

Résolution du Jeu SUDOKU par méta-heuristiques

Recuit Simulé



RÉSUMÉ

Le Présent rapport présente l'utilisation de la métaheuristique Recuit Simulé pour résoudre le jeu de Sudoku. Le Sudoku est un puzzle numérique qui implique le placement de chiffres de 1 à 9 dans une grille 9x9 de manière à ce que chaque ligne, chaque colonne et chaque région de 3x3 contiennent tous les chiffres de 1 à 9.

Le rapport commence par une présentation des concepts de base d'optimisation combinatoire, des différentes méthodes de résolution incluant les métaheuristiques arrivant au Recuit Simulé, qui est une technique de recherche locale pour résoudre des problèmes d'optimisation combinatoire. Les principales étapes de l'algorithme de Recuit Simulé sont expliquées en détail, y compris l'initialisation, la génération de solutions voisines, la sélection de la solution et la mise à jour de la température.

Le rapport présente ensuite l'application de l'algorithme de Recuit Simulé pour résoudre des puzzles Sudoku. Les résultats montrent que l'algorithme de Recuit Simulé est capable de résoudre des puzzles Sudoku de difficulté variable de manière efficace. L'algorithme est également capable de trouver des solutions optimales pour certains puzzles.

Le rapport conclut que l'utilisation de la métaheuristique Recuit Simulé pour résoudre des puzzles Sudoku est une approche efficace qui peut être utilisée pour résoudre des problèmes d'optimisation combinatoire similaires. Cependant, des recherches supplémentaires peuvent être nécessaires pour optimiser les paramètres de l'algorithme afin d'améliorer les performances.

TABLE DES MATIERES

résumé.....	2
Tables des figures	5
Chapitre 1 : Introduction.....	6
Optimisation Combinatoire	6
Classes de complexité.....	7
Problème P	7
Problème NP	8
Problème NP-difficile.....	8
Problème NP-complet	8
Méthodes de résolution en optimisation combinatoire	8
Classification des méthodes de résolution.....	9
Méthodes exactes	9
Méthodes approchées	9
Notion D'heuristique.....	10
Définition d'une heuristiQUE.....	10
Définition d'une métaheuristique	10
Métaheuristiques à solution unique	11
Métaheuristiques à population de solutions	12
Conclusion	12
Chapitre 2 : Le recuit Simulé	13
Définition de l'algorithme	13
Les étapes de base du recuit simulé.....	13
Initialiser la solution initiale	14
La méthode Gloutonne.....	15
Définir un voisinage	15
Sudoku et Recuit simulé.....	16
Présentation du jeu sudoku.....	16
Etapes de résolution du sudoku par recuit simulé	16

Chapitre 3 : Résolution du jeu sudoku	18
1 ^{er} étape : Affichage de la grille du sudoku	18
2 ^{ème} étape : Conversion des cases prédefinis en 1	19
3 ^{ème} étape : Définir la fonction objectif	20
4 ^{ème} étape : générer une solution initiale	21
5 ^{ème} étape : Définir un voisinage.....	23
Choisir un nouvel état	27
Choisir un nombre d'iteration	29
Calculer la temperature initiale	29
Résoudre le sudoku.....	31
Solution.....	34
Conclusion	35
References.....	36
Code source	37

TABLES DES FIGURES

Figure 1: Illustration des inclusions des classes de complexité	7
Figure 2: La taxonomie des méthodes de résolution des problèmes d'optimisation	9
Figure 3: Sudoku à résoudre	18
Figure 4: convertir d'une chaîne de caractère à un tableau sudoku	18
Figure 5: affichage de la grille du sudoku	19
Figure 6: convertir les valeurs remplies en 1	20
Figure 7: Calculer le nombre d'erreurs par ligne & colonne	20
Figure 8: Calculer le nombre total d'erreurs	20
Figure 9: Créer une liste de blocs	21
Figure 10: Remplissage Aléatoire du bloc	22
Figure 11: Calculer la somme des valeurs d'un bloc	23
Figure 12: Choisir 2 cases aléatoire dans un bloc	24
Figure 13: Retourner deux boxes	25
Figure 14: proposition d'un nouvel état	26
Figure 15: Choisir un nouvel état	27
Figure 16: Choisir le nombre d'itérations	29
Figure 17: Calculer la température initiale	30
Figure 18: Résoudre le sudoku	33
Figure 19: Affichage du résultat	33
Figure 20: Sudoku Résolu	34

CHAPITRE 1 : INTRODUCTION

OPTIMISATION COMBINATOIRE

L'optimisation combinatoire est une branche de l'optimisation en mathématiques appliquées et en informatique, également liée à la recherche opérationnelle, l'algorithmique et la théorie de la complexité. On parle également d'optimisation discrète.

DÉFINITION :

Un problème d'optimisation combinatoire consiste à trouver la meilleure solution dans un ensemble discret dit ensemble des solutions réalisables. En général, cet ensemble est fini mais compte un très grand nombre d'éléments, et il est décrit de manière implicite, c'est-à-dire par une liste, relativement courte, de contraintes que doivent satisfaire les solutions réalisables.

Pour définir la notion de meilleure solution, une fonction, dite fonction objective, est introduite. Pour chaque solution, elle renvoie un réel et la meilleure solution (ou solution optimale) est celle qui minimise ou maximise la fonction objective. Clairement, un problème d'optimisation combinatoire peut avoir plusieurs solutions optimales.

RÉSOLUTION :

Trouver une solution optimale dans un ensemble discret et fini est un problème facile en théorie : il suffit d'essayer toutes les solutions, et de comparer leurs qualités pour voir la meilleure. Cependant, en pratique, l'énumération de toutes les solutions peut prendre trop de temps ; or, le temps de recherche de la solution optimale est un facteur très important et c'est à cause de lui que les problèmes d'optimisation combinatoire sont réputés si difficiles. La théorie de la complexité donne des outils pour mesurer ce temps de recherche. De plus, comme l'ensemble des solutions réalisables est défini de manière implicite, il est aussi parfois très difficile de trouver ne serait-ce qu'une solution réalisable.

Quelques problèmes d'optimisation combinatoire peuvent être résolus (de manière exacte) en temps polynomial par exemple par un algorithme glouton, un algorithme de programmation dynamique ou en montrant que le problème peut être formulé comme un programme linéaire en variables réelles.

Dans la plupart des cas, le problème est NP complet et, pour le résoudre, il faut faire appel à des algorithmes de branch and bound, à la programmation linéaire en nombres entiers ou encore à la programmation par contraintes. En pratique, on se contente très souvent d'avoir une solution approchée, obtenue par une heuristique ou une métaheuristique. Pour certains problèmes, on peut prouver une garantie de performance, c'est-à-dire que l'écart entre la solution obtenue et la solution optimale est borné.

CLASSES DE COMPLEXITÉ

La théorie de la complexité permet de classer les problèmes de décision en fonction de la complexité des algorithmes qui existent pour les résoudre.

On distingue les problèmes décidables pour lesquels il existe au moins un algorithme pour les résoudre, de ceux indécidables.

Il existe plusieurs classes de complexité, mais les plus connues sont :

- Classe P
- Classe NP
- Classe NP-difficile
- Classe NP-complet

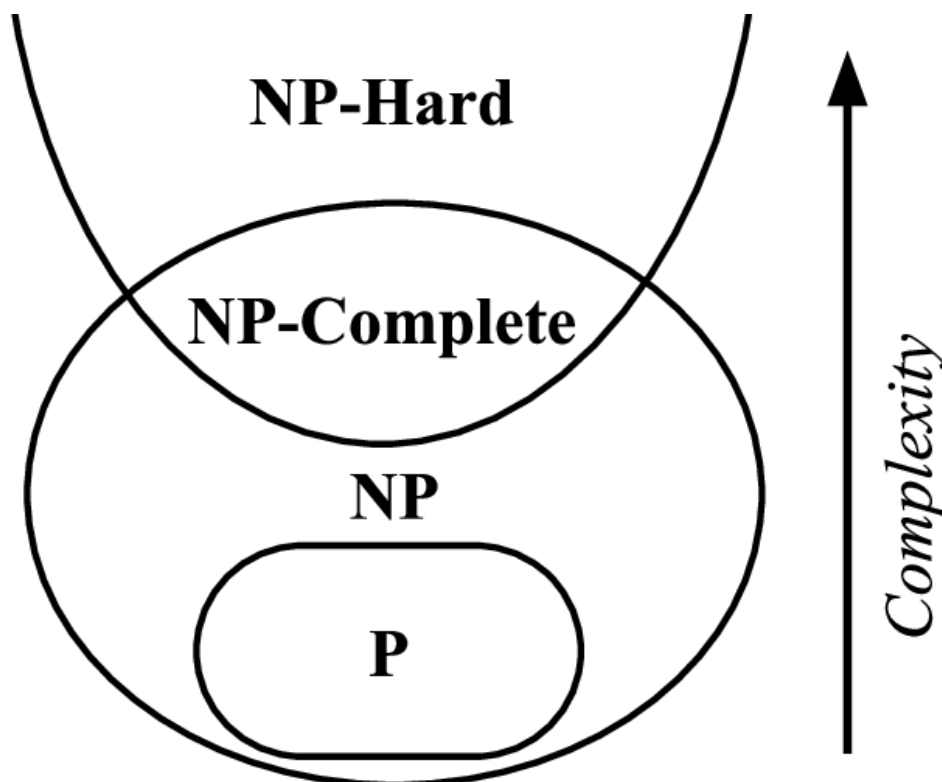


Figure 1: Illustration des inclusions des classes de complexité

PROBLÈME P

P est une classe de complexité qui représente l'ensemble des problèmes de décision pouvant être résolus en temps polynomial. Autrement dit, étant donné un exemple du problème, la réponse « oui » ou « non » peut être décidée en temps polynomial.

Puisqu'ils peuvent être résolus en temps polynomial, ils peuvent également être vérifiés en temps polynomial.

Par conséquent, P est un sous-ensemble de NP.

EXEMPLE :

Avec un graphe G connecté, peut-on colorer ses sommets en utilisant deux couleurs afin qu'aucun bord ne soit monochromatique ?

Algorithme : commencez avec un sommet arbitraire, colorez-le en vert et tous ses voisins en bleu et continuez. Arrêtez-vous lorsque vous êtes à court de sommets ou que vous devez créer une arête pour que ses deux extrémités aient la même couleur.

PROBLÈME NP

Un problème est attribué à la classe NP (temps polynomial non déterministe) si elle est résolue en temps polynomial par une machine de Turing non déterministe.

Un problème P (dont le temps de résolution est limité par un polynôme) est toujours aussi NP. Si un problème est connu comme étant NP et si une solution au problème est connue, la démonstration de l'exactitude de la solution peut toujours être réduite à une vérification P (temps polynomial). Si P et NP ne sont pas équivalents, la solution des problèmes de NP nécessite (dans le pire des cas) une recherche exhaustive.

La programmation linéaire, connue depuis longtemps comme étant NP et supposée ne pas être P, a été démontrée par L. Khachian en 1979. Il est important de déterminer si tous les problèmes apparemment liés aux NP sont réellement P.

PROBLÈME NP-DIFFICILE

Un problème est dit NP-difficile si un algorithme pour le résoudre peut-être traduit en un pour résoudre tout autre problème NP. Il est beaucoup plus facile de montrer qu'un problème est NP que de montrer que c'est NP-difficile. Un problème à la fois NP et NP-difficile est appelé un problème NP-complet.

PROBLÈME NP-COMPLET

Un problème x qui est dans NP est également dans NP-Complete si et seulement si tous les autres problèmes dans NP peuvent être rapidement (c'est-à-dire en temps polynomial) transformés en x.

En d'autres termes :

- x est dans NP, et
- Chaque problème dans NP est réductible à x

Donc, ce qui rend NP-Complete si intéressant, c'est que si l'un des problèmes NP-Complete devait être résolu rapidement, alors tous les problèmes NP peuvent être résolus rapidement.

MÉTHODES DE RÉOLUTION EN OPTIMISATION COMBINATOIRE

Face à un problème d'optimisation combinatoire donné, la question à laquelle on doit répondre est la résolution du problème. Ce dernier possède généralement un nombre énorme de solutions réalisables. La résolution la plus évidente est de lister toutes les combinaisons possibles afin de trouver celles qui sont valides et meilleures.

On appelle ce type d'algorithme : Algorithmes Exhaustifs

Ces algorithmes sont très gourmands en termes de complexité. Ils passent d'algorithmes polynomiaux à non-polynomiaux avec l'augmentation de la dimension du problème à traiter.

CLASSIFICATION DES MÉTHODES DE RÉOLUTION

Les méthodes de l'optimisation combinatoire peuvent être classées en deux grandes familles de classes : les méthodes exactes et les méthodes approchées.

La figure ci-dessous illustre la taxonomie des méthodes de résolution des problèmes d'optimisation.

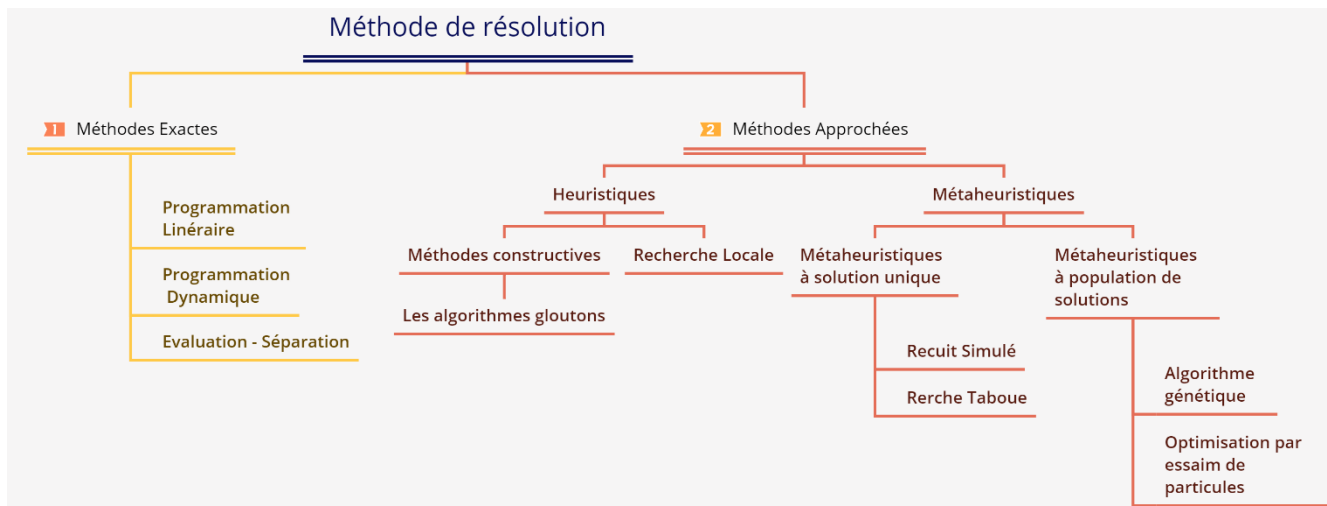


Figure 2: La taxonomie des méthodes de résolution des problèmes d'optimisation

MÉTHODES EXACTES

Les méthodes exactes (appelées aussi complètes) produisent une solution optimale pour une instance de problème d'optimisation donné. Elles se reposent généralement sur la recherche arborescente et sur l'énumération partielle de l'espace de solutions.

Elles sont utilisées pour trouver au moins une solution optimale d'un problème. Les algorithmes exactes les plus connus sont la méthode de séparation et évaluation, la programmation dynamique, et la programmation linéaire.

L'inconvénient majeur de ses méthodes est l'explosion combinatoire : Le nombre de combinaisons augmente avec l'augmentation de la dimension du problème. L'efficacité de ces algorithmes n'est prometteuse que pour les instances de problèmes de petites tailles.

MÉTHODES APPROCHÉES

Contrairement aux méthodes exactes, les méthodes approchées ne fournissent pas forcément une solution optimale, mais seulement une bonne solution (de qualité raisonnable) dans un temps raisonnable.

Les méthodes approchées de résolution des problèmes d'optimisation combinatoire cherchent à fournir des solutions de qualité raisonnable, tout en ayant des temps de calculs raisonnables. Voici quelques exemples de telles méthodes :

- **Heuristiques** : Les heuristiques sont des algorithmes de recherche locale qui se concentrent sur une région spécifique de l'espace de recherche, cherchant à atteindre une solution satisfaisante localement, sans garantie d'optimalité globale.

- **Méthodes de construction :** Les méthodes de construction visent à construire une solution réalisable en combinant différentes composantes de manière heuristique. Des exemples de telles méthodes incluent la construction gloutonne, la recherche par coûts, la recherche par flots, etc.

NOTION D'HEURISTIQUE

DÉFINITION D'UNE HEURISTIQUE

Une heuristique est une méthode ou une stratégie de résolution de problèmes qui fournit une solution approchée, souvent rapidement, mais sans garantie de l'optimalité de cette solution. Les heuristiques sont généralement utilisées pour résoudre des problèmes complexes, qui ne peuvent pas être résolus de manière exacte en un temps raisonnable. Les heuristiques peuvent également être utilisées pour améliorer les solutions initiales produites par d'autres méthodes.

Les heuristiques se concentrent souvent sur des régions spécifiques de l'espace de recherche, en explorant certaines solutions prometteuses, plutôt que de considérer l'ensemble de toutes les solutions possibles. Elles peuvent également utiliser des règles de décision heuristiques, qui ne sont pas forcément basées sur des principes mathématiques ou sur une analyse exhaustive des données.

Les heuristiques sont utilisées dans une grande variété de domaines, y compris en optimisation combinatoire, en intelligence artificielle, en apprentissage automatique, en planification et en gestion de projet, entre autres. Les exemples de heuristiques courantes sont la recherche taboue, la recherche locale itérative, la recherche de voisinage, les algorithmes génétiques, les méthodes de Monte Carlo, etc.

DÉFINITION D'UNE MÉTAHEURISTIQUE

Le mot métaheuristique est composé de deux mots grecs : méta et heuristique. Le mot méta est un suffixe signifiant au-delà c'est-à-dire de niveau supérieur.

Les métaheuristiques sont des méthodes généralement inspirées de la nature. Contrairement aux heuristiques, elles s'appliquent à plusieurs problèmes de nature différentes. Pour cela on peut dire qu'elles sont des heuristiques modernes, de plus haut niveau, dédiées particulièrement à la résolution des problèmes d'optimisation. Leur but est d'atteindre un optimum global tout en échappant les optima locaux.

Une métaheuristique est une méthode de résolution de problèmes d'optimisation combinatoire qui utilise des stratégies de recherche itératives et des techniques de diversification pour explorer efficacement l'espace de recherche des solutions, tout en évitant de rester bloqué dans des minimums locaux.

Les métaheuristiques sont des algorithmes généraux qui peuvent être appliqués à une grande variété de problèmes d'optimisation combinatoire, sans nécessiter de connaissances spécifiques du problème. Elles sont souvent utilisées pour résoudre des problèmes complexes, pour lesquels les méthodes exactes ne sont pas pratiques ou pour lesquels il n'existe pas de méthode exacte connue qui permette de trouver la solution optimale dans un temps raisonnable.

Les métaheuristiques utilisent souvent des mécanismes de recherche itératifs qui comprennent des étapes de génération de solutions aléatoires, de modification de ces solutions, d'évaluation de leur qualité et d'acceptation ou de rejet de ces solutions modifiées. Ces mécanismes de recherche peuvent être combinés avec des techniques de diversification, qui visent à explorer des régions de l'espace de recherche qui n'ont pas encore été visitées ou qui ont été visitées avec peu de succès.

Les exemples de métaheuristiques courantes sont le recuit simulé, la recherche taboue, les algorithmes génétiques, les colonies de fourmis, les essaims de particules, la recherche de voisinage variable, la recherche en essaims, etc.

Les métaheuristiques ont largement contribué à améliorer la qualité des solutions pour de nombreux problèmes d'optimisation combinatoire et ont été utilisées avec succès dans de nombreux domaines d'application, tels que l'ingénierie, la finance, la planification de la production, la logistique, etc.

Les métaheuristiques regroupent des méthodes qui peuvent se diviser en deux classes :

1. Métaheuristiques à solution unique
2. Métaheuristiques à population de solutions

MÉTAHEURISTIQUES À SOLUTION UNIQUE

Les métaheuristiques à solution unique, ou Single Solution Metaheuristics en anglais, sont des méthodes de résolution de problèmes d'optimisation combinatoire qui travaillent sur une seule solution à la fois. Contrairement aux métaheuristiques multi-solutions qui génèrent et manipulent un ensemble de solutions en parallèle, les métaheuristiques à solution unique se concentrent sur l'amélioration progressive d'une seule solution.

Les métaheuristiques à solution unique ont souvent un avantage en termes de temps de calcul et de mémoire, car elles ne nécessitent pas de stocker et de manipuler un grand nombre de solutions simultanément. Elles peuvent également être plus efficaces pour des problèmes dans lesquels la qualité de la meilleure solution est plus importante que la diversité des solutions, tels que des problèmes de conception ou d'ingénierie.

Les exemples de métaheuristiques à solution unique comprennent le recuit simulé, la recherche taboue, la recherche de voisinage variable, la recherche de descente de gradient, etc. Ces méthodes cherchent à améliorer la solution courante en explorant les solutions voisines, soit en modifiant la solution courante, soit en explorant l'espace de recherche en utilisant des techniques aléatoires.

Bien que les métaheuristiques à solution unique puissent ne pas être aussi puissantes que les métaheuristiques multi-solutions dans certaines situations, elles peuvent être une alternative efficace pour des problèmes spécifiques, en particulier lorsqu'ils sont combinés avec d'autres méthodes telles que des techniques de modélisation, des heuristiques ou des algorithmes de traitement du signal.

MÉTAHEURISTIQUES À POPULATION DE SOLUTIONS

Les métaheuristiques à population de solutions, ou Population-based Metaheuristics en anglais, sont des méthodes de résolution de problèmes d'optimisation combinatoire qui travaillent sur un ensemble de solutions en parallèle. Ces méthodes génèrent et manipulent une population de solutions pour explorer efficacement l'espace de recherche, avec l'objectif de trouver la meilleure solution possible.

Les métaheuristiques à population de solutions sont souvent plus efficaces pour résoudre des problèmes d'optimisation combinatoire qui ont plusieurs optima locaux ou qui nécessitent une exploration de l'espace de recherche plus étendue. Elles peuvent également être plus robustes, car elles sont moins susceptibles de rester coincées dans un minimum local.

Les exemples de métaheuristiques à population de solutions comprennent les algorithmes génétiques, les colonies de fourmis, les essaims de particules, la recherche en essaims, etc. Ces méthodes travaillent sur une population de solutions qui est générée et modifiée à chaque itération de l'algorithme. La population est souvent soumise à des opérateurs de variation, tels que la mutation, la recombinaison, l'insertion ou la suppression de solutions. Les solutions dans la population sont évaluées en fonction d'une fonction objectif, et les solutions de meilleure qualité sont sélectionnées pour former la population suivante.

Les métaheuristiques à population de solutions sont utilisées dans de nombreux domaines d'application, tels que l'ingénierie, la finance, la logistique, la biologie, etc. Elles ont largement contribué à améliorer la qualité des solutions pour de nombreux problèmes d'optimisation combinatoire et peuvent être utilisées pour résoudre des problèmes complexes qui ne peuvent pas être résolus de manière exacte en un temps raisonnable.

CONCLUSION

Afin de mieux appréhender la résolution de problème d'optimisation combinatoire nous allons tenter de résoudre par méta-heuristiques le jeu du SUDOKU. Le métaheuristique qu'on va utiliser est le Recuit Simulé (Simulated Annealing).

CHAPITRE 2 : LE RECUIT SIMULÉ

DÉFINITION DE L'ALGORITHME

Le recuit simulé (ou "simulated annealing" en anglais) est une méthode d'optimisation globale utilisée pour résoudre des problèmes d'optimisation combinatoire difficiles. Cette méthode est inspirée de la technique de recuit en métallurgie, qui consiste à refroidir lentement un matériau fondu pour lui donner une structure cristalline ordonnée.

En recuit simulé, on cherche à trouver la solution optimale d'un problème en explorant l'espace de recherche de manière probabiliste. La méthode est appelée "simulé" parce qu'elle simule un processus physique de recuit, où la température diminue lentement.

Le processus commence par une solution initiale, qui peut être choisie de manière aléatoire ou par une heuristique. Ensuite, à chaque itération, une nouvelle solution est générée en modifiant légèrement la solution courante. La qualité de la nouvelle solution est évaluée en termes d'une fonction objectif, et la nouvelle solution est acceptée ou rejetée en fonction d'une probabilité déterminée par la différence entre les valeurs de la fonction objectif de la nouvelle et de la solution courante, et la "température" actuelle du processus.

Au fur et à mesure que le processus avance, la "température" diminue, ce qui réduit la probabilité d'accepter des solutions moins bonnes. Cela permet au processus de se concentrer progressivement sur les régions les plus prometteuses de l'espace de recherche. Le processus se termine lorsque la "température" atteint une valeur suffisamment basse ou que le nombre maximal d'itérations est atteint.

Le recuit simulé est une méthode heuristique très utilisée pour résoudre des problèmes d'optimisation combinatoire difficiles, tels que le voyageur de commerce ou le placement de composants électroniques.

LES ÉTAPES DE BASE DU RECUIT SIMULÉ

1. **Initialisation** : La première étape consiste à initialiser la solution initiale du problème. Cette solution peut être choisie de manière aléatoire ou par une heuristique.
2. **Perturbation** : À chaque itération, une nouvelle solution est générée en modifiant légèrement la solution courante. Cette perturbation est souvent réalisée en utilisant un voisinage défini sur l'espace des solutions.
3. **Évaluation** : La qualité de la nouvelle solution est évaluée en termes d'une fonction objectif. Cette fonction objective peut être une fonction de coût ou une fonction de profit, selon la nature du problème à résoudre.
4. **Acceptation ou rejet de la nouvelle solution** : La nouvelle solution est ensuite acceptée ou rejetée en fonction d'une probabilité déterminée par la différence entre les valeurs de la fonction objectif de la

nouvelle et de la solution courante, et la "température" actuelle du processus. Si la nouvelle solution est meilleure que la solution courante, elle est toujours acceptée. Si la nouvelle solution est pire que la solution courante, elle peut être acceptée ou rejetée en fonction de la probabilité calculée.

5. **Réduction de la température** : Au fur et à mesure que le processus avance, la "température" diminue, ce qui réduit la probabilité d'accepter des solutions moins bonnes. Cette réduction de la température est souvent réalisée en utilisant une fonction de refroidissement qui permet de réduire la température de manière progressive.
6. **Critère d'arrêt** : Le processus se termine lorsque la "température" atteint une valeur suffisamment basse ou que le nombre maximal d'itérations est atteint.

Ces étapes sont répétées jusqu'à ce qu'une solution satisfaisante soit trouvée ou que le processus atteigne le critère d'arrêt. Le recuit simulé est une méthode heuristique très utile pour résoudre des problèmes d'optimisation combinatoire difficiles, car elle permet d'explorer l'espace de recherche de manière probabiliste et de trouver des solutions proches de l'optimum global.

INITIALISER LA SOLUTION INITIALE

Le choix de la solution initiale pour le recuit simulé peut avoir un impact significatif sur la performance globale de la méthode. Une solution initiale de bonne qualité peut permettre de réduire le temps de convergence et de trouver des solutions plus proches de l'optimum global. Il existe plusieurs méthodes pour générer une solution initiale pour le recuit simulé :

1. **Solution aléatoire** : La méthode la plus simple consiste à générer une solution initiale aléatoire en choisissant les valeurs de chaque variable de manière uniforme au hasard. Cette méthode peut être efficace pour les problèmes de petite taille, mais elle peut être inefficace pour les problèmes de grande taille.
2. **Solution basée sur des heuristiques** : Des heuristiques peuvent être utilisées pour générer une solution initiale basée sur des connaissances a priori ou des hypothèses sur la structure du problème. Par exemple, pour le problème du voyageur de commerce, une heuristique simple consiste à générer une solution initiale en parcourant les villes dans un ordre aléatoire.
3. **Solution gloutonne** : La méthode gloutonne peut également être utilisée pour générer une solution initiale en choisissant la meilleure option locale à chaque étape. Cette méthode peut être efficace si elle permet de trouver une solution de qualité acceptable rapidement.

Il est important de noter que le choix de la méthode pour générer la solution initiale dépend du problème spécifique et des contraintes de temps et de ressources. Il peut également être utile d'explorer plusieurs approches pour trouver la meilleure solution initiale avant d'appliquer la méthode de recuit simulé.

LA MÉTHODE GLOUTONNE

Voici les étapes pour appliquer la méthode gloutonne pour trouver une solution initiale pour le recuit simulé :

1. **Initialisation** : Choisissez une solution initiale aléatoire ou basée sur des informations a priori si disponibles.
2. **Évaluation** : Évaluez la qualité de la solution initiale en utilisant la fonction objective ou le critère de performance approprié.
3. **Itération** : Répétez les étapes suivantes jusqu'à ce qu'une solution satisfaisante soit trouvée :
 - a. **Génération de la solution voisine** : Générez une solution voisine à partir de la solution courante en effectuant une perturbation locale, telle que le déplacement d'un élément ou l'échange de deux éléments.
 - b. **Évaluation de la qualité de la solution voisine** : Évaluez la qualité de la solution voisine en utilisant la fonction objective ou le critère de performance approprié.
 - c. **Sélection de la meilleure solution voisine** : Sélectionnez la meilleure solution voisine parmi toutes les solutions voisines générées.
 - d. **Mise à jour de la solution courante** : Mettez à jour la solution courante en utilisant la meilleure solution voisine sélectionnée.
 - e. **Vérification de la condition d'arrêt** : Vérifiez si une condition d'arrêt est atteinte, telle qu'un nombre maximal d'itérations, une durée maximale ou une qualité minimale de la solution.
4. **Retourner la solution finale** : Retournez la dernière solution courante comme solution initiale pour la méthode de recuit simulé.

La méthode gloutonne peut être efficace pour trouver des solutions de qualité acceptable en un temps raisonnable, mais elle peut également conduire à des solutions sous-optimales si la recherche est bloquée dans une solution locale. Le recuit simulé peut être utilisé pour explorer l'espace de recherche de manière plus globale et trouver des solutions de meilleure qualité.

DÉFINIR UN VOISINNAGE

Dans le contexte du recuit simulé, un type de voisinage basé sur une recherche locale est une méthode pour générer des solutions voisines en utilisant une recherche locale avant d'appliquer la méthode de recuit simulé. La recherche locale est une méthode de résolution de problèmes d'optimisation qui explore localement l'espace de recherche autour d'une solution donnée pour trouver un optimum local.

Le type de voisinage basé sur une recherche locale pour le recuit simulé consiste à générer des solutions voisines en effectuant une recherche locale à partir de la solution courante. Cette méthode permet de générer des solutions de qualité supérieure en explorant plus en détail la région locale autour de la solution courante.

Le choix de la méthode de recherche locale dépend du problème spécifique et des contraintes de temps et de ressources. Il est également possible d'utiliser plusieurs méthodes de recherche locale pour générer des solutions voisines plus diversifiées et explorer plus efficacement l'espace de recherche.

En somme, l'utilisation d'un type de voisinage basé sur une recherche locale peut améliorer la performance globale du recuit simulé en générant des solutions de meilleure qualité et en explorant plus efficacement l'espace de recherche.

SUDOKU ET RECUIT SIMULÉ

PRÉSENTATION DU JEU SUDOKU

Le Sudoku est un puzzle de chiffres qui se compose d'une grille de 9x9 cases. La grille est divisée en neuf sous-grilles de 3x3 cases, appelées "régions". Le but du Sudoku est de remplir chaque case vide de la grille avec un chiffre de 1 à 9 de sorte que chaque colonne, chaque ligne et chaque région contienne tous les chiffres de 1 à 9 exactement une fois.

Voici les règles détaillées du Sudoku :

- Une grille de Sudoku est constituée de neuf colonnes et neuf lignes, ce qui donne 81 cases au total.
- Chaque ligne doit contenir tous les chiffres de 1 à 9 exactement une fois.
- Chaque colonne doit également contenir tous les chiffres de 1 à 9 exactement une fois.
- Chaque région de 3x3 cases doit également contenir tous les chiffres de 1 à 9 exactement une fois.
- Le Sudoku commence avec une grille partielle qui contient déjà quelques chiffres placés dans des cases. Le but est de remplir les cases vides avec des chiffres manquants.
- Le chiffre zéro est souvent utilisé pour représenter une case vide.
- Pour résoudre le Sudoku, on commence par examiner les chiffres déjà placés dans la grille et on cherche des indices sur lesquels chiffres peuvent être placés dans les cases vides.
- Évitez de placer le même chiffre dans une même ligne, colonne ou région.
- Utilisez la méthode d'élimination pour trouver les chiffres manquants dans les cases vides en fonction des chiffres déjà présents dans la grille.
- Lorsque vous remplissez une case avec un chiffre, vérifiez que cette case ne viole pas les règles du Sudoku en examinant les chiffres présents dans les mêmes lignes, colonnes et régions que cette case.
- Continuez à remplir les cases jusqu'à ce que toutes les cases soient remplies et que toutes les règles du Sudoku soient respectées.

Le Sudoku est un jeu qui demande de la patience et de la logique pour résoudre la grille en respectant les règles établies.

ETAPES DE RÉOLUTION DU SUDOKU PAR RECUIT SIMULÉ

Afin de résoudre un jeu de Sudoku comme un problème d'optimisation combinatoire en utilisant l'algorithme méta-heuristique du recuit simulé, il faut suivre certaines étapes :

1. Représenter le plateau de jeu comme une matrice 9x9, les cellules vides étant représentées comme des variables.
2. Définir la fonction objectif : l'objectif est de minimiser le nombre de contraintes violées, de sorte que toutes les lignes, colonnes et sous-grilles 3x3 contiennent tous les chiffres de 1 à 9 sans répétition.
3. Définissez la température initiale (T) et le programme de refroidissement pour l'algorithme de recuit simulé.
4. Générez une solution aléatoire en remplissant les cellules vides avec des valeurs qui respectent les contraintes grâce à une méthode gloutonne.
5. Évaluez la fonction objectif pour la solution actuelle.
6. Générer une nouvelle solution en apportant un petit changement aléatoire à la solution actuelle.
7. Évaluez la fonction objectif de la nouvelle solution
8. Calculez la probabilité d'acceptation à l'aide de la formule : $P = e^{\left(-\left(\frac{\Delta E}{T}\right)\right)}$, où ΔE est la différence entre les valeurs de la fonction objectif pour la solution actuelle et la nouvelle solution.
9. Si la probabilité d'acceptation est supérieure à un nombre aléatoire compris entre 0 et 1, acceptez la nouvelle solution comme solution actuelle.
10. Répétez les étapes 6 à 9 jusqu'à ce que la température ait refroidi jusqu'à un critère d'arrêt ou qu'une solution soit trouvée où la fonction objectif est minimisée.
11. Vérifiez la solution pour vous assurer qu'elle est correcte en vérifiant que toutes les contraintes sont satisfaites.
12. Si la solution est correcte, le jeu de Sudoku est résolu. Si la solution est incorrecte, il faut répéter le processus à partir de l'étape 4 avec un point de départ ou une température différents.

CHAPITRE 3 : RÉOLUTION DU JEU SUDOKU

Dans ce chapitre, nous allons présenter la résolution du jeu sudoku avec l'algorithme du recuit simulé à l'aide d'un programme en langage python.

Ci-joint une grille de sudoku de niveau moyen que nous allons résoudre :

2		7		1		6		5
	9	5	6			4	2	3
8					4			
7						2	6	
			7	2	6		1	8
	2			9			3	
	5					3		7
6		4	5	3				
	1	2		7	9			

Figure 3: Sudoku à résoudre

1^{ER} ÉTAPE : AFFICHAGE DE LA GRILLE DU SUDOKU

Cette ligne de code crée un tableau numpy sudoku à partir d'une chaîne de caractères sudokuDeDepart qui représente l'état initial d'une grille de Sudoku.

La méthode split() est utilisée pour diviser la chaîne de caractères en une liste de chaînes, chaque chaîne représentant une ligne de la grille.

Ensuite, une liste en compréhension est utilisée pour créer une liste de listes, où chaque élément de la liste est converti en un entier à l'aide de la fonction int(). Cela signifie que chaque chiffre de la chaîne est converti en un nombre entier.

Enfin, cette liste de listes est passée à la fonction np.array(), qui crée un tableau numpy 2D à partir des éléments de la liste de listes. Le tableau numpy résultant représente la grille de Sudoku initiale, où chaque cellule est représentée par un nombre entier.

```
#!/ crée un tableau numpy sudoku à partir d'une chaîne de caractères sudokuDeDepart qui représente l'état initial d'une grille de Sudoku
sudoku = np.array([[int(i) for i in ligne] for ligne in sudokuDeDepart.split()])
```

Figure 4: convertir d'une chaîne de caractère à un tableau sudoku

Cette fonction, afficher_sudoku(sudoku), prend en entrée un tableau à deux dimensions de 9x9 (c'est-à-dire une liste de listes) et l'imprime dans un format qui représente une grille de Sudoku.

La fonction commence par imprimer un caractère de saut de ligne pour commencer sur une nouvelle ligne. Ensuite, elle itère à travers chaque ligne (c'est-à-dire chaque sous-liste) du sudoku d'entrée en utilisant une boucle

for avec la fonction range(). À l'intérieur de cette boucle, la fonction crée une variable de chaîne vide appelée "ligne" qui sera utilisée pour construire la sortie imprimée pour chaque ligne.

Si l'indice i est soit 3 ou 6, la fonction imprime une ligne horizontale pour séparer chacune des trois boîtes de la grille. Cela est réalisé en utilisant une instruction if et la fonction print() avec une chaîne qui contient 21 caractères "-" pour former une ligne horizontale.

Pour chaque cellule dans la ligne courante, la fonction itère ensuite à travers chaque colonne (c'est-à-dire chaque élément dans la sous-liste courante) en utilisant une autre boucle for avec la fonction range().

Si l'indice j est soit 3 ou 6, la fonction ajoute un caractère "|" à la variable "ligne" pour séparer les cellules à l'intérieur de chaque boîte. La fonction str() est utilisée pour convertir la valeur à chaque position dans le tableau sudoku en une chaîne de caractères afin qu'elle puisse être concaténée avec la variable "ligne".

Enfin, la fonction utilise la fonction print() pour afficher la variable "ligne", qui contient la chaîne formatée pour la ligne actuelle de la grille de Sudoku. Cela est répété pour chaque ligne dans le tableau sudoku jusqu'à ce que toute la grille soit imprimée.

```
def afficher_sudoku(sudoku):
    print("\n")
    for i in range(len(sudoku)):
        ligne = ""
        if i == 3 or i == 6:
            print("-----")
        for j in range(len(sudoku[i])):
            if j == 3 or j == 6:
                ligne += "| "
            ligne += str(sudoku[i,j])+" "
        print(ligne)
```

Figure 5: affichage de la grille du sudoku

2^{ÈME} ÉTAPE : CONVERSION DES CASES PRÉDEFINIS EN 1

La fonction ValSudokuFixe (sudoku_fixe) prend en entrée un tableau NumPy de 9x9, sudoku_fixe, représentant une grille de Sudoku partiellement remplie, où les valeurs remplies sont représentées par des entiers de 1 à 9 et les cellules vides sont représentées par des 0.

La fonction convertit toutes les valeurs remplies en 1, laissant les cellules vides comme des 0. Cette conversion permet au programme de facilement distinguer les cellules remplies et vides lors de la vérification de la validité de la solution.

La fonction utilise une boucle for imbriquée pour itérer à travers chaque cellule de la grille de Sudoku. Si la cellule n'est pas vide (c'est-à-dire qu'elle contient un entier différent de zéro), la fonction définit la valeur de la cellule à 1. La fonction retourne ensuite la grille de Sudoku modifiée.

```
def ValSudokuFixe(sudoku_fixe):
    for i in range (0,9):
        for j in range (0,9):
            if sudoku_fixe[i,j] != 0:
                sudoku_fixe[i,j] = 1

    return(sudoku_fixe)
```

Figure 6: convertir les valeurs remplies en 1

3^{ÈME} ÉTAPE : DÉFINIR LA FONCTION OBJECTIF

Cette fonction prend en entrée trois arguments : ligne, colonne et sudoku, et retourne le nombre d'erreurs dans une ligne et une colonne donnée d'une grille de Sudoku.

La fonction calcule d'abord le nombre de valeurs uniques dans la colonne donnée de la matrice de sudoku en utilisant la fonction NumPy `np.unique(sudoku[:, colonne])`. Elle soustrait ensuite cette valeur de 9 (le nombre total de valeurs possibles dans une ligne ou une colonne de Sudoku) pour obtenir le nombre de valeurs répétées dans la colonne.

La fonction fait la même chose pour la ligne donnée de la matrice de sudoku en utilisant `np.unique(sudoku[ligne, :])`, et ajoute le nombre de valeurs répétées dans la ligne au nombre de valeurs répétées dans la colonne pour obtenir le nombre total d'erreurs dans la ligne et la colonne données.

En résumé, cette fonction calcule le nombre d'erreurs dans une ligne et une colonne donnée d'une grille de Sudoku, et peut être utile pour identifier les zones qui doivent être corrigées pour résoudre la grille.

```
def CalculerNbrErreursLigneColonne(ligne, colonne, sudoku):
    Nbr_erreurs = (9 - len(np.unique(sudoku[:,colonne]))) + (9 - len(np.unique(sudoku[ligne,:])))
    return(Nbr_erreurs)
```

Figure 7: Calculer le nombre d'erreurs par ligne & colonne

Ce code est une fonction qui calcule le nombre total d'erreurs dans une grille de Sudoku donnée.

La fonction parcourt chaque ligne et colonne de la grille de Sudoku à l'aide d'une boucle for, et appelle la fonction « CalculerNbrErreursLigneColonne » avec les indices de ligne et de colonne actuels pour calculer le nombre d'erreurs dans cette ligne et cette colonne.

Ensuite, la fonction ajoute ce nombre à un total cumulé stocké dans la variable « Nbr_erreurs ».

Enfin, la fonction retourne le nombre total d'erreurs calculées pour toutes les lignes et colonnes de la grille de Sudoku.

```
def CalculerNbrErreurs(sudoku):
    Nbr_erreurs = 0
    for i in range (0,9):
        Nbr_erreurs += CalculerNbrErreursLigneColonne(i ,i ,sudoku)
    return(Nbr_erreurs)
```

Figure 8: Calculer le nombre total d'erreurs

4^{ÈME} ÉTAPE : GÉNÉRER UNE SOLUTION INITIALE

La fonction « CreerListe3x3Blocs » crée une liste de listes où chaque liste interne contient les indices de ligne et de colonne d'un bloc 3x3 dans une grille de Sudoku. La boucle externe itère sur les 9 lignes de la grille, et pour chaque ligne, elle calcule les indices de ligne des trois blocs 3x3 qui chevauchent cette ligne en utilisant l'opérateur % et la fonction `math.trunc()`.

La boucle interne itère sur les cellules de chaque bloc et ajoute leurs indices de ligne et de colonne à la liste temporaire (`tmpListe`).

Enfin, la `tmpListe` est ajoutée à la liste finale des blocs (`ListeDeBlocsFinal`), et la boucle externe continue sur la ligne suivante, répétant le processus jusqu'à ce que toutes les 9 lignes aient été traitées.

La liste finale des blocs (`ListeDeBlocsFinal`) contient 9 listes internes, chacune correspondant à un bloc 3x3 dans la grille.

```
def CreerListe3x3Blocs ():
    ListeDeBlocsFinal = []
    for r in range (0,9):
        tmpListe = []
        block1 = [i + 3*((r)%3) for i in range(0,3)]
        block2 = [i + 3*math.trunc((r)/3) for i in range(0,3)]
        for x in block1:
            for y in block2:
                tmpListe.append([x,y])
        ListeDeBlocsFinal.append(tmpListe)
    return(ListeDeBlocsFinal)
```

Figure 9: Créer une liste de blocs

```
block1 = [i + 3*((r)%3) for i in range(0,3)]
```

La ligne crée une liste de 3 éléments contenant les indices des colonnes pour une rangée spécifique de la grille de sudoku.

- `range(0,3)` crée une liste de 3 éléments : [0, 1, 2].
- `((r)%3)` calcule le reste de la division de `r` par 3. Cela permet de déterminer à quel bloc de 3 colonnes (dans une rangée) l'indice actuel doit être ajouté. Il y a 3 blocs de colonnes dans une rangée de la grille de sudoku, chacun contenant 3 colonnes. Par conséquent, le reste de la division par 3 donne 0, 1 ou 2, en fonction de la position de la rangée dans le bloc de 3x3.
- `i + 3*((r)%3)` ajoute le reste de la division par 3 à l'indice de colonne actuel, ce qui permet de déterminer l'emplacement exact de la colonne dans la rangée.

En fin de compte, cette ligne de code crée une liste de 3 éléments contenant les indices des colonnes pour une rangée spécifique, en se basant sur la position de la rangée dans le bloc de 3x3 correspondant.

```
block2 = [i + 3*math.trunc((r)/3) for i in range(0,3)]
```

La ligne crée une liste de 3 éléments contenant les positions des colonnes pour un bloc 3x3 dans la matrice.

Plus précisément, `math.trunc((r)/3)` renvoie la partie entière de la division de `r` par 3, qui est utilisée pour déterminer la "ligne" du bloc dans la matrice, tandis que `i + 3 * math.trunc((r)/3)` renvoie la position de la colonne pour un élément `i` donné dans la liste `[0, 1, 2]`.

Ainsi, la boucle `for i in range(0,3)` crée une liste contenant les positions de colonnes relatives à un bloc 3x3 en parcourant les éléments `[0, 1, 2]` et en leur ajoutant un multiple de 3, correspondant à la position de la ligne du bloc dans la matrice.

```
for x in block1:
    for y in block2:
        tmpListe.append([x,y])
    ListeDeBlocsFinal.append(tmpListe)
return(ListeDeBlocsFinal)
```

Ces lignes de code permettent de générer des listes de blocs de 3x3 dans la grille Sudoku.

La boucle `for "x in block1"` permet de parcourir les éléments de la liste `block1`, qui contient les coordonnées des colonnes dans la grille pour chaque bloc 3x3.

La boucle `for "y in block2"` permet de parcourir les éléments de la liste `block2`, qui contient les coordonnées des lignes dans la grille pour chaque bloc 3x3.

La ligne `" tmpListe.append([x,y])"` permet d'ajouter les coordonnées de chaque case de la grille correspondant à un bloc 3x3 dans une liste temporaire `tmpList`.

Enfin, la liste temporaire est ajoutée à la liste finale `ListeDeBlocsFinal`, qui contient toutes les listes de blocs 3x3.

REEMPLISSAGE

Cette partie de code vérifie si la case du sudoku située aux coordonnées indiquées par la variable `"box"` est vide, c'est-à-dire si elle contient la valeur 0. Si tel est le cas, la variable `"BlocCourant"` est créée pour stocker les valeurs contenues dans le bloc 3x3 correspondant à cette case. Ensuite, la case est remplie avec une valeur aléatoire choisie parmi les chiffres de 1 à 9 qui ne sont pas déjà présents dans le bloc 3x3 en question, grâce à la fonction `"choice()"` de la bibliothèque `random`.

```
def RemplissageAleatoire3x3blocs(sudoku, listDeBlocs):
    for block in listDeBlocs:
        for box in block:
            if sudoku[box[0],box[1]] == 0:
                BlocCourant = sudoku[block[0][0]:(block[-1][0]+1),block[0][1]:(block[-1][1]+1)]
                sudoku[box[0],box[1]] = choice([i for i in range(1,10) if i not in BlocCourant])
    return sudoku
```

Figure 10: Remplissage Aléatoire du bloc

```
BlocCourant = sudoku[block[0][0]:(block[-1][0]+1),block[0][1]:(block[-1][1]+1)]
```

La ligne de code permet de créer une sous-matrice qui représente le bloc de la grille 3x3 qui contient la case `box` :

- `block[0][0]` correspond à la coordonnée en ligne de la première case du bloc (coin supérieur gauche)
- `block[-1][0]` correspond à la coordonnée en ligne de la dernière case du bloc (coin inférieur droit)

- `block[0][1]` correspond à la coordonnée en colonne de la première case du bloc (coin supérieur gauche)
- `block[-1][1]` correspond à la coordonnée en colonne de la dernière case du bloc (coin inférieur droit)

Ainsi, `BlocCourant` est une sous matrice qui couvre toutes les cases du bloc qui contient la case `box`. Cette sous-matrice est extraite de la grille de sudoku « `sudoku` ».

```
sudoku[box[0],box[1]] = choice([i for i in range(1,10) if i not in BlocCourant])
```

Cette ligne de code modifie une case vide (0) dans la grille sudoku avec une valeur aléatoire qui ne se trouve pas déjà dans le bloc 3x3 auquel la case appartient.

`sudoku[box[0],box[1]]` sélectionne la case vide actuelle dans la grille sudoku à l'emplacement de la ligne `box[0]` et la colonne `box[1]`.

`choice()` est une fonction de la bibliothèque `random` qui choisit un élément au hasard dans une liste. Dans ce cas, la liste contient toutes les valeurs possibles pour une case de Sudoku, sauf celles qui sont déjà présentes dans le bloc actuel.

La liste est créée par la compréhension de liste `[i for i in range(1,10) if i not in BlocCourant]`, qui itère sur les entiers de 1 à 9 et ne sélectionne que ceux qui ne se trouvent pas déjà dans le bloc actuel.

`BlocCourant` est le sous-tableau 3x3 de la grille sudoku qui contient la case actuelle.

Pour déterminer `BlocCourant`, nous utilisons les coordonnées de la première case `block[0][0]` et `block[0][1]` du bloc actuel, ainsi que les coordonnées de la dernière case `block[-1][0]` et `block[-1][1]` du bloc actuel, et nous les utilisons pour extraire le sous-tableau 3x3 correspondant.

5^{EME} ÉTAPE : DÉFINIR UN VOISINAGE

La fonction « `SommeDUnBloc` » prend en entrée une grille de sudoku « `sudoku` » et une liste de coordonnées qui représentent un bloc de 3x3. La fonction parcourt chaque coordonnée dans la liste et ajoute la valeur de la case correspondante à une somme finale. Enfin, la somme finale est retournée. En d'autres termes, la fonction calcule la somme des valeurs des cases dans un bloc de 3x3 spécifique de la grille de sudoku.

```
def SommeDUnBloc (sudoku, UnBloc):
    SommeFinal = 0
    for box in UnBloc:
        SommeFinal += sudoku[box[0], box[1]]
    return(SommeFinal)
```

Figure 11: Calculer la somme des valeurs d'un bloc

```
SommeFinal = 0
```

On initialise la variable `SommeFinal` à 0, qui va contenir la somme des chiffres des cases du bloc.

```
for box in UnBloc :
```

On parcourt chaque case du bloc en utilisant une boucle `for` avec la variable `box` qui prend successivement les coordonnées de chaque case.

```
SommeFinal += sudoku[box[0], box[1]]
```

Pour chaque case du bloc, on ajoute sa valeur à la variable SommeFinal. Les coordonnées de la case sont récupérées depuis box qui contient un tuple de deux valeurs : la ligne et la colonne de la case.

```
return (SommeFinal)
```

On retourne la somme finale des chiffres des cases du bloc

CHOIX ALEATOIRE DE BLOCS

La fonction « DeuxCasesAleatoireDansUnBloc » prend en entrée une grille de sudoku partiellement remplie (certaines cases sont déjà remplies et d'autres sont vides) ainsi qu'un bloc de 9 cases à l'intérieur de cette grille. Cette fonction retourne un tuple contenant deux cases aléatoires différentes appartenant au bloc donné en entrée.

Le code de la fonction utilise une boucle while (1) qui permet de continuer à chercher deux cases aléatoires différentes tant qu'aucune paire valide n'est trouvée.

La première case est sélectionnée aléatoirement en utilisant la fonction random.choice(), tandis que la deuxième case est sélectionnée à partir de la liste des cases du bloc qui ne sont pas égales à la première case, en utilisant la fonction choice() de la bibliothèque numpy.

Ensuite, la fonction vérifie que les deux cases sélectionnées ne sont pas déjà remplies avec une valeur fixe de la grille (représentée par la valeur 1), afin de garantir que ces deux cases peuvent être échangées sans violer les règles du sudoku.

Si les deux cases sélectionnées ne sont pas déjà remplies, la fonction retourne un tuple contenant ces deux cases.

```
def DeuxCasesAleatoireDansUnBloc(SudokuCorrige, block):
    while (1):
        PremierBox = random.choice(block)
        DeuxiemeBox = choice([box for box in block if box is not PremierBox ])

        if SudokuCorrige[PremierBox[0], PremierBox[1]] != 1 and SudokuCorrige[DeuxiemeBox[0], DeuxiemeBox[1]] != 1:
            return([PremierBox, DeuxiemeBox])
```

Figure 12: Choisir 2 cases aléatoire dans un bloc

```
while (1):
```

On entre dans une boucle infinie pour trouver deux cases aléatoires dans le bloc qui ne sont pas déjà fixes (valeur de la matrice initiale).

```
PremierBox = random.choice(block)
```

On sélectionne une case aléatoire dans le bloc à l'aide de la fonction random.choice().

```
DeuxiemeBox = choice([box for box in block if box is not PremierBox ])
```

On sélectionne une autre case aléatoire qui n'est pas la première case sélectionnée à l'aide de la fonction choice(), qui prend une liste en entrée.

La liste est créée à partir des cases du bloc qui ne sont pas égales à la première case sélectionnée.

```
if fixedSudoku[firstBox[0], firstBox[1]] != 1 and fixedSudoku[secondBox[0],
secondBox[1]] != 1:
```


On vérifie si les deux cases sélectionnées ne sont pas déjà fixes (avec une valeur égale à 1) dans la matrice de Sudoku.

```
return([PremierBox, DeuxiemeBox])
```

Si les deux cases sélectionnées ne sont pas déjà fixes, la fonction renvoie une liste contenant ces deux cases.

RETOURNER DES BOXES

La fonction `RetournerBoxes` prend en entrée un sudoku et deux coordonnées `boxesARetourner` correspondant à deux cases qui seront échangées.

La première ligne crée une copie du sudoku initial, afin de ne pas modifier l'original et de travailler sur une copie temporaire.

Ensuite, la deuxième ligne sauvegarde la valeur de la première case `boxesARetourner [0]` dans une variable temporaire `placeReserve`.

La troisième ligne écrase la valeur de la première case avec celle de la deuxième case `boxesARetourner [1]`.

La quatrième ligne écrase la valeur de la deuxième case avec celle qui a été sauvegardée dans `placeReserve`, donc la valeur initiale de la première case.

Enfin, la fonction renvoie la copie modifiée du sudoku initial `SudokuPropose`.

```
def RetournerBoxes(sudoku, boxesARetourner):
    SudokuPropose = np.copy(sudoku)
    PlaceReserve = SudokuPropose[boxesARetourner[0][0], boxesARetourner[0][1]]
    SudokuPropose[boxesARetourner[0][0], boxesARetourner[0][1]] = SudokuPropose[boxesARetourner[1][0], boxesARetourner[1][1]]
    SudokuPropose[boxesARetourner[1][0], boxesARetourner[1][1]] = PlaceReserve
    return (SudokuPropose)
```

Figure 13: Retourner deux boxes

```
SudokuPropose = np.copy(sudoku)
```

Crée une copie du sudoku original et la stocke dans la variable "SudokuPropose".

```
placeReserve = SudokuPropose[boxesARetourner[0][0], boxesARetourner[0][1]]
```

Stocke la valeur de la première case à inverser dans la variable "placeReserve".

```
SudokuPropose[boxesARetourner[0][0], boxesARetourner[0][1]] =
```

```
SudokuPropose[boxesARetourner[1][0], boxesARetourner[1][1]]
```

Remplace la valeur de la première case à inverser avec la valeur de la deuxième case à inverser.

```
SudokuPropose[boxesARetourner [1][0], boxesARetourner [1][1]] = placeReserve
```

Remplace la valeur de la deuxième case à inverser avec la valeur stockée dans "placeReserve".

```
return (SudokuPropose)
```

Retourne le sudoku modifié avec les deux cases inversées.

PROPOSER UN NOUVEAU ÉTAT

La fonction "EtatPropose" génère une proposition d'état pour le sudoku en entrée en effectuant des changements aléatoires à deux cases d'un bloc choisi au hasard.

Elle prend en entrée trois paramètres : "sudoku" qui représente l'état courant du sudoku, "SudokuFixe" qui est un sudoku avec les chiffres fixes qui ne peuvent pas être modifiés, et "listDeBlocs" qui est une liste des blocs du sudoku.

Elle commence par choisir un bloc aléatoire à partir de "listDeBlocs".

Si la somme des chiffres fixes dans le bloc choisi est supérieure à 6, cela signifie que le bloc est déjà rempli en grande partie et qu'il est peu probable que la proposition de changement mène à une solution valide.

Dans ce cas, la fonction retourne simplement l'état de sudoku d'entrée, ainsi que deux valeurs de sortie "1" pour signaler qu'aucun changement n'a été effectué.

Si la somme des chiffres fixes est inférieure ou égale à 6, la fonction appelle la fonction "DeuxCasesAleatoireDansUnBloc" pour sélectionner deux cases aléatoires à partir du bloc choisi.

Elle appelle ensuite la fonction "RetournerBoxes" pour permuter les valeurs des deux cases sélectionnées et stocke le nouvel état du sudoku proposé dans la variable "SudokuPropose".

Enfin, la fonction renvoie le nouvel état de sudoku proposé, ainsi que les coordonnées des deux cases modifiées.

```
def EtatPropose (sudoku, SudokuFixe, listDeBlocs):
    BlocAleatoire = random.choice(listDeBlocs)

    if SommeDUnBloc(SudokuFixe, BlocAleatoire) > 6:
        return(sudoku, 1, 1)
    boxesARetourner = DeuxCasesAleatoireDansUnBloc(SudokuFixe, BlocAleatoire)
    SudokuPropose = RetournerBoxes(sudoku, boxesARetourner)
    return([SudokuPropose, boxesARetourner])
```

Figure 14: proposition d'un nouvel état

```
BlocAleatoire = random.choice(listDeBlocs)
```

La première instruction de la fonction est de choisir un bloc aléatoire parmi ceux de listDeBlocs en utilisant la fonction random.choice(listDeBlocs) et stocke le bloc choisi dans la variable BlocAleatoire.

```
if SommeDUnBloc(SudokuFixe, BlocAleatoire) > 6:
    return(sudoku, 1, 1)
```

La condition if SommeDUnBloc (SudokuFixe, BlocAleatoire) > 6 : vérifie si le nombre de cases déjà remplies dans le bloc choisi est supérieur à 6 en appelant la fonction SommeDUnBloc qui retourne la somme des valeurs des cases du bloc donné.

```
boxesARetourner = DeuxCasesAleatoireDansUnBloc(SudokuFixe, BlocAleatoire)
```

Si la condition précédente est fausse, la fonction appelle la fonction DeuxCasesAleatoireDansUnBloc (SudokuFixe, BlocAleatoire) pour obtenir les coordonnées de deux cases aléatoires à échanger dans le bloc choisi.

SudokuPropose = RetournerBoxes(sudoku, boxesARetourner)

La fonction **RetournerBoxes** (sudoku, boxesARetourner) échange les valeurs des deux cases sélectionnées et stocke le nouveau tableau de sudoku proposé dans la variable **SudokuPropose**.

return([SudokuPropose, boxesARetourner])

La fonction retourne une liste contenant **SudokuPropose** et les coordonnées des cases échangées **boxesARetourner**.

CHOISIR UN NOUVEL ÉTAT

Cette fonction sert à choisir un nouvel état pour le sudoku en utilisant l'algorithme de recuit simulé

La fonction prend en entrée le sudoku actuel (**SudokuCourant**), le sudoku fixe (**SudokuFixe**), la liste des blocs du sudoku (**listDeBlocs**) et le paramètre **temperature** qui détermine l'amplitude des changements proposés.

La variable «proposition» stocke le nouvel état proposé par la fonction «**EtatPropose**» qui est appelée avec les arguments «**SudokuCourant**», «**SudokuFixe**» et «**listDeBlocs**».

La variable **nouveauSudoku** contient la proposition de la fonction **EtatPropose** pour le nouveau sudoku.

La variable **boxesAVerifier** contient les deux cases du sudoku qui ont été changées par la fonction **EtatPropose**.

La variable **CoutCourant** stocke le coût actuel de l'état courant. Pour cela, la fonction **CalculerNbrErreursLigneColonne** est appelée avec les coordonnées des cases à vérifier.

La variable **nouvelCout** stocke le coût de la nouvelle proposition. Encore une fois, la fonction **CalculerNbrErreursLigneColonne** est appelée avec les coordonnées des cases à vérifier.

La variable **DifferenceDeCout** contient la différence de coût entre l'état actuel et le nouvel état proposé.

La variable **rho** calcule la probabilité d'accepter la nouvelle proposition en fonction de la différence de coût et du paramètre **sigma**.

Si la probabilité d'accepter la proposition est supérieure à un nombre aléatoire généré par la fonction **np.random.uniform**, alors la nouvelle proposition est acceptée et renvoyée avec la différence de coût.

Si la proposition est rejetée, l'état courant est renvoyé avec un coût de 0.

```
def ChoisirNouvelEtat (SudokuCourant, SudokuFixe, listDeBlocs, temperature):
    proposition = EtatPropose(SudokuCourant, SudokuFixe, listDeBlocs)
    nouveauSudoku = proposition[0]
    boxesAVerifier = proposition[1]
    CoutCourant = CalculerNbrErreursLigneColonne(boxesAVerifier[0][0], boxesAVerifier[0][1], SudokuCourant) + CalculerNbrErreursLigneColonne(boxesAVerifier[1][0], boxesAVerifier[1][1], SudokuCourant)
    nouvelCout = CalculerNbrErreursLigneColonne(boxesAVerifier[0][0], boxesAVerifier[0][1], nouveauSudoku) + CalculerNbrErreursLigneColonne(boxesAVerifier[1][0], boxesAVerifier[1][1], nouveauSudoku)
    # nouvelCout = CalculerNbrErreurs(nouveauSudoku)
    DifferenceDeCout = nouvelCout - CoutCourant
    rho = math.exp(-DifferenceDeCout/temperature)
    if(np.random.uniform(1,0,1) < rho):
        return([nouveauSudoku, DifferenceDeCout])
    return([SudokuCourant, 0])
```

Figure 15: Choisir un nouvel état

proposition = EtatPropose(SudokuCourant, SudokuFixe, listDeBlocs)

proposition est une variable qui contient un nouvel état proposé pour le Sudoku courant. Elle est obtenue en appelant la fonction **EtatPropose** avec **SudokuCourant**, **SudokuFixe** et **listDeBlocs** comme arguments.

nouveauSudoku = proposition[0]

boxesAVerifier = proposition[1]

nouveauSudoku est une variable qui contient le nouvel état proposé pour le Sudoku courant qui est retourné par la fonction EtatPropose. La variable boxesAVerifier contient les cases du Sudoku qui doivent être vérifiées pour calculer le coût de l'état.

```
CoutCourant = CalculerNbrErreursLigneColonne (boxesAVerifier[0][0],  
boxesAVerifier[0][1], SudokuCourant) +  
CalculerNbrErreursLigneColonne (boxesAVerifier[1][0], boxesAVerifier[1][1],  
SudokuCourant)  
nouvelCout=CalculerNbrErreursLigneColonne (boxesAVerifier[0][0],boxesAVerifier[0][1],  
nouveauSudoku)+CalculerNbrErreursLigneColonne (boxesAVerifier[1][0],boxesAVerifier[1][1], nouveauSudoku)
```

CoutCourant est une variable qui contient le coût de l'état courant, c'est-à-dire le nombre d'erreurs dans les lignes et les colonnes qui contiennent les cases spécifiées dans boxesAVerifier.

nouvelCout est une variable qui contient le coût du nouvel état proposé, c'est-à-dire le nombre d'erreurs dans les lignes et les colonnes qui contiennent les cases spécifiées dans boxesAVerifier.

```
DifferenceDeCout = nouvelCout - CoutCourant  
rho = math.exp(-DifferenceDeCout/temperature)
```

DifferenceDeCout est une variable qui contient la différence de coût entre l'état courant et le nouvel état proposé.

rho est une variable qui contient la probabilité d'accepter le nouvel état proposé, calculée en utilisant la formule de Metropolis-Hastings.

La formule de Metropolis-Hastings est une méthode probabiliste utilisée dans l'algorithme de recuit simulé pour accepter ou rejeter un nouvel état proposé en fonction de la différence d'énergie entre l'état actuel et le nouvel état proposé. Elle est utilisée pour éviter de rester bloqué dans un minimum local de l'espace de recherche. La formule est la suivante :

$$p = \min(1, e^{(-\Delta E / T)})$$

où p est la probabilité d'accepter le nouvel état proposé, ΔE est la différence d'énergie entre l'état actuel et le nouvel état proposé, et T est la température courante de l'algorithme de recuit simulé :

- Si ΔE est négatif, cela signifie que le nouvel état proposé est meilleur que l'état actuel et donc qu'il sera toujours accepté.

- Si ΔE est positif, la probabilité d'acceptation diminue avec l'augmentation de T et donc la probabilité d'accepter le nouvel état proposé diminue également.

```
if(np.random.uniform(1,0,1) < rho):  
    return ([nouveauSudoku, DifferenceDeCout])  
return ([SudokuCourant, 0])
```

Si un nombre aléatoire entre 0 et 1 est inférieur à ρ , la fonction retourne le nouvel état proposé et la différence de coût entre l'état courant et le nouvel état proposé. Sinon, la fonction retourne l'état courant et 0 comme différence de coût.

CHOISIR UN NOMBRE D'ITERATION

La fonction `ChoisirNombreDIterations` prend en entrée une grille de sudoku `sudoku_fixe`. Elle initialise le nombre d'itérations à 0.

Ensuite, elle utilise deux boucles imbriquées pour parcourir chaque case de la grille.

Si une case contient une valeur différente de zéro, cela signifie qu'elle est fixée et ne peut pas être modifiée lors de la résolution du puzzle.

Par conséquent, le nombre d'itérations est augmenté de 1.

En fin de compte, la fonction renvoie le nombre total d'itérations nécessaires pour résoudre le puzzle.

La fonction `ChoisirNombreDIterations` prend en entrée une grille de sudoku partiellement remplie `sudoku_fixe` et retourne le nombre d'itérations nécessaires pour remplir la grille.

```
def ChoisirNombreDIterations(sudoku_fixe):
    nombreDIterations = 0
    for i in range (0,9):
        for j in range (0,9):
            if sudoku_fixe[i,j] != 0:
                nombreDIterations += 1
    return nombreDIterations
```

Figure 16: Choisir le nombre d'itérations

```
nombreDIterations = 0
```

Initialise la variable `numberOfIterations` à 0.

```
for i in range (0,9):
```

```
    for j in range (0,9):
```

Boucle sur les indices de ligne de la grille de sudoku (de 0 à 8 inclus).

Boucle sur les indices de colonne de la grille de sudoku (de 0 à 8 inclus).

```
if sudoku_fixe [i,j] != 0:
```

```
    nombreDIterations += 1
```

Vérifie si la case correspondant à l'indice (i, j) de la grille de sudoku n'est pas vide.

Si la case n'est pas vide, incrémente le compteur `nombreDIterations`.

```
return nombreDIterations
```

Retourne le nombre total de cases non vides dans la grille de sudoku.

CALCULER LA TEMPERATURE INITIALE

Cette fonction calcule la valeur initiale de la température utilisé dans l'algorithme de recuit simulé.

La fonction prend en entrée trois paramètres : le tableau de sudoku courant (sudoku), la version fixe du tableau (SudokuFixe) et une liste de blocs (listDeBlocs).

Une liste vide appelée listDeDifferences est créée pour stocker les différences dans le nombre d'erreurs entre les itérations consécutives.

Une copie temporaire du tableau de sudoku est créée et assignée à la variable tmpSudoku.

Une boucle est démarrée qui itère sur les nombres de 1 à 9.

Dans la boucle, la variable tmpSudoku est mise à jour vers un nouvel état proposé en utilisant la fonction EtatPropose, en passant le tmpSudoku courant, SudokuFixe et listDeBlocs.

Le nombre d'erreurs dans le nouveau tmpSudoku est calculé en utilisant la fonction CalculerNbrErreurs, et le résultat est ajouté à la listDeDifferences.

Après la fin de la boucle, l'écart type de la listDeDifferences est calculé en utilisant la fonction pstdev du module statistics, et le résultat est retourné comme valeur initiale de la température.

```
def CalculerTemperatureInitiale (sudoku, SudokuFixe, listDeBlocs):  
    listDeDifferences = []  
    tmpSudoku = sudoku  
    for i in range(1,10):  
        tmpSudoku = EtatPropose(tmpSudoku, SudokuFixe, listDeBlocs)[0]  
        listDeDifferences.append(CalculerNbrErreurs(tmpSudoku))  
    return (statistics.pstdev(listDeDifferences))
```

Figure 17: Calculer la temperature initiale

```
listDeDifferences = []
```

cette ligne crée une liste vide appelée "listDeDifferences" pour stocker les différences dans le nombre d'erreurs entre les itérations consécutives.

```
tmpSudoku = sudoku
```

cette ligne crée une copie temporaire de la grille de sudoku actuelle "sudoku" et l'assigne à la variable "tmpSudoku".

```
for i in range(1,10):
```

cette ligne commence une boucle qui va itérer de 1 à 9.

```
tmpSudoku = EtatPropose(tmpSudoku, SudokuFixe, listDeBlocs)[0]
```

cette ligne appelle la fonction "EtatPropose" en passant les paramètres "tmpSudoku", "SudokuFixe" et "listDeBlocs", puis met à jour la variable "tmpSudoku" avec la première valeur renvoyée par la fonction (la nouvelle grille proposée).

```
listDeDifferences.append(CalculerNbrErreurs(tmpSudoku))
```

Cette ligne calcule le nombre d'erreurs dans la nouvelle grille "tmpSudoku" en appelant la fonction "CalculerNbrErreurs", puis ajoute ce nombre à la liste "listDeDifferences".

```
return (statistics.pstdev(listDeDifferences))
```

cette ligne calcule l'écart-type de la liste des différences "listDeDifferences" à l'aide de la fonction "pstdev" du module "statistics", puis renvoie cette valeur en tant que température initiale pour l'algorithme de recuit simulé.

RÉSOUTRE LE SUDOKU

Cette fonction résout un sudoku en utilisant l'algorithme du recuit simulé.

La fonction prend en entrée un tableau numpy représentant le sudoku à résoudre.

```
f = open("scores.txt", "a")
solutionTrouve = 0
```

Un fichier texte est ouvert en mode ajout avec la commande `open("scores.txt", "a")`.

Ce fichier servira à stocker les scores de chaque itération de l'algorithme.

Une variable `solutionTrouve` est initialisée à 0 pour indiquer si une solution a été trouvée.

```
while (solutionTrouve == 0):
    FacteurDeDecroissance = 0.99
    decomppteBloque = 0
    SudokuFixe = np.copy(sudoku)
    afficher_sudoku(sudoku)
    ValSudokuFixe(SudokuFixe)
    listDeBlocs = CreerListe3x3Blocs()
    tmpSudoku = RemplissageAleatoire3x3blocs(sudoku, listDeBlocs)
    temperature = CalculerTemperatureInitiale(sudoku, SudokuFixe, listDeBlocs)
    score = CalculerNbrErreurs(tmpSudoku)
    iterations = ChoisirNombreDIterations(SudokuFixe)
    if score <= 0:
        solutionTrouve = 1
```

Une boucle `while` est initiée pour continuer l'exécution de l'algorithme tant qu'une solution n'a pas été trouvée.

Un facteur de diminution est initialisé à 0,99 pour diminuer la valeur de la température à chaque itération.

Un compteur de blocage est initialisé à 0 pour compter le nombre de fois que l'algorithme reste bloqué.

Une copie du sudoku initial est créée avec la commande `np.copy(sudoku)`.

Cette copie sera utilisée pour stocker les valeurs fixées du sudoku initial.

La fonction `afficher_sudoku` est appelée pour afficher le sudoku initial.

La fonction `ValSudokuFixe` est appelée pour fixer les valeurs initiales du sudoku initial dans `SudokuFixe`.

Une liste de blocs de 3x3 est créée avec la commande `CreerListe3x3Blocs`.

La fonction `RemplissageAleatoire3x3blocs` est appelée pour remplir aléatoirement les blocs 3x3 de `tmpSudoku`.

La valeur initiale de la température est calculée avec la fonction `CalculerTemperatureInitiale`.

Le nombre d'erreurs est calculé avec la fonction `CalculerNbrErreurs` pour `tmpSudoku` et stocké dans `score`.

Le nombre d'itérations est calculé avec la fonction `ChoisirNombreDIterations` pour `SudokuFixe` et stocké dans `iterations`.

Si `score` est inférieur ou égal à 0, cela signifie qu'une solution a déjà été trouvée et `solutionTrouve` est donc initialisée à 1.

```
while solutionTrouve == 0:
    ScorePrecedent = score
    for i in range (0, iterations):
```

```

nouveauEtat = ChoisirNouveauEtat(tmpSudoku, SudokuFixe, listDeBlocs,
temperature)

tmpSudoku = nouveauEtat[0]
DifferenceScore = nouveauEtat[1]
score += DifferenceScore
print(score)
f.write(str(score) + '\n')
if score <= 0:
    solutionTrouve = 1
    break

temperature *= FacteurDeDecroissance

```

Une autre boucle while est initiée pour exécuter l'algorithme du recuit simulé.

La valeur de score précédente est stockée dans ScorePrecedent.

Une boucle for est initiée pour effectuer iterations itérations de l'algorithme du recuit simulé.

La fonction ChoisirNouveauEtat est appelée pour choisir une nouvelle configuration pour tmpSudoku.

Le nouveau tableau obtenu est stocké dans nouveauEtat[0].

La différence de score entre le nouveau tableau et l'ancien est stockée dans DifferenceScore.

score est mis à jour en ajoutant scoreDiff.

Le score est écrit dans le fichier texte avec la commande f.write(str(score) + '\n').

Si le score est inférieur ou égal à 0, cela signifie qu'une solution a été trouvée et solutionTrouve est donc initialisée à 1 et la boucle while est interrompue.

La valeur de la température est mise à jour en la multipliant par FacteurDeDecroissance.

```
if score >= ScorePrecedent
```

vérifie si le score est resté le même que le score précédent.

Si c'est le cas : `decompteBloque += 1` incrémente le compteur de scores bloqués.

Sinon, `else: decompteBloque = 0` réinitialise le compteur de scores bloqués.

```
if (decompteBloque > 80): temperature += 2 :
```

augmente la valeur de la temperature si le score est bloqué depuis un certain nombre d'itérations.

```
`if(CalculerNbrErreurs (tmpSudoku)==0):
```

Ces dernières lignes sont exécutées une fois que la solution a été trouvée.

Elles vérifient si le nombre d'erreurs dans la grille de sudoku tmpSudoku est égal à zéro.

Si c'est le cas, cela signifie que toutes les cases ont été correctement remplies et la fonction affiche la solution finale en appelant la fonction afficher_sudoku sur la grille tmpSudoku.

Enfin, le fichier ouvert au début de la fonction est fermé et la fonction renvoie la grille résolue tmpSudoku.


```

def ResoudreSudoku (sudoku):
    f = open("scores.txt", "a")
    solutionTrouve = 0
    while (solutionTrouve == 0):
        FacteurDeDecroissance = 0.99
        decomppteBloque = 0
        SudokuFixe = np.copy(sudoku)
        afficher_sudoku(sudoku)
        ValSudokuCorrige(SudokuFixe)
        listDeBlocs = CreerListe3x3Blocs()
        tmpSudoku = RemplissageAleatoire3x3blocs(sudoku, listDeBlocs)
        temperature = CalculerTemperatureInitiale(sudoku, SudokuFixe, listDeBlocs)
        score = CalculerNbrErreurs(tmpSudoku)
        iterations = ChoisirNombreDIterations(SudokuFixe)
        if score <= 0:
            solutionTrouve = 1

    while solutionTrouve == 0:
        ScorePrecedent = score
        for i in range (0, iterations):
            nouveauEtat = ChoisirNouveauEtat(tmpSudoku, SudokuFixe, listDeBlocs, temperature)
            tmpSudoku = nouveauEtat[0]
            DifferenceScore = nouveauEtat[1]
            score += DifferenceScore
            print(score)
            f.write(str(score) + '\n')
            if score <= 0:
                solutionTrouve = 1
                break

        temperature *= FacteurDeDecroissance
        if score <= 0:
            solutionTrouve = 1
            break
        if score >= ScorePrecedent:
            decomppteBloque += 1
        else:
            decomppteBloque = 0
        if (decomppteBloque > 80):
            temperature += 2
        if(CalculerNbrErreurs(tmpSudoku)==0):
            afficher_sudoku(tmpSudoku)
            break
    f.close()
    return(tmpSudoku)

```

Figure 18: Résoudre le sudoku

Ces trois lignes de code permettent de résoudre le sudoku passé en argument (variable "sudoku") en utilisant la fonction "ResoudreSudoku" et de stocker la solution dans la variable "solution". Ensuite, le nombre d'erreurs dans la solution est calculé en utilisant la fonction "CalculerNbrErreurs" et affiché avec la fonction "print". Finalement, la solution est affichée à l'aide de la fonction "afficher_sudoku".

```

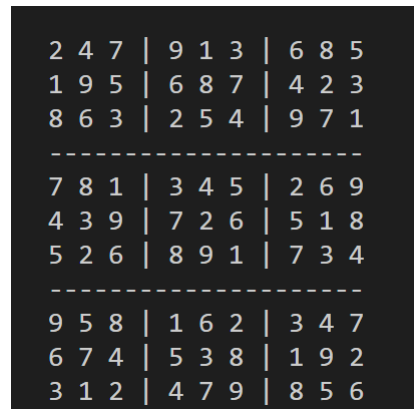
solution = ResoudreSudoku(sudoku)
print(CalculerNbrErreurs(solution))
afficher_sudoku(solution)

```

Figure 19: Affichage du résultat

SOLUTION

Suite à l'exécution du programme, on peut voir l'affichage du nombre d'erreurs à chaque itération, le nombre diminue jusqu'à atteindre 0. Ensuite on affiche le sudoku résolue.



2	4	7		9	1	3		6	8	5
1	9	5		6	8	7		4	2	3
8	6	3		2	5	4		9	7	1

7	8	1		3	4	5		2	6	9
4	3	9		7	2	6		5	1	8
5	2	6		8	9	1		7	3	4

9	5	8		1	6	2		3	4	7
6	7	4		5	3	8		1	9	2
3	1	2		4	7	9		8	5	6

Figure 20: Sudoku Résolu

CONCLUSION

En conclusion, le rapport démontre de manière convaincante que l'utilisation de la métaheuristique Recuit Simulé est une approche efficace pour résoudre le puzzle de Sudoku. Cette technique offre une méthode simple et rapide pour résoudre des problèmes d'optimisation combinatoire, tels que le Sudoku, en explorant un grand nombre de solutions possibles et en trouvant les meilleures solutions.

L'étude montre également que l'algorithme de Recuit Simulé est capable de trouver des solutions optimales pour certains puzzles de Sudoku, même pour ceux de difficulté élevée. En outre, cette approche peut être utilisée pour résoudre d'autres problèmes d'optimisation combinatoire similaires, tels que la planification, la logistique et l'ingénierie.

Cependant, il est important de souligner que la performance de l'algorithme de Recuit Simulé peut être affectée par les paramètres utilisés pour l'initialisation et la mise à jour de la température. Des recherches supplémentaires sont donc nécessaires pour optimiser les paramètres de l'algorithme afin d'améliorer les performances et la précision des résultats.

En fin de compte, la métaheuristique Recuit Simulé offre une approche prometteuse pour la résolution du Sudoku, ainsi que pour d'autres problèmes d'optimisation combinatoire. Cette technique peut être utile pour résoudre des problèmes dans de nombreux domaines, notamment la logistique, la planification et l'ingénierie, où il est nécessaire de trouver des solutions optimales à des problèmes complexes en un temps limité. Par conséquent, l'algorithme de Recuit Simulé est un outil puissant pour résoudre des problèmes d'optimisation combinatoire dans un large éventail de domaines.

REFERENCES

<https://www.techno-science.net/definition/6352.html#:~:text=Un%20probl%C3%A8me%20d'optimisation%20combinatoire,dit%20ensemble%20des%20solutions%20r%C3%A9alisables.>

<https://elearning-facsci.univ-annaba.dz/course/view.php?id=147>

http://sirdeyre.free.fr/Papiers_etc/2006_Sudokus_et_algorithmes_de_recuit_2.pdf

<https://www.math.univ-paris13.fr/~chaussar/Teaching/2015-2016/Info1-InfoBase/TP5/TP5.pdf>

<https://learn.microsoft.com/fr-fr/archive/msdn-magazine/2016/november/test-run-solving-sudoku-using-combinatorial-evolution>

CODE SOURCE

```
import random
import numpy as np
import math
from random import choice
import statistics

#? Prend en entrée un tableau à deux dimensions de 9x9 (c'est-à-dire une liste de listes)
et l'imprime dans un format qui représente une grille de Sudoku
def afficher_sudoku(sudoku):
    print("\n")
    for i in range(len(sudoku)):
        ligne = ""
        if i == 3 or i == 6:
            print("-----")
        for j in range(len(sudoku[i])):
            if j == 3 or j == 6:
                ligne += "| "
            ligne += str(sudoku[i,j])+" "
        print(ligne)

#? La fonction convertit toutes les valeurs remplies en 1, laissant les cellules vides
comme des 0
def ValSudokuFixe(sudoku_fixe):
    for i in range (0,9):
        for j in range (0,9):
            if sudoku_fixe[i,j] != 0:
                sudoku_fixe[i,j] = 1

    return(sudoku_fixe)

#? Cette fonction prend en entrée trois arguments et retourne le nombre d'erreurs dans une
ligne et une colonne données d'une grille de Sudoku
def CalculerNbrErreursLigneColonne(ligne, colonne, sudoku):
    Nbr_erreurs = (9 - len(np.unique(sudoku[:,colonne]))) + (9 -
len(np.unique(sudoku[ligne,:])))
    return(Nbr_erreurs)

#? une fonction qui calcule le nombre total d'erreurs dans une grille de Sudoku donnée
def CalculerNbrErreurs(sudoku):
    Nbr_erreurs = 0
    for i in range (0,9):
        Nbr_erreurs += CalculerNbrErreursLigneColonne(i ,i ,sudoku)
    return(Nbr_erreurs)

#? crée une liste de listes où chaque liste interne contient les indices de ligne et de
colonne d'un bloc 3x3 dans une grille de Sudoku
def CreerListe3x3Blocs ():
    ListeDeBlocsFinal = []
    for r in range (0,9):
```

```

    tmpListe = []
    block1 = [i + 3*((r)%3) for i in range(0,3)]
    block2 = [i + 3*math.trunc((r)/3) for i in range(0,3)]
    for x in block1:
        for y in block2:
            tmpListe.append([x,y])
    ListeDeBlocsFinal.append(tmpListe)
    return(ListeDeBlocsFinal)

#? remplir les cases vide avec une valeur aléatoire choisie parmi les chiffres de 1 à 9
qui ne sont pas déjà présents dans le bloc 3x3 en question, grâce à la fonction "choice"
de la bibliothèque random
def RemplissageAleatoire3x3blocs(sudoku, listDeBlocs):
    for block in listDeBlocs:
        for box in block:
            if sudoku[box[0],box[1]] == 0:
                BlocCourant = sudoku[block[0][0]:(block[-1][0]+1),block[0][1]:(block[-
1][1]+1)]
                sudoku[box[0],box[1]] = choice([i for i in range(1,10) if i not in
BlocCourant])
    return sudoku

#? La fonction parcourt chaque coordonnée dans la liste et ajoute la valeur de la case
correspondante à une somme finale
def SommeDUnBloc (sudoku, UnBloc):
    SommeFinal = 0
    for box in UnBloc:
        SommeFinal += sudoku[box[0], box[1]]
    return(SommeFinal)

#? la fonction vérifie que les deux cases sélectionnées ne sont pas déjà remplies avec une
valeur fixe de la grille,Si les deux cases sélectionnées ne sont pas déjà remplies, la
fonction retourne un tuple contenant ces deux cases
def DeuxCasesAleatoireDansUnBloc(SudokuFixe, block):
    while (1):
        PremierBox = random.choice(block)
        DeuxiemeBox = choice([box for box in block if box is not PremierBox ])

        if SudokuFixe[PremierBox[0], PremierBox[1]] != 1 and SudokuFixe[DeuxiemeBox[0],
DeuxiemeBox[1]] != 1:
            return([PremierBox, DeuxiemeBox])

#? La fonction prend en entrée un sudoku et deux coordonnées correspondant à deux cases
qui seront échangées
def RetournerBoxes(sudoku, boxesARetourner):
    SudokuPropose = np.copy(sudoku)
    PlaceReserve = SudokuPropose[boxesARetourner[0][0], boxesARetourner[0][1]]
    SudokuPropose[boxesARetourner[0][0], boxesARetourner[0][1]] =
SudokuPropose[boxesARetourner[1][0], boxesARetourner[1][1]]
    SudokuPropose[boxesARetourner[1][0], boxesARetourner[1][1]] = PlaceReserve
    return (SudokuPropose)

```

```

#? génère une proposition d'état pour le sudoku en entrée en effectuant des changements
aléatoires à deux cases d'un bloc choisi au hasard
def EtatPropose (sudoku, SudokuFixe, listDeBlocs):
    BlocAleatoire = random.choice(listDeBlocs)

    if SommeDUnBloc(SudokuFixe, BlocAleatoire) > 6:
        return(sudoku, 1, 1)
    boxesARetourner = DeuxCasesAleatoireDansUnBloc(SudokuFixe, BlocAleatoire)
    SudokuPropose = RetournerBoxes(sudoku, boxesARetourner)
    return([SudokuPropose, boxesARetourner])

#?
def ChoisirNouveauEtat (SudokuCourant, SudokuFixe, listDeBlocs, temperature):
    proposition = EtatPropose(SudokuCourant, SudokuFixe, listDeBlocs)
    nouveauSudoku = proposition[0]
    boxesAVerifier = proposition[1]
    #CoutCourant = CalculerNbrErreursLigneColonne(boxesAVerifier[0][0],
boxesAVerifier[0][1], SudokuCourant) +
CalculerNbrErreursLigneColonne(boxesAVerifier[1][0], boxesAVerifier[1][1], SudokuCourant)
    #nouvelCout = CalculerNbrErreursLigneColonne(boxesAVerifier[0][0],
boxesAVerifier[0][1], nouveauSudoku) +
CalculerNbrErreursLigneColonne(boxesAVerifier[1][0], boxesAVerifier[1][1], nouveauSudoku)
    CoutCourant = CalculerNbrErreurs(SudokuCourant)
    nouvelCout = CalculerNbrErreurs(nouveauSudoku)
    DifferenceDeCout = nouvelCout - CoutCourant
    rho = math.exp(-DifferenceDeCout/temperature)
    if(np.random.uniform(1,0,1) < rho):
        return([nouveauSudoku, DifferenceDeCout])
    return([SudokuCourant, 0])

#?cette fonction calcule le nombre de cases non vides dans la grille de sudoku, ce qui
correspond également au nombre d'itérations nécessaires pour remplir complètement la
grille
def ChoisirNombreDIterations(sudoku_fixe):
    nombreDIterations = 0
    for i in range (0,9):
        for j in range (0,9):
            if sudoku_fixe[i,j] != 0:
                nombreDIterations += 1
    return nombreDIterations

#?
def CalculerTemperatureInitiale (sudoku, SudokuFixe, listDeBlocs):
    listDeDifferences = []
    tmpSudoku = sudoku
    for i in range(1,10):
        tmpSudoku = EtatPropose(tmpSudoku, SudokuFixe, listDeBlocs)[0]
        listDeDifferences.append(CalculerNbrErreurs(tmpSudoku))
    return (statistics.pstdev(listDeDifferences))

#?
def ResoudreSudoku (sudoku):
    f = open("scores.txt", "a")

```

```

solutionTrouve = 0
while (solutionTrouve == 0):
    FacteurDeDecroissance = 0.99
    decomppteBloque = 0
    SudokuFixe = np.copy(sudoku)
    afficher_sudoku(sudoku)
    ValSudokuFixe(SudokuFixe)
    listDeBlocs = CreerListe3x3Blocs()
    tmpSudoku = RemplissageAleatoire3x3blocs(sudoku, listDeBlocs)
    temperature = CalculerTemperatureInitiale(sudoku, SudokuFixe, listDeBlocs)
    score = CalculerNbrErreurs(tmpSudoku)
    iterations = ChoisirNombreDIterations(SudokuFixe)
    if score <= 0:
        solutionTrouve = 1

    while solutionTrouve == 0:
        ScorePrecedent = score
        for i in range (0, iterations):
            nouveauEtat = ChoisirNouveauEtat(tmpSudoku, SudokuFixe, listDeBlocs,
temperature)

            tmpSudoku = nouveauEtat[0]
            DifferenceScore = nouveauEtat[1]
            score += DifferenceScore
            print(score)
            f.write(str(score) + '\n')
            if score <= 0:
                solutionTrouve = 1
                break

            temperature *= FacteurDeDecroissance
            if score <= 0:
                solutionTrouve = 1
                break
            if score >= ScorePrecedent:
                decomppteBloque += 1
            else:
                decomppteBloque = 0
            if (decomppteBloque > 80):
                temperature += 2
            if(CalculerNbrErreurs(tmpSudoku)==0):
                afficher_sudoku(tmpSudoku)
                break

f.close()
return(tmpSudoku)

sudokuDeDepart = """
    207010605
    095600423
    800004000
    700000260
    000726018
    020090030
    050000307

```



```

        604530000
        012079000
    """
    #? crée un tableau numpy sudoku à partir d'une chaîne de caractères sudokuDeDepart qui
    représente l'état initial d'une grille de Sudoku
    sudoku = np.array([[int(i) for i in ligne] for ligne in sudokuDeDepart.split()])

    solution = ResoudreSudoku(sudoku)
    print(CalculerNbrErreurs(solution))
    afficher_sudoku(solution)

```