Cahier des charges du projet Benchtrack

Alexis Lenoir - Xin He - Zhuangzhuang Yang

Mercredi 24 février 2021

1 Contexte et présentation

L'origine de ce projet part du constat qu'il est difficile de comparer des implémentations différentes d'algorithmes avancés. En effet, celles-ci dépendent de nombreux facteurs comme une grande diversité dans les formats d'entrée et de sortie, des conditions d'utilisation et plus simplement du langage de programmation utilisé.

Il existe déjà plusieurs frameworks qui s'occupe de gèrer des tests sur des implémentations, comme le module *pytest* et la solution logicielle *Jenkins*. Cependant ils s'attachent à tester pour s'assurer de la validité du code, ils interviennent dans son élaboration, l'objectif du projet se place en aval. En effet, le but du projet n'est pas de développer des implémentations, mais en comparer des préexistantes. Le maître mot est donc comparaison.

L'objectif du projet est donc d'élaborer un framework python, le plus générique possible, de comparaison d'implémentation d'algorithmes complexes qui permettra d'observer et de comparer les perfomances des différentes implémentations sur différents critères. Le framework générera ainsi un *benchmark*, c'est-à-dire un banc d'essai qui permet d'évaluer la pertinence d'une implémentation à l'aide de comparaisons. Ce benchmark sera ici un site statique qui contiendra tous les résultats des comparaisons. Ce site de comparaison devra pouvoir être mis à jour automatiquement et régulièrement.

2 Besoins et contraintes

Le problème est formalisé de la manière suivante. Le but est de générer un *benchmark* pour une *infrastructure de test*. Cette infrastructure de tests est caractérisée par un ensemble d'implémentation d'algorithme, que l'on appelera *target* ; ainsi qu'un ensemble de tâches(*tasks*). Une *task* rassemble un ensemble de *targets* qui décrivent une tâche commune. Les tasks sont eux-même regroupés par thème (*theme*). Le regroupement des *targets* est assez structuré mais il sera toujours possible d'indiquer un nom par défaut. De ce formalisme découle naturelement une arborescence *infrastructure_de_test/theme/task/target*.

Une problématique majeure est le formalisme adopté : il doit être clair et efficace, tout en restant le plus générique possible.

L'intérêt de ce benchmark dépend fortement des critères pris en compte. Le plus trivial est la vitesse d'exécutions. Cependant, il n'est pas chose aisée de la calculer.

Comme il a été dit précédemment, le *benchmark* sera impérativement un site statique, ce framework produira donc un fichier html.

Afin d'optimiser le développement, sera adopté une approche modulaire : le projet sera séparé en deux. La première partie consitera à générer les résultats dans un fichier *benchmark.csv*. La deuxième partie produira le benchmark à l'aide de *benchmark.csv*.

Les benchmarks seront générés à l'aide d'un script python.

2.1 Description générale d'une *infrastructure de test* :

La structure principale

Ce framework prend en entrée une *infrastructure de test*, ce sera un répertoire. Les *themes* et les *tasks* seront représentés par des sous-répertoires. Les *targets* seront représentés par des fichiers de codes.

Dans le répertoire de l'infrastructure de test, il y aura deux sous-répertoires : *targets*(voir les fichiers annexes) et *tasks* qui contiendra les repertoires *themes*, qui eux-même contiendront les répertoires *tasks*. Chaque *target* qui décrit une *task* sera présent dans le sous-répertoire correspondant.

Les fichiers annexes

Après avoir abordé les données brutes, il y aussi les fichiers indispensables de configuration et de documentation. Dans le répertoire de l'*infrastructure de test* se trouvera un répertoire *targets*. Pour chaque *target* sera associée un sous-répertoire */nom_infrastructure_test/targets/nom_target*. Il contiendra un fichier de configuration *config.ini* qui précisera le langage :

```
langage = nom_langage
```

Et un fichier *read.rst* qui décrira la target.

Pour chaque *task*, on trouvera dans le répertoire associée un sous-répertoire *data* qui contiendra éventuellement les données d'entrée de la tâche, tous les formats seront acceptées. Un fichier *read.rst* décrivant la *task* sera aussi présent dans le répertoire.

Format général :

```
/nom_infrastructure_test
    /targets
        /target1
            /config.ini
            /read.rst
        /target2
            /config.ini
            /read.rst
    /tasks
        /theme1
            /task1
                /read.rst
                /data
                /target1.py
                /target2.r
            /task2
                /read.rst
                /data
                    data_task2.csv
                /target2.r
        /theme2
            /task3
                /read.rst
                /data
                /target1.py
                /target2.r
```

2.2 Exemple d'une *infrastructure de test* : *PGM*

L'infrastructure de test *PGM* est donc un répertoire.
Les *themes* sont :

- Learning
- Inference
- Modelisation

Les *themes* regroupent les *tasks* de la manière suivante :

1. Learning (*theme*)
   (a) parameterLearning (*task*)
   (b) parameterLearningWithMissingValues (*task*)
   (c) structuralLearning (*task*)
2. Inference (*theme*)
   (a) exactInference (*task*)

   (b) multipleExactInference (*task*)
   (c) approximatedInferenceGibbsSampling (*task*)
3. Modelisation (*theme*)
   (a) modelingAsia (*task*)
   (b) modelingAlarm (*task*)
   (c) modelingPRM (*task*)

Les *targets* sont par exemple les librairies :
- pyAgrum (Python)
- pyPGM (Python)
- BNLearn (R)

On a ainsi l'arborescence suivante :

```
/PGM
    /targets
        /pyAgrum
            /config.ini
            /read.rst
        /pyPGM
            /config.ini
            /read.rst
        /BNLearn
            /config.ini
            /read.rst
    /tasks
        /learning
            /parameterLearning
                /read.rst
                /pyAgrum.py
                /pyPGM.py
                /BNLearn.r
            /parameterLearningWithMissingValues
                /read.rst
                /pyAgrum.py
                /BNLearn.r
            /structuralLearning
                /read.rst
                /pyAgrum.py
                /pyPGM.py
                /BNLearn.r
        /inference
            /exactInference
                /read.rst
                /pyAgrum.py
                /pyPGM.py
            /multipleExactInference
```

```
            /read.rst
            /pyAgrum.py
            /pyPGM.py
        /approximatedInferenceGibbsSampling
            /read.rst
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
    /modelisation
        /modelingAsia
            /read.rst
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
        /modelingAlarm
            /read.rst
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
        /modelingPRM
            /read.rst
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
```

3 Objectifs d'une première version

3.1 Fonctionnalités générales :

Un script python qui en prenant en paramètre une infrastructure de test(un repertoire) génère un fichier html(benchmark).

Les critères utilisées dans l'étude sont :
— La vitesse d'éxécution

3.2 Fonctionnement du framework :

Notre framework sera rassemblé dans le fichier *benchmark.py*. Il se servira de deux autres fichiers : *benchmark_results.py* et *benchmark2html.py*. On utilisera le framework de la manière suivante :

```
python benchmark.py infrastructure_de_test
```

où *infrastructure_de_test* sera un répertoire respectant le formalisme qui a été définit précédemment.
Plus précisément, *benchmark.py* appellera *benchmark_results.py* avec l'infrastructe de test et obtiendra le fichier *benchmark.csv* généré.

Il appellera ensuite *benchmark2html.py* avec *benchmark.csv* pour produire *benchmark.html*. Le fichier csv sera en fait d'abbord transformé en un fichier rst. Ensuite à l'aide de la librairie *pelican* sera produit le fichier html.

3.3 Le critère de vitesse d'exécution :

Dans un premiers temps, on calculera la vitesse d'exécution en prenant simplement la soustraction de deux temps, l'un pris avant l'exécution et l'autre après. Cette opération sera répétée plusieurs fois pour augmenter la précision.

4 Améliorations futures

Le framework pourra posséder les améliorations suivantes :

- Augmenter le nombre de critère dans l'évaluation de la comparaison
- Prendre en compte plus de langages : Java, C++, Julia ...
- Améliorer l'affichage du site statique
- Conserver en mémoire les résultats, pour les comparer entre eux, montrer ainsi l'évolution
- Personnaliser d'avantage l'interaction avec le framework en augmentant les commandes possibles avec le terminale :

```
python benchtrack.py --ouput HTML_FOLDER
python benchtrack.py --verbose
python benchtrack.py --targets-list
python benchtrack.py --tasks-list
python benchtrack.py --target-info TOTO
python benchtrack.py --task-info A
python benchtrack.py --targets-include=TOTO,TITI
python benchtrack.py --tasks-include=A,B,C
python benchtrack.py --targets-exclude=TOTO,TITI
python benchtrack.py --tasks-exclude=A,B,C
```

- Faciliter la création d'une infrastructure de test, en générant automatiquement l'arborescence à partir d'un fichier de configuration.
- Pouvoir vérifier la précision des résultats obtenus, en connaissant les résultats attendus qui seraient sauvegardés dans un répertoire *expected*

5 Délai

Une première version fonctionnelle doit être achever avant le lundi 1er mars 2021.
La dernière version de notre projet doit être réaliser avant le lundi 17 mai 2021.