Homework 6 Released Tonight: Due April 23 at 11:59 PM on Gradescope

11.6 LP Rounding: Vertex Cover

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: 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 $x_i$.

$$
\begin{align*}
\text{Model} & \quad \text{of each vertex } i \text{ using a 0/1 variable } x_i, \\
\text{Vertex covers in 1-1 correspondence with 0/1 assignments:} & \quad S = \{i \in V : x_i = 1\}
\end{align*}
$$

- Objective function: minimize $\sum w_i x_i$.

- Must take either $i$ or $j$: $x_i + x_j \geq 1$.

Weighted Vertex Cover: IP Formulation

Weighted vertex cover. Integer programming formulation.

$$(ILP) \min \quad \sum w_i x_i$$

s.t.  \quad $x_i + x_j \geq 1 \quad \forall (i, j) \in E$

$x_i \in \{0, 1\} \quad \forall i \in V$

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

**Observation.** Vertex cover formulation proves that integer programming is NP-hard search problem.

\[
\begin{align*}
\text{max} & \quad c'x \\
\text{s.t.} & \quad \sum_{j} a_{ij} x_j \geq b_i \\
& \quad x_j \geq 0 & 1 \leq i \leq m \\
& \quad x_j \text{ integral} & 1 \leq j \leq n
\end{align*}
\]

Linear Programming

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

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


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
  \[
  \sum_{i \in S^*} w_i \geq \sum_{i \in S} w_i x^*_i \geq \frac{1}{2} \sum_{i \in S} w_i
  \]
- LP is a relaxation \( x^*_i \geq \frac{1}{2} \)

Weighted Vertex Cover

Theorem. 2-approximation algorithm for weighted vertex cover.

**Theorem.** [Dinur-Safra 2001] If \( P \neq \text{NP} \), then no \( \rho \)-approximation for \( \rho < 1.3607 \), even with unit weights.

Open research problem. Close the gap.

**Theorem.** [Khot-Regev 2003] No polynomial time \( \rho \)-approximation for any constant \( \rho < 2 \) under a stronger conjecture called the "Unique Games Conjecture."
12.1 Landscape of an Optimization Problem

Vertex Cover

Given a graph $G = (V, E)$, find a subset of nodes $S$ of minimal cardinality such that for each $u-v$ in $E$, either $u$ or $v$ (or both) are in $S$.

Neighbor relation. If $S \sim S'$, then $S'$ can be obtained from $S$ by adding or deleting a single node. Each vertex cover $S$ has at most $n$ neighbors.

Gradient descent. Start with $S = V$. If there is a neighbor $S'$ that is a vertex cover and has lower cardinality, replace $S$ with $S'$.

Alternative. Run 2-approx alg for Vertex-Cover $S = S_{apx}$ to obtain $Gradient$ Descent with to improve the solution.

Remark. Algorithm terminates after at most $n$ steps since each update decreases the size of the cover by one.

Gradient Descent: Vertex Cover

Local optimum. No neighbor is strictly better.

optimum = center node only
local optimum = all other nodes

optimum = all nodes on left side
local optimum = all nodes on right side

optimum = even nodes
local optimum = omit every third node

Local Search

Local search. Algorithm that explores the space of possible solutions in sequential fashion, moving from a current solution to a “nearby” one.

Neighbor relation. Let $S \sim S'$ be a neighbor relation for the problem.

Gradient descent. Let $S$ denote current solution. If there is a neighbor $S'$ of $S$ with strictly lower cost, replace $S$ with the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.

A funnel
A jagged funnel

11.8 Knapsack Problem

Polynomial Time Approximation Scheme

PTAS. $(1 + \epsilon)$-approximation algorithm for any constant $\epsilon > 0$.

- 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

Given \( n \) objects and a "knapsack." Item \( i \) has value \( v_i > 0 \) and weighs \( w_i > 0 \). Knapsack can carry weight up to \( W \). Goal: fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)

Knapsack Problem: Dynamic Programming I

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

Case 1: \( \text{OPT} \) does not select item \( i \).
- \( \text{OPT} \) selects best of \( 1, \ldots, i-1 \) using up to weight limit \( w \)

Case 2: \( \text{OPT} \) selects item \( i \).
- new weight limit = \( w - w_i \)
- \( \text{OPT} \) selects best of \( 1, \ldots, i-1 \) using up to weight limit \( w - w_i \)

\( \text{OPT}(i, w) = \begin{cases} 
\text{OPT}(i-1, w) & \text{if } i = 0 \text{ or } w_i > w \\
\max (\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w - w_i)) & \text{otherwise} 
\end{cases} \)

Running time: \( O(nW) \)

\( W \) is weight limit.
- Not polynomial in input size.

Knapsack: FPTAS

Intuition for approximation algorithm.
- Round all values up to lie in smaller range.
- Run dynamic programming algorithm on rounded instance.
- Return optimal items in rounded instance.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)

Knapsack Problem: Dynamic Programming II

Def. \( \text{OPT}(v) \) = min weight subset of items \( 1, \ldots, i \) that yields value exactly \( v \).

Case 1: \( \text{OPT} \) does not select item \( i \).
- \( \text{OPT} \) selects best of \( 1, \ldots, i-1 \) that achieves exactly value \( v \)

Case 2: \( \text{OPT} \) selects item \( i \).
- consumes weight \( w_i \), new value needed = \( v - v_i \)
- \( \text{OPT} \) selects best of \( 1, \ldots, i-1 \) that achieves exactly value \( v \)

\( \text{OPT}(v) = \begin{cases} 
\infty & \text{if } v = 0 \\
\text{OPT}(v-1, v) & \text{if } v > 0 \\
\min (\text{OPT}(v-1, v), w_i + \text{OPT}(v-1, v-v_i)) & \text{otherwise} 
\end{cases} \)

Running time: \( O(nV^*) = O(n^2v_{max}) \)

\( V^* \) = optimal value = maximum \( v \) such that \( \text{OPT}(n, v) \leq W \).
- Not polynomial in input size.

Knapsack is NP-Complete

\text{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:

\[ \sum_{i \in S} w_i \leq W \]

\[ \sum_{i \in S} v_i \geq V \]

\text{SUBSET-SUM}: Given a finite set \( X \), nonnegative values \( u_i \), and an integer \( U \), is there a subset \( S \subseteq X \) whose elements sum to exactly \( U \)?

Claim. \( \text{SUBSET-SUM} \leq \text{P KNAPSACK} \).

Pf. Given instance \( \{u_1, \ldots, u_n, U\} \) of \( \text{SUBSET-SUM} \), create \( \text{KNAPSACK} \) instance:

\[ v_i = w_i = u_i \]

\[ V = W = U \]

\[ \sum_{i \in S} u_i \leq U \]

\[ \sum_{i \in S} v_i \geq U \]

Knapsack FPTAS

Round up all values: \( \bar{v}_i = \left\lceil \frac{v_i}{\delta} \right\rceil \) \( \bar{\theta} = \left\lceil \frac{\theta}{\delta} \right\rceil \)

- \( v_{\text{max}} \) = largest value in original instance
- \( \delta \) = precision parameter
- \( \theta \) = scaling factor = \( v_{\text{max}} / n \)

Observe. Optimal solution to problems with \( \bar{v} \) or \( \bar{\theta} \) are equivalent.

Intuition. \( \bar{v} \) close to \( v \) so optimal solution using \( \bar{v} \) is nearly optimal; \( \bar{\theta} \) small and integral so dynamic programming algorithm is fast.

Running time: \( O(n^2/\delta^2) \)

- Dynamic program II running time is \( O(n^2v_{\text{max}}) \), where

\[ \bar{v}_{\text{max}} = \left\lceil \frac{v_{\text{max}}}{\theta} \right\rceil = \left\lceil \frac{v_{\text{max}}}{\delta} \right\rceil \]

\[ \bar{v}_{\text{max}} = \left\lceil \frac{v_{\text{max}}}{\theta} \right\rceil = \left\lceil \frac{v_{\text{max}}}{\delta} \right\rceil \]
Knapsack: FPTAS

Knapsack FPTAS. Round up all values: \( \theta = \left\lceil \frac{W}{w} \right\rceil \)

Theorem. If S is solution found by our algorithm and \( S^* \) is any other feasible solution then
\[
\sum_{i \in S} v_i \leq \sum_{i \in S^*} v_i \\
\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + \left(1 + \frac{\theta}{1 + \theta} \right) \sum_{i \in S \setminus S^*} v_i \\
\]

Pf. Let \( S^* \) be any feasible solution satisfying weight constraint.
\[
\sum_{i \in S} v_i \leq \sum_{i \in S^*} v_i \\
\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + \left(1 + \frac{\theta}{1 + \theta} \right) \sum_{i \in S \setminus S^*} v_i \\
\]

* 11.7 Load Balancing Reloaded

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.

Def. Let \( J(i) \) be the subset of jobs assigned to machine \( i \).

Def. The makespan is the maximum load on any machine: \( \max_i L_i = \max_i \sum_{j \in J(i)} t_j \).

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

Generalized Load Balancing: Integer Linear Program and Relaxation

ILP formulation. \( x_{ij} = \) time machine \( i \) spends processing job \( j \).

\[
\begin{align*}
\text{(LP)} \min & \quad L \\
\text{s.t.} & \quad \sum_{j} x_{ij} = t_j \quad \text{for all } j \in J \\
& \quad \sum_{i} x_{ij} \leq L \quad \text{for all } i \in M \\
& \quad x_{ij} \in \{0, t_j\} \quad \text{for all } j \in J \text{ and } i \in M_j \\
& \quad x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j
\end{align*}
\]

LP relaxation.

\[
\begin{align*}
\text{(LP)} \min & \quad L \\
\text{s.t.} & \quad \sum_{j} x_{ij} = t_j \quad \text{for all } j \in J \\
& \quad \sum_{i} x_{ij} \leq L \quad \text{for all } i \in M \\
& \quad x_{ij} \geq 0 \quad \text{for all } j \in J \text{ and } i \in M_j \\
& \quad x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j
\end{align*}
\]

Generalized Load Balancing: Lower Bounds

Lemma 1. Let \( L \) be the optimal value to the LP. Then, the optimal makespan \( L^* \geq \max_j t_j \).

Pf. LP has fewer constraints than 2P formulation.

Lemma 2. The optimal makespan \( L^* \geq \max_i L_i \).

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

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 machine \( i \) to job \( j \) if \( x_{ij} > 0 \). Then \( G(x) \) is acyclic.

Pf. (deferred)

Generalized Load Balancing: Lower Bounds

Lemma 1. Let \( L \) be the optimal value to the LP. Then, the optimal makespan \( L^* \geq \max_j t_j \).

Pf. LP has fewer constraints than 2P formulation.

Lemma 2. The optimal makespan \( L^* \geq \max_i L_i \).

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

Proof. 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

Lemma 5. If job $j$ is a leaf node and machine $i = \text{parent}(j)$, then $x_{ij} = t_j$.

Proof. Since $i$ is a leaf, $x_{ij} = 0$ for all $j \neq \text{parent}(i)$. LP constraint guarantees $\sum_j x_{ij} = t_j$.

Lemma 6. At most one non-leaf job is assigned to a machine.

Proof. The only possible non-leaf job assigned to machine $i$ is parent($i$).

Theorem. Rounded solution is a 2-approximation.

Proof. 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.

\[
\begin{align*}
\sum_{j} x_{ij} &= t_j \quad \text{for all } j \in J \\
\sum_{j} x_{ij} &\leq L \quad \text{for all } i \in M \\
x_{ij} &\geq 0 \quad \text{for all } j \in J \text{ and } i \in M \\
x_{ij} &= 0 \quad \text{for all } j \in J \text{ and } i \not\in M \\
\end{align*}
\]

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

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: given $L$, find feasible flow if it exists. Binary search to find $L^*$.

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.
11.4 The Pricing Method: 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.

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.

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

\[
\sum_{e \in E} p_e \leq \sum_{i \in S} w_i = w(S).
\]

Pf. For each node in \( S \):

\[
\sum_{e \in E} p_e \leq \sum_{i \in S} w_i \leq \sum_{i \in S} p_i = w(S).
\]

Weighted-Vertex-Cover-Approx(G, w) {
    foreach e in E
        \( p_e = 0 \)
    while (\exists \text{ edge i-j such that neither i nor j are tight})
        select such an edge \( e \)
        increase \( p_e \) as much as possible until i or j tight
    \( S \leftarrow \text{set of all tight nodes} \)
    return \( S \)
}

Theorem. Pricing method is a 2-approximation.

Pf.

1. Algorithm terminates since at least one new node becomes tight after each iteration of while loop.

2. Let \( S \) = 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.

3. Let \( S^* \) be optimal vertex cover. We show \( w(S) \leq 2w(S^*) \).

\[
w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{j \in \text{adj}(i)} p_{ij} \leq \sum_{i \in S} \sum_{j \in \text{adj}(i)} p_{ij} = 2 \sum_{e \in E} p_e \leq 2w(S^*).
\]
Claim. Load balancing is hard even if only 2 machines.
Pf. NUMBER-PARTITIONING \leq_p LOAD-BALANCE.

NP-complete by Exercise 8.26

Claim. Load balancing is hard even if only 2 machines.
Pf. NUMBER-PARTITIONING \leq_p LOAD-BALANCE.

NP-complete by Exercise 8.26

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 \((2 - \varepsilon)\)-approximation algorithm for k-center to solve DOMINATING-SET in poly-time.

Let \( G = (V, E) \), \( k \) be an instance of DOMINATING-SET. 

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 \( k \) if and only if there exists \( k \) centers \( C^* \) with \( r(C^*) = 1 \).

Thus, if \( G \) has a dominating set of size \( k \), a \((2 - \varepsilon)\)-approximation algorithm on \( G' \) must find a solution \( C^* \) with \( r(C^*) = 1 \) since it cannot use any edge of distance 2.

see Exercise 8.29