CS264: Beyond Worst-Case Analysis
Lecture #20: Application-Specific Algorithm Selection*

Tim Roughgarden†

March 16, 2017

1 Introduction

A major theme of CS264 is to use theory to obtain accurate guidance about which algorithm to use to solve a given problem in a given domain. For most problems, there is no "one size fits all" algorithm, and the right algorithm to use depends on the set of inputs relevant for the application. In today's lecture, we'll turn this theme into a well-defined mathematical problem, formalized via statistical learning theory. Alternatively, we can think of today's lecture as a more general approach to the self-improving algorithms discussed in the last lecture.

Before presenting the formalism, we consider three motivating examples.

1.1 SATzilla

You may or may not know that, in every odd calendar year, there is a SAT competition. Anyone can submit a SAT solver, which is then run on a number of instances (both synthetic and real instances from various application domains). A team from the University of British Columbia [4] wanted to enter the 2007 competition but didn't really want to write a new SAT solver from scratch. So they came up with the idea of making intelligent use of existing solvers, rather than designing a new one.

The UBC team used a portfolio of 7 state-of-the-art SAT solvers, including winners of previous competitions. An interesting fact about these solvers is that each exhibits dramatic variation in running time across different inputs of the same size (many orders of magnitude). Also, different solvers tend to do well on different types of inputs. These facts suggest using a "meta-algorithm" that runs the most appropriate SAT solver for the instance at hand. The basic idea behind the meta-algorithm (called "SATzilla") is the following, for a given SAT formula φ:

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

1. Spend a small amount of time (like 1 second or .1 second) computing various features of φ. Some features are obvious, like the number of variables and clauses. Other features come from the *factor graph*, which is the bipartite graph with clauses on one side and variables on the other side, with edges between each clause and the variables in the clause. One clever and (as it turns out) very useful feature is to run a simple algorithm, like local search or unit propagation, for a small amount of time and see how much progress it makes (e.g., in terms of number of satisfied clauses).

2. For each of the 7 solvers, learn a *prediction function*, which predicts the running time of the solver on a given input, as a function of the input's features. With a good set of features, one can even get away with a linear predictor (i.e., a predicted running time that is a linear function of the features). Standard machine learning techniques were used to come up with these prediction functions. This involves finding training data (in the form of benchmark instances), which can be obtained by sampling from a generative model (e.g., random-SAT with a given clause/variable ratio) or considering publicly available benchmarks. One can then run the SAT solver on all of these benchmark instances, and use linear regression to compute the best-fit linear predictor.

3. Compute the predicted running time of each solver (using the prediction function for that solver). Run the solver with the smallest predicted running time.

Remarkably, SATzilla cleaned up at the 2007 SAT competition, winning multiple gold and silver medals across different categories. Two years later, a new version of the algorithm also did very well [6]. Nowadays, such meta-algorithms are explicitly banned from much of the competition.

1.2 Parameter Tuning

Much of the human and computing cycles used in applied machine learning are devoted to *parameter tuning*—figuring out the best values to use for one of more parameters of an algorithm. For example, you may think of gradient descent as a single algorithm, but any implementation depends on the (hard-coded) value of a couple parameters, most interestingly its step size.[1] In practice, it makes sense to experiment with value of the step size. One goal might be to minimize the number of iterations before convergence (for some fixed stopping criterion, like a target bound on the gradient's norm), on average over the "relevant inputs."

Another example in machine learning is setting a regularization parameter (e.g., in ridge regression). When training a model via optimization, regularization involves adding a penalty

[1] Recall that gradient descent is effectively a local search algorithm for unconstrained minimization. Given is a differentiable function from $\mathbb{R}^d$ to $\mathbb{R}$ that you want to minimize. The algorithm repeatedly computes the gradient at the current point, and moves in the direction of the negative gradient (the direction of "steepest descent") to obtain the next point. How far should one move in that direction? The answer is given by the step size. Set the step size too small and gradient descent will only make progress in tiny steps, requiring many iterations to converge. Set the step size too big and you run the risk of repeatedly shooting over all local minima, never converging at all.

term to the optimization problem to encourage "simple" solutions, and thereby hopefully avoiding "overfitting" the solution to the training data. (For example, if feasible solutions are vectors in $\mathbb{R}^d$, one might want to penalize solutions with large norm.) The regularization parameter controls how severe the penalty term is. Set the parameter too small and you risk overfitting. Set the parameter too big and you risk optimizing only the penalty term, thereby throwing out the baby with the bathwater. In practice, regularization parameters are typically set via trial-and-error, for example by minimizing error on a holdout set.

The problem of parameter tuning is not confined to machine learning. For example, consider CPLEX, one of the state-or-the-art solvers for linear programs and integer linear programs. CPLEX has no less than 135 parameters, requiring a 221-page manual [5]. And what guidance does the manual have to offer on choosing good values for the parameters? "You may need to experiment with them." (Gee, thanks CPLEX...)

1.3 Parameterized Heuristics

Our final example can also be considered a parameter-tuning problem, though the context is different, concerning fast inexact heuristics for NP-hard problems. For concreteness, let's focus on the weighted independent set (WIS) problem. Recall that an input of WIS is specified by an undirected graph $G = (V, E)$, and that each vertex $v \in V$ has a nonnegative weight w_v. An independent set is a subset of mutually non-adjacent vertices (a set $S \subseteq V$ such that $(u, v) \notin E$ for all $u, v \in S$). The objective is to compute an independent set with the largest-possible total weight. This problem is NP-hard, even to approximate (in the worst case), so typically one resorts to heuristics. As with most problems, greedy algorithms are a natural place to start.

Perhaps the most natural greedy algorithm takes a single pass over the vertices, from highest-weight to lowest-weight, always selecting the next vertex if it is feasible to do so.

Greedy Heuristic #1

1. Initialize $S = \emptyset$.
2. Sort the vertices in order of nonincreasing weight.
3. For each vertex v in turn:
   (a) If S does not contain a neighbor of v, then add v to S.
4. Return S.

For example, in the graph in Figure 1, the algorithm first selects the vertex in the lower left. It then has to skip all of the weight-3 vertices, and concludes by selecting the rightmost vertex. This independent set has total weight 6, whereas the optimal solution has total weight 8.

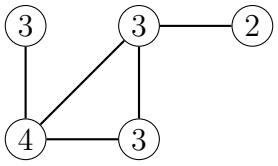

Figure 1: An instance of WIS. Each vertex is labeled with its weight.

The next idea is to pay attention not only to the weights of vertices, but also to their degrees. Intuitively, it makes sense to penalize high-degree vertices, because their selection excludes so many other vertices from future consideration. One way to do this is to sort vertices by "bang-per-buck" $w_v/(\deg(v)+1)$—reflecting that the selection of v nets a gain of w_v but "uses up" $\deg(v)+1$ vertices.

Greedy Heuristic #2

1. Initialize $S = \emptyset$.
2. Sort the vertices in order of nonincreasing $\frac{w_v}{\deg(v)+1}$.
3. For each vertex v in turn:
   (a) If S does not contain a neighbor of v, then add v to S.
4. Return S.

Note that the vertices are sorted once and for all up front. There is also an adaptive version of the algorithm, where every iteration the remaining vertices are re-sorted, with only the still-eligible neighbors contributing to the denominator.

For the example in Figure 1, this second greedy heuristic computes the optimal solution. But don't read too much into this: it's easy to find examples where the first heuristic recovers the optimal solution and the second does not.

A natural interpolation between these two greedy algorithms is to sort the vertices in nonincreasing order of $w_v/(\deg(v)+1)^p$. We call this algorithm Greedy(p). The first and second greedy heuristics correspond to Greedy(0) and Greedy(1), respectively.

Why bother with this interpolation? Intermediate values of p do not seem to be helpful in worst-case analyses. But if you're actually implementing this heuristic, you definitely want to experiment with different choices of p. There have been real-world applications where, on representative benchmark instances, intermediate values like $p = .4$ and $p = .6$ outperformed the extreme values of $p = 0$ and $p = 1$.

2 Formalism

We now describe our formalism for reasoning about application-specific algorithm selection. Fix a problem, like SAT, unconstrained convex minimization, or weighted independent set.

Fix a cost measure, like wall-clock time, number of iterations, or solution quality. Let $\mathcal{C}$ denote a set of algorithms, like a finite set of 7 SAT solvers, gradient descent with all possible step sizes, or the WIS heuristics GREEDY(p) for all $p \in [0,1]$. Note that $\mathcal{C}$ could be either finite or infinite.

The "best" algorithm in $\mathcal{C}$ (for the given problem, according to the given cost measure) generally depends on which instances of the problem are relevant to the application domain. Abstractly, and like the last lecture on self-improving algorithms, we identify a "domain" with an input distribution $\mathcal{D}$ that is a priori unknown. The goal is then to identify the best algorithm for the domain, meaning an algorithm in

$$\operatorname*{argmin}_{A \in \mathcal{C}} \mathbf{E}_{z \sim \mathcal{D}}[\text{COST}(A, z)], \tag{1}$$

at least up to a small amount of error.

Different algorithms from $\mathcal{C}$ will be better for different input distributions $\mathcal{D}$, so we need at least some information about what $\mathcal{D}$ is. In practice, performance is generally measured using agreed-upon benchmarks that are thought to representative of "real inputs" (whether it be in computer architecture, compiler design, machine learning, etc.) So we assume that some number s of benchmark instances are available, in the form of i.i.d. samples from the underlying distribution (again, like in self-improving algorithms).

3 Sample Complexity

Given samples/benchmarks $z_1, \ldots, z_s \sim \mathcal{D}$ from an unknown distribution $\mathcal{D}$, there is an obvious way to guess as to which algorithm $A \in \mathcal{C}$ is best for $\mathcal{D}$—just select the algorithm that is best on the samples. That is, select an algorithm in

$$\operatorname*{argmin}_{A \in \mathcal{C}} \frac{1}{s} \sum_{i=1}^{s} \text{COST}(A, z_i).$$

For the moment, let's study this approach without worrying about how to actually implement it (i.e., how to compute a minimizer efficiently).[2]

How well does this approach work? Is the algorithm that is best on the benchmark instances actually the best one for the underlying distribution? The answer depends on the number s of samples available. Intuitively, with few samples (e.g., $s = 1$), there is a risk of "overfitting," meaning that the empirically best algorithm is an artifact of the particular samples, rather than capturing anything important about the true distribution. At the other extreme, with an infinite number of samples, doing well on the samples is the same thing as doing well on the true distribution, and so the approach works perfectly. Thus the interesting question is a quantitative one: *how many* samples are necessary and sufficient to identify the best algorithm? That is, what is the *sample complexity* of this method? The

[2]Readers who have studied machine learning might recognize this approach as a version of the "empirical risk minimization" or "ERM" algorithm.

answer can be parameterized by the error ϵ that can be tolerated in approximating (1), the probability δ of failure (over the choice of samples), and any parameters of the particular problem being studied. As we'll see, these questions are right in the wheelhouse of a field known as statistical learning theory (as covered in CS229T).

4 Some Statistical Learning Theory

Let's start with a modest goal: what if we just want to estimate the expected performance of a single algorithm A from samples?

Lemma 4.1 *Let A be an algorithm with $\text{COST}(A, z) \in [0, 1]$ for every input z, and $\mathcal{D}$ an input distribution. Fix constants $\epsilon, \delta > 0$. Let $s = \frac{c}{\epsilon^2} \log \frac{1}{\delta}$ for a sufficiently large constant c (independent of ϵ and δ). Then with probability at least $1 - \delta$ over i.i.d. samples $z_1, \ldots, z_s \sim \mathcal{D}$,*

$$\left| \mathbf{E}_{z \sim \mathcal{D}}[\text{COST}(A, z)] - \frac{1}{s} \sum_{i=1}^{s} \text{COST}(A, z_i) \right| < \epsilon. \tag{2}$$

That is, the sample complexity of estimating the expected performance of a single algorithm (up to $\pm\epsilon$, with failure probability δ) is $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$. The proof is a straightforward application of the Hoeffding bound, and is left to Homework #10. The assumption that A's cost always lies in $[0, 1]$ reduces to the assumption that its cost is bounded (by scaling). When costs are not already bounded (like an algorithm that might run for ever), a bound can often be imposed with little loss in utility (e.g., imposing a timeout on the algorithm).

Now let's make the problem harder: suppose we have a set $\mathcal{C}$ of algorithms, and we want to estimate the expected performance of *every* algorithm $A \in \mathcal{C}$. First let's consider the easy case where $\mathcal{C}$ is finite (like the 7 SAT solvers in Section 1.1). Fix a desired failure probability δ, and set $\delta' = \delta / |\mathcal{C}|$. Lemma 4.1 implies that after

$$O\left(\tfrac{1}{\epsilon^2} \log \tfrac{1}{\delta'}\right) = O\left(\tfrac{1}{\epsilon^2} \left(\log \tfrac{1}{\delta} + \log |\mathcal{C}|\right)\right) \tag{3}$$

samples, the probability that a fixed algorithm $A \in \mathcal{C}$ fails to satisfy (2) is at most δ'. Taking a union bound over the algorithms of $\mathcal{C}$ gives the following.[3]

Lemma 4.2 *Let $\mathcal{C}$ be a finite set of algorithms, with $\text{COST}(A, z) \in [0, 1]$ for every input z and algorithm $A \in \mathcal{C}$, and $\mathcal{D}$ an input distribution. Fix constants $\epsilon, \delta > 0$. Let $s = \frac{c}{\epsilon^2}(\log \frac{1}{\delta} + \log |\mathcal{C}|)$ for a sufficiently large constant c (independent of ϵ, δ, and $|\mathcal{C}|$). Then with probability at least $1 - \delta$ over i.i.d. samples $z_1, \ldots, z_s \sim \mathcal{D}$,*

$$\left| \mathbf{E}_{z \sim \mathcal{D}}[\text{COST}(A, z)] - \frac{1}{s} \sum_{i=1}^{s} \text{COST}(A, z_i) \right| < \epsilon$$

for every algorithm $A \in \mathcal{C}$.

[3] In learning theory, this is sometimes called a "uniform convergence" result, since a single set of samples estimates simultaneously the performance of every algorithm in $\mathcal{C}$.

Lemma 4.2 immediately implies an error guarantee for the algorithm that is best on the samples (w.h.p.). If you know the expected performance of every algorithm up to $\pm\epsilon$, then you also know the algorithm with the best expected performance, up to 2ϵ.

Corollary 4.3 *Let $\mathcal{C}$, ϵ, δ, and s be as in Lemma 4.2. Then with probability at least $1-\delta$, if*

$$\hat{A} \in \operatorname*{argmin}_{A \in \mathcal{C}} \frac{1}{s} \sum_{i=1}^{s} \textsc{cost}(A, z_i),$$

then

$$\mathbf{E}_{z \sim \mathcal{D}}\Big[\textsc{cost}(\hat{A}, z)\Big] \leq \min_{A \in \mathcal{C}} \mathbf{E}_{z \sim \mathcal{D}}[\textsc{cost}(A, z)] + 2\epsilon.$$

What if $\mathcal{C}$ is an infinite set, like in our parameter tuning algorithms? If the parameter set is bounded and algorithm performance is continuous in the parameters, then one can reduce to the finite case via discretization (a la grid search). But statistical learning theory offers a more elegant solution, which works directly with infinite sets that are "low-dimensional" in some sense. The "dimension" of the set plays the role of $\log|\mathcal{C}|$ in Lemma 4.2, even though $\mathcal{C}$ is an infinite set. Before defining the notion of dimension that we use (called "pseudodimension"), we state the corresponding sample complexity guarantee.

Theorem 4.4 ([3, 1]) *Let $\mathcal{C}$ be a set of algorithms with pseudodimension d, and with $\textsc{cost}(A, z) \in [0,1]$ for every input z and algorithm $A \in \mathcal{C}$. Let $\mathcal{D}$ be an input distribution. Fix constants $\epsilon, \delta > 0$. Let $s = \frac{c}{\epsilon^2}(\log\frac{1}{\delta} + d)$ for a sufficiently large constant c (independent of ϵ, δ, and d). Then with probability at least $1-\delta$ over i.i.d. samples $z_1, \ldots, z_s \sim \mathcal{D}$,*

$$\left|\mathbf{E}_{z \sim \mathcal{D}}[\textsc{cost}(A, z)] - \frac{1}{s}\sum_{i=1}^{s} \textsc{cost}(A, z_i)\right| < \epsilon$$

for every algorithm $A \in \mathcal{C}$.

Given this theorem, the analog of Corollary 4.3 follows immediately. Thus *low pseudodimension implies low sample complexity.*

The two obvious questions now are: How is the pseudodimension of a set defined, exactly? And do sets that we care about actually have low pseudodimension?

The definition is a bit of a mouthful. The best way to understand it is through examples (see Homework #10). The definition concerns sets of real-valued functions, while we care about algorithms. So how do we think of an algorithm A as a real-valued function? As the map $z \mapsto \textsc{cost}(A, z)$ from inputs to algorithm performance.

Definition 4.5 (Pseudodimension of Real-Valued Functions) Let $\mathcal{F}$ be a set of real-valued functions with domain X.

(a) Let $Y \subseteq X$ be a finite set, and $t(y)$ a real-valued threshold for each $y \in Y$. The set $\mathcal{F}$ *shatters Y with respect to t* if for every subset $Z \subseteq Y$, there exists a function $f \in \mathcal{F}$ such that

(i) $f(y) \geq t(y)$ for every $y \in Z$;

(ii) $f(y) < t(y)$ for every $y \notin Z$.

(b) $\mathcal{F}$ *shatters* a finite set $Y \subseteq Z$ if there exist thresholds $t : Y \to \mathbb{R}$ such that $\mathcal{F}$ shatters Y with respect to t.

(c) The *pseudodimension* $\text{pdim}(\mathcal{F})$ of $\mathcal{F}$ is the maximum size of a shattered set $Y \subseteq X$ (or $+\infty$, if $\mathcal{F}$ shatters arbitrarily large finite sets).

A few comments. A different way to think about what's happening in part (a) is to think of each function $f \in \mathcal{F}$ as inducing a 0-1 labeling (w.r.t. t) of the points of Y, according to whether or not the function is above a point's threshold or not. Then (a) is saying that all $2^{|Y|}$ different 0-1 labelings of Y arise as one ranges over all functions $f \in \mathcal{F}$.[4] Intuitively, you should only see so many different function behaviors (for large $|Y|$) if $\mathcal{F}$ is a "complex" set of functions. For a finite set $\mathcal{F}$, it is almost immediate that $\text{pdim}(\mathcal{F}) \leq \log_2 |\mathcal{F}|$ (Homework #10), and in this sense the pseudodimension generalizes the notion of "size" to infinite sets of functions.

Infinite sets of "simple" functions can have finite pseudodimension (and hence finite sample complexity for our estimation problem in Theorem 4.4). The canonical example is the set of affine functions from $\mathbb{R}^d$ to $\mathbb{R}$, meaning functions of the form $f(x_1, \ldots, x_d) = a_0 + \sum_{i=1}^{d} a_i x_i$ for some $a_0, a_1, \ldots, a_d \in \mathbb{R}$. It turns out that the pseudodimension of this set is exactly $d + 1$ (see Homework #10 for the $d = 2$ case).

Since we can view an algorithm as a real-valued function (with respect to a performance measure), it makes sense to speak about the pseudodimension of a set of algorithms.

Definition 4.6 Let $\mathcal{C}$ be a set of algorithms and COST a performance measure with range $[0, 1]$. The *pseudodimension* $\text{pdim}(\mathcal{C})$ of $\mathcal{C}$ is the pseudodimension of the induced set $\mathcal{F} = \{z \mapsto \text{COST}(A, z)\}_{A \in \mathcal{C}}$ of real-valued functions.

The functions induced by algorithms are not at all linear. Do the sets of functions relevant for application-specific algorithm selection have low pseudodimension?

5 The Pseudodimension of Greedy Algorithms

Let's return to the GREEDY(p) heuristics for the weighted independent set problem (Section 1.3).[5] Recall what the GREEDY(p) algorithm does: first it sorts vertices in nonincreasing order of $w_v/(\deg(v) + 1)^p$, and then it greedily picks every vertex in turn that is not a neighbor of an already-chosen vertex. There are an infinite number of such algorithms, and the

[4] This should be familiar to readers who have studied the notion of the VC dimension of a set of binary-valued functions. Essentially, we're defining the pseudodimension of a set of real-valued functions as the VC dimension of the thresholded versions of these functions, for a worst-case (i.e., maximizing) choice of thresholds.

[5] The same results hold for all "singly-parameterized greedy algorithms," see [2].

behavior of these algorithms is not at all continuous in the parameter p.[6] Nevertheless, we will show that the pseudodimension of the corresponding set of functions is quite low. Our measure COST here is the quality of the solution output by the algorithm.[7]

Theorem 5.1 ([2]) *If $\mathcal{C} = \{\text{GREEDY}(p) : p \in [0,1]\}$ and the input distribution $\mathcal{D}$ is over weighted independent set problems with at most n vertices, then $pdim(\mathcal{C}) = O(\log n)$.*

Proof: We need to show that the real-valued functions induced by $\mathcal{C}$ cannot shatter any set $Y \subseteq X$ with size larger than $\Theta(\log n)$. So fix a finite set $Y \subseteq X$—remember, for us, Y is a finite set of instances of the weighted independent set problem. Fix thresholds $t : Y \to \mathbb{R}$.

Call two parameter values $p, q \in [0,1]$ *equivalent* if GREEDY(p) and GREEDY(q) operate identically for each input in Y, meaning that they sort the vertices of each input $z \in Y$ in exactly the same order. In particular, the solutions returned by GREEDY(p) and GREEDY(q) are identical for all $z \in Y$.

The key claim is that there are only $O(|Y|n^2)$ different equivalence classes. To see this, first note that the behavior of an algorithm GREEDY(p) on an instance z is entirely determined by the result of comparisons of the form

$$\frac{w_u}{(\deg(u)+1)^p} \text{ vs. } \frac{w_v}{(\deg(v)+1)^p} \tag{4}$$

for a pair u, v of vertices. The result of this comparison can flip as you vary p, but it flips at most once. (If p is such that equality holds in (4), then the vertex with the higher weight and higher degree wins for all smaller p, while the vertex with lower weight and lower degree wins for all larger p.) Since there are only $\binom{n}{2}$ such comparisons per input and only $|Y|$ inputs, as one varies $p \in [0,1]$, the behavior of GREEDY(p) changes $O(|Y|n^2)$ times. The set of p's between two such "transition points" belong to the same equivalence class, so there are $O(|Y|n^2)$ equivalence classes.

Now, suppose that $\mathcal{C}$ shatters the set Y with respect to the thresholds t. Since this requires inducing all $2^{|Y|}$ possible 0-1 labelings of Y with respect to t, this requires at least $2^{|Y|}$ different algorithm behaviors. (If GREEDY(p) and GREEDY(q) behave identically on all inputs in Y, then they induce the same 0-1 labeling, no matter what the thresholds are.) The key claim then implies that $2^{|Y|} = O(|Y|n^2)$, which can only be true if $|Y| = O(\log n)$. ■

Combining Theorems 4.4 and 5.1 shows that the best value of $p \in [0,1]$ (up to error 2ϵ in expected performance) can be learned from $O(\epsilon^{-2} \log n)$ samples. The proof of Theorem 5.1 also shows that this value of p can be computed in polynomial time—given a set Y of $\Theta(\epsilon^{-2} \log n)$ samples, just try one value of p from each of the $O(\epsilon^{-2} n^2 \log n)$ equivalence classes (note that the transition points are easy to identify), run the corresponding

[6] For example, arbitrarily small changes to p can change which vertex appears first in the ordering, which can then lead to a totally different algorithm execution and returned solution.

[7] Actually, since it's a maximization problem, we mean the negative of the solution quality (shifted and scaled to lie in $[0,1]$). Or alternatively, we could work with solution quality directly and redefine the ERM approach to pick the algorithm with maximum average performance on the given samples.

GREEDY(p) heuristic on every input in Y, and use the value of p with the best average performance on Y.[8]

References

[1] M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

[2] R. Gupta and T. Roughgarden. Application-specific algorithm selection. *SIAM Journal on Computing*, 46(3):992–1017, 2017.

[3] D. Haussler. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation*, 100(1):78–150, 1992.

[4] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.

[5] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *Proceedings of the RCRA workshop on combinatorial explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 16–30, 2011.

[6] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2012.

[8] This is for the non-adaptive version of GREEDY(p), where the vertices are sorted once and for all up front. Similar arguments work for the adaptive version, where degrees are recomputed in each iteration, based on the subgraph of vertices that are still eligible for future inclusion. The only difference is that instead of $O(n^2)$ comparisons of the form (4) for each input, there are now $O(n^4)$ such comparisons—one for each choice of u, v, and their current degrees (which are in $\{0, 1, 2, \ldots, n-1\}$). This still yields a pseudodimension bound of $O(\log n)$.