

Quiz

A quiz reflects several things to me:

- How much do you understand lecture materials?
A: At around 60%.
- How clever you are?
A: Reasonable, but in general not very clever.
- Do you have the techniques that lead you to success?
A: Okay generally, but not as good as I expect.

Stats:

N	65
Median	55
Mean	53.69
SD	20.18

ALG/TITCS L07-1

Five levels of understanding

When I read an answer, I look for signs showing that the student:

- Don't understand the question at all.
- Understands the question, but don't know anything to answer it.
- Knows what is needed to answer it, but don't know how to get it.
- Knows what is needed to answer it, but only have part of the ideas to get it.
- Started putting what is needed to answer it, but gets stuck in the middle.
- Is capable to solve the problem (need not have a perfect solution).

If you are on bullet three above, act like that. **Don't pretend that you are at bullet 6!** To one who know everything, one who pretend is on bullet 1/2.

ALG/TITCS L07-2

Example: paging

- Answer 1: Comparing the algorithm with Least-Recently-Used, we can make an example that this algorithm is better. So it is optimal. (Bullet 1.)
- Answer 2: Because the algorithm always chooses the page that will not be requested again for longest, it is clearly optimal. (Bullet 2.)
- Answer 3: Because the algorithm always chooses the page that will not be requested again for longest, it is will minimize the frequency of loading, and is thus optimal. (Bullet 2.)
- Answer 4: We need greedy choice and optimal substructure to solve the problem. Greedy choice means that we always have an optimal schedule of paging replacing the page not requested for longest. ... (something wrong.) (Bullet 3.)

ALG/TITCS L07-3

Example: paging (cont'd)

- Answer 5: We need greedy choice and optimal substructure. For greedy choice, assume we have an optimal schedule that does not replace the page that will not be accessed longest. We need to build another optimal solution that does do that. We can make this by choosing to use that page. It cannot use more page because ... (something with bug). With that we show optimal substructure. ... (Bullet 4.)
- Answer 6: We need greedy choice and optimal substructure. For greedy choice, assume we have an optimal schedule that does not replace the page that will not be accessed longest. We can build another optimal solution that does do that, by modifying the optimal schedule so that it chooses to use that page. It cannot use more page because ... (something with bug).
N.B.: There seems to be a problem: how can I argue that future requests will not cause our algorithm to perform worse if the memory is different?
Let's continue with showing optimal substructure. ... (Bullet 5.)

ALG/TITCS L07-4

Key to success in exams

- Show you know what is the question.
- Show you know what is required to answer the question.
- If you know there is a bug in your answer, show that you know it.
- If you seems having a clue, tell it. Explain why it seems to be a clue.
- If you have a conjecture, write down that conjecture, and clearly tells that it is just a conjecture, not a proof (otherwise you drop back to bullet 1/2).
- If you get stuck in the middle of the first part but believe that what you want to prove is right, continue to prove other parts of the answer.
- Answer a question that you know first!

ALG/TITCS L07-5

New topic: approximation algorithm

- In the last few week, we learnt basically that some problems are so hard that probably they cannot be done in polynomial time.
- CIRCUIT-SAT, SAT, 3CNF-SAT, SET-COVER, VERTEX-COVER, 3COLORING, HAMILTONIAN-CYCLE, 1-in-3CNF-SAT and 3DM are all among them.
- Since these problems are NP -complete, their associated optimization problems are NP -hard. (N.B.: Only decision problems can be NP -complete.)
- Many NP -hard optimization problems are very important problems, that even if they are NP -complete, we want a way to solve it.
- If input is small, we might be able to find algorithms that grow "slow enough".
- If the input is large, we must resort to suboptimal solutions.

ALG/TITCS L07-6

Approximation algorithms

- Even if we can accept suboptimal algorithms, we don't usually want to get solutions that are arbitrarily bad.
- *Approximation algorithms* are those algorithms that run in polynomial time, and always give you a solution that is not very bad.
- How good is the algorithm doing? We judge it by a *ratio bound*.
- Consider an instance I of the problem. Denote the cost of an optimal solution as $C^*(I)$, and the cost of the solution given by the algorithm as $C(I)$.
- An algorithm is said to have ratio bound (approximation factor) $\rho(n)$ if for any instance I of size n ,

$$\frac{1}{\rho(n)} \leq \frac{C(I)}{C^*(I)} \leq \rho(n)$$

ALG/TITCS L07-7

Different types of ratio bounds

- For some problems, and the ratio bound must grow with n . I.e., when the problem is larger, we need to spend more time to deal with it, and dealing with it resulting in a solution with worse guarantee.
- For other problems, it is possible to find algorithms that gives constant ratio bounds, i.e., does not depend on n .
- Some of these problems have *polynomial-time approximation schemes* (PTAS). That is, given any positive real number ϵ , we can find a polynomial-time algorithm with ratio bound $1 + \epsilon$. Here, ϵ is consider fixed, so the algorithm might well be exponential in $1/\epsilon$ (or worse).
- Some of these problems have very good approximation scheme, that the ratio bound is polynomial in $1/\epsilon$ as well. We call such approximation schemes "fully-polynomial-time approximation schemes" (FPTAS).

ALG/TITCS L07-8

Vertex cover

We know that the decision problem VERTEX-COVER is NP -complete. How about the optimization variant Vertex-cover?

Given a graph $G = (V, E)$, find the smallest subset C that is a vertex cover, i.e., every edge is adjacent to at least one vertex of C .

If we had an algorithm A to solve this problem in polynomial time, we can solve VERTEX-COVER in polynomial time: call A and find C . If C contains no more element than the bound k , say yes. Otherwise say no.

Therefore, VERTEX-COVER reduces to it, and thus Vertex-cover is NP -hard.

If the graph is a small one, we can try all the possibilities. It requires $n!$ time if the graph has n vertices.

ALG/TITCS L07-9

Approximating vertex cover

- If the graph is large, we cannot do things like this.
- However, vertex cover is quite easy to approximate. Note that an edge only has two vertex which can cover it, so it is not too horrible to choose both.
- Once we allow ourselves to choose both, an algorithm which chooses only one cannot be better than us.
- So intuitively, we know we can have an algorithm with ratio-factor 2.

ALG/TITCS L07-10

Algorithm APPROX-VERTEX-COVER

The following algorithm APPROX-VERTEX-COVER approximates Vertex-cover with approximation factor 2.

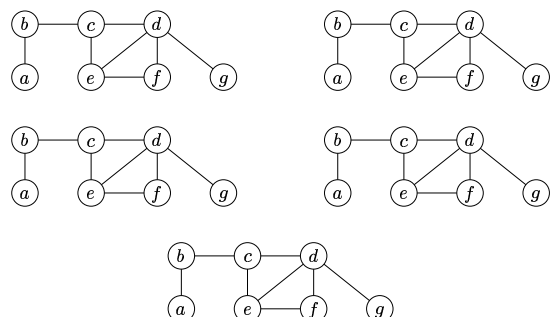
```

C ← ∅
E' ← E
while E' ≠ ∅
do let (u, v) be any edge in E'
   C ← C ∪ {u, v}
   Remove all edges adjacent to u or v from E'
return C
    
```

That is, every time we find an edge (u, v) not yet covered by C , and add both u and v to C . Repeat this until all edges are covered.

ALG/TITCS L07-11

Example



ALG/TITCS L07-12

Showing approximation ratio

- To show a ratio bound of any approximation algorithm, it is important to have a good bound of an optimal solution.
- We do not really know what is an optimal solution (otherwise we don't need to approximate it!).
- What we do is to compare the solution produced by the approximation algorithm and a hypothetical optimal solution, and see what the optimal solution must do.
- If we can bound the cost of each step of our algorithm by the cost in an independent part of the optimal solution, then we get a bound.

ALG/TITCS L07-13

APPROX-VERTEX-COVER has ratio bound of 2

- In each iteration of the loop, we put two new vertices u and v into C , and hence increase the cost of the solution by 2.
- The optimal algorithm must choose at least one of u or v : if it chooses neither, it cannot cover the edge (u, v) .
- Therefore, if our algorithm iterates m times, then our algorithm returns a set of size $2m$, while the optimal algorithm must at least choose m elements.
- Suppose an optimal solution is C^* . Then we have

$$1 \leq \frac{|C|}{|C^*|} \leq \frac{2m}{m} = 2$$

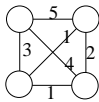
Hence the algorithm has a ratio bound of 2.

ALG/TITCS L07-14

The Travelling salesman problem (TSP)

The travelling salesman problem is the following problem:

Given a complete graph $G(V, E)$ and a cost function c from E to non-negative integers, find a complete tour which minimize the total cost.



This problem is NP-hard because the decision problem HAMILTONIAN-CYCLE reduces to it...

ALG/TITCS L07-15

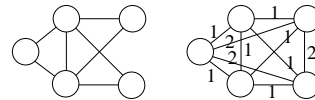
TSP is NP-complete

Suppose we have an algorithm A that solves TSP.

We can solve HAMILTONIAN-CYCLE as follows:

Given a graph $G = (V, E)$, we build a complete graph G' with the same vertices, so that the edges has cost 1 if it is in G , and has cost 2 if it is not.

Then we call A on G' . If the returned cycle has weight $|V|$, then G has a Hamiltonian cycle, so we accept it. Otherwise, we reject it.



ALG/TITCS L07-16

TSP is very hard to approximate

In fact, the same reduction can be used to show that unless $P = NP$, there is no algorithm that approximates TSP with any ratio bound:

- Suppose we have a polynomial-time algorithm A that approximates TSP within any ratio bound ρ (whether or not ρ grows with the size of graph).
- Then we can solve HAMILTONIAN-CYCLE as follows: given a graph $G = (V, E)$, build a complete graph G' so that the edges has cost 1 if it is in G , and has cost $\rho|V| + 1$ otherwise.
- Now call A with G' . If there is a Hamiltonian cycle, the optimal cycle is of cost $|V|$, so the approximation algorithm returns a cycle of cost $\rho|V|$.
- If there is no Hamiltonian cycle, the approximation algorithm must choose a "costly edge", which alone costs $\rho|V| + 1$. So we have a polynomial time algorithm for HAMILTONIAN-CYCLE, i.e., $P = NP$.

ALG/TITCS L07-17

TSP with restrictions

- While TSP itself is impossible to approximate, there may be specific cases that can be approximated.
- Let's consider the case for which the cost function follows the triangle inequality. That is, $c((u, v)) + c((v, w)) \geq c((u, w))$ for all u, v and w .
- The problem is still NP-hard—An exercise for you (37.2-1, page 973)!
- However, this time the problem has an approximation algorithm that approximates the optimal solution to a ratio bound of 2.

ALG/TITCS L07-18

Example

Algorithm APPROX-TSP-TOUR

The idea is: we can find a MST very easily. Due to the triangle inequality, we can use the MST to build a TSP that is not too costly.

Arbitrarily choose a vertex v as the root.

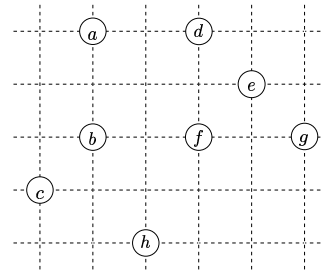
Find an MST T of G .

$H \leftarrow \emptyset$

Do a depth-first search on T . Whenever we visit a node v for the first time, we add v at the end of H .

Return H .

To avoid writing a real lot of cost values, we use the Euclidean distance as the cost function. . .



ALG/TITCS L07-19

ALG/TITCS L07-20

APPROX-TSP-TOUR has ratio bound of 2

- Any tour has no less cost than the MST T , since it is itself a tree plus one edge.
- Thus the cost of the optimal tour H^* is no smaller than $c(T)$.
- If we visit the DFS tree and follow the links there, we end up traversing each edge twice, so the cost is exactly $2c(T)$.
- Due to the triangle inequality, $c(H)$ is no more than the above value.
- As a result,

$$1 \leq \frac{c(H)}{c(H^*)} \leq \frac{2c(T)}{c(T)} = 2$$

- One more exercise: 37.1-4 (page 968).

ALG/TITCS L07-21