



Projet Informatique 1ère Année
CSC 3502

Projet Puzzle Eternity II

Antoine LACHAUD
Charles-Emmanuel MARTIN
Alban NICOLAS
Yann SIONNEAU

Enseignant responsable : **Sébastien LERICHE**, département Informatique

Sommaire

1 Introduction.....	3
2 Cahier des charges.....	3
3 Développement.....	4
3.1 Analyse du problème et spécification fonctionnelle.....	4
3.2 Conception préliminaire.....	4
3.3 Conception détaillée.....	8
3.3.1 Chargement du fichier.....	8
3.3.2 Interface Graphique.....	12
3.3.3 Module de résolution.....	16
3.3.4 Divers.....	20
3.4 Tests unitaires.....	21
3.4.1 Module chargement.....	21
3.4.2 Module IHM.....	21
3.4.3 Module Résolution.....	22
3.5 Tests intégration.....	23
3.5.1 Interaction Chargement/IHM.....	23
3.5.2 Interaction Résolution/IHM.....	23
3.5.3 Interaction Chargement/Résolution.....	24
3.5.4 Interaction Chargement/Résolution/IHM.....	24
4 Conclusion.....	25
5 Bibliographie.....	26
6 Annexes.....	27
6.1 Annexe 1 : Gestion de projet.....	27
6.1.1 Plan de charge.....	28
6.1.2 Suivi d'activités.....	29
6.2 Annexe 2 : Code source.....	30

1 Introduction

D'après le site officiel, Eternity II est un jeu unique en son genre : mi puzzle, mi jeu de logique, il est composé de 256 pièces carrées portant chacune 4 motifs géométriques parmi 22 motifs de base, en plus des bords. Il s'agit d'agencer correctement ces pièces de manière à faire correspondre les motifs entre eux.

La complexité du jeu est telle qu'une résolution manuelle n'est pas envisageable dès que le nombre de pièces devient grand. Même pour une taille très réduite du jeu (par exemple le 4x4 à essayer en ligne), plusieurs minutes sont nécessaires pour en venir à bout.

Nous avons chacun été intéressé par ce projet pour différentes raisons. Antoine, obnubilé par l'immensité théorique de l'espace des solutions, a été motivé par le côté mathématique théorique de la résolution. Charles, à qui l'exemple en ligne n'a pas résisté bien longtemps, souhaitait vérifier si son raisonnement intuitif pouvait se généraliser à des tailles plus grandes à l'aide de l'informatique. Alban et Yann, intéressés par la programmation avant tout, n'avaient pas d'aversion particulière pour ce projet, et se sont donc joints à l'équipe.

2 Cahier des charges

Le but du projet est de développer une application informatique capable de résoudre les puzzles indices du jeu Eternity II, qui soit doté d'une interface graphique permettant de visualiser des grilles solution. Il faut pour cela mettre en oeuvre une application complète écrite en C, avec des aspects variés (interface graphique GTK, gestion de fichiers, algorithmique...).

Le logiciel à développer doit répondre aux critères suivants:

- avoir une interface graphique claire
- respecter le format des fichiers
- être capable de résoudre un puzzle 6x6 dans un temps raisonnable

Il faut pour cela développer en parallèle l'interface et l'algorithme de résolution, faire les tests unitaires pour permettre de valider le logiciel. Le point fort du projet reste toutefois la partie liée à la résolution du puzzle, qui nécessite un bon investissement au niveau algorithmique.

3 Développement

3.1 Analyse du problème et spécification fonctionnelle

L'enjeu principal de notre projet est d'optimiser la recherche de la solution. En effet l'espace des solutions est trop riche (sans compter les rotations on a au moins $36! = 3.71 \cdot 10^{41}$ configurations possibles). Un algorithme de brute force pur ne pourra donc s'exécuter entièrement avant la fin de ce millénaire sur aucun des ordinateurs actuellement disponibles dans le monde. Il est donc indispensable d'éliminer le plus possible de configurations qui ne sont pas solution au cours de la résolution. Les données à manipuler sont les différentes pièces, qui après avoir été chargées se verront affecter des entiers correspondants chacun à un des motifs de base.

3.2 Conception préliminaire

Les structures de données sont définies dans types.h comme suit :

```
typedef enum {false, true} Booleen;
```

```
struct changement {  
    int id;  
    int r;  
    int x;  
    int y;  
};
```

```
struct piece {  
    int id;  
    int haut;  
    int droite;  
    int bas;  
    int gauche;  
    int rotations;  
    Booleen used;  
};
```

```
struct place {  
    int x;  
    int y;  
};
```

```
typedef struct piece ** Puzzle;  
typedef struct piece ** Pioche;  
typedef struct place * Place;  
typedef struct piece * Piece;
```

```
struct backtrack_node {
    struct backtrack_node *pere;
    Place place;
    struct piece **table;
    struct backtrack_node *fils;
    int nb_elem;
};

typedef struct backtrack_node * backtrack_tree;
```

Description :

La structure pièce représente une pièce à l'aide des numéros respectifs des motifs qui la composent.

Elle contient également un champ id qui indexe les pièces selon leur ordre de chargement par le programme, un champ rotations qui quantifie le nombre de rotations qui ont été appliquée à la pièce, et un drapeau used qui permet de savoir si la pièce est libre ou a déjà été placée.

La structure place est simplement un condensé des deux coordonnées x et y qui représentent l'emplacement d'une pièce sur le puzzle.

La structure backtrack_node est la base de l'arbre de backtrack : dès que l'emplacement de la prochaine pièce à poser est déterminé par le tableau de contraintes, un backtrack_node est créé pour garder la trace des pièces qui sont posées à cet endroit de l'arbre.

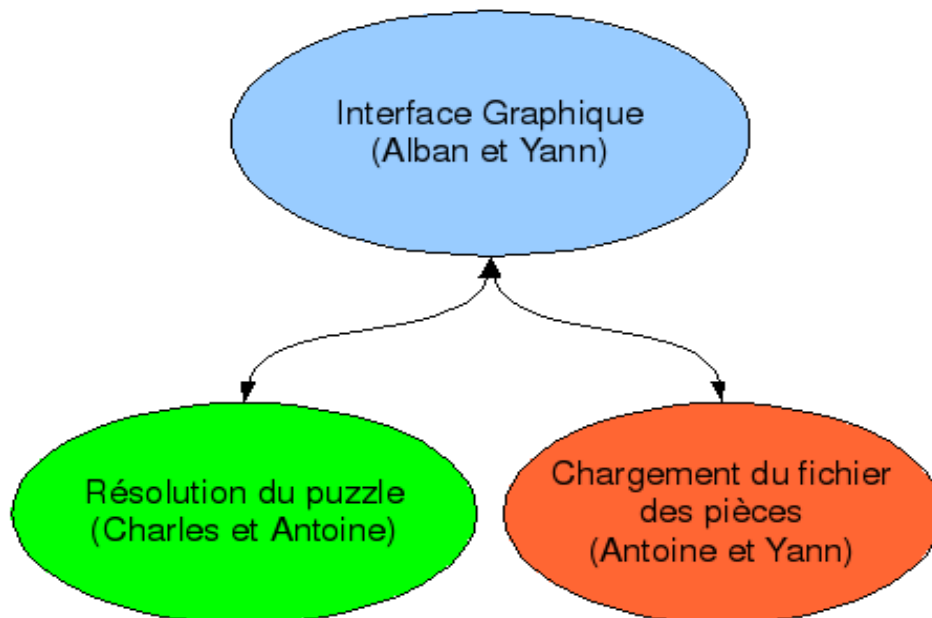
Les différents typedef servent simplement à définir directement des pointeurs vers toutes ces structures.

La structure changement permet la communication entre le module résolution et le module ihm.

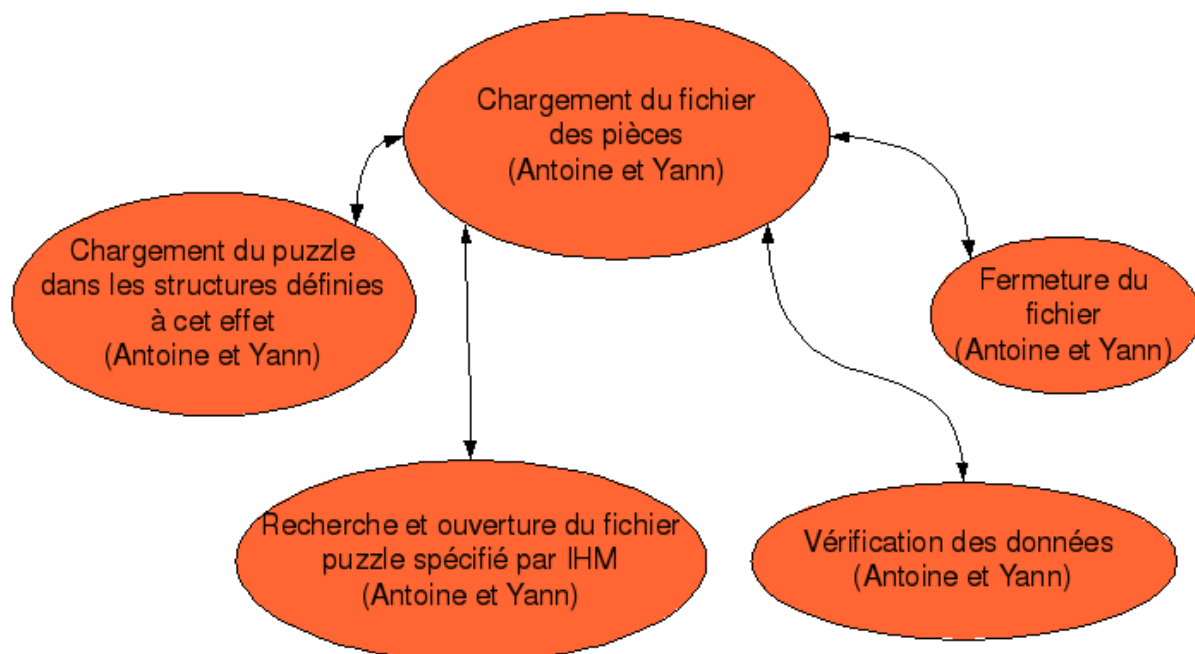
On remplit cette structure avec les coordonnées d'une pièce à ajouter ou à enlever du puzzle, et on la passe au module ihm par l'intermédiaire d'une Queue asynchrone (gérée par les fonctions g_async_queue_* de la glibc) .

Organisation au sein de l'équipe :

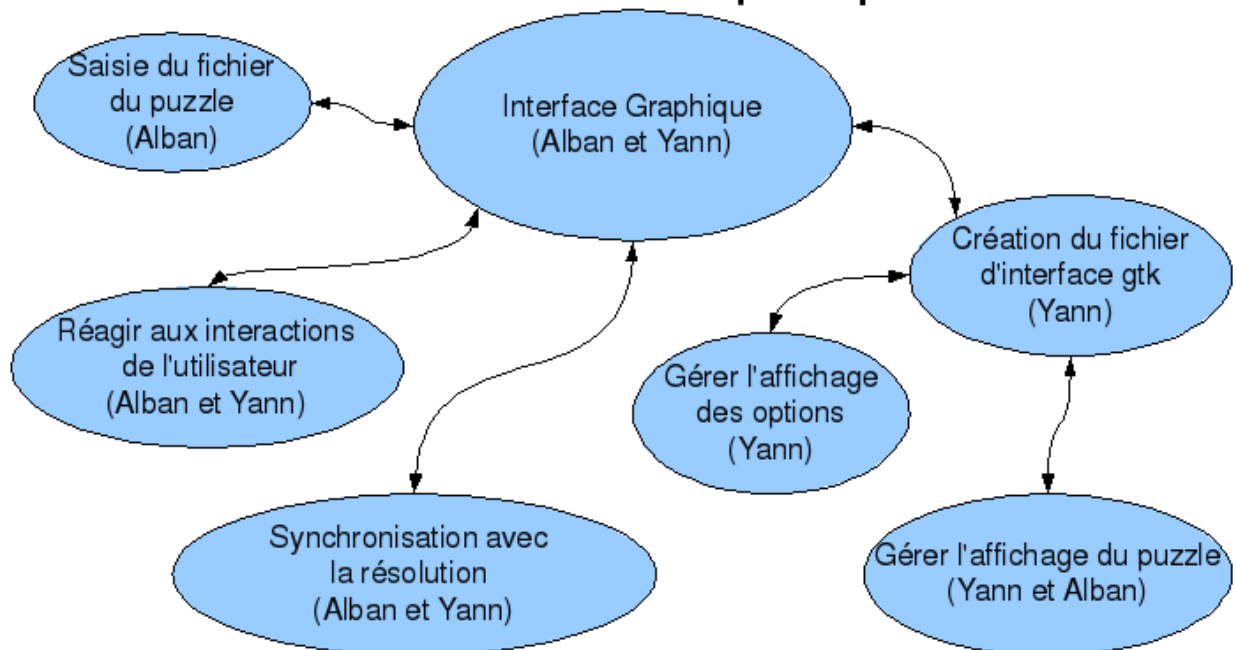
Modules



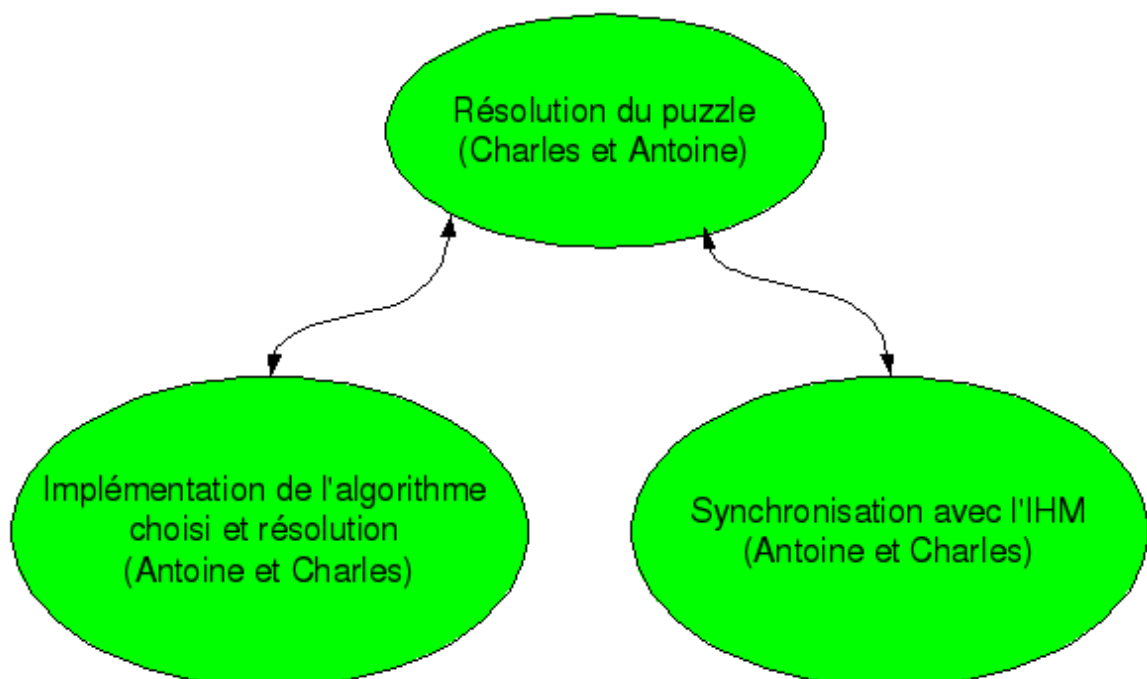
Module Chargement



Module Interface Graphique



Module Résolution



3.3 Conception détaillée

3.3.1°) Chargement du fichier

Général

(on connaît le nom du fichier et les dimensions du puzzle en arrivant ici.)

- Allocation de la structure Puzzle selon les dimensions du puzzle.
- Ouverture en mode lecture seule du fichier spécifié.
- Chargement d'une pièce du fichier.
- Vérification de la validité de la pièce.
 - * Si Invalide, avertir l'utilisateur et annuler tout.
 - * Si Valide, placer la pièce dans la structure Puzzle.
- Boucler sur le chargement tant qu'on a pas lu assez de Pieces pour compléter le puzzle.
- Vérifier que le fichier est fini.
 - * Si c'est le cas, tout va bien.
 - * Sinon, il y a des pièces "en trop", avertir l'utilisateur mais continuer.
- Renvoyez un lien vers la structure Puzzle.

Chargement d'une pièce

- Lire un entier dans le fichier.
- Le mettre dans le champ Haut de la structure Piece.
- Passer l'espace ou la tabulation.
- Recommencer ce qui précède pour les champs Droite, Bas et Gauche respectivement.

Vérification de la validité de la pièce

- Vérifier si les 4 champs Haut Droite Bas Gauche sont bien des entiers < 23
 - * Si Non, renvoyer Invalide.
 - * Si Oui, renvoyer Valide.

Génération du puzzle

Ce module contient aussi une fonction dont le but est de générer les images bmp 24 bits correspondant aux pièces lues dans le fichier, sans oublier de générer une image par rotation possible de la pièce.

Format du fichier du puzzle à résoudre

La première ligne est de la forme « #n m » avec n le nombre de lignes et m le nombre de colonnes.

Le fichier comporte autant de lignes que le puzzle comporte de pièces.

Chaque motif est représenté par un entier non nul, 0 désignant le « bord ».

Sur chaque ligne on met 4 entiers, le premier étant l'entier représentant le motif du haut, puis séparé par un espace, on met celui représentant le motif de droite et ainsi de suite avec le motif du bas et de gauche.

Format du fichier exporté par notre programme une fois la résolution terminée

On utilise le même format que précédemment mais on écrit les pièces dans l'ordre trouvé lors de la résolution en commençant en haut à gauche, en allant de gauche à droite et de haut en bas.

Prototypes du module chargement

Dans le module Chargement (chargement.h , chargement.c) :

```
void rotate_piece(Piece p) {  
    /* Prend une pièce en argument et effectue une rotation de 90° vers la droite */  
}
```

```
Pioche load_puzzle_file(const char *filename) {  
  
    /* Prend en argument le nom du fichier puzzle à charger. Alloue, initialise et  
    remplit la structure globale pioche. Renvoie un pointeur vers celle-ci. */  
}
```

```
void create_pieces(void) {
```

```
    /* Est chargée d'ouvrir tous les fichiers .bmp du répertoire motifs. Appelle ensuite  
    create_piece pour chacune des pièces de la pioche. Au final toutes les pièces de la pioche et  
    leurs diverses rotations ont été générées. */
```

```
}
```

```
void create_piece(int id, char *mtop, char *mleft, char *mbottom, char *mright, int r) {
```

```
    /* Prend en argument des pointeurs vers les différents motifs, chargés en mémoire,  
    qui composent la pièce. Génère un .bmp pour chaque rotation de chaque pièce à partir d'une  
    image vide black.bmp à l'aide des fonctions overwrite_* et finalize_piece pou terminer. */
```

```
}
```

```
void overwrite_left(char *black, char *left) {
```

```
    /* Appose le motif pointé par left sur la partie gauche de l'image de la pièce pointée  
    par black. */
```

```
}
```

```
void overwrite_right(char *black, char *right) {
```

```
    /* Appose le motif pointé par right sur la partie droite de l'image de la pièce pointée  
    par black. */
```

```
}
```

```
void overwrite_top(char *black, char *top) {
```

```
    /* Appose le motif pointé par top sur la partie haute de l'image de la pièce pointée  
    par black. */
```

```
}
```

```
void overwrite_bottom(char *black, char *bottom) {
```

```
    /* Appose le motif pointé par botom sur la partie basse de l'image de la pièce  
    pointée par black. */
```

```
}
```

```
void finalize_piece(char *black) {
```

```
    /* Fonction cosmétique qui prend l'image de la pièce après les appels successifs des  
    fonctions overwrite_* et noircit les bords ainsi que les diagonales pour lisser */
```

```
}
```

```
void *ErrorDataFormat(const char *filename) {
```

```
    /* Fonction générique qui affiche une erreur si le fichier puzzle spécifié n'a pas un  
    format valide, et tente ensuite de quitter le programme. */
```

```
}
```

3.3.2°) Interface Graphique :

Liste des options disponibles :

- Start : bouton permettant de démarrer la résolution [button]
*quand on clique dessus ça lance la résolution et le label du bouton devient grisé,

Boite de sélection du fichier du puzzle :

- Path : indique le chemin du fichier puzzle [1 text field input]
- n x m : indique la taille du puzzle [2 text field input]
- OK : valide l'entrée du fichier et de la taille [button]

Prototypes du module IHM

Dans le module IHM (interface.h, interface.c, callbacks.h, callbacks.c) :

```
GtkWidget* create_MainWindow (void) {
```

```
    /* Centre nerveux de GTK, cette fonction crée la fenêtre principale et initialise la  
    plupart des widgets. */
```

```
}
```

```
void start_clicked(GtkButton *b, gpointer funcdata) {
```

```
    /* Cette fonction est appelée dès que l'utilisateur presse le bouton Start.  
    Elle démarre deux nouveaux threads, chargés de la résolution et de la synchronisation. */
```

```
}
```

```
void close_eternity( GtkWidget *widget,gpointer *data) {
```

```
    /* Cette fonction est appelée dès que l'utilisateur ferme la fenêtre à l'aide de  
    l'icône en haut à droite de la celle-ci. Elle provoque l'arrêt du programme. */
```

```
}
```

```
void store_puzzlename (GtkWidget *widget, gpointer user_data) {  
    /* Cette fonction reçoit en argument le chemin vers le fichier puzzle, stocke  
    globalement ce chemin et appelle la fonction load_puzzle_file. */  
}  
  
void create_file_selection (void) {  
    /* Cette fonction crée la fenêtre de sélection du fichier puzzle. Dès que l'utilisateur  
    a fait son choix et clique sur un des boutons, la fonction store_puzzlename est appelée  
    automatiquement par GTK. */  
}  
  
void synchro( GAsyncQueue * q ) {  
    /* Cette fonction s'assure que le programme sera au courant de la fin du thread de  
    résolution.Elle ne fait rien avant que ce dernier lui envoie un signal de terminaison.  
    A ce moment là, elle gère l'affichage graphique de la solution. */  
}  
  
gboolean uptime_refresh(char *string) {  
    /* Cette fonction met à jour l'affichage du temps écoulé depuis le début de la  
    résolution.La variable « string » contient ce temps au format « %d h %d m %d s » */  
}
```

```
static void manage_uptime(void) {  
    /* Cette fonction est en fait le thread qui gère le calcul du temps écoulé depuis  
    le début de la résolution. Le thread s'exécute tant que le flag running est à true et  
    calcule la différence entre le temps de départ et le temps actuel et formate une chaîne  
    de caractère contenant le temps écoulé qu'il passe à la fonction uptime_refresh en le  
    mettant dans la queue d'actions de gtk via la fonction g_idle_add, ce thread attend une  
    seconde entre chaque tour de boucle grâce à un appel à la fonction sleep */  
}
```

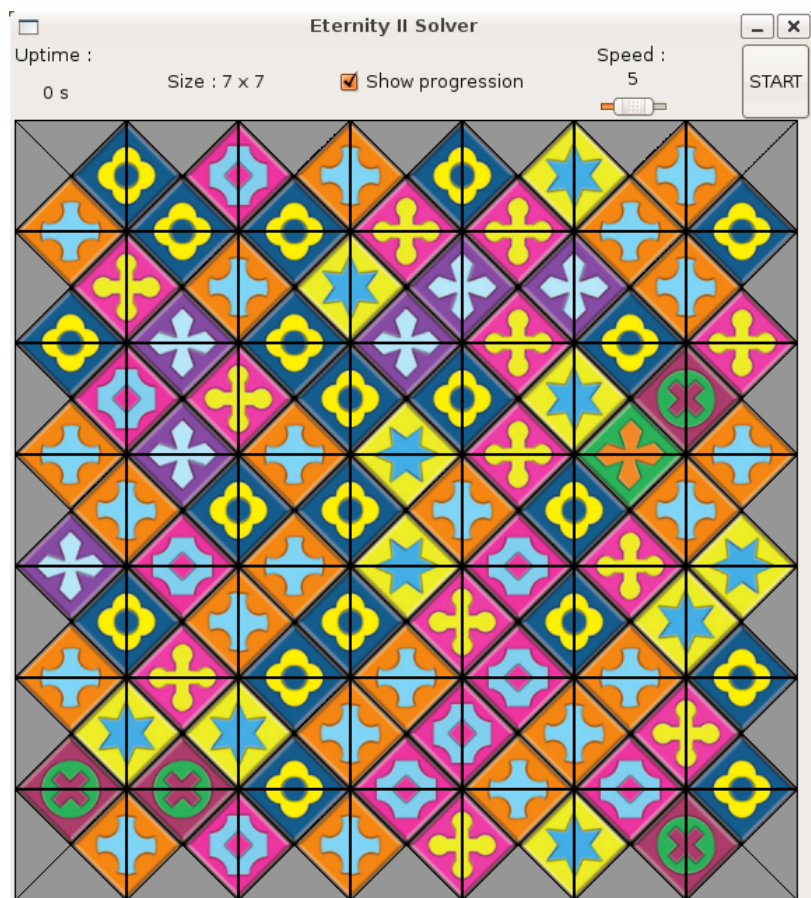
```
static gboolean remove_piece_from_puzzle(Changement p) {  
    /* Cette fonction détruit le widget correspondant à l'image de la pièce identifiée  
    par la structure p puis libère la mémoire occupée par p. Cette fonction est appelée  
    quand on veut enlever une pièce du puzzle lors du backtrack.*/  
}
```

```
static gboolean show_piece_in_puzzle(Changement p) {  
    /* Cette fonction crée le widget correspondant à l'image de la pièce identifiée  
    par la structure p, insère cette image dans la tableau GTK qui fait office de puzzle,  
    affiche l'image puis libère la mémoire occupée par p */  
}
```

```
void do_click(GtkWidget *widget, gpointer data) {  
    /* Cette fonction est appelée quand on clique sur le bouton OK du popup qui  
    s'affiche quand la résolution est terminée. Elle ferme juste le popup*/  
}
```

```
gboolean set_running_zero( char *q) {  
    /* Cette fonction remet juste à 0 la variable globale « running »  
}
```

Capture d'écran de l'ihm :



3.3.3°) Module de résolution

Principe général :

Nous avons décidé de poser tout d'abords les coins, puis les bords et enfin le centre du puzzle.

On choisi tout d'abords l'endroit où nous allons poser une pièce, cet endroit est sélectionné grâce à un tableau (en deux dimensions) contenant les contraintes de chaque case du plateau de jeu, calculées de cette manière : Chaque case a pour contrainte le nombre de cases voisines vides. On cherche donc le minimum du tableau de contraintes afin d'avoir une case comportant le plus de contraintes.

Maintenant qu'on a l'endroit où l'on veut poser une pièce, nous cherchons une pièce inutilisée et qui correspond au type de pièce que l'on veut placer (coin, bord ou centre).

On essaie toutes les pièces qui correspondent en leur faisant faire des rotations (3 au plus).

Si on trouve une pièceposable à cet endroit, alors on la pose et on sauvegarde cet essaie dans l'arbre de backtrack.

Si on ne trouve aucune pièceposable à cet endroit, alors on « backtrack », c'est à dire on remonte d'un étage dans l'arbre de backtrack : on enlève la dernière carte posée tout en la laissant dans l'arbre pour signaler qu'on l'a déjà essayé et qu'il ne faut pas la remettre au même endroit dans le même contexte.

Dans tous les cas (qu'on vienne de backtrack ou qu'on vienne de poser une pièce) on revient dans la boucle principale et on essaie à nouveau de poser une nouvelle pièce.

Bien évidemment si on vient d'en poser une, on va essayer d'en poser une ailleurs en mettant à jour le tableau des contraintes, et si on vient d'en enlever une on met aussi à jour les contraintes mais on essaie de poser une autre pièce au même endroit.

On répète ce procédé jusqu'à ce que le puzzle soit résolu.

Interaction avec le module IHM :

- Il faut un flag « running » qui s'il est à 1 stoppe la résolution
- Il faut un thread de synchronisation entre le module de résolution et le module d'interface graphique, afin de permettre à l'affichage de se mettre à jour à partir des données modifiées par le module de résolution.
- Il faut aussi un thread qui gère la mise à jour du champs « Uptime » afin d'afficher le temps écoulé depuis le début de la résolution.

– Prototypes du module résolution

Dans le module Résolution (resolution.h , resolution.c) :

```
void cleanup(void) {
```

```
    /* Fonction de nettoyage chargée notamment de libérer une grande partie de la
    mémoire allouée au cours du programme lorsqu'une solution a été trouvée. */
```

```
}
```

```
void solve(void) {
```

```
    /* Fonction principale lancée dans un thread à part par l'IHM lors d'un click sur le
    bouton Start. Elle contient notamment la boucle principale de résolution qui veille à poser
    le bon nombre de pièces de chaque type (COIN, BORD ou CENTRE) */
```

```
}
```

```
int nature_piece(const Piece p) {
```

```
    /* Prend en argument une pièce qu'on ne modifiera pas ici. Cette fonction renvoie
    simplement le type (COIN, BORD ou CENTRE) de la pièce suivant le nombre de motifs
    BORD qu'elle contient. */
```

```
}
```

```
Booleen poser_coin(void) {
```

```
    /* Parcours la pioche à la recherche d'une pièce à poser à la place courante, sachant
    que c'est un coin. Appelle poser_effectivement_coin si une pièce inutilisée qui n'a pas
    encore été utilisée à cet endroit (grâce à l'arbre de backtrack ) est disponible. Elle renvoie
    true si une pièce a été posée, false sinon. */
```

```
}
```

```
Booleen poser_bord(void) {
```

```
    /* Parcours la pioche à la recherche d'une pièce à poser à la place courante, sachant
    que c'est un bord. Appelle poser_effectivement_bord si une pièce inutilisée qui n'a pas
    encore été utilisée à cet endroit (grâce à l'arbre de backtrack ) est disponible. Elle renvoie
    true si une pièce a été posée, false sinon. */
}
```

```
Booleen poser_centre(void) {
```

```
    /* Parcours la pioche à la recherche d'une pièce à poser à la place courante, sachant
    que c'est un centre. Appelle poser_effectivement_centre si une pièce inutilisée qui n'a pas
    encore été utilisée à cet endroit (grâce à l'arbre de backtrack ) est disponible. Elle renvoie
    true si une pièce a été posée, false sinon. */

}
```

```
Booleen poser_effectivement_coin(Piece p, const Place place) {
```

```
    /* Teste la pièce proposée par poser_coin contre les contraintes de la position pour
    chaque degré de rotation. Renvoie true si la pièce correspond, false sinon */

}
```

```
Booleen poser_effectivement_bord(Piece p, const Place place) {
```

```
    /* Teste la pièce proposée par poser_bord contre les contraintes de la position pour
    chaque degré de rotation. Renvoie true si la pièce correspond, false sinon */

}
```

```
Booleen poser_effectivement_centre(Piece p, const Place place) {
```

```
    /* Teste la pièce proposée par poser_centre contre les contraintes de la position
    pour chaque degré de rotation. Renvoie true si la pièce correspond, false sinon */

}
```

```
void initialiser_contraintes(void) {
```

```
    /* Alloue et initialise le tableau des contraintes. */
```

```
}
```

```
void calculer_contraintes(Place place, int a ) {
```

```
    /* Fonction chargée de mettre à jour le tableau des contraintes après le retrait ou  
    l'ajout d'une pièce. Le paramètre a différencie ces deux cas pour s'assurer que les  
    contraintes sont changées correctement. */
```

```
}
```

```
Place trouver_case_avec_le_plus_de_contraintes(int type) {
```

```
    /* Parcours le tableau des contraintes pour trouver l'emplacement du puzzle avec le  
    maximum de contraintes qui correspond à une pièce de nature type, et renvoie la position  
    de cette pièce dans le puzzle. */
```

```
}
```

```
void backtrack(void) {
```

```
    /* Fonction appelée lorsqu'aucune pièce restante dans la pioche ne peut être posée à  
    la position courante. On revient sur nos pas. */
```

```
}
```

```
void log_add(Piece piece, Place place) {
```

```
    /* Fonction appelée après l'ajout d'une pièce au puzzle. Elle met à jour l'arbre de  
    backtrack */
```

```
}
```

```
void rotate_piece(Piece p) {  
    /* Prend une pièce en argument et effectue une rotation de 90° vers la droite */  
}  
  
int deja_posee(Piece p) {  
    /* Vérifie l'arbre de backtrack pour savoir si la pièce p a déjà été posée à cet endroit  
    précédemment. */  
}  
  
int nature_place(int x, int y) {  
    /* Renvoie la nature de la place dans le puzzle selon ses coordonnées : BORD  
    COIN ou CENTRE. */  
}
```

3.3.4) Divers

Dans support.h et support.c :

Ces fichiers ont été générés par le programme Glade avec lequel l'interface de base a été conçue, et contiennent diverses fonctions utilitaires qui simplifient la programmation GTK.

3.4 Tests unitaires

3.4.1°) Module chargement

Donnée en entrée :

- un nom de fichier dont le contenu respecte le format de fichier.

Données attendues en sortie :

- les fichiers bmp représentant les pièces dans leur état initial ainsi qu'après rotations, générées à partir des motifs triangulaires.
Ces fichiers doivent être générés avec le nom « pN-R.bmp » avec N le numéro de la pièce et R le numéro de la rotation (R variant donc de 0 à 3).
Les fichiers images des motifs et des pièces sont au format bmp 24 bits (3 octets par couleurs, en suivant la règle RGB)).
Les fichiers de pièces sont la concaténation des 4 motifs triangulaires, avec un contour noir et des diagonales noires.

- Une structure pioche remplie, qui est en fait un tableau à deux dimensions contenant des pointeurs vers les pièces initialisées aux valeurs fournies dans le fichier du puzzle.

- Les dimensions nxm du problème, lues dans le fichier puzzle.

3.4.2°) Module IHM

Données en entrée :

- Une structure pioche remplie avec des pièces.
- Les dimensions nxm du problème.
- Les images bmp des pièces du puzzle.

Données attendues en sortie :

- L'affichage de la pioche sous la forme d'un puzzle nxm en GTK, en utilisant les images bmp des pièces. Les pièces doivent s'afficher dans l'ordre de gauche à droite puis de haut en bas.

3.4.3°) Module Résolution

Donnée en entrée :

- Une structure pioche remplie avec des pièces.
- les dimensions nxm du problème.

Données attendues en sortie :

- La solution du puzzle sous la forme d'un affichage en console.

L'affichage se fait sous la forme suivante :

N : H G B D [R]

où N est l'identifiant de la pièce (son numéro)

H l'identifiant du motif du haut de la pièce (après éventuelles rotations).

G l'identifiant du motif de gauche de la pièce (après éventuelles rotations).

B l'identifiant du motif du bas de la pièce (après éventuelles rotations).

D l'identifiant du motif de droite de la pièce (après éventuelles rotations).

R le nombre de rotations pour arriver à cette configuration sur la pièce N à partir de la configuration initiale dans la pioche.

3.5 Tests d'intégration

3.5.1°) Interaction Chargement/IHM

Données en entrée :

- Le nom du fichier texte contenant les informations sur le puzzle.

Données attendues en sortie :

- Génération des fichiers images bmp 24 bits des pièces (ainsi que les rotations).

- Remplissage de la pioche par le module de chargement.

- Les dimensions nxm du problème.

- Transmission de la pioche à l'IHM.

- Affichage graphique de la pioche (générée par le module de chargement) par l'IHM en utilisant les fichiers images des pièces créés par le module de chargement.

On affiche les pièces graphiquement comme un puzzle, en suivant l'ordre indiqué dans le fichier du puzzle, de gauche à droite puis de haut en bas.

3.5.2°) Interaction Résolution/IHM

Données en entrée :

- Une pioche remplie avec des pièces.

- Les dimensions du problème.

- Les images des pièces (ainsi que de leurs rotations) au format bmp 24 bits.

Données attendues en sortie :

- Affichage de la solution (fournie par le module de résolution) dans l'IHM, grâce aux images des pièces, sous la forme d'un puzzle complété.

-

3.5.3°) Interaction Chargement/Résolution

Données en entrée :

- Le nom du fichier texte contenant les informations sur le puzzle.

Données attendues en sortie :

- La pioche remplie avec les pièces.
- Génération des fichiers bmp 24 bits représentant les pièces à partir des motifs.
- Les dimensions nxm du problème.
- L'affichage de la solution du problème en console par le module de résolution, à partir des données transmises par le module de chargement.
L'affichage de la solution se fait toujours en respectant le format déjà cité :
N : H G B D [R]

3.5.4°) Interaction Chargement/résolution/IHM

Données en entrée :

- On entre le nom du fichier du puzzle dans la fenêtre de sélection de l'interface graphique.

Données attendues en sortie :

- La pioche remplie avec les pièces.
- Les dimensions nxm du problème.
- Génération des fichiers bmp 24 bits représentant les pièces à partir des motifs.
- Affichage graphique de la pioche (en respectant l'ordre du fichier puzzle).
- Résolution du puzzle quand on appuie sur le bouton « START ».
- Affichage graphique de la solution quand la résolution est finie.

4 Conclusion

L'équipe avait d'abord été attirée par l'idée d'un jeu auquel on peut gagner gros, un thème récurrent de nos jours. Peu à peu, des aspects plus poussés du projet ont pris le pas sur cette attraction primaire. Après de nombreux cours abstraits d'algorithmique, une application concrète fut accueillie chaleureusement par toute l'équipe. La recherche de l'optimisation, traduite à la fois par des améliorations purement algorithmiques et des optimisations du code lui-même, s'est avérée être un puissant moteur vers une plus grande maîtrise de l'architecture matérielle d'un ordinateur. Habitué à des affichages sur des terminaux, l'entrée en scène de GTK fût également une bouffée d'air frais.

Nettoyer et simplifier le code tout comme l'interface furent également plus compliqué que prévu. En fin de compte, l'installation et l'utilisation du programme ont été simplifiés le plus possible pour toucher un public plus large.

Par ailleurs, une seconde version du programme, plus complexe et interne à l'équipe pour l'instant, a été développée en parallèle du projet. Son intérêt réside dans la possibilité théorique de suivre graphiquement l'évolution du puzzle au cours de la résolution. A la fois pratique pour vérifier le fonctionnement du module Résolution, car beaucoup plus parlant pour l'être humain que des lignes de texte sur un terminal, et motivant de par les problèmes de synchronisation entre les threads qu'il pose, ce programme reste néanmoins très instable pour le moment.

Les points noirs à noter concernent en grande partie une aversion globale de l'équipe envers la documentation, toujours considérée plus comme une corvée qu'autre chose, ce qui a engendré de nombreux retards sur le planning, heureusement sans grandes conséquences au final. L'organisation fût également difficile au début, en grande partie à cause d'une communication interne mal rodée qui s'est affinée avec le temps.

5 Bibliographie

<http://fr.eternityii.com/>

<http://www.eternity2.fr/>

pages du manuel du programmeur UNIX.

Langage C, G. Willms, édition Micro Application, série PC Poche.

et l'indispensable <http://google.fr>

Annexe : <http://library.gnome.org/devel/gtk/unstable/>

6 Annexes

6.1 Annexe 1 : Gestion de projet

Plan de charge Prévisionnel						
Description de l'activité	charge en %	charge en H	Charges en heures / participant			
			LACHAUD	MARTIN	NICOLAS	SIONNEAU
total	100%	209	50	50	51	58
Gestion de projet						
Spécification	21%	43	12	12	8	11
conception préliminaire	4%	8	2	2	2	2
conception détaillée	15%	32	8	8	8	8
codage	19%	40	8	8	12	12
intégration	26%	54	12	12	13	17
soutenance	8%	16	4	4	4	4
	8%	16	4	4	4	4

Suivi d'Activités (charges consommées)						
Description de l'activité	charge en %	charge en H	Charges en heures / participant			
			LACHAUD	MARTIN	NICOLAS	SIONNEAU
total	119%	249	60	60	64	65
Gestion de projet						
Spécification	32%	80	27	27	12	14
conception préliminaire	3%	8	2	2	2	2
conception détaillée	8%	20	5	5	5	5
codage	16%	41	9	9	12	11
intégration	22%	56	8	8	20	20
soutenance	10%	24	4	4	8	8
	8%	20	5	5	5	5

6.1.1°) Plan de charge**Plan de charge Prévisionnel**

Description de l'activité	charge en %	charge en H	Charges en heures / participant			
			LACHAUD	MARTIN	NICOLAS	SIONNEAU
total	100%	209	50	50	51	58
Gestion de projet	21%	43	12	12	8	11
Réunion de lancement	2%	4	1	1	1	1
Planning prévisionnel et suivi d'activité	4%	8	2	2	2	2
Réunion de suivi	8%	16	4	4	4	4
Rédaction	6%	12	5	5	1	1
Site web	1%	3	0	0	0	3
Spécification	4%	8	2	2	2	2
Définition des fonctionnalités	4%	8	2	2	2	2
Conception préliminaire	15%	32	8	8	8	8
définition d'un modèle de données	4%	8	2	2	2	2
Définition d'un format de fichiers	2%	4	1	1	1	1
Définition des fonctionnalités	6%	12	3	3	3	3
Définition des différents modules(.h)	4%	8	2	2	2	2
conception détaillée	19%	40	8	8	12	12
Définition des structures de données	4%	8	1	1	3	3
Conceptions de la base de données	3%	6	1	1	2	2
Définition des fonctions	6%	12	3	3	3	3
Définition des tests unitaires	4%	8	2	2	2	2
Maquettages des IHM (ascii,web,...)	3%	6	1	1	2	2
codage	26%	54	12	12	13	17
Ecriture des interfaces (.h)	4%	8	2	2	2	2
Ecriture des makefiles	4%	8	1	1	1	5
Ecriture des différentes fonctions	13%	28	7	7	7	7
Tests unitaires	5%	10	2	2	3	3
intégration	8%	16	4	4	4	4
Intégration des différents modules	4%	8	2	2	2	2
Tests d'intégration	4%	8	2	2	2	2
soutenance	8%	16	4	4	4	4
Préparation de la soutenance	6%	12	3	3	3	3
Soutenance	2%	4	1	1	1	1

6.1.2°) Suivi d'activités**suivi d'activités (charges consommées)**

Description de l'activité	charge en %	charge en H	Charges en heures / participant			
			LACHAUD	MARTIN	NICOLAS	SIONNEAU
total	100%	249	60	60	64	65
Gestion de projet	32%	80	27	27	12	14
Réunion de lancement	2%	4	1	1	1	1
Planning prévisionnel et suivi d'activité	3%	8	2	4	1	1
Réunion de suivi	11%	28	7	7	7	7
Rédaction	14%	36	17	15	2	2
Site web	2%	4	0	0	1	3
Spécification	3%	8	2	2	2	2
Définition des fonctionnalités	3%	8	2	2	2	2
conception préliminaire	8%	20	5	5	5	5
définition d'un modèle de données	2%	4	1	1	1	1
Etat de fichiers associés au modèle données	2%	4	1	1	1	1
Définition des fonctionnalités	2%	4	1	1	1	1
Définition des différents modules(.h)	3%	8	2	2	2	2
conception détaillée	16%	41	9	9	12	11
Définition des structures de données	2%	6	1	1	2	2
Conceptions de la base de données	2%	4	1	1	1	1
Définition des fonctions	6%	15	4	4	4	3
Définition des tests unitaires	3%	8	2	2	2	2
Maquettages des IHM (ascii,web,...)	3%	8	1	1	3	3
codage	22%	56	8	8	20	20
Ecriture des interfaces (.h)	2%	6	1	1	2	2
Ecriture des makefiles	2%	4	1	1	1	1
Ecriture des différentes fonctions	13%	32	4	4	12	12
Tests unitaires	6%	14	2	2	5	5
intégration	10%	24	4	4	8	8
Intégration des différents modules	5%	12	2	2	4	4
Tests d'intégration	5%	12	2	2	4	4
soutenance	8%	20	5	5	5	5
Préparation de la soutenance	6%	16	4	4	4	4
Soutenance	2%	4	1	1	1	1

6.2 Annexe 2 : Code source

Le code source est disponible à l'adresse suivante:

<http://eternity.yozora-irc.net/eternity.tar.gz>