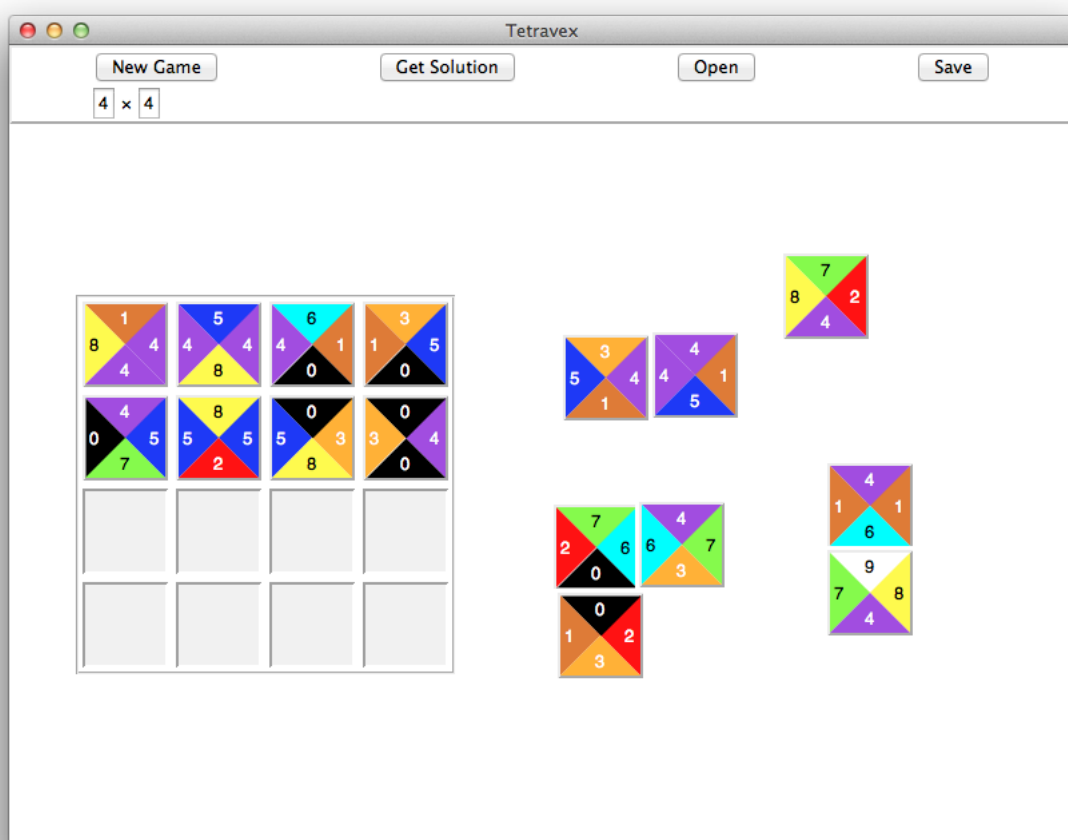


Projet de Programmation 2

TETRAVEX



Cédric Foucault

Introduction

Tetravex

Tetravex est un puzzle constitué d'une grille à remplir de taille $m \times n$ et d'un ensemble de $m.n$ tuiles carrées distinctes. Chaque tuile est scindée par ses diagonales en 4 faces qui possèdent chacune une couleur. Deux tuiles ne peuvent être posées côte à côte que si leurs faces adjacentes ont la même couleur, les faces au bord de la grille ne sont pas contraintes. Le but du jeu consiste à paver la grille avec les tuiles en respectant ces restrictions.

Objectif du projet

L'objectif de ce projet était double.

Il s'agissait d'une part de concevoir une librairie qui modélise et interface des données de puzzle Tetravex, permettant de générer et de résoudre des puzzles, de lire, d'afficher et de sauvegarder une instance dans un format spécifié (voir `README.txt`).

D'autre part, il fallait réaliser une interface graphique permettant à l'utilisateur de visualiser et d'interagir avec les données simplement.

J'ai été un peu plus loin en proposant une application de jeu complète réunissant les deux objectifs mentionnés mais permettant du même à l'utilisateur de réelement jouer au Tetravex.

Un effort particulier était demandé sur la structuration du programme, la documentation et la clarté du code.

Vue d'ensemble

Présentation du programme

Application de jeu

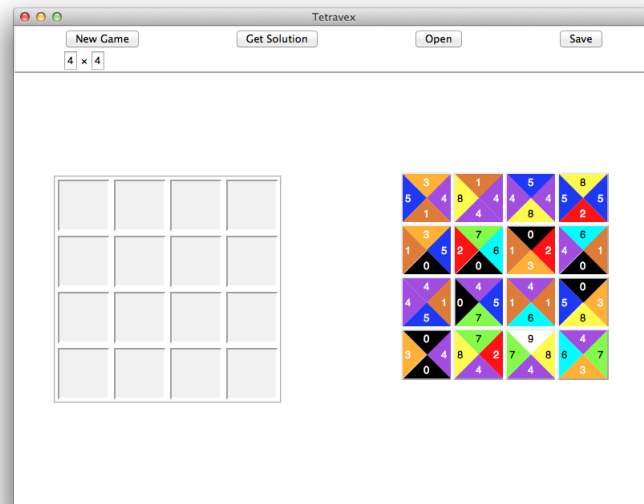


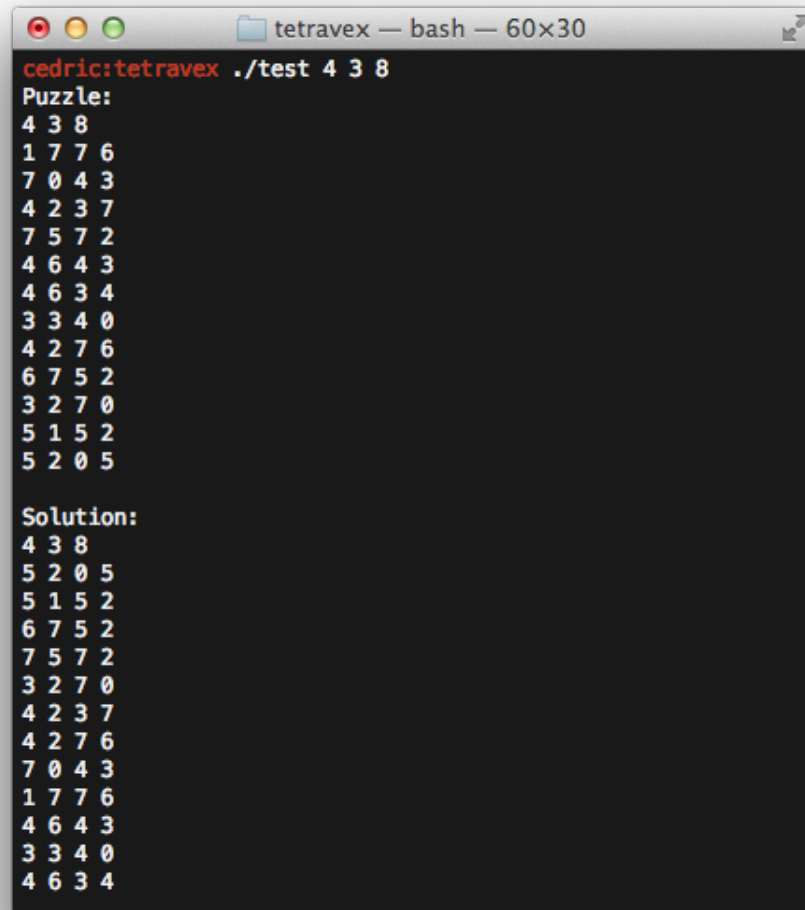
FIGURE 1 – Screenshot de l'application

L'application présente au joueur une grille de cases initialement vide, ainsi qu'un ensemble de tuiles aux faces colorées et numérotées. Celui-ci peut déplacer les tuiles par *drag-and-drop* n'importe où sur la surface de jeu et les assigner à une position sur la grille.

L'utilisateur peut demander au programme via le menu de générer un nouveau puzzle de la taille de son choix ou bien d'en ouvrir un depuis un fichier, de sauvegarder l'instance courante dans un fichier et d'afficher la solution sur la grille.

Librairie de résolution

Le programme s'appuie sur le module `TetravexModel` qui forme lui-même un ensemble indépendant et que l'on peut utiliser directement. Le projet comprend ainsi un script `test.ml` qui génère aléatoirement et résout des puzzles bien formés en affichant l'instance et la solution directement dans la sortie standard. La taille et le nombre de couleurs sont passés en argument.

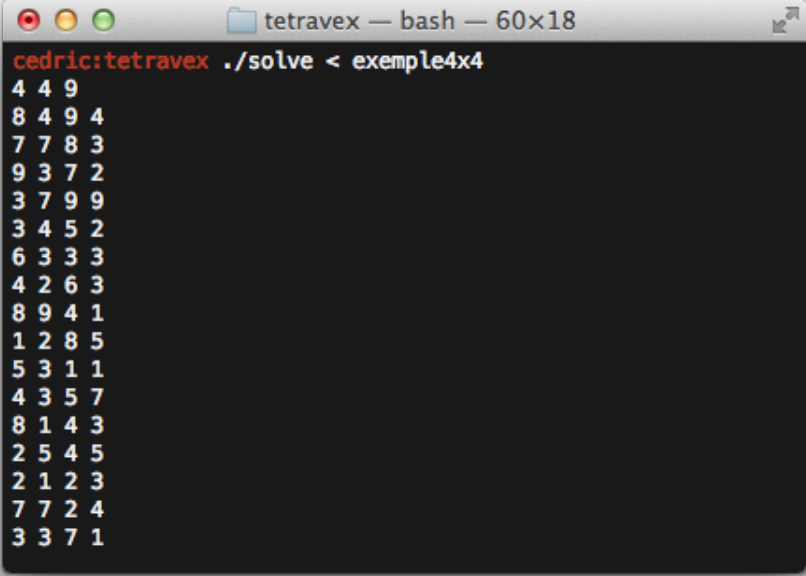
A terminal window titled "tetravex — bash — 60x30" with standard macOS window controls. The prompt is "cedric:tetravex". The command executed is "./test 4 3 8". The output shows a "Puzzle:" section with a 10x3 grid of numbers and a "Solution:" section with a 10x3 grid of numbers. The puzzle grid is: 4 3 8, 1 7 7 6, 7 0 4 3, 4 2 3 7, 7 5 7 2, 4 6 4 3, 4 6 3 4, 3 3 4 0, 4 2 7 6, 6 7 5 2, 3 2 7 0, 5 1 5 2, 5 2 0 5. The solution grid is: 4 3 8, 5 2 0 5, 5 1 5 2, 6 7 5 2, 7 5 7 2, 3 2 7 0, 4 2 3 7, 4 2 7 6, 7 0 4 3, 1 7 7 6, 4 6 4 3, 3 3 4 0, 4 6 3 4.

```
cedric:tetravex ./test 4 3 8
Puzzle:
4 3 8
1 7 7 6
7 0 4 3
4 2 3 7
7 5 7 2
4 6 4 3
4 6 3 4
3 3 4 0
4 2 7 6
6 7 5 2
3 2 7 0
5 1 5 2
5 2 0 5

Solution:
4 3 8
5 2 0 5
5 1 5 2
6 7 5 2
7 5 7 2
3 2 7 0
4 2 3 7
4 2 7 6
7 0 4 3
1 7 7 6
4 6 4 3
3 3 4 0
4 6 3 4
```

FIGURE 2 – Une exécution de `test`

Un autre script `solve.ml` lit une instance dans l'entrée standard, la résoud et l'affiche dans la sortie standard.



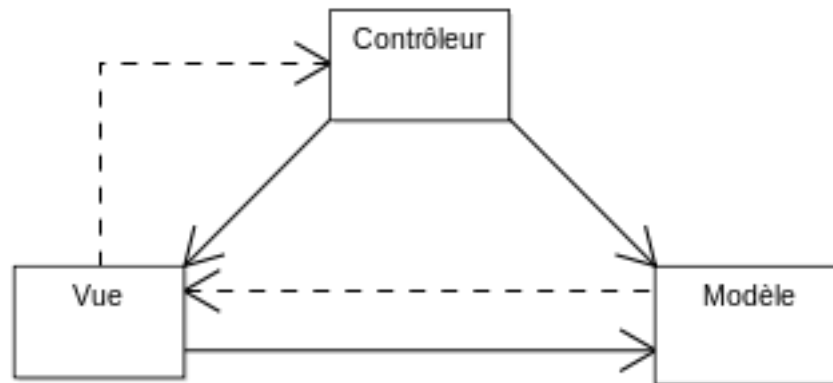
```
cedric:tetravex ./solve < exemple4x4
4 4 9
8 4 9 4
7 7 8 3
9 3 7 2
3 7 9 9
3 4 5 2
6 3 3 3
4 2 6 3
8 9 4 1
1 2 8 5
5 3 1 1
4 3 5 7
8 1 4 3
2 5 4 5
2 1 2 3
7 7 2 4
3 3 7 1
```

FIGURE 3 – Une exécution de `solve`

On peut écrire d'autres scripts en utilisant les fonctions de ce module, par exemple pour générer et tenter de résoudre des problèmes qui n'ont pas nécessairement de solution.

Architecture du programme : Modèles, Vues, Contrôleur

L'organisation du programme suit le *pattern* Modèle-Vue-Contrôleur, très adapté à ce type d'application. Il s'agit d'une correspondance directe, puisque le projet est composé des trois modules `TetravexModel`, `TetravexView`, et `TetravexController`.



Rien de mieux qu'un exemple pour illustrer les interactions entre ces trois modules.

Déroulement de l'évènement où l'utilisateur demande à l'application de générer une nouvelle instance :

- L'utilisateur saisit les valeurs de m et n dans entrées et clique sur le bouton "Get Solution". Les entrées et les boutons sont des *vues* du la *vue* `TetravexView.Menu`. La vue reçoit l'action "clic de bouton" et envoie cet évènement au contrôleur.
- Le *contrôleur* prend en charge cet évènement et enlence les actions à effectuer.

Il appelle les fonctions de génération de `TetravexModel.Puzzle` pour créer deux nouveau *modèles*, de type `TetravexModel.Puzzle.t`, un pour l'ensemble de tuiles (l'instance du problème), l'autre pour la grille (initialement vide).

- Le *contrôleur* crée deux nouvelles vues qu'il lie à ces modèles, l'ensemble de tuiles correspondant à la *vue* `TetravexView.tile_set` et la grille à la *vue* `TetravexView.puzzle_grid`. Le *contrôleur* affiche les nouvelles *vues* à l'ecran à l'endroit voulu.

Cette architectre confère une grande maintenabilité au projet. Si l'on veut par exemple modifier le design graphique des tuiles ou du menu, il suffit de modifier la vue correspondante, sans toucher aux autres modules et sans se soucier d'autre chose que des données purement graphiques. Un autre avantage de cette architecture et de pouvoir procurer plusieurs vues pour un modèle, par exemple un puzzle modélisé par `TetravexModel.Puzzle` peut correspondre à la vue "grille", instance de `TetravexView.puzzle_grid`, où à la vue "ensemble de tuiles", instance de `TetravexView.tile_set`.

Pour plus de détails sur chacun des modules, on se réfèrera à leur documentation respective, générée grâce à `ocaml doc`. Toutes les fonctions de ces modules y sont expliquées.

Application de jeu et interface graphique

J’ai réalisé l’interface graphique à l’aide de la librairie `labltk` (bindings Tcl/Tk pour OCaml).

À chaque nouveau puzzle chargé, le calcul de la solution commence directement après son affichage à l’écran, et s’effectue dans un nouveau thread. Ainsi, lorsque le joueur demande à obtenir la solution, il ne reste à faire qu’une recopie sur la grille de la solution précalculée (en attendant néanmoins la terminaison du thread si le calcul n’est pas fini). Cela permet de conserver une interface graphique réactive, du moins en théorie (on observerait une latence lorsque l’on cliquerait sur le bouton “Get Solution” si l’on ne calculait qu’à ce moment-là et que le calcul est de l’ordre de la seconde). Bien entendu, cela ne règle le problème que si le programme a le temps de calculer la solution entre le moment où le joueur charge un nouveau puzzle et demande sa solution.

J’ai préféré que le joueur puisse disposer librement des tuiles plutôt que de les organiser en 2 grilles comme cela se fait dans d’autres applications car cela lui donne un bon confort de jeu et une meilleure souplesse, en lui permettant par exemple d’organiser les tuiles en petits groupes pour visualiser et trouver plus rapidement la solution.

Par contre, cela complexifie un peu le code, et notamment la gestion du *drag-and-drop* qui, du coup, est asymétrique. Pour l’implémenter, j’ai dû gérer individuellement les événements `press`, `motion`, `release` des boutons de la souris, et les lier en mémorisant temporairement les éléments concernés. Lorsque l’utilisateur déplace une tuile de la grille, celle-ci est supprimée de la grille et est ajoutée à nouveau à l’ensemble de tuiles du jeu. À cause de cela, et parce que je n’ai pas trouvé comment rediriger le *focus* d’un événement `motion` entre un événement `press` et un événement `release`, le comportement du *drag-and-drop* pour retirer une tuile de la grille est un peu spécial : l’utilisateur doit d’abord cliquer sur la tuile concernée (`press` + `release`), puis la déplacer avec la souris sans re-cliquer, et enfin cliquer une deuxième fois pour la déposer à l’endroit voulu.

J’ai donné la possibilité au joueur de choisir la taille de l’instance de puzzle qu’il souhaite obtenir, mais pas celle de changer le nombre de couleurs différentes des tuiles générées car j’ai jugé que cela ne correspondait pas à un besoin réel du joueur (en plus, la palette de couleurs rendue n’aurait pas été si jolie :P). Ce paramètre peut par contre être modifié à souhait lorsqu’on interagit directement avec la librairie `TetravexModel`.

Résolution du puzzle

Le problème de la résolution d’un puzzle Tetravex peut se modéliser comme un problème de satisfaction de contraintes (tout comme SAT, le pro-

blème des huit dames, Sudoku, et bien d'autres...).

Implémenter cette modélisation pour effectuer une résolution de contraintes, plutôt que de résoudre directement par recherche exhaustive, présente plusieurs avantages :

Clarté Une fois le problème bien défini, sa traduction effective en code est directe puisque la programmation par contraintes consiste à **déclarer** explicitement puis à demander à les résoudre.

Généralité On a abstrait le problème de telle manière que la résolution effective est la même que pour n'importe quel problème. On peut ainsi imaginer résoudre des variantes de ce puzzle avec des règles légèrement différentes, il suffirait alors de modifier les contraintes

Efficacité Cette résolution est efficace *a priori* parce que les problèmes de satisfaction de contraintes ont été étudiés abondamment et que l'on dispose d'algorithmes qui, par propagation de contraintes, envisagent le moins de solutions possible. Cependant, leur efficacité est très dépendante des contraintes posées, *i.e.* de la manière dont on modélise le problème, ainsi que de l'ordre dans lequel on les exprime (le

graphe de contraintes résultant sera différent).

Par exemple, dans ma première modélisation j'avais contraint les tuiles incompatibles à ne pas être voisines plutôt que de poser des contraintes d'égalité, et la résolution était beaucoup plus lente (de l'ordre de 400 fois plus lente). J'ai aussi accéléré ma résolution en éliminant les redondances.

Modélisation

J'ai modélisé par le problème de satisfaction de contraintes par le triplet (Pos, D, C) , où :

- Pos , l'ensemble des variables, est l'ensemble des positions de chaque tuile. Les tuiles et les cellules étant numérotées de 1 à $m.n$, on dit que la tuile k est positionnée à la cellule (i, j) si $Pos(k) = (i, j)$.
- D , le domaine de valeurs, est le domaine des cellules, numérotées de 1 à $m.n$: $D = \{1, \dots, m.n\}$.
- C est l'ensemble des contraintes :

All Different Traduit le fait que chaque tuile doit être placée à une position différente. Cela ne fait que contraindre l'ensemble des valeurs de positions à l'ensemble des permutations de $\{1, \dots, n\}$

Color Match Traduit le fait que deux tuiles ne peuvent être voisines que si leurs couleurs adjacentes correspondent. On distingue les contraintes horizontales des contraintes verticales.

Horizontalement, soit une tuile est sur le bord droit de la grille, soit sa tuile voisine à droite est l'une des tuiles dont la couleur gauche est égale à la couleur droite de la tuile.

Soit les contraintes :

$$C_{hor,k} = Colonne(k) = n \vee \bigvee_{k' \setminus droite(k)=gauche(k')} (Pos(k) = Pos(k') - 1)$$

Verticalement, on a par analogie les contraintes :

$$C_{ver,k} = Ligne(k) = m \vee \bigvee_{k' \setminus bas(k)=haut(k')} (Pos(k) = Pos(k') - n)$$

Ces contraintes sont nécessaires et suffisantes (par décalage d'indice).

Ces contraintes permettent d'élaguer considérablement l'arbre de recherche.

Par exemple, si une tuile k n'a aucune tuile compatible à droite dans l'ensemble des tuiles, elle sera directement placée sur le bord droit de la grille. Aussi, à chaque fois que l'on fixe une tuile à un endroit de la grille, par **propagation de contraintes** cela va supprimer des possibilités dans les contraintes C_{hor} et C_{ver} , et le système va arriver à de nouvelles déductions (fixer d'autres positions, ou arriver à une impossibilité et backtracker).

Pour effectuer de la programmation par contraintes en OCaml, j'ai utilisé la librairie FaCiLe (Functional Constraint Library)¹, qui permet un style proche de Prolog.

Résultats

Pour une instance de taille 4×4 avec $c = 10$ couleurs différentes, sur ma machine personnelle, la résolution est de l'ordre de 0.1s en moyenne. Avec le même c , j'ai pu résoudre des instances 6×3 en temps $< 1s$. Ce temps a tendance à croître si l'on diminue c , puisque l'on va avoir plus de compatibilités de tuiles et on pourra moins élaguer, la durée est critique entre 2 et 5. Inversement, en augmentant c le temps diminue, si l'on met un c très grand (ex : 20) on peut même résoudre des instances 5×5 , les couleurs "parlent d'elles-mêmes".

Génération

On peut générer un puzzle bien formé (*i.e.* qui a une solution) en appliquant le principe récursif suivant :

Un puzzle est bien formé si sa première ligne est constituée de tuiles horizontalement compatibles et son sous-puzzle bas est bien formé et est verticalement compatible avec la première ligne.

1. <http://www.recherche.enac.fr/opti/facile/>

Conclusion

En suivant une architecture rigoureuse, ce projet m'a appris à structurer et à organiser mon code de façon cohérente et efficace.

Du fait des nombreuses parties à implémenter, de leur natures diverses et de la polyvalence du langage OCaml, il m'a appris à utiliser plusieurs styles de programmation (classes ? modules ? impératif ? fonctionnel ? déclaratif ?) et à les interfacer.

Enfin, il m'a surtout appris à me débrouiller avec une librairie (`lablgtk`) dont la documentation est quasi-inexistante ! (J'ai été regarder directement les sources de la librairie, en les comparant avec la documentation officielle de Tcl/Tk dans d'autres langages)