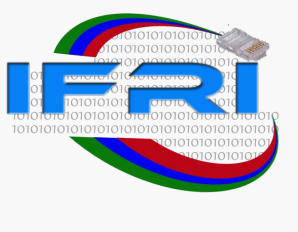
RÉPUBLIQUE DU BÉNIN



MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET
DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ D'ABOMEY-CALAVI (UAC)

INSTITUT DE FORMATION ET DE RECHERCHE
EN INFORMATIQUE (IFRI - UAC)

MÉMOIRE DE FIN DE FORMATION

POUR L'OBTENTION DU

Diplôme de Master en Science Informatique

THEME :

Évaluation d'heuristiques de recherche informée : Cas du n-puzzle

Présenté par :
IDJATON Koubouratou Olowountogni

Sous la direction de:
Dr VANDER MEULEN José, PhD
Ing HOUNDJI V. Ratheil, PhD Student

Année Académique : 2015-2016

3ème Promotion

Table des matières

Liste des algorithmes

Table des figures

Liste des tableaux

Dédicace

A mon papa chéri Aboudou Raïmi IDJATON,
pour ces trente années de passion, de travail acharné et de sacrifices continus.

Remerciements

Je tiens à adresser mes sincères remerciements :

A tous ceux qui ont d'une quelconque manière oeuvré à la réalisation de ce travail.

Aux professeurs Norbert HOUNKONNOU et Marc LOBELLE, les promoteurs du CeFRI.

Au feu professeur Semiyou ADEDJOUMA, promoteur adjoint du CeFRI et au professeur Ezinvi BALOICHA, coordinateur des études du CeFRI.

Au professeur Eugène EZIN, Directeur de l'IFRI et à toute son administration.
A tous les professeurs qui ont contribué à nous former pendant tout notre cursus.

A nos encadrants pour leur précieux appui dans la réalisation de ce travail.

A tous mes camarades de l'IFRI toutes promotions confondues. Tout particulièrement Alfred, Armand, Fabrice, Jaurès, Jean-Romuald, Lionel et Zansouyé.

A mes amis. Une pensée particulière à Marius et Solange.

A ma famille. Une pensée particulière pour ma mère, mes frères Mauriel et Babatoundé, et mon fiancé Arêmou.

Je ne saurais finir sans exprimer ma profonde gratitude à l'endroit des membres du jury pour avoir accepté d'évaluer ce travail.

Résumé

La résolution de problème par la recherche est l'un des domaines d'étude de l'Intelligence Artificielle. Elle se fait à travers deux grandes familles de techniques de recherche : les techniques de recherche non informée et les techniques de recherche informée.
Les techniques de recherche informée utilisent une fonction appelée heuristique afin d'opérer des choix relativement intelligents dans le processus de recherche.

Dans ce mémoire nous avons étudié différentes heuristiques (Distance de Manhattan, Linear conflict, Corner tile, Last move) qui ont été proposées pour résoudre le jeu de n-puzzle.

Notre contribution a été :

1. l'évaluation des performances des heuristiques existantes
2. la proposition d'une nouvelle heuristique que nous avons nommée Diagonal conflict. Une heuristique dont la version non admissible offre d'assez bonnes performances. Mais lorsque l'admissibilité est forcée les performances sont considérablement réduites. Cette heuristique offre de meilleures performances sur les instances du 15-puzzle et plus encore lorsque combiné à d'autres heuristiques existantes.

Mots clés : Intelligence artificielle, recherche informée, heuristique, diagonal conflict, all conflict

Abstract

The problem solving by search is one of the areas of study of Artificial intelligence. It is done through two major research techniques families : uninformed search techniques and informed search techniques.
Informed search techniques use a function called heuristic to make relatively smart choices in the research process.

In this paper, we studied different heuritics (Manhattan distance, Linear conflict, Corner tile, Last move) that were proposed to solve the n-puzzle game.

Our contribution was :

1. the improvement of the performance of existing heuristics
2. the proposition of a new heuristic which we named Diagonal conflict. An heuristic whose non admissible version offer a good performance. But when admissibility is forced performance is significantly reduce. This heuristic offers better performance on instances of the 15-puzzle and more when combined with others existing heuristics.

Keywords : Artificial intelligence, informed search, heuristic, diagonal conflict, all conflict

Introduction

En Intelligence artificielle, la résolution de problème par la recherche consiste à formuler le problème comme un problème de recherche puis à construire à base de cette formulation un arbre de recherche qui sera parcouru pour retrouver un but. Ce parcours de l'arbre se fait à l'aide de techniques de recherches qui pourrait être catégorisées en deux grandes familles. Les techniques de recherche non-informées et les techniques de recherche informées.

Les techniques de recherches non-informées consistent à explorer de manière aveugle l'arbre de recherche à la recherche d'un état but. En effet, il s'agit d'explorer les chemins possibles partant d'un point de départ (état initial) afin de déterminer si l'un des chemins explorés conduit à un état but. Ces techniques se révèlent parfois très coûteuses voir impossible en fonction du nombre d'états dans l'arbre de recherche. C'est pourquoi les techniques de recherches informées ont été développées.

Les techniques de recherches informées, utilisent des fonctions (heuristiques) afin de déterminer quels états parcourir afin d'atteindre un état but par un chemin optimal.

L'objectif principal de ce travail est de :

1. Étudier les différentes heuristiques qui ont été proposées pour résoudre le jeu de n-puzzle.
2. Implémenter et comparer ces heuristiques.
3. Proposer des améliorations de ces heuristiques.

Ainsi, notre travail a consisté, dans un premier temps à, réaliser une synthèse des différentes heuristiques (Distance de Manhattan, Linear conflict, Last move, Pattern database, Disjoint pattern database) qui ont été proposées pour résoudre le jeu de n-puzzle. Ces heuristiques ont ensuite été implémentées et comparées. Enfin, nous avons proposé de nouvelles heuristiques pour tenter d'améliorer l'existant.

Le présent document contient trois grandes parties. La première partie, État de l'art, où nous avons présenté les études et notions théoriques sur lesquelles ce sont appuyées nos travaux. Ainsi le chapitre 1 fait un rappel théorique de quelques notions de base en Intelligence Artificielle. Le chapitre 2 présente le jeu de n-puzzle et les travaux antérieurs effectués sur le n-puzzle. Ensuite la deuxième partie, Méthode, Outils et Développement, où nous expliquons d'abord à travers le chapitre 3 la méthode utilisée pour réaliser le travail, le choix de la méthode ; nous y présentons aussi la machine de test et le langage utilisé. Puis à travers le chapitre 4 où nous décrivons ce qui a été développé. Enfin la troisième partie, Résultats et Discussion, où nous présentons à travers les chapitres 5 et 6, nos propositions d'heuristiques inspirées de l'étude des heuristiques existantes et les résultats de leurs évaluations.

Première partie

État de l'art

Introduction

Dans cette partie, nous faisons un rappel théorique de quelques notions de base en Intelligence Artificielle. La formulation d'un problème comme problème de recherche, la recherche non-informée et la recherche informée ont été abordés. Puis, nous présentons le jeu de n-puzzle et les travaux antérieurs effectués sur le n-puzzle.

Chapitre 1

Rappel sur l'Intelligence Artificielle

1.1 Introduction sur l'Intelligence Artificielle

Le terme intelligence artificielle (couramment abrégé en IA) est né en été 1956 au cours d'une conférence au campus de Dartmouth College [1, 2], une université privée du nord-est des États-Unis.
Ce terme a depuis lors fait l'objet de plusieurs définitions au point où il devient assez difficile de toutes les regrouper en une seule et unique définition à laquelle toute la communauté scientifique s'accorderait. Ainsi l'Intelligence Artificielle est :

- « l'étude des facultés mentales à l'aide des modèles de type calculatoires » [3]
- « l'étude de comment faire-faire aux ordinateurs des choses pour lesquelles, pour le moment, les hommes sont meilleurs » [2]
- « l'étude de la conception d'agents intelligents » [4]
- « la science qui permet de faire faire aux machines des choses qui nécessiteraient de l'intelligence si elles étaient faites par l'homme » [18]
- « l'étude d'entité ayant un comportement intelligent » [5]
- « l'étude des moyens informatiques qui rendent possible la perception, le raisonnement et l'action » [8]

L'Intelligence Artificielle peut être subdivisé en plusieurs axes de recherches au sein desquels divers travaux sont effectués. Nous pouvons cités [9] :

- la conception de systèmes experts ;
- la représentation des connaissances ;
- la reconnaissance de la parole, de l'écriture et des formes ;

- la robotique ;
- l'apprentissage automatique ;
- la résolution de problèmes par la recherche.

C'est dans ce dernier axe cité que se situe ce travail. La résolution de problèmes par la recherche a pour but de représenter, d'analyser et de résoudre des problèmes concrets avec des algorithmes de recherche. Ces différents algorithmes sont souvent appliqués aux jeux de réflexions tels que les échecs, le jeu de dames, le sudoku, les puzzles, le jeu de go etc. En 1997, un programme du nom de Deep Blue de IBM a réussit à battre le champion du monde aux échecs Garry Kasparov [11], ce fut un des premiers exploits de l'IA dans ce domaine. En mars 2016, un programme du nom de AlphaGo a gagné 4-1 contre le légendaire Lee Sedol, le meilleur joueur de Go dans le monde au cours de la dernière décennie [12], ce fut l'un des plus récent progrès de l'IA dans le domaine des jeux.
Aujourd'hui la grande majorité de ces jeux sus cités sont facilement résolus sur la plupart des systèmes d'ordinateurs ou mobiles. Néanmoins, bien que déjà résolus, ces jeux demeurent d'intéressants sujets d'études en vue du perfectionnement des algorithmes existant pour obtenir de meilleurs résultats.

1.2 Formulation d'un problème comme un problème de recherche

L'utilisation de la recherche dans le processus de résolution de problèmes requiert une formulation abstraite du problème [9, 10, 17]. Cette abstraction nous permet d'ignorer un certain nombre de détails de la vie réelle, relatif au problème, afin de nous concentrer uniquement sur les éléments qui contribuent directement à résoudre le problème. Une formulation abstraite du problème n'est efficace que lorsque toute solution issue de cette formulation peut-être aisément appliquée au problème en situation réel.
La formulation d'un problème est définie à l'aide d'une machine à état par les différents éléments ci-après [9, 10] :

- État initial : Il faut préciser ici la représentation d'un état et représenter l'état de départ ou état initial.
- Actions ou successors function : permet de passer d'un état à un autre par une opération. Il faut définir ici l'ensemble des actions ou opérations possible sur un état. La fonction des successeurs (successors function) prend en entrée un état quelconque x et retourne $s(x)$ une liste de l'ensemble des successeurs possible de l'état x.
- Goal test : Il faut définir ici comment tester un état pour déterminer si il s'agit d'un état but ou pas. Les différents critères de l'état but recherché sont précisés ici.

- Fonction de coût : Cette fonction assigne un coût au chemin entre deux états. Soit un état E, le chemin représente l'ensemble des états parcourus de l'état initial I à E. La fonction de coût effectue la somme des coûts des différentes actions effectuées le long du chemin pour obtenir le coût du chemin.

Prenons par exemple le jeu de 8-puzzle, le tableau de jeu est une matrice 3×3 contenant des pions numérotés de 1 à 8 et un espace vide. Seul les pions adjacents à l'espace vide peuvent être déplacé dans l'espace vide. Le jeu consiste à quitter un tableau de jeu où les pions sont désorganisés, communément appeler l'état initial (Cf. État initial figure 1.1), et à parvenir à les réorganiser suivant un tableau de jeu représentant le but, communément appelé l'état but (Cf. État but figure 1.1).
Voici une formulation possible du 8-puzzle :

- État initial : Un état serait représenté par la position des pions sur le tableau de jeu. L'état initial est aléatoirement défini.
- Actions : Déplacer le vide vers le haut, le bas, la gauche, la droite ; si possible.
- Goal test : L'état but est pré-défini et connu. Le test consistera à vérifier si la position des pions dans l'état actuel correspond à celle des pions dans l'état but.
- Fonction de coût : 1 point par déplacement.

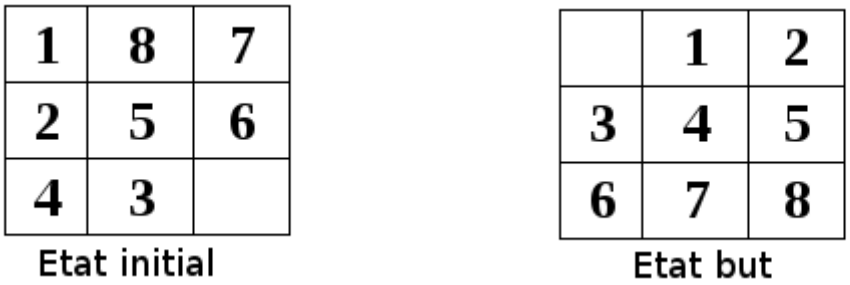


Figure 1.1 – 8-puzzle exemple

Cette formulation définies $(3 \times 3)! = 9! = 362880$ états initiaux possibles. Lorsque pour un état quelconque E, le pion 0 est de même parité que les pions du tableau de jeu, cet état est considéré solvable. Le test de solvabilité permet de savoir si un état initial quelconque du n-puzzle peut-être résolu ou non. En d'autres termes savoir si cet état initial peut aboutir à l'état but ou non.
Les travaux de Johnson et Storey en 1879 [32] ont révélé que pour un tableau de jeu $n \times n$ du n-puzzle seulement la moitié des états initiaux possibles sont solvables. Si on considère que chaque état est représenté par la suite des numéros des pions avec la cellule vide représentée par le numéro zero (0), un état initial peut-être résolu si et seulement si l'état but est une permutation paire de l'état initial. Ainsi il existe $(3 \times 3)! \div 2 = 9! \div 2 = 181440$ états initiaux

solvables. Soit la moitié des états initiaux possibles.
La parité du pion 0, s'obtient comme suit :

$$P = (\mid l_c^0 - l_b^0 \mid + \mid c_c^0 - c_b^0 \mid)\%2$$

avec l_c^0 le numéro de ligne du pion 0 dans sa position courante ; l_b^0 le numéro de ligne du pion 0 dans sa position but ; c_c^0 le numéro de colonne du pion 0 dans sa position courante ; c_b^0 le numéro de colonne du pion 0 dans sa position but et $P \in [0, 1]$.
La parité des pions du tableau de jeu s'obtient comme suit :

$$P = (\Sigma p(x))\%2$$

avec $p(x)$ une fonction qui lorsque le pion x n'est pas à sa position finale, effectue une permutation pour le ramener à sa position finale puis retourne 1 ; et lorsque le pion x est à sa position finale retourne 0.

La recherche de solution consiste en l'examination des différents états solvables dans le but de trouver un chemin menant de l'état initial à l'état but.
Une solution représente un ensemble d'actions ou un chemin qui, de l'état initial conduit à l'état but.
La solution optimale est le chemin ayant le coût le plus petit [15].

1.3 Les algorithmes de recherches

Dans le processus de recherche de solution, une large variété d'algorithmes peuvent être utilisées. Ces différents algorithmes sont souvent évalués et comparés entre eux suivant des critères bien précis. Il s'agit de :

- La complétude : Elle est vérifiée si l'algorithme garantie de trouver une solution si il en existe.
- L'optimalité : Elle est vérifiée si l'algorithme garantie que chaque solution trouvée est optimale.
- La complexité de temps : Il s'agit du temps que met l'algorithme pour trouver une solution. Elle est aussi représentée par le nombre de noeuds créés.
- La complexité d'espace : Il s'agit de l'espace mémoire utilisée par l'algorithme. Elle est aussi représentée par le nombre maximum de noeuds en mémoire. [9, 10].

Les complexités de temps et d'espace sont mesurées en fonction des variables telles que :

- b : le facteur de branchement maximal de l'arbre de recherche.

- d : la profondeur de la solution la moins coûteuse.
- m : la profondeur maximale de l'arbre de recherche.

Il existe deux principaux type d'implémentation des algorithmes de recherches : L'implémentation de type arbre (tree-search) et l'implémentation de type graphe (graph-search).
L'implémentation en arbre consiste à représenter le problème comme un graphe orienté où les noeuds sont les différents états. Cependant, l'implémentation en graph peut conduire à une boucle où les noeuds déjà visités sont revisités indéfiniment.
L'implémentation en graphe consiste à représenter le problème en prenant soin de ne pas répéter les noeuds déjà visités. Ce qui évite les boucles. Cependant, elle garde en mémoire tous les noeuds déjà visités.
Dans le cadre de ce travail, nous utiliserons l'implémentation de type graphe.

Il existe deux grandes familles d'algorithmes de recherche : La recherche non-informée et la recherche informée.
Recherche non-informée : La recherche non-informée est un ensemble de méthodes de recherche qui effectuent un parcours aveugle de l'arbre de recherche en vue de retrouver un but. Elles se révèlent parfois très coûteuses suivant la profondeur de l'arbre de recherche à parcourir.
Nous avons par exemple :

- Breadth First Search : Consiste à parcourir le graphe par la largeur. C'est à dire parcourir le graphe niveau par niveau en partant de la gauche vers la droite. Dans l'exemple de la figure 1.2 ci-dessous, les chiffres en noir représentent l'ordre dans lequel les différents états seront parcouru.
  Le Breadth first search est complet si b (le facteur de branchement maximal) est fini, ce qui est souvent le cas dans la plupart des problèmes courant dont le n-puzzle. La complexité en temps de cet algorithme est :

$$1 + b + b^2 + b^3 + ... + b^d + (b^d + 1 - b) = O(b^d + 1) = O(b^d)$$

  En effet, l'algorithme crée et examine tous les noeuds de profondeur au plus d et en plus, lors de examen des noeuds de profondeur d, crée tous les noeuds de profondeur $d + 1$ sauf b noeuds qui représentent les successeurs du noeud but. La complexité en espace est aussi de $O(b^d)$. Tous les noeuds générés sont gardés en mémoire au cours de l'exécution du Breadth first search. La mémoire est donc un véritable problème. Il est optimal si le coût de chaque action est de 1 [1, 2, 9].

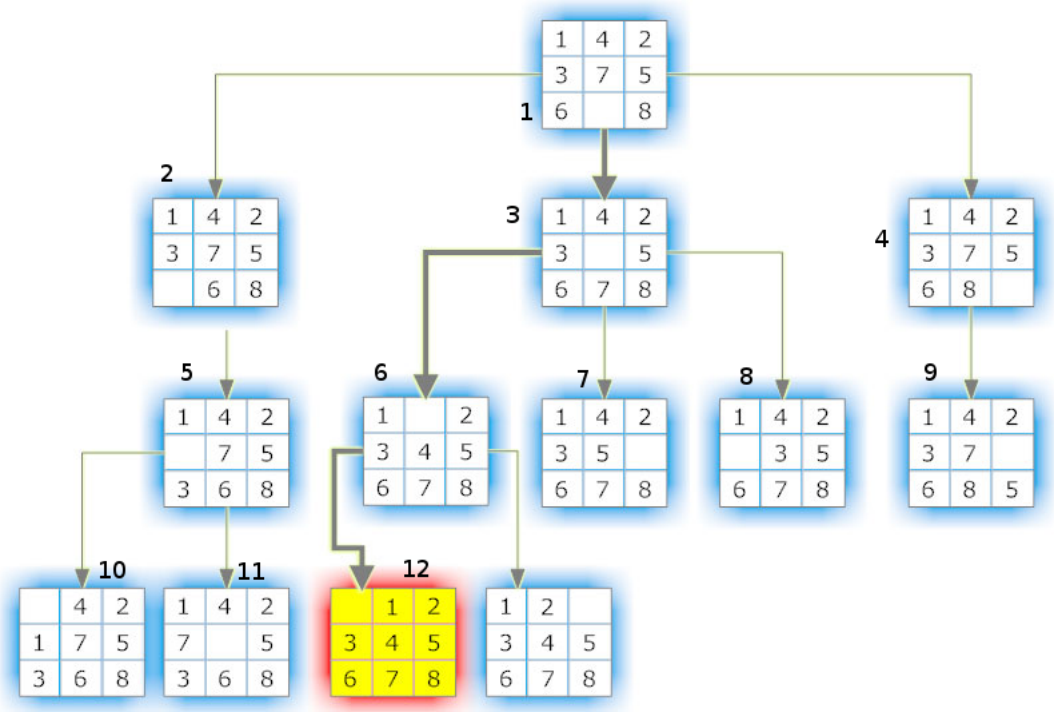


Figure 1.2 – Exemple du breadth first search pour le 8-puzzle

- Depth First Search : Consiste à parcourir le graphe en profondeur. Il s'agit d'aller à chaque fois à la plus basse profondeur avant de revenir explorer d'autres noeuds. Cette technique n'est pas optimale, ne garantie pas que le but trouvé est optimal. Sa compléxité en temps est de $O(b^m)$. En effet,au pire des cas l'algorithme crée et examine tous les noeuds du graphe. Le Depth first search a une complexité en espace de $O(b^m)$; elle offre une meilleure complexité en espace $O(mb)$ lorsqu'il s'agit d'un tree-search ; en considérant qu'au pire des cas le but se retrouve à la profondeur maximale. [9, 10, 15].

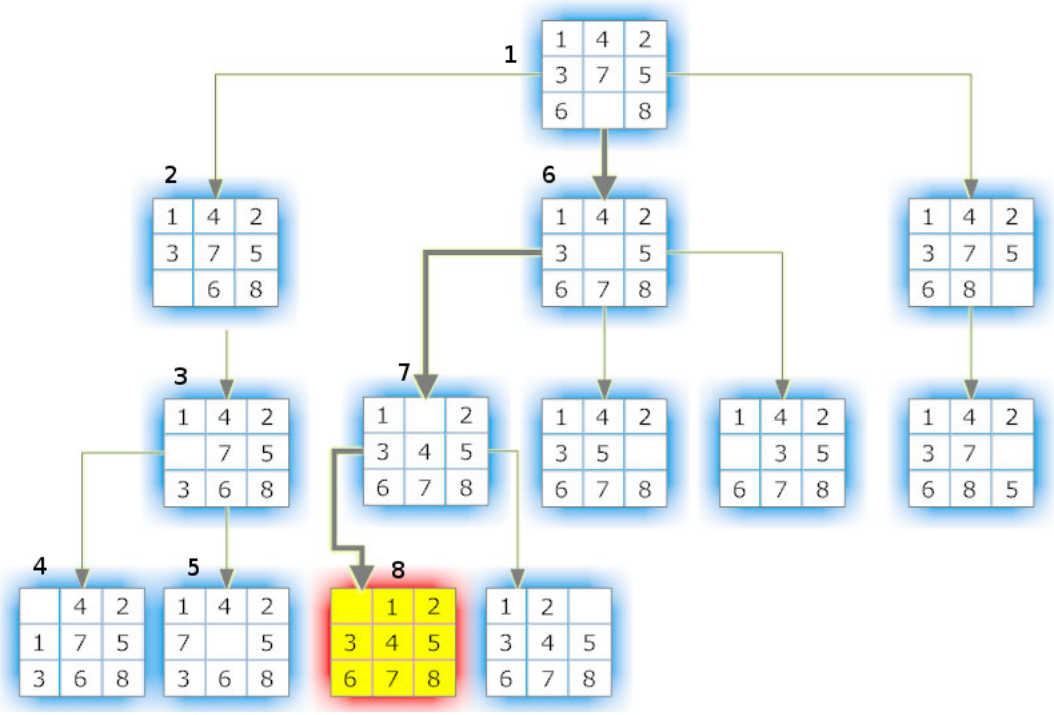


Figure 1.3 – Exemple du depth first search pour le 8-puzzle

- Limited Depth First Search : Cet algorithme vient pallier au problème de profondeur infinie du depth first search. Ainsi le Limited depth first search consiste à effectuer la recherche en profondeur(Depth First Search) en limitant la profondeur à atteindre lors du parcours de l'arbre de recherche. L'inconvénient principal de cette technique est le choix de la profondeur à fixée comme limite.
  En considérant l la limite choisie, Limited depth search n'est pas complet si la limite choisie est inférieure à la profondeur de la solution la moins coûteuse ($l < d$) et ne garantie pas l'optimalité si la limite choisie est supérieure à la profondeur de la solution la moins coûteuse $l > d$. Sa complexité en temps est $O(b^l)$. En effet,au pire des cas l'algorithme crée et examine tous les noeuds du graphe jusqu'à la limite l. Sa complexité en espace $O(b^l)$, mais $O(bl)$ lorsqu'il s'agit d'un tree-search ; en considérant qu'au pire des cas le but se retrouve à la profondeur maximale. [1, 9, 10, 15].

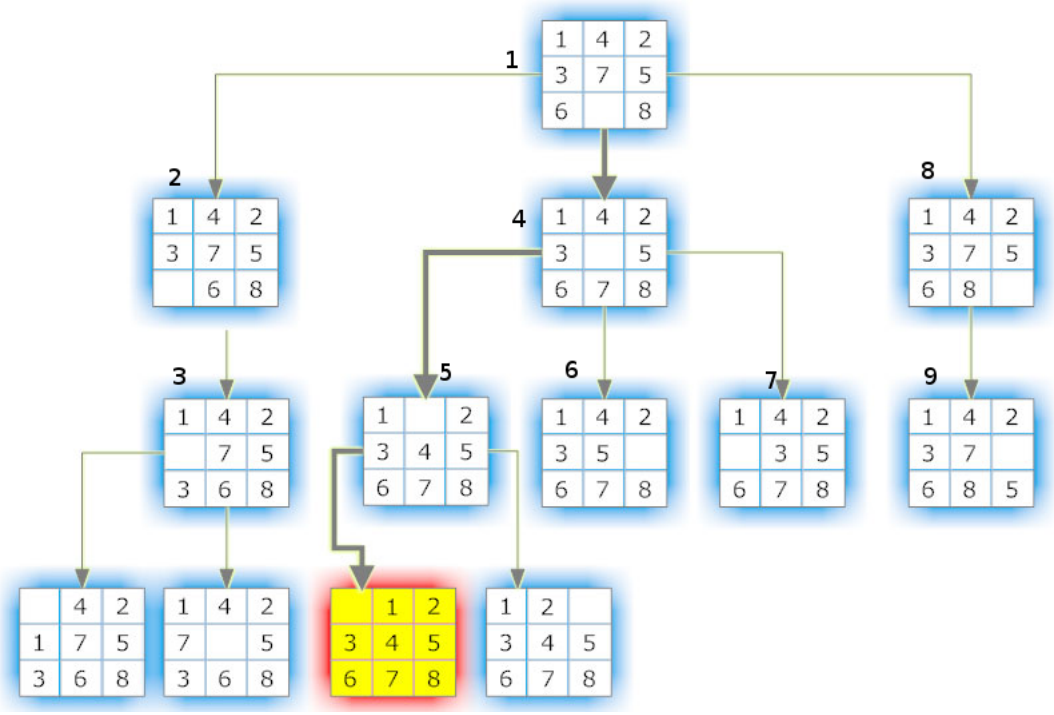


Figure 1.4 – Exemple du limited depth first search pour le 8-puzzle. Limite = 2

- Iterative Deepening Search : Il vient pallier au problème de choix de la profondeur à fixer pour la limite que pose le Limited depth first search. Ainsi, le Depth first search consiste à répéter la Limited depth first search en augmentant à chaque fois la limite jusqu'à atteindre le but ou la profondeur maximale de l'arbre de recherche. Dans l'exemple ci-après (figure 1.5), la limite a dans un premier temps été fixée à 2 (nombre en noir). Puis le but n'étant pas atteint et la profondeur maximale n'étant pas atteinte non plus, la limite a été augmentée et fixée à 3 (nombre en rouge).
  Iterative deepening est complet et est optimal si le coût de chaque action est de 1. Sa complexité en temps est de :

$$(d+1)b^0 + db^1 + (d-1)b^2 + ... + b^d = O(b^d)$$

  Sa complexité en espace de $O(b^d)$, mais $O(bd)$ lorsqu'il s'agit d'un tree-search [1, 2, 9, 15].

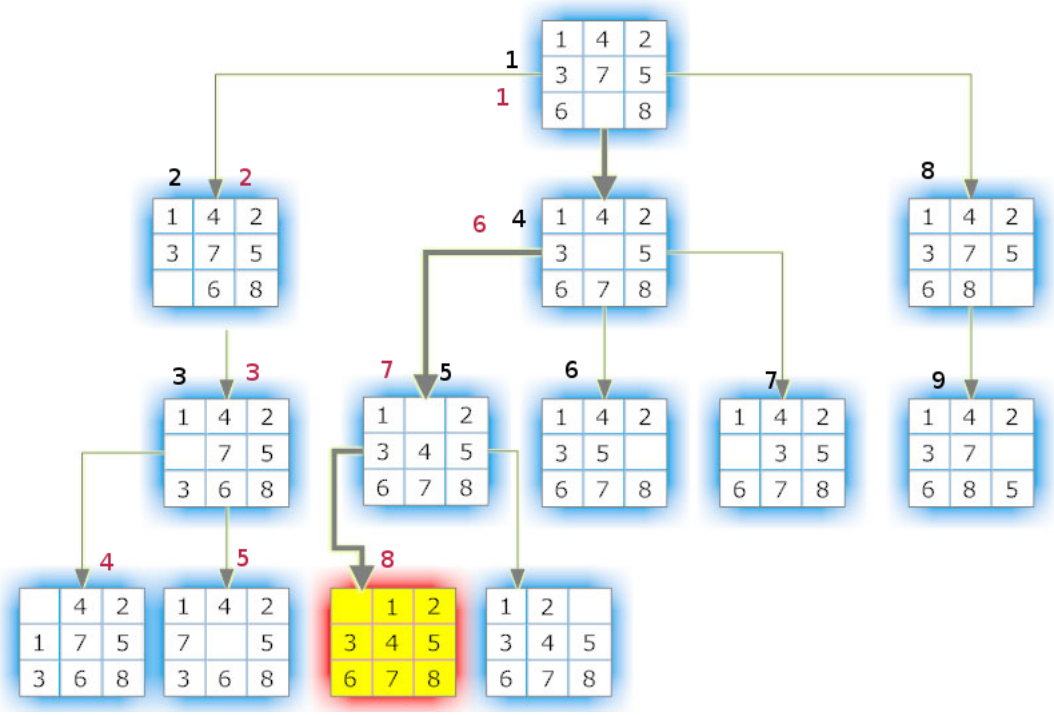


Figure 1.5 – Exemple du iterative deepening depth first search pour le 8-puzzle.

- Uniform Cost Search : Cette technique consiste à parcourir l'arbre de recherche en explorant en priorité les noeuds ayant les coûts les plus bas.
  Dans le cas du n-puzzle, le coût d'un déplacement étant de 1, appliquer Uniform Cost Search reviendrait à faire du Breadth First Search.
  Uniform cost est complet si le coût de chaque étape est strictement positif. Sa complexité en temps et en espace est difficile à établir avec précision et il est optimal [2, 9, 10, 15].

Recherche Informée : La recherche informée constitue un ensemble de méthodes de recherches qui effectuent le parcours de l'arbre de recherche en choisissant à chaque fois les noeuds les plus prometteurs. Ce choix des états prometteurs se fait grâce à une fonction appelée heuristique.

Une heuristique est une fonction qui permet d'évaluer la distance d'un noeud au noeud but [15]. Elle est souvent notée $h(n)$ avec n le noeud qu'on évalue. Cette fonction est utilisée par les algorithmes de recherche informée pour déterminer si un état est plus prometteur qu'un autre.

Une heuristique peut avoir différentes propriétés telles que [9, 10] :

- Admissible : Une heuristique est dite admissible lorsque pour tout noeud n, $h(n) < C(n)$, avec $C(n)$ le coût réel du noeud n au noeud but. Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but.

- Consistante (ou Monotone) : Une heuristique est dite monotone lorsque $\forall n$, $\forall n'$ fils de n, $h(n) \leq C(n, n') + h(n')$, où $C(n, n'$ représente le coût réel de n à n'.
  Toute fonction consistante est aussi admissible.

- Minorante : Une heuristique est dite minorante lorsque $\forall n$, $h(n) \leq h^\star(n)$, où $h^\star(n)$ est la longueur du chemin le plus court joignant n au but.

L'approche décrite ici s'appelle le Best-first Search. Elle consiste à utiliser une fonction d'évaluation, souvent notée $f(n)$ avec n le noeud qu'on évalue, et à toujours choisir le noeud qui minimise le plus la fonction $f(n)$. Il existe différentes approches de définition de la fonction $f(n)$:

- une approche consiste à choisir le noeud jugé le plus proche du noeud but par la fonction heuristique.
- une autre approche consiste à choisir le noeud dont le coût du chemin de la racine(noeud de départ) à la solution(noeud but) est jugée le plus petit.

Ces différentes approches différencient ces algorithmes de recherche les unes des autres. Au nombre des algorithmes de recherche informée existant, nous pouvons citer :

- Greedy Best-first Search : Il parcours l'arbre de recherche par le noeud indiqué par la fonction heuristique $h(n)$ comme étant celui se rapprochant le plus d'un noeud but. Sa fonction d'évaluation est donc égale à la fonction heuristique : $f(n) = h(n)$.
  Greedy Best-first Search n'est pas optimal et n'est pas complet, il peut se retrouver dans un cas de profondeur infinie. [1, 9, 10, 15].
- $A^\star$ Search : Il est une amélioration du Greedy best first search. Ainsi au lieu de considérer seulement le noeud indiqué par la fonction heuristique, $A^\star$ rajoute une information supplémentaire qui est la distance du noeud initial au noeud évalué, noté $g(n)$. Il parcours donc l'arbre de recherche en explorant en priorité le noeud dont le coût du chemin de la racine à la solution est jugé le plus petit. Sa fonction d'évaluation est la somme du coût du chemin de la racine au noeud évalué (n), noté $g(n)$, et de la fonction heuristique $h(n)$. On a ainsi : $f(n) = h(n) + g(n)$.
  $A^\star$ est complet et optimal si b est fini et si l'heuristique $h(n)$ utilisée est admissible ou consistante. Dans certain cas, $A^\star$ pose un problème d'espace car il garde en mémoire tous les noeuds générés [9, 10, 15, 17].
- Iterative Deepening $A^\star$ ($IDA^\star$) Search : Il est une variante de $A^\star$. Il combine l'approche du Iterative deepening avec celle du $A^\star$. $IDA^\star$ utilise une fonction d'évaluation $f(n)$ qui a les caractéristiques déjà exigées par $A^\star$; par contre, $IDA^\star$ ne suit pas strictement le principe de « développer le noeud qui minimise le plus $f(n)$ d'abord » mais plutôt « développer partiellement le noeud le plus profond d'abord pourvu que son évaluation ne dépasse pas un seuil alloué courant S » (Initialement S vaut $h(I)$, avec I l'état initial). $IDA^\star$ produit un premier fils X de I, puis un premier fils X' de X et ainsi de suite tant que l'évaluation de l'état le plus profond est inférieure ou égale à S. Si cette évaluation dépasse S, alors $IDA^\star$ remonte au carrefour de choix immédiatement précédent, soit Y l'état correspondant, et engendre un nouveau fils Y' de Y. S'il ne reste plus de choix, $IDA^\star$ lance une nouvelle fois l'exploration en profondeur, depuis I, mais en assignant

maintenant à S le minimum des valeurs d'évaluation ayant dépassé la valeur en vigueur jusqu'ici. $IDA^\star$ s'arrête lorsqu'il advient que le sommet à développer est le but ou lorsqu'il achève l'exploration de tout l'arbre. $IDA^\star$ vient pallier au problème de mémoire que pose $A^\star$ en gardant le moins possible de noeuds en mémoire [9, 10, 15, 17, 21].

Chapitre 2

Le Jeu de n-puzzle

2.1 Jeu de n-puzzle et Intelligence Artificielle

Un jeu peut-être définit comme étant une « activité d'ordre physique ou mental, non imposée, ne visant aucune fin utilitaire, et à laquelle on s'adonne pour se divertir, en tirer un plaisir » [6]. Les jeux ont pendant longtemps été et restent encore un terrain d'expérimentation idéal pour l'intelligence artificielle. Ils sont généralement fait de règles simples et peu nombreuses et de situations bien définis ce qui permet à l'ordinateur ou aux chercheurs en intelligence artificielle de travailler sur une échelle de problème plus réduite et assez simple à cerner pour pouvoir mesurer plus simplement l'évolution des techniques de résolutions [7].

Le succès de l'intelligence artificielle dans les jeux a commencé depuis BKG, conçu pour jouer au jeu de Backgammon par Hans Berliner dans les années 1970 et qui à sa version 9.8 en 1979 a été suffisamment fort pour battre le champion du monde Luigi Villa au Backgammon [9]. Un peu plus tard, Deep Blue, conçu par IBM qui en 1997 bat Garry Kasparov, champion du monde aux Échec[11]. En 2008, le monde de l'intelligence artificielle a connu aussi Chinook, impossible à battre au jeu de dame, mise au point par des scientifiques canadiens[13]. Ensuite vint en 2011, Watson, un système expert conçu par IBM qui a battu 2 champions au Jeopardy, un jeu dans lequel on donne la réponse et où il faut deviner la question.Watson a beaucoup évolué depuis[14]. Puis en 2016, AlphaGo, une intelligence artificielle conçue par DeepMind, une filiale de Google, a battu le champion Lee Sedol, meilleur joueur de Go dans le monde au cours de la dernière décennie [12].
En dehors de ces jeux dont le succès a marqué l'histoire, beaucoup d'autres jeux ont également fait l'objet d'études en intelligence artificielle. Au nombre de ceux-ci, nous pouvons citer : l'Othello, le pousse-pousse, le rubik's cube, le Marienbad, l'Awalé, le taquin ou n-puzzle qui est l'objet de cette étude.

Le jeu de taquin, aurait été imaginé par Sam Loyd dans les années 1870. Il est représenté par une matrice carrée de 4×4 dont 15 cellules sont remplies de pions numérotés de 1 à 15 et la dernière cellule laissée vide comme représenté sur la figure 2.1 ci-dessous.

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

Figure 2.1 – Jeu de taquin ou 15-puzzle

Le but du jeu est de partir d'une situation de départ où les pions ne sont pas dans l'ordre numérique correcte et d'aboutir à la situation finale où chaque pion est disposé tel qu'on obtient un ordre numérique correcte comme sur la figure 2.2 ci-après.

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

Figure 2.2 – Exemple de situation finale du jeu de taquin

Un déplacement dans le jeu consiste à pousser un pion de sa position horizontalement ou verticalement dans la cellule vide. Ainsi, on ne peut déplacer que les pions adjacent à la cellule vide.

Le jeu de taquin ou n-puzzle a évolué depuis sa création et possède aujourd'hui plusieurs variantes : 3×3, 4×4, 5×5,... $n \times n$ avec $n \epsilon N^{\star}$.

	1	2
3	4	5
6	7	8

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

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

Figure 2.3 – Exemple de quelques variantes de n-puzzle

« La maîtrise des n-puzzle pourrait être directement utile pour traiter des problèmes pratiques de parcage, magasinage, circulation (automobiles, wagons, marchandises, matériaux, informations...) » [20].

2.2 Heuristiques classiques du jeu de n-puzzle

Les principales heuristiques classiques existantes pour résoudre le jeu de taquin sont :

- Distance d'Euclide : La distance d'Euclide est égale à la racine carré de la somme des distances au carré entre chaque pion et sa position finale.

$$h(n) = \sqrt{\mid p_c^1 - p_b^1 \mid^2 + \mid p_c^2 - p_b^2 \mid^2 + ... + \mid p_c^{t\times t} - p_b^{t\times t} \mid^2}$$

  avec n le noeud évalué, t la taille du n-puzzle, p_c^x le pion x dans sa position courante et p_b^x le pion x dans sa position but.
  Prenons l'exemple de la situation de départ de la figure 2.4 ci-dessous :

	8	**1**
5	**7**	**4**
6	**2**	**3**

Figure 2.4 – Exemple de situation de départ pour évaluer les heuristiques

La distance d'Euclide est :

$$h(n) = \sqrt{1^2 + 3^2 + 3^2 + 1^2 + 2^2 + 0^2 + 1^2 + 3^2}$$

$$h(n) = 5.83$$

- Misplaced tile : Elle consiste à compter le nombre de pions qui ne sont pas à la position qui leur est destinée dans l'état but. Cette heuristique est admissible car elle relâche le problème en considérant qu'un seul mouvement suffit pour que tout pion qui n'est pas à sa place la rejoigne.

  $$h(n) = \Sigma P(x)$$

  avec n le noeud évalué ; $P(x)$ une fonction qui retourne 1 si le pion x n'est pas dans sa position but et 0 sinon.
  En considérant l'exemple de situation de départ de la figure 2.4 ci-dessus :

  $$h(n) = (1 + 1) + (1 + 1 + 1) + (0 + 1 + 1)$$

  $$h(n) = 7$$

  Nous avons regroupé entre parenthèse les chiffres correspondants aux pions situés sur la même ligne.

- Tiles out of row and column : Elle consiste à compter le nombre de pions qui ne se situent pas dans la ligne de leur position finale et additionner au nombre de pions qui ne se situent pas dans la colonne de leur position finale. Cette heuristique est admissible, car elle relâche le problème en ignorant 2 contraintes que sont : les pions se situant entre le pion évalué et la ligne et la colonne de sa position finale ; et la position réel du pion dans la ligne et la colonne de sa position finale.

  $$h(n) = \Sigma Pl(x) + \Sigma Pc(x)$$

  avec n le noeud évalué ; $Pl(x)$ une fonction qui retourne 1 si le pion x n'est dans la ligne de sa position but et 0 sinon ; $Pc(x)$ une fonction qui retourne 1 si le pion x n'est dans la colonne de sa position but et 0 sinon.
  En considérant l'exemple de situation de départ de la figure 2.4 ci-dessus :

  $$h(n) = (0 + 1 + 1 + 0 + 0 + 0 + 1 + 1) + (1 + 1 + 1 + 1 + 1 + 0 + 0 + 1)$$

  $$h(n) = 10$$

  Nous avons regroupé dans la première parenthèse les chiffres correspondants aux pions situés hors de la ligne de leur position finale puis dans la deuxième parenthèse les chiffres correspondant aux pions situés hors de la colonne de leur position finale.

- Distance de Manhattan [22] : C'est l'une des heuristiques les plus connues pour résoudre le n-puzzle. La distance de Manhattan consiste à compter pour chaque pion du tableau

de jeu, le nombre de cellules qui sépare sa position courante de sa position finale ; puis à sommer ce nombre pour tous les pions du tableau de jeu. On a :

$$h(n) = \Sigma(\mid p_c^x - p_b^x \mid)$$

avec n le noeud évalué ; p_c^x le pion x dans sa position courante et p_b^x le pion x dans sa position but. La distance de Manhattan est une heuristique admissible car elle relâche le problème en ignorant les différents déplacements que devront faire les pions qui occupent les cellules qui séparent tout pion de sa position finale.
Si on considère que sur le tableau de jeu chaque pion peut être identifié par son numéro de ligne et son numéro de colonne, une version plus détaillée de cette fonction donne :

$$h(n) = \Sigma(\mid l_c^x - l_b^x \mid + \mid c_c^x - c_b^x \mid)$$

avec n le noeud évalué ; l_c^x le numéro de ligne du pion x dans sa position courante ; l_b^x le numéro de ligne du pion x dans sa position but ; c_i^x le numéro de colonne du pion x dans sa position initiale et c_b^x le numéro de colonne du pion x dans sa position but.
En considérant l'exemple de la situation de départ de la figure 2.4 ci-dessus, la distance de Manhattan donne :

$$h(n) = (1+3) + (3+1+2) + (0+1+3)$$

$$h(n) = 14$$

Nous avons regroupé entre parenthèse les chiffres correspondants aux pions situés sur la même ligne.

- Linear conflict [22] : Elle constitue une variante de la distance de Manhattan. Le Linear conflict s'applique lorsque deux (2) pions sont dans la ligne ou la colonne de leur position finale mais sont inversés l'un par rapport à l'autre en fonction de leurs positions finales respectives.
  Par exemple, le pion portant le chiffre 1 et le pion portant le chiffre 2. S'ils se retrouvent sur la première ligne mais le pion 2 venait avant le pion 1, on serait dans un cas où il faut appliquer la Linear conflict. Cette heuristique consiste à prendre en compte, en plus de la distance de Manhattan, le fait qu'il faudrait que l'un des pions se déplace hors de la ligne ou de la colonne concernée pour laisser-passer l'autre puis y revienne afin de résoudre la permutation. Ainsi on ajoute 2 déplacements supplémentaires à la distance de Manhattan. Dans le cas où un état présente plusieurs conflits en ligne ou en colonne, il faut gérer les rapports entre les conflits pour ne pas surévaluer les coûts.
  Cette heuristique est admissible car elle relâche le problème en considérant que la cellule dans laquelle doit se déplacer le pion qui quitte sa position finale pour laisser-passer l'autre est vide.
  En considérant l'exemple de situation de départ de la figure 2.4 ci-dessus, nous obser-

vons que les pions 4 et 5 présentent un cas de linear conflict. De ce fait, il nous faudra ajouter 2 déplacements à la distance de Manhattan. La distance de Manhattan calculée précédemment est :

$$h(n) = 14$$

la linear conflict donne donc :

$$h(n) = 14 + 2 = 16$$

- Last moves [22] : La last move heuristic constitue aussi l'une des améliorations de la distance de Manhattan. Elle considère le dernier mouvement effectué pour obtenir l'état but. Il s'agit du mouvement pour permettre de positionner la cellule vide à sa place. En effet pour positionner la cellule vide, le pion 1*oun* est déplacé de la position de la cellule vide à sa positon finale. Ce qui signifie le pion 1 est dans la première colonne ou le pion n dans la première ligne. Étant donné que la distance de Manhattan ne tient pas compte de ce mouvement supplémentaire, deux déplacements peuvent-être ajoutés à la distance de Manhattan tout en maintenant l'admissibilité. De la même manière, le concept peut être élargit à l'avant-dernier mouvement effectué pour obtenir l'état but.

- Corner tiles [22] [20] : Elle constitue aussi l'une des améliorations de la Manhattan distance. La Corner Tile heuristic concerne les pions situés dans les coins du tableau de jeu. Lorsque pour une situation quelconque, différente de la situation finale, on se retrouve dans un cas où les pions adjacents à un coin sont à leurs positions finales et que le pion devant rester dans ce coin ne l'ai pas encore ; alors 4 mouvements supplémentaires peuvent-être ajouter au calcul de la distance de Manhattan pour prendre en compte le fait que ces pions devront se déplacer pour permettre au pion devant occuper le coin de rejoindre sa position finale.
  Cette heuristique est admissible car elle relâche le problème en considérant que la cellule dans laquelle doit se déplacer le pion qui quitte sa position finale pour laisser-passer le pion devant occupé le coin est vide.
  Considérons par exemple l'état initial de la figure 2.5 ci-dessous :

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

Figure 2.5 – Corner tile exemple

Les pions 2 et 7 sont à leurs positions finales respectives. Mais le pion 3 n'est pas encore à sa position finale. Alors 4 mouvements supplémentaires peuvent être ajouter au calcul de

la distance de Manhattan pour prendre en compte le fait que ces pions devront se déplacer pour permettre au pion 3 de rejoindre sa position finale. La distance de Manhattan de cet état initial donne :

$$h(n) = (2 + 0 + 6) + (1 + 3 + 2 + 0) + (1 + 2 + 3 + 2) + (1 + 2 + 1 + 0) = 26$$

La Corner tile donne donc :

$$h(n) = 26 + 4 = 30$$

La Corner tile permet d'ajouter jusqu'à 12 mouvements à la distance de Manhattan dans le cas du 15-puzzle [22]. Notons que dans le cas du 8-puzzle, il faut éviter de recompter des mouvements supplémentaires pour les pions adjacents à deux angles simultanément.

2.3 Patterns database

Les Patterns database consistent à exploiter une base de données préétablie des solutions pour des sous-problèmes du jeu (appelés patterns) qui doivent être résolus pour arriver à résoudre le problème initial. Les Patterns Database enregistrent les différentes solutions optimales possibles pour chaque patterns du jeu [23, 24].
Sur la figure 2.6 ci-dessous, nous présentons un exemple classique de Patterns database où deux patterns sont représentés : fringe et corner. Les pions dans les cellules rouges constituent la fringe pattern. Il s'agit des pions 3,7,11,12,13,14 et 15. Les pions de couleur bleue constituent la corner pattern. Il s'agit des pions 8,9,10,12,13,14 et 15. Le but est de résoudre toutes les permutations possibles de chaque pattern sans se soucier de la position des autres pions n'appartenant pas au pattern. Puis de stocker la solution optimale obtenue de la résolution des différents patterns dans une base de données. Ensuite lors de l'évaluation d'un quelconque noeud du jeu, on recherche dans la base de donnée préalablement construit la résolution optimale de chaque pattern et on choisit pour heuristique de ce noeud la résolution optimale ayant la plus grande valeur. Les patterns database offrent la particularité d'être à la fois admissible et offrant aussi une valeur heuristique assez proche de la réalité. Leur inconvénient majeur est l'utilisation d'une assez forte mémoire.

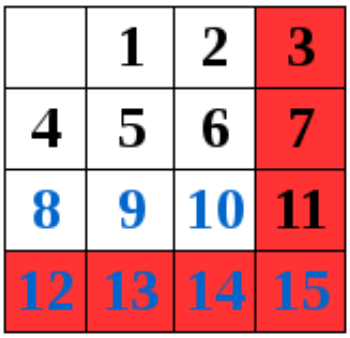


Figure 2.6 – Exemple de Pattern database fringe et corner pattern

Divers types de patterns database existent. Nous pouvons citer :

- Multi Pattern database : On parle de multi patterns database lorsque plusieurs patterns sont considérés sur le tableau de jeu. Par exemple sur la figure 2.7 ci-dessous, trois patterns sont considérés.L'un est composé des pions 1, 2, 4, 5, 8 et 9. L'autre est composé des pions 3, 6, 7, 10, 11 et 15. Enfin le troisième est composé des pions 12, 13 et 14.

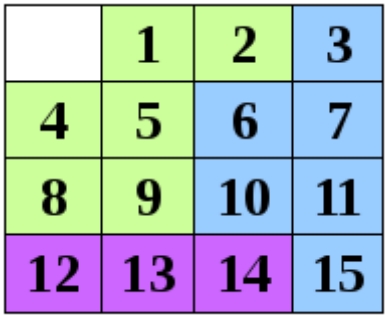


Figure 2.7 – Exemple de Multiple patterns database

- Disjoint Pattern database : On parle de disjoint patterns database lorsque les patterns choisis sont constitués de pions strictement distincts. En d'autres termes, aucun pions n'appartient simultanément à plus d'un pattern comme dans la figure 2.8 ci-dessous.

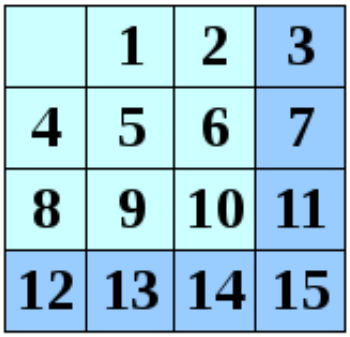


Figure 2.8 – Exemple de Disjoint pattern database

Les disjoints patterns database ont la particularité de permettre d'additionner les différentes valeurs de résolutions des patterns tout en maintenant l'admissibilité. Ainsi nous obtenons une valeur plus proche de la valeur de la solution optimale pour la résolution de tout le tableau de jeu.

Conclusion

Cette partie a traité des techniques de recherche non informées et informées. Elle a aussi abordé la formulation d'un problème comme problème de recherche et les heuristiques existantes pour résoudre le jeu de n-puzzle. Nous présentons dans la partie suivante les méthodes et outils utilisés dans la réalisation du travail.

Deuxième partie

Méthodes, Outils et Développement

Introduction

Dans cette partie, nous présentons la méthode utilisée pour réaliser le travail (le cycle semi-itératif), le choix de la méthode. Puis nous faisons une description de la machine et du langage utilisés, des algorithmes et des tests.

Chapitre 3

Méthodes et Outils

3.1 Choix de la méthode

Le choix de la méthode de travail a constitué une phase importante dans le travail effectué. Nous avons d'abord mis en comparaison des méthodes usuellement utilisées. A cet effet, nous avons étudié le modèle en cascade, le cycle en V, le cycle semi-itératif et le SCRUM.

Notre choix s'est tourné vers le cycle semi-itératif qui offre la possibilité de tester l'une après l'autre chaque fonctionnalité(heuristique) implémentée au fur et à mesure que le projet évoluait. La simplicité et la flexibilité ont aussi été prise en compte.

3.2 Présentation du cycle semi-itératif

Le cycle semi-itératif est une méthode de développement conçue par James Martin en 1991 [25]. Il constitue la base de la plupart des méthodes de développement agiles actuellement utilisées. Le cycle semi-itératif permet une souplesse dans le processus de développement. Il facilite notamment l'ajout progressif de fonctionnalités au travail existant. Il comprend les principales phases suivantes :

- L'expression des besoins : Elle consiste à recueillir les besoins et attentes vis à vis du travail à réaliser ;
- La conception : Elle se fait conformément aux besoins décrit lors de la phase précédente. Elle consiste en la traduction des besoins en fonctions techniques. De cette phase sort l'architecture du travail à réaliser. La phase suivante, théoriquement, n'est que l'implémentation de cette phase ;
- La construction : Elle constitue la phase la plus importante du processus. C'est durant cette phase que les différentes fonctions techniques décrites plus haut sont réalisées, de manière concrète. La phase de construction se présente sous la forme d'itérations courtes.

Ainsi à chaque itération une fonction est codée, testée, validée puis rajoutée au projet complet. Les itérations sont répétées jusqu'à l'obtention du travail complet qui sera exploité ;

- Les tests : Elle est faite d'un ensemble de tests sur le travail complet afin de s'assurer qu'il respecte bien les spécifications et fonctions décidées lors de l'étape de conception.
- La mise en production : C'est l'aboutissement de toutes les phases décrites plus haut. L'étape où le travail complet est prêt pour l'exploitation.

3.3 Python

Le choix d'utiliser Python comme langage de programmation a fortement été motivé par l'existence de travaux effectués en python dans le cadre de la résolution de problème par la recherche. Il s'agit des travaux de Stuart Russell, Peter Norvig disponibles dans leur livre Artificial Intelligence : A Modern Approach [9], un bestseller dans le domaine de l'Intelligence Artificielle et cité 28314 fois (statistiques tirés de google scholar au 31 Août 2016) par différents livres et articles scientifiques sur le sujet. En effet, ces auteurs ont dans leur ouvrage mis en place une bibliothèque comportant des classes écrites en Python qui fournissent des bases pour :

- la formulation du problème comme un problème de recherche ;
- la définition des états et des noeuds ;
- la définition des actions et successeurs.

Ces classes implémentent aussi quelques algorithmes de recherches, Breadth First Search, Limited Depth First Search, Iterative Deepening Search, $A^\star$, et bien d'autres encore.

Aussi, Python est un langage de programmation, créé par Guido van Rossum, dont la première version est sortie en 1991. Il est actuellement associé à une organisation à but non lucratif, la Python Software Foundation , créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques les Monty Python [26].
Python fonctionne sous différentes variantes d'Unix, Windows, Mac OS, BeOs, NextStep, et par le biais de différentes implémentations.
Avec Python, aucune compilation n'est nécessaire. De plus, Python permet de programmer objet mais ne l'impose pas : il reste possible de faire des petits scripts déstructurés [27].
Python dispose aussi de plusieurs bibliothèques, qui permettent de l'utiliser dans la quasi-totalité des domaines. Il existe des extensions pour une large variétés de besoins. Par exemple, développer des interfaces graphiques en Python (TKinter, PyGTK, PyQt4), faire du calcul scientifique (numeric, math), intégrer du code MatLab (matlab), utiliser des frameworks web (Zope, Plone), utiliser des moteurs de jeux (Pygame) et toute une myriade d'applicatifs [27, 28].
Il faut retenir que ce langage est conçu pour produire du code de qualité, lisible et maintenable.

Ceci grâce à sa syntaxe claire, cohérente et concise, et l'indentation obligatoire du code. Il permet aussi au développeur d'être rapide et plus productif car certains aspects du développement sont gérés automatiquement, comme par exemple le typage des données.

3.4 Les tests

Dans un premier temps, une base de toutes les instances solvables du 8-puzzle a été créée. Pour ce faire, on génère tout les états initiaux possibles dont on teste la solvabilité. Chaque état est représenté comme une liste contenant les numéros des pions avec la cellule vide représentée par le numéro 0. Les tests ont été effectués avec l'algorithme $A^\star$ sur l'ensemble de toutes les instances solvables du 8-puzzle, soit 181440 instances ; et avec l'algorithme $IDA^\star$ sur un ensemble de 1000 instances solvables du 15-puzzle prises aléatoirement.

3.5 Matériel de test

Le développement et les tests ont été effectués sur un ordinateur de type PC possédant les caractéristiques ci-après :

- processeur intel core i5 ;
- mémoire RAM 8 Go ;
- disque dur SATA 750 Go ;
- système d'exploitation 64 bits Debian 8.1 jessie ;

Chapitre 4

Développement

4.1 Fonctions additionnelles

Test de Solvabilité

L'algorithme ci-dessous a été implémenté pour tester la solvabilité sur un état quelconque. Il est utilisé lors de la mise en place de la base de toutes les instances solvables du 8-puzzle et de la base des 1000 instances du 15-puzzle utilisées lors des tests. Dans un premier temps, nous déterminons la parité du vide, nommé *blankParity*. Puis nous déterminons la parité du tableau de jeu, nommé *parity*. Enfin nous comparons pour déterminer si l'état pris en paramètre est solvable ou pas.

Algorithme 4.1 Reachable state

```
1 : state : état évalué
2 : liste(state) : Fonction qui formate l'état sous la forme d'une liste
3 : calculateN(liste) : Fonction qui calcule la taille n du puzzle
4 : permutation(X, Y) : Fonction qui effectue la permutation des éléments X et Y dans liste
5 : abs(X) : Fonction qui calcule la valeur absolue de la valeur X
6 : DIV : Opérateur qui effectue la division entière
7 : % : Opérateur qui calcul le %
8 : isAtGoalPosition(X) : Retourne vrai si le pion X est à sa position but et faux sinon
9 : function ReachableState(state)
10 :     liste ← liste(state)
11 :     n ← calculateN(liste)
12 :     if liste[1] != 0 then
13 :         blankPosition ← liste.index(0) // La position actuelle du pion vide dans liste
14 :         permutation(liste[1],liste[blankPosition])
15 :         parity ← 1
16 :         blankParity ← abs((blankPosition DIV n)) + abs((blankPosition % n))
```

```
17 :         blankParity ← blankParity % 2
18 :     else
19 :         blankParity ← 0
20 :         parity ← 0
21 :     end if
22 :     i ← 2
23 :     while i <= length of liste do
24 :         if ! isAtGoalPosition(liste[i]) then
25 :             elmtPosition ← liste.index(liste[i])
26 :             permutation(liste[elmtPosition],liste[i])
27 :             parity ← parity + 1
28 :         end if
29 :         i ← i + 1
30 :     end while
31 :     parity ← parity % 2
32 :     if parity == 0 and blankParity == 0 then
33 :         return True
34 :     end if
35 :     if parity != 0 and blankParity != 0 then
36 :         return True
37 :     end if
38 :     return False
39 : end function
```

$A^\star$

L'algorithme $A^\star$ ci-dessous a été implémenté pour effectuer les tests avec les heuristiques sur la base de toutes les instances solvables du 8-puzzle. Pour éviter que les états visités ne soient revisités, une autre fois, l'algorithme tient une liste des états déjà visités, nommé *closedList*.

Algorithme 4.2 $A^\star$

```
1 : function Astar(noeudRacine, h)
2 :     fringe.ajouter(noeudRacine) //  Ajouter le noeud racine à la fringe
3 :     while !empty(fringe) do
4 :         node ← fringe.retirer(min, f) // Le noeud retirer est celui ayant le plus petit f(n)
5 :         if isGoal(state(node)) then
6 :             return node
7 :         end if
8 :         if node not in closedList then
9 :             closedList ← node
10 :            fringe.ajouter(successeurs(node)) // Ajouter les successeurs du noeud à la fringe
11 :        end if
12 :    end while
```

```
13 :     return None
14 : end function
15 : function f(node, h)
16 :     // Calculer f à partir de l'heuristique h et du coût du chemin pathCost
17 :     f ← pathCost(node) + h(node)
18 :     return f
19 : end function
```

$IDA^\star$

L'algorithme $IDA^\star$ ci-dessous a été implémenté pour effectuer les tests avec les heuristiques sur la base de 1000 instances solvables du 15-puzzle. Pour éviter que les états visités ne soient revisités, une autre fois, l'algorithme tient une liste des états déjà visités, nommé *closedList*.

Algorithme 4.3 $IDA^\star$

```
1 : function IDAstar(noeudRacine, h)
2 :     S ← h(noeudRacine)
3 :     result ← None
4 :     while result == None do
5 :         nextSeuil ← Seuil
6 :         Seuil ← ""
7 :         (result, Seuil) ← recursiveFunction(problem, h, nextSeuil)
8 :         return result
9 :     end while
10 : end function
11 : function recursiveFunction(noeudRacine, h, Seuil)
12 :     fringe.ajouter(noeudRacine) // Ajouter le noeud racine à la fringe
13 :     nextSeuil ← 10000
14 :     while fringe n'est pas vide do
15 :         node ← fringe.retirer() // Retirer un noeud de la fringe
16 :         if isGoal(node) then
17 :             return (node,nextSeuil)
18 :         end if
19 :         if state(node) not in closedList then
20 :             closedList.ajouter(state(node))
21 :             f ← pathCost(node) + h(node)
22 :             if f <= Seuil then
23 :                 fringe.ajouter(successeurs(node))
24 :             else
```

```
25 :                 if f < nextSeuil then
26 :                     nextSeuil ← f
27 :                 end if
28 :             end if
29 :         end if
30 :     end while
31 :     return (None, nextSeuil)
32 : end function
```

4.2 Heuristiques

Pour effectuer les tests, différentes heuristiques décrites dans la partie précédente ont été développées. Voici ci-dessous les algorithmes de ces heuristiques.

Distance de Manhattan

Algorithme 4.4 Distance de Manhattan

```
 1 : INPUT :
 2 : node : noeud évalué
 3 : function ManhattanDistance(node)
 4 :     state ← state(node)
 5 :     liste ← liste(state)
 6 :     n ← calculateN(liste)
 7 :     h ← 0
 8 :     for i ← 1 to length of liste do
 9 :         if liste[i] != 0 and ! isAtGoalPosition(liste[i]) then
10 :             addrInGoal ← address of liste[i] in goal
11 :             h ← h + (abs((i DIV n - addrInGoal DIV n))
12 :              + abs((i % n - addrInGoal % n)))
13 :         end if
14 :     end for
15 :     return h
16 : end function
```

Corners Tiles

L'algorithme 4.5 implémente le corner tile comme décrit dans la partie précédente.

Algorithme 4.5 Corners Tiles

```
1 : INPUT :
2 : node : noeud évalué
3 : function CornersTiles(node)
4 :     state ← state(node)
```

```
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     // Coin supérieur droit / Upper right corner
9 :     if (liste[n - 1] != 0) and (! isAtGoalPosition(liste[n - 1])) then
10 :        if (isAtGoalPosition(liste[n - 2])) and (isAtGoalPosition(liste[2 * n - 1])) then
11 :            h ← h + 4
12 :        end if
13 :    end if
14 :    // Coin inférieur droit / lower right corner
15 :    if (liste[n * n - 1] != 0) and (! isAtGoalPosition(liste[n * n - 1])) then
16 :        if  (isAtGoalPosition(liste[(n - 1) * n - 1])) and (isAtGoalPosition(liste[n * n - 2]))
    then
17 :            h ← h + 4
18 :        end if
19 :    end if
20 :    // Coin inférieur gauche / lower left corner
21 :    if (liste[(n - 1) * n] != 0) and (! isAtGoalPosition(liste[(n - 1) * n])) then
22 :        if (isAtGoalPosition(liste[(n - 2) * n])) and (isAtGoalPosition(liste[(n - 1) * n + 1]))
    then
23 :            h ← h + 4
24 :        end if
25 :    end if
26 :    return h
27 : end function
```

Corners Tiles pour 8-puzzle

L'algorithme 4.6 présente une implémentation de corner tile spécifique au 8-puzzle. La particularité de cet algorithme est qu'il tient compte du fait que deux coins du tableau de jeu partagent le même pion dans le cas du 8-puzzle. Ce contrôle se fait avec la fonction $isInvolvedInOtherCorner()$ utilisée ci-dessous.

Algorithme 4.6 Corners Tiles pour 8-puzzle

```
1 : INPUT :
2 : node : noeud évalué
3 : function CornersTiles8(node)
4 :     state ← state(node)
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     // Coin supérieur droit / Upper right corner
9 :     if (liste[n - 1] != 0) and (! isAtGoalPosition(liste[n - 1])) then
```

```
10 :         if (isAtGoalPosition(liste[n - 2])) and (isAtGoalPosition(liste[2 * n - 1])) then
11 :             h ← h + 4
12 :             isInvolvedInOtherCorner(liste[n ∗ n − 2]) ← True
13 :         end if
14 :     end if
15 :     // Coin inférieur droit / lower right corner
16 :     if (liste[n * n - 1] != 0) and (! isAtGoalPosition(liste[n * n - 1])) then
17 :         if   (isAtGoalPosition(liste[(n - 1) * n - 1])) and (isAtGoalPosition(liste[n * n - 2]))
     then
18 :             if isInvolvedInOtherCorner(liste[n * n - 2]) == False then
19 :                 h ← h + 4
20 :                 isInvolvedInOtherCorner(liste[(n − 1) ∗ n + 1]) ← True
21 :             end if
22 :         end if
23 :     end if
24 :     // Coin inférieur gauche / lower left corner
25 :     if (liste[(n - 1) * n] != 0) and (! isAtGoalPosition(liste[(n - 1) * n])) then
26 :         if (isAtGoalPosition(liste[(n - 2) * n])) and (isAtGoalPosition(liste[(n - 1) * n + 1]))
     then
27 :             if (isInvolvedInOtherCorner(liste[(n - 1) * n + 1]) == False) then
28 :                 h ← h + 4
29 :             end if
30 :         end if
31 :     end if
32 :     return h
33 : end function
```

Last Move

L'algorithme 4.7 implémente last move comme décrit dans la partie précédente.

Algorithme 4.7 Last Move

```
1 : INPUT :
2 : node : noeud évalué
3 : function LastMove(node)
4 :     state ← state(node)
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     if (columnNumber(1) != 0) and (lineNumber(n) != 0) then
9 :         h ← h + 2
10 :    end if
11 :    return h
12 : end function
```

Linear conflict

L'algorithme 4.8 implémente linear conflict comme décrit plus haut dans la partie précédente. Pour ce faire, nous traitons d'abord les conflits horizontaux, en rapport aux lignes du tableau de jeu ; puis les conflits verticaux, en rapport aux colonnes du tableau de jeu.

Algorithme 4.8 Linear conflict

```
1 : INPUT :
2 : node : noeud évalué
3 : function LinearConflict(node)
4 :     state ← state(node)
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     // Horizontal conflict
9 :     line ← 1
10 :    while line <= length of liste do
11 :        tmpMax ← −1
12 :        column ← 1
13 :        while column <= n do
14 :            i ← column + line
15 :            if (liste[i] != 0) and (isAtGoalLine(liste[i]) then
16 :                if liste[i] > tmpMax then
17 :                    tmpMax ← liste[i]
18 :                else
```

```
19 :                        h ← h + 2
20 :                        tmpMax ← liste[i]
21 :                    end if
22 :                end if
23 :                column ← column + 1
24 :            end while
25 :            line ← line + n
26 :        end while
27 :        // Vertical conflict
28 :        column ← 1
29 :        while column <= n do
30 :            tmpMax ← −1
31 :            line ← 1
32 :            while line <= length of liste do
33 :                i ← column + line
34 :                if (liste[i] != 0) and (isAtGoalColumn(liste[i]) then
35 :                    if liste[i] > tmpMax then
36 :                        tmpMax ← liste[i]
37 :                    else
38 :                        h ← h + 2
39 :                        tmpMax ← liste[i]
40 :                    end if
41 :                end if
42 :                line ← line + n
43 :            end while
44 :            column ← column + 1
45 :        end while
46 :        return h
47 : end function
```

Diagonal Conflict non admissible

L'algorithme 4.9 implémente la version non admissible de la diagonal conflict, une heuristique que nous proposons pour tenter d'améliorer les heuristiques existantes. Elle est décrite dans le chapitre suivant.

Algorithme 4.9 Diagonal Conflict non admissible

```
1 : INPUT :
2 : node : noeud évalué
3 : function DiagonalConflictNonAd(node)
4 :     state ← state(node)
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     i ← 1
9 :     while i <= length of liste do
10 :        Gi ← goal[i] // l'élément supposé être à la position i dans le but
11 :        if (liste[i] != 0) and (Gi != 0) and (! isAtGoalPosition(liste[i]) then
12 :            // Coin supérieur gauche de la position actuelle
13 :            if (i DIV n) > 0 and (i % n) > 0 then
14 :                if liste[i - n - 1] == Gi then
15 :                    if (liste[i - n] != 0) and (isAtGoalPosition(liste[i - n])) then
16 :                        h ← h + 2
17 :                    else
18 :                        if (liste[i - 1] != 0) and (isAtGoalPosition(liste[i - 1])) then
19 :                            h ← h + 2
20 :                        end if
21 :                    end if
22 :                end if
23 :                // Coin supérieur droit de la position actuelle
24 :            else
25 :                if (i DIV n) > 0 and (i % n) < (n - 1) then
26 :                    if liste[i - n + 1] == Gi then
27 :                        if (liste[i + 1] != 0) and (isAtGoalPosition(liste[i + 1])) then
28 :                            h ← h + 2
29 :                        else
30 :                            if (liste[i - n] != 0) and (isAtGoalPosition(liste[i - n])) then
31 :                                h ← h + 2
32 :                            end if
33 :                        end if
34 :                    end if
35 :                    // Coin inferieur droit de la position actuelle
36 :                else
37 :                    if (i DIV n) < (n - 1) and (i % n) < (n - 1) then
```

```
38 :                         if liste[i + n + 1] == Gi then
39 :                             if (liste[i + n] != 0) and (isAtGoalPosition(liste[i + n])) then
40 :                                 h ← h + 2
41 :                             else
42 :                                 if (liste[i + 1] != 0) and (isAtGoalPosition(liste[i + 1])) then
43 :                                     h ← h + 2
44 :                                 end if
45 :                             end if
46 :                         end if
47 :                         // Coin inférieur gauche de la position actuelle
48 :                     else
49 :                         if (i DIV n) < (n - 1) and (i % n) > 0 then
50 :                             if liste[i + n - 1] == Gi then
51 :                                 if (liste[i - 1] != 0) and (isAtGoalPosition(liste[i - 1])) then
52 :                                     h ← h + 2
53 :                                 else
54 :                                     if     (liste[i + n] != 0) and (isAtGoalPosition(liste[i + n]))
    then
55 :                                         h ← h + 2
56 :                                 end if
57 :                             end if
58 :                         end if
59 :                     end if
60 :                 end if
61 :             end if
62 :         end if
63 :         end if
64 :         i ← i + 1
65 :     end while
66 :     return h
67 : end function
```

Diagonal Conflict admissible

L'algorithme 4.10 implémente la version admissible de la diagonal conflict, une heuristique que nous proposons pour tenter d'améliorer les heuristiques existantes. Elle est décrite dans le chapitre suivant.

Algorithme 4.10 Diagonal Conflict admissible

```
1 : INPUT :
2 : node : noeud évalué
3 : function DiagonalConflictAd(node)
4 :     state ← state(node)
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     i ← 1
9 :     while i <= length of liste do
10 :        Gi ← goal[i] // l'élément supposé être à la position i dans le but
11 :        if (liste[i] != 0) and (Gi != 0) and (! isAtGoalPosition(liste[i]) then
12 :            // Coin supérieur gauche de la position actuelle
13 :            if (i DIV n) > 0 and (i % n) > 0 then
14 :                if liste[i - n - 1] == Gi then
15 :                    if (liste[i - n] != 0) and (isAtGoalPosition(liste[i - n])) then
16 :                        if (liste[i - 1] != 0) and (isAtGoalPosition(liste[i - 1])) then
17 :                            h ← h + 2
18 :                        end if
19 :                    end if
20 :                end if
21 :                // Coin supérieur droit de la position actuelle
22 :            else
23 :                if (i DIV n) > 0 and (i % n) < (n - 1) then
24 :                    if liste[i - n + 1] == Gi then
25 :                        if (liste[i + 1] != 0) and (isAtGoalPosition(liste[i + 1])) then
26 :                            if (liste[i - n] != 0) and (isAtGoalPosition(liste[i - n])) then
27 :                                h ← h + 2
28 :                            end if
29 :                        end if
30 :                    end if
31 :                    // Coin inferieur droit de la position actuelle
32 :                else
33 :                    if (i DIV n) < (n - 1) and (i % n) < (n - 1) then
34 :                        if liste[i + n + 1] == Gi then
35 :                            if (liste[i + n] != 0) and (isAtGoalPosition(liste[i + n])) then
36 :                                if (liste[i + 1] != 0) and (isAtGoalPosition(liste[i + 1])) then
```

```
37 :                                    h ← h + 2
38 :                                end if
39 :                            end if
40 :                        end if
41 :                        // Coin inférieur gauche de la position actuelle
42 :                    else
43 :                        if (i DIV n) < (n - 1) and (i % n) > 0 then
44 :                            if liste[i + n - 1] == Gi then
45 :                                if (liste[i - 1] != 0) and (isAtGoalPosition(liste[i - 1])) then
46 :                                    if     (liste[i + n] != 0) and (isAtGoalPosition(liste[i + n]))
    then
47 :                                        h ← h + 2
48 :                                    end if
49 :                                end if
50 :                            end if
51 :                        end if
52 :                    end if
53 :                end if
54 :            end if
55 :        end if
56 :        i ← i + 1
57 :    end while
58 :    return h
59 : end function
```

Diagonal Conflict étendue

L'algorithme 4.11 implémente une autre version admissible de la diagonal conflict, nommée diagonal conflict étendue, une heuristique que nous proposons pour tenter d'améliorer les heuristiques existantes. Elle est décrite dans le chapitre suivant.

Algorithme 4.11 Diagonal Conflict étendue

```
1 : INPUT :
2 : node : noeud évalué
3 : function DiagonalConflictEtd(node)
4 :     state ← state(node)
5 :     liste ← liste(state)
6 :     n ← calculateN(liste)
7 :     h ← ManhattanDistance(node)
8 :     i ← 1
9 :     while i <= length of liste do
10 :        Gi ← goal[i] // l'élément supposé être à la position i dans le but
11 :        if (liste[i] != 0) and (Gi != 0) and (! isAtGoalPosition(liste[i]) then
```

```
12 :            indexGi ← liste.index(Gi) // index de Gi dans liste
13 :            // Coin supérieur gauche de la position actuelle
14 :            if (i DIV n) > (indexGi DIV n) and (i % n) > (indexGi % n) then
15 :                if (liste[i - n] != 0) and (isAtGoalPosition(liste[i - n])) then
16 :                    if (liste[i - 1] != 0) and (isAtGoalPosition(liste[i - 1])) then
17 :                        h ← h + 2
18 :                    end if
19 :                end if
20 :                // Coin supérieur droit de la position actuelle
21 :            else
22 :                if (i DIV n) > (indexGi DIV n) and (i % n) < (indexGi % n) then
23 :                    if (liste[i + 1] != 0) and (isAtGoalPosition(liste[i + 1])) then
24 :                        if (liste[i - n] != 0) and (isAtGoalPosition(liste[i - n])) then
25 :                            h ← h + 2
26 :                        end if
27 :                    end if
28 :                    // Coin inferieur droit de la position actuelle
29 :                else
30 :                    if (i DIV n) < (indexGi DIV n) and (i % n) < (indexGi % n) then
31 :                        if (liste[i + n] != 0) and (isAtGoalPosition(liste[i + n])) then
32 :                            if (liste[i + 1] != 0) and (isAtGoalPosition(liste[i + 1])) then
33 :                                h ← h + 2
34 :                            end if
35 :                        end if
36 :                        // Coin inférieur gauche de la position actuelle
37 :                    else
38 :                        if (i DIV n) < (indexGi DIV n) and (i % n) > (indexGi % n) then
39 :                            if (liste[i - 1] != 0) and (isAtGoalPosition(liste[i - 1])) then
40 :                                if (liste[i + n] != 0) and (isAtGoalPosition(liste[i + n])) then
41 :                                    h ← h + 2
42 :                                end if
43 :                            end if
44 :                        end if
45 :                    end if
46 :                end if
47 :            end if
48 :        end if
49 :        i ← i + 1
50 :    end while
51 :    return h
52 : end function
```

Conclusion

Cette partie a présenté le cycle semi-itératif, les outils utilisés, les algorithmes développés et les caractéristiques de la machine de travail. Elle a aussi expliqué le choix du langage python comme langage de programmation et présenté quelques algorithmes des fonction heuristiques développées. Dans la partie suivante, nous présentons les résultats de l'implémentation.

Troisième partie

Résultats et Discussion

Introduction

Dans cette partie, nous décrivons les propositions de nouvelles heuristiques inspirées de l'étude des heuristiques existantes. Puis nous présentons les résultats des évaluations effectuées à l'issue de leur implémentation.

Chapitre 5

Diagonal Conflict et 8-puzzle

L'idée qui a motivé la diagonal conflict est d'apporter un peu plus d'informations sur les interactions entre les différents pions du jeu. Cette nouvelle heuristique met en avant une forme de conflit entre les pions, jusque là inexploitée.

En effet, la diagonal conflict consiste en une amélioration de la distance de Manhattan. Elle s'applique lorsqu'un pion, x par exemple dont la cellule de la position finale est c, remplit les conditions ci-après :

- x se retrouve dans une des cellules : $c-t-1$, $c-t+1$, $c+t+1$, $c+t-1$; avec t la taille du n-puzzle. Il s'agit des cellules situées sur les diagonales et voisines à la position finale du pion x ;
- l'un des deux pions adjacents à la cellule où se retrouve le pion x est à sa position finale.

En effet, la diagonal conflict prend en compte le fait que le pion adjacent qui est à sa position finale se déplace de sa position finale pour laisser-passer le pion x avant d'y revenir. Ceci permet d'ajouter deux déplacements supplémentaires à la Manhattan Distance afin de prendre en compte ces déplacements.

$$h(n) = MD(n) + \Sigma[(isCp(c-1) \vee isCp(c-t) \wedge (gP(c) = aP(c-t-1)))$$

$$\vee(isCp(c-t) \vee isCp(c+1) \wedge (gP(c) = aP(c-t+1)))$$

$$\vee(isCp(c+1) \vee isCp(c+t) \wedge (gP(c) = aP(c+t+1)))$$

$$\vee(isCp(c+t) \vee isCp(c-1) \wedge (gP(c) = aP(c+t-1)))]$$

avec n le noeud évalué ; $MD(n)$ la distance de Manhattan du noeud évalué ; t la taille du n-puzzle ; c la position de la cellule courante évaluée ; $isCp(z)$ une fonction qui retourne 1 si le pion dans la cellule z est bien dans sa position but et 0 sinon ; $gP(c)$ qui retourne le pion dont la position but est la cellule c ; $aP(z)$ qui retourne le pion actuellement situé dans la cellule z. Considérons l'exemple de la figure 5.1 ci-dessous.

3	**1**	**5**
6	**2**	**4**
7	**8**	

Figure 5.1 – Exemple de diagonal conflict (non admissible) pour le 8-puzzle

Dans cet exemple, le pion 1 est déjà à sa position finale. Mais le pion 2 quant à lui se retrouve dans une cellule voisine à sa position finale, située sur l'une des diagonales à laquelle appartient sa position finale. Toutes les conditions de la diagonal conflict étant remplient, nous pouvons ajouter deux déplacements supplémentaires à la distances de Manhattan.

Cette nouvelle heuristique n'est pas admissible car en reprenant l'exemple de la figure 5.1 ci-dessus, le pion 2 n'est pas contraint de passer par la cellule du pion 1 pour rejoindre sa position finale. Il pourrait tout simplement passer la cellule où se trouve actuellement le pion 4. Ceci éviterait d'ajouter les deux déplacements supplémentaires.
Cependant, bien que n'étant pas admissible, on observe à l'issu des tests réalisés que la diagonal conflict trouve le chemin optimale sur 180781 instances parmi toutes les instances solvables du 8-puzzle. Soit seulement 659 instances pour lesquelles il manque de trouver le chemin optimal.

Pour y remédier, nous avons donc proposé une version améliorée, et surtout admissible de la diagonal conflict. Dans cette version, la diagonal conflict s'applique suivant les conditions ci-après :

- Se retrouve dans une des cellules voisines à sa position finale, située sur l'une des diagonales à laquelle appartient la cellule de sa position finale ;
- Les deux pions adjacents à la cellule où se retrouve le pion x sont à leur position finale respective.

En effet, cette version de la diagonal conflict prend en compte le fait que l'un des pions adjacents à la cellule où se trouve le pion x devra forcément se déplacer de sa position finale pour laisser-passer le pion x avant d'y revenir. Ceci permet donc d'ajouter deux déplacements supplémentaires à la Manhattan Distance afin de prendre en compte ces déplacements et ceci en maintenant l'admissibilité.

$$h(n) = MD(n) + \Sigma[(isCp(c-1) \wedge isCp(c-t) \wedge (gP(c) = aP(c-t-1)))$$

$$\vee(isCp(c-t) \wedge isCp(c+1) \wedge (gP(c) = aP(c-t+1)))$$

$$\vee(isCp(c+1) \wedge isCp(c+t) \wedge (gP(c) = aP(c+t+1)))$$

$$\vee(isCp(c+t) \wedge isCp(c-1) \wedge (gP(c) = aP(c+t-1)))]$$

avec n le noeud évalué ; $MD(n)$ la distance de Manhattan du noeud évalué ; t la taille du n-puzzle ; c la position de la cellule courante évaluée ; $isCp(z)$ une fonction qui retourne 1 si le

pion dans la cellule z est bien dans sa position but et 0 sinon ; $gP(c)$ qui retourne le pion dont la position but est la cellule c ; $aP(z)$ qui retourne le pion actuellement situé dans la cellule z. Considérons l'exemple de la figure 5.2 ci-dessous.

1	8	7
5	4	2
6	3	

Figure 5.2 – Exemple de diagonal conflict (admissible) pour le 8-puzzle

Nous observons que les pions 4 et 6 sont déjà à leur position finale respectives. Mais le pion 3 quant à lui se retrouve dans une cellule voisine à sa position finale, située sur l'une des diagonales à laquelle appartient sa position finale. Toutes les conditions de la diagonal conflict admissible étant remplies, nous pouvons ajouter deux déplacements supplémentaires à la Distances de Manhattan.
Cette version de la diagonal conflict offre des résultats optimales sur toutes les instances solvables du 8-puzzle.

Nous essayons d'améliorer cette version de la diagonal conflict en considérant qu'il n'est pas strictement nécessaire que le pion x se retrouve dans une des cellules : $c-t-1$, $c-t+1$, $c+t+1$, $c+t-1$; avec t la taille du n-puzzle et c la position finale de x. Ainsi, elle pourrait aussi s'appliquer lorsque le pion x se retrouve dans une cellule positionnée comme décrite dans l'exemple de la figure 5.3 ci-après :

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

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

Figure 5.3 – Exemple de diagonal conflict étendue

Dans l'exemple ci-dessus, on évalue la cellule contenant le pion 3. Cette cellule constitue la position finale du pion 10 comme le montre l'état but représenté à droite. Cette version s'applique lorsque :

- les pions 9 et 6 sont à leur position finale et le pion 10 se retrouve dans l'une des cellules contenant actuellement les pions 10,8,1 et 4 ;
- les pions 6 et 12 sont à leur position finale et le pion 10 se retrouve dans l'une des cellules contenant actuellement les pions 5 et 13 ;

- les pions 12 et 2 sont à leur position finale et le pion 10 se retrouve dans la cellule contenant actuellement le vide ;
- les pions 2 et 9 sont à leur position finale et le pion 10 se retrouve dans l'une des cellules contenant actuellement les pions 15 et 7.

5.1 Diagonal conflict comparée à distance de Manhattan

Les résultats des tests révèlent que pour :

- 89, 59% des instances (soit 162550 instances) la diagonal conflict non admissible explore moins de noeuds que la distance de Manhattan ;
- 9, 55% des instances (soit 17322 instances) la diagonal conflict non admissible explore plus de noeuds que la distance de Manhattan ;
- 0, 86% des instances (soit 1568 instances) la diagonal conflict non admissible explore le même nombre de noeuds que la distance de Manhattan.

Nous observons aussi que pour :

- 92, 16% des instances (soit 167227 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de la distance de Manhattan ;
- 7, 83% des instances (soit 14200 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de la distance de Manhattan ;
- 0, 01% des instances (soit 13 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de la distance de Manhattan.

Dans le cas de la diagonal conflict admissible, les résultats des tests montrent que pour :

- 91, 54% des instances (soit 166093 instances) la diagonal conflict admissible explore moins de noeuds que la distance de Manhattan ;
- 0, 90% des instances (soit 1623 instances) la diagonal conflict admissible explore plus de noeuds que la distance de Manhattan ;
- 7, 56% des instances (soit 13724 instances) la diagonal conflict admissible explore le même nombre de noeuds que la distance de Manhattan.

Nous observons aussi que pour :

- 9, 20% des instances (soit 16686 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de la distance de Manhattan ;
- 90, 79% des instances (soit 164735 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de la distance de Manhattan ;

- $0,01\%$ des instances (soit 21 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de la distance de Manhattan.

Dans le cas de la diagonal conflict étendue, les résultats des tests montrent que pour :

- $93,54\%$ des instances (soit 169722 instances) la diagonal conflict étendue explore moins de noeuds que la distance de Manhattan ;
- $3,80\%$ des instances (soit 6888 instances) la diagonal conflict étendue explore plus de noeuds que la distance de Manhattan ;
- $2,66\%$ des instances (soit 4830 instances) la diagonal conflict étendue explore le même nombre de noeuds que la distance de Manhattan.

Nous observons aussi que pour :

- $32,69\%$ des instances (soit 59314 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de la distance de Manhattan ;
- $67,30\%$ des instances (soit 122102 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de la distance de Manhattan ;
- $0,01\%$ des instances (soit 25 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de la distance de Manhattan.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à la distance de Manhattan.

Table 5.1 – Tableau récapitulatif de diagonal conflict comparée à distance de Manhattan

-	distance de Manhattan	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)89,59\%$ $(+)9,55\%$ $(=)0.86\%$	$(-)92,16\%$ $(+)7,83\%$ $(=)0,01\%$
Diagonal conflict admissible	$(-)91,54\%$ $(+)0,90\%$ $(=)7.56\%$	$(-)9,20\%$ $(+)90,79\%$ $(=)0,01\%$
Diagonal conflict étendue	$(-)93,54\%$ $(+)3,80\%$ $(=)2,66\%$	$(-)32,69\%$ $(+)67,30\%$ $(=)0,01\%$

Ces résultats permettent de conclure que toutes les versions de la diagonal conflict améliorent considérablement la distance de Manhattan en terme de nombre de noeuds explorés. Cependant cette amélioration s'accompagne d'une augmentation du temps d'exécution dans le cas de la diagonal conflict admissible et de la diagonal conflict étendue.

5.2 Diagonal conflict comparée à linear conflict

Les résultats des tests par rapport à linear conflict révèlent que pour :

- $19,67\%$ des instances (soit 35692 instances) la diagonal conflict non admissible explore moins de noeuds que linear conflict ;
- $79,87\%$ des instances (soit 144913 instances) la diagonal conflict non admissible explore plus de noeuds que linear conflict ;
- $0,46\%$ des instances (soit 835 instances) la diagonal conflict non admissible explore le même nombre de noeuds que linear conflict.

Nous observons aussi que pour :

- $47,74\%$ des instances (soit 86615 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de linear conflict ;
- $52,25\%$ des instances (soit 94804 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de linear conflict ;
- $0,01\%$ des instances (soit 22 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de linear conflict.

Dans le cas de la diagonal conflict admissible, les résultats des tests par rapport à linear conflict montrent que pour :

- $24,14\%$ des instances (soit 43795 instances) la diagonal conflict admissible explore moins de noeuds que linear conflict ;
- $75,19\%$ des instances (soit 136423 instances) la diagonal conflict admissible explore plus de noeuds que linear conflict ;
- $0,67\%$ des instances (soit 1222 instances) la diagonal conflict admissible explore le même nombre de noeuds que linear conflict.

Nous observons aussi que pour :

- $4,8\%$ des instances (soit 8711 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de linear conflict ;

- 95, 19% des instances (soit 172720 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de linear conflict ;
- 0, 01% des instances (soit 11 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de linear conflict.

Dans le cas de la diagonal conflict étendue, les résultats des tests par rapport à linear conflict montrent que pour :

- 21, 59% des instances (soit 39181 instances) la diagonal conflict étendue explore moins de noeuds que linear conflict ;
- 77, 74% des instances (soit 141051 instances) la diagonal conflict étendue explore plus de noeuds que linear conflict ;
- 0, 67% des instances (soit 1208 instances) la diagonal conflict étendue explore le même nombre de noeuds que linear conflict.

Nous observons aussi que pour :

- 9, 53% des instances (soit 17294 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de linear conflict ;
- 90, 46% des instances (soit 164125 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de linear conflict ;
- 0, 01% des instances (soit 22 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de linear conflict.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à linear conflict.

Table 5.2 – Tableau récapitulatif de diagonal conflict comparée à linear conflict

-	linear conflict	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)19,67\%$ $(+)79,87$ $(=)0,46\%$	$(-)47,74\%$ $(+)52,25\%$ $(=)0,01\%$
Diagonal conflict admissible	$(-)24,14\%$ $(+)75,19\%$ $(=)0,67\%$	$(-)4,80\%$ $(+)95,19\%$ $(=)0,01\%$
Diagonal conflict étendue	$(-)21,59\%$ $(+)77,74\%$ $(=)0,67\%$	$(-)9,53\%$ $(+)90,46\%$ $(=)0,01\%$

Ces résultats amènent à conclure que linear conflict offre des résultats nettement meilleurs que toutes les versions de la diagonal conflit.

5.3 Diagonal conflict comparée à corners tiles

Les résultats des tests par rapport à corners tiles révèlent que pour :

- $17,93\%$ des instances (soit 32525 instances) la diagonal conflict non admissible explore moins de noeuds que corners tiles ;
- $81,52\%$ des instances (soit 147926 instances) la diagonal conflict non admissible explore plus de noeuds que corners tiles ;
- $0,55\%$ des instances (soit 989 instances) la diagonal conflict non admissible explore le même nombre de noeuds que corners tiles.

Nous observons aussi que pour :

- $45,23\%$ des instances (soit 82058 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de corners tiles ;
- $54,76\%$ des instances (soit 99362 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de corners tiles ;
- $0,01\%$ des instances (soit 21 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de corners tiles.

Dans le cas de la diagonal conflict admissible, les résultats des tests par rapport à corners tiles montrent que pour :

- $18,17\%$ des instances (soit 32976 instances) la diagonal conflict admissible explore moins de noeuds que corners tiles ;
- $81,24\%$ des instances (soit 147401 instances) la diagonal conflict admissible explore plus de noeuds que corners tiles ;
- $0,59\%$ des instances (soit 1063 instances) la diagonal conflict admissible explore le même nombre de noeuds que corners tiles.

Nous observons aussi que pour :

- $3,49\%$ des instances (soit 6341 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de corners tiles ;
- $96,51\%$ des instances (soit 175094 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de corners tiles ;
- $0,00\%$ des instances (soit 7 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de corners tiles.

Dans le cas de la diagonal conflict étendue, les résultats des tests par rapport à corners tiles montrent que pour :

- $15,82\%$ des instances (soit 28695 instances) la diagonal conflict étendue explore moins de noeuds que corners tiles ;
- $83,60\%$ des instances (soit 151694 instances) la diagonal conflict étendue explore plus de noeuds que corners tiles ;
- $0,58\%$ des instances (soit 1051 instances) la diagonal conflict étendue explore le même nombre de noeuds que corners tiles.

Nous observons aussi que pour :

- $9,6\%$ des instances (soit 17410 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de corners tiles ;
- $90,39\%$ des instances (soit 164014 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de corners tiles ;
- $0,01\%$ des instances (soit 18 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de corners tiles.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à corners tiles.

Table 5.3 – Tableau récapitulatif de diagonal conflict comparée à corner tile

-	corners tiles	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)17,93\%$ $(+)81,52\%$ $(=)0,55\%$	$(-)45,23\%$ $(+)54,76\%$ $(=)0,01\%$
Diagonal conflict admissible	$(-)18,17\%$ $(+)81,24\%$ $(=)0,59\%$	$(-)3,49\%$ $(+)96,51\%$ $(=)0,00\%$
Diagonal conflict étendue	$(-)15,82\%$ $(+)83,60\%$ $(=)0,58\%$	$(-)9,60\%$ $(+)90,39\%$ $(=)0,01\%$

Ces résultats amènent à conclure que corner tiles offre des résultats nettement meilleurs que toutes les versions de la diagonal conflit.

5.4 Diagonal conflict comparée à last move

Les résultats des tests par rapport à last move révèlent que pour :

- 78, 79% des instances (soit 142949 instances) la diagonal conflict non admissible explore moins de noeuds que last move ;
- 21, 15% des instances (soit 38385 instances) la diagonal conflict non admissible explore plus de noeuds que last move ;
- 0, 06% des instances (soit 106 instances) la diagonal conflict non admissible explore le même nombre de noeuds que last move.

Nous observons aussi que pour :

- 95, 03% des instances (soit 172423 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de last move ;
- 4, 96% des instances (soit 9000 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de last move ;
- 0, 01% des instances (soit 19 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de last move.

Dans le cas de la diagonal conflict admissible, les résultats des tests par rapport à last move montrent que pour :

- 72, 58% des instances (soit 131690 instances) la diagonal conflict admissible explore moins de noeuds que last move ;
- 27, 30% des instances (soit 49531 instances) la diagonal conflict admissible explore plus de noeuds que last move ;
- 0, 12% des instances (soit 219 instances) la diagonal conflict admissible explore le même nombre de noeuds que last move.

Nous observons aussi que pour :

- 63, 64% des instances (soit 115466 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de last move ;
- 36, 34% des instances (soit 65947 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de last move ;
- 0, 02% des instances (soit 31 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de last move.

Dans le cas de la diagonal conflict étendue, les résultats des tests par rapport à last move montrent que pour :

- $77,10\%$ des instances (soit 139896 instances) la diagonal conflict étendue explore moins de noeuds que last move ;
- $22,82\%$ des instances (soit 41406 instances) la diagonal conflict étendue explore plus de noeuds que last move ;
- $0,08\%$ des instances (soit 138 instances) la diagonal conflict étendue explore le même nombre de noeuds que last move.

Nous observons aussi que pour :

- $67,05\%$ des instances (soit 121657 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de last move ;
- $32,94\%$ des instances (soit 59765 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de last move ;
- $0,01\%$ des instances (soit 20 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de last move.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à last move.

Table 5.4 – Tableau récapitulatif de diagonal conflict comparée à last move

-	last move	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)78,79\%$ $(+)21,15\%$ $(=)0,06\%$	$(-)95,03\%$ $(+)4,96\%$ $(=)0,01\%$
Diagonal conflict admissible	$(-)72,58\%$ $(+)27,30\%$ $(=)0,12\%$	$(-)63,64\%$ $(+)36,34\%$ $(=)0,02\%$
Diagonal conflict étendue	$(-)77,10\%$ $(+)22,82\%$ $(=)0,08\%$	$(-)67,05\%$ $(+)32,94\%$ $(=)0,01\%$

Ces résultats permettent de conclure que toutes les versions de la diagonal conflict offrent des résultats nettement meilleurs que last move.

Chapitre 6

Diagonal Conflict et 15-puzzle

Observons à présent les performances des différentes versions de la diagonal conflict sur une plus grande instance du n-puzzle. Pour ce faire, les tests ont été effectués sur 1000 instances aléatoires du 15-puzzle. A l'issue de ces tests, la diagonal conflict non admissible ne trouve pas le chemin optimal pour 21 instances sur les 1000 instances utilisées pour les tests.

6.1 Diagonal conflict comparée à distance de Manhattan

Les résultats des performances des différentes versions de diagonal conflict par rapport à la distance de Manhattan se présentent comme suit. Pour :

- $45,20\%$ des instances (soit 452 instances) la diagonal conflict non admissible explore moins de noeuds que la distance de Manhattan ;
- $10,50\%$ des instances (soit 105 instances) la diagonal conflict non admissible explore plus de noeuds que la distance de Manhattan ;
- $44,30\%$ des instances (soit 443 instances) la diagonal conflict non admissible explore le même nombre de noeuds que la distance de Manhattan.

Les résultats des tests révèlent aussi que pour :

- $40,00\%$ des instances (soit 400 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de la distance de Manhattan ;
- $59,70\%$ des instances (soit 597 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de la distance de Manhattan ;
- $0,30\%$ des instances (soit 3 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de la distance de Manhattan.

Dans le cas de la diagonal conflict admissible, les résultats des tests montrent que pour :

- $13,10\%$ des instances (soit 131 instances) la diagonal conflict admissible explore moins de noeuds que la distance de Manhattan ;
- $10,00\%$ des instances (soit 10 instances) la diagonal conflict admissible explore plus de noeuds que la distance de Manhattan ;
- $85,90\%$ des instances (soit 859 instances) la diagonal conflict admissible explore le même nombre de noeuds que la distance de Manhattan.

Nous observons aussi que pour :

- $14,60\%$ des instances (soit 146 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de la distance de Manhattan ;
- $85,40\%$ des instances (soit 854 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de la distance de Manhattan ;
- $0,00\%$ des instances (soit 0 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de la distance de Manhattan.

Dans le cas de la diagonal conflict étendue, les résultats des tests montrent que pour :

- $27,60\%$ des instances (soit 276 instances) la diagonal conflict étendue explore moins de noeuds que la distance de Manhattan ;
- $2,90\%$ des instances (soit 29 instances) la diagonal conflict étendue explore plus de noeuds que la distance de Manhattan ;
- $69,50\%$ des instances (soit 695 instances) la diagonal conflict étendue explore le même nombre de noeuds que la distance de Manhattan.

Nous observons aussi que pour :

- $15,40\%$ des instances (soit 154 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de la distance de Manhattan ;
- $84,40\%$ des instances (soit 844 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de la distance de Manhattan ;
- $0,20\%$ des instances (soit 2 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de la distance de Manhattan.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à la distance de Manhattan.

Table 6.1 – Tableau récapitulatif de diagonal conflict comparée à distance de Manhattan

-	distance de Manhattan	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)45,20\%$ $(+)10,50\%$ $(=)44,30\%$	$(-)40,00\%$ $(+)59,70\%$ $(=)0,30\%$
Diagonal conflict admissible	$(-)13,10\%$ $(+)10,00\%$ $(=)85,90\%$	$(-)14,60\%$ $(+)85,40\%$ $(=)0,00\%$
Diagonal conflict étendue	$(-)27,60\%$ $(+)2,90\%$ $(=)69,50\%$	$(-)15,40\%$ $(+)84,40\%$ $(=)0,20\%$

Ces résultats permettent de conclure que sur une plus grande instance du n-puzzle, la diagonal conflict non admissible améliorent moyennement la distance de Manhattan ; tandis que la diagonal conflict admissible et la diagonal conflict étendue n'apportent pas de grandes amélioration à la distance de Manhattan. Néanmoins, ces deux dernières offrent les même performances que la distance de Manhattan pour la grande majorité des instances testées.

6.2 Diagonal conflict comparée à linear conflict

Les résultats des tests par rapport à linear conflict révèlent que pour :

- $31,70\%$ des instances (soit 317 instances) la diagonal conflict non admissible explore moins de noeuds que linear conflict ;
- $37,30\%$ des instances (soit 373 instances) la diagonal conflict non admissible explore plus de noeuds que linear conflict ;
- $31,00\%$ des instances (soit 310 instances) la diagonal conflict non admissible explore le même nombre de noeuds que linear conflict.

Nous observons aussi que pour :

- $72,10\%$ des instances (soit 721 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de linear conflict ;

- 27, 80% des instances (soit 278 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de linear conflict ;
- 0, 10% des instances (soit 1 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de linear conflict.

Dans le cas de la diagonal conflict admissible, les résultats des tests par rapport à linear conflict montrent que pour :

- 9, 00% des instances (soit 90 instances) la diagonal conflict admissible explore moins de noeuds que linear conflict ;
- 41, 30% des instances (soit 413 instances) la diagonal conflict admissible explore plus de noeuds que linear conflict ;
- 49, 70% des instances (soit 497 instances) la diagonal conflict admissible explore le même nombre de noeuds que linear conflict.

Nous observons aussi que pour :

- 64, 10% des instances (soit 641 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de linear conflict ;
- 35, 90% des instances (soit 359 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de linear conflict ;
- 0, 00% des instances (soit 0 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de linear conflict.

Dans le cas de la diagonal conflict étendue, les résultats des tests par rapport à linear conflict montrent que pour :

- 13, 40% des instances (soit 134 instances) la diagonal conflict étendue explore moins de noeuds que linear conflict ;
- 39, 70% des instances (soit 397 instances) la diagonal conflict étendue explore plus de noeuds que linear conflict ;
- 46, 90% des instances (soit 469 instances) la diagonal conflict étendue explore le même nombre de noeuds que linear conflict.

Nous observons aussi que pour :

- 65, 20% des instances (soit 652 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de linear conflict ;
- 34, 70% des instances (soit 347 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de linear conflict ;
- 0, 10% des instances (soit 1 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de linear conflict.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à linear conflict.

Table 6.2 – Tableau récapitulatif de diagonal conflict comparée à linear conflict

-	linear conflict	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)31,70\%$ $(+)37,30$ $(=)31,00\%$	$(-)72,10\%$ $(+)27,80\%$ $(=)0,10\%$
Diagonal conflict admissible	$(-)9,00\%$ $(+)41,30\%$ $(=)49,70\%$	$(-)64,10\%$ $(+)35,90\%$ $(=)0,00\%$
Diagonal conflict étendue	$(-)13,40\%$ $(+)39,70\%$ $(=)46,90\%$	$(-)65,20\%$ $(+)34,70\%$ $(=)0,10\%$

Ces résultats permettent de conclure que sur une plus grande instance du n-puzzle, linear conflict offre des performances nettement meilleur à toutes les versions de la diagonal conflict pour environ la moitié des instances testées. Pour approximativement l'autre moitié, elle offre les mêmes performances que les différentes versions de la diagonal conflict.

6.3 Diagonal conflict comparée à corners tiles

Les résultats des tests par rapport à corners tiles révèlent que pour :

- $44,60\%$ des instances (soit 446 instances) la diagonal conflict non admissible explore moins de noeuds que corners tiles ;
- $11,40\%$ des instances (soit 114 instances) la diagonal conflict non admissible explore plus de noeuds que corners tiles ;
- $44,00\%$ des instances (soit 440 instances) la diagonal conflict non admissible explore le même nombre de noeuds que corners tiles.

Nous observons aussi que pour :

- $45,20\%$ des instances (soit 452 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de corners tiles ;
- $54,70\%$ des instances (soit 547 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de corners tiles ;

- 0, 10% des instances (soit 1 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de corners tiles.

Dans le cas de la diagonal conflict admissible, les résultats des tests par rapport à corners tiles montrent que pour :

- 12, 10% des instances (soit 121 instances) la diagonal conflict admissible explore moins de noeuds que corners tiles ;
- 2, 90% des instances (soit 29 instances) la diagonal conflict admissible explore plus de noeuds que corners tiles ;
- 85, 00% des instances (soit 850 instances) la diagonal conflict admissible explore le même nombre de noeuds que corners tiles.

Nous observons aussi que pour :

- 17, 00% des instances (soit 170 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de corners tiles ;
- 82, 80% des instances (soit 828 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de corners tiles ;
- 0, 20% des instances (soit 2 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de corners tiles.

Dans le cas de la diagonal conflict étendue, les résultats des tests par rapport à corners tiles montrent que pour :

- 26, 90% des instances (soit 269 instances) la diagonal conflict étendue explore moins de noeuds que corners tiles ;
- 3, 70% des instances (soit 37 instances) la diagonal conflict étendue explore plus de noeuds que corners tiles ;
- 69, 40% des instances (soit 694 instances) la diagonal conflict étendue explore le même nombre de noeuds que corners tiles.

Nous observons aussi que pour :

- 17, 80% des instances (soit 178 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de corners tiles ;
- 81, 90% des instances (soit 819 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de corners tiles ;
- 0, 30% des instances (soit 3 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de corners tiles.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à corners tiles.

Table 6.3 – Tableau récapitulatif de diagonal conflict comparée à corner tile

-	corners tiles	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)44,60\%$ $(+)11,40\%$ $(=)44,00\%$	$(-)45,20\%$ $(+)54,70\%$ $(=)0,10\%$
Diagonal conflict admissible	$(-)12,10\%$ $(+)2,90\%$ $(=)85,00\%$	$(-)17,00\%$ $(+)82,80\%$ $(=)0,20\%$
Diagonal conflict étendue	$(-)26,90\%$ $(+)3,70\%$ $(=)69,40\%$	$(-)17,80\%$ $(+)81,90\%$ $(=)0,30\%$

Ces résultats permettent de conclure que sur une plus grande instance du n-puzzle, la diagonal conflict non admissible offre des performances nettement meilleures à corners tiles pour environ la moitié des instances testées et pour l'autre moitié approximativement, elle offre les mêmes performances que corners tiles. Mais la diagonal conflict admissible et la diagonal conflict étendue offrent juste les mêmes performances que corners tiles pour la majorité des instances testées en terme de nombre de noeuds explorés.

6.4 Diagonal conflict comparée à last move

Les résultats des tests par rapport à last move révèlent que pour :

- $64,40\%$ des instances (soit 644 instances) la diagonal conflict non admissible explore moins de noeuds que last move ;
- $35,20\%$ des instances (soit 352 instances) la diagonal conflict non admissible explore plus de noeuds que last move ;
- $0,40\%$ des instances (soit 4 instances) la diagonal conflict non admissible explore le même nombre de noeuds que last move.

Nous observons aussi que pour :

- $96,60\%$ des instances (soit 966 instances) la diagonal conflict non admissible a un temps d'exécution moins que celui de last move ;

- 3, 40% des instances (soit 34 instances) la diagonal conflict non admissible a un temps d'exécution plus que celui de last move ;
- 0, 00% des instances (soit 0 instances) la diagonal conflict non admissible a un temps d'exécution égale à celui de last move.

Dans le cas de la diagonal conflict admissible, les résultats des tests par rapport à last move montrent que pour :

- 64, 90% des instances (soit 649 instances) la diagonal conflict admissible explore moins de noeuds que last move ;
- 34, 90% des instances (soit 349 instances) la diagonal conflict admissible explore plus de noeuds que last move ;
- 0, 20% des instances (soit 2 instances) la diagonal conflict admissible explore le même nombre de noeuds que last move.

Nous observons aussi que pour :

- 95, 90% des instances (soit 959 instances) la diagonal conflict admissible a un temps d'exécution moins que celui de last move ;
- 4, 10% des instances (soit 41 instances) la diagonal conflict admissible a un temps d'exécution plus que celui de last move ;
- 0, 00% des instances (soit 0 instances) la diagonal conflict admissible a un temps d'exécution égale à celui de last move.

Dans le cas de la diagonal conflict étendue, les résultats des tests par rapport à last move montrent que pour :

- 65, 80% des instances (soit 658 instances) la diagonal conflict étendue explore moins de noeuds que last move ;
- 34, 00% des instances (soit 340 instances) la diagonal conflict étendue explore plus de noeuds que last move ;
- 0, 20% des instances (soit 2 instances) la diagonal conflict étendue explore le même nombre de noeuds que last move.

Nous observons aussi que pour :

- 96, 10% des instances (soit 961 instances) la diagonal conflict étendue a un temps d'exécution moins que celui de last move ;
- 3, 90% des instances (soit 39 instances) la diagonal conflict étendue a un temps d'exécution plus que celui de last move ;
- 0, 00% des instances (soit 0 instances) la diagonal conflict étendue a un temps d'exécution égale à celui de last move.

Le tableau ci-dessous récapitule les résultats des performances des différentes versions de la diagonal conflict par rapport à last move.

Table 6.4 – Tableau récapitulatif de diagonal conflict comparée à last move

-	last move	
-	Noeuds explorés	Temps
Diagonal conflict non admissible	$(-)64,40\%$ $(+)35,20\%$ $(=)0,40\%$	$(-)96,60\%$ $(+)3,40\%$ $(=)0,00\%$
Diagonal conflict admissible	$(-)64,90\%$ $(+)34,90\%$ $(=)0,20\%$	$(-)95,90\%$ $(+)4,10\%$ $(=)0,00\%$
Diagonal conflict étendue	$(-)65,80\%$ $(+)34,00\%$ $(=)0,20\%$	$(-)96,10\%$ $(+)3,90\%$ $(=)0,00\%$

Ces résultats permettent de conclure que sur une plus grande instance du n-puzzle, toutes les versions de la diagonal conflict offrent des performances nettement meilleures à last move pour la grande majorité des instances testées.

6.5 Combinaison entre heuristiques

A présent, nous combinons les heuristiques entre elles afin d'observer leur complémentarité et l'impact sur les résultats. Pour ce faire nous avons utilisé la diagonal conflict admissible. Puis nous avons combiné linear conflict, corners tiles et last move (LC+CT+LM) dans un premier temps. Ensuite linear conflict, diagonal conflict et corners tiles (LC+DC+CT) ont été combinés. Enfin, linear conflict, diagonal conflict et last move (LC+DC+LM) ont été combinés.
Le tableau ci-dessous récapitule les résultats des performances des différentes combinaisons d'heuristiques effectuées les unes par rapport aux autres.

Table 6.5 – Tableau récapitulatif des résultats des combinaisons d'heuristiques

-	LC+CT+LM		LC+DC+CT		LC+DC+LM	
-	Noeuds explorés	Temps	Noeuds explorés	Temps	Noeuds explorés	Temps
LC+CT+LM	-	-	(−)48.20% (+)43.40% (=)8.40%	(−)5.70% (+)94.20% (=)0.10%	(−)12.80% (+)14.00% (=)73.20%	(−)19.30% (+)80.10% (=)0.60%
LC+DC+CT	(−)43.40% (+)48.20% (=)8.40%	(−)94.20% (+)5.70% (=)0, 10%	-	-	(−)46.30% (+)46.10% (=)7.60%	(−)86.90% (+)13.00% (=)0.10%
LC+DC+LM	(−)14.00% (+)12.80% (=)73.20%	(−)80.10% (+)19.30% (=)0.60%	(−)46.10% (+)46.30% (=)7, 60%	(−)13.00% (+)86, 90% (=)0, 10%	-	-

Ces résultats permettent de conclure que la combinaison LC+DC+CT est meilleure que les autres car offrant de meilleurs résultats par rapport aux autres pour près de la moitié des instances traitées. Suivie de la combinaison LC+CT+LM qui offre des résultats légèrement meilleurs à ceux de LC+DC+LM.

Conclusion

Cette partie a présenté les nouvelles heuristiques proposées diagonal conflict et ses différents versions. Elle a aussi présenté les résultats de l'évaluation de ces heuristiques et discuté de l'apport des nouvelles heuristiques proposées sur l'existant.

Conclusion

Dans ce travail, nous nous sommes intéressés à la résolution de problème par la recherche, un des axes de recherche de l'Intelligence artificielle. Nous y avons étudié le jeu de n-puzzle et les différentes heuristiques existantes pour la résolution du jeu de n-puzzle.

Nous avons proposé et implémenté de nouvelles heuristiques : diagonal conflict admissible, non admissible et étendue. La diagonal conflict donne d'assez bonnes performances mais n'est pas admissible. Lorsque l'admissibilité est forcée, les performances sont considérablement réduites. La diagonal conflict donne de meilleures performances sur le 15-puzzle par rapport au 8-puzzle. Combinée la diagonal conflict admissible à d'autres heuristiques permet d'obtenir des résultats appréciables.

Les travaux réalisés montrent que de meilleures heuristiques peuvent encore être créés pour la résolution du jeu de n-puzzle. Aussi des améliorations peuvent être apporté au travail.
Pour de futurs travaux, on peut envisager :

- faire des tests sur plus d'instances du 15-puzzle afin de mieux observer l'impact de l'amélioration apportées par les différentes versions de la diagonal conflict ;
- développer d'autres heuristiques qui exploitent les forces de ces nouvelles heuristiques et des autres pour en tirer les meilleurs résultats suivant l'état ;
- faire des tests de ces heuristiques sur de plus grandes instances du n-puzzle.

Bibliographie

[1] Introduction to Artificial Intelligence, Wolfgang Ertel, 2011, Springer, 329 pages.

[2] Artificial Intelligence, Elaine Richie, 3rd Ed., 1983, McGraw–Hill, 436 pages.

[3] Introduction to artificial intelligence, E. Charniak and D. McDermott, Addison-Wesley Publ., 1985.

[4] Computational Intelligence : A Logical Approach, David Poole, Alan Mackworth, and Randy Goebel, Oxford University Press, 1998.

[5] Artificial Intelligence : A New Synthesis, Nils J. Nilsson, 1st Ed., 1998, Morgan Kaufmann, 513 pages.

[6] Dictionnaire Larousse, LAROUSSE, Edition 2014, .

[7] A World Championship Caliber Checkers Program, J. Schaeffer , J. Culberson , N. Treloar , B. Knight , P. Lu and D. Szafron, Departement of Computing Science, University of Alberta, Canada.

[8] Artificial Intelligence, Patrick Henry Winston, 3rd Ed., 1999, Addison-Wesley.

[9] Artificial Intelligence : A Modern Approach, Stuart Russell et Peter Norvig, 3rd Ed., PEARSON, 1152 pages.

[10] Intelligent Systems : A Modern Approach, Crina Grosan and Ajith Abraham, Volume 17, 2011, Springer, 465 pages.

[11] https ://www.research.ibm.com/deepblue/ Consulter le 30 Août 2016

[12] https ://deepmind.com/alpha-go Consulter le 30 Août 2016.

[13] https ://webdocs.cs.ualberta.ca/ chinook/ Consulter le 30 Août 2016.

[14] http ://www.ibm.com/watson/ Consulter le 30 Août 2016.

[15] Heuristic Search - Theory and Applications, Stefan Edelkamp and Stefan Schrodl, 2012, ELSEVIER, 865 pages.

[16] Problem-Solving Methods in Artificial Intelligence, Nils J. Nilsson, 1971, McGraw-Hill, 256 pages.

[17] Artificial Intelligence for Computer Games, Pedro Antonio González-Calero and Marco Antonio Gómez-Martín 2011, Springer-Verlag, 212 pages.

[18] Soft Computing and Intelligent Systems Design : Theory, Tools and Applications, Fakhreddine O. Karray and Clarence W De Silva, 1st Ed., 2004, PEARSON, 584 pages.

[19] Problem Solving Methods - Understanding Description Development and Reuse, Dieter Fensel, 2000, Springer, 173 pages.

[20] Des heuristiques plus performantes si avec la fonction on garde la raison, Henri Farreny, 2002, 13ème Congrès Francophone AFRIF-AFIA de Reconnaissance des Formes et d'Intelligence Artificielle (RFIA 2002), Angers-France, 08/01/2002-10/01/2002.

[21] Iterative-deepening-A* : an optimal admissible tree search, Richard E. Korf Proceedings Ninrh International Joint Conference on Artificial Intelligence, Los Angeles, 1985.

[22] Finding optimal solutions to the twenty-four puzzle, R. E. Korf, L. A. Taylor, Proceedings of AAAI-96, pp 1202-1207, Portland, 1996.

[23] Pattern Databases, J. Culberson, J. Schaeffer, Computational Intelligence , Vol.14, N°4, pp 318-334, 1998.

[24] Disjoint Pattern Database Heuristics, Richard E. Korf, Ariel Felner ———

[25] Rapid Application Development, MARTIN, Macmillan James, 1991, Macmillan Coll Div, 736 pages.

[26] Apprenez à programmer en python, Vincent LE GOFF, 2011, OpenClassrooms, 418 pages.

[27] Programmation Python - Conception et optimisation, Tarek Ziadé, 2nd Ed.,Eyrolles , 594 pages.

[28] Apprendre à programmer avec Python 3, Gérard Swinnen, 3rd Ed., 2012, Eyrolles , 435 pages.

[29] Git : Gestionnaire de version, http ://git-scm.com/about, Consulter le 17 Septembre 2015.

[30] Git : Gestionnaire de version, http ://git-scm.com/documentation, Consulter le 17 Septembre 2015.

[31] Pragmatic guide to Git, Travis Swicegood, 2010, Pragmatic Bookshelf, 160 pages.

[32] Notes on the '15' puzzle, W. W. Johnson, W. E. Storey, American Journal of Mathematics , Vol. 2, pp 397-404, 1879.

Annexe A

Annexes

Code sources de quelques fonctions

Dans cette section, nous présentons quelques codes sources de fonctions utilisées dans le travail effectué.

Distance de Manhattan

```
def manhattanDistance(node):
    state = node.state
    n = find_n(state.liste)
    h = 0
    i = 0
    while i < len(state.liste):
        if((state.liste[i] != 0) and (state.liste[i] != goal_state.liste[i])):
            elmt_addr_in_goal = goal_state.liste.index(state.liste[i])
            k = abs((i // n - elmt_addr_in_goal // n)) + abs((i % n - elmt_addr_in_goal % n))
            h += k
        i += 1
    return h
```

Linear Conflict

```
def linearConflict(node):
    state = node.state
    n = find_n(state.liste)
    h = manhattanDistance(node)
    # Horizontal conflict
    line = 0
    while line < len(state.liste):
        tmpMax = -1
        col = 0
        while col < n:
            i = int(line + col)
            if ((state.liste[i] != 0) and (state.liste[i] // n == goal_state.liste[i] // n)):
                if state.liste[i] > tmpMax:
                    tmpMax = state.liste[i]
                else:
                    h += 2
            col += 1
        line += n
    # Vertical conflict
    col = 0
    while col < n:
```

```
        tmpMax = -1
        line = 0
        while line < len(state.liste):
            i = int(col + line)
            if ((state.liste[i] != 0) and (state.liste[i] % n == goal_state.liste[i] % n)):
                if state.liste[i] > tmpMax:
                    tmpMax = state.liste[i]
                else:
                    h += 2
            line += n
        col += 1

    return h
```

Corner Tile

```
def cornersTiles(node): # Corners-Tiles
    state = node.state
    n = find_n(state.liste)
    h = manhattanDistance(node)
    commonCorner1 = 0
    commonCorner2 = 0
    commonCorner3 = 0
    commonCorner4 = 0
    # Coin supérieur gauche / Upper left corner
    if (state.liste[0] != goal_state.liste[0]):
        if (state.liste[1] == goal_state.liste[1]):
            h += 2
            commonCorner1 = 1
        if (state.liste[n] == goal_state.liste[n]):
            h += 2
            commonCorner4 = 1
    # Coin supérieur droit / Upper right corner
    if ((state.liste[n - 1] != 0) and (state.liste[n - 1] != goal_state.liste[n - 1])):
        if ((state.liste[n - 2] == goal_state.liste[n - 2]) and (commonCorner1 == 0)):
            h += 2
        if (state.liste[2 * n - 1] == goal_state.liste[2 * n - 1]):
            h += 2
            commonCorner2 = 1
    # Coin inférieur droit / lower right corner
    if ((state.liste[n * n - 1] != 0) and (state.liste[n * n - 1] != goal_state.liste[n * n - 1])):
        if ((state.liste[(n - 1) * n - 1] == goal_state.liste[(n - 1) * n - 1])
        and (commonCorner2 == 0)):
            h += 2
        if (state.liste[n * n - 2] == goal_state.liste[n * n - 2]):
            h += 2
            commonCorner3 = 1
    # Coin inférieur gauche / lower left corner
    if ((state.liste[(n - 1) * n] != 0)
    and (state.liste[(n - 1) * n] != goal_state.liste[(n - 1) * n])):
        if ((state.liste[(n - 1) * n + 1] == goal_state.liste[(n - 1) * n + 1])
        and (commonCorner3 == 0)):
            h += 2
        if ((state.liste[(n - 2) * n] == goal_state.liste[(n - 2) * n]) and (commonCorner4 == 0)):
            h += 2

    return h
```

Diagonal Conflict (non admissible)

```
def diagonalConflict1(node):
    state = node.state
    n = find_n(state.liste)
    h = manhattanDistance(node)
    i = 0
```

```
    while i < len(state.liste):
        if ((state.liste[i] != 0) and (goal_state.liste[i] != 0)
        and (state.liste[i] != goal_state.liste[i])):
            # Vérifie si le coin supérieur gauche de la prosition actuelle existe et le traite
            if ((i // n) > 0 and (i % n) > 0):
                if (state.liste[i - n - 1] == goal_state.liste[i]):
                    if ((state.liste[i - n] != 0)
                    and (state.liste[i - n] == goal_state.liste[i - n])):
                        h += 2
                    elif ((state.liste[i - 1] != 0)
                    and (state.liste[i - 1] == goal_state.liste[i - 1])):
                        h += 2
            # Vérifie si le coin supérieur droit de la prosition actuelle existe et le traite
            elif ((i // n) > 0 and (i % n) < (n - 1)):
                if (state.liste[i - n + 1] == goal_state.liste[i]):
                    if ((state.liste[i + 1] != 0)
                    and (state.liste[i + 1] == goal_state.liste[i + 1])):
                        h += 2
                    elif ((state.liste[i - n] != 0)
                    and (state.liste[i - n] == goal_state.liste[i - n])):
                        h += 2
            # Vérifie si le coin inferieur droit de la prosition actuelle existe et le traite
            elif ((i // n) < (n - 1) and (i % n) < (n - 1)):
                if (state.liste[i + n + 1] == goal_state.liste[i]):
                    if ((state.liste[i + n] != 0)
                    and (state.liste[i + n] == goal_state.liste[i + n])):
                        h += 2
                    elif ((state.liste[i + 1] != 0)
                    and (state.liste[i + 1] == goal_state.liste[i + 1])):
                        h += 2
            # Vérifie si le coin inférieur gauche de la prosition actuelle existe et le traite
            elif ((i // n) < (n - 1) and (i % n) > 0):
                if (state.liste[i + n - 1] == goal_state.liste[i]):
                    if ((state.liste[i - 1] != 0)
                    and (state.liste[i - 1] == goal_state.liste[i - 1])):
                        h += 2
                    elif ((state.liste[i + n] != 0)
                    and (state.liste[i + n] == goal_state.liste[i + n])):
                        h += 2
        i += 1
    return h
```

Diagonal Conflict (admissible)

```
def diagonalConflict2(node):
    state = node.state
    n = find_n(state.liste)
    h = manhattanDistance(node)
    i = 0
    while i < len(state.liste):
        if ((state.liste[i] != 0) and (goal_state.liste[i] != 0)
        and (state.liste[i] != goal_state.liste[i])):
            # Vérifie si le coin supérieur gauche de la prosition actuelle existe et le traite
            if ((i // n) > 0 and (i % n) > 0):
                if (state.liste[i - n - 1] == goal_state.liste[i]):
                    if ((state.liste[i - n] != 0) and (state.liste[i - n] == goal_state.liste[i - n])
                    and (state.liste[i - 1] != 0) and (state.liste[i - 1] == goal_state.liste[i - 1])):
                        h += 2
            # Vérifie si le coin supérieur droit de la prosition actuelle existe et le traite
            elif ((i // n) > 0 and (i % n) < (n - 1)):
                if (state.liste[i - n + 1] == goal_state.liste[i]):
                    if ((state.liste[i + 1] != 0) and (state.liste[i + 1] == goal_state.liste[i + 1])
                    and (state.liste[i - n] != 0) and (state.liste[i - n] == goal_state.liste[i - n])):
                        h += 2
```

```
            # Vérifie si le coin inferieur droit de la prosition actuelle existe et le traite
            elif ((i // n) < (n - 1) and (i % n) < (n - 1)):
                if (state.liste[i + n + 1] == goal_state.liste[i]):
                    if ((state.liste[i + n] != 0) and (state.liste[i + n] == goal_state.liste[i + n])
                    and (state.liste[i + 1] != 0) and (state.liste[i + 1] == goal_state.liste[i + 1])):
                        h += 2
            # Vérifie si le coin inférieur gauche de la prosition actuelle existe et le traite
            elif ((i // n) < (n - 1) and (i % n) > 0):
                if (state.liste[i + n - 1] == goal_state.liste[i]):
                    if ((state.liste[i - 1] != 0) and (state.liste[i - 1] == goal_state.liste[i - 1])
                    and (state.liste[i + n] != 0) and (state.liste[i + n] == goal_state.liste[i + n])):
                        h += 2
        i += 1
    return h
```

All Conflict

```
def allConflict(node):
  state = node.state
  n = find_n(state.liste)
  h = manhattanDistance(node)
  flagLine = [0,0,0,0]
  flagCol = [0,0,0,0]
  # Horizontal conflict
  line = 0
  while line < len(state.liste):
    tmpMax = -1
    col = 0
    while col < n:
      i = int(line + col)
      if ((state.liste[i] != 0) and (state.liste[i] // n == goal_state.liste[i] // n)):
        if state.liste[i] > tmpMax:
          tmpMax = state.liste[i]
        else:
          h += 2
          flagLine[line/n] = 1
      col += 1
    line += n
  # Vertical conflict
  col = 0
  while col < n:
    tmpMax = -1
    line = 0
    while line < len(state.liste):
      i = int(col + line)
      if ((state.liste[i] != 0) and (state.liste[i] % n == goal_state.liste[i] % n)):
        if state.liste[i] > tmpMax:
          tmpMax = state.liste[i]
        else:
          h += 2
          flagCol[col/n] = 1
      line += n
    col += 1

  i = 0
  while i < len(state.liste):
    if ((flagCol[i%n] != 1) and (flagLine[i//n] != 1)
     and (state.liste[i] != goal_state.liste[i])):
      if ((state.liste[i] != 0) and (goal_state.liste[i] != 0)
       and (state.liste[i] != goal_state.liste[i])):
        # Vérifie si le coin supérieur gauche de la prosition actuelle existe et le traite
        if ((i // n) > 0 and (i % n) > 0):
          if (state.liste[i - n - 1] == goal_state.liste[i]):
            if ((state.liste[i - n] != 0) and (state.liste[i - n] == goal_state.liste[i - n])
```

```
48              and (state.liste[i - 1] != 0) and (state.liste[i - 1] == goal_state.liste[i - 1])):
49                h += 2
50          # Vérifie si le coin supérieur droit de la prosition actuelle existe et le traite
51          elif ((i // n) > 0 and (i % n) < (n - 1)):
52            if (state.liste[i - n + 1] == goal_state.liste[i]):
53              if ((state.liste[i + 1] != 0) and (state.liste[i + 1] == goal_state.liste[i + 1])
54               and (state.liste[i - n] != 0) and (state.liste[i - n] == goal_state.liste[i - n])):
55                h += 2
56          # Vérifie si le coin inferieur droit de la prosition actuelle existe et le traite
57          elif ((i // n) < (n - 1) and (i % n) < (n - 1)):
58            if (state.liste[i + n + 1] == goal_state.liste[i]):
59              if ((state.liste[i + n] != 0) and (state.liste[i + n] == goal_state.liste[i + n])
60               and (state.liste[i + 1] != 0) and (state.liste[i + 1] == goal_state.liste[i + 1])):
61                h += 2
62          # Vérifie si le coin inférieur gauche de la prosition actuelle existe et le traite
63          elif ((i // n) < (n - 1) and (i % n) > 0):
64            if (state.liste[i + n - 1] == goal_state.liste[i]):
65              if ((state.liste[i - 1] != 0) and (state.liste[i - 1] == goal_state.liste[i - 1])
66               and (state.liste[i + n] != 0) and (state.liste[i + n] == goal_state.liste[i + n])):
67                h += 2
68      i += 1
69    return h
```