Recherche de solutions optimales au jeu de taquin

Sujet proposé par François Pottier
mailto:Francois.Pottier@inria.fr

Francois.Pottier[at]inria.fr *Difficulté : difficile (***)*

1 Préambule

1.1 Le jeu de taquin

Le jeu de taquin (*Fifteen Puzzle*) est constitué d'un plateau carré, subdivisé en seize cases, sur lequel on pose quinze pièces carrées numérotées de un à quinze. Il reste donc une case libre ou *trou*. Le seul mouvement ou *coup* autorisé consiste à faire glisser l'une des pièces adjacentes au trou vers celui-ci, ce qui revient à échanger leurs positions respectives.

Ce jeu, imaginé par Sam Loyd dans les années 1870, suscita immédiatement un grand intérêt à travers l'Occident. L'une des raisons peut-être de ce rapide succès était une récompense de mille dollars promise par Loyd à quiconque parviendrait à transformer une position de départ en une position d'arrivée fixées.

De façon relativement abstraite, on peut voir l'espace des configurations du jeu de taquin comme un graphe où chaque sommet représente une configuration et où les successeurs d'un sommet sont les configurations obtenues grâce à un coup quelconque. Chaque sommet a donc deux, trois ou quatre successeurs. Le jeu consiste alors à trouver un chemin à travers ce graphe entre un sommet de départ et un sommet d'arrivée fixés.

Loyd était taquin : les positions de départ et d'arrivée qu'il avait fixées ne sont en fait reliées par aucun chemin dans le graphe des configurations. Ce fait a été publié dès 1879, et se démontre aujourd'hui facilement [2]. Dans le cadre de ce projet, la position de départ sera variable, et la position d'arrivée sera celle illustrée à gauche de la figure 1.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

10	6	4	12
1	14	3	7
5	15	11	13
8		2	9

FIG. 1 – Notre position idéale et une position engendrée aléatoirement

1.2 Parcours en profondeur à aprofondissement progressif

La difficulté, d'un point de vue algorithmique, réside dans la grande taille du graphe. Le nombre de configurations possibles, donc le nombre de sommets du graphe, est de 16!, c'est-à-dire environ 2.10^{13}. On démontre mathématiquement que le graphe est divisé en deux composantes connexes, dont le nombre de sommets est donc de l'ordre de 10^{13}. Il est clair qu'un tel graphe ne peut être logé entièrement dans la mémoire vive d'une machine actuelle. Par conséquent, un algorithme de recherche de plus courts chemins tel que celui de Dijkstra [4, 3, 13], dont la complexité *en espace* est linéaire vis-à-vis du nombre de sommets du graphe, est inutilisable. Les algorithmes classiques de parcours en profondeur d'abord ou en largeur d'abord sont également impuissants : en effet, pour garantir leur terminaison, ces algorithmes colorent les sommets déjà rencontrés, ce qui leur confère également une complexité en espace linéaire vis-à-vis du nombre de sommets du graphe.

Il nous faut donc un moyen de réduire la complexité en espace de la recherche, quitte à augmenter sa complexité en temps. Comment faire ?

Une réponse originale à cette question a été proposée par plusieurs auteurs, dont Korf [9]. Puisque les algorithmes de parcours en largeur (*breadth-first*, de Dijkstra, et A* (*best-first*) utilisent des files d'attente dont la taille peut être linéaire vis-à-vis du nombre de sommets du graphe, on ne peut y recourir. À l'inverse, l'algorithme de parcours en profondeur (*depth-first*) exploite une pile dont la taille est seulement celle du chemin en cours d'exploration. Cela le rend beaucoup plus intéressant. Néanmoins, rappelons qu'il est impossible d'attribuer une couleur à chaque sommet, par manque d'espace. On doit donc recourir à une variante de l'algorithme de parcours en profondeur qui ne colore pas les sommets. Mais l'absence de coloration empêchera la détection des cycles ! Pour garantir la terminaison de l'algorithme, on devra alors stopper l'exploration à une certaine profondeur k. Et comme on ne connaît pas à l'avance la longueur du chemin optimal recherché, on devra effectuer *plusieurs* parcours en profondeur successifs, pour des valeurs croissantes du seuil k. On obtient ainsi un algorithme de *parcours en profondeur à aprofondissement progressif* (*depth-first iterative deepening*). Si le seuil k est incrémenté de 1 à chaque itération, on a la certitude que le premier chemin découvert sera optimal. Cette approche peut sembler coûteuse, car on parcourt de nombreuses fois les sommets proches du sommet initial. Mais, en réalité, le gros du travail se fait loin du sommet initial, et cette répétition n'est donc guère coûteuse, comme le démontre Korf [9].

1.3 Emploi d'une estimation de distance

Une idée indépendante, et également intéressante, est à la base de l'algorithme A* [6, 7], une amélioration de l'algorithme de Dijkstra. (Comme Dijkstra, A* est de complexité linéaire en espace.) L'idée est d'utiliser une notion externe de distance, par exemple une distance géométrique entre sommets, pour obtenir une approximation inférieure de la distance au sens du graphe. On exploite cette estimation de distance pour explorer en priorité les chemins qui semblent nous rapprocher le plus du but (*best-first search*). Cette stratégie permet de stopper l'exploration d'un chemin dès qu'il devient certain qu'il ne pourra être prolongé en un chemin optimal. On parle alors d'*élagage* de l'arbre de recherche. Par exemple, imaginons une carte routière de France, représentée sous forme d'un graphe dont les villes forment les sommets et dont les routes, étiquetées par leur longueur, forment les arêtes. La distance à vol d'oiseau entre deux villes fournit alors une approximation inférieure de la distance routière, c'est-à-dire de la distance au sens du graphe. De plus, elle est facile à calculer, si l'on connaît les coordonnées géographiques de chaque ville. On peut donc l'utiliser pour guider l'algorithme A* de recherche de plus courts chemins. Supposons, par exemple, que l'on recherche un chemin optimal de Paris à

Marseille. La distance routière entre Paris et Marseille est inférieure à la somme de la distance routière entre Paris et Lille, d'une part, et de la distance à vol d'oiseau entre Lille et Marseille, d'autre part. De ce fait, l'algorithme A*, lancé à partir de Paris, n'étudiera aucun chemin passant par Lille : il préférera étudier d'abord d'autre chemins, plus prometteurs, qui le conduiront à trouver un chemin optimal vers Marseille avant d'avoir dû examiner les chemins passant par Lille. Cela peut sembler naturel à un humain, qui n'aurait pas l'idée de passer par Lille. Néanmoins, un algorithme plus simple, comme celui de Dijkstra, étudierait tous les chemins en cercles concentriques autour de Paris, et examinerait donc des chemins passant par Lille, Bruxelles, et au-delà, avant de découvrir un chemin optimal vers Marseille ! On conçoit donc bien l'intérêt de cette idée.

Cette idée est-elle applicable au problème du jeu de taquin ? Autrement dit, disposons-nous d'une façon simple d'estimer inférieurement la distance entre deux configurations ? La réponse est oui. Puisque chaque coup ne déplace une pièce que d'une seule case, le nombre de coups nécessaires pour amener une pièce donnée de sa position courante à sa position idéale est borné inférieurement par la distance L_1 entre ces deux positions. De plus, puisque chaque coup ne déplace qu'une pièce, ce raisonnement s'applique indépendamment à toutes les pièces, et le nombre de coups nécessaires pour transformer une configuration donnée en la configuration idéale est donc borné inférieurement par la *somme*, prise pièce par pièce, des distances L_1 entre position courante et position idéale. Cette estimation de distance est connue sous le nom de *distance de Manhattan*, et est facile à calculer. Voici donc une précieuse première idée pour réduire l'espace de recherche : exploiter une estimation de distance externe, obtenue grâce à notre connaissance de la structure du graphe.

1.4 Combinaison

Les deux idées principales exposées ci-dessus peuvent être combinées : on peut définir un algorithme de parcours en profondeur à aprofondissement progressif exploitant une estimation inférieure de distance. Le plus simple de ces algorithmes est connu sous le nom de IDA* [9]. On peut alors parfois incrémenter le seuil k de plus de 1 entre deux itérations : la valeur de k adoptée pour chaque itération est la longueur estimée d'un chemin optimal passant par le plus proche sommet découvert mais non exploré (car situé au-delà du seuil) au cours de l'itération précédente. Un pseudo-code assez lisible pour IDA* est donné par Reinefeld et Marsland [12, figure 1].

1.5 Raffinements

On peut imaginer divers raffinements des idées décrites ci-dessus.

1.5.1 Calcul incrémental de l'estimation de distance

L'estimation de la distance entre une configuration donnée et la configuration idéale peut facilement être maintenue à jour lorsque la première est modifiée par un coup donné. Ce calcul incrémental permet d'économiser du temps.

1.5.2 Amélioration de l'estimation de distance

Dans le cas du jeu de taquin, on peut améliorer l'estimation de distance suggérée plus haut, à savoir la distance de Manhattan, en tenant compte des *conflits linéaires*. L'idée est que, si deux pièces se trouvent dans la ligne (ou colonne) souhaitée, *et si leurs positions sont inversées par rapport à celles souhaitées*, alors l'une d'elle devra nécessairement quitter temporairement cette ligne (ou colonne). La distance de Manhattan ne prend pas cela en compte, et on peut donc lui ajouter 2 sans risque de surestimer la distance réelle. Dans certaines positions, il existe plusieurs conflits de la sorte, et on peut donc ainsi obtenir une estimation de distance nettement plus précise que la distance de Manhattan [5].

Explorer un sommet coûte alors plus cher en temps, puisque le calcul de distance devient plus complexe. En contrepartie, le nombre de sommets à explorer diminue de façon importante, ce qui permet des gains importants en pratique.

1.5.3 Élagage des séquences de coups redondantes

Un raffinement important est d'élaguer immédiatement les séquences de coups *redondants*, en un sens que nous allons définir.

Dans le cas du jeu de taquin, chaque coup est réversible : il peut être immédiatement suivi d'un coup symétrique, de sorte que la séquence de ces deux coups n'a aucun effet. Il est clair que tout chemin qui contient une telle séquence ne peut être optimal, et peut être ignoré (élagué) sans danger. Si on étiquette les coups par H, B, G et D, selon que le trou est déplacé vers le haut, vers le bas, vers la gauche ou vers la droite, alors tout chemin contenant les mots HB, BH, GD ou DG peut être élagué. On peut implanter ce test en présentant l'étiquette de chaque coup, successivement, à un automate fini, qui peut exiger l'élagage lorsque certains états sont atteints. Cette optimisation simple permet un gain de temps considérable. En effet, le facteur de branchement (c'est-à-dire le nombre de successeurs de chaque sommet) est effectivement réduit de 4 (au pire) à 3 (au pire), ce qui signifie que la complexité en temps de l'algorithme passe de $O(4^k)$ à $O(3^k)$.

Pour reconnaître les quatre mots précédents, un automate à cinq états suffit, que l'on peut construire manuellement. Mais ces mots ne sont pas les seuls redondants. Par exemple, on peut vérifier que, à partir d'une configuration quelconque, les séquences de coups DBGHDB et BDHGBD soit sont toutes deux interdites, soit conduisent à la même configuration. En d'autres termes, ces séquences identifient des cycles de longueur 12 dans le graphe des configurations. Par conséquent, s'il existe un chemin optimal contenant la séquence DBGHDB, alors il en existe également qui ne la contient pas – à sa place, il contient la séquence BDHGBD. Nous pouvons donc élaguer tous les chemins contenant la séquence DBGHDB, à condition de conserver ceux contenant la séquence BDHGBD. Bien sûr, le choix inverse serait également possible. En utilisant ce même raisonnement, on peut interdire en tout quatre mots de longueur 6, en plus des quatre mots de longueur 2 déjà identifiés, et reconnaître ces mots à l'aide d'un automate à 29 états.

En poussant ce raisonnement, on peut identifier des séquences de coups redondantes de plus en plus longues. Naturellement, il serait illusoire d'espérer découvrir ces séquences et construire l'automate correspondant de façon purement manuelle. Heureusement, il est possible de réaliser un apprentissage automatique [14].

2 Détail du sujet

Le but est de trouver un chemin optimal entre une configuration de départ quelconque et la configuration idéale donnée à gauche de la figure 1, et ce dans un temps raisonnable.

J'insiste sur le fait que la complexité de l'algorithme proposé est *exponentielle* vis-à-vis de la longueur du chemin recherché, et que, selon la configuration initiale, certaines instances du problème peuvent être dix, cent, mille ou dix mille fois plus difficiles que d'autres. Par conséquent, on ne cherchera pas à grignoter un facteur deux en efficacité en modifiant le code de telle ou telle façon superficielle. Seuls des choix profonds, de nature algorithmique, sont à même d'apporter de véritables gains de temps.

Le langage d'implantation n'est pas imposé. Suivant la tradition propre au cours d'Informatique Fondamentale, on recommande d'employer Java ou Objective Caml.

2.1 Interface

2.1.1 Entrée

Votre programme lira la configuration de départ sur son entrée standard. Une configuration s'écrit, sous forme textuelle, comme une liste de seize entiers distincts compris entre 0 et 15 séparés par des blancs et terminée par un retour à la ligne. Le i-ème entier de cette liste indique le numéro de la pièce qui occupe la i-ème case du plateau, la pièce numéro 0 représentant, par convention, le trou. Les cases du plateau sont numérotées de 0 à 15 de gauche à droite puis de haut en bas, de sorte que la configuration idéale située à gauche dans la figure 1 s'écrit

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

Pour prendre un autre exemple, la configuration représentée sous forme textuelle par la ligne

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

est donnée sous forme visuelle à droite de la figure 1.

2.1.2 Sortie

Votre programme affichera sur sa sortie standard une ligne de résumé, suivie éventuellement d'un chemin.

La ligne de résumé sera `FAILURE` si aucun chemin n'existe entre la configuration initiale proposée et la configuration idéale. Elle sera de la forme `OK n` si de tels chemins existent et si le plus court d'entre eux est de longueur `n`. (Nous écrivons donc `n` pour la représentation décimale à un ou plusieurs chiffres d'un entier.)

Dans le second cas, c'est-à-dire lorsqu'un chemin existe, votre programme exhibera ensuite un chemin optimal quelconque entre la configuration initiale proposée et la configuration idéale. Ce chemin sera affiché sous forme d'une suite de configurations, à raison d'une par ligne, la première étant la configuration initiale proposée et la dernière étant la configuration idéale. Cette suite de configurations comprendra donc $\mathtt{n} + 1$ éléments.

2.1.3 Jeu de tests

Si l'interface requise est spécifiée de façon précise, c'est parce que votre programme sera testé de façon automatique. Veillez donc bien à remplir les exigences ci-dessus.

Vous pouvez vous-même tester la correction de votre programme en exécutant un jeu de tests que je vous fournis [11]. Chaque fichier dont le nom est de la forme `*.[0-9]+` contient une configuration de départ, et le fichier dont le nom est obtenu en ajoutant le suffixe `.ref` contient la ligne de résumé que votre programme doit produire. (Les chemins optimaux n'étant pas uniques, le fichier `.ref` n'en contient aucun.)

Tous ces tests ne sont pas de la même difficulté – certains demandent d'explorer un plus grand nombre de sommets, et exigent donc un temps plus important. Pour ce qui concerne les tests proposés par Korf [9, Table 1], une idée de la difficulté est donnée par la colonne “Total Nodes”. Le nombre de sommets que votre programme explore ne sera pas nécessairement égal à au nombre de sommets explorés par l'algorithme de Korf, car les optimisations que vous aurez implantées ne seront pas nécessairement les mêmes, mais cette table montre clairement que la différence de difficulté entre deux tests peut être de plusieurs ordres de grandeur.

Votre programme sera testé à l'aide d'un jeu de tests qui comprend celui-ci.

2.2 Algorithme

Je vous propose de procéder de la façon suivante. L'ordre des diverses étapes n'est qu'indicatif et il vous est possible de procéder différemment.

1. Décidez quelles opérations doivent être proposées par la structure de données qui implante le graphe des configurations. En bref, il devra s'agir d'un graphe où un sommet – la configuration idéale – est distingué ; où les arêtes sont étiquetées par les lettres H, B, G ou D ; où, étant donné un sommet, on peut obtenir une liste des arêtes qui en sont issues ; et où, étant donné un sommet, on peut obtenir une estimation de sa distance à la configuration idéale. Cette spécification pourra être concrétisée, en Java, par l'écriture d'une interface `GrapheConf`, ou bien, en Objective Caml, par l'écriture d'un type de modules `GrapheConf`.
2. Écrivez une implantation concrète du graphe des configurations. Cela prendra la forme, en Java, d'une classe qui implémente l'interface `GraphConf`, ou bien, en Objective Caml, d'un module de type `GraphConf`. En plus des opérations ci-dessus, il faudra ajouter une opération permettant de transformer une configuration, représentée sous forme externe (§2.1.1), en un sommet représenté sous forme interne. Je rappelle que l'efficacité de ces opérations n'est pas un souci premier. On les implantera donc, dans un premier temps, de la façon la plus simple possible. On pourra les optimiser, par exemple en utilisant des tables pré-calculées pour accélérer le calcul de l'estimation de distance, dans un second temps.
3. Écrivez une implantation de l'algorithme IDA*. L'algorithme devra effectuer une recherche à partir d'un sommet fixé et produire un chemin optimal vers le sommet idéal ou bien échouer. Pour vous guider, vous pourrez consulter la définition de l'algorithme A* [6], qui explique le principe de l'estimation de distance, ainsi que la présentation originale de l'algorithme IDA* [9]. Un pseudo-code assez lisible pour IDA* est donné par Reinefeld et Marsland [12, figure 1], dans un article par ailleurs sans grand intérêt. IDA* est un algorithme très simple, dont la réalisation ne devrait pas dépasser une centaine de lignes de code. N'hésitez pas à insérer des instructions d'affichage pour connaître en temps réel le nombre d'itérations effectuées, le nombre de sommets explorés à chaque itération, etc. Votre programme peut afficher ce qu'il souhaite sur le canal d'erreurs standard sans gêner le jeu de tests automatique.
4. Combinez tout cela pour écrire un programme satisfaisant la spécification. Vérifiez qu'il passe avec succès les tests les plus faciles, par exemple `facile.*`, `korf.009` ou `korf.012`.

Ceci constitue l'effort minimal. Ensuite, pour faire mieux, vous pourrez enchaîner avec les étapes suivantes :

1. Si vous le souhaitez, vous pouvez engendrer aléatoirement vos propres configurations de départ. Dans ce cas, l'article d'Archer [2] vous sera utile pour éviter d'engendrer des configurations inaccessibles. Un peu de réflexion est nécessaire pour comprendre comment déterminer si une configuration est ou non accessible.

2. Généralisez votre implantation de l'algorithme IDA* pour que l'étiquette de chaque coup soit soumise à un automate et pour que l'exploration cesse immédiatement si l'automate reconnaît une séquence de coups. Écrivez à la main un automate reconnaissant les séquences HB, BH, GD et DG. Combinez le tout et appréciez le gain de temps obtenu !
3. Faites en sorte que l'estimation de distance associée au sommet en cours d'exploration soit mise à jour de façon incrémentale à chaque coup (§1.5.1).
4. Raffinez l'estimation de distance pour prendre en compte les conflits linéaires (§1.5.2). La définition exacte de la nouvelle estimation de distance est donnée par Hansson *et al.* [5]. Un peu de réflexion est nécessaire pour l'implanter efficacement. Les auteurs de cet article suggèrent d'employer des tables pour précalculer une partie des informations utiles, mais ne donnent pas de détails...
5. Construisez un second programme capable d'effectuer un apprentissage automatique de l'automate d'élagage (§1.5.3), selon les idées de Taylor et Korf [14]. Ce programme explorera le graphe des configurations d'un jeu de taquin de taille 7x7, en partant d'une configuration dans laquelle le trou est au centre ; cela pour éviter les problèmes liés aux bords sur un plateau 4x4. Il effectuera une série de parcours du graphe, avec une borne de profondeur accrue de 1 à chaque parcours. À l'issue de chaque parcours, on déterminera quelles séquences de coups mènent aux mêmes configurations, et on en déduira si certaines séquences sont préférables à d'autres. Une séquence s_1 est *préférable* à une séquence s_2 si et seulement si (i) ces deux séquences mènent toujours à la même configuration lorsqu'elles sont appliquées en partant d'une même configuration ; (ii) la séquence s_1 est plus courte, au sens large ; et (iii) la séquence s_1 est applicable à toute configuration à laquelle la séquence s_2 est applicable. (La troisième condition est nécessaire pour prendre en compte les effets dûs aux bords du plateau.) Il s'agit d'un préordre, que l'on peut transformer en un ordre strict en ajoutant la condition suivante : (iv) la séquence s_1 est inférieure à la séquence s_2 pour un ordre total strict quelconque, par exemple un ordre lexicographique. L'ensemble des séquences pour lesquelles une séquence strictement préférable existe pourra alors être considéré comme redondant. On construira un automate qui reconnaît cet ensemble, en utilisant par exemple l'algorithme d'Aho et Corasick [1]. L'automate sera enregistré dans un fichier. On pourra utiliser cet automate au sein même du programme d'apprentissage, dès le parcours suivant. On peut espérer mener cette phase d'apprentissage jusqu'à la profondeur 13 ou 14, semble-t-il, après quoi elle devient trop coûteuse et de toute façon peu rentable. On utilisera enfin l'automate au sein du programme principal (lequel le retrouvera dans le fichier où on l'aura stocké) pour guider l'algorithme IDA*.

3 Pour aller au-delà

On peut s'attaquer à un plateau encore plus grand [10]... ou changer de jeu : la littérature regorge de références à propos du Rubik's Cube, de Sokoban ou d'autres jeux encore. Un recueil de courts articles supervisé par Haym [8] constitue une introduction intéressante au domaine.

Références

[1] Alfred V. Aho and Margaret J. Corasick. `http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/`

papers/Aho-Corasick-1975.pdf Efficient string matching : an aid to bibliographic search. *Communications of the ACM*, 18(6) :333–340, 1975.

[2] Aaron F. Archer. http://www.research.att.com/~aarcher/Research/15puzzle.ps A modern treatment of the 15 puzzle. *American Mathematical Monthly*, 106 :793–799, November 1999.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction à l'algorithmique : Cours et exercices*. Sciences Sup. Dunod, 2002.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[5] Othar Hansson, Andrew Mayer, and Moti Yung. http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/papers/Hansson-Mayer-Yung-1992.pdf Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3) :207–227, September 1992.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/papers/Hart-Nilsson-Raphael-1968.pdf A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2) :100–107, July 1968.

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37 :28–29, 1972.

[8] Haym Hirsh. http://www.computer.org/intelligent/ex1999/pdf/x6008.pdf Playing with AI. *IEEE Intelligent Systems*, pages 8–18, November 1999.

[9] Richard E. Korf. http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/papers/Korf-1985.pdf Depth-first iterative-deepening : An optimal admissible tree search. *Artificial Intelligence*, 27(1) :97–109, September 1985.

[10] Richard E. Korf and Larry A. Taylor. http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/papers/Korf-Taylor-1996.ps.gz Finding optimal solutions to the Twenty-Four Puzzle. In *National Conference on Artificial Intelligence (AAAI)*, pages 1202–1207, 1996.

[11] François Pottier. http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/test.tar.gz Jeu de tests.

[12] Alexander Reinefeld and T. A. Marsland. http://www.cs.ualberta.ca/~tony/RecentPapers/pami94.pdf Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7) :701–710, 1994.

[13] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java : Graph Algorithms*. Addison-Wesley, 2003.

[14] Larry A. Taylor and Richard E. Korf. http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Pottier/pi/2005/papers/Taylor-Korf-1993.ps.gz Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 756–761, 1993.