Unit 10. Approximation Algorithms

Algorithms

EE3980

June 3, 2019

0/1 Knapsack Problem

- Given n objects, each with profit p_i and weight w_i, $1 \le i \le n$, to be placed into a sack that can hold maximum of m weight. However, there is an additional constraint that each object must be placed as a whole into the sack, or not at all. That is, find x_i, $1 \le i \le n$, such that

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} p_i x_i, \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \le m, \\ \text{and} \quad & x_i = 0 \text{ or } 1, \qquad 1 \le i \le n. \end{aligned} \tag{10.1.1}$$

- We need $\sum_{i=1}^{n} w_i > m$ for nontrivial solutions.
- It is assumed that the n objects are ordered by p_i/w_i in a nonincreasing order.
- It is also assumed that the optimal profit is p^*.
- The following greedy algorithm can find a feasible but not necessarily the optimal solution.

0/1 Knapsack Problem – Greedy Algorithm

Algorithm 10.1.1. Greedy Knapsack

```
// Solving knapsack using greedy method.
// Input: n, p[], w[], m; Output: solution x[].
Algorithm GKnap0(n, p, w, x, m)
{ // The objects are assumed to be sorted by p[i]/w[i] in nonincreasing order.
    for i := 1 to n do x[i] := 0;
    i := 1; fp1 := 0;
    while (m ≥ w[i]) do {
        x[i] := 1; fp1 := fp1 + p[i]; m := m − w[i]; i := i + 1;
    }
}
```

- At the end of the algorithm GKnap0 object i is placed into the sack if $x[i] = 1$, and fp_1 is the final profit.
- It is easy to see that $fp_1 \le p^*$, and $fp_1 < p^*$ most of the time.

0/1 Knapsack Problem – An example

- An example of the knapsack problem:
  Given n objects, $p_i = 1$ and $w_i = 1$ for $i = 1, \ldots, n-1$, and $p_n = k \cdot n - 1$, $w_n = m = k \cdot n$, $k \gg 1$.
- The optimal profit for this problem is $p^* = k \cdot n - 1$ with $x_n = 1$ and $x_i = 0$, $i = 1, \ldots, n-1$.
- Note that $p_i / w_i = 1$ for $i = 1, \ldots, n-1$ and $p_n / w_n = (k \cdot n - 1)/(k \cdot n) = 1 - 1/(k \cdot n) < 1$. Thus, the objects are already in a nonincreasing order.
- The Greedy Knapsack algorithm finds a solution $x_i = 1$, $i = 1, \ldots, n-1$, and $x_n = 0$ with a profit $fp_1 = n - 1$.
- The ratio $p^*/fp_1 = (k \cdot n - 1)/(n - 1) \gg 1$.
- The greedy Knapsack algorithm can be modified as the following to fix this problem.

0/1 Knapsack Problem – Revised Greedy Algorithm

Algorithm 10.1.2. Revised Greedy Knapsack

```
   // Revised knapsack algorithm using greedy method.
   // Input: n, p[], w[], m; Output: solution x[].
 1 Algorithm GKnap(n, p, w, x, m)
 2 { // The objects are assumed to be sorted by p[i]/w[i] in nonincreasing order.
 3      for i := 1 to n do x[i] := 0 ;
 4      i := 1 ; fp2 := 0 ; m' := m ;
 5      while (m' ≥ w[i]) do { // Greedy method.
 6           x[i] := 1 ; fp2 := fp2 + p[i] ; m' := m' − w[i] ; i := i + 1 ;
 7      }
 8      Find j such that p[j] = max(p[1 : n]) ; // Object j has the max profit.
 9      if ( p[j] > fp2 and w[j] ≤ m ) then { // Choose the object j.
10           for i := 1 to n do x[i] := 0 ;
11           x[j] := 1 ; fp2 := p[j] ;
12      }
13 }
```

- This revised algorithm adds lines 8-12 for the possibility of choosing the object with the largest profit.

0/1 Knapsack Problem – The Profit

- In the preceding algorithm, let $i = h$ when the **while** loop on line 8 terminates.
- At this time, we have

$$fp_1 = \sum_{i=1}^{h-1} p_i < p^* < fp_1 + p_h \cdot \frac{m'}{w_h} < fp_1 + p_h.$$

- Consider two cases
  - Case 1: $p_h < fp_1$ then

$$p^* < fp_1 + p_h < 2 \cdot fp_1 \le 2 \cdot fp_2.$$

  - Case 2: $p_h > fp_1$, then

$$p^* < fp_1 + p_h < 2 \cdot p_h \le 2 \cdot \max\{p_i\} \le 2 \cdot fp_2.$$

- Thus, we have the following lemma.

0/1 Knapsack Problem – Bound of The Profit

Lemma 10.1.3.

Given a 0/1 knapsack problem, let the optimal profit be p^* and the profit found by Algorithm (10.1.2) be fp_2, then

$$\frac{p^*}{fp_2} \leq 2. \tag{10.1.2}$$

- The greedy algorithm to solve the knapsack problem always finds a profit fp_2 such that $\frac{p^*}{2} < fp_2 < p^*$.
- This algorithm finds an approximate solution given the bound above. Though it is not an optimal solution, it has very low time complexity.

Approximation Algorithms

- There are no known polynomial time algorithms to solve $\mathcal{NP}$-complete problems.
- Solving these problems can take a long time if the problem size is not small.
- But, there are many practical problems that are $\mathcal{NP}$-complete.
- Heuristics might be used with existing algorithms to reduce solution time.
  - Backtracking and branch and bound algorithms.
  - The solution quality can vary significantly from instance to instance.
  - Exponential time complexity can still take formidable time.
- Instead of finding the optimal solution, a different approach is to find an approximate solution, which is a feasible solution with value close the optimal solution.
- An approximation algorithm for a problem $\mathcal{Q}$ is an algorithm that generates approximate solutions for $\mathcal{Q}$.

Approximation Algorithms — Definitions

- Let $\mathcal{Q}$ be a problem such as the knapsack (or the traveling salesperson) problem.
- Let I is an instance of problem $\mathcal{Q}$ and $F^*(I)$ be the value of an optimal solution to I.
- An approximation algorithm generally produces a feasible solution to I whose value $\hat{F}(I)$ is less than (greater than) $F^*(I)$ if $\mathcal{Q}$ is a maximization (minimization) problem.

Definition. 10.1.4. Absolute approximation.

$\mathcal{A}$ is an absolute approximation algorithm for problem $\mathcal{Q}$ if and only if for every instance I of $\mathcal{Q}$, $|F^*(I) - \hat{F}(I)| \leq k$ for some constant k.

Definition. 10.1.5.

$\mathcal{A}$ is an $f(n)$-approximate algorithm for problem $\mathcal{Q}$ if and only if for every instance I of size n, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq f(n)$ for $F^*(I) > 0$.

Approximation Algorithms — Definitions, II

Definition. 10.1.6.

An ϵ-approximate algorithm is an $f(n)$-approximate algorithm for which $f(n) \leq \epsilon$ for some constant ϵ.

- Note that for maximization problems, $|F^* - \hat{F}(I)|/F^* \leq 1$ for every feasible solution to I.
  - Thus, $\epsilon < 1$ is usually required for ϵ-approximate algorithms.
- In the following, we assume ϵ is an input to algorithm $\mathcal{A}$.

Definition. 10.1.7.

$\mathcal{A}(\epsilon)$ is an approximation scheme if and only if for every given $\epsilon > 0$ and problem instance I, $\mathcal{A}(\epsilon)$ generates a feasible solution such that $|F^*(I) - \hat{F}(I)|/F^* \leq \epsilon$. ($F^* > 0$ is assumed.)

Approximation Algorithms — Definitions, III

Definition. 10.1.8.

An approximation scheme is a polynomial time approximation scheme if and only of for every fixed $\epsilon > 0$, it has computing time that is polynomial in the problem size.

Definition. 10.1.9.

An approximation scheme whose computing time is a polynomial both in problem size and in $1/\epsilon$ is a fully polynomial time approximation scheme.

- For most $\mathcal{NP}$-complete problems, it can be shown the absolute approximation algorithms exist only if $\mathcal{P}$=$\mathcal{NP}$-complete.
  - For certain $\mathcal{NP}$-complete problems, the existence of $f(n)$-approximate algorithm is also shown only when $\mathcal{P}$=$\mathcal{NP}$-complete.

Absolute Approximations

- There are very few $\mathcal{NP}$-hard optimization problems for which polynomial time absolute approximation algorithms are known.
- The problem of determining the minimum number of colors to color a planar graph is an exception.
  - It has been proven that every planar graph is four colorable.
  - One can also determine a planar graph is zero, one or two colorable.

Algorithm. 10.1.10. Planar Graph Coloring.

```
// Approximate algorithm to determine minimum color for planar graph G(V, E).
// Input: graph G; Output: minimum number of colors.
Algorithm AColor(G)
{
    if (V = ∅) then return 0;
    else if (E = ∅) then return 1;
    else if (G is bipartite ) then return 2;
    else return 4;
}
```

Planar Graph Coloring

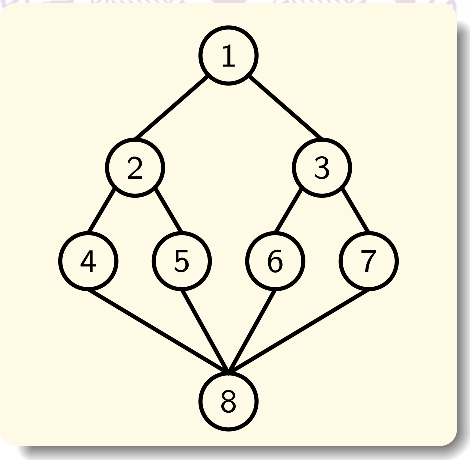
- The time complexity of Algorithm (10.1.10) is dominated by line 6 which checks if the graph is bipartite.
- Checking the bipartite property of a graph can be done in $\mathcal{O}(|V| + |E|)$ time.
- Thus, Algorithm (10.1.10) is a polynomial time algorithm.
- Note that the planar graph coloring problem is $\mathcal{NP}$-hard since three color decision problem is $\mathcal{NP}$-complete.
- Algorithm (10.1.10) does not check for three color solution, thus avoiding the long execution time by returning an approximate solution.
- Algorithm (10.1.10) is an absolute approximation algorithm because $|F^*(I) - \hat{F}(I)| \leq 1$.

Bipartite Graph

Definition. 10.1.11. Bipartite Graph.

An undirected graph $G(V, E)$ is bipartite if V can be partitioned into two disjoint sets V_1 and $V_2 = V - V_1$ such that no two vertices in V_1 are adjacent, and no two vertices in V_2 are adjacent.

- Example: The graph below is bipartite with $V_1 = \{1, 4, 5, 6, 7\}$ and $V_2 = \{2, 3, 8\}$.



- Determine if a graph is bipartite can be done in $\mathcal{O}(|V| + |E|)$ time.

Maximum Programs Stored Problem

- Given n programs and two storage devices. The ith program is of length ℓ_i and each storage device has capacity of L. The maximum programs stored problem is to determine the maximum number of programs that can be stored on these two storage devices without splitting any program.
- This maximum programs stored problem is $\mathcal{NP}$-hard because of the following theorem.
- Example: Four programs with the lengths as $(\ell_1, \ell_2, \ell_3, \ell_4) = (2, 4, 5, 6)$ and storage device capacity $L = 10$.
  - The optimal solution is 4, which can be achieved by storing programs 1 and 4 on one device, and programs 2 and 3 on the other device.

Theorem. 10.1.12.

Partition problem $\propto$ maximum programs stored problem.

- Proof please see textbook [Horowitz] p. 581.

Maximum Programs Stored Problem, II

- Assume the lengths of the n program is stored in array $\ell[1:n]$.
- Sort array $\ell[1:n]$ in nondecreasing order, $\ell[i] \le \ell[i+1]$, $1 \le i \le n$.

Algorithm. 10.1.13. Approximate algorithm to store programs.

```
   // Store n programs with ℓ[1 : n] lengths to 2 devices.
   // Input: ℓ[], n; Output: storage assignment.
Algorithm PStore(ℓ, n, L)
{
      i := 1;
      for j := 1 to 2 do { // store to device 1 then 2
            sum := 0; // Amount of device used.
            while (sum + ℓ[i] ≤ L) do {
                  write (" store program ", i, " on device ", j);
                  sum := sum + ℓ[i]; i := i + 1;
                  if i > n then return;
            }
      }
}
```

Maximum Programs Stored Problem, III

Theorem 10.1.14.

Let I be any instance of the maximum programs stored problem. Let $F^*(I)$ be the maximum number of programs that can be stored on two devices each with length L. Let $\hat{F}(I)$ be the number of programs stored using the function PStore. Then $|F^*(I) - \hat{F}(I)| \le 1$.

Proof. Consider the case that only one device with length $2L$ is used to store the programs, and p programs are stored. Then $p > F^*(I)$ and $\sum_{i=1}^{p} \ell_i \le 2L$. Let j be the largest index such that $\sum_{i=1}^{j} \ell_i \le L$. We must have $j \le p$ and that PStore assign the first j programs to device 1. Also,

$$\sum_{i=j+1}^{p-1} \ell_i \le \sum_{i=j+2}^{p} \ell_i \le L.$$

Hence, PStore assigns at least $j+1, j+2, \cdots, p-1$ to device 2. So, $\hat{F}(I) \ge p-1$ and $|F^*(I) - \hat{F}(I)| \le 1$. □

- Algorithm PStore can be extended to be a $k-1$ absolute approximation algorithm for the case of k devices.

$\mathcal{NP}$-hard Absolute Approximations

- For a majority of the $\mathcal{NP}$-hard problems, however, the polynomial absolute approximation algorithm exists if and only if the original program has a polynomial time algorithm.
- For example, we have the following theorem.

Theorem. 10.1.15.

The absolute knapsack problem is $\mathcal{NP}$-hard.

Proof. Suppose that we have a polynomial time algorithm to find $|F^*(I) - \hat{F}(I)| \le k$ for every instance I and a fixed k. Let (p_i, w_i), $1 \le i \le n$ and m be the instance. Furthermore, we assume p_i are integers. Form a new instance I' by $((k+1)p_i, w_i)$, $1 \le i \le n$, and m. Note that any feasible solution for I is also a feasible solution for I', and $F^*(I') = (k+1)F^*(I)$ and I and I' have the same optimal solutions. Since p_i are integers, the feasible solutions of I' must have difference $\ge (k+1)$ due to the way I' is constructed. Now, suppose the absolute algorithm A finds the optimal solution such that $|F^*(I') - \hat{F}(I')| \le k$, then $\hat{F}(I')$ must be $F^*(I')$. Thus, the polynomial algorithm can be used to find the optimal solution, which is not possible. □

$\mathcal{NP}$-hard Absolute Approximations, II

- Another example of absolute approximation algorithm is $\mathcal{NP}$-hard.

Theorem. 10.1.16.

Max clique $\propto$ absolute approximation max clique.

Proof. Suppose there is an absolute approximation algorithm that finds a solution such that $|F^*(I) - \hat{F}(I)| \leq k$. For a given graph $G(V, E)$ construct a new graph $G^{'}(V^{'}, E^{'})$ so that $G^{'}$ consists of $(k+1)$ copies of G connected together such that there is an edge between every two vertices in distinct copies of G. That is, if $V = \{v_1, v_2, \cdots, n\}$, then

$$V^{'} = \bigcup_{i=1}^{k+1} \{v_1^i, v_2^i, \cdots, v_n^i\},$$

$$\text{and} \quad E^{'} = (\bigcup_{i=1}^{k+1} \{(v_p^i, v_r^i) | (v_p, v_r) \in E\}) \bigcup \{(v_p^i, v_r^j) | i \neq j\}$$

Then the maximum clique size is q if and only if the maximum clique size if $G^{'}$ is $(k+1)q$. Furthermore, any clique in $G^{'}$ that is within k of the maximum clique in $G^{'}$ must contain a subclique of size q in G. Thus, we can use this absolute approximation algorithm to find the maximum clique of the original problem in polynomial time since constructing $G^{'}$ is of polynomial time. □

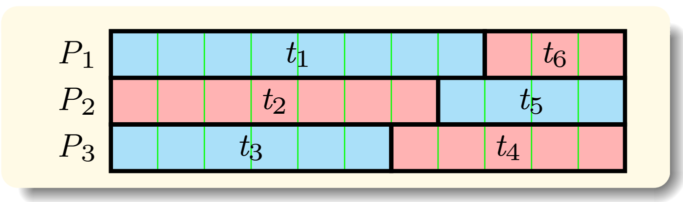
ϵ-Approximations

- Given a set of n tasks with processing time t_i each and m identical processors, the minimum finish time schedule assign the tasks to the processors to achieve the minimum finish time.
- This minimum finish time scheduling problem has been shown to be $\mathcal{NP}$-hard.
- In this section we study a polynomial time scheduling algorithm.

Definition. 10.1.17. LPT Schedule.

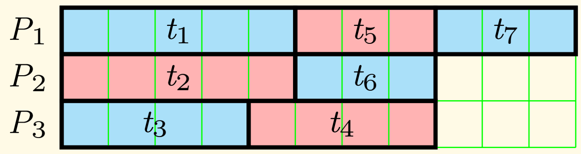
An LPT schedule is one that is the result of an algorithm that, whenever a processor becomes free, assigns to that processor a task whose processing time is the longest of those tasks not yet assigned. Ties are broken in an arbitrary manner.

- Example: $m = 3$, $n = 6$ and $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 7, 6, 5, 4, 3)$. The following is the result of a LPT schedule, which is also an optimal solution.

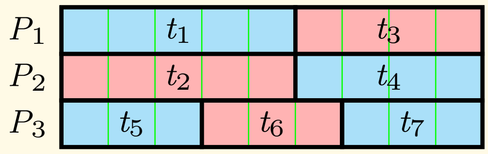


LPT Scheduling

- Example 2: $m = 3$, $n = 7$ and $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (5, 5, 4, 4, 3, 3, 3)$. The LPT schedule and the optimal schedule are shown below.



LPT schedule.



Optimal schedule.

Theorem. 10.1.18.

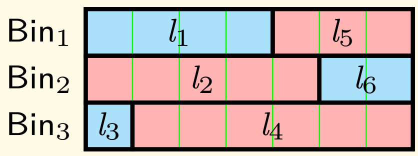
Let $F^*(I)$ be the finish time of an optimal m-processor schedule for instance I of the task scheduling problem. Let $\hat{F}(I)$ be the finish time of an LPT schedule for the same instance. Then

$$\frac{|F^*(I) - \hat{F}(I)|}{|F^*(I)|} \le \frac{1}{3} - \frac{1}{3m}. \tag{10.1.3}$$

- Proof please see textbook [Horowitz] pp. 586-587.

Bin Packing Problem

- Given n objects of l_i units each to be placed in bins with equal capacity L. The bin packing problem is to determine the minimum number of bins to accommodate all objects.
- Example: $n = 6$, $(l_1, l_2, l_3, l_4, l_5, l_6) = (4, 5, 1, 6, 3, 2)$ and $L = 7$. An optimal solution is:



- This bin packing problem has many applications. The followings are examples.
  - n tasks with t_i processing time and all tasks must be completed before deadline L. Find the minimum number of processors, m.
  - n programs with l_i lengths each to be stored on devices with capacity L. Find the minimum number of storage devices, m.

Bin Packing Problem, II

Theorem 10.1.19.

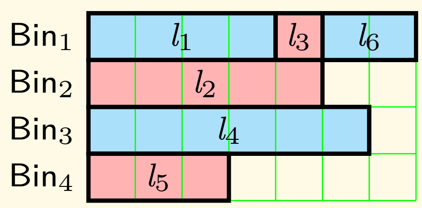
The bin packing problem is $\mathcal{NP}$-hard.

Proof. Let $\{a_1, a_2, \cdots, a_3\}$ be an instance of partition problem. A bin packing problem can be constructed by assigning $l_i = a_i$, $1 \le i \le n$, and $L = \sum_{i=1}^{n} a_i$. The minimum number of bins is 2 and the solution can be found if there is a partition for $\{a_1, a_2 \cdots, a_n\}$. Since the partition problem is $\mathcal{NP}$-hard, the bin packing problem is also $\mathcal{NP}$-hard. □

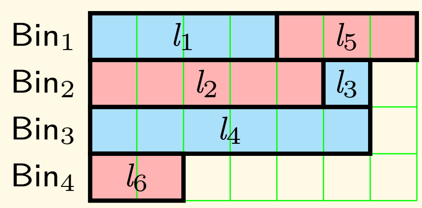
- Thus, finding the optimal solution for the bin packing problem can take long time if the number of input, n, is large.
- Heuristics can be used to find good feasible solutions.
  - These solutions are usually not optimal.

Bin Packing Problem, III

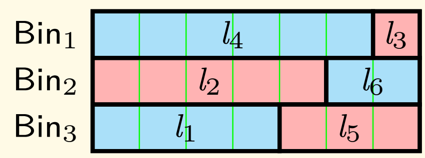
- Four heuristics are possible:
  1. First Fit (FF): Pack objects sequentially from 1 to n. All bins are initially filled to level zero. To pack object i, find the least index j such that bin j is filled to a level r, $r \le L - l_i$. Pack object i into bin j. Bin j is now filled to the level $r + l_i$.
  2. Best Fit (BF): The initial conditions on the bins and objects are the same as above. To pack object i, find the least j such that bin j is filled to a level r, $r \le L - l_i$ and is as large as possible. Pack object i into bin j. Bin j is now filled to the level $r + l_i$.
  3. First Fit Decreasing (FFD): Reorder the objects is a nonincreasing order, then use First Fit to pack the objects.
  4. Best Fit Decreasing (BFD): Reorder the objects is a nonincreasing order, then use Best Fit to pack the objects.
- Example: $n = 6$, $(l_1, l_2, l_3, l_4, l_5, l_6) = (4, 5, 1, 6, 3, 2)$, and $L = 7$.



FF.



BF.



FFD and BFD.

Bin Packing Problem, IV

Theorem. 10.1.20.

Let I be an instance of the bin packing problem and $F^*(I)$ be the minimum number of bins needed for this instance. The packing generated by either FF or BF uses no more than

$$\frac{17}{10}F^*(I) + 2 \tag{10.1.4}$$

bins. The packing generated by either FFD or BFD used no more than

$$\frac{11}{9}F^*(I) + 4 \tag{10.1.5}$$

bins. These bounds are the best possible for the respective algorithms.

Proof. See the paper: D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham, "Worst-case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal on Computing* 3, No. 4, 1974, pp. 299-325. □

- Note these are worst-case bounds.
  - For some instances, these heuristics are capable of generating the optimal solutions.
- For large n, the FFD and BFD heuristics have the smaller bounds.

$\mathcal{NP}$-hard ϵ-approximation Problems

- Many $\mathcal{NP}$-hard optimization problems their corresponding ϵ-approximation problems are also $\mathcal{NP}$-hard.
- Few examples are given here.

Theorem. 10.1.21.

Hamiltonian cycle problem $\propto$ ϵ-approximation traveling problem.

- Proof please see textbook [Horowitz] p. 591.

Theorem. 10.1.22.

Partition problem $\propto$ ϵ-approximation integer programming problem.

- Proof please see textbook [Horowitz] p. 592.

Theorem. 10.1.23.

Hamiltonian cycle problem $\propto$ ϵ-approximation quadratic assignment problem.

- Proof please see textbook [Horowitz] p. 593.

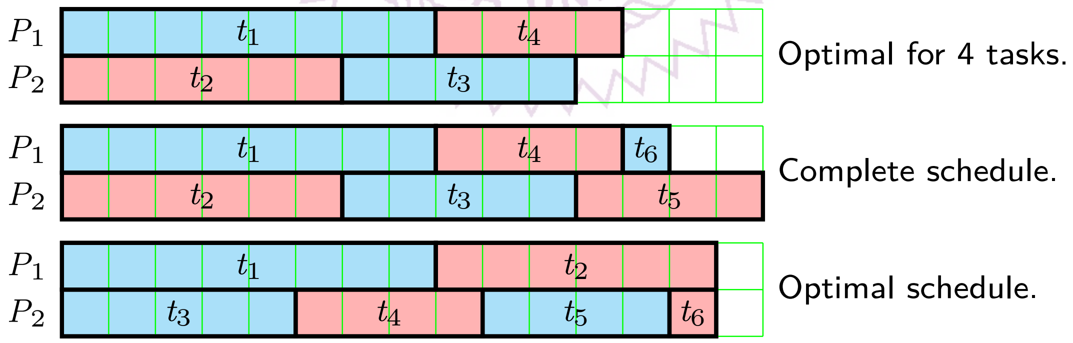
Polynomial Time Approximation Schemes

- A different approximation scheme of the independent task scheduling problem.

Algorithm 10.1.24. Scheduling by Graham

```
// Schedule n tasks with processing time t[1 : n] on m processors.
// Input: n, m, k, t[ ]; Output: task schedule.
Algorithm Graham(n, m, k, t)
{
      Find the optimal schedule for the k longest tasks ;
      Perform LPT scheduling for the rest of the tasks ;
}
```

- Example: $n = 6$, $m = 2$, $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 6, 5, 4, 4, 1)$.



Polynomial Time Approximation Schemes, II

Theorem. 10.1.25. Graham Scheduling.

Let I be an m-processor instance of the scheduling problem. Let $F^*(I)$ be the finish time of an optimal schedule for I and let $\hat{F}(I)$ be the finish time of the schedule generated by the algorithm Graham. Then,

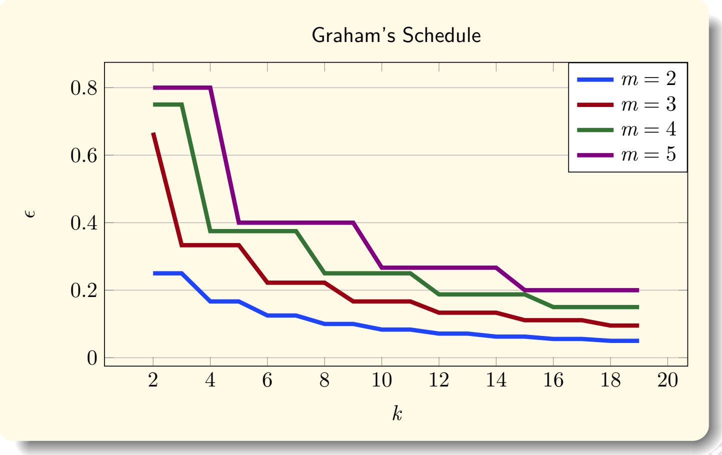
$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \le \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}. \tag{10.1.6}$$

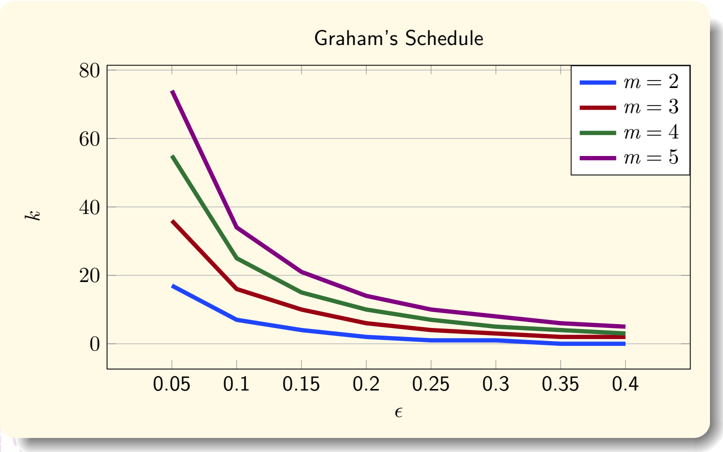
- Proof please see textbook [Horowitz] pp. 598-599.
- Given any ϵ, one can find

$$k \ge \frac{m-1}{\epsilon} - m \tag{10.1.7}$$

then the schedule generated is $\epsilon \cdot F^*(I)$.

Polynomial Time Approximation Schemes, III





- In the Graham's algorithm ϵ can be made small, but then k can be large.
- The first part of the Graham's algorithm, line 4, can take $\mathcal{O}(m^k)$ time.
- Before applying Graham's algorithm, the input needs to be sorted, time complexity $\mathcal{O}(n \lg n)$.
- Thus, the total time complexity is $\mathcal{O}(n \lg n + m^k)$.
  - This is not exactly a polynomial time algorithm for large k.

Solving $\mathcal{NP}$-complete Problems

- Finding solutions for $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems can take formidable amount of time.
- Approximation algorithms do not attempt to find the optimal solution but to find a feasible solution close to the optimal one.
  - The bound, if can be derived, is of great value.
- Basic methods for approximate algorithms are the ones we have studied
  - Divide-and-conquer
  - Greedy method
  - Dynamic programming
  - Local search instead of all space search
  - The key is the bounding function.
- Other heuristic approaches have been developed
  - Construction heuristics
  - Local search heuristics
  - Simulated annealing
  - Genetic algorithms
  - Tabu search

Summary

- Approximation algorithms.
- Absolution approximations.
  - Planar graph coloring problem.
  - Maximum programs stored problem.
  - $\mathcal{NP}$-hardness.
- ϵ-approximations.
  - Scheduling problem.
  - Bin packing problem.
  - $\mathcal{NP}$-hardness.
- Polynomial time approximation scheme.
  - Graham's algorithm.