Intelligence Artificielle 1

Cours 13 - Prog. Dynamique

Table des matières

1 Définition

2 Fibonacci normal vs. programmation dynamique

3 Fibonacci - Arbre de décision

4 Fibonacci - Programmation Dynamique - Astruce

Définition

La programmation dynamique est une méthodologie algorithmique pour résoudre des problèmes en le décomposant en sous-problème (comme la récursivité), puis en sauvegardant le résultat obtenu de celui-ci dans une “cache de mémorisation” (memoization cache).

Le but est de retrouver la réponse pour un sous-problème dans la cache au lieu de calculer la branche du sous-problème à nouveau et ainsi éviter de perdre du temps.

Définition

Cette technique augmente grandement la vitesse d'exécution, mais il est généralement plus dur de programmer de tels algorithmes puisqu'il faut insérer la structure de données représentant la mémoire (cache).

La cache est souvent un dictionnaire, puisqu'il est très rapide d'accéder à une valeur par clé. Sinon, les tableaux et matrices sont souvent utilisés.

Fibonacci normal

```python
# Steve Levesque, All rights reserved

def fib(n):
    # Base cases (stop cases)
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

Fibonacci programmation dynamique

```python
# Steve Levesque, All rights reserved

def fib_mem(n, memo={}):
    # Base cases (stop cases)
    if n == 0:
        return 0
    if n == 1:
        return 1
    # Search in the dictionary for branches already computed
    if n in memo:
        return memo[n]

    memo[n] = fib_mem(n - 1, memo) + fib_mem(n - 2, memo)
    return memo[n]
```

Fibonacci - benchmark récursivité normale vs. programmation dynamique

```python
# Steve Levesque, All rights reserved
import time
from fibonacci_normal import fib
from fibonacci_mem import fib_mem

def benchmark(n):
    time_start_fib = time.perf_counter()
    print(fib(n))
    time_end_fib = time.perf_counter()
    time_start_fib_mem = time.perf_counter()
    print(fib_mem(n))
    time_end_fib_mem = time.perf_counter()

    runtime_fib = time_end_fib - time_start_fib
    runtime_fib_mem = time_end_fib_mem - time_start_fib_mem
    print('Took fib('+str(n)+f')  {runtime_fib:.3f} seconds')
    print('Took fib_mem('+str(n)+f') {runtime_fib_mem:.3f} seconds')

benchmark(40)
```

Fibonacci - benchmark récursivité normale vs. programmation dynamique

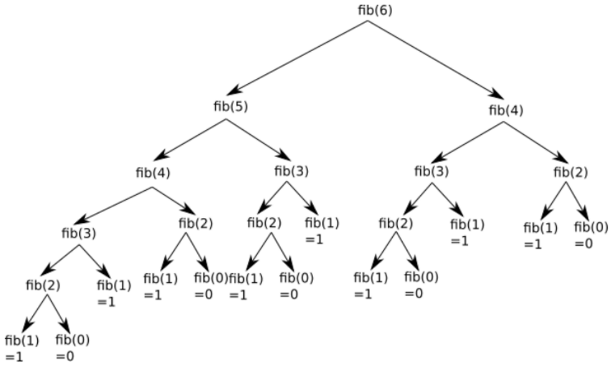
```
{2: 1}
{2: 1, 3: 2}
{2: 1, 3: 2, 4: 3}
{2: 1, 3: 2, 4: 3, 5: 5}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
8
```

Figure: Résultat du dictionnaire et résultat final pour fib(6)

```
102334155
102334155
Took fib(40)     13.113 seconds
Took fib_mem(40) 0.000 seconds
```

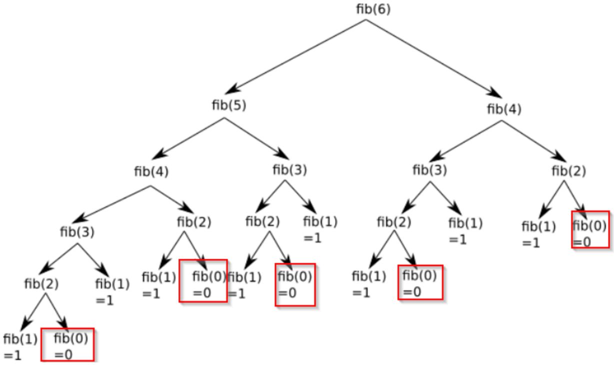
Figure: Benchmarking de fib(40) et fib_mem(40)

Arbre de décision pour fib(6)



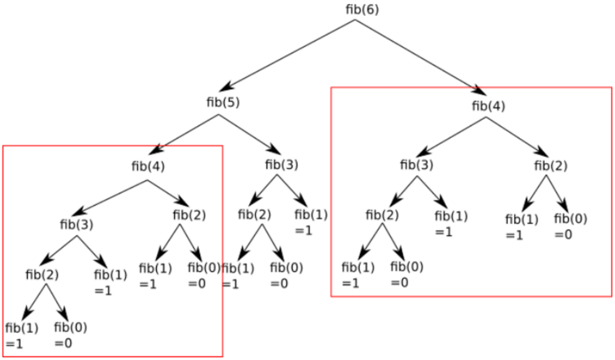
Fibonacci - Arbre de décision

Combien de fois calcule-t-on fib(0) ? $\Longrightarrow$ 5



Fibonacci - Arbre de décision

Combien de fois calcule-t-on fib(4) ? $\implies$ 2



Fibonacci - Programmation Dynamique - Astuce

L'astuce : après la première branche de fib(4) visitée, le résultat est retrouvé par la clé (n) dans le dictionnaire (la cache de mémorisation) lors dans la deuxième occurence de fib(4).

Le principe est le même pour tout les n ! (qui ne sont pas des cas de base). Peu importe si la cache est un tableau, une matrice, etc.

```
{2: 1}
{2: 1, 3: 2}
{2: 1, 3: 2, 4: 3}
{2: 1, 3: 2, 4: 3, 5: 5}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
8
```

Figure: Partie annotée rouge : à gauche la clé représentant fib(4), à droite la valeur étant le résultat de fib(4) sauvegardé depuis la première occurence $\Longrightarrow$ 3

```
# Search in the dictionary for branches already computed
if n in memo:
    return memo[n]
```

Figure: La valeur $n = 4$ de fib(4) est utilisé comme clé dans le dictionnaire (cache de mémorisation)

Bibliographie

- https://coin-or.github.io/pulp/main/index.html