How to write a bachelor/master thesis

Andreas Wiese

When you write your bachelor or master thesis, probably you have not written many theses like that before in your life. In fact, writing a thesis is very different from attending lectures, going to TA-classes, and doing homework exercises as you did it in the courses that you have taken before. However, many students really like working on their theses and I am sure that you will learn a lot while you do it. In this document, I list some points that can help you while you do the research for your thesis and during the writing process.

If you have any suggestions what to add or change in this document, or you have any comments or feedback in general, I am happy to hear from you: `andreas.wiese@tum.de`.

- **Organize yourself well.** When you write your thesis, suddenly there is no fixed schedule due to the courses you take (lecture every Monday from 10am-12pm, TA-class every Tuesday from 2pm-4pm, hand in your assignment sheet every Wednesday by 4pm, etc.). Instead, you need to decide completely for yourself when you work on your thesis, for how long, what you want to have finished until when; it all depends on you. Therefore, it is important that you schedule well when you work on your thesis, what you do when, etc. My experience is that for some people this is quite hard, so please do not underestimate this aspect. On the flip side, you can find a schedule that works best for you, in which you work during times of the day when you are most productive and that fits well with your other activities.

- **A thesis is a marathon, not a sprint.** Your thesis takes about 3-6 months (depending on whether it is a bachelor or a master thesis and on the regulations of your university). This is a long time. Therefore, I recommend you to do your thesis in the way you would run a marathon, and not the way you would run a sprint: make sure that you work regularly and steadily on your thesis, but not too much at a time. For example, it is better to work for a few hours every day than working for 14 hours one day and then not at all for the rest of the week.

- **Motivation varies over time.** At the beginning most students are very motivated for their thesis. Typically, after some time the initial enthusiasm degrades, and there can be moments of frustration, in particular when you do not make progress for some time. This is normal. The thing with math and theoretical computer science is that it is well possible that think about a problem for a whole day and make no progress at all. This does not feel good; neither for you nor for very experienced researchers. When you do not feel motivated, it is important that you still keep going. In such a situation, it can help that you set yourself small achievable goals for the day, so that you feel that you accomplished something by the end of the day.

- **Start writing early.** Even if you have all results for your thesis ready, it will probably take you quite some time to write down everything in detail, polish your proofs, write an introduction, have your chapters proof-read by friends etc. *Do not underestimate how long all this takes, for most people it takes longer than they expect.* Also, probably this is the first or the second thesis that you write which makes it even harder for you to estimate how long things take. If you have the first version of some thesis chapters ready, it is probably a good idea to send them to your advisor so that she or he can give you feedback on your writing. In this way, you can use these suggestions directly for the other chapters in your thesis. Discuss with your advisor when and how much you should send to her or him.

- **Write down all ideas.** Ideally, write down every idea that you have, every partial result, every useful looking thought etc. on a notepad or on your computer. You will probably not include everything in your thesis that you write in this way. But the benefit is that it will allow you to, say, clear your head and it will force you to make precise what you have in mind (which is good!). And it is likely that in the process you discover aspects that you did not think of before, find potential problems with your approach early, etc. Try to write down your ideas as detailed as possible; in this way, you will discover more potential issues. Admittedly, this takes a bit of discipline, but there are a lot of benefits for you.

- **Meet with your advisor regularly.** In particular, meet with her/him also when you did not make progress since your last meeting, even though you tried very hard. Chances are that your advisor can give you new ideas and directions, or might even tell you that some obstacle seems to be too hard to overcome (in particular until your thesis deadline) and that you should do something else instead. Also, it helps you to keep going with your thesis.

- **If you get stuck** then there are several things that you can do.

  - **Simplify your problem.** For example, this could mean that you try to construct the smallest possible example or the simplest special case in which the problem arises that you cannot solve right now. Then focus on that smaller case and try to solve it. For example, if your want to prove some theorem about general polygons in the two-dimensional plane, you can start with the special case of convex polygons, or rectangles, or squares, or squares of unit size. Once you proved your theorem for unit squares, you can try to generalize your proof to arbitrary squares, then to rectangles, then to arbitrary convex polygons, etc. It is a bit of an art to construct the "right" special case that allows you to make progress. A good guideline is that you should define a setting in which one but not all of the difficulties of the general case arises. Another approach is to partition your problem into two smaller subproblems, e.g., one for convex polygons and one for non-convex polygons.

  - **Use another approach.** There are typically several different ways to approach a problem. If you get stuck with one of them, it makes sense to simply try a different one. For example, in order to compute a solution for a combinatorial problem, you can use a greedy algorithm, or a local search algorithm, or write down an LP-formulation and try to round it, or design a primal-dual algorithm for it, or formulate a dynamic program that solves it. If you tried one of these approaches for a long time and it did not work out, then it simply does not make sense anymore to keep on trying. Instead, you invest your time better if you try something else. However, this is a good topic to discuss with your advisor.

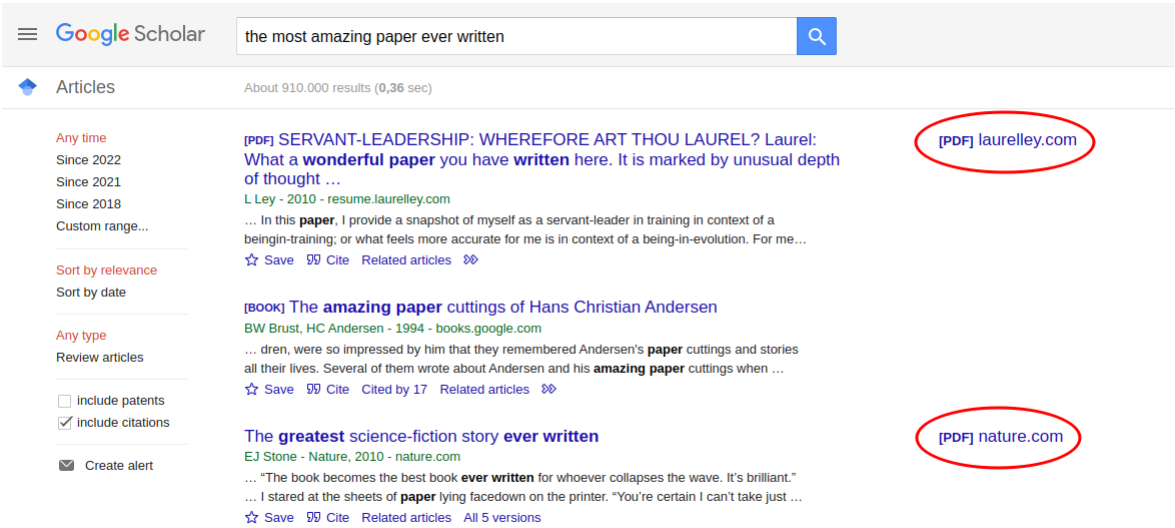
  - **Talk to your advisor.** Chances are that she or he can give you suggestons on how to overcome an obstacle, give you useful advice on whether you should try an alternative approach, or suggest a good special case to look at. Of course, you should try for some time for yourself to make progress, after all it is your thesis. But your advisor will be happy to help you if you get stuck.

- **Use LaTeX efficiently.** A thesis in mathematics or computer science is written in LaTeX. As a matter of fact, when you learn LaTeX there is quite a bit of a learning curve. So if you do not have much experience with LaTeX, it is useful if you start learning it soon, before you even start with your thesis. There are several programs that make it more comfortable to write LaTeX documents, like Kile, TeXnicCenter, or LyX. My experience is that everybody has her or his favorite programs for writing LaTeX, so you need to try and figure out what works best for you. I personally use LyX since there you see directly how your document will look like, including formulae and tables, and you do not need to worry much about LaTeX commands. For drawing figures I use IPE which has all the features you need to draw mathematical figures and it allows you to include LaTeX formulae directly in your figures (just type, e.g., α in a text box). However, figure out what works best for you, and make sure that you know LaTeX before you start with your thesis. Also, you can check whether your university offers a template for theses, following its thesis regulations (like the template of the TU of Munich).

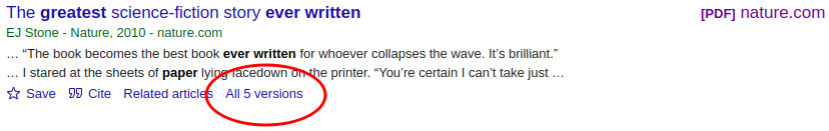
Finally, if you have written a large part of text and believe that you will not need it at the end, instead of deleting it it might be better to comment it out or save it in some other file, just in case.

- **Writing mathematics.** It is important that in your thesis all proofs are mathematically precise and correct. It might take some time for you to polish your proofs until everything is spelled out precisely, every special case is taken care of, and every inaccuracy is fixed. In particular, it might take you a long time compared to the amount of text that you produce. This is normal when writing theses or papers. It can easily take you a whole morning to write a proof that is only half a page long at the end. It is important though that you invest this time: only when you write down a formal proof of your claims, you can be absolutely certain that what you believe is really correct. In many cases something seems "intuitively obviously true" but when you write down the formal proof you realize that you missed something. This happens also to very experienced researchers. On the other hand, please also give a lot of intuition to the reader. It is hard to read a mathematical text in which the formalism is precise and correct, but in which no intuition is given. For example, it is good to say something about the general structure of a section, a proof, or an algorithm, before you go into details. You may think of your thesis as a story that you tell to somebody. What structure would be good? What should come first and what should come only later?

- **Web search.** When you search for papers on the web, Google Scholar and DBLP are great tools. In particular, from them you can download bibtex-entries of papers so that you can cite them easily with bibtex. Also, when you do the research for your thesis topic, it can be very useful to read papers in which other people did something similar before. When you search, you might find the website of a journal etc. in which the paper was published, but you cannot download the paper without paying for it. If this happens, often you can still download the paper when you are in your university (being connected to the internet via the university WLAN) or when you connect to your university via a VPN. The reason is that universities pay for subscriptions to journals and in this way get access to such papers. If there is a freely available version of a paper, Google Scholar tries to give you a link to the PDF as you can see here (circled in red):



However, sometimes Google Scholar is wrong here and gives you a link like the red circled ones, but this link does not send you to a free version of the paper you are looking for. Then it can help to ask Google Scholar for other versions of that paper



and if you are lucky one of them will really be freely available.

- **Approximation algorithms.** If you are supervised by me, chances are that the topic of your thesis is to design an approximation algorithm for some problem. For this, there is no general method or technique that always works, but in Appendix A there are some strategies that might help you.
- **Useful resources**
  - "Writing Mathematical Papers in English: A practical guide" by Jerzy Trzeciak and "Das ist o.B.d.A. trivial!" by Albrecht Beutelspacher (in German) which contain many helpful suggestions for writing mathematics well.
  - "The Grammar According to West" by Douglas B. West which contains many grammatical suggestions when writing mathematics.
  - "Writing a Bachelor Thesis in Computer Science" by Siegfried Nijssen which many suggestions on how to structure a thesis in computer science (and IMO most of it also applies to a thesis in mathematics)

I hope that you will write a great thesis, and that you enjoy doing the research for it and finally writing it. All the best!

Acknowledgements.

I would like to thank Holger Dell for pointing out his website on the project process of a thesis with many pointers to helpful resources, and Lars Rohwedder for pointing out the book "Writing Mathematical Papers in English: A practical guide". Also, I would like to thank Paul Deuker, Alexander Keil, Anja Kirschbaum, Alexandra Lassota, and Stefan Weltge for useful comments for this document.

A How to design an approximation algorithm

There is no general method for designing an approximation algorithm for a problem, i.e., a method that one could simply follow and that always works (like Gaussian elemination for solving systems of linear equations). Instead, every problem is different and one needs to design an approximation algorithm specifically for it. However, there are some general approaches and techniques that are often useful. Below I list some strategies that you can use if you want to design an approximation algorithm for a given problem.

- **NP-hardness.** Before designing an approximation algorithm, it is useful to check whether your problem is actually NP-hard. For many problems this is already known. For a new problem however, this might not be known. When you try to prove that a problem is NP-hard, you can use the following approaches:
  - If the problem contains numbers in the input, like the size of an item in KNAPSACK or the processing time of a job in a scheduling problem, then in many cases you can reduce PARTITION or 3-PARTITION to the problem, and in this way prove (weak or strong) NP-hardness.
  - If the problem is about selecting a subset of some given objects according to some combinatorial constraints, sometimes it is useful to reduce INDEPENDENT SET to the problem, so that the combinatorial constraints model the input graph of INDEPENDENT SET. Note that INDEPENDENT SET is already NP-hard on planar cubic graphs, i.e., on graphs in which all vertices have a degree of three [2, problem GT20 in Appendix A1.2], which might simplify the reduction.
  - Otherwise, I would recommend you to try reducing 3-SAT to your problem by building some suitable gadgets for the variables and the clauses of a given formula. There is a special case of 3-SAT called 3-BOUNDED 3-SAT which is still NP-hard [1]. In this special case, each variable appears in at most three clauses, so you can assume w.l.o.g. that it appears either twice as a positive and once as a negative literal, or twice as a negative literal and once as a positive literal.
  - In principle, you could try to construct a reduction from any NP-hard problem; however, in my experience the problems above are the ones I would try first.
- **Greedy algorithms.** It is a good idea to try first whether some simple algorithmic approaches are already good enough to get a (decent) approximation ratio for your problem. Greedy algorithms are among the simplest type of algorithms. For many problems, they are not good enough to get a reasonable approximation ratio, but it makes sense to rule them out first before you try more complicated techniques. In particular, when you find counterexamples for a greedy algorithm, then you often find "difficult" instances on which you can "test" other algorithmic approaches that you try (and maybe rule out quickly that they are useful).
- **Small and large objects.** There are many problems in which you can partition the input items easily into small and large objects. For example, in the KNAPSACK problem you can define that an item i is *small* if its size a_i satisfies $a_i < \epsilon B$ where B is the capacity of the knapsack and ϵ is a fixed (small) positive constant, and you define that i is *large* otherwise. Then you can look at the small and large items separately. For small objects, greedy or LP-based methods often work well, e.g., for KNAPSACK you can get a $(1+\epsilon)$-approximation with both approaches. For large objects, dynamic programming or complete enumeration often works well, e.g., for Knapsack you can find the optimal solution consisting of large items only by complete enumeration in time $n^{O(1/\epsilon)}$. Often, you can show that you can round the sizes of the large items (e.g., to powers of $1+\epsilon$), while losing only a factor of $1+\epsilon$ in the approximation guarantee. For KNAPSACK this does not work, but for BIN-PACKING it does. Finally, you can take the best solution among the small and large items (while losing only a factor of 2 in the approximation guarantee) or try to combine your approaches for small and large items to a joint algorithm.
- **Polynomial bounded input numbers.** If your problem contains numbers, e.g., the size of an item in a BIN-PACKING instance, try first to find an approximation algorithm assuming that the input

numbers are bounded by a polynomial in the input, e.g., that they are in the range $\{1, 2, ..., n^k\}$ for some constant k. Often, if you find a good algorithm in this setting, you can sacrifice a factor of $1 + \epsilon$ in the approximation ratio in order to generalize your algorithm to the general case. Typically, this last step makes the algorithm more technical, though the key ideas are still the same as in the case of bounded input data. On the other hand, if the input data are polynomially bounded it is often easier to design a good algorithm.

- **LP-approaches.** A standard approach in approximation algorithms is to formulate your problem as a linear program (LP), solve it in polynomial time (e.g., via the Ellipsoid method) and then try to round it. Typically, it is easy to formulate a problem as an LP. However, rounding a given (optimal) fractional solution can be difficult. A first good step is to look for instances in which the integrality gap is large. If there is such an instance, then you can rule out quickly that this LP helps you (but maybe a stronger LP does, e.g., if you add further constraints to your LP or use configuration-LP based approaches). Often, LPs are bad for problems that have many symmetries (e.g., for scheduling a given set of jobs on identical machines to minimize the makespan) since then the LP can simply "smear" all input objects across the instance and you do not get any non-trivial bound from the LP. If you have the impression that your LP is good (i.e., has a good integrality gap) then then you can try the following techniques.
  - **Rounding up** all fractional variables x_i that are higher than some threshold, i.e., you set $\bar{x}_i = 1$ for your rounded solution $\bar{x}$ if and only if $x_i \geq 1/10$. For some simple problems this is already good enough, e.g., for Vertex Cover. Typically, this simple approach is not good enough, but it is useful to rule it out first.
  - **Randomized rounding** where the probability of rounding up a variable x_i to 1 is simply the value of x_i in the optimal solution. Together with the Chernoff bounds, this often gives a simple $O(\log n)$-approximation which is already something. But there are more sophisticated randomized rounding schemes that give better approximation guarantees. For example, *contention resolution schemes* (sometimes also called *randomized rounding with alteration)* is also a technique that works in many settings, see e.g., [3] (as shown in this paper, the technique works even for maximizing submodular functions which is a generalization of maximizing linear functions).
  - **Primal-dual algorithms** are another idea you can try for a given LP. Often you can design a primal-dual algorithm in a straight-forward way as follows:
    - start with the (infeasible) primal solution $x = 0$ (so each primal variable equals to 0) and the (feasible) dual solution $y = 0$
    - raise one or all dual variables until some dual constraint becomes tight
    - take the primal variable x_i that corresponds to the tight dual constraint, i.e., set $x_i := 1$
    - repeat this process until the solution x is feasible
    - do backward deletion, i.e., go through the x_i variables in the *reverse* order in which you set them to 1, and for each such variable x_i check whether you can set it to 0 without making your solution infeasible

    Suppose that each primal constraint is of the form $a_i^T x \geq b_i$. Try to show the following statement: "There is a value $\alpha \geq 1$ such for each dual variable y_i with $y_i > 0$, for the corresponding primal constraint $a_i^T x^* \geq b_i$ it holds that $a_i^T x^* \leq \alpha \cdot b_i$." Then you get an α-approximation algorithm.
- **Guessing the value of OPT.** For almost every problem, you can use a *binary search framework* (sometimes also called *bisection search*) that in each iteration gives you an estimate T for OPT. Assuming that your problem is a minimization problem, then you need only an algorithm that either asserts that $\mathsf{OPT} > T$ or that finds a solution with a value of at most $\alpha \cdot T$, where α is your desired approximation algorithm. Often, it is easier to design an algorithm this type than a "complete" approximation algorithm for your given problem. The binary search does not cost you anything in the approximation guarantee, and it increases the running time typically only by a logarithmic factor.

Researchers often say that this "guesses the value of OPT". Here is why: when you design and analyze your approximation algorithm, do this first while assuming that $T = \mathsf{OPT}$. Typically, if $T \neq \mathsf{OPT}$ your algorithm either gives you a solution of value at most $\alpha \cdot T$ or finds out that $T < \mathsf{OPT}$.

- **Counterexamples** can be very useful for you. If you find an example showing that some approach or technique does not work, then you can save a lot of time (which you can then invest in another approach that actually works)! Thus, if you get stuck in proving that some algorithm or algorithmic approach gives you a good approximation ratio, then try proving the opposite. In particular, if you already tried proving that some candidate algorithm is good, then you probably have some intuition for how bad instances need to look like (e.g., because for some special cases you already proved that you algorithm is good). On the other hand, if you fail at finding a counterexample, this might give you a clue for how you could prove that you algorithm actually *is* good.

References

[1] T. Ebenlendr, M. Krcal, and J. Sgall. Graph balancing: a special case of scheduling unrelated parallel machines. In Shang-Hua Teng, editor, *SODA*, pages 483–490. SIAM, 2008.

[2] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness.* Freeman NY, 1979.

[3] Jan Vondrák, Chandra Chekuri, and Rico Zenklusen. Submodular function maximization via the multilinear relaxation and contention resolution schemes. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 783–792, 2011.