LAB 4 - Heuristic Search Techniques

Auteur : Badr TAJINI - Introduction à l'IA - ESIEE-IT - 2024/2025

Dans ce chapitre, nous allons apprendre les techniques de recherche heuristique. Les techniques de recherche heuristique sont utilisées pour explorer l'espace des solutions afin de trouver des réponses. La recherche est effectuée en utilisant des heuristiques qui guident l'algorithme de recherche. Cette heuristique permet à l'algorithme d'accélérer le processus, qui autrement prendrait beaucoup de temps pour aboutir à la solution.

À la fin de ce chapitre, vous connaîtrez les éléments suivants :

- Qu'est-ce que la recherche heuristique ?
- Recherche non informée vs recherche informée
- Problèmes de satisfaction de contraintes
- Techniques de recherche locale
- Recuit simulé
- Construction d'une chaîne de caractères à l'aide de la recherche gloutonne
- Résolution d'un problème avec contraintes
- Résolution du problème de coloration de régions
- Construction d'un résolveur de puzzle 8-pièces
- Construction d'un résolveur de labyrinthe

La recherche heuristique est-elle de l'intelligence artificielle ?

Dans le *Chapitre 2*, *Cas d'utilisation fondamentaux pour l'intelligence artificielle*, nous avons appris les cinq tribus définies par Pedro Domingos. L'une des tribus les plus "anciennes" est la tribu *symboliste*. Pour moi du moins, ce fait n'est pas surprenant. En tant qu'êtres humains, nous essayons de trouver des règles et des motifs dans tout. Malheureusement, le monde est parfois chaotique et tout ne suit pas des règles simples.

C'est la raison pour laquelle d'autres tribus ont émergé pour nous aider lorsque nous n'avons pas un monde ordonné. Cependant, lorsque nos espaces de recherche sont petits et que le domaine est limité, l'utilisation d'heuristiques, de satisfaction de contraintes et d'autres techniques présentées dans ce chapitre est utile pour cet ensemble de problèmes. Ces techniques sont utiles lorsque le nombre de combinaisons est relativement restreint et que l'explosion combinatoire est limitée. Par exemple, résoudre le problème du voyageur de commerce est simple avec ces techniques lorsque le nombre de villes est d'environ 20. Si nous

essayons de résoudre le même problème pour n = 2000, d'autres techniques devront être utilisées qui n'explorent pas l'espace complet et donnent seulement une approximation du résultat.

Qu'est-ce que la recherche heuristique ?

La recherche et l'organisation des données sont des sujets importants au sein de l'intelligence artificielle. Il existe de nombreux problèmes qui nécessitent de rechercher une réponse dans le domaine des solutions. Il y a de nombreuses solutions possibles à un problème donné et nous ne savons pas lesquelles sont correctes. En organisant efficacement les données, nous pouvons rechercher des solutions rapidement et efficacement.

Souvent, il existe tellement d'options possibles pour résoudre un problème donné qu'aucun algorithme unique ne peut être développé pour trouver une solution optimale définitive. De plus, passer en revue chaque solution n'est pas possible car cela serait prohibitivement coûteux. Dans de tels cas, nous nous appuyons sur une règle empirique qui nous aide à réduire la recherche en éliminant les options qui sont évidemment incorrectes. Cette règle empirique est appelée **heuristique**. La méthode consistant à utiliser des heuristiques pour guider la recherche est appelée **recherche heuristique**.

Les techniques heuristiques sont puissantes car elles accélèrent le processus. Même si l'heuristique n'est pas capable d'éliminer certaines options, elle aidera à ordonner ces options de sorte que de meilleures solutions soient susceptibles d'apparaître en premier. Comme mentionné précédemment, les recherches heuristiques peuvent être coûteuses en termes de calcul. Nous allons maintenant apprendre comment nous pouvons prendre des *raccourcis* et *élaguer* l'arbre de recherche.

Recherche non informée vs recherche informée

Si vous êtes familier avec l'informatique, vous avez peut-être entendu parler de techniques de recherche telles que la **Recherche en Profondeur** (**DFS**), la **Recherche en Largeur** (**BFS**) et la **Recherche de Coût Uniforme** (**UCS**). Ce sont des techniques de recherche couramment utilisées sur des graphes pour atteindre la solution. Ce sont des exemples de recherches non informées. Elles n'utilisent aucune information préalable ou règle pour éliminer certains chemins. Elles vérifient tous les chemins plausibles et choisissent le chemin optimal.

Une recherche heuristique, en revanche, est appelée une **recherche informée** car elle utilise des informations préalables ou des règles pour éliminer des chemins inutiles. Les techniques de

recherche non informées ne prennent pas en compte l'objectif. Les techniques de recherche non informées recherchent à l'aveugle et n'ont aucune connaissance a priori de la solution finale.

Dans le problème du graphe, les heuristiques peuvent être utilisées pour guider la recherche. Par exemple, à chaque nœud, nous pouvons définir une fonction heuristique qui renvoie un score représentant l'estimation du coût du chemin depuis le nœud actuel jusqu'à l'objectif. En définissant cette fonction heuristique, nous informons la technique de recherche de la bonne direction pour atteindre l'objectif. Cela permettra à l'algorithme d'identifier quel voisin mènera à l'objectif.

Nous devons noter que les recherches heuristiques ne trouvent pas toujours la solution la plus optimale. Cela est dû au fait que nous n'explorons pas toutes les possibilités et que nous nous appuyons sur une heuristique. La recherche est garantie de trouver une bonne solution en un temps raisonnable, ce qui est ce que nous attendons d'une solution pratique. Dans les scénarios réels, nous avons besoin de solutions rapides et efficaces. Les recherches heuristiques fournissent une solution efficace en arrivant rapidement à une solution raisonnable. Elles sont utilisées dans les cas où les problèmes ne peuvent pas être résolus autrement ou prendraient beaucoup de temps à résoudre. Une autre façon d'*élaguer* l'arbre est d'exploiter les contraintes inhérentes aux données. Dans la section suivante, nous apprendrons davantage de techniques d'élagage en tirant parti de ces contraintes.

Problèmes de satisfaction de contraintes

Il existe de nombreux problèmes qui doivent être résolus sous contraintes. Ces contraintes sont essentiellement des conditions qui ne peuvent pas être violées pendant le processus de résolution du problème.

Ces problèmes sont appelés **Problèmes de Satisfaction de Contraintes** (**CSP**).

Pour une compréhension intuitive, examinons rapidement un exemple de Sudoku. Le Sudoku est un jeu où nous ne pouvons pas avoir le même nombre deux fois sur une ligne horizontale, une ligne verticale ou dans le même carré. Voici un exemple de plateau de Sudoku :

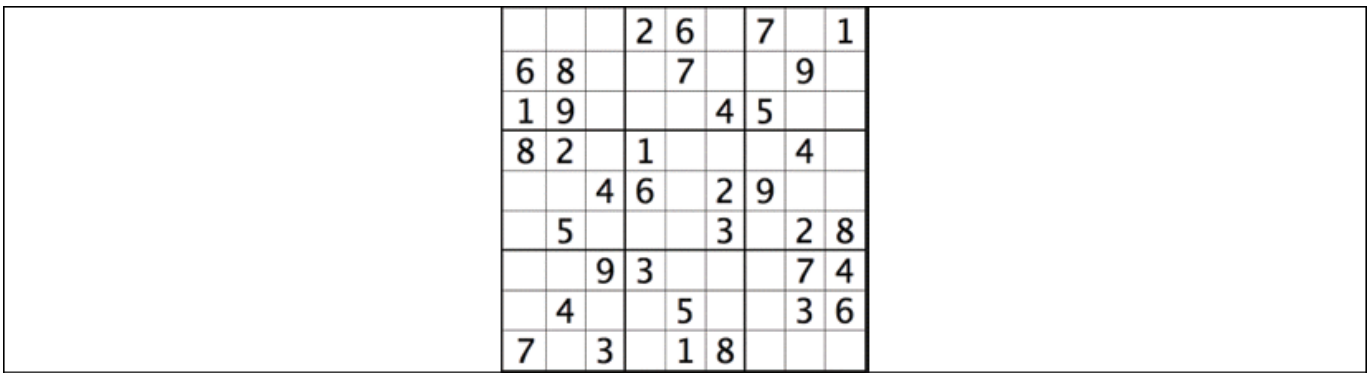

Figure 1 : Exemple d'un plateau de Sudoku

En utilisant la satisfaction de contraintes et les règles du Sudoku, nous pouvons rapidement déterminer quels nombres essayer et quels nombres ne pas essayer pour résoudre le puzzle. Par exemple, dans ce carré :

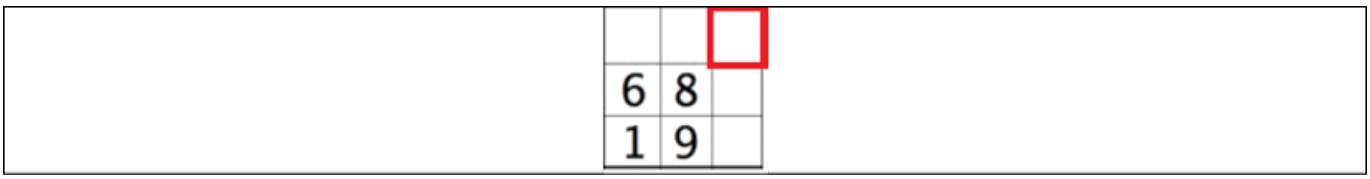

Figure 2 : Considération d'un problème dans Sudoku

Si nous n'utilisons pas de CSP, une approche par force brute consisterait à essayer toutes les combinaisons de nombres dans les cases puis à vérifier si les règles sont respectées. Par exemple, notre première tentative pourrait consister à remplir toutes les cases avec le numéro 1, puis à vérifier le résultat.

En utilisant le CSP, nous pouvons élaguer les tentatives avant de les essayer.

Passons en revue ce que nous pensons que le nombre devrait être pour la case surlignée en rouge. Nous savons que le nombre ne peut pas être 1, 6, 8 ou 9 car ces nombres existent déjà dans le carré. Nous savons également qu'il ne peut pas être 2 ou 7 car ces nombres existent sur la ligne horizontale. Nous savons également qu'il ne peut pas être 3 ou 4 car ces nombres sont déjà sur la ligne verticale. Il ne reste donc que la seule possibilité : 5.

Les CSP sont des problèmes mathématiques définis comme un ensemble de variables devant satisfaire certaines contraintes. Lorsque nous arrivons à la solution finale, les états des variables doivent respecter toutes les contraintes. Cette technique représente les entités impliquées dans un problème donné comme une collection d'un nombre fixe de contraintes sur les variables. Ces variables doivent être résolues par des méthodes de satisfaction de contraintes.

Ces problèmes nécessitent une combinaison d'heuristiques et d'autres techniques de recherche pour être résolus en un temps raisonnable. Dans ce cas, nous utiliserons des techniques de

satisfaction de contraintes pour résoudre des problèmes sur des domaines finis. Un domaine fini consiste en un nombre fini d'éléments. Puisque nous traitons des domaines finis, nous pouvons utiliser des techniques de recherche pour arriver à la solution. Pour une clarté supplémentaire sur les CSP, nous allons maintenant apprendre comment utiliser des techniques de recherche locale pour résoudre les problèmes de CSP.

Techniques de recherche locale

La recherche locale est une manière de résoudre un CSP. Elle optimise continuellement la solution jusqu'à ce que toutes les contraintes soient satisfaites. Elle met à jour itérativement les variables jusqu'à ce que nous arrivions à la destination. Ces algorithmes modifient la valeur à chaque étape du processus qui nous rapproche de l'objectif. Dans l'espace des solutions, la valeur mise à jour est plus proche de l'objectif que la valeur précédente. C'est pourquoi elle est connue sous le nom de recherche locale.

Un algorithme de recherche locale est un type d'algorithme de recherche heuristique. Ces algorithmes utilisent une fonction qui calcule la qualité de chaque mise à jour. Par exemple, elle peut compter le nombre de contraintes qui sont violées par la mise à jour actuelle ou voir comment la mise à jour affecte la distance par rapport à l'objectif. Cela est appelé le coût de l'affectation. L'objectif global de la recherche locale est de trouver la mise à jour à coût minimal à chaque étape.

Le **hill climbing** est une technique de recherche locale populaire. Il utilise une fonction heuristique qui mesure la différence entre l'état actuel et l'objectif. Lorsque nous commençons, il vérifie si l'état est l'objectif final. Si c'est le cas, il s'arrête. Sinon, il sélectionne une mise à jour et génère un nouvel état. Si ce nouvel état est plus proche de l'objectif que l'état actuel, il en fait l'état actuel. Sinon, il l'ignore et continue le processus jusqu'à ce qu'il vérifie toutes les mises à jour possibles. Il grimpe essentiellement la colline jusqu'à atteindre le sommet.

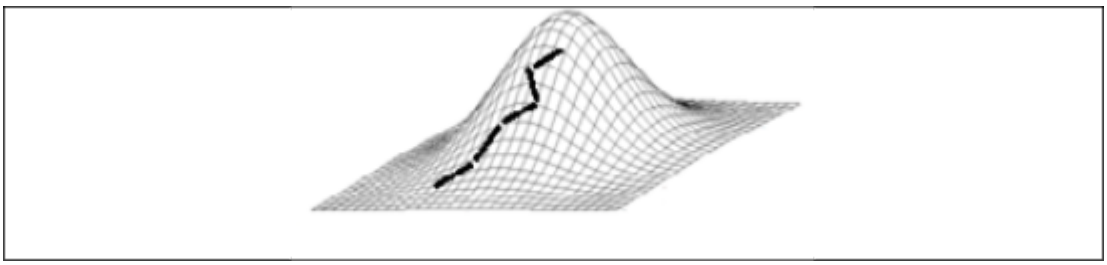

Figure 3 : Hill climbing

Recuit simulé (Simulated annealing)

Le recuit simulé est un type de recherche locale, ainsi qu'une technique de recherche stochastique. Les techniques de recherche stochastique sont largement utilisées dans divers domaines, tels que la robotique, la chimie, la fabrication, la médecine et l'économie. Les algorithmes stochastiques sont utilisés pour résoudre de nombreux problèmes du monde réel : nous pouvons par exemple optimiser la conception d'un robot, déterminer les stratégies de synchronisation pour le contrôle automatisé dans les usines et planifier le trafic. Le recuit simulé est une variante de la technique de l'escalade de colline. L'un des principaux problèmes de l'escalade de colline est qu'elle finit par grimper de faux contreforts, ce qui signifie qu'elle reste bloquée dans des maxima locaux. Il est donc préférable d'examiner l'ensemble de l'espace avant de prendre des décisions d'escalade. Pour ce faire, l'espace entier est d'abord exploré pour voir à quoi il ressemble. Cela nous permet d'éviter de rester bloqués sur un plateau ou des maxima locaux. Dans le recuit simulé, nous reformulons le problème et le résolvons par minimisation, plutôt que par maximisation. Nous descendons donc dans des vallées au lieu de grimper des collines. Nous faisons à peu près la même chose, mais d'une manière différente.

Nous utilisons une fonction objective pour guider la recherche. Cette fonction objective sert d'heuristique. La raison pour laquelle on l'appelle *recuit simulé* est qu'il est dérivé du processus métallurgique. Dans ce processus, nous chauffons d'abord les métaux, ce qui permet aux atomes de se diffuser à l'intérieur du métal, puis nous les laissons refroidir jusqu'à ce qu'ils atteignent l'état optimal souhaité en termes d'arrangement structurel des atomes. Il s'agit généralement de modifier les propriétés physiques d'un métal pour qu'il devienne plus souple et plus facile à travailler.

La vitesse à laquelle nous refroidissons le système est appelée **programme de recuit**. La vitesse de refroidissement est importante car elle a un impact direct sur le résultat. Dans le cas concret des métaux, si la vitesse de refroidissement est trop rapide, le métal finit par se stabiliser trop rapidement dans un état non idéal (structure atomique). Par exemple, si le métal chauffé est plongé dans l'eau froide, il finit par se fixer rapidement dans une structure qui n'était pas souhaitée, rendant le métal cassant, par exemple. Si la vitesse de refroidissement est lente et contrôlée, le métal a une chance d'atteindre la structure atomique optimale, ce qui lui confère les propriétés physiques souhaitées. Dans ce cas, les risques de faire de grands pas rapidement vers une pente sont plus faibles. Comme la vitesse de refroidissement est lente, il faudra du temps pour atteindre l'état optimal. Il est possible de procéder de la même manière avec les données.

Nous évaluons d'abord l'état actuel et vérifions s'il a atteint l'objectif. Si c'est le cas, nous nous arrêtons. Si ce n'est pas le cas, nous fixons la meilleure variable d'état à l'état actuel. Nous définissons ensuite un programme de recuit qui contrôle la vitesse à laquelle il descend dans une vallée. La différence est calculée entre l'état actuel et le nouvel état. Si le nouvel état n'est pas meilleur, nous en faisons l'état actuel avec une certaine probabilité prédéfinie. Pour ce faire, on utilise un générateur de nombres aléatoires et on décide en fonction d'un seuil. S'il est

supérieur au seuil, nous définissons le meilleur état comme étant cet état. Sur cette base, le programme de recuit est mis à jour en fonction du nombre de nœuds. Nous continuons ainsi jusqu'à ce que nous atteignions l'objectif. Une autre technique de recherche locale est l'algorithme de recherche avide. Nous en apprendrons plus à ce sujet dans la section suivante.

Construction d'une chaîne de caractères à l'aide de la recherche gloutonne (Greedy)

La **recherche gloutonne (Greedy)** est un paradigme algorithmique qui fait le choix localement optimal à chaque étape afin de trouver l'optimum global. Cependant, dans de nombreux problèmes, les algorithmes gloutons ne produisent pas des solutions globalement optimales. Un avantage d'utiliser des algorithmes gloutons est qu'ils produisent une solution approximative en un temps raisonnable. L'espoir est que cette solution approximative soit raisonnablement proche de la solution optimale globale.

Les algorithmes gloutons ne raffinent pas leurs solutions en fonction des nouvelles informations pendant la recherche. Par exemple, supposons que vous planifiez un voyage en voiture et que vous souhaitez prendre le meilleur itinéraire possible. Si vous utilisez un algorithme glouton pour planifier l'itinéraire, il pourrait vous demander de prendre des routes plus courtes en distance mais qui pourraient finalement prendre plus de temps. Il peut également vous conduire sur des chemins qui semblent plus rapides à court terme mais qui pourraient mener à des embouteillages plus tard. Cela se produit parce que les algorithmes gloutons ne voient que l'étape suivante et non la solution finale optimale globale.

Voyons comment résoudre un problème en utilisant une recherche gloutonne. Dans ce problème, nous allons essayer de recréer la chaîne de caractères d'entrée basée sur les alphabets. Nous demanderons à l'algorithme de rechercher dans l'espace des solutions et de construire un chemin vers la solution.

Nous utiliserons un paquet appelé `simpleai` tout au long de ce chapitre. Il contient diverses routines utiles pour construire des solutions en utilisant des techniques de recherche heuristique. Il est disponible sur https://github.com/simpleai-team/simpleai. Nous devons apporter quelques modifications au code source afin de le rendre compatible avec Python 3. Un fichier appelé `simpleai.zip` a été fourni avec le code du livre. Décompressez ce fichier dans un dossier appelé `simpleai`. Ce dossier contient toutes les modifications nécessaires à la bibliothèque originale pour la faire fonctionner avec Python 3. Placez le dossier `simpleai` dans le même répertoire que votre code et vous pourrez exécuter votre code sans problème.

Créez un nouveau fichier Python et importez les bibliothèques suivantes :

```
import argparse
import simpleai.search as ss
```

Définissez une fonction pour analyser les arguments d'entrée :

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Crée la chaîne de caractères
d\'entrée \
            en utilisant l\'algorithme glouton')
    parser.add_argument("--input-string", dest="input_string", required=True,
            help="Chaîne de caractères d'entrée")
    parser.add_argument("--initial-state", dest="initial_state", required=False,
            default='', help="Point de départ pour la recherche")
    return parser
```

Créez une classe qui contient les méthodes nécessaires pour résoudre le problème. Cette classe hérite de la classe `SearchProblem` disponible dans la bibliothèque. Quelques méthodes doivent être remplacées pour résoudre le problème en question. La première méthode `set_target` est une méthode personnalisée qui définit la chaîne cible :

```
class CustomProblem(ss.SearchProblem):
    def set_target(self, target_string):
        self.target_string = target_string
```

La méthode `actions` est une méthode de `SearchProblem` qui doit être remplacée. Elle est responsable de prendre les bonnes étapes vers l'objectif. Si la longueur de la chaîne actuelle est inférieure à celle de la chaîne cible, elle renverra la liste des alphabets possibles à choisir. Sinon, elle renverra une liste vide :

```
    # Vérifie l'état actuel et prend la bonne action
    def actions(self, cur_state):
        if len(cur_state) < len(self.target_string):
            alphabets = 'abcdefghijklmnopqrstuvwxyz'
            return list(alphabets + ' ' + alphabets.upper())
        else:
            return []
```

Créez une méthode pour calculer le résultat en concaténant la chaîne actuelle et l'action à prendre. Cette méthode fait partie de `SearchProblem` et nous la remplaçons :

```
    # Concatène l'état et l'action pour obtenir le résultat
    def result(self, cur_state, action):
        return cur_state + action
```

La méthode `is_goal` fait également partie de `SearchProblem` et est utilisée pour vérifier si l'objectif a été atteint :

```
    # Vérifie si l'objectif a été atteint
    def is_goal(self, cur_state):
        return cur_state == self.target_string
```

La méthode `heuristic` fait aussi partie de `SearchProblem` et nous devons la remplacer. Une heuristique est définie qui sera utilisée pour résoudre le problème. Un calcul est effectué pour voir à quelle distance se trouve l'objectif et utiliser cela comme heuristique pour guider vers l'objectif :

```
    # Définit l'heuristique qui sera utilisée
    def heuristic(self, cur_state):
        # Compare la chaîne actuelle avec la chaîne cible
        dist = sum([1 if cur_state[i] != self.target_string[i] else 0
                    for i in range(len(cur_state))])

        # Différence entre les longueurs
        diff = len(self.target_string) - len(cur_state)
        return dist + diff
```

Initialisez les arguments d'entrée :

```
if __name__ == '__main__':
    args = build_arg_parser().parse_args()
```

Initialisez l'objet `CustomProblem` :

```
    # Initialiser l'objet
    problem = CustomProblem()
```

Définissez le point de départ ainsi que l'objectif que nous voulons atteindre :

```
    # Définir la chaîne cible et l'état initial
    problem.set_target(args.input_string)
    problem.initial_state = args.initial_state
```

Exécutez le solveur :

```
    # Résoudre le problème
    output = ss.greedy(problem)
```

Affichez le chemin vers la solution :

```
    print('\nChaîne cible :', args.input_string)
    print('\nChemin vers la solution :')
    for item in output.path():
        print(item)
```

Le code complet est disponible dans le fichier `greedy_search.py`. Si vous exécutez le code avec un état initial vide :

```
$ python3 greedy_search.py --input-string 'Artificial Intelligence' --initial-state ''
```

Vous obtiendrez la sortie suivante :

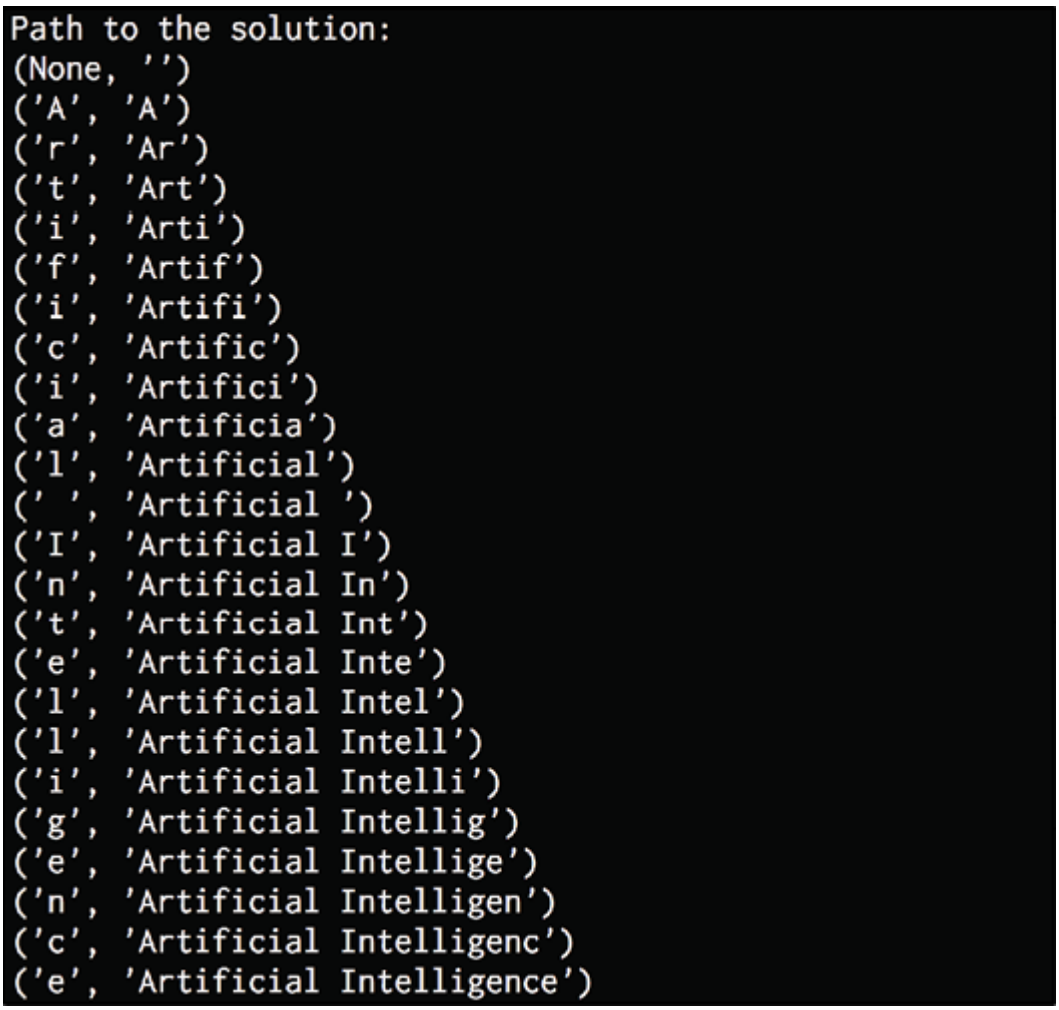

Figure 4 : Sortie du code exécuté avec un état initial vide

Si vous exécutez le code avec un point de départ non vide :

```
$ python3 greedy_search.py --input-string 'Artificial Intelligence with Python' --
initial-state 'Artificial Inte'
```

Vous obtiendrez la sortie suivante :

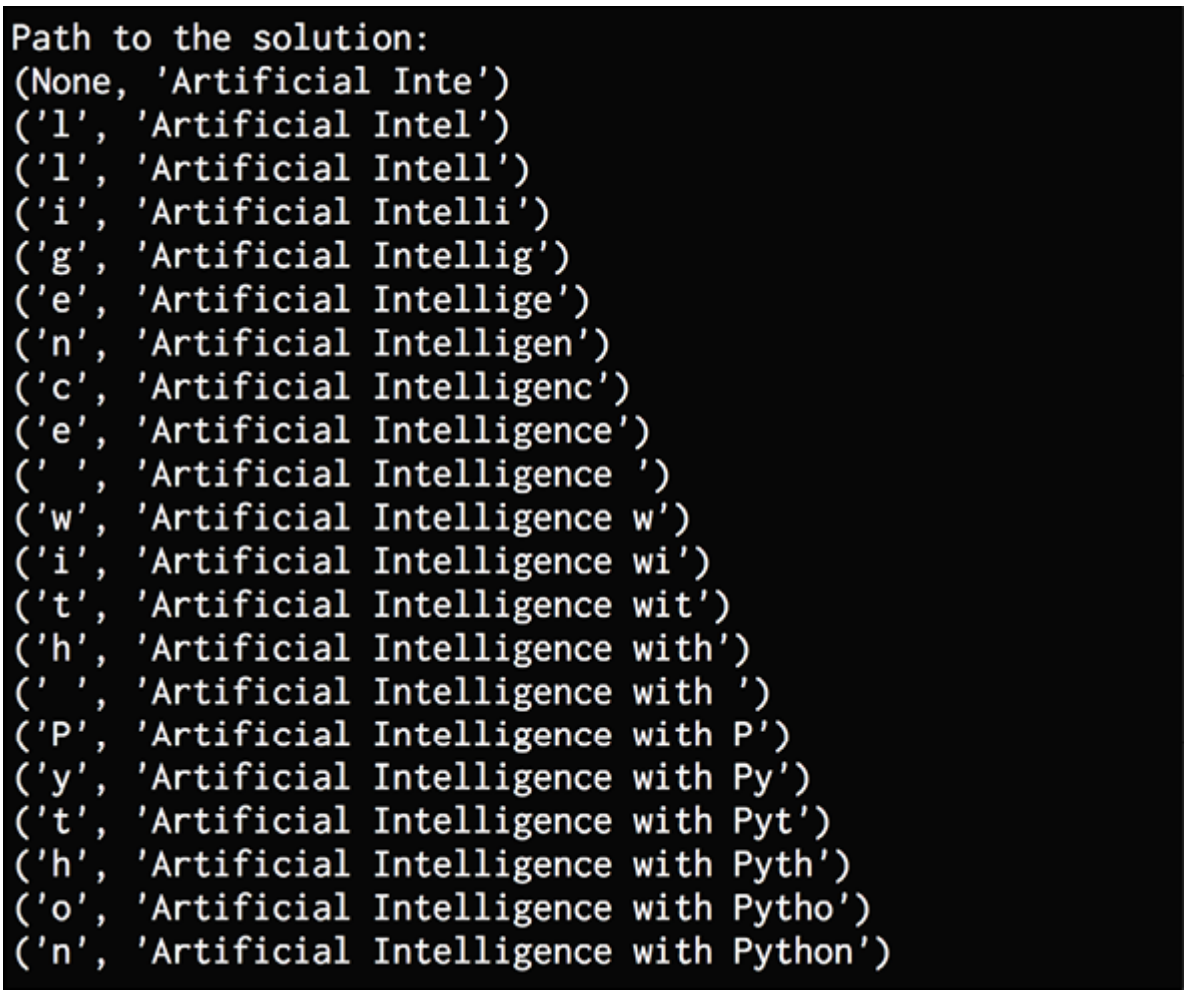

Figure 5 : Sortie du code exécuté avec un état initial non vide

Maintenant que nous avons couvert certaines techniques de recherche populaires, nous allons passer à la résolution de problèmes du monde réel en utilisant ces algorithmes de recherche.

Résolution d'un problème avec contraintes

Nous avons déjà discuté de la manière dont les CSP sont formulés. Appliquons-les à un problème du monde réel. Dans ce problème, nous avons une liste de noms et chaque nom peut prendre un ensemble de valeurs fixes. Nous avons également un ensemble de contraintes entre ces personnes qui doivent être respectées. Voyons comment procéder.

Créez un nouveau fichier Python et importez les bibliothèques suivantes :

```
from simpleai.search import CspProblem, backtrack, \
        min_conflicts, MOST_CONSTRAINED_VARIABLE, \
        HIGHEST_DEGREE_VARIABLE, LEAST_CONSTRAINING_VALUE
```

Définissez la contrainte qui spécifie que toutes les variables dans la liste d'entrée doivent avoir des valeurs uniques :

```
# Contrainte qui attend que toutes les variables différentes
# aient des valeurs différentes
def constraint_unique(variables, values):
    # Vérifie si toutes les valeurs sont uniques
    return len(values) == len(set(values))
```

Définissez la contrainte qui spécifie que la première variable doit être supérieure à la deuxième variable :

```
# Contrainte qui spécifie qu'une variable
# doit être supérieure à une autre
def constraint_bigger(variables, values):
    return values[0] > values[1]
```

Définissez la contrainte qui spécifie que si la première variable est impaire, alors la deuxième variable doit être paire et vice versa :

```
# Contrainte qui spécifie qu'il doit y avoir
# une variable impaire et une paire dans les deux variables
def constraint_odd_even(variables, values):
    # Si la première variable est paire, alors la deuxième doit
    # être impaire et vice versa
    if values[0] % 2 == 0:
        return values[1] % 2 == 1
    else:
        return values[1] % 2 == 0
```

Définissez la fonction `main` et spécifiez les variables :

```
if __name__ == '__main__':
    variables = ('John', 'Anna', 'Tom', 'Patricia')
```

Définissez la liste des valeurs que chaque variable peut prendre :

```
domains = {
    'John': [1, 2, 3],
    'Anna': [1, 3],
    'Tom': [2, 4],
    'Patricia': [2, 3, 4],
}
```

Définissez les contraintes pour divers scénarios. Dans ce cas, nous spécifions trois contraintes comme suit :

- John, Anna et Tom doivent avoir des valeurs différentes
- La valeur de Tom doit être supérieure à celle d'Anna
- Si la valeur de John est impaire, alors la valeur de Patricia doit être paire et vice versa

Utilisez le code suivant :

```
constraints = [
    (('John', 'Anna', 'Tom'), constraint_unique),
    (('Tom', 'Anna'), constraint_bigger),
    (('John', 'Patricia'), constraint_odd_even),
]
```

Utilisez les variables et les contraintes pour initialiser l'objet `CspProblem` :

```
# Résoudre le problème
problem = CspProblem(variables, domains, constraints)
```

Calculez la solution et affichez-la :

```
print('\nSolutions :\n\nNormal :', backtrack(problem))
```

Calculez la solution en utilisant l'heuristique `MOST_CONSTRAINED_VARIABLE` :

```
print('\nVariable la plus contrainte :', backtrack(problem,
        variable_heuristic=MOST_CONSTRAINED_VARIABLE))
```

Calculez la solution en utilisant l'heuristique `HIGHEST_DEGREE_VARIABLE` :

```
    print('\nVariable avec le plus haut degré :', backtrack(problem,
            variable_heuristic=HIGHEST_DEGREE_VARIABLE))
```

Calculez la solution en utilisant l'heuristique `LEAST_CONSTRAINING_VALUE` :

```
    print('\nValeur la moins contraignante :', backtrack(problem,
            value_heuristic=LEAST_CONSTRAINING_VALUE))
```

Calculez la solution en utilisant l'heuristique `MOST_CONSTRAINED_VARIABLE` pour les variables et `LEAST_CONSTRAINING_VALUE` pour les valeurs :

```
    print('\nVariable la plus contrainte et valeur la moins contraignante :',
            backtrack(problem, variable_heuristic=MOST_CONSTRAINED_VARIABLE,
            value_heuristic=LEAST_CONSTRAINING_VALUE))
```

Calculez la solution en utilisant l'heuristique `HIGHEST_DEGREE_VARIABLE` pour les variables et `LEAST_CONSTRAINING_VALUE` pour les valeurs :

```
    print('\nPlus haut degré et valeur la moins contraignante :',
            backtrack(problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE,
            value_heuristic=LEAST_CONSTRAINING_VALUE))
```

Calculez la solution en utilisant l'heuristique de conflits minimaux :

```
    print('\nConflits minimaux :', min_conflicts(problem))
```

Le code complet est disponible dans le fichier `constrained_problem.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

```
Solutions:
Normal: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}
Most constrained variable: {'Patricia': 2, 'John': 3, 'Anna': 1, 'Tom': 2}
Highest degree variable: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}
Least constraining value: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}
Most constrained variable and least constraining value: {'Patricia': 2, 'John': 3, 'Anna': 1, 'Tom': 2}
Highest degree and least constraining value: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}
Minimum conflicts: {'Patricia': 4, 'John': 1, 'Anna': 3, 'Tom': 4}
```

Figure 6 : Résolution de la solution avec l'heuristique de conflits minimaux

Vous pouvez vérifier les contraintes pour voir si les solutions satisfont toutes ces contraintes.

Résolution du problème de coloration de régions

Utilisons le cadre de satisfaction de contraintes pour résoudre le problème de coloration de régions. Considérez la figure suivante :

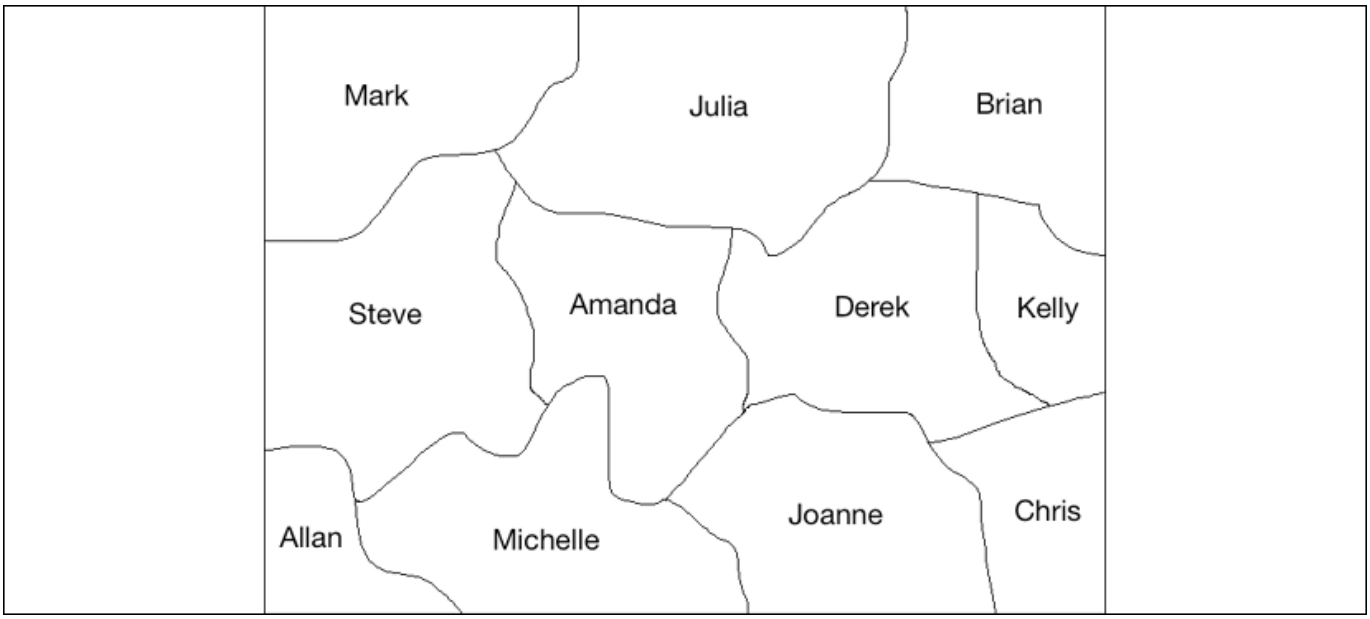

Figure 7 : Cadre pour le problème de coloration de régions

Nous avons quelques régions dans la figure précédente qui sont étiquetées avec des noms. L'objectif est de les colorier avec quatre couleurs de sorte qu'aucune région adjacente n'ait la même couleur.

Créez un nouveau fichier Python et importez les bibliothèques suivantes :

```
from simpleai.search import CspProblem, backtrack
```

Définissez la contrainte qui spécifie que les valeurs doivent être différentes :

```
# Définir la fonction qui impose la contrainte
# que les voisins doivent avoir des couleurs différentes
def constraint_func(names, values):
    return values[0] != values[1]
```

Définissez la fonction `main` et spécifiez la liste des noms :

```
if __name__ == '__main__':
    # Spécifier les variables
    names = ('Mark', 'Julia', 'Steve', 'Amanda', 'Brian',
            'Joanne', 'Derek', 'Allan', 'Michelle', 'Kelly')
```

Définissez la liste des couleurs possibles :

```
    # Définir les couleurs possibles
    colors = dict((name, ['red', 'green', 'blue', 'gray']) for name in names)
```

Nous devons convertir les informations de la carte en quelque chose que l'algorithme peut comprendre. Définissons les contraintes en spécifiant la liste des personnes qui sont adjacentes les unes aux autres :

```
    # Définir les contraintes
    constraints = [
        (('Mark', 'Julia'), constraint_func),
        (('Mark', 'Steve'), constraint_func),
        (('Julia', 'Steve'), constraint_func),
        (('Julia', 'Amanda'), constraint_func),
        (('Julia', 'Derek'), constraint_func),
        (('Julia', 'Brian'), constraint_func),
        (('Steve', 'Amanda'), constraint_func),
        (('Steve', 'Allan'), constraint_func),
        (('Steve', 'Michelle'), constraint_func),
        (('Amanda', 'Michelle'), constraint_func),
        (('Amanda', 'Joanne'), constraint_func),
        (('Amanda', 'Derek'), constraint_func),
        (('Brian', 'Derek'), constraint_func),
        (('Brian', 'Kelly'), constraint_func),
        (('Joanne', 'Michelle'), constraint_func),
        (('Joanne', 'Amanda'), constraint_func),
        (('Joanne', 'Derek'), constraint_func),
        (('Joanne', 'Kelly'), constraint_func),
        (('Derek', 'Kelly'), constraint_func),
    ]
```

Utilisez les variables et les contraintes pour initialiser l'objet :

```
    # Résoudre le problème
    problem = CspProblem(names, colors, constraints)
```

Calculez la solution et affichez-la :

```
    # Afficher la solution
    output = backtrack(problem)
    print('\nMappage des couleurs :\n')
    for k, v in output.items():
        print(k, '==>', v)
```

Le code complet est disponible dans le fichier `coloring.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

```
Color mapping:

Derek ==> blue
Michelle ==> gray
Allan ==> red
Steve ==> blue
Julia ==> green
Amanda ==> red
Joanne ==> green
Mark ==> red
Kelly ==> gray
Brian ==> red
```

Figure 8 : Sortie du mappage des couleurs

Si vous coloriez les régions en fonction de cette sortie, vous obtiendrez ce qui suit :

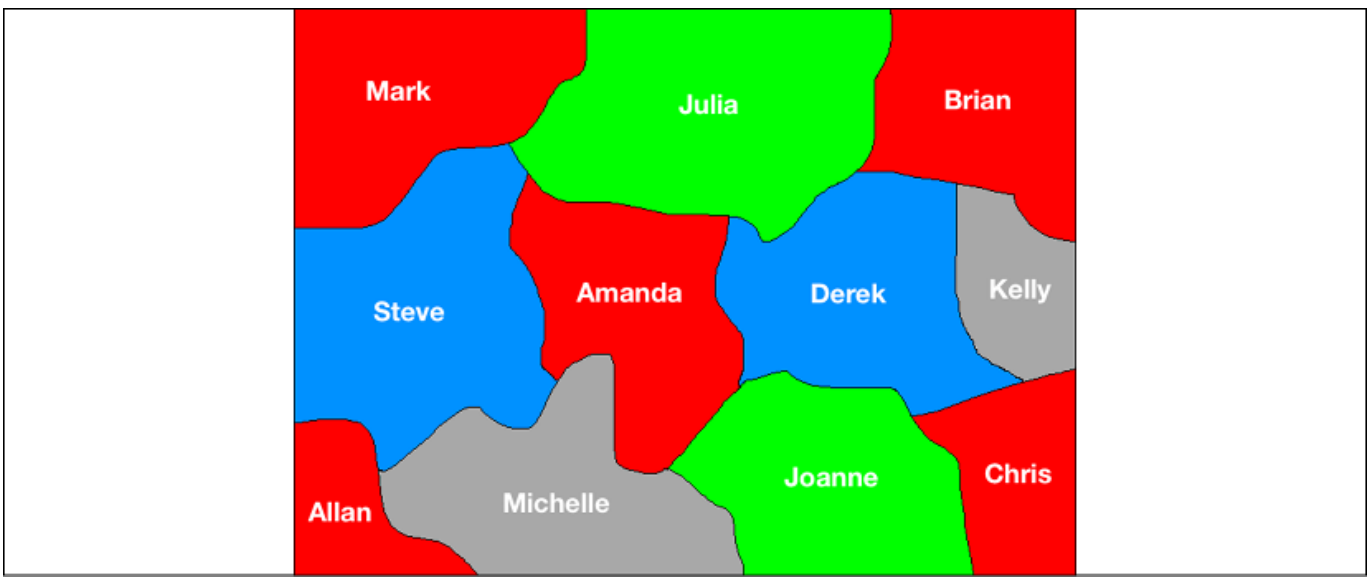

Figure 9 : Solution au problème de coloration de régions

Vous pouvez vérifier qu'aucune deux régions adjacentes n'ont la même couleur.

Construction d'un résolveur de puzzle 8-pièces

Le puzzle 8-pièces est une variante du puzzle 15-pièces. Vous pouvez le consulter sur https://fr.wikipedia.org/wiki/Puzzle_15. Vous serez présenté avec une grille aléatoire et votre objectif est de la ramener à la configuration ordonnée originale. Vous pouvez jouer au jeu pour vous familiariser avec lui sur http://mypuzzle.org/sliding.

Nous utiliserons un **algorithme A*** pour résoudre ce problème. C'est un algorithme utilisé pour trouver des chemins vers la solution dans un graphe. Cet algorithme est une combinaison de l'**algorithme de Dijkstra** et d'une recherche gloutonne best-first. Au lieu de deviner aveuglément où aller ensuite, l'algorithme A* choisit celui qui semble le plus prometteur. À chaque nœud, la liste de toutes les possibilités est générée, puis celle avec le coût minimal nécessaire pour atteindre l'objectif est choisie.

Voyons comment définir la fonction de coût. À chaque nœud, le coût doit être calculé. Ce coût est essentiellement la somme de deux coûts : le premier coût est le coût d'atteindre le nœud actuel et le second coût est le coût d'atteindre l'objectif depuis le nœud actuel.

Nous utilisons cette sommation comme heuristique. Comme nous pouvons le voir, le second coût est essentiellement une estimation qui n'est pas parfaite. Si elle l'était, l'algorithme A* arriverait rapidement à la solution. Mais ce n'est généralement pas le cas. Il faut du temps pour trouver le meilleur chemin vers la solution. A* est efficace pour trouver les chemins optimaux, cependant, et est l'une des techniques les plus populaires.

Utilisons l'algorithme A* pour construire un résolveur de puzzle 8-pièces. Il s'agit d'une variante de la solution donnée dans la bibliothèque `simpleai`. Créez un nouveau fichier Python et importez les bibliothèques suivantes :

```
from simpleai.search import astar, SearchProblem
```

Définissez une classe qui contient les méthodes pour résoudre le puzzle 8-pièces :

```
# Classe contenant les méthodes pour résoudre le puzzle
class PuzzleSolver(SearchProblem):
```

Remplacez la méthode `actions` pour l'aligner avec le problème actuel :

```
    # Méthode d'action pour obtenir la liste des
    # nombres possibles à déplacer dans l'espace vide
    def actions(self, cur_state):
        rows = string_to_list(cur_state)
        row_empty, col_empty = get_location(rows, 'e')

        actions = []
        if row_empty > 0:
            actions.append(rows[row_empty - 1][col_empty])
        if row_empty < 2:
            actions.append(rows[row_empty + 1][col_empty])
        if col_empty > 0:
            actions.append(rows[row_empty][col_empty - 1])
        if col_empty < 2:
            actions.append(rows[row_empty][col_empty + 1])
        return actions
```

Remplacez la méthode `result`. Convertissez la chaîne en liste et extrayez l'emplacement de l'espace vide. Générez le résultat en mettant à jour les emplacements :

```
    # Retourne l'état résultant après avoir déplacé
    # une pièce dans l'espace vide
    def result(self, state, action):
        rows = string_to_list(state)
        row_empty, col_empty = get_location(rows, 'e')
        row_new, col_new = get_location(rows, action)
        rows[row_empty][col_empty], rows[row_new][col_new] = \
                rows[row_new][col_new], rows[row_empty][col_empty]
        return list_to_string(rows)
```

Vérifiez si l'objectif a été atteint :

```
    # Retourne vrai si un état est l'état objectif
    def is_goal(self, state):
        return state == GOAL
```

Définissez la méthode `heuristic`. Nous utiliserons l'heuristique qui calcule la distance entre l'état actuel et l'état objectif en utilisant la distance de Manhattan :

```
    # Retourne une estimation de la distance d'un état à
    # l'objectif en utilisant la distance de Manhattan
    def heuristic(self, state):
        rows = string_to_list(state)
        distance = 0
        for number in '12345678e':
            row_new, col_new = get_location(rows, number)
            row_new_goal, col_new_goal = goal_positions[number]
            distance += abs(row_new - row_new_goal) + abs(col_new - col_new_goal)
        return distance
```

Définissez une fonction pour convertir une liste en chaîne de caractères :

```
# Convertir une liste en chaîne de caractères
def list_to_string(input_list):
    return '\n'.join(['-'.join(x) for x in input_list])
```

Définissez une fonction pour convertir une chaîne de caractères en liste :

```
# Convertir une chaîne de caractères en liste
def string_to_list(input_string):
    return [x.split('-') for x in input_string.split('\n')]
```

Définissez une fonction pour obtenir l'emplacement d'un élément donné dans la grille :

```
# Trouver l'emplacement 2D de l'élément d'entrée
def get_location(rows, input_element):
    for i, row in enumerate(rows):
        for j, item in enumerate(row):
            if item == input_element:
                return i, j
```

Définissez l'état initial et l'objectif final que nous voulons atteindre :

```
# Résultat final que nous voulons atteindre
GOAL = '''1-2-3
4-5-6
7-8-e'''
```

```
# Point de départ
INITIAL = '''1-e-2
6-3-4
7-5-8'''
```

Suivez les positions cibles pour chaque pièce en créant une variable :

```
# Créer un cache pour la position cible de chaque pièce
goal_positions = {}
rows_goal = string_to_list(GOAL)
for number in '12345678e':
    goal_positions[number] = get_location(rows_goal, number)
```

Créez l'objet solveur A* en utilisant l'état initial défini précédemment et extrayez le résultat :

```
# Créer l'objet solveur
result = astar(PuzzleSolver(INITIAL))
```

Affichez la solution :

```
# Afficher la solution
for i, (action, state) in enumerate(result.path()):
    print()
    if action == None:
        print('Configuration initiale')
    elif i == len(result.path()) - 1:
        print('Après avoir déplacé', action, 'dans l\'espace vide. Objectif atteint
!')
    else:
        print('Après avoir déplacé', action, 'dans l\'espace vide')
```

Le code complet est disponible dans le fichier `puzzle.py`. Si vous exécutez le code, vous obtiendrez une sortie longue.

Elle commencera comme suit :

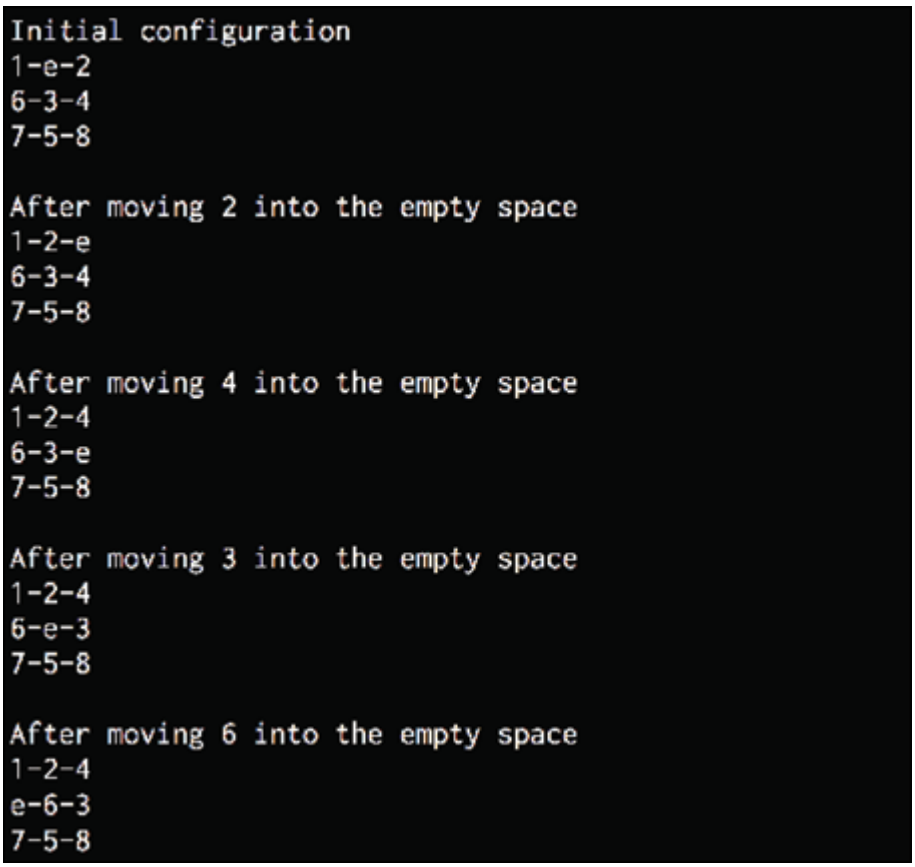

Figure 10 : Sortie de PuzzleSolver

Si vous faites défiler, vous verrez les étapes prises pour arriver à la solution. À la fin, vous verrez ce qui suit :

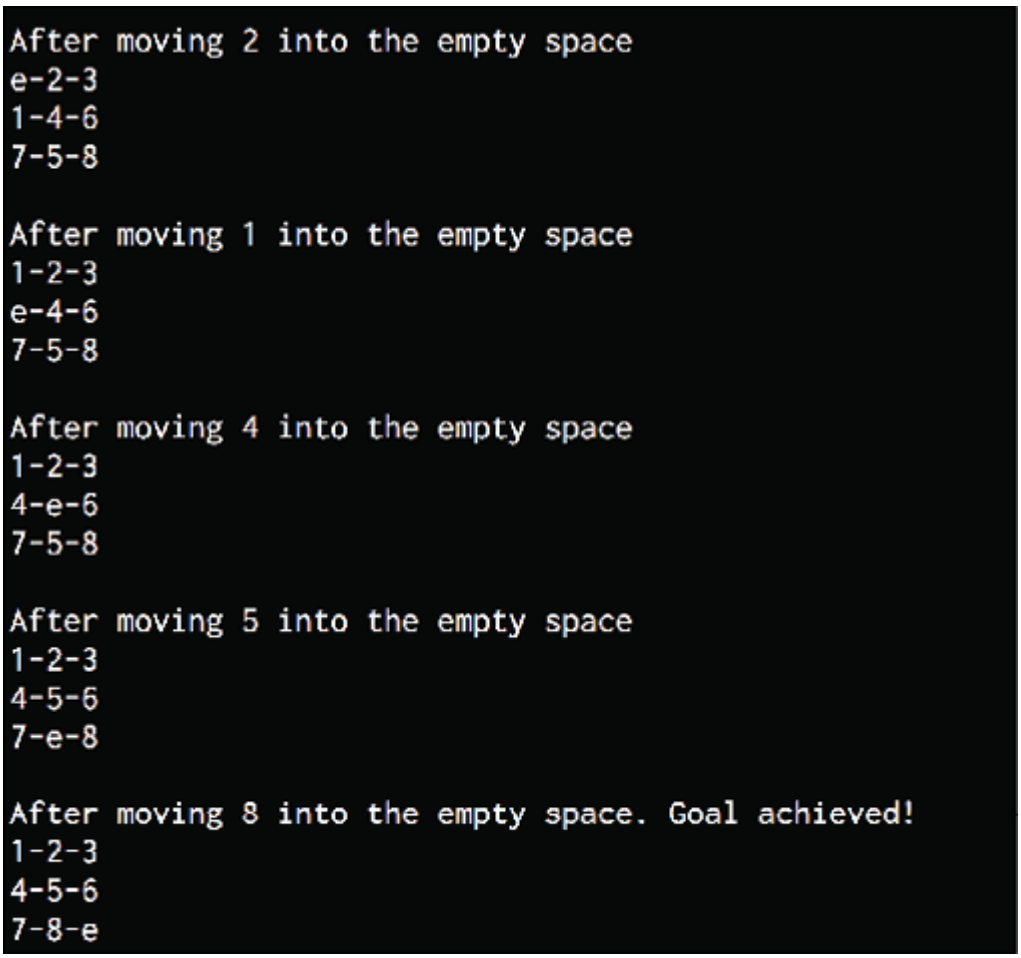

Figure 11 : Fin de la sortie PuzzleSolver - Objectif atteint !

Comme vous pouvez le voir, l'objectif est atteint et le puzzle est résolu.

Construction d'un résolveur de labyrinthe

Utilisons l'algorithme A* pour résoudre un labyrinthe. Considérez la figure suivante :

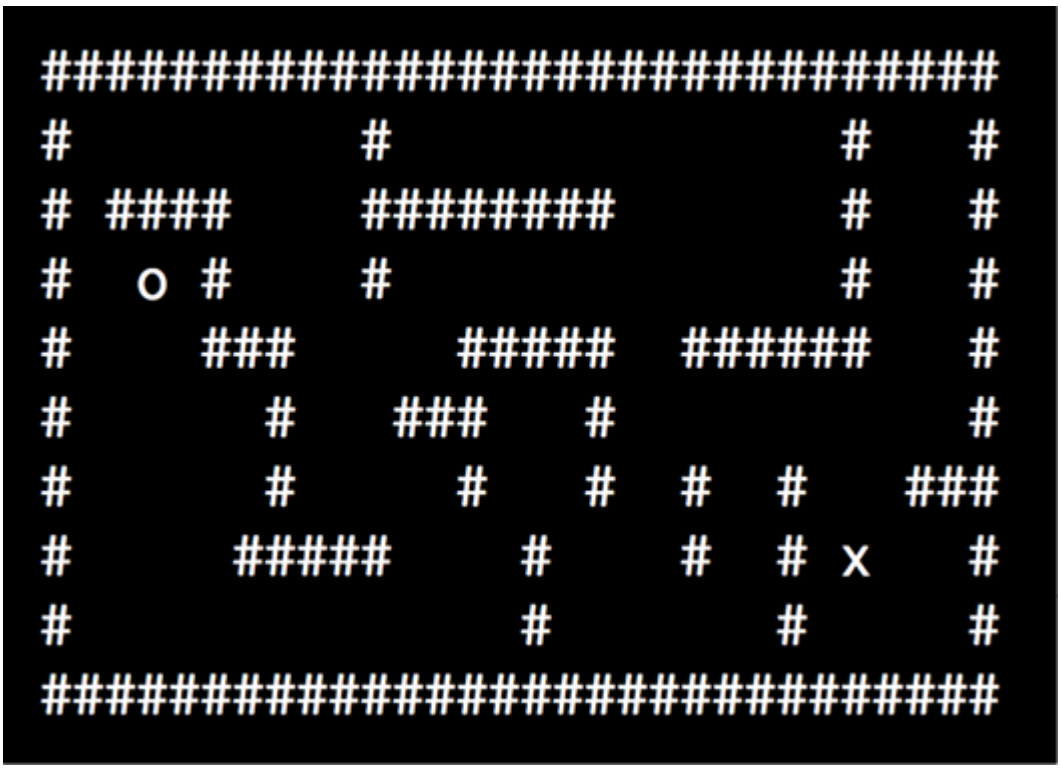

Figure 12 : Exemple de problème de labyrinthe

Les symboles # indiquent des obstacles. Le symbole **o** représente le point de départ, et **x** représente l'objectif. L'objectif est de trouver le chemin le plus court du début jusqu'au point d'arrivée. Voyons comment le faire en Python. La solution suivante est une variante de la solution fournie dans la bibliothèque `simpleai`. Créez un nouveau fichier Python et importez les bibliothèques suivantes :

```
import math
from simpleai.search import SearchProblem, astar
```

Créez une classe qui contient les méthodes nécessaires pour résoudre le problème :

```
# Classe contenant les méthodes pour résoudre le labyrinthe
class MazeSolver(SearchProblem):
```

Définissez la méthode d'initialisation :

```
    # Initialiser la classe
    def __init__(self, board):
        self.board = board
        self.goal = (0, 0)

        for y in range(len(self.board)):
            for x in range(len(self.board[y])):
                if self.board[y][x].lower() == "o":
                    self.initial = (x, y)
                elif self.board[y][x].lower() == "x":
                    self.goal = (x, y)

        super(MazeSolver, self).__init__(initial_state=self.initial)
```

Remplacez la méthode `actions`. À chaque position, nous devons vérifier le coût de déplacement vers les cellules voisines et ensuite ajouter toutes les actions possibles. Si la cellule voisine est bloquée, cette action n'est pas considérée :

```
    # Définir la méthode qui prend des actions
    # pour arriver à la solution
    def actions(self, state):
        actions = []
        x, y = state
        directions = [("up", (x, y-1)), ("down", (x, y+1)),
                      ("left", (x-1, y)), ("right", (x+1, y))]
        for action, (newx, newy) in directions:
            if 0 <= newx < len(self.board[0]) and 0 <= newy < len(self.board):
                if self.board[newy][newx] != "#":
                    actions.append(action)
        return actions
```

Remplacez la méthode `result`. En fonction de l'état actuel et de l'action d'entrée, mettez à jour les coordonnées `x` et `y` :

```
    # Mettre à jour l'état en fonction de l'action
    def result(self, state, action):
        x, y = state

        if action == "up":
            y -= 1
        elif action == "down":
            y += 1
        elif action == "left":
            x -= 1
        elif action == "right":
            x += 1

        return (x, y)
```

Vérifiez si nous avons atteint l'objectif :

```
    # Vérifie si nous avons atteint l'objectif
    def is_goal(self, state):
        return state == self.goal
```

Nous devons définir la fonction `cost`. C'est le coût de prendre une action, et il est différent pour les mouvements verticaux/horizontaux et les mouvements diagonaux. Nous définirons ces coûts plus tard :

```
    # Calculer le coût de prendre une action
    def cost(self, state, action, state2):
        return COSTS[action]
```

Définissez l'heuristique qui sera utilisée. Dans ce cas, nous utiliserons la distance Euclidienne :

```
    # Heuristique que nous utilisons pour arriver à la solution
    def heuristic(self, state):
        x, y = state
        gx, gy = self.goal
        return math.sqrt((x - gx) ** 2 + (y - gy) ** 2)
```

Définissez la fonction `main` et définissez la carte dont nous avons discuté précédemment :

```
if __name__ == "__main__":
    # Définir la carte
    MAP = """
    ##############################
    #         #              #   #
    # ####    ########       #   #
    #  o #    #              #   #
    #    ###     #####  ######   #
    #      #   ###   #           #
    #      #     #   #  #  #   ###
    #     #####    #    #  # x   #
    #              #       #     #
    ##############################
    """
```

Convertissez les informations de la carte en une liste :

```
    # Convertir la carte en une liste
    print(MAP)
    MAP = [list(x) for x in MAP.split("\n") if x]
```

Définissez le coût de déplacement dans la carte. Un mouvement diagonal est plus coûteux que les mouvements horizontaux ou verticaux :

```
    # Définir le coût de déplacement dans la carte
    cost_regular = 1.0
    cost_diagonal = 1.7
```

Attribuez les coûts aux mouvements correspondants :

```
    # Créer le dictionnaire de coûts
    COSTS = {
        "up": cost_regular,
        "down": cost_regular,
        "left": cost_regular,
        "right": cost_regular,
        "up left": cost_diagonal,
        "up right": cost_diagonal,
        "down left": cost_diagonal,
        "down right": cost_diagonal,
    }
```

Créez un objet solveur en utilisant la classe personnalisée définie précédemment :

```
# Créer l'objet résolveur de labyrinthe
problem = MazeSolver(MAP)
```

Exécutez le solveur sur la carte et extrayez le résultat :

```
# Exécuter le solveur
result = astar(problem, graph_search=True)
```

Extrayez le chemin du résultat :

```
# Extraire le chemin
path = [x[1] for x in result.path()]
```

Affichez la sortie :

```
# Afficher le résultat
print()
for y in range(len(MAP)):
    for x in range(len(MAP[y])):
        if (x, y) == problem.initial:
            print('o', end='')
        elif (x, y) == problem.goal:
            print('x', end='')
        elif (x, y) in path:
            print('·', end='')
        else:
            print(MAP[y][x], end='')
    print()
```

Le code complet est disponible dans le fichier `maze.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

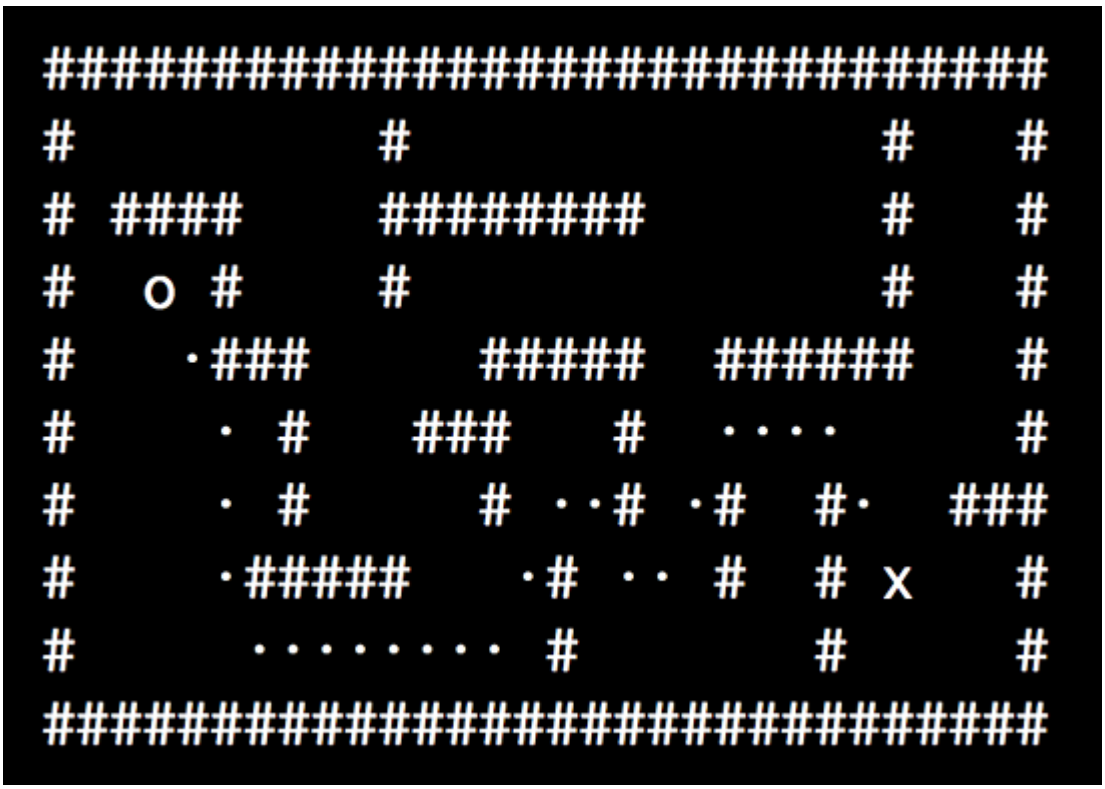

Figure 13 : Solution au problème de labyrinthe

Comme vous pouvez le voir, l'algorithme a laissé une trace de points et a trouvé la solution du point de départ **o** au point d'arrivée **x**. Cela conclut notre démonstration de l'algorithme A* dans cette dernière section du chapitre.

Résumé

Dans ce chapitre, nous avons appris comment fonctionnent les techniques de recherche heuristique. Nous avons discuté de la différence entre les recherches non informées et informées. Nous avons appris ce que sont les problèmes de satisfaction de contraintes et comment nous pouvons résoudre des problèmes en utilisant ce paradigme. Nous avons discuté de la manière dont fonctionnent les techniques de recherche locale et pourquoi le recuit simulé est utilisé en pratique. Nous avons implémenté une recherche gloutonne pour un problème de chaîne de caractères. Nous avons résolu un problème en utilisant la formulation CSP.

Nous avons utilisé cette approche pour résoudre le problème de coloration de régions. Nous avons ensuite discuté de l'algorithme A* et de la manière dont il peut être utilisé pour trouver les chemins optimaux vers la solution. Nous l'avons utilisé pour construire un résolveur de puzzle 8-pièces ainsi qu'un résolveur de labyrinthe. Dans le prochain chapitre, nous discuterons des algorithmes génétiques et de la manière dont ils peuvent être utilisés pour résoudre des problèmes du monde réel.

```
end
```