CS420+500: Advanced Algorithm Design and Analysis

Lectures: Mar. 22 + Mar. 24, 2017

Prof. Will Evans *Scribe: Vincent Hui*

Announcements (posted on webpage):

- The final review will be on Friday, Apr. 7th in DMP 110, hopefully.
- Please complete teaching evaluations.

As a first aside, we note the Euclidean TSP problem, with edge weights obeying the triangle inequality, is in fact NP-Hard. This was not explicitly stated last time. The result was first shown by Papadimitriou.

In this lecture, we:

- discussed hardness of approximation, and (in particular) hardness of approximation of the Travelling Salesman Problem
- defined online algorithms
- introduced competitive analysis as a technique for comparing runtime of online algorithms

1 Hardness of Approximation

We motivate our discussion by considering the generalization of Travelling Salesman problem. We will see that while Euclidean TSP was NP-hard but easy to approximate, the general problem is NP-hard to even approximate. We will use the NP-hardness of the Hamiltonian Cycle problem to show that obtaining a c-approximation of the general TSP problem is NP-hard.

Note that hardness of approximation typically implies the original problem belongs in an even harder complexity class, assuming the complexity hierarchy is true.

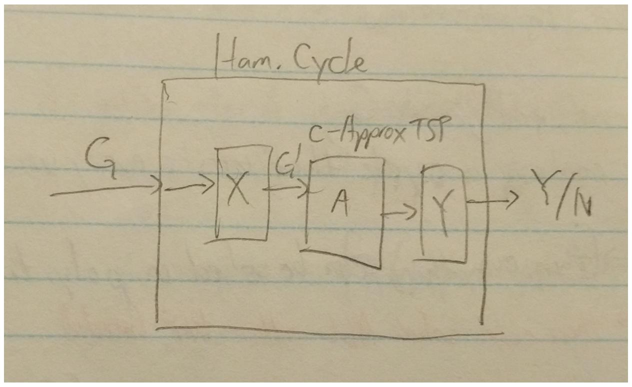
Definition 1 (Hamiltonian Cycle)**.** Given unweighted graph G, does G contain a cycle that visits every vertex once?

Claim 2 (NP-hardness of appproximating TSP)**.** *If $P \neq NP$, then there is no polynomial time c-approximation algorithm for TSP.*

Remark. *To show this general result, observe we will need show that any c-approximation algorithm, of which we know **NOTHING** about, is NP-hard.*

The natural idea will be to reduce the Hamiltonian-cycle (NP-hard) problem to the approximation. Note our task is trivial if the target problem was an exact TSP solver. Since it is not, we seek to transform input graph G in such a way to make it "horrible" if and only if there is no H-cycle - so that we can detect from even the *approximation* whether an H-cycle exists in the original graph. Consider a scheme to modify G so that using any added edge $e \in E(G')$ in the Hamiltonian cycle would blow up the cost of such cycle. (can you do this?)

Proof. Suppose A is a polynomial time c-approximation algorithm for TSP. We use A to solve Hamiltonian Cycle.



Transform X: Create G' from $G = (V, E)$, $|V| = n$. G' has all edges, and is the complete graph.

$$w(u,v) = \begin{cases} 1 & \text{if } (u,v) \in G \\ c|V| + 1 & \text{if } (u,v) \notin G \end{cases}$$

Transform Y (answer of $TSP_A(G')$):

$$\text{Return: } \begin{cases} \text{Yes} & \text{if } |TSP_A(G')| \leq c|V| \\ \text{No} & \text{otherwise} \end{cases}$$

□

- **Why does this work?** Edges not in the original graph are so costly that there is a gap between cost of tour if G contains a Hamiltonian cycle (cost $= n$) and cost of tour if G doesn't (cost $> c|V|$).

Last, if we assumed the approximating algorithm does nothing when Hamiltonian cycle does not exist, there is no need to transform the graph. However, this is being unnecessarily restrictive on target problem.

2 Online Algorithms

We will see our next topic, online algorithms, is closely related to approximation algorithms. Online algorithms respond to a sequence of requests, and we would like to know how they perform, even if the sequence could be infinitely long. Examples include a data structure, or operating system.

Definition 3 (Online Algorithms)**.** For input sequence $p_1, p_2, \ldots p_n$, an <u>online</u> algorithm must produce an output, given $p_1, p_2, \ldots p_i$ (without seeing $p_{i+1} \ldots p_n$) for each i.

Example 4 (Page replacement in cache)**.**

$p_1, p_2, \dots p_n$	is a sequence of page requests, made by a program
k	is cache size (# pages)

At i^{th} page request p_i, the cache contains some k pages. If p_i is not in cache (page fault occurs; some page **MUST** be evicted from cache to make room for p_i, then p_i is added to cache.

The **cost** of a page replacement algorithm A on a sequence $p_1, p_2, \dots p_n$ is:

$$f_A(p_1, p_2 \dots p_n) = \text{ \# faults on } p_1, p_2 \dots p_n$$

Online algorithms must decide what page to evict without knowing the future requests. Some reasonable page replacement policies include:

Least Recently Used(LRU): Evict page whose most recent request occured furthest in the past.

Least Frequently Used(LFU): Evict page that has been requested least often.

Marking Algorithm: Randomly evicts unmarked pages. Consider this an approximation to LRU by using a "mark" bit to replace timestamp. This is currently in use in Linux.

First In First Out(FIFO): Evict page that has been in cache longest.

Farthest in the Future(Optimal): This is impossible, but if entire sequence of requests is known offline, evicting page used farthest in future is optimal - result was proven by Bellady.

- **How do we decide on a best online algorithm?**

  1. Worst-case performance

  $$\max_{p_1,p_2 \dots p_n} \begin{cases} f_{LRU}(p_1, p_2 \dots p_n) = n \\ f_{LFU}(p_1, p_2 \dots p_n) = n \\ f_{FIFO}(p_1, p_2 \dots p_n) = n \end{cases} \quad \text{where } n = \text{total \# of pages possibly requested.}$$

  2. Average case performance

  Expected # of page faults on sequence of randomly, uniformly, independently chosen pages:

  $$E[f_{LRU}(p_1, p_2 \dots p_n)] = E[f_{LFU}(p_1, p_2 \dots p_n)] = E[f_{FIFO}(p_1, p_2 \dots p_n)] = (1 - \frac{k}{m})n$$

  In all cases, the expectation is the sum of expectation for each request, which has probability density uniformly distributed over all pages. In particular,

  $$P(\text{not in cache}) = 1 - \frac{k}{m}$$

Clearly, neither technique yields interesting results, so we need to try something beyond conventional analysis techniques. A possible direction would be to empirically find a more accurate probability distribution of future page requests, and this is in fact the subject of much research - (beyond the scope of our course).

We will instead compare online algorithms to the optimal offline algorithm. In doing so, we will be reminded of the relative method of analysing approximation algorithms. In both cases, we compare the non-optimal solution of a resource-constrained algorithm against an "infinite-resource" algorithm's optimal solution. Running time constraints force non-optimality in approximation algorithms, while non-optimality is now due to online nature of problem/not knowing future.

3 Competitive Analysis

How does an algorithm's performance compare to the optimal offline algorithm?

Definition 5 (c-competitiveness)**.** An online algorithm A is **c-competitive** if:
There exists b, s.t. for all $p_1, p_2 \ldots p_n$:

$$f_A(p_1, p_2 \ldots p_n) \leq \mathbf{c} f_{OPT}(p_1, p_2 \ldots p_n) + b$$

Theorem 6. *LRU and FIFO are k-competitive.*

Theorem 7. *If A is a deterministic online algorithm for paging, then $c \geq k$.*

Remark. *Randomization does better than k-competitive.*