TP5 : Le recuit simulé

Dans ce TP, vous allez découvrir un algorithme d'optimisation, appelé le recuit simulé, qui tout comme la descente de gradient permet de trouver le minimum d'une fonction.

Vous pouvez récupérer les fichiers de l'énoncé ici : `https://goo.gl/wSLEpF`.

1 Résoudre un Sudoku

Le sudoku consiste en une grille de 9x9 cases, contenant une valeur entre 1 et 9, ou rien du tout. Le principe du puzzle est de remplir les cases vides avec un chiffre de 1 à 9 de façon à ce que chaque chiffre n'apparaisse qu'une seule fois sur chaque ligne, chaque colonne, et chaque carré 3x3 (il y a en tout 9 carrés 3x3 qui composent la grille générale).

1. Commencez par regarder le code contenu dans le fichier *sudoku.c*. Ce programme lit un des fichiers puzzle, le charge dans deux tableaux et affiche le résultat. Les cases vides représentent les valeurs à trouver pour résoudre le sudoku.
2. Le schéma général de résolution que l'on va adopter sera le suivant :

```
1 Remplir les cases vides avec des chiffres entre 1 et 9, de façon à ce que chaque
  chiffre n'apparaisse exactement qu'une seule fois sur chaque ligne ;
2 while le puzzle n'est pas résolu do
3 |   Choisir une ligne au hasard du puzzle, et y permuter deux nombres (ne faisant
  |   pas partie des nombres données au départ dans le puzzle) ;
4 |   Si cette permutation donne une "meilleure" solution, on la conserve ; sinon, on
  |   la rejette ;
5 end
```

   Pour l'initialisation de l'algorithme, nous aurons besoin de remplir la grille de sudoku avec des chiffres. Faîtes une fonction *initialiser_grille* prenant en paramètre une grille de sudoku (un tableau 9x9) et remplissant cette dernière de façon à ce que chaque chiffre entre 1 et 9 n'apparaisse qu'une seule fois exactement sur chaque ligne.
3. Faîtes une fonction *permutation* qui échange deux chiffres d'une ligne de la grille de sudoku, tous passés en paramètre. Gardez à l'esprit qu'un second appel à la même fonction avec les mêmes paramètres devrait permettre d'annuler la permutation.
4. Il nous faut un moyen d'évaluer à quel point notre grille se rapproche de la solution souhaitée. Pour cela, on comptera le nombre de chiffres en double sur chaque colonne ainsi que chaque carré 3x3 de notre grille : plus ce nombre sera élevé, et moins bonne sera la solution ; au contraire, si ce nombre atteint 0, nous aurons trouvé la solution.

   Proposez une fonction *score_grille* prenant en paramètre une grille de sudoku, et calculant son score selon la méthode expliquée précédemment.

5. Il ne vous reste plus qu'à implémenter la boucle principale décrite précédemment, dans la fonction main, et à tester votre programme avec l'un des puzzles proposés dans le fichier joint au TP. Pensez à limiter le nombre de tours de la boucle principale avec un compteur : par exemple, si aucune permutation ne produit une meilleure grille après N tours, on s'arrête.

6. Normalement, votre programme ne devrait pas parvenir à trouver une solution : cela vient du fait que la boucle principale réalise ce que l'on appelle une descente de gradient - on conserve une solution si elle est meilleure que la solution présente.

   Il faut implémenter, pour ce genre de problème, un recuit simulé : on conservera tout de même une solution moins bonne avec une probabilité qui décroit dans le temps. Voici la boucle principale à réaliser :

```
1  Remplir les cases vides avec des chiffres entre 1 et 9, de façon à ce que chaque
   chiffre n'apparaisse exactement qu'une seule fois sur chaque ligne ;
2  while le puzzle n'est pas résolu do
3      for T allant de 1 à 0 par pas de 0.05 do
4          for i allant de 0 à 1000 do
5              Choisir une ligne au hasard du puzzle, et y permuter deux nombres (ne
               faisant pas partie des nombres données au départ dans le puzzle) ;
6              if score(nouvelleGrille) ¡ score(ancienneGrille) then
7                  Conserver la nouvelle grille ;
8              end
9              else
10                 Choisir un nombre c au hasard entre 0 et 1 ;
11                 if c < exp(score(ancienneGrille)−score(nouvelleGrille) / T) then
12                     Conserver la nouvelle grille ;
13                 end
14             end
15         end
16     end
17 end
```

Implémentez cette nouvelle boucle principale et testez votre programme.