Travaux pratiques : Hypergraphes

Comme vu en cours, l'intérêt du formalisme des hypergraphes pour la programmation dynamique est qu'il permet de séparer l'espace de recherche de l'application algorithmique. Au cours de ce TP, nous allons illustrer cette séparation en codant tout d'abord quelques algorithmes génériques vus en cours, puis en construisant des espaces de recherches associés à deux problèmes classiques (Alignement par l'algorithme de Needleman-Wunsch et repliement avec Nussinov).

`hypergraph.py` – Une bibliothèque minimale d'hypergraphes

`http://www.lix.polytechnique.fr/~ponty/enseignement/hypergraph.py`

On a recodé une petite bibliothèque permettant de manipuler des hypergraphes :

- Chaque **sommet** sera identifié par son **nom**. Celui-ci pourra être de n'importe quel type [1], mais devra nécessairement être **unique** et **hashable** [2].
- Les **sommets** sont créés *à la volée* quand des hyper-arêtes sont crées.
- Un **sommet** s deviendra *terminal* quand un arc $s \rightarrow$ `[]` est ajouté au graphe.
- Un **arc** peut être ajouté directement via la méthode `addFArc` de la classe `FGraph`. Les arguments à fournir à celle-ci sont :
  - `origin` : Le nom du sommet à l'origine de l'arc
  - `destinations` : Une liste de noms de sommets destination de l'arc
  - `weight` : Un poids optionnel (Default : `0.0`)
  - `type` : Un type optionnel (Default : `None`)

  Ces données peuvent être récupérées via des méthodes `getOrigin`, `getDestinations`, `getWeights` et `getType` respectivement.
- Les **arcs** ayant pour origine un sommet s donné peuvent être obtenus via la méthode `getOutList` de la classe `FGraph`.

N'hésitez pas à aller regarder dans les commentaires du code source les options disponibles.

Exemple de code

```
g = FGraph()
g.addFArc(0,[1])
g.addFArc(0,[2,3])
g.addFArc(1,[])
g.addFArc(2,[])
g.addFArc(3,[])
print g
print g.getOutList(1)
for e in g.getOutList(0):
    for name in e.getDestinations():
        print name
```

Sortie

```
Vertices:
[0, 1, 2, 3]
Arcs:
  0->[1](w:0.0)
  0->[2, 3](w:0.0)
```

1. C'est mal, je sais, mais bien trop pratique pour pouvoir s'en passer ici. Ne le faites pas à la maison !
2. Cette dernière contrainte n'est pas très forte, car la plupart des types primitifs (ceux dits *immutables*) sont hashables. En particulier, vous pourrez utiliser des `strings`, `int`, `float` ..., des `tuple`, mais pas des `list` ou `dict` comme nom pour les sommets ! !

```
  1->[](w:0.0)
  2->[](w:0.0)
  3->[](w:0.0)

[1->[](w:0.0)]
1
2
3
```

La mémoïsation : le rêve du programmeur paresseux ...

On simplifiera l'implémentation des applications en utilisant une **mémoïsation**, terme qui décrit la *mise en cache* automatique par le langage des valeurs retournées par une fonction. L'avantage de cette technique est qu'elle vous évite d'avoir à vous soucier de l'ordre dans les calculs, au prix d'une dégradation (constante) des performances.

Comme cette fonction n'est pas présente par défaut en `python`, on a du l'émuler grâce à un `decorator memoize` implémenté au sein du fichier `hypergraph.py`. Plus précisément la fonction de fibonacci calculée **en temps exponentiel** par ce code :

```
def fib(n):
  if n<=1: return 1
  return fib(n-1)+fib(n-2)
```

sera calculée **en temps linéaire** par ce code :

```
from hypergraph import *
@memoize
def fib(n):
  if n<=1: return 1
  return fib(n-1)+fib(n-2)
```

Objectifs du TP

1. Coder dans un fichier `applications.py` une fonction `count(hypergraph,name)`, qui compte les F-chemins originaires d'un sommet `name` dans un hypergraphe `hypergraph`. Vous utiliserez la technique de mémoisation décrite ci-dessus.
   Tester votre fonction grâce au code (`testHypergraph` est défini dans `applications.py`) :

   ```
   if __name__=="__main__":
       g = testHypergraph()
       print count(g,0)
       # Affiche : 12
   ```

2. Ajouter une fonction `maximalScore(hypergraph,name)` qui calcule et renvoie une paire `(path,score)`, où `path` est le F-chemin de poids maximal, représenté par une liste de F-Arcs, et score le score associé.

   ```
   if __name__=="__main__":
       g = testHypergraph()
       print maximalScore(g,0)
   ```

```
4    # Affiche : ([0->[2](w:0.0), 2->[5, 7](w:2.0), 5->[](w:1.0),
5    #            7->[](w:0.0)], 3.0)
```

3. Coder dans un nouveau fichier `hypergraphNW.py` une fonction `buildAlignmentGraph(seq1,seq2)` qui construit l'hypergraphe associé à l'alignement de deux séquences `seq1` et `seq2`. On utilisera une pénalité de -2 pour les insertions/déletions, de -3 pour les mismatch et un bonus de +2 pour les correspondances. On prendra soin de bien définir les sommets terminaux.
4. En outre, on codera une fonction affichant le chemin obtenu comme un alignement classique (il est possible d'associer un type à chaque arc, ce qui devrait faciliter la mise en forme de l'alignement).
5. Ajouter à `applications.py` une fonction `minimalScore(hypergraph,name)` qui calcule le chemin/score minimal, et coder dans un nouveau fichier `hypergraphNussinov.py` une fonction `buildNussinovGraph(seq)` qui construit l'hypergraphe associé au repliement d'une séquence d'ARN `seq`. On utilisera des énergies de -3/-2/-1 kcal.Mol^{-1} respectivement pour les paires (G/C,C/G), (A/U,U/A) et (G/U,U/G). Vérifiez votre implémentation sur les exemples du cours numéro 1.
6. Coder une petite fonction permettant la mise en forme de la structure secondaire sous la forme d'une notation bien parenthésée.
7. Ajouter à `applications.py` une fonction `weightedCount(hypergraph,name)` qui implémente le comptage pondéré vu en cours. Ajouter au fichier `hypergraphNussinov.py` une fonction `buildBoltzmannNussinovGraph(seq)` qui construit l'hypergraphe de Nussinov, pondéré par des facteurs de Boltzmann $e^{E/RT}$.
8. Ajouter une fonction de génération aléatoire pondérée `rangGen(hypergraph,name)` qui engendre un F-chemin avec probabilité proportionnelle à son poids, suivant les principes vus en cours. On pourra tester l'absence de biais sur l'hypergraphe renvoyé par la fonction `testHypergraph()`, qui contient exactement 12 chemins.
9. Ajouter enfin à `applications.py` une fonction `arcProba(hypergraph)` qui calcule les probabilités associées à tous les F-arcs de l'hypergraphe `hypergraph`.
10. De part le choix d'implémentation dans `applications.py`, les arcs sont stockés un par un et de façon explicite. La complexité en mémoire des algorithmes est donc égale à celle en temps (et est égale à la taille de l'hypergraphe). Proposez une représentation compact pour économiser de l'espace.