CS 580: Algorithm Design and Analysis

Jeremiah Blocki
Purdue University
Spring 2019

Upcoming Workshops at Purdue

Algorithms, Mathematical, and Statistical Foundations of Data Science and Applications
April 12-13, 2019

https://datafoundations.cs.purdue.edu/index.html

https://sites.google.com/view/midwesttheoryday2019/home

Midterm Exam 2

Minimum Value 54.5
Maximum Value 133.5
Average 90.5
Median 91.33
Standard Deviation 18.83

Re-grade Requests: You can submit re-grade requests directly on GradeScope (Standard Caveat: Your grade may go up or down)

Competitive Facility Location

Claim. COMPETITIVE-FACILITY is PSPACE-complete.

Pf.

Known PSPACE Complete Problem

- To solve in poly-space, use recursion like QSAT, but at each step there are up to n choices instead of 2.
- To show that it’s complete, we show that QSAT polynomial reduces to it. Given an instance of QSAT, we construct an instance of COMPETITIVE-FACILITY such that player 2 can force a win iff QSAT formula is false.

- Include a node for each literal and its negation and connect them.
- Choose c ≥ k+2, and put weight c on literal x and its negation; set B = c^n + c^{n-2} + ... + c + 1.
- Ensures variables are selected in order xn, xn-1, ..., x1.
- As is, player 2 will lose by 1 unit: c^n + c^{n-2} + ... + c + 1.

- Assume n is odd
Competitive Facility Location

Construction. Given instance \( \Phi(x_1, \ldots, x_n) = \exists x_0 \forall x_0 \cdots \exists x_0 \forall x_0 \cdots \exists x_0 \forall x_0 \) of QSAT.

- Give player 2 one last move on which she can try to win.
- For each clause \( C_j \), add node with value 1 and an edge to each of its literals.
- Player 2 can make last move iff truth assignment defined alternately by the players failed to satisfy some clause.

\[
x_1 \lor x_2 \lor \overline{x_3}
\]

Technical Detail: Eliminate pointless clauses

\[
x_1 \lor x_2 \lor \overline{x_3}
\]

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:

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

\( \rho \)-approximation algorithm.

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

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

Load Balancing

Input: \( m \) identical machines; \( n \) jobs, job \( j \) has processing time \( t_j \).

- Job \( j \) must run contiguously on one machine.
- A machine can process at most one job at a time.

Def. Let \( J(i) \) be the subset of jobs assigned to machine \( i \). The load of machine \( i \) is \( L_i = \sum_{j \in J(i)} t_j \).

Def. The makespan is the maximum load on any machine \( L = \max_i L_i \).

Load balancing: Assign each job to a machine to minimize makespan.

M=2 Machines. Subset Sum problem in disguise

\[
\text{List-scheduling algorithm.}
\]

- Consider \( n \) jobs in some fixed order.
- Assign job \( j \) to machine whose load is smallest so far.

\[
\text{List-Scheduling}(m, n, t_1, t_2, \ldots, t_n) \{
\text{for } i = 1 \text{ to } m \{
L_i \leftarrow 0 \quad \text{load on machine } i
\}
J[1] \leftarrow \emptyset \quad \text{jobs assigned to machine 1}
\}
\text{for } j = 1 \text{ to } n \{
\text{if } i = \text{argmin}_i L_i \text{ then }
J[i] \leftarrow J[i] \cup \{j\}
L_i \leftarrow L_i + t_j \text{ update load of machine } i
\}
\text{return } J(1), \ldots, J(m)
\}
\]

Implementation. \( O(n \log m) \) using a priority queue.

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan $L^*$.

**Lemma 1.** The optimal makespan $L^* \geq \sum_{i=1}^{m} y_i$.

- Some machine must process the most time-consuming job.

**Lemma 2.** The optimal makespan $L^* \geq \frac{2}{n} \sum_{i=1}^{m} y_i$.

- The total processing time is $\sum_{i=1}^{m} y_i$.
- One of the machines must do at least a $\frac{1}{n}$ fraction of total work.

**Theorem.** Consider load $L_i$ of bottleneck machine $i$.

- Let $j$ be last job scheduled on machine $i$.
- When job $j$ assigned to machine $i$, it had smallest load. Its load before assignment is $L_i - t_j = L_k + t_j$ for all $1 \leq k \leq m$.

**Proof.** Consider load $L_i$ of bottleneck machine $i$.

- Let $j$ be last job scheduled on machine $i$.
- When job $j$ assigned to machine $i$, it had smallest load. Its load before assignment is $L_i - t_j = L_k + t_j$ for all $1 \leq k \leq m$.

**Lemma 1** 

$$L_i = \sum_{j=1}^{n} t_j$$

**Lemma 2**

Now

$$L_i = \left( \sum_{j=1}^{n} t_j \right) + t_j \leq 2L^*$$

**Q.** Is our analysis tight?

**A.** Essentially yes.

**Ex:** $m$ machines, $m(m-1)$ jobs length 1 jobs, one job of length $m$

**LPT Rule**

Longest processing time (LPT). Sort $n$ jobs in descending order of processing time, and then run list scheduling algorithm.

```
LPT-List-Scheduling(m, n, t1, t2, ..., tn) {
    Sort jobs so that $t_1 \geq t_2 \geq ... \geq t_n$
    for $i = 1$ to $m$
        $L_i \leftarrow 0$
        $J(i) \leftarrow \phi$
    for $j = 1$ to $n$
        $i = \text{arg} \min_{k} L_k$
        $J(i) \leftarrow J(i) \cup \{j\}$
        $L_i \leftarrow L_i + t_j$
    return $J(1), ..., J(m)$
}
```

**Q.** Is our analysis tight?

**A.** Essentially yes.

**Ex:** $m$ machines, $m(m-1)$ jobs length 1 jobs, one job of length $m$

**Q.** Is our analysis tight?

**A.** Essentially yes.

**Ex:** $m$ machines, $m(m-1)$ jobs length 1 jobs, one job of length $m$
Load Balancing: LPT Rule

Observation. If at most m jobs, then list-scheduling is optimal.
Pf. Each job put on its own machine.

Lemma 3. If there are more than m jobs, \( L^* > 2m + 1 \).
Pf.
  1. Consider first m+1 jobs \( t_1, \ldots, t_{m+1} \).
  2. Since the \( t_i \)'s are in descending order, each takes at least \( t_{m+1} \) time.
  3. There are m+1 jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs.

Theorem. LPT rule is a 3/2 approximation algorithm.
Pf. Same basic approach as for list scheduling.

\[ L_i = \left( L_i - t_j \right) + t_j \leq \frac{3}{2} L^* \]

11.2 Center Selection

Input. Set of n sites \( s_1, \ldots, s_n \) and integer \( k > 0 \).

Center selection problem. Select \( k \) centers \( C \) so that maximum distance from a site to nearest center is minimized.

Notation.
  1. \( \text{dist}(x, y) \) = distance between \( x \) and \( y \).
  2. \( \text{dist}(s, C) = \min_{c \in C} \text{dist}(s, c) \) = distance from \( s \) to closest center.
  3. \( r(C) = \max \text{dist}(s, C) \) = smallest covering radius.

Goal. Find set of centers \( C \) that minimizes \( r(C) \), subject to \( |C| = k \).

Distance function properties.
  1. \( \text{dist}(x, x) = 0 \) (identity)
  2. \( \text{dist}(x, y) = \text{dist}(y, x) \) (symmetry)
  3. \( \text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y) \) (triangle inequality)

Center Selection Example

Ex: each site is a point in the plane, a center can be any point in the plane, \( \text{dist}(x, y) \) = Euclidean distance.

Remark: search can be infinite.
Greedy Algorithm: A False Start

**Greedy algorithm.** Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

**Remark:** arbitrarily bad!

Greedy algorithm. Repeatedly choose the next center to be the site farthest from any existing center.

**Remark.** Greedy algorithm always places centers at sites, but is still within a factor of $2$ of best solution that is allowed to place centers anywhere. E.g., points in the plane.

**Remark.** Is there hope of a $3/2$-approximation? $4/3$?

**Theorem.** Unless $P = NP$, there is no $\rho$-approximation for center-selection problem for any $\rho < 2$.
Weighted Vertex Cover

**Definition.** Given a graph \( G = (V, E) \), a vertex cover is a set \( S \subseteq V \) such that each edge in \( E \) has at least one end in \( S \).

**Weighted vertex cover.** Given a graph \( G \) with vertex weights, find a vertex cover of minimum weight.

\[
\begin{align*}
\text{weight} = 2 + 2 + 4 \\
\text{weight} = 11
\end{align*}
\]

Weighted Vertex Cover: IP Formulation

**Weighted vertex cover.** Given an undirected graph \( G = (V, E) \) with vertex weights \( w_i \geq 0 \), find a minimum weight subset of nodes \( S \) such that every edge is incident to at least one vertex in \( S \).

**Integer programming formulation.**

- Model inclusion of each vertex \( i \) using a 0/1 variable \( x_i \).
  \[
  x_i = \begin{cases} 
  0 & \text{if vertex } i \text{ is not in vertex cover} \\
  1 & \text{if vertex } i \text{ is in vertex cover}
  \end{cases}
  \]
- Vertex covers in 1-1 correspondence with 0/1 assignments: \( S = \{i \in V : x_i = 1\} \).
- Objective function: minimize \( \sum_i w_i x_i \).
- Must take either \( i \) or \( j \): \( x_i + x_j \geq 1 \).

**Observation.** If \( x^* \) is optimal solution to (ILP), then \( S = \{i \in V : x^*_i = 1\} \) is a min weight vertex cover.

\[
\begin{align*}
\text{(ILP)} \min \quad & \sum_i w_i x_i \\
\text{s. t.} \quad & x_i + x_j \geq 1 \quad (i,j) \in E \\
& x_i \in \{0,1\} \quad i \in V
\end{align*}
\]

**Linear Programming**

**Linear programming.** Max/min linear objective function subject to linear inequalities.

- Input: integers \( c_j, b_i, a_{ij} \).
- Output: real numbers \( x_j \).

Linear: \( ax^2 \), \( xy \), \( \arccos(x) \), \( x(1-x) \), etc.

- **Simplex algorithm.** [Dantzig 1947] Can solve LP in practice.
- **Ellipsoid algorithm.** [Khachiyan 1979] Can solve LP in poly-time.
LP Feasible Region

LP geometry in 2D.

LP Relaxation

Weighted vertex cover. Linear programming formulation.

\[
(LP) \begin{align*}
\text{min} & \quad \sum_{i \in V} w_i x_i \\
\text{s.t.} & \quad x_i + x_j \geq 1 \quad (i,j) \in E \\
x_i \geq 0 & \quad i \in V
\end{align*}
\]

Observation. Optimal value of (LP) is \( \leq \) optimal value of (ILP).

Pf. LP has fewer constraints.

Note. LP is not equivalent to vertex cover.

Q. How can solving LP help us find a small vertex cover?

A. Solve LP and round fractional values.

Weighted Vertex Cover

Theorem. If \( x^* \) is optimal solution to (LP), then \( S = \{ i \in V : x^*_i \geq \frac{1}{2} \} \) is a vertex cover whose weight is at most twice the min possible weight.

Pf. \( \{ S \) is a vertex cover\}
- Consider an edge \( (i,j) \in E \).
- Since \( x^*_i + x^*_j \geq 1 \), either \( x^*_i \geq \frac{1}{2} \) or \( x^*_j \geq \frac{1}{2} \) \( \Rightarrow (i,j) \) covered.

Pf. \( \{ S \) has desired cost\}
- Let \( S^* \) be optimal vertex cover. Then
  \[
  \sum_{i \in S^*} w_i = \sum_{i \in S^*} \sum_{j \in \delta(i)} \frac{x^*_j}{2} \geq \frac{1}{2} \sum_{j \in \delta(i)} x^*_j \geq \frac{1}{2} \sum_{j \in \delta(i)} x^*_j
  \]

LP is a relaxation

11.8 Knapsack Problem

Polynomial Time Approximation Scheme

PTAS. \((1 + \varepsilon)\)-approximation algorithm for any constant \( \varepsilon > 0 \).
- Load balancing. (Hochbaum-Shmoys 1987)
- Euclidean TSP. (Arora 1996)

Consequence. PTAS produces arbitrarily high quality solution, but trades off accuracy for time.

This section. PTAS for knapsack problem via rounding and scaling.
Knapsack Problem

Knapsack problem:
- Given $n$ objects and a "knapsack" $W$.
- Item $i$ has value $v_i > 0$ and weight $w_i > 0$.
- A subset $S$ of items $1, \ldots, n$.
- Objective: maximize total value subject to $\sum_{i \in S} w_i \leq W$.

Knapsack is NP-Complete

**KNAPSACK**

Given a finite set $X$, nonnegative weights $w_i$, nonnegative values $v_i$, a weight limit $W$, and a target value $V$, is there a subset $S \subseteq X$ such that:

$$\sum_{i \in S} w_i \leq W$$

$$\sum_{i \in S} v_i \geq V$$

**SUBSET-SUM**

Given a finite set $X$, nonnegative values $v_i$ and an integer $U$, is there a subset $S \subseteq X$ whose elements sum to exactly $U$?

Claim: **SUBSET-SUM** $\leq$ **KNAPSACK**.

Proof: Given instance $(v_1, \ldots, v_n, U)$ of **SUBSET-SUM**, create **KNAPSACK** instance:

$$v_i = v_i - u_i$$

$$W = U$$

$$V = W = U$$

**Intuition for approximation algorithm.**

- Simple and fast.
- Returns optimal among rounded instances.

**Running time.** $O(n W)$.

**Def.** $OPT(i, v) =$ maximum value subset of items $1, \ldots, i$ with weight limit $v$.

**Goal:** Fill knapsack so as to maximize total value. Knapsack can carry weight up to $W$.

Item $i$ has value $v_i > 0$ and weighs $w_i > 0$.

**Knapsack:** $FPTAS$.

**Intuition for approximation algorithm.**

- Simple and fast.
- Runs dynamic programming algorithm on rounded instance.
- Returns optimal items in rounded instance.

**Running time.** $O(n W)$.

**Def.** $OPT(i, w) =$ max value subset of items $1, \ldots, i$ with weight limit $w$.

- Case 1: $OPT$ does not select item $i$.
  - $OPT$ selects best of $1, \ldots, i-1$ using up to weight limit $w - w_i$.
  - consumes weight $w_i$, new value needed = $v_i$.
- Case 2: $OPT$ selects item $i$.
  - $OPT$ selects best of $1, \ldots, i-1$ that achieves exactly value $v$.

**Knapsack:** $FPTAS$.

**Intuition for approximation algorithm.**

- Simple and fast.
- Runs dynamic programming algorithm on rounded instance.
- Returns optimal items in rounded instance.

**Running time.** $O(n W)$.

**Def.** $OPT(i, v) =$ maximum value subset of items $1, \ldots, i$ with weight limit $v$.

- Case 1: $OPT$ does not select item $i$.
  - $OPT$ selects best of $1, \ldots, i-1$ using up to weight limit $v$.
- Case 2: $OPT$ selects item $i$.
  - $OPT$ selects best of $1, \ldots, i-1$ that achieves exactly value $v$.

**Dynamic Program II:**

- Running time is $O(n W).$
- Optimal solution to problems with $V_i$ and $V_*$ are equivalent.
- Dynamic program II running time is $O(n W)$. where

$$W = \frac{V_*}{\epsilon}$$
Knapsack: FPTAS

Knapsack FPTAS. Round up all values: \( \bar{V} = \frac{V}{\theta} \cdot \theta \)

**Theorem.** If \( S \) is solution found by our algorithm and \( S^* \) is any other feasible solution then
\[
\sum_{i \in S} v_i \leq \sum_{i \in S^*} \bar{v}_i \leq \sum_{i \in S} v_i + \theta n
\]
\[
\sum_{i \in S} \bar{v}_i \leq \sum_{i \in S^*} v_i + \theta n
\]
\[
\sum_{i \in S} v_i \leq \sum_{i \in S^*} v_i + \theta n
\]

**Pf.** Let \( S^* \) be any feasible solution satisfying weight constraint.

Always round up solve rounded instance optimally

Never round up by more than

\( \theta \) for each node

Always round up solve rounded instance optimally

Weighted Vertex Cover

**Definition.** Given a graph \( G = (V, E) \), a vertex cover is a set \( S \subseteq V \) such that each edge in \( E \) has at least one end in \( S \).

**Weighted vertex cover.** Given a graph \( G \) with vertex weights, find a vertex cover of minimum weight.

**Weighted Vertex Cover**

weight = 2 + 2 + 4

weight = 11

Pricing Method

**Pricing method.** Each edge must be covered by some vertex.

Edge \( e = (i, j) \) pays price \( p_e \geq 0 \) to use vertex \( i \) and \( j \).

**Fairness.** Edges incident to vertex \( i \) should pay \( \leq w_i \) in total.

for each vertex \( i \), \( \sum_{e \in E} p_e \leq w_i \)

**Lemma.** For any vertex cover \( S \) and any fair prices \( p_e \):

\[
\sum_{i \in S} \sum_{e \in E} p_e \leq \sum_{i \in S} w_i
\]

**Pf.**

\[
\sum_{i \in S} \sum_{e \in E} p_e \leq \sum_{i \in S} w_i
\]

for each edge \( e \) covered by at least one node in \( S \)

and fairness inequalities for each node in \( S \)
Pricing Method: Analysis

Theorem. Pricing method is a 2-approximation.

Proof.
- Algorithm terminates since at least one new node becomes tight after each iteration of while loop.
- Let $S$ be set of all tight nodes upon termination of algorithm. $S$ is a vertex cover: if some edge $i-j$ is uncovered, then neither $i$ nor $j$ is tight. But then while loop would not terminate.
- Let $S^*$ be optimal vertex cover. We show $w(S) \leq 2w(S^*)$.

$$w(S) = \sum_{i \in S} x_i = \sum_{i \in S, x_i > 0} p_i \leq \sum_{i \in S, x_i > 0} \sum_{e \in \leftarrow(i)} p_e \leq 2 \sum_{i \in S^*} p_i \leq 2w(S^*) \quad \text{term set lemma}$$

Generalized Load Balancing

Input. Set of $m$ machines $M$; set of $n$ jobs $J$.
- Job $j$ must run contiguously on an authorized machine in $M_j \subseteq M$.
- Each machine can process at most one job at a time.

Definition. Let $J(i)$ be the subset of jobs assigned to machine $i$. The load of machine $i$ is $L_i = \sum_{j \in J(i)} t_j$.

Definition. The makespan is the maximum load on any machine $\max_i L_i$.

Generalized load balancing. Assign each job to an authorized machine to minimize makespan.

ILP formulation. $x_{ij}$ = time machine $i$ spends processing job $j$.

$$(IP) \quad \min \ L \quad s.t. \begin{align*} \sum_{j \in J} x_{ij} &= t_j \quad \text{for all } j \in J \ 
\sum_{i \in M} x_{ij} &\leq L \quad \text{for all } i \in M \ 
x_{ij} &\in \{0, t_j\} \quad \text{for all } j \in J \text{ and } i \in M_j \ 
x_{ij} &\geq 0 \quad \text{for all } j \in J \text{ and } i \in M_j \ 
x_{ij} &= 0 \quad \text{for all } j \in J \text{ and } i \not\in M_j \end{align*}$$

LP relaxation.

$$(LP) \quad \min \ L \quad s.t. \begin{align*} \sum_{j \in J} x_{ij} &= t_j \quad \text{for all } j \in J \\
\sum_{i \in M} x_{ij} &\leq L \quad \text{for all } i \in M \\
x_{ij} &\geq 0 \quad \text{for all } j \in J \text{ and } i \in M_j \\
x_{ij} &= 0 \quad \text{for all } j \in J \text{ and } i \not\in M_j \end{align*}$$

Generalized Load Balancing: Lower Bounds

Lemma 1. Let $L$ be the optimal value to the LP. Then, the optimal makespan $L^* \geq L$.

Proof. LP has fewer constraints than IP formulation.

Lemma 2. The optimal makespan $L^* \geq \max_j t_j$.

Proof. Some machine must process the most time-consuming job.

* 11.7 Load Balancing Reloaded

Generalized Load Balancing: Structure of LP Solution

Lemma 3. Let $x$ be solution to LP. Let $G(x)$ be the graph with an edge from job $j$ to machine $i$ if $x_{ij} > 0$. Then $G(x)$ is acyclic.

Proof. (deferred)
Generalized Load Balancing: Rounding

**Rounded solution.** Find LP solution $x$ where $G(x)$ is a forest. Root forest $G(x)$ at some arbitrary machine node $r$.

- If job $j$ is a leaf node, assign $j$ to its parent machine $i$.
- If job $j$ is not a leaf node, assign $j$ to one of its children.

**Lemma 4.** Rounded solution only assigns jobs to authorized machines.

**Pf.** If job $j$ is assigned to machine $i$, then $x_{ij} > 0$. LP solution can only assign positive value to authorized machines.

Generalized Load Balancing: Analysis

**Theorem.** Rounded solution is a 2-approximation.

**Pf.** Let $J(i)$ be the jobs assigned to machine $i$.
- By Lemma 6, the load $L_i$ on machine $i$ has two components:
  - leaf nodes
  - parent($i$)
- Thus, the overall load $L_i \leq 2L^*$.

Generalized Load Balancing: Flow Formulation

**Flow formulation of LP.**

- **Observation.** Solution to feasible flow problem with value $L$ are in one-to-one correspondence with LP solutions of value $L$.

Generalized Load Balancing: Structure of Solution

**Lemma 3.** Let $(x, L)$ be solution to LP. Let $G(x)$ be the graph with an edge from machine $i$ to job $j$ if $x_{ij} > 0$. We can find another solution $(x', L)$ such that $G(x')$ is acyclic.

**Pf.** Let $C$ be a cycle in $G(x)$.
- Augment flow along the cycle $C$.
- Flow conservation is maintained.
- At least one edge from $C$ is removed (and none are added).
- Repeat until $G(x')$ is acyclic.

Conclusions

**Running time.** The bottleneck operation in our 2-approximation is solving one LP with $mn + 1$ variables.

**Remark.** Can solve LP using flow techniques on a graph with $m+n+1$ nodes.

**Extensions: unrelated parallel machines.** [Lenstra-Shmoys-Tardos 1990]
- Job $j$ takes $t_j$ time if processed on machine $i$.
- 2-approximation algorithm via LP rounding.
- No 3/2-approximation algorithm unless $P = NP$. 

Copyright 2000, Kevin Wayne
**Load Balancing on 2 Machines**

**Claim.** Load balancing is hard even if only 2 machines.

**Pf.** NUMBER-PARTITIONING ≤ \_ \_ \_ \_ LOAD-BALANCE.

---

**Center Selection: Hardness of Approximation**

**Theorem.** Unless P = NP, there is no \( \rho \)-approximation algorithm for metric k-center problem for any \( \rho < 2 \).

**Pf.** We show how we could use a \((2 - \epsilon)\)-approximation algorithm for k-center to solve DOMINATING-SET in poly-time.

- Let \( G = (V, E) \) be an instance of DOMINATING-SET. \( \ldots \) see Exercise 8.29
- Construct instance \( G' \) of k-center with sites \( V \) and distances
  - \( d(u, v) = 2 \) if \((u, v) \in E\)
  - \( d(u, v) = 1 \) if \((u, v) \notin E\)
- Note that \( G' \) satisfies the triangle inequality.
- **Claim:** \( G \) has dominating set of size \( k \) iff there exists k centers \( C^* \) with \( r(C^*) = 1 \).
- Thus, if \( G \) has a dominating set of size \( k \), a \((2 - \epsilon)\)-approximation algorithm on \( G' \) must find a solution \( C^* \) with \( r(C^*) = 1 \) since it cannot use any edge of distance 2.

---

Copyright 2000, Kevin Wayne