Table des matières

1 Introduction

Ce projet Informatique que j'ai eu à réaliser est la concrétisation de plusieurs mois de dur labeur durant lesquelles j'ai eu à beaucoup me documenter sur mon sujet. Ma curiosité et mon amour pour les jeux de type puzzle ou "casse-tête" par exemple : taquin, le jeu 2048, Sudoku ... m'ont poussé à me poser la question à savoir comment ces types de jeux pouvaient être résolus de manière automatique par des programmes prévus à cet effet.

Ce sujet étant un peu trop vaste j'ai décidé d'orienter mes recherches sur les outils et méthodes existant pour résoudre un N-puzzle ou taquin en français de manière optimale et la plus rapide possible.

A cet effet nous allons développer dans la partie suivante le procédé de création du jeu, ces règles et les solutions trouvées pour répondre à cette problématique.

2 Présentation du Projet

2.1 Description du jeu

Nous avons une grille 3*3/4*4/5*5 composée de plusieurs cases où une case est enlevée ; cette case enlevée permet le déplacement de cases adjacentes à la case vide vers l'emplacement de cette dernière et ainsi l'ancien emplacement de la case active devient à son tour vide.

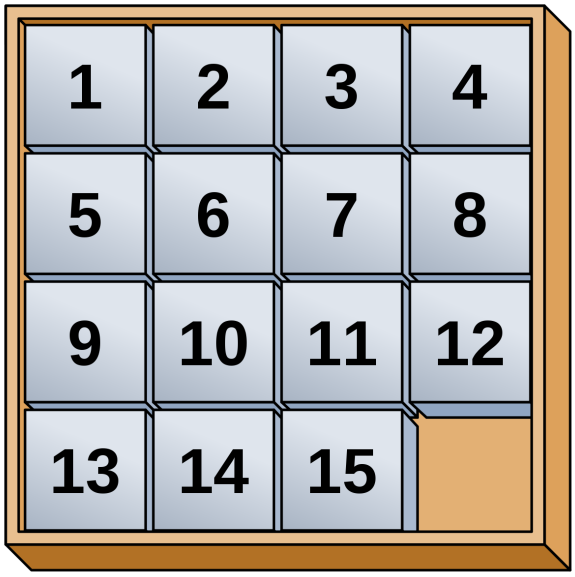

FIGURE 1 – 15-Puzzle

2.2 Règles du jeu

Les règles du taquin sont très simples . Au début du jeu le puzzle est mélangé et l'objectif pour le joueur est de revenir à l'état initial du puzzle . Les seuls déplacements de case autorisés sont horizontaux et verticaux vers la case vide.

2.3 Les outils utilisés

Les principaux outils utilisés sont :

- Unity : Un moteur de jeu
- Visual Studio 2019 : un logiciel de développement
- Gimp : pour la création de la texture du jeu

2.4 Création du Taquin sous Unity

Une fois que le jeu , ses règles et les outils de création sont connus , il ne reste plus que son développement.
Le Taquin peut être vu comme un ensemble de cellules indépendantes qui sont soumises aux mêmes lois.Donc la base du jeu est cette cellule ou case qui est dupliquée autant de fois qu'il le faut et placée au bon endroit.

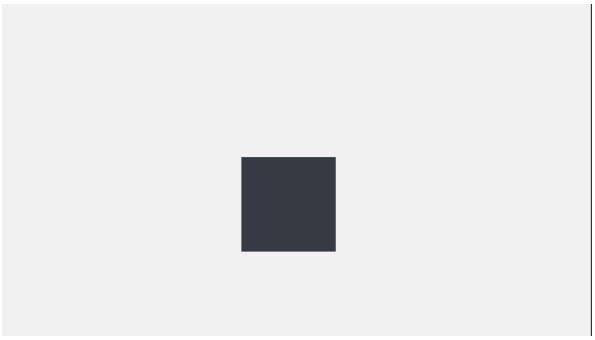

FIGURE 2 – Case Puzzle

Donc sous Unity cette case est un 3D object (Plane Mesh Filter) auquel on ajoute un mesh collider pour la gestion des collisions et pour qu'on puisse le déplacer ,ainsi qu'un mesh render pour qu'il soit visible et pour pouvoir appliquer un shader dessus

Grâce a Gimp , Nous avons crée la texture du Puzzle qui va être appliquée a ces cases . Suivant que le jeu soit un 3*3 ou un 4*4 un découpage de la texture est fait et est appliqué à chaque case du puzzle .

A l'initialisation chaque case est placé suivant un ordre prédéfini

FIGURE 3 – représentation du Puzzle 3*3 sous Unity

Ensuite on désactive le mesh render et le mesh collider de la 9 ème case ce qui permettra aux cases adjacentes de pouvoir se déplacer vers cette case. Au début du jeu un nombre aléatoire de déplacement sont effectués ce qui permet de casser le Puzzle.

2.5 Modèles de Résolution

Le jeu est enfin crée et est totalement jouable. Il suffit pour le joueur de remettre en ordre le puzzle pour gagner.Mais forcement il existe des cas largement plus corsés et complexes que d'autres.L'aide d'un algorithme de résolution pourrait être la bienvenue.

Un Puzzle a un état T pourrait être vu comme un Graphe , en considérant chaque déplacement possible vers la case vide comme étant un fils de cet état. De ce fait Nous pouvons donc nous imaginer parcourir ce Graphe jusqu'à trouver le bon état du puzzle.

Un type de parcours répond nécessairement a plusieurs critères :

- La complétude : la garantie de trouver une solution si elle existe
- L'optimalité : la garantie que la solution trouvée est optimale
- La complexité de temps : le temps que met l'algorithme a trouvé une solution
- La complexité d'espace : l'espace mémoire utilisé par l'algorithme

Il existe plusieurs type de parcours de Graphe divisés en 2 grandes famille :

- La recherche non-informée
- La recherche informée

La recherche non-informée est une recherche en Brute Force sans distinction des noeuds parcourus, tous les noeuds se valent.

La recherche non informé comporte entre autre :

- Breadth First Search(BFS)
- Depth First Search(DFS)
- Limited Depth First Search(LDFS)
- Iterative Deepening Search(IDS)
- Uniform Cost Search(UCS)

Ces différents algorithmes de parcours de Graphe ne sont pas adaptés a la recherche de solution pour un puzzle ; généralement soit l'optimalité ou la completude n'est pas garantie , soit la complexité d'espace est monstrueuse.

La recherche informée par contre est une recherche privilégiant les noeuds les plus prometteurs grâce a une fonction appelée Heuristique.

La recherche informée comporte entre autre :

- Greedy Best-first Search(GBFS)
- A$\star$ Search (A*)
- Iterative Deepening A$\star$ (IDA*)

Les algorithme A* et IDA* sont optimaux et complets , ce qui n'est pas le cas du Greedy Best-first Search.

Nos algorithmes de résolution seront donc le A* et le IDA*

3 Aspect Technique

3.1 Les Heuristiques

Une heuristique est une fonction qui permet d'estimer la distance du noeud actuel au noeud objectif.

Elle doit être admissible , consistante et minorante.

- Une heuristique est dite admissible si la valeur réelle du coût séparant le noeud actuel du noeud final est supérieur ou égale a la valeur d'heuristique .
- Une heuristique est dite consistante quand la valeur d'heuristique est inférieure ou égale au coût réelle entre le noeud actuel et un noeud fils + la valeur d'heuristique du noeud fils
- Une heuristique est dite minorante lorsque la valeur d'heuristique est inférieure a la longueur du chemin le plus court entre le noeud actuel et le noeud objectif

Parmi les heuristiques existantes pour la résolution de N-puzzle, J'en ai implémenter quelques une

3.1.1 La distance de Hamming

La distance de Hamming ou Misplaced Tiles est le nombre de cases du puzzle mal placés , en ne comptant pas la case vide.Pour un puzzle 3*3 la valeur d'heuristique peut aller de 0 a 8.

3.1.2 La distance de Manhattan

La distance de manhattan est l'une des heuristiques les plus connus pour résoudre un N-puzzle.La valeur d'heuristique d'un état du puzzle est égale a la somme des distances de manhattan de chaque case excepté pour la case vide avec :

$$m_i = |X_B - X_A| + |Y_B - Y_A|$$

(X_B, Y_B) : Coordonnés de l'emplacement de la bonne position de la case et (X_A, Y_A) :Coordonnés actuelle de la case.

3.1.3 La Linear Conflict

Le Linear Conflict est une amélioration de l'heuristique de manhattan. La distance estimée de manhattan ne prend pas en compte le fait que deux pièces du puzzle d'un état peuvent se bloquer entre elles.Dans ce cas deux déplacements verticaux supplémentaires sont a prévoir en plus pour chaque conflit linéaire qui existe.

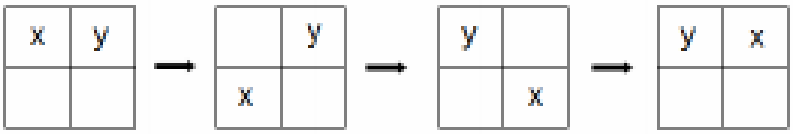

FIGURE 4 – Exemple linear conflict

LC = manhattandistance(etat) + 2*nombredelinearconflict

3.1.4 Additive Pattern Database

L'additive Pattern Database est l'heuristique la plus performante que j'ai eu a implémenter .Elle est uniquement implémentée pour le puzzle 4*4

Une Pattern Database est une base de donnés contenant toutes les positions et permutations possibles d'une zone du puzzle

ainsi que la valeur du nombre de coût minimale pour passer de l'état actuel de la zone de ce puzzle a la zone correctement agencée de ce puzzle.

Comme son nom l'indique l'additive Pattern Database consiste a partitionner le puzzle en plusieurs parties de manières disjointes et d'appliquer un Breadth First Search sur chaque partie pour pouvoir stocker toutes les permutations d'une partie ainsi que le nombre de coup minimale pour le résoudre.

Une fois toutes les bases de données remplit la valeur d'heuristique d'un état s'obtient en additionnant le nombre de coup minimale a effectue pour résoudre chaque sous-partie du puzzle.

Dans mon projet il y'a 2 types de partitionnement : Le 6-6-3 et le 5-5-5

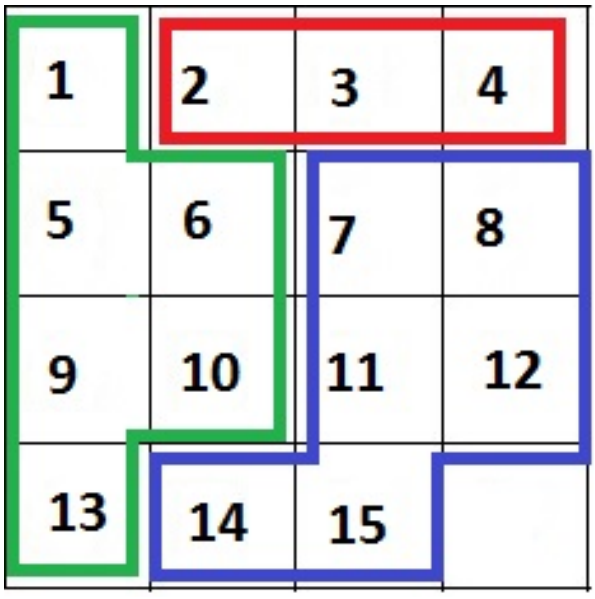

FIGURE 5 – Le Partitionnement 6-6-3

Les bases de donnés sont très longues a générer , il m'a fallut une moyenne de 10 heures pour obtenir les (16 !)/(16-7) ! permutations d'un pattern a 6 éléments , la ou un pattern a 5 éléments avec (16 !)/(16-6) ! permutations prend en moyenne 5 minutes.

3.2 L'algorithme A*

L'algorithme A* comme vu précédemment est un algorithme de recherche informé qui est optimal et complet.

Les éléments principaux qui caractérisent le A* sont :

- Une Open List : c'est la liste des noeuds pas encore visités
- Une Closed List : la liste des noeud déjà visités
- Un noeud de Depart : l'état du puzzle a résoudre
- Un noeud d'arrivé : l'état du puzzle résolu
- Une fonction de score = $f(n) = h(n)+g(n)$ avec $h(n)$ valeur d'heuristique de l'état et $g(n)$ la valeur de la génération ou profondeur a partir du noeud initial

Le fonctionnement de l'algorithme est simple . Au début de l'algorithme le noeud initial est mis dans la Closed List et ses fils dans l'Open List qui est une pile prioritaire.

La priorité d'un noeud est déterminée par la fonction $f(n)$ .Ainsi le premier noeud de l'Open List est dépilé et aussitôt mis la Closed List et ses fils sont mis dans l'Open List si elle n'y appartenait pas déjà.

Les principaux avantages de A* sont qu'un noeud déjà rencontré n'est jamais repris donc pas de cycle et que grâce a une bonne heuristique les noeuds les plus prometteurs sont toujours explores en priorité donc le nombre de noeuds en mémoire est largement moins que pour un BFS

L'inconvénient majeur de cet algorithme est la complexité d'espace : tous les noeuds sont gardés en mémoire ce qui rend son usage impossible pour un puzzle supérieur au 3*3

3.3 IDA*

L'IDA* ou Iterative Deepening A* est un peu le mélange entre un Iterative Deepening Search (IDS) et l'algorithme A* car il a la caractéristique d'être optimal et d'utiliser des heuristiques comme le A* et d'avoir un fonctionnement de IDS.

Les éléments principaux qui caractérisent le IDA* sont :

- Un Treeshold : la limite du score $f(n)$ qui régit la profondeur maximale de recherche
- Un noeud de Départ : l'état du puzzle a résoudre
- Un noeud d'arrivé : l'état du puzzle résolu
- Une fonction de score = $f(n) = h(n)+g(n)$ avec $h(n)$ valeur d'heuristique de l'état et $g(n)$ la valeur de la generation ou profondeur a partir du noeud initial

Le fonctionnement du IDA* est pratiquement le même que celui de l'IDS, a la différence ou la valeur du Treeshold est definit par le score $f(n)$

L'IDA* est une fonction récursive qui a chaque itération augmente la valeur du treeshold jusqu'à trouver le noeud Objectif.La nouvelle valeur de Treeshold est le plus petit $f(n)$ rencontrée supérieur au treeshold actuel.

Pour faire simple pendant le parcours si le noeud suivant a une valeur de $f(n)$ inférieur ou égale au Treeshold alors on regarde ses fils sinon on passe au noeud suivant . Tout cela de manière récursive donc on ne stocke pas de noeud . On ne stocke que le chemin actuel pour éviter les cycles.

Le principal avantage de l'IDA* est qu'il ne consomme presque pas de mémoire comparé a A* mais aussi son plus grand inconvénient est qu'il peut repasser X fois sur le même noeud.

4 Résultats

Tous les Algorithmes et Heuristiques présentés ci dessus on était implémentés dans le jeu de Taquin.

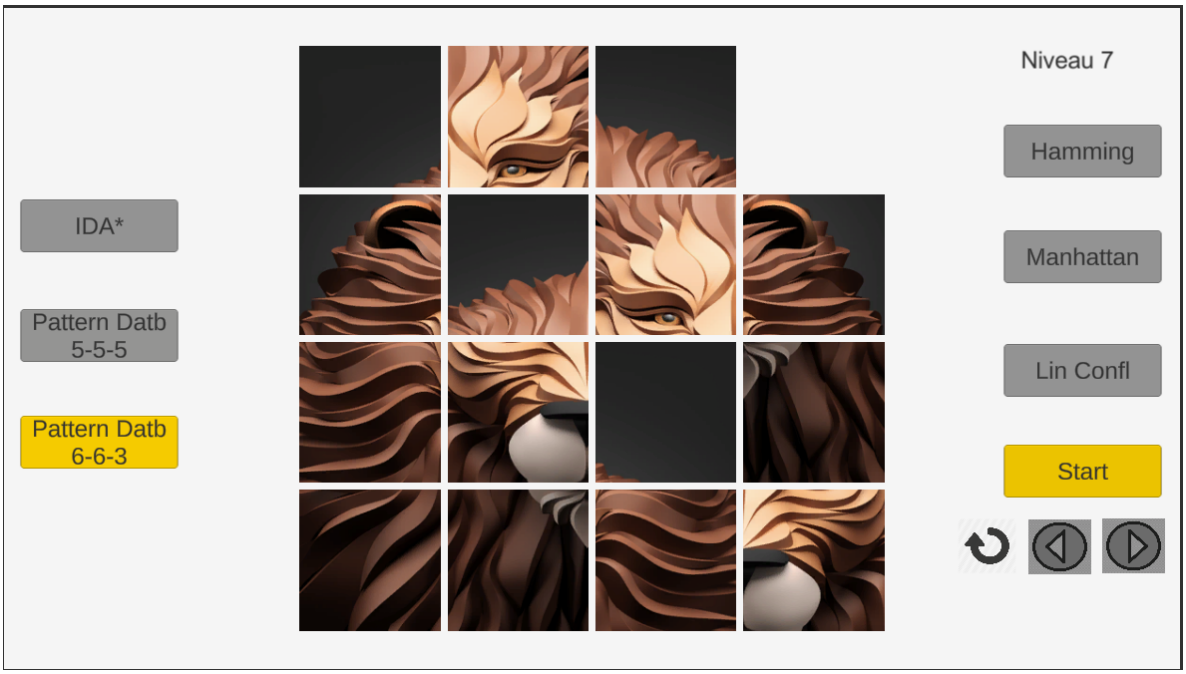

FIGURE 6 – Interface finale jeu

Les algorithmes et heuristiques implémentés sont jugés suivant trois critères qui sont : Le temps mis pour résoudre le puzzle,le nombre de noeud générés et la précision de l'estimation de la valeur d'heuristique.

La comparaison de l'IDA* et de A* montre que utilisé avec les mêmes heuristique l'IDA* est plus rapide que l'A*.Cela n'empêche pas l'A* d'être aussi très rapide (cf. tableau ci dessous)

Nous avons aussi remarqué une corrélation entre la précision distance estimée et le nombre de noeud généré : plus la distance s'approche de la réalité , moins le nombre de noeud exploré est grand.

TEMPS MIS

	Moy 1er	Moy 2eme	Moy 3eme	Moy 4eme
3*3 Puzzle				
A* Hamming	00 :00 :00,02	00 :02 :06,10	00 :01 :33,37	00 :24 :10,10
A* Manhattan	00 :00 :00,00	00 :00 :01,45	00 :00 :00,64	00 :00 :07,05
A* Manh+linc	00 :00 :00,00	00 :00 :01,46	00 :00 :00,62	00 :00 :05,61
IDA* linc	00 :00 :00,00	00 :00 :00,35	00 :00 :00,34	00 :00 :00,73
4*4 Puzzle				
IDA* linc	00 :00 :00,00	00 :00 :04,95	00 :02 :39,39	00 :34 :00,42
IDA* PD 5-5-5	00 :00 :00,00	00 :00 :00,81	00 :00 :09,19	00 :00 :50,80
IDA* PD 6-6-3	00 :00 :00,00	00 :00 :00,81	00 :00 :02,31	00 :00 :06,54

NBR DE NOEUDS

	Moy 1er	Moy 2eme	Moy 3eme	Moy 4eme
3*3 Puzzle				
A* Hamming	182,4	17135,2	10942,2	52848
A* Manhattan	42,6	1894	1112,2	3980,5
A* Manh+linc	47,6	1700	1011,8	3423
IDA* linc	96,2	3802,6	3187,6	8423
4*4 Puzzle				
IDA* linc	52,8	69248	1760741,6	27619026,5
IDA* PD 5-5-5	23,4	17131	147457,6	845967,5
IDA* PD 6-6-3	42,4	14845	45986,8	133772,5

DISTANCE ESTIMEE

	Moy 1er	Moy 2eme	Moy 3eme	Moy 4eme
3*3 Puzzle	11,6	23,8	22,4	27
A* Hamming	6	7	5,6	8
A* Manhattan	8,8	13	12,4	17
A* Manh+linc	9,2	16,2	12,8	18
IDA* linc	9,2	16,2	12,8	18
4*4 Puzzle	12,4	34,6	49	54
IDA* linc	10,4	23,4	36,2	42
IDA* PD 5-5-5	12,4	25,4	45	43
IDA* PD 6-6-3	10,8	24,6	42,6	45

Le tableau ci-dessus est la moyenne de temps , nombre de noeuds générés et distance sur 4 séries de test comportant chacun 5 niveaux du jeu . Ces séries de test nous permettent de dresser un constat :

- L'IDA* est plus efficace que l'A*
- Le A* est inutilisable pour un puzzle plus grand que du 3*3
- Les heuristiques sont classés suivant leurs efficacités dans cette ordre : $Hamming \leq Manhatann \leq LinearConflict \leq PatternDatabase5-5-5 \leq PatternDatabase6-6-3$

L'IDA* combiné avec le LinearConflict est capable de résoudre n'importe quelle puzzle 3*3 en moins de 1 seconde . Le Devil Configuration sensé être le puzzle 3*3 le plus dure est résolu en 0,63 seconde.

L'IDA* combiné avec le PatternDatabase 6-6-3 est capable de résoudre en un temps raisonnable la majeure partie des puzzles 4*4. Pour les cas extrêmes de Puzzle 4*4 il faudra compté plusieurs heures ainsi que des milliards de noeuds générés.

5 Conclusion

Ce projet a été un réel plaisir tant au niveau de la partie algorithmique et programmation que la partie recherche.

L'algorithme est capable de résoudre n'importe quelle puzzle 3*3 en moins de 31 coups qui est son nombre de coup optimal max et n'importe quelle puzzle 4*4 en moins de 80 coups.

D'autres Heuristique aurait pu être implementé comme la Walking Distance, La distance de Tchebychev,La fringe Database,les Corner Tile,Pattern Database 7-8 etc . Mais les heuristiques les plus essentielles en mon sens sont présentes dans le projet.

Une manière d'améliorer ce projet serait d'accroître la vitesse du IDA* qui traite que $\simeq$ 800000 noeuds/minutes la ou les programmes les plus performants en traite des centaines de millions par minutes ou réduire le poids des Databases qui pèsent actuellement $\simeq$ 250 MegaOctects de mémoire.

Références

[1] https://www.ijcai.org/Proceedings/93-1/Papers/035.pdf

[2] https://www.aaai.org/Papers/JAIR/Vol22/JAIR-2209.pdf

[3] https://www.blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288

[4] https://courses.cs.washington.edu/courses/cse473/12sp/slides/04-heuristics.pdf

[5] https://algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/implementing-a-star-to-solve-n-puzzle/

[6] https://www.ics.uci.edu/~kkask/Fall-201420CS271/slides/03-InformedHeuristicSearch.pdf