Création d'une Interface Graphique Pour la Résolution de Puzzles

Projet de Techniques de Développement Logiciel

Matthieu ROUX, Théo VIEL

2017-2018

1 Introduction

Dans le cadre de notre projet MoPSi, nous avons implémenté deux algorithmes de résolution d'un puzzle particulier. Le premier aborde le problème de façon probabiliste et trouve une solution par avec un **recuit simulé**, et le deuxième est déterministe et énumère l'ensemble des solutions par la méthode des **dancing links**.
Le puzzle étudié est un puzzle bidimensionnel. Il s'agit d'un plateau carré sur lequel le but est d'encastrer toutes les pièces. Il y a deux types de pièces:

- Un cylindre (bloc), qui a pour caractéristique de ne pas pouvoir être déplacé au cours de la résolution. On choisit de le placer avant la résolution.
- Les autres pièces sont des **polyominos**: une réunion de carrés unitaires ayant une arrête coïncidente au moins.

Le puzzle qui nous a été présenté est de taille 7x7 (voir figure 1). Nous avons généralisé l'étude à d'autres tailles.

Figure 1: Le puzzle étudié.

2 Objectifs

Le but de notre projet est de développer une interface graphique pour les deux algorithmes afin d'exploiter au maximum le potentiel de la résolution de ce puzzle. Le premier point était tout d'abord d'obtenir une mise en forme interactive pour la méthode probabiliste. En effet, dans le cadre du projet MoPSi, nous avons utilisé le module `Pygame` qui permettait d'afficher naïvement la solution une fois le programme exécuté. Une fois lancée, l'interface graphique a pour but premier de permettre les interactions élémentaires suivantes:

- Lancer et mettre en pause l'exécution
- Modifier la position du bloc
- Modifier le choix du voisinage pour le recuit.
- Modifier la période d'affichage

Ensuite, nous avons proposé deux modes d'utilisation supplémentaire, permettant d'ajouter de la difficulté pour la résolution de puzzles:

- L'utilisateur peut créer son propre puzzle
- Il peut également placer des pièces d'un puzzle existant sur la grille

Enfin, le dernier point est d'importer les résultats obtenu par l'algorithme des des dancing links, qui a été implémenté en `C++`, et de les afficher dans notre interface.

3 Choix techniques

Dans la continuité des TPs du cours de technique de développement logiciel, il nous a paru logique de développer notre interface sous `Python`, en utilisant la bibliothèque `PyQt`. Nous avons opté pour une interface de départ Modèle-Vue-Contrôleur (MVC), que nous détaillerons après.
Le problème de Python se trouve être ses performances, pour le recuit simulé cela ne devait pas poser trop de problèmes, mais il en est tout autrement pour la méthode déterministe, sur laquelle nous visons des performances convenables. Cette dernière a été implémentée en `C++` et utilise les pointeurs et l'allocation dynamique permise par ce dernier.

Le dernier point de notre projet, une fois l'interface développée, consiste à pouvoir appeler les fonctions développées en `C++` depuis celle-ci. Nous avons opté pour l'appel depuis un exécutable. Ce choix d'un exécutable n'était pas le premier retenu, car nous avions tout d'abord envisagé d'encapsuler les fonctions `C++` dans un package Python. Néanmoins, pour des raisons que nous détaillerons par la suite, nous avons opté pour un compromis moins élégant, mais qui a le mérite de remplir son rôle.

4 L'interface graphique Python

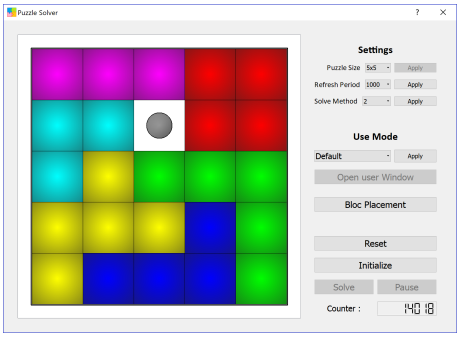

Figure 2: L'interface graphique PyQt

4.1 Modèle

La partie modèle s'inscrit dans la partie MoPSi de notre projet, c'est pourquoi nous ne la détaillerons que très peu dans le rapport Voici cependant le diagramme des classes UML utilisé pour représenter le problème. Le bloc est le cylindre que l'on ne peut pas bouger, la grille est l'endroit où l'on place les pièces. Les fonctions fondamentales sont celles qui définissent les voisinages, et se trouvent dans la classe `Piece`. Le reste du modèle se trouve dans le fichier `Fonctions.py` et sort totalement de la partie développement logiciel du projet. Pour plus d'informations, les codes sont commentés et nous vons invitons à les consulter, ou à regarder le poster réalisé pour le projet MoPSi.

Piece		
mat		color
x		y
__init__(self)		
permut_possible(self, grid, piece)		
permut(self, grid, piece)		
varV_permut(self, grid, piece)		
rotate_possible(self, grid)		
rotate(self, grid)		
varV_rotate(self, grid)		
move_possible(self, grid, dir)		
move(self, grid, direction)		
varV_move(self, grid, direction)		

Grid
n
m
__init__(self, n, m)
clear(self)
affiche(self)
add_piece(self, piece)
add_bloc(self, bloc)
V(self)
refresh(self)

Bloc
x
y

Figure 3: Diagramme des classes UML du modèle

4.2 Fonctionnalités Vue-Contrôleur

Notre interface permet les interactions suivantes :

- Changer la taille du puzzle résolu : de 3x3 à 7x7
- Changer la période d'affichage des pièces (pour le recuit)
- Changer la méthode de résolution
- Placer le bloc (détaillé après)
- Créer des pièces (détaillé après également)
- Placer des pièces (idem)
- Lancer, réinitialiser et mettre en pause la résolution
- Afficher le nombre d'itérations
- Ouvrir une fenêtre en cas de résolution ou d'échec

Celles non détaillées dans la suite sont implémentées de façon assez simple, chaque bouton est lié à une fonction correspondante. Chaque fonction assure que les boutons ne sont activés que quand leur utilisation est appropriée.

5 Fonctions interactives de l'interface graphique

5.1 L'interface de placement du bloc

Cette interface auxiliaire est la première que nous avons implémentée. Pour elle ainsi que les 2 suivantes, nous avons opté pour une interface Modèle-Vue-Contrôleur simplifiée, au sens où pour ne pas rendre le projet trop lourd, la vue et le contrôleur sont regroupés dans la même classe. La partie Vue étant principalement contenue dans la fonction `__init__`.
Son but est de laisser l'utilisateur placer le bloc. On rappelle qu'on a appelé bloc la pièce cylindrique du puzzle, et qui est fixe au cours de la résolution.
Le fonctionnement est très simple, l'utilisateur clique dans une case que la fonction `getPos` identifie, et la fonction `DrawCercle` dessine le cercle en cet endroit. Une fois la position choisie, l'utilisateur clique sur le bouton Ok, qui est lié à la fonction `Ok` qui permet de transmettre la position du bloc choisie à l'application.

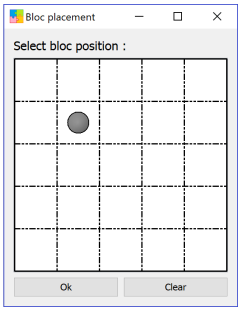

Figure 4: Interface Bloc Placement

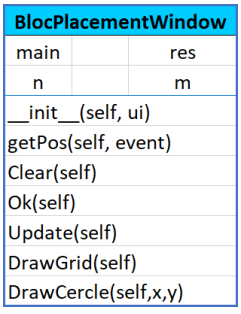

Figure 5: La classe associée

5.2 L'interface de placement de pièces

Le but de cette interface est de permettre à l'utilisateur de pré-positionner des pièces du puzzle afin de voir si l'algorithme trouve une solution à partir de cette configuration. L'interface se sépare en trois :

- À gauche la fenêtre de placement de la pièce
- À droite la fenêtre de sélection de la pièce
- En bas les boutons de validation/annulation.

Le fonctionnement est un peu plus complexe que l'interface de placement du bloc, mais repose sur les mêmes idées.

Avec l'interface de droite, l'utilisateur choisit une pièce à partir du menu défilant et clique sur Apply, connecté à `PreviewPiece`, qui l'affiche dans la fenêtre en dessous. Il peut choisir l'orientation en cliquant sur le bouton Rotate qui est connecté à la fonction `RotatePiece`. Ensuite, l'utilisateur clique sur une case de la grille correspondant à l'endroit où il souhaite placer le coin supérieur gauche de la pièce sur la pré-visualisation.
Quand une case est cliquée, la fonction `Placepiece` l'affiche à condition qu'il n'y ait pas de conflit détecté par `Placable`, c'est à dire si on ne sort pas de la grille, ou si on ne se trouve pas sur une des cases occupées (stockées dans la liste `occupied`). Pour confirmer un emplacement, l'utilisateur clique sur Confirm Placement qui appelle la fonction associée `ConfirmPlacement`. Cette fonction met `occupied` à jour, ajoute cette pièce à `placedpiece` et l'enlève du menu défilant. On peut ensuite placer une autre pièce.
Quand le bouton Ok est cliqué, la fonction du même nom est appelée et le contrôleur stocke dans la liste `fixedpieces` les pièces qu'on ne pourra pas bouger dans la recherche de solution, et les retire de la liste `pieces`.

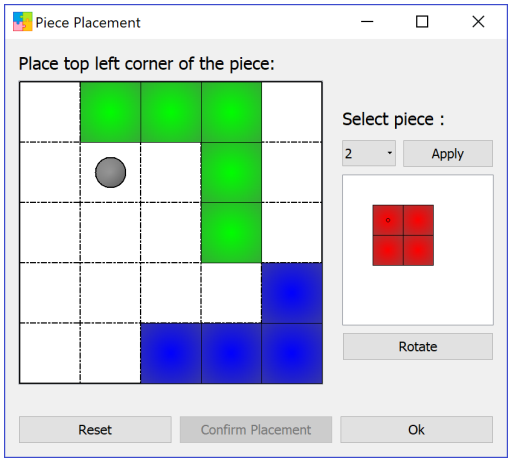

Figure 6: Interface Piece Placement

PiecePlacementWindow		
main		res, resp
index		pieces
occupied		placedpieces
__init__(self, main)		
PreviewPiece(self)		
AffichePiecePreview(self, piece)		
RotatePiece(self)		
getPos(self, event)		
Placepiece(self,x,y)		
Placable(self,piece, x, y)		
AffichePiece(self, piece)		
ConfirmPlacement(self)		
Reset(self)		
Refresh(self)		
Ok(self)		
Error(self, type)		

Figure 7: La classe associée

5.3 L'interface de création de pièces

Cette interface permet à l'utilisateur de dessiner son propre puzzle. Elle met à disposition une grille de la taille du puzzle dans laquelle il peut dessiner une pièce, en cliquant sur les cases choisies. Une des contraintes est la cohérence de la forme de la pièce, vérifiée par la fonction `connexe`. Les cases cliquées sont stockées dans la matrice `mat`. Si le dessin de la pièce convient à l'utilisateur, il clique sur le bouton Add Piece, sinon sur Clear. Cette première appelle la fonction `Addpiece` qui convertit la matrice des clics `mat` en pièce avec la fonction `matToPiece` et l'ajoute à la variable `pieces`.
Le compteur en haut de la fenêtre indique à l'utilisateur le nombre de cases que l'utilisateur peut utiliser pour dessiner des pièces. Il est mis à jour par la fonction `RefreshCompt` . En effet, une condition nécessaire pour que notre puzzle de taille n soit résolu est que les pièces et le bloc remplissent les n^2 cases. Une fois cette condition satisfaite, le bouton Ok est débloqué. Il transmet via la fonction du même nom la nouvelle liste de pièces au contrôleur.

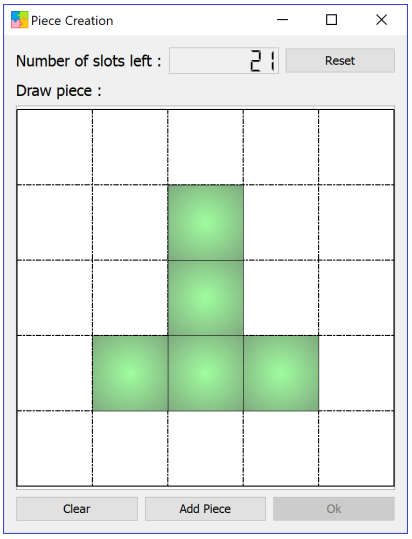

Figure 8: Interface Piece Creation

PieceCreationWindow		
main		res
mat		pieces
compt		clics
__init__(self, main)		
getPos(self, event)		
Addpiece(self)		
MatToPiece(self)		
connexe(self, mat)		
SetNewCol(self)		
Update(self)		
RefreshCompt(self, int)		
Ok(self)		
Clear(self)		
Reset(self)		
Error(self, type)		

Figure 9: La classe associée

Les 3 interfaces peuvent être utilisées ensemble,
Ceci à condition de suivre l'ordre naturel :
Création des pièces → Placement du bloc → Placement des pièces

6 Interfaçage C++ et fenêtre graphique Python

6.1 Objectifs et grandes lignes

Comme dit précédemment, nous avons choisi les langages de programmation que nous maîtrisions et qui nous semblaient les plus appropriés à chaque tâche, à savoir une interface graphique utilisateur et des calculs efficaces. Le choix de calculs en C++ s'est fait naturellement lorsqu'il a fallu implémenter une méthode de calcul relativement lourde, et principalement basée sur les notions de pointeurs avec de grosses quantités de données. Mais il fallait ensuite être capable de faire communiquer ce code avec le code Python. Étant donné nos objectifs globaux initiaux, l'interfaçage devait essentiellement réaliser deux fonctions :

- Faire passer les résultats finaux obtenus par le code C++ vers le code Python
- Avoir une communication entre les 2 langages suffisamment établie afin qu'il soit possible que le code Python connaisse également les états intermédiaires du code C++ (et pas uniquement la solution finale)

6.2 Première approche : la librairie Boost.python

Dans un premier temps, nous nous sommes rendu compte qu'une bonne manière de faire communiquer les deux langages pouvait être d'utiliser la librairie **Boost.python**. En effet, celle-ci permet de compiler des bibliothèques de code C++ en des packages compréhensibles par Python. Cette méthode était donc parfaitement adaptée puisqu'elle permettait une encapsulation totale des fonctionnalités de calcul dans le code Python et donc aurait été très pratique pour communiquer sur des résultats intermédiaires. Malheureusement, nous avons rencontré un grand nombre de difficultés lors de la mise en oeuvre de cette méthode.

En effet, la procédure d'installation de la librairie Boost.python est assez complexe, basée sur des formats .bjam. Cette bibliothèque doit être compilée pour pouvoir être utilisée, mais la documentation étant inexistante sur la procédure d'installation et de première compilation sous MinGW, nous avons du demander de l'aide à M. Martinez et M. Monasse :

6.2.1 Sous Windows

Nous avons d'abord tenté une installation sous Windows. Néanmoins, les procédures glanées sur les quelques forums ne nous permirent pas d'aboutir, malgré diverses tentatives. Il semblerait que la bibliothèque n'ait pas été construite pour être aisément compilée avec MinGW.

6.2.2 Sous Linux

Conscients que "tout est plus facile" sous Linux, notre second réflexe a donc été de tenter avec ce système d'exploitation. Malheureusement, malgré les nombreux efforts de notre encadrant M. Martinez pour nous aider à comprendre cette librairie (et que nous remercions chaleureusement au passage), nous n'y sommes pas parvenus. En fait, nous avons finalement pu compiler un bout de code et réaliser une encapsulation correcte, mais tout était fait dans le terminal, avec une longue suite d'instructions que nous ne comprenions pas toujours très bien. Incapables de produire un CMakeLists propre, nous avons finalement décidé d'abandonner cette méthode, car nous y avions déjà consacré beaucoup de temps, et que le partage du code serait beaucoup plus difficile si nous ne disposions pas de CMakeList robuste afin de pouvoir le partager avec d'autres machines, sans devoir recommencer la procédure dans le terminal, et qui devrait être adaptée pour chaque machine.

6.3 Seconde approche : appel à un exécutable

Comme nous avions passé beaucoup de temps déjà sur Boost, nous avons décidé d'opter finalement pour une solution certes moins élégante, mais qui au moins avait le mérite de marcher (et dans un premier temps nous nous devions d'avoir au moins une première version du logiciel qui fonctionne, quitte à la faire évoluer plus tard) : passer par un **exécutable** appelé par Python. Le principe est simple: on compile un exécutable `C++`, que le code Python devra ensuite appeler. Se pose alors la question de la communication, et du transfert de données. Comment envoyer les paramètres du problème, définis dans l'interface graphique, au code `C++` ? Et comment renvoyer les résultats obtenus ?

6.3.1 Données d'entrée

Premièrement, nous avons choisi d'envoyer les données de définition du problème, dans le sens `Python` vers `C++`, avec un fichier au format `.txt`. Il doit contenir des informations sur :

- La taille de la grille du puzzle
- Le nombre de pièces
- Le nombre de cases fixes (identifiées par leur numéro)
- La définition des pièces du puzzle

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 10: Méthode d'identification des cases pour une grille 5x5

```
#taille plateau
N

#nombre de cases fixees
nbCasesFixes
#numero des cases fixees
n1;n2...;nk

#nombre de pieces
nbPieces

#Pour piece pi
#dimensions hauteur x largeur
hauteur; largeur
#cases occupees = 1
0;1;1
1;1;1
```

Description du format d'entrée des données

```
7

1
10

9

1;5
1;1;1;1;1

3;3
0;1;1
1;1;1
0;1;1

2;1
1
1
```

Exemple de fichier entrée

Figure 11: Format d'entrée des données du puzzle

6.3.2 Données de sortie

Après le calcul des solutions par le code `C++`, il faut donc envoyer les résultats à l'interface Python. Pour ce faire, on utilise le format .csv afin de décrire toutes les solutions trouvées. De plus, on produit un deuxième fichier .csv de sortie, contenant le résumé des informations sur le calcul effectué (synthèse de la structure de l'arbre d'exploration). De manière simple, une solution est décrite par la description des cases occupées par chaque pièce :

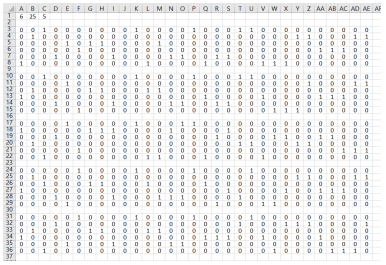

Figure 12: Format de sortie : description des solutions

6.3.3 Le problème de la communication en des états intermédiaires du calcul

Le seul inconvénient de cette méthode d'interfaçage, passant par un exécutable, est qu'elle rend plus difficile la communication sur des résultats intermédiaires. En effet, lorsque `Python` lance l'exécutable, il n'y a guère, de moyen simple pour permettre une communication en temps réel sur les états intermédiaires du code de calcul `C++`. Par conséquent, nous avons malheureusement été contraints d'abandonner cet objectif.

7 Conclusion

Pour conclure, nous pouvons dire que nous avons globalement rempli les objectifs tels qu'ils étaient au départ. Notre interface graphique permet bien de calculer les solutions de puzzles relativement simples, en choisissant la méthode de résolution, avec également d'autres paramètres. Il est également possible de définir son propre puzzle, ainsi que de voir s'il existe des solutions à un puzzle partiellement résolu. Néanmoins, il faut reconnaître que certaines difficultés ont été difficiles à lever.
La librairie `Pyqt4` reste obscure en certains points. Nous avons réussi à lever un problème d'overflow qui faisait que le programme ralentissait sans aucune raison, en ajoutant des fonctions adéquates sans bien comprendre pourquoi le problème était résolu. De plus, l'interface graphique développée s'affiche différemment selon la machine depuis laquelle elle est exécutée. Si le projet était à refaire nous utiliserions une autre bibliothèque d'affichage que `Pyqt`.

Le problème de l'interfaçage `Python-C++` nous a pris beaucoup de temps, pour vraiment peu de résultats. Dans un soucis de pragmatisme, nous avons décidé de contourner le problème en passant par un exécutable, mais malheureusement cela s'est fait au détriment de l'objectif de suivi de la méthode déterministe, car on ne peut finalement pas accéder aux états intermédiaires du code déterministe `C++`.
Une autre solution à explorer consiste à utiliser `SWIG`, ce que nous souhaitions faire mais n'avons pas pu, faute de temps.

Ce projet de développement logiciel fut malgré tout très enrichissant. Nous avons entre autre pris nos marques avec le logiciel de versioning Git, et ainsi développé les compétences nécessaires du travail en groupe autour d'un projet informatique. De plus, même si l'objectif est partiellement atteint de ce côté-là, le temps investi à comprendre comment interfacer efficacement deux langages de programmation saura nécessairement nous être utile par la suite.

8 Remerciements

Tout d'abord à Pierre-André Zitt pour son aide au cours du projet MoPSi sans lequel ce projet TDLog n'aurait pas eu lieu.
Ensuite à Pascal Monasse et à Thierry Martinez pour leur aide sur les problèmes d'interfaçage.
Merci également au KI Club Info pour la formation `Git` et en particulier à Guillaume Desforges pour ses conseils.