CS 580
ALGORITHM DESIGN AND ANALYSIS

Approximation Algorithms:
Part 1

Vassilis Zikas

SO FAR

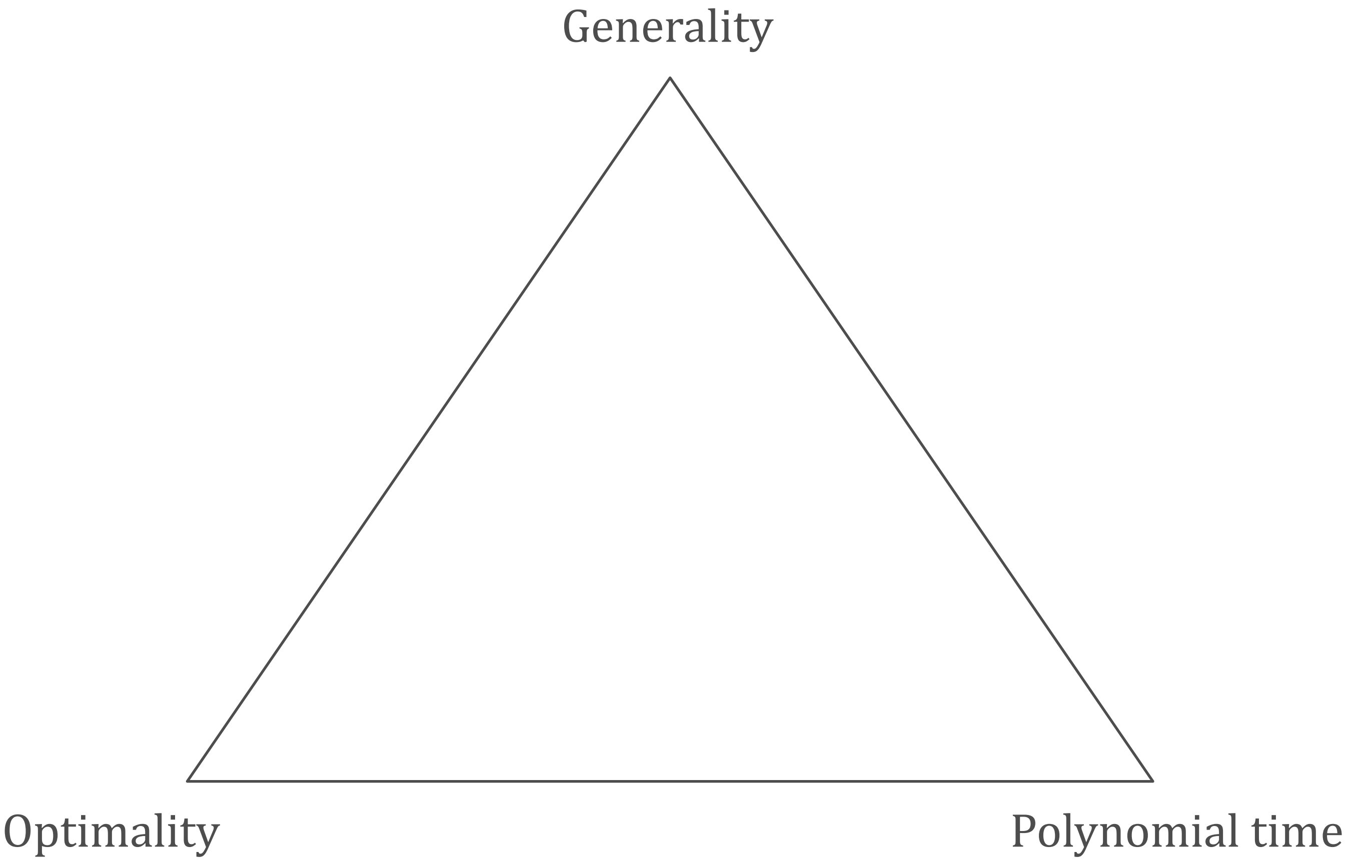
- A wild problem appears
- So far:
  1. Find a polynomial time algorithm 😀
  2. Prove NP-completeness and give up 😠

- But, what if I <u>really</u> need to solve the problem (even if it is NP-hard)?

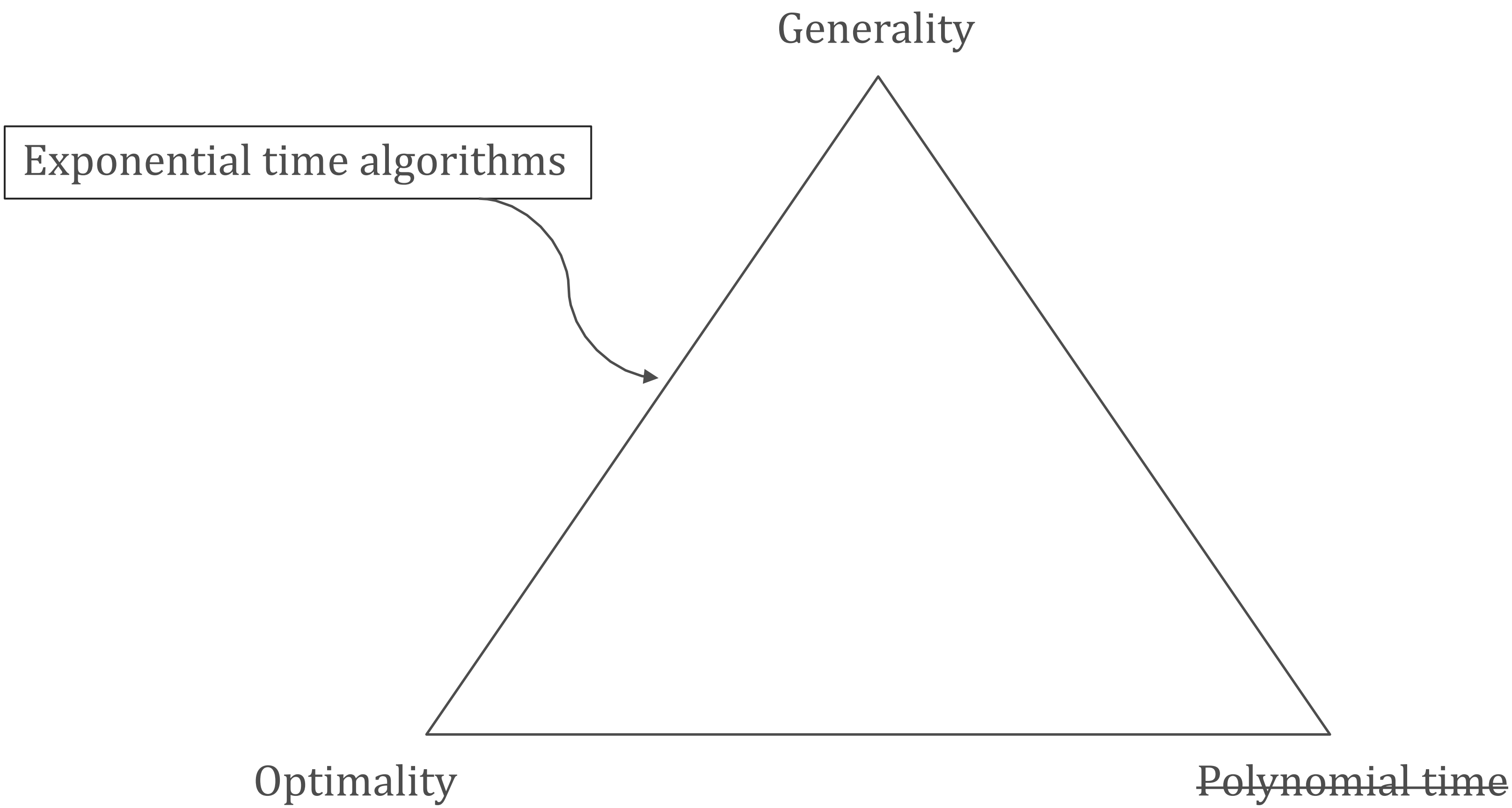
PICK YOUR POISON

- Three features:
  - Generality
  - Optimality
  - Polynomial time
- Pick two!

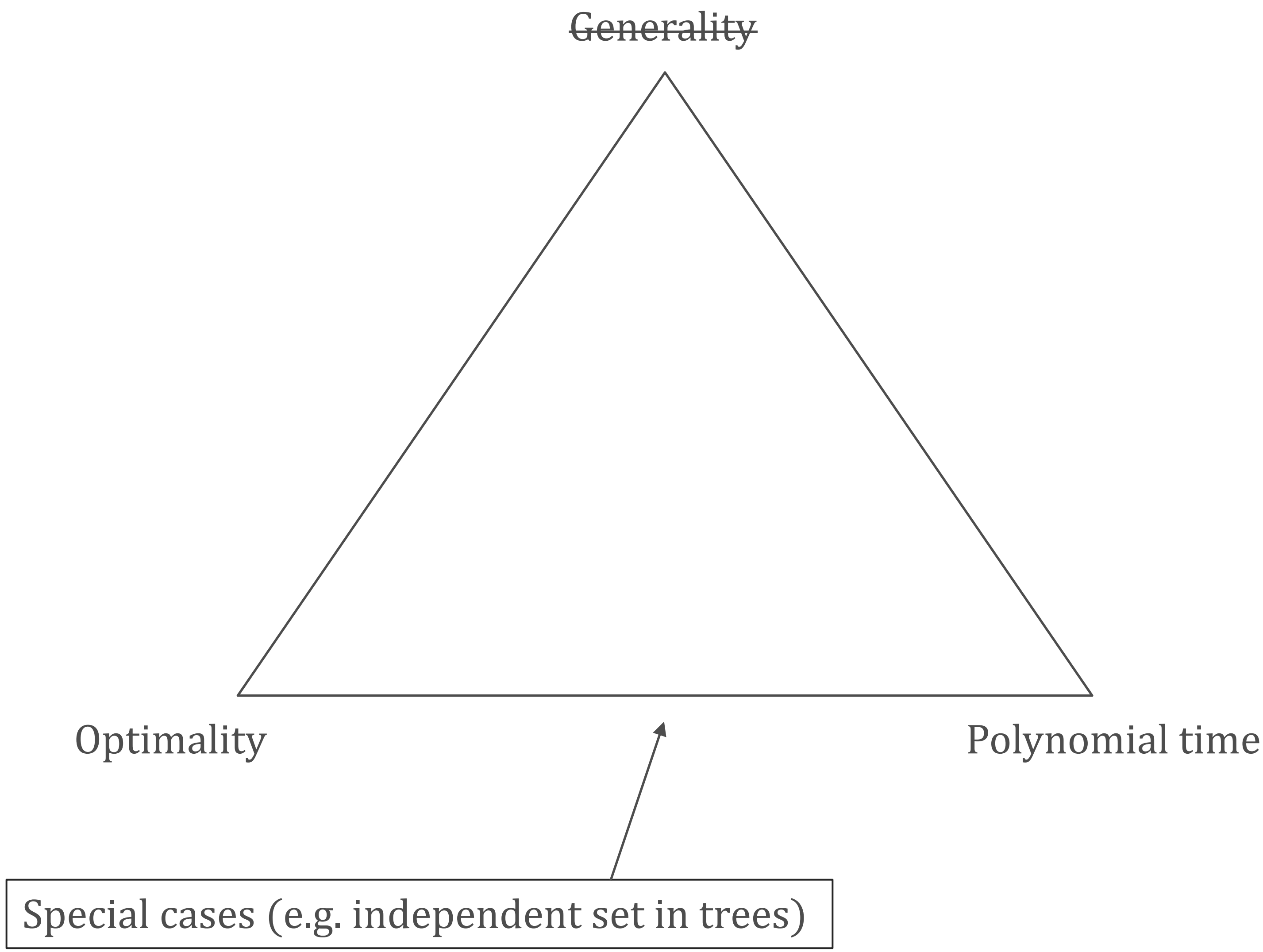
PICK YOUR POISON



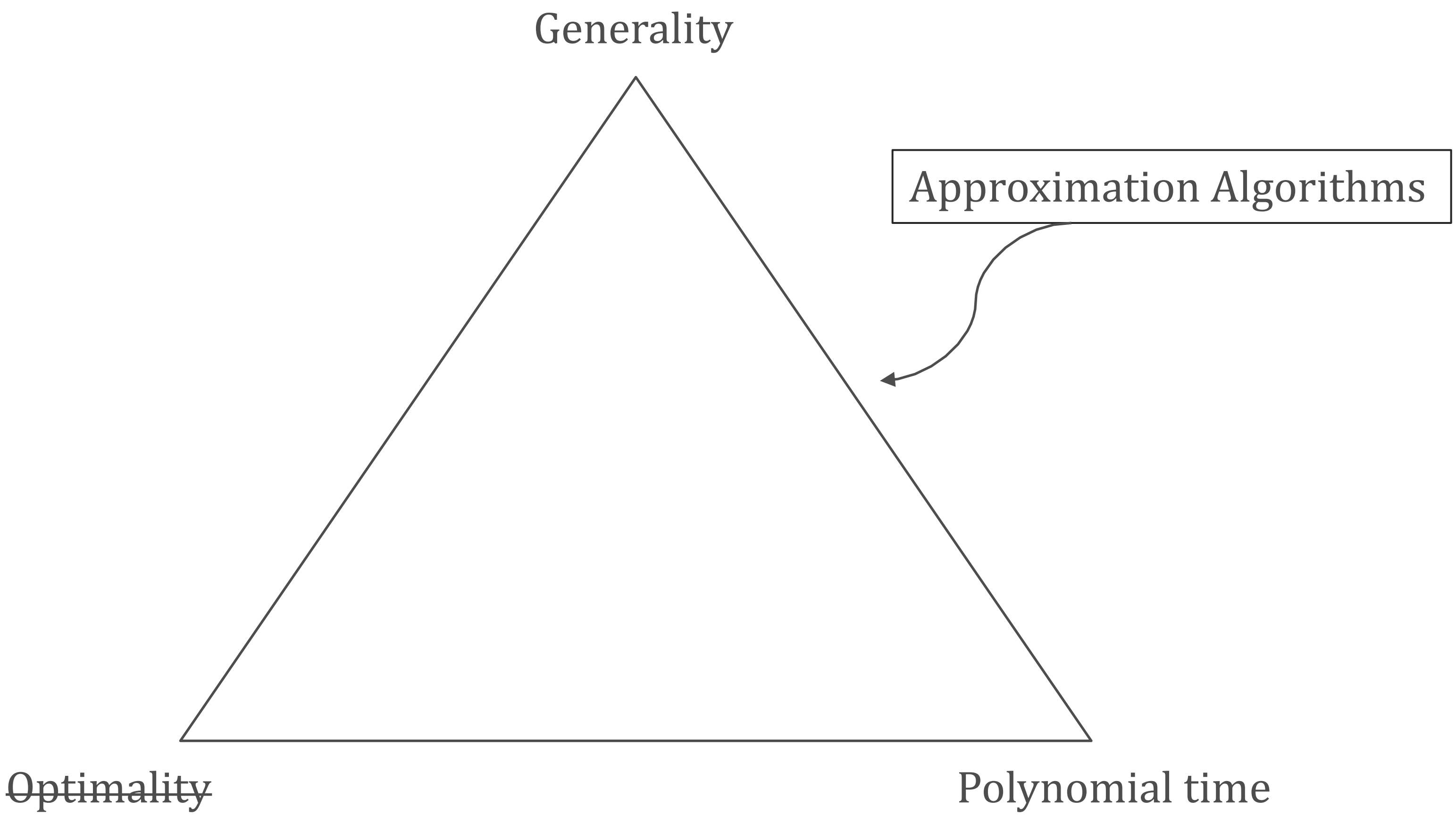
PICK YOUR POISON



PICK YOUR POISON



PICK YOUR POISON



APPROXIMATION ALGORITHMS

- ρ-approximation algorithm
  - Runs in polynomial time
  - The ratio of the true optimum to the output of the algorithm is within ρ

APPROXIMATION ALGORITHMS

- E.g., a 2-approximation algorithm for INDEPENDENT SET would return an independent set whose size is at least ½ the size of the maximum independent set (maximization problems)
- A 2-approximation algorithm for VERTEX COVER would return a vertex cover whose size is at most twice the size of the minimum vertex cover (minimization problems)

TODAY

- LOAD BALANCE (11.1)
- SET COVER (11.3)
- VERTEX COVER

LOAD BALANCING

- Input:
  - m machines: $M_1, \ldots, M_m$
  - n jobs: job j has processing time t_j
- Output:
  - Assign jobs to machines so that machine loads are "balanced"
  - Let $A(i)$ be the set of jobs assigned to M_i
  - Load of M_i is $T_i = \sum_{j \in A(i)} t_j$
  - Goal: Minimize the makespan $T = \max_i T_i$

LOAD BALANCING

- Theorem: LOAD-BALANCING is NP-complete
  - We will not show this, but notice that $m = 2$ is essentially SUBSET SUM
- Our interest:
  - Find, in polynomial time, an assignment with good makespan

LOAD BALANCING

- Greedy algorithm:
  - Take an arbitrary job *j*
  - Assign it to the machine with the smallest load
  - Repeat

```
List-Scheduling(m, n, t1,t2,…,tn) {
   for i = 1 to m {
      Li ← 0        ← load on machine i
      A(i) ← φ      ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmink Lk
      A(i) ← A(i) ∪ {j}
      Li ← Li + tj
   }
   return A(1), …, A(m)
}
```

LOAD BALANCING

- <u>Theorem</u> [Graham 1966!]: Greedy is a 2-approximation

- Lemma 1: The optimal makespan T^* is at least $\max_i t_i$
- Proof:
  - Some machine must process the largest job

LOAD BALANCING

- Lemma 2: The optimal makespan T^* is at least $\frac{1}{m}\sum_j t_j$
- Proof:
  - The total processing time is $\sum_j t_j$
  - One of the m machines must have an at-least-average makespan $\frac{1}{m}\sum_j t_j$

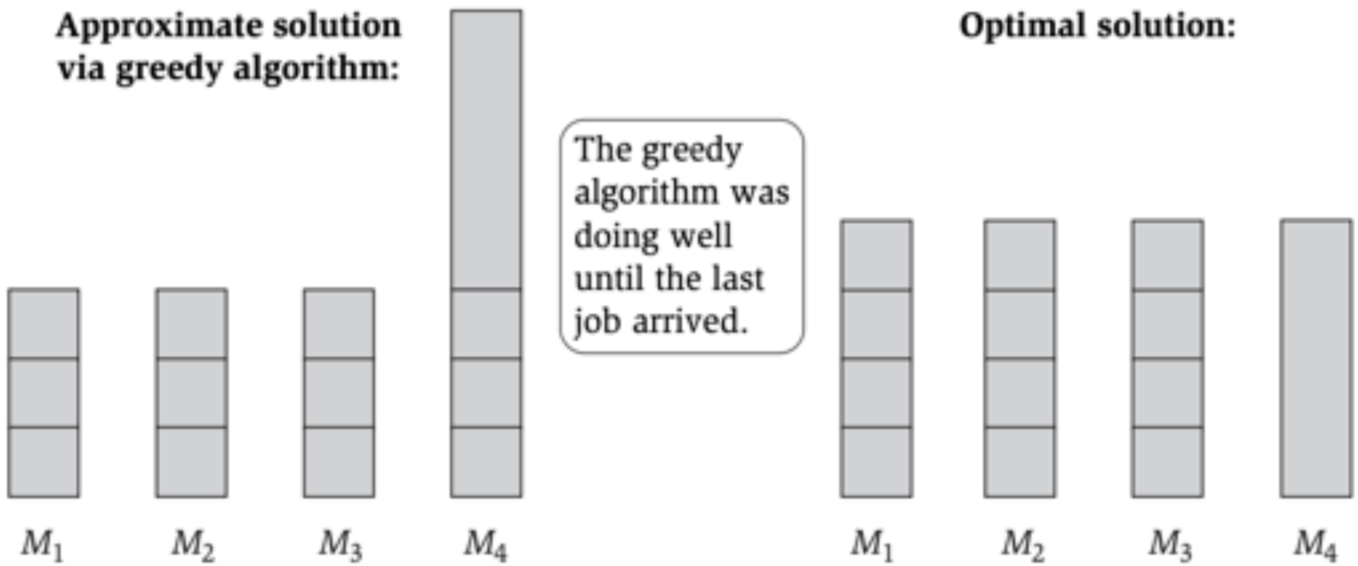
PROOF OF 2-APPROXIMATION

- Let L_i be the load of machine M_i at the end of the algorithm
- Let z be the last job we added to M_i
- When we added z, M_i had the smallest load at that point
  - $L_i - t_z \leq L_k$, for all other machines $k \neq i$
- Sum up over all k and divide by m:
  - $\frac{1}{m}\sum_k L_k \geq L_i - t_z$
- But, $\sum_k L_k = \sum_j t_j$, the total load in the system
- By Lemma 2: $L_i - t_z \leq T^*$
- Overall, $L_i = t_z + (L_i - t_z) \leq 2T^*$

TIGHT ANALYSIS

- Is the factor of 2 tight?
  - Does there exist an instance in which we output a solution exactly 2 times worse than the optimal one?
- Tight example:
  - m machines, $n = m(m-1)$+1 jobs
  - The first $m(m-1)$ take time $t_j = 1$
  - The last job takes time $t_n = m$
  - OPT saves a machine for the last job
  - Greedy evenly balances the first $n-1$ jobs, and then has to add the large job to an already loaded machine

TIGHT ANALYSIS



LOAD BALANCING: CAN WE DO BETTER?

- What went wrong in our tight example?
- We processed the largest job last!
  - Perhaps if we start from the large jobs we can get a better bound for the "damage" the smaller jobs can do
- New algorithm: Sorted-Balance
  1. Sort jobs in decreasing time $t_1 \geq \cdots \geq t_n$
  2. Run Greedy on the sorted jobs:
     - For $j = 1 \ldots n$:
       - $M_i = argmin_k T_k$
       - Assign job j to machine M_i
       - $T_i += t_j$

SORTED-BALANCE ANALYSIS

- If we have fewer than m jobs, then obviously Greedy and Sorted-Balance are optimal
- Claim: If $n > m$, then $T^* \geq 2t_{m+1}$
- Proof:
  - The first m jobs take time at least t_{m+1}
  - There exists a machine assigned at least two of the first $m + 1$ jobs
  - That machine will have makespan at least $2t_{m+1}$

SORTED-BALANCE ANALYSIS

- Theorem: Sorted-Balance is a 3/2 approximation algorithm
- Proof:
  - Let M_i be the machine with the maximum load
  - Assume this machine has at least two jobs, and let j be the last job assigned to it
  - $j \geq m + 1$, since M_i has two jobs
  - $L_i = t_j + \left(L_i - t_j\right) \leq \mathrm{t_{m+1}} + \mathrm{T}^* \leq \frac{3}{2} T^*$
  - If M_i has one job, then our schedule is optimal (since the optimal schedule would also need to put the single job in M_i somewhere)

SORTED-BALANCE

- Is the factor 3/2 tight?
- No!
- Theorem (Graham 1969): Sorted-Balance is a 4/3 approximation

- Is the factor 4/3 tight?
- Yes

SORTED-BALANCE

- 4/3 example
- m machines, $n = 2m + 1$ jobs
- 2 jobs of length m, 2 jobs of length $m + 1$,..., 2 jobs of length $2\mathrm{m} - 1$
- 1 job of length m
- One machine gets 3 jobs (by pigeonhole)
- OPT: $3m$
- Sorted-Balance:
  - All but one of the machines get two jobs, with processing time $3m - 1$
  - One machine gets an extra job of processing time m

WEIGHTED SET COVER

- Input:
  - A set U of n elements
  - A list $S_1, \ldots, S_m$ of subsets of U, with associated weights $w_1, \ldots, w_m$
- Output:
  - A collection of these subsets that covers U and has minimum weight

WEIGHTED SET COVER

- Greedy approach: built up the solution one set at a time
- Which are good sets to include?
- Smaller weight is better
- More elements covered is better
- Pick by $w_i/|S_i|$!
- Almost…
- Pick by $\frac{w_i}{|S_i \cap R|}$, where R is the set of remaining elements (as the algorithm progresses)

WEIGHTED SET COVER

- Greedy:
  - Start with $R = U$
  - While $R \neq \emptyset$:
    - Pick S_i that minimizes $w_i / |R \cap S_i|$
    - Let $c_s = w_i / |R \cap S_i|$ for all $s \in R \cap S_i$ (we'll need this line for the analysis)
    - Delete S_i from R

WEIGHTED SET COVER - ANALYSIS

- Key question in approximation algorithms: what is our benchmark for OPT?
- Typically, we do not know clean expressions for OPT (otherwise, why would we need suboptimal algorithms in the first place?)
- For weighted set cover, what should we lower bound (it's a minimization problem) OPT by?
- On the flip side, what should we upper bound the performance of our algorithm by?

WEIGHTED SET COVER - ANALYSIS

- Observation: If C is the set obtained by greedy, then $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$

WEIGHTED SET COVER - ANALYSIS

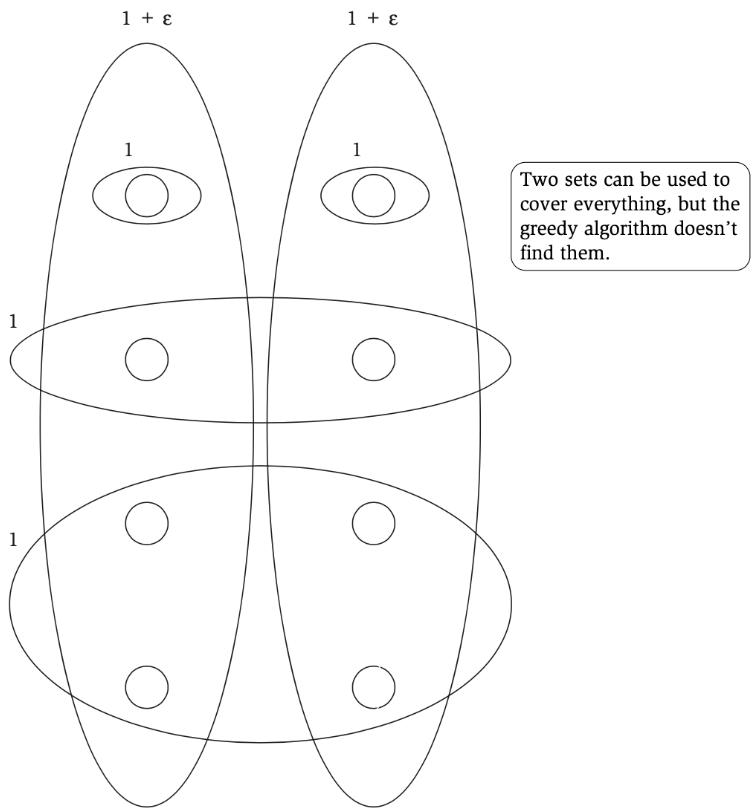
- Lemma 1: For every set S_k, $\sum_{s \in S_k} c_s$ is at most $\mathrm{w_k} \cdot H_{|S_k|}$
  - $H_n = \sum_{i=1}^{n} 1/i$ is the n-th harmonic number
- Proof:
  - $S_k = \{s_1, s_2, \dots, s_d\}$ without loss of generality (and labeled in ordered in which they are assigned a cost)
  - s_j is covered by Greedy at some iteration
  - We know that $s_{j+1}, \dots, s_d \in R$ in that iteration
  - Thus $|S_k \cap R| \geq d - j + 1$
  - $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d-j+1}$
  - Greedy selects the set with minimum average cost S_i
  - $c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d-j+1}$
  - $\sum_{s \in S_k} c_s \leq w_k \cdot \sum_{j=1}^{d} \frac{1}{d-j+1} = w_k \cdot \sum_{j=1}^{d} \frac{1}{j} = w_k \cdot H_d$

WEIGHTED SET COVER - ANALYSIS

- Theorem: Greedy is a H_{d^*} approximation to OPT, where $d^* = \max_i |S_i|$
- Proof:
  - Let C^* be the optimal set cover
  - $w^* = \sum_{S_i \in C^*} w_i$
  - For each set S_i Lemma 1 implies
$$w_i \geq \frac{\sum_{s \in S_i} c_s}{H_{|S_i|}} \geq \frac{\sum_{s \in S_i} c_s}{H_{d^*}}$$
  - $\sum_{S_i \in C^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s$, since C^* is a cover
  - Putting everything together
$$w^* \geq \sum_{S_i \in C^*} \frac{\sum_{s \in S_i} c_s}{H_{d^*}} \geq \frac{\sum_{s \in U} c_s}{H_{d^*}} = \frac{\sum_{S_i \in C} w_i}{H_{d^*}}$$

WEIGHTED SET COVER

- Is this tight?
- Yes!
- Example:
  - Two tall columns of $\frac{n}{2}$ elements each
  - Two sets with weight $1 + \epsilon$ that can cover them
  - $O(logn)$ sets of size $\frac{n}{2}, \frac{n}{4}, \ldots$ with weight 1



WEIGHTED SET COVER

- Can we do better?
- Probably not…
- Unless $P = NP$, there is no o(log(n)), i.e. better than order log(n), approximation algorithm that runs in polytime

VERTEX COVER

- Input: An undirected graph $G = (V, E)$
- Output: The smallest vertex cover, i.e. the smallest subset of vertices S such that for all $e = (u, v) \in E$, either $u \in S$ or $v \in S$

- VERTEX COVER is a special case of SET COVER
  - $U = E, S_i = \{u, v\}$ if $(u, v) \in E$
- Immediately gives a $O(\log(n))$ approximation for VERTEX COVER
  - Repeatedly delete the vertex with the largest degree and include it in the vertex cover

VERTEX COVER

- Can we do better?
- Yes!
- Repeatedly include both endpoints of an edge!

- Greedy:
  - $C = \emptyset$
  - While there exists an edge $e = (u, v)$ such that $u \notin C$ and $v \notin C$:
    - $C = C \cup \{u, v\}$

VERTEX COVER-ANALYSIS

- Observation: Any vertex cover is at least as large as any matching in G
  - A vertex cover has at least one vertex for every edge
  - In a matching, a vertex touches at most one edge
- Therefore, OPT $\geq |M|$, where OPT is the minimum vertex cover, and M is any matching
- Greedy selects a matching M_C!
  - It just puts both endpoints in
- $|C| = 2|M_C| \leq 2OPT$

VERTEX COVER-ANALYSIS

- Can we do better?
- Probably not!
- If UGC (the Unique Games Conjecture) is true, then then minimum vertex cover cannot be approximated within any constant factor better than 2.

SUMMARY

- LOAD BALANCING (11.1)
- SET COVER (11.3)
- VERTEX COVER (DPV 9.2.1)