Humbleloop

Le but de ce projet est de programmer un jeu s'inspirant du jeu Infinityloop.

Un vérificateur de solutions, un générateur de grilles et un solveur sont demandés.

Architecture

Le jeu Infinityloop peut être modélisé comme un CSP (Constraints Satisfaction Problem). Les variables sont les cases, les domaines sont les orientations possibles et les contraintes sont binaires entre les variables. Chaque variable doit être connectée avec son voisin.

Dans une version expérimentale non aboutie, la liste des valeurs possibles est séparé de la cellule et stockée dans une Map.

Répartition du travail

Ayant passé beaucoup de temps sur les CSP et les algorithmes de résolution, le solveur a été codé par Philippe Nguyen alors que le vérificateur, le générateur ainsi que l'interface graphique ont été codés par Charles Nourry.

Vérificateur

Étant donné une liste de contraintes d'un jeu, le vérificateur doit vérifier si toutes les contraintes sont satisfaites.

Solveur

La méthode la plus basique est une recherche en profondeur ou backtracking. Cette méthode a pour inconvénient d'être lente d'où la nécessité d'un forward checking voire un algorithme de réduction de domaine AC(3).

Cela consiste à enlever une valeur possible d'une variable s'il n'existe pas de valeur possible chez un voisin tel que la contrainte est satisfaites.

L'algorithme AC3 étant plus difficile à mettre en place en multithreading (DisAC3) et très lent en mono thread, on a choisi d'effectuer l'algorithme d'AC plusieurs fois

en multithreading.

Dans le solveur on commence par rendre les constraintes arc-consistantes
puis on effectue un forward checking.

Pour choisir la variable à assigner, on a préféré prendre celle qui à le moins
de valeurs possibles de manière dynamique. Ce choix est le plus utilisé de manière
générale. Il est également possible de prendre celui qui crée le plus
de conflits avec ses voisins mais ce choix est moins efficace en pratique.
On a également tenté de prendre les plus éloignés sur centre (distance Euclidienne)
mais ce choix est également moins performant et est plus apte à créer des conflits.

Le problème rencontré à ce niveau est la restauration du domaine en cas de non
consistance de la dernière assignation mais également la restauration du
domaine induit par l'assignation des valeurs précédentes.

Un bug non-résolu existe cependant dans la restauration des valeurs induites.
En effet, on redonne toutes les valeurs possibles à une variables bien qu'elles
ne peuvent pas correspondre à une orientation valide.

Les algorithmes de maintenance de l'arc-consistance (MAC) sont lourds en terme
de complexité et dans notre cas, il ne permet pas de réduire les domaines
efficacement d'où l'utilisation d'un forward checking plutôt qu'un algorithme MAC.

Générateur

Notre générateur part d’une grille entièrement vide, à laquelle on associe une
liste de contraintes binaires qui correspond à toutes les potentielles liaisons,
sans doublons.
On parcourt ensuite cette liste de contraintes, chaque contrainte
à 60% de chance d’être créée. Si c’est le cas, on met à jour les données dans les deux
cellules concernées. A la fin du parcours de la liste, on se retrouve donc avec une grille
réalisable. Il ne nous reste plus qu’à trouver le type final de chaque pièce (en fonction de leurs
liaisons) puis de mélanger la grille en faisant pivoter chaque pièce aléatoirement. On obtient
alors une grille réalisable mélangée.

Interface graphique

Nous avons choisi d’utiliser le framework java Vaadin, ce qui nous a permis de créer une
application sur un serveur local. Nous avons pu construire une première application web à
l’aide des composants fournis par le framework. Les nombreux add-ons permettent de faire

de belles applications web, nous en avons utilisé un. Pour notre projet, nous avons utilisé l'add-on Canvas pour Vaadin de Henri Hezamu. Ce qui nous a permis de tracer les formes de notre interface.