SATFC MANUAL

[Auction]omics

ALEXANDRE FRÉCHETTE*, KEVIN LEYTON-BROWN*

January 8, 2015

CONTENTS

* *Department of Computer Science, University of British Columbia, British Columbia, Canada.*

1 INTRODUCTION

ABSTRACT SATFC (*SAT-based Feasibility Checker*) solves radio-spectrum repacking feasibility problems arising in the FCC's upcoming reverse auction. It combines a formulation of feasibility checking based on propositional satisfiability with a heuristic pre-solver and a SAT solver tuned for the types of instances observed in auction simulations.

AUTHORS & COLLABORATORS SATFC is the product of the ideas and hard work of Auctionomics, notably Alexandre Fréchette, Neil Newman, Guillaume Saulnier-Comte, Nick Arnosti, and Kevin Leyton-Brown.

CONTACT Questions, bug reports and feature suggestions should be directed to Alexandre Fréchette – afrechet@cs.ubc.ca.

1.1 Licenses

SATFC is released under the GNU General Public License (GPL) - `http://www.gnu.org/copyleft/gpl.html`.

1.2 System Requirements

SATFC is primarily intended to run on Unix-like platforms. It requires Java 8 to run and Gradle for building. One also needs our modified version of the SAT solver CLASP v2.2.3 compiled for JNA library usage, which in turn necessitates GCC v4.8.1 or higher as well as the standard Unix C libraries (see Section 3 to learn how to compile CLASP).

1.3 Version

This manual is for SATFC v1.2.

2 DESCRIPTION

2.1 Problem

At the core of the reverse auction part of the radio-spectrum incentive auction lies the problem of (re)assigning television stations to a smaller range of broadcast channels than what they currently occupy subject to various interference constraints (often referred to as the STATION PACKING PROBLEM or the FEASIBILITY CHECKING PROBLEM). These constraints are pairwise between stations, hence the whole problem can be cast to an (extended) GRAPH COLORING PROBLEM, where stations are nodes of the graph, edges correspond to interference constraints and colors to available broadcast channels. Unfortunately, the latter problem is known to be NP-complete, *i.e.* computationally challenging. Furthermore, in the latest reverse auction designs, feasibility checking must be executed very frequently, sometimes upwards of a thousand times per auction round. This problem is thus the fundamental computational bottleneck of the auction. Fortunately, the distribution of feasibility checking problems encountered in a typical auction is very specific, and that is what SATFC leverages.

2.2 SATFC

To take advantage of the specific distribution of encountered feasibility checking problems, SATFC first translates feasibility checking into its propositional satisfiability (SAT) version. This allows us to leverage the body of research on high performance SAT solvers developed over the last years. Through extensive

empirical evaluations, we have identified the SAT solver CLASP as the best solver when tuned on our specific instance set using SMAC, a sequential, model-based algorithm configurator. In addition to using a highly configured version of clasp, SATFC solves easy instances using a heuristic pre-solving algorithm based on previous partial satisfiable assignments. Finally, some engineering was required to make the whole pipeline as efficient as possible.

2.3 Efficient Usage

SATFC was developed to work particularly well on the auction designs that are being studied by the FCC. For example, in these designs, one expects to encounter many problems to be solved sequentially, a good fraction of which are simple. SATFC thus extensively leverages any partial feasible assignments to get rid of easy instances quickly, and has a main solver tuned for a specific (empirically defined) distribution of harder instances.

3 USAGE

3.1 Setting Up & Building SATFC

The releases folder in SATFC 's repository should contain a compiled, ready-to-go compressed version of SATFC . Packaged with SATFC are its source code, necessary libraries as well as a copy of CLASP .

SATFC uses the Gradle build system and maven for dependency resolution. To compile from source and build your own command-line executable version of SATFC , execute the following:

```
> cd <SATFC repository>
> ./gradlew installApp
```

This builds and packages SATFC in `<SATFC repository>/build/install/`, and may be a lengthy process as it (possibly) installs Gradle, downloads all of SATFC 's dependencies (from external repositories) and builds the project. Note that it can sometimes fail while trying to download dependencies - restarting the process fixes this. Note that SATFC 's versioning is based on branch and sha1 hash of the latest commit of its git repository.

Build tasks other than `installApp` are also available. A list can be obtained by doing

```
> ./gradlew tasks
```

For instance, the `distTar` or `distZip` packages a SATFC release in `<SATFC repository>/build/distributions/`, and the `javadoc` task produces all the SATFC javadocs.

In any case, CLASP may need to be compiled for your machine. To compile the CLASP packaged with a SATFC release, do the following:

```
> cd <SATFC release directory>
> cd clasp/
> bash compile.sh
```

To compile the source's CLASP (so that your custom built SATFC has an already compiled CLASP), instead execute:

```
> cd <SATFC repository>
> cd src/dist/clasp
> bash compile.sh
```

3.2 Standalone

Warning

Every time SATFC is launched from the command line, it will have to load the Java Virtual Machine, necessary libraries as well as any constraint data corresponding to the specified problem. This is a significant overhead compared to the time required to solve easy instances. If it is necessary to solve large numbers of instances, many of which are easy, we suggest using SATFC as a Java library.

To use SATFC from the command line to solve feasibility checking problems, go in the SATFC directory and execute the following:

```
> bash bin/SATFC -DATA-FOLDERNAME <data folder> -DOMAINS <station domains>
```

where

- `data folder` points to a folder containing the *broadcasting interference constraints data* (`Domain.csv` and `Interference_Paired.csv` files) - see Section 3.2.1 for further information, and
- `station domains` is a string listing a set of stations to be packed, and for each station a set of eligible channels. This *domains string* consists of *single station domains strings* joined by ';', where each single domain string consists of a station numerical ID, a ':', and a list of integer channels joined by ','.

  For example, the problem of packing stations 1 and 2 into channels 14,15,16,17,18,19,20 and station 3 into channels 14,15,16 would be specified by the following string:

  `1:14,15,16,17,18,19,20;2:14,15,16,17,18,19,20;3:14,15,16`

One can also run

```
> bash satfc --help
```

to get a list of the SATFC options and parameters.

3.2.1 *Data*

The broadcast interference constraint data (as specified by the FCC) consists of two files, one specifying station domains, and one specifying pairwise interference between stations. As one of its required arguments, command-line SATFC expects a path to a folder containing both of these files, named `Domain.csv` and `Interference_Paired.csv`, respectively.

DOMAINS The domains file consists of a CSV with no header where each row encodes a station's domain (the list of channels on which a station can broadcast). The first entry of a row is always the `DOMAIN` keyword, the second entry is the station ID, and all subsequent entries are domain channels. Note that SATFC uses this file to define the set of available stations (so it should contain all the station IDs that will be used in defining problems).

INTERFERENCES The interferences file consists of a CSV with no header where each row is a representation of many pairwise interference constraints between a single subject station on a single subject channels with possibly many target stations. Specifically, the entries of a row are, in order, a key indicating the type of constraint, a subject and target channel on which the constraint applies, the subject station of the constraint, and then a list of target stations to which the constraint applies. Here it is in more compact format:

```
<key>,<subject channel>,<target channel>,<subject station>,<target station 1>,...
```

There are three possible constraint keys: `CO`, `ADJ+1` and `ADJ-1`. The former indicates a *co-channel constraint*, stating that the subject station and any target station cannot be on the same target/subject channel. The second describes an *adjacent plus one constraint* which implies that the subject station cannot be on its

subject channel c together with any target station on the target channel $c+1$. The third one is an *adjacent minus one constraint*, which is just an adjacent plus one constraint with subject station and channels interchanged with target station and channel, respectively.

Here are two examples of interference constraints:

```
CO,4,4,1,2,3
```

This constraint means that neither stations 1 and 2 nor stations 1 and 3 can be jointly assigned to channel 4

```
ADJ+1,8,9,1,2,3
```

This constraint implies that neither stations 1 and 2 nor stations 1 and 3 can be on channels 8 and 9 respectively.

```
ADJ-1,8,7,1,2,3
```

This constraint implies that neither stations 1 and 2 nor stations 1 and 3 can be on channels 8 and 7 respectively.

3.2.2 *Parameters & Options*

In addition to the required arguments discussed above, command-line SATFC also exposes the following optional parameters:

- `--help` – display SATFC options and parameters, with short descriptions and helpful information.
- `-PREVIOUS-ASSIGNMENT` – a partial, previously satisfiable channel assignment. This is passed in a string similar to the `-DOMAINS` string: the station and previous channel pairs are separated by ':', and the different pairs are joined by ','. For example, "`1:14,15,16;2:14,15`" means pack station 1 into channels 14,15 and 16 and station 2 into channels 14 and 15.
- `-CUTOFF` – a cutoff time, in seconds, for the SATFC execution.

  > **Warning**
  > SATFC was optimized for runtimes of about one minute. Thus, components might not interact in the most efficient way if cutoff times vastly different than a minute are enforced, especially much shorter ones. Moreover, cutoff times below a second may not always be respected.

- `-SEED` – the seed to use for any (non-CLASP) randomization done in SATFC .
- `-CLASP-LIBRARY` – a path to the compiled "`.so`" CLASP library to use.
- `--log-level` – SATFC 's logging level. Can be one of `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE` (listed in increasing order of verbosity).

3.3 As a Library

The most efficient way of using SATFC is as a Java library. This source code is packaged with SATFC 's release. The code is well documented; the simplest entry point is the `SATFCFacade` object, and its corresponding builder `SATFCFacadeBuilder`. The reader may find it helpful to consult Sections 3.2 and 3.2.1 to understand the components at play.

Even though SATFC is built using gradle, native components such as CLASP are not yet packaged in SATFC 's maven repository. Hence, to build a project that uses SATFC , one must strictly depend on it at a project level for it to find a copy of CLASP , for example.

4 ACKNOWLEDGEMENTS

Steve Ramage deserves a special mention as he is indirectly responsible for much of SATFC 's quality, via his package AEATK, one of the main libraries in SATFC that was used throughout its development for various prototypes, as well as different other miscellaneous features. He also was a constant source of technical help and knowledge when it came to the software design of SATFC .

We would also like to acknowledge various members of the Auctionomics team. Ilya Segal provided the main idea behind the pre-solver used in SATFC . Ulrich Gall and his team members offered much useful feedback after working with SATFC during its early stages.