

What this course teaches

Course aim at the beginning of the course:

- To learn some methodologies that are widely used in designing algorithms.
- To learn some metrics in which algorithms are designed for.
- To understand that some problems are inherently difficult, and what to do to deal with them.

My own target: after taking this course, you should have better access to the literature of theoretical computer science.

That is, more papers of the proceedings of top theory conferences like SODA or FOCS should make sense to you now.

Exam format

- CSIS6924 (Algorithms): 2 hr exam, 4 questions, answer all. One is about the upcoming presentation, and should require no more than 100 words to answer adequately (but can be much fewer).
- CSIS0324 (TITCS): 3 hr exam, 5 questions, answer all.

Apart from the question about presentation, all questions will require you proving something. Difficulty is somewhat less than quiz.

ALG/TITCS L11-1

ALG/TITCS L11-2

Topic 1: greedy algorithm

- We considered optimization problems that can be solved by using a greedy algorithm.
- “Greedy” is a design strategy of algorithms.
- Different people have different idea about what is greedy. If every time the algorithm do a step that seems to be the best in some way, then it can be said to be a greedy algorithm.
- On the other hand, “optimal” is an objective measurement. It means “this is the best that we can do, and there is no way to do better”.
- Therefore, optimality is a very strict requirement. It does not mean to “be-have good”. It means to be best.

General advice

- We are in the business of analysing algorithms. We are to make judgements of algorithms about its property. We are to make broad statements about algorithms. Don't have any prejudice about the algorithm before proving the algorithm.
- At all time, be very clear about what you are going to do, and what you're going to get if you can do it, before you actually do something.
- For example, if your are going to show an algorithm to be optimal, don't think like “is it better than this algorithm”: it gives you nothing. Think “why it is the best”, i.e., why there is no way to do better.

ALG/TITCS L11-3

ALG/TITCS L11-4

How to show a greedy algorithm is optimal

During the course, we show algorithms to be optimal by one of three ways:

1. Show that if there is a step that increment the cost, then every algorithm really need the new, increased cost.
2. Show that the problem exhibits the greedy-choice property and the optimal substructure property. This means that the first step never makes optimality impossible, and after the first step the problem is reduced to a smaller problem.
3. Show that the problem is in the form of “maximum independent subset problem in matroid”.

ALG/TITCS L11-5

Choosing a method

- It is nearly always possible to use the basic principle, i.e., method 2 above.
- But to apply method 2, you must be able to reduce the problem to itself after a step. It is sometimes necessary to extend the problem a bit to apply the method.
- If method 1 can be applied, it usually gives a very simple way to argue for optimality. But not many problem can be argued this way.
- Method 3 has a very characteristic property: all maximal solutions will have the same size, since all maximal independent sets in a matroid are of the same size. This usually show directly whether such an argument is possible.

ALG/TITCS L11-6

Topic 2: NP completeness

- Informally, we can associate the “difficulty” of a problem by the question “what is the efficiency of the best algorithm that solve the problem”.
- For some problems, it is known that polynomial time algorithm exists. We say that these problems are “easy”. For example: Minimum Spanning Tree.
- For some problems, it is known that polynomial time algorithm does not exist. We say that these problems are “very difficult”. For example: knowing whether a winning strategy exists for chess.
- Other problems: NO VERDICT. Some such problems are factoring a large integer, check whether two graphs are isomorphic, finding a best TSP, etc.

ALG/TITCS L11-7

The class NP

- A significant number of problems are in the last category. Some of them can be grouped together by using non-determinism.
- We say a **decision problem** is in NP if it can be solved in polynomial time using non-determinism.
- Non-determinism is a very asymmetric way to compute: the algorithm can use guess. If answer of the problem is “yes” there must be a way to guess so that the algorithm answer yes.
- On the other hand, if the answer of the problem is “no”, **there must be no way** for the algorithm to answer “yes”.
- We can also use the notion of polynomial time verification to define the complexity class NP: there must be a certificate for each “yes” instance that can convince a polynomial time algorithm that it is really one.

ALG/TITCS L11-8

NP completeness

- Being in NP does not make a problem difficult. It actually means the problem cannot be too difficult.
- On the other hand, it is possible to say that a problem is “among the most difficult problems in NP”. Such problems are said to be NP-complete.
- We do this by reducing all problems in NP to the problem. That means, if we can solve the problem “under any reasonable model”, then we can also solve all problems in NP “under the same model”.
- We actually know one NP complete problem this way: the circuit satisfiability problem.

ALG/TITCS L11-9

Proving new NP complete problem

- Once we have an NP complete problem, it is much less tedious to prove another algorithm to be NP complete.
- Instead of reducing all problems in NP, we just need to reduce one NP-complete problem. That every problem in NP can be reduced to the new problem is automatic, by the transitive property of reductions.
- The key idea here is **reduction**. A valid reduction must:
 - Be polynomial time. Given an instance of the old problem, we must be able to compute an instance of the new problem.
 - Gives an equivalent problem. If the answer to the old problem is “yes”, then the answer to the new problem must be “yes”. **And vice-versa**.

ALG/TITCS L11-10

Common pitfall

- It is mentioned many times, but still people get it wrong: we have to reduce known NP hard problem to the algorithm, not the other way round.
- If we are going to solve a new problem, of course we will try to reduce the new problem to a known problem with known solution. Then we can use the solution of the known problem to solve the new problem.
- But we are doing the reverse: we are showing that a new problem is very difficult to solve. So of course we will do the reverse: to reduce a known hard problem to the new problem!

ALG/TITCS L11-11

Methods for reduction

- If we know a related problem is NP hard, proving NP hardness of a new problem can sometimes be done by reducing this related problem.
- Otherwise, we usually end up having to have direct reduction from 3CNF-SAT.
- To reduce 3CNF-SAT, we must have some way to do a number of things:
 1. Represent a variable in 3CNF-SAT using the new problem.
 2. Represent a clause in 3CNF-SAT using the new problem.
 3. Ensure that each clause have at least one true term.
 4. Ensure that the variables take consistent values.

ALG/TITCS L11-12

Topic 3: approximation algorithms

- If a decision problem is NP-complete, its associated optimization problem is NP-hard.
- To deal with these problems, we have two choice: to use exponential time, or to use an approximation algorithm.
- To use an approximation algorithm means we modify the problem. Instead of asking for the optimal solution, we ask for a "good enough" solution.
- Here, "good enough" means it can provide some performance guarantee. We usually use a ratio bound: if the optimal solution costs c , our algorithm should costs no more than ρc for some fixed ρ .

ALG/TITCS L11-13

Difficulty of approximation

- NP complete problems are not all equal.
- Some are very easy to approximate, leading to polynomial time approximation schemes. That is, given any number ρ larger than 1, we can find an algorithm with ratio bound ρ . (e.g., subset sum)
- Some are less easy to approximate, but is still approximable to constant ratio bound. (e.g., vertex cover, TSP-with-triangle-inequality)
- Some are hard to approximate, and can only be approximated to a ratio bound that increases with problem size. (e.g., set cover)
- Some are essentially impossible to approximate. (e.g., TSP)

ALG/TITCS L11-14

Showing ratio bounds

- Proofs of ratio bounds are very problem specific.
- Usually, what we need is that a significant portion of the cost of the algorithm translates directly to the cost of the optimal solution.
- This way the cost of the solution given by our algorithm can be related to the solution given by the optimal algorithm. (Vertex cover, set cover.)
- At some times, a related polynomial time algorithm can give rise to a good approximation algorithm (TSP).
- On the other hand, most PTAS are modifications of exponential-time algorithms for the original problems, trying to bound the amount of time needed to polynomial by trimming the input.

ALG/TITCS L11-15

Other related topics

- We don't have chance to study a lot of interesting topics of approximation algorithms.
- One example is some very general approach to use linear programming and semi-definite programming to approximate NP-hard problems.
- Another is to make use of randomization. At many times, if we can tolerate expected behaviour instead of worst case behaviour, we can find a good approximation algorithm easier.

ALG/TITCS L11-16

Topic 4: Online algorithms

- Some algorithm have to make decisions based on incomplete information. In particular, on-line algorithm need to deal with the fact that it doesn't have future information.
- Under such circumstances, usually the algorithm cannot perform optimally.
- Just like the case for NP complete problems, we then aim for an algorithm that guarantees a ratio bound: competitive algorithms.
- And just like the case for approximation algorithm, we want to translate a significant portion of the cost of an on-line algorithm to the cost of the best off-line algorithm.

ALG/TITCS L11-17

Design principles

We have seen two design principles:

- Ski principle: when we have made the cheap choice for enough times, it is okay to make an expensive choice (ski rental, unbounded search).
- Force off-line cost to increase: if we cannot guarantee that we can perform good, it is equally good to make sure that the off-line algorithm performs bad (cache replacement, list access).

These are design principle—not proving technique! To prove your algorithm, you need to bound ratio of the costs between the on-line algorithm and the off-line algorithm.

ALG/TITCS L11-18

Lower bound techniques

- In order to show lower bounds for on-line problems, we treat the running of the algorithm as the game between two players.
- One of the player is an on-line algorithm. The other is called an adversary.
- To show a lower bound, we design an adversary. The behaviour of the adversary can depend on the behaviour of the on-line algorithm.
- The adversary will try to guarantee that the input it generates can be easily served by an off-line algorithm, while on the other hand makes sure that the on-line algorithm will behave poorly.

ALG/TITCS L11-19

Additional proving techniques

- It is usually the case that the on-line cost of each step cannot be directly related to the off-line cost of the corresponding step, even if on the whole an algorithm is competitive.
- One way to deal with this problem is to use the potential method.
- Intuitively, the potential of a configuration shows how "bad" is the configuration of the on-line algorithm. Usually this means how different is it from the optimal off-line solution.
- Then we can try to associate the on-line cost and off-line cost to the total potential change, in order to argue about the relationship between the on-line and the off-line cost.

ALG/TITCS L11-20