RAPPORT DE PROJET – IA41

IQ Puzzler Pro XXL Solver - 2025

UTBM

IA41 – Fondamentaux de l’IA

1. Introduction
    a. Contexte général et motivation
    b. Présentation du projet IQ Puzzler Pro
2. Rappel de l'Énoncé du Problème
    a. Description du jeu IQ Puzzler Pro
    b. Objectifs à atteindre dans le cadre du projet
3. Spécification (Formalisation) du Problème
    a. Définition des règles du jeu et des contraintes
    b. Représentation des données (plateau, pièces, positions)
    c. Formalisation des objectifs en termes de résolution algorithmique
4. Analyse du Problème
    a. Identification des difficultés (combinatoire, placement, etc.)
    b. Approches classiques pour résoudre des problèmes similaires
    c. Choix des stratégies pour aborder le problème
5. Méthodes Proposées
    a. Présentation de l'algorithme utilisé (exemple : backtracking, heuristiques)
    b. Explication de l'implémentation détaillée (structure du programme, choix des langages, modules utilisés)
    c. Ajout en annexe : listing complet et commenté du programme
6. Description des Situations Traitées
    a. Exemple(s) de configurations initiales du plateau
    b. Scénarios spécifiques testés avec le programme
7. Résultats Obtenus
    a. Présentation des solutions trouvées par le programme
    b. Analyse des performances (temps de calcul, efficacité)
    c. Limites identifiées dans les résultats
8. Difficultés Rencontrées
    a. Problèmes techniques (optimisation, bugs, etc.)
    b. Contraintes liées à la modélisation ou à l'implémentation
9. Améliorations Possibles
    a. Suggestions pour améliorer l'efficacité de la méthode de résolution
    b. Intégration d'autres approches ou algorithmes
10. Perspectives d'Ouverture
    a. Extensions possibles du sujet (variantes du jeu, application à d'autres domaines)
    b. Potentiel d'exploitation des solutions développées
11. Conclusion
    a. Résumé des principaux résultats et apprentissages
    b. Évaluation globale du projet
12. Annexes
    a. Listing complet et commenté du programme
    b. Images ou captures d'écran des situations testées
    c. Documentation technique supplémentaire

Introduction

Les jeux de logique comme le IQ Puzzler Pro captivent par leur capacité à mêler réflexion, stratégie et créativité. Ce projet s'est donné comme objectif de relever un défi ambitieux : concevoir un programme capable de résoudre automatiquement les casse-têtes proposés par ce jeu. Au-delà du plaisir de relever un défi algorithmique, il s'agit aussi d'une opportunité de plonger dans des problématiques complexes liées à la modélisation, à l'optimisation, et à l'efficacité des solutions.

L'idée centrale est simple : les joueurs doivent placer un ensemble de pièces sur un plateau en respectant certaines règles, jusqu'à trouver une solution valide. Mais derrière cette apparente simplicité se cache une véritable complexité combinatoire, avec des millions de configurations possibles à explorer. Dans ce cadre, notre rôle a été de formaliser le problème, d'en analyser les défis et de développer un programme informatique capable de proposer des solutions à ce casse-tête.

Ce rapport retrace toutes les étapes de ce projet, de l'analyse du problème aux résultats obtenus, en passant par les méthodes utilisées et les obstacles rencontrés. Il s'inscrit dans une démarche à la fois exploratoire et technique, où chaque défi rencontré a été l'occasion d'apprendre et de s'améliorer.

Contexte

Le IQ Puzzler Pro propose plusieurs modes de jeu, chacun impliquant des contraintes spécifiques. Certaines variantes demandent de remplir un plateau en 2D, tandis que d'autres explorent la dimension 3D avec des constructions en relief. Ces particularités rendent ce jeu particulièrement intéressant à résoudre par des algorithmes, car elles nécessitent une bonne gestion des contraintes spatiales et logiques.

À mesure que le nombre de pièces et de configurations possibles augmente, résoudre manuellement ces casse-têtes devient un défi de taille. L'idée de créer une solution automatisée est donc apparue comme une manière non seulement de faciliter cette tâche, mais aussi d'explorer des techniques avancées de résolution de problèmes combinatoires. Ce projet combine ainsi plusieurs domaines : algorithmique, programmation, et optimisation.

Langages de programmation

Python : pour la conception des algorithmes et l'implémentation principale du programme. (https://www.python.org/)

Bibliothèques et outils Python

NumPy : pour la gestion des tableaux multidimensionnels et la manipulation des données.

Matplotlib : pour la visualisation graphique des solutions.

Méthodes utilisées

Backtracking : pour explorer toutes les configurations possibles et trouver des solutions valides.

Programmation par contraintes : pour restreindre l'espace des solutions et accélérer la recherche.

Algorithmes heuristiques : pour améliorer les performances dans des cas où la recherche exhaustive est trop lente.

Environnements de développement

VS Code : pour l'écriture et le débogage du code.

Git : pour le suivi des versions du projet.

Documentation et collaboration

Mark down : pour rédiger et structurer la documentation du projet.

Sphinx (Python) : https://www.sphinx-doc.org/en/master/

Énoncé du Problème

Les défis se déclinent en plusieurs modes de jeu :

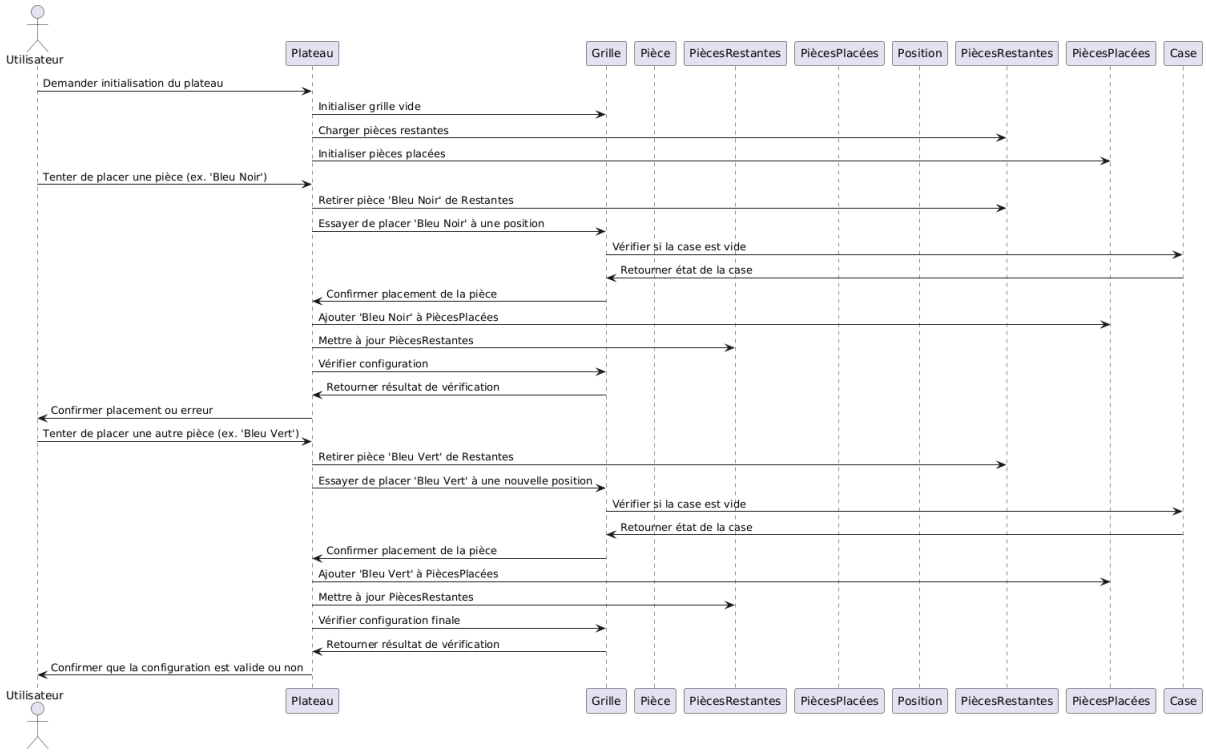
Mode 2D : Remplir une grille plane avec des pièces polygonales. Chaque pièce doit s'imbriquer parfaitement sans chevauchement ni espace vide.

Mode 3D : Construire une structure en relief en empilant les pièces sur une base donnée, en respectant les limites du plateau.

Chaque casse-tête a une ou plusieurs solutions possibles, mais trouver ces solutions devient rapidement complexe en raison de la multitude de combinaisons à explorer. La difficulté réside dans :

1. La gestion des pièces, chacune ayant une forme, une orientation et des contraintes spécifiques.
2. L'exploration exhaustive des configurations possibles sans tomber dans un délai de calcul prohibitif.

Problème posé



Créer un programme capable de résoudre automatiquement les casse-têtes du jeu en :

Formalisant le problème de manière informatique.

Développant un algorithme efficace pour trouver une ou plusieurs solutions valides pour un casse-tête donné.

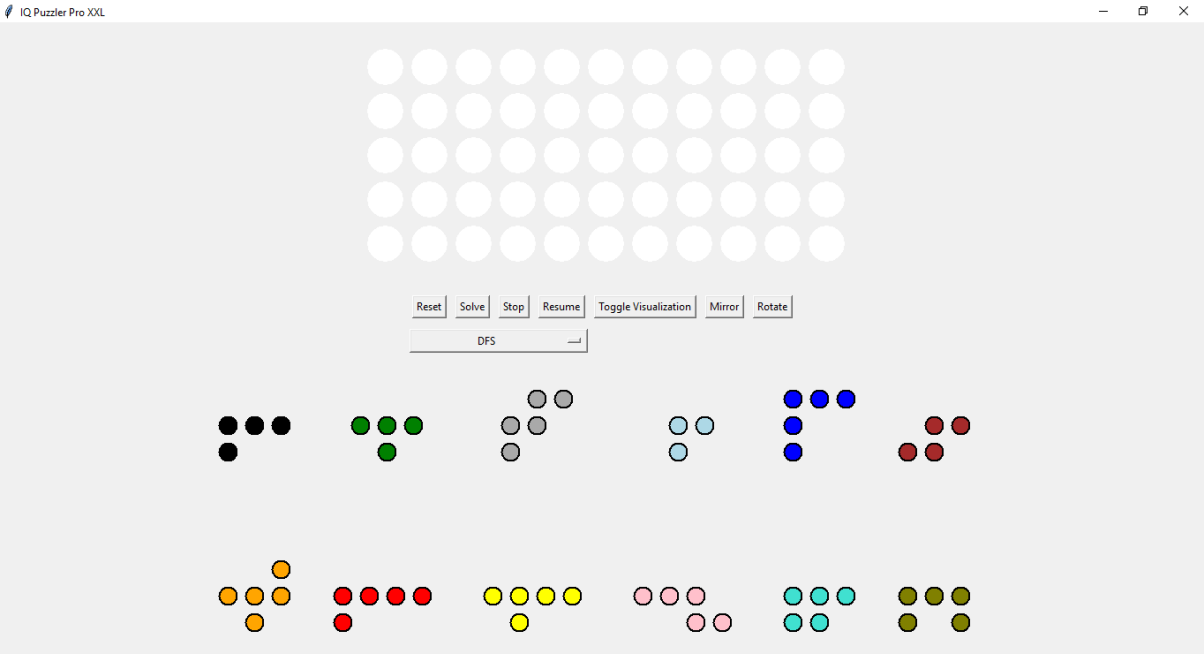
Permettant une visualisation claire des solutions obtenues.

Ce projet s'inscrit donc dans une logique de modélisation algorithmique, combinée à une exploration des techniques d'optimisation et de visualisation.

Spécification (Formalisation) du Problème

Le projet concerne la résolution d'un jeu de puzzle où l'objectif est de placer des pièces sur une grille tout en respectant certaines contraintes. Ce problème sera traité à l'aide d'une approche d'intelligence artificielle basée sur la recherche exhaustive et les algorithmes de satisfaction de contraintes.

Description du Jeu



Grille : La grille sur laquelle les pièces doivent être placées est une matrice de dimensions fixes (dans ce cas, 11x5). Elle est initialement vide, à l'exception des cases déjà occupées par des pièces de jeu.

Pièces : Le jeu comporte plusieurs types de pièces qui peuvent être placées sur la grille. Chaque pièce est représentée par une forme géométrique (par exemple, un tableau binaire ou une liste de coordonnées de cellules).

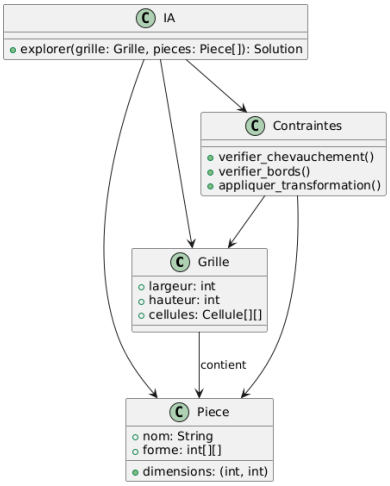
Contraintes de Placement :

Les pièces doivent être placées de manière à ne pas se chevaucher (c'est-à-dire, aucune cellule occupée par une pièce ne peut être occupée par une autre pièce).

Les pièces peuvent subir des transformations (rotations, symétries).

Certaines cases de la grille sont déjà occupées et ne peuvent pas être modifiées.

Chaque pièce a un nombre d'occurrences limité.

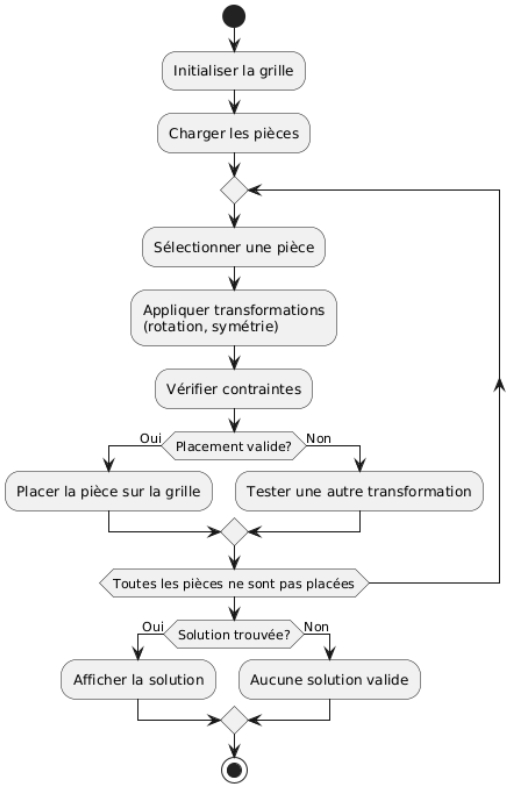


L'objectif est de placer toutes les pièces de manière valide sur la grille. L'IA doit explorer différentes configurations possibles en respectant les contraintes et trouver une solution optimale ou une solution valide (selon la définition du problème).

Grille : Une grille de taille n × m où chaque cellule peut être soit vide (0) soit occupée (non nul – dans ce projet une cellule contient une liste avec les spécifications de la pièce – nom, coordonnées, rotation, miroir etc...).

Pièces : Chaque pièce peut être représentée par un tableau binaire de taille p × q (par exemple, 3x3, 2x2). Chaque cellule de ce tableau peut être 0 (vide) ou occupée par la pièce.

Rotation et Symétrie : Chaque pièce peut avoir plusieurs variations obtenues par des rotations de 90°, 180°, 270° et des symétries horizontales et verticales.



Analyse du problème

L'espace de recherche est constitué de toutes les configurations possibles de placement des pièces sur la grille, en tenant compte des transformations (rotations et symétries). L'IA devra explorer cet espace en vérifiant chaque configuration pour s'assurer qu'elle respecte les contraintes.

L'algorithme de recherche (par exemple, recherche exhaustive, backtracking) peut être amélioré par l'utilisation de heuristiques pour accélérer la recherche de solutions. Une heuristique possible est de toujours placer la pièce qui occupe le plus d'espace ou celle qui a le moins de possibilités de placement restantes.

Les contraintes principales de ce problème sont :

Contrainte de non-chevauchement : Les pièces ne doivent pas se superposer sur la grille. Cela doit être vérifié lors de chaque tentative de placement.

Contrainte de transformation : Chaque pièce peut subir des rotations et des symétries. L'IA doit tester chaque variation de la pièce avant de la placer.

Contrainte de placement valide : Les pièces ne peuvent être placées que dans des zones spécifiques de la grille, et certaines cases sont déjà occupées par des pièces fixes.

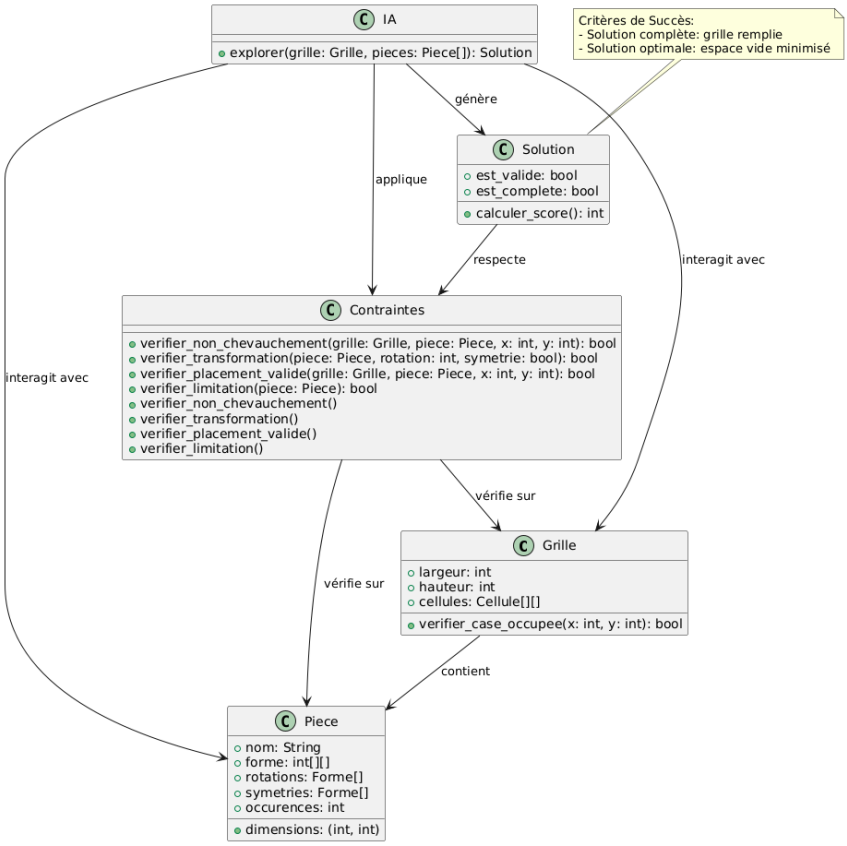
Limitation des pièces : Le nombre de pièces est fixe, et chaque pièce ne peut être utilisée qu'un nombre limité de fois.

La solution consiste à trouver un placement valide pour toutes les pièces, c'est-à-dire une configuration où :

Toutes les pièces sont placées correctement sur la grille.

Aucune pièce ne chevauche une autre pièce.

Les pièces sont placées en tenant compte des transformations possibles (rotations et symétries).



Critères de Succès

Solution complète : La grille est complètement remplie de manière valide avec toutes les pièces placées.

Solution optimale : Si plusieurs solutions sont possibles, la solution qui maximise certains critères (par exemple, minimiser l'espace vide) peut être considérée comme optimale.

Méthodes Proposées

Le problème sera abordé par un algorithme de backtracking ou de brute-force, combiné à des heuristiques pour réduire l'espace de recherche.

Brute Force

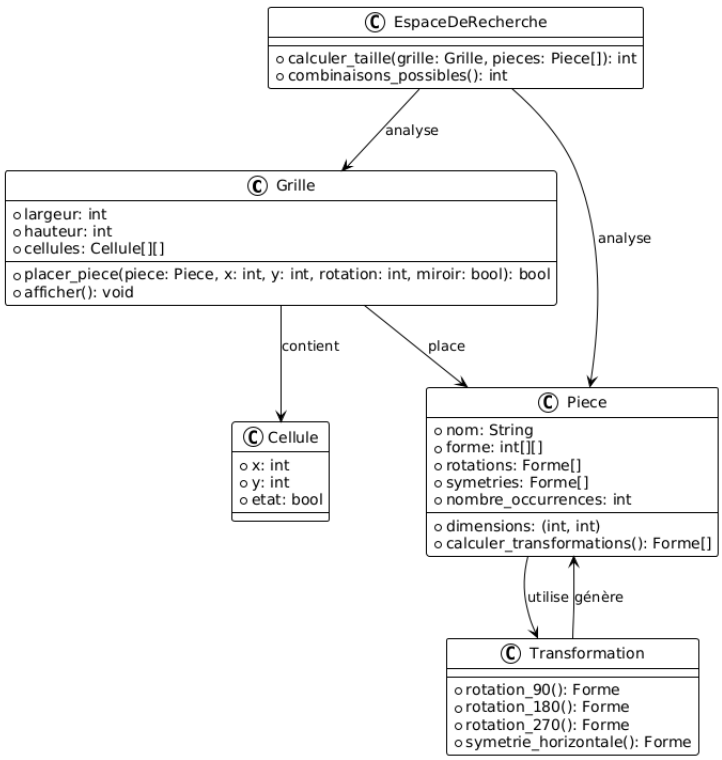
Pour chaque pièce, essayer toutes les variations (rotations et symétries).

Vérifier les placements possibles sur la grille.

Tester la validité de chaque placement (pas de chevauchement, respect des contraintes de placement).

Si une solution est trouvée, l'algorithme se termine.

Si une solution n'est pas trouvée après avoir testé toutes les configurations possibles, l'algorithme échoue.



Backtracking

Le backtracking améliorera l'algorithme de brute force en abandonnant plus tôt les chemins qui ne peuvent pas conduire à une solution valide. L'algorithme procédera ainsi :

Choisir une pièce.

Tester toutes les transformations possibles (rotations et symétries) et placements valides.

Avancer si un placement valide est trouvé.

Si une solution complète est trouvée, arrêter.

Si un chemin mène à une impasse, revenir en arrière (backtracking).

Évaluation de la Solution

L'évaluation peut se faire sur la base des critères suivants :

Temps de calcul : Le temps nécessaire pour trouver une solution valide, particulièrement pour les grilles de grandes tailles.

Qualité de la solution : En fonction de l'optimisation du placement des pièces, par exemple, en minimisant l'espace vide restant.

Précision : Le taux de solutions trouvées par rapport aux solutions possibles (si le problème admet plusieurs solutions valides).

Algorithmes DFS (Depth-First Search) et Q-learning

Définition de l'État et de la Pièce

Grille :
La grille peut être représentée par une liste bidimensionnelle de n lignes et m colonnes, où

chaque élément de la grille est soit 0 (vide) soit occupé. Une grille de taille 10×10 serait donc une liste de 10 sous-listes, chacune contenant 10 éléments (0 ou list).

grid = [[1, 0, 0, 0, 0, 1, 0, 1, 0, 0], [0, 1, 0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 1, 0, 1, 0, 0, 0, 0, 0], ...]

Pièce :

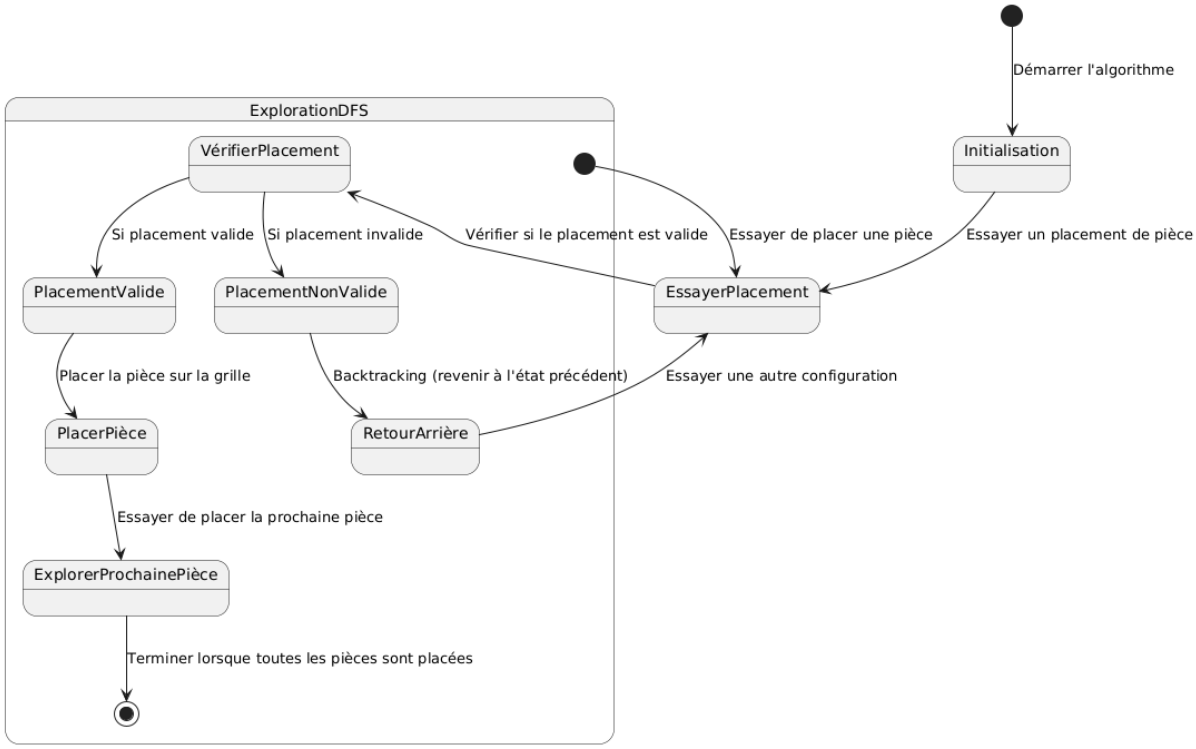
Une pièce est un sous-ensemble de cases de la grille, représentée par une forme géométrique. Elle peut subir des transformations (rotations et symétries). Chaque pièce peut être définie par ses coordonnées relatives et les transformations possibles.

piece = [(0, 0), (0, 1), (1, 0), (1, 1)]

DFS (Depth-First Search)

L'algorithme DFS est un algorithme de recherche dans lequel on explore le plus profondément possible un chemin avant de revenir en arrière lorsque l'on atteint une impasse. Ce type de recherche est particulièrement adapté aux problèmes où il faut explorer de manière exhaustive toutes les solutions possibles.

L'algorithme DFS pour ce problème de placement de pièces peut être décrit comme suit :

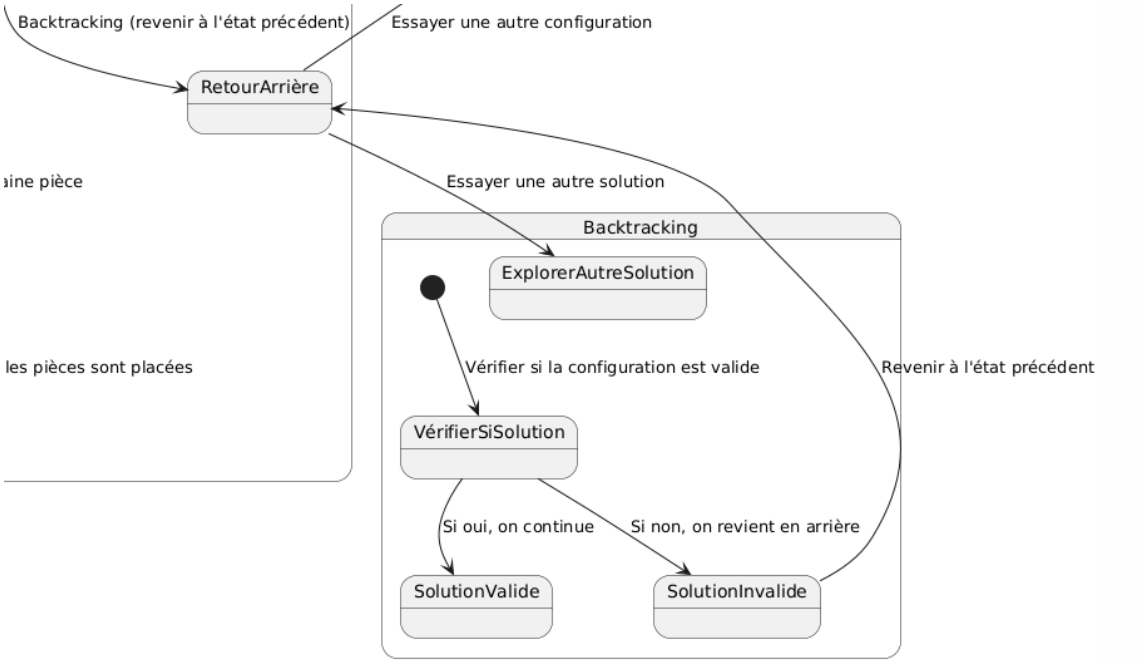


Initialisation :
La recherche commence à partir d'une grille vide (ou d'un état partiellement rempli). On essaie de placer une pièce dans la grille en respectant les contraintes (absence de chevauchement et respect des zones autorisées).

Exploration :
À chaque étape, on tente de placer une pièce de toutes les façons possibles (en

considérant toutes les transformations possibles de la pièce) dans la grille. Pour chaque tentative, on explore récursivement en essayant de placer la pièce suivante.

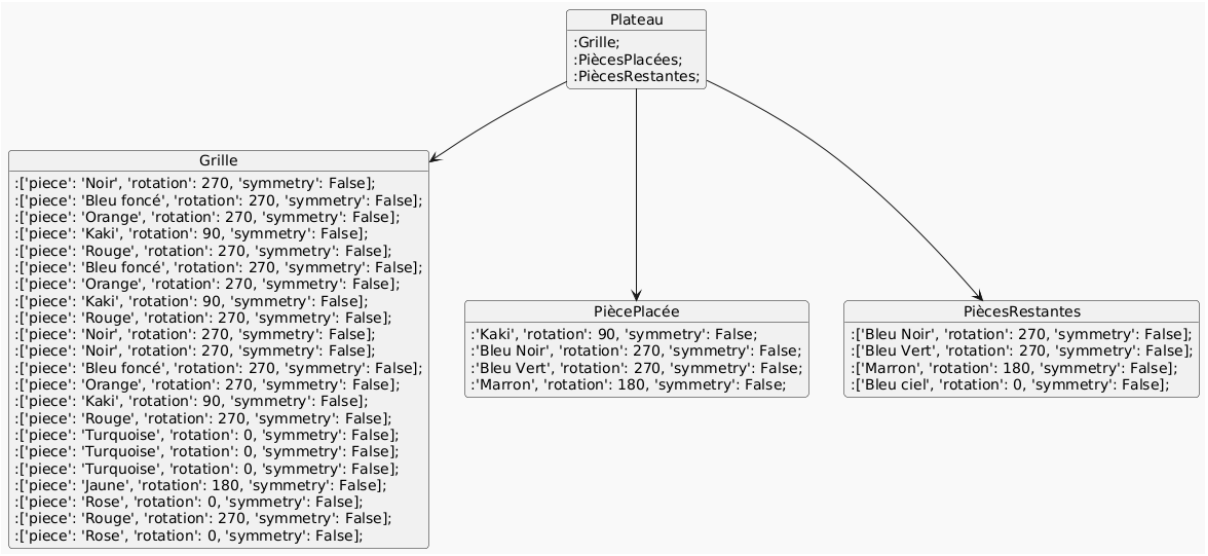
Backtracking :
Si une configuration de pièces ne mène pas à une solution valide (si une pièce ne peut pas être placée ou si la grille est pleine), l'algorithme revient à l'état précédent et essaie une autre configuration.



Condition de fin :
L'algorithme termine lorsqu'une solution valide est trouvée (toutes les pièces sont placées de manière correcte) ou lorsqu'il n'y a plus de configurations possibles à tester.

Complexité de DFS

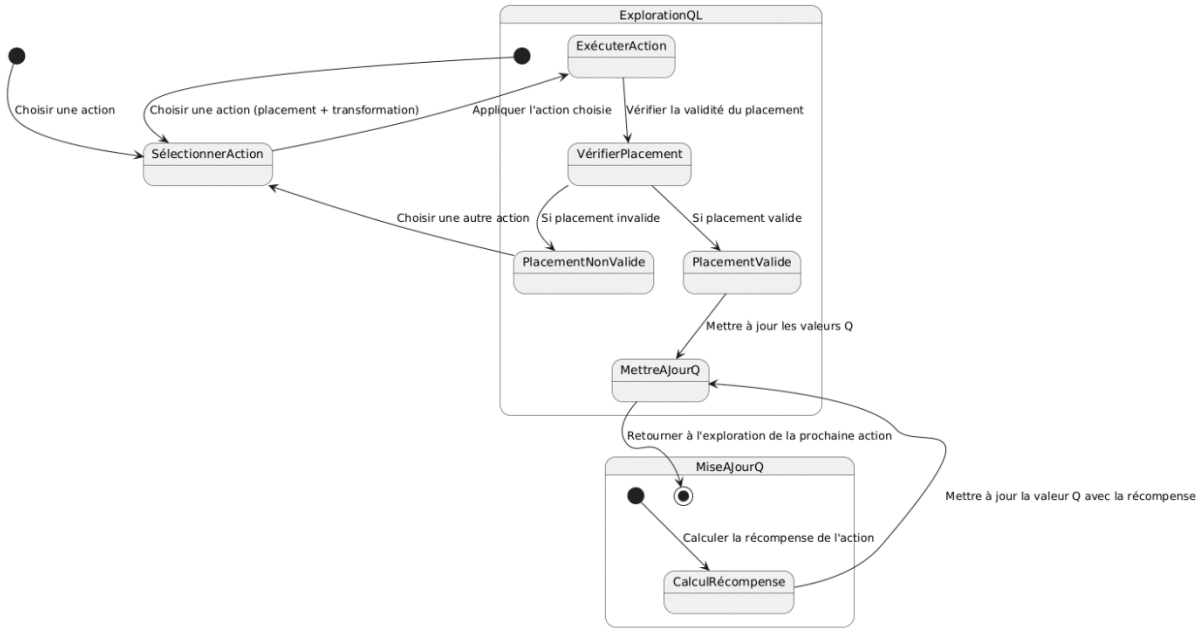
Espace de recherche : Le nombre de configurations possibles pour un placement de pièce dépend du nombre de pièces, de la taille de la grille et des transformations possibles. Chaque pièce peut avoir plusieurs orientations possibles, et pour chaque orientation, elle peut être placée à plusieurs positions.



Complexité en temps : La complexité de DFS dans ce contexte est exponentielle, car il faut explorer un grand nombre de configurations pour trouver une solution valide.

Q-Learning (Apprentissage par Renforcement)

Dans le contexte de la résolution du problème de placement de pièces, Q-learning peut être utilisé pour apprendre à l'IA la meilleure façon de placer les pièces en fonction des récompenses.



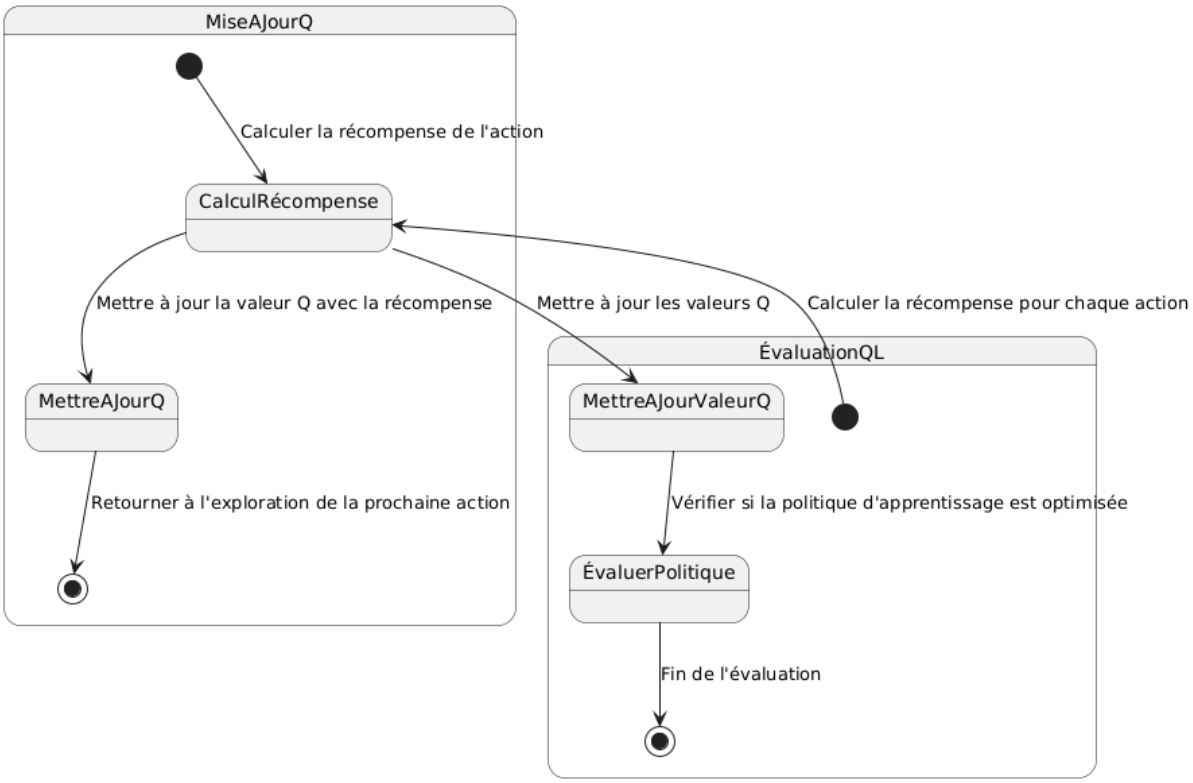
Initialisation :
L'agent (IA) commence avec une grille vide et aucune information sur l'espace de recherche. Un tableau Q est utilisé pour stocker les valeurs d'action (représentant les décisions sur le placement des pièces) pour chaque état.

Exploration :
À chaque étape, l'IA sélectionne une action (placer une pièce à un emplacement particulier et

appliquer une transformation) en utilisant une stratégie d'exploration (par exemple, epsilon-greedy).

Récompenses :
L'agent reçoit une récompense pour chaque action en fonction de la validité de la configuration résultante. Par exemple, une récompense positive si une pièce est placée correctement, et une récompense négative en cas de chevauchement.

Mise à jour des valeurs Q :
Après chaque action, la valeur Q est mise à jour en fonction de la récompense reçue, et l'agent ajuste sa politique pour maximiser la récompense cumulative sur le long terme.



Terminaison :
L'apprentissage continue jusqu'à ce que l'IA soit capable de placer toutes les pièces sur la grille de manière optimale (ou jusqu'à un nombre d'itérations fixé).

Comparaison des Performances

DFS :

Avantages : Simple à implémenter, garantissant une solution si elle existe.

Inconvénients : Très coûteux en termes de temps de calcul et d'espace mémoire. Peut-être impraticable pour de grandes grilles ou un grand nombre de pièces.

Q-learning :

Avantages : Plus flexible, capable d'apprendre une politique optimale sur le long terme. Il peut s'améliorer au fur et à mesure de l'apprentissage, et potentiellement plus rapide dans des scénarios complexes.

Inconvénients : Nécessite un grand nombre d'itérations pour converger vers une solution optimale. Plus difficile à implémenter et à ajuster.

Comparaison des Temps de Calcul

DFS : Le temps de calcul peut devenir très élevé pour des grilles de grande taille, car l'algorithme explore toutes les configurations possibles de manière exhaustive.

Q-learning : Le temps de calcul dépend du nombre d'itérations d'apprentissage et de la taille de l'espace d'états. Bien qu'il ne garantisse pas une solution immédiate, il peut s'améliorer au fil du temps.

Description des Situations Traitées

Le projet vise à résoudre le problème de placement optimal de pièces géométriques dans une grille donnée, en tenant compte des contraintes (définies précédemment).

Deux approches ont été utilisées pour traiter ce problème :

DFS (Depth-First Search) : une recherche exhaustive pour explorer toutes les configurations possibles.

Q-learning : un algorithme d'apprentissage par renforcement permettant à l'IA d'apprendre une politique optimale pour placer les pièces.

Plus un algorithme de brut force qui converge vers une solution également.

Résultats Obtenus

La convergence du DFS est assez lente et couteuse en ressources car la taille de la grille est grande. Pour le Q-Learning, l'implémentation d'une mémoire permet d'enregistrer les informations déjà traitées et le score qui leur est associé. Cela améliore l'apprentissage, mais la convergence est lente.

Enfin l'algorithme de brut force (simple ici, car des versions plus avancées, utilisant notamment des bibliothèques python telles que dlx) assure une solution, mais est aussi couteux en ressource. Pour améliorer un peu la computation, des systèmes de threading ont été implémentés.

Figure a: DFS avec heuristiques - grand espace occupé par la pièce en priorité

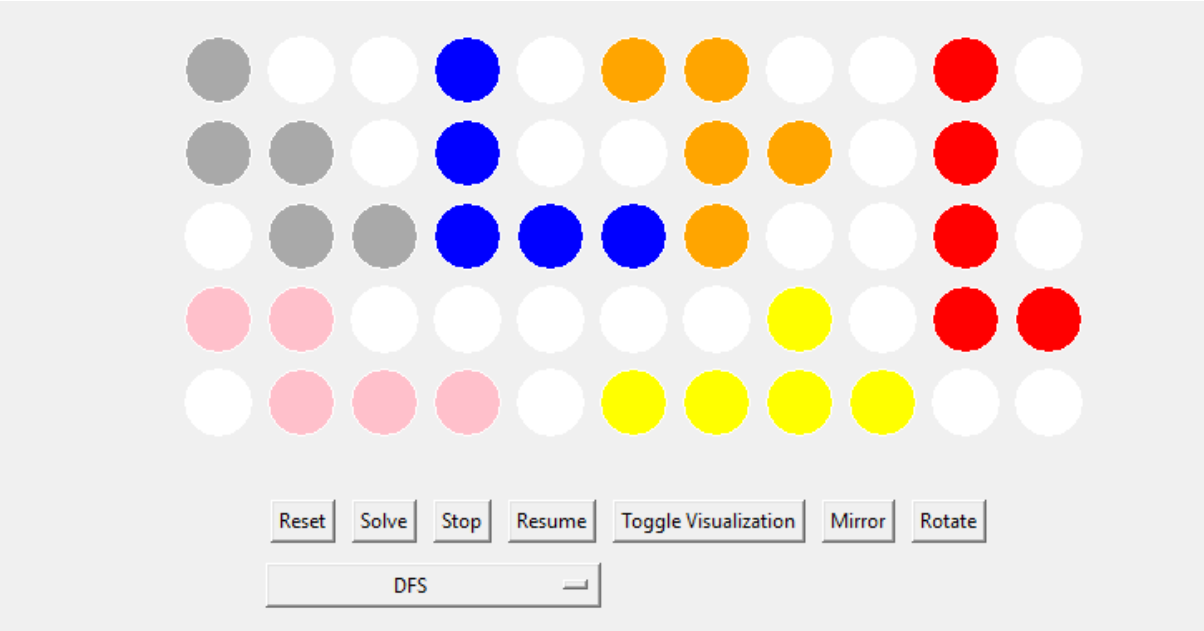


Figure b: Q-Learning agent avec mémoire et multithreading

Difficultés Rencontrées

DFS :

Explosion combinatoire : La taille de l'espace de recherche a considérablement ralenti les calculs pour des grilles et des pièces complexes.

Gestion de la mémoire : L'utilisation excessive de la pile a conduit à des erreurs de dépassement de mémoire.

Q-learning :

Taille de l'espace d'états : La gestion d'un tableau Q pour des grilles de grande taille a nécessité des simplifications (quantification des états).

Convergence lente : Le modèle a nécessité un réglage minutieux des hyperparamètres (taux d'apprentissage, epsilon) pour converger efficacement.

Complexité des Transformations :
Implémenter toutes les transformations possibles des pièces tout en évitant les configurations invalides a été un défi majeur.

Améliorations Possibles

DFS :

Implémenter une version avec heuristiques (par exemple, A* ou recherche en largeur limitée) pour explorer d'abord les configurations les plus prometteuses.

Réduire la complexité en limitant les positions ou transformations testées.

Q-learning :

Utiliser des algorithmes plus avancés comme le Deep Q-Learning (DQN) pour gérer de grands espaces d'états en utilisant des réseaux neuronaux.

Optimiser les hyperparamètres automatiquement avec des méthodes de recherche comme l'algorithme bayésien.

Transformations des Pièces :

Précalculer les transformations possibles des pièces pour réduire le coût computationnel pendant l'exécution.

Introduire une meilleure gestion des symétries pour éviter de tester des configurations redondantes.

Optimisation des Performances :

Implémenter l'algorithme en parallèle (par exemple, avec GPU ou threads) pour accélérer le traitement.

Réduire la granularité des états pour diminuer la taille du tableau Q.

Conclusion

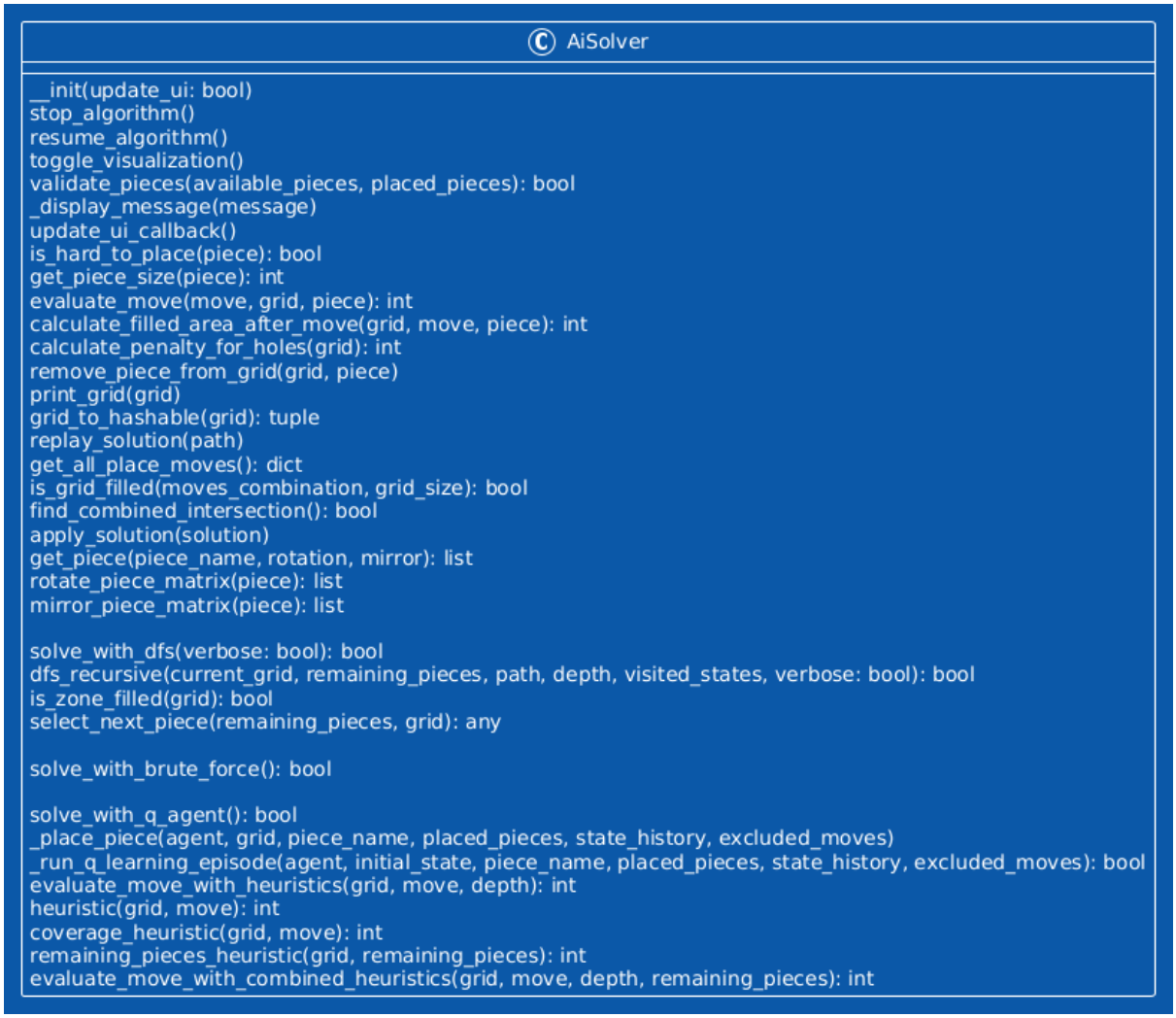
Ce projet a permis de modéliser un problème complexe de placement géométrique et d'explorer deux approches pour le résoudre : DFS et Q-learning.

- DFS a montré son efficacité pour des petits espaces de recherche mais est limité par sa complexité exponentielle.
- Q-learning s'est avéré prometteur pour des grilles de grande taille, bien qu'il nécessite un entraînement conséquent.

L'application future de méthodes plus avancées, telles que le Deep Q-Learning, pourrait offrir des solutions encore plus performantes. Ce projet a également mis en lumière l'importance de la formalisation des états et des actions dans les algorithmes d'intelligence artificielle.

ANNEXE 1 :

Méthodes principales de ai_solver.py – descriptions générées par Copilot.



- **__init__(update_ui=True)**
  Constructeur de la classe. Il initialise l'état de l'algorithme et les paramètres de l'interface utilisateur, optionnellement pour activer la mise à jour de l'interface après chaque modification.
- **stop_algorithm()**
  Arrête l'exécution de l'algorithme en cours. Cette méthode est utilisée pour suspendre les actions de l'agent lorsqu'il est en pause.
- **resume_algorithm()**
  Reprend l'exécution de l'algorithme après qu'il ait été arrêté par un appel à stop_algorithm().

- **toggle_visualization()**
  Permet d'activer ou de désactiver la visualisation de l'algorithme en cours (par exemple, afficher ou masquer les étapes du placement des pièces).
- **validate_pieces(available_pieces, placed_pieces): bool**
  Vérifie si les pièces disponibles et celles déjà placées sont compatibles avec l'état actuel du plateau. Renvoie True si la validation est réussie, sinon False.
- **_display_message(message)**
  Affiche un message dans l'interface utilisateur (peut être utilisé pour des informations de débogage ou de statut).
- **update_ui_callback()**
  Fonction de rappel pour mettre à jour l'interface utilisateur après un changement dans l'algorithme, comme un placement de pièce ou un changement d'état.
- **is_hard_to_place(piece): bool**
  Vérifie si une pièce est difficile à placer sur le plateau, probablement en raison de sa forme ou de l'espace disponible.
- **get_piece_size(piece): int**
  Renvoie la taille d'une pièce, généralement la surface occupée sur le plateau.
- **evaluate_move(move, grid, piece): int**
  Évalue la qualité d'un mouvement en cours en fonction de la configuration du plateau et de la pièce. Cela pourrait être basé sur la minimisation des pénalités, des espaces vides, etc.
- **calculate_filled_area_after_move(grid, move, piece): int**
  Calcule la zone remplie après l'application d'un mouvement. Cela peut être utile pour estimer l'impact d'un mouvement sur l'état global du plateau.
- **calculate_penalty_for_holes(grid): int**
  Calcule une pénalité pour les trous créés sur le plateau. Les trous sont des espaces vides qui ne peuvent pas être remplis par d'autres pièces.
- **remove_piece_from_grid(grid, piece)**
  Retire une pièce de la grille après qu'elle ait été placée, pour annuler un mouvement ou tester une configuration différente.
- **print_grid(grid)**
  Affiche l'état actuel de la grille dans un format lisible pour l'utilisateur, utile pour le débogage ou l'analyse.
- **grid_to_hashable(grid): tuple**
  Convertit l'état de la grille en une structure de données hashable (comme un tuple) pour faciliter la comparaison d'états dans les algorithmes de recherche (par exemple, DFS ou Q-learning).
- **replay_solution(path)**
  Rejoue la solution en suivant les étapes stockées dans le chemin path, montrant ainsi l'évolution de l'algorithme de placement des pièces.
- **get_all_place_moves(): dict**
  Renvoie un dictionnaire de tous les mouvements possibles pour placer les pièces restantes sur le plateau.
- **is_grid_filled(moves_combination, grid_size): bool**
  Vérifie si la grille est complètement remplie après une combinaison de mouvements. Cela peut être utilisé comme critère d'arrêt dans des algorithmes comme DFS.
- **find_combined_intersection(): bool**
  Cherche une intersection combinée dans les configurations actuelles des pièces ou des mouvements, pour une optimisation des placements.

- **apply_solution(solution)**
  Applique la solution trouvée à la grille, plaçant toutes les pièces aux positions appropriées.
- **get_piece(piece_name, rotation, mirror): list**
  Récupère une pièce en fonction de son nom, de sa rotation et de son miroir. Cela permet de créer des versions différentes de la même pièce.
- **rotate_piece_matrix(piece): list**
  Effectue une rotation de la matrice représentant la pièce, en modifiant son orientation.
- **mirror_piece_matrix(piece): list**
  Applique un miroir à la matrice de la pièce, retournant la pièce dans l'axe vertical ou horizontal.
- **solve_with_dfs(verbose=True): bool**
  Résout le problème en utilisant l'algorithme de recherche en profondeur (DFS). Renvoie True si une solution a été trouvée, sinon False.
- **dfs_recursive(current_grid, remaining_pieces, path, depth, visited_states, verbose=True): bool**
  Fonction récursive utilisée dans DFS pour explorer les différentes configurations possibles du plateau en ajoutant des pièces à chaque étape.
- **is_zone_filled(grid): bool**
  Vérifie si une zone spécifique du plateau est remplie ou si elle contient des espaces vides.
- **select_next_piece(remaining_pieces, grid): any**
  Sélectionne la prochaine pièce à placer en fonction des pièces restantes et de la configuration du plateau.
- **solve_with_brute_force(): bool**
  Résout le problème en utilisant une approche brute-force, où chaque combinaison de placements est testée jusqu'à ce qu'une solution soit trouvée.
- **solve_with_q_agent(): bool**
  Résout le problème en utilisant un agent Q-learning, qui apprend à placer les pièces en fonction des récompenses reçues à chaque mouvement.
- **_place_piece(agent, grid, piece_name, placed_pieces, state_history, excluded_moves)**
  Place une pièce sur le plateau en utilisant l'agent Q-learning, en mettant à jour l'état de la grille et en enregistrant l'historique des états et des mouvements exclus.
- **_run_q_learning_episode(agent, initial_state, piece_name, placed_pieces, state_history, excluded_moves): bool**
  Exécute une session de Q-learning, où l'agent explore différentes actions et met à jour ses connaissances pour améliorer ses performances.
- **evaluate_move_with_heuristics(grid, move, depth): int**
  Évalue un mouvement en utilisant des heuristiques qui prennent en compte la profondeur de l'exploration et l'état du plateau.
- **heuristic(grid, move): int**
  Calcule une valeur heuristique pour un mouvement donné, afin d'estimer sa qualité en termes de progression vers la solution.
- **coverage_heuristic(grid, move): int**
  Évalue un mouvement en fonction de sa capacité à couvrir l'espace vide du plateau, minimisant ainsi les zones inutilisées.
- **remaining_pieces_heuristic(grid, remaining_pieces): int**
  Évalue un mouvement en fonction du nombre et du type de pièces restantes à placer sur le plateau.

➲ **evaluate_move_with_combined_heuristics(grid, move, depth, remaining_pieces): int**
Combine plusieurs heuristiques pour évaluer la qualité d'un mouvement, en prenant en compte la configuration actuelle du plateau, la profondeur de l'exploration et les pièces restantes.

ANNEXE 2 :

Diagramme UML du projet

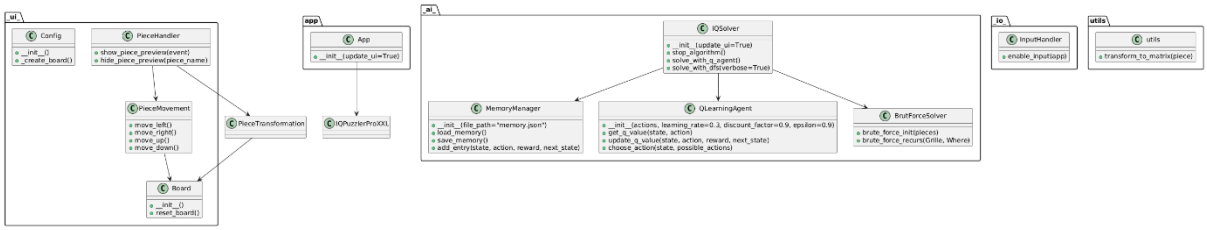


Figure c : Diagramme simplifié du projet

Figure d : Diagramme complet