CS264: Beyond Worst-Case Analysis
Lecture #5: Parameterized Approximation and Running Time Guarantees*

Tim Roughgarden†

January 24, 2017

1 Preamble

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. This is our final lecture that focuses squarely on the parameterized analysis of algorithms, though the theme will resurface frequently in forthcoming lectures.

Parameterized analysis can 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 finer-grained analyses can achieve the Comparison Goal, allowing us to differentiate algorithms (like LRU vs. FIFO) that coarser analyses deem equivalent.

Second, a parameterized performance guarantee suggests *when* — meaning on which inputs, or 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

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

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 (even when it has 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).

These three reasons are all motivated by the quest for more faithful and meaningful analyses of existing algorithms. The fourth reason is motivated by the Design Goal. 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 theme will be more prominent in this lecture than in previous lectures.

2 NP-Hard Problems

This lecture, and a majority of the remainder of the course, concerns computational problems that are *NP-hard*. These notes assume that you've seen the basics of NP-completeness at some point in your life, but really all you need to remember is:

1. NP-hard problems are ubiquitous in practice — you will certainly encounter some in your own work after you graduate.

2. An NP-hard problem admits no (worst-case) polynomial-time and always-correct algorithm, unless $P = NP$. (And $P = NP$ would be surprising, to say the least!)

It's important to recognize NP-hard problems as such, so that you don't waste any time trying to come up with an exact and polynomial-time algorithm for it. But your problem doesn't go away just because it's NP-hard. What can you do?

A natural approach is to relax at least one of the conditions of "worst-case polynomial running time" and "always-correct."

> **Escape route #1:** Relax correctness. That is, allow an algorithm to be wrong (or, for optimization problems, suboptimal) on some inputs.
>
> **Escape route #2:** Relax the polynomial time constraint. That is, allow an algorithm to run in super-polynomial time on some inputs.

Of course, we still want our algorithms to be non-trivial. If we relax correctness, we still want to be correct much more often than random guessing. If we relax the polynomial-time constraint, we still want to beat naive exhaustive search on all or many inputs.

The main point of today's lecture is that, for both of these escape routes, parameterized analysis can assist in obtaining meaningful guarantees. The next section illustrates parameterized approximation guarantees for polynomial-time algorithms for NP-hard problems (which are not always correct/optimal). Section 4 considers parameterized running time bounds of exact algorithms that do not always run in polynomial time.

3 Parameterized Approximation Guarantees

We will content ourselves with a simple example of a parameterized approximation guarantee, but the idea is widely useful. Homework #3 showcases a couple of the many examples.

3.1 Approximation Algorithms

In this section, we assume that our performance measure corresponds to the objective function of an optimization problem (and not the running time of the algorithm). For example: the length of a traveling salesperson tour; the number of clauses satisfied by a truth assignment in an instance of the satisfiability (SAT) problem; the total value of the items packed in a knapsack; the size of a clique; and so on.

We consider algorithms that are not always optimal. But hopefully they are not too suboptimal. This idea motivates the next definition.

Definition 3.1 (α-Approximation Algorithm) For $\alpha \geq 1$, an *α-approximation algorithm A* for a minimization problem satisfies

$$\text{cost}(A, z) \leq \alpha \cdot \text{cost}(OPT, z). \tag{1}$$

for every instance z of the problem.

In Definition 3.1, $\text{cost}(OPT, z)$ denotes the smallest objective function value of any feasible solution to the instance z (the shortest traveling salesperson tour, etc.), while $\text{cost}(A, z)$ is the objective function value of the solution returned by A on z. For maximization problems, the inequality goes the opposite direction, and $\alpha \leq 1$.[1] Generally, one thinks about approximation algorithms that are restricted to run in polynomial time; this will be the case for us, unless otherwise noted. This is a mathematical abstraction of a "fast heuristic."

So while a competitive ratio (Lecture #3) of α is the same as an instance-optimality guarantee (with approximation factor α) with the additional restriction that the designed algorithm be online, an approximation ratio of α translates to an instance-optimality guarantee with the additional restriction that the designed algorithm runs in polynomial time.

[1] And the notation "cost" should probably be replaced by something more appropriate, like "value."

3.2 The Knapsack Problem

We'll use the KNAPSACK problem, familiar from undergraduate algorithms, to make a number of points. Recall the setup: the input consists of n items, with values $v_1, v_2, \ldots, v_n$ and weights $w_1, w_2, \ldots, w_n$, and a knapsack capacity W. Assume that all numbers are positive integers. The objective is to identify a subset $S \subseteq \{1, 2, \ldots, n\}$ of the items to maximize the total value $\sum_{i \in S} v_i$, subject to the capacity constraint $\sum_{i \in S} w_i \leq W$. The KNAPSACK problem is very basic and comes up all the time in real life—whenever you need to share a bounded amount of a resource, you have a KNAPSACK problem (scheduling courses in a classroom, assigning computing jobs to servers, etc.).

The following fact is usually proved in an undergraduate complexity course (like CS154).

Fact 3.2 *The* KNAPSACK *problem is NP-hard.*

So let's think about fast heuristics. In the design of exact or approximation algorithms, it's often best to start by thinking about greedy algorithms. It's usually easy to devise one or more natural greedy algorithms, and in many cases it can be quickly determined whether or not any of them are any good. The most obvious greedy approach to the KNAPSACK problems is to process the items greedily in some order. But which order? All else being equal, we prefer items with higher values, and lower weights. This suggests assigning a "score" to each item that is increasing in value and decreasing in weight. There are many such functions, but probably the most natural one is the ratio v_i/w_i, also called the *density* (or "bang-per-buck").

Here's the greedy heuristic:

A Greedy Heuristic for Knapsack

1. Sort and re-index the items so that they are ordered by density:
$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$
2. Pack items into the knapsack in this order, subject to the capacity constraint (i.e., skip an item if there isn't enough room in the knapsack for it).

Is this greedy heuristic an α-approximation algorithm for the KNAPSACK problem, for a reasonable constant $\alpha > 0$? To see the issue, suppose:

- item #1 has value $v_1 = 2$ and weight $w_1 = 1$;
- item #2 has value $v_2 = 1000$ and weight $w_2 = 1000$;
- the knapsack capacity W is 1000.

The greedy algorithm packs the first item (its density is 2), which in turn does not leave enough room for the second item (with density 1). So the greedy solution has value 2, while the optimal solution packs only the second item and has value 1000. Since we can replace "1000" with any number that we want, this greedy algorithm is not an α-approximation algorithm for any constant $\alpha > 0$.

To get a meaningful approximation guarantee, we either need to fix the algorithm (making it smarter) or fix the analysis (by restricting or parameterizing the inputs). Typical algorithms courses focus on the first approach. For example, you can take the better solution of those produced by two different greedy algorithms: the above algorithm that packs item in order of decreasing density, and the algorithm that packs items greedily in order of decreasing values. It turns out that this "best-of-2" greedy algorithm is a $\frac{1}{2}$-approximation algorithm for the KNAPSACK problem. One can also work harder and use dynamic programming (after rounding the input data) to obtain a $(1-\epsilon)$-approximation algorithm for any ϵ (with the running time depending on $\frac{1}{\epsilon}$).

In this course, we are interested in the second approach. Maybe the simple sort-by-density heuristic already does well on all of the inputs that we care about?

The sort-by-density greedy heuristic has the following parameterized approximation guarantee: it is a $(1-\beta)$-approximation algorithm for instances in which no item takes up more than a β fraction of the knapsack.

Theorem 3.3 *If $w_i \leq \beta W$ for every $i \in \{1, 2, \ldots, n\}$ and some $\beta \in [0, 1]$, then the output of the greedy algorithm has value at least $1 - \beta$ times that of the maximum possible.*

For example, if every item consumes at most 10% of the shared resource, then the sort-by-density heuristic outputs a solution with value at least 90% times that of the optimal solution. Homework #3 asks you to prove Theorem 3.3.

Analogous parameterized guarantees are possible for many different approximation algorithms. Often such guarantees are implicit or buried in proofs of worst-case guarantees; it's always valuable to state the strongest, most finely-parameterized guarantee than you can.

4 Parameterized Running Times

The second approach to coping with NP-hard problems is to keep insisting on correctness (for all instances) but relax the constraint that the algorithm runs in polynomial time on every input. For an NP-hard problem, we expect that the running time of any correct algorithm will be exponential in the input size on some inputs. But can we parameterize the running time of such an algorithm so that it's exponential on some inputs but provably subexponential (or even polynomial) on all inputs for which the parameter is "small"? This question leads us to the vibrant field of *parameterized algorithms* and *fixed-parameter tractability*, which has been one of the hottest subareas of algorithms over the past decade. One could easily teach a semester-long course just on this subject (e.g., using the recent book [2]), but we'll only have time for a brief glimpse of it.

4.1 Knapsack Revisited

To see how a parameterized running time bound for an algorithm for an NP-hard problem might work, let's return to the KNAPSACK problem (Section 3.2). If you learned anything about the KNAPSACK problem in undergrad algorithms, it would have been an exact dynamic programming algorithm, running time $O(nW)$, where n is the number of items and W is the capacity of the knapsack. Note this is already an interesting example of a parameterized running time bound, with two parameters (n and W).

A common point of confusion is: given that the KNAPSACK problem is NP-hard, why doesn't this dynamic programming algorithm imply that $P = NP$? Well, what's the "input size" of an instance of KNAPSACK, as a function of n and W? We can describe an instance by writing down $2n+1$ positive integers, and a positive integer x can be described in $\lceil \log_2 x \rceil$ bits. Assuming for simplicity that the v_i's and w_i's are at most the knapsack capacity W, it follows that the description length of an instance of KNAPSACK is $O(n \log W)$. This makes sense: the input length is supposed to correspond to the number of keystrokes necessary to communicate the problem instance to a computer, and it doesn't take a million keystrokes to communicate the number "1,000,000," just one keystroke per digit.

With an input size of $O(n \log W)$, we see that the $O(nW)$-time dynamic programming algorithm is *not* a polynomial-time algorithm for the KNAPSACK problem. For example, suppose $W = 2^n$. Then the input length is $O(n^2)$ while the running time of the algorithm is $O(n2^n)$. On the other hand, our parameterized running time bound immediately identifies a subset of instances for which the algorithm *does* run in polynomial time, namely those instances for which W is bounded by a polynomial in n.

Said another way, the running time of the dynamic programming algorithm does run in time exponential in one parameter of the input ($\log W$), as one would expect for an exact algorithm for an NP-hard problem, but it runs in time polynomial in the other parameter (n). This is the basic idea behind fixed-parameter algorithms. To get more experience, let's look at a second example of a running time bound that is exponential in one parameter of the input but polynomial in another parameter.

4.2 The Vertex Cover Problem

In the VERTEX COVER problem, the input is an undirected graph $G = (V, E)$, and the goal to compute a vertex cover with minimum-possible cardinality, where a *vertex cover* is a subset $S \subseteq V$ of vertices that contains at least one endpoint of every edge of E. Or in English, given a bunch of (overlapping) two-person groups (e.g., two people capable of performing the same task), the goal is to hire as few people as possible while hiring a representative from every group.

For example, the minimum size of a vertex cover of a star graph is 1 (no matter how many vertices n there are); of a clique is $n - 1$; and of a cycle is $\lceil \frac{n}{2} \rceil$. (See Figure 1.)

The VERTEX COVER problem is a canonical NP-hard problem (as typically proved in a course like CS154). But could there be an algorithm better than brute-force search, that is guaranteed to run in polynomial time on a significant subset of instances?

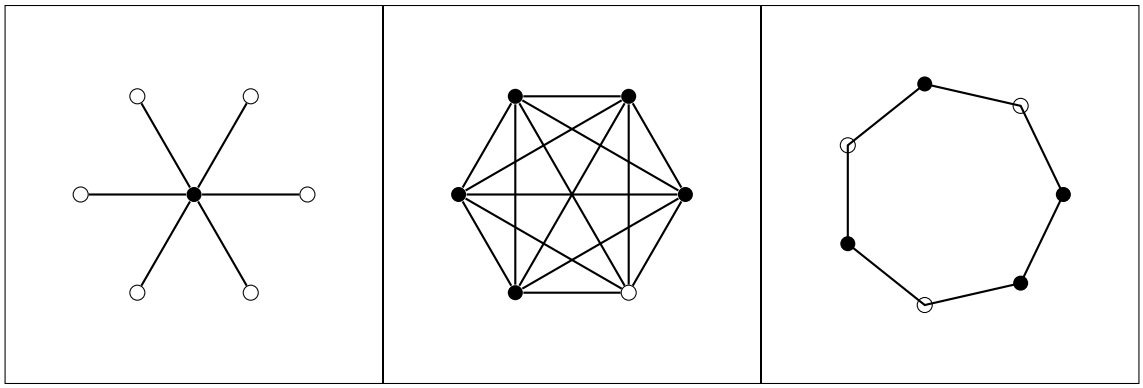

Figure 1: Minimum size vertex covers for a star graph, clique, and cycle.

4.3 Bounded-Depth Search Trees

We focus on the decision problem of checking whether or not an instance of Vertex Cover admits a vertex cover of size at most k (for a given target k). This problem is no easier than the general problem, since the latter reduces to the former by trying all possible values of k. Here, you should think of k as "small," for example between 10 and 20. The graph G can be arbitrarily large, but think of the number of vertices as somewhere between 100 and 1000. We'll show how to beat brute-force search for small k. (Perhaps hiring people is expensive, and we're only interested in a vertex cover if it's within budget.)

The naive brute-force search algorithm for checking whether or not there is a vertex cover of size at most k is: for every subset $S \subseteq V$ of k vertices, check whether or not S is a vertex cover. The running time of this algorithm scales as $\binom{n}{k}$, which is $\Theta(n^k)$ when k is small. While technically polynomial for any constant k, there is no hope of running this algorithm unless k is extremely small (like 3 or 4). Can we do better?

We consider the following algorithm (ignoring the base cases).

1. Pick an edge $e = (u, v)$ arbitrarily. (If there are no edges, then the empty set is already a vertex cover.)

   [Note: every vertex cover contains u, or v, or both.]

2. Recursively check if there is a vertex cover S of size $k-1$ in the graph $G \setminus \{u\}$ (i.e., G with u and u's incident edges removed). If so, return $S \cup \{u\}$.

   [This corresponds to "guessing" that u belongs to a small vertex cover, and proceeding recursively under the working hypothesis that this guess is correct.]

3. Recursively check if there is a vertex cover of size $k-1$ in the graph $G \setminus \{v\}$ (i.e., G with v and v's incident edges removed). If so, return $S \cup \{v\}$.

   [Ditto, for v.]

4. Halt and declare that G has no vertex cover of size k.

Why is this algorithm correct? If the first recursive call successfully computes a vertex cover S of size $k-1$, then $S \cup \{u\}$ is indeed the desired vertex cover of size k (S includes at least one endpoint of each of the edges of $G \setminus \{u\}$, and u covers the rest). Similarly for the second recursive call. What if neither one succeeds? This can only happen if there is no vertex cover of G of size k—if there were one, it would have to contain at least one of the vertices u, v, the remaining graph $G \setminus \{u\}$ or $G \setminus \{v\}$ would contain the other $k-1$ vertices of the vertex cover, and the corresponding recursive call would succeed.

The running time analysis is equally straightforward. The algorithm is recursive, so it's useful to think about its recursion tree (with nodes corresponding to recursive calls, and children of a node corresponding to the calls that it initiates). The branching factor of this recursion tree is 2 (one recursive call in each of the second and third steps). Since the target vertex cover size drops by 1 with each recursion level, the depth of the recursion tree is bounded by k. This means that there are at most 2^k recursive calls. Since the work done in each recursive call is minimal (easily implemented in linear time), the total running time of the algorithm is $O(2^k(m+n))$, where m and n denote the number of edges and vertices, respectively. This algorithm is efficient enough to handle values of k in the 20–30 range, for reasonable-sized graphs (hundreds of vertices, say). It has been implemented a number of times with reasonable results (even if no "killer app" has been identified).

We have now seen two parameterized running time bounds that are exponential in one but not all of the relevant parameters—one for our Knapsack algorithm with running time polynomial in n but exponential in $\log W$, and one for the algorithm above for Vertex Cover which is polynomial in n and exponential in k. In general, we say that a problem is *fixed-parameter tractable (FPT)* with respect to a parameter k if there is an algorithm that always solves the problem correctly and runs in time at most $f(k) \cdot \text{poly}(n,k)$, where n denotes the description length of the instance, f is an arbitrary (computable) function of k only, and $\text{poly}(n,k)$ is some polynomial in n and k (with exponent independent of n and k).[2] Obviously, we'd like the polynomial term to be as small as possible (ideally, linear). We'd also like the function $f(k)$ to be as small as possible, but for NP-hard problems, we expect $f(k)$ to be at least exponential in k. Every such fixed-parameter problem runs in polynomial time when the parameter k is sufficiently small (e.g., if $f(k) = 2^{\Theta(k)}$, then for all $k = O(\log n)$). Singly-exponential dependence on a parameter, like in the Vertex Cover algorithm above, is pretty much the best-case scenario for an exact algorithm for an NP-hard problem.

Just as some problems admit good approximation algorithms and others do not (assuming $P \neq NP$), some problems (and parameters) admit fixed-parameter algorithms while others do not (under appropriate complexity assumptions). This is made precise primarily via the theory of "W-hardness," which parallels the familiar theory of NP-hardness. For example, the independent set problem, despite its close similarity to the vertex cover problem, is "$W[1]$-hard" and hence does not seem to admit a fixed-parameter tractable algorithm (parameterized by the size of the largest independent set).

[2] $k = O(n)$ in most natural examples, in which case we can write the running time bound simply as $f(k) \cdot \text{poly}(n)$.

4.4 Kernelization Algorithms

There are a few general techniques for designing fixed-parameter tractable algorithms. One is the "bounded search tree" approach of the previous section, which can also be applied to several other problems (see [2] and Homework #3). Another one, and probably the most interesting one theoretically, is "kernelization." The idea is to preprocess the "easy part" of an instance, and hope that the remaining "hard part" is small, and so solvable by brute-force search, if nothing else. (For a concrete example, think of unit propagation in SAT problems, or see below.) Thus the study of kernelization algorithms is sort of a theory about when preprocessing is guaranteed to make a lot of progress. We will again illustrate the idea through the Vertex Cover problem, but know that this technique also applies to many other problems (especially graph problems, again see [2]).

We claim that the following is an FPT algorithm for the minimum-cardinality vertex cover problem (with budget k).

FPT Algorithm for Vertex Cover

set $S = \{v \in V : \deg(v) \geq k+1\}$
set $G' = G \setminus S$
set G'' equal to G' with all isolated vertices removed
if G'' has more than k^2 edges **then**
    return "no vertex cover with size $\leq k$"
else
    compute a minimum-size vertex cover T of G'' by brute-force search
    return "yes" if and only if $|S| + |T| \leq k$

The first five lines can be thought of as preprocessing, with the hard work only coming in line 7.

We next explain why the algorithm is correct. First, notice that if G has a vertex cover S of size at most k, then every vertex with degree at least $k+1$ must be in S. For if such a vertex v is not in S, then the other endpoint of each of the (at least $k+1$) edges incident to v must be in the vertex cover; but then $|S| \geq k+1$. In the second step, G' is obtained from G by deleting S and all edges incident to a vertex in S. The edges that survive in G' are precisely the edges not already covered by S. Thus, the vertex covers of size at most k in G are precisely the sets of the form $S \cup T$, where T is a vertex cover of G' with size at most $k - |S|$. Given that every vertex cover with size at most k contains the set S, there is no loss in discarding the isolated vertices of G' (all incident edges of such a vertex in G are already covered by vertices in S). Thus, G has a vertex cover of size at most k if and only if G'' has a vertex cover of size at most $k - |S|$. In the fourth step, if G'' has more than k^2 edges, then it cannot possibly have a vertex cover of size at most k (let alone $k - |S|$). The reason is that every vertex of G'' has degree at most k (all higher-degree vertices were placed in S), so each vertex of G'' can only cover k edges, so G'' has a vertex cover of size at most k only if it has at most k^2 edges. The final step computes the minimum-size vertex cover of G'' by brute force, and so is clearly correct.

Next, observe that in the final step (if reached), the graph G'' has at most k^2 edges (by assumption) and hence at most $2k^2$ vertices (since every vertex of G'' has degree at least 1). It follows that the brute-force search step can be implemented in $2^{O(k^2)}$ time. Steps 1–4 can be implemented in linear time, so the overall running time is $O(m) + 2^{O(k^2)}$, and hence the algorithm is fixed-parameter tractable. In FPT jargon, the graph G'' is called a *kernel* (of size $O(k^2)$), meaning that the original problem (on an arbitrarily large graph, with a given budget k) reduces in polynomial time to the same problem on a graph whose size depends only on k. Using linear programming techniques, it is possible to show that every unweighted vertex cover instance actually admits a kernel with size at most $2k$ (see Homework #3), leading to a running time dependence on k of $2^{O(k)}$ (as in our first algorithm) rather than $2^{O(k^2)}$.

In general, a *kernelization algorithm* (or simply *kernel*) for a parameterized problem accepts as input an instance (I, k) of the problem, runs in time polynomial in n (the description length of I in bits) and k, and outputs an instance (I', k') of the same problem such that: (i) k' and the description length of I' are bounded by some (computable) function $g(k)$ of k only; (ii) (I', k') is a "yes" instance if and only if (I, k) is a "yes" instance. The value $g(k)$ is the *size* of the kernel. If we allow g to be an arbitrary function of k, then the existence of a kernelization algorithm turns out to be equivalent to being fixed-parameter tractable (see Homework #3). For polynomial-size kernels, the story is more interesting: some fixed-parameter tractable problems have them (like Vertex Cover, above) while others (like Longest Path, below) do not, under appropriate complexity assumptions.

4.5 Color Coding

The study of fixed-parameter tractability has, to this point, been a primarily theoretical enterprise (and a quite interesting one). Some fixed-parameter algorithms are effectively unimplementable, because the parameter dependence is doubly exponential or worse. Other are perfectly implementable but not obviously superior to competing techniques (like the bounded-depth search tree algorithm for Vertex Cover). But there is one very nice example of a genre of fixed-parameter algorithms crossing over into practice; these are based on a technique known as *color coding*, introduced by Alon et al. [1].

As usual, we illustrate the general technique through the study of a canonical application, in this case the Longest Path problem. The input here is a graph $G = (V, E)$ (undirected, say) and a target path length k. The goal is to determine whether or not G has a k-path (a sequence of k distinct vertices $v_1, \ldots, v_k$, with $(v_{i-1}, v_i) \in E$ for $i = 2, 3, \ldots, k$).

One motivation for solving the Longest Path problem in practice is to find "motifs" in real-world networks. For example, in protein-protein interaction (PPI) networks, paths correspond to "linear pathways," which are of biological interest. In social networks, one generally looks for more complex (but still small) structures, and the color coding technique can be correspondingly extended. In both of these application domains, color coding led to advances in the state-of-the-art (see e.g. [3, 5, 4]).

Unlike the shortest-path problem, the Longest Path problem is NP-hard. For example, checking whether or not there is a $(n-1)$-path is equivalent to the Hamiltonian Path

problem, which should be familiar from a course like CS154. One can check for a k-path in time $O(n^k)$ using exhaustive search. Could the problem have a fixed-parameter algorithm?

The answer is yes, and the algorithm is very simple. It is randomized, and runs T independent trials of the following experiment (for T to be chosen shortly):

1. Independently assign a uniformly random color in $\{1, 2, \ldots, k\}$ to each vertex $v \in V$.
2. If there exists a panchromatic k-path (where each color appears exactly once), halt and return "yes."

If all T trials fail, then the algorithm returns "no."

Where did this algorithm come from? What makes the Longest Path problem hard is the constraint that the k vertices in the path are distinct. A simple sufficient (but not necessary) condition for a collection of vertices to be distinct is to have distinct colors. Intuitively, checking for this sufficient condition is easier than solving the original problem because, when building up a path edge-by-edge, one only has to keep track of the subset of colors already in your path (2^k possibilities) rather than the subset of vertices already in your path ($\approx n^k$ possibilities). This intuition may become particularly clear after you prove the following, using dynamic programming (Homework #3).

Lemma 4.1 *Checking if a given graph with vertex colors in $\{1, 2, \ldots, k\}$ has a panchromatic k-path can be done in $O(2^k \cdot poly(n))$ time.*

The algorithm is clearly correct whenever it says "yes" (since it provides a certificate). What is the probability that it says "no," even though the graph G contains a k-path? You will prove the following on Homework #3, using counting and Stirling's approximation (where $e = 2.718\ldots$).

Lemma 4.2 *If G has a k-path, then each trial succeeds with probability at least e^{-k}.*

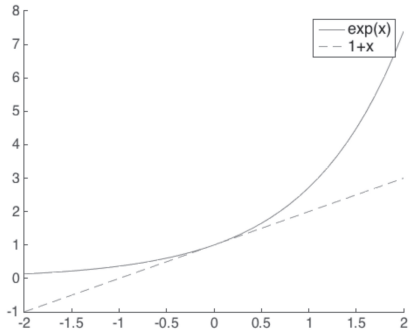

Figure 2: The inequality $1 + x \leq e^x$ holds for all real-valued x.

Recalling that $1 + x \leq e^x$ for all real-valued x (Figure 2), if we set $T = e^k \ln(\frac{1}{\delta})$, then at least one trial succeeds with probability at least

$$1 - \left(1 - e^{-k}\right)^T \leq 1 - e^{e^{-k}T} = 1 - e^{\ln(1/\delta)} = 1 - \delta.$$

Note that δ is a parameter than we can pick as small as we want. The total running time is then $O((2e)^k \text{poly}(n) \ln \frac{1}{\delta})$, which shows that the LONGEST PATH problem is fixed-parameter tractable (at least with a randomized algorithm). The algorithm can be modified to replace the constant $2e$ with a smaller constant, and it can also be derandomized; see [1, 2] for details.

References

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42:844–856, 1995.

[2] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[3] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 52:114–132, 2008.

[4] L. Romijn, B. Ó Nualláin, and L. Torenvliet. Discovering motifs in real-world social networks. In *Proceedings of the 41st International Conference on Theory and Practice of Computer Science (SOFSEM)*, pages 462–475, 2015.

[5] Z. Zhao, M. Khan, V. S. Anil Kumar, and M. V. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Proceedings of the 39th International Conference on Parallel Processing*, 2010.