Applied Algorithm Design
Lecture 5

Pietro Michiardi

Eurecom

Approximation Algorithms

Introduction

- In the first lectures, we discussed about NP-completeness and the idea of computational intractability in general
- How should we design algorithms for problems where polynomial time is probably an unattainable goal?

Approximation algorithms:

- Run in polynomial time
- Find solutions that are **guaranteed** to be **close** to optimal

Introduction

- Since we will not be seeking at optimal solutions, polynomial running time becomes feasible
- We are interested in proving that our algorithms find solutions that are guaranteed to be close to the optimum

But how can you compare to and reason about the optimal solution that is computationally very hard to find?

Approximation techniques

1. **Greedy algorithms**: simple and fast algorithms that require finding a greedy rule that leads to solution probably close to optimal
2. **Pricing method**: motivated by an economic perspective, these algorithms consider a price one has to pay to enforce each constraint of the problem. A.k.a. *Primal-dual technique*
3. **Linear programming and rounding**: these algorithms exploit the relationship between the computational feasibility of linear programming and the expressive power of *integer programming*
4. **Dynamic programming and rounding**: complex technique that achieves extremely good approximations

The Greedy Approach

Greedy Algorithms and Bounds on the Optimum

To illustrate this first technique we consider a fundamental problem: **Load Balancing**.

Load Balancing:

- This is a problem with many facets
- A simple instance of this problem arises when multiple servers need to process a set of jobs or requests
- We look at the case in which all servers are identical and each can be used to serve any of the requests

This problem is useful to learn how to compare an approximate solution with an optimum solution that we cannot compute efficiently.

Load Balancing: The Problem

Definition:

- Given a set $M = \{M_1, M_2, ..., M_m\}$ of m machines
- Given a set $J = \{1, 2, ...n\}$ of n jobs, with job j having a processing time t_j

The goal is to assign each job to one of the machines so that the loads placed on all machines are as "balanced" as possible

Load Balancing: Some Useful Definitions

Assignment:

Let $A(i)$ denote the set of jobs assigned to machine M_i

Load:

Under the assignment defined above, machine M_i needs to work for a total time of:

$$T_i = \sum_{j \in A(i)} t_j$$

Makespan:

We denote the maximum load on any machine to be a quantity known as the makespan:

$$T = \max_i T_i$$

Our Goal, Revisited

Our goal is thus to minimize the makespan

Although we will not prove it, the scheduling problem of finding an assignment of minimum makespan is NP-hard

Designing the algorithm

- Consider a simple greedy algorithm, which makes one pass through the jobs in any order
- When it comes to job j, it assigns j to the machine whose load is the smallest so far

---

Algorithm 1: Greedy-Balance:

Start with no jobs assigned
Set $T_i = 0$ and $A(i) = \emptyset$ for all machines M_i
for $j = 1 \cdots n$ **do**
  $i \leftarrow \arg\min_k T_k$
  $A(i) \leftarrow A(i) \cup \{j\}$
  $T_i \leftarrow T_i + t_j$
end

Question: running time? implementation?

Analyzing the 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 T^*

- We need to compare our solution to the optimal value T^*, which is unknown
- We need a **lower bound** on the optimum: a quantity that no matter how good the optimum is, it cannot be less than this bound

Analyzing the algorithm

Lemma 1

The optimal makespan is $T^* \geq \max_j t_j$

Proof.

Some machine must process the most time-consuming job □

Lemma 2

The optimal makespan is $T^* \geq \frac{1}{m} \sum_j t_j$

Proof.

The total processing time is $\sum_j t_j$
One of m machines must do at least a $1/m$ fraction of total work □

Analyzing the algorithm

Theorem

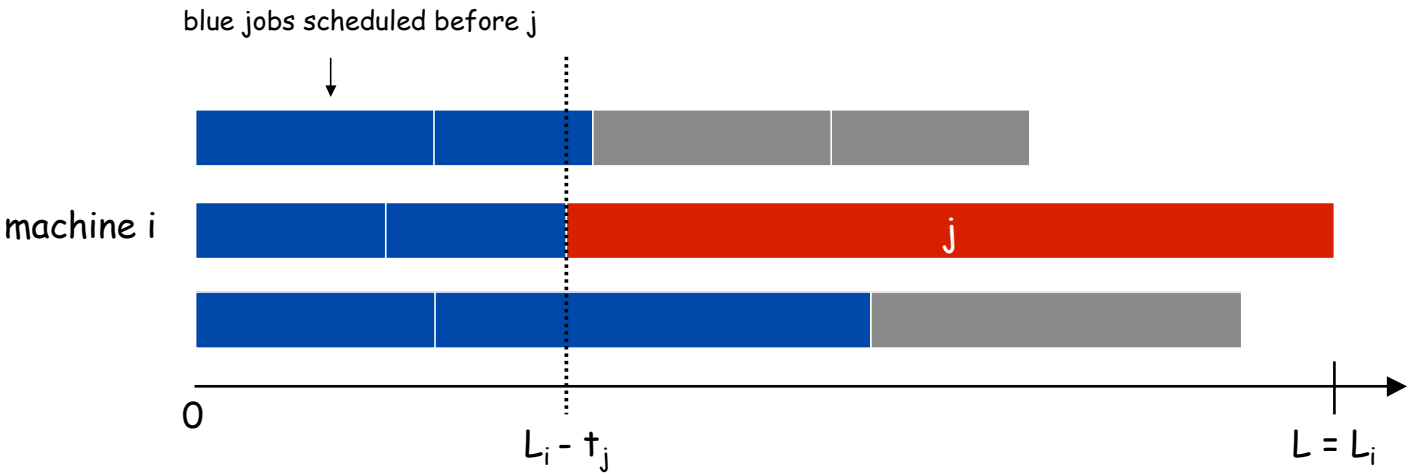
Greedy algorithm is a 2-approximation, that is it produces an assignment of jobs to machines with a makespan $T \leq 2T^*$

Proof.

- Consider load T_i of "bottleneck" machine M_i
- Let j be last job scheduled on machine M_i
- When job j assigned to machine M_i, M_i had smallest load
- Its load before assignment is $T_i - t_j$

$$\Rightarrow T_i - t_j \leq T_k \forall k \in \{1 \cdots m\}$$

Analyzing the algorithm



Analyzing the algorithm

Proof.

- Sum inequalities over all k and divide by m:

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

$$= \frac{1}{m} \sum_k t_k$$

$$\leq T^* \quad \leftarrow \text{From Lemma 2}$$

- Now:

$$T_i = (T_i - t_j) + t_j \leq 2 \cdot T^* \quad \leftarrow \text{From Lemma 1}$$

Example

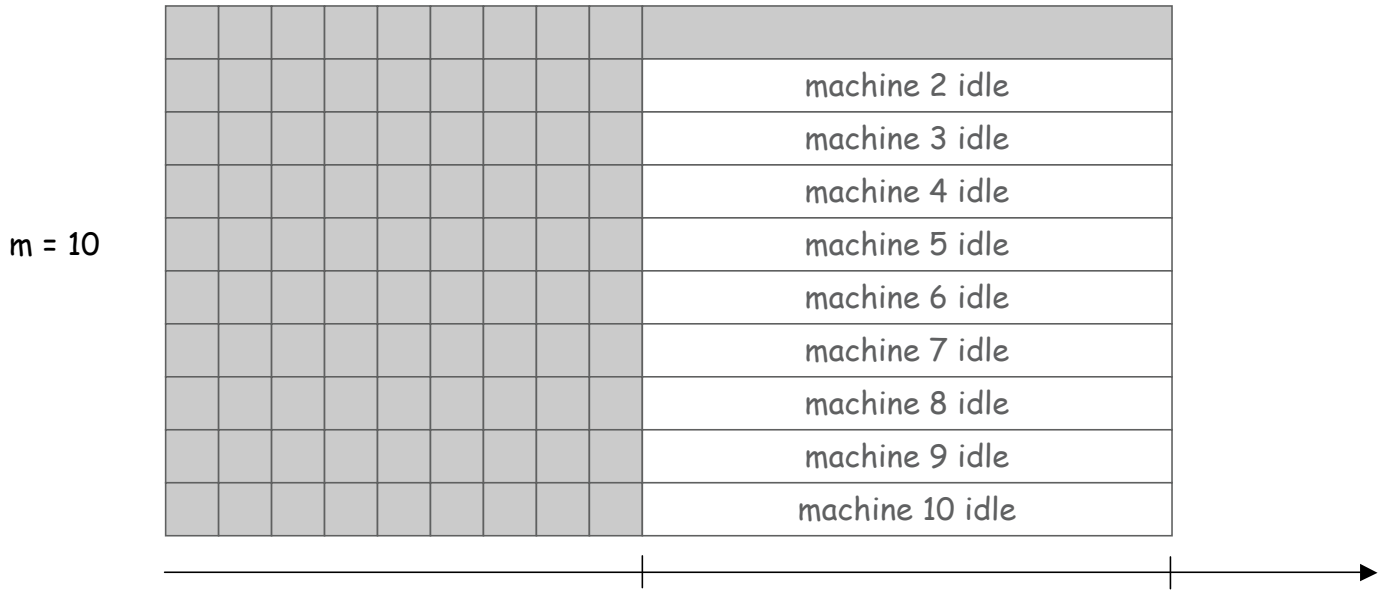
- Suppose we have m machines and $n = m(m-1)+1$ jobs
- The first $m(m-1) = n-1$ jobs each require $t_j = 1$
- The last job requires $t_n = m$

What does our greedy algorithm do?

- It evenly balances the first $n-1$ jobs
- Add the last giant job n to one of them

$\rightarrow$ The resulting makespan is $T = 2m-1$

Example: greedy solution



Example

What would the optimal solution look like?

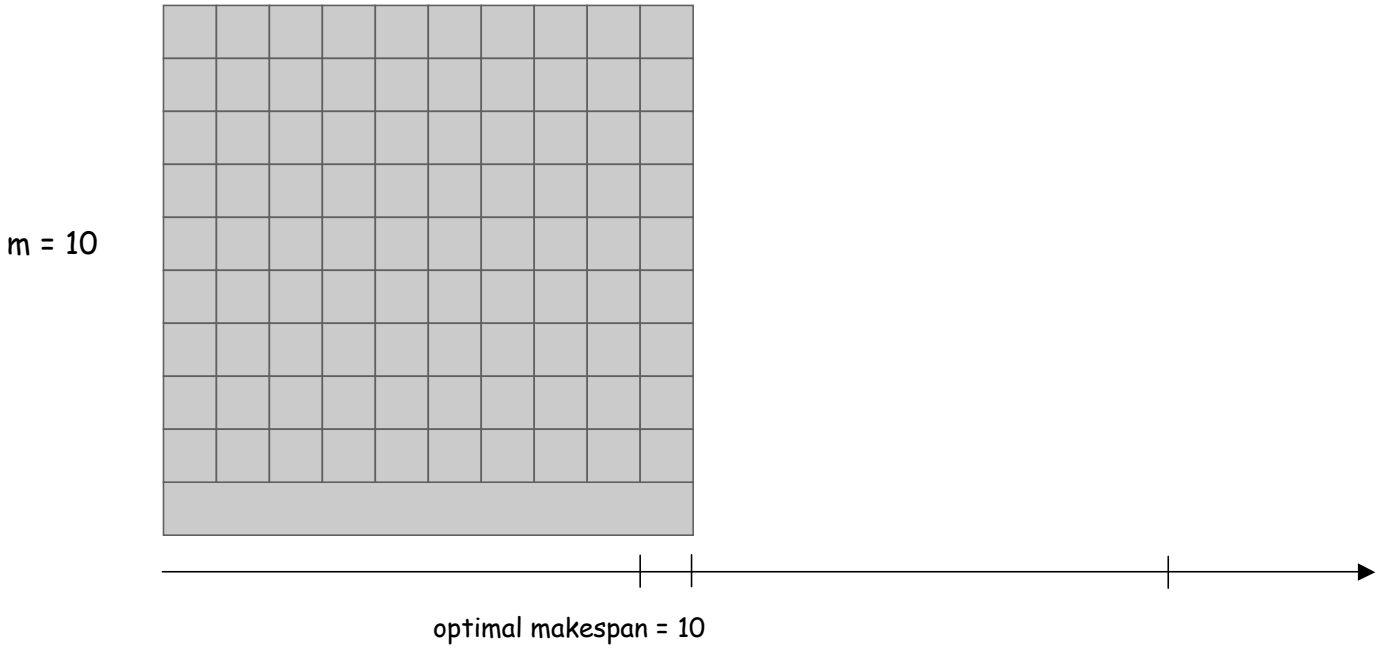
- It assigns the giant job n to one machine, say M_1
- It spreads evenly the remaining jobs on the other $m-1$ machines

$\rightarrow$ The resulting makespan is $T^* = m$

As a consequence the ratio between the greedy and optimal solution is:

$$\frac{(2m-1)}{m} = 2 - \frac{1}{m} \rightarrow 2 \text{ when } m \text{ is large}$$

Example: optimal solution



An improved approximation algorithm

- Can we do better, *i.e.*, guarantee that we're always within a factor of strictly less than 2 away from the optimum?
- Let's think about the previous example:
  - We spread everything evenly
  - A last giant job arrived and we had to compromise
- Intuitively, it looks like it would help to get the largest jobs arranged nicely first
- Smaller jobs can be arranged later, in any case they do not hurt much

An improved approximation algorithm

Algorithm 2: Sorted-Balance:

Start with no jobs assigned
Set $T_i = 0$ and $A(i) = \emptyset$ for all machines M_i
Sort jobs in decreasing order of processing time t_j
Assume that $t_1 \geq t_2 \geq \cdots \geq t_n$
for $j = 1 \cdots n$ **do**
    $i \leftarrow \arg\min_k T_k$
    $A(i) \leftarrow A(i) \cup \{j\}$
    $T_i \leftarrow T_i + t_j$
end

Analyzing the improved algorithm

Observation:

If we have fewer than m jobs, then everything gets arranged nicely, one job per machine, and the greedy is optimal

Lemma 3:

If we have more than m jobs, then $T^* \geq 2t_{m+1}$

Proof.

- Consider first $m+1$ jobs $t_1, ..., t_{m+1}$
- Since the t_i's are in descending order, each takes at least t_{m+1} time
- There are $m+1$ jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs

Analyzing the improved algorithm

Theorem

Sorted-balance algorithm is a 1.5-approximation, that is it produces an assignment of jobs to machines with a makespan $T \leq \frac{3}{2} T^*$

Proof.

- Let's assume (by the above observation and Lemma 3) that machine M_i has at least two jobs
- Let t_j be the last job assigned to the machine
- Note that $j \geq m + 1$, since the algo assigns the first m jobs to m distinct machines

$$T_i = (T_i - t_j) + t_j \leq \frac{3}{2} T^*$$

$$(T_i - t_j) \leq T^*$$

$$\text{Lemma 3} \rightarrow t_j \leq \frac{1}{2} T^*$$

The center selection problem

The Center Selection Problem

- As usual, let's start informally
- The center selection problem can also be related to the general task of allocating work across multiple servers
- The issue here, however, is where it is best to place the servers
- We keep the formulation simple here, and don't incorporate also the notion of load balancing in the problem
- As we will see, a simple greedy algorithm can approximate the optimal solution with a gap that can be arbitrarily bad, while a simple modification can lead to results that are always near optimal

The Center Selection Problem

The problem

- We have a set $S = \{s_1, s_2, \cdots, s_n\}$ of n sites to serve
- We have a set $C = \{c_1, c_2, \cdots, c_k\}$ of k centers to place

Select k centers C placement so that maximum distance form a site to the nearest center is minimized

The Center Selection Problem

Definition: distance

- We consider instances of the problem where the sites are points in the plane
- We define the distance as the Euclidean distance between points
- Any point in the plane can be a potential location of a center

Note that the algorithm we develop can be applied to a broader notion of distance, including:

- latency
- number of hops
- cost

The Center Selection Problem

Metric Space:

We allow any function that satisfies the following properties:

- $dist(s, s) = 0 \forall s \in S$
- **Symmetry**: $dist(s, z) = dist(z, s) \forall s, z \in S$
- **Triangle inequality**: $dist(s, z) + dist(z, h) \geq dist(s, h)$

The Center Selection Problem

Let's put down some assumptions

- Service points are served by the closest center:

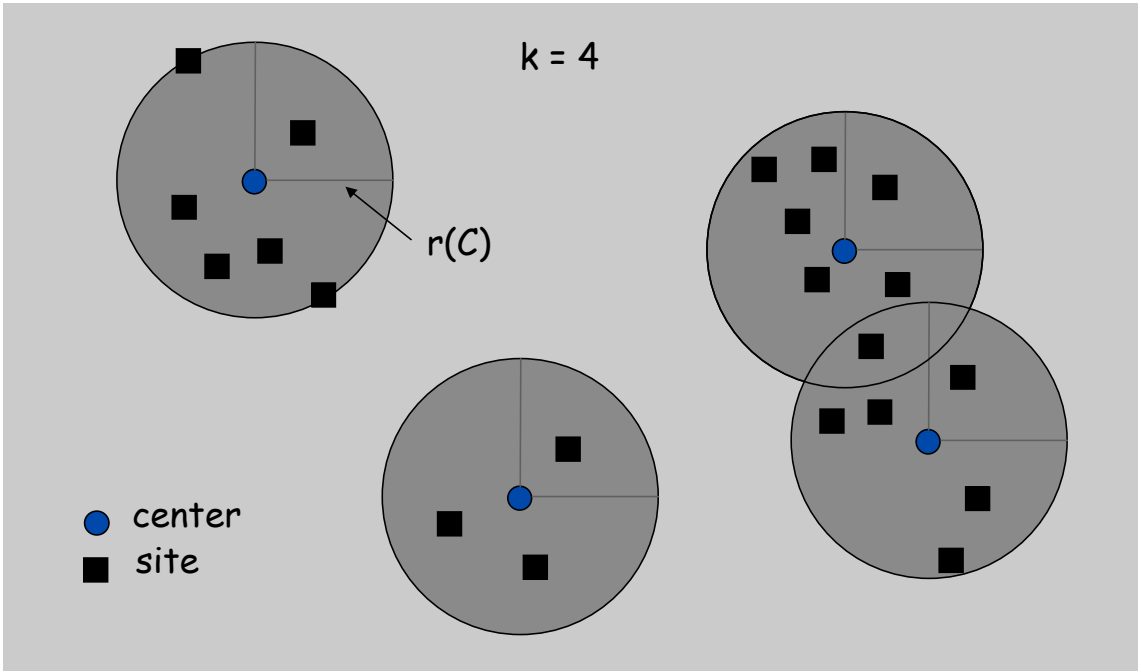
$$dist(s_i, C) = \min_{c \in C} dist(s_i, c)$$

- Covering radius:

$$r(C) = max_i dist(s_i, C)$$

- C forms a **cover** if $dist(s_i, C) \leq r(C) \forall s_i \in S$

Example



Designing the Algorithm

- Let's start with a simple greedy rule
- Consider an algorithm that selects centers one by one in a myopic fashion, without considering what happens to other centers
- Put the first center at the best possible location (for a single center)
- Keep adding centers so as to reduce the covering radius by as much as possible
- Done, once all the k centers have been placed

Example 1

Example where bad things can happen

- Consider only two sites s, z and $k = 2$
- Let $d = dist(s, z)$
- The algorithm would put the first center exactly at half-way:

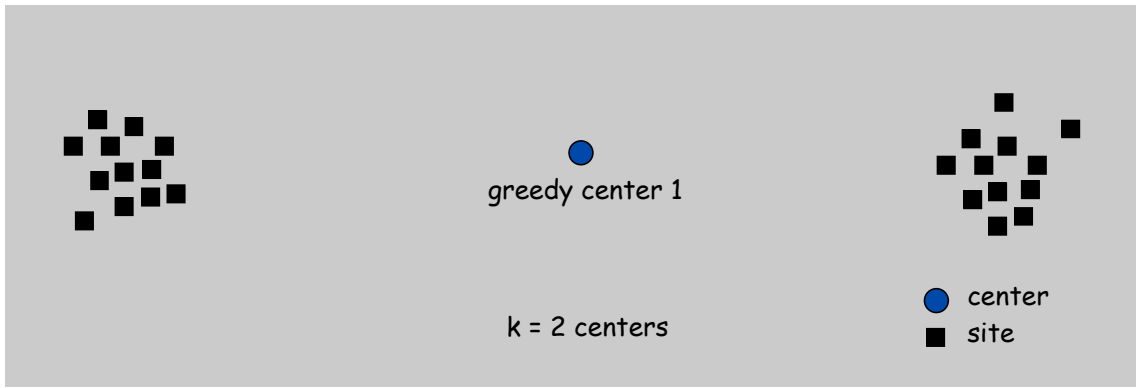
$$r(\{c_1\}) = d/2$$

- Now, we're stuck, and no matter what we do with the second center, the covering radius will always be $d/2$

Where is the optimal location to place the two centers?

Example 2

Here's another similar example, with clusters of sites



Designing the Algorithm

- Suppose for a minute that someone told us what the optimum covering radius r is
- That is, suppose we **know** there is a set of k centers C^* with $r(C^*) \leq r$
- Would this information help?
- Our job would be to find some set of k centers whose covering radius is not much more than r
- It turns out that finding the above set for $r(C) \leq 2r$ is easy

Designing the Algorithm

The idea is the following

- Consider any site $s \in S$
- There must be a center $c^* \in C^*$ that covers site s, with a distance at most r
- Take s as a center in our solution instead of c^* (we don't know where c^* is)
- We would like that our center could cover all the sites that c^* covers in the unknown solution C^*
- This is accomplished by expanding the radius from r to $2r$
  - Note that this is true because of the triangle inequality
  - All the sites that were at distance at most r from c^* are at distance at most $2r$ from s

Designing the Algorithm

Algorithm 3: Greedy-Center placement

S' is the set of sites that still need to be covered
Initialize $S' = S$
Let $C = \emptyset$
while $S' \neq \emptyset$ **do**
  Select any site $s \in S'$ and add s to C
  Delete all sites from S' that are at distance at most $2r$ from s
end
if $|C| \leq k$ **then**
  Return C as the selected set of sites
else
  Claim that there is no set of k centers with covering radius at most r
end

Designing the Algorithm

Proposition

Any set of centers C returned by the Greedy-algorithm above has covering radius $r(C) \leq 2r$

Proposition

Suppose the Greedy-algorithm above selects more than k centers. Then, for any set C^* of size at most k, the covering radius is $r(C^*) > r$

Question

What about designing an algorithm for the center selection problem without knowing in advance the optimal covering radius?

Discussion

Methodology 1

- Assume that you know the value achieved by an optimal solution
- Design your algorithm under this assumption and then convert it into one that achieves a comparable performance guarantee
- Basically, try out a range of “guesses” as to what the optimal solution value might be
- Over the course of your algorithm, this sequence of guesses gets more and more accurate, until an approximate solution is reached

Discussion

Methodology 1: application to the center selection problem

- We know that the optimal solution is larger than 0 and smaller than $r_{max} = \max dist(s_i, s_j) \forall s_i, s_j \in S$
- Let $r = r_{max}/2$

1. The Greedy-algorithm above tells us there is a set of k centers with a covering radius at most $2r$
2. The Greedy-algorithm above terminates with a negative answer
   - In case 1) we can lower our initial guess of the optimal radius
   - In case 2) we have to raise our initial guess

$\rightarrow$ We can do "binary-search" on the radius, and stop when our estimate gets close enough

Discussion

- We saw a general technique that can be used when dropping the assumption of known optimal solution
- Next, we look at a simple greedy algorithm that approximate well the optimal solution without requiring it to be known in advance and without using the previous general methodology

Designing the Algorithm

A greedy algorithm that works

Repeatedly choose the next center to be the site **farthest** from any existing center

---

Algorithm 4: Greedy-farthest

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

Analyzing the Algorithm

Observation:

Upon termination all centers in C are pairwise at least $r(C)$ apart

Theorem:

- Let C^* be an optimal set of centers
- Then the covering radius achieved by the greedy algorithm satisfies $r(C) \leq 2r(C^*)$

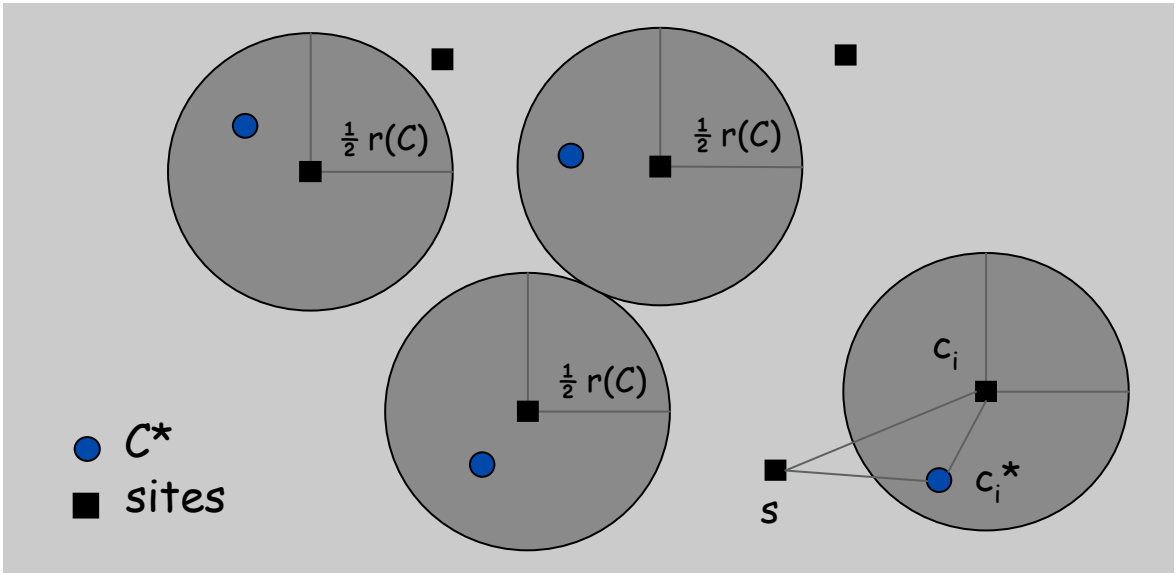
Analyzing the Algorithm

Proof.

- By contradiction, assume $r(C^*) < 1/2r(C)$
- For each site $c_i \in C$, consider a ball of radius $1/2r(C)$ around it
- Exactly one c_i^* in each ball; let c_i be the site paired with c_i^*
- Consider any site s and its closest center $c_i^* \in C^*$
- $dist(s, C) \leq dist(s, c_i) \leq dist(s, c_i^*) + dist(c_i^*, c_i) \leq 2r(C*)$
  - First inequality, derives from triangle inequality
  - The two terms of the triangle inequality $\leq r(C^*)$ since c_i^* is the closest center
- Thus $r(C) \leq 2r(C*)$

Analyzing the Algorithm

The proof in images



Analyzing the Algorithm

Theorem

Greedy algorithm is a 2-approximation for center selection problem

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

Question

Is there hope of a 3/2-approximation? 4/3?

Theorem

Unless P = NP, there no α-approximation for center-selection problem for any $\alpha < 2$

The pricing method

The Pricing Method

- We now turn to our second technique for designing approximation algorithms: the **pricing method**
- We already outlined the Vertex Cover Problem, and its related parent, the Set Cover Problem

Note:

- We being the section with a general discussion on how to use **reductions** in the design of approximation algorithms
- What are reductions?

The Vertex Cover Problem

The Vertex Cover Problem

- You are given a graph $G = (V, E)$
- A set $S \subseteq V$ is a **vertex cover** if each edge $e \in E$ has at least one end in S

The Weighted Vertex Cover Problem

- You are given a graph $G = (V, E)$
- Each $v \in V$ has a **weight** $w_i \geq 0$
- The weight of a set S of vertices is denoted $w(S) = \sum_{i \in S} w_i$
- A set $S \subseteq V$ is a **vertex cover** if each edge $e \in E$ has at least one end in S
- Find a vertex cover S of minimum weight $w(S)$

Discussion

Unweighted Vertex Problem

When all weights in the weighted vertex problem are equal to 1, deciding if a vertex cover of weight at most k is the **standard decision version** of the Vertex Cover

- Vertex Cover is easily **reducible** to Set Cover
- There is an approximation algorithm to the Set Cover (not seen in class)

$\rightarrow$ What does this imply about the approximability of the Vertex Cover?

Discussion

Polynomial-time Reductions

We will outline some of the subtle ways in which approximation results interact with this technique, which indicates ways of reducing hard problems to polynomial-time problems

- Consider an unweighted vertex cover problem: we look at a vertex cover of minimum size
- We can show that Set Cover is NP-complete using a **reduction from the decision version** of unweighted Vertex Cover

$$\text{Vertex Cover} \leq_P \text{Set Cover}$$

Discussion

Polynomial-time Reduction

$$\text{Vertex Cover} \leq_P \text{Set Cover}$$

If we had a polynomial time algorithm to solve the Set Cover Problem, then we could use this algorithm to solve the Vertex Cover Problem in polynomial time

$\rightarrow$ Now, since we know the Vertex Cover is NP-complete, it is impossible to use a poly-time algo for the Set Cover to solve it, hence there is no poly-time algo to solve the Set Cover as well

Discussion

- Now, we know that a polynomial-time approximation algorithm for the Set Cover exists

Does this imply that we can use it to formulate an approximation algorithm for the Vertex Cover?

Proposition:

One can use the Set Cover approximation algorithm to give an approximation algorithm for the weighted Vertex Cover Problem

Discussion

Cautionary example:

It is possible to use the Independent Set Problem to prove that the Vertex Cover Problem is NP-Complete:

$$\text{Independent Set} \leq_P \text{Vertex Cover}$$

If we had a polynomial-time algorithm that solves the Vertex Cover Problem, then we could use this algorithm to solve the Independent Set Problem in polynomial time

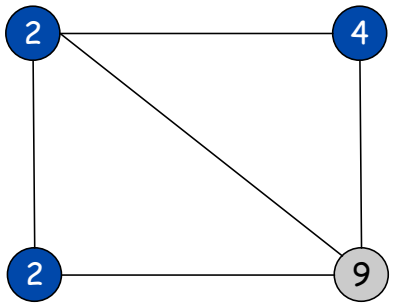
- Can we use this reduction to say we can use an approximation algorithm for the minimum-size vertex cover to design a comparably good approximation for the maximum-size independent set?

$\rightarrow$ NO!

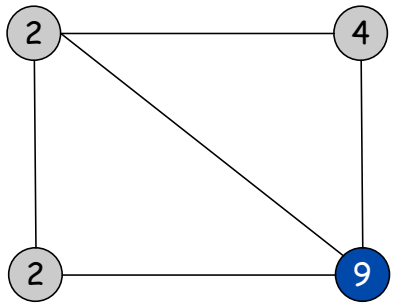
Example of Vertex Cover Problem

Recall:

Weighted vertex cover: Given a graph G with vertex weights, find a vertex cover of minimum weight



weight = 2 + 2 + 4



weight = 9

The Pricing Method

- Also known as the **Primal-Dual method**
- Motivated by an economic perspective
- For the case of the Vertex Cover problem:
  - We will think of the weights on the nodes as **costs**
  - We will think of each edge as having to pay for its "share" of the costs of the vertex cover we find
- An edge is seen as an independent "agent" who is willing to "pay" something to the node that covers it.

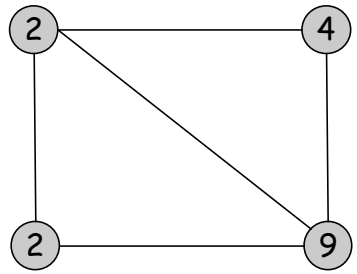
Our algorithm will find a vertex cover S and determine the prices $p_e \geq 0$ for each edge $e \in E$ so that if each edge pays p_e, this will in total approximately cover the cost of S

The 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, that is:

$$\sum_{e=(i,j)} p_e \leq w_i$$

for each vertex i : $\sum_{e=(i,j)} p_e \leq w_i$



The Pricing Method

Lemma ("The Fairness Lemma"):

For any vertex cover S and any fair prices p_e, we have:

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

Proof.

- The following holds for each edge covered by at least one node in S:

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

- Now, if we sum the fairness inequalities for each node in S, we have that:

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

$\rightarrow$ We have our claim: $\sum_{e \in E} p_e \leq w(S)$

The Approximation Algorithm

Definition: tightness

A node is tight (or "paid for") if $\sum_{e=(i,j)} p_e = w_i$

Algorithm 5: Vertex-Cover Approx (G, w)

Set $p_e = 0 \, \forall e \in E$

while $\exists \, e = (i, j)$ *such that neither i nor j are* ***tight*** **do**

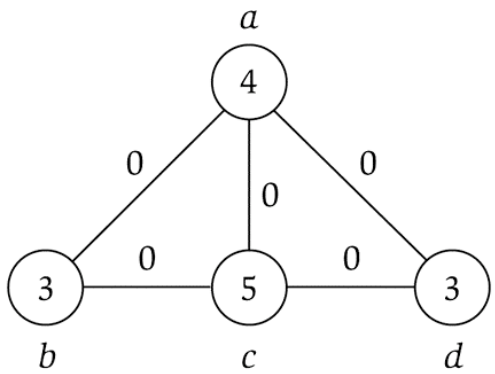
  Select such an edge e

  Increase p_e without violating fairness

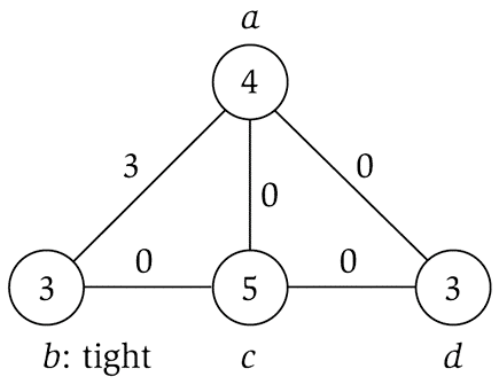
end

$S \leftarrow$ set of all **tight** nodes

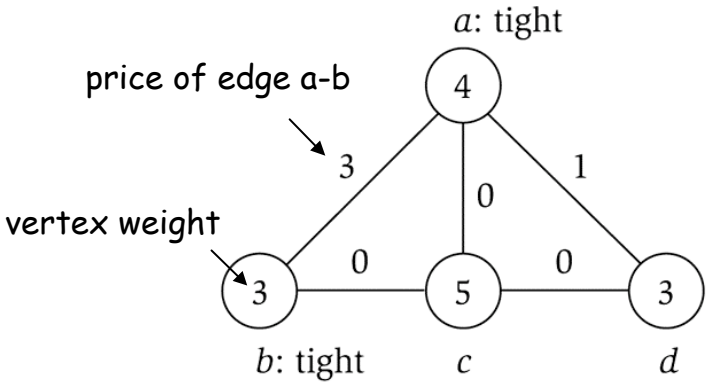
The Approximation Algorithm



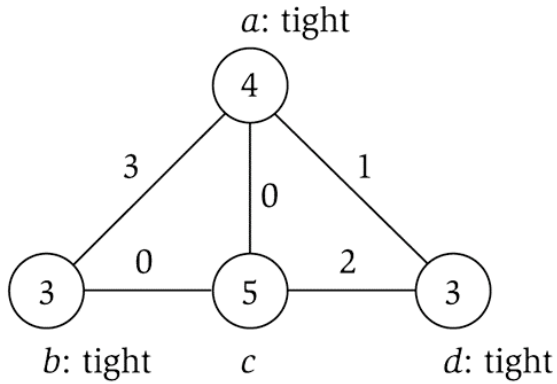
(a)



(b)



(c)



(d)

Analyzing the Algorithm

Theorem

The Pricing method for the weighted Vertex Cover Problem is a 2-approximation

Proof.

- Algorithm terminates since at least one new node becomes tight after each iteration of while loop
- Let $S =$ set of all tight nodes upon termination of algorithm
- S is a vertex cover
  - If some edge $e = (i, j)$ is uncovered, then neither i nor j is tight
  - But then while loop would not terminate

Analyzing the Algorithm

Proof.

- Let S^* be optimal vertex cover
- We show that $w(S) \leq 2w(S^*)$. Indeed:
  - *since all nodes in S are tight*:

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

  - *since* $S \subseteq V$ *and* $p_e \geq 0$:

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

  - *since each edge is counted twice*:

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

Analyzing the Algorithm

Proof.

- *since each edge is counted twice*:

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

- *From the fairness Lemma*:

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

Linear Programming and Rounding

Linear Programming and Rounding

- We now look at a third technique used to design approximation algorithms
- This method derives from **operation research**: linear programming (LP)
- Linear programming is the subject of entire courses, here we'll just give a crash introduction to the subject
- Our goal is to show how LP can be used to approximate NP-hard optimization problems

LP as a General Technique

- Recall, from **linear algebra**, the problem of a system of equations
- Using a matrix-vector notation, we have a vector x of unknown real numbers, a given matrix A, and a given vector b
- The goal is to solve:

$$Ax = b$$

Gaussian elimination is a well-known efficient algorithm for this problem

LP as a General Technique

- The basic LP problem can be viewed as a more complex version of this, with inequalities in place of equations
- The goal is to determine a vector x that satisfies:

$$Ax \geq b$$

- Each coordinate of the vector Ax should be greater than or equal to the corresponding coordinate of the vector b
- Such system of inequalities define regions in the space

LP as a General Technique

Example

- Suppose $x = (x_1, x_2)$
- Our system of inequalities is:

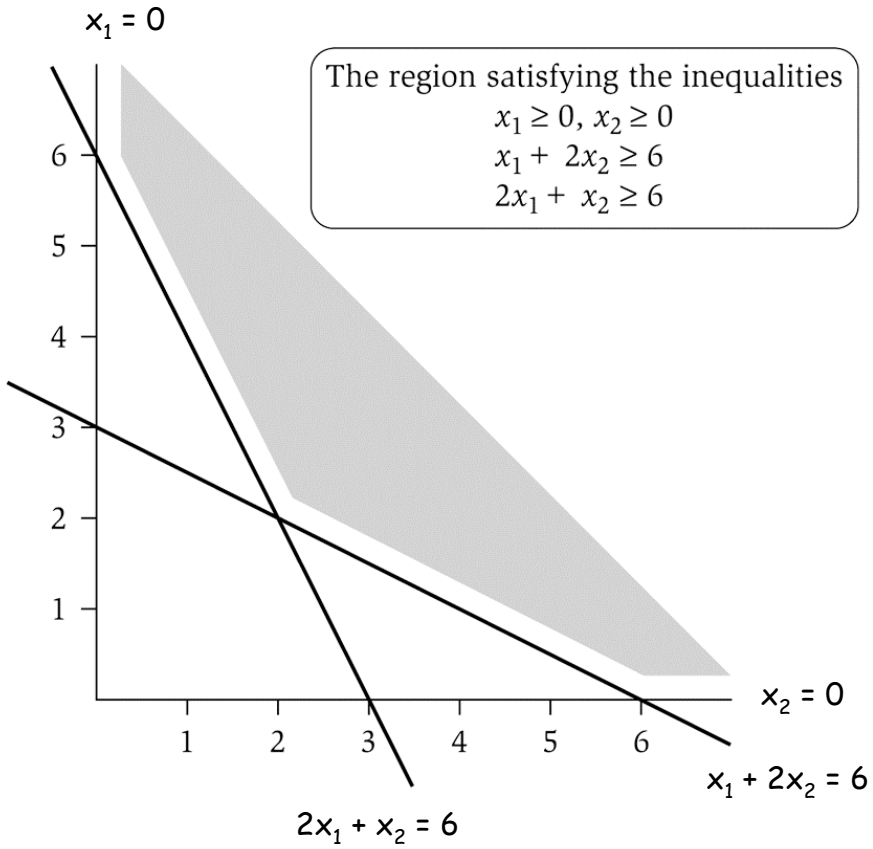
$$x_1 \geq 0\,,\; x_2 \geq 0$$

$$x_1 + x_2 \geq 6$$

$$2x_1 + x_2 \geq 6$$

LP as a General Technique

The example tells that the set of solutions is in the region in the plane shown below



LP as a General Technique

- Given a region defined by $Ax + b$, LP seeks to minimize a linear combination of the coordinates of x, for all x belonging to the region defined by the set of inequalities
- Such a linear combination, called **objective function**, can be rewritten as $c^t x$, where c is a vector of coefficients, and $c^t x$ denotes the inner product of two vectors.

LP in Standard Form

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

$$\min c^t x$$

$$x \geq 0$$

$$Ax \geq b$$

LP as a General Technique

Example: continued...

- Assume we have $c = (1.5, 1)$
- The objective function is $1.5x_1 + x_2$
- This function should be minimized over the region defined by $Ax \geq b$
- The solution would be to choose the point $x = (2, 2)$, where the two slanting lines cross, which yields a value of $c^t x = 5$

LP as a General Technique

LP as a Decision Problem

- Given a matrix A, vectors b and c, and a bound γ, does there exist x so that $x \geq 0$, $Ax \geq b$, and $c^t x \leq \gamma$?

Computational complexity of LP

- The decision version of LP is NP
- Historically, several methods to solve such problems have been developed:
  - **Interior methods**: practical poly-time algorithms
  - **The Simplex method**: practical method that competes with poly-time algorithms
  - ...

How can linear programming help us when we want to solve combinatorial problems such as Vertex Cover?

Vertex Cover as an Integer Program

The Weighted Vertex Cover Problem

- You are given a graph $G = (V, E)$
- Each $v \in V$ has a **weight** $w_i \geq 0$
- The weight of a set S of vertices is denoted $w(S) = \sum_{i \in S} w_i$
- A set $S \subseteq V$ is a **vertex cover** if each edge $e \in E$ has at least one end in S
- Find a vertex cover S of minimum weight $w(S)$

Vertex Cover as an Integer Program

- We now try to formulate a linear program that is in close correspondence with the Vertex Cover problem
- LP is based on the use of **vectors of variables**
- We use a **decision variable** x_i for each node $i \in V$ to model the choice of whether to include node i in the vertex cover: $x_i = 0 \Rightarrow i \notin S$ and $x_i = 1 \Rightarrow i \in S$
- We can now create a n-dimensional vector x of decision variables

Vertex Cover as an Integer Program

How do we proceed now?

- We use linear inequalities to encode the requirement that the selected nodes form a vertex cover
- We use the objective function to encode the goal of minimizing the total weight
- For each edge $(i, j) \in E$, it must have one end in the vertex cover $\Rightarrow x_i + x_j \geq 1$ ("whether one, the other or both ends in the cover are ok")
- We write the set of node weights as a n-dimensional vector w, and seek to minimize $w^t x$

Vertex Cover as an Integer Program

VC.IP

$$\min \sum_{i \in V} w_i x_i$$

$$\text{s.t. } x_i + x_j \geq 1 \ (i,j) \in E$$

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

Proposition:

- *S is a vertex cover in G iff the vector x, defined as* $x_i = 0 \Rightarrow i \notin S$ *and* $x_i = 1 \Rightarrow i \in S$*, satisfies the constraints in VC.IP*
- Furthermore, we have $w(S) = w^t x$

Vertex Cover as an Integer Program

We can now put this system into the matrix form discussed before

- Define a matrix A whose columns are the nodes in V, and whose rows are the edges in E: $A[e, i] = 1$ if node i is an end of the edge e, and $A[e, i] = 0$ otherwise ("Each row has exactly two non-zero entries")
- The system of inequalities can be rewritten as:

$$Ax \geq \mathbf{1}$$

$$\mathbf{0} \leq x \leq \mathbf{1}$$

Vertex Cover as an Integer Program

- Keep in mid that we crucially have required that all coordinates in the solution be either 0 or 1

$\rightarrow$ This is an instance of an **Integer Program**

- Instead, in linear programs, the coordinates can be arbitrary real numbers

Integer Programming

- Integer Programming (IP) is considerably harder than LP
- Our discussion really constitutes a reduction from the Vertex Cover to the decision version of IP

$$\textit{Vertex Cover} \leq_P \textit{Integer Programming}$$

Using Linear Programming for Vertex Cover

- Trying to solve the IP problem (VC.IP) optimally is clearly not the right way to go, as this is NP-hard
- We thus exploit the fact that LP is not as hard as IP

LP version of (VC.IP)

- We drop the requirement that $x_i \in \{0, 1\}$ and assume $x_i \in \Re\{0, 1\}$
- This gives us an instance of the problem we call (VC.LP), and we can solve it in polynomial time

Using Linear Programming for Vertex Cover

(VC.LP)

- Find a set of real values $\{x_i^*\} \in \{0, 1\}$
- Subject to $x_i^* + x_j^* \geq 1 \ \forall e = (i, j) \in E$
- The goal is to minimize $\sum_i w_i x_i^*$

- Let x^* denote the solution vector
- Let $w_{LP} = w^t x^*$

Using Linear Programming for Vertex Cover

Proposition

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

Proof.

- Vertex Cover of G corresponds to integer solutions of (VC.IP)
- We have that the minimum $(\min w^t x : \mathbf{0} \leq x \leq \mathbf{1},\ Ax \geq \mathbf{1})$ over all integer x vectors is exactly the minimum-weight vertex cover
- To get the minimum of the linear program (VC.LP), we allow x to take arbitrary real-number values and so the minimum of (VC.LP) is no larger than that of (VC.IP)

Using Linear Programming for Vertex Cover

Note:

The previous proposition is one of the crucial ingredient we need for an approximation algorithm: a good **lower bound** on the optimum, in the form of the efficiently computable quantity w_{LP}

- Note that w_{LP} can be definitively smaller than $w(S^*)$

Example

- If the graph G is a triangle and all weights are 1, then the minimum vertex cover has a weight of 2
- But, in a LP solution, we can set $x_i = 1/2$ for all three vertices, and so get a solution with weight 3/2

Using Linear Programming for Vertex Cover

Question

How can solving the LP help us actually **find** a near-optimal vertex cover?

- The idea is to work with the values x_i^* and to infer a vertex cover S from them
- If we have integral values for x_i^*, then there is no problem: $x_i^* = 0$ implies that node i is not in the cover, $x_i^* = 1$ implies that node i is in the cover
- What to do with the fractional values in between? The natural approach is to **round**

Rounding

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

Using Linear Programming for Vertex Cover

Proposition:

The set S defined as $S = \{i \in V : x_i^* \geq 1/2\}$ (rounding) is a vertex cover, and $w(S) \leq 2w_{LP}$

S is a vertex cover.

- Consider and edge $e = (i, j) \in E$
- Since $x_i^* + x_j^* \geq 1$ and $x_i^* \geq 1/2$ or $x_j^* \geq 1/2$

$\Rightarrow$ (i, j) is covered

Using Linear Programming for Vertex Cover

Proposition:

The set S defined as $S = \{i \in V : x_i^* \geq 1/2\}$ (rounding) is a vertex cover, and $w(S) \leq 2w_{LP}$

S has desired cost.

- Let S^* be the optimal vertex cover
- Since LP is a relaxation:

$$\sum_{i \in S^*} w_i \geq \sum_{i \in S} w_i x_i^*$$

- Since $x_i^* \geq 1/2$, we have

$$\sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i$$