CS264: Beyond Worst-Case Analysis Lecture #5: Computing Independent Sets: A Parameterized Analysis *

Tim Roughgarden†

October 8, 2014

1 Preamble

This is our final lecture that focuses squarely on the parameterized analysis of algorithms, though the theme will resurface frequently in forthcoming lectures. This lecture's case study is on simple heuristics for approximating the maximum-weight independent set of a graph. We conclude the lecture with a tour d'horizon of parameters that are commonly used to parameterize problems and inputs.

Recall the two-step approach of parameterized analysis. The first step is to choose a natural parameterization of the inputs, which intuitively measures the "easiness" of an input. The second step is a performance analysis of an algorithm — running time, cost incurred, etc. — as a function of this parameter. We have already seen several examples. In Lecture #2, we gave three parameterized running time analyses of the Kirkpatrick-Seidel algorithm: a bound parameterized solely by the input size n ($O(n \log n)$), a bound parameterized by both the input size and the output size h ($O(n \log h)$), and a more complex "entropy parameter" necessary for proving the instance-optimality of the algorithm. Last lecture, we analyzed the page fault rate of the LRU paging algorithm, parameterized by the "degree of locality" of the input.

Recall the many motivations of parameterized analysis, which has the potential to make inroads on all three of our goals (the Prediction/Explanation Goal, the Comparison Goal, and the Design Goal). First, parameterized analysis is strictly stronger and more informative than worst-case analysis parameterized solely by the input size. Last lecture we saw how such more fine-grained analyses can achieve the Comparison Goal, allowing us to differentiate algorithms (like LRU vs. FIFO) that coarser analyses deem equivalent.

*©2014, Tim Roughgarden.

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

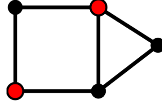

Figure 1: A graph in which the maximum size of an independent set is 2..

Second, a parameterized performance guarantee suggests *when* — meaning on which inputs, in which domains — a given algorithm should be used. (Namely, on the inputs where the performance of the algorithm is good!) Such advice is progress toward the Prediction Goal and is often useful in practice, in scenarios where someone has no time and/or interest in developing their own novel algorithm for a problem, and merely wishes to be an educated client of existing algorithms. For a simple example, for many graph problems the algorithms of choice are different for sparse and for dense graphs — this shows the utility of parameterizing the running time of a graph algorithm by the edge density of the input graph.

Third, parameterized analysis offers a two-step approach to explaining the good empirical performance of an algorithm with poor worst-case performance. The first step is to prove that the algorithm works well on "easy" inputs, such as request sequences with a high degree of locality. The second step is to demonstrate that "real-world" inputs are "easy" according to the chosen parameter, either empirically (e.g., by computing the parameter on various benchmarks) or mathematically (e.g., by positing a generative model and proving that it typically generates "easy" inputs).

The fourth reason is motivated by the Design Goal, unlike the previous three, which are motivated by the quest for more faithful and meaningful analyses of existing algorithms. A novel parameterization of performance sets up a principled way to explore the algorithm design space, which in turn can guide one to new and potentially better algorithms for a problem. This lecture is the first to illustrate this fourth motivation.

2 Case Study: The Maximum-Weight Independent Set Problem

2.1 Preliminaries

Recall that an instance of the maximum-weight independent set (MWIS) problem is defined by an undirected graph $G = (V, E)$ with a nonnegative weight $w_v \geq 0$ on each vertex $v \in V$. The goal is to output an independent set S with maximum-possible total weight $\sum_{v \in S} w_v$. Recall that S is an independent set if it contains no adjacent vertices — if $x, y \in S$ implies that $(x, y) \notin E$. See Figure 1. In the *unweighted* special case, $w_v = 1$ for every vertex v.

The (decision version of the) MWIS problem is NP-complete. You've learned that all NP-complete problems are equivalent from the standpoint of exact polynomial-time computation, but they are diverse in many other respects. Even among NP-hard optimization

problems, the MWIS problem is *really* hard.

Fact 2.1 ([6]) *For every $\epsilon > 0$, it is NP-hard to approximate the MWIS problem within a factor of $n^{1-\epsilon}$.*

Here's what we mean. For $c \geq 1$, a *c-approximation algorithm* for a maximization problem always returns a solution with value at least $1/c$ times the maximum possible. For the MWIS problem, it is trivial to achieve an n-approximation, where n is the number of vertices — just return the maximum-weight vertex. (This is an independent set, and no independent set has weight greater than $\sum_{v \in V} w_v$.) Fact 2.1 states that there is no significantly better polynomial-time approximation algorithm for the problem, unless $P = NP$ (in which case you might as well solve it exactly). Fact 2.1 holds also for the unweighted special case of MWIS.

Fact 2.1 implies that, if we judge polynomial-time heuristics by their worst-case approximation ratios, then there is not much to say about the MWIS problem — all we can do is narrow the window between n and $n^{1-\epsilon}$ (like $n/\sqrt{\log n}$, $n/\log n$, etc.). There has been work along such lines (e.g. [5]), but this lecture pursues a different approach. Our quandary is reminiscent of where things stood after we applied traditional competitive analysis to the online paging problem, with all options on the table being equally bad. (Note in the paging problem our handicap was not knowing the future; here it's having only a polynomial amount of time.) The lesson is the same: if an analysis framework is preventing us from reasoning about an important problem, devise an alternative analysis approach that is more informative.

2.2 The Randomized Greedy Algorithm

Our first idea, which is simple but good, is to parameterize the performance of heuristics for the MWIS problem by the maximum degree Δ of the graph. (You should be used to parameterizing graph algorithms by the number of vertices n and edges m.) Once we study some specific algorithms, there will be clear intuition about why the MWIS problem gets easier as Δ decreases. Vaguely, a heuristic that erroneously includes a vertex v in its independent set "blocks" the $\deg(v)$ neighboring vertices from being included, so small degrees suggest that no mistakes will be overly costly.

Next we give a parameterized analysis of the following *randomized greedy (RG) algorithm.* The algorithm does a single pass through the vertices, in a random order, greedily constructing an independent set in the obvious way (Figure 2). Remarkably, the algorithm doesn't even look at vertices' weights.

A simple lemma is the key to analyzing the RG algorithm.

Lemma 2.2 *For every vertex v,*

$$\mathbf{Pr}[v \in S] \geq \frac{1}{\deg(v) + 1}, \tag{1}$$

where the probability is over the random ordering chosen in the first step of the algorithm.

1. Order the vertices $V = \{1, 2, \ldots, n\}$ according to a random permutation.
2. $S = \emptyset$.
3. For $i = 1$ to n:
   (a) if $S \cup \{i\}$ is an independent set
   add i to S.
4. Return S.

Figure 2: The randomized greedy (RG) algorithm.

Proof: Let v be an arbitrary vertex v and $N(v)$ its set of $\deg(v)$ neighbors. Every relative ordering of $v \cup \{N(v)\}$ is equally likely, so v is first among these with probability exactly $1/(\deg(v) + 1)$. Whenever this event occurs, v is added to the output set S. ■

Observe that (1) need not hold with equality; a vertex v might get lucky and be included in the output set S even if one of its neighbors w appears earlier in the ordering — w might have been blocked by one of *its* earlier neighbors.

With Lemma 2.2 in hand, it is easy to give a parameterized approximation guarantee for the RG algorithm.

Corollary 2.3 *If S is the output of the RG algorithm, then*

$$\mathbf{E}\left[\sum_{v \in S} w_v\right] \geq \sum_{v \in V} \frac{w_v}{\deg(v) + 1},$$

where the expectation is over the random ordering chosen by the algorithm.

Proof: The corollary follows immediately from Lemma 2.2 and linearity of expectation. In more detail, let W_v denote the contribution of vertex v to the total weight of S — this is w_v if $v \in S$ and 0 otherwise. Then,

$$\mathbf{E}\left[\sum_{v \in S} w_v\right] = \mathbf{E}\left[\sum_{v \in V} W_v\right] = \sum_{v \in V} \mathbf{E}[W_v] = \sum_{v \in V} w_v \cdot \mathbf{Pr}[v \in S] \geq \sum_{v \in V} \frac{w_v}{\deg(v) + 1},$$

where the inequality follows from Lemma 2.2. ■

Recalling that $\sum_{v \in V} w_v$ is a trivial upper bound on the weight of MWIS, we have the following.

Corollary 2.4 *For every graph with maximum degree Δ, the expected total weight of the independent set output by the RG algorithm is at least $1/(\Delta + 1)$ times that of an optimal solution.*

Two comments. First, Corollary 2.4 continues to hold for an inferior version of the randomized greedy algorithm, where vertex i is added to S if and only if none of its $\deg(v)$ neighbors have been seen yet. (The output is a subset of that of the Randomized Greedy algorithm, and need not be a maximal independent set.) Lemma 2.2 holds with equality for the modified algorithm, so the two corollaries continue to hold for it. Second, the Randomized Greedy algorithm can be derandomized without any loss in the approximation guarantee; we leave the details to Homework #3.

2.3 The Recoverable Value

The good news is that Corollary 2.4 is another successful parameterized analysis of a natural algorithm for a fundamental problem. Its performance is bad in the worst case — Δ can be as large as $n-1$ — but we expected this in light of Fact 2.1. In graphs with a small maximum degree, however, it offers a reasonably good approximation guarantee. For the unweighted special case of MWIS, it also has an approximation guarantee of Δ_{avg}, where $\Delta_{avg} = 2m/n$ is the average degree of the graph (see Homework #3).

The bad news is that the parameter Δ is "brittle," in the sense that small changes to a graph can change the parameter a lot. For example, suppose G is a graph with a large number n of vertices and a small maximum degree Δ. Corollary 2.4 implies that the RG algorithm outputs an independent set with weight reasonably close to the maximum possible on this graph. Now obtain the graph G' from G by adding one new vertex v^*, connected to every vertex of G. Assuming the weight of v^* is not massive, the maximum-weight independent set is unchanged. The performance of the RG algorithm is almost the same — the only difference is that once in great while (with probability $1/(n+1)$) vertex v^* will be considered first, resulting in a bad solution. Thus, the expected value of the solution returned by the RG algorithm is still at least roughly a $1/(\Delta+1)$ fraction of the maximum possible. But our analysis, in the form of Corollary 2.4, only proves an approximation ratio of $n+$).[1]

Once again, a stupid example motivates modifying our method of analysis. The key observation about this example is: while G' has a large maximum degree, the optimal solution I^* comprises only vertices with low degree (at most $\Delta+1$ in G'). Thus, starting from Corollary 2.3, we have

$$\mathbf{E}\left[\sum_{v \in S} w_v\right] \geq \sum_{v \in V} \frac{w_v}{\deg(v)+1} \geq \sum_{v \in I^*} \frac{w_v}{\deg(v)+1}, \tag{2}$$

which is enough to imply Corollary 2.4!

Motivated by this observation, our second and more refined idea is to parameterize *according to the degrees in the optimal solution.* We haven't seen a parameter quite like this before; the closest analog occurred in Lecture #2, when we discussed parameterizing the

[1] Adding a bunch of vertices like v^* can blow up the average degree without significantly affecting the performance of the RG algorithm; thus even the analysis for the unweighted special case is vulnerable to this argument.

running time of the Kirkpatrick-Seidel algorithm in terms of the output size h (in addition to the input size n).

Generalizing the argument in (2), the RG algorithm satisfies

$$\mathbf{E}\left[\sum_{v \in S} w_v\right] \geq \underbrace{1}_{\text{“recoverable value”}} \cdot \max_{\text{indep set } I} \sum_{v \in I} \frac{w_v}{\deg(v)+1}. \tag{3}$$

The RG algorithm satisfies a stronger statement, with I ranging over *all* subsets of V, not just independent sets. The point of (3) is that every algorithm that satisfies it has guaranteed good performance whenever there is a near-optimal independent set comprising only low-degree vertices: if there is a c-approximate solution with all vertex degrees at most Δ, then the algorithm's approximation ratio is at most $c(\Delta + 1)$. For unweighted instances of MWIS, this guarantee holds also with the maximum degree replaced by the average degree.[2]

Do “typical” instances of the MWIS problem have near-optimal solutions with only or mostly low-degree vertices? Beyond the vague intuition that lower degree vertices are generally better than higher-degree ones, because they “block” the inclusion of fewer other vertices, it's hard to say. It's also hard to check — the condition references the optimal MWIS solution, which we know is hard to compute or approximate well. This may seem like a fatal flaw if you insist on interpreting the condition literally, but don't forget that striving for mathematical performance guarantees is also the means to other important ends. First, the set of instances for which one can prove guaranteed good performance of an algorithm is often a small subset of the instances for which the algorithm enjoys good empirical performance. Thus proving performance guarantees for a broad class of instances suggests that the algorithm could enjoy robustly good performance in practice.[3] Second, proposing a performance guarantee like (3) to shoot for naturally guides the design of new algorithms, which can be interesting and useful independent of any specific guarantee. The next section nicely illustrates this point.

2.4 Optimizing the Recoverable Value

We now shift our focus, for the first time in these lectures, from algorithm analysis to algorithm design. Having posited the novel type of performance guarantee in (3), the Pavlovian response of any good algorithm designer is to ask: *can we do better?* Meaning, can we replace the “1” on the right-hand side of (3) with a bigger number?[4] The constant “1” is the best possible for the RG algorithm (see Homework #3), so improving it requires designing a new algorithm. This constant is called the *recoverable value*; the original definition is due to [2], and the instantiation here for the MWIS problem is from [3]. While we motivated

[2]The next sequence of lectures on clustering problems expands on this theme of “performing well whenever the optimal solution is ‘well-structured’ ”.

[3]A hypothesis which, of course, should then be tested on “real” data.

[4]Strictly speaking, we mean replacing, for every vertex v, the coefficient $1/(\deg(v)+1)$ by a coefficient $\min\{1, c/(\deg(v)+1)\}$ for $c > 1$. For example, if G is a bunch of isolated vertices (all with degree 0), we clearly can't improve over the “1”.

1. Order the vertices $V = \{1, 2, \ldots, n\}$ according to a random permutation.
2. $T = \emptyset$.
3. For $i = 1$ to n:
   (a) if at most one of i's neighbors has already been seen:
   add i to T.
4. Return a maximum-weight independent set S of the graph $G[T]$ induced by T.[6]

Figure 3: The Feige-Reichman (FR) algorithm.

this parameter as a more "robust" parameterization of MWIS instances than the maximum degree of the input graph, in the spirit of our Explanation/Prediction Goal, we'll see next that it naturally guides us to some cool new algorithms, thereby making progress on our Design Goal.

We first point out that the window of opportunity for improvement is fairly small: unless $P = NP$, we cannot replace the "1" with a "4" for a polynomial-time algorithm. To see this, consider instantiating (3) on a cubic graph ($\deg(v) = 3$ for all v) with the "1" replaced by a 4. The guarantee would say that the algorithm's output has weight at least that of every independent set — i.e., that the algorithm solves the problem exactly. The MWIS problem is NP-hard on cubic graphs (see [4]), so this guarantee is unachievable unless $P = NP$. This holds even for the special case of unweighted instances.

The main result of this section is a neat polynomial-time algorithm that does improve the constant.

Theorem 2.5 ([3]) *There is a randomized polynomial-time algorithm that, for every MWIS instance, outputs an independent set with expected weight at least*

$$\max_{\textit{indep set } I} \sum_{v \in I} w_v \cdot \min\left\{1, \frac{2}{\deg(v) + 1}\right\}.$$

As a bonus, the algorithm seems like a novel and worthy heuristic to try out — it's simple but smart — which provides some justification for the mathematical problem of trying to maximize the recoverable value.[5]

The algorithm is shown in Figure 3. If we replace the text "at most one neighbor" with "at most zero neighbors," we recover the slightly inferior version of the RG algorithm mentioned at the end of Section 2.2.

[5]For the special case of unweighted instances, more complex variations of the following algorithm and ideas achieve a recoverable value of 7/3 [3].

[6]Recall this means the graph with the vertex set T and the edges of G that have both endpoints in T.

Both the performance guarantee of Theorem 2.5 and its running time require explanation. The intuition is that T is a "sweet spot," retaining enough of the original graph G to enable a large recoverable value, while possessing simple enough structure that the MWIS S of $G[T]$ can be computed in polynomial time.

We recommend proving the following lemma yourself, before glancing at the (short) proof.

Lemma 2.6 *With probability 1 (over the random ordering), the graph $G[T]$ is a forest.*

Proof: If C is a cycle of G, then the last vertex of C in the ordering is surely omitted from T — its two neighbors on C have already been seen. Since at least one vertex from each cycle of G is omitted, the graph $G[T]$ that remains is acyclic (i.e., a forest). ■

Since the maximum-weight independent set of an acyclic graph can be computed in polynomial time — linear time even, see Homework #3 — the following corollary is immediate.

Corollary 2.7 *The FR algorithm can be implemented in polynomial time.*

We turn to the performance analysis.

Proof of Theorem 2.5: Fix an independent set I of G. With probability 1, $I \cap T$ is an independent set of $G[T]$. Since the FR algorithm returns the maximum-weight independent set of $G[T]$, we have

$$\mathbf{E}\left[\sum_{v \in S} w_v\right] \geq \mathbf{E}\left[\sum_{v \in I \cap T} w_v\right].$$

The point of this maneuver is that the right-hand side is much easier to understand — a vertex $v \in I$ contributes to the right-hand side if and only if it is included in T, which occurs if and only if it is first or second in the induced ordering on v and its neighbors $N(v)$, which by uniformity occurs with probability exactly $\min\{1, \frac{2}{|\{v\} \cup N(v)|}\} = \min\{1, \frac{2}{\deg(v)+1}\}$ (cf., Lemma 2.2). Applying linearity of expectation as in Corollary 2.3 completes the proof. ■

There is also a deterministic polynomial-time algorithm that achieves a recoverable value of 2 (see Homework #3). In contrast to the RG algorithm and its derandomization (Homework #3), this deterministic algorithm is completely different from the FR algorithm.

3 Parameterized Analysis: A User's Guide

3.1 Recap of Applications

We've taken a tour through three parameterized analyses: of the running time of the Kirkpatrick-Seidel algorithm (en route to instance optimality), in terms of an "entropy measure" of point sets; of the page fault rate of the LRU algorithm, in terms of the "locality" of a request sequence; and of the approximation guarantees of heuristics for the maximum-weight independent set problem, as a function of the degrees of the vertices in the optimal solution.

Given that you're unlikely to remember any of the details of these parameters or analyses for very long, what should you take away from them?

The first take-away of the last few lectures is simply: *you should aspire toward parameterized bounds* whenever possible. Let's review the reasons.

1. A parameterized bound is a mathematically stronger statement, containing strictly more information about an algorithm's performance, than a worst-case bound (parameterized only by the input size).

2. Fine-grained performance characterizations can differentiate algorithms when worst-case analysis cannot. We saw an example with the LRU and FIFO paging algorithms.

3. Parameterized analysis can explain why an algorithm has good "real-world" performance even when its worst-case performance is bad. The approach is to first show that the algorithm performs well for "easy" values of the parameter, and then make a case that "real-world" instances are "easy" in this sense. For example, our lectures in smoothed analysis will follow this paradigm closely.

4. A novel parameterization of algorithm performance provides a yardstick that can inspire new algorithmic ideas, with today's lecture being a good example.

5. Formulating a good parameter often forces the analyst to articulate a notion of "structure" in data, with paging being a great example. Ideas for algorithms that explicitly exploit such structure usually follow soon thereafter.

Given the "why" of parameterized analysis, what about the "how"? Given an algorithm for a problem, how does one figure out the "right" parameter? Our three examples have given you a bit of practice, but all of the parameters seem quite problem-specific (by necessity), and it's not clear that any of them are directly useful for other problems. This brings us to our second take-away: *learn a toolbox of tried-and-true ways to parameterize problems, inputs, and algorithm performance.* This is an important point that we'll unfortunately give short shrift, discussing it explicitly only in this section and in Homework #3. Hopefully our brief treatment encourages you to learn more, for whatever types of problems and goals are most relevant to your own work.

3.2 How Parameters Differ

Parameters necessarily come in different flavors, depending on the nature of the problem (e.g., geometric vs. graphical data), the quantity being parameterized, and the goal of the parameterization. Below we list four different axes that can be used to compare and contrast them.

1. *What quantity is being parameterized?* Are you analyzing the running time of an algorithm, the approximation quality of a heuristic, or something else? Our three parameterized analyses were for three different quantities: running time (of the Kirkpatrick-Seidel algorithm); absolute performance (page fault rates of online paging algorithms); and relative performance (the approximation ratio of MWIS heuristics).

2. *Input- vs. solution-defined parameters.* We've seen about a 50/50 split of the two different types so far. For example, recall our bound of $O(n \log h)$ on the Kirkpatrick-Seidel algorithm, where n (the input size) and h (the output size) nicely illustrate the two different types of parameters. Our parameter $\alpha_f(k)$ for online paging directly measures a statistic of the input. In our study of MWIS heuristics, we initially used an input parameter (the maximum degree), and then switched to the recoverable value, a solution-based parameter.

3. *For algorithm analysis or algorithm design?* (Or both.) Some parameters generally show up only in the analysis of an algorithm, whereas others are meant to quantify nice structure of an input to be explicitly exploited by an algorithm. Most of our parameterized analyses thus far have been of the former type — the Kirkpatrick-Seidel algorithm is defined independently of the entropy measure we used to analyze it, similarly with the LRU algorithm and our measure of locality. Today's lecture is an ambiguous example: one can imagine discovering the FR algorithm without knowing the definition of the recoverable value, but the latter led directly to the former. Several of the examples below (explored in Homework #3) are much clearer examples, with algorithms' descriptions directly referencing the parameter.

4. *In what sense is the input or solution "small" or "well structured"?* Parameters are meant to measure the "easiness" of a problem or of an instance of a problem, and generally translate to something being "small" or "simple." The appropriate measure of simplicity varies with the type of problem — it's hard to imagine many definitions relevant for both graphs and Euclidean space, for example — but several such notions, some described below, have broad applicability.

3.3 A Tour d'Horizon

It would be impossible to survey all of the different ways that computational problems and algorithms are commonly parameterized. The following list is meant to highlight some useful examples, illustrate the differences discussed in the previous section, and whet your appetite for further investigation.

1. As a warm-up, Homework #3 includes two exercises, about the Knapsack problem and an online scheduling problem, that measure the performance of greedy algorithms as a function of the "smallness" of the input (items or jobs). In terms of Section 3.2, these parameters are input-based, are designed to aid the analysis of algorithms (not their design), specifically their approximation ratios.

2. For geometric data, a common approach is to parameterize by the "dimension" of the input or output. For example, in computational geometry problems where the input is points in $\mathcal{R}^d$, the dimension d can be much more significant for the difficulty of a problem than the input size n (a.k.a. the "curse of dimensionality").

There are many useful notions of dimension. For points in Euclidean space, a natural one is to assume that the input or solution lies (approximately) in a low-dimensional subspace. But there are also useful definitions of dimension for non-Euclidean data. For example, for a metric space (X, d) — e.g., representations of a bunch of images X with a (dis)similarity measure d — the notion of "doubling dimension" is a useful way to parameterize many problems and algorithms, including nearest neighbor search (see Homework #3). For another example, recent work has defined the "highway dimension" of graph to help guide the design of very practical algorithms that compute the shortest route (e.g., driving directions) while using little time and space [1].[7]

Definitions of dimension are most commonly input-based parameters, and guide the design (not just the analysis) of algorithms.

3. For graphs, there are many useful parameters that quantify how "easy" a graph is. We saw a simple one, the maximum degree, in this lecture. An example of a more sophisticated graph parameter is the "treewidth", which quantifies how "tree-like" a graph is. Many NP-hard problems, including MWIS, can be solved in polynomial time on graphs of bounded treewidth, generally by dynamic programming (see Homework #3 for an example). This parameter was originally defined with purely theoretical motivations (see [8]), but is now regularly an important part of sophisticated optimization and machine learning algorithms. Like notions of dimension for geometric data, graph algorithms don't run quickly on graphs of bounded treewidth by accident — to be useful, the property needs to be exploited explicitly in algorithms.

4. A simple solution-based parameter is the "size" of an optimal solution. Many NP-hard problems can be solved in polynomial time when the optimal solution is small. Consider, for example, the problem of deciding whether or not a graph has a vertex cover of size k.[8] When k is a constant, the problem can be solved in polynomial time (roughly n^k) by brute-force search. Much more interesting is the fact that it can be solved in time 2^k times a polynomial function of n and m (see e.g. [7]). That is, the exponential dependence that we expect in any algorithm for the (NP-hard) Vertex Cover problem can factored out as a function of the output size k only, independent of the size of the graph. Problems with this property are called *fixed-parameter tractable (FPT)*, and the vibrant field of *parameterized complexity* classifies, among other things, which problems are FPT and which ones are not.[9] For the Vertex Cover problem, this means that the problem can be solved in polynomial time for all $k = O(\log n)$ (not just $k = O(1)$), and can be solved efficiently in practice for decent-sized graphs for all k up to 10 or 20 (not just 2 or 3). See Homework #3 for another example of an FPT result, for the NP-hard Hitting Set problem. Like the last couple of examples,

[7]Learning theory (e.g. [9]) offers several notions of dimensions for sets that have no obvious geometry, such as "VC dimension"; further discussion is outside the scope of these notes.

[8]Recall that vertex covers are complements of independent sets: so a set S of vertices is a vertex cover if it contains at least one endpoint of every edge of the graph.

[9]For much more on this topic, see CS266, taught by my colleague Ryan Williams.

FPT algorithms are generally designed explicitly for inputs with a small value of the relevant parameter.

5. Our final genre of parameters express the idea that the optimal solution is "meaningful" or "pronounced" in some way. Such parameters can be useful both for analyzing when general-purpose algorithms work well, and for designing algorithms that are tailored to inputs that are "easy" in this sense. One canonical example is the notion of "margin," common in machine learning. For example, consider the problem of computing a linear separator (i.e., a halfspace) that is consistent with labelled data (with labels "+" and "-") in $\mathcal{R}^d$. One common assumption is that, not only is there a linear separator consistent with all the data (+'s on one side of the boundary, -'s on the other), but there is one with a large margin, with all data points far from the separator's boundary. The size of the margin measures how "pronounced" the optimal separator is, and machine learning problems with large margins tend to be qualitatively easier than those without them.

   For another classical example, consider the power iteration method. This is the algorithm where, given an $n \times n$ matrix A (symmetric, say), one starts with a random unit vector x_0, and iteratively computes $x_i = Ax_{i-1}/\|Ax_{i-1}\|$. The intent of the algorithm is to compute the eigenvector of A that has the largest eigenvalue (in magnitude). Simple linear algebra shows that the convergence rate of this algorithm is governed by the extent to which the biggest eigenvalue exceeds that of the second-biggest (in magnitude). Thus, the more the top eigenvector "sticks out," the faster it will be found by the power iteration method.

   The next several lectures on "stable clustering" are another great example of parameterizing "easiness" according to how meaningful or robust the optimal solution is.

References

[1] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 782–793, 2010.

[2] U. Feige, N. Immorlica, V. S. Mirrokni, and H. Nazerzadeh. PASS approximation: A framework for analyzing and designing heuristics. *Algorithmica*, 66(2):450–478, 2013.

[3] U. Feige and D. Reichman. Recoverable values for independent sets. *Random Structures & Algorithms*, 2014. To appear.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[5] M. Halldórsson. Approximations of weighted independent set and hereditary subset problems. *Journal of Graph Algorithms and Applications*, 4(1):1–16, 2000.

[6] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, 2001.

[7] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2005.

[8] N. Robertson and P. D. Seymour. Graph minors II: Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[9] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, 2014.