RAPPORT

TP AI

ASKRI MARYEM

PROBLEMATIQUE DU TAQUIN

Le problème du Taquin est un casse-tête classique qui consiste en un puzzle à n cases disposées dans une grille carrée, avec une case vide servant à déplacer les autres cases pour atteindre une configuration donnée. Ce problème est un exemple typique de problème de recherche d'état, dans lequel il s'agit de trouver une séquence d'actions (déplacements) permettant de passer de l'état initial à l'état objectif.t

DESCRIPTION DU PROBLÈME

LLe Taquin se compose d'une grille de taille N×N, où N est un entier représentant la dimension de la grille (par exemple, N=3 pour une grille 3x3 ou N=4 pour une grille 4x4). La grille contient des cases numérotées de 1 à N^2 - 1, et une case vide représentée par le nombre 0. L'objectif est de déplacer les cases sur la grille en utilisant la case vide, en respectant les mouvements permutables (haut, bas, gauche, droite), afin de réaliser une configuration cible, généralement appelée "état objectif".

Objectifs du problème

L'objectif est de développer des algorithmes de recherche capables de résoudre ce problème du Taquin pour diverses tailles de grilles (par exemple, 3×3 et 4×4) en trouvant une séquence de mouvements pour transformer l'état initial en état objectif. Les algorithmes doivent être comparés en fonction de :

- **Le nombre de nœuds développés : Combien d'états intermédiaires doivent être générés avant de trouver la solution ?**
- **Le temps d'exécution : Combien de temps prend l'algorithme pour trouver la solution ?**
- **Optimalité de la solution : L'algorithme trouve-t-il la solution optimale, c'est-à-dire le nombre minimum de mouvements pour atteindre l'état objectif ?**

MÉTHODES DE RECHERCHE

Pour résoudre ce problème, trois techniques de recherche seront comparées :

1. **Recherche en largeur (BFS) :**
   - **L'algorithme explore tous les nœuds à chaque niveau de la recherche avant de passer au niveau suivant.**
   - **Il garantit de trouver la solution optimale, mais peut être très coûteux en mémoire pour des grilles de grande taille.**
2. **Recherche en profondeur (DFS) :**
   - **L'algorithme explore les nœuds de manière récursive en profondeur avant de revenir en arrière pour explorer d'autres options.**
   - **Bien qu'il puisse être plus rapide en termes de nombre de nœuds développés, il ne garantit pas une solution optimale et peut être inefficace pour des problèmes complexes.**

- **Algorithme A* (A-star) :**
  - **Utilise une fonction heuristique pour guider la recherche vers les solutions les plus prometteuses.**
  - **L'heuristique utilisée ici est la distance de Manhattan, qui mesure la distance entre chaque case et sa position dans l'état objectif. L'algorithme A* est souvent plus efficace que BFS et DFS car il combine exploration exhaustive et information heuristique pour privilégier les chemins les plus prometteurs.**

Algorithme de BFS:

Initialiser une file queue contenant l'état initial et un tableau vide pour stocker le chemin.
Initialiser un ensemble de visites visited pour éviter de revisiter les mêmes états.
Tant que la file n'est pas vide :
Dépiler le premier état et le chemin de la file.
Si l'état actuel est l'état objectif, retourner le chemin.
Sinon, générer les voisins de l'état et les ajouter à la file si ce sont des états non visités.
Si aucun chemin n'est trouvé, retourner "Pas de solution".

AAlgorithme de DFS:

1. **Initialiser une pile stack contenant l'état initial et un tableau vide pour stocker le chemin.**
2. **Initialiser un ensemble de visites visited pour éviter de revisiter les mêmes états.**
3. **Tant que la pile n'est pas vide :**
   - **Dépiler le dernier état et le chemin de la pile.**
   - **Si l'état actuel est l'état objectif, retourner le chemin.**
   - **Sinon, générer les voisins de l'état et les ajouter à la pile s'ils ne sont pas visités.**
4. **Si aucun chemin n'est trouvé, retourner "Pas de solution".**

Algorithme A*:

1. **Initialiser une file de priorité open_list avec l'état initial et un coût de 0.**
2. **Initialiser un ensemble de visites closed_list pour les états déjà explorés.**
3. **Tant que la file de priorité n'est pas vide :**
   - **Dépiler l'état avec le coût total le plus faible de la liste.**
   - **Si l'état actuel est l'état objectif, retourner le chemin.**
   - **Sinon, générer les voisins de l'état et les ajouter à la liste de priorité avec un coût estimé.**
4. **Si aucun chemin n'est trouvé, retourner "Pas de solution".**

CODE BFS

```python
# BFS
Tabnine | Edit | Test | Explain | Document
def bfs(initial_state, goal_state):
    queue = [(initial_state, [])]  # (état, chemin)
    visited = set()
    visited.add(tuple(tuple(row) for row in initial_state))
    nodes_expanded = 0  # Compteur pour les nœuds développés

    while queue:
        state, path = queue.pop(0)
        nodes_expanded += 1  # Incrémente le nombre de nœuds développés

        if is_goal(state, goal_state):
            return path, nodes_expanded

        for neighbor in get_neighbors(state):
            state_tuple = tuple(tuple(row) for row in neighbor)
            if state_tuple not in visited:
                visited.add(state_tuple)
                queue.append((neighbor, path + [neighbor]))

    return None, nodes_expanded
```

CODE DFS

```python
# DFS
Tabnine | Edit | Test | Explain | Document
def dfs(initial_state, goal_state):
    stack = [(initial_state, [])]  # (état, chemin)
    visited = set()
    visited.add(tuple(tuple(row) for row in initial_state))
    nodes_expanded = 0  # Compteur pour les nœuds développés

    while stack:
        state, path = stack.pop()
        nodes_expanded += 1  # Incrémente le nombre de nœuds développés

        if is_goal(state, goal_state):
            return path, nodes_expanded

        for neighbor in get_neighbors(state):
            state_tuple = tuple(tuple(row) for row in neighbor)
            if state_tuple not in visited:
                visited.add(state_tuple)
                stack.append((neighbor, path + [neighbor]))

    return None, nodes_expanded
```

CODE A*

```python
# A*
Tabnine | Edit | Test | Explain | Document
def a_star(initial_state, goal_state):
    open_list = []
    heappush(open_list, (0 + manhattan_distance(initial_state, goal_state), 0, initial_state, []))  # (f, g, état, chemin)
    closed_list = set()
    closed_list.add(tuple(tuple(row) for row in initial_state))
    nodes_expanded = 0  # Compteur pour les nœuds développés

    while open_list:
        f, g, state, path = heappop(open_list)
        nodes_expanded += 1  # Incrémente le nombre de nœuds développés

        if is_goal(state, goal_state):
            return path, nodes_expanded

        for neighbor in get_neighbors(state):
            state_tuple = tuple(tuple(row) for row in neighbor)
            if state_tuple not in closed_list:
                closed_list.add(state_tuple)
                f_new = g + 1 + manhattan_distance(neighbor, goal_state)
                heappush(open_list, (f_new, g + 1, neighbor, path + [neighbor]))

    return None, nodes_expanded
```

COMPARAISON

- **BFS garantit une solution optimale, mais peut être extrêmement coûteuse en termes de mémoire et de temps, en particulier pour des tailles de grilles plus grandes comme le 4×44 \times 44×4.**
- **DFS est plus rapide en termes de développement de nœuds, mais ne garantit pas de solution optimale et peut mener à des résultats sous-optimaux ou à un temps d'exécution élevé.**
- **A* combine le meilleur des deux mondes en explorant de manière plus efficace et en garantissant la solution optimale tout en réduisant le nombre de nœuds développés par rapport à BFS.**

En résumé, A* est généralement la méthode la plus efficace en termes de nombre de nœuds développés et de temps d'exécution, tandis que BFS et DFS ont leurs propres avantages et inconvénients selon la taille de la grille et les contraintes de performance.

RESULTAT

```
Recherche en largeur (BFS)...
Solution trouvée en BFS ! Nombre de nœuds développés : 3357
Temps d'exécution (BFS) : 0.0623 secondes

Recherche en profondeur (DFS)...
Solution trouvée en DFS ! Nombre de nœuds développés : 55982
Temps d'exécution (DFS) : 365.4279 secondes

Algorithme A*...
Solution trouvée en A* ! Nombre de nœuds développés : 92
Temps d'exécution (A*) : 0.0165 secondes
```