CSE 431/531: Analysis of Algorithms

Approximation and Randomized Algorithms

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Outline

Approximation Algorithms

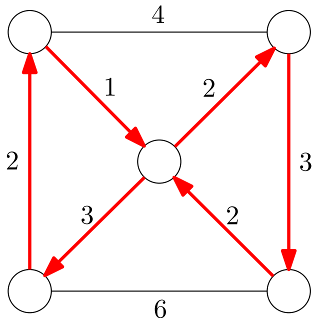
An algorithm for an optimization problem is an α-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an α-factor of the cost (or value) of the optimum solution.

- opt: cost (or value) of the optimum solution
- sol: cost (or value) of the solution produced by the algorithm
- α: approximation ratio
- For minimization problems:
  - $\alpha \geq 1$ and we require sol $\leq \alpha \cdot$ opt
- For maximization problems, there are two conventions:
  - $\alpha \leq 1$ and we require sol $\geq \alpha \cdot$ opt
  - $\alpha \geq 1$ and we require sol $\geq$ opt$/\alpha$

Outline

Recall: Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



Travelling Salesman Problem (TSP)

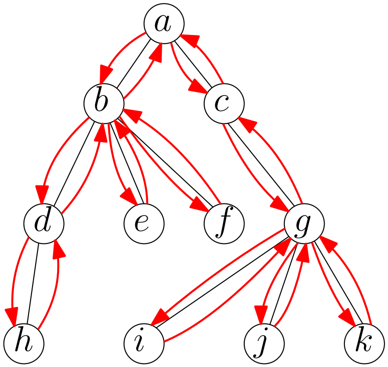
Input: a graph $G = (V, E)$, weights $w : E \to \mathbb{R}_{\geq 0}$

Output: a traveling-salesman tour with the minimum cost

2-Approximation Algorithm for TSP

TSP1(G, w)

1. $MST \leftarrow$ the minimum spanning tree of G w.r.t weights w, returned by either Kruskal's algorithm or Prim's algorithm.
2. Output tour formed by making two copies of each edge in MST.



2-Approximation Algorithm for TSP

Lemma Algorithm TSP1 is a 2-approximation algorithm for TSP.

Proof

- mst $=$ cost of the minimum spanning tree
- tsp $=$ cost of the optimum travelling salesman tour
- then mst $\leq$ tsp, since removing one edge from the optimum travelling salesman tour results in a spanning tree
- sol $=$ cost of tour given by algorithm TSP1
- sol $= 2 \cdot$ mst $\leq 2 \cdot$ tsp. $\square$

1.5-Approximation for TSP

Def. Given $G = (V, E)$, a set $U \subseteq V$ of even number of vertices in V, a matching M over U in G is a set of $|U|/2$ paths in G, such that every vertex in U is one end point of some path.

Def. The cost of the matching M, denoted as $\text{cost}(M)$ is the total cost of all edges in the $|U|/2$ paths (counting multiplicities).

Theorem Given $G = (V, E)$, a set $U \subseteq V$ of even number of verticies, the minimum cost matching over U in G can be found in polynomial time.

1.5-Approximation for TSP

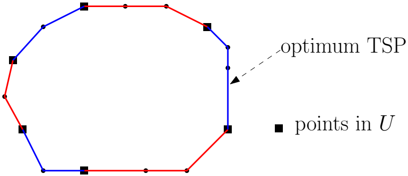
Lemma Let T be a spanning tree of $G = (V, E)$; let U be the set of odd-degree vertices in MST ($|U|$ must be even, why?). Let M be a matching over U, then, $T \uplus M$ gives a traveling salesman's tour.

Proof.

Every vertex in $T \uplus M$ has even degree and $T \uplus M$ is connected (since it contains the spanning tree). Thus $T \uplus M$ is an Eulerian graph and we can find a tour that visits every edge in $T \uplus M$ exactly once. $\square$

1.5-Approximation for TSP

Lemma Let U be a set of even number of vertices in G. Then the cost of the cheapest matching over U in G is at most $\frac{1}{2}$tsp.



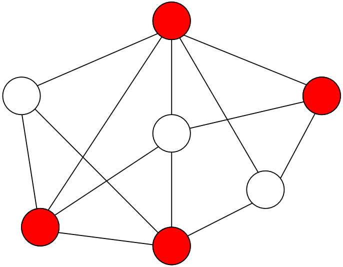
Proof.

- Take the optimum TSP
- Breaking into read matching and blue matching over U
- cost(blue matching)$+$ cost(red matching) $=$ tsp
- Thus, cost(blue matching) $\leq \frac{1}{2}$tsp or cost(red matching) $\leq \frac{1}{2}$tsp
- cost(cheapeast matching) $\leq \frac{1}{2}$tsp $\square$

Outline

Vertex Cover Problem

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



Vertex-Cover Problem

Input: $G = (V, E)$

Output: a vertex cover S with minimum $|S|$

First Try: Greedy Algorithm

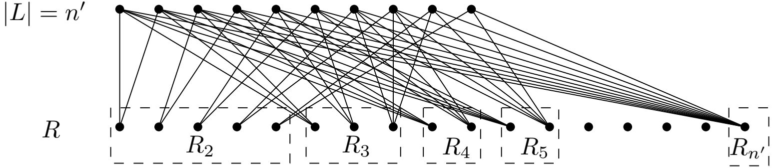
Greedy Algorithm for Vertex-Cover

1. $E' \leftarrow E, S \leftarrow \emptyset$
2. while $E' \neq \emptyset$
3. let v be the vertex of the maximum degree in (V, E')
4. $S \leftarrow S \cup \{v\}$,
5. remove all edges incident to v from E'
6. output S

Theorem Greedy algorithm is an $O(\lg n)$-approximation for vertex-cover.

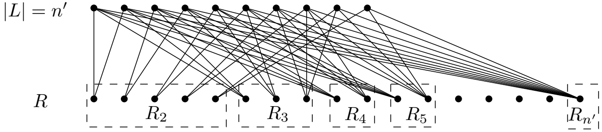
- We are not going to prove the theorem
- Instead, we show that the $O(\lg n)$-approximation ratio is tight for the algorithm

Bad Example for Greedy Algorithm



- L: n' vertices
- R_2: $\lfloor n'/2 \rfloor$ vertices, each connected to 2 vertices in L
- R_3: $\lfloor n'/3 \rfloor$ vertices, each connected to 3 vertices in L
- R_4: $\lfloor n'/4 \rfloor$ vertices, each connected to 4 vertices in L
- $\cdots$
- $R_{n'}$: 1 vertex, connected to n' vertices in L
- $R = R_2 \cup R_3 \cup \cdots \cup R_{n'}$

Bad Example for Greedy Algorithm

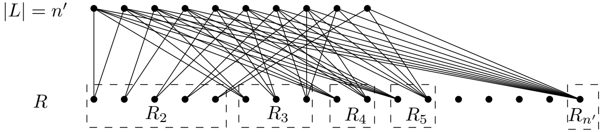


- Optimum solution is L, where $|L| = n'$
- Greedy algorithm picks $R_{n'}, R_{n'-1}, \cdots, R_2$ in this order
- Thus, greedy algorithm outputs R

$$\begin{aligned}|R| &= \sum_{i=2}^{n} \left\lfloor \frac{n'}{i} \right\rfloor \geq \sum_{i=1}^{n} \frac{n'}{i} - n' - (n'-1) \\ &= n' H(n') - (2n'-1) = \Omega(n' \lg n')\end{aligned}$$

- where $H(n') = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n'} = \Theta(\lg n')$ is the n'-th number in the harmonic sequence.

Bad Example for Greedy Algorithm



- Let $n = |L \cup R| = \Theta(n' \lg n')$
- Then $\lg n = \Theta(\lg n')$
- $\frac{|R|}{|L|} = \frac{\Omega(n' \lg n')}{n'} = \Omega(\lg n') = \Omega(\lg n)$.
- Thus, greedy algorithm does not do better than $O(\lg n)$.

- Greedy algorithm is a very natural algorithm, which might be the first algorithm some one can come up with
- However, the approximation ratio is not so good
- We now give a somewhat "counter-intuitive" algorithm,
- for which we can prove a 2-approximation ratio.

2-Approximation Algorithm for Vertex Cover

1. $E' \leftarrow E, S \leftarrow \emptyset$
2. while $E' \neq \emptyset$
3.   let (u, v) be any edge in E'
4. $S \leftarrow S \cup \{u, v\}$,
5.   remove all edges incident to u and v from E'
6. output S

- The counter-intuitive part: adding both u and v to S seems to be wasteful
- Intuition for the 2-approximation ratio: the optimum solution must cover the edge (u, v), using either u or v. If we select both, we are always ahead of the optimum solution. The approximation factor we lost is at most 2.

2-Approximation Algorithm for Vertex Cover

1. $E' \leftarrow E, S \leftarrow \emptyset$
2. while $E' \neq \emptyset$
3.   let (u, v) be any edge in E'
4. $S \leftarrow S \cup \{u, v\}$,
5.   remove all edges incident to u and v from E'
6. output S

- Let E^* be the set of edges (u, v) considered in Statement 3
- Observation: E^* is a matching and $|S| = 2|E^*|$
- To cover all edges in E^*, the optimum solution needs $|E^*|$ vertices

Theorem The algorithm is a 2-approximation algorithm for vertex-cover.

Outline

Max 3-SAT

Input: n boolean variables $x_1, x_2, \cdots, x_n$

m clauses, each clause is a disjunction of 3 literals from 3 distinct variables

Output: an assignment so as to satisfy as many clauses as possible

Example:

- clauses: $x_2 \vee \neg x_3 \vee \neg x_4, \quad x_2 \vee x_3 \vee \neg x_4,$ $\neg x_1 \vee x_2 \vee x_4, \quad x_1 \vee \neg x_2 \vee x_3, \quad \neg x_1 \vee \neg x_2 \vee \neg x_4$
- We can satisfy all the 5 clauses: $x = (1, 1, 1, 0, 1)$

Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability $1/2$, independent of other variables

Lemma Let m be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

Proof.

- for each clause C_j, let $Z_j = 1$ if C_j is satisfied and 0 otherwise
- $Z = \sum_{j=1}^{m} Z_j$ is the total number of satisfied clauses
- $\mathbb{E}[Z_j] = 7/8$: out of 8 possible assignments to the 3 variables in C_j, 7 of them will make C_j satisfied
- $\mathbb{E}[Z] = \mathbb{E}\left[\sum_{j=1}^{m} Z_j\right] = \sum_{j=1}^{m} \mathbb{E}[Z_j] = \sum_{j=1}^{m} \frac{7}{8} = \frac{7}{8}m$, by linearity of expectation. $\square$

Randomized Algorithm for Max 3-SAT

Lemma Let m be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

- Since the optimum solution can satisfy at most m clauses, lemma gives a randomized $7/8$-approximation for Max-3-SAT.

Theorem ([Hastad 97]) Unless $\mathrm{P} = \mathrm{NP}$, there is no ρ-approximation algorithm for MAX-3-SAT for any $\rho > 7/8$.

Outline

Outline

Quicksort vs Merge-Sort

	Merge Sort	**Quicksort**
Divide	Trivial	Separate small and big numbers
Conquer	Recurse	Recurse
Combine	Merge 2 sorted arrays	Trivial

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

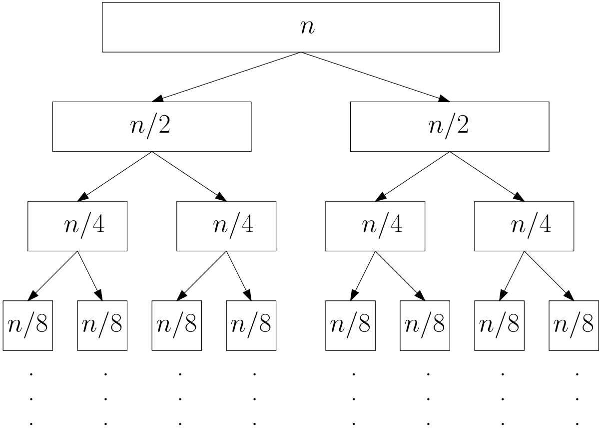
25	15	17	29	38	45	37	64	82	75	94	92	69	76	85

Quicksort

quicksort(A, n)

if $n \leq 1$ then return A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x \\ Divide
$A_R \leftarrow$ elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

- Recurrence $T(n) \leq 2T(n/2) + O(n)$
- Running time $= O(n \lg n)$



- Each level has total running time $O(n)$
- Number of levels $= O(\lg n)$
- Total running time $= O(n \lg n)$

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

$i\ j$

17	37	15	29	38	45	25	64	69	76	94	92	75	82	85

- To partition the array into two parts, we only need $O(1)$ extra space.

Outline

Randomized Quicksort Algorithm

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ a random element of A (x is called a pivot)
3. $A_L \leftarrow$ elements in A that are less than x \\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

Variant of Randomized Quicksort Algorithm

quicksort(A, n)

```
if n ≤ 1 then return A
repeat
    x ← a random element of A (x is called a pivot)
    A_L ← elements in A that are less than x            \\ Divide
    A_R ← elements in A that are greater than x         \\ Divide
unitl A_L.size ≤ 3n/4 and A_R.size ≤ 3n/4
B_L ← quicksort(A_L, A_L.size)                          \\ Conquer
B_R ← quicksort(A_R, A_R.size)                          \\ Conquer
t ← number of times x appear A
return the array obtained by concatenating B_L, the array
    containing t copies of x, and B_R
```

Analysis of Variant

3. $x \leftarrow$ a random element of A
4. $A_L \leftarrow$ elements in A that are less than x
5. $A_R \leftarrow$ elements in A that are greater than x

Q: What is the probability that A_L.size $\leq 3n/4$ and A_R.size $\leq 3n/4$?

A: At least $1/2$

Analysis of Variant

2. repeat
3. $\quad x \leftarrow$ a random element of A
4. $\quad A_L \leftarrow$ elements in A that are less than x
5. $\quad A_R \leftarrow$ elements in A that are greater than x
6. unitl A_L.size $\leq 3n/4$ and A_R.size $\leq 3n/4$

Q: What is the expected number of iterations the above procedure takes?

A: At most 2

- Suppose an experiment succeeds with probability $p \in (0, 1]$, independent of all previous experiments.

1. repeat
2.   run an experiment
3. until the experiment succeeds

Lemma The expected number of experiments we run in the above procedure is $1/p$.

Fact For $q \in (0,1)$, we have $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$.

Lemma The expected number of experiments we run in the above procedure is $1/p$.

Proof

$$\begin{aligned}
\text{Expectation} &= p + (1-p)p \times 2 + (1-p)^2 p \times 3 + (1-p)^3 p \times 4 \\
&\quad + \cdots \\
&= p \sum_{i=1}^{\infty} (1-p)^{i-1} i \quad = \quad p \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} (1-p)^{i-1} \\
&= p \sum_{j=1}^{\infty} (1-p)^{j-1} \frac{1}{1-(1-p)} \quad = \quad \sum_{j=1}^{\infty} (1-p)^{j-1} \\
&= (1-p)^0 \frac{1}{1-(1-p)} = 1/p
\end{aligned}$$

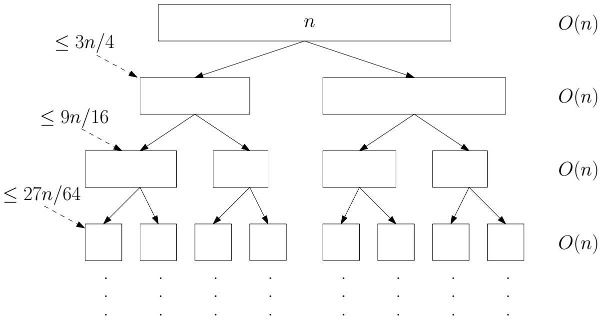
Variant Randomized Quicksort Algorithm

quicksort(A, n)

```
if n ≤ 1 then return A
repeat
    x ← a random element of A (x is called a pivot)
    A_L ← elements in A that are less than x          \\ Divide
    A_R ← elements in A that are greater than x       \\ Divide
unitl A_L.size ≤ 3n/4 and A_R.size ≤ 3n/4
B_L ← quicksort(A_L, A_L.size)                        \\ Conquer
B_R ← quicksort(A_R, A_R.size)                        \\ Conquer
t ← number of times x appear A
return the array obtained by concatenating B_L, the array
containing t copies of x, and B_R
```

Analysis of Variant

- Divide and Combine: takes $O(n)$ time
- Conquer: break an array of size n into two arrays, each has size at most $3n/4$. Recursively sort the 2 sub-arrays.



- Number of levels $\le \lg_{4/3} n = O(\lg n)$

Randomized Quicksort Algorithm

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ a random element of A (x is called a pivot)
3. $A_L \leftarrow$ elements in A that are less than x \\\\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\\\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\\\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\\\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

- Intuition: the quicksort algorithm should be better than the variant.

Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the expected running time of the randomized quicksort algorithm on n elements
- Assuming we choose the element of rank i as the pivot.
- The left sub-instance has size at most $i - 1$
- The right sub-instance has size at most $n - i$
- Thus, the expected running time in this case is $\big(T(i-1) + T(n-i)\big) + O(n)$
- Overall, we have

$$
\begin{aligned}
T(n) &= \frac{1}{n}\sum_{i=1}^{n}\big(T(i-1) + T(n-i)\big) + O(n) \\
&= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + O(n)
\end{aligned}
$$

- Can prove $T(n) \le c(n \lg n)$ for some constant c by reduction

Analysis of Randomized Quicksort Algorithm

The induction step of the proof:

$$
\begin{aligned}
T(n) &\le \frac{2}{n}\sum_{i=0}^{n-1} T(i) + c'n \le \frac{2}{n}\sum_{i=0}^{n-1} ci\lg i + c'n \\
&\le \frac{2c}{n}\left(\sum_{i=0}^{\lfloor n/2\rfloor -1} i\lg\frac{n}{2} + \sum_{i=\lfloor n/2\rfloor}^{n-1} i\lg n\right) + c'n \\
&\le \frac{2c}{n}\left(\frac{n^2}{8}\lg\frac{n}{2} + \frac{3n^2}{8}\lg n\right) + c'n \\
&= c\left(\frac{n}{4}\lg n - \frac{n}{4} + \frac{3n}{4}\lg n\right) + c'n \\
&= cn\lg n - \frac{cn}{4} + c'n \le cn\lg n \qquad \text{if } c \ge 4c'
\end{aligned}
$$

Exercise: Coupon Collector

Coupon Collector

Each box of cereal contains a coupon. There are n different types of coupons. Assuming all boxes are equally likely to contain each coupon, in expectation, how many boxes before you have all coupon types?

- Break into n stages $1, 2, 3, \cdots, n$
- Stage i terminates when we have collected i coupon types
- X_i: number of coupons collected in stage i
- $X = \sum_{i=1}^{n} X_i$: total number of coupons collected

Exercise: Coupon Collector

- X_i: number of coupons collected in stage i
- $X = \sum_{i=1}^{n} X_i$: total number of coupons collected
- In stage i: with probability $\frac{n-(i-1)}{n}$, a random coupon has type different from the $i-1$ types already seen
- Thus, $\mathbb{E}[X_i] = \frac{n}{n-(i-1)}$.
- By linearity of expectation:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \frac{n}{n-(i-1)} = \sum_{i=1}^{n} \frac{n}{i} = nH(n),$$

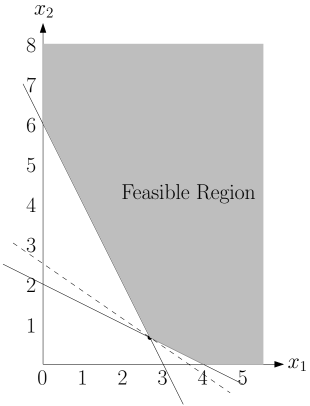
  where $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \Theta(\lg n)$ is called the n-th Harmonic number.
- $\mathbb{E}[X] = \Theta(n \lg n)$.

Outline

Outline

Example of Linear Programming

$$\begin{aligned} \min \quad & 4x_1 + 5x_2 \quad \text{s.t.} \\ & 2x_1 + x_2 \geq 6 \\ & x_1 + 2x_2 \geq 4 \\ & x_1, x_2 \geq 0 \end{aligned}$$



- optimum point: $x_1 = \frac{8}{3}, x_2 = \frac{2}{3}$
- value $= 4 \times \frac{8}{3} + 5 \times \frac{2}{3} = 14$

Standard Form of Linear Programming

$$
\begin{aligned}
\min \quad & c_1x_1 + c_2x_2 + \cdots + c_nx_n \quad \text{s.t.} \\
\sum A_{1,1}x_1 + A_{1,2}x_2 + \cdots + A_{1,n}x_n &\geq b_1 \\
\sum A_{2,1}x_1 + A_{2,2}x_2 + \cdots + A_{2,n}x_n &\geq b_2 \\
\vdots \qquad \vdots \qquad \vdots \qquad \vdots & \\
\sum A_{m,1}x_1 + A_{m,2}x_2 + \cdots + A_{m,n}x_n &\geq b_m \\
x_1, x_2, \cdots, x_n &\geq 0
\end{aligned}
$$

Standard Form of Linear Programming

$$\text{Let } x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \qquad c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix},$$

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

$$\text{Then, LP becomes} \qquad \min \quad c^{\mathrm{T}}x \quad \text{s.t.}$$
$$Ax \geq b$$
$$x \geq 0$$

- $\geq$ means coordinate-wise greater than or equal to

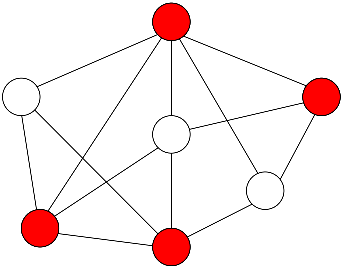
- Linear programmings can be solved in polynomial time

Algorithms for Solving LPs

- Simplex method: exponential time in theory, but works well in practice
- Ellipsoid method: polynomial time in theory, but slow in practice
- Internal point method: polynomial time in theory, works well in practice

Outline

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



Weighted Vertex-Cover Problem

Input: $G = (V, E)$ with vertex weights $\{w_v\}_{v \in V}$

Output: a vertex cover S with minimum $\sum_{v \in S} w_v$

Integer Programming for Weighted Vertex Cover

- For every $v \in V$, let $x_v \in \{0, 1\}$ indicate whether we select v in the vertex cover S
- The integer programming for weighted vertex cover:

$$
\begin{aligned}
(\text{IP}_{\text{WVC}}) \qquad \min \quad & \sum_{v \in V} w_v x_v \qquad \text{s.t.} \\
x_u + x_v &\geq 1 \qquad \forall (u, v) \in E \\
x_v &\in \{0, 1\} \qquad \forall v \in V
\end{aligned}
$$

- $(\text{IP}_{\text{WVC}}) \Leftrightarrow$ weighted vertex cover
- Thus it is NP-hard to solve integer programmings in general

- Integer programming for WVC:

$$
\begin{aligned}
(\mathsf{IP}_{\mathsf{WVC}}) \qquad \min \qquad & \sum_{v \in V} w_v x_v \qquad \text{s.t.} \\
x_u + x_v \geq 1 & \qquad \forall (u,v) \in E \\
x_v \in \{0,1\} & \qquad \forall v \in V
\end{aligned}
$$

- Linear programming relaxation for WVC:

$$
\begin{aligned}
(\mathsf{LP}_{\mathsf{WVC}}) \qquad \min \qquad & \sum_{v \in V} w_v x_v \qquad \text{s.t.} \\
x_u + x_v \geq 1 & \qquad \forall (u,v) \in E \\
x_v \in [0,1] & \qquad \forall v \in V
\end{aligned}
$$

- let IP $=$ value of $(\mathsf{IP}_{\mathsf{WVC}})$, LP $=$ value of $(\mathsf{LP}_{\mathsf{WVC}})$
- Then, LP $\leq$ IP

Algorithm for Weighted Vertex Cover

Algorithm for Weighted Vertex Cover

1. Solving (LP_{WVC}) to obtain a solution $\{x_u^*\}_{u \in V}$
2. Thus, $\text{LP} = \sum_{u \in V} w_u x_u^* \leq \text{IP}$
3. Let $S = \{u \in V : x_u \geq 1/2\}$ and output S

Lemma S is a vertex cover of G.

Proof.

- Consider any edge $(u, v) \in E$: we have $x_u^* + x_v^* \geq 1$
- Thus, either $x_u^* \geq 1/2$ or $x_v^* \geq 1/2$
- Thus, either $u \in S$ or $v \in S$. $\square$

Algorithm for Weighted Vertex Cover

Algorithm for Weighted Vertex Cover

1. Solving (LP$_{\text{WVC}}$) to obtain a solution $\{x_u^*\}_{u\in V}$
2. Thus, LP $= \sum_{u\in V} w_u x_u^* \leq$ IP
3. Let $S = \{u \in V : x_u \geq 1/2\}$ and output S

Lemma S is a vertex cover of G.

Lemma cost$(S) := \sum_{u\in S} w_u \leq 2 \cdot$ LP.

Proof.

$$\text{cost}(S) = \sum_{u\in S} w_u \leq \sum_{u\in S} w_u \cdot 2x_u^* = 2\sum_{u\in S} w_u \cdot x_u^*$$
$$\leq 2\sum_{u\in V} w_u \cdot x_u^* = 2 \cdot \text{LP}.$$

□

Algorithm for Weighted Vertex Cover

Algorithm for Weighted Vertex Cover

1. Solving (LP$_{\text{WVC}}$) to obtain a solution $\{x_u^*\}_{u\in V}$
2. Thus, LP $= \sum_{u\in V} w_u x_u^* \leq$ IP
3. Let $S = \{u \in V : x_u^* \geq 1/2\}$ and output S

Lemma S is a vertex cover of G.

Lemma cost$(S) := \sum_{u\in S} w_u \leq 2 \cdot$ LP.

Theorem Algorithm is a 2-approximation algorithm for WVC.

Proof.

cost$(S) \leq 2 \cdot$ LP $\leq 2 \cdot$ IP $= 2 \cdot$ cost(best vertex cover). □