Approximation Basics

Milestones, Concepts, and Examples

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course @ Shanghai Jiao Tong University

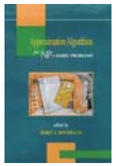
History of Approximation

1966	**Graham**: First analyzed algorithms by approximation ratio
1971	**Cook**: Gave the concepts of NP-Completeness
1972	**Karp**: Introduced plenty NP-Hard combinatorial optimization problems
1970's	Approximation became a popular research area
1979	**Garey & Johnson**: Computers and Intractability: A guide to the Theory of NP-Completeness

Books

CS 351 Stanford Univ

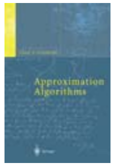
(1991-1992) Rajeev Motwani
Lecture Notes on Approximation Algorithms Volume I



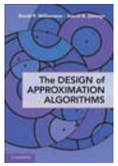
(1997) Hochbaum (Editor)
Approximation Algorithms for NP-Hard Problems

(1999) Ausiello, Crescenzi, Gambosi, etc.
Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties

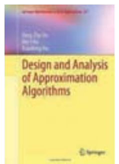
Books (2)



(2001) Vijay V. Vazirani
Approximation Algorithms



(2010) D.P. Williamson & D.B. Shmoys
The Design of Approximation Algorithms



(2012) D.Z Du, K-I. Ko & X.D. Hu
Design and Analysis of Approximation Algorithms

NP Optimization Problem

An NP Optimization Problem P is a fourtuple $(I, sol, m, goal)$ s.t.

- I is the set of the instances of P and is recognizable in polynomial time.
- Given an instance x of I, $sol(x)$ is the set of short feasible solutions of x and $\forall x$ and $\forall y$ such that $|y| \leq p(|x|)$, it is decidable in polynomial time whether $y \in sol(x)$.
- Given an instance x and a feasible solution y of x, $m(x, y)$ is a polynomial time computable measure function providing a positive integer which is the value of y.
- $goal \in \{\max, \min\}$ denotes maximization or minimization.

An Example of NP Optimization Problem

Example: Minimum Vertex Cover

Given a graph $G = (V, E)$, the Minimum Vertex Cover problem (MVC) is to find a vertex cover of minimum size, that is, a minimum node subset $U \subseteq V$ such that, for each edge $(v_i, v_j) \in E$, either $v_i \in U$ or $v_j \in U$.

Justification $\to$ MVC is an NP Optimization Problem

- $I = \{G = (V, E) \mid G \text{ is a graph}\}$; *poly-time decidable*
- $sol(G) = \{U \subseteq V \mid \forall (v_i, v_j) \in E[v_i \in U \vee v_j \in U]\}$; *short feasible solution set and poly-time decidable*
- $m(G, U) = |U|$; *poly-time computable function*
- $goal = \min$.

NPO Class

Definition: (NPO Class)

The class NPO is the set of all NP optimization problems.

Definition: (Goal of NPO Problem)

The goal of an NPO problem with respect to an instance x is to find an *optimum solution*, that is, a feasible solution y such that $m(x, y) = goal\{m(x, y') : y' \in sol(x)\}$.

What is Approximation Algorithm?

Definition: (Approximation Algorithm)

Given an NP optimization problem $P = (I, sol, m, goal)$, an algorithm A is an approximation algorithm for P if, for any given instance $x \in I$, it returns an approximate solution, that is a feasible solution $A(x) \in sol(x)$ with guaranteed quality.

Note:

- Guaranteed quality is the difference between approximation and heuristics.
- Approximation for PO, NPO and NP-hard Optimization.
- Decision, Optimization, and Constructive Problems.

r-Approximation

Definition: (Approximation Ratio)

Let P be an NPO problem. Given an instance x and a feasible solution y of x, we define the performance ratio of y with respect to x as

$$R(x,y) = \max\left\{\frac{m(x,y)}{opt(x)}, \frac{opt(x)}{m(x,y)}\right\}.$$

Definition: (r-Approximation)

Given an optimization problem P and an approximation algorithm A for P, A is said to be an r-approximation for P if, given any input instance x of P, the performance ratio of the approximate solution $A(x)$ is bounded by r, say, $R(x, A(x)) \le r$.

APX Class

Definition: (F-APX)

Given a class of functions F, an NPO problem P belongs to the class F-APX if an r-approximation polynomial time algorithm A for P exists, for some function $r \in F$.

Example:

- F is constant functions $\rightarrow P \in$ APX.
- F is $O(\log n)$ functions $\rightarrow P \in$ log-APX.
- F is $O(n^k)$ functions (polynomials) $\rightarrow p \in$ poly-APX.
- F is $O(2^{n^k})$ functions $\rightarrow P \in$ exp-APX.

Special Case

Definition: (Polynomial Time Approximation Scheme $\rightarrow$ PTAS)

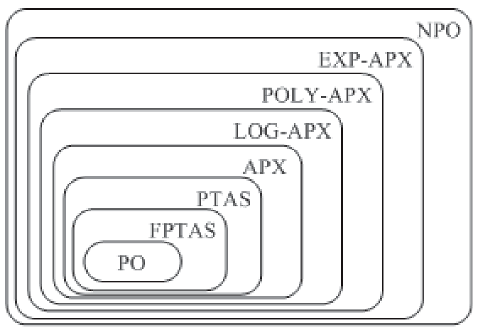
An NPO problem P belongs to the class PTAS if an algorithm A exists such that, for any rational value $\epsilon > 0$, when applied A to input (x, ϵ), it returns an $(1+\epsilon)$-approximate solution of x in time polynomial in $|x|$.

Definition: (Fully PTAS $\rightarrow$ FPTAS)

An NPO problem P belongs to the class FPTAS if an algorithm A exists such that, for any rational value $\epsilon > 0$, when applied A to input (x, ϵ), it returns a $(1+\epsilon)$-approximate solution of x in time polynomial both in $|x|$ and in $\frac{1}{\epsilon}$.

Approximation Class Inclusion

If $P \neq NP$, then FPTAS $\subseteq$ PTAS $\subseteq$ APX $\subseteq$ Log-APX $\subseteq$ Poly-APX $\subseteq$ Exp-APX $\subseteq$ NPO



- Constant-Factor Approximation (APX)
  - Reduce App. Ratio
  - Reduce Time Complexity
- PTAS ($(1+\epsilon)$-Appx)
  - Test Existence
  - Reduce Time Complexity

Procedure

Given:

- An instance of the problem specifies a set of items

Goal:

- Determine a subset of the items that satisfies the problem constraints
- Maximize or minimize the measure function

Steps:

- Sort the items according to some criterion
- Incrementally build the solution starting from the empty set
- Consider items one at a time, and maintain a set of "selected" items
- Terminate when break the problem constraints

Set Cover Problem

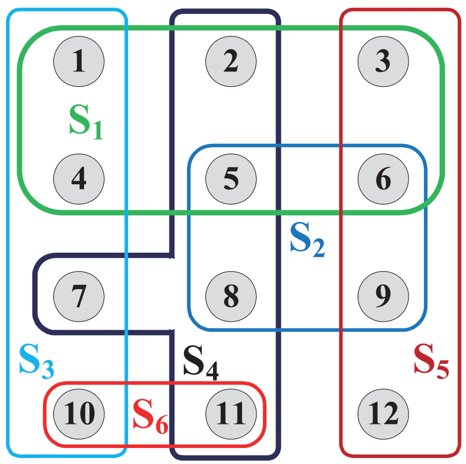
Problem

Instance: *Given a universe* $U = \{e_1, \cdots, e_n\}$ *of* n *elements, a collection of subsets* $\mathbf{S} = \{S_1, \ldots, S_m\}$ *of* U*, and a cost function* $c : \mathbf{S} \to \mathbb{Q}^+$.

Solution: *A subcollection* $\mathbf{S}' \subseteq \mathbf{S}$ *that covers all elements of* U.

Measure: *Total cost of the chosen subcollection,* $\sum_{S_i \in \mathbf{S}'} c(S)$.

An Example



$U = \{1, 2, \cdots, 12\}$

$\mathbf{S} = \{S_1, S_2, \cdots, S_6\}$

$S_1 = \{1, 2, 3, 4, 5, 6\}$

$S_2 = \{5, 6, 8, 9\}$

$S_3 = \{1, 4, 7, 10\}$

$S_4 = \{2, 5, 7, 8, 11\}$

$S_5 = \{3, 6, 9, 12\}$

$S_6 = \{10, 11\}$

Optimal Solution:

$\mathbf{S}' = \{S_3, S_4, S_5\}$

Greedy Algorithm

Algorithm 1 Greedy Set Cover

Input: U with n item; $\mathbf{S}$ with m subsets; cost function $c(S_i)$.
Output: Subset $\mathbf{S}' \subseteq \mathbf{S}$ such that $\bigcup_{e_i \in S_k \in \mathbf{S}'} e_i = U$.

$C \leftarrow \emptyset$
while $C \neq U$ **do**
  Find the most cost-effective set S.
  $\forall e \in S \backslash C$, set $price(e) = \frac{c(S)}{|S - C|}$. Set $C \leftarrow C \cup S$.
end while
Output selected S.

The cost-effectiveness of a set S is the average cost at which it covers new elements; The price of an element e is the average cost when e is covered.

Time Complexity

Theorem: Greedy Set Cover has time complexity $O(mn)$.

Proof:
(1). There are at most $O(\min\{m, n\})$ iterations to select the subcollection. Within each iteration to find the minimum cost-effectiveness, it requires $O(m)$ times;

(2). There are totally n elements, and each e_i, the price modification will perform at most $O(m)$ times, each with linear operations. Totally the price updating procedure requires $O(mn)$ time.

Thus the total running time is
$O(\min\{m, n\}) \cdot O(m) + O(mn) = O(mn)$. □

Approximation Ratio

Theorem: Greedy Set Cover is an H_n factor approximation algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$. ← Harmonic Number (Log-APX)

Proof: Let $m_g(U)$ be the cost of Greedy Set Cover, $m^*(U)$ be the cost of the optimal solution.

Number the elements of U in the order in which they were covered by the algorithm.

Let $e_1, \ldots, e_n$ be this numbering (resolving ties arbitrarily).

Observation: For each $k \in \{1, \ldots, n\}$, $price(e_k) \leq \dfrac{m^*(U)}{n-k+1}$.

Proof (Continued)

Since in any iteration, the optimal solution can cover the remaining elements $\overline{C}$ with cost $m^*(U)$.

Therefore, among the remaining sets, there must be one having cost-effectiveness of at most $m^*(U)/|\overline{C}|$.

In the iteration in which e_k was covered, $|\overline{C}| \geq n - k + 1$. Thus

$$price(e_k) \leq \frac{m^*(U)}{|\overline{C}|} \leq \frac{m^*(U)}{n-k+1}.$$

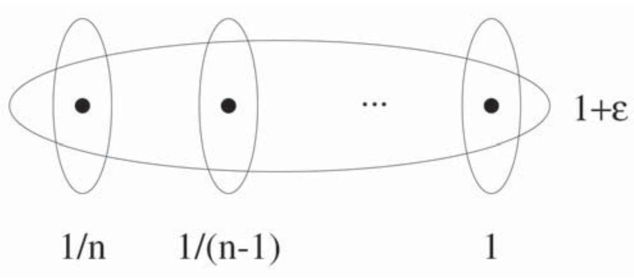
Proof (Continued)

The total cost of the sets picked by this algorithm is equal to $\sum_{k=1}^{n} price(e_k)$. Then

$$\begin{aligned} m_g(U) &= \sum_{k=1}^{n} price(e_k) \\ &\leq (1 + \frac{1}{2} + \cdots + \frac{1}{n}) \cdot m^*(U) \\ &= H_n \cdot m^*(U) \end{aligned}$$

□

Tight Example



The optimal cover has a cost of $1+\epsilon$. While the greedy algorithm will output a cover of cost $\dfrac{1}{n}+\dfrac{1}{n-1}+\cdots+1=H_n$.

Maximum Knapsack Problem

Problem

Instance: *Given finite set X of items and a positive integer b, for each $x_i \in X$, it has value $p_i \in Z^+$ and size $a_i \in Z^+$.*

Solution: *A set of items $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leq b$.*

Measure: *Total value of the chosen items, $\sum_{x_i \in Y} p_i$.*

Greedy Algorithm

Algorithm 2 Greedy Knapsack

Input: X with n item and b; for each $x_i \in X$, value p_i, and a_i.
Output: Subset $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leq b$.

```
Sort X in non-increasing order with respect to the ratio p_i/a_i
                              ▷ Let x_1, ..., x_n be the sorted sequence
Y = ∅;
for i = 1 to n do
    if b ≥ a_i then
        Y = Y ∪ {x_i};
        b = b − a_i;
    end if
end for
Return Y
```


Time Complexity

Theorem: Greedy Knapsack has time complexity $O(n \log n)$.

Proof: Consider items in non-increasing order with respect to the profic/occupancy ratio.

(1). To sort the items, it requires $O(n \log n)$ times;

(2). and then the complexity of the algorithm is linear in their number.

Thus the total running time is $O(n \log n)$. □

Approximation Ratio

Theorem: The solution of Greedy Knapsack can be arbitrarily far from the optimal value.

Proof: (A Worst Case Example)

- Consider an instance X of Maximum Knapsack with n items. $p_i = a_i = 1$ for $i = 1, \cdots, n-1$. $p_n = b-1$ and $a_n = b = kn$ where k is an arbitrarily large number.
- Let $m^*(X)$ be the size of optimal solution, and $m_g(X)$ the size of Greedy Knapsack solution. Then, $m^*(X) = b-1$, while $m_g(X) = n-1$,
- $\dfrac{m^*(X)}{m_g(X)} > \dfrac{kn-1}{n-1} > k.$

□

Improvement

The poor behavior of Greedy Knapsack if due to the fact that the algorithm does not include the element with highest profit in the solution while the optimal solution contains only this element.

Theorem

Given an instance X of the Maximum Knapsack problem, let $m_H(X) = \max\{p_{max}, m_g(X)\}$. where p_{max} is the maximum profit of an item in X. Then $m_H(X)$ satisfies the following inequality:

$$\frac{m^*(X)}{m_H(X)} < 2. \quad \textit{(Constant-Factor Approximation)}$$

Proof (1)

Let j be the first index of an item in the order that cannot be included. The profit achieved so far (up to item j) is:

$$\overline{p_j} = \sum_{i=1}^{j-1} p_i \leq m_g(X).$$

The total occupancy (size) is

$$\overline{a_j} = \sum_{i=1}^{j-1} a_i \leq b.$$

Observation: $m^*(X) < \overline{p_j} + p_j$.

Proof (2)

x_i are ordered by $\frac{p_i}{a_i}$, so any exchange of any subset of the chosen items $x_1, \cdots, x_{j-1}$ with any subset of the unchosen items $x_j, \cdots, x_n$ that does not increase $\overline{a_j}$ will not increase the overall profit.

Thus $m^*(X)$ is bounded by $\overline{p_j}$ plus the maximum profit from filling the remaining space.

Since $\overline{a_j} + a_j > b$ (otherwise x_j will be selected), we obtain:

$$m^*(X) \leq \overline{p_j} + (b - \overline{a_j}) \cdot \frac{p_j}{a_j} < \overline{p_j} + p_j.$$

Proof (3)

To complete the proof we consider two possible cases.

- If $p_j \le \overline{p_j}$, then

$$m^*(X) < \overline{p_j} + p_j \le 2\overline{p_j} \le 2m_g(X) \le 2m_H(X).$$

- If $p_j > \overline{p_j}$, then $p_{max} > \overline{p_j}$, and

$$m^*(X) < \overline{p_j} + p_{max} \le 2p_{max} \le 2m_H(X)$$

Thus Greedy Knapsack is 2-approximation. □

Maximum Independent Set Problem

Definition

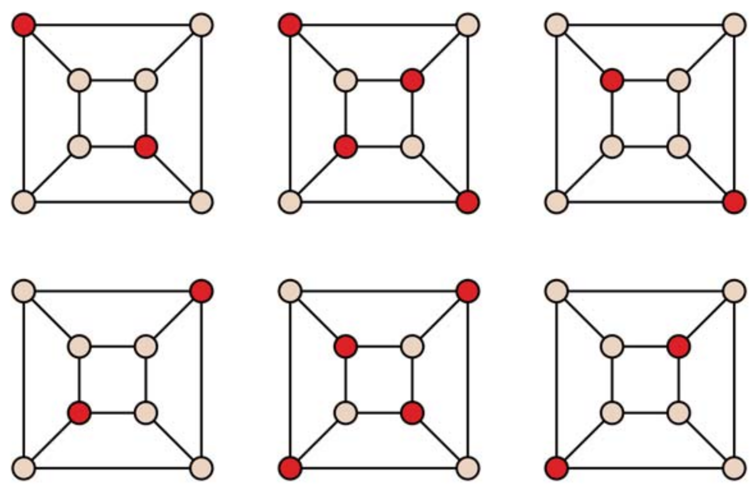
Instance: Given a graph $G = (V, E)$

Solution: An independent set $V' \subseteq V$ on G, such that for any $(u, v) \in E$, either $u \notin V'$ or $v \notin V'$.

Measure: Cardinality of the independent set, $|V'|$.

An Example

The cube has 6 maximal independent sets (red nodes).



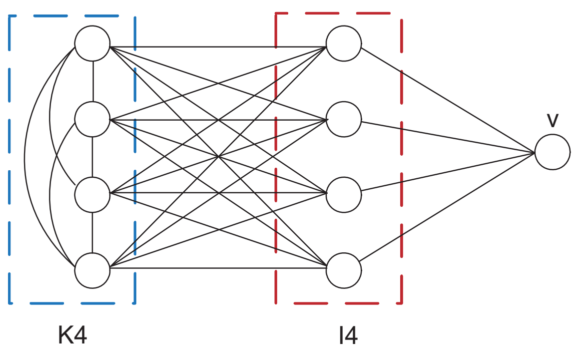
Greedy Algorithm

Algorithm 3 Greedy Independent Set

Input: Graph $G = (V, E)$.
Output: Independent Node Subset $V' \subseteq V$ in G.

$V' = \emptyset$;
$U = V$;
while $U \ne \emptyset$ **do**
  x = vertex of minimum degree in the graph induced by U.
  $V' = V' \cup \{x\}$.
  Eliminate x and all its neighbors from U.
end while
Return V'.

Worst Case Example



Let K_4 be a clique with four nodes and I_4 an independent set of four nodes. v is the first to be chosen by algorithm, and the resulting solution contains this node and exactly one node of K_4. The optimal solution contains I_4. Thus $\frac{m^*(X)}{m_g(X)} \geq \frac{n}{2}$.

Approximation Ratio

Theorem: Given a graph G with n vertices and m edges, let $\delta = \frac{m}{n}$. The approximation ratio of Greedy Independent Set is

$$\frac{m^*(X)}{m_g(X)} \leq \delta + 1. \qquad (\textit{Poly-APX})$$

Proof:

- Define V^* the optimal independent set for G.
- x_i the vertex chosen at i^{th} iteration of Greedy Algorithm.
- d_i the degree of x_i, then each time remove $d_i + 1$ vertices.
- k_i the number of vertices in V^* that are among $d_i + 1$ vertices deleted in the i^{th} iteration.

Proof (2)

Since algorithm stops when all vertices are eliminated,

$$\sum_{i=1}^{m_g(G)} (d_i + 1) = n. \tag{1}$$

k_i represent distinct vertices set in V^*,

$$\sum_{i=1}^{m_g(G)} k_i = |V^*| = m^*(G). \tag{2}$$

Each iteration the degree of the deleted vertices is at least $d_i(d_i + 1)$ and an edge cannot have both its endpoints in V^*, the number of deleted edges is at least $\frac{d_i(d_i+1)+k_i(k_i-1)}{2}$,

$$\sum_{i=1}^{m_g(G)} \frac{d_i(d_i+1)+k_i(k_i-1)}{2} \leq m = \delta n. \tag{3}$$

Proof (3)

Notation: Define V_i^g as the deleted vertex set in the ith iteration, V_i^* as the vertices in V^* that are deleted in this iteration.

$$|V_i^g| = d_i + 1, \ |V_i^*| = k_i, \ V_i^* \subseteq V_i^g.$$

The number of deleted edges $\geq \frac{d_i(d_i+1)}{2}$. (this lower bound implies a virtual clique with $d_i + 1$ vertices)

However, vertices in V_i^g are independent to each other, so they cannot "contribute" to the above clique. Thus we need to repay the degrees back. Correspondingly,

$$\begin{aligned}\text{no. of deleted edges} &\geq \text{edges of clique with } d_i + 1 \text{ vertices} \\ &\quad + \text{repaid edges of a clique with } k_i \text{ vertices} \\ &= \frac{d_i(d_i+1)+k_i(k_i-1)}{2}\end{aligned}$$

Proof (4)

Adding inequalities (1), (2), and (3) together, we have

$$\sum_{i=1}^{m_g(G)} \Big(d_i(d_i+1)+k_i(k_i-1)+(d_i+1)+k_i\Big) \le 2\delta n + n + m^*(G)$$

$$\implies \sum_{i=1}^{m_g(G)} ((d_i+1)^2+k_i^2) \le n(2\delta+1)+m^*(G).$$

By applying the Cauchy-Schwarz Inequality, the left part is minimized when $d_i + 1 = \frac{n}{m_g(G)}$ and $k_i = \frac{m^*(G)}{m_g(G)}$, hence,

$$\frac{n^2+m^*(G)^2}{m_g(G)} \le \sum_{i=1}^{m_g(G)} ((d_i+1)^2+k_i^2) \le n(2\delta+1)+m^*(G),$$

C-S: $\left(\sum_{i=1}^{n} x_i\right)^2 \le n\sum_{i=1}^{n} x_i^2$, equality holds when $x_1 = \cdots = x_n$. (run twice here)

Proof (5)

Thus,

$$m_g(G) \ge \frac{n^2+m^*(G)^2}{n(2\delta+1)+m^*(G)} = m^*(G)\frac{\frac{n^2}{m^*(G)}+m^*(G)}{n(2\delta+1)+m^*(G)}$$

We have

$$\frac{m^*(G)}{m_g(G)} \le \frac{2\delta+1+\frac{m^*(G)}{n}}{\frac{n}{m^*(G)}+\frac{m^*(G)}{n}}.$$

Easy to know that $m^*(G) \le n$.

Let $x = \frac{m^*(G)}{n} \le 1$. For any $x \in (0,1]$, $x + \frac{1}{x}$ decreases when x increases, so $\frac{x+a}{x+\frac{1}{x}}$ is maximized when $x = 1$.

Proof (6)

When $m^*(G) = n$, the right-hand inequality is maximized,

$$\frac{m^*(G)}{m_g(G)} \le \frac{2\delta+1+1}{1+1} = \delta+1.$$

The Greedy Independent Set Algorithm yields an approximation ratio of $\delta+1$.

Thus Maximum Indecent Set Problem is a Poly-APX. □

Note: $\max(m) = \frac{n(n-1)}{2}$ when G is a K_n clique, and $\delta = \frac{n-1}{2}$.

Procedure

Given:

- An instance of the problem specifies a set of items $I = \{x_1, \cdots, x_n\}$

Goal:

- Determine a suitable partition that satisfies the problem constraints
- Maximize or minimize the measure function

Steps:

- Sort the items according to some criterion.
- Build the output partition P sequentially.
- Note that when algorithm considers item x_i, it is not allowed to modify the partition of items x_j, for $j < i$ (only assign once).

Maximum Cut Problem

Problem

Instance: *Given a graph* $G = (V, E)$.

Solution: *A partition of* V *into sets* S *and* $\overline{S}$.

Measure: *Maximize the number of edges running between* S *and* $\overline{S}$.

Sequential Algorithm

Algorithm 4 Sequential Maximum Cut

```
Input: G = (V, E);
Output: Partition of V = S ∪ S̄.
Pick v1, v2 from V arbitrarily. Set A ← {v1}; B ← {v2}
for v ∈ V − {v1, v2} do
    if d(v, A) ≥ d(v, B) then
        B ← B ∪ {v}
    else
        A ← A ∪ {v}
    end if
end for
Return A, B.
```

$d(v, A)$ is the number of edges between v and A

Approximation Ratio

Theorem. Greedy Sequential has approximation ratio 2.

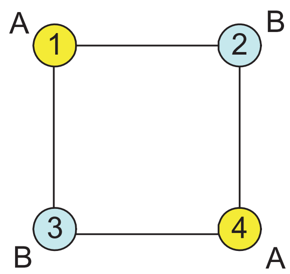
Proof. Consider each edge (v_i, v_j). Whether it belongs to the cut is determined when v_i is fixed and at the moment when v_j is fixed. Thus we can partition the edge set by its “decision vertex".

At each iteration, by the algorithm strategy at least half of edges in each partition will be assigned to the cut, and will never change again.

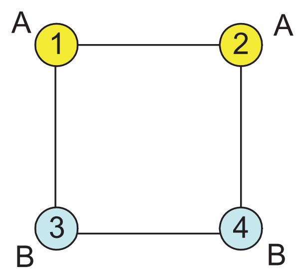
Thus $|A_g| \geq \frac{|E|}{2}$. It is easy to see that $|OPT| \leq |E|$. Hence

$$\frac{|OPT|}{|A_g|} \leq \frac{|E|}{|E|/2} = 2. \qquad \square$$

Tight Example



OPT



Ag

Minimum Scheduling on Identical Machines

Problem

Instance: *Given set of jobs T, number p of machines, length l_j for executing job $t_j \in T$.*

Solution: *A p-machine schedule for T, i.e., a function $f : T \mapsto [1, \cdots, p]$.*

Measure: *Minimize the schedule's makespan, i.e.,*

$$\min \Big(\max_{i \in [1, \cdots, p]} \sum_{t_j \in T : f(t_j) = i} l_j \Big).$$

Note: This problem is NP-Hard even in the case of $p = 2$.

Sequential Algorithm

Algorithm 5 Largest Processing Time Sequential Algorithm

Input: Set T with n jobs, each has length l_j, p machines;
Output: Partition P of T.

Sort l in non-increasing order w.r.t. their processing time
 ▷ Let $t_1, \cdots, t_n$ be the obtained sequence, $l_1 \geq \cdots \geq l_n$.
$P = \{\{t_1\}, \emptyset, \cdots, \emptyset\}$
for $i = 2$ to n **do**
 Find machine p_j with minimum finish time

$$A_j(i-1) = \min_{1 \leq j \leq p} \sum_{1 \leq k \leq i-1 : f(t_k) = j} l_k$$

 Append t_i into p_j.
end for
Return P.

Approximation Ratio

Theorem: Greedy Sequential has approximation ratio $\frac{4}{3} - \frac{1}{3p}$.

Proof: Let j be the job of T that is last considered by Greedy Sequential and let l_{min} be its length (the shortest one).

Consider two cases: $l_{min} > \frac{m^*(T)}{3}$ and $l_{min} \leq \frac{m^*(T)}{3}$.

Proof (2)

If $l_{min} > \frac{m^*(T)}{3}$, then at most two jobs may have been assigned to any machine (otherwise it will violate the definition of $m^*(T)$). There are p machines in the system, so

$$p < |T| \leq 2p.$$

Let $m_L(T)$ be the length of the Greedy Sequential solution. Next, we prove that

$$m_L(T) = m^*(T) \text{ for } |T| \leq 2p.$$

We can setup $2p - |T|$ virtual jobs with length 0 such that $|T| = 2p$.

Easy to see, either greedy approach or optimal solution will divide those $2p$ jobs into p pairs.

Proof (3)

Assume $m_L(T)$ is the length of the ith machine (obviously $i \leq p$, and the ith machine is the makespan). Then

$$m_L(T) = l_i + l_{2p-i+1}.$$

If $l_{2p-i+1} = 0$, then it means l_i forms the makespan. Thus $m_L(T) = m^*(T) = l_i$.

If $l_{2p-i+1} > 0$, then it means that the ith machine has two jobs with length$>$0. Assume $m_L(T) > m^*(T)$ at this scenario.

Consider the new matching pair on the ith machine in an optimal solution. However, the pairs containing $\{l_1, \cdots, l_{i-1}\}$ must match an l_j ($2p - i + 2 \leq j \leq 2p$), otherwise the new matching is greater than $m_L(T)$. Impossible to get one!

Proof (4)

If $l_{min} \leq \dfrac{m^*(T)}{3}$, let $W = \sum\limits_{k=1}^{|T|} l_k$, then we have $m^*(T) \geq \dfrac{W}{p}$.

Use Contradiction. Assume theorem doesn't hold (i.e., the approximation ratio is larger than $\frac{4}{3} - \frac{1}{3p}$) and let T violates the claim having the minimum number of jobs.

Since T is a minimum counter-example, T' obtained from T by removing job t_j satisfies the claim ($m_L(T) > m_L(T')$).

Thus Greedy Sequential assigns t_j to machine h that will have the largest processing time.

Proof (5)

Since t_j was assigned to the least loaded machine, then the finish time of any other machine is at least $A_h(|T|) - l_j$. Then $W \geq p(A_h(|T|) - l_j) + l_j$ and we obtain that

$$m_L(T) = A_h(|T|) \leq \frac{W}{p} + \frac{p-1}{p} l_{min}$$

Since $m^*(T) \geq \frac{W}{p}$ and $l_{min} \leq \frac{m^*(T)}{3}$, we have

$$m_L(T) \leq m^*(T) + \frac{p-1}{3p} m^*(T) = (\frac{4}{3} - \frac{1}{3p}) m^*(T).$$

Thus, the contradiction cannot hold. □