Projet Link-Pix

Ambroise Bernhardt - ambroise.bernhardt@etud.univ-montp2.fr
Florian Galinier - florian.galinier@etud.univ-montp2.fr
Adrien Plazas - adrien.plazas@etud.univ-montp2.fr
Chloé Tronc - chloe.tronc@etud.univ-montp2.fr

Résumé

Ce rapport est le compte rendu du projet Link-Pix exécuté par les auteurs et proposé par Philippe Janssen pour l'unité d'enseignement GLIN405 Projet Informatique du quatrième semestre du parcours Licence Informatique de la Faculté de Sciences de Montpellier en 2012-2013.

Remerciements

Nous remercions tout particulièrement Philippe Janssen pour nous avoir brillament encadré et soutenu tout le long de la réalisation de ce projet.

Table des matières

Chapitre 1

Introduction

1.1 Généralités

Le projet Link-Pix s'inscrit dans le cadre de l'UE GLIN405 proposée au semestre 4 du parcours Informatique. L'objectif de ce projet est la mise en place d'un travail de groupe pour la conception et la réalisation d'un projet informatique. L'organisation, la distribution du travail, la communication sont des éléments essentiels à la réussite du projet, et apporteront aux participants un savoir-faire pratique du travail de groupe.

Le projet devra être fini aux dates de soutenance, c'est à dire les 30 et 31 mai 2013.

Le groupe pour le projet Link-Pix se compose de six membres, une répartition du travail devra par conséquent être faite.

Chaque groupe est encadré par un tuteur, Philippe Janssen dans notre cas, chargé d'encadrer et de guider le groupe.

1.2 Le sujet

L'objectif de ce projet est la réalisation d'un solveur pour les puzzles de type Link-Pix, dont les détails seront donnés ci-dessous.

1.2.1 Principe du Link-Pix

Les puzzles de type Link-Pix (aussi connus sous le nom de Link-a-Pix) sont des puzzles images. Le puzzle est une grille rectangulaire dont certaines cases (appelées indice par la suite) contiennent des nombres positifs.

La résolution du puzzle fait apparaître une image, sous la forme d'une grille de même taille dont chaque case est soit noire, soit blanche.

Cette image est obtenue en reliant par un chemin, comprenant les indices, chaque couple d'indices selon les contraintes suivantes :

- un indice de valeur 1 est lié à lui-même ;
- un indice ne peut être relié qu'à un seul autre indice de même valeur ;
- deux indices de valeur n doivent être reliés par un chemin de n cases ;
- deux chemins ne peuvent pas se croiser ;
- un chemin ne peut pas se recouper.

Ainsi, deux cases d'indice 3 devront être reliées avec un chemin de longueur 3.

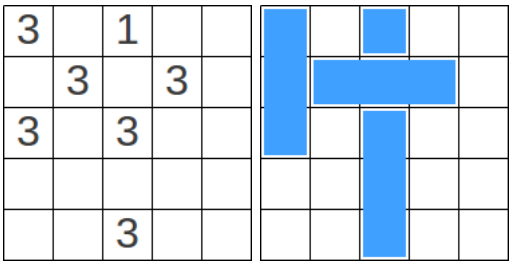


FIGURE 1.1 – Exemple de puzzle 5×5 de type Link-Pix et son unique solution.

1.2.2 Sujet

L'objectif du projet se divise en deux parties :
- la première partie est la création d'un solveur, un programme capable d'analyser une grille et de rendre l'image résultat ;
- la seconde partie est l'analyse d'une image dans l'objectif de créer une grille correspondant à cette image.

Ainsi, le programme final devra être capable de résoudre des puzzles de type Link-Pix, dévoilant l'image cachée par le problème.

1.3 Cahier des charges

1.3.1 Fonctionnalités

Le solveur devra, à partir d'un puzzle, analyser la grille et trouver la solution de celle-ci.

L'analyseur devra quant à lui être capable, à partir d'une image donnée, d'analyser cette image et de renvoyer une grille de type Link-Pix qui possède une unique solution.

1.3.2 Contraintes

Le solveur doit être capable de lire une grille enregistrée dans un fichier, selon un format défini, afin de l'analyser.

Il a été décidé que le solveur ne traiterait que le cas des puzzles à solution unique, ceci afin de faciliter le développement.

Il doit également se composer d'une interface graphique permettant de suivre l'évolution pas à pas de la résolution.

En outre, le développement de la seconde partie, l'analyse de l'image et la création d'une grille correspondante, a été abandonné quasiment dès le début. Ceci est principalement dû au désistement de certains membres du groupe, ne nous permettant pas de travailler parallèlement sur les deux parties. La partie solveur a ainsi été privilégiée.

Chapitre 2

Organisation du projet

2.1 Organisation du travail

2.1.1 Réunions de travail

Une réunion hebdomadaire était tenue tous les mardis, d'abord à 13h15, rapidement décalée à 11h20. Certaines réunions exceptionnelles ont pû être rajoutées le jeudi.

2.1.2 Répartition des tâches

Les tâches étaient assignées au dévouement ou, si certaines tâches n'étaient pas assignées et que des personnes ne s'étaient pas dévouées, par l'encadrant.

2.1.3 Planification du développement

On peut analyser la méthode de développement utilisée de la manière suivante :
— analyse globale du problème ;
— analyse des structures de données nécessaires à la résolution du problème ;
— implémentation des structures et de leurs fonctions associées ;
— analyse en profondeur du problème et écriture des algorithmes nécessaires à sa résolution ;
— implémentation des algorithmes en s'appuyant sur les structures déjà implémentées ;
— construction de l'application finale à partir des briques logicielles implémentées.

2.1.4 Élection d'un chef de projet

Aucun chef de projet n'a été élu, les décisions ont été prises lors des réunions, où tout le monde pouvait faire des propositions qui étaient alors débattues. Le groupe de travail était suffisamment restreint pour qu'un tel système fonctionne.

2.1.5 Gestion du groupe d'étudiants

La gestion du groupe s'est faite par courriel, lors des réunions et lors de rencontres hors du cadre du projet. Elle s'est avérée compliquée et de nombreux problèmes de communication et tout particulièrement de non comunication et d'absentéisme sont apparus et ont retardé le projet.

2.2 Choix des outils de développement

2.2.1 Choix du langage

Le langage initialement proposé par notre encadrant était le C. Ce langage étant le mieux connu de toute l'équipe, il avait alors un avantage subjectif mais réel.

Il s'avère que le C répondait en outre assez bien aux besoins du projet, qui n'a nécessité que des structures de données assez simples.

La question posée a été celle de la version du langage à utiliser : C89 ou C99 ? Au final le C89 a été choisi, afin de forcer l'équipe à utiliser un langage plus strict.

2.2.2 Choix de l'éditeur

Le C étant un langage permettant un choix extrêmement souple d'éditeurs, il a été décidé de ne pas associer le projet à un éditeur particulier et de favoriser l'utilisation d'outils simples. Ainsi le choix d'un simple éditeur de texte et d'un émulateur de terminal a été fait par la majorité, sinon la totalité, des membres de l'équipe.

2.2.3 Choix du compilateur

GCC est le compilateur C utilisé par tous à la faculté, de plus certains membres de l'équipe sont attachés aux outils GNU, ainsi son choix a été fait sans se poser de questions particulières.

Le choix a été fait sur le tard d'en activer les options `--ansi` et `--pedantic` pour plus de rigueur dans le code.

2.2.4 Choix du débogueur

Pour des raisons similaires à celles du choix du compilateur, GDB a été choisi pour nos besoins de déboguage.

2.2.5 Choix du gestionnaire de projet et du gestionnaire de versions

Le service informatique de la faculté a mis à disposition des étudiants et professeurs un service d'hébergement et de gestion de projet utilisant GitLab aux alentours du démarrage du projet. Il a rapidement été décidé de l'adopter, ainsi Git a été choisi comme gestionnaire de version car c'est celui utilisé par GitLab.

2.2.6 Choix des outils d'analyse

Aucun outil d'analyse n'a été utilisé.

2.2.7 Choix des outils de documentation

Concernant la documentation du code, Doxygen a été proposé par notre encadrant et a été accepté par les membres de l'équipe. Quant au rapport, le choix de LaTeXa été fait car il permet de modulariser et de simplifier le travail à plusieurs sur un document par rapport à des outils de traitement de texte plus classiques utilisant un format de fichier binaire et monolithique.

Chapitre 3

Analyse du projet

La résolution comporte deux parties :
— le calcul des couples d'indices qui seront reliés ;
— le calcul du chemin reliant chaque couple d'indices.

3.1 Calcul des couples d'indices

Cette partie de la résolution comporte trois sous-problèmes :
— le calcul des ensembles de voisins ;
— la propagation des ensembles de voisins ;
— le est d'existence de chemin.

3.1.1 Calcul des ensembles de voisins

Pour chaque indice, on calcule l'ensemble des indices auxquels il peut être relié (appelé par la suite ensemble des voisins possibles). Deux indices sont des voisins possibles si :
— ils ont la même valeur d'indice ;
— leur distance ($abs(x_1 - x_2) + abs(y_1 - y_2) + 1$ pour deux indices de coordonnées (x_1, y_1) et (x_2, y_2)) est inférieure et a la même parité que leur valeur d'indice.

Ces conditions sont des conditions nécessaires à l'existence d'un chemin liant les deux indices, mais pas suffisantes. Les couples impossibles seront progressivement éliminés par les analyses suivantes jusqu'à ce que chaque indice n'ait plus qu'un voisin.

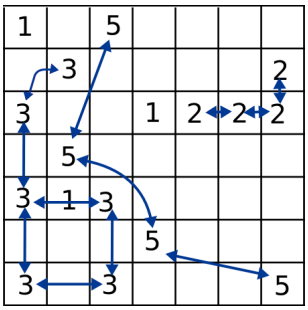


FIGURE 3.1 – Ensembles de voisins.

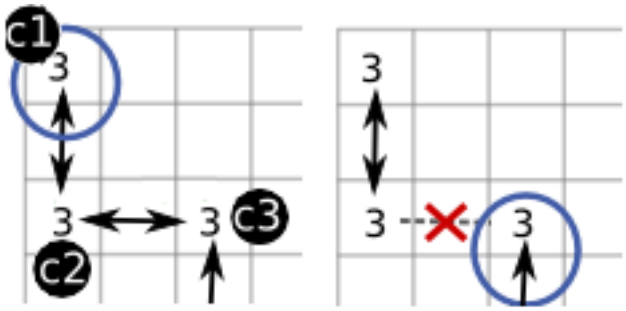
3.1.2 Formation des couples

Au cours de la résolution du problème le solveur devra parvenir à retrouver les couples, c'est-à-dire associer chaque indice à son unique voisin.

Lorsqu'un indice c_1 n'a qu'un voisin c_2, c_1 et c_2 forment un couple.

c_2 ne peut plus former de couple avec un autre indice, il faut donc supprimer c_2 des autres ensembles de voisins possibles.

Cette suppression peut entraîner la formation de nouveaux couples. En effet, il est possible que parmi les indices qui n'ont plus c_2 pour voisin, certains ne comportent plus qu'un seul autre voisin (figure 3.2).



(a) c_1 a c_2 pour unique voisin.

(b) c_1 et c_2 forment donc un couple. À son tour c_3 n'a plus qu'un voisin, avec lequel il forme donc un couple.

FIGURE 3.2 – Affectation des couples

Pour certains problèmes cette phase de propagation suffit à déterminer l'ensemble des couples.

Dans le cas général il faut raffiner les ensembles de voisins en vérifiant l'existence de chemins.

3.1.3 Test de l'existence d'un chemin entre deux cases

Au cours de la résolution, il sera nécessaire de tester l'existence d'un chemin entre deux coordonnées. Cela permettra notamment d'éliminer des couples potentiels.

Lorsqu'un indice c_1 possède plusieurs voisins, on vérifie pour chacun d'entre eux s'il existe au moins un chemin le liant à c_1 dans la grille.

Un chemin est constitué d'une suite de cases : la première case est la case de départ, et les cases suivantes conduisent à la case d'arrivée.

Ce chemin ne doit passer que par des cases libres. Une case est libre si elle ne correspond pas à une case indice et si elle n'est empruntée par aucun chemin.

La longueur du chemin doit être égale à la valeur des indices.

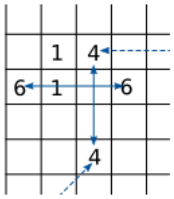
Pour tester l'existence d'un chemin, on peut ainsi partir de la case de départ puis ajouter une case suivante, selon les conditions suivantes :
— la case doit être une case adjacente de la case précédente ;
— la case ne doit pas être déjà occupée par un autre chemin ou un autre indice ;
— la case ne doit pas déjà faire partie du chemin ; en effet le chemin ne doit pas se recouper.
On continue ainsi récursivement jusqu'à arriver à la case d'arrivée ou bien à un chemin trop long.

Ce test permet de réduire les ensembles de voisins et ainsi, avec la phase de propagation, de découvrir de nouveaux couples d'indices.

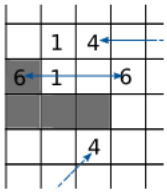
3.2 Calcul du chemin entre un couple d'indices

Il est utile de déterminer par quelles cases passe le chemin qui relie un couple.

En effet, aucun autre chemin ne pourra emprunter ces cases. Cela pourra donc permettre l'élimination de certains couples de voisins lors de la suite de l'exécution du programme (figure 3.3).



(a) Les deux indices 4 sont un couple de voisins possibles.



(b) Après la construction du début de chemin entre les deux 6, il n'y a plus de chemin valide entre les deux quatres.

FIGURE 3.3 – Le calcul de chemin permet d'éliminer des voisins possibles.

Lorsque plusieurs chemins relient un couple d'indices (c_1, c_2), on ne peut pas choisir a priori l'un d'entre eux. Cependant dans le cas où tous ces chemins ont un préfixe commun, on peut marquer les cases de cet unique début de chemin comme étant occupées (figure 3.4).

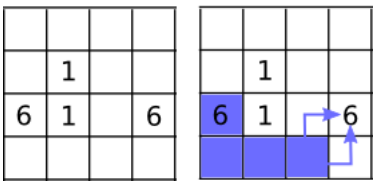


FIGURE 3.4 – Dans cet exemple, seules les quatre premières cases du chemin peuvent être construites. Au delà, il y a plus d'une possibilité.

S'il y a un unique début de chemin, il passe forcément par une des quatre cases adjacentes de c_1. Pour chaque case adjacente on peut alors tester l'existence d'un chemin la reliant à l'indice d'arrivée. S'il y a plus d'une case possible, il n'y a pas de début unique de chemin. Sinon on a trouvé un unique début de chemin. On peut ensuite recommencer le même test à partir de cette case de manière récursive.

Si c_3 est l'extrémité du préfixe de tous les chemins entre c_1 et c_2, il reste à découvrir le chemin de c_3 à c_2. Pour mémoriser cette information, on remplace l'indice c_1 par le nouvel indice c_3 et on remplace le couple d'indices (c_1, c_2) par le couple (c_3, c_2) (figure 3.5).

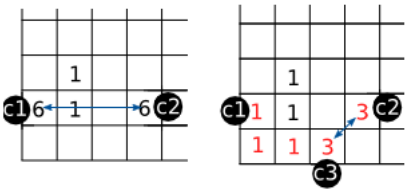


FIGURE 3.5 – le couple d'indices (c_1, c_2) est remplacé par le couple (c_3, c_2).

3.3 Enchaînement des traitements

Les deux parties (calcul des couples d'indices, calcul des chemins) sont interdépendantes : lorsqu'un nouveau couple d'indices est trouvé, il faut rechercher le chemin les reliant et la découverte d'un nouveau chemin ou d'une partie de chemin permet de trouver de nouveaux couples d'indices.

L'enchaînement de ces deux traitements se fait selon l'algorithme suivant :

Algorithme 1 : Enchainement des traitements

```
Calculer les ensembles de voisins initiaux (calculés avec la distance) ;
Propager les ensembles de voisins ;
Répéter
    Reduire les ensembles de voisins en vérifiant l'existence de chemins ;
    Propager les ensembles de voisins ;
    Pour chaque couple d'indices non reliés par un chemin faire
        Rechercher un chemin entre les deux indices ;
        Mettre à jour la grille des cases occupées ;
    finpour
jusqu'à pas de modification de la grille;
```

Chapitre 4

Définition des structures

4.1 Les structures

La résolution du problème requiert des structures de données afin de représenter :
— des coordonnées, représentées par la structure `Coordonnees` (voir **??**) contenant deux entiers x, l'abscisse, et y, l'ordonnée et permettant de situer une case dans un puzzle ;
— des indices, représentés par la structure `Indice` (voir **??**) contenant la valeur d'une case et sa coordonnée ;
— un ensemble d'indices représenté par une liste ;
— un ensemble de coordonnées également représenté par une liste ;
— l'ensemble des « cases » contenant les indices et la liste des voisins possibles pour cet indice ;
— la grille correspondant à l'image résultat (donnant donc les cases « noircies »).
Nous avons choisi de représenter ces deux derniers points par une unique structure `TabVoisins`.

4.1.1 Les listes chainées

Nous avons défini deux types de listes chainées : les listes de coordonnées et les listes d'indices.

De façon générale, la liste est composée d'une donnée, sa première valeur, et de l'adresse de l'élément suivant :
— `ListeCoord` (voir **??**) est donc une liste dont les éléments sont des `Coordonnees` ;
— `ListeInd` (voir **??**) est, elle, une liste d'`Indice`.
Les principales fonctions associées aux listes sont celles de création (constructeur / destructeur), d'insertion et de suppression. Tout d'abord, la fonction `creerLCoord`, respectivement `creerLInd`, permet la création d'une liste. Elle prend donc en paramètre l'élément tête de la liste ainsi qu'une liste équivalente à la queue de la liste que l'on souhaite créer et pouvant être égale à la liste vide. La fonction `detruitLCoord` (respectivement `detruitLInd`) supprime l'intégralité de la liste passée en paramètre. Les fonctions d'insertion permettent, elles, d'ajouter un élément dans la liste, soit au début soit à la fin. Enfin, les fonctions de suppression (comme `supprimerpremierLCoord` ou `supprimerdansLCoord`) permettent d'enlever soit le premier soit un élément quelconque de la liste. Ces deux derniers types de fonctions ne renvoient rien mais modifient des listes.

Ces listes sont définis dans les modules d'opération sur les listes de coordonnées et d'indices situés respectivement dans les fichiers `operationLCoord.h`, `operationLCoord.c`, `operationLInd.h` et `operationLInd.c` (voir A.1).

4.1.2 Un tableau de voisins

La structure `TabVoisins`, définie dans les fichiers `TabVoisins.h` (??) et `TabVoisins.c` (voir A.1), est un tableau de cases qui donne pour chaque case sa valeur et la liste de ses potentiels voisins. Pour cela, elle utilise la structure `CaseVoisins` (voir ??) associant donc :

- la valeur d'une case, qui peut être :
  - 0 si la case est libre ;
  - 1 si la case est un indice de valeur 1 ou une case empruntée par un chemin ;
  - $n > 1$ si la case est un indice de valeur n.
- sa liste `ListeCoord` qui est la liste des coordonnées des voisins de la case si celle-ci est un indice de valeur supérieure à 1.

`TabVoisins` a également en donnée les dimensions du puzzle d'origine.

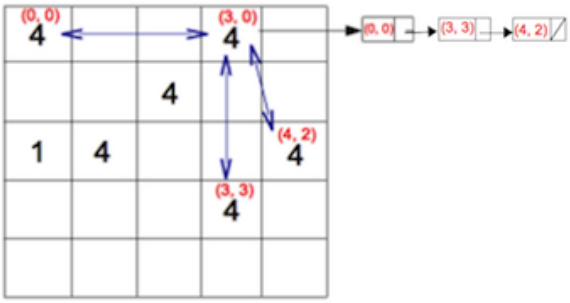


FIGURE 4.1 – Exemple de TabVoisins avec la case $(3, 0)$ qui a la valeur 4 et sa liste de voisins possibles

On utilise sur cette structure des fonctions « basiques » de création et d'accesseurs :

- `new_TabVoisins_from_Probleme` qui permet de créer un nouveau TabVoisins représentateur d'un problème donné ;
- `TabVoisins_getWidth` qui renvoie sa largeur ;
- `TabVoisins_getHeigth` qui renvoie sa hauteur ;
- `TabVoisins_setValeurs` qui permet d'assigner une valeur à une case existante ;
- `TabVoisins_getValeurs` qui renvoie la valeur d'une case existante ;
- `TabVoisins_setVoisins` qui permet de spécifier les voisins potentiels d'une case ;
- `TabVoisins_getVoisins` qui renvoie la liste des voisins possibles d'une case.

Chapitre 5

Résolution du puzzle Link-Pix

Le projet a connu différentes étapes de développement. En effet, le solveur étant lui-même découpé en plusieurs parties, nous avons tout d'abord travaillé sur la partie « mise à jour » des voisins, avant de passer à la construction des chemins. Nous avons ainsi une partie « raffinage » qui a pour objectif de simplifier le problème avant la résolution.

5.1 De sous-problème en sous-problème

5.1.1 Modification des voisins

La première partie du solveur vise à réduire l'ensemble des voisins possibles pour un indice. Ainsi, une coordonnée n'ayant qu'un voisin est, dans le cadre de puzzles qui ont une solution, le seul voisin possible de ce voisin-ci. On peut donc mettre à jour la liste des voisins de cette coordonnée.

Mais les anciens voisins peuvent également se retrouver à ce moment-là avec un unique voisin, il faut par conséquent répercuter la fonction sur eux si c'est le cas.

Algorithme 2 : affecterVoisin, qui modifie la liste des voisins de c_2, avec c_2 étant l'unique voisin de c_1

Données : c_1,c_2 des coordonnées, avec c_1 n'ayant que c_2 comme voisin ;
TabVoisin la structure contenant tous les voisins pour chaque coordonnée ;
Variables : L1, une liste de coordonnées ;
$L1 \leftarrow TabVoisins\_getVoisins(c_2)$;
Pour chaque *coordonnée e* **de** *L1* **faire**
  Si $e \neq c_1$ **alors**
    $oterVoisin(c_2, e, TabVoisins)$;
    $supprimerdansLCoord(e, L1)$;
  finsi
  $TabVoisins\_setVoisins(c_2, L1)$;
finpour

Cette fonction appelle pour chaque coordonnée voisine de c_2 la fonction `oterVoisin`. Cette dernière modifie la liste des voisins de e pour supprimer c_2, et appelle `affecterVoisin` si la liste des voisins à la fin est de longueur 1.

Ainsi, lorsque l'on a une coordonnée c ayant un unique voisin, un appel d'`affecterVoisin` sur c et son unique voisin se répercutera sur les autres coordonnées « en contact » avec ce couple-là. On peut ainsi espérer isoler de la sorte des couples de coordonnées, simplifiant ainsi le problème.

Algorithme 3 : oterVoisin, qui supprime c_1 de la liste des voisins de c_2

Données : c_1,c_2 des coordonnées ;
TabVoisins la structure contenant tous les voisins pour chaque coordonnées ;
Variables : L1, une liste de coordonnées ;
$L1 \leftarrow TabVoisins\_getVoisins(c_2)$;
$supprimerdansLCoord(c_1, L1)$;
$TabVoisins\_setVoisins(c_2, L1)$;
Si *L1 est de longueur 1* **alors**
  $affecterVoisin(c_2$, l'élément de L1 $, TabVoisins)$;
finsi

Ces fonctions se situent dans le module « Mise à jour des voisins » (fichiers `majVoisins.c` et `majVoisins.h`) du programme (voir A.1).

5.1.2 Existence d'un chemin

Un deuxième algorithme nous permettant de réduire la liste des voisins possibles est l'algorithme qui teste pour deux coordonnées données, supposées voisines, s'il existe un chemin entre ces deux. Ainsi, si aucun chemin n'existe, on peut en déduire que ces deux coordonnées ne sont en réalité pas voisines.

Algorithme 4 : existeChemin, qui teste s'il existe un chemin entre c_1 et c_2

Données : c_1, c_2 des coordonnées ;
d la distance qui doit être parcourue entre c_1 et c_2 ;
TabVoisins la matrice contenant les chemins et les indices ;
L une liste de coordonnées contenant les cases déjà parcourues ;
Résultat : Le booléen qui vaut Vrai s'il existe un chemin.
Variables : L1, une liste de coordonnées ;
b, un booléen ;
Si $NON(c_1 \in L)$ **alors**
  $L \leftarrow creerListe(c_1, L)$;
  $L1 \leftarrow$Liste des cases adjacentes de c_1 ;
  Si $d = 2$ **alors**
    Renvoyer $c_2 \in L1$;
  sinon
    tant que $NON\ b\ ET\ L1 \neq NULL\ ET\ L1->info$ *n'est pas occupée* **faire**
      $b \leftarrow existeChemin(L1->info, c_2, d-1, TabVoisins, L)$;
    fin
    Renvoyer b ;
  finsi
finsi

Cet algorithme s'appelle de façon récursive et teste tous les chemins possibles, sans repasser deux fois par la même case.

La condition d'arrêt de cet algorithme est dans le cas où on a un chemin à parcourir de longueur 2. Dans ce cas-là, on teste si les cases sont adjacentes. Si ce n'est pas le cas, on progresse comme indiqué dans la partie analyse, en se déplaçant sur les cases adjacentes et en testant s'il existe un chemin de longueur $d-1$.

On pourra ainsi réduire la liste des coordonnées voisines possibles grâce à cette fonction de « tamisage », avant de passer à la partie construction.

5.1.3 Construction du début d'un chemin

La construction d'un chemin, ou plus « simplement » du début d'un chemin, peut permettre d'éliminer de nouvelles possibilités pour les listes de voisins possibles.

Ainsi, un chemin, même incomplet, permet la progression de la résolution du problème. L'algorithme que nous avons écrit est un algorithme récursif, qui écrit un morceau de chemin et qui, s'il le peut, le complète au fur et à mesure. On a ainsi la fonction qui construira les chemins, mais aussi une fonction qui permettra de « raffiner » les listes de voisins.

Algorithme 5 : consChemin, qui construit le début d'un chemin (si possible) entre c_1 et c_2

```
Données : c1, c2 des coordonnées qui sont uniques voisines l'une de l'autre ;
d la distance qui doit être parcourue entre c1 et c2 ;
TabVoisins la matrice contenant les chemins et les indices ;
Résultat : Un booléen valant Vrai si la construction du chemin a progressé, Faux sinon.
Variables : L1, une liste de coordonnées ;
s, un entier ;
c, une coordonnée ;
Si d = 2 alors
    Mettre à jour la grille avec 1 comme nouvelle valeur pour c1 et c2 ;
    Renvoyer Vrai ;
sinon
    s ← 0 ;
    L ← Liste des cases adjacentes joignables ;
    Pour chaque coordonnées e de L faire
        Si existeChemin(e,c2,TabVoisins,NULL) alors
            s ← s + 1 ;
            c ← e ;
        finsi
    finpour
    Si s = 1 alors
        Mettre à jour la grille avec 1 pour c1, et d − 1 pour c et c2 ;
        consChemin(c, c2, d − 1, TabVoisins) ;
        Renvoyer Vrai ;
    sinon
        Renvoyer Faux ;
    finsi
finsi
```

Cette fonction va s'arrêter si elle arrive dans le cas de deux indices adjacents et de chemin 2, ou dans le cas où il existe plus d'un début de chemin entre c_1 et c_2. S'il n'existe qu'un seul début de chemin possible, elle va alors construire ce début de chemin. Elle va ensuite se positionner sur ce début de chemin, via l'appel récursif, et tenter à nouveau de construire un morceau de chemin.

On pourra aussi appeler cette fonction dans l'autre « sens ». Car, en effet, il est possible que l'on ne puisse pas construire de morceau de chemin d'une coordonnée vers l'autre, mais que ce soit possible dans l'autre sens.

5.2 Solveur

Le solveur en lui-même est une succession d'appel sur les fonctions précédentes. On va ainsi « tamiser » le problème, tenter de voir si l'on a alors des chemins possibles, puis re-tamiser le problème, et ainsi de

suite jusqu'à ce que la solution apparaisse. La structure globale du solveur sera alors :

Algorithme 6 : Structure globale du solveur

tant que *le problème n'est pas résolu* **faire**
    Tamiser le problème ;
    Construire des morceaux de chemins ;
fin

La boucle principale se décomposera ainsi en deux sous-parties, appelant chacune des fonctions décrites dans la section précédente.

5.2.1 Boucle principale

Pour représenter le fait que le problème n'est pas encore résolu, nous avons fait le choix d'utiliser deux listes.

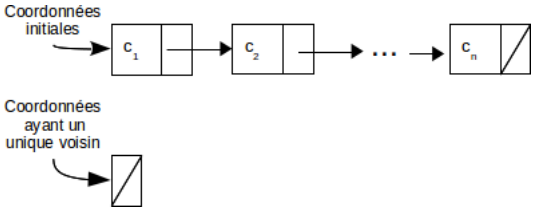


FIGURE 5.1 – Valeur des listes au début de la résolution

La première liste contient toutes les coordonnées de la grille qui n'ont pas été traitées, tandis que la seconde liste contient les coordonnées qui auront été analysées par le « tamisage » mais dont on n'aura pas encore construit le chemin. Ainsi, après le tamisage, les listes seraient de la forme :

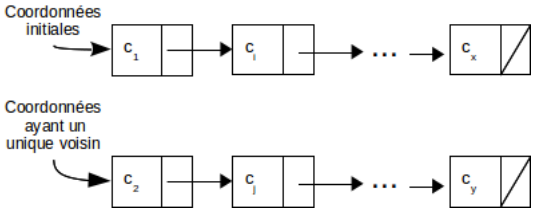


FIGURE 5.2 – Valeur des listes après tamisage

La partie construction, quant à elle, ne toucherait pas pas à la liste des coordonnées initiales mais modifierait la liste des coordonnées ayant un unique voisin, supprimant celles qui ont leurs chemins complétés.

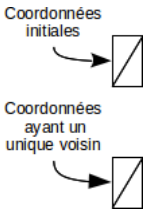


FIGURE 5.3 – Valeur des listes à la fin de la résolution

La résolution serait alors finie quand les deux listes seraient vides.

5.2.2 Tamisage

La partie tamisage du solveur serait principalement composé des fonctions de mise à jour des voisins dans le cas de coordonnées ayant un voisin unique ainsi que de la fonction qui teste s'il existe un chemin entre deux coordonnées.

La première partie du tamisage serait alors d'essayer, pour chaque coordonnées n'ayant qu'un unique voisin, de faire se répercuter ceci sur les autres coordonnées.

Algorithme 7 : Première partie du tamisage

Pour chaque *coordonnées c* **de** *la liste initiale* **faire**
  Si *c n'a qu'un seul voisin* **alors**
    $affecterVoisin(c, voisindec, TabVoisins)$;
  finsi
finpour

La fonction `affecterVoisin` devrait ainsi également modifier la liste initiale, retirant les éléments modifiés et n'ayant plus qu'un unique voisin.

La seconde partie du tamisage consisterait, pour chaque couple de coordonnées supposées voisines, à tester si un chemin existe entre elles. Si ce n'est pas le cas, alors, on doit supprimer le lien de « voisinage » qui existe entre elles.

Algorithme 8 : Deuxième partie du tamisage

Pour chaque *coordonnée c* **de** *la liste initiale* **faire**
  Pour chaque *coordonnée* c_2 **de** *la liste des voisins de c* **faire**
    Si $NON(existeChemin(c, c_2, valeurdec, TabVoisins, NULL)$ **alors**
      Retirer c de la liste des voisins de c_2 ;
      Retirer c_2 de la liste des voisins de c ;
    finsi
  finpour
finpour

On pourra également après appeler affecterVoisin sur ces coordonnées dans le cas où, après retrait, il n'existerait plus qu'un seul voisin possible.

5.2.3 Construction

Dans cette partie, le solveur tenterait de créer des chemins ou, si un chemin complet n'est pas possible, des morceaux de chemins dans le but d'affiner le problème lors du tamisage suivant.

Algorithme 9 : Construction des chemins

Pour chaque *coordonnées c* **de** *la liste des coordonnées ayant un unique voisin* **faire**
  consChemin(c,voisin de c,valeur de c,TabVoisins) ;
finpour

La fonction `consChemin` devrait alors également modifier une liste du solveur. En effet, celle-ci devrait supprimer les coordonnées dont on a fini le chemin de la liste, ne laissant dedans que celles en attente de pouvoir progresser dans la construction du chemin.

Chapitre 6

Manuel d'utilisation

6.1 Compilation du programme

Afin de compiler l'application, suivez ces différentes étapes :
— assurez-vous de disposer de GCC ;
— assurez-vous de disposer d'une version pour développeurs de GTK+ 2 ;
— depuis un terminal, naviguez jusqu'au dossier des sources de Link-Pix ;
— lancez `make`.

6.2 Démarrage du programme

Pour démarrer Link-Pix il suffit d'exécuter le binaire `link-pix` généré par `make`, que ce soit en ligne de commande ou par votre gestionnaire de fichiers.

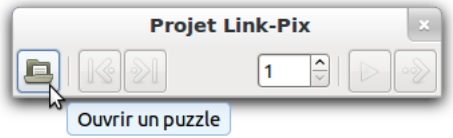


FIGURE 6.1 – Link-Pix à son lancement.

6.3 Ouverture d'un puzzle

Pour ouvrir un puzzle il suffit de cliquer sur le bouton prévu à cet effet dans la fenêtre de Link-Pix. Une boîte de dialogue apparaîtra alors, vous proposant de charger un fichier `.lnpx` (voir 6.2). Quelques fichiers `.lnpx` sont fournis avec les sources de l'application.

6.4 Résolution du puzzle

6.4.1 Fonctionnalités de l'interface graphique

L'interface graphique vous permet :
— d'ouvrir un fichier de puzzle ;
— de recharger le puzzle ;

— de résoudre le puzzle ;
— de définir le nombre d'étapes à résoudre à chaque pas, afin d'accélérer la résolution ;
— d'avancer pas à pas automatiquement dans la résolution du puzzle ;
— d'avancer pas à pas manuellement dans la résolution du puzzle ;
— d'afficher le puzzle tout au long de sa résolution avec des couleurs différentes indiquant son état.

6.4.2 Légende des couleurs

Lors de la résolution, chaque case peut apparaître d'une des manières suivantes :
— une case blanche sans nombre est une case actuellement vide ;
— une case blanche avec nombre est une case encore non résolue ;
— une case noire est une case remplie lors de la résolution du puzzle ;
— une case bleue est une case remplie au cours de la dernière étape de résolution, elle deviendra noire au tour suivant ;
— une case rouge est une case appairée dont la résolution a avancé au cours du dernier tour de résolution mais pas encore totalement résolue, le nombre est la distance restant à parcourir jusqu'à son pair.

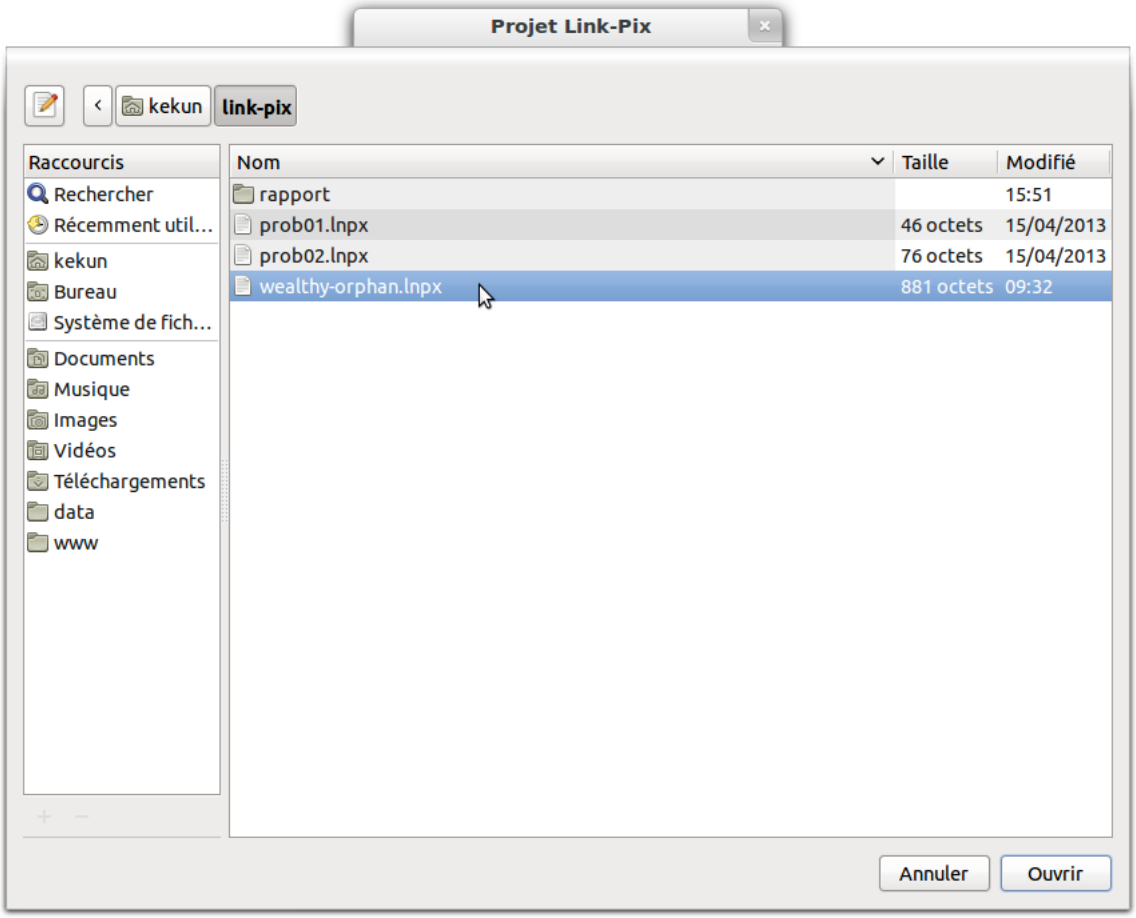


FIGURE 6.2 – La boîte de dialogue d'ouverture de puzzle de Link-Pix.

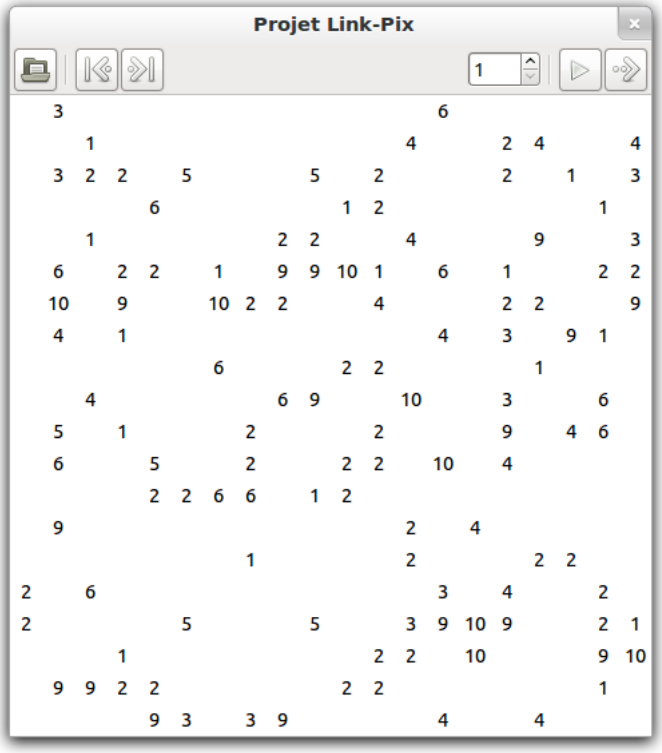


FIGURE 6.3 – Link-Pix lorsqu'un puzzle est ouvert.

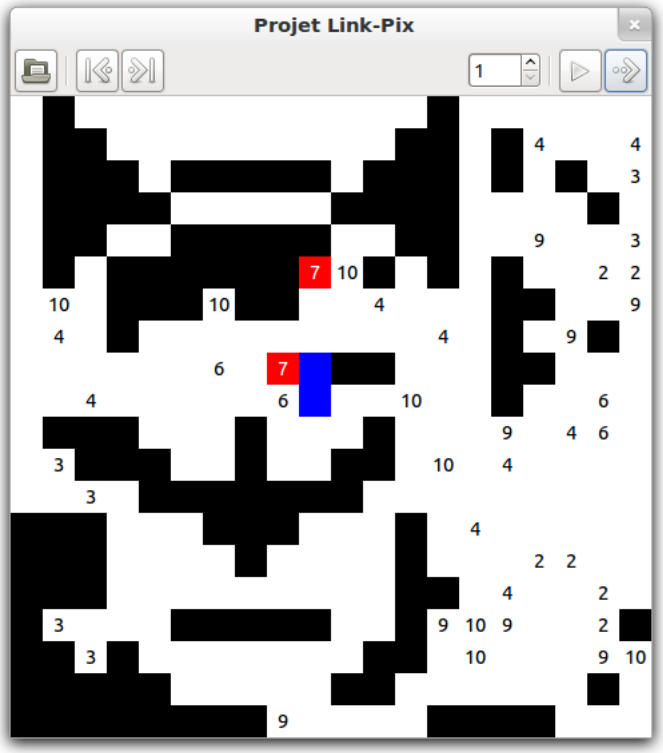


FIGURE 6.4 – Link-Pix en cours de résolution d'un puzzle.

Chapitre 7

Perspectives et conclusions

7.1 Perspectives

Le solveur que nous avons élaboré peut-être amélioré de bien des manières.

Un des premiers ajouts que l'on peut imaginer est l'intégration de la génération de puzzle à partir d'une image, partie du sujet non traitée faute de temps et de personnes. Cet ajout pourrait permettre par la suite, éventuellement, la création d'un logiciel de jeu Link-Pix, proposant des grilles à résoudre qui ont été générées par le logiciel à partir d'image.

Un ajout plus « simple » serait la gestion des images de couleurs. En effet, on peut imaginer ajouter une caractéristique pour chaque case qui serait la couleur. La possibilité de lier deux indices serait ainsi dépendante de cette couleur, réduisant finalement le champ des possibilités. Une réécriture simple du code permettrait d'intégrer cet ajout.

7.2 Conclusions

7.2.1 Fonctionnement de l'application

7.2.1.1 Généralités

Bien que notre solveur ait résolu les puzzles que nous lui avons proposés, nous n'avons pas la certitude qu'il puisse résoudre tous les puzzles à solution unique. Ainsi, nous construisons les débuts de chemins dont nous sommes certains, mais nous n'analysons pas si d'autres portions de chemins, situées non pas en début mais au milieu, sont sûres.

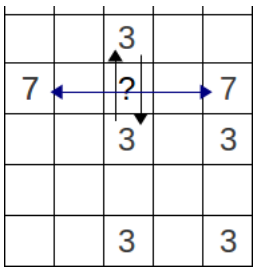


FIGURE 7.1 – On ne peut pas construire de façon sûre les débuts du chemin qui relie les 7, mais il passe forcément entre les deux 3, ce qui peut réduire les possibilités de voisinage.

En outre, notre solveur a été testé sur des puzzles de tailles raisonnables, et nous ne pouvons pas

savoir avec certitude si les performances sur de plus gros puzzles seront identiques. Cela dit, le solveur fonctionne sur les problèmes auquels nous l'avons soumis, et il semble efficace tant en utilisation mémoire qu'en temps de calcul.

Ainsi, malgré les problèmes d'organisation que nous avons eu, nous avons atteint notre objectif qui était de fournir un solveur opérationnel, et nous avons même pu créer une interface graphique, alors que nous ne pensions pas en avoir le temps.

7.2.1.2 Choix des outils

Le langage choisi pour développer ce projet, le langage C, aurait sans doute pu être remplacé par un langage de plus haut niveau, ce qui nous aurait permis un développement plus rapide, nous affranchissant ainsi des problèmes de gestion de mémoire. C'était cependant le langage le mieux connu de tous les membres de l'équipe, ce qui nous a permis de travailler plus facilement en groupe.

L'utilisation du gestionnaire de version Git nous a permis de travailler plus facilement à distance, peut-être au détriment de réunions de travail qui auraient pu être plus fréquentes. De même, du retard a été pris suite à certains oublis de soumission, faisant que la version en ligne n'était pas la plus à jour. Il s'est quand même globalement avéré être un outil de choix et a efficacement soutenu notre projet du début à la fin, nous aidant ainsi énormément.

L'utilisation du générateur de documentation Doxygen fut aussi positive. En effet, cet outil s'est avéré facile à utiliser et relativement puissant, nous fournissant ainsi une documentation claire. Nous avions ainsi accès aux caractéristiques d'une fonction beaucoup plus aisément que dans une recherche directement dans le code source.

7.2.2 Fonctionnement du groupe de travail

Les démarrages du projet ont été difficiles, notamment dû au fait que certains membres de l'équipe n'assistaient pas aux réunions. Nous avons ainsi pris du retard au début, les choix ne parvenant pas à être fixés dû aux manques de rigueur des absents.

Cela dit, au milieu du semestre, le retrait complet des étudiants inactifs sur le projet nous a permis de nous mettre au travail en groupe, de façon relativement efficace. Seule une semaine relativement chargée en contrôles nous a fait baisser d'intensité de travail, mais nous avons de façon globale eu un travail assez régulier, jusqu'à la fin où nous avons eu un rythme un peu plus élevé afin d'essayer de terminer dans les temps.

Au final, nous avons réussi à nous entendre, travaillant ensemble et en s'entraidant, de sorte que nous avons pu être relativement efficaces. Les erreurs trouvées étaient rapidement corrigées, notamment grâce à l'utilisation de Git, et nous avons pris du plaisir à travailler ensemble.

Annexe A

Documents d'analyse

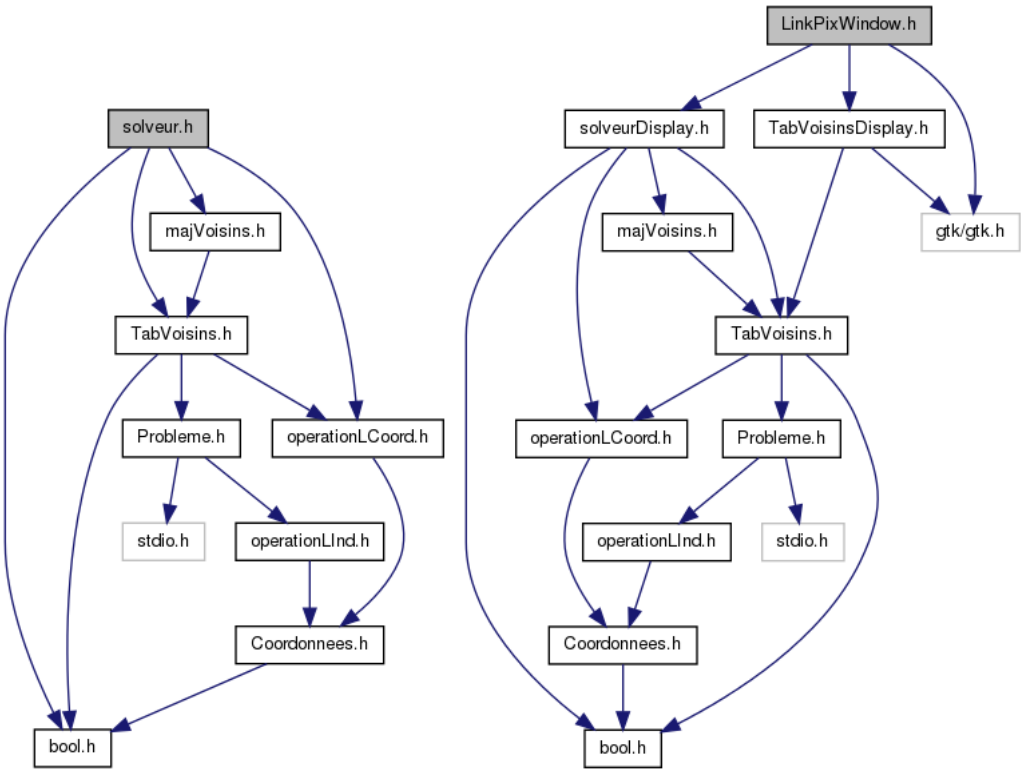


FIGURE A.1 – Graphe des dépendances du solveur à gauche et de du solveur graphique à droite

Les fichiers `solveur.h` et `solveur.c` ne sont pas utilisés dans le cadre du logiciel, mais sont conservés car ils contiennent le solveur en un seul morceau (et non pas des itérations découpées comme dans `solveurDisplay.h` et `solveurDisplay.c`, qui sont des adaptations du solveur pour la surcouche graphique).

Annexe B

Manuel d'utilisation

B.1 Compilation du programme

Afin de compiler l'application, suivez ces différentes étapes :
— assurez-vous de disposer de GCC ;
— assurez-vous de disposer d'une version pour développeurs de GTK+ 2 ;
— depuis un terminal, naviguez jusqu'au dossier des sources de Link-Pix ;
— lancez `make`.

B.2 Démarrage du programme

Pour démarer Link-Pix il suffit d'exécuter le binaire `link-pix` généré par `make`, que ce soit en ligne de commande ou par votre gestionnaire de fichiers.

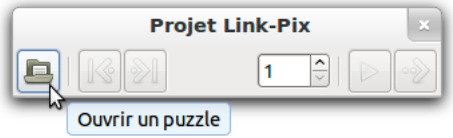


FIGURE B.1 – Link-Pix à son lancement.

B.3 Ouverture d'un puzzle

Pour ouvrir un puzzle il suffit de cliquer sur le bouton prévu à cet effet dans la fenêtre de Link-Pix. Une boîte de dialogue apparaîtra alors, vous proposant de charger un fichier `.lnpx` (voir 6.2). Quelques fichiers `.lnpx` sont fournis avec les sources de l'application.

B.4 Résolution du puzzle

B.4.1 Fonctionnalités de l'interface graphique

L'interface graphique vous permet :
— d'ouvrir un fichier de puzzle ;
— de recharger le puzzle ;

— de résoudre le puzzle ;
— de définir le nombre d'étapes à résoudre à chaque pas, afin d'accélérer la résolution ;
— d'avancer pas à pas automatiquement dans la résolution du puzzle ;
— d'avancer pas à pas manuellement dans la résolution du puzzle ;
— d'afficher le puzzle tout au long de sa résolution avec des couleurs différentes indiquant son état.

B.4.2 Légende des couleurs

Lors de la résolution, chaque case peut apparaître d'une des manières suivantes :
— une case blanche sans nombre est une case actuellement vide ;
— une case blanche avec nombre est une case encore non résolue ;
— une case noire est une case remplie lors de la résolution du puzzle ;
— une case bleue est une case remplie au cours de la dernière étape de résolution, elle deviendra noire au tour suivant ;
— une case rouge est une case appairée dont la résolution a avancé au cours du dernier tour de résolution mais pas encore totalement résolue, le nombre est la distance restant à parcourir jusqu'à son pair.

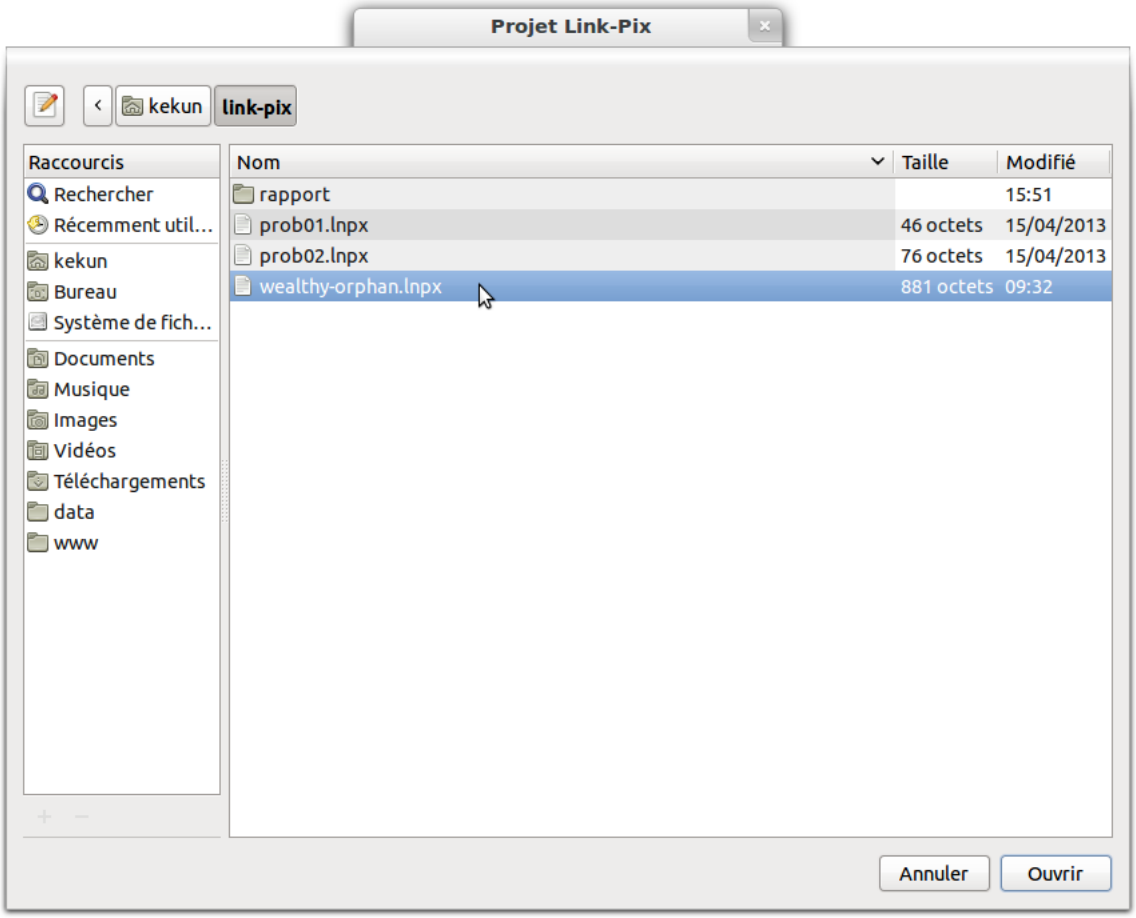


FIGURE B.2 – La boîte de dialogue d'ouverture de puzzle de Link-Pix.

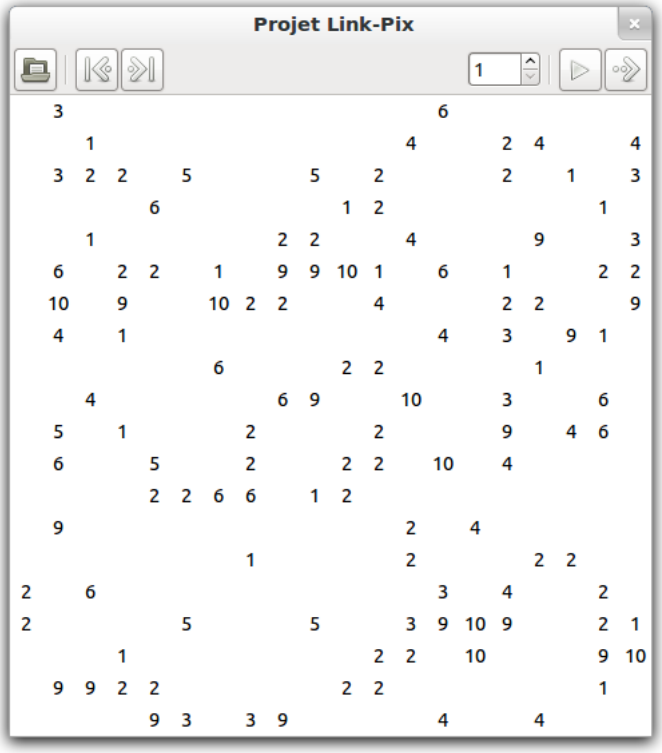


FIGURE B.3 – Link-Pix lorsqu'un puzzle est ouvert.

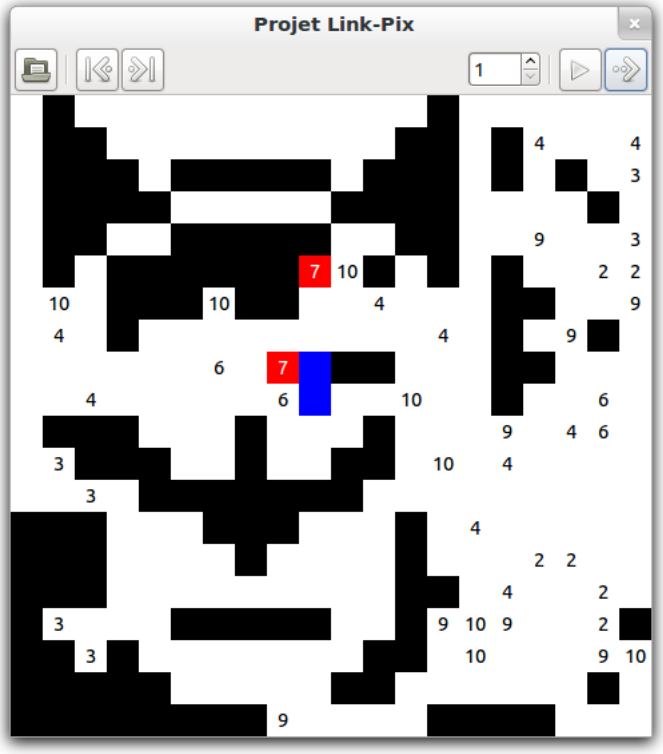


FIGURE B.4 – Link-Pix en cours de résolution d'un puzzle.