtient la solution suivante :

```
# Objective value = 2
P(1,2) 0
P(1,3) 0
P(1,4) 0
P(1,5) 1
P(2,3) 0
P(2,4) 1
P(2,5) 0
P(3,4) 0
P(3,5) 0
P(4,5) 0
```

Figure 6 : Fichier généré par Gurobi en prenant en entrée le fichier de la figure 5

L'appariement trouvé contient donc deux paires situées aux positions (1,5) et (2,4) qui correspondent aux paires (A,U) et (C,G). Cet appariement respecte bien toutes les contraintes citées précédemment.

2.5. Etude du temps de génération et de résolution

Maintenant que nous avons bien compris le fonctionnement de Gurobi, nous pouvons effectuer des tests sur des séquences de nucléotides de tailles différentes. Pour cela, nous générons des séquences aléatoires de nucléotides grâce au code Python donné en annexes (*Annexe 1*). Pour obtenir des résultats assez représentatifs, on procède comme suit : on génère 15 séquences aléatoires de taille 10, puis de taille 20, puis on va de 10 en 10 jusqu'à une taille 100. Pour cela, nous avons écrit un script python permettant à partir de ces séquences de sauvegarder le temps de création de chacune dans un fichier Excel en fonction de leur taille (*Annexe 2*).

On récupère le temps de génération de chaque fichier par le programme first-RNA, puis on exécute Gurobi pour chaque fichier généré, pour récupérer le temps de résolution donné par le logiciel. On obtient alors les deux courbes suivantes :

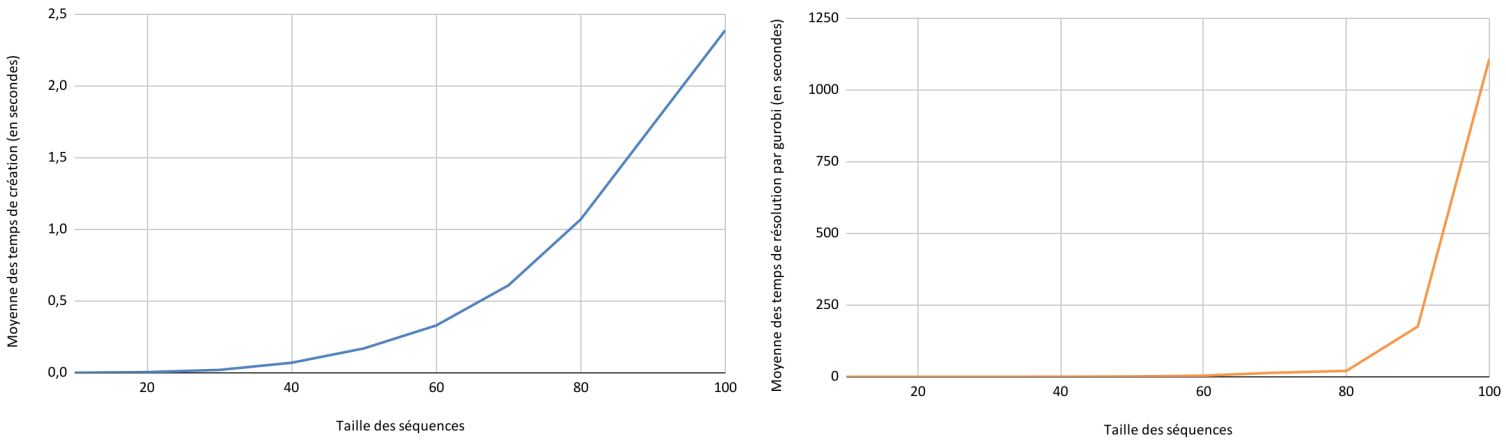


Figure 7 : Représentation du temps de création des fichiers et du temps de résolution pour la première modélisation en fonction des tailles des séquences

On remarque que le temps de génération des fichiers et de résolution du problème avec le solveur Gurobi sont représentés par une fonction exponentielle. L'ARN pouvant contenir plusieurs milliers de nucléotides, notre code semble peu efficace pour des séquences très longues. De plus, d'après le graphique, Gurobi est un solveur efficace pour de petites séquences d'ARN (quelques centaines de nucléotides) mais risque d'être très long pour une séquence avec plusieurs milliers de nucléotides. On se demande alors si l'ajout de contraintes supplémentaires augmentera le temps de résolution.

Plus tard, nous réitérerons donc les mêmes tests pour chaque nouvelle contrainte ajoutée à la modélisation, afin de comparer les résultats obtenus.

3. Vers une modélisation plus proche de la réalité

Après ce premier exemple, il est évident que nous devons apporter plus de contraintes à notre modèle de base pour le rapprocher le plus possible de la réalité. Pour ce faire nous devons ajouter 4 contraintes, que nous traiterons de la même manière que l'exemple précédent. Chaque contrainte étant programmée différemment, nous devons avoir une idée de comment évoluent la génération du code pour Gurobi via python et la résolution du problème posé pour chaque contrainte.

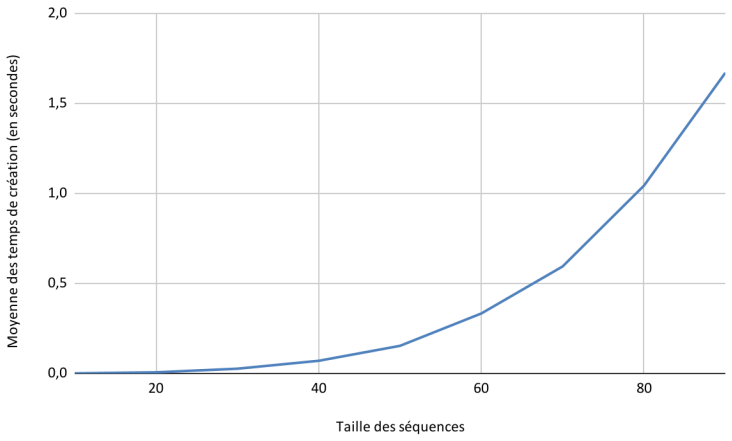
3.1. Fréquence d'apparition pour chaque nucléotide

Lors de la création d'une séquence d'ARN, la fréquence d'apparition des différents nucléotides n'est pas équiprobable. Pour modéliser le plus fidèlement une séquence dans un fichier, il est nécessaire de créer aléatoirement la séquence en prenant en compte les coefficients d'apparition de chaque nucléotide (*Annexe 3*). En revanche, n'ayant pas eu accès à ces coefficients, nous serons contraints de poursuivre la modélisation avec une fréquence d'apparition équiprobable pour les différents nucléotides.

3.2. Distance minimale entre deux nucléotides appariés

L'ARN étant très fragile, il ne peut pas former une arête entre deux nucléotides consécutifs. Il y a un pas minimum à respecter pour former la première arête afin de respecter cette contrainte chimique.

Pour modéliser cette contrainte, il suffit de déterminer une distance minimale notée d puis de parcourir l'ensemble des positions i et j, avec i < j, et lorsque la différence j-i est inférieure à d, on affecte la valeur de P(i,j) à 0 afin que les nucléotides i et j ne soient pas appariés (*Annexe 4*). En considérant les données biologiques mises à notre disposition, nous avons choisi $d = 4$.



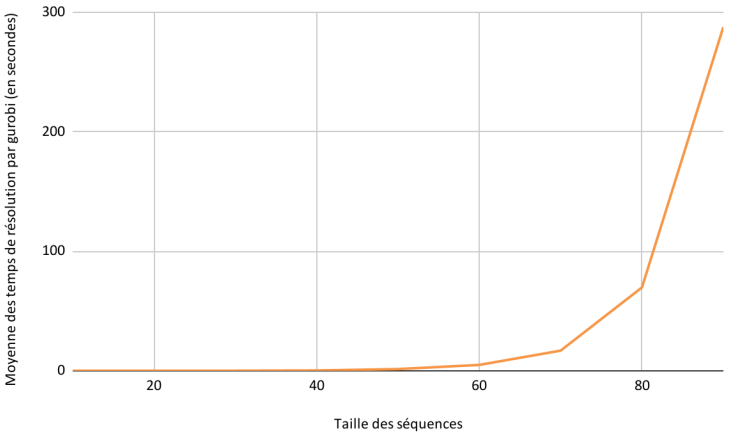


Figure 8 : Représentation du temps de création des fichiers et du temps de résolution pour la contrainte : distance minimale entre deux nucléotides en fonction des tailles des séquences

3.3. Association d'un poids à chaque combinaison complémentaire

Les combinaisons complémentaires (A-U et C-G) n'ont pas le même poids. Le lien entre C et G contient 3 atomes d'hydrogène et sera donc plus solide qu'entre A et U. Ainsi, l'appariement de nucléotides C-G sera plus stable que celui de A-U. On cherche donc à favoriser les appariements C-G et à leur associer un poids supérieur au poids de l'appariement A-U.

Afin de modéliser cette contrainte, il suffit de modifier la fonction objective en testant pour chaque paire de nucléotides situés à la position (i,j) quels sont les nucléotides associés. Dans le cas où l'on a une paire A-U ou U-A, on multiplie la variable binaire P(i,j) par une constante *a*, qui correspond au poids de la paire A-U. Dans le cas d'une paire C-G ou G-C, on multiplie P(i,j) par une constante *b* qui correspond au poids de la paire C-G, avec $a < b$ (*Annexe 5*). En pratique, le poids de la paire A-U est égal à 1 et le poids de la paire C-G est compris entre et 1,33 et 1,5, nous avons donc choisi b = 1,4.

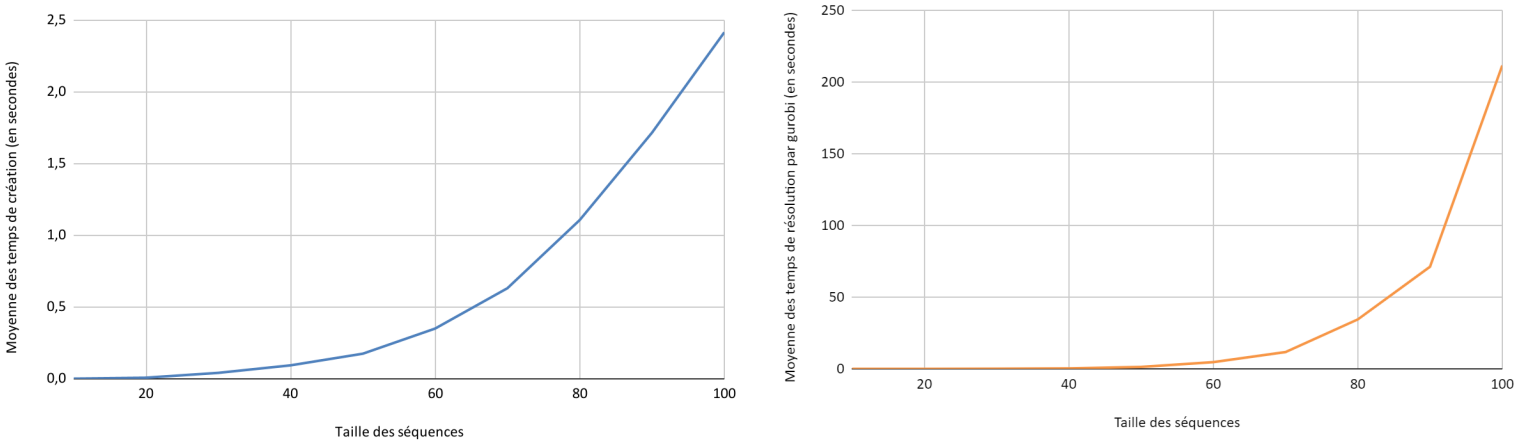


Figure 9 : Représentation du temps de création des fichiers et du temps de résolution pour la contrainte : association d'un poids à chaque combinaison complémentaire

3.4. Choix des arcs formant des piles

La structure de l'ARN est plus stable suivant le placement des arcs. En effet, des appariements empilés la rendront plus stable que ceux formés qui se suivent. Ainsi les arcs privilégiés dans le repliement de l'ARN sont ceux qui forment une "pile". Il faut donc maximiser le nombre de paires imbriquées. Pour cela, on va considérer une paire de nucléotides situés à la position (i,j), qu'on écrira seulement (i,j) pour faciliter l'explication.

Étudions donc le quadruplet (i, i+1, j-1, j), qu'on associera ensuite à la variable Q(i,j). On cherche à affecter la valeur 1 à ce quadruplet si et seulement si les paires (i,j) et (i+1,j-1) sont également affectées à la valeur 1. On peut noter cela de la manière suivante :

Q(i,j) = 1 si et seulement si P(i,j) =1 et P(i+1, j-1) = 1

Transformons cela en deux équations linéaires afin de l'implémenter dans Gurobi :
On a $P(i,j)\ et\ P(i+1,\ j-1) \Rightarrow Q(i,j)$ et d'autre part, $Q(i,j) \Rightarrow P(i,j)\ et\ P(i+1,\ j-1)$.

Ce qui est équivalent aux deux équations suivantes :

$$P(i,j) + P(i+1,\ j-1) - Q(i,j) \leq 1 \qquad (1)$$
$$2Q(i,j) - P(i,j) - P(i+1,\ j-1) \leq 0 \qquad (2)$$

On veut à présent maximiser le nombre de quadruplets pour avoir la structure la plus stable possible. Nous allons donc tester les fonctions objectives suivantes : *(Annexes 6 et 7)*

- $f_2 = max \sum_{i<j} [P(i,j) + Q(i,j)]$
- $f_3 = max \sum_{i<j} Q(i,j)$

On remarquera par la suite qu'on peut réduire notre système d'équations à une seule équation. En retirant l'équation (1) ci-dessus de la modélisation, on obtiendra les mêmes résultats que pour le système entier, comme peuvent en attester les courbes ci-dessous :

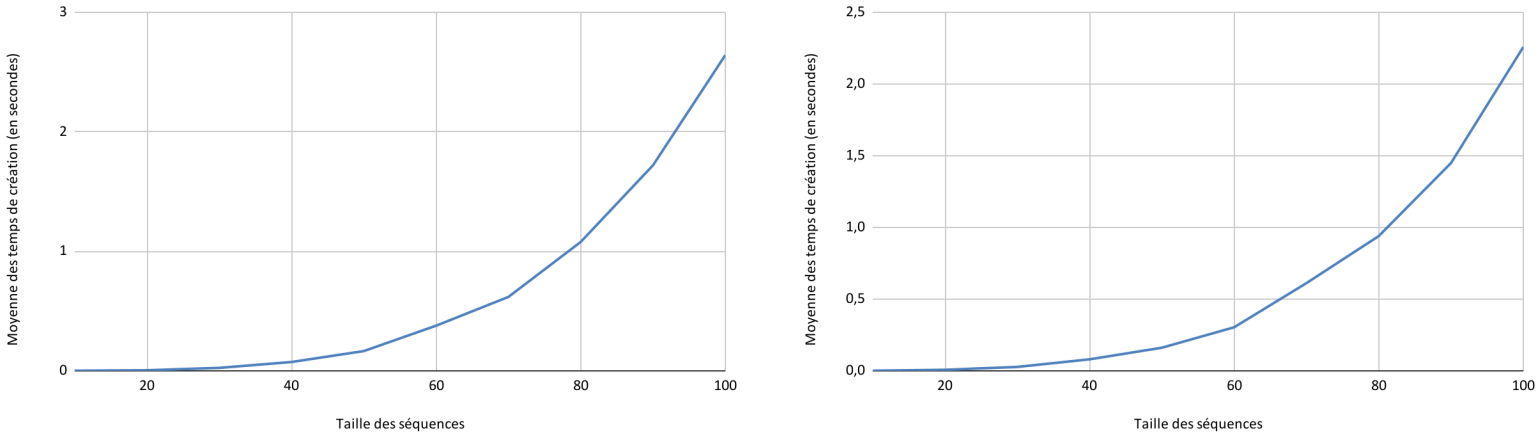


Figure 10 : Représentation du temps de création des fichiers pour la fonction objective f_2 pour la contrainte des piles avec deux équations (à gauche) et une équation (à droite)

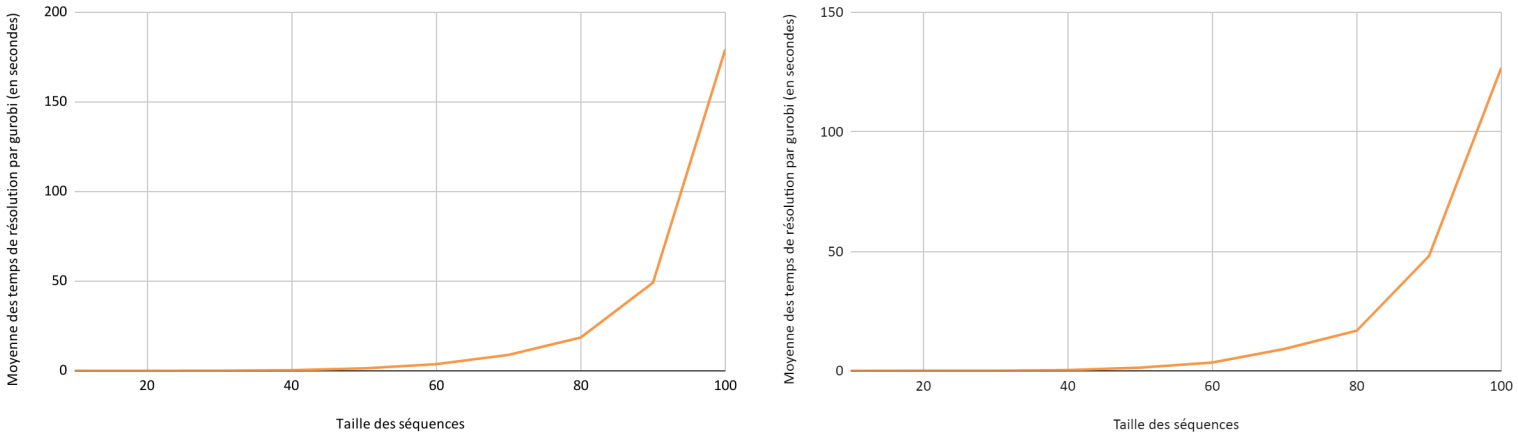


Figure 11 : Représentation du temps de résolution pour la fonction objective f_2 pour la contrainte des piles avec deux équations (à gauche) et une équation (à droite)

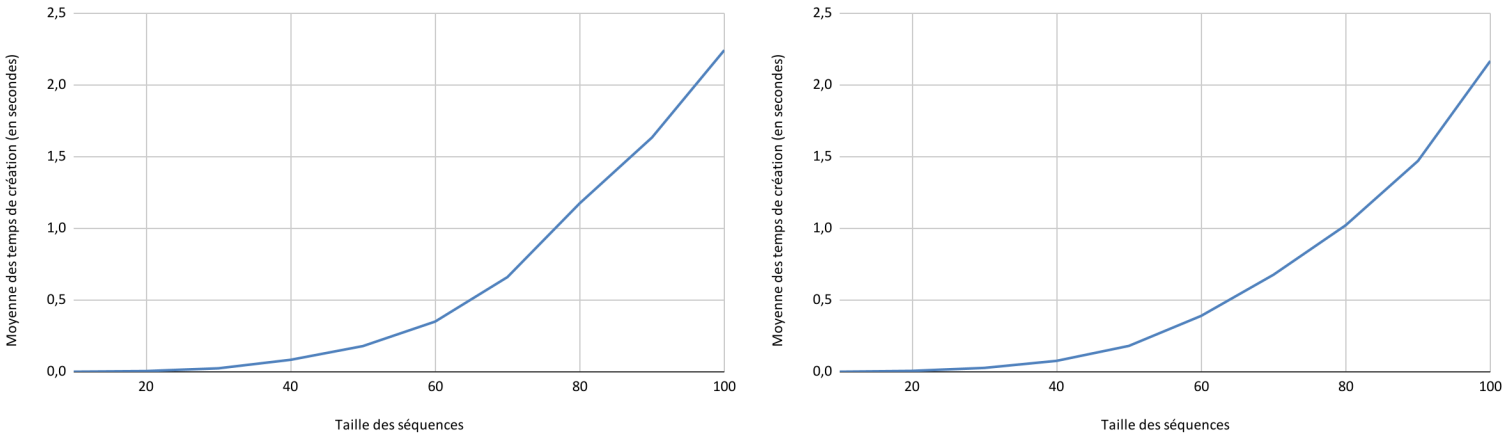


Figure 12 : Représentation du temps de création des fichiers pour la fonction objective f_3 pour la contrainte des piles avec deux équations (à gauche) et une équation (à droite)

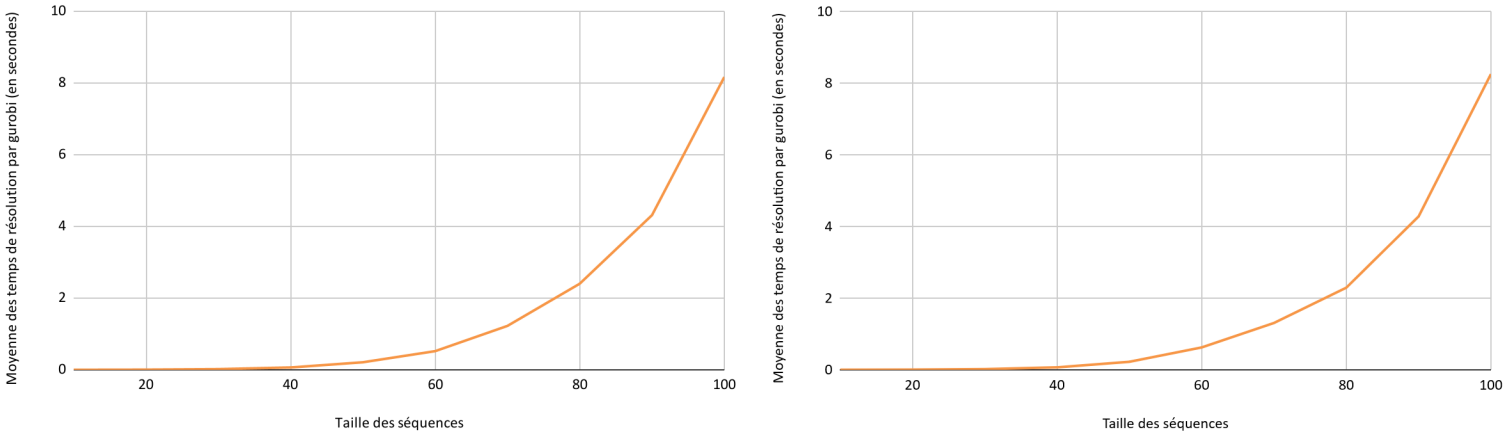


Figure 13 : Représentation du temps de résolution pour la fonction objective f_3 pour la contrainte des piles avec deux équations (à gauche) et une équation (à droite)

Nous avons donc modélisé toutes nos contraintes et avons créé les graphes des temps de génération et de résolution pour chacune d'entre elles. Ceux-ci nous permettent d'avoir une vision sur l'ensemble des contraintes.

4. Analyse et comparaison des résultats

Après avoir modélisé chacune de nos contraintes et les avoir testées pour différents fichiers, nous souhaitons maintenant analyser les graphes que nous avons obtenus.

4.1. Analyse et comparaison des temps de génération et de résolution pour chacune des contraintes

Intéressons-nous d'abord au temps de création des fichiers exécutables par Gurobi. Comme nous pouvons le voir sur les graphiques de la partie précédente, la création de toutes les contraintes est représentée par une fonction quadratique. On observe que pour les plus grandes séquences de taille 100, le temps de création des fichiers varie entre 1.5 et 2.5 secondes. Il n'y a donc pas d'écart notable pour nos différentes contraintes. Cependant, le temps de résolution varie selon la contrainte étudiée.

On remarque que les temps de résolution sont représentés par des fonctions exponentielles. Pour chaque contrainte, pour les séquences de taille inférieure à 50, le temps de résolution est inférieur à 5 secondes. A partir des séquences de taille supérieure à 60, le temps de résolution augmente très fortement. Regardons les temps pour les séquences de taille 100. On observe que le temps de résolution lorsque nous n'avions pas ajouté de contrainte s'élevait à environ 1125 secondes *(Figure 7)*. Pour les contraintes de fréquence d'apparition des nucléotides, de distance minimale et de poids, nous obtenons une nette diminution du temps de résolution par Gurobi, avec une solution trouvée en 200-300 secondes. Enfin, lors de nos tests concernant la contrainte des piles, on remarque que selon la fonction objective utilisée, le temps de résolution n'est pas du tout le même. En effet, on trouve plus de 100 secondes pour la fonction f_2, contre seulement 8 secondes pour f_3. De plus, pour la fonction f_2, on observe une différence de 50 secondes environ entre les fichiers créés avec les deux équations ou simplement la seconde. La résolution est plus rapide pour les fichiers créés avec une seule équation. On ne constate en revanche pas de différence pour les fichiers créés avec la fonction f_3.

On peut maintenant se demander quels seront les résultats lorsque toutes les contraintes seront appliquées en même temps selon la fonction objective choisie. La résolution étant plus rapide pour les fichiers créés avec une seule équation, nous décidons d'ajouter la contrainte des quadruplets avec une seule équation.

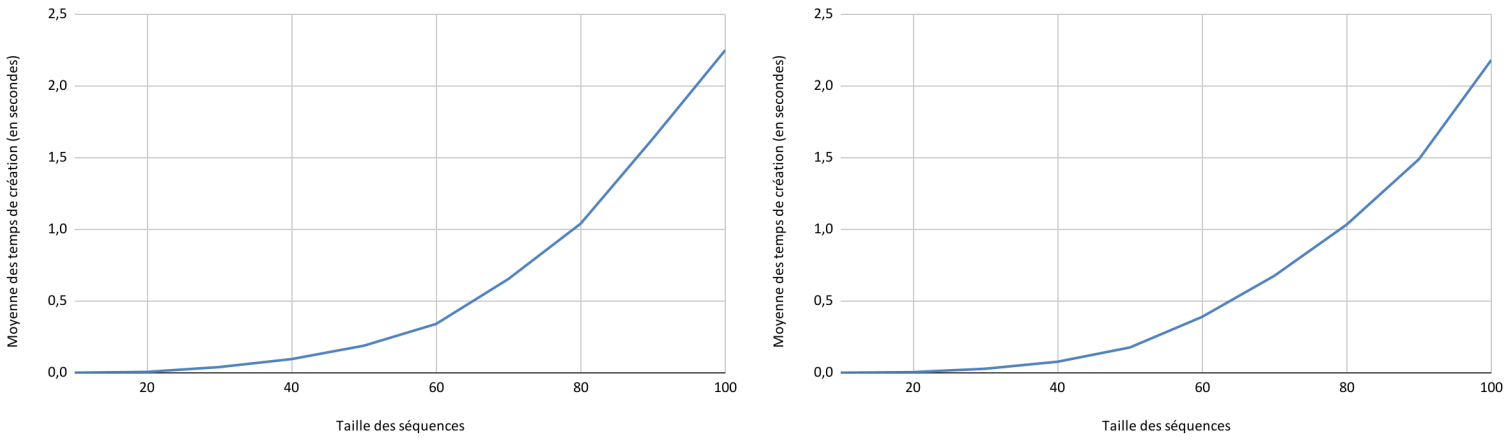


Figure 14 : Représentation du temps de création des fichiers pour les fonctions objectives f_3 (à gauche) et f_2 (à droite) avec toutes les contraintes en fonction des tailles des séquences

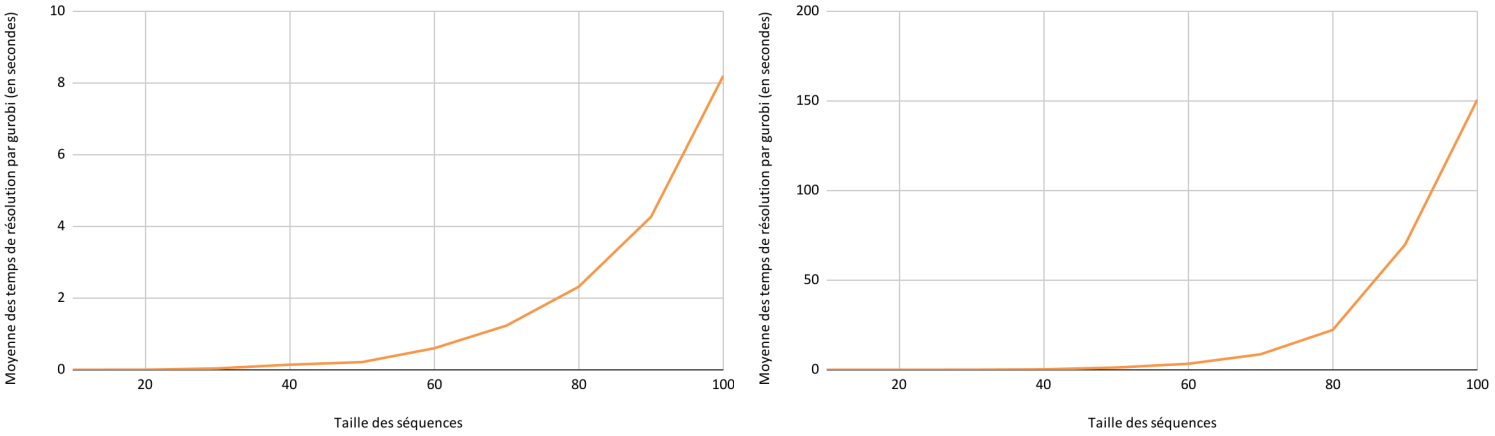


Figure 15 : Représentation du temps de résolution pour les fonctions objectives f_3 (à gauche) et f_2 (à droite) avec toutes les contraintes en fonction des tailles des séquences

Avec cette étude, nous remarquons qu'avec toutes les contraintes, la résolution de Gurobi est plus rapide. On peut alors supposer que plus on ajoute de contraintes, plus le temps de résolution diminue.

Pour prendre un exemple plus concret, nous allons traiter la séquence GCCAUUGAUGACCUGAGCGGGAUGGCCCAAGU, que nous représenterons par des graphiques montrant clairement l'appariement optimal pour chaque contrainte étudiée.

4.2. Comparaison des séquences de nucléotides appariés obtenues pour chaque contrainte

Nous avons représenté la position des nucléotides sur une ligne graduée. Les arcs de cercle représentent les liaisons complémentaires que le solveur Gurobi a trouvées. Les graphiques ont été réalisés via Python *(Annexe 8)*.

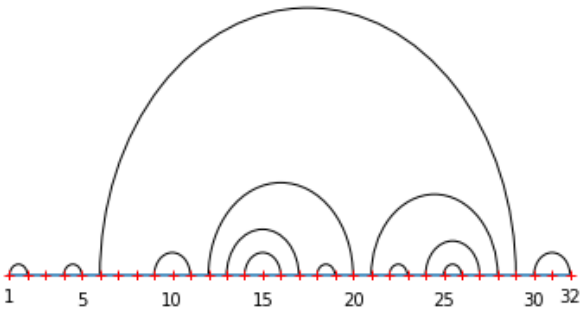


Figure 16 : Représentation graphique de la séquence avec les contraintes de base

Sur la figure ci-dessus, on observe que le nombre de paires a été maximisé. Toutefois, cette modélisation a des limites, puisqu'en réalité la stabilité de l'ARN n'est pas due seulement au nombre de paires. Nous devons ajouter plusieurs contraintes qui viennent perfectionner notre modèle. Par exemple, deux nucléotides consécutifs sont trop proches pour être appariés.

Représentons maintenant les séquences obtenues avec les différentes contraintes, en commençant par les contraintes de distance et de poids.

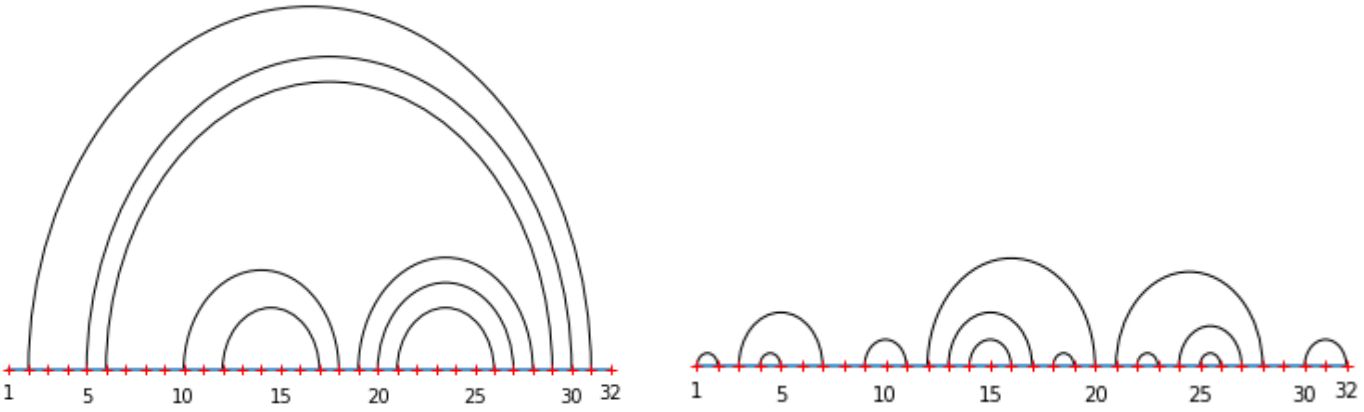


Figure 17 : Représentation graphique de la séquence avec la contrainte de distance (à gauche) et la contrainte de poids (à droite)

Sur cette figure, à droite, nous pouvons voir que les nucléotides 1 et 2 sont appariés, alors qu'en réalité ils ne pourraient pas l'être car ils sont trop proches. On remarque aussi que la contrainte de distance produit une séquence contenant beaucoup moins de paires que celle du poids (8 paires contre 13). En revanche, les paires de la contrainte de distance sont toutes imbriquées. L'imbrication des paires participe à la stabilité de l'ARN.

Afin de vérifier que la contrainte des piles maximise les imbrications, représentons maintenant l'appariement de la séquence avec celle-ci.

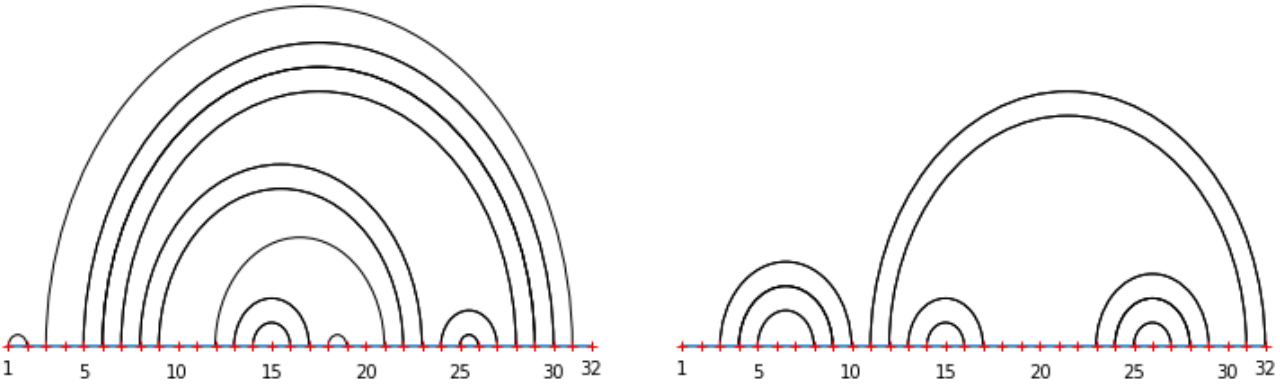


Figure 18 : Représentation graphique de la séquence avec la fonction objective f_2 (à gauche) et la fonction objective f_3 (à droite)

On observe ci-dessus qu'avec la fonction objective f_2 , on obtient beaucoup plus de paires qu'avec simplement la fonction f_3 (13 paires contre 10). On remarque aussi que le nombre maximum d'imbrications produit à gauche est de 9 tandis qu'à droite il est seulement de 5. On peut en déduire que le résultat produit par la fonction objective f_2 semble être beaucoup plus représentatif de la réalité que celui produit par la fonction f_3.

Nous allons maintenant combiner toutes ces contraintes afin d'observer le résultat obtenu et de comparer celui-ci par rapport aux fonctions objectives .

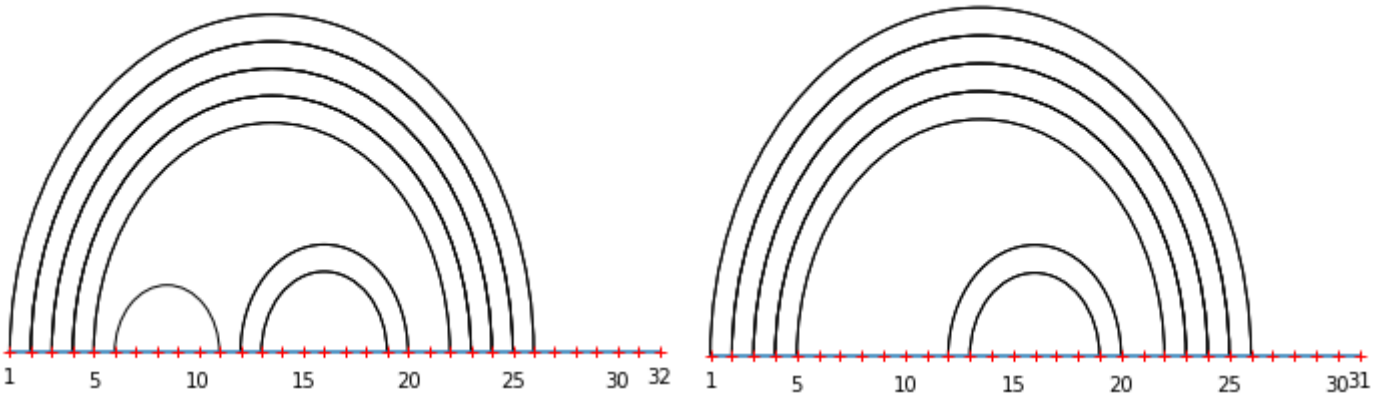


Figure 19 : Représentation graphique de la séquence avec la fonction objective f_2 (à gauche) et la fonction objective f_3 (à droite), toutes contraintes réunies

Tout comme nous l'avions remarqué sur les graphes de la figure 16, on distingue plus de paires pour la fonction objective f_2 que pour f_3 (8 paires contre 7). En revanche, le nombre maximal d'imbrications ne varie pas, il est de 7 dans les deux cas. Nous pouvons aussi observer que nos deux contraintes de poids et de distance sont également respectées.

Ainsi, il nous semble que la forme la plus stable de notre modèle est celle que nous avons calculée avec la fonction objective f_2 et l'ensemble des contraintes réunies. En effet, chaque contrainte étant primordiale à la stabilité de l'ARN, le résultat obtenu en combinant toutes les contraintes nous semble être la solution optimale.

5. Conclusion

La structure de l'ARN est bien plus compliquée que ce que nous pouvons penser. De sa création à sa transformation en protéine, il entre dans une structure stable dont nous ne connaissons pas grand chose. De plus, il est très difficile d'envisager une modélisation exacte, considérant la complexité du problème du repliement de l'ARN et de notre niveau et domaine d'études.

En revanche, nous avons pu avoir quelques informations nous permettant de nous rapprocher de la réalité. Nous avons vu que la structure de l'ARN gagne en stabilité lorsque nous maximisons le nombre de paires complémentaires, qui ont un poids d'appariement maximal et qui respectent une distance minimale entre ses liaisons. Étant partis d'un modèle de base pour comprendre la situation, nous avons ainsi pu programmer nos contraintes avec Python et les ajouter dans un fichier texte compréhensible par le solveur Gurobi.

Finalement, il serait intéressant de comparer nos tests et résultats avec l'avancée biologique du problème, mais il faudrait alors sortir du cadre de ce projet. De plus, l'ARN nous est encore peu familier puisque ce brin d'ADN n'est connu que depuis les années 1960. L'informatique que nous pratiquons nous permet de réaliser des tests (de comportement par exemple), et nous avons réussi à recréer une partie de la réalité. Cependant, savoir modéliser l'ARN dans tous ses états et toutes ses formes serait une avancée majeure en médecine mais aussi dans la compréhension du corps humain.

Finalement, toucher au cœur de notre génétique serait un moyen de combattre les différentes maladies, dues aux mutations et aux transformations de notre corps. Mais ne serait-il pas dangereux de comprendre et de modifier certains systèmes aussi près de notre génome ? Les recherches qui mènent au progrès scientifique du monde peuvent aussi être utilisées à mauvais escient si on ne pose pas de limites éthiques. Il est ainsi primordial d'anticiper les effets et conséquences du progrès, surtout dans un domaine aussi vaste et controversé que la médecine. Pouvoir changer la nature de l'Homme, c'est un sujet auquel il faut être particulièrement attentif.

6. Annexes

```python
import random as rd

def creation_rd_RNA(length,path):
    file = "randomstring"
    OUT = open(path + "/"+file, 'a')
    choice = ['A', 'U', 'C', 'G']

    for i in range(length):
        OUT.write(rd.choice(choice))
    OUT.write("\n")
```

Annexe 1 : Code Python permettant de générer des séquences aléatoires de taille "length"

```python
import openpyxl
workbook = openpyxl.Workbook()

def fichier_excel(length):
    times = []
    sequence = []
    path = 'ARN_taille_' + str(length)
    if not os.path.exists(path):
        os.mkdir(path)
    else:
        shutil.rmtree(path)
        os.mkdir(path)

    for i in range(15):
        randomstring_creation.creation_rd_RNA(length, path)

    IN = open(path + "/randomstring", 'r')
    for RNA in IN.readlines():
        sequence.append(RNA)
        file = RNA + ".lp"
        OUT = open(path + "/" + file, 'w')
        start = time.time()
        ecriture_fichier(RNA, OUT)
        times.append(time.time() - start)

    sheet = workbook.create_sheet("Temps pour ARN de taille" + str(length), 0)

    sheet.cell(1, 1).value = "Sequence de ARN"
    sheet.cell(1, 2).value = "Temps de création"
    sheet.cell(1, 3).value = "Temps de résolution avec gurobi"

    x = 2
    for i in sequence:
        sheet.cell(x, 1).value = i
        x += 1
    x = 2
    for j in times:
        sheet.cell(x, 2).value = j
        x += 1

    workbook.save('Tps.xlsx')
```

Annexe 2 : Code Python permettant de générer le graphique représentant les solution trouvées

```
import random
import time

def randomstring (a, u, c, g, longueur):

    file = "randomstring"
    OUT = open(file, 'a')


      string = ""
    liste_nucleotide =[]
    condition = True
    long = longueur

    while condition :
        if len(liste_nucleotide) < long:

            for i in range (round(a*long)):
                        liste_nucleotide.append("A")
            for i in range (round(u*long)):
                        liste_nucleotide.append("U")
            for i in range (round(c*long)):
                        liste_nucleotide.append("C")
            for i in range (round(g*long)):
                        liste_nucleotide.append("G")

        if len(liste_nucleotide) > long :
            while len(liste_nucleotide) > long:
                liste_nucleotide = liste_nucleotide[:-1]

        if len(liste_nucleotide) == longueur:
            condition = False

    random.shuffle(liste_nucleotide)
    string = ''.join(liste_nucleotide)

    for i in range(len(string)):
        OUT.write(string[i])
    OUT.write("\n")

    return string
```

Annexe 3 : Code Python permettant de générer des séquences aléatoires de taille donnée avec un poids d'apparition pour les nucléotides

```
file = "testdistance.lp"
OUT = open(file, 'w')

# Collect information from the user
print("Will you type in the RNA sequence, or read it from file `randomstring'?")

inchoice = input("Type 't' for typed input, 'f' for file. ")
if (inchoice == 't'):
    RNA = input("Write an RNA sequence (using A,U,C,G), and hit return \n")
else:
    IN = open("randomstring", 'r')
    RNA = IN.readline().strip()

minDist = input("Quelle est la distance minimale entre deux nucléotides appariés ?")
print("You input the sequence: %s" % RNA)
```

```
def ecriture_fichier(minDist,RNA,OUT):
   length = len(RNA)

   #Generate the objective function

   OUT.write("Maximize \n")
   for i in range(1, length):
      for j in range(i + 1, length + 1):
         OUT.write("+ P(%d,%d) \n" % (i, j))
  OUT.write("such that \n")

  # Generate the ILP inequalities to ensure that only complementary nucleotides pair.
  for i in range(1, length):
      for j in range(i + 1, length + 1):
          if (RNA[i - 1] == 'A') and (RNA[j - 1] != 'U'):
            OUT.write("P(%d,%d) = 0 \n" % (i, j))
          if (RNA[i - 1] == 'U') and (RNA[j - 1] != 'A'):
            OUT.write("P(%d,%d) = 0 \n" % (i, j))
          if (RNA[i - 1] == 'C') and (RNA[j - 1] != 'G'):
            OUT.write("P(%d,%d) = 0 \n" % (i, j))
          if (RNA[i - 1] == 'G') and (RNA[j - 1] != 'C'):
              OUT.write("P(%d,%d) =  0 \n" % (i, j))

  # Generate the ILP equalities to ensure that two positions
  # can not be paired if the distance between them is smaller than 4
  for i in range(1, length):
      for j in range(i+1, length+1):
          if((j-i) <= int(minDist)):
              OUT.write("P(%d,%d) = 0 \n" % (i,j))

  # Generate the ILP inequalities to ensure that each position is paired to at most one other
position
  for i in range(1, length + 1):
      inequality = ""
      for j in range(1, i):
          inequality = inequality + " + P(%d,%d)" % (j, i)
      for j in range(i + 1, length + 1):
          inequality = inequality + " + P(%d,%d)" % (i, j)
      inequality = inequality + ' <= 1'
      OUT.write(inequality)
      OUT.write("\n")

  # Generate the ILP inequalities to ensure that no pairs cross.
  for h in range(1, length - 2):
      for i in range(h + 1, length - 1):
          for j in range(i + 1, length):
              for k in range(j + 1, length + 1):
                  OUT.write("P(%d,%d) + P(%d,%d) <= 1 \n" % (h, j, i, k))

  # Write out the list of binary variables
  OUT.write("Binary \n")
  for i in range(1, length):
      for j in range(i + 1, length + 1):
          OUT.write("P(%d,%d) \n" % (i, j))


  OUT.write("end \n")

ecriture_fichier(minDist,RNA,OUT)
print("The ILP file is ", file)
```

Annexe 4 : Code Python permettant de générer les équations-inégalités pour la contrainte : Distance minimale entre un appariement de nucléotides

```
OUT = open ("RNA1.lp", 'w')

# Collect information from the user
print ("Will you type in the RNA sequence, or read it from file `randomstring'?" )

inchoice = input("Type 't' for typed input, 'f' for file. ")
if (inchoice == 't'):
   RNA = input("Write an RNA sequence (using A,U,C,G), and hit return \n")
else:
   IN = open("randomstring", 'r')
   RNA = IN.readline().strip()

a = input("Quel est le poids de l'arc A->U ou U->A ? ")
b = input("Quel est le poids de l'arc C->G ou G->C ? ")

print ("You input the sequence: %s" % RNA)

def ecriture_fichier(a,b,RNA, OUT):
   length = len(RNA)
   OUT.write("Maximize \n")
   for i in range(1, length):
       for j in range(i + 1, length + 1):
           if (RNA[i - 1] == 'A') and (RNA[j - 1] == 'U'):
               x = ("+ " + a + " P(%d,%d) \n" % (i, j))
               OUT.write(x)
           if (RNA[i - 1] == 'U') and (RNA[j - 1] == 'A'):
               x = ("+ " + a + " P(%d,%d) \n" % (i, j))
               OUT.write(x)
           if (RNA[i - 1] == 'C') and (RNA[j - 1] == 'G'):
               x = ("+ " + b + " P(%d,%d) \n" % (i, j))
               OUT.write(x)
           if (RNA[i - 1] == 'G') and (RNA[j - 1] == 'C'):
               x = ("+ " + b + " P(%d,%d) \n" % (i, j))
               OUT.write("+ " + b + " P(%d,%d) \n" % (i, j))
           else:
               OUT.write("+ P(%d,%d) \n" % (i, j))

   OUT.write("such that \n")

   for i in range(1, length):
       for j in range(i + 1, length + 1):

           if (RNA[i - 1] == 'A') and (RNA[j - 1] != 'U'):
               OUT.write("P(%d,%d) = 0 \n" % (i, j))

           if (RNA[i - 1] == 'U') and (RNA[j - 1] != 'A'):
               OUT.write("P(%d,%d) = 0 \n" % (i, j))

           if (RNA[i - 1] == 'C') and (RNA[j - 1] != 'G'):
               OUT.write("P(%d,%d) = 0 \n" % (i, j))

           if (RNA[i - 1] == 'G') and (RNA[j - 1] != 'C'):
               OUT.write("P(%d,%d) =  0 \n" % (i, j))

   for i in range(1, length + 1):
       inequality = ""

       for j in range(1, i):
           inequality = inequality + " + P(%d,%d)" % (j, i)

       for j in range(i + 1, length + 1):
           inequality = inequality + " + P(%d,%d)" % (i, j)
```

```
        inequality = inequality + ' <= 1'
        OUT.write(inequality)
        OUT.write("\n")

    for h in range(1, length - 2):
        for i in range(h + 1, length - 1):
            for j in range(i + 1, length):
                for k in range(j + 1, length + 1):
                    OUT.write("P(%d,%d) + P(%d,%d) <= 1 \n" % (h, j, i, k))

    OUT.write("Binary \n")
    for i in range(1, length):
        for j in range(i + 1, length + 1):
            OUT.write("P(%d,%d) \n" % (i, j))

    OUT.write("end \n")

ecriture_fichier(a,b,RNA,OUT)
print ("The ILP file is RNA1.lp")
```

Annexe 5 : Code Python permettant de générer le code gurobi avec un poid pour chaque combinaison complémentaire

```
OUT = open ("RNA1.lp", 'w')
print ("Will you type in the RNA sequence, or read it from file `randomstring'?" )

inchoice = input("Type 't' for typed input, 'f' for file. ")
if (inchoice == 't'):
    RNA = input("Write an RNA sequence (using A,U,C,G), and hit return \n")
else:
    IN = open("randomstring", 'r')
    RNA = IN.readline().strip()

print ("You input the sequence: %s" % RNA)

def ecriture_fichier(RNA, OUT):
    length = len(RNA)
    # Generate the objective function : SOMME DES P+Q

    OUT.write("Maximize \n")
    for i in range(1, length):
        for j in range(i + 1, length + 1):
            OUT.write("+ P(%d,%d) \n" % (i, j))
    for k in range(1, length - 2):
        for l in range(k + 3, length + 1):
            OUT.write("+ Q(%d,%d) \n" % (k, l))
    OUT.write("such that \n")

    for i in range(1, length):
        for j in range(i + 1, length + 1):

            if (RNA[i - 1] == 'A') and (RNA[j - 1] != 'U'):
                OUT.write("P(%d,%d) = 0 \n" % (i, j))

            if (RNA[i - 1] == 'U') and (RNA[j - 1] != 'A'):
                OUT.write("P(%d,%d) = 0 \n" % (i, j))

            if (RNA[i - 1] == 'C') and (RNA[j - 1] != 'G'):
                OUT.write("P(%d,%d) = 0 \n" % (i, j))
```

```
            if (RNA[i - 1] == 'G') and (RNA[j - 1] != 'C'):
                OUT.write("P(%d,%d) =  0 \n" % (i, j))

    for i in range(1, length + 1):
        inequality = ""
        for j in range(1, i):
            inequality = inequality + " + P(%d,%d)" % (j, i)
        for j in range(i + 1, length + 1):
            inequality = inequality + " + P(%d,%d)" % (i, j)

        inequality = inequality + ' <= 1'
        OUT.write(inequality)
        OUT.write("\n")

    ## Représentation des piles
    for i in range(1, length - 2):
        for j in range(i + 3, length + 1):
            OUT.write("P(%d,%d) + P(%d,%d) - Q(%d,%d) <= 1 \n" % (i, j, i + 1, j - 1, i, j))
            OUT.write("2 Q(%d,%d) - P(%d,%d) - P(%d,%d) <= 0 \n" % (i, j, i, j, i + 1, j - 1))

    for h in range(1, length - 2):
        for i in range(h + 1, length - 1):
            for j in range(i + 1, length):
                for k in range(j + 1, length + 1):
                    OUT.write("P(%d,%d) + P(%d,%d) <= 1 \n" % (h, j, i, k))

  OUT.write("Binary \n")
    for i in range(1, length):
        for j in range(i + 1, length + 1):
            OUT.write("P(%d,%d) \n" % (i, j))

    for k in range(1, length - 2):
        for l in range(k + 3, length + 1):
            OUT.write("Q(%d,%d) \n" % (k, l))

    OUT.write("end \n")

ecriture_fichier(RNA,OUT)
print ("The ILP file is RNA1.lp")
```

Annexe 6 : Code Python permettant de générer le code gurobi pour avoir le plus d'arcs possibles pour la fonction objective : $f_2 = max \sum_{i<j} [P(i,j) + Q(i,j)]$

```
# Generate the objective function : SOMME DES Q

OUT.write ("Maximize \n")

for i in range(1, length-2):
  for j in range(i+3, length+1):
          OUT.write ("+ Q(%d,%d) \n" % (i,j))
OUT.write ("such that \n")
```

Annexe 7 : Code Python (fonction objective uniquement) permettant de générer le code gurobi pour avoir le plus d'arcs possibles pour la fonction objective : $f_3 = max \sum_{i<j} Q(i,j)$

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# On récupère les valeurs contenues dans le fichier RNA1.sol
def lectureFichier(f):
    fichier = open(f)
    objective_value = fichier.readline()
    P=[]
    Q=[]
    for line in fichier.readlines():
        sep1=line.find("(")
        sep2=line.find(",")
        sep3=line.find(")")
        sep4=line.find(" ")
        indice_i=int(line[sep1+1:sep2])
        indice_j=int(line[sep2+1:sep3])
        nombre=int(line[sep4+1])
        if line[0]=="P" :
            P.append([[indice_i, indice_j], nombre])
        else :
            Q.append([[indice_i, indice_j], nombre])
    return objective_value, P, Q

objective_value, P, Q = lectureFichier("RNA1.sol")
indice_max = P[-1][0][-1]
axes = plt.gca()

y=[0 for i in range (1,indice_max+1)]
x=[i for i in range(1,indice_max+1)]

axes.set_ylim(-0.1,indice_max/2)
axes.set_xlim(0.5,indice_max+0.5)
axes.get_yaxis().set_visible(False)
axes.set_frame_on(False)
axes.set_xticks([1,indice_max], minor = True)
axes.xaxis.set_ticklabels(['1', indice_max], minor = True)
axes.xaxis.set_ticks_position('none')
plt.plot(x,y)
for i in range(1,indice_max+1):
    plt.plot(i,0, marker='+', color ='red')

# Parcours des listes P et Q, si la valeur est associée à 1, alors on dessine un arc de cercle entre
# les nucléotides i et j
for liste in Q:
    if liste[1]==1:
        i=liste[0][0]
        j=liste[0][1]
        i2=i+1
        j2=j-1
        k = (i+j)/2
        k2= (i2+j2)/2

        axes.add_artist(patches.Arc((k, 0), j-i, (j-i), 180, 180, 0))
        axes.add_artist(patches.Arc((k2, 0), j2-i2, (j2-i2), 180, 180, 0))
for liste in P:
    if liste[1]==1:
        i=liste[0][0]
        j=liste[0][1]
        k = (i+j)/2
        axes.add_artist(patches.Arc((k, 0), j-i, (j-i), 180, 180, 0))
```

Annexe 8 : Code Python permettant de générer le graphique représentant les solutions trouvées

7. Bibliographie

- Qu'est ce que l'ADN ?
https://fr.wikipedia.org/wiki/Acide_d%C3%A9soxyribonucl%C3%A9ique

- Qu'est ce que l'ARN ?
https://fr.wikipedia.org/wiki/Acide_ribonucl%C3%A9ique

- Étapes de la synthèse des protéines :
https://youtu.be/J9k0r3zDUUM
https://youtu.be/sS5eVGzHbv0
https://youtu.be/FUmq3XiZQlY
https://youtu.be/TFZG2w9e8vY
https://youtu.be/qFD04mSKNPA

- Structure secondaire de l'ARN :
https://fr.wikipedia.org/wiki/Structure_de_l%27ARN#Structure_secondaire

- Repliement de l'ARN :
https://www.lix.polytechnique.fr/~ponty/talks/2009-11-LIPN-AnalyseAlgorithmeARN-Ponty.pdf

- Fonctionnement et principe de la programmation linéaire en entiers :
https://fr.wikipedia.org/wiki/Optimisation_lin%C3%A9aire_en_nombres_entiers