Dans le cadre de ce devoir, il nous est demandé de créer un programme d'intelligence artificielle capable de résoudre un jeu de taquin de manière autonome. L'intelligence artificielle correspond à l'ensemble des théories et des techniques développant des programmes informatiques complexes capables de simuler certains traits de l'intelligence humaine, les jeux sont donc un excellent domaine d'application car ils constituent une véritable épreuve intellectuelle.

Pour nous aider dans un notre mission nous pouvons utiliser des heuristiques, soit des méthodes utilisées pour fournir une solution réalisable lorsqu'un problème est trop complexe, notamment en raison de l'explosion combinatoire (grand nombre de critères à prendre en compte).

Concernant le jeu en lui même, un taquin n × n est constitué de n2 – 1 carreaux à l'intérieur d'un carré pouvant contenir n2 jetons : on a donc une case vide et le but est de remettre dans l'ordre les carreaux à partir d'une configuration initiale quelconque, une forme de puzzle. Pour le programme, nous avons décidé de mettre en place la résolution du taquin 3x3 et 4x4.

I. Théorie

Avant de décrire le code plus en détail, nous allons commencé par voir son fonctionnement théorique. Nous avons décidé de mettre en place trois modes de résolution pour venir à bout de ce casse-tête. Dans cette partie théorique nous nous contenterons de développer uniquement la méthode A* car c'est celle demandée par le sujet

Parcours A*

L'algorithme de recherche A*, comme le parcours en profondeur ou en largeur, est un algorithme de recherche de chemins. Il permet de trouver une solution pour arriver d'un état initial à un état final. A* est en théorie optimisé pour nous permettre de trouver l'itinéraire le plus court pour ce chemin.

A* fonctionne alors de la manière suivante : le programme commence par un noeud donné de poids 0, on applique ensuite les différentes heuristiques implémentés. Une heuristique est un bout de code nous permettant de guider notre algorithme en calculant un coût pour chaque noeud à explorer. Dans le cas de notre programme, 7 heuristique sont misent en place.
Une fois ce travail fait et les différentes solutions "notées", A* est chargé de les trier en fonction de leur poids. Celui qui coûte le mois est choisi. S'il correspond à l'état final, l'algorithme renvoie le chemin par lequel nous sommes passé pour en arriver là, sinon l'algorithme recommence en expansant cette fois le nouveau noeud.

Pour réaliser ce projet notre code s'organise en 6 class bien distinctes. Cette décisions de fractionner les parties s'inscrit dans une volonté de notre part de simplifier au maximum la compréhension et la clarté au moment de la lecture du code.

Pour commencer, avant même la première class, nous avons importé les bibliothèques indispensables au bon fonctionnement du programme.
Puis 2 variables sont créées « data » et « etat_final », elles représentent respectivement l'état dans lequel se trouve la grille du Taquin au moment de son initialisation et le résultat attendu à la fin du jeu.

Passons maintenant à la première class : Node. Elle est centrale dans le fonctionnement du code puisqu'elle permet:

- **La création et le parcours des nœuds** : grâce à l'appel des fonctions « actions » qui permet comme son nom l'indique d'effectuer une action (ici une action entendue au sens de mouvement de pièces) et « path» qui garde dans un tableau le chemin emprunté par le programme lors du parcours récursif du graphe.

- **D'évaluer la solvabilité des grilles** (propriété « solved » que nous verrons plus tard)

Enfin la fonction d'évaluation : « function_eval » correspond à la distance parcourue ajoutée à la distance qu'il reste potentiellement à parcourir avant d'arriver à la solution finale et qui permet de donner une mesure de l'utilité d'un noeud, de savoir si ce dernier est proche de la solution ou non.

Continuons sur la deuxième class : Solver. Elle regroupe trois fonctions correspondant aux différents types de parcours d'arbre possible pour la résolution du problème.
Les trois fonctions on la même base,

Au dessus de cette fonction de base il existe une donc trois variants correspondant chacun à une manière différente de résoudre ce taquin :

- **A*** : prend tous les noeuds présents dans la liste *queu* puis les tris de manière croissante suivant la fonction d'évaluation vue en amont. La fonction prend ensuite le premier de cette liste (celui avec la FE, la plus petite de l'itération courante). Elle permet dans un premier temps de vérifier si le noeud est l'état final, si oui elle retourne le chemin qui a permis d'y accéder sinon elle expense un nouveau nouveau noeud qu'elle ajoute a la liste des noeuds vue si il n'y était pas deja

```
if node.solved:··
    return node.path·

for move, action in node.actions:
    child = Node(move(), node, action)

    if child.state not in seen:
        queue.append(child)
        seen.add(child.state)·
```

3 Parcours en largeur
Meme chose que a* sans le trie qui permet de prendre la plus petite valeur
2 Parcours en profondeur

La prochaine class que nous allons analyser est la class puzzle.Elle complete les fonctions initiées dans la class node on y retrouve

« solved » qui prend le noeud str et le compare la solotion attendu avec un range égale a « 1234567890 » qui est le résultat attendu

« Action » qui permet de faire se déplacer le 0 qui correspond a la case vide dans le 4 directions haut bas droite gauche en tenant compte de limite de la map grace a la condition

```
if r >= 0 and c >= 0 and r < len(self.data) and c < len(self.data) and self.data[r][c] == 0:
```

Pour ce faire action s'appuis un un deuxième fonction: « create_move » qui permet d'affecter un déplacement ajouter a un l

Image est une classe uniquement réservée a l'interface graphique ,elle permet d'afficher les images correspondantes a chaque cases du tableau et permet de les faire se déplacer en fonction des différents coups joués

Melangeur est notre 5 eme class elle pemet grace trois méthodes « isSolvable », « permutation » et « melange » de mélanger les pieces jusqu'à trouver une solution solvable .On l'utilise pour relancer une partie sans avoir a quitter le programme

Enfin nous arrivons a la fin du code et la dernière class: Jeu.Elle est divisée en deux parties
La premiere chargée d'initié le jeu et de mettre en forme les solutions en s'aidant des fonctions présente dans solver
La deuxièmes correspondant au main qui lance les grace aux touches assignées les différents mode de jeu précédemment codés

"Jeu" est donc la class bilan c'est qui réunie tout les element précédemment codé pour les faire fonctionner de cohort

STAT

Passons maintenant au test d'efficacité indispensable pour observer les performances des différentes heuristiques. Pour ces tests, nous avons essayé chaque stratégie sur 5 grilles différentes. Les résultats obtenus sont donc les moyennes des différents tests.
Les différents critères d'évaluation que nous avons choisis de mesurer sont : le nombre de coup, le nombre de noeuds vus, le nombre de noeuds expansés et le temps de résolution.

Représentation du nombre de coup en fonction des heuristiques

Commençons par le nombre de coup. Il représente le nombre de mouvement a effectué avant de résoudre le Taquin ; plus cette valeur est basse plus l'algorithme est efficace.
On remarque grâce au test effectué que le nombre de coup reste plutôt stable en fonction des différentes heuristiques malgré un léger avantage pour les *Manhattan* avec un coefficient de normalisation de 1.
Cependant on assiste à une véritable explosion de la valeur lors du parcours en profondeur, ce qui était a prévoir quand on sait que ce type de parcours n'est absolument pas optimisé pour ce genre de problème.

Représentation du nombre de noeuds vus

Le nombre de noeuds vus est un autre bon indicateur pour évaluer la performance de notre programme puisqu'il nous montre le nombre de noeud que compare l'algorithme avant d'arriver a la solution attendue. On peut voir que pour les *Manhattan* pondérés, le nombre noeuds vus est inversement proportionnel au nombre de coups joués, ce qui traduit le fait que plus le programme voit de noeuds plus il est capable de prendre une décision allant dans le sens d'une minimisation des mouvements de pièces.
De part leur fonctionnement plus « hasardeux », les parcours en largeurs et en profondeur voient énormément de noeuds ce qui n'est pas vraiment conseillé dans une optique d'optimisation du temps.

Représentation du nombre de noeuds expansé

Comme pour le test précédent, tester le nombre de noeuds expansés offre un bel aperçu des performances. Ici il nous sert à vérifier le test des noeuds vus, et il s'avère en effet qu'entre les deux résultats on observe un très forte similitude se qui atteste de la cohérence des mesures.

Représentation du temps d'exécution

Pour le temps d'exécution, il va sans dire qu'il s'agit d'un facteur essentiel à prendre en compte lorsque l'on veut tester notre programme. Ici, dans l'optique de protéger nos processeurs, les parcours en profondeur et en largeur n'ont pas d'interphase graphique ce qui biaise le résultat, aucun commentaire ne sera donc fait à leur sujet.

En ce qui concerne les autres heuristiques on peut voir que celles qui voient et expansent le plus de noeuds sont aussi celles qui prennent le moins de temps. *Mal placé* met plusieurs dizaines de secondes avant de terminer contre 0,001 seconde pour certaines autres heuristiques beaucoup moins gourmandes en noeuds. Mais, comme vu précédemment, plus un algorithme expanse et voit de possibilités différentes, plus il est efficace pour trouver une solution minimale en nombre de coups.

Pour conclure au sujet des statistiques et des tests, nous pouvons voir qu'en fonction des différentes heuristiques utilisées les résultats changent du tout au tout. Malgré cela, ces différences reste explicables et logiques c'est donc à nous d'équilibrer au mieux le programme en fonction des leviers qui sont à notre disposition afin d'obtenir le meilleur résultat en gardant un temps d'exécution raisonnable en limitant le nombre de noeuds vus et expansés.

Extension

Nous avons essayé, afin de répondre aux besoins de ce devoir, le plus de fonctionnalités possibles. Cependant, il reste des pistes a exploré comme la mise en place d'un mode de jeu « manuel » ou d'un mode « versus » contre la machine.
Il nous est aussi possible d'ajouter d'autres mesures de grilles puisqu'ici seul les taquin 3x3 et 4x4 sont pris en compte mais il est possible d'étendre à 5x5 6x6 etc.
Voilà selon nous les possibilités d'extension possibles pour avoir un programme complet qui permet à la fois la résolution du casse-tête mais aussi un moyen pour l'utilisateur de s'amuser.

Conclusion

Ainsi, la réalisation de ce programme nous a permis d'en apprendre davantage sur l'intelligence artificielle et plus précisément les heuristiques. En effet, ce concept et dans un cadre plus large, l'algorithme A*, que nous voyons pour la première fois, est indispensable à cette discipline en ce qu'il permet de réduire grandement et d'améliorer les algorithmes de recherche de chemin. C'est donc un exercice qui nous a été très formateur et nous servira dans la suite de notre cursus.
De plus, chaque travail de groupe nous permet de nous familiariser davantage avec l'environnement dans lequel il nous sera demandé d'évoluer à l'avenir, et d'apprendre à nous organiser de plus en plus dans la gestion des tâches dans le cadre de gros projets.