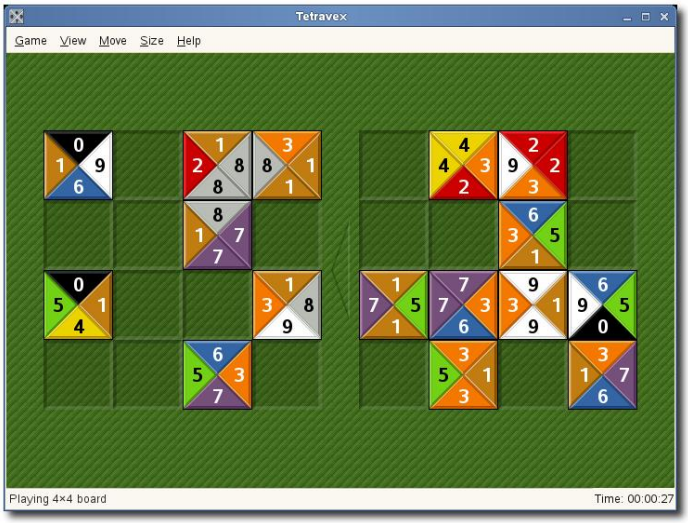
Compte rendu : Tetravex

Arris ZAIDI
Racim ZENATI
Augustin MANIAK

INM - 03

INF402

Introduction à la logique et démonstration automatique



Problème et Modélisation du Tetravex

1.1 Introduction

Tetravex est un jeu de puzzle dans lequel des tuiles carrées doivent être disposées sur une grille de taille $n \times n$. Chaque tuile possède quatre côtés, chacun portant un numéro compris entre 0 et 9. L'objectif est d'agencer les tuiles de manière à ce que les numéros des côtés adjacents soient identiques.

Ce projet a pour objectif de modéliser et de résoudre automatiquement des puzzles **Tetravex** à l'aide de la **logique propositionnelle** et d'un **solveur SAT**. Il combine plusieurs technologies :

- **OCaml** pour la génération des clauses logiques au format **DIMACS CNF**.
- Un **solveur SAT** pour résoudre le problème de manière algorithmique.
- **Python avec Pygame** pour l'interface graphique permettant d'afficher la solution du puzzle.

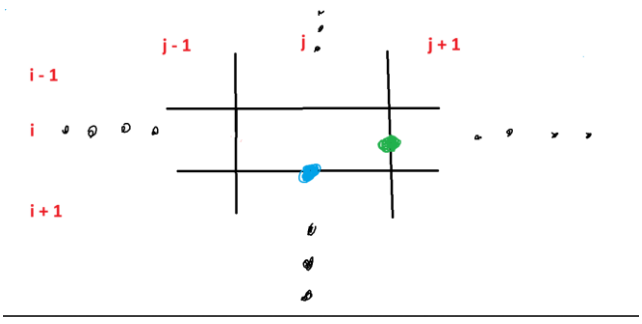
L'intérêt de ce projet est de démontrer comment un problème combinatoire visuel peut être traduit en une formule logique satisfaisable, résolu efficacement, puis restitué de façon interactive à l'utilisateur.

1.2 Modélisation en logique du premier ordre

Pour formaliser ce problème, nous définissons le prédicat $T(i, j, t)$, qui signifie que la tuile t est placée à la position (i, j) de la grille. À partir de cette définition, nous établissons les contraintes suivantes :

- **Unicité des tuiles dans la grille** :
  - Chaque case contient exactement une tuile :
    $$\forall (i, j), \exists! \, t \quad T(i, j, t)$$
  - Chaque tuile est placée une seule fois :
    $$\forall t, \exists! \, (i, j) \quad T(i, j, t)$$
- **Correspondance des bords adjacents** :
  - Pour toute paire de tuiles adjacentes horizontalement :
    $$\forall (i, j, t_1, t_2), \quad T(i, j, t_1) \wedge T(i, j+1, t_2) \Rightarrow \text{Droite}(t_1) = \text{Gauche}(t_2)$$
    (Représenté par le point vert dans le schéma)
  - Pour toute paire de tuiles adjacentes verticalement :
    $$\forall (i, j, t_1, t_2), \quad T(i, j, t_1) \wedge T(i+1, j, t_2) \Rightarrow \text{Bas}(t_1) = \text{Haut}(t_2)$$
    (Représenté par le point bleu dans le schéma)

Cette formalisation permet d'exprimer le problème de manière rigoureuse, facilitant ainsi son traitement algorithmique ou sa résolution à l'aide de techniques de programmation logique.



1.3 Modélisation en Forme Normale Conjonctive (FNC)

Nous transformons les contraintes du problème en une **forme normale conjonctive (FNC)**, couramment utilisée par les solveurs SAT. Cette transformation permet d'exprimer le problème sous forme de clauses conjonctives de littéraux disjonctifs.

Contraintes en FNC : **Ces clauses traduisent les règles fondamentales du jeu en un modèle logique exploitable par un solveur SAT.**

- **Chaque case contient au moins une tuile** :

$$(x_{1,1,a} \vee x_{1,1,b} \vee x_{1,1,c} \vee x_{1,1,d}) \wedge \dots$$

(Cette règle s'applique pour chaque case de la grille, où $x_{i,j,t}$ indique que la tuile t est placée en position (i,j)).

- **Une tuile ne peut être placée qu'à un seul endroit** :
Pour toute tuile t et pour toute paire de positions distinctes $(i_1,j_1) \neq (i_2,j_2)$:

$$\neg(x_{i_1,j_1,t} \wedge x_{i_2,j_2,t})$$

(Autrement dit, une même tuile ne peut pas apparaître à deux endroits différents dans la grille).

- **Contraintes d'adjacence** :
Ces contraintes assurent que les bords adjacents des tuiles placées côte à côte ont des valeurs correspondantes.
  - **Correspondance horizontale** (pour les tuiles à gauche et à droite) :

$$\forall(i,j,t_1,t_2), \quad (x_{i,j,t_1} \wedge x_{i,j+1,t_2}) \Rightarrow \text{droite}(t_1) = \text{gauche}(t_2)$$

  - **Correspondance verticale** (pour les tuiles en haut et en bas) :

$$\forall(i,j,t_1,t_2), \quad (x_{i,j,t_1} \wedge x_{i+1,j,t_2}) \Rightarrow \text{bas}(t_1) = \text{haut}(t_2)$$

1.4 Modélisation Globale sous Forme FNC

Voici la **formule globale sous forme normale conjonctive (FNC)** :

$$\bigwedge_{i,j}(x_{i,j,A} \vee x_{i,j,B} \vee x_{i,j,C} \vee x_{i,j,D})$$

$$\wedge \quad \bigwedge_{T}\bigwedge_{(i_1,j_1)\neq(i_2,j_2)}(\neg x_{i_1,j_1,T} \vee \neg x_{i_2,j_2,T})$$

$$\wedge \quad \bigwedge_{i,j,T_1,T_2}\left(\neg x_{i,j,T_1} \vee \neg x_{i,j+1,T_2} \vee (\text{Droite}(T_1) = \text{Gauche}(T_2))\right)$$

$$\wedge \quad \bigwedge_{i,j,T_1,T_2}\left(\neg x_{i,j,T_1} \vee \neg x_{i+1,j,T_2} \vee (\text{Bas}(T_1) = \text{Haut}(T_2))\right)$$

Cette **formulation unique en FNC** peut être directement exploitée par un solveur SAT pour déterminer une affectation satisfaisant toutes les contraintes du problème.

EXEMPLE :

1. Définition du Problème

Nous avons une grille **2×2** et **4 tuiles** (1, 2, 3, 4), où chaque tuile possède **quatre bords** (Haut, Bas, Gauche, Droite). L'objectif est de placer les tuiles de façon à ce que les valeurs des bords adjacents correspondent.

Tuiles et leurs valeurs de bords :

Tuile	Haut	Bas	Gauche	Droite
1	1	2	3	4
2	1	3	4	2
3	2	4	3	1
4	3	1	2	4

Grille 2×2 et notation des positions :

(i=1, j=1)	**(i=1, j=2)**
(i=2, j=1)	**(i=2, j=2)**

Chaque variable booléenne $x_{i,j,T}$ signifie que la tuile T est placée en position **(i,j)**.

---

2. Modélisation en FNC

a. Chaque case contient au moins une tuile

Chaque position de la grille doit contenir au moins une tuile parmi **1, 2, 3, 4** :

$$(x_{1,1,1} \vee x_{1,1,2} \vee x_{1,1,3} \vee x_{1,1,4}) \wedge$$

$$(x_{1,2,1} \vee x_{1,2,2} \vee x_{1,2,3} \vee x_{1,2,4}) \wedge$$

$$(x_{2,1,1} \vee x_{2,1,2} \vee x_{2,1,3} \vee x_{2,1,4}) \wedge$$

$$(x_{2,2,1} \vee x_{2,2,2} \vee x_{2,2,3} \vee x_{2,2,4})$$

---

b. Chaque tuile est placée exactement une fois

Formulation Générale :

$$\forall T, \forall (i_1, j_1) \neq (i_2, j_2), \quad \neg(x_{i_1,j_1,T} \wedge x_{i_2,j_2,T})$$

Explicitation pour chaque tuile :

$$\neg(x_{1,1,1} \wedge x_{1,2,1}) \quad \wedge \quad \neg(x_{1,1,1} \wedge x_{2,1,1}) \quad \wedge \quad \neg(x_{1,1,1} \wedge x_{2,2,1}) \ldots$$

Ce même schéma est appliqué pour **2, 3 et 4**.

---

c. Contraintes d'adjacence

Les bords des tuiles doivent correspondre avec ceux de leurs voisines.

Adjacence horizontale (droite/gauche)

Si une tuile T_1 est placée à **(i,j)** et une tuile T_2 à **(i,j+1)**, alors **Droite(T_1) = Gauche(T_2)** :

$$\neg x_{1,1,1} \vee \neg x_{1,2,2} \vee (4 = 4) \wedge$$

$$\neg x_{2,1,3} \vee \neg x_{2,2,4} \vee (1 = 2)$$

(On répète pour toutes les combinaisons de tuiles et positions.)

Adjacence verticale (haut/bas)

Si une tuile T_1 est placée à **(i,j)** et une tuile T_2 à **(i+1,j)**, alors **Bas(T_1) = Haut(T_2)** :

$$\neg x_{1,1,1} \vee \neg x_{2,1,3} \vee (2 = 2)$$

$$\neg x_{1,1,1} \vee \neg x_{2,1,4} \vee (2 = 3)$$

(On répète pour toutes les combinaisons de tuiles et positions.)

---

3. Formule globale en FNC

Nous combinons toutes les contraintes en une seule formule :

$$\bigwedge_{i,j} (x_{i,j,1} \vee x_{i,j,2} \vee x_{i,j,3} \vee x_{i,j,4})$$

$$\wedge$$

$$\bigwedge_{T} \bigwedge_{(i_1,j_1) \neq (i_2,j_2)} (\neg x_{i_1,j_1,T} \vee \neg x_{i_2,j_2,T})$$

$$\wedge$$

$$\bigwedge_{i,j,T_1,T_2} \left(\neg x_{i,j,T_1} \vee \neg x_{i,j+1,T_2} \vee (\text{Droite}(T_1) = \text{Gauche}(T_2))\right)$$

$$\wedge$$

$$\bigwedge_{i,j,T_1,T_2} \left(\neg x_{i,j,T_1} \vee \neg x_{i+1,j,T_2} \vee (\text{Bas}(T_1) = \text{Haut}(T_2))\right)$$

Cette formulation permet à un **solveur SAT** de résoudre le problème en déterminant une configuration valide pour les tuiles dans la grille.

4. Interface homme-machine et entrées/sorties

Le pipeline complet de résolution d'un puzzle est le suivant :

Tuiles Tetravex → tetravex_sat.ml → SAT solveur → interface.py → Affichage du puzzle résolu

a. Type OCaml d'une tuile et exemple d'entrée

L'entrée des tuiles **Tetravex** se fait en fournissant au programme une liste **OCaml** nommée tuiles contenant des éléments du type suivant :

```
(* Définition du type tuile *)
type tuile = {
  id : int;              (* numéro de la tuile *)
  haut : int;            (* Nombre associé à la face haute *)
  bas : int;             (* Nombre associé à la face basse *)
  gauche : int;          (* Nombre associé à la face gauche*)
  droite : int;          (* Nombre associé à la face droite *)
}
```

Ainsi qu'un entier n définissant la taille du puzzle (nombre de tuiles par côté).
On doit avoir : List.length tuiles = n * n.

Exemple d'entrée : Un ensemble de tuiles qui correspondrait potentiellement à une grille 2x2.

```
let tuiles = [
  {id=0; haut=1; bas=2; gauche=1; droite=2};
  {id=1; haut=1; bas=2; gauche=2; droite=3};
  {id=2; haut=2; bas=3; gauche=1; droite=2};
  {id=3; haut=2; bas=3; gauche=2; droite=3};
] in
let n = 2 in
```

b. Génération d'un nombre unique

```
(* Fonction pour générer un identifiant unique pour chaque variable *)
let var_id (i : int) (j : int) (t : int) (n : int) (nb_tuiles : int) : int =
  (i * n * nb_tuiles) + (j * nb_tuiles) + t + 1
```

Cette fonction a pour objectif de **générer un identifiant unique pour chaque variable logique** utilisée dans la modélisation du puzzle **Tetravex** en vue de sa résolution par un SAT-solveur.

L'idée est de transformer une combinaison d'indices (ligne i, colonne j, tuile t) en un **nombre entier distinct**. Cette transformation est essentielle car un solveur **SAT** ne manipule que des variables représentées par des entiers positifs.

Grâce à cette **formule mathématique injective**, chaque triplet d'informations est encodé de manière univoque. Cela garantit qu'il n'y a **aucune collision d'identifiants**, ce qui est fondamental pour exprimer correctement les contraintes du problème dans le fichier DIMACS. Ce mécanisme assure donc la **fiabilité et l'intégrité** de la traduction entre le problème combinatoire Tetravex et son équivalent en logique propositionnelle.

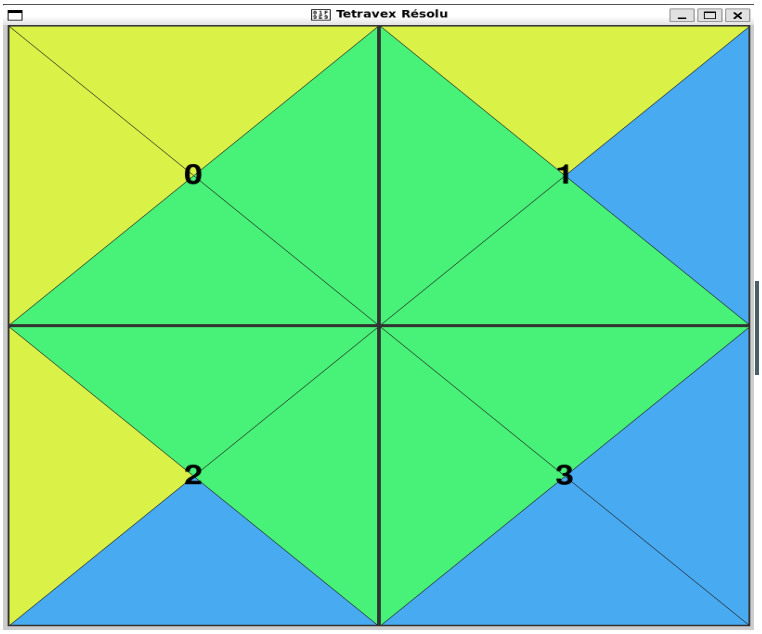
c. Format des sorties générées

Le programme tetravex_sat.ml génère une sortie dans un fichier au format **DIMACS**.
Ce format indique le nombre de variables et de clauses, puis énumère les clauses logiques.
Le fichier est largement commenté pour en faciliter la lecture humaine.
Le script interface.py prend ensuite en entrée un result.sat, qui est :

- Soit une ligne **SAT** suivie de l'affectation des variables (si les clauses sont satisfiables) ;
- Soit une ligne **UNSAT** (si aucune solution n'existe).

Exemple de sortie : result.sat généré par le SAT solver pour l'exemple d'entrée de **4.a**

```
SAT
1 -2 -3 -4 -5 6 -7 -8 -9 -10 11 -12 -13 -14 -15 16 0
```



*Grille **Tetravex** Généré par interface.py*

5. Jeux de tests

Dans le fichier tetravex_sat.ml, on trouve un **test de grille 2×2** permettant de :

- Vérifier manuellement la conformité de la sortie **DIMACS**,
- Tester la validité des contraintes générées,
- S'assurer du bon fonctionnement sans surcharger le **SAT**-solveur.

De plus, dans le dossier ex_inputs/, on dispose d'un ensemble de grilles **satisfiables et insatisfaisables** de différentes tailles.
Ces tests servent à :

- Évaluer la **scalabilité** de l'algorithme de transformation en **FNC**,
- Tester les **performances** du **SAT**-solveur développé pour ce projet.

Conclusion

Ce projet a permis d'illustrer concrètement la puissance de la logique propositionnelle appliquée à un problème combinatoire : la résolution du puzzle **Tetravex** via un encodage **SAT**. À travers différentes étapes — modélisation des contraintes, génération automatique des clauses en format **DIMACS**, résolution par un solveur **SAT** maison, et interprétation graphique du résultat — nous avons conçu une chaîne complète de traitement, de l'entrée brute à l'affichage utilisateur.
Ce travail nous a permis de manipuler des concepts fondamentaux de l'informatique théorique (FNC, satisfiabilité), de développer une interface homme-machine intuitive en Python, et d'intégrer plusieurs langages et outils **(OCaml, C, Python, JSON)** dans un projet cohérent. De plus, les tests réalisés sur plusieurs jeux de tuiles démontrent la solidité de l'approche et la scalabilité du système pour des tailles croissantes.
En conclusion, ce projet constitue un excellent exemple d'intégration entre algorithmique, logique et interface utilisateur, tout en mettant en avant la pertinence de la méthode **SAT** pour résoudre efficacement des problèmes **NP**-complets en pratique.

Sources et Ressources utilisées

- **Format DIMACS CNF**
  Pour la génération du fichier d'entrée du solveur SAT :
  https://wackb.gricad-pages.univ-grenoble-alpes.fr/inf402/satformat.pdf
- **Algorithmes de résolution SAT**
  Implémentation de base inspirée du **DPLL (Davis–Putnam–Logemann–Loveland)** et de l'étude de simplifications heuristiques usuelles.
- **Langage OCaml**
  Documentation officielle : https://ocaml.org/docs
- **Langage Python**
  Utilisé pour l'interface graphique. Référence : https://docs.python.org/3/
- **JSON (JavaScript Object Notation)**
  Format utilisé pour représenter les tuiles graphiquement :
  https://www.json.org/
- **Visualisation graphique en Python avec Pygame**
  Utilisé pour l'affichage graphique de la solution du puzzle :
  https://www.pygame.org/docs/
- **Inspiration du jeu Tetravex**
  Présentation du puzzle :
  https://en.wikipedia.org/wiki/Tetravex