RAPPORT PROJET INF402

Introduction :

Ceci est le compte-rendu du projet d'INF402 portant sur le jeu du Norinori.

Description du Norinori :

Le Norinori est un jeu de réflexion impliquant un rectangle divisé en plusieurs carrés égaux, chaque carré étant associé à une région délimitée par des lignes épaisses. L'objectif est d'avoir exactement deux carrés noirs par région, avec la particularité qu'un carré noir ne touche qu'un seul autre carré noir.

Modélisations sous forme logique :

Pour exprimer les contraintes du problème, nous utilisons des représentations symboliques où r représente une région, et x et y les coordonnées de chaque case. Ainsi, nous noterons les relations suivantes nécessaires à la bonne résolution du problème :

- N(x) : case noire aux coordonnées x, y, z.
- R(x, y, r) : case noire aux coordonnées x, y dans la région r.
- V(x,y) : présence d'une case noire voisin à une case de coordonnées x

Grâce à ces notations nous pouvons formaliser les contraintes issues des règles de ce casse-tête.

A. Contraintes :

- Il est nécessaire d'avoir aux plus deux carrés noirs par région de la grille de jeu :

  $\forall r, \forall x, \forall y, \forall z, (x \neq y) \wedge (y \neq z) \wedge (x \neq z) \Rightarrow \neg R(x, r) \vee \neg R(y, r) \vee \neg R(z, r)$

- Il est essentiel d'avoir au moins deux carrés noirs par région de la grille :

  $\forall r, \exists x, \exists y, \exists z, \exists w, (x \neq w) \vee (y \neq z) \Rightarrow R(x,y,r) \wedge R(w,z,r)$

- Chaque case noircie a exactement une seule et unique case voisine noircie également :

  $\forall x, \forall y, \forall z, (x \neq y) \wedge (x \neq z) \wedge (y \neq z) \Rightarrow N(x) \wedge ((V(x, y) \wedge \neg V(x, z)) \vee (V(x, z) \wedge \neg V(x, y)))$

B. Modélisation en forme normale conjonctive :

Lorsque nous rassemblons les conditions évoquées ci-dessus nous obtenons la formule logique suivante :

$\forall r, \forall x, \forall y, \forall z, (x \neq y) \wedge (x \neq z) \wedge (y \neq z) \Rightarrow N(x) \wedge ((V(x, y) \wedge \neg V(x, z)) \vee (V(x, z) \wedge \neg V(x, y))) \wedge R(x,y,r) \wedge R(w,z,r) \wedge (\neg R(x, r) \vee \neg R(y, r) \vee \neg R(z, r))$

Nous la passons donc en forme normale conjonctive afin de faciliter la programmation.

$\forall r, \forall x, \forall y, \forall z, (x \neq y) \wedge (x \neq z) \wedge (y \neq z) \Rightarrow N(x) \wedge (V(x, y) \vee V(x,z)) \wedge (\neg V(x,z) \vee \neg V(x,y)) R(x,y,r) \wedge R(w,z,r) \wedge (\neg R(x, r) \vee \neg R(y, r) \vee \neg R(z, r))$

Grâce à celle-ci nous pouvons passer à l'implémentations des différents programmes nécessaires.

C. Exemple de transformation :

Pour assurer au moins deux carrés noirs dans la région r1 :

$(R(0,0,1) \vee R(0,1,1) \vee R(1,0,1)) \wedge (R(0,1,1) \vee R(1,0,1) \vee R(1,1,1)) \wedge (R(1,0,1) \vee R(1,1,1) \vee R(0,0,1)) \wedge (R(1,1,1) \vee R(0,0,1) \vee R(0,1,1))$

Pour garantir aux plus deux carrés noirs dans la région r1 :

$(\neg R(0,0,1) \vee \neg R(0,1,1) \vee \neg R(1,0,1)) \wedge (\neg R(0,0,1) \vee \neg R(0,1,1) \vee \neg R(1,1,1)) \wedge (\neg R(0,0,1) \vee \neg R(1,1,1) \vee \neg R(1,0,1)) \wedge (\neg R(0,1,1) \vee \neg R(1,0,1) \vee \neg R(1,1,1))$

RAPPORT PROJET INF402

Pour les voisins des cases noircies, reprenons l'exemple du carré de 4 cases de côté :

(¬ N(1,0) ∨ ¬ V(1,0,0,0) ∨ ¬ V(1,0,1,1)) ∧ (¬ N(1,0) ∨ ¬ V(1,0,0,0) ∨ ¬ V(1,0,2,0)) ∧ (¬ N(1,0) ∨ ¬ V(1,0,1,1) ∨ ¬ V(1,0,2,0)) ∧ (¬ N(1,0) ∨ V(1,0,0,0) ∨ V(1,0,1,1) ∨ V(1,0,2,0))

Implémentation des programmes :

Lors de l'implémentation en Python, nous avons géré plusieurs aspects. Tout d'abord, la création de toutes les clauses en utilisant les données du fichier qui contient le nombre de colonnes, de lignes et les régions. Nous avons séparé ce processus en deux parties :

Pour les régions avec au plus deux carrés noirs : assembler toutes les cases sauf une, en alternant pour éviter les clauses uniques.

Pour les régions avec au moins deux carrés noirs : chaque case apparaît autant de fois avec toutes les autres, puis négation de toutes les clauses.

Nous avons ensuite traité la partie voisine, en tenant compte des 4 voisins possibles par case. Chaque case est dans toutes les clauses concernées, et nous avons pris en compte si une case n'est pas noire.

Explications sur les programmes fournis :

Fichier main.py :

Ce fichier sert à appeler les autres fichiers pour créer les clauses puis les résoudre avec le solveur et enfin afficher la solution.

Fichier probleme.py :

ecrire_dimacs(nom_fichier, clauses, nb_cases):

Cette fonction prend en entrée un nom de fichier de sortie, une liste de clauses et le nombre total de cases dans la grille de jeu. Elle écrit ensuite les clauses au format DIMACS dans le fichier de sortie afin d'être donné au SAT-solver.

bonne_region(region, coord):

Vérifie si toutes les cases dans une région donnée sont adjacentes les unes aux autres.

tableau_coord(regions, lignes, colonnes):

Crée un dictionnaire associant chaque case à ses coordonnées (ligne, colonne) dans la grille.

probleme():

Fonction principale du script qui se décompose en quatre parties :

- Lecture du fichier donné en entrée et vérification des données contenues
- Génération de clauses en prenant en compte les contraintes de règles du jeu (deux cases noires exactement par région et une case noire a toujours exactement un voisin noir)
- Ecriture au format DIMACS, fichier sortie avec extension.cnf, chaque clauses s'écrit sur une ligne terminée par un 0.

RAPPORT PROJET INF402

En résumé, le fichier probleme.py lit les informations du jeu Norinori à partir d'un fichier d'entrée, génère les clauses logiques correspondantes en utilisant les règles du jeu, et écrit ces clauses dans un fichier au format DIMACS pour être résolu par un SAT-solver.

Fichiers clauses.py :

negatif(clauses) :
Cette fonction prend en entrée une liste de clauses et la transforme pour trouver la négation pour les contraintes où au plus deux cases noires sont autorisées par région.

est_voisin(coord1, coord2) :

Prend en entrée les coordonnées (ligne, colonne) de deux cases et vérifie si elles sont adjacentes dans ce cas elle retourne True, sinon False.

clause_voisin(case, voisins) :

Génère les clauses correspondantes à la contrainte selon laquelle chaque case noire a exactement un voisin case noire. Elle prend en entrée une case et une liste de ses voisins, puis génère les clauses nécessaires pour garantir cette contrainte. Les clauses générées sont ainsi conformes au format attendu pour le SAT-solve.

un_voisin_case_noir_par_case_noir(coord) :

Crée les clauses qui assurent qu'une case noire a exactement un voisin case noire.

Elle parcourt toutes les cases de la grille, identifie leurs voisins et génère les clauses correspondantes à la règle.

au_moins_deux_case(region) :

Cette fonction génère les groupes de clauses nécessaires pour garantir qu'il y a au moins deux cases noires dans une région donnée. Elle génère tous les sous-ensembles de taille n-1 de la région, où n est le nombre total de cases dans la région. Chaque sous-ensemble représente un groupe de clauses où au moins une des cases est noire.

au_plus_deux_case(region) :

Crée les groupes de clauses nécessaires pour garantir que la grille possède aux plus deux cases noires dans une région donnée en argument. Elle génère tous les sous-ensembles de taille 3 possibles de la région, à l'exception des cas où la région contient déjà moins de 3 cases. Chaque sous-ensemble représente un groupe de clauses où au plus deux des cases doivent être noires.

deux_case_noires_par_regions(regions) :

Cette fonction prend en entrée une liste de régions et génère les clauses correspondant à la règle d'avoir exactement deux cases noires par région. Elle utilise les fonctions au_moins_deux_case(region) et au_plus_deux_case(region) pour générer ces clauses. Elle renvoie toutes les clauses nécessaires pour appliquer cette règle à toutes les régions de la grille.

En résumé, le fichier "clauses.py" a pour but de générer les clauses logiques nécessaires à la modélisation des règles du jeu Norinori sous forme de contraintes logiques

RAPPORT PROJET INF402

Fichier slv.py :

Il faut installer PySAT via le terminal avec la commande : `pip install pysat`

Fichier du solveur SAT.

sol(fichier, clauses) :

Prend en arguments un nom de fichier et une liste de clauses puis crée une formule sous forme normale et utilise un solveur pour résoudre le problème. Si le solveur renvoie vrai, l'ensemble est satisfiable et la solution est écrite dans un fichier sortie.

solveur() :

Lit le contenu du fichier pris en argument de ligne de commande et en retient uniquement les clauses. Ces clauses sont transformées en listes d'entiers et appelle la fonction sol pour établir la solution.

En résumé, ce fichier lit un fichier contenant des clauses au format DIMACS, utilise un solveur SAT pour décider si les clauses sont satisfiables ou non et écrit le résultat dans un fichier.

Fichier interface.py :

Fichier chargé de l'affichage de la solution au travers d'une interface graphique.

affichage(colonne, ligne, tab_info) :

Prend en entrée un nombre de colonnes, de lignes et les données du fichier fourni. Permet l'ouverture d'une fenêtre graphique avec Tkinter et calcule la taille de la fenêtre à afficher puis rempli les cases du tableau créé en fonction des données fournies dans le fichier lu.

solution() :

Lit les arguments de la ligne de commande afin d'obtenir le nombre de colonnes, de lignes et le nom du fichier à lire puis stocke les informations lue dans un fichier tab_info avant d'appeler affichage pour lui fournir les données requises.

En résume, ce fichier crée une interface graphique en utilisant Tkinter pour afficher une fenêtre représentant la solution du jeu Norinori.

Conclusion :

Pour conclure, nous avons eu à transformer les règles du jeu Norinori en formules logiques puis en forme normale conjonctive afin d'écrire un programme python capable d'interpréter ces clauses en tenant compte des règles du jeu pour produire un fichier DIMACS vérifiable par un SAT-solver qui vérifie si l'ensemble des clauses est satisfiable ou non avant d'afficher la solution.