Approximation Algorithms

or: How I Learned to Stop Worrying and Deal with NP-Completeness

Ong Jit Sheng, Jonathan (A0073924B)

March, 2012

Key Results (I)

General techniques:

- Greedy algorithms
- Pricing method (primal-dual)
- Linear programming and rounding
- Dynamic programming on rounded inputs

Key Results (II)

Approximation results:

- $\frac{3}{2}$-approx of Load Balancing
- 2-approx of Center Selection
- $H(d^*)$-approx of Set Cover
- 2-approx of Vertex Cover
- $(2cm^{1/(c+1)} + 1)$-approx of Disjoint Paths
- 2-approx of Vertex Cover (with LP)
- 2-approx of Generalized Load Balancing
- $(1 + \epsilon)$-approx of Knapsack Problem

Outline

Load Balancing: Problem Formulation

Problem

- m machines $M_1, \ldots, M_m$
- n jobs; each job j has processing time t_j

Goal

- Assign each job to a machine
- Balance loads across all machines

Load Balancing: Problem Formulation

Concrete formulation

- Let $A(i)$ = set of jobs assigned to M_i
- Total load on M_i: $T_i = \sum_{j \in A(i)} t_j$
- Want to minimize the makespan (the max load on any machine), $T = \max_i T_i$

Load Balancing: Algorithm

1st Algo:

procedure GREEDY-BALANCE
  1 pass through jobs in any order.
  Assign job j to to machine with current smallest load.
end procedure

However, this may not produce an optimal solution.

Load Balancing: Analysis of Algorithm

We want to show that resulting makespan T is not much larger than optimum T^*.

Theorem

(11.1) Optimal makespan is at least $T^* \geq \frac{1}{m} \sum_j t_j$.

Proof.

One of the m machines must do at least $\frac{1}{m}$ of total work. $\square$

Theorem

(11.2) Optimal makespan is at least $T^* \geq \max_j t_j$

Load Balancing: Analysis of Algorithm

Theorem

(11.3) GREEDY-BALANCE *produces an assignment with makespan* $T \leq 2T^*$.

Proof.

Let T_i be the total load of machine M_i.
When we assigned job j to M_i, M_i had the smallest load of all machines, with load $T_i - t_j$ before adding the last job.
$\implies$ Every machine had load at least $T_i - t_j$
$\implies \sum_k T_k \geq m(T_i - t_j) \implies T_i - t_j \leq \frac{1}{m} \sum_k T_k$
$\therefore T_i - t_j \leq T^*$
From (11.2), $t_j \leq T^*$ so $T_i = (T_i - t_j) + t_j \leq 2T^*$
$\therefore T = T_i \leq 2T^*$ $\square$

Load Balancing: An Improved Approx Algo

procedure SORTED-BALANCE
  Sort jobs in descending order of processing time (so $t_1 \geq t_2 \geq \cdots \geq t_n$)
  for $j = 1, \ldots, n$ **do**
    Let M_i = machine that achieves $\min_k T_k$
    Assign job j to machine M_i
    Set $A(i) \leftarrow A(i) \cup \{j\}$
    Set $T_i \leftarrow T_i + t_j$
  end for
end procedure

Load Balancing: Analysis of Improved Algo

Theorem

(11.4) If there are more than m jobs, then $T^* \geq 2t_{m+1}$

Proof.

Consider the first $m+1$ jobs in the sorted order.
Each takes at least t_{m+1} time.
There are $m+1$ jobs but m machines.
$\therefore$ there must be a machine that is assigned two of these jobs.
The machine will have processing time at least $2t_{m+1}$. $\square$

Load Balancing: Analysis of Improved Algo

Theorem

(11.5) SORTED-BALANCE *produces an assignment with makespan* $T \leq \frac{3}{2}T^*$

Proof.

Consider machine M_i with the maximum load.
If M_i only holds a single job, we are done.
Otherwise, assume M_i has at least two jobs, and let $t_i =$ last job assigned to M_i.
Note that $j \geq m + 1$, since first m jobs go to m distinct machines.
Thus, $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$ (from 11.4).
Similar to (11.3), but now:
$T_i - t_j \leq T^*$ and $t_j \leq \frac{1}{2}T^*$, so $T_i = (T_i - t_j) + t_j \leq \frac{3}{2}T^*$. □

Outline

Center Selection: Problem Formulation

Problem

- Given a set S of n sites
- Want to select k centers such that they are central

Formally:
We are given an integer k, a set S of n sites and a distance function.
Any point in the plane is an option for placing a center.
Distance function must satisfy:

- $\mathrm{dist}(s, s) = 0 \ \forall s \in S$
- symmetry: $\mathrm{dist}(s, z) = \mathrm{dist}(z, s) \ \forall s, z \in S$
- triangle inequality: $\mathrm{dist}(s, z) + \mathrm{dist}(z, h) \geq \mathrm{dist}(s, h)$

Center Selection: Problem Formulation

Let C be a set of centers.

- Assume people in a given town will shop at the closest mall.
- Define distance of site s from the centers as $\text{dist}(s, C) = \min_{c \in C}\{\text{dist}(s, c)\}$
- Then C forms an r-*cover* if each site is within distance at most r from one of the centers, i.e. $\text{dist}(s, C) \leq r \; \forall s \in S$.
- The minimum r for which C is an r-cover is called the *covering radius* of C, denoted $r(C)$.

Goal

Select a set C of k centers that minimizes $r(C)$.

Center Selection: Algorithm

Simplest greedy solution:

1. Find best location for a single center
2. Keep adding centers so as to reduce by as much as possible.

However, this leads to some bad solutions.

Center Selection: Algorithm

Suppose we know the optimum radius r.
Then we can find a set of k centers C such that $r(C)$ is at most $2r$.
Consider any site $s \in S$. There must be a center $c^* \in C^*$ that covers s with distance at most r from s.
Now take s as a center instead.
By expanding the radius from r to $2r$, s covers all the sites c^* covers.
(i.e. $\mathrm{dist}(s,t) \leq \mathrm{dist}(s,c^*) + \mathrm{dist}(c^*,t) = 2r$)

Center Selection: Algorithm

Assuming we know r:

procedure $\textsc{Center-Select-1}$
    // S' = sites still needing to be covered
    Init $S' = S$, $C = \emptyset$
    while $S' \neq \emptyset$ **do**
        Select any $s \in S'$ and add s to C
        Delete all $t \in S'$ where $\mathrm{dist}(t, s) \leq 2r$
    end while
    if $|C| \leq k$ **then**
        Return C as the selected set of sites
    else
        Claim there is no set of k centers with covering radius at most r
    end if
end procedure

Theorem

(11.6) Any set of centers C returned by the algo has covering radius $r(C) \leq 2r$.

Center Selection: Analysis of Algorithm

Theorem

(11.7) Suppose the algo picks more than k centers. Then, for any set C^* of size at most k, the covering radius is $r(C^*) > r$.

Proof.

Assume there is a set C^* of at most k centers with $r(C^*) \leq r$.
Each center $c \in C$ selected by the algo is a site in S, so there must be a center $c^* \in C^*$ at most distance r from c. (Call such a c^* *close* to c.)
Claim: no $c^* \in C^*$ can be close to two different centers in C.

> Each pair of centers $c, c' \in C$ is separated by a distance of more than $2r$, so if c^* were within distance r from each, this would violate the triangle inequality.

$\implies$ each center $c \in C$ has a close optimal center $c^* \in C^*$, and each of these close optimal centers is distinct
$\implies$ $|C^*| \geq |C|$, and since $|C| > k$, this is a contradiction. $\square$

Center Selection: Actual algorithm

Eliminating the assumption of knowing the optimal radius

We simply select the site s that is furthest from all previously selected sites.

procedure CENTER-SELECT
  Assume $k \leq |S|$ (else define $C = S$)
  Select any site s and let $C = \{s\}$
  while $|C| < k$ **do**
    Select a site $s \in S$ that maximizes $\mathrm{dist}(s, C)$
    Add s to C
  end while
  Return C as the selected set of sites
end procedure

Center Selection: Analysis of Actual Algorithm

Theorem

(11.8) This algo returns a set C of k points such that $r(C) \leq 2r(C^*)$.

Proof.

Let $r = r(C^*)$ denote the minimum radius of a set of k centers.
Assume we obtain a set C of k centers with $r(C) > 2r$ for contradiction.
Let s be a site more than $2r$ from every center in C.
Consider an intermediate iteration (selected a set C' and adding c' in this iteration).
We have $\mathrm{dist}(c', C') \geq \mathrm{dist}(s, C') \geq \mathrm{dist}(s, C) > 2r$.
Thus, $\textsc{Center-Select}$ is a correct implementation of the first k iterations of the while loop of $\textsc{Center-Select-1}$.
But $\textsc{Center-Select-1}$ would have $S' \neq \emptyset$ after selecting k centers, as it would have $s \in S'$, and so it would select more than k centers and conclude that k centers cannot have covering radius at most r.
This contradicts our choice of r, so we must have $r(C) \leq 2r$. □

Outline

Set Cover: Problem Formulation

Problem

Given a set U of n elements and a list $S_1, \ldots, S_m$ of subsets of U, a *set cover* is a collection of these sets whose union is equal to U.
Each set S_i has weight $w_i \geq 0$.

Goal

Find a set cover C minimizing the total weight $\sum_{S_i \in C} w_i$.

Set Cover: Algorithm

Designing the algo (greedy)

- Build set cover one at a time
- Choose next set by looking at $\frac{w_i}{|S_i|}$, "cost per element covered". Since we are only concerned with elements still left uncovered, we maintain the set R of remaining uncovered elements and choose S_i that minimizes $\frac{w_i}{|S_i \cap R|}$.

Set Cover: Algorithm

procedure GREEDY-SET-COVER
  Start with $R = U$ and no sets selected
  while $R \neq \emptyset$ **do**
    Select set S_i that minimizes $\frac{w_i}{|S_i \cap R|}$
    Delete set S_i from R
  end while
  Return the selected sets
end procedure

Note: GREEDY-SET-COVER can miss optimal solution.

Set Cover: Analysis

To analyze the algo, we add the following line after selecting S_i:

Define $c_s = \frac{w_i}{|S_i \cap R|} \; \forall s \in S_i \cap R$

i.e. record the cost paid for each newly covered element

Theorem

(11.9) If C is the set cover obtained by GREEDY-SET-COVER, *then* $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$

We will use the harmonic function $H(n) = \sum_{i=1}^{n} \frac{1}{i}$, bounded above by $1 + \ln n$ and below by $\ln(n+1)$, so $H(n) = \Theta(\ln n)$.

Set Cover: Analysis

Theorem

(11.10) For every set S_k, $\sum_{s \in S_k} c_s$ is at most $H(|S_k|)w_k$.

Proof.

For simplicity, assume S_k contains the first $d = |S_k|$ elements of U, i.e. $S_k = \{s_1, \ldots, s_d\}$. Also assume these elements are labelled in the order in which they are assigned a cost c_{s_j} by the algo.
Consider the iteration when s_j is covered by the algo, for some $j \leq d$
At the start of this iteration, $s_j, s_{j+1}, \ldots, s_d \in R$ by our labelling.
So $|S_k \cap R| \geq d - j + 1$, and so the average cost of S_k is at most $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d-j+1}$
In this iteration, the algo selected a set S_i of min average cost, so S_i has average cost at most that of S_k.
So $c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d-j+1}$, giving us $\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d-j+1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \cdots + \frac{w_k}{1} = H(d)w_k$ □

Set Cover: Analysis

Let $d^* = \max_i |S_i|$ denote the max size of any set. Then

Theorem

(11.11) The set cover C selected by Greedy-Set-Cover *has weight at most $H(d^*)$ times the optimal weight w^*.*

Proof.

Let C^* be the optimum set cover, so $w^* = \sum_{S_i \in C^*} w_i$.
For each of the sets in C^*, (11.10) implies $w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s$.
Since these sets form a set cover, $\sum_{S_i \in C^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s$.
Combining these with (11.9), we get
$w^* = \sum_{S_i \in C^*} w_i \geq \sum_{S_i \in C^*} \left[\frac{1}{H(d^*)} \sum_{s \in S_i} c_s \right] \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} w_i$. □

Set Cover: Analysis

The greedy algo finds a solution within $O(\log d^*)$ of optimal.
Since d^* can be a constant fraction of n, this is a worst-case upper bound of $O(\log n)$.
More complicated means show that no polynomial-time approximation algo can achieve an approximation bound better than $H(n)$ times optimal, unless P=NP.

Outline

Vertex Cover: Problem Formulation

Problem

A *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ such that each edge has at least one end in S. We give each vertex $i \in V$ weight $w_i \geq 0$, and weight of a set S of vertices $w(S) = \sum_{i \in S} w_i$.

Goal

Find a vertex cover S that minimizes $w(S)$.

Vertex Cover: Approximations via Reduction

Note that Vertex Cover $\leq_p$ Set Cover

Theorem

(11.12) One can use the Set Cover approximation algo to give an $H(d)$-approximation algo for weighted Vertex Cover, where d is the maximum degree of the graph.

However, not all reductions work similarly.

Vertex Cover: Pricing Method

Pricing Method (aka primal-dual method)

- Think of weights on nodes as costs
- Each edge pays for its "share" of cost of vertex cover
- Determine prices $p_e \geq 0$ for each edge $e \in E$ such that if each e pays p_e, this will approximately cover the cost of S
- Fairness: selecting a vertex i covers all edges incident to i, so it is "unfair" to charge incident edges in total more than the cost of i
- Prices p_e fair if $\sum_{e=(i,j)} p_e \leq w_i$ (don't pay more than the cost of i)

Vertex Cover: Algorithm

Theorem

(11.13) For any vertex cover S^*, and any non-negative and fair prices p_e, we have $\sum_{e \in E} p_e \leq w(S^*)$.

Proof.

Consider vertex cover S^*.
By definition of fairness, $\sum_{e=(i,j)} p_e \leq w_i \; \forall i \in S^*$.

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*)$$

Since S^* is a vertex cover, each edge e contributes to at least one term p_e to the LHS.
Prices are non-negative, so LHS is at least as large as the sum of all prices, i.e.
$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq w(S^*)$. $\square$

Vertex Cover: Algorithm

We say a node i is *tight* (or "paid for") if $\sum_{e=(i,j)} p_e = w_i$.

procedure $\textsc{Vertex-Cover-Approx}(G, w)$
  Set $p_e = 0$ for all $e \in E$
  while $\exists$ edge $e = (i, j)$ such that neither i nor j is tight **do**
    Select e
    Increase p_e without violating fairness
  end while
  Let S = set of all tight nodes
  Return S.
end procedure

Vertex Cover: Analysis

Theorem

(11.14) The set S *and prices* p *returned by* $\textsc{Vertex-Cover-Approx}$ *satisfy* $w(S) \leq 2\sum_{e\in E} p_e$.

Proof.

All nodes in S are tight, so $\sum_{e=(i,j)} p_e = w_i \; \forall i \in S$.
Adding over all nodes: $w(S) = \sum_{i\in S} w_i = \sum_{i\in S}\sum_{e=(i,j)} p_e$.
An edge $e = (i,j)$ can be included in the sum in the RHS at most twice, so

$$w(S) = \sum_{i\in S}\sum_{e=(i,j)} p_e \leq 2\sum_{e\in E} p_e$$

□

Vertex Cover: Analysis

Theorem

(11.15) The set S returned by VERTEX-COVER-APPROX *is a vertex cover, and its cost is at most twice the min cost of any vertex cover.*

Proof.

Claim 1: S is a vertex cover.

Suppose it does not cover edge $e = (i, j)$. Then neither i nor j is tight, contradicting the fact that the while loop terminated.

Claim 2: Approximate bound.

Let $p =$ prices set by the algo, and S^* be an optimum vertex cover. By (11.14), $2\sum_{e\in E} p_e \geq w(S)$, and by (11.13) $\sum_{e\in E} p_e \leq w(S^*)$, so

$$w(S) \leq 2\sum_{e\in E} p_e \leq 2w(S^*)$$

□

Outline

Disjoint Paths: Problem Formulation

Problem

- Given:
  - a directed graph G
  - k pairs of nodes $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$
  - an integer capacity C
- Each pair (s_i, t_i) is a routing request for a path from s_i to t_i
- Each edge is used by at most c paths

A solution consists of a subset of the requests to satisfy, $I \subseteq \{1, \dots, k\}$, together with paths that satisfy them while not overloading any one edge.

Goal

Find a solution with $|I|$ as large as possible.

Disjoint Paths: First Algorithm

A Greedy Approach (when $c = 1$)

procedure GREEDY-DISJOINT-PATHS
  Set $I = \emptyset$
  until no new path can be found
    Let P_i be the shortest path (if one exists) that is edge-disjoint from previously selected paths, and connects some unconnected (s_i, t_i) pair
    Add i to I and select path P_i to connect s_i to t_i
  end until
end procedure

Disjoint Paths: Analysis of First Algo

Theorem

(11.16) GREEDY-DISJOINT-PATHS *is a* $(2\sqrt{m}+1)$*-approximation algo for Max Disjoint Paths (where* $m = |E|$ *= number of edges).*

Proof.

Let I^* = set of pairs connected in an optimal solution, and P_i^* for $i \in I^*$ be the selected paths.
Let I = the set of pairs selected by the algo, and P_i for $i \in I$ be the corresponding paths.
Call a path *long* if it has at least $\sqrt{m}$ edges, and *short* otherwise.
Let I_s^* be indices in I^* with short P_i^*, and similarly define I_s for I.
G has m edges, and long paths use at least $\sqrt{m}$ edges, so there can be at most $\sqrt{m}$ long paths in I^*.
Now consider short paths I^*. In order for I^* to be much larger than I, there must be many connected pairs that are in I^* but not in I.

Proof.

Consider pairs connected by the optimum by a short path, but not connected by our algo.
Since P_i^* connecting s_i and t_i in I^* is short, the greedy algo would have picked it before picking long paths if it was available.
Since the algo did not pick it, one of the edges e along P_i^* must occur in a path P_j selected earlier by the algo.
We say edge e blocks path P_i^*.
Lengths of paths selected by the algo are monotone increasing.
P_j was selected before P_i^*, and so must be shorter: $|P_j| \leq |P_i^*| \leq \sqrt{m}$, so P_j is short.
Paths are edge-disjoint, so each edge in a path P_j can block at most one path P_i^*.
So each short path P_j blocks at most $\sqrt{m}$ paths in the optimal solution, and so

$$|I_s^* - I| \leq \sum_{j \in I_s} |P_j| \leq |I_s|\sqrt{m} \qquad (*)$$

Proof.

I^* consists of three kinds of paths:

- long paths, of which there are at most $\sqrt{m}$;
- paths that are also in I; and
- short paths that are not in I, fewer than $|I_s|\sqrt{m}$ by (*).

Note that $|I| \geq 1$ whenever at least one pair can be connected.
So $|I^*| \leq \sqrt{m} + |I| + |I_s^* - I| \leq \sqrt{m} + |I| + \sqrt{m}|I_s| \leq \sqrt{m}|I| + |I| + \sqrt{m}|I| = (2\sqrt{m} + 1)|I|$. □

Disjoint Paths: Second Algorithm

Pricing Method (for $c > 1$)

- Consider a pricing scheme where edges are viewed as more expensive if they have been used.
- This encourages "spreading out" paths.
- Define cost of an edge e as its length ℓ_e, and length of a path to be $\ell(p) = \sum_{e \in P} \ell_e$.
- Use a multiplicative parameter β to increase the length of an edge each time it is used.

Disjoint Paths: Second Algorithm

procedure GREEDY-PATHS-WITH-CAPACITY
Set $I = \emptyset$
Set edge length $\ell_e = 1$ for all $e \in E$
until no new path can be found
Let P_i be the shortest path (if one exists) such that adding P_i to the selected set of paths does not use any edge more than c times, and P_i connects some unconnected (s_i, t_i) pair
Add i to I and select P_i to connect s_i to t_i
Multiply the length of all edges along P_i by β
end until
end procedure

Disjoint Paths: Analysis of Second Algo

For simplicity, focus on case $c = 2$. Set $\beta = m^{\frac{1}{3}}$.
Consider a path P_i selected by the algo to be *short* if its length is less than β^2.
Let I_s denote the set of short paths selected by the algo.
Let I^* be an optimal solution, and P_i^* the paths it uses.
Let $\bar{\ell}$ be the length function at the first point at which there are no short paths left to choose.
Consider a path P_i^* in I^* *short* if $\bar{\ell}(P_i^*) < \beta^2$, and *long* otherwise.
Let I_s^* denote the set of short paths in I^*.

Disjoint Paths: Analysis of Second Algo

Theorem

(11.17) Consider a source-sink pair $i \in I^*$ that is not connected by the approx algo; i.e. $i \notin I$. Then $\bar{\ell}(P_i^*) \geq \beta^2$.

Proof.

As long as short paths are being selected, any edge e considered for selection by a third path would already have length $\ell_e = \beta^2$ and hence be long.

Consider the state of the algo with length $\bar{\ell}$. We can imagine the algo running up to this point without caring about the limit of c.

Since s_i, t_i of P_i^* are not connected by the algo, and since there are no short paths left when the length function reaches $\bar{\ell}$, it must be that path P_i^* has length of at least β^2 as measured by $\bar{\ell}$. □

Disjoint Paths: Analysis of Second Algo

Finding the total length in the graph $\sum_e \bar{\ell}_e$

$\sum_e \bar{\ell}_e$ starts out at m (length 1 for each edge). Adding a short path to the solution I_s can increase the length by at most β^3, as the selected path has length at most β^2 (for $c = 2$), and the lengths of the edges are increased by a factor of β along the path. Thus,

Theorem

(11.18) The set I_s of short paths selected by the approx algo, and the lengths $\bar{\ell}$, satisfy the relation $\sum_e \bar{\ell}_e \leq \beta^3 |I_s| + m$.

Disjoint Paths: Analysis of Second Algo

Theorem

(11.19) GREEDY-PATHS-WITH-CAPACITY, *using* $\beta = m^{\frac{1}{3}}$, *is a* $(4m^{\frac{1}{3}} + 1)$*-approx algo for capacity* $c = 2$.

Proof.

By (11.17), we have $\bar{\ell}(P_i^*) \geq \beta^2 \; \forall i \in I^* - I$.
Summing over all paths in $I^* - I$, $\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \geq \beta^2 |I^* - I|$.
On the other hand, each edge is used by at most two paths in I^*, so $\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \leq \sum_{e \in E} 2\bar{\ell}_e$.
Combining these with (11.18): $\beta^2|I^*| \leq \beta^2|I^* - I| + \beta^2|I| \leq \sum_{i \in I^* - I} \bar{\ell}(P_i^*) + \beta^2|I| \leq \sum_{e \in E} 2\bar{\ell}_e + \beta^2|I| \leq 2(\beta^2|I| + m) + \beta^2|I|$.
Dividing throughout by β^2, using $|I| \geq 1$ and setting $\beta = m^{\frac{1}{3}}$, $|I^*| \leq (4m^{\frac{1}{3}} + 1)|I|$. □

Disjoint Paths: Analysis of Second Algo

The same algo works for any capacity $c > 0$. To extend the analysis, choose $\beta = m^{\frac{1}{c+1}}$ and consider paths to be short if their length is at most β^c.

Theorem

(11.20) GREEDY-PATHS-WITH-CAPACITY, *using* $\beta = m^{\frac{1}{c+1}}$, *is a* $(2cm^{\frac{1}{c+1}} + 1)$*-approx algo when the edge capacities are* c.

Outline

Vertex Cover: Problem Formulation

Linear programming (LP)

Given an $m \times n$ matrix A, and vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, find a vector $x \in \mathbb{R}^n$ to solve the following optimization problem:

$$\min(c^t x \text{ s.t. } x \geq 0; Ax \geq b)$$

The most widely-used algo is the simplex method.

Vertex Cover: Integer Programming

Vertex Cover as an Integer Program

Consider a graph $G = (V, E)$ with weight $w_i \geq 0$ for each node $i \in V$.
Have a decision variable x_i for each node i:

$$x_i = \begin{cases} 0 & \text{if } i \text{ not in vertex cover} \\ 1 & \text{otherwise} \end{cases}$$

Create a single n-dimensional vector x where the i-th coordinate corresponds to x_i.
Use linear inequalities to encode the requirement that selected nodes form a vertex cover, and an objective function to encode the goal of minimizing weight.

$$\begin{array}{lll} \text{(VC.IP)} & \text{Min} & \sum_{i \in V} w_i x_i & \\ & \text{s.t.} & x_i + x_j \geq 1 & (i,j) \in E \\ & & x_i \in \{0, 1\} & i \in V \end{array}$$

Theorem

(11.21) S is a vertex cover in G iff vector x satisfies the constraints in (VC.IP). Further, $w(S) = w^t x$.

In matrix form, define matrix A whose columns correspond to nodes in V and whose rows correspond to edges in E:

$$A[e,i] = \begin{cases} 1 & \text{if node } i \text{ is an end of edge } e \\ 0 & \text{otherwise} \end{cases}$$

Then we can write $Ax \geq \vec{1}, \vec{1} \geq x \geq \vec{0}$.

So our problem is:

$$\min(w^t x \text{ subject to } \vec{1} \geq x \geq \vec{0}, Ax \geq \vec{1}, x \text{ has integer coords}) \qquad (\dagger)$$

Theorem

(11.22) Vertex Cover $\leq_p$ Integer Programming

Vertex Cover: Linear Programming

Using Linear Programming for Vertex Cover

Drop the requirement that $x_i \in \{0, 1\}$ and let $x_i \in [0, 1]$ to get an instance of LP, (VC.LP), which we can solve in polynomial time.
Call such a vector x^* and let $w_{LP} = w^t x^*$ be the value of the objective function.

Theorem

(11.23) Let S^* denote a vertex cover of minimum weight. Then $w_{LP} \leq w(S^*)$.

Proof.

Vertex covers of G correspond to integer solutions of (VC.IP), so the minimum of (†) over all integer x vectors is exactly the minimum-weight vertex cover.
In (VC.LP), we minimize over more choices of x, so the minimum of (VC.LP) is no larger than that of (VC.IP). □

Vertex Cover: Linear Programming

Rounding

Given a fractional solution $\{x_i^*\}$, define $S = \{i \in V : x_i^* \geq \frac{1}{2}\}$.

Theorem

(11.24) The set S defined this way is a vertex cover, and $w(S) \leq 2w_{LP}$.

Proof.

Claim 1: S is a vertex cover.

Consider an edge $e = (i, j)$. Claim: at least one of i or j is in S.
Recall that one inequality is $x_i + x_j \geq 1$. So any solution x^* satisfies this: either $x_i^* \geq \frac{1}{2}$ or $x_j^* \geq \frac{1}{2}$.
$\therefore$ at least one of them will be rounded up, and i or j will be in S.

Vertex Cover: Linear Programming

Proof.

Claim 2: $w(S) \leq 2w_{LP}$.

Consider $w(S)$. S only has vertices with $x_i^* \geq \frac{1}{2}$.

Thus, the linear program "paid" at least $\frac{1}{2}w_i$ for node i, and we only pay w_i at most twice as much, i.e.

$w_{LP} = w^t x^* = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S)$.

□

Theorem

(11.25) The algo produces a vertex cover S of at most twice the minimum possible weight:

$$w(S) \overset{(11.24)}{\leq} 2w_{LP} \overset{(11.23)}{\leq} 2w(S^*)$$

Outline

Generalized Load Balancing: Problem Formulation

Problem

- Generalization of the Load Balancing Problem.
- Given a set J of n jobs, and a set M of m machines.
- Additional requirement: for each job, there is just a subset of machines to which it can be assigned, i.e. each job j has a fixed given size $t_j \geq 0$ and a set of machines $M_j \subseteq M$ that it may be assigned to.
- An assignment of jobs is *feasible* if each job j is assigned to a machine $i \in M_j$.

Goal

Minimize load on any machine: Using $J_i \subseteq J$ as the jobs assigned to a machine $i \in M$ in a feasible assignment, and $L_i = \sum_{j \in J_i} t_j$ to denote the resulting load, we seek to minimize $\max_i L_i$.

Generalized Load Balancing: Algorithm

Designing and Analyzing the Algorithm

$$
\text{(GL.IP)} \quad \begin{array}{ll}
\text{Min } L & \\
\sum_i x_{ij} = t_j & \forall j \in J \\
\sum_j x_{ij} \leq L & \forall i \in M \\
x_{ij} \in \{0, t_j\} & \forall j \in J, i \in M_j \\
x_{ij} = 0 & \forall j \in J, i \notin M_j
\end{array}
$$

Theorem

(11.26) An assignment of jobs to machines has load at most L iff the vector x, defined by setting $x_{ij} = t_j$ whenever job j is assigned to machine i, and $x_{ij} = 0$ otherwise, satisfies the constraints in (GL.IP), with L set to the maximum load of the assignment.

Generalized Load Balancing: Algorithm

Consider the corresponding linear program (GL.LP) obtained by replacing the requirement that each $x_{ij} \in \{0, t_j\}$ by $x_{ij} \geq 0 \, \forall j \in J$ and $i \in M_j$.

Theorem

(11.27) If the optimum value of (GL.LP) is L, then the optimal load is at least $L^* \geq L$.

We can use LP to obtain such a solution (x, L) in polynomial time.

Theorem

(11.28) The optimal load is at least $L^* \geq \max_j t_j$. (see 11.2)

Generalized Load Balancing: Algorithm

Rounding the solution when there are no cycles

Problem: the LP solution may assign small fractions of job j to each of the m machines.

Analysis: Some jobs may be spread over many machines, but this cannot happen to too many jobs.

Consider the bipartite graph $G(x) = (V(x), E(x))$: nodes are $V(x) = M \cup J$, and there is an edge $(i, j) \in E(x)$ iff $x_{ij} > 0$.

Theorem

(11.29) Given a solution (x, L) of (GL.LP) s.t. $G(x)$ has no cycles, we can use this solution x to obtain a feasible assignment with load at most $L + L^*$ in $O(mn)$ time.

Proof.

$G(x)$ has no cycles $\implies$ each connected component is a tree.
Pick one component (it is a tree with jobs and machines as nodes).
1. Root the tree at an arbitrary node.
2. Consider a job j. If its node is a leaf of the tree, let machine node i be its parent.
2a. Since j has degree 1 (leaf node), machine i is the only machine that has been assigned any part of job j. $\therefore x_{ij} = t_j$. So this will assign such a job j to its only neighbour i.
2b. For a job j whose node is not a leaf, we assign j to an arbitrary child of its node in the rooted tree.
This method can be implemented in $O(mn)$ time. It is feasible, as (GL.LP) required $x_{ij} = 0$ whenever $i \notin M_j$.

Proof.

Claim: load is at most $L + L^*$.
Let i be any machine, and J_i be its set of jobs.
The jobs assigned to i form a subset of the neighbours of i in $G(x)$: J_i contains those children of node i that are leaves, plus possibly the parent $p(i)$ of node i.
Consider each $p(i)$ separately. For all other jobs $j \neq p(i)$ assigned to i, we have $x_{ij} = t_j$.
$\therefore \sum_{j \in J_i, j \neq p(i)} t_j \leq \sum_{j \in J} x_{ij} \leq L$
For the parent $j = p(i)$, we use $t_j \leq L^*$ (11.28).
Adding the inequalities, we get $\sum_{j \in J_i} x_{ij} \leq L + L^*$. □

By (11.27), $L \leq L^*$, so $L + L^* \leq 2L^*$ (twice of optimum), so we get:

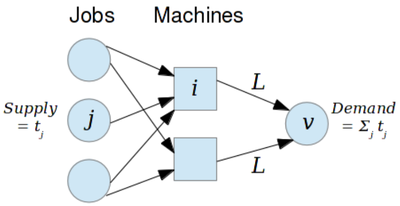
Theorem

(11.30) Restatement of (11.29), but with load at most twice the optimum.

Generalized Load Balancing: Algorithm

Eliminating cycles from the LP solution

We need to convert an arbitrary solution of (GL.LP) into a solution x with no cycles in $G(x)$.
Given a fixed load value L, we use a flow computation to decide if (GL.LP) has a solution with value at most L.
Consider the directed flow graph $G = (V, E)$, with $V = M \cup J \cup \{v\}$, where v is a new node.



Generalized Load Balancing: Algorithm

Theorem

(11.31) The solutions of this flow problem with capacity L are in one-to-one correspondence with solutions of (GL.LP) with value L, where x_{ij} is the flow value along edge (i, j), and the flow value on edge (i, v) is the load $\sum_j x_{ij}$ on machine i.

Thus, we can solve (GL.LP) using flow computations and a binary search for optimal L.
From the flow graph, $G(x)$ is an undirected graph obtained by ignoring directions, deleting v and all adjacent edges, and deleting all edges from J to M that do not carry flow.

Generalized Load Balancing: Algorithm

Theorem

(11.32) Let (x, L) be any solution to (GL.LP) and C be a cycle in $G(x)$. In time linear in the length of the cycle, we can modify x to eliminate at least one edge from $G(x)$ without increasing the load or introducing any new edges.

Proof.

Consider the cycle C in $G(x)$.
$G(x)$ corresponds to the set of edges that carry flow in the solution x.
We augment the flow along cycle C. This will keep the balance between incoming and outgoing flow at any node, and will eliminate one backward edge from $G(x)$.

Generalized Load Balancing: Algorithm

Proof.

Assume the nodes along the cycle are $i_1, j_1, i_2, j_2, \ldots, i_k, j_k$, where i_l is a machine node and j_l is a job node.

We decrease the flow along all edges (j_l, i_l) and increase the flow on the edges (j_l, i_{l+1}) for all $l = 1, \ldots, k$ (where $k+1$ is used to denote 1), by the same amount δ.

This does not affect flow conservation constraints.

By setting $\delta = \min_{l=1}^{k} x_{i_l j_l}$, we ensure that the flow remains feasible and the edge obtaining the minimum is deleted from $G(x)$. $\square$

Generalized Load Balancing: Algorithm

Theorem

(11.33) Given an instance of the Generalized Load Balancing Problem, we can find, in polynomial time, a feasible assignment with load at most twice the minimum possible.

Outline

Knapsack Problem: Problem Formulation

Problem

n items to pack in a knapsack with capacity W.
Each item $i = 1, \dots, n$ has two *integer* parameters: weight w_i and value v_i.

Goal

- Find a subset S of items of maximum value s.t. total weight $\leq W$, i.e. maximize $\sum_{i \in S} v_i$ subject to the condition $\sum_{i \in S} w_i \leq W$.
- In addition, we take a parameter ϵ, the desired precision.
- Our algo will find a subset S satisfying the conditions above, and with $\sum_{i \in S} v_i$ at most a $(1 + \epsilon)$ factor below the maximum possible.
- The algo will run in polynomial time for any fixed choice of $\epsilon > 0$.
- We call this a *polynomial-time approximation scheme*.

Knapsack Problem: Algorithm

Designing the Algorithm

We use a DP algo (given later) with running time $O(n^2 v^*)$ (pseudopolynomial) where $v^* = \max_i v_i$.
If values are large, we use a rounding parameter b and consider the values rounded to an integer multiple of b.
For each item i, let its rounded value be $\bar{v_i} = \lceil \frac{v_i}{b} \rceil b$. The rounded and original values are close to each other.

Theorem

(11.34) For each item i, $v_i \leq \bar{v_i} \leq v_i + b$.

Knapsack Problem: Algorithm

Instead of solving with the rounded values, we can divide all values by b and get an equivalent problem.
Let $\hat{v}_i = \frac{\bar{v}_i}{b} = \lceil \frac{v_i}{b} \rceil b$ for $i = 1, \ldots, n$.

Theorem

(11.35) The Knapsack Problem with values $\bar{v}_i$ and the scaled problem with values $\hat{v}_i$ have the same set of optimum solutions, the optimum values differ exactly by a factor of b, and the scaled values are integral.

Knapsack Problem: Algorithm

Assume all items have weight at most W. Also assume for simplicity that ϵ^{-1} is an integer.

procedure $\textsc{Knapsack-Approx}(\epsilon)$
  Set $b = \frac{\epsilon}{2n} \max_i v_i$
  Solve Knapsack Problem with values $\hat{v}_i$ (equivalently $\bar{v}_i$)
  Return the set S of items found
end procedure

Theorem

(11.36) The set of items S returned by $\textsc{Knapsack-Approx}$ has total weight $\sum_{i \in S} w_i \leq W$.

Knapsack Problem: Analysis

Theorem

(11.37) Knapsack-Approx runs in polynomial time for any fixed $\epsilon > 0$.

Proof.

Setting b and rounding item values can be done in polynomial time.
The DP algo we use has running time $O(n^2 v^*)$ $(v^* = \max_i v_i)$ for integer values.
In this instance, each item i has weight w_i and value $\hat{v}_i$.
The item j with max value $v_j = \max_i v_i$ also has maximum value in the rounded problem, so $\max_i \hat{v}_i = \hat{v}_j = \lceil \frac{v_j}{b} \rceil = 2n\epsilon^{-1}$.
Hence, the overall running time is $O(n^3 \epsilon^{-1})$. □

Note: This is polynomial for any fixed $\epsilon > 0$, but dependence on ϵ is not polynomial.

Theorem

(11.38) If S is the solution found by Knapsack-Approx, and S^* is any other solution, then $(1+\epsilon)\sum_{i\in S} v_i \geq \sum_{i\in S^*} v_i$.

Proof.

Let S^* be any set satisfying $\sum_{i\in S^*} w_i \leq W$.
Our algo finds the optimal solution with values $\bar{v}_i$, so we have $\sum_{i\in S^*} \bar{v}_i \leq \sum_{i\in S} \bar{v}_i$. Thus,

$$\sum_{i\in S^*} v_i \overset{(11.34)}{\leq} \sum_{i\in S^*} \bar{v}_i \leq \sum_{i\in S} \bar{v}_i \overset{(11.34)}{\leq} \sum_{i\in S} (v_i + b) \leq nb + \sum_{i\in S} v_i \qquad (\ddagger)$$

Let j be the item with the largest value; by our choice of b, $v_j = 2\epsilon^{-1} nb$ and $v_j = \bar{v}_j$.
By our assumption $w_i \leq W \ \forall i$, we have $\sum_{i\in S} \bar{v}_i \geq \bar{v}_j = 2\epsilon^{-1} nb$.
From $(\ddagger)$, $\sum_{i\in S} v_i \geq \sum_{i\in S} \bar{v}_i - nb$, and thus $\sum_{i\in S} v_i \geq (2\epsilon^{-1} - 1)nb$.
Hence $nb \leq \epsilon \sum_{i\in S} v_i$ for $\epsilon \leq 1$, so
$\sum_{i\in S^*} v_i \leq \sum_{i\in S} v_i + nb \leq (1+\epsilon)\sum_{i\in S} v_i$. □

Knapsack Problem: DP Algo

The Dynamic Programming Algo

Our subproblems are defined by i and a target value V, and $\overline{\mathrm{OPT}}(i, V)$ is the smallest knapsack weight W so that one can obtain a solution using a subset of items $\{1, \ldots, i\}$ with value at least V.
We will have a subproblem for all $i = 0, \ldots, n$ and values $V = 0, \ldots, \sum_{j=1}^{i} v_j$.
If v^* denotes $\max_i v_i$, then the largest V can get is $\sum_{j=1}^{n} v_j \leq nv^*$.
Thus, assuming values are integral, there are at most $O(n^2 v^*)$ subproblems.
We are looking for the largest value V s.t. $\overline{\mathrm{OPT}}(n, V) \leq W$.

Knapsack Problem: DP Algo

Recurrence relation:

Consider whether last item n is included in the optimal solution $\mathcal{O}$.

- If $n \notin \mathcal{O}$, then $\overline{\mathrm{OPT}}(n, V) = \overline{\mathrm{OPT}}(n-1, V)$.
- If $n \in \mathcal{O}$, then $\overline{\mathrm{OPT}}(n, V) = w_n + \overline{\mathrm{OPT}}(n-1, \max(0, V - v_n))$.

Theorem

(11.39) If $V > \sum_{i=1}^{n-1} v_i$, then $\overline{\mathrm{OPT}}(n, V) = w_n + \overline{\mathrm{OPT}}(n-1, V - v_n)$. Otherwise,
$\overline{\mathrm{OPT}}(n, V) = \min(\overline{\mathrm{OPT}}(n-1, V), w_n + \overline{\mathrm{OPT}}(n-1, \max(0, V - v_n)))$.

Theorem

(11.40) $\textsc{Knapsack}(n)$ takes $O(n^2 v^*)$ time and correctly computes the optimal values of the subproblems.

Knapsack Problem: DP Algo

procedure $\textsc{Knapsack}(n)$
  Array $M[0 \dots n, 0 \dots V]$
  for $i = 0, \dots, n$ **do**
    $M[i, 0] = 0$
  end for
  for $i = 1, 2, \dots, n$ **do**
    for $V = 1, \dots, \sum_{j=1}^{i} v_j$ **do**
      if $V > \sum_{j=1}^{i-1} v_j$ **then**
        $M[i, V] = w_i + M[i-1, V]$
      else
        $M[i, V] = \min(M[i-1, V],$
          $w_i + M[i-1, \max(0, V - v_i)])$
      end if
    end for
  end for
  Return the maximum value V s.t. $M[n, V] \leq W$
end procedure

Conclusion

Many important problems are NP-complete. Even if we may not be able to find an efficient algorithm to solve these problems, we would still like to be able to approximate solutions efficiently. We have looked at some techniques for proving bounds on the results of some simple algorithms, and techniques for devising algorithms to obtain approximate solutions.

LP rounding algorithms for Generalized Load Balancing and Weighted Vertex Cover illustrate a widely used method for designing approximation algorithms:

- Set up an integer programming formulation for the problem
- Transform it to a related linear programming problem
- Round the solution

Conclusion

One topic not covered is *inapproximability*.
Just as one can prove that a given NP-hard problem can be approximated to within a certain factor in polynomial time, one can sometimes establish lower bounds, showing that if the problem can be approximated better than some factor c in polynomial time, then it could be solved optimally, thereby proving P=NP.

The End

The End

$\square$