ALGORITHMIQUE AVANCÉE

Voyageur de commerce : 2. programmation dynamique

PROGRAMMATION DYNAMIQUE

Ce concept a été introduit au début des années 1950 par Richard Bellman. À l'époque, le terme « programmation » signifie planification et ordonnancement. Une anecdote rapporte que Bellman a rajouté le terme « dynamique » pour obtenir des financements plus facilement.

Question 1. *Mémoïsation du calcul des distances.*

1. Observez dans les performances que le temps de calcul des distances est significatif et que les distances sont calculées de multiples fois.
   Par exemple, examinez la trace de *TEST 1.4.e*.

Nous allons utiliser un dictionnaire pour mémoriser les distances déjà calculées et alors tenter de remplacer du calcul par de la recherche de résultats dans une structure de données.

2. Implémentez tout cela dans la fonction `distance_memo` qui se trouve dans le fichier point.py

   ```
   def distance_memo(p:Point,q:Point)->float:
   ```

   en utilisant le dictionnaire vide `distances_memo` déclaré juste au-dessus et mentionné en global au début de la fonction.
3. Vérifiez la correction via *TEST 2.1.a* et les améliorations dans les performances en temps via *TEST 2.2.a*. Ces deux tests répètent dix fois l'expérience suivante : tirer aléatoirement 8 points dans le carré unité puis faire 300000 calculs avec toutes les fonctions de distance entre 2 des 8 points tirés au hasard.
4. Vous pouvez surveiller la mémoire pour les grandes instances avec *TEST 2.1.c* où nous tirons 10000 points au lieu de 8. Pourquoi cette limitation n'est pas un problème dans notre contexte ?

Cette technique de mémorisation est si efficace et répandue qu'il est parfois possible de l'utiliser directement via le module `functools`. Cela revient à indiquer, pour une fonction f donnée, que les résultats des appels à f sont stockés quelque part en mémoire et qu'à chaque appel, on cherche si les arguments n'ont pas déjà été utilisés. Pour le faire sur notre fonction `distance` il suffit de décommenter juste avant sa déclaration la ligne : `@fun.lru_cache(maxsize=None)`.

Elle fait exactement ce que vous venez de programmer avec des outils plus efficaces car de plus bas niveau.

Remplacez des calculs répétés par une mémorisation de la première occurrence du calcul est le thème de cette séance. C'est une des clefs de l'efficacité de la programmation dynamique.

Attention : dans certains contextes recalculer est plus efficace que de rechercher si et où on a mémorisé la première exécution de ce calcul.

Question 2. *Mise au point de la version récursive de la récurrence. Assimilation de la récurrence vue en cours.*

Dans cette partie on va s'intéresser à l'algorithme de programmation dynamique vu en cours basé sur un sous-ensemble S de villes et une ville t incluse dans S.

La ville est un entier mais pour l'instant S est une liste d'entiers.

Nous pourrions écrire une fonction listant tous les sous-ensembles de villes mais pour des optimisations futures, nous allons coder par des entiers ces sous-ensembles.

1. Implémentez la description récursive vue en cours :

```python
def solve_by_dynamic_programming1(tsp:Tsp)->(float,Permutation):
    # <EXO 2.2>
    def rec_solve(s:List[int],t:int)-> (float,List[int]):
        # - s sous-ensemble de ville sous forme de liste (triée).
        # - t dernière ville à visiter parmi toutes celles de s
        # - retourne la distance et une liste donnant l'ordre des visites des villes de s
        # ecrire ici la recurrence vue en cours
        return (0,0)
    # </EXO 2.2>
    (length,solution) = rec_solve(None,None)[1] # paramètres initiaux à trouver
    return (length,Permutation(solution))
```

Le but est d'écrire une version correcte sans se soucier de l'efficacité.

Il est même suggéré d'utiliser la fonction `min` de python pour trouver la meilleure solution parmi les possibilités listées.

Indication : `min(une_liste_de_tuples,key=lambda p:p[0])` L'optimisation de ce calcul du minimum est repoussé jusqu'à la toute fin de cette étape, mais elle pourrait se faire dès maintenant.

Une fois cette version correcte mise au point, nous ajouterons progressivement diverses optimisations par des transformations de cette version initiale. Nous n'aurons plus qu'à nous soucier de préserver l'expression de la récurrence (et non de la formuler au milieu des optimisations).

2. Le test *TEST 2.2.a* permet de vérifier que les résultats sont identiques aux approches par force brute. Notez l'apparition probable des problèmes numériques liés à l'ordre de sommation de nombres réels.
3. Le test *TEST 2.2.b* observe les performances en temps de cette version comparée à la meilleure version en force brute.
4. Le test *TEST 2.2.c* observe l'usage de la mémoire. Pourquoi les performances en mémoire semblent bonnes malgré la multitude des cas envisagés ?
5. Décommentez la ligne `@fun.lru_cache(maxsize=None)` juste avant la fonction `rec_solve`. Essayez de comprendre le message d'erreur expliquant l'échec. Ce n'est pas un obstacle fondamental, mais comme l'esprit de cette feuille est de mettre explicitement en œuvre la mémoïsation nous n'en discuterons pas plus.

Question 3. *Codage/Décodage de sous-ensemble par des entiers.*

Nous interprétons le codage binaire d'un entier $N = [01001101]_2$ comme un sous-ensemble de villes, le bit de poids 2^i correspondant à la ville i.

Par exemple, on a $N = 77 = 2^0 + 2^2 + 2^3 + 2^6$ qui correspond au sous-ensemble $S = [0, 2, 3, 6]$.

1. Implémentez les fonctions

   ```python
   def convert_int_to_subset(n:int)->List[int]:
   ```

   et

   ```python
   def convert_subset_to_int(List[int])->int:
   ```

2. Lancez les exemples des docstring via le test *TEST 2.3.a*.

Question 4. *Utilisez des entiers pour les subsets en passage de paramètres. Préparation à l'utilisation d'une table indexée par deux entiers comme en cours.*

1. Implémentez la version n'utilisant que des entiers comme paramètres. Une fois n'est pas coutume, vous êtes incités à débuter par un COPIER-coller [1] du corps de la fonction `rec_solve` de la fonction `solve_by_dynamic_programming1`. Remarquez que les deux sous-fonctions `rec_solve` sont distinctes car elles sont déclarées dans des fonctions différentes.

   ```python
   def solve_by_dynamic_programming2(tsp:Tsp)->(float,Permutation):
       def rec_solve(s_int:int,t:int)-> (float,List[int]):
           # <EXO 2.4>
           # adapter ici le corps de la fonction rec_solve de solve_by_dynamic_programming1
           return (0,[])
       (length,solution) = rec_solve(None,None) # paramètres initiaux à trouver
       return (length,Permutation(solution))
       # </EXO 2.4>
   ```

   Dans cette version, utilisez les entiers représentant les sous-ensembles uniquement au plus près des appels à `rec_solve`. Dans un exercice ultérieur, nous manipulerons encore plus les sous-ensembles sous forme de leur codage en entiers.

1. Alors que normalement la programmeuse idéal(isée) ne fait que du COUPER-coller car elle factorise au maximum son code.

2. Validez cette implémentation par le test *2.4.a*, comparez ses performances en temps par rapport à la première version et la meilleure en brute force par le test *2.4.b* et observer son utilisation de la mémoire par *2.4.c*.
3. Activez la mémoïsation automatique en décommentant la ligne `@fun.lru_cache(maxsize=None)` juste avant `rec_solve` et repassez les tests de performances.
4. Avez-vous transformé du temps en espace ?

Question 5. Partie optionnelle. *Manipuler les sous-ensembles uniquement sous forme d'entiers.*

Pour trouver le bit de poids fort d'un entier, nous disposons de la fonction `int.bit_length()`.

Dans le sous-ensemble décrit par l'entier `s:int`, nous pouvons ainsi trouver le plus grand élément.

De plus, connaissant un élément `i:int` de l'ensemble `s:int` une formule simple permet de calculer l'entier `t:int` représentant l'ensemble s privé de l'élément i.

Pour calculer des puissances de 2, l'opérateur `<<` est pratique pour multiplier par deux : `1<<2` donne 4, `1<<3` donne 8,...

Le complément est `~`, le « et logique » est `&`.

1. À l'aide de toutes ces observations nous pouvons réécrire plus efficacement la boucle sur les sous-ensembles dans une troisième version :
```python
def solve_by_dynamic_programming3(tsp:Tsp)->(float,Permutation):
    def rec_solve(s_int:int,t:int)-> (float,List[int]):
        # <EXO 2.5>
        # adapter ici le corps de la fonction rec_solve de solve_by_dynamic_programming2
        return (0,[])
    (length,solution) = rec_solve(None,None) # paramètres initiaux à trouver
    return (length,Permutation(solution))
    # </EXO 2.5>
```
2. Validez les tests *TODO 2.5.a*, puis comparez les performances en temps des fonctions `rec_solve` avec *TODO 2.5.b*.
3. Que pensez vous de la lisibilité de cette version ?
4. Justifie-t-elle le gain en temps ?

Lancez et étudiez les tests *TEST 2.5.a*,*TEST 2.5.b*.

Question 6. *Mémoïsation explicite dans une table des D[(s,t)] en remplissant la table avec les solutions partielles.*

Nous allons pour cette question oublier la possibilité d'automatiser la mémoïsation.

1. Implémentez la version calculant la table D vue en cours sous la forme d'un dictionnaire dont les clefs sont les paires d'entiers (s, t).
Attention la table contient encore la solution partielle `List[int]` (nous la supprimerons bientôt).
```python
def solve_by_dynamic_programming4(tsp:Tsp)->(float,Permutation):
    D = {} # dict storing explicitely results of rec_solve
    def rec_solve(s:int,t:int)-> (float,List[int]):
     # <EXO 2.6>
        return D[(s,t)]
    for s in range(1,2**tsp.n): # Pourquoi ?
        for t in range(tsp.n):
            rec_solve(s,t)
    return  D[(None,None)] # paramètres initiaux à trouver
    # </EXO 2.6>
```
2. Lancez et étudiez les tests *TEST 2.6.a*,*TEST 2.6.b*.

Question 7. *Élimination des appels récursifs à* `rec_solve`.

Pour éviter le coût de multiples appels récursifs à `rec_solve` et la vérification que le résultat a bien été calculé et stocké dans D, vous avez vu en cours qu'il était possible de parcourir les sous-cas dans un ordre bien choisi.

Un de ces ordres est donné dans le squelette de la version suivante notamment par la boucle `for s in range(1,2**tsp.n):`.

S'il est simple à décrire, comprendre pourquoi cela fonctionne mérite une explication.

```
def solve_by_dynamic_programming5(tsp:Tsp)->(float,Permutation):
    D = {} # dict storing explicitely results of rec_solve
    def rec_solve(s:int,t:int)-> None:
     # <EXO 2.7>
     # sans appel récursif à rec_solve
        return
    for s in range(1,2**tsp.n): # Pourquoi ?
        for t in range(tsp.n):
        if True: # find the test
               rec_solve(s,t)
    (length,solution) = D[(None,None)]
    return  (length,Permutation(solution)) # paramètres initiaux à trouver
    # </EXO 2.7>
```

1. Remarquez que cette version va bien jusqu'à `2**tsp.n-1` et non `2**(tsp.n-1)-1`. C'est (significativement) pénalisant pour les performances mais cela simplifie l'implémentation. Une prochaine question corrigera tardivement cette erreur de développement.
2. Lancez et étudiez les tests *TEST 2.7.a* et *TEST 2.7.b*.

Question 8. *Compression de la table (on ne retient que l'avant-dernière ville).*

Pour diminuer la taille de la table, au lieu de retourner la solution complète, on ne retient que l'avant-dernière ville.

1. Vérifiez ce qu'il se passe pour les cas de base.

Finalement, l'information à stocker est un tableau à double entrée `s:int` et `t:int` et à double valeur `length:float` et `tt:int`.

Cela permet d'utiliser directement deux tableaux/matrices `Dlength` et `Dtt` pour stocker plus efficacement.

Le prix à payer est une fonction extrayant une solution optimale à partir du tableau `Dtt`.

Nous vous proposons une fonction réalisant cette opération :

```
def extract_solution(s:int,t:int,Dtt)->Permutation:
```

2. Implémentez.

```
def solve_by_dynamic_programming6(tsp:Tsp)->(float,Permutation):
    D = {} # dict storing explicitely results of rec_solve
    Dlength = {} # 2D array
    Dtt = {} # 2D array
    def rec_solve(s:int,t:int)-> None:
     # <EXO 2.8>
     # sans appel récursif à rec_solve
        return
    for s in range(1,2**tsp.n):
        for t in range(tsp.n):
        if True: # find the test
               rec_solve(s,t)
    (length,solution) = extract_solution(None,None,Dtt)
    return  (length,Permutation(solution)) # paramètres initiaux à trouver
    # </EXO 2.8>
```

3. Lancez et étudiez les tests *TEST 2.8.a*,*TEST 2.8.b* et *TEST 2.8.c*.

Question 9. *Au fait, ce min, il est encore là ?*

Dans les tests de performances vous devriez avoir constaté que la fonction `min` prend un certain temps.

1. Faites-la disparaitre dans

   ```
   def solve_by_dynamic_programming7(tsp:Tsp)->(float,Permutation):
   ```

2. Lancez et étudiez les tests *TEST 2.9.a*,*TEST 2.9.b* et *TEST 2.9.c*.

Question 10. `s in range(1,2**(tsp.n-1))`

1. Implémentez dans

   ```
   def solve_by_dynamic_programming8(tsp:Tsp)->(float,Permutation):
   ```

   une version où la boucle sur `s` parcourt bien `range(1,2**(tsp.n-1))` puisque que la dernière ville visitée est bien `tsp.n-1`. Notez bien l'instruction supplémentaire à laquelle nous souhaitions initialement ne pas réfléchir.
2. Lancez et étudiez les tests *TEST 2.10.a*,*TEST 2.10.b* et *TEST 2.10.c*.