Solving the Station Repacking Problem

Alexandre Fréchette and **Neil Newman** and **Kevin Leyton-Brown**
Department of Computer Science
University of British Columbia, Canada
{afrechet, newmanne, kevinlb}@cs.ubc.ca

Abstract

We investigate the problem of repacking stations in the FCC's upcoming, multi-billion-dollar "incentive auction". Early efforts to solve this problem considered mixed-integer programming formulations, which we show are unable to reliably solve realistic, national-scale problem instances. We describe the result of a multi-year investigation of alternatives: a solver, SATFC, that has been adopted by the FCC for use in the incentive auction. SATFC is based on a SAT encoding paired with a wide range of techniques: constraint graph decomposition; novel caching mechanisms that allow for reuse of partial solutions from related, solved problems; algorithm configuration; algorithm portfolios; and the marriage of local-search and complete solver strategies. We show that our approach solves virtually all of a set of problems derived from auction simulations within the short time budget required in practice.

1 Introduction

The US government will soon hold an innovative "incentive auction" for radio spectrum, in which television broadcasters are paid to relinquish broadcast rights via a "reverse auction", remaining broadcasters are repacked into a narrower band of spectrum, and cleared spectrum is sold to telecommunications companies. The stakes are enormous: the auction is forecast to net the government tens of billions of dollars, as well as creating massive economic value by reallocating spectrum to more socially beneficial uses (Congressional Budget Office 2015). As a result of both its economic importance and its conceptual novelty, the auction has been the subject of considerable recent study by the research community, mostly focusing on elements of the auction design (Bazelon, Jackson, and McHenry 2011; Kwerel, LaFontaine, and Schwartz 2012; Milgrom et al. 2012; Calamari et al. 2012; Marcus 2013; Milgrom and Segal 2014; Dütting, Gkatzelis, and Roughgarden 2014; Vohra 2014; Nguyen and Sandholm 2014; Kazumori 2014). After considerable study and discussion, the FCC has selected an auction design based on a descending clock (FCC 2014c; 2014a). Such an auction offers each participating station a price for relinquishing its broadcast rights, with this price offer falling for a given station as long as it remains repackable. A consequence of this design is that the auction must (sequentially!) solve hundreds of thousands of such repacking problems. This is challenging, because the repacking problem is NP-complete. It also makes the performance of the repacking algorithm extremely important, as every failure to solve a single, feasible repacking problem corresponds to a lost opportunity to lower a price offer. Given the scale of the auction, individual unsolved problems can cost the government millions of dollars each.

This paper is the first to show how the station repacking problem can be solved exactly and reliably at the national scale. It describes the results of an extensive, multi-year investigation into the problem, which culminated in a solver that we call SATFC. This solver combines a wide variety of techniques: SAT encoding; algorithm configuration; algorithm portfolios; and a bevy of problem-specific speedups, including a powerful and novel caching scheme that generalizes from different but related problem instances. Overall, SATFC solves virtually all problems in a previously-unseen test set—99.6%—within one minute. It has been adopted by the FCC for use in the incentive auction (FCC 2014b). SATFC is open-source; pointers both to the solver and to data used in this paper are available at http://www.cs.ubc.ca/labs/beta/Projects/SATFC.

In what follows, we begin by defining the station repacking problem and explaining some of its salient properties (Section 2). We then discuss the encodings we considered and the tools we leverage (Section 3) before detailing the problem-specific speedups that make our approach effective (Section 4). We report experimental results throughout, establishing a baseline in Section 2 and then showing the extent to which each of our extensions strengthens our solver.

2 The Station Repacking Problem

Each US television station $s \in S$ is currently assigned a channel $c_s \in C \subseteq \mathbb{N}$ that ensures that it will not excessively interfere with other, nearby stations. The FCC reasons about what interference would be harmful via a complex, grid-based physical simulation ("OET-69" (FCC 2013)), but has also processed the results of this simulation to obtain a CSP-style formulation listing forbidden pairs of stations and channels, which it has publicly released (FCC 2014e). Let $I \subseteq (S \times C)^2$ denote a set of *forbidden station–channel pairs* $\{(s, c), (s', c')\}$, each representing the proposition that stations s and s' may not concurrently be assigned to channels c and c', respectively. The effect of the auction will be to remove some broadcasters from the airwaves completely, and to reassign channels to the remaining stations from a reduced set. This reduced set will be defined by a *clearing target*: some channel $\overline{c} \in C$ such that all stations are only eligible to be assigned channels from $\overline{C} = \{c \in C : c < \overline{c}\}$. The sets of channels *a priori* available to each station are given by a *domain* function $D : S \to 2^{\overline{C}}$ that maps from stations to these reduced sets. The *station repacking problem* is then the task of finding a repacking $\gamma : S \to \overline{C}$ that assigns each station a channel from its domain that satisfies the interference constraints: i.e., for which $\gamma(s) \in D(s)$ for all $s \in S$, and $\gamma(s) = c \Rightarrow \gamma(s') \neq c'$ for all $\{(s, c), (s', c')\} \in I$. It is easy to see that the station repacking problem is NP-complete; e.g., it generalizes graph coloring (see below). It also falls under the umbrella of *frequency assignment problems* (see Aardal et al. (2007) for a survey and a discussion of applications to mobile telephony, radio and TV broadcasting, satellite communication, wireless LANs, and military operations).

Luckily, there is reason to hope that this problem could nevertheless be solved effectively in practice. First, we only need to be concerned with problems involving subsets of a fixed set of stations and a fixed set of interference constraints: those describing the television stations currently broadcasting in the United States. Channels can be partitioned into three equivalence classes: LVHF (channels 1–6), HVHF (channels 7–13), and UHF (channels 14–$\overline{c}$, excepting 37), where $\overline{c} \leq 51$ is the largest available UHF channel set by the auction's clearing target, and 37 is never available. No interference constraints span the three equivalence classes of channels, giving us a straightforward way of decomposing the problem. A problem instance thus corresponds to a choice of stations $S \subseteq \mathcal{S}$ and channels $C \subseteq \mathcal{C}$ to pack into, with domains D and interference constraints I implicitly being restricted to S and C; we call the resulting restrictions $\mathcal{D}$ and $\mathcal{I}$.

Let us define the *interference graph* as an undirected graph in which there is one vertex per station and an edge exists between two vertices s and s' if the corresponding stations participate together in any interference constraint: i.e., if there exist $c, c' \in C$ such that $\{(s, c), (s', c')\} \in I$. Figure 1 shows the US interference graph. We know that every repacking problem we will encounter will be derived from the restriction of this interference graph to some subset of S. This suggests the possibility of doing offline work to capitalize on useful structure present in the interference graph. However, this graph involves a total of $|\mathcal{S}| = 2173$ stations, and the number of possible subsets is exponential in this number. Thus, it is not possible to exhaustively range over all possible subsets. Nevertheless, meaningful structure exists, whether leveraged explicitly or implicitly (see e.g., a computational analysis of unavoidable constraints by Kearns and Dworkin (2014)).

Interference constraints are more structured than in the general formulation: they come in only two kinds. *Co-channel constraints* specify that two stations may not be assigned to the same channel; *adjacent-channel constraints* specify that two stations may not be assigned to two adjacent channels. Hence, any forbidden station–channel pairs are of the form $\{(s, c), (s', c)\}$ or $\{(s, c), (s', c + 1)\}$ for some stations $s, s' \in \mathcal{S}$ and channel $c \in \mathcal{C}$. Note that if we were dealing exclusively with co-channel constraints, we would face a graph coloring problem.

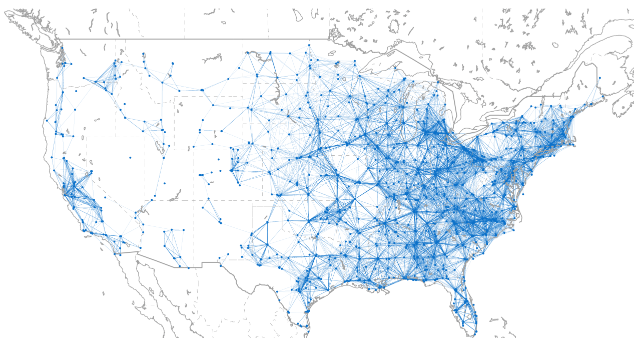
Figure 1: Interference graph derived from the FCC's May 2014 constraint data (FCC 2014e).

Interestingly, our cost function is asymmetric: it is terrible to incorrectly conclude that a repacking problem is feasible (this could make the auction outcome infeasible) whereas the consequences are fairly mild for wrongly claiming that a given repacking is infeasible (this prevents a price offer from being lowered, costing the government money, but does not pose a fundamental problem for the auction itself). Thus, whenever a problem cannot be solved within the given amount of time, we can safely treat it as though it was proven infeasible—albeit at some financial cost.

The last key property that gives us reason to hope that this large, NP-complete problem can be tamed in practice is that we are not interested in worst-case performance, but rather in good performance on the sort of instances generated by actual reverse auctions. The question of which stations will need to be repacked in which order depends on the stations' valuations, which depend in turn (among many other factors) on the size and character of the population reached by their broadcasts. The distribution over repacking orders is hence far from uniform. Second, descending clock auctions repeatedly generate station repacking problems by adding a single station s^+ to a set S^- of provably repackable stations. This means that every station repacking problem $(S^- \cup \{s^+\}, C)$ comes along with a partial assignment $\gamma^- : S^- \to C$ which we know is feasible on restricted station set S^-.

In order to validate the auction design, the FCC has run extensive simulations of the auction, based on a wide variety of assumptions about station participation and bidding behavior. We obtained anonymized versions of some repacking problems from five such simulations,[1] and randomly partitioned them into a training set of 100,572 examples, a validation

[1]This small set of simulations explores a very narrow set of answers to the questions of which stations would participate and how bidders would interact with the auction mechanism; it does not represent a statement either by us or by the FCC about how these questions are likely to be resolved in the real auction. While this data represents the best proxy currently available for the sorts of computational problems that would arise in practice, it is of course impossible to guarantee that variations in the assumptions would not yield computationally different problems.

set of 1,000 examples, and a test set of 10,000 examples. All of these problems were nontrivial in the sense that the auction's previous solution could not be directly augmented with an assignment to the newly introduced station; such trivial problems (80–90% of the total encountered in a typical auction simulation) are solved directly in the FCC's software without calling an external solver. Preliminary experiments showed that repacking problems in the VHF bands are very easy, because these bands contain at most 7 channels. We thus constrained ourselves to the much harder UHF band, fixing the interference model and setting the clearing target such that $|C| = 16$. The encoded test problems contained between 453 and 16,299 variables (averaging 8,654) and between 3,197 and 342,450 clauses (averaging 146,871) of which between 210 and 228,042 (averaging 86,849) were interference clauses. Station domains ranged from 1–16 channels (averaging 14). We used test instances solely to perform the benchmarking reported here, having performed preliminary experimentation and optimization using the training and validation sets. We chose a cutoff time of 60 seconds, reflecting the constraints involved in solving up to hundreds of thousands of problems sequentially in a real auction.

3 Encoding and Tools

3.1 Initial Efforts

The FCC's initial investigations included modeling the station repacking problem as a mixed-integer program (MIP) and using off-the-shelf solvers paired with problem-specific speedups (FCC 2014d). Unfortunately, the problem-specific elements of this solution were not publicly released, so we do not discuss them further in this paper. Instead, to assess the feasibility of a MIP approach, we ran what are arguably the two best-performing MIP solvers—CPLEX and Gurobi—on the test set described above; the results are summarized as the dashed lines in Figure 2. To encode the station repacking problem as a MIP, we create a variable $x_{s,c} \in \{0, 1\}$ for every station–channel pair, representing the proposition that station s is assigned to channel c. We then add the constraints $\sum_{c \in D(s)} x_{s,c} = 1 \forall s \in S$, and $x_{s,c} + x_{s',c'} \leq 1 \forall \{(s, c), (s', c')\} \in I$. These constraints ensure that each station is assigned to exactly one channel, and that interference constraints are not violated.

On our experimental data, both solvers solved under half of the instances within our cutoff time of one minute. Such performance would likely be insufficient for deployment in practice, since it means that most stations would be paid unnecessarily high amounts due to computational constraints (recall that each station gives rise to many feasibility checking problems over the course of a single auction).

3.2 SAT Encoding

We propose instead that the station repacking problem should be encoded as a propositional satisfiability (SAT) problem. This formalism is well suited to station repacking, which is a pure feasibility problem with only combinatorial constraints.[2]

[2]Of course, it may nevertheless be possible to achieve good performance with MIP or other techniques; we did not investigate such alternatives in depth.

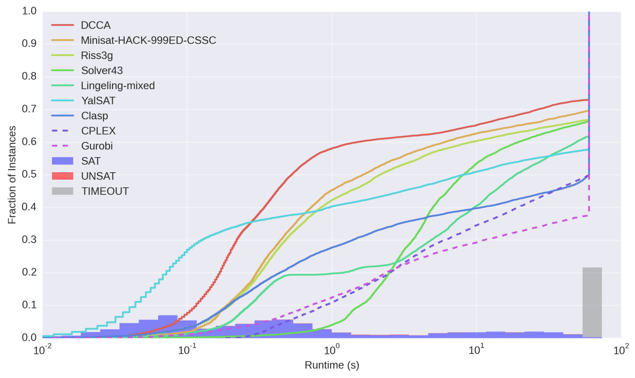

Figure 2: ECDF of runtimes for default MIP and SAT solvers. The bars show fraction of SAT and UNSAT instances binned by their (fastest) runtime. Although present, unsatisfiable instances form an insignificant portion of instances solved.

The SAT reduction is straightforward (and similar to the MIP reduction just described): given a station repacking problem (S, C) with domains D and interference constraints I, we create a boolean variable $x_{s,c} \in \{\top, \bot\}$ for every station–channel pair $(s, c) \in S \times C$, representing the proposition that station s is assigned to channel c. We then create three kinds of clauses: (1) $\bigvee_{d \in D(s)} x_{s,d} \; \forall s \in S$ (each station is assigned at least one channel); (2) $\neg x_{s,c} \vee \neg x_{s,c'} \; \forall s \in S, \forall c, c' \neq c \in D(s)$ (each station is assigned at most one channel); (3) $\neg x_{s,c} \vee \neg x_{s',c'} \; \forall \{(s, c), (s', c')\} \in I$ (interference constraints are respected).

Besides parsimony, a SAT encoding has the advantage of making it possible to leverage the research community's vast investment into developing high-performance SAT solvers (see e.g., Järvisalo et al. (2012)). We experimented with 18 different SAT solvers, obtained mainly from SAT solver competition entries collected in AClib (Hutter et al. 2014b). The performance of the seven best solvers, measured according to the number of instances solved by the cutoff time, is summarized as the solid lines in Figure 2. We observed a range of performance, but found that no solver did well enough to recommend use in practice: the best solver could not even solve three quarters of the instances. The best was DCCA (Luo et al. 2014), which as a local search algorithm can only prove satisfiability. This turns out not to be a major impediment because, due to the way a descending clock auction works, the instances we encounter are predominantly satisfiable. More specifically, out of the 9963 test instances that we were able to solve by any means, including the solvers introduced later in this paper, 9871 (99.07%) were satisfiable and only 92 (0.93%) unsatisfiable. We illustrate the distribution of instances labeled by feasibility at the bottom of Figure 2.

3.3 Meta-Algorithmic Techniques

In recent years, there has been increasing development of artificial intelligence techniques that reason about how existing heuristic algorithms can be modified or combined together to yield improved performance on specific problem domains of interest. These techniques are called *meta-algorithmic* because they consist of algorithms that take other algorithms as part of their input. For example, *algorithm configuration* consists of setting design decisions exposed as parameters to optimize an algorithm's average performance across an instance distribution. This approach has proven powerful in the SAT domain, as many SAT solvers expose parameters that can drastically modify their behavior, from probability of random restarts to choice of search heuristics or data structures (Hutter et al. 2014a). We performed configuration using the Sequential Model-based Algorithm Configuration algorithm, or SMAC (Hutter, Hoos, and Leyton-Brown 2011).

Unfortunately, even after performing algorithm configuration, it is rare to find a single algorithm that outperforms all others on instances of an NP-hard problem such as SAT. This inherent variability across solvers can be exploited by *algorithm portfolios* (Gomes and Selman 2001; Nudelman et al. 2003). Most straightforwardly, one selects a small set of algorithms with complementary performance on problems of interest and, when asked to solve a new instance, executes them in parallel.

Finally, algorithm configuration and portfolios can be combined. Hydra (Xu, Hoos, and Leyton-Brown 2010) is a technique for identifying sets of complementary solvers from highly parameterized design spaces via algorithm configuration, by greedily adding configurations that make the greatest possible marginal contribution to an existing portfolio. Specifically, we create a (parallel) portfolio by greedily selecting the algorithm that most improves its performance. In our experiments we measured improvement by percentage of instances solved within a one-minute cutoff. Therefore, we started off by picking the best solver, then used algorithm configuration to construct many new solvers that complement it well, then identified the next best solver given that solver, and so on.

Performing this procedure on our experimental data, we committed to the best default SAT solver, `DCCA` (which unfortunately exposes no parameters), and configured the remaining 17 solvers with the objective of maximizing marginal contribution. The solver `clasp` (Gebser et al. 2007) was the best in this regard, improving the number of instances solved in our validation set by 4% when executed in parallel with `DCCA`. We could then have performed further Hydra iterations; however, our first two enhancements from Section 4 end up altering our configuration scenario. Hence, we revisit the impact of Hydra at the end of Section 4.2.

4 Problem-Specific Enhancements

We now describe the novel methods we developed to bring our SAT-based station repacking solver to the point where it could deliver high performance in practice.[3] In what follows, we assume that the sets $\mathcal{S}$ and $\mathcal{C}$ consist of all stations and channels, domains $\mathcal{D}$ and interference constraints $\mathcal{I}$ are fixed, and that we are given a station repacking instance $(S = S^- \cup \{s^+\}, C)$ along with a feasible, partial assignment γ^-.

[3]While we only discuss positive results, we unsucessfully explored many other avenues, notably including incremental SAT solvers, various heuristics, and local consistency techniques.

4.1 Incremental Station Repacking

Local Augmenting. On a majority of problem instances, a simple transformation of γ^- is enough to yield a satisfiable repacking: we consider whether it is possible to assign s^+ to a channel and update the channel assignments of the stations in s^+'s neighborhood, holding the rest of γ^- fixed. Specifically, we find the set of stations $\Gamma(s^+) \subseteq S$ that neighbor s^+ in the interference graph, then solve the reduced repacking problem in which all non-neighbors $S \setminus \Gamma(s^+)$ are fixed to their assignments in γ^-. Observe that a feasible repacking for this reduced problem is also feasible on the full set; on the other hand, if we prove that the reduced problem is infeasible, we cannot conclude anything. The value of this approach is in its speed: the sparseness of the interference graph often yields very small neighborhoods, hence small reduced problems.

To evaluate this technique experimentally, we performed local augmentation on all our instances and used the training set to configure each SAT solver for best performance on this new instance distribution. `DCCA` was once again the best-performing solver. The performance of this altered version of `DCCA`, dubbed *`DCCA-preSAT`*, is shown as one of the dotted lines in Figure 5: it solved 78.5% of the instances and then stagnated within 0.1 seconds.

Starting Assignment for Local Search Solvers. Local search solvers such as `DCCA` work by searching a space of complete assignments and seeking a feasible point, typically following gradients to minimize an objective function that counts violated constraints, and periodically randomizing. When working with such solvers we can leverage γ^- in a second way, by assigning the stations in γ^- to their channels in γ^- and randomly assigning a channel for s^+. If a solution does indeed exist near this starting point, such an initialization can help us to find it much more quickly (although there is no guarantee that the solver will not immediately randomize away to another part of the space). We observe that this approach does not generalize the "local augmenting" approach, as we do not constrain the local search algorithm to consider only s^+'s (extended) neighborhood.

The performance of `DCCA` starting from the previously feasible assignment, which we dub *`DCCA+`*, is shown in Figure 5. It solved 85.4% of the instances, clearly dominating the original, randomly initialized `DCCA`.

4.2 Problem Simplification

We now describe two ways of simplifying problem instances.

Graph Decomposition. First, the subgraph of the interference graph induced by the set of stations considered in a particular problem instance is usually disconnected. It can therefore help to identify disconnected components and solve each separately, giving smaller instances to our SAT solvers. An additional benefit is that if we identify a single component as infeasible, we can immediately declare the entire problem infeasible without looking at all of its components. In practice, we decompose each problem into its connected components and solve each component sequentially, starting with the smallest component. While we could have instead solved each component in parallel, we found that in practice runtimes were almost always dominated by the cost of solving the largest component, so that it did not make much of a difference whether or not the components were solved simultaneously. This did not mean that decomposition was not worth doing—we also found that the largest component was often considerably smaller than the full problem (e.g., first, second and third quartiles over number of stations of 346, 458, and 559 after simplification as compared to 494, 629, and 764 before).

Underconstrained Station Removal. Second, in some cases we can delete stations completely from a repacking problem and thereby reduce its size. This occurs when there exist stations for which, regardless of how every other station is assigned, there always exists some channel into which they can be packed. Verifying this property exactly costs more time than it saves; instead, we check it via the sound but incomplete heuristic of comparing a station's available channels to its number of neighboring stations. This problem simplification complements graph decomposition: we perform it first in order to increase the number of components into which we will be able to decompose the graph. We observe that a few important stations of high degree are often underconstrained; these are the stations whose removal makes the biggest difference to graph decomposition.

4.3 Hydra Revisited

The problem-specific enhancements we have discussed so far impact the Hydra procedure: incremental solvers solve many instances extremely quickly, allowing the remaining solvers in the portfolio to concentrate their efforts elsewhere; our problem simplifications change instances enough to reduce correlation between solvers. We thus augmented the set of solvers available to Hydra to include `DCCA-presat`, `DCCA+`, and all base SAT solvers given simplified problem instances. Our first two rounds of Hydra, already described, identified our base `DCCA-preSAT` and `DCCA+` solvers. Our third round selected a configured version of `clasp`; we dub this new contributor *`clasp-h1`*. The fourth iteration found a second `clasp` configuration that operates on simplified instances; we dub that *`clasp-h2`*. The (test-set) performance of these two `clasp` configurations is shown in Figure 5. `clasp-h1` solves 2.8% of the (test-set) instances previously unsolved by the (`DCCA-preSAT`; `DCCA+`) portfolio, and `clasp-h2` solves an additional 0.9% that were unsolved by the 3-solver portfolio. The next Hydra step yielded only a 0.2% (validation-set) improvement, and we found it desirable to obtain a portfolio that could be run on a 4-core workstation, so we stopped with this set of 4 solvers.

4.4 Caching Instances

So far, we have concentrated on building the best station repacking solver possible, based on no contextual information except a previous assignment. However, our advance knowledge of the constraint graph means that we have considerably more context. Furthermore, it is feasible to invest an enormous amount of offline computational time before the actual auction, in order to ensure that the repacking problem can be solved quickly online. We investigated a wide range of strategies for leveraging such offline computation (including incremental SAT solving), but had the most success with a novel caching scheme we call *containment caching*.

Containment Caching. "Caching" means storing the result of every repacking problem solved on our training set, for reference at test time. Unfortunately, in experiments on our validation set, we observed that it was extremely rare to encounter previously-seen problems, even given our training set of over 100,000 problems. However, observe that if we know whether or not it is possible to repack a particular set of stations S, we can also answer many different but related questions. Specifically, if we know that S was packable then we know the same for every $S' \subseteq S$ (and indeed, we know the packing itself—the packing for S restricted to the stations in S'). Similarly, if we know that S was unpackable then we know the same for every $S' \supseteq S$. This observation dramatically magnifies the usefulness of each cached entry S, as S can be used to answer queries about an exponential number of subsets or supersets (depending on the feasibility of repacking S).

We call a cache meant to be used in this way a *containment cache*, because it is queried to determine whether one set contains another (i.e., whether the query contains the cache item or vice versa). To the best of our knowledge, containment caching is a novel idea. A likely reason why this scheme is not already common is that querying a containment cache is nontrivial; see below. We observe that containment caching is applicable to any family of feasibility testing problems generated as subsets of a master set of constraints, not just to spectrum repacking.

In more detail, containment caching works as follows. We maintain two caches, a *feasible cache* and an *infeasible cache*, and store each problem we solve (including both full instances and their components resulting from problem simplifications) in the appropriate cache. When asked whether it is possible to repack station set S, we proceed as follows. First, we check whether the feasible cache contains a superset of S, in which case the original problem is feasible. If we find no matches, we check to see whether a subset of S belongs to the infeasible cache, in which case the original problem is infeasible. If both queries fail, we simplify and decompose the given instance and query each component in the feasible cache. We do not check the infeasible cache again after problem simplification, because a subset of a component of the original instance is also a subset of the original instance.

Querying the Containment Cache. Containment caching is less straightforward than traditional caching, because we can not simply index entries with a hash function. Instead, an exponential number of keys could potentially match a given query. We were nevertheless able to construct an algorithm that solved this problem extremely quickly: within an average time of 30 ms on a cache of nearly 200,000 entries.[4]

[4]There is a literature on efficiently finding subsets and supersets of a query set (Hoffmann and Koehler 1999; Savnik 2013; Charikar, Indyk, and Panigrahy 2002; Patrascu 2011). However, our algorithm was so fast in our setting that we did not explore alternatives; indeed,

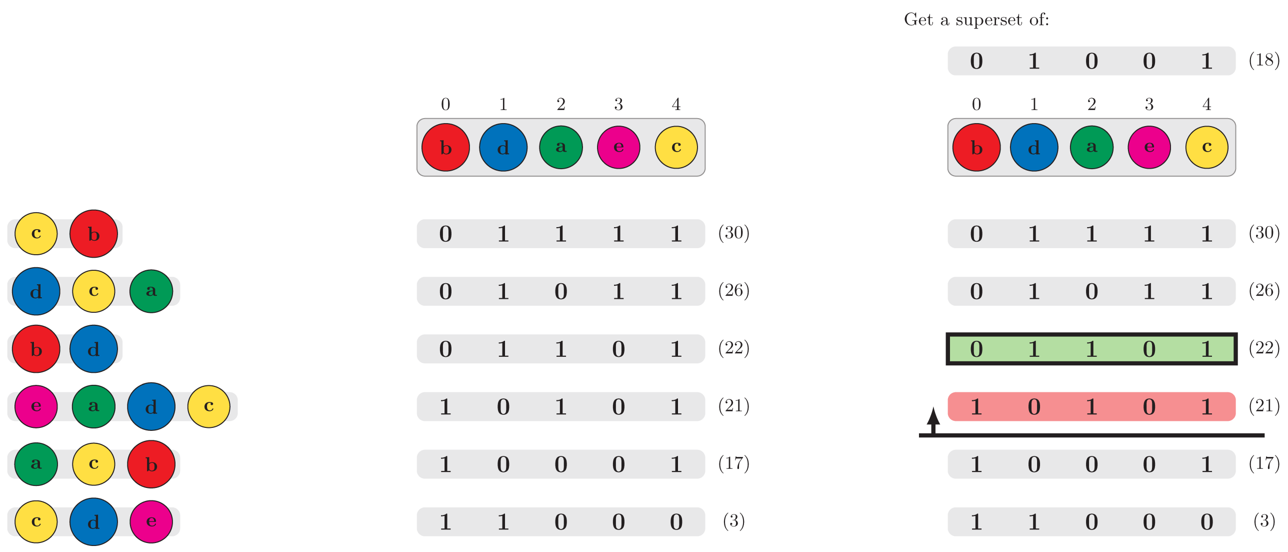

Figure 3: Containment caching example. Left: six elements of the power set $2^{\{a,b,c,d,e\}}$. Center: a secondary cache defined by a random ordering over the five elements, with each of the sets interpreted as a bit string and sorted in descending order. Right: the result of querying the containment cache for supersets of $\{c, d\}$. The query (18) does not exist in the cache directly; the next largest entry (21) is not a superset (i.e., 01001 does not bitwise logically imply 10101); the cache returns $\{a, c, d\}$ (22).

Specifically, our approach proceeds as follows. Offline, we build (1) a traditional cache $\mathfrak{C}$ indexed by a hash function and—in the case of feasible problems—storing solutions along with each problem; and (2) a secondary cache $\mathfrak{C}_o$ containing only a list of station sets that appear in $\mathfrak{C}$. This secondary cache is defined by an ordering o over the stations, which we choose uniformly at random. We represent each station set stored in $\mathfrak{C}_o$ as a bit string, with the bit in position k set to 1 if and only if the k-th station in ordering o belongs to the given station set. We say that one station set is larger or smaller than another by interpreting both station set bit strings as integers under the ordering o and then comparing the integers. Appealing to this ordering, we sort the entries of $\mathfrak{C}_o$ in descending order. We give an example in Figure 3: the left and center diagrams respectively illustrate a set of six subsets of the power set $2^{\{a,b,c,d,e\}}$ and a secondary cache constructed based on these sets along with a random ordering over their elements. As the figure suggests, secondary caches are very compact: a cache of 200,000 entries, each consisting of 2,000 stations/bits, occupies only 50 MB. We can thus afford to build multiple secondary caches $\mathfrak{C}_{o_1}, \ldots, \mathfrak{C}_{o_\ell}$, based on the same set of station sets but ℓ different random orderings.

We now explain how to query for a superset, as we do to test for feasible solutions; the algorithm for subsets is analogous. (A sample execution of this algorithm is illustrated in Figure 3 (right) and explained in the caption.) Given a query S, we perform binary search on each of the ℓ secondary caches to find the index corresponding to S itself (if it is actually stored in the cache) or of the smallest entry larger than S (if not); denote the index returned for cache $\mathfrak{C}_{o_k}$ as i_k. If we find S, we are done: we retrieve its corresponding solution from the main cache. Otherwise, the first i_1 entries in cache $\mathfrak{C}_{o_1}$ contain a mix of supersets of S (if any exist) and non-supersets that contain one or more stations not in S that appear early in the ordering o_1. Likewise, the first i_2 entries in $\mathfrak{C}_{o_2}$ contain the same supersets of S (because $\mathfrak{C}_{o_1}$ and $\mathfrak{C}_{o_2}$ contain exactly the same elements) and a different set of non-supersets based on the ordering o_2, and so on. We have to search through the first i_k entries of some cache $\mathfrak{C}_{o_k}$; but it does not matter which $\mathfrak{C}_{o_k}$ we search. We thus choose the shortest list: $k = \arg\min_j i_j$. This protects us against unlucky situations where the secondary cache's ordering yields a large i_k: this is very unlikely to happen under all ℓ random orderings. The superset search itself can be performed efficiently by testing whether the cached bit string is bitwise logically implied by the query bit string. If we find a superset, we query the main cache to retrieve its solution.

Evaluation. To build the cache we used to evaluate our approach, we ran our 4-solver parallel portfolio with a 24-hour cutoff time on all instances from both our training and validation sets,[5] along with all of their simplified versions,

our approach may be of independent interest.

[5]A word of warning: Our experiments may exaggerate cache performance because of the way we partitioned our training and test data: related instances from the same auction run can appear in different sets. We would have preferred to use a test set consisting of entirely distinct auction simulations; however, the data provided to us by the FCC did not distinguish the auction runs from which each instance was taken. We do note that SATFC achieved very strong performance even without the cache—solving 98% of the instances in under a minute—meaning that even if we do overestimate cache performance here, our overall qualitative findings would not change. Subsequent, informal experimentation on new data (which we are not yet free to publish) assures us that containment caching continues to "solve" a large fraction of instances even when entire auction runs are excluded from training, albeit usually fewer than the 98%

terminating runs when one solver completed. To speed up this process, we constructed the cache in a bootstrapped fashion: we made use of a partial cache even as we were working through the set of instances designed to populate the cache, thereby solving subsequent instances more quickly. In the end, we obtained a cache of 185,750 entries at a cost of roughly one CPU month. The largest feasible problem in our cache contained 1170 stations, and the smallest infeasible problem contained 2 stations. We built $\ell = 5$ secondary caches based on different random station orderings.

We interpret the containment cache as a standalone solver that behaves as follows: (1) checking whether a full instance has a superset in the feasible cache; (2) if not, checking whether a subset of the instance belongs to the infeasible cache; (3) simplifying the instance and then asking whether all of its components have supersets in the feasible cache. This solver's runtime is equal to the appropriate cache lookup time(s) plus the time to perform problem simplification, if initial cache lookup fails; we report its performance in Figure 5. It far outperformed all other algorithms, solving 98.2% of the instances on its own; however, this "solver" obviously works only because we were able to obtain a database of solved instances via our other algorithms. We also note that its performance would continue to improve if we obtained an even larger training set and performed more offline computation, as we intend to do in preparation for the incentive auction.

To investigate more deeply how the cache functioned, Figure 4 shows a scatter plot relating the frequency with which different keys were "hit" in the cache and the amount of time the best set of remaining solvers would have taken to solve the instances if the cache had not been used. This analysis shows that only a handful of keys were hit more than ten times, but that infrequently hit keys contributed significantly to the total time saved by the cache. Furthermore, many hits saved more than our one-minute cutoff time, justifying our investment in very long runs while populating the cache.

5 Conclusions: Putting It All Together

Station repacking is an economically important problem that initially seemed impractical to solve exactly. We have shown how to combine state-of-the-art SAT solvers, recent meta-algorithmic techniques, and further speedups based on domain-specific insights to yield a solver that meets the performance needs of the real incentive auction. Specifically, we identified a powerful parallel portfolio[6] of four solvers: a containment cache followed by `DCCA-preSAT`; `DCCA+`; `clasp-h1`; and `clasp-h2`. This portfolio, which we named *`SATFC 2.0`* (for SAT-based Feasibility Checker) achieved impressive performance (shown in Figure 5) as the SATFC 2.0 line), solving 99.0% of test instances in under 0.2 seconds, and 99.6% in under a minute. Moreover, the contribution of one of its main components, the containment cache, will continue to increase at negligible (online) CPU cost as we base

reported below.

[6]Our intention is for SATFC to be run in parallel. However, since our portfolio achieves excellent performance in under a second, sequential execution would therefore achieve the same performance in under four seconds.

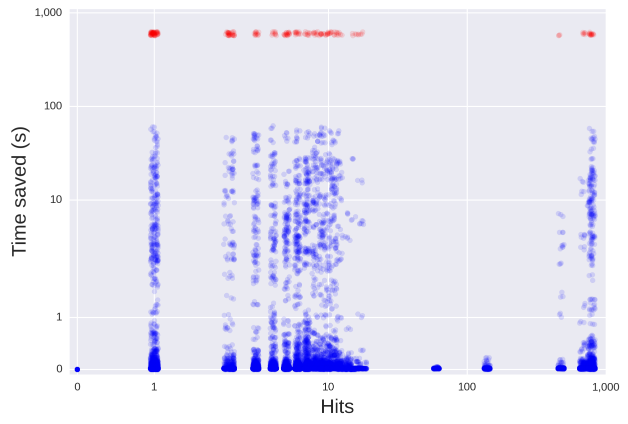

Figure 4: Time saved per cache hit. Each point represents a cache hit on a particular key: the x-axis represents the number of times the corresponding key was hit, while the y-axis represents the amount of time each individual cache hit saved. For visualization purposes we count the runtime of unsolved instances as 10 times the cutoff time and color such points in red.

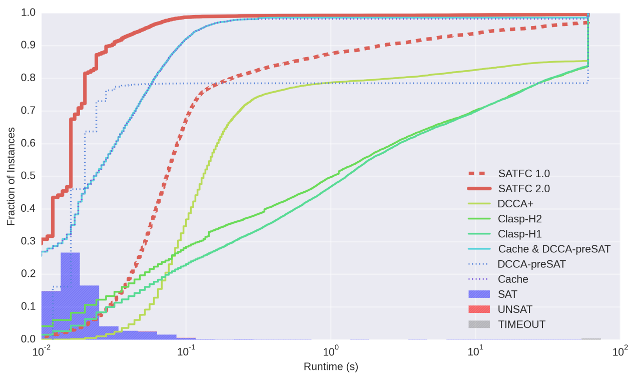

Figure 5: ECDF of runtimes of the SAT solvers we include in our final portfolio. The bars show fraction of SAT and UNSAT instances binned by their (fastest) runtime.

it on more data.

Finally, we note `SATFC 2.0`'s significant improvement over our previous solver `SATFC`, which was adopted and officially released by the FCC in November 2014 (FCC 2014b), albeit never previously discussed in an academic publication. Most of the new ideas presented in this paper go beyond `SATFC`, which is a single-processor sequential portfolio: it works by carrying out a SAT encoding, performing the "local augmenting" idea described in Section 4, and then executing a version of `clasp` that we identified via algorithm configuration. Although `SATFC` achieved quite good performance within long cutoff times, it is both dramatically less able to solve instances within very short timescales and able to solve considerably fewer instances overall.

Acknowledgments. We gratefully acknowledge support from Auctionomics and the FCC; helpful conversations with Paul Milgrom, Ilya Segal, and James Wright; assistance from past research assistants Nick Arnosti, Guillaume Saulnier-Comte, Ricky Chen, Alim Virani, and Chris Cameron; experimental infrastructure assistance from Steve Ramage; and help gathering data from Ulrich Gall, Rory Molinari, Karla Hoffman, Brett Tarnutzer, Sasha Javid, and others at the FCC. This work was funded by Auctionomics and by NSERC via the Discovery Grant and E.W.R. Steacie Fellowship programs.

References

Aardal, K. I.; Van Hoesel, S. P.; Koster, A. M.; Mannino, C.; and Sassano, A. 2007. Models and solution techniques for frequency assignment problems. *Annals of Operations Research* 153(1):79–129.

Bazelon, C.; Jackson, C. L.; and McHenry, G. 2011. An engineering and economic analysis of the prospects of reallocating radio spectrum from the broadcast band through the use of voluntary incentive auctions. TPRC.

Calamari, M.; Kharkar, O.; Kochard, C.; Lindsay, J.; Mulamba, B.; and Scherer, C. B. 2012. Experimental evaluation of auction designs for spectrum allocation under interference constraints. In *Systems and Information Design Symposium*, 7–12. IEEE.

Charikar, M.; Indyk, P.; and Panigrahy, R. 2002. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *Automata, Languages and Programming*. Springer. 451–462.

Congressional Budget Office. 2015. Proceeds from auctions held by the Federal Communications Commission. https://www.cbo.gov/publication/50128.

Dütting, P.; Gkatzelis, V.; and Roughgarden, T. 2014. The performance of deferred-acceptance auctions. In *Proc. of EC*, EC '14, 187–204. New York, NY, USA: ACM.

FCC. 2013. Office of engineering and technology releases and seeks comment on updated OET-69 software. *FCC Public Notice* DA 13-138.

FCC. 2014a. Comment sought on competitive bidding procedures for broadcast incentive auction 1000, including auctions 1001 and 1002. *FCC Public Notice* 14-191.

FCC. 2014b. FCC feasibility checker. http://wireless.fcc.gov/incentiveauctions/learn-program/repacking.html.

FCC. 2014c. In the matter of expanding the economic and innovation opportunities of spectrum through incentive auctions. *FCC Report & Order* FCC 14-50. particularly section IIIB.

FCC. 2014d. Information related to incentive auction repacking feasibility checker. https://www.fcc.gov/document/information-related-incentive-auction-repacking-feasability-checker.

FCC. 2014e. Repacking constraint files. http://data.fcc.gov/download/incentive-auctions/Constraint_Files/.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*. Springer. 260–265.

Gomes, C. P., and Selman, B. 2001. Algorithm portfolios. *AIJ* 126(1):43–62.

Hoffmann, J., and Koehler, J. 1999. A new method to index and query sets.

Hutter, F.; Lindauer, M.; Bayless, S.; Hoos, H.; and Leyton-Brown, K. 2014a. Configurable SAT solver challenge (CSSC) (2014). http://aclib.net/cssc2014/index.html.

Hutter, F.; López-Ibáñez, M.; Fawcett, C.; Lindauer, M.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2014b. AClib: A benchmark library for algorithm configuration. In *LION*. Springer. 36–40.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION*, 507523.

Järvisalo, M.; Le Berre, D.; Roussel, O.; and Simon, L. 2012. The international SAT solver competitions. *AI Magazine* 33(1):89–92.

Kazumori, E. 2014. Generalizing deferred acceptance auctions to allow multiple relinquishment options. *SIGMETRICS Performance Evaluation Review* 42(3):41–41.

Kearns, M., and Dworkin, L. 2014. A computational study of feasible repackings in the FCC incentive auctions. *CoRR* abs/1406.4837.

Kwerel, E.; LaFontaine, P.; and Schwartz, M. 2012. Economics at the FCC, 2011–2012: Spectrum incentive auctions, universal service and intercarrier compensation reform, and mergers. *Review of Industrial Organization* 41(4):271–302.

Luo, C.; Cai, S.; Wu, W.; and Su, K. 2014. Double configuration checking in stochastic local search for satisfiability. In *AAAI*.

Marcus, M. J. 2013. Incentive auction: a proposed mechanism to rebalance spectrum between broadcast television and mobile broadband [spectrum policy and regulatory issues]. *Wireless Communications* 20(2):4–5.

Milgrom, P., and Segal, I. 2014. Deferred-acceptance auctions and radio spectrum reallocation. In *Proc. of EC*. ACM.

Milgrom, P.; Ausubel, L.; Levin, J.; and Segal, I. 2012. Incentive auction rules option and discussion. *Report for Federal Communications Commission. September* 12.

Nguyen, T.-D., and Sandholm, T. 2014. Optimizing prices in descending clock auctions. In *Proc. of EC*, 93–110. ACM.

Nudelman, E.; Leyton-Brown, K.; Andrew, G.; Gomes, C.; McFadden, J.; Selman, B.; and Shoham, Y. 2003. Satzilla 0.9. Solver description, International SAT Competition.

Patrascu, M. 2011. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing* 40(3):827–847.

Savnik, I. 2013. Index data structure for fast subset and superset queries. In *Availability, Reliability, and Security in Information Systems and HCI*, 134–148. Springer.

Vohra, A. 2014. On the near-optimality of the reverse deferred acceptance algorithm.

Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, 210–216.