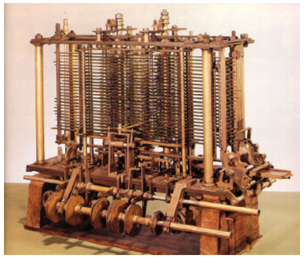
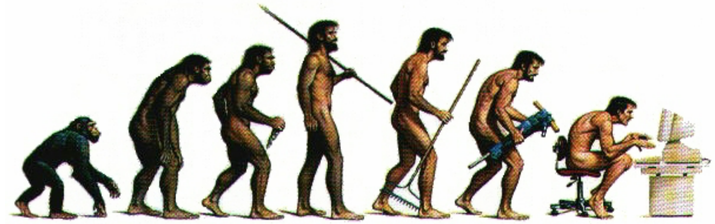
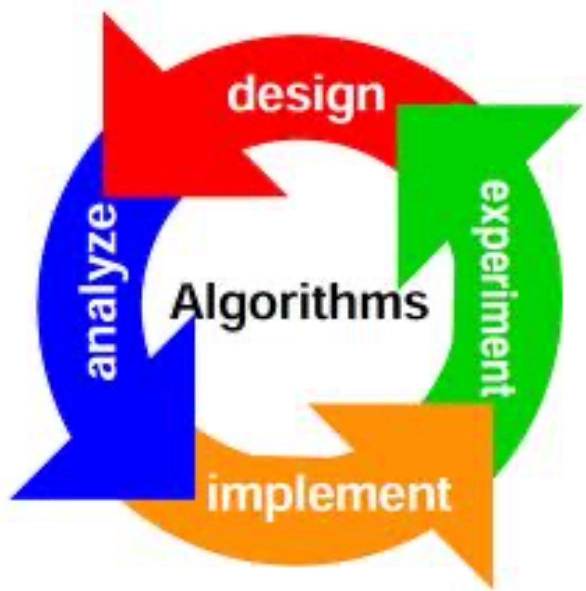




Data Structures and Algorithms II
CMPSC 130B



Approximation Algorithms

Q. Suppose I need to solve an NP-hard problem. What should I do?

A. Theory says you're unlikely to find a poly-time algorithm.

Must sacrifice one of three desired features.

1. Solve problem to optimality.
2. Solve problem in poly-time.
3. Solve arbitrary instances of the problem.

ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.

- Challenge. Need to prove a solution's value is close to optimum, without even knowing what the optimum value is!

Measuring Quality of Approximations

- The standard measure to measure the quality of approximation algorithm A is the ratio:

$$\rho(n) = \max_{inputs\ of\ size\ n} \left(\frac{\text{cost}(A)}{\text{cost}(OPT)} \right)$$

- If the objective is a maximization, then we choose

$$\rho(n) = \max_{inputs\ of\ size\ n} \left(\frac{\text{profit}(OPT)}{\text{profit}(A)} \right)$$

Announcements (March 10)

- Assignments
  - PA2 due Friday March 12
- Final exam @ 8 am on Wed March 17
  - Optional
  - Opt-in decision by 8:30 am on Wed March 17
  - Comprehensive
- Today's plan:
  - Approximations
  - Review
  - Some ongoing work in my group

Please fill out the ESCI surveys by Friday

Vertex Cover

- Given a graph G = (V, E), find a vertex cover C of minimize size; that is, for each edge (u, v) in E, either u or v is in C.
- cost(A) = size of VC found by algorithm A
- cost(OPT) = optimal VC size
- $\rho(n) = \max\left(\frac{\text{cost}(A)}{\text{cost}(OPT)}\right)$ over all n node graphs.

Vertex Cover: Greedy Attempt #1

This problem seems well-suited for greedy strategies. Perhaps the most obvious and natural greedy scheme is the following:

- Repeatedly choose the vertex that covers most edges until no edges left uncovered.

The number of edges covered by a vertex v is its degree d(v). So, we repeatedly

- Choose the max-deg vertex
- Delete all its incident edges
- Until no edges left.

Greedy Attempt #1

- How does this algorithm perform in worst case? Is the worst-case ratio constant (doesn't grow with n)?
- Unfortunately not!
- Counter-example is a bipartite graph: $G_n = (L + R, E)$, where
  - L is a set of n vertices 1..n
  - Then, for each i = 2..n, we add another set of vertices R_i where $|R_i| = \left\lfloor \frac{n}{i} \right\rfloor$, and each vertex of R_i is then joined to i distinct vertices of L. Thus, each vertex of R_i has degree i.
  - R_2 has $\left\lfloor \frac{n}{2} \right\rfloor$ vertices, each of degree 2
  - R_n has one vertex, connected to every vertex in L.

Notes

Greedy Attempt #1

- If we run the greedy algorithm on this example, we will first choose R_n, then R_{n-1}, and so on until we choose all the vertices of R.
- How many vertices are in R?

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{3} \right\rfloor + ..+ 1 = \Theta(n \log n)$$

- So, cost(A) = n log n.
- However, OPT could have just chosen all the vertices of L, of which there are only n.
- So, the ratio is (log n).

Greedy Attempt #2

Now, instead consider another simple greedy scheme:

While E not empty,

1. pick an arbitrary edge $e = (u, v)$
2. add both u and v to the vertex cover
3. delete all edges from E incident to u or v

Theorem: The algorithm chooses a vertex cover whose size is at most twice the optimal.

Notes

Greedy Attempt #2

- Analysis. Let E be the set of edges picked by the greedy. Since no two edges in E can share a vertex, each of them requires a separate vertex in OPT to cover. So, $|OPT| \geq$. $|E|$ On the other hand, our greedy cover has size $2|E|$.
- As we saw, the more natural greedy strategy of repeatedly picking the vertex of max degree can only achieve an approximation ratio of log n.

Traveling Salesman Problem

- Input: $G = (V, E)$, a complete graph, where each edge e has a positive cost $w(e)$.
- Output: The minimum cost tour visiting all the vertices of G exactly once.
- Impossibility Result. There is no polynomial-time algorithm to approximate TSP to any factor unless NP = P.
- Proof: Suppose there is an algorithm that guarantees factor X approximation for TSP in polynomial time. We prove its impossibility by showing that such an algorithm can solve the Hamiltonian cycle problem in polynomial time.

Traveling Salesman Problem

- Given an instance G = (V, E) of the Hamiltonian cycle problem, construct a TSP instance G' as follows.
- Set $w(e) = 1$ if e in E, and $w(e) = (nX + 1)$ otherwise.
- If G contains a Hamiltonian cycle, then the corresponding TSP has cost n, using the Hamiltonian cycle edges, each of cost one.
- The approximation algorithm must find a cycle with cost $\leq nX$.
- Since each edge that is not in G has cost $> nX$, it cannot use that edge. So, the only tours that lead to acceptable approximation are the Hamiltonian cycles in G.
- Conversely, if TSP returns a tour that costs more than nX, it must mean that G does not contain a Hamiltonian cycle.

Notes

Traveling Salesman Problem

TSP with Triangle Inequality

- The good news about TSP is that if the edge costs satisfy triangle inequality, that is, $w(a, b) \leq w(a, c) + w(c, b)$, for any a, b, c, then we can approximate the tour within a factor of 2 using a minimum spanning tree (MST).

Load Balancing: Minimizing makespan

- Problem: Given a set of m machines $M_{1,\ldots,}M_m$, and a set of n jobs, where job j needs t_j time for processing, the goal is to schedule the jobs on these machines so that all the jobs are finished as soon as possible.
- That is, find the minimum time (called makespan) T by which all jobs can be executed collectively.
- Let A_i be the set of jobs assigned to machine M_i. Then, M_i needs time

$$L_i = \sum_{j \in A_i} t_j$$

This is called the load on machine M_i.

- We wish to minimize $L = \max_i L_i$

which is called the makespan.

Load Balancing: Minimizing makespan

- The decision problem is NP-complete: Does there exist a makespan = T?
- It is in NP because we can decide whether the makespan is T or not, given the assignments to machines.
- For NP-completeness, we can reduce subset sum to it (scheduling on two machines).

Load Balancing: Greedy Algorithm

- The greedy algorithm makes one pass over the jobs in any order, and assigns the next job j to the machine with the lightest current load.
- *for* $j = 1..n$
  - Let M_i be the machine with the minimum L_i
  - Assign j to M_i
  - $A_i = Ai \cup \{j\}$
  - $L_i = L_i + t_j$

Time complexity?

Load Balancing: Greedy Algorithm

- For instance, given 6 jobs, with sizes 2, 3, 4, 6, 2, 3, and three machines, the algorithm generates the assignments (2, 6), (3, 2), (4, 3) for a makespan of 8.
- The optimal makespan is 7: (3, 4), (6), (2, 2, 3).

Notes

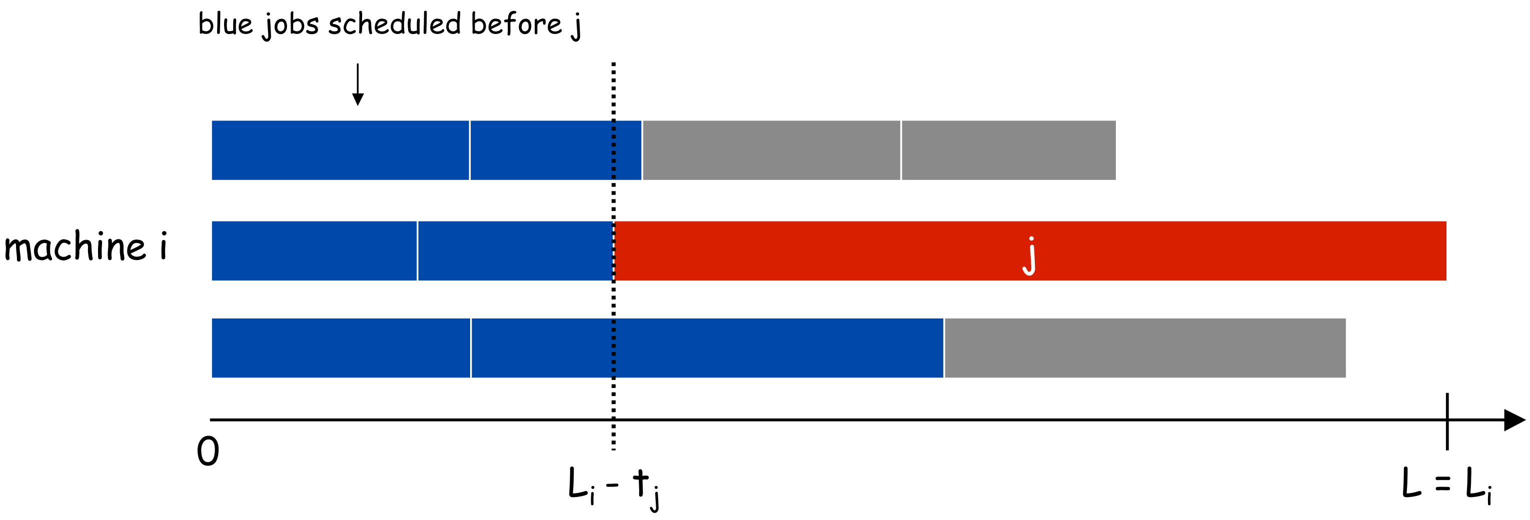
Load Balancing: Greedy Algorithm

- Theorem. [Graham, 1966] Greedy algorithm is a 2-approximation.
- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan L*.
- Lemma 1. $L^* \geq \max_j t_j \geq t_i$
  - Proof. Some machine must process the most time-consuming job.
- Lemma 2. $L^* \geq \frac{1}{m} \sum_j t_j$
  - Proof. One of m machines must do at least a $\frac{1}{m}$ fraction of total work.

Greedy Algorithm Analysis

- Consider load L_i of *bottleneck* machine i.
- Let j be last job scheduled on machine i.
- When job j is assigned to machine i, it had the smallest load. Its load before assignment is $L_i - t_j$. Therefore,

$$L_i - t_j \leq L_k \text{ for all } 1 \leq k \leq m.$$



Greedy Algorithm Analysis

- $L_i - t_j \leq L_k$ for all $1 \leq k \leq m$
- Summing over all k and dividing by m,

$$L_i - t_j \leq \frac{1}{m}\sum_k L_k$$

$$= \frac{1}{m}\sum_j t_j$$

$$\leq L^*$$

Applying Lemma 2

- So, $L_i \leq t_j + L^*$, or $L_i \leq 2\, L^*$ Applying Lemma 1
- It is easy to construct examples where this bound is tight.
- There is a better approximation: if we first sort the jobs in the decreasing order of lengths, and assign them using the greedy strategy, then the approximation factor is 3/2.

Notes