



Résolution de problèmes

TP du Taquin

AJALE Saâd
BENHADDOU Lahcen

30/03/2024

Ce rapport résume les différentes étapes de la réalisation du TP du taquin

TABLES DE MATIERES :

I.	INTRODUCTION.....	3
II.	CONCEPTION.....	4
III.	MODELISATION.....	5
1.	RECHERCHE INFORME A*	6
2.	GRAPHE DE RESOLUTION.....	8
IV.	SIMULATION.....	11
V.	Probleme et solutions.....	16
VI.	CONCLUSION.....	17

I. INTRODUCTION

Le jeu de Taquin, avec sa grille de 3x3 cases numérotées et une case vide pour les déplacements, offre un défi intrigant qui a captivé les amateurs de casse-tête depuis des décennies. Son histoire fascinante remonte au 19ème siècle, où il a vu le jour comme un passe-temps inventif et stimulant. La nécessité de déplacer les tuiles dans un ordre spécifique met en lumière la complexité de la résolution de ce puzzle, exigeant une réflexion stratégique et une planification minutieuse à chaque étape. Tout comme d'autres énigmes de cette époque, le jeu de Taquin invite les joueurs à explorer la dynamique complexe entre les éléments du jeu, offrant une expérience stimulante et gratifiante de résolution de problèmes.

II. CONCEPTION

Pour concevoir le jeu de Taquin, Nous avons commencé par établir une structure de données pour représenter l'état actuel du puzzle. Cette structure comprend une grille de 3x3 cases numérotées et une case vide pour les déplacements. Chaque case est représentée par une valeur numérique, permettant ainsi de suivre les déplacements des tuiles.

Ensuite, nous avons développé un ensemble de règles pour permettre aux joueurs de déplacer les tuiles en toute sécurité. Ces règles dictent les mouvements autorisés en fonction de la position de la case vide et des tuiles adjacentes. Par exemple, une tuile peut être déplacée vers la case vide si elle se trouve dans la même ligne ou colonne que la case vide.

Enfin, nous avons implémenté un algorithme de recherche efficace pour explorer l'ensemble des états possibles du puzzle et trouver la solution optimale. Cet algorithme utilise une approche heuristique pour guider la recherche vers les configurations les plus prometteuses, réduisant ainsi le temps de calcul nécessaire pour résoudre le puzzle.

III. MODELISATION

La modélisation du jeu de taquin consiste à définir un ensemble d'états possibles, des règles de transition et un état but à atteindre.

Espace d'États:

L'espace d'états du jeu de Taquin est représenté par différentes configurations de la grille 3x3, où chaque case contient un nombre spécifique ou est vide. Chaque état est défini par la disposition des tuiles dans la grille.

États Initial et But:

L'état initial du jeu de Taquin est donné par une configuration spécifique de la grille, où les tuiles sont mélangées de manière aléatoire avec une seule case vide. L'utilisateur définit l'état but en déterminant la configuration finale souhaitée de la grille.

Règles:

Les règles de déplacement dictent comment les tuiles peuvent être déplacées dans la grille. Une tuile adjacente à la case vide peut être déplacée vers cette dernière. Le déplacement est autorisé uniquement dans les directions verticales ou horizontales, en alignant les tuiles avec la case vide.

Règle 1: La case vide peut être déplacée vers le haut si une tuile est située au-dessus d'elle dans la grille. Ce déplacement consiste à échanger la position de la case vide avec celle de la tuile située au-dessus.

Règle 2: La case vide peut être déplacée vers le bas si une tuile est située en dessous d'elle dans la grille. Ce déplacement implique l'échange de position entre la case vide et la tuile située en dessous.

Règle 3: La case vide peut être déplacée vers la droite si une tuile est positionnée à sa droite dans la grille. Cette action consiste à intervertir la position de la case vide avec celle de la tuile à sa droite.

Règle 4: La case vide peut être déplacée vers la gauche si une tuile est située à sa gauche dans la grille. Ce déplacement implique l'échange de position entre la case vide et la tuile située à sa gauche.

Contrainte :

Pour garantir la validité des mouvements dans le jeu de Taquin, certaines contraintes doivent être respectées. Tout d'abord, la case vide ne peut être déplacée que dans les directions où une tuile est présente dans la grille. En d'autres termes, les règles de déplacement ne peuvent être appliquées que si une tuile adjacente à la case vide existe dans la direction spécifiée. De plus, il est interdit de déplacer une tuile en dehors des limites de la grille. Ces contraintes garantissent l'intégrité structurelle du puzzle et maintiennent sa résolubilité.

Objectif de la Modélisation:

L'objectif de la modélisation du jeu de Taquin est de trouver une séquence de mouvements permettant de passer de l'état initial à l'état but. Cela implique l'utilisation d'algorithmes de recherche A* et de techniques d'optimisation pour explorer l'espace d'états et trouver la solution optimale.

Implémentation des fonctions :

Extraction du premier élément de la liste : Cette stratégie est choisie pour réduire la complexité du parcours de la liste lors de l'extraction d'un nœud. En récupérant toujours le premier élément, nous évitons de parcourir toute la liste à chaque extraction, ce qui garantit une complexité constante ou linéaire en moyenne. Cette approche contribue à optimiser les performances de l'algorithme en minimisant le temps nécessaire pour extraire des nœuds.

Calcul de la fonction d'évaluation f sans stockage explicite : Bien que nous devions calculer la fonction d'évaluation f pour chaque nœud, nous ne stockons pas explicitement cette valeur. Cette approche est choisie pour réduire l'utilisation de la mémoire, car le stockage de la valeur f pour chaque nœud pourrait entraîner une surcharge de mémoire, surtout pour des problèmes de grande taille. En calculant f au besoin lors de l'évaluation des nœuds, nous économisons de l'espace mémoire tout en maintenant les performances de l'algorithme.

Insertion des nouveaux nœuds à la fin de la liste : C'est une stratégie avantageuse qui contribue à optimiser l'algorithme de tri utilisé dans A^* . En ajoutant les nouveaux nœuds à la fin de la liste, nous évitons de parcourir toute la liste à chaque insertion. Cette approche réduit donc le temps nécessaire pour insérer de nouveaux éléments, ce qui est particulièrement bénéfique lorsque la liste est déjà triée ou partiellement triée. En minimisant le coût de l'insertion, cette stratégie contribue à maintenir les performances globales de l'algorithme, car elle permet de conserver l'ordre de tri de la liste tout en réduisant la complexité temporelle de l'opération d'insertion. Ainsi, en insérant les nouveaux nœuds à la fin de la liste ouverte, nous optimisons le processus de tri utilisé dans l'algorithme A^* .

1. RECHERCHE INFORME A^*

Dans le contexte de la résolution du jeu de Taquin, la recherche informée, comme l'algorithme A^* , offre une approche plus efficace que la recherche systématique, qu'elle soit en largeur ou en profondeur. Contrairement à la recherche systématique qui explore systématiquement tous les états possibles, A^* utilise une fonction d'évaluation pour guider la recherche vers les états les plus prometteurs, réduisant ainsi le nombre total d'états à explorer.

L'algorithme A^* utilise une combinaison judicieuse de deux composantes dans sa fonction d'évaluation : le coût réel pour atteindre l'état actuel à partir de l'état initial ($g(n)$), et une estimation du coût restant pour atteindre l'état final depuis l'état actuel ($h(n)$). En additionnant ces deux composantes, A^* sélectionne le prochain état à explorer en fonction de sa promesse de mener à une solution optimale.

Ainsi, par rapport à la recherche en largeur, qui peut souvent générer et explorer un grand nombre d'états sans discernement, l'algorithme A^* est

capable de guider la recherche vers des chemins plus prometteurs, ce qui peut conduire à une résolution plus rapide et à une utilisation plus efficace des ressources.

Dans le cadre de notre algorithme de résolution du jeu de Taquin, nous avons introduit deux fonctions clés pour évaluer les états et guider notre recherche vers une solution optimale. Ces fonctions, $h(n)$ et $g(n)$, sont fondamentales pour l'algorithme A^* que nous utilisons pour trouver la solution optimale.

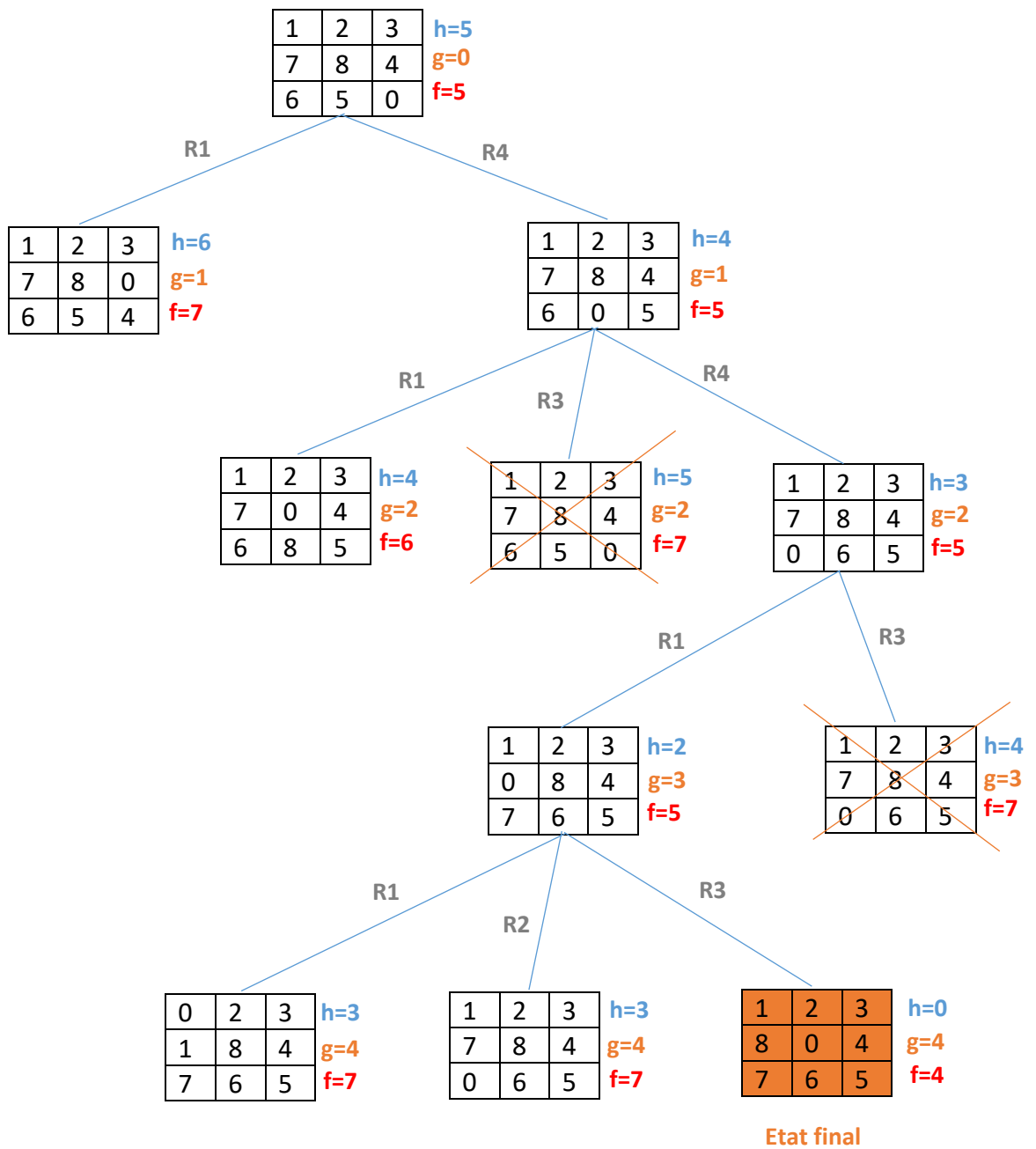
La fonction $h(n)$ représente une estimation heuristique du coût restant pour atteindre l'état final à partir de l'état actuel n . Cette fonction parcourt la grille de l'état actuel et compare chaque tuile avec sa position dans l'état final, calculant ainsi le nombre de tuiles qui ne sont pas à leur position finale. Cette valeur est utilisée comme estimation du nombre de mouvements nécessaires pour atteindre l'état final à partir de l'état actuel.

La fonction $g(n)$ représente le coût réel pour atteindre l'état actuel n à partir de l'état initial. Cette valeur est stockée dans la structure de données du nœud sous le champ g . À chaque fois qu'un nœud est généré, son coût réel est calculé en ajoutant 1 au coût du nœud parent. Cela garantit que chaque nœud a un coût correctement calculé pour représenter le nombre de mouvements nécessaires pour atteindre cet état à partir de l'état initial.

En combinant $g(n)$ et $h(n)$ dans la fonction d'évaluation $f(n)=g(n)+h(n)$, l'algorithme A^* sélectionne le prochain nœud à explorer en fonction de sa promesse de mener à une solution optimale. Ainsi, en utilisant $g(n)$ et $h(n)$, votre code parvient à évaluer efficacement les états et à guider la recherche vers la solution optimale.

2. GRAPHE DE RESOLUTION

Voici un graphe de resolution pour atteindre la solution final



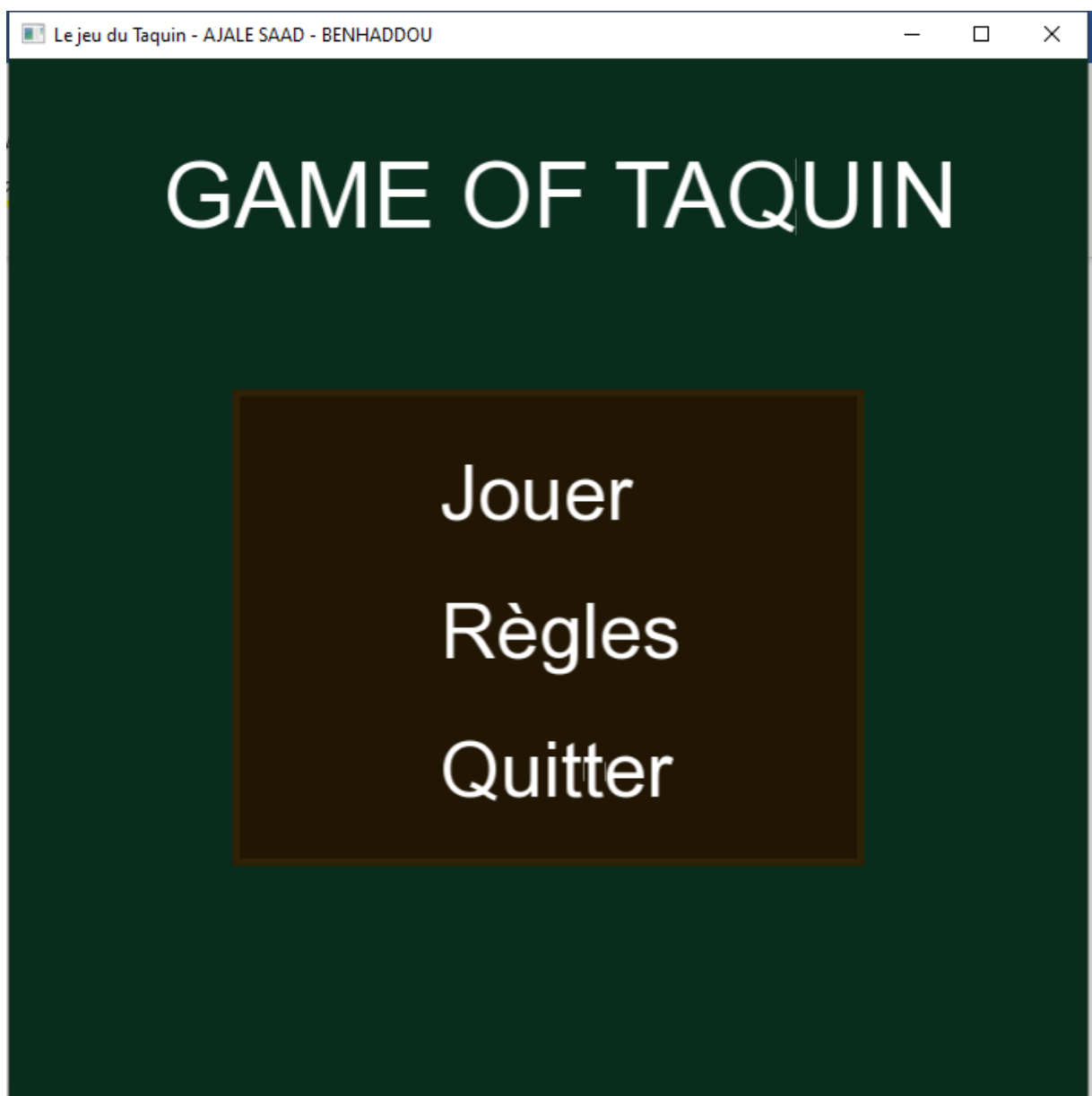
Nous avons affiché le chemin optimal depuis l'état initial jusqu'à l'état final sur la partie console à l'aide des fonctions d'affichage des nœuds implémentées dans le code.

```
Succes !  
le chemin est :  
 1  2  3  
 8  0  4  
 7  6  5  
-----  
  ^  
  ||  
 1  2  3  
 0  8  4  
 7  6  5  
-----  
  ^  
  ||  
 1  2  3  
 7  8  4  
 0  6  5  
-----  
  ^  
  ||  
 1  2  3  
 7  8  4  
 6  0  5  
-----  
  ^  
  ||  
 1  2  3  
 7  8  4  
 6  5  0  
-----  
Succes ! Arrer sur l'etat Final a partir de l'etat:  
 1  2  3  
 7  8  4  
 6  5  0  
-----  
Process returned 0 (0x0)   execution time : 0.021 s
```

IV. simulation:

Partie graphique:

- **Bibliothèque:** utilisé SFML pour C++
- **Menu:** 3 options
 - Jouer
 - Règle
 - Quitter



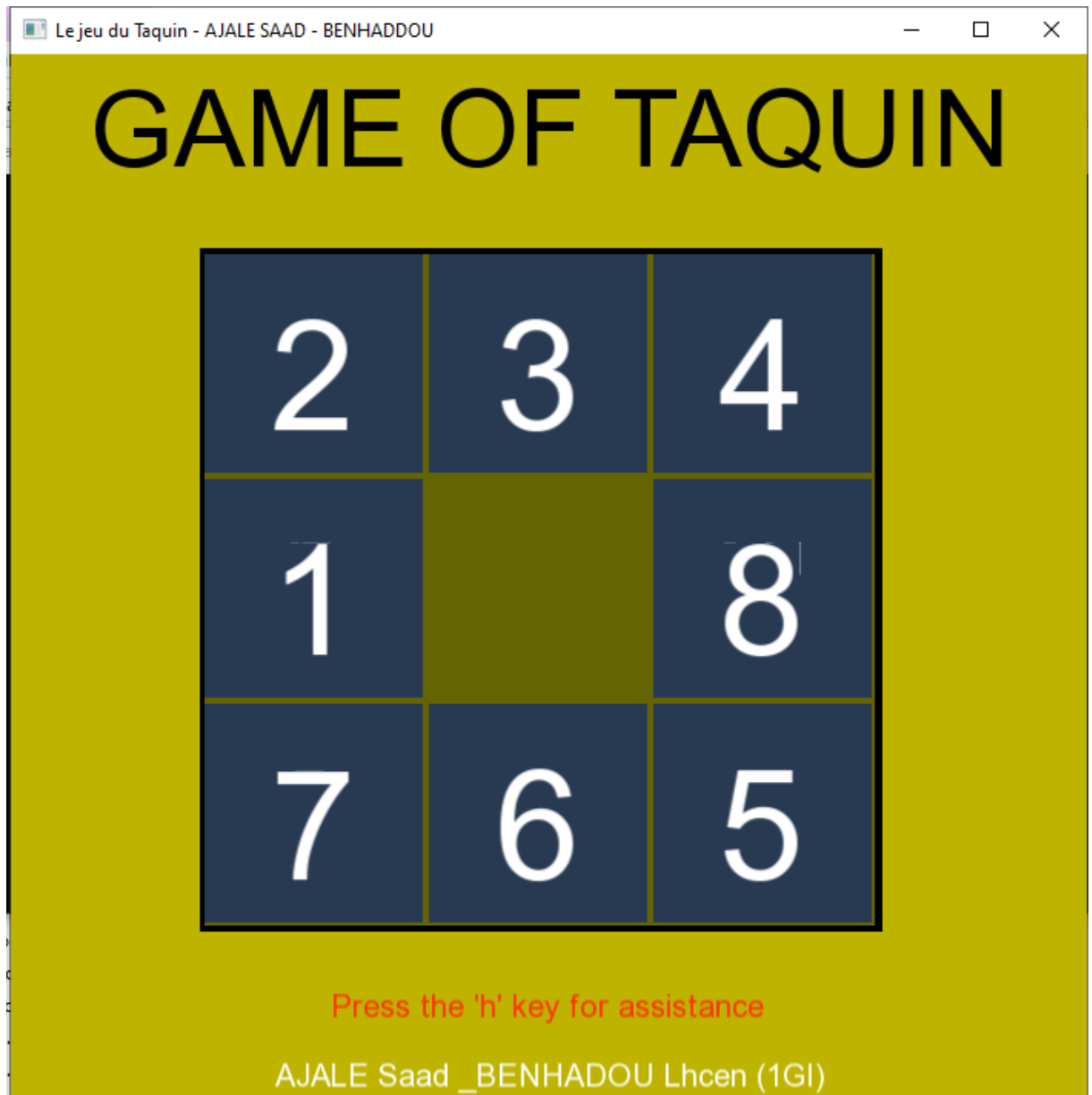
Règle:

- Fenêtre affichant les règles générales du jeu



Jouer:

- Fenêtre de jeu de taquin
- Déplacement de la case vide avec les flèches du clavier
- Le bouton 'h' sur clavier pour lancer la recherche A* (implémentée dans la partie console)



État final:

- Une fenêtre contenant deux boutons s'affiche
 - Quitter
 - Rejouer



Fonctionnement:

Le menu principal permet au joueur de choisir entre jouer, consulter les règles ou quitter le jeu.

L'option "Jouer" ouvre une fenêtre de jeu où le joueur peut déplacer la case vide à l'aide des flèches du clavier. La touche "h" permet d'activer la fonction de recherche d'étoile implémentée dans la partie console et par la suite l'utilisateur sera assisté par la machine.

Lorsque l'état final du jeu est atteint, deux boutons s'affichent : "Quitter" et "Rejouer". Le bouton "Quitter" ferme la partie et retourne au menu principal, tandis que le bouton "Rejouer" permet de recommencer la partie depuis le début.

La simulation a permis de tester et de valider les fonctionnalités de l'interface graphique du jeu. Les résultats sont satisfaisants et confirment que l'interface est intuitive et facile à utiliser.

Remarque :

Pour exécuter notre application, il suffit de lancer le fichier `main.exe` situé dans le dossier `./x64/Debug/main.exe`.

V. Problème et solution :

Problème:

Nous avons observé que certains états nécessitent un temps d'exécution **beaucoup plus long** que d'autres, même s'ils ne diffèrent que d'un ou deux coups. Ceci est illustré dans les images ci-jointes.

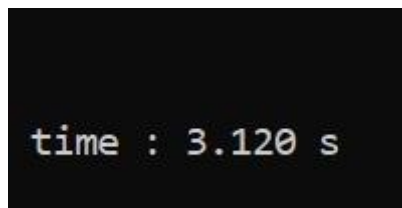
Cas1 :

```
int main() {  
    int tabini[TAILLE_GRILLE]  
        {8, 2, 4}, //{2, 0, 3}  
        {6, 1, 5},  
        {0, 7, 3}  
};
```



: 37.908 s

Cas2 :



time : 3.120 s

```
int main() {  
    int tabini[TAILLE_GRILLE]  
        {8, 2, 4}, //{2, 0, 3}  
        {6, 1, 5},  
        {7, 3, 0}  
};
```

Solution:

Pour remédier à ce problème, nous avons décidé de stocker les états déjà traités et leur chemin. Cela permettra de **réduire** l'effet de ce problème en **évitant de recalculer** les chemins pour les états déjà explorés.

Avantages de la solution:

- Réduction du temps d'exécution
- Amélioration de l'efficacité de l'algorithme
- Gain de mémoire

Conclusion:

La solution proposée permettra de **résoudre** le problème du temps d'exécution excessif pour certains états.

VI. CONCLUSION :

En conclusion, notre projet de jeu de taquin nous a plongés dans un voyage captivant à travers les méandres de l'algorithme A* et de la conception d'interfaces utilisateur. En développant des versions console et graphique avec SFML, nous avons été confrontés à des défis variés, de la gestion efficace des ressources à la création d'une expérience utilisateur immersive. Cette expérience nous a permis d'acquérir une compréhension approfondie de la résolution de problèmes algorithmiques. En fin de compte, ce projet nous a permis de consolider nos compétences techniques, tout en fournissant une application pratique et divertissante pour résoudre des casse-têtes de manière interactive.