Metrics : Mission Expérimentations

Thibault Falque[1] **Romain Wallon**[2] **Hugues Wattez**[3]

[1] Exakis Nelite

[2] LIX, Laboratoire d'Informatique de l'X, École Polytechnique, Chaire X-Uber

[3] CRIL, Univ Artois & CNRS

thibault.falque@exakis-nelite.com wallon@lix.polytechnique.fr wattez@cril.fr

Résumé

Le développement de solveurs de contraintes s'accompagne nécessairement d'une phase d'expérimentation, permettant d'évaluer les performances des fonctionnalités implantées. Pour chacun des solveurs considérés, il faut alors collecter et analyser des statistisques relatives à leur exécution et qui, le plus souvent, restent les mêmes. Néanmoins, il existe probablement autant de scripts permettant d'extraire ces statistiques et de tracer les figures associées que de chercheurs du domaine. Un outil unifiant et facilitant les analyses de résultats expérimentaux paraît donc nécessaire, notamment pour favoriser le partage et la reproductibilité de ces résultats. Partant de ce constat, nous avons développé la bibliothèque *Metrics*, dont l'ambition est de fournir une chaîne complète d'outils conçue dans cette optique. Dans cet article, nous présentons cette bibliothèque et illustrons son utilisation en rejouant l'analyse des résultats de la compétition XCSP'19.

Abstract

Developing constraint solvers comes with the need to perform experiments, in order to evaluate the performance of the implemented features. For each considered solver, one needs to collect and analyze statistics about its execution which, most of the time, remain the same. However, there exist probably as many scripts allowing to extract such statistics and to draw the associated figures as researchers in the domain. A unifying tool making easier the analysis of experimental results seems thus required, especially to favor the sharing and the reproducibility of these results. Based on this observation, we have developed the *Metrics* library, whose ambition is to provide a complete toolchain designed in that purpose. In this paper, we present this library and illustrate its use through the analysis of the results of the XCSP'19 competition.

1 Introduction

Dans le cadre de la recherche en informatique (et plus généralement, dans n'importe quel domaine nécessitant la conception de logiciels), il est nécessaire de réaliser des expérimentations pour s'assurer que les programmes développés fonctionnent comme prévu. En particulier, il est important de vérifier que les ressources utilisées par ces programmes restent raisonnables. Dans ce but, différentes solutions logicielles telles que *runsolver* [7] ont été proposées, à la fois pour mesurer et limiter l'utilisation des ressources temporelles et spatiales par le programme en cours d'évaluation. Cependant, respecter les limites fixées est en général insuffisant pour évaluer le comportement du programme. Il est souvent nécessaire de collecter des statistiques supplémentaires, qui peuvent être fournies par le programme lui-même (par exemple, via ses *logs*) ou par l'environnement d'exécution (par exemple, *runsolver*). Les données collectées doivent ensuite être aggrégées pour évaluer la qualité des résultats du programme au travers d'une analyse statistique.

Dans ce domaine, de nombreux outils mathématiques peuvent être utilisés, et faire un choix parmi ceux-ci peut introduire des biais dans les résultats ou leur analyse. Il est donc important d'appliquer les principes de *science ouverte* et de *reproductibilité* pour permettre de rejouer l'analyse des résultats. Ainsi, ces principes ont fait l'objet d'une recommendation de l'OCDE [6], et des chercheurs ont déjà introduit diverses approches favorisant la reproductibilité dans le contexte d'expérimentations logicielles [3, 5]. En particulier, il est recommandé de rendre disponible le code source des logiciels étudiés (ou, à minima, leurs exécutables sous forme binaire), et de diffuser les données utilisées pour les évaluer (par exemple, dans des forges logicielles). Il

est également important de rendre l'analyse des résultats reproductibles, en utilisant des outils tels que le *RMarkdown* ou les *notebooks Jupyter*.

Dans la communauté CP (tout comme dans les communautés SAT, PB, MaxSAT ou QBF par exemple), il y a souvent peu de différences sur la manière d'exécuter les solveurs. En effet, ces derniers doivent proposer des interfaces en lignes de commande qui respectent les règles imposées par l'environnement dans lequel ils sont exécutés (par exemple, durant les compétitions). De plus, la plupart des données collectées lors de l'exécution des solveurs restent le plus souvent les mêmes (par exemple, le temps d'exécution, l'utilisation de la mémoire, etc.). Dans ce contexte, la création d'un outil permettant d'exécuter le programme, de collecter les données qu'il produit et de les analyser présenterait plusieurs avantages : tester de nouvelles fonctionnalités serait plus simple, à la fois en termes d'exécution et d'analyse, et la reproductibilité des résultats serait systématiquement assurée.

Partant de ces observations, cet article présente *Metrics* (*mETRICS* signifie *rEproducible sofTware peRformance analysIs in perfeCt Simplicity*), une bibliothèque Python visant à unifier et faciliter l'analyse des expérimentations réalisées sur des solveurs. L'ambition de *Metrics* est de fournir une chaîne complète d'outils de l'exécution du solveur à l'analyse de ses performances. Actuellement, cette bibliothèque possède deux composants principaux : *Scalpel* (*sCAlPEL* signifie *extraCting dAta of exPeriments from softwarE Logs*) et *Wallet* (*wALLET* signifie *Automated tooL for expLoiting Experimental resulTs*). D'une part, *Scalpel* est conçu pour simplifier la récupération des données expérimentales. Ce module est capable de gérer une grande variété de fichiers, incluant les formats CSV, XML, JSON ou encore la sortie produite par le solveur, qui peut être décrite par l'utilisateur dans un fichier de configuration. Cette approche fait de *Scalpel* un outil à la fois flexible et simple à configurer. D'autre part, *Wallet* propose une interface simple d'utilisation pour tracer les diagrammes les plus communément utilisés pour l'analyse de solveurs (tels que les *scatter plots* et les *cactus plots*) et pour calculer diverses statistiques relatives à leur exécution (en particulier, leurs scores en utilisant différentes mesures classiques). La conception de *Wallet* facilite l'intégration de l'analyse dans des *notebooks Jupyter* qui peuvent être facilement partagés en ligne (par exemple, *GitHub* ou *GitLab* sont capables d'afficher de tels fichiers), favorisant également la reproductibilité de l'analyse.

Dans cet article, nous présentons l'utilisation de *Metrics* de la manière suivante. En guise de préliminaires, nous introduisons les différentes notions de vocabulaires utilisées par la suite. Nous enchaînons avec la présentation des différentes étapes d'une analyse expérimentale réalisée avec *Metrics*, de l'extraction des données au tracé des diagrammes permettant leur exploitation, en rejouant l'analyse des résultats de la compétition XCSP'19 [1]. Nous concluons enfin avec quelques perspectives d'amélioration pour *Metrics*, qui permettront d'en faire une bibliothèque unifiée pour l'expérimentation de solveurs.

2 Vocabulaire de *Metrics*

Cette section introduit quelques notions de vocabulaire utilisées pour identifier les données manipulées par *Metrics* et pour décrire les analyses réalisées par cette bibliothèque.

L'objet central dans *Metrics* est la *campagne*, qui contient toutes les données expérimentales qui ont été collectées, et définit la configuration de l'environnment d'exécution des solveurs (limites temporelle et spatiale, configuration des machines, etc.). Au cours d'une campagne des *expérimentaciels* (ou *experimentwares* en anglais) sont évalués. Ce néologisme est utilisé pour caractériser n'importe quel logiciel pouvant être expérimenté, en guise de notion plus générale que le terme de *solveur*. Notons que différentes configurations d'un même programme sont considérées comme différents expérimentaciels, de même que des programmes complètement différents. Une campagne se caractérise par l'ensemble des fichiers d'entrée (*input-set*) utilisés pour cette campagne. Dans ce contexte, *tous* les expérimentaciels sont exécutés sur le *même* ensemble de fichiers. Enfin, une *expérimentation* correspond à l'exécution d'un expérimentaciel donné sur un fichier d'entrée donné, de sorte que l'ensemble des expérimentations correspond au produit cartésien de l'ensemble des fichiers d'entrée et de celui des expérimentaciels. Chaque expérimentation est caractérisée par les données qui sont pertinentes au vu des analyses à réaliser, telles que le temps d'exécution ou la mémoire utilisée par l'expérimentaciel (et bien d'autres, dépendant de la configuration fournie par l'utilisateur).

Exemple 1. *Considérons une* campagne *dans laquelle nous souhaitons comparer les solveurs* AbsCon *et* Nacre *[4]. Ces deux solveurs sont nos* expérimentaciels. *Supposons que nous souhaitions comparer ces deux solveurs sur deux* entrées, *à savoir* `AllInterval.xml` *et* `CarSequencing.xml`. *L'ensemble des fichiers d'entrée se compose de ces deux fichiers. Les expérimentations de cette campagne sont alors :*

- *l'exécution d'*AbsCon *sur* `AllInterval.xml` ;
- *l'exécution d'*AbsCon *sur* `CarSequencing.xml` ;
- *l'exécution de* Nacre *sur* `AllInterval.xml` ;
- *l'exécution de* Nacre *sur* `CarSequencing.xml`.

3 Présentation détaillée de *Metrics*

Dans cette section, nous proposons une description de notre bibliothèque *Metrics*. En particulier, nous motivons l'intérêt d'une telle bibliothèque, et illustrons son utilisation en rejouant l'analyse des résultats de la compétition XCSP'19 [1].

3.1 Motivation

La bibliothèque *Metrics* a été conçue dans le but de fournir tous les outils nécessaires pour l'analyse de résultats expérimentaux obtenus dans le cadre de l'évaluation de solveurs. Dans la communauté, plusieurs outils sont fréquemment utilisés dans ce but. C'est par exemple le cas de *GNUPlot*, qui est une bonne bibliothèque pour tracer des diagrammes (mais un peu classique), qui ne permet cependant pas de gérer toutes les fonctionnalités envisagées pour *Metrics*. L'utilisation du langage *R* a un temps été envisagée, notamment en raison de son support natif de la reproductibilité au travers du format *RMarkdown*. Son utilisation reste toutefois limitée au sein de la communauté, ce qui peut être un frein pour l'adoption de *Metrics*.

Notre choix s'est donc porté sur le langage Python, qui offre un large écosystème pour l'analyse de données. *Metrics* est en particulier fondé sur deux bibliothèques bien connues dans ce domaine, à savoir *pandas*, qui propose une implantation de *data-frames* nous permettant de gérer et manipuler les données extraites des expérimentations réalisées, et *matplotlib*, que nous utilisons pour tracer des figures statiques (des figures dynamiques sont également proposées grâce à la bibliothèque *plotly*). Concernant les figures, nous notons que des outils ont déjà été implantés en Python par la communauté, comme par exemple *mkplot*[1], mais ceux-ci ne fournissent pas une chaîne complète d'outils comme *Metrics* souhaite le faire.

3.2 *Metrics* en action

Pour présenter les capacités de *Metrics*, nous proposons dans cette section de rejouer l'analyse des résultats de la compétition XCSP'19 [1]. Dans la *main track* de cette compétition, 26 expérimentaciels (ou *solveurs*) ont été soumis et évalués sur 600 entrées (en l'occurence, 300 problèmes de décision et 300 problèmes d'optimisation décrits au format XCSP3 [2]). Chaque expérimentation (c'est-à-dire, chaque exécution d'un solveur donné sur une instance donnée) était limitée à 2400 secondes, et ne pouvait utiliser plus de 16 Go de mémoire pour les solveurs séquentiels, et 32 Go pour les solveurs parallèles.

1. https://github.com/alexeyignatiev/mkplot.git

Le lecteur intéressé pourra retrouver et reproduire l'analyse proposée ci-après en consultant le dépôt *GitHub* de *Metrics*[2].

3.2.1 Extraction des données avec *Scalpel*

La première étape pour analyser les résultats est d'extraire les données nécessaires à l'analyse. C'est le rôle du module *sCAlPEL* (« *extraCt dAta of exPeriments from softwarE Logs* ») fourni par *Metrics*, qui construit la représentation d'une campagne à partir d'un ou plusieurs fichiers produits au cours de celle-ci.

Plus précisément, *Scalpel* est capable de lire nativement différents formats de fichiers classiquement utilisés, tels que le format CSV (et ses variantes), ainsi que les formats JSON et XML (en utilisant une notation pointée pour identifier chacun des attributs apparaissant dans le fichier). *Scalpel* est également capable d'extraire des données directement depuis la sortie des solveurs, sans que ceux-ci n'aient à respecter un format particulier. Dans ce dernier cas, *Scalpel* permet notamment d'explorer une hiérarchie complète de fichiers de log, en extrayant les données de chacun des fichiers pertinents rencontrés au cours de cette exploration. Ceci est rendu possible par l'utilisation d'un fichier de configuration, dans lequel is est possible de décrire et de nommer les données pertinentes apparaissant dans cess fichiers, en utilisant des expressions régulières, ou des motifs simplifiés, permettant d'identifier des valeurs communes (comme des valeurs booléennes, entières, réelles, ou des mots).

Dans le cas qui nous intéresse ici, nous procédons à l'analyse des résultats de la compétition XCSP'19 à partir du fichier texte fourni sur le site web de la compétition. Il s'agit d'un fichier ayant un format particulier, que nous appellons `evaluation` dans la suite. Comme ce format est utilisé dans un certain nombre de compétitions, *Scalpel* propose un lecteur natif pour les fichiers de ce type. La configuration de *Scalpel* présentée ici est donc relativement simple, et par souci de concision, nous ne pouvons pas décrire toute la puissance de lecture offerte par ce module. Le lecteur intéressé pourra retrouver plus d'informations sur la documentation de *Metrics*[3].

Construisons maintenant la configuration de *Scalpel* pour le cas particulier de la compétition XCSP'19. Cette configuration s'effectue dans un fichier YAML. Les premières informations à saisir sont celles permettant d'identifier de manière unique la campagne.

```
name: Compétition XCSP 2019
date: 1 Octobre 2019
```

2. https://github.com/crillab/metrics
3. https://metrics.readthedocs.io

Puis, il faut donner la configuration de l'environnement où les solveurs ont été exécutés, ce qui permet de pouvoir facilement reproduire les expérimentations par la suite.

```
setup:
  os: Linux CentOS 7 (x86_64)
  cpu: Intel XEON X5550
  ram: 32GB
  timeout: 2400
  memout: 16384
```

Il faut ensuite préciser où se trouvent les fichiers que *Scalpel* doit lire, au travers du champ `source` du fichier de configuration.

```
source:
  path: path/to/XCSP19.txt
  format: evaluation
  is-success:
    - ${Checked answer} in ["SAT", "UNSAT"]
```

Ici, nous pouvons reconnaître le chemin vers le fichier de la campagne, en l'occurence `path/to/XCSP19.txt`. Notons qu'une liste de fichiers peut également être précisée, auquel cas *Scalpel* extraiera les données de chacun des fichiers de la liste. Ici, le format `evaluation` est précisé pour indiquer à *Scalpel* comment lire le fichier de la campagne, comme l'extension `.txt` est trop vague pour déterminer son format.

Attardons-nous maintenant quelques instants sur la valeur de `is-success` donnée dans cet exemple. Afin de déterminer si une expérimentation donnée s'est terminée correctement, il est possible de saisir des conditions dans un langage d'expressions relativement simple (la liste des conditions étant interprétée conjonctivement). Ici, une expérimentation est considérée comme un succès si la réponse produite par le solveur est soit `SAT`, soit `UNSAT`. Cette information se trouve dans la colonne `Checked answer` du fichier de la compétition.

Considérons maintenant ce qui est probablement la partie la plus importante du fichier de configuration, qui décrit *comment* retrouver les données à partir des fichiers fournis à *Scalpel*. Dans notre exemple, comme il s'agit d'un format reconnu par *Scalpel*, il y a relativement peu d'informations à donner dans le champ `data` présenté ici. La seule information pertinente à considérer est le `mapping`, qui permet d'identifier quelle(s) colonne(s) du fichier correspond(ent) aux données attendues par *Scalpel*.

```
data:
  mapping:
    experiment_ware:
      - Solver name
      - Solver version
    cpu_time: CPU time
    input: Instance name
```

Notons que les champs indiqués dans cet exemple doivent obligatoirement être précisés, comme les noms des colonnes ne respectent pas les conventions de nommage de *Metrics*. De plus, les informations relatives aux expérimentaciels se trouvent dans deux colonnes distinctes, d'où la liste mentionnée ici.

Comme mentionné plus haut, l'analyse de la compétition XCSP'19 ne nécessite pas de lire des fichiers de log produits au cours de la campagne, les données nécessaires ayant déjà été extraites et mises à disposition dans le fichier structuré `XCSP19.txt`. Toutefois, par souci de complétude, nous présentons brièvement comment des fichiers de log pourraient être lus par *Scalpel*. Dans ce contexte les données à extraire doivent être décrite de la manière suivante.

```
source:
  path: path/to/my/campaign
  format: dir
data:
  raw-data:
    - log-data: status
      file: execution.out
      pattern: 's {word}'
    - log-data: cpu_time
      file: execution.out
      regex: 'c real time : (\d+\.\d+)'
      group: 1
```

La configuration présentée ci-dessus permet d'extraire des données à partir d'une hiérarchie de fichiers. Dans cette hiérarchie, les fichiers produits par les solveurs lors d'une expérimentation sont placés dans un répertoire dédié à cette expérimentation. Dans ce répertoire, le fichier qui va nous intéresser est `execution.out`, qui contient deux données qui vont être extraites par *Scalpel* : le statut de la résolution (décrit via un motif simplifié) et le temps CPU de l'exécution du solveur (décrit via une expression régulière). Notons qu'il est également possible d'extraire des informations à partir du nom des fichiers ou des répertoires explorés, en utilisant le champ `file_name_meta` et en appliquant une configuration similaire à celle illustrée ici.

D'autres informations relatives à la campagne peuvent par ailleurs être précisées dans ce fichier, comme par exemple la liste des expérimentaciels et celle des entrées considérées dans la campagne. Par défaut, il n'est pas nécessaire de les préciser : *Scalpel* les découvrira automagiquement en lisant les fichiers de la campagne. Dans le cas de la compétition XCSP'19, il n'est donc pas nécessaire de préciser ces informations.

Nous remarquons cependant que cela peut s'avérer utile dans le cas où des informations pertinentes relatives aux expérimentaciels ne sont pas disponibles dans les fichiers fournis à *Scalpel* (par exemple, l'empreinte SHA des logiciels, leur ligne de commande, etc.).

Ceci termine la présentation du module *Scalpel* de *Metrics*. Conscients de la violence des informations présentées dans cette section, nous préférons vous montrer cette illustration, consacrée au logo de *Metrics*, avant de passer à la suite.

FIGURE 1 – Le logo de *Metrics*. Contrairement à une idée largement répandue, le panda et le python s'entendent très bien, notamment pour développer une bibliothèque plus humaine.

3.2.2 Exploiter les données avec *Wallet*

Une fois les données extraites, analysons-les ![4] C'est le rôle du module *wALLET* (*« Automated tooL for expLoiting Experimental resulTs »*) fourni par *Metrics*. Il permet, entre autres, de calculer des statistiques et de tracer des diagrammes communément utilisés dans le cadre de l'expérimentation de solveurs. *Wallet* est principalement conçu pour être utilisé au sein d'un *notebook Jupyter*, mais il est parfaitement envisageable de l'utiliser dans n'importe quel environnement Python convenablement configuré.

Wallet est capable de produire différents types de figures. Un premier type correspond aux graphiques statiques communément utilisés par la communauté, notamment les *scatter plots* et *cactus plots*, ainsi que leurs homologues, les *CDF plots*, que nous détaillons plus bas, et les *box plots* (aussi appelées *boîtes à moustaches*). Ces graphiques sont générés en interne par la bibliothèque *matplotlib* et peuvent être exportées sous différents formats tels que des images PNG ou vectorisées (SVG ou EPS). Ces graphiques statiques se veulent hautement personnalisables afin de s'adapter à leur environnement final (présentation, article, etc.). Entre autres, il est possible de modifier la police d'écriture, d'interpréter des formules LaTeX, de lier les expérimentaciels à différentes couleurs et différents styles, de personnaliser la légende, etc.

Un second type de figures correspond aux versions dynamisées des figures mentionnéees ci-dessus, générées grâce à la bibliothèque *plotly*. Cette bibliothèque rend possible des interactions avec l'utilisateur, par exemple, en permettant de zoomer en avant ou en arrière sur les graphiques, de sélectionner un sous-ensemble de la légende ou d'afficher des données supplémentaires par le biais d'événements tels que le survol de la souris. L'utilisateur peut aussi personnaliser ces graphiques à travers l'interface proposée par les *notebooks Jupyter* et les différents paramètres mis à disposition lors de la création des graphiques.

Enfin, *Wallet* propose aussi différentes figures sous forme tabulaire, affichant diverses statistiques discriminant les expérimentaciels sous plusieurs points de vue. Ces données tabulaires sont exportables en LaTeX.

Manipulation des données Afin de réaliser l'analyse, la première étape est de charger la campagne grâce au fichier de configuration de *Scalpel* préparée dans la précédente section :

```
from metrics.wallet import Analysis
analysis = Analysis(
  'path/to/config.yml', [...]
)
```

Une *analyse* est un objet *Metrics* permettant de gérer et de manipuler simplement les expérimentations. Par souci de clarté et de concision, nous ne présentons ici qu'une forme simplifiée [5]. Il est néanmoins possible, grâce à des informations supplémentaires fournies par l'utilisateur, de vérifier la cohérence des résultats obtenus, à la fois pour chaque expérimentation individuelle et à l'échelle d'une instance. Par exemple, il serait incohérent d'observer qu'un solveur trouve une solution pour une instance, et qu'un autre solveur identifie cette même instance comme insatisfaisable. Naturellement, l'analyse remplace les expérimentations manquantes ou incohérentes par leurs ajouts invalidés et interprétés

4. « C'est un alexandrin »

5. L'analyse complète est ici : https://github.com/crillab/metrics/tree/master/example/xcsp-19/

comme un *timeout*. L'utilisateur est prévenu de toute opération appliquée sur l'analyse et peut vérifier les instances et expérimentaciels concernés.

Une fois cette première étape consolidant les données de l'analyse effectuée, l'utilisateur peut enfin les manipuler. Dans le cadre de la compétition XCSP'19, et souhaitant nous attarder sur l'analyse de la résolution *séquentielle* de CSPs (*Problème de Satisfaction de Contraintes*), une première opération doit être réalisé :

```
analysis_no_para = analysis.filter_analysis(
    function=lambda x:
      'parallel' not in x['experiment_ware']
)
```

Cette opération permet de filtrer l'ensemble des solveurs comprenant le mot-clé *parallel* dans leur nom et de ne garder que les solveurs séquentiels. Une autre manipulation, extrayant la famille des instances, permettra d'analyser par la suite plus finement la campagne. Ici, nous produisons une nouvelle variable (une nouvelle métrique) permettant d'associer directement chacune des instances à sa famille :

```
import re
family_re = re.compile(r'^XCSP\d\d/(.*?)/')

new_analysis = analysis.add_variable(
    new_var='family',
    function=lambda x:
      family_re.match(x['input']).group(1)
)
```

Les instances de la compétition étant décrites sous forme de lien où chacune d'elles se trouve dans le dossier de sa famille, il est donc possible d'appliquer une expression régulière pour récupérer la donnée intéressante. Ainsi, la méthode `add_variable` permet de produire un nouvelle variable `family` stockant désormais la famille de l'instance. Nous proposons une dernière manipulation de l'analyse actuelle en créant un solveur virtuel :

```
from metrics.wallet
  import find_best_cpu_time_input

analysis_plus_vbs =
  analysis.add_virtual_experiment_ware(
      function=find_best_cpu_time_input,
      name='VBS'
  )
```

Parmi les solveurs virtuels, nous pouvons citer le *VBS* (pour *Virtual Best Solver*), obtenu ici en renseignant la fonction `find_best_cpu_time_input`. De manière générale, un solveur virtuel est un solveur composé d'expérimentations de solveurs réels (pour chaque instance, l'expérimentation du solveur souhaité est choisie). La particularité de la fonction importée est de choisir, pour chaque instance, l'expérimentation d'un solveur réel ayant produit le meilleur résultat (le temps le plus faible dans ce cas). L'utilisateur est libre de personnaliser cette fonction afin de produire un solveur virtuel selon d'autres critères.

Comme il n'est pas possible de présenter l'entièreté des manipulations d'analyse mise à disposition, nous résumons celles-ci par la capacité à :

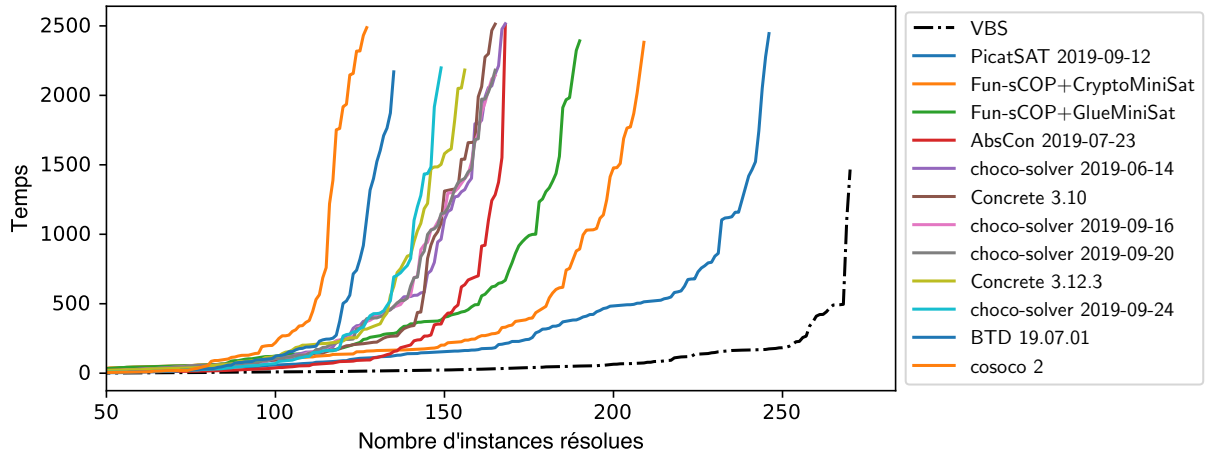
- ajouter et supprimer des variables ;
- ajouter une analyse ou une data-frame déjà existante tout en appliquant les précédents tests de cohérences ;
- filtrer l'analyse (par exemple, en supprimant des solveurs ou des instances) ;
- grouper l'analyse (par exemple, par famille) et sous-analyses ;
- produire autant d'analyse que de paires de solveurs pour les analyser plus finement.

Génération des Figures Une fois notre analyse entièrement configurée, nous pouvons construire l'ensemble des figures proposées par *Metrics*. Tout d'abord, faisons un tour d'horizon des solveurs soumis à la compétition avec un cactus plot. Pour ce faire, il suffit d'appeler la méthode `cactus_plot()` depuis l'analyse de la campagne :

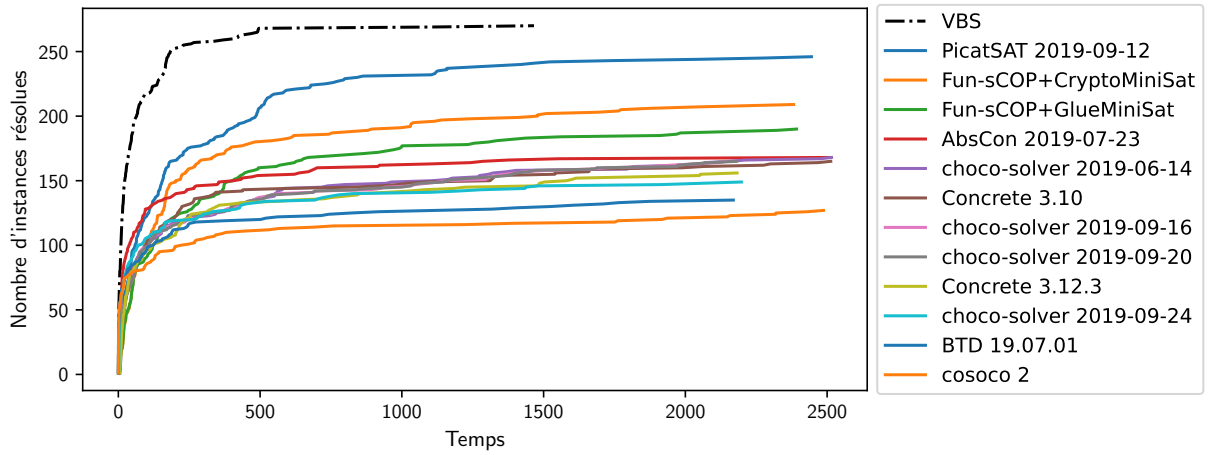
```
analysis.cactus_plot(
  cactus_col='cpu_time',
  x_min=75,
  [...]
)
```

Ici, nous avons explicitement défini `cactus_col` à `cpu_time` (la valeur par défaut) pour montrer la capacité du cactus plot (et des autres graphiques) à adapter son comportement à tout type de valeur numérique. Au-delà de ce comportement, le cactus plot et les autres graphiques produits par *Wallet* sont configurables par le biais de nombreux paramètres. En outre, il est possible de :

- changer le titre du graphique et de ses axes ;
- modifier les limites des axes x et y et changer leur échelle (en logarithmique par exemple) ;
- adapter la taille de la figure ;
- afficher des marqueurs (personnalisables selon le graphique) ;
- associer le nom des solveurs à une couleur ou un motif de tracé de courbe ;
- modifier la police d'écriture (famille, taille, couleur) ;
- interpréter du LaTeX.



(a) Cactus plot des solveurs soumis à la compétition XCSP'19.



(b) CDF plot des solveurs soumis à la compétition XCSP'19.

FIGURE 2 – Cactus plot vs. CDF plot [6]

La Figure 2a nous présente une vue globale des solveurs séquentiels soumis et expérimentés pour la compétition XCSP'19. Nous pouvons observer distinctement l'ensemble des solveurs et noter l'avance qu'a le gagnant de la compétition : le solveur *PicatSAT*. Avec plus d'avance, nous pouvons observer le VBS ayant récupéré les meilleurs contributions de chaque solveur.

Plus le nombre de solveurs est important, plus le graphique est complexe à interpréter. Dans le cas où cela se produit, il existe deux manières de répondre à ces complications. D'une part, *Wallet* est doté d'un autre module graphique permettant de passer d'un modèle statique à un modèle dynamique par le simple changement du paramètre `dynamic` à `True`. Une fois ce passage fait, le modèle dynamique dévoile des outils supplémentaires :

— ajouter ou supprimer des solveurs par simple clic sur leur nom dans la légende ;
— zoomer en avant ou en arrière sur une partie intéressante du graphique ;
— survoler pour obtenir des informations supplémentaires (le temps CPU exact, le nom d'une instance, etc.).

Notons que l'ensemble des graphiques statiques ont leur homologue dynamique. Une autre possibilité pour améliorer la lisibilité des graphiques est de restreindre l'analyse des solveurs au sous-ensemble de ces derniers qui nous intéresse dans le cadre de l'analyse.

De manière équivalente, le CDF plot (*Cumulative Distribution Function*) montre les mêmes résultats avec les axes x et y inversés et le nombre d'instances résolues (Figure 2b). Le CDF est intéressant car il est plus communément utilisé dans d'autres communautés et sa lecture est assez naturelle : l'ordre des courbes suit exactement l'ordre de la légende. Comme dit précédemment, il est également important de considérer le temps

d'exécution des différents solveurs afin de diversifier les informations à considérer. Nous pouvons observer sur ces deux graphiques que *PicatSAT* est nettement devant avec presque 250 instances de résolues.

L'extraction des métriques n'étant pas précise à travers les graphiques, il est aussi intéressant d'observer plus en détails celles-ci à travers le format tabulaire :

```
analysis.stat_table([...])
```

Par souci de concision, nous omettons certains paramètres, qui permettent de personnaliser les nombres en les entourant de dollars et en séparant les ordres de grandeur de 10^3 par des virgules. Dans leur finalité, ces tableaux sont aussi exportables au format LaTeX.

Le Tableau 1 montre, de haut en bas, les solveurs ordonnés par le nombre d'instances résolues. Évidemment, nous pouvons observer que le VBS possède le meilleur classement. Il est suivi par les solveurs *PicatSAT* et *Fun-sCOP+CryptoMiniSat*. Ce Tableau est composé d'une diversité de colonnes présentant le nombre de résolution (`count`), la somme du temps CPU (`sum`), les méthodes `PARx` pénalisant les instances non-résolues pour un solveur en multipliant son timeout par x (de ce fait, `sum` et `PAR1` sont semblables), le nombre d'instances communément résolues (`common count`) et le temps CPU pour les résoudre (`common sum`), `uncommon count` étant le nombre d'instances qu'au moins un solveur n'a pas su résoudre (celles-ci sont considérées plus compliquées) et enfin le nombre `total` d'instances.

Une autre information intéressante à extraire du VBS est la contribution de chaque solveur qui le compose. Le Tableau 2 fournit la contribution de l'ensemble des solveurs en comparaison au VBS. Les quatre premières colonnes correspondent au VBS contraint par une *delta* minimale en secondes. Par exemple, 0*s* correspond au VBS original, tandis que 100*s* correspond aux instances pour lesquelles le meilleur solveur possède une avance de 100 secondes sur le second meilleur solveur. Ainsi, ces quatre colonnes exhibent la distribution des instances communes entre le VBS et les solveurs de la compétition en fonction de ce delta de temps. La dernière colonne `contribution` correspond au nombre d'instances que seul le solveur courant a réussi à résoudre. Ainsi, nous pouvons observer que *PicatSAT* est le plus grand contributeur concernant le monopole de résolutions : il a le plus grand nombre d'instances que seul lui a résolu. Nous pouvons aussi remarquer que *PicatSAT* et *Fun-sCOP+CryptoMiniSat* ont respectivement résolu 25 et 18 instances 100 secondes plus rapidement que le second meilleur solveur (pour chaque instance).

Une autre façon d'observer en détail le comportement de *PicatSAT* et *Fun-sCOP+CryptoMiniSat* est

6. « Le cactus ne s'associe pas avec le CDF. »

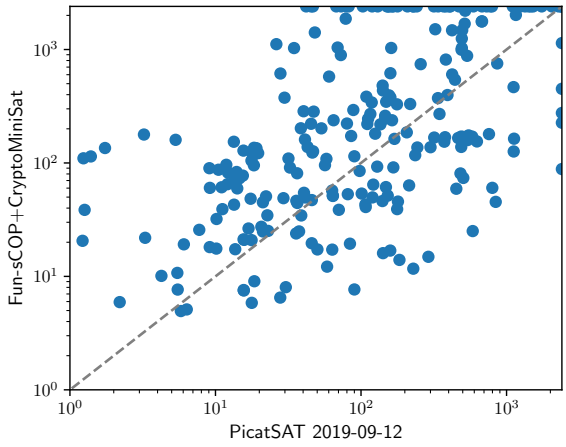


FIGURE 3 – Scatter plot des deux meilleurs solveurs de la compétition XCSP'19.

d'utiliser un scatter plot :

```
analysis.scatter_plot(
  scatter_col='cpu_time',
  [...]
)
```

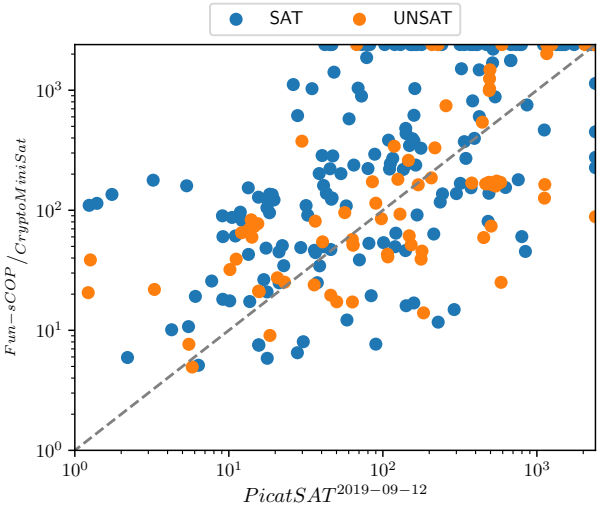


FIGURE 4 – Scatter plot des deux meilleurs solveurs de la compétition XCSP'19.

Le scatter plot, tel qu'illustré dans la Figure 3, montre que beaucoup d'instances résolues rapidement par *PicatSAT* sont en *timeout* pour *Fun-sCOP+CryptoMiniSat*. Si nous omettons les instances en *timeout*, il y a peu de différences entre les deux solveurs. L'utilisation du paramètre `color_col="Checked answer"`, dans la Figure 4, met en évidence les instances *SAT*/*UNSAT* et permet

	count	sum	PAR1	PAR2	PAR10	common count	common sum	uncommon count	total
VBS	270	90,388	90,388	162,388	738,388	65	405	205	300
PicatSAT 2019-09-12	246	192,377	192,377	321,977	1,358,777	65	11,093	181	300
Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	209	274,323	274,323	492,723	2,239,923	65	16,472	144	300
Fun-sCOP order+GlueMiniSat (2019-06-15)	190	320,070	320,070	584,070	2,696,070	65	14,632	125	300
AbsCon 2019-07-23	168	341,387	341,387	658,187	3,192,587	65	2,805	103	300
choco-solver 2019-06-14	168	369,846	369,846	686,646	3,221,046	65	7,875	103	300
Concrete 3.10	165	369,615	369,615	693,615	3,285,615	65	5,182	100	300
choco-solver 2019-09-16	165	372,266	372,266	696,266	3,288,266	65	7,790	100	300
choco-solver 2019-09-20	165	372,316	372,316	696,316	3,288,316	65	7,754	100	300
Concrete 3.12.3	156	386,276	386,276	731,876	3,496,676	65	7,198	91	300
choco-solver 2019-09-24	149	390,634	390,634	753,034	3,652,234	65	2,570	84	300
BTD 19.07.01	135	421,087	421,087	817,087	3,985,087	65	6,718	70	300
cosoco 2	127	448,425	448,425	863,625	4,185,225	65	6,810	62	300

TABLE 1 – Tableau des instances résolues et des statistiques de temps d'exécution des principaux solveurs de la compétition XCSP'19.

	vbew simple	vbew 1s	vbew 10s	vbew 100s	contribution
PicatSAT 2019-09-12	70	65	49	25	3
BTD 19.07.01	49	29	4	1	0
Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	41	37	30	18	1
cosoco 2	41	25	14	1	0
AbsCon 2019-07-23	17	16	11	2	0
choco-solver 2019-09-24	16	13	7	2	0
Fun-sCOP order+GlueMiniSat (2019-06-15)	15	10	8	2	0
Concrete 3.10	6	5	2	1	0
choco-solver 2019-06-14	6	5	4	1	0
Concrete 3.12.3	4	2	1	0	0
choco-solver 2019-09-16	4	2	0	0	0
choco-solver 2019-09-20	1	1	0	0	0

TABLE 2 – Tableau de contribution des principaux solveurs de la compétition XCSP'19.

de montrer, par exemple, que les *timeout* de *Fun-sCOP+CryptoMiniSat* sont le plus souvent sur des instances *SAT*. Cette même figure montre quelques manipulations possibles dans la transformation des noms de solveurs avec interprétation LaTeX. Enfin, *Metrics* donne la possibilité de survoler les points avec la souris et d'afficher les métadonnées des différentes instances (pour la version dynamique du scatter plot avec l'option `dynamic=True`). Cet outil peut être très pratique pour obtenir plus d'informations sur les instances en *timeout* par exemple.

Enfin, nous présentons un nouveau type de graphique qui complète les informations fournies par le cactus plot : le box plot. Ce graphique affiche des informations sur la moyenne et les quartiles. Comme précédemment, nous avons juste besoin d'appeler la méthode `box_plot()` de l'analyse que nous souhaitons observer :

```
analysis.box_plot(
  box_col='cpu_time',
  [...]
)
```

Les box plots résultants sont présentés dans la Figure 5. Chaque box plot est composé, de gauche à droite, de la valeur minimale, du premier et du troisième quartile (la boîte) et du maximum. Les valeurs aberrantes sont représentées par des cercles.

Nous pouvons observer que le premier solveur à résoudre des instances est *PicatSAT*. Dans la box, entre le premier et troisième quartile nous pouvons observer la médiane. Nous observons aussi que *PicatSAT* est le seul solveur à ne pas avoir son troisième quartile (représentant donc 75% des instances) confondu avec la moustache droite représentant la valeur maximale calculée.

Enfin, à travers les statistiques et les chiffres fournis par *Metrics*, nous observons que posséder une vision globale de la campagne pour la compétition XCSP'19 est important. Grâce aux opérations de filtrage, de génération de vues et de statistiques, nous avons rapidement identifié les meilleurs solveurs et avons pu les comparer pour obtenir des conclusions plus précises. Il reste difficile de trouver une unique méthode pour départager les solveurs à travers leurs performances : c'est pourquoi une diversité d'outils ont été conçus pour clarifier la situation.

4 Conclusion

Dans cet article, nous avons présenté *Metrics*, une bibliothèque fournissant une chaîne d'outils simple d'utilisation pour collecter et analyser des résultats expérimentaux en calculant des statistiques et en traçant différents types de diagrammes. Les utilisateurs sont

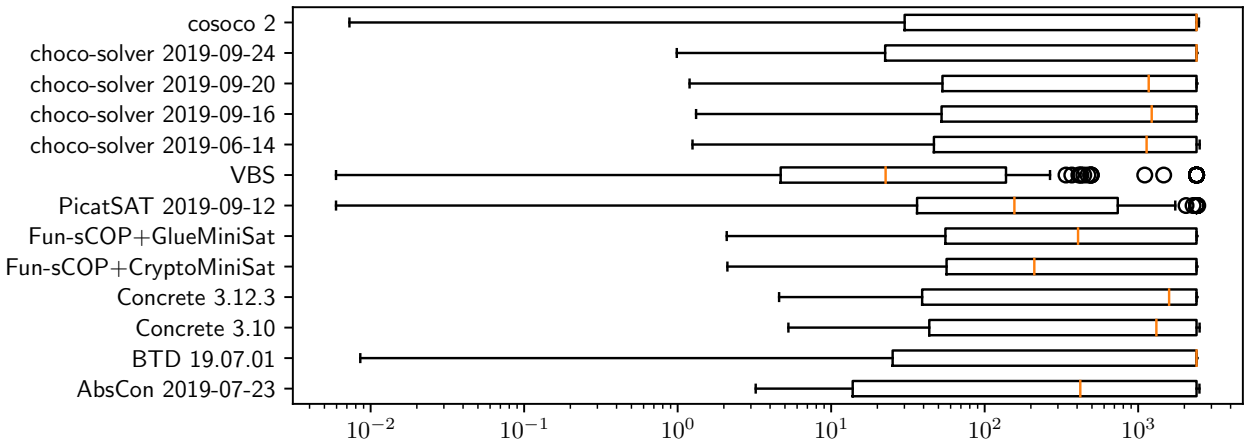


FIGURE 5 – Box plots des temps d'exécution des meilleurs solveurs de la compétition XCSP'19.

laissés libres d'organiser et de personnaliser l'analyse produite selon leurs besoins grâce à l'utilisation de *notebooks Jupyter*. Ceci permet de partager les résultats des expérimentations réalisées tout en rendant possible la reproductibilité de l'analyse.

A ce jour, *Metrics* est un projet *jeune*, qui propose un œil neuf sur la manière d'analyser ses expérimentations. Nous envisageons d'ajouter de nouvelles fonctionnalités à cette bibliothèque, de sorte à permettre des analyses plus approfondies des résultats, par exemple en permettant l'utilisation d'autres types de figures (par exemple, dans le cas particulier des problèmes d'optimisation). Bien que les utilisateurs peuvent utiliser le *data-frame* de la campagne et réaliser leurs analyses avec les fonctions déjà fournies par *pandas* et *matplotlib*, nous pensons que *Metrics* doit être en mesure de fournir ces fonctionnalités par défaut, afin de simplifier les actions à réaliser par l'utilisateur.

De plus, pour devenir une chaîne *complète* d'outils, *Metrics* a l'ambition d'étendre ses capacités en proposant une interface en lignes de commande ainsi qu'une interface web au travers desquelles il serait possible d'automatiser le processus d'exécution des solveurs et celui de collecte et d'analyse des données. En particulier, en configurant leurs installations de *Metrics* suivant leurs besoins, les utilisateurs pourront soumettre leur(s) logiciel(s) directement dans l'application, de sorte que toutes les étapes entre l'exécution du solveur (éventuellement, sur des machines distantes, sur un cluster ou dans le *cloud*) à la production d'un rapport complet, en minimisant les efforts à fournir par l'utilisateur. Notre but ultime est de faire de *Metrics* un outil incontournable : pas de *Metrics*, pas d'expé', pas d'expé', pas de papier, pas de papier… pas de papier !

Références

[1] 2019 XCSP3 Competition. http://www.cril.univ-artois.fr/XCSP19/, 2019 (accessed April 13, 2021).

[2] Gilles AUDEMARD, Frédéric BOUSSEMART, Christophe LECOUTRE, Cédric PIETTE et Olivier ROUSSEL : Xcsp^3 and its ecosystem. *Constraints An Int. J.*, 25(1-2):47–69, 2020.

[3] Juliana FREIRE, Norbert FUHR et Andreas RAUBER : Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.

[4] Gael GLORIAN, Jean-Marie LAGNIEZ et Christophe LECOUTRE : NACRE - A nogood and clause reasoning engine. *In* Elvira ALBERT et Laura KOVÁCS, éditeurs : *LPAR 2020 : 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 de *EPiC Series in Computing*, pages 249–259. EasyChair, 2020.

[5] Yang-Min KIM, Jean-Baptiste POLINE et Guillaume DUMAS : Experimenting with reproducibility : a case study of robustness in bioinformatics. *GigaScience*, 7(7):giy077, juillet 2018.

[6] Dirk PILAT et Yukiko FUKASAKU : Oecd principles and guidelines for access to research data from public funding. *Data Science Journal*, 6:4–11, 06 2007.

[7] Olivier ROUSSEL : Controlling a Solver Execution : the runsolver Tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.