1 Prelude

One goal of algorithms and complexity theory is to develop ways to formally measure the *quality* of algorithms, so we can compare why we would choose to run one algorithm over another. Many problems we want to solve are often 3SUM-hard, APSP-hard, SETH-hard, or worse, but they need to get solved anyway. What can we do? Over the years, multiple paradigms for coping with hard problems have been introduced:

- **Average-case analysis.** The instances we want to solve in practice aren't "worst case" but instead are "random" in some sense. (This is actually true for some problems in crypto.) There are notions of average-case fine-grained complexity and fine-grained cryptography, as your list of project papers shows!
- **Randomized algorithms.** Put randomness in the algorithm and see what happens. But complexity theorists widely believe that $\mathsf{P} = \mathsf{BPP}$ (polynomial time equals randomized polynomial time) so it's not clear that randomized algorithms are much more powerful!
- **Approximation algorithms.** Here the idea is to not try to solve the problem exactly, but get an approximately good solution instead. There are now well-known limitations on approximation as well, the most famous example being the PCP theorem [ALM+98, Din07]. Also, sometimes the notion of approximation doesn't make sense: for example, often the SAT problem is used in cases of "mission-critical verification": you want to satisfy *all* the clauses and if not all clauses are satisfied, the rocket will crash! In that case, you don't want to risk approximations.
- **Focus on solving "structured" practical instances.** Suppose you are studying a graph problem that looks hard to solve, but the graphs come from a real-world map. You might decide to focus on planar graphs where you can draw the graph on the two-dimensional plane without edges crossing. Or, if the graph is a *tree*, problems like Vertex Cover, Hamiltonian Path, and Independent Set become solvable in polynomial time. (If you haven't seen such algorithms before, think about how they might work!)

  More generally, one can ask, how "close" is a given graph to a tree? The notion of treewidth was designed to measure this. Without going into details about the actual definition of treewidth (probably a later lecture or problem set will?), trees have treewidth 1, cycles have treewidth 2, etc. **The treewidth of a given graph can be seen as a parameter measuring how close that graph is to a tree.** Moreover, treewidth can be used as a measure of how "hard" a given graph is to solve NP-hard problems on! It is known that the treewidth of a graph closely corresponds to how well algorithms can solve various NP-hard problems on graphs. For example,

  - Independent Set on graphs of treewidth at most k can be solved in $2^k \cdot \text{poly}(n)$ time [Nie06].
    (In fact, this running time cannot be improved to $(2-\varepsilon)^k \cdot \text{poly}(n)$ time, unless ETH is false! [LMS18].)
  - Hamiltonian Path on graphs of treewidth at most k can be solved in $2^{O(k)} \cdot \text{poly}(n)$ time [BCKN15].
- **"Exact" algorithms: Just try to definitively beat the "trivial" algorithm that tries all possible solutions.** Assuming $\mathsf{P} \neq \mathsf{NP}$, we forgo all expectations that this algorithm will actually run in polynomial time.

  Formally, suppose you have an NP problem V with a verifier algorithm V, and you know:

  1. $V(x, y)$ runs in $t(|x|, |y|)$ time, where $t(n, m)$ is a polynomial in n and m.
  2. If $x \in L$, then there is a **N***ifty* **P***roof* string y of length at most $b(|x|)$ bits such that $V(x, y)$ accepts.
  3. If $x \notin L$ then no such y exists.

Then, L can be decided in $O(2^{b(|x|)} \cdot t(|x|, |y|))$ time: on any input x, simply try all strings y of length up to $b(|x|)$, plugging these y into $V(x, y)$, and see if V ever accepts.

The witness length $b(|x|)$ can be viewed as a **parameter** to the problem L, and the length of this parameter can determine whether or not the problem is "hard".

> Question: When can we decide L in a way that reduces this $2^{b(|x|)}$ factor in the runtime, for "natural" verifiers?

What is a natural verifier? It depends on the problem! For a problem like SAT, the natural verifier treats the input x as a boolean formula, and y as a potential satisfying assignment on n variables. Then exhaustive search takes $2^n \cdot \text{poly}(|x|)$ time. Can we do better than 2^n? In some cases, yes! For example, we saw in the early lectures that 3SAT can be solved in less than $2^{0.3n}$ time.

For some problems, this proof length function $b(|x|)$ can be much smaller than $|x|$. For example, suppose we only look at the Boolean formulas which have $\log(n)$ variables, but the formula has size n. Then we can solve SAT for those instances in polynomial time.

What about the formulas with $(\log^2 n)$ variables? It's possible they could be solved in polynomial time as well. This wouldn't imply $\mathsf{P} = \mathsf{NP}$, but would still be a major advance, it would in fact refute ETH!

Although such improvements may look silly at first, it turns out that universal improvements over exhaustive search would have major implications for computational complexity. In fact, looking for algorithms that run slightly faster than brute-force is currently the only plausible-looking program we know for proving lower bounds in circuit complexity! (See for example [Wil17].)

2 Parameterization

The idea of parameterization subsumes the last two approaches discussed above. To introduce it, let's set up some notation. Let Σ be some finite alphabet; typically $\Sigma = \{0, 1\}$.

Definition 2.1 *A* **parametric problem** *is a subset of* $\Sigma^\star\{a\}^\star$*, where* a *is a symbol not in* Σ*.*

(Contrast with the usual definition of a decision problem, as a subset of $\Sigma^\star$.) We think of a parametric problem instance xa^k as having two items: an input $x \in \Sigma^\star$ and a parameter $k \in \mathbb{N}$. *In what follows, we will think of parametric problems as subsets of* $\Sigma^\star \times \mathbb{N}$*, for simplicity.* In particular, the notation is a little cleaner: an instance xa^k becomes (x, k).

The parameter can have many interpretations, as mentioned above. You might think of k as a "hardness measure" which tells us how hard the problem instance is. But often in practice, we don't know how hard a problem instance is, until we start trying to solve it! It's better to think of k as just some extra input, an extra constraint on the problem instance.[1]

Some Important Examples of Parametric Problems. Here are a few parametric problems to start. Each of them use the parameter to bound the size of a solution to the instance.

- Recall a graph $G = (V, E)$ has a vertex cover of size k if there is a subset $S \subseteq V$ such that $|S| \leq k$ and for all $\{u, v\} \in E$, either $u \in S$ or $v \in S$. Then we define:
$$k\text{-Vertex Cover} = \{(G, k) \mid G \text{ has a vertex cover of size at most } k\}.$$

[1] There are some natural situations in practice where we may not know the parameter in advance. (There are sorting algorithms which run faster if the input is "close" to being sorted, but you don't know how far it is from being sorted until you run the algorithm. There, the parameter would be "distance from sorted order" in swap distance, but we wouldn't know that parameter until the data's actually sorted!) This is an interesting but different notion of parameterization.

- k-Clique $= \{(G, k) \mid G$ has a clique of size at least $k\}$
- k-SUM $= \{(S, k) \mid S$ is a set of numbers and there are k numbers in S that sum to $0\}$
- A graph $G = (V, E)$ has a dominating set of size k if there is a subset $S \subseteq V$ such that $|S| = k$ and for all $v \notin S$, v is a neighbor of some $w \in S$. Then we define
$$k\text{-Dominating Set} = \{(G, k) \mid G \text{ has a dominating set of size } k\}.$$

All of these problems have the property that, when k is a fixed constant, they are solvable in polynomial time, indeed in $O(n^{k+c})$ time for a constant $c \geq 1$. To represent a set of k nodes or k numbers, we need only $(k \log n) + O(1)$ bits, so we can just enumerate and check all of them. We can "blame the difficulty" of Vertex Cover, Clique, and Subset-Sum on the fact that the solutions can sometimes be large: if k is large, in each of these problems we don't know of a significantly faster way to find the object of size k, than to try all possible choices for the object.

2.1 Key Definitions in Fixed-Parameter Tractability (FPT)

Those are examples of problems. Let's now introduce our notion of feasibility: what will count as a "feasible" parametric problem? First, we define a notion of a "feasible running time" function. (The below notion takes the place of "polynomial running time" in parametrized complexity.)

Definition 2.2 *A function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is* **FPT** *if there is a computable function $f : \mathbb{N} \to \mathbb{N}$ and a constant $c \geq 1$ such that for all n, k, $g(n, k) \leq f(k) \cdot n^c$.*

For example, $f(n, k) = 2^k \cdot n^2$ and $f(n, k) = 2^{2^k} \cdot n^3$ are FPT functions.

Definition 2.3 *A parameteric problem L is* **FPT** *if there is an FPT function g and an algorithm A such that for all $(x, k) \in \Sigma^\star \times \mathbb{N}$, $A(x, k)$ runs in $O(g(|x|, k))$ time and $(x, k) \in L \iff A(x, k)$ accepts.*[2]

Definition 2.4 FPT *is the class of parametric L which are FPT.*

This notion was formally introduced by Rod Downey and Mike Fellows in 1992. The intuition is that for *every* constant value of the parameter k, we have an algorithm for L that runs in $O(n^c)$ time, where c is fixed and does not depend on k. (The function $f(k)$ is a "constant" factor.) But of course, we put no real bounds on how big f can be! This is the power of FPT algorithms: as long as the extra parameter is "small", we're guaranteed to be polynomial time, and the degree of the polynomial is *independent* of the parameter k. And it's our hope that, for the instances of the parametrized problem that we *want* to solve, this parameter is indeed small. (And of course we want f to grow as slowly as possible too.)

Here is a silly example of a parameterization. Every decidable problem has *some* parameterization which makes it FPT: for every decidable L, you could define $L' = \{(x, |x|) \mid x \in L\}$.

Exercise: Show that L' is in FPT.

But this is not an interesting parameterization! For one, we want the parameter to be *smaller* than the input length. There is an art to defining a "good" parametric problems: you want to choose a parameter which is

(a) likely to be small, but

(b) the instances with a small parameter are still a large and interesting set that you want to solve.

[2] You might ask: what is the algorithm here? What is the computational model? It's the one from your algorithms courses – the "random access machine" model, where we allow constant-time memory accesses and all that good stuff. Generally speaking though, it doesn't really matter much: almost everything we say will also hold for multitape Turing machines, for example, especially up to polynomial factors.

The following "O-star" notation can be useful.

Definition 2.5 *A function $f(n,k)$ is $O^\star(g(k))$ if there is a constant c such that for all (n,k), we have $f(n,k) \leq g(k) \cdot n^c$.*

This notation lets us suppress polynomial-in-n factors, and focus on the parameter dependence.

2.2 Case Study in Parameterization: CNF Satisfiability.

Different parameterizations of the CNF-SAT lead to different ways of thinking about this hard problem.

- Consider k-SAT $= \{(F,k) \mid F$ is a satisfiable k-CNF formula$\}$.
  This is *not* a "good" parameter choice, for the following reason.

> **Exercise:** Prove that $\mathsf{P} \neq \mathsf{NP}$ if and only if k-SAT is not FPT.

  So the FPT-ness of this problem depends on whether $\mathsf{P} = \mathsf{NP}$; the problem is easy for $k = 2$ and insanely hard for $k = 3$ already.

- Now consider the parametric problem CNF-SAT-VARS $= \{(F,k) \mid F$ is a satisfiable CNF formula on k variables$\}$.

  CNF-SAT-VARS is FPT, as it can be solved in $O^\star(2^k)$ time. Is there a faster algorithm, running in $O^\star(1.999^k)$ time? That question is basically what the Strong Exponential Time Hypothesis asks.

- Consider CNF-SAT-CLAUSES $= \{(F,k) \mid F$ is a satisfiable CNF formula on k clauses$\}$.

  As you showed on your problem set, CNF-SAT-CLAUSES is FPT, solvable in time $O^\star(2^k)$. So for this parameterization, a less-than-2^k time bound *is* possible here. A paper recently appearing on the arXiv claims the best-known dependence on k, with an algorithm running in $O^\star(1.23^k)$ time [CXZ20].

- Consider CNF-SAT-LENGTH $= \{(F,k) | Fis a satisfiable CNF formula with k total literal occurrences\}$.

> **Exercise:** Show that any $O^\star(f(k))$ time algorithm for CNF-SAT-CLAUSES yields an $O^\star(f(k))$ time algorithm for CNF-SAT-LENGTH.

  Therefore, CNF-SAT-LENGTH is also FPT. I think the best known algorithm for CNF-SAT-LENGTH runs in $O^\star(1.08^k)$ time [Wah05].[3]

  So CNF-SAT-VARS, CNF-SAT-CLAUSES, and CNF-SAT-LENGTH are FPT, but the bases of the runtime exponents are different. Once a problem becomes FPT, we still aren't satisfied. It may be in $f(k) \cdot n^c$ time, but we also want to know how small the function f can get. Do any of the above problems have an $(1+\varepsilon)^k \cdot n^c$ time algorithm, for every $\varepsilon > 0$? If so, that would refute ETH! (If this isn't immediate, think about it for a moment, and don't forget the sparsification lemma.)

- WEIGHTED-CNF-SAT $= \{(F,k) \mid$ CNF F has a satisfying assignment with **exactly** k variables set to true$\}$.
  The best known algorithm simply tries all $\binom{n}{k}$ assignments with exactly k true variables.

  Is WEIGHTED-CNF-SAT in FPT? This is a major open problem. It is believed the answer is no: if the problem was in FPT, then ETH would be false. (This is not easy to see; we'll prove it later.)

In summary, different parameterizations of satisfiability lead to different ways of thinking about this hard problem.

[3] Note there may well have been a better result published in the last 15 years, but Google scholar doesn't know about it...

2.3 Playing With the FPT Notion

Let's return to the original FPT definitions. Why did we define an FPT function as being of the form $f(k) \cdot n^c$? We wanted to capture the intuition that for every fixed k, the function is a polynomial that is $O(n^c)$. What if we defined FPT as slightly differently, like $f(k) + n^c$ instead? Turns out that doesn't matter, either!

Theorem 2.1 *$L \in \mathsf{FPT} \iff$ L can be solved in $O(n^c + f(k))$ time, for some computable f and constant c.*

> **Exercise:** Prove the theorem. *Hints: One direction is obvious. For the other direction, you could use the fact that $(x-y)^2 \geq 0$ to derive a useful inequality.*

Sometimes it's hard to even tell when your function is FPT! There could be a weird dependence on n and k which leads to the function being upper bounded by $f(k) \cdot n^c$. (This is actually useful sometimes...)

Proposition 2.1 *Let $s : \mathbb{N} \to \mathbb{N}$ be an unbounded (computable) function, and suppose there is a computable $g : \mathbb{N} \to \mathbb{N}$ satisfying $g(n) \leq n^{1/s(n)}$ for all n. Then for every computable h, $g(n)^{h(k)}$ is an FPT function.*

Here, the computable function of k is in *the exponent*, but because $g(n)$ is slow-growing, the overall function turns out to be FPT. (Recall we defined an FPT function to be any function of n and k that is *upper bounded* by a function of the form $f(k) \cdot n^c$.)

For example, if $s(n) = (\log n)^{1/2}$ and $g(n) \leq (\log n)^3$, then $g(n) \leq n^{1/s(n)}$, so we can conclude that the function $f(n,k) = (\log n)^{3h(k)}$ is FPT for every computable $h(k)$.

Proof. Define $s^{-1}(n)$ to output the largest integer m such that $s(m) = n$, and output 1 if such an m does not exist. (Note the latter case will only hold for finitely many m.)

Define $f(k) := g(s^{-1}(h(k)))^{h(k)}$. Note that f is computable.

> **Exercise:** Prove that for all $(n,k) \in \mathbb{N} \times \mathbb{N}$, we have $g(n)^{h(k)} \leq f(k) + n$. *Hint: Consider two cases, based on how n and k relate to each other.*

Therefore, $g(n)^{h(k)}$ is an FPT function. □

3 An Interesting FPT Algorithm

To conclude these introductory notes on FPT, we'll give an example of an interesting FPT algorithm for the k-Vertex Cover problem. Recall that

$$k\text{-Vertex Cover} = \{(G,k) \mid G \text{ has a vertex cover of size at most } k\}.$$

The obvious algorithm for determining if G has a vertex cover, is to try all possible $\binom{n}{k}$ options for the vertex cover, and check each one. This takes $\Omega(n^k)$ time for fixed k, so its running time is not FPT.

Theorem 3.1 *k-Vertex Cover is in $O^\star(2^k)$ time.*

Proof. We define a recursive algorithm with a working set S of vertices, which is initially empty. Our algorithm A will take a graph G and vertex subset S, and accept if and only if there is a vertex cover C in G such that $|C| \leq k$. (The working set S holds the "progress" on the vertex cover so far.)

Let $G = (V,E)$. For a vertex $u \in V$, we define $G - v$ to be the graph G with the vertex v removed (all edges to and from v are removed, along with v itself).

$A(G = (V, E), S)$:
If $|S| = k$ then { if G has no vertices, then **accept** else **reject** }. Pick any edge $\{u, v\} \in E$. **Accept** if and only if $A(G - u, S \cup \{u\})$ accepts or $A(G - v, S \cup \{v\})$ accepts.

To find a k-vertex cover in G, we run $A(G, \emptyset)$ and output its answer.

Exercise: Prove that $A(G, \emptyset)$ accepts when there is a k-vertex cover in G, and rejects when there is no k-vertex cover in G.

Let's consider the time complexity of $A(G, \emptyset)$. Let $T(n, \ell)$ be the running time of $A(G, S)$ where $\ell = k - |S|$. For $\ell = 0$, we have $|S| = k$ and

$$T(n, 0) \leq \text{poly}(n),$$

because we only have to just check if G has no vertices. Then for $\ell > 0$, we have $|S| < k$, and two recursive calls, where each call has one more vertex in the set S and one less vertex in the graph G. Therefore

$$T(n, \ell) \leq 2T(n - 1, \ell - 1) + \text{poly}(n).$$

This recurrence solves to $T(n, \ell) \leq 2^{\ell} \cdot \text{poly}(n)$. Therefore the running time of $A(G, \emptyset)$ is $O^{\star}(2^k)$. □

References

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[BCKN15] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Inf. Comput.*, 243:86–111, 2015.

[CXZ20] Huairui Chu, Mingyu Xiao, and Zhe Zhang. An improved upper bound for SAT. *CoRR*, abs/2007.03829, 2020.

[Din07] Irit Dinur. The PCP theorem by gap amplification. *J. ACM*, 54(3):12, 2007.

[LMS18] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. *ACM Trans. Algorithms*, 14(2), April 2018.

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[Wah05] Magnus Wahlström. An algorithm for the SAT problem for formulae of linear length. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2005.

[Wil17] Ryan Williams. Some ways of thinking algorithmically about impossibility. *ACM SIGLOG News*, 4(3):28–40, July 2017.