

📍 Avenue V. Maistriau 8a
B-7000 Mons

☎ +32 (0) 65 33 81 54

📧 scitech-mons@heh.be

WWW.HEH.BE

Projet d'optimisation combinatoire

Nonogram

Recuit Simulé

Rapport présenté dans le cadre du diplôme de :

Master en sciences de l'ingénieur industriel
orientation Informatique option IA et Big Data

Auteur·e·s :

**Theo Pluvinage
Guillaume Tricknot
Axel Vanhaverbeke
Xavier Tomisinec
Jonathan Szczepanski**

Table des matières

1	Introduction	1
2	Contexte	2
2.1	Le jeu Nonogram	2
2.2	Choix du langage	2
3	Recuit simulé	3
3.1	Adaptation au Nonogram	4
3.2	Recherche de voisinage	5
3.2.1	Sélection aléatoire d'un groupe	5
3.2.2	Déplacement à gauche ou à droite	5
3.2.3	Validation de la configuration	5
3.3	Implémentation	6
3.3.1	Initialisation des contraintes de la matrice	6
3.3.2	Prétraitement	6
3.3.3	Calcul du score initial	7
3.3.4	Recuit simulé	7
3.4	Needleman-Wunsch	10
3.4.1	Fonctionnement de l'algorithme	10
3.4.2	Objectifs	11
3.4.3	Modification de l'algorithme	11
3.4.4	Détails de l'implémentation	11
3.4.5	Exemple de résultat	12
3.4.6	Analyse	12
3.4.7	Conclusion sur l'utilisation de Needleman-Wunsch	12
3.5	Parallélisation	13
4	Résultats	14
4.1	Paramètres utilisés pour les tests	14
4.2	Résultats nonogram 10x10	15
4.3	Résultats nonogram 15x15	16
4.4	Résultats nonogram 30x30	17
5	Conclusion	18
5.1	Discussion	18
5.2	Perspectives	18
	Bibliographie	20
	Annexe	21
A	Code source	21
A.1	Frontend	38
A.2	Backend	38
B	Timesheets	38

1 Introduction

Afin d'introduire ce rapport nous allons voir les différents points abordé au long de ce dernier :

Tous d'abord nous allons évoquer le contexte dans lequel ce projet a été réalisé, ainsi qu'une explication sur qu'est ce que le nonogram, son histoire et la répartition des tâches au long du projet. Nous évoquerons aussi pourquoi nous avons utilisé le C comme langage de programmation pour ce projet.

Ensuite nous expliqueront plus en détail le fonctionnement du recuit simulé et quels sont les stratégies que nous avons développées pour résoudre le nonogram et l'implémenter.

Nous analyserons également les résultats obtenus lors des différents testes sur les différents nonogram donnés.

Pour finir avec une conclusion et une discussion au sujet des divers problèmes rencontrés et perspectives d'améliorations.

2 Contexte

Dans le cadre du cours d'optimisation combinatoire, il nous a été attribué par groupes différentes métaheuristiques que nous avons dû développer afin de résoudre le jeu du nonogram le plus vite et "efficacement" possible. Nous avons donc dû apprendre le fonctionnement de notre algorithme qui, dans notre cas, est le recuit simulé et le développer avec le langage de programmation de notre choix.

2.1 Le jeu Nonogram

Qu'est-ce que le nonogram ? Il s'agit d'un puzzle japonais né à la fin des années 80 qui a pour principe de remplir une grille avec des cases noires ou blanches à partir de consignes numériques appliquées pour chaque lignes et colonnes. Créé par Non Ishida, graphiste japonais qui eut l'idée du design en participant à un concours visant à créer une nouvelle campagne publicitaire pour le métro de Tokyo. Cependant, au même moment, un monsieur passionné de casses-têtes et de logique nommé Tetsuya Nishio proposa une idée similaire. Ces deux idées seront développées en parallèle permettant la popularisation rapide du concept au Japon sous le nom de "Picross", il arrivera ensuite en Europe sous le nom de "nonogram".

Comme évoqué brièvement plus tôt, le jeu est constitué d'une grille $N \times N$ dans laquelle chaque ligne et chaque colonne aura une suite de nombre associée. Ces nombres représenteront des blocs de cases noires qui devront à minima être séparés par au moins une case blanche. La logique de ce jeu demandera de recouper l'information de chaque ligne et chaque colonne afin de déterminer pas à pas le dessin qui se cache dans cette grille et ce par déduction en découvrant au fil de la résolution le dessin apparaître. Les nonogram peuvent être de taille diverse et dispose de possibilité infinie de dessins. [1]

2.2 Choix du langage

Pour ce projet, nous avons choisi d'utiliser le C++ et ce pour diverses raisons :

Tout d'abord pour une raison de performance, en effet, le C++ est bien plus performant que des langages interprétés tel que le python notamment d'un point de vue de vitesses d'exécutions et de contrôle de la mémoire. Ces points étant très important lors de l'implémentation d'algorithmes de recherches et ou d'optimisation comme le recuit simulé, d'autant plus que nous n'avons qu'une minute pour résoudre le nonogram qui nous sera donné lors de l'examen et que nous ne connaissons pas la dimension de ce dernier et par conséquent le nombres d'itérations nécessaires à sa résolution.

De plus le C++ est un langage de bas niveau permettant donc une bonne gestion des ressources, qui est un facteur très important quand l'on souhaite optimiser l'exécution de notre algorithme.

Enfin le fait que le C/C++ soit le langage que nous avons le plus utilisé au cours de nos études à également été un choix décisif concernant son utilisation.

3 Recuit simulé

Inspiré du processus physique de recuit en métallurgie, le recuit simulé simule le refroidissement progressif d'un matériau chauffé pour atteindre une configuration stable et optimale.

Le principe de base repose sur l'idée de rechercher une solution optimale dans un espace de solutions en acceptant temporairement des solutions moins bonnes pour éviter de rester bloqué dans des minima locaux. Ce principe est contrôlé dans l'application principale de cet algorithme par un paramètre clé : la température, qui diminue progressivement au fil des itérations.

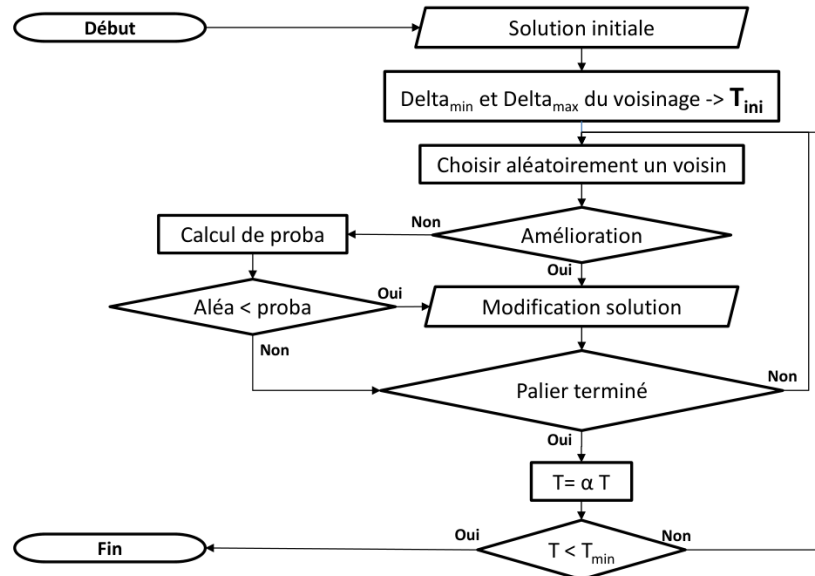


Figure 3.1 – Schéma général recuit simulé

L'algorithme commence par une initialisation où une solution initiale est choisie. Ensuite, les valeurs minimales et maximales des variations de la fonction objectif (Δ_{\min} et Δ_{\max}) dans le voisinage sont utilisées pour calculer une température initiale (T_{ini}). Cette température contrôle la probabilité d'accepter des solutions moins bonnes.

À chaque itération, un voisin de la solution actuelle est choisi aléatoirement. Si ce voisin représente une amélioration (c'est-à-dire que la fonction objectif diminue), la solution est automatiquement mise à jour. Dans le cas contraire, une probabilité d'acceptation est calculée en fonction de la différence entre les coûts (ΔE) et de la température actuelle (T). Si un tirage aléatoire est inférieur à cette probabilité, la solution est également mise à jour, même si elle est moins bonne.

Une fois qu'un certain nombre d'itérations est terminé à température constante, la température est réduite selon un facteur de refroidissement ($T = \alpha T$, avec $\alpha < 1$). Ce processus se répète jusqu'à ce que la température atteigne un seuil minimal (T_{\min}), moment auquel l'algorithme s'arrête.

L'algorithme alterne entre accepter des solutions moins bonnes et améliorer directement des solutions tout en diminuant progressivement sa capacité à accepter des solutions sous-optimales grâce au refroidissement progressif. [2] [3]

3.1 Adaptation au Nonogram

Pour rappel, notre problème consiste à remplir une grille de dimensions spécifiques en respectant des contraintes. Toutes ces données sont à notre disposition dans un fichier (.pti).

Voici un schéma général des différentes étapes de notre approche afin de résoudre le problème :

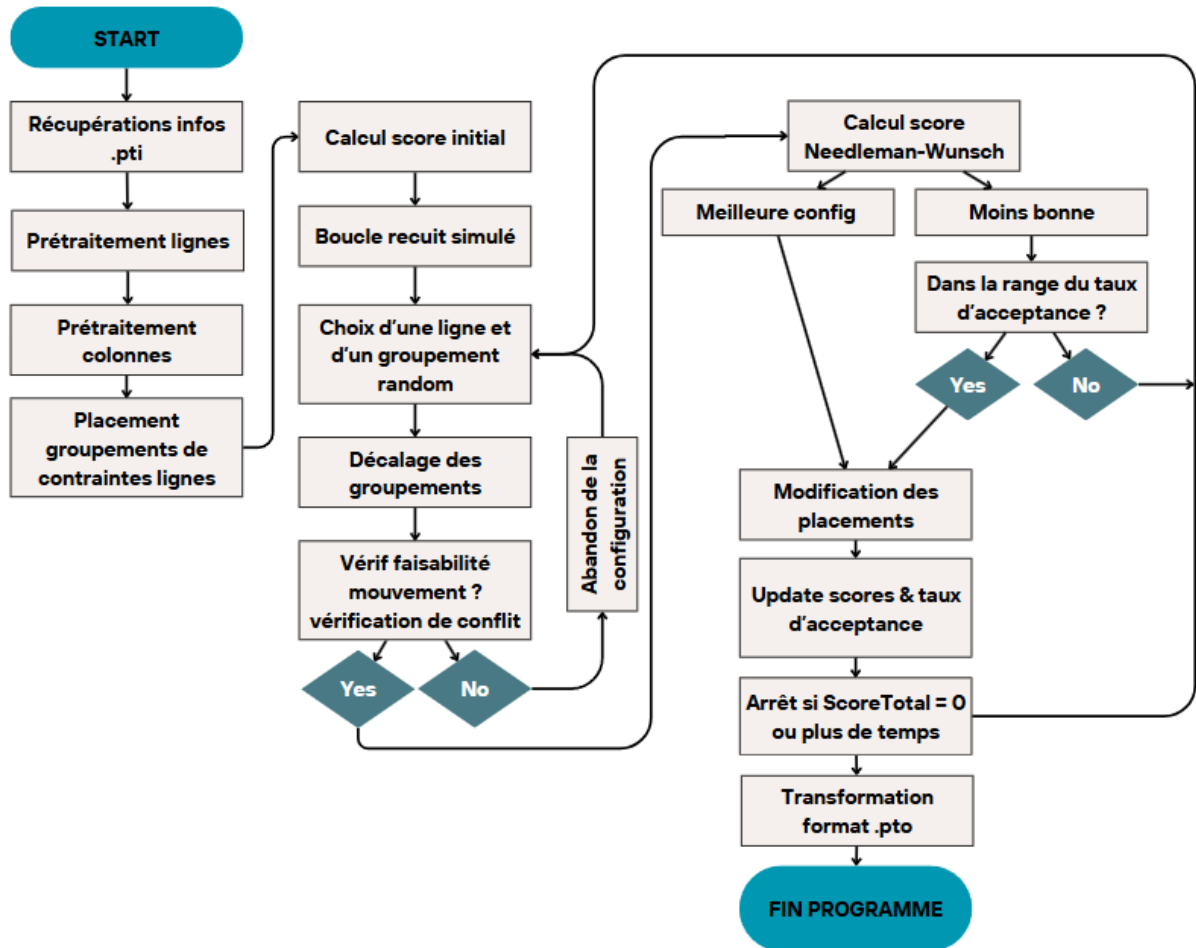


Figure 3.2 – Schéma des différentes étapes du projet

3.2 Recherche de voisinage

La recherche de voisinage dans notre application consiste à déplacer un ensemble de cases noires potentielles d'une case vers la gauche ou la droite sur une ligne donnée.

3.2.1 Sélection aléatoire d'un groupe

On choisit d'abord une ligne et un groupe de cases noires potentielles. Ces groupes sont définis par leurs bornes de début et de fin, ce qui permet de repérer facilement leur position dans la ligne.

3.2.2 Déplacement à gauche ou à droite

Pour générer un voisin, on tente de décaler ce groupe d'une seule case vers la gauche ou la droite (via les fonctions `movingLeft()` et `movingRight()`). Si l'espace est insuffisant parce que nous approchons d'un autre groupement, un effet "auto-tamponneuse" est fait, la fonction de mouvement va être répercutée sur le groupement approché. Cependant, à l'approche d'une bordure de grille, il cessera sa tentative de mouvement.

3.2.3 Validation de la configuration

Lorsqu'un groupe est déplacé, on recopie cette nouvelle position dans une version temporaire de la grille, ensuite on vérifie qu'elle est compatible avec les cases noires/blanches dont on a la certitude de la position. Si la configuration est invalide, on rejette directement et on retente un autre déplacement.

Une fois validé, le groupe décalé donne une nouvelle solution voisine de la précédente, qui servira de point de départ pour la prochaine itération du recuit simulé.

Cette démarche permet donc d'explorer progressivement tout l'espace de solutions en jouant avec la position des groupes sur chaque ligne du nonogram.

3.3 Implémentation

Cette partie va décrire en détail notre implémentation.

3.3.1 Initialisation des contraintes de la matrice

La première étape de notre approche consiste à définir les contraintes des lignes et des colonnes. Ces contraintes sont spécifiées aux fichiers `.pti` fournis en entrée. Ensuite, une matrice est créée et initialisée avec la valeur indéterminée (0), indiquant que les cellules n'ont pas encore été déterminées.

3.3.2 Prétraitement

Un prétraitement est effectué pour initialiser certaines cellules de la matrice en fonction des contraintes des lignes et des colonnes. Ce prétraitement permet de réduire l'espace de recherche en plaçant des cellules noires là où elles sont obligatoires et logiques, ce qui permet de simplifier le problème pour les étapes suivantes. Il est appliqué aux lignes et aux colonnes.

Pour se faire, on calcule la taille totale des groupements et des espaces blancs entre les groupements. Si la somme des tailles des groupements et des espaces blancs est supérieure à la moitié de la taille de la ligne ($\text{RowSize}/2$), cela signifie que certaines cellules doivent être définitivement noires. Pour chaque groupement, on place les cellules noires là où elles sont obligatoires, en tenant compte des espaces blancs nécessaires. Le même traitement est appliqué aux colonnes.

Exemple de prétraitement ligne de taille 10 et contrainte de 7 :

[0	0	0	0	0	0	0	0	0	0]
[1	1	1	1	1	1	1	0	0	0]
[0	1	1	1	1	1	1	1	0	0]
[0	0	1	1	1	1	1	1	1	0]
[0	0	0	1	1	1	1	1	1	1]

Valeurs dont on est sûr :

[0	0	0	1	1	1	1	0	0	0]
---	---	---	---	---	---	---	---	---	---	----

Une fois ceci fait, afin de compléter efficacement nos lignes, nous comparerons nos contraintes actuelles et "verrons" les groupements manquants; ceux-ci seront placés sur la ligne au premier emplacement possible afin de s'assurer que l'entière des contraintes lignes soit respectée.

3.3.3 Calcul du score initial

Le score est calculé en alignant les contraintes avec les séquences détectées et en utilisant le carré de la contrainte si cette dernière n'est pas respectée. Les scores sont mis à jour après chaque itération pour refléter les modifications apportées par l'algorithme.

3.3.4 Recuit simulé

Notre boucle de recuit simulé itère sur des propositions de modifications de la grille, évalue ces modifications et décide de les accepter ou de les rejeter en fonction d'un critère d'acceptation.

3.3.4.1 Initialisation et conditions d'arrêt

On démarre avec une matrice de travail `tableau2DReplika`, copie de la matrice initiale `tableau2D`.

Des variables de contrôle, comme `ScoreTotal`, `iteration` et `TrueIteration`, permettent de mesurer la qualité de la solution et d'enregistrer le nombre d'itérations déjà effectuées.

La boucle principale tourne tant que `RecuitRunStep` est à `true`, et s'arrête si `ScoreTotal` tombe à 0 (solution parfaite) ou si on dépasse un nombre d'itérations spécifié.

3.3.4.2 Génération d'un mouvement aléatoire

À chaque itération, on efface la liste `MovedPlace`, qui stocke les indices de colonnes potentiellement affectés.

On choisit aléatoirement :

- Une ligne (`seedRow`) via `std::uniform_int_distribution<> distribRow(0, ColSize - 1)`.
- Un indice de groupe sur cette ligne (`seedGrp`), toujours de manière aléatoire.
- Une direction de déplacement (`seedDir`), qui détermine si on tente de décaler ce groupe vers la gauche ou vers la droite. Du fait du placement des groupements sur la gauche par la pré-traitement, nous avons décidé de donner un poids plus grand aux seeds favorisant légèrement un déplacement vers la droite.

3.3.4.3 Application du mouvement

En fonction de la direction, on appelle soit `movingRight()`, soit `movingLeft()`.

Ces deux fonctions essaient de décaler un groupement « noir possible » (BLACKMAYBE) au sein de la ligne en question, en respectant des contraintes de proximité entre groupes.

Si le mouvement est autorisé :

- On recopie cette nouvelle configuration dans la matrice `tableau2DReplica`.
- On stocke dans `MovedPlace` les indices des colonnes modifiées.

3.3.4.4 Vérification de la faisabilité du changement

Pour chaque place modifiée, on vérifie la condition de faisabilité via la fonction `fitModif()`.

Si cette vérification échoue, on abandonne immédiatement ce mouvement et l'algorithme recommence directement une nouvelle itération.

Dans ce cas, la configuration n'est pas conservée car elle entre en conflit avec les cellules noires et blanches dont on est certain.

3.3.4.5 Calcul du score et critère d'acceptation

- Si le mouvement est faisable :
 1. On appelle l'algorithme Needleman-Wunsch (`needlemanWunsch()`) pour comparer la nouvelle répartition des groupes à la contrainte voulue. Le résultat est un coût qui reflète l'écart entre la solution actuelle et les contraintes.
 2. On recalcule la différence de score pour les colonnes concernées.
 3. On obtient alors :
 - (a) L'ancien score des colonnes (stocké dans `ScoresCols[x]`).
 - (b) Le nouveau score (calculé dans `ScoresModified[x]`).
 4. On compare ces scores :
 - (a) Si le nouveau score réduit le score total (`ScoreTotal`), on accepte directement cette configuration.
 - (b) Sinon, on utilise un critère probabiliste basé sur la variable d'acceptation des erreurs (`ERRORACCEPTANCEPROBABILITY`) pour éventuellement accepter un mouvement moins bon. Cela correspond au principe classique du recuit simulé, qui maintient une part d'exploration aléatoire. `ERRORACCEPTANCEPROBABILITY` se réfère à la valeur obtenue par cette formule pour éventuellement accepter un mouvement moins bon :

$$P = \exp\left(-\frac{\Delta E}{T}\right)$$

- Si on accepte la modification :
 1. On met à jour la matrice principale (`tableau2D`) avec les valeurs de la matrice temporaire (`tableau2DReplica`).
 2. On adapte le score total (`ScoreTotal`), on incrémente le compteur global des itérations réussies (`TrueIteration`), et on réduit progressivement la probabilité d’accepter des solutions moins bonnes en multipliant l’acceptation des erreurs par un facteur réducteur (`ERRORACCEPTANCEREDUCTOR`).

Sinon :

1. On restaure l’état initial de la matrice temporaire (`tableau2DReplica`).
2. On passe à l’itération suivante sans conserver ce mouvement.

Grâce à cette logique, le programme peut sortir de minima locaux, tout en affinant progressivement la grille de cases noires et blanches jusqu’à trouver (idéalement) une solution satisfaisant toutes les contraintes.

3.4 Needleman-Wunsch

Cette partie traite plus en détail du fonctionnement et l'utilisation de l'algorithme de Needleman-Wunsch dans notre solution.

On appelle l'algorithme Needleman-Wunsch (**needlemanWunsch()**) pour comparer la nouvelle répartition des groupes à la contrainte voulue. Le résultat est un coût qui reflète mieux l'écart entre la solution actuelle et les contraintes.

3.4.1 Fonctionnement de l'algorithme

L'algorithme de Needleman-Wunsch fonctionne en deux étapes principales :

1. Création de la matrice d'alignement

Une matrice $(m+1) \times (n+1)$ est construite, où m et n sont les longueurs des deux séquences à aligner. Chaque case de la matrice représente un sous-problème d'alignement.

- Les premières lignes et colonnes sont initialisées avec des pénalités pour les gaps.
- Ensuite, chaque cellule est remplie en prenant le maximum entre :

Le score pour un match/mismatch (diagonale),

Le score pour introduire un gap dans l'une des séquences (ligne ou colonne).

2. Backtracking pour trouver l'alignement optimal

Une fois la matrice remplie, le chemin qui maximise le score global est déterminé par un processus de backtracking. Partant de la dernière cellule (en bas à droite), on remonte vers la première (en haut à gauche) en suivant les cellules qui ont contribué au score optimal (diagonale, haut ou gauche). Cela permet de reconstruire les deux séquences alignées, avec des gaps si nécessaire.

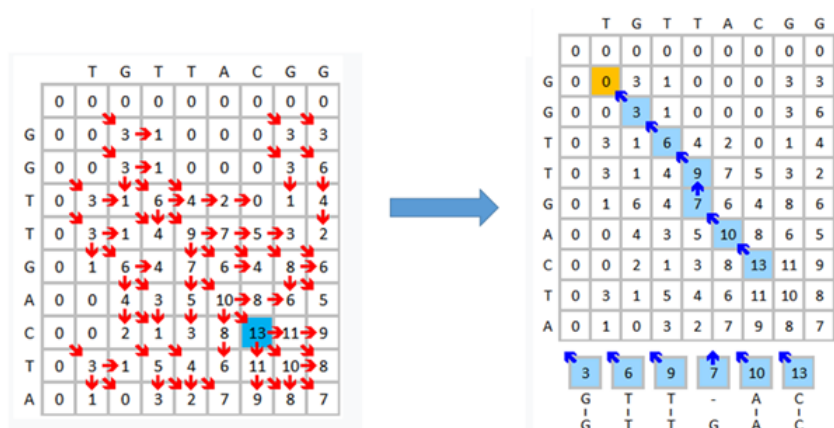


Figure 3.3 – Création et Backtracking

3.4.2 Objectifs

Le but du code modifié est d'adapter l'algorithme classique pour :

1. **Favoriser les grandes valeurs** : Assurer que, dans la mesure du possible, les chiffres de grande amplitude soient alignés, car leur correspondance est considérée comme plus significative.
2. **Maintenir un alignement global optimal** : Garantir que la modification reste compatible avec l'approche classique de Needleman-Wunsch pour produire un alignement optimal globalement.

3.4.3 Modification de l'algorithme

Dans l'algorithme original de Needleman-Wunsch, le score de match est une constante (souvent une valeur fixe positive) lorsqu'une correspondance est détectée. Dans notre modification, ce score a été redéfini comme le carré de la valeur alignée, augmentant ainsi significativement le poids des grandes valeurs dans le calcul du score total.

3.4.4 Détails de l'implémentation

Les pénalités pour les mismatches (-1) et les gaps (-1) ont été conservées.

La principale modification se situe au niveau du calcul du score de match :

```
1 int match = dpMatrix[[]i - 1[]][[]j - 1[]] + (sequence1[[]i - 1[]] ==
    sequence2[[]j - 1[]] ? (sequence1[[]i - 1[]] - * (sequence1[[]i - 1[]] -
        : mismatchPenalty);
```

Listing 3.1 – Modification calcul score de match

Ainsi :

Si deux chiffres identiques sont alignés (ex. : 5 avec 5), le score est égal à $5^2 = 25$.

En cas de mismatch, une pénalité fixe de -1 est appliquée.

Les gaps sont également pénalisés par une valeur fixe de -1.

3.4.5 Exemple de résultat

Pour tester la modification, nous avons aligné les séquences suivantes :

Séquence 1 : 8256

Séquence 2 : 2568

La **matrice dynamique** générée est la suivante :

0	-1	-2	-3	-4
-1	-1	-2	-3	61
-2	3	2	1	60
-3	2	28	27	59
-4	1	27	64	63

Table 3.1 – Matrice Dynamique

L'**alignement optimal** obtenu est :

8256-
-2568

Table 3.2 – Alignement optimal

3.4.6 Analyse

Cet alignement montre que l'algorithme a favorisé l'alignement des grandes valeurs communes :

Les chires 2, 5 et 6 sont alignés en raison de leurs poids élevés ($2^2=4$, $5^2=25$ et $6^2=36$).

La valeur 8 de la première séquence n'est pas alignée, car introduire un gap à cet endroit maximise le score global.

3.4.7 Conclusion sur l'utilisation de Needleman-Wunsch

Cette modification de l'algorithme de Needleman-Wunsch permet de privilégier l'alignement des grandes valeurs, tout en maintenant un alignement global optimal. Elle permet également la possibilité de calculer un score plus proche de la réalité, tout en permettant une plus grande flexibilité dans les groupements, favorisant ainsi des mouvements plus libres et adaptatifs lors de l'alignement des séquences. [4]

3.5 Parallélisation

Dans notre implémentation, nous lançons plusieurs threads pour exécuter en parallèle la fonction de recuit simulé, chacun manipulant sa propre copie de la grille et maintenant un score individuel. Une fois tous les threads terminés, nous comparons les scores finaux pour ne conserver que la meilleure solution. Cette approche permet de converger plus rapidement vers une solution optimale en exploitant les différents cœurs disponibles, au prix de dupliquer la matrice initiale afin que chaque thread progresse indépendamment sans se gêner mutuellement.

4 Résultats

Dans cette section, nous présentons les résultats obtenus en appliquant notre algorithme de recuit simulé à la résolution de différents nonogram. Nous pourrions alors évaluer la performance de notre solution en termes de convergence vers une solution valide en se référant aux fichiers de solution fournis ainsi qu'au calcul de score global.

Les résultats présentés sont accompagnés de configurations intermédiaires.

4.1 Paramètres utilisés pour les tests

Les paramètres utilisés ont été sélectionnés car ils offraient un bon compromis entre vitesse de convergence et qualité des solutions.

Concrètement :

- Si l'on augmente les pénalités mismatch ou gap (par exemple à -2), le score sera plus strict. Cela peut accélérer la convergence mais accroît le risque de rester bloqué dans un minimum local.
- Réduire la probabilité d'acceptation initiale ou baisser le ratio de température initial rend l'algorithme plus conservateur dès le départ. Cela limite l'exploration des solutions moins bonnes.
- Au contraire, augmenter ce ratio et la probabilité d'acceptation favorisera l'exploration. Cependant, cela peut rallonger le temps nécessaire pour converger.
- Enfin, un facteur de réduction de 0,99 (plutôt lent) prolonge la phase d'exploration aléatoire. À l'inverse, une valeur de 0,95 engagerait un refroidissement plus rapide et resserrerait la recherche autour de solutions déjà prometteuses.

4.2 Résultats nonogram 10x10



Figure 4.1 – Évolution de la matrice au cours des différentes étapes de l'algorithme.

4.2.0 Résumé des résultats et paramètres

Voici les différents paramètres utilisés pour les tests sur la 10x10.

Paramètre	Valeur
Pénalité mismatch	-1
Pénalité gap	-1
Nombre de threads	12
Prob accepter sol moins bonne	0.6
Facteur de réduction de la température	0.99
Ratio initial de la température basé sur le score initial	0.6

Table 4.1 – Paramètres utilisés dans l'implémentation du recuit simulé

Le tableau ci-dessous résume les résultats obtenues à la fin de l'algorithme, incluant le nombre total d'itérations effectuées, le nombre d'itérations ayant réellement contribué à une amélioration (itérations utiles), et le score global.

Temps	Nombre d'itérations	Itérations utiles	Score
Après 1 minute	4447	447	0

Table 4.2 – Résumé des résultats.

4.3 Résultats nonogram 15x15

```
0000000000000000
0000000000000000
0000001010000000
0000001111000000
0000001000000000
0000001000000000
0000101000000000
11011111111101
0000100011000000
0000100011100000
0000000010000000
0000001000010000
0000001000000000
0000000000000000
0000000000000000
```

(a) Après prétraitement.

```
3333333000000000
3033333300000000
3300001310000000
330033111103300
3330031000000000
3333031300000000
3330131000000000
11011111111101
3333103311000000
3333100011100000
3333303313000000
3330301033310000
3030331300000000
3033000000000000
3330330333300000
```

(b) Placement initial.

```
000000003333333
030000003333330
0003301310000000
330033111103300
3330001300000000
3333031300000000
3330131000000000
11011111111101
0033133011330000
0033133011100000
0333330013330000
3330301033310000
0030301333000000
0000003000000330
000033303303333
```

(c) Après 1 minute.

Figure 4.2 – Évolution de la matrice au cours des différentes étapes de l'algorithme.

4.3.0 Résumé des résultats et paramètres

Voici les différents paramètres utilisés pour les tests sur la 10x10.

Paramètre	Valeur
Pénalité mismatch	-1
Pénalité gap	-1
Nombre de threads	12
Prob accepter sol moins bonne	0.6
Facteur de réduction de la température	0.99
Ratio initial de la température basé sur le score initial	0.6

Table 4.3 – Paramètres utilisés dans l'implémentation du recuit simulé

Le tableau ci-dessous résume les résultats obtenues à la fin de l'algorithme, incluant le nombre total d'itérations effectuées, le nombre d'itérations ayant réellement contribué à une amélioration (itérations utiles), et le score global.

Temps	Nombre d'itérations	Itérations utiles	Score
Après 1 minute	276884	1052	84

Table 4.4 – Résumé des résultats.

4.4 Résultats nonogram 30x30

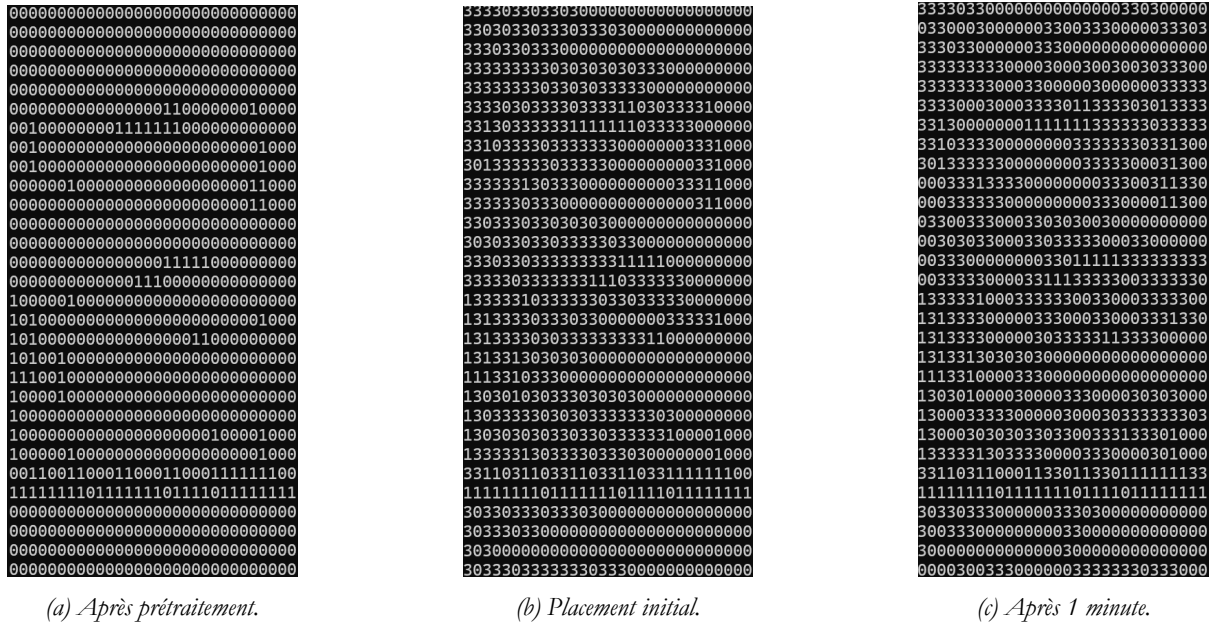


Figure 4.3 – Évolution de la matrice au cours des différentes étapes de l'algorithme.

4.4.0 Résumé des résultats et paramètres

Voici les différents paramètres utilisés pour les tests sur la 10x10.

Paramètre	Valeur
Pénalité mismatch	-1
Pénalité gap	-1
Nombre de threads	12
Prob accepter sol moins bonne	0.6
Facteur de réduction de la température	0.99
Ratio initial de la température basé sur le score initial	0.6

Table 4.5 – Paramètres utilisés dans l'implémentation du recuit simulé

Le tableau ci-dessous résume les résultats obtenues à la fin de l'algorithme, incluant le nombre total d'itérations effectuées, le nombre d'itérations ayant réellement contribué à une amélioration (itérations utiles), et le score global.

Temps	Nombre d'itérations	Itérations utiles	Score
Après 1 minute	142788	4244	231

Table 4.6 – Résumé des résultats.

5 Conclusion

Le projet démontre comment l'algorithme de recuit simulé peut être adapté pour résoudre un nonogram. L'algorithme a tiré parti des différentes optimisations telles que le prétraitement, l'implémentation d'un score basé sur Needleman-Wunsch et la parallélisation via plusieurs threads. L'ensemble de ces choix nous a permis d'améliorer la convergence vers de meilleures solutions.

5.1 Discussion

Nous avons résolu le problème par un "habile" mélange de prétraitement purement logique afin d'alléger la partie algorithmique liée au recuit simulé et donc de celui-ci, nous avons également décidé de calculer les scores par les algorithmes de Needleman-Wunsch. Les résultats obtenus pour un algorithme comme le recuit simulé sont relativement satisfaisants.

5.2 Perspectives

- Rendre le prétraitement plus sophistiqué pour identifier encore davantage de placements sûrs, diminuant le nombre d'itérations nécessaires par la suite.
- Rendre notre "simple random" plus malin, afin qu'il ne choisisse pas des groupements non déplaçables (car déjà validés) ou encore des lignes complètes.
- Étudier un éventuel système "BitFlip", celui-ci serait beaucoup moins lourd en complexité et pourrait aller beaucoup plus vite pour proposer des solutions, cependant, au prix d'une convergence moins rapide qu'actuellement.
- Changer le côté brute force du placement de groupement par quelque chose de plus "équitable" en fonction de la logique des contraintes, afin d'arriver sur une configuration plus favorable au recuit simulé.
- Supprimer la matrice Replica et directement travailler dans 2D en substituant TempRow afin de gagner en mémoire.
- Établissement d'un système plus smart afin de trouver des paramètres optimaux dans la température initiale et le facteur de refroidissement.

Bibliographie

- [1] “Theory — nonogram solver,” Stevocity.me.uk, 2025. [Online]. Available : <https://stevocity.me.uk/nonogram/theory>
- [2] C. aux projets Wikimedia, “méthode d’optimisation,” Wikipedia.org, 08 2004. [Online]. Available : https://fr.wikipedia.org/wiki/Recuit_simul%C3%A9
- [3] lb@laurentbloch.org and W. de, “Algorithme de needleman et wunsch - [site www de laurent bloch],” Laurentbloch.net, 2019. [Online]. Available : <https://www.laurentbloch.net/MySpip3/Algorithme-de-Needleman-et-Wunsch>
- [4] “Needleman-wunsch algorithm in c++ - javatpoint,” www.javatpoint.com, 2025. [Online]. Available : <https://www.javatpoint.com/needleman-wunsch-algorithm-in-cpp>

Annexe

A Code source

```
1  #include <iostream>
3  #include <vector>
   #include <random>
5  #include <algorithm>
   #include <fstream>
7  #include <string>
   #include <sstream>
9  #include <chrono>
   #include <thread>
11 #include <cmath>

13 /*-----//Définitions//-----*/
   #define INDETERMINE 0
15 #define BLACK100 1
   #define WHITE100 2
17 #define BLACKMAYBE 3

19 #define MISMATCHPENALTY -1
   #define GAPPENALTY -1
21
   #define NUMBEROFTHREADS 1
23
   #define ERRORACCEPTANCEPROBALITY 0.6
25 #define ALPHARED 0.99
   #define TEMPERATURERATIODESOLUTIONINITIALE 0.6
27
   #define RATIODECALAGEVERSDROITE 3 // 0-9
29
   const bool EvalPunitive = false;
31 // false : Met au carré la différence entre contrainte et groupement
   // true  : Met au carré les mauvaises contraintes
33
   const bool Preprocess = false;
35
   std::string filename = "pti\\10x10.pti"; // Chemin du fichier pt
37 auto MaxTiming = std::chrono::seconds(5);
   /*-----//CODE//-----*/
39 int RowSize = 0;
   int ColSize = 0;
41
   void needlemanWunsch(std::vector<short>& sequence1, std::vector<short>&
       sequence2) {
43
       int m = (int)sequence1.size();
45       int n = (int)sequence2.size();

47       // Initialize the dynamic programming matrix
       std::vector<std::vector<int>> dpMatrix(m + 1, std::vector<int>(n + 1,
           0));
49
```

```

51 // Initialize the first row and column with gap penalties
for (int i = 1; i <= m; ++i) {
53     dpMatrix[i][0] = dpMatrix[i - 1][0] + GAPPENALTY;
}

55 for (int j = 1; j <= n; ++j) {
57     dpMatrix[0][j] = dpMatrix[0][j - 1] + GAPPENALTY;
}

59 // Fill the dynamic programming matrix
for (int i = 1; i <= m; ++i) {
61     for (int j = 1; j <= n; ++j) {
63         int match = dpMatrix[i - 1][j - 1]
+ (sequence1[i - 1] == sequence2[j - 1]
? sequence1[i - 1] * sequence1[i - 1]
65         : MISMATCHPENALTY);
67         int gap1 = dpMatrix[i - 1][j] + GAPPENALTY;
        int gap2 = dpMatrix[i][j - 1] + GAPPENALTY;

69         dpMatrix[i][j] = std::max({ match, gap1, gap2 });
71     }
}

73 // Backtrack to find the optimal alignment
int i = m, j = n;
75 std::vector<short> alignedSequence1, alignedSequence2;

77 while (i > 0 || j > 0) {
79     int scoreCurrent = dpMatrix[i][j];
    int scoreDiag = (i > 0 && j > 0)
? dpMatrix[i - 1][j - 1]
81     + (sequence1[i - 1] == sequence2[j - 1] ? sequence1[i - 1] *
sequence1[i - 1] : MISMATCHPENALTY)
: -999999;
83     int scoreUp = (i > 0) ? dpMatrix[i - 1][j] + GAPPENALTY : -999999;
    int scoreLeft = (j > 0) ? dpMatrix[i][j - 1] + GAPPENALTY :
-999999;

85     if (i > 0 && j > 0 && scoreCurrent == scoreDiag) {
87         alignedSequence1.insert(alignedSequence1.begin(), sequence1[i -
1]);
        alignedSequence2.insert(alignedSequence2.begin(), sequence2[j -
1]);
89         --i;
        --j;
91     }
    else if (i > 0 && scoreCurrent == scoreUp) {
93         alignedSequence1.insert(alignedSequence1.begin(), sequence1[i -
1]);
        alignedSequence2.insert(alignedSequence2.begin(), 0);
95         --i;
    }
    else {
97         alignedSequence1.insert(alignedSequence1.begin(), 0);
        alignedSequence2.insert(alignedSequence2.begin(), sequence2[j -
99         1]);
        --j;
101     }
}

```

```

    }
sequence1 = alignedSequence1;
sequence2 = alignedSequence2;
}

bool movingRight(std::vector<short>& tempRowEnd, std::vector<short>&
tempRowStart, std::vector<short>& MovedPlace, short seedGrp)
{
    if (seedGrp < (short)tempRowStart.size() - 1) // si pas dernier GRP
    {
        if (tempRowStart[seedGrp + 1] - tempRowEnd[seedGrp] > 2) // SI
espace de bougé
        {
            for (short i = tempRowStart[seedGrp]; i < tempRowEnd[seedGrp] +
2; i++) {
                MovedPlace.push_back(i);
            }
            //group can move
            tempRowStart[seedGrp] += 1;
            tempRowEnd[seedGrp] += 1;
            return true;
        }
        else { // Suivant trop proche
            // MOVING NEXT FIRST ?
            if (movingRight(tempRowEnd, tempRowStart, MovedPlace, seedGrp +
1)) {
                for (short i = tempRowStart[seedGrp]; i < tempRowEnd[
seedGrp] + 2; i++) {
                    MovedPlace.push_back(i);
                }
                //group can move
                tempRowStart[seedGrp] += 1;
                tempRowEnd[seedGrp] += 1;
                return true;
            }
            else {
                return false;
            }
        }
    }
    else { // si Dernier GRP
        if (tempRowEnd[seedGrp] < (ColSize - 1))
        {
            for (short i = tempRowStart[seedGrp]; i < tempRowEnd[seedGrp] +
2; i++) {
                MovedPlace.push_back(i);
            }
            //group can move
            tempRowStart[seedGrp] += 1;
            tempRowEnd[seedGrp] += 1;
            return true;
        }
        else {
            return false;
        }
    }
}
}

```

```

bool movingLeft(std::vector<short>& tempRowEnd, std::vector<short>&
tempRowStart, std::vector<short>& MovedPlace, short seedGrp)
155 {
157     if (seedGrp > 0) // si pas premier GRP
    {
159         if (tempRowStart[seedGrp] - tempRowEnd[seedGrp - 1] > 2) // SI
espace de bougé
        {
161             for (short i = (short)(tempRowStart[seedGrp] - 1); i <
tempRowEnd[seedGrp] + 1; i++) {
                MovedPlace.push_back(i);
            }
163             //group can move
            tempRowStart[seedGrp] -= 1;
165             tempRowEnd[seedGrp] -= 1;
            return true;
167         }
        else { // Suivant trop proche
169             // MOVING NEXT FIRST ?
            if (movingLeft(tempRowEnd, tempRowStart, MovedPlace, seedGrp -
171 1)) {
                for (short i = (short)(tempRowStart[seedGrp] - 1); i <
tempRowEnd[seedGrp] + 1; i++) {
173                     MovedPlace.push_back(i);
                }
                //group can move
175                 tempRowStart[seedGrp] -= 1;
                tempRowEnd[seedGrp] -= 1;
177                 return true;
            }
179             else {
                return false;
181             }
        }
183     }
    else { // si Premier GRP
185         if (tempRowStart[seedGrp] > 0)
        {
187             for (short i = (short)(tempRowStart[seedGrp] - 1); i <
tempRowEnd[seedGrp] + 1; i++) {
                MovedPlace.push_back(i);
189             }
            //group can move
191             tempRowStart[seedGrp] -= 1;
            tempRowEnd[seedGrp] -= 1;
193             return true;
        }
195         else {
            return false;
197         }
    }
199 }

201 bool fitModif(short* tableauOrig, short* tableauProp)
{
203     // tableauOrig et tableauProp représentent une ligne
    // On itère donc sur ColSize (nb de colonnes)
205     for (short j = 0; j < ColSize; j++)

```

```

207     {
208         if (tableauProp[j] == BLACKMAYBE) {
209             if (tableauOrig[j] == WHITE100) {
210                 return false;
211             }
212         }
213         else {
214             if (tableauOrig[j] == BLACK100) {
215                 return false;
216             }
217         }
218     }
219     return true;
220 }
221
222 void MoveGroupsFirst(short* line, std::vector<short>& contraintesRows) {
223     std::vector<short> gStart;
224     // Detection des groupements
225     if (line[0] == BLACKMAYBE) {
226         // start new grp
227         gStart.push_back(0);
228     }
229     for (short i = 1; i < ColSize; ++i) {
230         if (line[i] == BLACKMAYBE && line[i - 1] == INDETERMINE) {
231             // start new grp
232             gStart.push_back(i);
233         }
234     }
235
236     std::vector<short> gEnd;
237     for (short i = 0; i < (short)gStart.size(); ++i) {
238         gEnd.push_back(gStart[i] + contraintesRows[i] - 1);
239     }
240
241     // Vérifie si le dernier groupement peut être déplacé
242     short cursor = (short)gStart.size() - 1;
243     if (gEnd[cursor] < (ColSize - 1))
244     {
245         gStart[cursor] += 1;
246         gEnd[cursor] += 1;
247     }
248     else {
249         cursor -= 1;
250         bool notDecal = true;
251         while (notDecal && cursor >= 0) {
252             if (gStart[cursor + 1] - gEnd[cursor] > 2) {
253                 gStart[cursor] += 1;
254                 gEnd[cursor] += 1;
255                 for (short i = cursor + 1; i < (short)gStart.size(); ++i) {
256                     gStart[i] = gEnd[i - 1] + 2;
257                     gEnd[i] = gStart[i] + contraintesRows[i] - 1;
258                 }
259                 notDecal = false;
260             }
261             else {
262                 cursor -= 1;
263             }
264         }
265     }
266 }

```

```

    }
265
    // Remet la ligne à INDETERMINE avant de positionner les BLACKMAYBE
267    for (short j = 0; j < ColSize; ++j) {
        line[j] = INDETERMINE;
269    }
    // Positionne les groupes
271    for (short i = 0; i < (short)gStart.size(); ++i) {
        for (short j = gStart[i]; j <= gEnd[i]; ++j) {
273            line[j] = BLACKMAYBE;
        }
275    }
}
277

void createPTOFile(const char* InFilePath, short** matrix, int score, int
rows, int cols) {
279    std::string filePath = "ptoOutput\\" + std::string(InFilePath) + ".pto"
;
    std::ofstream outFile(filePath);
281    if (!outFile) {
        std::cerr << "Erreur lors de la création du fichier PTO." << std::
endl;
283        return;
    }
285
    outFile << "Name: Test Solution " << std::string(InFilePath) << "\n";
287    outFile << "Solver: Recuit Simule\n";
    outFile << "Rows: " << rows << "\n";
289    outFile << "Cols: " << cols << "\n";
    outFile << "Score: " << score << "\n\n";
291
    for (int i = 0; i < rows; ++i) {
293        for (int j = 0; j < cols; ++j) {
            // Transformation 'daffichage
            short valueToWrite;
            if (matrix[i][j] == 3) {
297                valueToWrite = 1;
            }
            else if (matrix[i][j] == 2) {
299                valueToWrite = 0;
            }
            else {
301                valueToWrite = matrix[i][j];
            }
303
            outFile << valueToWrite;
305
        }
307        outFile << "\n";
309    }
}
311

void runSimulatedAnnealing(
313    short** tableau2D,
    int& ScoreTotal,
315    std::vector<short> ScoresCols,
    const std::vector<std::vector<short>>& contraintesCols,
317    const std::vector<std::vector<short>>& contraintesRows,
    std::vector<std::vector<short>> gStart,

```

```

319     std::vector<std::vector<short>>> gEnd)
{
321     float Temperature = TEMPERATURERATIODESOLUTIONINITIALE * ScoreTotal; //
TEMPERATURE

323     auto StartingChrono = std::chrono::high_resolution_clock::now();

325     // Initialise le générateur avec une graine aléatoire
std::random_device rd;
327     std::mt19937 gen(rd());

329     std::vector<short> tempRowStart;
std::vector<short> tempRowEnd;
331     std::vector<short> MovedPlace;

333     //DynamiqueVersion : copie du tableau 2D
short** tableau2DReplica = new short* [RowSize];
335     for (short i = 0; i < RowSize; i++) {
        tableau2DReplica[i] = new short[ColSize];
337         for (short j = 0; j < ColSize; j++) {
            tableau2DReplica[i][j] = tableau2D[i][j];
339         }
    }
341 }

343     bool RecuitRunStep = true;
int iteration = 0;
345     int TrueIteration = 0;

347     while (RecuitRunStep) {
        MovedPlace.clear();

349         //seedRow
std::uniform_int_distribution<> distribRow(0, RowSize - 1);
351         short seedRow = (short)distribRow(gen);

353         //seedGrp
std::uniform_int_distribution<> distribGrp(0, (int)contraintesRows[
seedRow].size() - 1);
355         short seedGrp = (short)distribGrp(gen);
//seedDir
357         std::uniform_int_distribution<> distribDir(0, 9);
short seedDir = (short)distribDir(gen);
359

        tempRowStart = gStart[seedRow];
361         tempRowEnd = gEnd[seedRow];

363         // Tentative de déplacement
bool moved = false;
365         if (seedDir > RATIODECALAGEVERSDROITE) { // DROITE
            moved = movingRight(tempRowEnd, tempRowStart, MovedPlace,
seedGrp);
367         }
        else { // GAUCHE
369             moved = movingLeft(tempRowEnd, tempRowStart, MovedPlace,
seedGrp);
        }
371         if (moved) {

```

```

373         // On remet la ligne seedRow à INDETERMINE dans la replica,
        // puis on repositionne les BLACKMAYBE pour chaque groupe de la
ligne
375         for (short j = 0; j < ColSize; ++j) {
            tableau2DReplica[seedRow][j] = INDETERMINE;
377         }
        for (short g = 0; g < (short)tempRowStart.size(); ++g) {
379             for (short c = tempRowStart[g]; c <= tempRowEnd[g]; ++c) {
                tableau2DReplica[seedRow][c] = BLACKMAYBE;
381             }
        }
383
        // Vérifie si ce changement ne contredit pas des cases déjà
BLACK100 ou WHITE100
385         if (!fitModif(tableau2D[seedRow], tableau2DReplica[seedRow])) {
            // Si modif impossible, on restaure la ligne dans la
replica
387             for (short c = 0; c < ColSize; c++) {
                tableau2DReplica[seedRow][c] = tableau2D[seedRow][c];
389             }
        }
        else {
391             // On recalcule le score pour chaque colonne impactée dans
MovedPlace
393             // MovedPlace = liste de colonnes qui ont bougé
            int ActualScore = 0;
395             std::vector<short> ScoresModified(MovedPlace.size(), 0);
            int TOTScoresModified = 0;
397
            std::vector<short> gdetect2;
399
            for (size_t iMP = 0; iMP < MovedPlace.size(); ++iMP)
401             {
                short col = MovedPlace[iMP]; // la colonne affectée
403                 gdetect2.clear();

                bool onTrack = false;
                short TrackLen = 0;
405
407                 // On balaie la colonne col en parcourant les lignes
0..RowSize-1
409                 for (short row = 0; row < RowSize; row++) {
                    if (tableau2DReplica[row][col] == WHITE100
411                        || tableau2DReplica[row][col] == INDETERMINE)
                    {
                        if (onTrack) {
413                            // FIN DE GROUPEMENT
                            onTrack = false;
415                            gdetect2.push_back(TrackLen);
                            TrackLen = 0;
417                        }
                    }
                    else {
421                        // BLACKMAYBE ou BLACK100
                        onTrack = true;
                        TrackLen++;
423                    }
                }
425             }
        }

```

```

427         // Si le dernier était noir, on ferme le groupement
428         if (TrackLen != 0) {
429             onTrack = false;
430             gdetect2.push_back(TrackLen);
431             TrackLen = 0;
432         }
433
434         signed short deltaNW = (signed short)contraintesCols[
col].size() - (signed short)gdetect2.size();
435
436         std::vector<short> seq1 = contraintesCols[col];
437         std::vector<short> seq2 = gdetect2;
438         needlemanWunsch(seq1, seq2);
439
440         // Calcul de la pénalité
441         short penalty = 0;
442         for (short idx = 0; idx < (short)seq1.size(); idx++) {
443             if (EvalPunitive) {
444                 if (seq1[idx] != seq2[idx]) penalty += (short)
std::pow((float)seq1[idx], 2.f);
445             }
446             else {
447                 penalty += (short)std::pow((float)(seq1[idx] -
seq2[idx]), 2.f);
448             }
449         }
450         penalty += (short)std::pow((float)deltaNW, 2.f);
451         ScoresModified[iMP] = penalty;
452
453         // On ajoute l'ancien score et le nouveau
454         ActualScore += ScoresCols[col];
455         TOTScoresModified += penalty;
456     }
457
458     // On décide si 'lon accepte ou non la solution
459     if (ActualScore >= TOTScoresModified) {
460         // Meilleure solution -> on accepte
461         for (short c = 0; c < ColSize; c++) {
462             if (tableau2D[seedRow][c] != BLACK100 && tableau2D[
seedRow][c] != WHITE100) {
463                 tableau2D[seedRow][c] = tableau2DReplica[
seedRow][c];
464             }
465         }
466         TrueIteration++;
467         ScoreTotal = ScoreTotal + (TOTScoresModified -
ActualScore);
468
469         // On met à jour gStart / gEnd du master
470         gStart[seedRow] = tempRowStart;
471         gEnd[seedRow] = tempRowEnd;
472
473         // On met à jour les scores de colonnes
474         for (size_t iMP = 0; iMP < MovedPlace.size(); ++iMP) {
475             ScoresCols[MovedPlace[iMP]] = ScoresModified[iMP];
476         }
477     }
478     else {

```

```

// Solution moins bonne -> on tente 'l'acceptation
probabiliste (recuit)
479     int DeltaE = TOTScoresModified - ActualScore;
        if (std::exp(-DeltaE / Temperature) >=
ERRORACCEPTANCEPROBABILITY) {
481         // On accepte quand même
            for (short c = 0; c < ColSize; c++) {
483                 if (tableau2D[seedRow][c] != BLACK100 &&
tableau2D[seedRow][c] != WHITE100) {
                        tableau2D[seedRow][c] = tableau2DReplica[
seedRow][c];
485                 }
            }
            TrueIteration++;
            ScoreTotal += DeltaE;
            gStart[seedRow] = tempRowStart;
            gEnd[seedRow] = tempRowEnd;
491
            for (size_t iMP = 0; iMP < MovedPlace.size(); ++iMP
) {
493                 ScoresCols[MovedPlace[iMP]] = ScoresModified[
iMP];
            }
            // On refroidit la température
            Temperature = Temperature * ALPHARED;
497        }
        else {
499            // On refuse -> on restaure la replica
            for (short c = 0; c < ColSize; c++) {
501                tableau2DReplica[seedRow][c] = tableau2D[
seedRow][c];
            }
503        }
    }
505 }
}
507 // else -> on n'a pas pu bouger du tout

iteration++;
auto CurrentChrono = std::chrono::high_resolution_clock::now();
511 auto Duration = std::chrono::duration_cast<std::chrono::seconds>(
CurrentChrono - StartingChrono);

513 // Critères d'arrêt
    if (ScoreTotal <= 0 || Duration > MaxTiming)
515         RecuitRunStep = false;
}
517 std::cout << std::endl << "Iteration " << iteration << std::endl;
std::cout << std::endl << "True Iteration " << TrueIteration << std:::
endl;
519 // Libération du tableau replica
    for (short i = 0; i < RowSize; i++) {
521         delete[] tableau2DReplica[i];
    }
523 delete[] tableau2DReplica;
}
525
int main()

```

```

527 {
    auto StartingChrono = std::chrono::high_resolution_clock::now();
529
    /*-----//Lecture PTI//-----*/
531    std::vector<std::vector<short>> contraintesCols;
    std::vector<std::vector<short>> contraintesRows;
533    std::string cuttedName = filename.substr(filename.find_last_of("\\/") +
        1);
    cuttedName = cuttedName.substr(0, cuttedName.find_last_of("."));
535    const char* cuttedNamechar = cuttedName.c_str();

537    std::fstream myfile(filename);
    std::string myline;
539    short nbrenoir = 0;
    bool firstEmptyLine = false;
541    bool secondEmptyLine = false;

543    while (std::getline(myfile, myline)) {
        if (myline.rfind("Rows:", 0) == 0) {
545            sscanf_s(myline.c_str(), "Rows: %d", &RowSize);
        }
547        if (myline.rfind("Cols:", 0) == 0) {
            sscanf_s(myline.c_str(), "Cols: %d", &ColSize);
549        }

551        if (myline.empty()) {
            if (!firstEmptyLine) {
553                firstEmptyLine = true;
                continue;
555            }
            else if (!secondEmptyLine) {
557                secondEmptyLine = true;
                continue;
559            }
            else {
561                break;
            }
563        }

565        if (firstEmptyLine && !secondEmptyLine) {
            std::stringstream ss(myline);
            std::vector<short> constraints;
            short value;
567            while (ss >> value) {
                nbrenoir += value;
                constraints.push_back(value);
569            }
            contraintesCols.push_back(constraints); // Ajouter aux
571            colonnes
        }

573
575        if (secondEmptyLine) {
            std::stringstream ss(myline);
            std::vector<short> constraints;
            short value;
577            while (ss >> value) {
                constraints.push_back(value);
579            }
581        }
    }

```

```

583         contraintesRows.push_back(constraints); // Ajouter aux lignes
584     }
585 }
586 myfile.close();
587
588 // Vérif de la taille lue
589 // contraintesCols.size() devrait être = ColSize
590 // contraintesRows.size() devrait être = RowSize
591
592 // Création du tableau 2D
593 short** tableau2D = new short* [RowSize];
594 for (short i = 0; i < RowSize; i++) {
595     tableau2D[i] = new short[ColSize];
596     for (short j = 0; j < ColSize; j++) {
597         tableau2D[i][j] = INDETERMINE;
598     }
599 }
600
601 /*-----//Pre-Process//-----*/
602 if (Preprocess) {
603     // ROW PREPROCESS
604     short HalfRows = (short)(ColSize / 2); // Hard Rounded : le "milieu
605     " sur la longueur 'd'une ligne (nb colonnes)
606     for (short i = 0; i < RowSize; i++)
607     {
608         short tmp = (short)(contraintesRows[i].size() - 1);
609         for (short j = 0; j < (short)contraintesRows[i].size(); j++)
610         {
611             // count black groupe
612             tmp += contraintesRows[i][j];
613         }
614         // Si somme (noir + espaces) > la moitié de la ligne,
615         // on place quelques cases BLACK100 en fin de groupe, etc.
616         if (tmp > HalfRows)
617         {
618             short VariableSpace = 0;
619             for (short j = 0; j < (short)contraintesRows[i].size(); j
620 ++)
621             {
622                 for (short k = 0; k < contraintesRows[i][j]; k++)
623                 {
624                     // si k >= (ColSize - tmp), on "fixe" en noir
625                     if (k >= (ColSize - tmp)) {
626                         // i : la ligne, (k + VariableSpace) : la
627                         colonne
628                         tableau2D[i][k + VariableSpace] = BLACK100;
629                     }
630                 }
631                 VariableSpace += contraintesRows[i][j] + 1;
632             }
633         }
634     }
635
636     // COLS PREPROCESS
637     short HalfCols = (short)(RowSize / 2); // Hard Rounded : le "milieu
638     " sur la hauteur 'd'une colonne (nb lignes)
639     for (short j = 0; j < ColSize; j++)
640     {

```

```

637     short tmp = (short)(contraintesCols[j].size() - 1);
639     for (short c = 0; c < (short)contraintesCols[j].size(); c++)
641     {
        tmp += contraintesCols[j][c];
643     }
    if (tmp > HalfCols)
    {
        short VariableSpace = 0;
645        for (short c = 0; c < (short)contraintesCols[j].size(); c
++))
        {
647            for (short k = 0; k < contraintesCols[j][c]; k++)
            {
649                // si k >= (RowSize - tmp), on "fixe" en noir
                if (k >= (RowSize - tmp)) {
651                    // k + VariableSpace = index de ligne, j =
colonne
                    tableau2D[k + VariableSpace][j] = BLACK100;
653                }
            }
655            VariableSpace += contraintesCols[j][c] + 1;
        }
657    }
}

659 // AFFICHAGE SOL PREPROCESS
661 std::cout << "\nAFTER PREPROCESS\n";
663 for (short i = 0; i < RowSize; i++)
{
    for (short j = 0; j < ColSize; j++)
665    {
        std::cout << tableau2D[i][j];
667    }
    std::cout << std::endl;
669 }
}

671 /*-----//Placement des groupements//-----*/
673 // On stocke les positions de départ/fin de groupes pour chaque ligne
std::vector<std::vector<short>>> gStart;
std::vector<std::vector<short>>> gEnd;
675 gStart.resize(RowSize);
gEnd.resize(RowSize);

677 for (short i = 0; i < RowSize; i++)
679 {
    short* tableau1DG = new short[ColSize];
681    for (short j = 0; j < ColSize; j++) {
        tableau1DG[j] = INDETERMINE;
683    }
    short VariableSpace = 0;

685    // On place a priori les groupes bout à bout
687    for (short j = 0; j < (short)contraintesRows[i].size(); j++)
    {
689        gStart[i].push_back(VariableSpace);
        for (short k = 0; k < contraintesRows[i][j]; k++)
691        {
            tableau1DG[k + VariableSpace] = BLACKMAYBE;

```

```

693     }
        gEnd[i].push_back((short)(contraintesRows[i][j] + VariableSpace
- 1));
695     VariableSpace += contraintesRows[i][j] + 1;
        }
697
        if (fitModif(tableau2D[i], tableau1DG)) {
699             // On recopie dans la solution
            for (short j = 0; j < ColSize; j++)
701             {
                if (tableau2D[i][j] != BLACK100 && tableau2D[i][j] !=
WHITE100) {
703                     tableau2D[i][j] = tableau1DG[j];
                }
705             }
        }
707     else {
        // Sinon, on essaye de faire des décalages successifs
709     bool notOK = true;
        while (notOK) {
711             // On essaye de bouger le dernier groupe
            short cursor = (short)gStart[i].size() - 1;
713             if (gEnd[i][cursor] < (ColSize - 1))
            {
715                 gStart[i][cursor] += 1;
                gEnd[i][cursor] += 1;
717             }
            else {
719                 cursor -= 1;
                bool notDecal = true;
721                 while (notDecal && cursor >= 0) {
                    if (gStart[i][cursor + 1] - gEnd[i][cursor] > 2) {
723                         gStart[i][cursor] += 1;
                        gEnd[i][cursor] += 1;
725                         for (short kk = cursor + 1; kk < (short)gStart[
i].size(); ++kk) {
                            gStart[i][kk] = (short)(gEnd[i][kk - 1] +
2);
727                             gEnd[i][kk] = (short)(gStart[i][kk] +
contraintesRows[i][kk] - 1);
                        }
729                         notDecal = false;
                    }
                    else {
731                         cursor -= 1;
                    }
                    }
733             }
        }
735
        // On reconstruit tableau1DG
737     for (short j = 0; j < ColSize; j++) {
        tableau1DG[j] = INDETERMINE;
739     }
        for (short idx = 0; idx < (short)gStart[i].size(); ++idx) {
741             for (short col = gStart[i][idx]; col <= gEnd[i][idx];
++col) {
                tableau1DG[col] = BLACKMAYBE;
743             }
        }
    }
}

```

```

745         if (fitModif(tableau2D[i], tableau1DG)) {
747             for (short j = 0; j < ColSize; j++)
749                 {
751                     if (tableau2D[i][j] != BLACK100 && tableau2D[i][j]
753 != WHITE100) {
755                         tableau2D[i][j] = tableau1DG[j];
757                     }
759                     notOK = false;
761                 }
763             }
765         delete[] tableau1DG;
767     }

769     // AFFICHAGE Placement initial
771     std::cout << "\nPlacement initial\n";
773     for (short i = 0; i < RowSize; i++)
775     {
777         for (short j = 0; j < ColSize; j++)
779         {
781             std::cout << tableau2D[i][j];
783             std::cout << std::endl;
785         }

787         /*-----//Recuit Simulé PREPA//-----*/
789         // Calcul du score initial par COLONNE
791         std::vector<short> ScoresCols;
793         ScoresCols.resize(ColSize, 0);

795         int ScoreTotal = 0;

797         // Pour chaque colonne j
799         for (short j = 0; j < ColSize; j++) {
801             std::vector<short> gdetect;
803             bool onTrack = false;
805             short TrackLen = 0;
807             // on scanne la colonne j, en parcourant les lignes i=0..RowSize-1
809             for (short i = 0; i < RowSize; i++) {
811                 if (tableau2D[i][j] == WHITE100 || tableau2D[i][j] ==
813 INDETERMINE) {
815                     if (onTrack) {
817                         onTrack = false;
819                         gdetect.push_back(TrackLen);
821                         TrackLen = 0;
823                     }
825                     else {
827                         // 'cest un noir possible
829                         onTrack = true;
831                         TrackLen++;
833                     }
835                 }
837             }
839             // fermeture de groupe éventuelle
841             if (TrackLen != 0) {
843                 onTrack = false;
845                 gdetect.push_back(TrackLen);

```

```

801     TrackLen = 0;
      }
803
      // On compare gdetect à contraintesCols[j] via Needleman-Wunsch
805     signed short deltaNW = (signed short)contraintesCols[j].size() - (
signed short)gdetect.size();

807     std::vector<short> seq1 = contraintesCols[j];
      std::vector<short> seq2 = gdetect;
809     needlemanWunsch(seq1, seq2);

811     short penalty = 0;
      for (short idx = 0; idx < (short)seq1.size(); idx++) {
813         if (EvalPunitive) {
            if (seq1[idx] != seq2[idx]) {
815                 penalty += (short)std::pow((float)seq1[idx], 2.f);
            }
817         }
            else {
819                 penalty += (short)std::pow((float)(seq1[idx] - seq2[idx]),
2.f);
            }
821         }
            penalty += (short)std::pow((float)deltaNW, 2.f);
823
            ScoresCols[j] = penalty;
825             ScoreTotal += penalty;
        }

827     /*-----//Recuit Simulé MPx//-----*/
829     std::vector<std::thread> threads;
      std::vector<int> ScoringTable;
831     std::vector<short**> TableFromThread;

833     // DUPLICATION
      for (int t = 0; t < NUMBEROFTHREADS; ++t) {
835         ScoringTable.push_back(ScoreTotal);

837         short** tableau2DNEW = new short* [RowSize];
            for (short i = 0; i < RowSize; i++) {
839                 tableau2DNEW[i] = new short[ColSize];
                    for (short j = 0; j < ColSize; j++) {
841                         tableau2DNEW[i][j] = tableau2D[i][j];
                    }
843             }
            TableFromThread.push_back(tableau2DNEW);
845         }

847     // Lancement des threads
      for (int i = 0; i < NUMBEROFTHREADS; ++i) {
849         threads.emplace_back(
            runSimulatedAnnealing,
851             TableFromThread[i],
            std::ref(ScoringTable[i]),
853             ScoresCols,
            std::cref(contraintesCols),
855             std::cref(contraintesRows),
            gStart,

```

```

857         gEnd
858     );
859 }
860
861 // Attente de la fin de tous les threads
862 for (auto& t : threads) {
863     if (t.joinable()) {
864         t.join();
865     }
866 }
867
868 // On regarde le meilleur score
869 short IDBestOne = 0;
870 int TmpScore = ScoringTable[0];
871 for (int i = 0; i < NUMBEROFTHREADS; ++i) {
872     if (ScoringTable[i] < TmpScore) {
873         TmpScore = ScoringTable[i];
874         IDBestOne = (short)i;
875     }
876 }
877
878 for (int i = 0; i < NUMBEROFTHREADS; ++i) {
879     std::cout << "\nScore Thread " << i << " : " << ScoringTable[i] <<
std::endl;
880 }
881
882 createPTOFile(cuttetdNamechar, TableFromThread[IDBestOne], ScoringTable[
IDBestOne], RowSize, ColSize);
883
884 // AFFICHAGE SOL FINAL
885 std::cout << "\n\nFINAL\n";
886 for (short i = 0; i < RowSize; i++)
887 {
888     for (short j = 0; j < ColSize; j++)
889     {
890         std::cout << TableFromThread[IDBestOne][i][j];
891     }
892     std::cout << std::endl;
893 }
894
895 for (short i = 0; i < RowSize; i++) {
896     delete[] tableau2D[i];
897 }
898 delete[] tableau2D;
899
900 for (int t = 0; t < NUMBEROFTHREADS; t++) {
901     for (short i = 0; i < RowSize; i++) {
902         delete[] TableFromThread[t][i];
903     }
904     delete[] TableFromThread[t];
905 }
906
907 return 0;
908 }
909 }

```

Listing 1 – Modification calcul score de match

A.1 Frontend

/

A.2 Backend

/

B Timesheets

	Vanhaverbeke Axel	Pluinage Theo	Szczepanski Jonathan	Tricknot Guillaume	Tomisnec V. Xavier
PTI	1h	/	/	1h	1h
Prétraitement	2h	4h	6h	3h (ne s'était pas concerté avec ses collègues)	/
Placement des groupes	/	3h	/	/	/
Compréhension algorithme	3h	3h	3h	3h	3h
Compréhension métaheuristique	2h (presque compris mais non)	2h(+10h d'explication)	4h30	1h30	4h
Recuit simulé	/	4h	/	/	/
Scoring	/	3h	/	2h	/
N-W	/	/	/	/	4h30
Multiprocessing	1h30	2h	/	/	/
PTO	1h30	/	/	1h	/
Fusion du code	3h	2h	/	/	1h
Debug	4h(je déteste Théo)	3h(My bad sry Axel)	/	1h	/
Timer du code	/	2 sec	/	5 sec	/
Benchmarking	2h	30 min		30min	/
Rapport	3h	2h	3h	6h (est une vraie secrétaire *bruit de machine à écrire*)	3h
Prototype de code non utilisé	3h	4h	5h	3h	4h30
Total	26h	32h30min2sec	21h30	22h00	21h00

(a) TimeSheet du groupe recuit simulé