CS54 Lab #3 - Exhaustive Search and Backtracking

Objectifs

Objectifs principaux

- Réaliser plusieurs implémentations d'algorithmes de recherche exhaustive et de recherche avec retour arrière (*backtracking*) pour résoudre des problèmes classiques : problèmes du sac à dos, des pyramides et des récipients.

Objectifs annexes

- Utiliser la plateforme GitLab de l'école pour y déposer les réalisations ;
- Écrire des tests unitaires pour s'assurer de la correction des implémentations proposées.

Exercices

Exercice 1 : Sac à dos

Le problème du *sac à dos* est un problème d'optimisation classique. L'objectif est de choisir autant d'objets que peut en contenir un sac à dos dont la capacité est limitée. Dans cet exercice vous allez réaliser des algorithmes permettant de résoudre ce problème en appliquant une recherche exhaustive des solutions.

L'ensemble de l'exercice est à réaliser dans un fichier source Python dénommé `knapsack.py`.

Question 1 Écrire une fonction permettant de trouver une des meilleures solutions possibles.

Le profil attendu de la fonction est :

```
1 def find_one_best_solution(capacity : int, weights : List[int]) -> List
    [bool]
```

où :

- `capacity` est la capacité maximale du sac à dos
- `weight` est le tableau du poids de chaque objet (`weight[i]` contient le poids de l'objet *i*) La fonction retournera un tableau de valeurs booléennes indiquant si l'objet *i* doit être pris ou non.

Vous pouvez et nous vous le conseillons fortement d'écrire d'autres fonctions (ajout d'un objet, suppression d'un objet, évaluation du poids d'une solution, évaluation de la valeur d'une solution).

Question 2 Écrire une fonction permettant de trouver toutes les meilleures solutions possibles. Le profil attendu de la fonction est :

```
def find_all_best_solutions(capacity : int, weights : List[int]) ->
    List[List[bool]]
```

Question 3 Généraliser vos algorithmes pour résoudre les problèmes où la valeur d'un objet est décorrélé de son poids. Il s'agi donc maintenant de maximiser la valeur du contenu du sac en respectant les contraintes de poids.

Écrire une fonction permettant de trouver toutes les meilleures solutions possibles. Le profil attendu de la fonction est :

```
def find_all_best_loot_solutions(capacity : int, weights : List[int],
    prices : List[int]) -> List[List[bool]]
```

Question 4 Reprenez l'algorithme précédent (de la question 3) en donnant la possibilité de prendre plusieurs fois un même objet. Le nombre maximal d'instances utilisables est le même pour chaque objet et vous est fourni sous forme d'un entier dénommé stock. Le profil attendu de la nouvelle fonction est:

```
def find_all_best_loot_solutions_with_stock(capacity : int, weights :
    List[int], prices : List[int], stock : int) -> List[List[bool]]
```

Les plus avancés d'entre vous pourront remplacer l'entier stock par une valeur individualisée par objet. Dans ce cas, l'entier `stock` est remplacé par une liste stocks qui indique le nombre maximal d'instances disponibles par objets. Dans ce cas, une nouvelle fonction est à construire :

```
def find_all_best_loot_solutions_with_stocks(capacity : int, weights :
    List[int], prices : List[int], stocks : List[int]) -> List[List[int
    ]]
```

Exercice 2 : Remplissage de pyramides

On considère une pyramide la tête en bas de hauteur *h* comme celle représentée plus bas. On cherche à remplir toutes les cases avec chacun des entiers compris entre 1 et $\frac{h\times(h+1)}{2}$ en respectant les contraintes suivantes :

1. chaque nombre de $[1, \frac{h\times(h+1)}{2}]$ ne figure qu'une fois sur la pyramide ;
2. la valeur de chaque case est égale à la différence des deux cases placées au dessus d'elle.

La solution consistant à générer dans un premier temps toutes les permutations possibles puis à tester si chaque permutation correspond à une pyramide valide dans un second temps étant trop inefficace, nous ne vous demandons pas d'implémenter cette solution

L'ensemble de l'exercice est à réaliser dans un fichier source Python dénommé `pyramides.py`.

Question 1 : Algorithme de construction pas à pas Réaliser un algorithme permettant de résoudre le problème en vérifiant à chaque étape de la construction que les contraintes sont respectées et en interrompant dés qu'un choix mène à une situation interdite.

Dans cet exercice, nous supposerons que la pyramide est rangée par lignes dans le tableau (l'élément de la première ligne, suivi par les deux éléments de la seconde ligne, suivis par les autres éléments des autres lignes).

Écrire une fonction permettant de trouver une solution possible pour une hauteur donnée. Le profil attendu de la fonction est :

```
1 def find_one_solution(height : int) -> List[int]
```

Écrire une fonction permettant de trouver toutes les solutions possibles pour une hauteur donnée. Le profil attendu de la fonction est :

```
1 def find_all_solutions(height : int) -> List[List[int]]
```

Question 2 : Algorithme de construction par tranches (génération par propagation) Dupliquer vos algorithmes et modifier les pour remplir la pyramide par tranches (diagonale par diagonale).

Dans cet exercice, nous supposerons que la pyramide est rangée par diagonales dans le tableau (l'élément de la première diagonale, suivi par les deux éléments de la seconde diagonale, suivis par les autres éléments des autres diagonales).

```
1 def find_all_solutions_using_propagation(height : int) -> List[List[int
    ]]
```

Exercice 3 : Le problème des récipients

On dispose d'un certain nombre de récipients dont on connaît la capacité, et d'une fontaine d'eau.

On cherche quels sont les transvasements à réaliser pour faire en sorte que l'un des récipients contienne une quantité d'eau donnée. Seules les opérations suivantes sont autorisées :

- remplir complètement un récipient depuis la fontaine ;

- vider complètement un récipient dans la fontaine ;
- transvaser un récipient dans un autre jusqu'à ce que le récipient source soit complètement vide ou que le récipient de destination soit complètement plein.

L'ensemble de l'exercice est à réaliser dans un fichier source Python dénommé `containers.py`.

Question 1 : Parcours exhaustif mais borné en profondeur Après avoir modélisé le problème, écrire un algorithme effectuant une recherche exhaustive des solutions de transvasements possibles.

Écrire une fonction permettant de trouver une succession de transvasements pour une instance du problème donné et une profondeur maximale de recherche.

Le profil attendu de la fonction est :

```
1 def find_one_solution(depth : int, target : int, capacities : List[int
    ]) -> Tuple[bool, List[str]]
```

où :

- `depth` correspond à la profondeur maximale de recherche utilisée
- `capacities` est le tableau des capacités de chaque récipient Le résultat attendu est une liste d'action (chaîne de caractères) indiquant les actions de transvasement à réaliser.

Par exemple, l'appel `find_one_solution(7, 4, [5, 3])` devrait retourner la solution `(True, [ 'Remplir(0)', 'Transvaser(0,1)', 'Vider(1)', 'Transvaser(0,1)', 'Remplir(0)', 'Transvaser(0,1)'])`.

Question 2 : Parcours exhaustif mais en supprimant les situations déjà explorées Écrire une fonction permettant de trouver une succession de transvasements pour une instance du problème donné tout en évitant d'explorer à nouveau les situations déjà explorées.

Le profil attendu de la fonction est :

```
1 def find_one_solution_with_memoization(depth : int, target : int,
    capacities : List[int]) -> Tuple[bool, List[str]]
```

Pour limiter les descentes infinies, n'oublier pas de supprimer les actions qui ne mènent à rien (qui ne modifient pas la situation).

Question 3 : Parcours exhaustif en largeur Écrire une fonction permettant de trouver une succession de transvasements possible pour une instance du problème donné en réalisant un parcours en largeur d'abord des états à explorer.

Le profil attendu de la fonction est :

```python
def find_one_solution_breadth_first_search(target : int, capacities :
    List[int]) -> Tuple[bool, List[str]]
```

Bon courage ;)

Conseils et documentations

Quelques liens utiles

- le serveur gitlab : http://gitlab.telecomnancy.univ-lorraine.fr/ de l'école
- la page du cours CS54 sur Arche

La documentation sur :

- le tutoriel basique sur "git" sur Arche
- le langage Python : https://docs.python.org/3/

Gestion de versions (git)

Il vous est demandé :

- de *cloner* le dépôt présent sur la plateforme GitLab de l'école pour travailler en local sur votre machine personnelle ou une machine de l'école
- de *committer* chacune de vos versions de vos programmes et de *pousser* **régulièrement** ces commits sur la plateforme GitLab

Prenez l'habitude d'associer à vos commits des messages pertinents qui décrivent le contenu des modifications apportées par le commit.

Bonnes pratiques de développement

Vous veillerez à documenter/commenter le code Python que vous écrivez. Chaque fonction que vous définissez devrait être associée à commentaire pour préciser : le type et le sujet de chaque paramètre ; le type et la valeur de retour de la fonction ; les pré-conditions (et éventuellement post-conditions) de cette fonction.

Vous pouvez utiliser si vous le souhaitez :

- des indications de types (*Type Hinting*) disponibles depuis Python 3.5+ (https://docs.python.org/3/library/typing.h et définis dans la PEP 484. Ils vous permettent d'annoter des types à vos paramètres, valeur de retour ou encore variables.

- les *docstrings* (PEP 257)

L'hyperlien suivant pointe sur un article en ligne sur ces bonnes pratiques : https://www.codeflow.site/fr/article/docume
python-code

Tests unitaires (pytest) Nous vous invitons fortement à écrire des tests unitaires permettant de valider la correction et la résilience de vos algorithmes et de leur implémentation.

Pour cela, vous utiliserez la libraire pytest : https://docs.pytest.org/en/stable/.

Vous pouvez installer cette librairie dans votre environnement en exécutant la commande :

```
1 $ pip install -U pytest
```

Ensuite, pour exécuter vos tests (que vous écrirez dans des fichiers dénommés `xxx__test.py`), il vous suffit d'utiliser la commande :

```
1 $ pytest --verbose
```