On souhaite améliorer l’algorithme du sudoku en utilisant un algorithme de backtracking-search, en vous appuyant sur l’algorithme de backtracking-search développé pour la coloration des pays et en utilisant les mêmes types d’instructions en particulier dans les méthodes ***MRV*** et ***nextValue***, et dans la fonction principale.

1. **Création de la classe contenant le jeu**

* Créer une classe ***CSP*** qui contiendra les propriétés et les méthodes nécessaires à la résolution du problème.
* Cette classe devra contenir trois propriétés correspondant au jeu du sudoku (***grille***), aux cases à remplir (***toFill***) et aux domaines de définition de ces cases (***domain***).
* Le constructeur devra créer ces trois propriétés, lire la grille dans une fichier de texte et initialiser les deux autres propriétés à null.
* Information. Dans le programme on fera en sorte que la propriété ***toFill*** soit une liste de tuple (x,y) et une liste de liste pour ***domain***. Avec notre exemple de sudoku ces propriétés au premier appel du programme de ***backtracking\_search*** elles auront comme valeurs :

|  |  |
| --- | --- |
| **attribut** toFill  index 0 1  0 0 0  1 0 2  2 0 5  3 0 6  4 0 7  5 0 8  6 1 0  7 1 1  8 1 2  9 … … | **attribut** Domaine  index  0 [1, 2, 4, 5, 6]  1 [1, 4, 5]  2 [1, 3, 5, 6]  3 [1, 6]  4 [1, 3, 6]  5 [2, 3, 6]  6 [1, 4, 5, 6, 7]  7 [1, 5, 6, 7]  8 [1, 4, 5, 7]  9 … |

1. **Propriété de recherche des cases liées à une case donnée**

* Cette propriété ***constraints*** doit rechercher pour une case donnée repérée par ses coordonnées (x,y) quelles sont les cases libres et les cases occupées qui sont liées à cette case au sens du sudoku.
* Cette fonction devra remplir deux listes (***libre*** et ***occupée***) qui contiendront les cases qui sont sur la même ligne sur la même colonne et dans le même espace (3\*3) que la case (x,y).
* Pour cela la propriété devra parcourir dans un premier temps la ligne de (x,y), puis la colonne et enfin l’espace de cette case.
* Les cases sur la même ligne que (x,y) qui sont libres (valeur=0) seront placées dans la liste ***libre*** et celles déjà affectées dans la liste ***occupée***. Les valeurs mises dans ces listes seront des couples [x,y].
* Puis vous traiterez les cases qui sont sur la même colonne que (x,y).
* Enfin vous devrez faire de même pour les cases qui sont dans le même espace (3\*3) que la case (x,y). Attention dans cet espace les cases qui sont sur la même colonne et la même ligne que (x,y) ne doivent pas être prise en compte puisqu’elles auront été traitées par les deux boucles précédentes.
* Cette méthode recevra donc en entrée les coordonnées de la case à traiter retournera ensuite les deux listes ***libre*** et ***occupée***.
* Vous devez avant de passer à la suite tester votre méthode. Pour cela créez un objet sudoku de type ***CSP*** puis vous appellerez cette fonction avec des valeurs pour un (x,y) donné et vérifiez que vous obtenez bien les bons résultats.

1. **Recherche des domaines de définition des cases non encore affectées (de toFill).**

* Cette méthode recherche pour toutes les cases non encore affectées leurs domaines de définition. C’est-à-dire qu’elles sont les valeurs que l’on peut encore mettre dans ces différentes cases.
* On utilisera une liste locale ***domaine*** qui va contenir les différents domaines de définition de cases libres.
* La méthode consiste à parcourir la liste ***toFill*** et à rechercher pour chaque case quelles sont les valeurs acceptables. Pour cela vous devez grâce à la méthode ***constraints*** récupérer les listes des cases libres et occupées liées à la case que vous êtes en train de traiter.
* A partir du résultat de ***constraints*** et surtout de la liste des cases occupés vous pourrez trouver toutes les valeurs qui sont interdites pour la case en cours de traitement et ensuite en déduire celles que l’on peut encore choisir.
* A la fin de la fonction vous disposerez dans la variable local ***domaine*** les listes des valeurs acceptables pour chaque case à remplir. Cette liste de domaine sera retournée par la méthode.

1. **Méthode MRV**

* Cette méthode doit nous permettre de trouver la prochaine case à affecter. Elle se décompose en deux parties, la première est une heuristique qui sélectionne les cases qui ont le moins de valeurs compatibles. Si plusieurs cases sont dans ce cas une seconde heuristique permet de sélectionner dans cette liste les cases qui ont le plus de contraintes avec les cases non encore affectées.
* La méthode commence par rechercher via la méthode ***where*** de ***numpy*** toutes les cases qui ont comme valeur 0. Rappel : cette méthode retourne un couple (tuple) de listes contenant dans la première les coordonnées x des cases vides et dans la seconde les coordonnées y de ces mêmes cases (voir programme du sudoku avec backtracking).
* Le résultat de la requête where doit être traiter afin de transformer le couple de liste en une liste de couple. Pour cela il suffit de parcourir le éléments des deux listes puis d’insérer dans la liste ***toFill*** les coordonnées sous forme de couple (x,y).
* Si la liste ***toFill*** est vide on peut quitter la méthode en retournant cette liste puisque le programme est terminé, toutes les cases ont une valeur.
* Dans le cas contraire il faut évaluer les domaines de définition de cases à remplir en utilisant la méthode précédente, calculer la taille du domaine la plus petite : ***minMRV***, puis sélectionner les cases correspondant à cette valeur minimum ***MRVs***.
* Si une seule case correspond à cette requête ***MRVs*** peut être retournée car il n’y n’est pas nécessaire de faire un tri supplémentaire sur ***MRVs***.
* Si plusieurs cases ont un domaine de taille ***minMRV***, la sélection suivante doit se baser sur le degré d’heuristique. Pour chaque sommet de la liste ***MRVs*** on recherche quelles sont les cases liées et l’on enregistre ensuite le nombre de cases dans une liste spécifique *degres*.
* On sélectionne ensuite les cases de ***MRVs*** qui ont le degrés d’heuristiques maximum.

1. **Trie des valeurs acceptable en fonction du nombre de variables directement impactées.**

* La méthode ***nextValues*** reçoit l’indice ***indS*** de la case sélectionnée dans la liste ***toFill***. Cette méthode commence par récupérer la liste des cases liées à la case à traiter. Si aucune case n’est impactée par ***indS*** le domaine de ***indS*** peut être retourné directement sans tri. De même si le domaine de définition de ***indS*** ne contient qu’une seule valeur celle-ci peut être retournée.
* Pour toutes les valeurs compatibles de la case ***indS*** on doit chercher dans combien de domaines elles ont un impact. Ainsi pour chaque valeur possible pour la case ***indS*** on compte dans combien de domaine de ses cases liées elle est présente. Ces valeurs sont conservées dans une liste qui servira ensuite à trier les valeurs acceptables de ***indS.***
* Les valeurs du domaine de ***indS*** seront ensuite triées en fonction de cette liste et retournés par la méthode.

1. **Fonction de recherche par backtracking.**

* Il reste à implémenter la fonction principale permettant de remplir les cases du sodoku. La fonction de ***backtracking-search*** débute par le choix via la fonction ***MRV*** des cases à remplir en priorité. Si la liste des cases à remplir est vide le programme est terminé et la fonction est terminée.
* Dans le cas contraire il faut choisir une case dans la liste des cases retournée par la méthode ***MRV*** de la classe ***CSP***, puis classer les valeurs à choisir grâce à la méthode ***nextValues***.
* Il faut ensuite parcourir la liste de ces valeurs, modifier la case sélectionnée puis rappeler récursivement la fonction ***backtracking-search***.
* Au retour de cette fonction il faut effectuer le backtracking si la solution n’a pas été trouvée.
* Finalement pour terminer le programme il est nécessaire avant d’appeler la fonction ***backtracking-search*** de créer un jeu de sudoku à partir de la classe ***CSP***, et créer une variable booléenne ***Termine*** initialisée à ***False***.