CO454 - Scheduling Theory

(Notes Scans)

University of Waterloo

Nicholas Pun

Spring 2018

Contents

Summary

Sections 1 to 8 - See Section Headings (Note: For Section 8, apparently I didn't trust one of the proofs, I've included the offending proof somewhere near the end of the section)

I usually take notes on loose leaf in-class and then copy them into a dedicated notebook (with some colour-coding and rewording and whatnots) after the class. I guess I never got around to copying a good chunk of the notes, so below are just raw lecture notes.

In case it matters, the colour coding is as follows: Red for theorems, propositions, lemmas, etc., Black for definitions, Blue for generic descriptions

Lecture 16 - $P||\sum_{C_j}$, $R||\sum_{C_j}$

Lecture 17 - $R|\text{pmtn}|C_{\max}$, $O|\text{pmtn}|C_{\max}$

Lecture 18 - Finishing off $O|\text{pmtn}|C_{\max}$, Travelling Salesman Problem

Lecture 19 - Travelling Salesman Problem, Christofides Algorithm

Lecture 20 - $R||C_{\max}$

Lecture 21 - $R||C_{\max}$, Shop Scheduling, Flow Shop

Lecture 22 - $F2||C_{\max}$

Lecture 23 - $F||C_{\max}$

Lecture 24 - Sevast'janov-Steinitz Lemma (Steinitz lemma in coordinate form)

The main reference was *Scheduling: Theory, Algorithms, and Systems* by Pinedo. .

(1) Introduction

1 Introduction Examples:

1) Scheduling in CPU.

We want to be able to schedule the various processes which are running concurrently on a computer (i.e. Allocate chunks of CPU time towards each process)

We can have multiple objectives:
- Minimize average waiting time, or
- Maximize throughput.

Our tasks (i.e. The processes) have multiple properties:
- Processing time - Time needed to finish the proc
- Priorities - Weights indicating importance

Further,
- Tasks need not be known beforehand; in fact, neither does the processing time (These are known as real-time problems, we will mainly focus on problems with known quantities)
- Tasks need not run in contiguous chunks. They can be interrupted, and resumed at a later time. (This is known as preemption)

↳ Ex. 2, 3

2) Production Planning.
The Carts-for-U factory produces 3 types of carts:
- Shopping Carts - Airport Trolleys - Cargo Carts
Production of each type of cart goes through the same stages
- Producing metal skeleton
- Casting Metal
- Attaching parts to the skeleton
- Packaging of carts
Each stage consists of the same machines, not necessarily identical (i.e. could differ in speed or other capabilities)

We could have multiple objectives:
- Minimize # of late demands
  (The "demand" for each cart is the # required and due date)
- Minimize the "total penalty" of late demands
  i.e. Some demands may not be met, but we want to minimize (lateness)
- Minimize maximum lateness of demands.

3) Gate Assignments at an airport
An airport has n gates where planes land and take off. We assume all gates are identical. and there is a hanger where planes can wait.
Suppose plane j lands at time r_j, departs at time d_j and takes time $P_{j,i,i'}$ to go from gate i to i'. How should planes be assigned to gates to minimize total airline delay?

1.1

Def'n (Job, Machine)

Every scheduling problem consists of:

- Jobs - Entities that require resources. We will commonly use j, k, l to denote jobs
- Machines (MCs) - Resources that jobs require. We will use i, i', i'', etc. to denote MCs.

- Jobs:

A job j typically is associated with:

- Processing time (P_{ij})
  This is the time taken by machine i to process job j.
  Note:
  1) Always ≥ 0
  2) Can be infinity (i.e. This machine cannot process job j)
  3) If machines are identical, then j's processing time is independent of i, and we will use P_j to denote j's processing time.
- Release Date (r_j)
  Time when j enters the system.
  Note: j can only be scheduled on a machine at or after its release date
- Due Date (d_j)
  Time by which j must be completed
  Note: Could be a soft deadline that can incur penalty in our objective fn. or hard deadline
- Weights (w_j)
  Indicates the importance / priority of job j.
  $w_j \geq 0$ unless otherwise stated.

→ Machines

- Machines:

We can have multiple settings/environments:

- Single M/C
- Multiple M/C
  - Identical machines
  - Non-identical machines
- Shop Scheduling: Multiple machines and a job consists of various operations that need to be performed on various machines (Could be a sequence - See Ex. 2)

- Objective

1) Sum of completion times:

Let C_j denote the completion time of job j. We write:

$$\min \sum_j C_j$$

2) Weighted Completion time:

$$\min \sum_j w_j C_j$$

3) Total Waiting time:

$$\min \sum (C_j - r_j)$$

4) Maximum lateness:

Let $L_j := C_j - d_j$ be the lateness of j.

$$\min \max_j L_j$$

5) No. of late jobs.

Define $U_j := \begin{cases} 1 & C_j > d_j \\ 0 & \text{otherwise} \end{cases}$

$$\min \sum U_j$$

(2) Basic Single Machine Setup

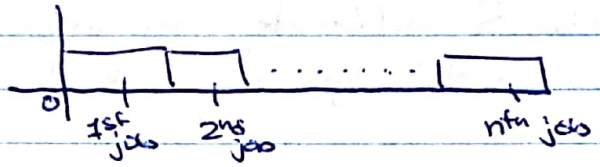
~~Basic Single Machine Setup~~

Basic Single Machine (BSM) Setup

The BSM setup consists of:

- 1 M/C
- All jobs are released at time 0.
- No precedence constraints between jobs forcing that some jobs must complete before others

Objective: Minimize total completion time $\sum C_j$.



$$\sum C_j = (P_{1st\,job}) \cdot n + (P_{2nd\,job}) \cdot (n-1) + \ldots + (P_{nth\,job}) \cdot 1 \quad (1)$$

Defn 2.1 (Shortest Processing time - SPT)
The SPT rule is to schedule jobs in increasing order of processing time (breaking ties arbitrarily)

Theorem 2.1:
In the BSM setup, a schedule minimizes $\sum C_j$ iff it is produced via the SPT rule. (called an "SPT schedule")

(Proof)
We will prove this by an interchange argument.

($\Rightarrow$) Suppose we have a schedule S that minimizes $\sum C_j$, but is NOT an SPT schedule

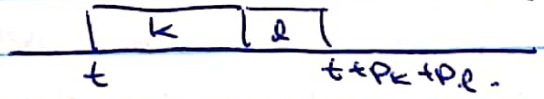
(Cont'd

Proof (cont'd)

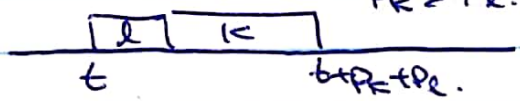
Then, there must be jobs j, j' such that $p_j > p_{j'}$, but j is scheduled before j'. (We call this an inversion). In fact there must be 2 consecutive jobs k, l s.t. $p_k > p_l$, but k is scheduled immediately before l in S. (Consider moving inwards from j, j' until you find k, l).

Let S' be the schedule obtained from S by interchanging k, l

S:



S':



Consider the change in the objective value.

$\sum_j C_j^{S'} - \sum_j C_j^S$

($C_j^{S'}$ denotes completion time of j in S')

$= (C_k^{S'} + C_l^{S'}) - (C_k^S + C_l^S)$

Since $C_j^{S'} = C_j^S \quad \forall j \neq k, l$

$= t + p_k + p_l + t + p_l - (t + p_k + t + p_k + p_l)$

$= p_l - p_k < 0$, Since $p_k > p_l$.

which contradicts that S is an optimal schedule.

Cont'd

[Proof] Con't)
So, we have shown that:
- Every optimal schedule is an SPT schedule
- All SPT schedules have the same objective value — We see this because we can always move from SPT schedule to another by interchanging pairs of jobs with equal processing times. The objective value is unchanged

These 2 statements imply that that all SPT schedules are optimal (which is exactly the reverse direction) □

Objective: Minimize $\sum w_j C_j$
(Note: If $w_j = p_j$, then this problem reduces to the previous one)

Def'n 2.2 (Weighted Shortest Processing time - WSPT)
The weighted shortest processing time rule is to schedule jobs in decreasing order of w_j/p_j.
(Known as the density of job j)

Theorem 2.2:
In the BSM setup, a schedule minimizes $\sum w_j C_j$ iff it is a WSPT schedule

[Proof]
We will use an interchange argument.

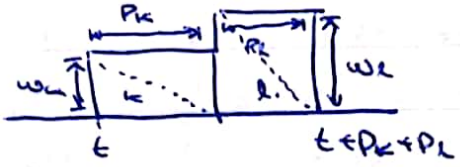
($\Rightarrow$) Let S be an optimal schedule that minimizes $\sum w_j C_j$, but is NOT a WSPT schedule.

Cont'

[Proof] (cont'd)

There must be consecutive jobs k, l s.t. k is scheduled immediately before l in S, but $w_k/p_k < w_l/p_l$.

S:



S':

Let S' be obtained from S by interchanging k, l. Then we get:

$$\sum w_j C_j^{S'} - \sum w_j C_j^{S}$$

$$= w_k C_k^{S'} + w_l C_l^{S'} - (w_k C_k^{S} + w_l C_l^{S})$$
$$= w_k (t + p_k + p_l) + w_l (t + p_l) - (w_k (t + p_k) + w_l (t + p_k + p_l))$$
$$= w_k p_l - w_l p_k$$
$$= p_l p_k \left(\frac{w_k}{p_k} - \frac{w_l}{p_l}\right) < 0, \text{ since } \frac{w_k}{p_k} < \frac{w_l}{p_l}.$$

which contradicts that S is an optimal schedule.

This also shows that interchanging equal density jobs preserves the objective value. Hence, all WSPT schedules have the same objective value, and so are optimal schedules; completing the proof.

□

Objective: Minimize maximum lateness

Recall that lateness is:

$$L_j := C_j - d_j$$

So, we minimize:

$$L_{max} := \max_j L_j$$

Def'n 2.3 (Earliest Due Date Rule - EDD)

The EDD rule is to schedule jobs in increasing order of d_j.

Theorem 2.3

The EDD rule minimizes L_{max} in the BSH setup

Note: There are optimal schedules that minimize L_{max}, but NOT EDD schedules.

[Proof]

Let S be an optimal schedule, but is NOT an EDD schedule. Then, there are consecutive jobs k, l s.t:

1) k is scheduled immediately before l in S.
2) $d_k > d_l$

Let S' be constructed from S by interchanging k, l. (We want to show $L_{max}^{S'} \leq L_{max}^{S}$)

We know that:

- $\forall j \neq k, l$, $L_j^{S'} = L_j^S$ (Since $C_j^{S'} = C_j^S$)
- $L_k^S = C_k^{S'} - d_k = C_j^S - d_k < C_l^S - d_l = L_l^S$
  $\Rightarrow L_k^{S'} < L_l^S$
- $L_l^{S'} = C_l^{S'} - d_l \leq C_l^S - d_l = L_l^S$

So,

$$\max(L_k^{S'}, L_l^{S'}) \leq L_l^S \leq \max(L_k^S, L_l^S)$$

$$\Rightarrow \max_j L_j^{S'} \leq \max_j L_j^S$$

□

Objective: Minimize f_{max}

We have a non-decreasing function:

$f_j : \mathbb{R}_{\geq 0} \to \mathbb{R}$ for each job j.

Where:

$f_j(C_j)$, $C_j :=$ Completion time for j

is the cost incurred for job j.

We want a schedule that minimizes:

$$f_{max} := \max_j f_j(C_j)$$

(Note: With $f_j(x) = x - d_j$, we retrieve our previous problem of minimizing max. lateness)

Motivation for LCL Rule:

We define:

$f^*_{max}(J) :=$ objective value of an optimal schedule for a set of jobs J

And also "relax" our definition of schedule:

A schedule specifies when each job in J starts (and hence finishes) and should be s.t. at most one job is processed at any point of time.

(Note: This allows us to have gaps between jobs)

We notice that:

An optimal schedule for J completes at time: $\sum_{j \in J} p_j$.

→ continued

(continued)
So, if r is the last job scheduled in an optimal schedule for J, then:

$$f^*_{max}(J) \geq f_r(C_r)$$
$$= f_r\left(\sum_{j \in J} p_j\right)$$
$$\geq \min_{k \in J} f_k\left(\sum_{j \in J} p_j\right) \quad (1)$$

And, for any $k \in J$,

$$f^*_{max}(J) \geq f^*_{max}(J \setminus \{k\}) \quad (2)$$

Since dropping jobs cannot increase the objective value.
So, combining (1) and (2), we get:
Let $l \in J$ be s.t.:

$$f_l\left(\sum_{j \in J} p_j\right) = \min_{k \in J} f_k\left(\sum_{j \in J} p_j\right)$$

Then,

$$f^*_{max}(J) \geq \max\left(f_l\left(\sum_{j \in J} p_j\right), f^*_{max}(J \setminus \{l\})\right)$$

Notice that this suggests that l should be the last job since $C_l = \sum_{j \in J} p_j$, and this also gives us that:

$$f^*_{max}(J) = \max\left(f_l\left(\sum_{j \in J} p_j\right), f^*_{max}(J \setminus \{l\})\right)$$

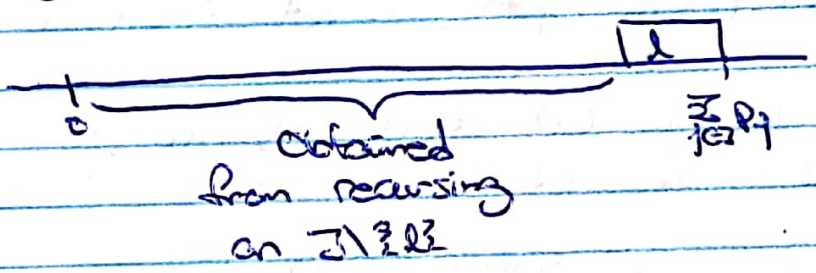
$\hookrightarrow$ def'n

Defn 2.4 (Lowest Cost Last Rule - LCL)

- $J \leftarrow \{1, \ldots, n\}$
- While $J \neq \emptyset$:
  - Choose $l \in J$ s.t.
    $$f_l\left(\sum_{j \in J} p_j\right) = \min_{k \in J} f_k\left(\sum_{k \in J} p_j\right)$$
  - Schedule l last among J (so that l completes at time $\sum_{j \in J} p_j$)
  - $J \leftarrow J \setminus \{l\}$

Note:

(i) Visually, we get the following:



(ii) There are 2 ways of constructing schedules: from time 0 to $\sum_{j \in J} p_j$ (i.e. front to back) or from time $\sum_{j \in J} p_j$ to time 0 (i.e from back to front). In this case, scheduling from the back is more convenient.

Thm Theorem

Theorem 2.4

In the BSM setup, every LCL schedule for a job-set J has objective value $\leq f^*_{max}(J)$, hence every LCL schedule minimizes f_{max}.

[Proof]

We prove by induction on $|J|$.

(BC) If $|J|=1$, then the statement is trivially true

(IH) Now, suppose this statement holds when $|J|=n$.

(IC) Consider job-set J with $|J|=n+1$.

Let l be as in LCL s.t.:

$$f_l\left(\sum_{j\in J} p_j\right) = \min_{k\in J} f_k\left(\sum_{j\in J} p_j\right)$$

And the objective value of the LCL schedule will be:

$$\max\left(f_l\left(\sum_{j\in J} p_j\right),\ f_{max} \text{ in schedule constructed by LCL for } J\setminus\{l\}\right)$$

$$\leq \max\left(f_l\left(\sum_{j\in J} p_j\right),\ f^*_{max}(J\setminus\{l\})\right) \quad \text{(by (IH))}$$

$$\leq f^*_{max}(J)$$

which completes the induction step

□

(3) Time Complexity I - Big-O Notation

3 Time Complexity - Big-O Notation

Def'n 3.1 (Running Time, Input Size)
We define the running time to be the # of elementary operations (i.e. arithmetic ops, comparisons, assignments) executed by an algorithm.
This will commonly be represented as a function of input size, which is the # of bits needed to represent the input

Def'n 3.2 (Big-O Notation)
Given $f: \mathbb{R}_+ \to \mathbb{R}_+$, $g: \mathbb{R}_+ \to \mathbb{R}_+$, we say that:
$f(n) \in O(g(n))$
if there exists constants $c \geq 0$, $n_0 \geq 0$ s.t.
$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

Ex:
1) $n \in O(2n)$ since $n \leq 1 \cdot 2n \quad \forall n \geq 0$
$2n+10 \in O(n)$ since $2n+10 \leq 3 \cdot n \quad \forall n \geq 10$

2) $3n \in O(n^2)$ since $3n \leq 3n^2 \quad \forall n \geq 0$
But,
$n^2 \notin O(3n)$
In general:
If $0 \leq \alpha_1 < \alpha_2$ and $\beta_1, \beta_2 \geq 0$:
$\beta_1 n^{\alpha_1} + \beta_2 \in O(n^{\alpha_2})$
But,
$n^{\alpha_2} \notin O(n^{\alpha_1})$

Continued

Ex: Continued)

3) $n\log_2(n) \in O(n^2)$, but not $O(n)$

4) $\log_2 n = \frac{\log_c n}{\log_c 2} = \frac{\log_d n}{\log_d 2}$

So, in general, for any $c, d \geq 1$:

$$\log_c n \in O(\log_d n)$$

5) $2^n \in O(3^n)$, but $3^n \notin O(2^n)$

6) $f(n) \in O(c) \equiv f(n) \leq c \quad \forall n \geq n_0$.

(i.e. This is shorthand for saying $f(n)$ is bounded by a constant)

Note:

Using the notation in (6), we write for: "$f(n)$ is bounded by some polynomial of n":

$$f(n) \in O(n^{O(1)})$$

Def'n 3.3 (Efficient)

An algorithm with running time $f(n)$ is efficient, if $f(n)$ is bounded by some fixed polynomial of n.

→ Analysis of our algorithms.

For SPT, WSPT, EDD, these algorithms involve sorting n numbers (jobs). So these require $|O(n \log n)|$ time

For LCL, we perform the analysis below:
There are n jobs, so:
- We perform n iterations (exactly)
- Each iteration $i = 1, \ldots, n$ requires finding the minimum of $(n-i+1)$ $f_j(\cdot)$'s at a given time t.

We can consider $f_j(\cdot)$ to be an elementary operation, and so LCL will have $|O(n^2)|$ running time:

Note:
Trying out all possible solutions is NOT an efficient algorithm, as this requires $n!$ time, but $n! \in O((\frac{n}{2})^n)$, and so it is NOT bounded by a poly in n.

(4) Approximation algorithms

Notation:

We will use Graham et al's notation for describing problems using triplet notation:

$$\alpha \mid \beta \mid \gamma$$

- α: machine environment
- β: Side Constraints
- γ: Objective (will usually be a min. objective, so we will omit 'min' :)

Ex:

α (M/C environment)

- 1: Denotes single M/C
- P: Multiple identical M/C's
- P2 or Pm: m identical M/C's

(... and more to come)

γ (Objective Function)

- $\sum C_j$, $\sum w_j C_j$, L_{max}, f_{max} (Things we have seen)
- $\sum f_j(C_j)$, $C_{max} := \max_j C_j$ (Other possible objective functions)

Note: This is trivial with a single machine

β (Side constraints)

- This can be empty (which is exactly what we've seen so far in the BBL setup)
  i.e. $1 \mid\mid \sum C_j$, $1 \mid\mid \sum w_j C_j$, $1 \mid\mid L_{max}$, $1 \mid\mid f_{max}$.
- There are other constraints we will see:
  - r_j: Release Dates
    Specifies that job j becomes available at time r_j
    (i.e. Cannot schedule j at time $< r_j$)

Continues

Ex: (continued)

- Prec: Precedence Constraint

A precedence constraint $j \to k$ specifies that j must complete before k can be started

Note:

Precedence is a transitive relation i.e. $j \to k$, $k \to l \Rightarrow j \to l$.

This also implies that we cannot have a cycle of precedence constraints:

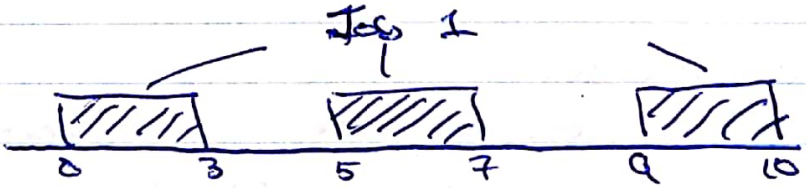
i.e. $j_1 \to j_2 \to \ldots \to j_k \to j_1$ is not possible)

Precedence constraints are represented by DAGs (i.e. Nodes := jobs, Arcs := Constraints)

- Pmtn/Pmp: Preemption

Can interrupt a job and resuming working on it later.

i.e. We can have:



as a valid schedule.

$1|prec|f_{max}$:

By our notation, this is the problem:

1 MC, min. f_{max}, and subject to precedence constraints $j \to k$.

We define:

Given a set J of jobs, let

$$L(J) := \{j \in J : j \text{ has no successors in } J\}$$
$$:= \{k \in J \text{ s.t. } j \to k\}$$

Observe: Only jobs in $L(J)$ can appear at the end of a schedule for J.

Defn 4.1 (Modified LCL for $1|prec|f_{max}$)

- $J \leftarrow \{1, \ldots, n\}$
- while $J \neq \emptyset$:
  - find $l \in L(J)$ s.t. $f_l(\sum_{j \in J} p_j) = \min_{k \in L(J)} f_k(\sum_{j \in J} p_j)$
  - Schedule l last
  - $J \leftarrow J \setminus \{l\}$ (and update $L(J)$).

Theorem 4.1:

The above algorithm gives an optimal schedule for $1|prec|f_{max}$.

[Proof]

Similar to the proof of Thm 2.4. □

↳ $1||r_j, \text{pmtn}|\sum C_j$

$1|r_j, prmtn|\sum C_j$

We have:

1 M/C, min $\sum C_j$, and with the constraints of release dates and allowing preemption.

(Note: without r_j, preemption is unnecessary)

Defn 4.2 (Shortest Remaining Processing Time Rule - SRPT)

At each point of time, schedule the job available with smallest remaining processing time, preempting a job if a job with smaller processing time is released

Theorem 4.2:

SRPT generates an optimal schedule for $1|r_j, prmtn|\sum C_j$.

[Proof]

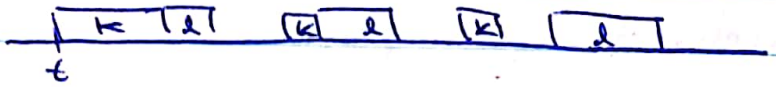
Let S^* be an optimal schedule that is NOT an SRPT schedule.

Then, at some point of time t, S^* schedules some job k with remaining proc time x_k, but there was another available job l with remaining proc time $x_l < x_k$ and $x_l > 0$

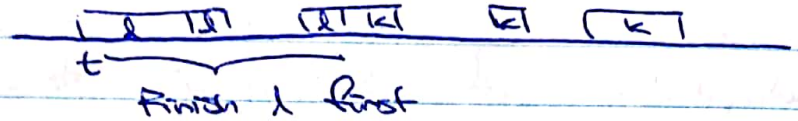
Continued

[Proof] (Continued)

S^*: (Placement of k, l arbitrary)



S:



We want to be careful with our interchange argument (Else we might affect the completion time of other j's)

Consider portions of time after t where k or l are scheduled in S^*, and interchange k, l by scheduling l in the first x_l of the time, followed by k to get schedule S.

This ensures that $C_j^S = C_j^{S^*}$ $\forall j \neq k, l$, and also $C_k^S = \max(C_k^{S^*}, C_l^{S^*})$.

Claim 1: $C_l^S < C_k^{S^*}$

Since k is processed for time $x_k > x_l$.

Claim 2: $C_l^S < C_l^{S^*}$

l is not scheduled at time t in S^*, so $C_l^{S^*}$ will be pushed back.

So, $C_l^S < \min(C_k^{S^*}, C_l^{S^*})$

$\Rightarrow C_l^S \cdot C_k^S < C_l^{S^*} \cdot C_k^{S^*}$

Completing the contradiction

□

$1|r_j|\sum C_j$:

Now, we look at the previous problem, but without preemption

Notice that SRPT generalizes SPT, so:

Q: Can we come up with generalizations or adaptations of SRPT/SPT to solve $1|r_j|\sum C_j$?

Attempt 1!

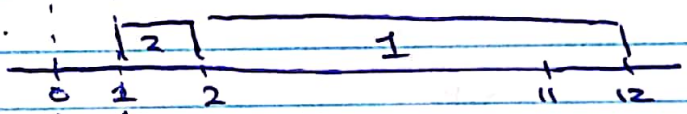
At each time t, schedule the available job with shortest processing time, but do not preempt

However, consider:

S:

$\sum C_j^S = 10+11 = 21$

S^*:



$\sum C_j^{S^*} = 2+12 = 14$.

Job 2: $r_j = 1$, $p_j = 1$

Job 1: $r_j = 0$, $p_j = 10$

Notice that S^* is more optimal than S, so our algorithm is no longer optimal.

The problem is: A small job got released soon after a big job was started

→ Attempt 2

Attempt 2!

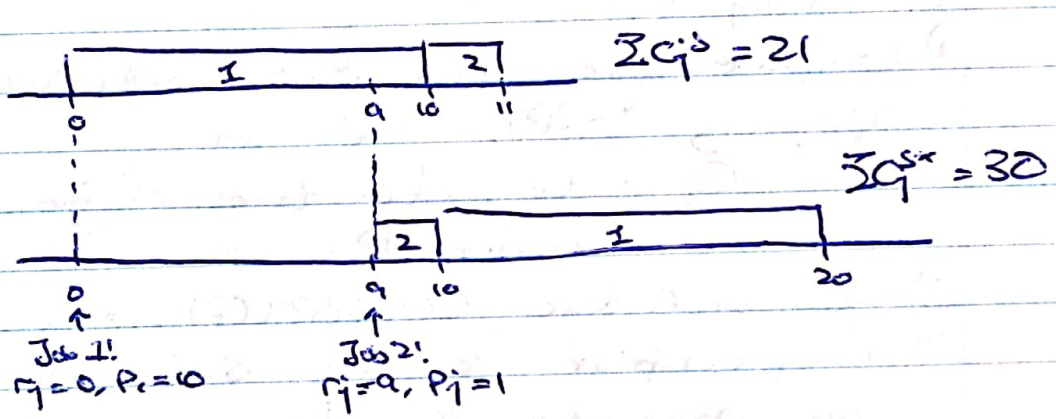
We might try to fix our previous example.

Consider:

Sort jobs in increasing p_j order, and process them in this order, waiting for job j, if j is not released by the completion time of the previous job.

(Notice that this suggests that it may be beneficial to leave the machine idle even if there are available jobs - called "enforced idle time")

However, Consider:



Again, our algorithm doesn't work since S is more optimal than S^*.

(An optimal solution to $1|r_j|\sum C_j$ is not known!)

What can we do?

- Abandon Efficiency
- Abandon Optimality.

→ Continued

Defⁿ 4.3 (α-approximation algorithm)
An α-approximation algorithm is an efficient algorithm that returns a schedule S whose objective value ≤ α · (optimum) (where optimum is the optimal objective value for our given problem, and we will commonly write it as OPT)
Further, α is known as the "approximation ratio" or "approximation guarantee".

We will design a 2-approximation algorithm for $1|r_j|\sum C_j$:
1) Compute the preemptive schedule P by applying SRPT, and let
C_j^P := Completion time of job j under P.
2) Run procedure CONVERT(P), which takes a preemptive schedule and returns a non-preemptive schedule

CONVERT(P):
1) Sort jobs in increasing order of C_j^P:
$C_1^P \le C_2^P \le \dots \le C_n^P$
2) Schedule jobs "non-preemptively" in the order defined above, and output the resulting schedule
(By "non-preemptively", we mean:
Start Job j at time
$t := \max(\text{Completion time of } j-1, r_j)$).

Theorem 4.3:
SRPT + CONVERT is a 2-approx. algorithm for $1|r_j|\sum C_j$.

To prove this, we need to following:

Claim 4.4:
Let $OPT :=$ optimal value for $1|r_j|\sum C_j$.
Then:

$$OPT \geq \sum C_j^P$$

[Proof]
We've proved that SRPT gives an optimal schedule for $1|r_j, pmtn|\sum C_j$ (Theorem 4.2). So:

$$\sum C_j^P = OPT_{1|r_j, pmtn|\sum C_j}$$

And:

$$OPT_{1|r_j, pmtn|\sum C_j} \leq OPT$$

Since any schedule for $1|r_j|\sum C_j$ is feasible for $1|r_j, pmtn|\sum C_j$. The preemptive problem is a relaxation of $1|r_j|\sum C_j$. □

Lemma 4.5:
For every job j (in the ordering after step (1) of CONVERT):

$$C_j^S \leq 2C_j^P$$

[Proof]

[Proof]

We begin with the following claim:

For every jobs j (under the order after Step 1):

$$C_j^P \geq \max_{k=1,\ldots,j} r_k \quad \text{, and} \quad (1)$$

$$C_j^P \geq \sum_{k=1}^{j} p_k \quad (2)$$

Since by definition:

$$C_j^P \geq C_k^P \quad \forall\, k=1,\ldots,j$$

$$\geq r_k \quad \forall\, k=1,\ldots,j$$

$$\geq \max_{k=1,\ldots,j} r_k$$

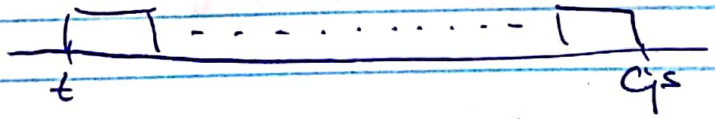
And, by time C_j^P, all jobs $1,\ldots,j$ have completed in the preemptive schedule P

So,

$$C_j^P \geq \sum_{k=1}^{j} p_k$$

Now, in schedule S, let $t = \max_{k=1,\ldots,j} r_k$, and [redacted], consider the interval $[t, C_j^S]$

S:



Goal: We want to bound this interval by C_j^P

→ Continued

[Proof] (continued)
Notice that:
- There is no idle time in $[t, C_j^S]$. Since the only way this can happen is if we are waiting for a job to be released. However, by time t, all jobs $1, \dots, j$ have been released and further, we would complete j before waiting for any other job l with $l > j$.
- This also means that in $[t, C_j^S]$, we are only processing jobs $1, \dots, j$.

Hence,

$$C_j^S = t + (\text{length of } [t, C_j^S])$$

$$\leq \max_{k=1,\dots,j} r_k + \sum_{k=1}^{j} p_k$$

(By the 2nd point)

$$\leq \underbrace{C_j^P}_{(1)} + \underbrace{C_j^P}_{(2)} = 2C_j^P \qquad \square$$

Note:
If we can solve a preemptive version of a non-preemptive problem, CONVERT and this lemma gives a strong guarantee for the completion times.

Now, we can prove Theorem 4.3:
[Proof] (of Theorem 4.3):
By Lemma 4.5, we get that:

$$\sum_j C_j^S \leq 2 \sum_j C_j^P$$

And by claim 4.4:

$$\sum_j C_j^S \leq 2 \sum_j C_j^P \leq 2 \cdot OPT.$$

$\square$.

Runtime Analysis:

SRPT requires:

- $O(n^2)$:
  - The schedule changes at most $2n$ time points corresponding to completion times + release dates of jobs
  - At each of these time points, we go through the list of remaining jobs in $O(n)$ time

  So, $O(n \cdot 2n) = O(n^2)$
- $O(n \log n)$
  - By using a min-heap or a sorted list, we can reduce the $O(n)$ lookup to $O(\log n)$

  So, $O(n \cdot \log n)$

Then, our 2-approx. alg takes:

SRPT + CONVERT

$O(n \log n)$ $O(n \log n)$ (sorting)

$= O(n \log n)$ time.

$1|r_j|\sum w_jC_j$ and $1|r_j, prmtn|\sum w_jC_j$

We know that $1|r_j, prmtn|\sum w_jC_j$ is a relaxation of $1|r_j|\sum w_jC_j$ and we have a general strategy to tackle preemptive problems and convert them to their non-preemptive counterparts. So, we want an α-approx. algorithm A for $1|r_j, prmtn|\sum w_jC_j$ so that A + CONVERT gives an 2α-approx algorithm for $1|r_j|\sum w_jC_j$.

Preemptive WSPT:
At each point of time, schedule available jobs with largest w_j/p_j ratio, preempting as necessary.
Equivalently:
1) Order jobs in decreasing order of density
2) Schedule jobs, preemptively, in this order
(i.e. At each point of time, schedule jobs that came earliest in this order)

Note: Preemptive WSPT does not coincide with SRPT, even when all jobs have weight 1, since p_j is the original processing time for j.

Theorem 4.6:
Preemptive WSPT is a 2-approx. algorithm for $1|r_j, prmtn|\sum w_jC_j$.

→ proof.

[Proof]

Let $OPT := OPT_{1|r_j, pmtn|\sum w_j C_j}$

Let jobs be ordered $1, \ldots, n$ so that:

$$\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \geq \cdots \geq \frac{w_n}{p_n}$$

and S be the schedule returned by preemptive WSPT

Observe that $OPT \geq \sum_{j=1}^{n} w_j C_j$ (1). Since the completion time $C_j \geq r_j$ in any schedule and, $OPT \geq OPT_{1|pmtn|\sum w_j C_j}$, which is the relaxation of the problem, with all release dates of 0. (Any schedule for $1|r_j, pmtn|\sum w_j C_j$ is a schedule $1|pmtn|\sum w_j C_j$)

Further $1|pmtn|\sum w_j C_j \equiv 1||\sum w_j C_j$, since preemption is unnecessary when $r_j = 0 \; \forall j$. So we can use WSPT, which produces an optimal schedule with objective value $OPT_{1||\sum w_j C_j}$. The schedule will also be exactly in the order $1, \ldots, n$, so:

$$\begin{aligned} OPT &\geq OPT_{1|pmtn|\sum w_j C_j} \\ &= OPT_{1||\sum w_j C_j} \\ &= \sum_{j=1}^{n} w_j \left(\sum_{k=1}^{j} p_k \right). \end{aligned} \quad (2)$$

Continued

[Proof] (Continued)
We are nearly done. Consider schedule S and job J:

S:

In the interval $[r_j, C_j^S]$, there is never any idle time, since job j is available. And, only jobs k $(1 \le k \le j)$ can be scheduled, since any job l with $l \ge j+1$ would contradict our ordering.
So:

$$C_j^S \le r_j + \sum_{k=1}^{j} p_k \quad \forall j = 1, \ldots, n$$

Then:

$$\sum_{j=1}^{n} w_j C_j^S \le \sum_{j=1}^{n} w_j \left(r_j + \sum_{k=1}^{j} p_k\right) \quad \text{(By above)}$$

$$= \sum_{j=1}^{n} w_j r_j + \sum_{j=1}^{n} w_j \left(\sum_{k=1}^{j} p_k\right)$$

$$\le \underset{(1)}{OPT} + \underset{(2)}{OPT} = 2 \cdot OPT.$$

□.

And so, Preemptive WSPT + CONVERT gives an algorithm for $1|r_j|\sum w_j C_j$.

Corollary 4.7:
Preemptive WSPT + CONVERT is a 4-approx. alg for $1|r_j|\sum w_j C_j$

5 Basic Computational Complexity

(5) Introduction to Computational Complexity

5.1

Defⁿ (P)

We define $\underset{\sim}{P}$ to be:

$P := \{$ All problems π s.t. there is a polytime algorithm for solving π $\}$

Ex: $1|L_{max} \in P$ since we can use EDD

Defⁿ 5.2 (Decision Problem)

A decision problem is one that can be posed as a YES-NO question of the input.

The decision problem of an optimization problem asks:

"Is there a solution of objective value at most K?"

Ex:

Suppose our problem is $1||L_{max}$, the decision version of the problem asks:

"Is there a schedule S with $L_{max} \leq K$?"

(i.e. $1||L_{max} \leq K$)

Notation:

We sometimes denote a decision problem by its collection of YES instances.

Ex:

$$\pi_{(1||L_{max} \leq K)} = \{\text{All instances of } (1||L_{max} \leq K) \text{ s.t. } \exists \text{ schedule of } L_{max}\text{-value at most } K\}$$

Polynomial Reduction:

Def'n 5.3

Given 2 problems A, B, we say that B is polytime reducible to A denoted by $B \leq_p A$, if given an algorithm T for A, we can solve B by using a polynomial number of calls to T + a polynomial # of elementary operations.

Remark 5.1:

Suppose $B \leq_p A$, then:

1) If $A \in P$, then $B \in P$
   (i.e. Polytime alg. for A admits a polytime alg. for B)
2) If B does not have a polytime alg (i.e $B \notin P$), then neither does A.

Ex: (Polytime Reductions)

1) $1||\sum w_j C_j \leq_p 1|r_j|\sum w_j C_j$
   Since $1||\sum w_j C_j$ is the special case with $r_j = 0 \ \forall j$.

   Also: $1|r_j|\sum C_j \leq_p 1|r_j|\sum w_j C_j$.

2) $(1|r_j|L_{max} \leq K) \leq_p 1|r_j|L_{max}$

3) $1|r_j|L_{max} \leq_p (1|r_j|L_{max} \leq K)$
   However, we need to be careful when searching for K (i.e. use a "smart" algorithm, like binary search)

Def^n 5.4 (NP, Verifier)
A decision problem Π is in the class NP if there exists a polynomial verifier V, s.t:
1) If $x \in \Pi$, there exists a certificate y s.t. $(|y| \in poly(|x|)$ AND $V(x,y) = YES$
2) If $x \notin \Pi$, $V(x,y) = NO$ for all y

Ex:
1) Is $\Pi_{SCHEDULE} \in NP$?
Yes!
Certificate: A schedule S where $\sum C_j \leq k$
Verifier: Takes S and checks:
- The schedule is valid
- $\sum C_j \leq k$

2) Π_{CLIQUE}
Def^n (Clique)
A clique in a graph $G=(V,E)$ is a subset $U \subseteq V(G)$ s.t. for all $u,v \in U$ there is an edge from u to v with $uv \in E(G)$.

And so,
$$\Pi_{CLIQUE} := \{(G,k) : G \text{ has a clique of size} \leq k\}$$

Π_{CLIQUE} is in NP.
Certificate: The subset U.
Verifier: For every 2 vertices $u,v \in U$ check that $uv \in E(G)$

↪ continued.

Ex (Continued)

3) $\Pi_{composite}$

$$\Pi_{composite} := \{\text{Positive Integers } x \text{ s.t. } x \text{ is composite}\}$$

$\Pi_{composite} \in NP$:
Certificate : p, q integers where $p, q \neq 1$ or x
Verifier : Check $pq = x$.

4) Π_{prime}

$$\Pi_{prime} := \{x \in \mathbb{Z}^+ \mid x \text{ is prime}\}$$

This problem seems harder, but, in fact, $\Pi_{prime} \in NP$, and the certificate is called the "Proof Certificate".

Claim 5.2: $P \subseteq NP$

[Proof]

Let decision problem $\Pi \in P$. Then, we have an algorithm A s.t. on input x, A returns YES if $x \in \Pi$ and NO if $x \notin \Pi$.

Consider the following verifier:

$$V(x, y) = \begin{cases} \text{YES} & \text{if } A \text{ returns YES on } x \\ \text{NO} & \text{if } A \text{ returns NO on } x \end{cases}$$

We claim that V shows that $\Pi \in NP$:

1) V runs in polytime, since A runs in polytime
2) Suppose $x \in \Pi$, we want to show that there exists a certificate y s.t. $V(x, y) = \text{YES}$.
   $y = \emptyset$ is a possible certificate, since $V(x, y) = \text{YES}$ for all YES instances
3) Suppose $x \notin \Pi$, then $V(x, y) = \text{NO}$ for all y □

Def'n 5.6 (NP-hard)
Problem A is NP-hard if $X \leq_p A$ for all $X \in NP$

Def'n 5.7 (NP-Complete)
A (decision) problem π is NP-Complete if $\pi \in NP$ and π is NP-hard

Remark 5.3:
Let π be NP-Complete:
(1) If $\pi \notin P$, then $P \neq NP$
(2) If $\pi \in P$, then $P = NP$
[Proof]
(1) Trivial, if $\pi \notin P$, then $\pi \in NP \setminus P$, so $P \subsetneq NP$.
(2) We want to show that $NP \subseteq P$. Let $X \in NP$, now, $X \leq_p \pi$ and π is solvable in polytime. Therefore, X is solvable in polytime. So $X \in P$. □

Strategy: (to proving problem π is NP-Complete)
1) Show $\pi \in NP$
2) Choose a suitable NP-Complete problem y
3) Show $y \leq_p \pi$

↳ Starting menu of NP-Complete problems

Starting Home of NP-Complete Problems

1) 3-SAT:
Given a boolean formula:
$F = C_1$ AND C_2 AND ... AND C_m
where each clause:
$C_i = (y_{i1}$ OR y_{i2} OR $y_{i3})$ $i \in \{1, \ldots, m\}$
each:
$y_{ij} = \begin{cases} x_k \\ \text{NOT } x_k \end{cases}$ for some k
Is there a setting (assignment) of TRUE/FALSE value to the x_k's under which F evaluates to TRUE?

2) CLIQUE:
Given a graph G and integer k, does G have a clique of size k?

3) SUBSET-SUM:
Given n integers $a_1, \ldots a_n \geq 0$ and a target integer B, is there a subset $S \subseteq \{1, \ldots, n\}$, s.t. $\sum_{i \in S} a_i = B$?

4) PARTITION:
Given n integers $a_1, \ldots, a_n \geq 0$ s.t. $\sum_{i=1}^{n} a_i = 2k$, where $k \in \mathbb{Z}$, is there $S \subseteq \{1, \ldots, n\}$ s.t. $\sum_{i \in S} a_i = k$?

5) 3-PARTITION:
Given $3n$ integers $a_1, \ldots, a_{3n} \geq 0$ with $\sum_{i=1}^{3n} a_i = nk$, $k \in \mathbb{Z}$, is there a partition $S_1, \ldots, S_n$ of $\{1, \ldots, 3n\}$ s.t.
$|S_i| = 3$ and $\sum_{i \in S_i} a_i = k$ $\forall i$?

Showing some scheduling problems are NP-Complete / NP-Hard

We will use three more and the strategy introduced earlier.

1) $1|r_j|L_{max} \leq 0$

Theorem 5.4 : $1|r_j|L_{max} \leq 0$ is NP-Complete

[Proof]

To prove this, we must show:

(i) $1|r_j|L_{max} \leq 0$ ∈ NP

(ii) Choose $Y \in$ NP-Complete, and show $Y \leq_p 1|r_j|L_{max} \leq 0$

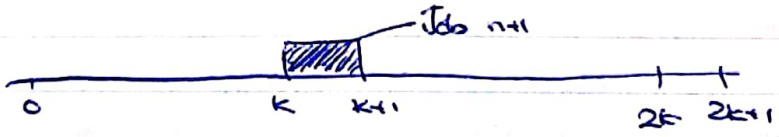
(i) is easy to show since a polysize certificate for a YES instance is the schedule itself; and the verifier will check if said schedule is feasible and $L_{max} \leq 0$

For (ii), we will show PARTITION $\leq_p 1|r_j|L_{max} \leq 0$

Let $a_1, \ldots, a_n \geq 0$ be integers with $\sum_{j=1}^{n} a_j = 2k$, k integer, be a partition instance.

Create n jobs with $p_j = a_j$ $\forall j = 1, \ldots, n$, $r_j = 0$ and $d_j = 2k+1$.

Create job $n+1$ with $p_{n+1} = 1$, $r_{n+1} = k$, $d_{n+1} = k+1$



Can't

[Proof] (cont)
We use the following lemma to complete the proof.

Lemma 5.4: There is a set $A \subseteq \{1, \dots, n\}$ with $\sum_{j \in A} = t$ iff there is a schedule S with $L_{max} \leq 0$.

[Proof] (Lemma)
($\Rightarrow$) If $\exists A \subseteq \{1, \dots, n\}$ with $\sum_{j \in A} a_j = t$, then, we schedule jobs in A first in $[0, t]$, schedule job $n+1$ in $[t, t+1]$, and jobs $\{1, \dots, n\} \setminus A$ in $[t+1, 2t+1]$. No jobs are late and so $L_{max} \leq 0$.

($\Leftarrow$) S cannot have idle time, and must schedule job $n+1$ in $[t, t+1]$. So, we can take $A = \{j \mid j$ is scheduled in $[0, t]\}$ and then $\sum_{j \in A} a_j = t$.

□ (Lemma + Theorem)

2) P2||Cmax:
P2: 2 identical / parallel machines
Cmax: Max completion time

Theorem 5.5: P2||Cmax is NP-hard
[Proof]
We will show:
PARTITION $\leq_p$ P2||Cmax
Let $a_1, \dots, a_n \geq 0$ be integers, $\sum_{j=1}^{n} a_j = 2k$ (k integer) be a PARTITION instance.
Create n jobs with $p_j = a_j$ $\forall j = 1, \dots, n$

Cont

[Proof] (cont.)

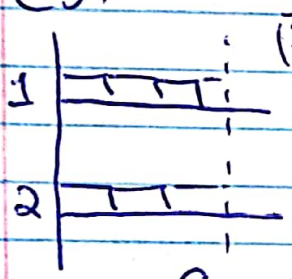
We finish the proof with the following lemma:

Lemma 5.5: C_{max} of the above $P||C_{max}$ instance equals k iff the PARTITION instance is a YES instance

[Proof] (Lemma)

($\Leftarrow$) If $\exists A \subseteq \{1,\dots,n\}$ with $\sum_{j \in A} a_j = k$, then schedule jobs in A on M/C 1 and jobs in $\{1,\dots,n\} \setminus A$ on M/C 2. This gives a schedule with $C_{max} = k$

($\Rightarrow$)



Since $C_{max}=k$ and $\sum_{j=1}^{n} p_j = 2k$, there cannot be any idle time on either machine in $[0, C_{max}=k]$.

So, we can take:

$A = \{j : j$ is scheduled on M/C $1\}$

Then, $\sum_{a_j \in A} a_j = k$, which is a YES instance of PARTITION.

□ (Lemma + Theorem)

So ■. continued

3) $1|prec|\sum U_j$:

We define:

$$U_j := \begin{cases} 1 & ; \text{ if } C_j > d_j \\ 0 & ; \text{ otherwise} \end{cases}$$

And so, the problem asks to minimize the number of late jobs

Theorem 5.6: $1|prec|\sum U_j$ is NP-hard

We will show:

$$CLIQUE \leq_p 1|prec|\sum U_j$$

Let $G=(V,E)$, integer k be a CLIQUE instance.
Let $|V|=n$, $|E|=m$, and define $l = \frac{k(k-1)}{2}$

Further:

- For every $u \in V$, create node-job j_u
- For every $e \in E$, create edge-job j_e

And $N = n+m$ = total # of jobs.

- For every edge $e=(u,v)$ of G, create 2 precedence constraints: $j_u \to j_e$, $j_v \to j_e$

And, for the schedule:

- Set $p_j = 1$ $\forall j$
- Set $d_{j_e} = k+l$ $\forall e \in E$
- Set $d_{j_u} = N = n+m$ $\forall u \in V$

(Since there are only $n+m$ jobs, node-jobs will never be late)

(Our goal is to force the schedule from $[0, k+l]$ to correspond to a clique)

Continued

[Proof] (Cont'd)
We complete the proof with the following lemma:

Lemma 5.6: G has a clique of size k iff there is a schedule with exactly $m-l$ late jobs

($\Rightarrow$) Let $A \subseteq V$ be a clique of size k. Schedule node-jobs j_u $\forall u \in A$ in $[0, k]$, and edge-jobs j_{uv} $\forall (u,v) \in A$ in $[k, k+l]$, and the remaining jobs arbitrarily in $[k+l, N]$. This gives l late jobs

($\Leftarrow$) Let S be a schedule with $m-l$ late jobs, so l edge jobs are scheduled in $[0, k+l]$. Define:

$$A = \{u \in V : j_u \text{ is scheduled in } [0, k+1]\}$$

$|A| \leq k$, since l edge jobs are scheduled in $[0, k+l]$.
By our precedence constraints for every edge job j_{uv} scheduled in $[0, k+l]$, we must have that j_u and j_v are also scheduled in $[0, k+l]$, so $u, v \in A$.
And we have $l = \frac{k(k-1)}{2}$, which can only be satisfied if $|A| = k$, so A is a clique of size k. □

6 LP Techniques (6) LP Techniques

1|prec|$\sum w_j C_j$ and 1|prec, r_j, pmtn|$\sum w_j C_j$

1|prec|$\sum w_j C_j$ is NP-hard (and hence so is 1|prec, r_j, pmtn|$\sum w_j C_j$). In fact, even the non-weighted problem 1|prec|$\sum C_j$ is NP-hard

Our goal is to write an LP relaxation for 1|prec|$\sum w_j C_j$, with C_j variables denoting the completion times of jobs, and work our way towards an approximation algorithm.

Notice that the basic set of constraints need not make a valid schedule:

i.e

$\min \sum_j w_j C_j$ $\quad$ J = set of all jobs, ($|J| = n$)

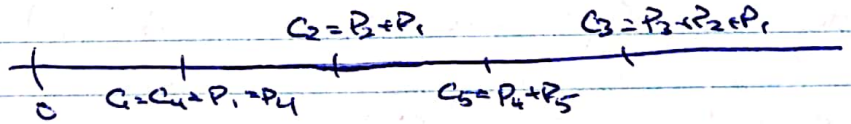
s.t.

$C_k \geq C_j + P_k$ $\quad$ if $j \to k$

$C_j \geq P_j$ $\quad$ $\forall j \in J$

But the following is a feasible solution:

$1 \to 2 \to 3$, $4 \to 5$



which is clearly not a valid schedule.

(continued)

Consider:

(LP)

$$\min \sum_{j \in J} w_j C_j$$

s.t.

$$C_k \geq C_j + P_k \quad \text{if } j \to k \qquad (1)$$

$$C_j \geq P_j \quad \forall j \in J \qquad (2)$$

$$\sum_{j \in A} P_j C_j \geq \frac{1}{2} P(A)^2 + \frac{1}{2} P^2(A) \quad \forall A \subseteq J \qquad (3)$$

where:

$$P(A) = \sum_{j \in A} P_j \quad \text{and} \quad P^2(A) = \sum_{j \in A} P_j^2$$

Lemma 6.1:

Let OPT be the optimal value of $1|prec|\sum w_j C_j$ and OPT_{LP} be the optimal value of (LP). Then,

$$OPT_{LP} \leq OPT.$$

[Proof]

We want to show that every feasible schedule yields a feasible solution to (LP), where $C_j = C_j^S \ \forall j$

Clearly the C_j's will satisfy (1) and (2).

Now, consider any $A \subseteq J$.

↪ continued.

[Proof] (cont'd)

We will say $k \le_S j$ if k is scheduled before j in S.

By defn:

$$C_j^S = C_j = \sum_{k \in S : k \le_S j} p_k \ge \sum_{k \in A : k \le_S j} p_k$$

Then,

$$\sum_{j \in A} p_j C_j \ge \sum_{j \in A} \sum_{\substack{k \in A : \\ k \le_S j}} p_j p_k$$

$$= \sum_{j \in A} p_j^2 + \sum_{\substack{j,k \in A \\ k \le_S j}} p_j p_k$$

$$= \frac{1}{2} \sum_{j \in A} p_j^2 + \frac{1}{2} \left(\sum_{j,k \in A} p_j p_k \right)^2$$

$$= \frac{1}{2} P^2(A) + \frac{1}{2} P(A)^2 \qquad \square$$

Using the LP, we can consider the following algorithm:

Algorithm A_1:

(1) Solve (LP) to obtain solution $\{C_j^*\}$

(Note: These C_j^*'s need not correspond to job completion times in a feasible schedule)

(2) Schedule jobs in increasing order of C_j^*

(Note: Schedule returned will be feasible, by design of the (LP): If $j \to k$, then $C_j^* \le C_k^*$)

→ Theorem 6.1

Theorem 6.1:
A is a 2-approx. alg. for $1|prec|\sum w_j C_j$

[Proof]
Let jobs be ordered $1, \dots, n$ so that $C_1^* \le \dots \le C_n^*$ and let S be the schedule outputted by A.

We want to show: $C_j^S \le 2C_j^*$ $\forall j = 1, \dots, n$.
So that:

$$\sum w_j C_j^S \le 2 \sum w_j C_j^*$$
$$\Rightarrow OPT_{A} \le 2 \cdot OPT_{1|prec|\sum w_j C_j}$$

Fix job j. Let $A = \{1, \dots, j\}$ and consider (3) for job-set A:

(i) $\sum_{k \in A} p_k C_k^* \ge \frac{1}{2}p(A)^2 + \frac{1}{2}p^2(A) \ge \frac{1}{2}p(A)^2$

(ii) $C_j^* \cdot \sum_{k \in A} p_k \ge \sum_{k \in A} p_k C_k^*$ (since $C_k^* \le C_j^*$)

$$\Rightarrow C_j^* \cdot \sum_{k \in A} p_k \ge \frac{1}{2}p(A)^2$$
$$\Rightarrow C_j^* \ge \frac{1}{2}p(A)^2 = \frac{1}{2}\sum_{k \in S_j} p_k = \frac{1}{2}C_j^S$$

□.

Q: Is A efficient? In particular, is solving (LP) efficient?
- Notice that if $|J| = n$, then there are 2^n subsets $A \subseteq J$, and hence 2^n constraints. Hence, in fact:

Theorem 6.2:
(LP) can be solved in polytime

[Proof] See Theorem 6.8

$1|prec, r_j|\sum w_j C_j$ and $1|prec, r_j, pmtn|\sum w_j C_j$

Claim 6.3:
We may assume WLOG that if $j \to k$, then $r_k \geq r_j + p_j$

[Proof]
The earliest time k can start is $r_j + p_j$, so if $r_k < r_j + p_j$, we can always advance the release date of k to $r_k \leftarrow \max(r_k, r_j + p_j)$ without affecting the feasibility of the schedule. □

As with $1|prec|\sum w_j C_j$, we will write an LP-relaxation for our problem.

(LP1)

$$\min \sum w_j C_j$$

s.t.

$$C_k \geq C_j + p_k \qquad \forall j \to k \qquad (1)$$

$$C_j \geq r_j + p_j \qquad \forall j \qquad (2)$$

$$\sum_{j \in A} p_j C_j \geq p(A) r(A) + \frac{1}{2} p(A)^2 + \frac{1}{2} p^2(A) \qquad \forall A \subseteq J \qquad (3)$$

where:

$$r(A) := \min_{j \in A} r_j \quad \text{for all } A \subseteq J.$$

Note:
It is easy to see:

$$OPT_{LP1} \leq OPT_{1|prec, r_j, pmtn|\sum w_j C_j} \leq OPT_{1|prec, r_j|\sum w_j C_j}$$

↳ Algo.

Algorithm A_2 (for $1|prec, r_j, pmtn|\sum w_j C_j$)

1) Solve (LP') to obtain $\{C_j^*\}$
2) Schedule preemptively in increasing C_j^* order.
   (i.e. At each point of time, schedule the available job with smallest C_j^* value, preempting as necessary)

Theorem 6.4:
A_2 produces a feasible schedule

[Proof]
We want to ensure that precedence constraints are respected.
Suppose $j \to k$. By Claim 3, if k is available at time t, either:
- j is also available at time t, or
- j is completed by time t

And also, $C_j^* \leq C_k^*$ by our ordering, unless j has completed, we will not process k. □

Theorem 6.5:
A_2 is a 2-approx. for $1|prec, r_j, pmtn|\sum w_j C_j$.

[Proof]
Let S be our schedule outputted by A_2, and order jobs so that:

$$C_1^S \leq C_2^S \leq \dots \leq C_n^S$$

We want to show that: $C_j^S \leq 2 \cdot C_j^* \ \forall j$

Cont

[Proof] (cont'd)

Consider job j and let t be the earliest time before C_j^S s.t.:

1) There is no idle time in $[t, C_j^S]$, and
2) m/c is only processing jobs k s.t. $k \leq j$ in $[t, C_j^S]$

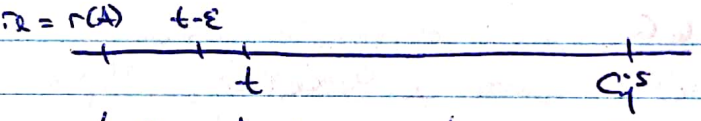
Let $A = \{\text{jobs } k : k \text{ scheduled in } [t, C_j^S]\}$

By defn: $\forall k \in A,\ C_k^* \leq C_j^*$

We claim: $\boxed{t = r(A).}$

Clearly $t \geq r(A)$, since some job $k \in A$ is scheduled at time t.

Suppose $t > r(A)$, and suppose let A s.t. $r_A = r(A)$.



At time $t-\varepsilon$, A is available since $r_A \leq t-\varepsilon$ and A has not completed since A is scheduled in $[t, C_j^S]$. So, at time $t-\varepsilon$ the m/c cannot be idle, and either:

- has scheduled A or some $A' \leq A$. but then we can move A to an earlier time
- has scheduled some $A' > j$, contradicting our schedule rule.

$\leftarrow$ cont

[Proof] (cont'd)

So, $t = r(A)$. Applying (3) in (LP'), we get:

- $\sum_{k \in A} p_k C_k^* \geq p(A) r(A) + \frac{1}{2} p(A)^2 + \frac{1}{2} p^2(A)$

  $\geq p(A) r(A) + \frac{1}{2} p(A)^2$

- $C_j^* p(A) \geq \sum_{k \in A} p_k C_k^*$

  (Since $C_k^* \leq C_j^*$)

$\Rightarrow C_j^* p(A) \geq p(A) r(A) + \frac{1}{2} p(A)^2$

$\Rightarrow C_j^* \geq r(A) + \frac{1}{2} p(A)$

So,

$C_j^S \leq t + p(A) = r(A) + p(A) \leq 2 C_j^*$ □.

Proposition 6.6:

A_2 + CONVERT gives a 4-approx $1|prec, r_j| \sum w_j C_j$

Algorithm A_3:

1) Solve (LP') to obtain $\{C_j^*\}$
2) Non-preemptively schedule in increasing C_j^* order

Theorem 6.7:

A_3 is a 3-approx. algo. for $1|prec, r_j| \sum w_j C_j$.

↳ Proof.

[Proof]
Let S be the schedule produced by A_3 and order jobs: $C_1^* \le \dots \le C_n^*$ (So, $C_1^S \le \dots \le C_n^S$)

Let $A = \{1, \dots, j\}$, then by (3) from (LP'), we get:

$$P(A) C_j^* \ge \sum_{k \in A} p_k C_k^* \ge \frac{1}{2} P(A)^2.$$

$$\Rightarrow \quad P(A) \le 2 C_j^*$$

So,

$$C_j^S \le \max_{k \le j} r_k + \sum_{k \le j} p_k$$

$$\le C_j^* + 2 C_j^* \quad (\text{since } r_k \le C_k^* \le C_j^*)$$
$$= 3 C_j^* \quad \forall j$$

which gives:

$$\sum w_j C_j^S \le 3 \cdot \sum w_j C_j^*$$

$$= 3 \, OPT_{(LP')}$$
$$\le 3 \cdot OPT_{1|prec, r_j| \sum w_j C_j}$$ □.

Ellipsoid Method:
First polytime algorithm to solve LPs

Defn 6.1: (Separation Oracle)
For an LP with feasible region:
$K := \{x \in \mathbb{R}^n : a_i^T x \le b \ \forall i = 1, \dots, m\}$
Given input $y \in \mathbb{R}^n$, the separation oracle is a procedure that correctly determines if $y \in K$, or finds a constraint $a_i^T x \le b_i$ of K that is violated by y.

Theorem 6.8 (Ellipsoid Method)
Consider an LP:

(P)

$$\min c^T x$$
$$\text{s.t.}$$
$$a_i^T x \le b_i \quad \forall i = 1, \ldots, m$$
$$x \in \mathbb{R}^n$$

Let $S := \max(\text{size of } c, \max_{i=1,\ldots,m}(\text{size of } (a_i, b_i)))$
Given a separation oracle A for (P), we can solve (P) using $\text{poly}(n, S)$ calls to A and $\text{poly}(n, S)$ [redacted] of extra operations.

Corollary 6.8:
Given (poly-time separation oracle for $\{x \in \mathbb{R}^n : a_i^T x \le b_i, \ i = 1, \ldots, m\}$, we can solve (P) in $\text{poly}(n, S)$ time.

7 Dynamic Programming

(7.1) Dynamic Programming

Knapsack Problem:
Given a set J of n items, knapsack of capacity $B \geq 0$ (B - integer). Each item has a value v_j and a weight w_j ($v_j, w_j \geq 0$, v_j, w_j integers).
The goal is: Choose a set $S \subseteq J$ of items of max. value whose total weight $\leq B$.

Observation:
Suppose S^* is an optimal solution to the above problem and $j \in S^*$. Notice that $S^* \setminus \{j\}$ must be an optimal solution to the subproblem with itemset $J \setminus \{j\}$ and capacity $B - w_j$.

Strategy:
Let $D(j, w)$ denote the subproblem with itemset $\{1, \ldots, j\}$ and capacity w. Our goal is to calculate $D(n, B)$ from the subproblems.

[Sol'n].
We first handle the base cases:
- $D(j, 0) = 0 \quad \forall j = 1, \ldots, n$
- $D(0, w) = 0 \quad \forall w = 1, \ldots, B.$

In the general case:

$$D(j, w) = \begin{cases} \max(v_j + D(j-1, w-w_j),\ D(j-1, w)) & ; w_j \leq w \\ D(j-1, w) & ; \text{otherwise} \end{cases}$$

(use item j; don't use item j)

This is known as the "DP recurrence".

Continued

Hilroy

Runtime Analysis:
Each $D(j, w)$ can be calculated in $O(1)$ time, and in total we have nB $D(j, w)$'s to calculate. This gives us a total runtime of $O(nB)$.

Note: Our DP recurrence gives us the optimal value. To obtain the optimal [redacted] itemset we can trace backwards from $D(n, B)$ (giving the sequence of steps required to obtain the optimal value)

Theorem 7.1
Knapsack is NP-hard

$O(nB)$ is not polynomial in the size of the input since B requires $\log B$ bits (and this is exponential). We call this pseudopolynomial

$1||\sum w_j U_j$:

Recall $U_j = \begin{cases} 1 & ; \text{if } C_j > d_j \\ 0 & ; \text{otherwise.} \end{cases}$, so $1||\sum w_j U_j$ is the problem of minimizing total weight of late jobs. This problem is NP-hard. Further, we may assume $p_j \leq d_j \ \forall j$.

Observation:
An optimal schedule for $1||\sum w_j U_j$ has the form:

On-time jobs	LATE jobs

Notice that all LATE jobs can be rearranged arbitrarily, and $\sum w_j U_j$ remains unchanged. Further, we can assume the on-time jobs are scheduled in increasing d_j order (by EDD rule).

We will show 2 solutions to $1||\sum w_j U_j$:
Order all jobs in $\uparrow d_j$ order, so $d_1 \leq d_2 \leq \dots \leq d_n$.

Solution 1:
We will define $P(j, w)$ to be the min. processing time of a set $S \subseteq \{1, \dots, j\}$ that can be completed on-time and whose total weight is $\geq w$.
Formally,

$$P(j,w) = \min \left\{ P(S) : \begin{array}{l} (1)\ S \subseteq \{1,\dots,j\} \\ (2)\ C_j \leq d_j \quad \forall j \in S \\ (3)\ \sum_{k \in S} w_k \geq w \end{array} \right\}$$

and if $\nexists S$ satisfying (1)-(3), then $P(j,w) = \infty$

↳ can't

Section I: (Cont'd)

How does this recurrence work?

Base Case:

- $P(0,0) = 0$
- $P(0,w) = \infty \quad \forall w > 0$
- $P(j,0) = 0 \quad \forall j = 0, \ldots, n.$

General Case:

1) $j \in$ optimal set S^* for $P(j,w)$:
Then, j must be scheduled last (by EDD rule). So,
$P(S^*) = P(S^* \setminus \{j\}) + p_j \le d_j$
$= P(j-1, w-w_j)$
So, $P(j,w) = p_j + P(j-1, w-w_j)$

Note: Here P denotes both the recurrence, and $\sum_{i=1}^{n} p_i$.

2) $j \notin S^*$:
Then, $P(j,w) = P(j-1,w)$

So, our DP recurrence is:

$$P(j,w) = \begin{cases} \min\{p_j + P(j-1, w-w_j),\ P(j-1,w)\} \\ \qquad ;\text{if } P(j-1, w-w_j) + p_j \le d_j \\ P(j-1,w) \qquad ;\text{otherwise} \end{cases}$$

The running time for $P(\cdot,\cdot)$ is then $O(n \sum_{i=1}^{n} w_i)$.

Note:
We can reduce the running time slightly, since if $P(j,w') = \infty$, then $P(j,w^*) = \infty$ for all $w^* \ge w'$. So, we can stop at the smallest w s.t. $P(j,w) = \infty$. This is precisely OPT, and so we get $O(n(\sum_{i=1}^{n} w_i - OPT))$.

Solution 2:
We modify our recurrence slightly.
Define:

$$Q(j,w) = \min\left\{ P(S) : \begin{array}{l} (1) S \subseteq \{1,\dots,j\} \\ (2) \text{All jobs in } S \text{ can be scheduled on time.} \\ (3) \sum_{k \in S} w_k \geq \sum_{k=1}^{j} w_k - w \end{array} \right\}.$$

Notice that (3) is the total weight of jobs from $\{1,\dots,j\} \setminus S$.

$Q(j,w) = P(j, \sum_{k=1}^{j} w_k - w)$, so this is the complement problem.
Then, our recurrence is:

$$Q(j,w) = \begin{cases} \min\{Q(j-1,w) + p_j,\ Q(j-1, w-w_j)\} & ; Q(j-1,w) + p_j \leq d_j \\ Q(j-1, w-w_j) & ; \text{otherwise} \end{cases}$$

The running time of $Q(\cdot,\cdot)$ is $O(n \cdot OPT)$, since we can stop once $Q(n,w) < \infty$. So, $Q(\cdot,\cdot)$ is more useful if OPT is small, and $P(\cdot,\cdot)$ is more useful if OPT is large.

→ Approximation Schemes

Hilroy

Approximation Schemes:

Dynamic algorithms are useful, but aren't efficient since they are pseudo-polynomial. We want to come up with efficient approximation algorithms using the exact ones.

Defn 7.1 (Polytime approximation scheme - PTAS)
Let X be a minimization problem. An algorithm A for X is called a polytime approximation scheme for X if for every instance I and every fixed $\varepsilon_0 > 0$, A returns a solution of:

$$\text{obj. value} \leq (1+\varepsilon_0) \cdot OPT(I)$$

in time: $f(\text{size}(I), \varepsilon_0) = \text{poly}(\text{size}(I))$

Note:
$f(\cdot, \varepsilon)$ can have an arbitrary dependence on ε. So, the following are valid:

(a) $n^{1/\varepsilon}$, (b) $(\frac{1}{\varepsilon})^{1/\varepsilon^2} n^{2 \cdot 1/\varepsilon}$

(c) $2^{1/\varepsilon} n$ (d) $(\frac{1}{\varepsilon})^2 n^2 \log n$

The important point is that ε is fixed.

Defn 7.2 (Fully polytime approx. scheme - FPTAS)
A fully polytime approx. scheme is an algorithm A, where

- A is a PTAS
- f satisfies:

$$f(\text{size}(I), \varepsilon) = \text{poly}(\text{size}(I), 1/\varepsilon)$$

$1||\sum w_j U_j$:

We want to come up with an FPTAS for $1||\sum w_j U_j$. We will need the following building blocks:

(1) An exact algorithm with running time $O(n \cdot OPT)$, or, more generally, $O(poly(n) * poly(OPT))$.
(We can get this from $Q(\cdot,\cdot)$)

(2) A lower bound on OPT s.t.
$$LB \le OPT \le poly(n) \cdot LB.$$
(This needs to be efficiently calculated).

We have (1), so let's work on (2). How can we obtain LB?

Notice that for any schedule S:
$$\max_j w_j U_j^S \le \sum_j w_j U_j^S \le n \cdot \max_j w_j U_j^S$$

So,
$$\min_{\text{schedule } S} \max_j w_j U_j^S \le OPT \le n \cdot \min_S \max_j w_j U_j^S$$

And we can solve $1||\max_j w_j U_j$ efficiently using the LCh rule.

Hence $LB = OPT_{1||\max_j w_j U_j}$

Algorithm:
- Choose suitable $\Delta > 0$
- Set $w'_j = \lfloor w_j/\Delta \rfloor$ $\forall j$ (then $\frac{w_j}{\Delta} - 1 \le w'_j \le \frac{w_j}{\Delta}$)
- Run $Q(\cdot,\cdot)$ DP on the $\{w'_j\}$-instance to get a schedule, and a set A' of late jobs, where
$$w'(A') = \sum_{j \in A'} w'_j = OPT'$$

Can't

Claim 7.2:
The previous algorithm is an FPTAS for $1||\sum w_j U_j$.

[Proof]

To show this, we must show:

(1) $w(A') \leq (1+\varepsilon) \cdot OPT$, and

(2) $O(n \cdot OPT')$ is in $poly(n, 1/\varepsilon)$

Let A^*: Set of late jobs in optimal schedule with $\{w_j\}$-weights

Note that:

$$OPT' \leq w'(A^*) \leq \frac{w(A^*)}{\Delta} \leq \frac{OPT}{\Delta}$$

(since each $w'_j \leq \frac{w_j}{\Delta}$)

which also shows that:

$$O(n \cdot OPT') = O\left(n \cdot \frac{OPT}{\Delta}\right)$$

We can bound $w(A')$ in terms of OPT and Δ to help us show (1).

$$w(A') = \sum_{j \in A'} w_j \leq \sum_{j \in A'} (w'_j + 1)\Delta \quad \left(\text{since } \frac{w_j}{\Delta} \leq w'_j + 1\right)$$

$$\leq w'(A') + n\Delta \quad (\text{since there are at most } n \text{ jobs})$$

$$\leq w'(A^*) + n\Delta \quad (\text{since } A' \text{ is optimal for } \{w'_j\}\text{-weights})$$

$$\leq OPT + n\Delta \quad (\text{by above})$$

Q.E.D.

[Proof] (cont'd)

Now, we choose, $\Delta = \frac{\varepsilon \cdot LB}{n}$,

and using that:

$$LB \le OPT \le n \cdot LB$$

We have:

(1) $w(A') \le OPT + n\Delta \le OPT + \varepsilon \cdot LB \le (1+\varepsilon) OPT$

(2)

$$O\left(n \cdot \frac{OPT}{\Delta \cdot LB}\right) = O\left(n \cdot \frac{OPT}{\frac{\varepsilon \cdot LB}{n}}\right) = O\left(\frac{n^2}{\varepsilon} \cdot \frac{OPT}{LB}\right)$$

$$= O\left(\frac{n^3}{\varepsilon}\right)$$

So, we have an FPTAS. □

(8) Multiple Machine Models.

8 Multiple Machine Models

Notation:

Recall we write $\alpha|\beta|\gamma$, where α = m/c environment

Common choices for α we will use in defining multiple machine environments are:

- P: multiple identical m/c's OR parallel m/c's
  Each job j takes time p_j to be processed on any m/c
- R: unrelated m/c's:
  job j requires time p_{ij} to be processed on machine i.
  (P is the special case where $p_{ij} = p_j \ \forall i, j$)
- Q: related m/c's
  Each m/c i has speed $s_i > 0$ and so the processing time for job j on m/c i is $p_{ij} = p_i/s_i$.
  (P is the special case where $s_i = 1 \ \forall i$)

In each of these cases, a job j only has to be processed on 1 m/c.

Sometimes, we write P_m, Q_m, or R_m, where m denotes the number of m/c's.

↳ P|pmtn|Cmax

History

P|prmtn|C_{max}:

We define:

$$C_{max} := \max_j C_j$$

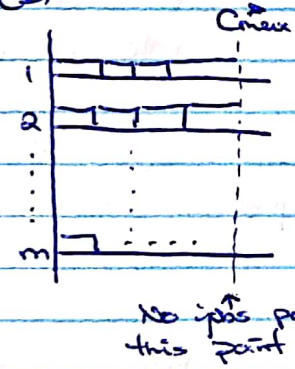
often called the makespan of a schedule.

Note: Preemption in a multiple m/c environment means that job j can be interrupted and resumed later on any m/c. However, at any point in time, at most 1 m/c can be processing the job.

Let us first find a lower bound on C_{max}:

(1) $C^*_{max} \geq p_j \ \forall j \equiv C^*_{max} \geq p_{max} := \max_j p_j$.

(2)



In the time interval $[0, C^*_{max}]$, we have mC^*_{max} total processing capacity on the m m/c's, and since there are no jobs past that point:

$$mC^*_{max} \geq \sum_{j=1}^{n} p_j$$

So, (1) and (2) give:

$$C^*_{max} \geq LB := \max\left\{ \sum_{j=1}^{n} p_j / m, \ p_{max} \right\}$$

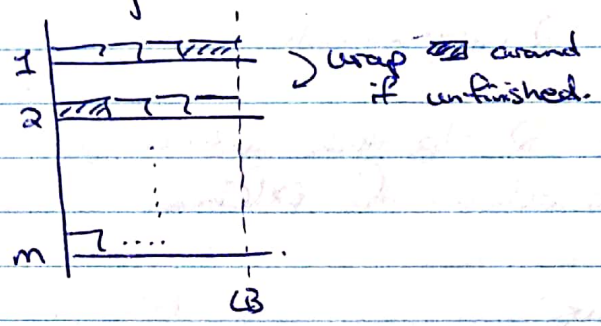
load-based lower bound — job-based lower bound

Cont'd

P|pmtn|Cmax: (cont'd)

McNaughton's Wraparound Rule:

Consider jobs in arbitrary order and begin packing them in the interval $[0, LB]$ on the mlc's. When we reach LB on mlc i, we move to mlc i+1 and resume with the unfinished portion of the current job.



Claim 8.1: We finish processing all jobs using this rule

We have enough space to pack all jobs since $m \cdot LB \geq \sum p_i$

Claim 8.2: No job is processed on multiple mlc's at the same point of time

$LB \geq p_{max}$, so if we have multiple chunks of a job, they cannot overlap.

Using this rule, we have an algorithm that obtains an optimal schedule for P|pmtn|Cmax.

$\Rightarrow P|pmtn|C_{max}$

$P||C_{max}$:

We can obtain an easy non-preemptive algorithm by simply not preempting a job if it comes around (in the schedule for $P|pmtn|C_{max}$). Since jobs have a max processing time of p_{max}, this gives:

$$LB + p_{max} \leq 2LB$$

which is a 2-approximation.

We will show 2 algorithms which give even better approximations for $P||C_{max}$.

List Scheduling:

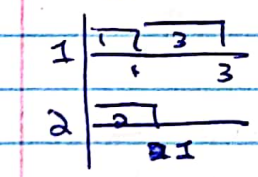
Consider a list of jobs in any order. Schedule the first unscheduled job on the list on the first available mc.

Ex:

$m = 2$, $P_1, P_2 = 1$, $P_3 = 2$

List = [1, 2, 3]

Using list scheduling this gives:



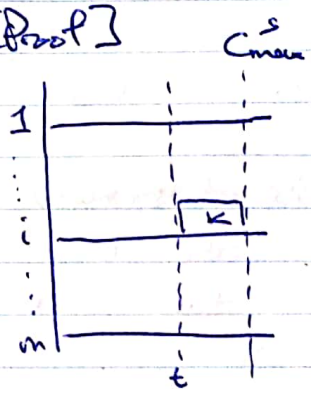
→ Notice that the makespan is 3 (But, clearly, the OPT makespan is 2).

Theorem 8.3:

List Scheduling is a $(2 - \frac{1}{m})$-approximation algorithm for $P||C_{max}$

→ [Proof]

[Proof]



Let S be the schedule constructed
Let k be the job to finish last in S, i.e. $C^S_k = C^S_{max}$
Let t be the time where k begins processing

In $[0,t]$, all m/c's are busy processing jobs other than k (Otherwise, we would schedule k earlier).
So,

$m \cdot t \leq$ Total processing time of jobs available other than k.

i.e.

$$m \cdot t \leq \sum_{j \neq k} P_j = \sum_{j=1}^{n} P_j - P_k$$

Then,

$$C^S_{max} = t + P_k \leq \underbrace{\sum_{j=1}^{n} P_j / m}_{\leq LB} - P_k/m + P_k$$

$$\leq LB - \underbrace{P_k}_{\leq P_{max} = LB}\left(1 - \frac{1}{m}\right)$$

$$\leq 2LB$$

So, we have a 2-approximation.

□.

→ Another algorithm

P||Cmax (cont'd)

List Scheduling w/ Largest Processing Time (LPT) Ordering:

(We abbreviate this to LPT-Schedule(Rule))

Sort jobs in decreasing p_j order and run LIST-SCHEDULING (LS) with this order.

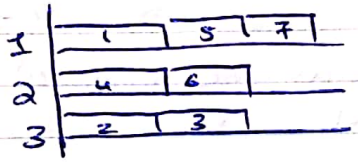
Ex:

$m=3$

j	1	2	3	4	5	6	7
p_j	5	4	4	5	3	3	3

→ This gives the ordering: 1, 4, 2, 3, 5, 6, 7.

S:



Theorem 8.4:

LS with the LPT-rule is a $(\frac{4}{3}-\frac{1}{3m})$-approx. for P||Cmax

[Proof]

Let S: Schedule constructed,

Let k: the last job that completes in S, so $C_k^S = C_{max}^S$

Let t: the point of time where k begins processing.

Let $J' = \{j : p_j \geq p_k\}$

→ We can focus on J' since under the LPT rule, the last job to start processing is the shortest job. If it also finishes last, then we have that job k is the last job. Otherwise, we can delete this job without affecting the makespan

Cont'd

[Proof] (cont'd)
Since it suffices to only focus on J', we have:
- The LPT-schedule for J' is the LPT-schedule for J up to k, call this S'.
- $C_{max}^{S'} = C_{max}^{S}$
- We can show that:

$$C_{max}^{S'} \leq \left(\frac{4}{3} - \frac{1}{3m}\right) OPT(J')$$

since $OPT(J') \leq OPT(J)$

Further, because of the simplification, we can say:
- $P_k = \min_{j \in J'} P_j$, and
- k is the last job to start and finish in S'.

Write $OPT' = OPT(J')$ and consider 2 cases:

Case 1: $P_k \leq OPT'/3$

From proof of Theorem 8.3, we have:

$$C_{max}^{S'} \leq \sum_{j \in J'} P_j + P_k\left(1 - \frac{1}{m}\right)$$

$$\leq OPT' + \frac{OPT'}{3}\left(1 - \frac{1}{m}\right) = \left(\frac{4}{3} - \frac{1}{3m}\right) OPT'$$

Case 2: $P_k > OPT'/3$.

So, $P_j > OPT'/3$ $\forall j$ since $P_j \geq P_k$ $\forall j$.
$\Rightarrow$ Every optimal schedule (for J') schedules ≤ 2 jobs per mc. So $n' := |J'| \leq 2m$.

(a) $n' \leq m$: LPT will schedule at most one job per machine and such a schedule must be optimal.

→ Cont'

Kilroy

[Proof] (cont)

(b) $m < n' \leq 2m$:

First, we claim that there is an optimal schedule that schedules at least one job per m/c. This is true since we can always move a job from an m/c to another, having 0 jobs, without increasing the makespan. Then, since $n' \geq m$, we have at least one job on each m/c.

So, there is an optimal schedule where there are [redacted] 1 or 2 jobs per m/c. We will show that we can define a "nice structure" for these schedules and argue that this must be an LPT schedule.

Label the jobs, in the optimal schedule, on m/c i as follows:

$(i,1)$, $(i,2)$, where $P_{(i,1)} \geq P_{(i,2)}$ with the convention where if i has only one job, then $(i,2) = \text{NULL}$ and $P_{(i,2)} = 0$.

Reorder the machines so that:

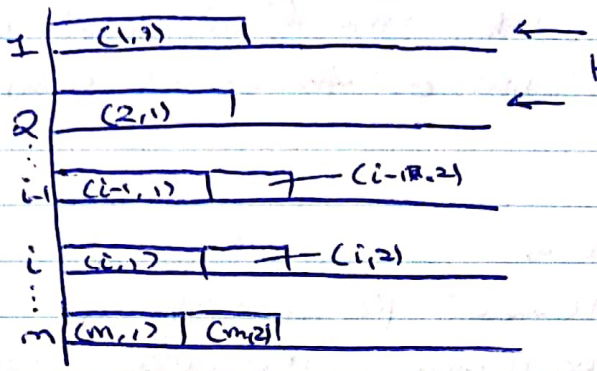
$P_{(i,1)} \geq P_{(i,2)} \geq \ldots \geq P_{(i,m)}$

Further, we can also assume that if $i^* < i'$, then $P_{(i',2)} \geq P_{(i,2)}$, with this ordering. Since we can always interchange the 2 without affecting the makespan.

Cont

Proof 3 (cont'd)

So, we have the following structure:



So, $P_{(1,1)} \geq P_{(2,1)} \geq \cdots \geq P_{(m,1)} \geq P_{(m,2)} \geq \cdots$
$\cdots \geq P_{(i,2)} \geq \cdots \geq P_{(1,2)}$

We claim all LPT schedules will look like this (and hence are also optimal). To show this, LPT schedules cannot assign a 3rd job to any m/c.

Suppose i' is the m/c on which LPT assigns a 3rd job for the first time.

The rest of this argument makes no sense, and is unnecessary.
Try a proof using contradiction.
i.e. LPT produces a $> (\frac{4}{3} - \frac{1}{3m})$ - approx.

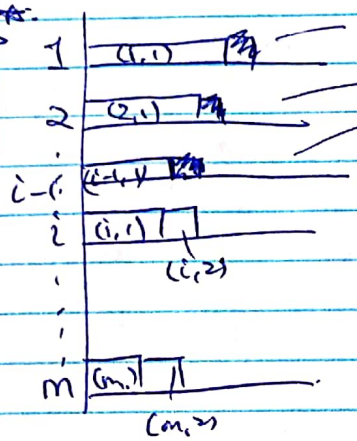
Kilroy

Theorem 8.4

The offending Proof

Proof (cont)

So, now we have the following pic structure of an optimal schedule for J

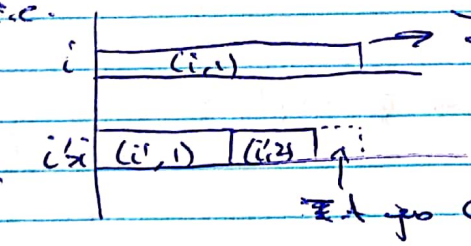


So,

$$P_{(1,1)} \geq P_{(2,1)} \geq \cdots P_{(i-1,1)} \geq P_{(i,1)} \geq \cdots$$
$$\cdots \geq P_{(n,1)} \geq P_{(n,2)} \geq P_{(n-1,2)} \geq \cdots \geq P_{(i,2)}$$

We want to show that S^* is an LPT-schedule. Both for this, we have to show that LPT does not assign a 3rd job on any machine.

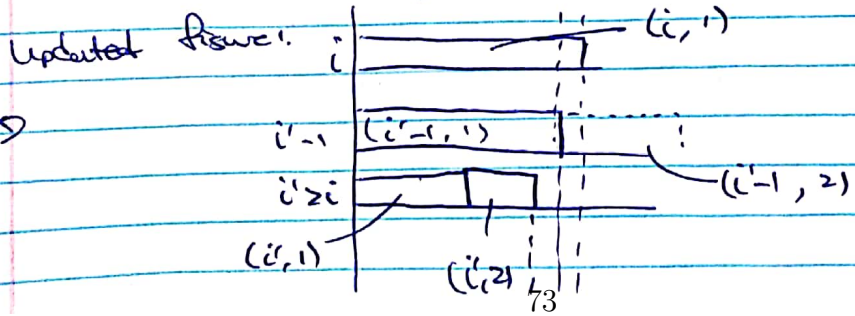
(i.e.



Intuitively, this means that (i,1) must be pretty long.

Suppose that i' is the mc on which LPT assigns a 3rd job for the first time

Updated figure:



Claim: $(i'-1, 2)$ is not null.

If $(i'-1, 2)$ is not null, then # of jobs n', is $\leq 1 \cdot (i'-1) + 2(m - i' + 1)$

- looking at OPT schedule
- $1 \cdot (i'-1)$: m/c's $1, \ldots i'-1$ have 1 job
- $2(m-i'+1)$: m/c's $i', \ldots, m$ have ≤ 2 jobs

But looking at OPT schedule ~~$\geq i'$~~ $n' > 1 - i' + 2(m - i' + 1)$ since it has assigned ≥ 1 job on $1, \ldots, i'-1$, and 2 jobs on $i', \ldots, m$, and is still left with more jobs

~~Given~~

Given our claim, the load on $i'-1$ in S^* is $P_{(i'-1,1)} + P_{(i'-1,2)}$

$\geq P_{(i',1)} + P_{(i',2)} > \frac{OPT'}{3}$

Since OPT is assigning a 3rd job on i'

$\Rightarrow > 2 \cdot \frac{OPT'}{3}$

$\Rightarrow P_{(i'-1,1)} + P_{(i'-1,2)} > \frac{2}{3} OPT'$,

which is a contradiction.

Generalized List Scheduling:

Consider a list of jobs in any given order. Schedule the first valid unscheduled job in the list on the first available mc

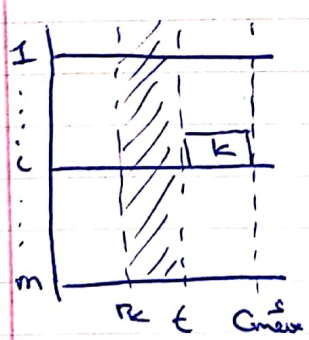
Note:
"Valid" means:
- For $P|r_j|C_{max}$: job is released
- For $P|prec|C_{max}$: Predecessors have completed
- For $P|r_j, prec|C_{max}$: Released and all predecessors have completed

Theorem 8.5:
(Generalized) LS is a 2-approx. for $P|r_j|C_{max}$.

[Proof]
We first refine our lower bound to account for release dates:

$$C^*_{max} \geq \max\left\{\sum P_j/m,\ \max_j (r_j + p_j)\right\}.$$

Call this LB'.



Let S: Schedule constructed,
Let k: job to complete last in S
Let t: time where k begins processing

Cont'd

[Proof] (cont'd)

In the interval $[r_k, t]$, mlc's $1, \ldots, m$ must be busy processing jobs other than k, since if not we could have scheduled k earlier. In other words:

$$m(t - r_k) \leq \sum_{j \neq k} p_j \leq \sum_{j=1}^{n} p_j$$

So,

$$\begin{aligned} C^S_{max} &= t + p_k \\ &= (r_k + p_k) + (t - r_k) \\ &\leq \max_j (r_j + p_j) + \sum_{j=1}^{n} p_j / m \\ &= 2 \cdot LB' \end{aligned}$$

□.

Theorem 8.6:
Generalized LS is a 2-approx. for $P|prec|C_{max}$

[Proof]

Again, we refine our lower bound to account for precedence constraints.

Suppose we have a chain of jobs: $j_1 \to j_2 \to \ldots \to j_r$ then,

$$C^*_{max} \geq p_{j_1} + p_{j_2} + \ldots + p_{j_r}.$$

So,

$$C^*_{max} \geq \max \left\{ \sum_{j=1}^{n} p_j / m, \max_{\text{all chains } j_1 \to \ldots \to j_r} (p_{j_1} + \ldots + p_{j_r}) \right\}$$

← Call this LB''

Let S and k be as defined in Theorem 8.5, and let t_k be the start time of k.

Cont'd

[Proof] (Cont'd)

We will construct a chain $C: k_r \to \cdots \to k_1 := k$, s.t. at every time $t \in [0, t_1]$, either:

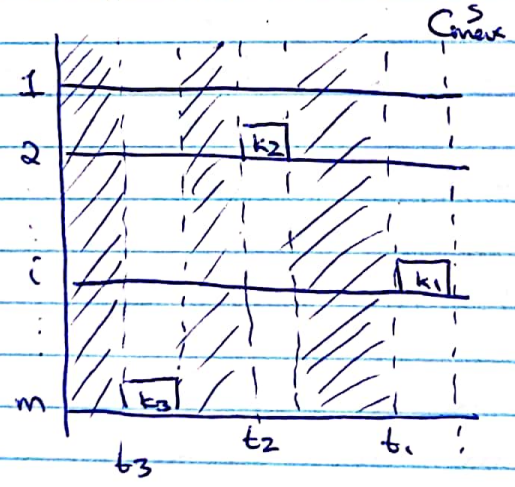
(1) Some job of C is processed at time t; or

(2) All m/c's are busy.

Let k_2 be the job s.t. $k_2 \to k_1$ and k_2 completes last among all predecessors of k_1. Let t_2 be the start time of k_2. (If k_1 has no predecessors, set $k_2 = \text{NULL}$, $t_2 = 0$ and $P_{k_2} = 0$)

And, continue this way: In general, k_{i+1} is the predecessor of k_i that completes last among all predecessors of k_i. t_{i+1} is the start time of k_{i+1}, and if k_i has no predecessors, set $k_{i+1} = \text{NULL}$, $t_{i+1} = 0$, $P_{k_{i+1}} = 0$

i.e.



This gives a chain: $k_r \to k_{r-1} \to \cdots \to k_1$ with $k_{r+1} = \text{NULL}$, $t_{r+1} = 0$, $P_{k_{r+1}} = 0$.

$\Rightarrow$ Cont'd

[Proof] (cont'd)

For any $d=1, \ldots, r$, all mc's must be busy within $[t_{k_{d-1}} + P_{k_{d-1}}, t_{k_d}]$, since all predecessors of k_d will have been completed, so if the mc's are not all busy, we can schedule k_d at an earlier time. And so:

$$m \cdot \left(\begin{array}{c} \text{total length of all} \\ [t_{k_{d-1}} + P_{k_{d-1}}, t_{k_d}] \\ \text{intervals} \end{array} \right) \leq \sum_{\substack{k \text{ not} \\ \text{in chain } C}} P_k \leq \sum_{i=1}^{n} P_i$$

Then,

$$\begin{aligned} C^S_{max} &= (\text{total length of all } [t_{k_{d-1}} + P_{k_{d-1}}, t_{k_d}] \text{ intervals}) \\ &\quad + (P_{k_1} + \cdots + P_{k_r}) \\ &\leq \sum_{i=1}^{n} P_i / m + \max_{\text{chain}} P_{i_1} + \cdots + P_{i_n} \\ &= 2 \cdot LB'' . \end{aligned}$$

□.

$\Rightarrow Q \| C_{max}$

Q||C_{max}:

Recall: Each m/c i has speed $s_i > 0$, and so takes time p_i/s_i to complete job j on m/c i.

Again, we begin by refining LB to account for the speeds s_i.

(1) In time $[0, C^*_{max}]$, each machine i can do $s_i(C^*_{max})$ amount of work.
So, in total, all m machines can do: $\left(\sum_{i=1}^{m} s_i\right) C^*_{max}$ amount of work. And this had better be $\geq \sum_{i=1}^{n} p_i$.
So, we have:

$$\left(\sum_{i=1}^{m} s_i\right) C^*_{max} \geq \sum_{i=1}^{n} p_i$$

$$\Rightarrow C^*_{max} \geq \sum_{i=1}^{n} p_i \Big/ \sum_{i=1}^{m} s_i.$$

(2) Order m/c's so that $s_1 \geq s_2 \geq \dots \geq s_m$
Order jobs so that $p_1 \geq p_2 \geq \dots \geq p_n$.
The fastest m/c should be able to complete the largest job by C^*_{max}:
So:

$$s_1 \cdot C^*_{max} \geq p_1 \Rightarrow C^*_{max} \geq p_1/s_1$$

And, in general, for any $k \leq m$, the k fastest m/c's should be able to complete the k largest jobs by C^*_{max}
So, we have:

$$C^*_{max} \geq \max_{k=1,\dots,m} \frac{\sum_{i=1}^{k} p_i}{\sum_{i=1}^{k} s_i}.$$

Q||Cmax: (Cont)

Combining (1) and (2), we have:

$$C^*_{max} \geq \max\left\{ \sum_{i=1}^{n} p_i / \sum_{i=1}^{m} s_i \;,\; \max_{k \leq m} \sum_{i=1}^{k} p_i / \sum_{i=1}^{k} s_i \right\}.$$

LS w/ LPT ordering (for Q||Cmax):

Order jobs in ↓ p_j order (so $p_1 \geq p_2 \geq \ldots \geq p_n$)

Repeatedly schedule the first unscheduled job on the mc on which it finishes earliest (given currently scheduled jobs).

Theorem 8.7:

LS with LPT rule is a $(2-\frac{1}{m})$-approx. for Q||Cmax

[Proof]

Let S: Schedule constructed, k: job that finishes last. And, as with P||Cmax, we can assume that $p_k = \min_{j=1,\ldots,n} p_j = p_n$.

Case 1: $m \leq n$

Since n was scheduled on mc i, we know that ∀ mc i', completion time if job n was scheduled on mc i' $\geq C^S_{max}$:

i.e.

$$\forall i' = 1, \ldots, m \quad \left(p_n + \sum_{\substack{j \neq i' \\ j \neq n}} p_j \right) / s_{i'} \geq C^S_{max}$$

So summing all m equations, we get:

$$m p_n + \sum_{j \neq n} p_j \geq C^S_{max} \cdot \sum_{i=1}^{m} s_i$$

$$\Rightarrow C^S_{max} \cdot \sum_{i=1}^{m} s_i \leq \sum_{j=1}^{n} p_j + (m-1) p_n = \sum_{j=1}^{n} p_j + \left(1 - \frac{1}{m}\right) m p_n.$$

$$\leq \sum_{j=1}^{n} p_j + \left(1 - \frac{1}{m}\right) n p_n.$$

Co can't

Hilroy

[Proof] (cont'd)

$$C_{max}^{S} \sum_{i=1}^{m} s_i \le \sum_{j=1}^{n} p_j + \left(1-\frac{1}{m}\right) n p_n$$

$$\le \sum_{j=1}^{n} p_j + \left(1-\frac{1}{m}\right) \sum_{j=1}^{n} p_j \quad (p_n \le p_j)$$

$$\Rightarrow C_{max}^{S} \le \frac{\left(2-\frac{1}{m}\right) \sum_{j=1}^{n} p_j}{\sum_{i=1}^{m} s_i} \le \left(2-\frac{1}{m}\right) C_{max}^{*}$$

Case 2: $m > n$

So, $m-n$ mc's will be idle, in fact, the machines from $n+1, \ldots, m$ will be idle.

Using the equation from Case 1, and adding $\forall i = 1, \ldots, n$.

$$\left(\sum_{i=1}^{n} s_i\right) \cdot C_{max}^{S} \le \sum_{j=1}^{n} p_j + (n-1) p_n$$

$$\le \sum_{j=1}^{n} p_j \left(2-\frac{1}{n}\right)$$

$$\Rightarrow C_{max}^{S} \le \left(2-\frac{1}{n}\right) \left(\sum_{j=1}^{n} p_j \Big/ \sum_{i=1}^{m} s_i\right)$$

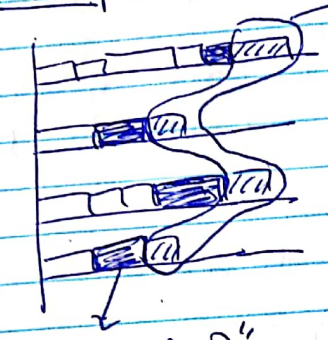
$$\le \left(2-\frac{1}{m}\right) C_{max}^{*} \quad (\text{Since } n < m)$$

□.

~~9~~ Lecture 16

454 - 16

$P||\sum C_j$:



If job j is assigned to slot k on mc i means that there are $k-1$ jobs on mc i after job j.

Why is this useful?:
Every job j assigned to slot 1 on mc i contributes p_j to our objective. And, in general, if j is assigned to slot k on a mc, it contributes $k \cdot p_j$ to $\sum C_j$. (Since it contributes p_j to its completion time, and $(k-1)p_j$ to the jobs that come after it on its mc).

Goal: Assign jobs to slots so as to minimize

$$\sum_{j \Rightarrow \text{slot } k} k p_j.$$

"Assign largest m jobs to m slot 1's
Next batch of m largest jobs to m slot 2's,
and so on."

i.e. Sort jobs so that $p_1 \geq \dots \geq p_n$. Let $J = \{1, \dots, n\}$.
$k=1, 1+1+1$

Hilroy

Repeat until $J = \emptyset$:
- Schedule $\min(m, |J|)$ jobs from J in first slot k of the m m/c's (arbitrarily)
- $J \leftarrow J \setminus \{$ first $\min(m, |J|)$ jobs of $J\}$
- $k \leftarrow k+1$.

This is clearly polynomial in time.

Theorem 1: The above algorithm produces an optimal schedule for $P||\sum C_j$.
[Proof] Exercise

Exercise: Show that running LIST-SCHEDULING with jobs sorted in increasing p_j order mimics the above algorithm and produces an optimal schedule.

$P||\sum C_j$:
(Recall: job j takes time p_{ij} to be processed on m/c i)

If $j \mapsto Slot(i,k)$, then there are $(k-1)$ jobs following j on m/c i, and j contributes $k \cdot p_{ij}$ to $\sum C_j$.

Then, our task can be restated as:

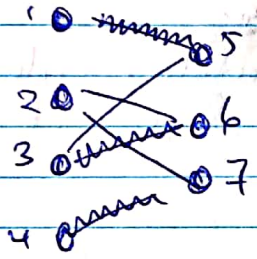
$\min \sum C_j \equiv$ [illegible]
Job → slot assignment problem
Assign jobs to (i,k) slot to
$\min \sum_{j \mapsto (i,k)} k p_{ij}$.

Excursion into matchings in bipartite graphs:

Let $G = (V := A \cup B, E)$ be a bipartite graph with bipartition $A \cup B$.

Defn: (Matching, S-perfect)
- A matching M is a set of edges s.t no 2 edges of M share an endpoint
- Given $S \subseteq V$, a matching M is said to be S-perfect, if $\forall v \in S$, $\exists e \in M$ incident to v.

Ex:



— matching M.

For $S = \{1, 5, 6, 7\}$, M is S-perfect.
For $S = \{2\}$, M is not S-perfect.

Defn (Fractional Matching, Fractional S-perfect)
A fractional matching is an assignment $\{x_e \geq 0\}_{e \in E}$ of numbers to edges s.t.

$$\sum_{e \in \delta(v)} x_e \leq 1 \quad \forall v \in V. \quad (\delta(v): \text{set of edges incident to } v).$$

Further, a fractional matching x is S-perfect (for $S \subseteq V$) if

$$\sum_{e \in \delta(v)} x_e = 1 \quad \forall v \in S.$$

Note: If $x \in \{0,1\}^E$, then fractional matching corresponds to matchings and similarly for S-perfect matching.

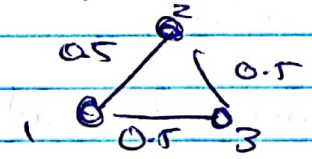
Hilroy

Fact 1: Given edge costs $\{c_e\}_{e \in E}$, and any $S \subseteq V$, we can decide if $\exists$ S-perfect matchings and if so compute a min cost S-perfect matching in poly time.
Cost of matching $M = \sum_{e \in M} c_e$

Fact 2: In a bipartite graph, given edge costs $\{c_e\}_{e \in E}$ and a fractional S-perfect matching x, we can compute (in poly time) an S-perfect matching M of cost $\leq \sum_{e \in E} c_e x_e$.

(In particular, an S-perfect matching always exists if there is a fractional S-perfect matching)

(Ex. Counter-example:



x: Fractional $\{1,2,3\}$-perfect matching.

But, there is no way to obtain a $\{1,2,3\}$-perfect matching.)

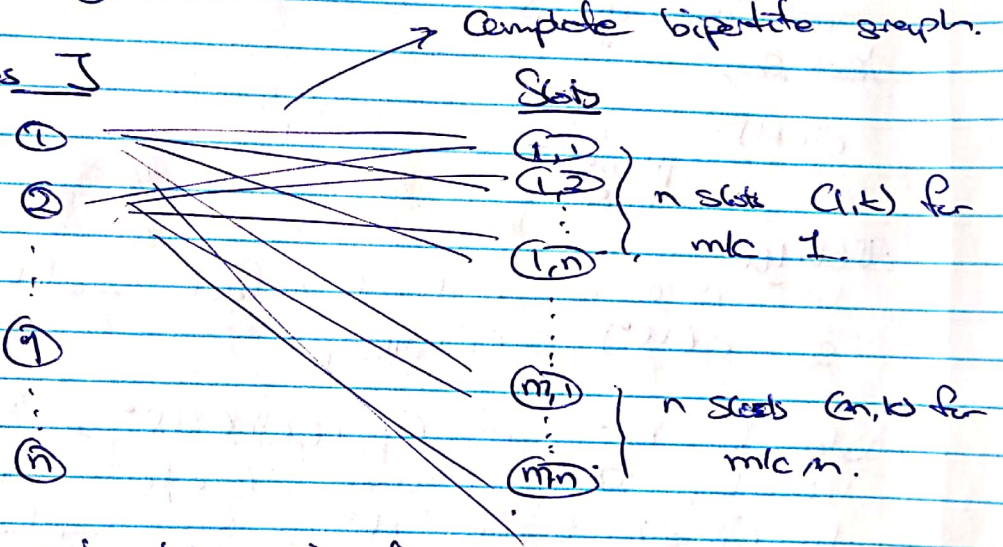
i.e. The assumption of bipartite graphs is important.

Back to $R||\sum C_j$

Recall we want to schedule jobs to slots to minimize $\sum_{j \to (i,k)} k p_{ij}$

Strategy: We will view this as a min-cost S-matching problem in a bipartite graph.

Bipartite graph:



Create node-jobs j for all jobs, and node (i,k) for each slot $\forall i=1,\ldots,m,\ k=1,\ldots,n$.

The cost on edge $(j,(i,k))$ $\forall$ jobs j, slot (i,k) is $k \cdot p_{ij}$.

Theorem:
A min cost J-perfect matching M^* yields an optimal schedule for $R||\sum C_j$ where $j \to (i,k)$ iff $(j,(i,k)) \in M^*$.

[Proof]

Suppose we have a schedule S. Then, consider the edge set $M = \{(j, (i,k)) : j$ is scheduled in slot (i,k) in $S\}$

Then M is a J-perfect matching since every job is assigned to exactly 1 slot and every slot is assigned at most one job.

So,

$$C(M) = \sum_{j \to (i,k) \text{ in } S} k p_{ij} = \sum C_j^S$$

M^* is a min-cost J-perfect matching.

So $C(M^*) \leq OPT_{R||\sum C_j}$

(Note: In M^* if $(j, (i,k)) \in M^*$, then $\forall k' < k$, can assume $\exists$ edge incident to (i,k') in M^*, since otherwise, we can replace $(j, (i,k))$ in M^* by $(j, (i,k'))$ to get lower cost J-perfect matching)

So, schedule constructed by $j \to (i,k)$ iff $(j, (i,k)) \in M^*$ is a valid schedule.

i.e. $j \to (i,k)$ means $k-1$ jobs follow j on mc i) and $\sum C_j$ of schedule $= C(M^*)$

$\leq OPT_{R||\sum C_j}$

□

June 28th

Lecture 17

R|prmtn|Cmax and O|prmtn|Cmax:

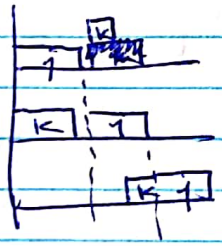
O: Open shop environment

- Each job j consists of m operations; operation i of job j has to be scheduled on mc i, and takes p_{ij} time to complete. Operations of a job can be performed in any order, but at any point of time, at most one operation of a job can be processed.
- The completion time of a job = time by which all operations of the job have completed.

So,

C_{max} = max completion time of a job
= max completion time of an operation

Ex:



$\sum_{i=1}^{m} p_{ij}$ Must spend time processing the operations of job j.

O|prmtn|Cmax:

$C_{max}^{*} \geq \max\left\{ \max_{i=1,\ldots,m} \sum_{j=1}^{n} p_{ij}, \max_{j=1,\ldots,n} \sum_{i=1}^{m} p_{ij} \right\}$

mc i must spend this much time processing the jth operation of jobs.

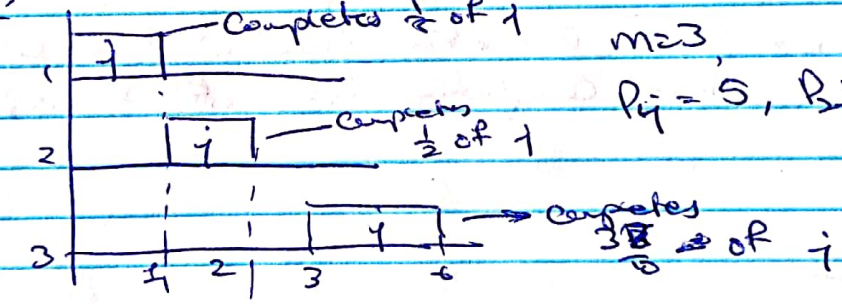
Call this LB

Theorem 1.
We can efficiently compute a schedule for O|pmtn|Cmax (with at most mn+m+n preemptions) with makespan = CB; and hence is an opt schedule.

Theorem 2:
R|pmtn|Cmax ≤p O|pmtn|Cmax

Completion time of j = time when j is processed to the extent of 1.

Ex:



m=3,
$P_{ij} = 5$, $P_{2j} = 2$, $P_{3j} = 10$

So, $\frac{1}{2} + \frac{1}{5} + \frac{3}{10} = 1$.

Proof] (Thm 2).
Will write down an LP-relaxation for R|pmtn|Cmax.
Let variables x_{ij}: Fraction of job j scheduled on m/c i
$\forall i = 1, \ldots, m$, $\forall j = 1, \ldots, n$.

↪ Cont

[Proof] (cont'd)

(LP) min C_{max} → Variable to denote makespan

s.t.

(1) $\sum_{i=1}^{m} x_{ij} = 1$ $\forall$ jobs j. (All jobs completed to the extent of 1)

(2) $\sum_{j=1}^{n} p_{ij} x_{ij} \le C_{max}$ $\forall$ m/c i.

(3) $\sum_{i=1}^{m} p_{ij} x_{ij} \le C_{max}$ $\forall$ jobs j

$C_{max}, x_{ij} \ge 0$ $\forall i,j$

Fact: $OPT_{LP} \le OPT_{R|pmtn|C_{max}}$.

Reduction: Find an opt. soln (x^{LP}, C_{max}^{LP}) to (LP). Create following input for $R|pmtn|C_{max}$

For every job j, create an operation O_{ij} with length $p_{ij} x_{ij}$ $\forall i = 1, \ldots, m$

Then,

1) LB for open-shop instance is $\le C_{max}^{LP}$ (follows from (2), (3))
2) Theorem 1 produces an open-shop schedule S of makespan $= LB \le C_{max}^{LP}$
3) S is feasible for $R|pmtn|C_{max}$.

On m/c i, we spend $p_{ij} x_{ij}$ time on job j, so since $\sum_{i=1}^{m} x_{ij} = 1$ $\forall j$, we completely process each job. The $p_{ij} x_{ij}$ time units spent on i for j do not overlap with the $p_{ij} x_{ij}$ time spent on some other machine i' for j.

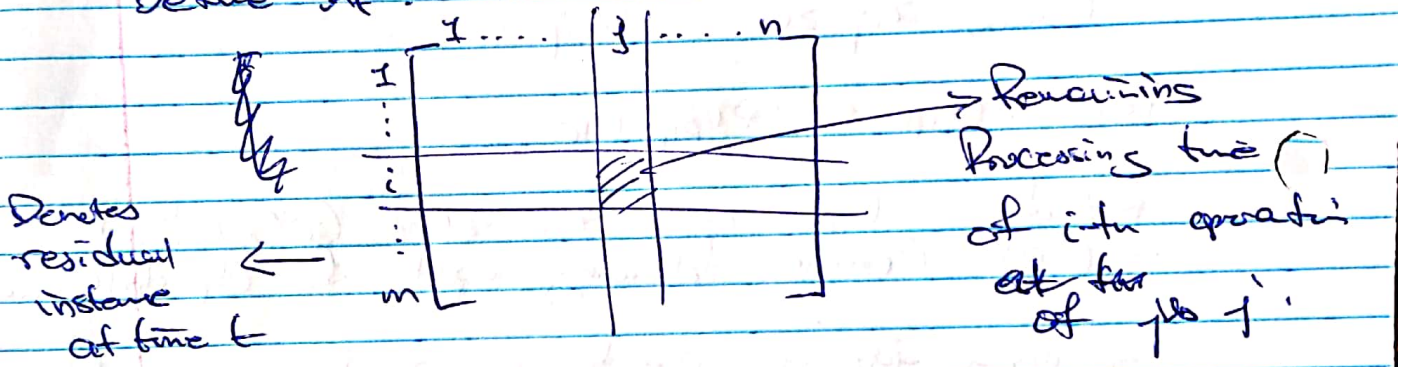
Hilroy

↪

[Proof] (cont)
(Every mle is processing ≤ 1 job at any time)
So, (1) + (2) + (3) ⇒ S optimal for R|pmtn|Cmax.
∎.

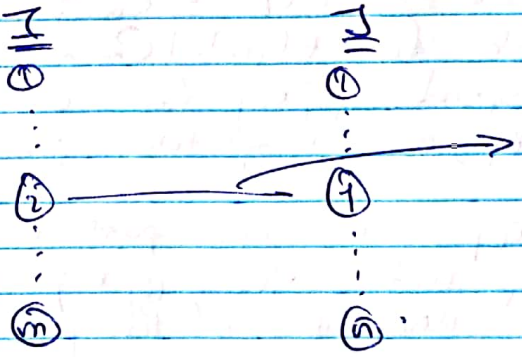
[Proof] (of Thm 1)

Suppose we have built a schedule up to time t.
Define A_t : m × n matrix



Let Δ_t = max row/col sum in A_t
At $t=0$, $\Delta_0 = B$

With A_t, have a corresponding bipartite graph



[Proof] (cont.)

Define:

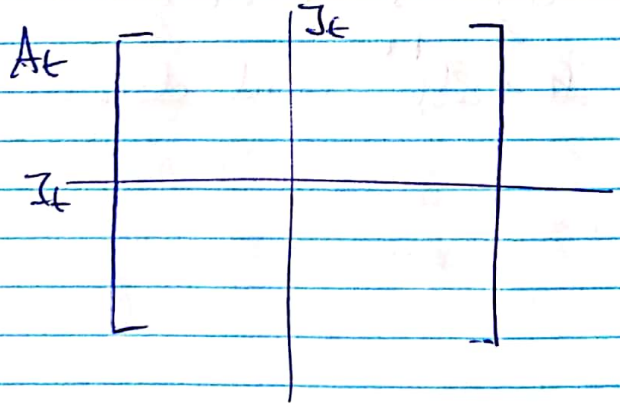
$I_t = \{ i \in \{1, \dots, m\} : \text{row-sum for } i = \Delta_t \}$

$J_t = \{ j \in \{1, \dots, n\} : \text{col-sum for } j = \Delta_t \}$.

Observations:

1) At any time t, the assignment of operations of jobs to m/c's forms a matching in G_t
2) $\Delta_t \geq \Delta_0 - t$
3) After time t, it takes $\geq \Delta_t$ units to complete all operations, so $C_{max} \geq t + \Delta_t \geq \Delta_0$.

So to get $C_{max} = \Delta_0$ we must have $\Delta_t = \Delta_0 - t$ and we can take any Δ_t time units after t to complete all operations.
$\Rightarrow$ At time $t+1$, we must have $\Delta_{t+1} = \Delta_t - 1$



To get $\Delta_{t+1} = \Delta_t - 1$, we must pick an $(I_t \cup J_t)$ - perfect matching in G_t

Algorithm:
Starting at time $t=0, 1, \ldots,$
- We pick an $(I_t \cup J_t)$-perfect matching M in G_t
- Run M for 1 time unit (assuming all operation lengths are integer)
- Update $A_t, \Delta_t, I_t, J_t, t$

This will return a schedule of makespan Δ_0.

2 Q's:
- Why should even a $(I_t \cup J_t)$-perfect matching exist

What prevents M from being an $(I_{t+1} \cup J_{t+1})$ perfect matching?
- The sets I_t and J_t could change, i.e. $I_{t+1} \neq I_t$ and $J_{t+1} \neq J_t$. (But, $I_{t+1} \supseteq I_t$ and $J_{t+1} \supseteq J_t$)
- An edge of M could disappear from G_t

July 3rd, 2018

11 Lecture 18

Recall: (Algorithm A) (*)

For $t = 0, 1, 2, \ldots, \Delta_0$

Let A_t = matrix of remaining processing times

Let Δ_t = max {row or column sum of A_t}

Let $I_t = \{i \in I : \text{sum of row } i \text{ in } A_t = \Delta_t\}$

Let $J_t = \{j \in J : \text{sum of col } j \text{ in } A_t = \Delta_t\}$

Let G_t = bipartite graph on $I \cup J$ s.t. $ij \in E(G_t)$ iff $[A_t]_{ij} > 0$

Compute M_t : a $(I_t \cup J_t)$-perfect matching in G_t

Schedule jobs according to M_t from time t to $t+1$ (**)

Recall!

We were finishing Optmtl Cmax.

$\Rightarrow$ We can reduce $R|pmtn|C_{max} \leq_p O|pmtn|C_{max}$

Claim (from last class)

Algorithm A computes a schedule of makespan Δ_0

Proof:

We showed that since $\Delta_{t+1} = \Delta_t - 1 \Rightarrow \Delta_t = \Delta_0 - t$

$\Rightarrow$ @ time Δ_0, all jobs are completed

$$C^*_{max} \geq \sum_{i \in I} p_{ij} \quad \forall j \in J$$

$$C^*_{max} \geq \sum_{j \in J} p_{ij} \quad \forall i \in I$$

$$C(LB) = \max\left\{\max_{j \in J} \sum_{i \in I} p_{ij}, \max_{i \in I} \sum_{j \in J} p_{ij}\right\}$$

Milroy

Issues to resolve:

(1) @ time t, why does a $J_t \cup I_t$-perfect matching exist?
(2) When can we reuse M_t as M_{t+1}?

Goal: Bound # of matchings we use $\leq mn + m + n$.

Idea from last class:
What prevents M_t from being a $(J_{t+1} \cup J_{t+1})$-perfect matching in G_{t+1}?
$\leq m_1$) Complete a job if $p_{ij} \leq 1$.
$\leq m_2$) $J_{t+1} \neq J_t$, there's a new tight machine
$\leq n$ 3) $J_{t+1} \neq J_t$, there's a new tight job

How long to use M_t?
"Run" the schedule given by M_t for a time interval of length δ UNTIL
- one of the p_{ij}'s goes to 0,
- $\Delta_t - \delta$ becomes equal to row $\hat{i}$'s sum for $\hat{i}$ not matched by M_t
- $\Delta_t - \delta$ becomes equal to column $\hat{j}$'s sum for $\hat{j}$ not matched by M_t

Now we can replace (in Alg ** A)
→ (*1 w/ : Let $t=0$, while $t < \Delta_0$
→ (**1 w/ :
Compute:

$$\delta = \min \begin{cases} \min_{ij \in M_t} p_{ij} \\ \min_{\hat{i} \text{ not matched by } M_t} (\Delta_t - \text{row sum}_{M_t}(\hat{i})) \\ \min_{\hat{j} \text{ not matched by } M_t} (\Delta_t - \text{col sum}_{M_t}(\hat{j})) \end{cases}$$

Schedule according to M_t from time t to $t+\delta$. Advance $t \leftarrow t+\delta$

Claim:
We can always find a $(I_t \cup J_t)$-perfect matching in G_t at time t

[Proof]
If G_t is bipartite, recall given fractional S-perfect matching, can find integral S-perfect matching

Consider:

$$x_{ij} = \frac{p_{ij}}{\Delta t} \geq 0$$

For $i \in I$

$$\sum_{j \in J} x_{ij} = \frac{1}{\Delta t} \underbrace{\sum_{j \in J} p_{ij}}_{\leq \Delta t} \leq 1$$

$\longrightarrow$ and $=1$ iff $i \in I_t$.

For $j \in J$:

$$\sum_{i \in I} x_{ij} = \frac{1}{\Delta t} \underbrace{\sum_{i \in I} p_{ij}}_{\leq \Delta t} \leq 1$$

$\longrightarrow$ and $=1$ iff $j \in J_t$.

(This is a fractional S-perfect matching, so there is an integral S-perfect matching)

Hilroy

7

Ex:

Iteration 1 (t=0)

$$P = \begin{bmatrix} 2 & 1 & 0 & 2 \\ 0 & 4 & 1 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix} = A_0$$

1. Δ_{t0} = max row sum or colsum

$$\begin{bmatrix} 2 & 1 & 0 & 2 \\ 0 & 4 & 1 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix} \begin{matrix} =5 \\ =5 \\ =4 \end{matrix}$$

$\quad 4 \quad 5 \quad 1 \quad 4$

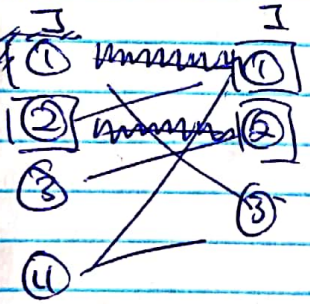
So, $\Delta_0 = 5$

2. $I_0 = \{1, 2\}$

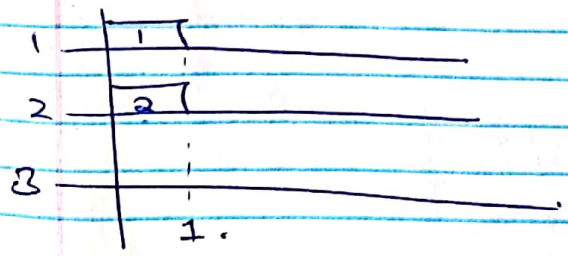
3. $J_0 = \{2\}$

4. H_0 (J, I)

$H_0 = \{(1,1), (2,2)\}$



5. $\xi = \min \begin{cases} \min\{2, 4\} & \leftarrow P(1,1),\ P(2,2) \\ \min\{5-4\} & = 1 \\ \min\{5-1, 5-4\} & \end{cases}$



cut

Iteration 2 ($t=1$)

$$A_1 = \begin{bmatrix} \textcircled{1} & 1 & 0 & 2 \\ 0 & \textcircled{3} & 1 & 0 \\ 2 & 0 & 0 & \textcircled{2} \end{bmatrix} \begin{matrix} =4 \\ =4 \\ =4 \end{matrix}$$

$$\begin{matrix} " & " & " & " \\ 3 & 4 & 1 & 4 \end{matrix}$$

$\Delta_1 = 4$

$I_1 = \{1, 2, 3\}$

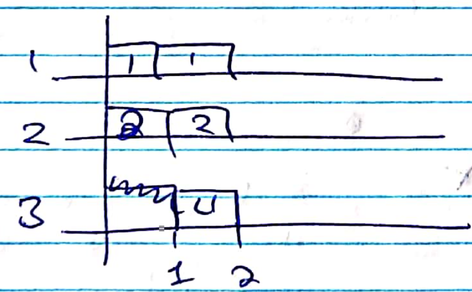
$J_1 = \{2, 4\}$

$M_1 =$ (See circles)

(J, I)

→ So, we have! $\{(1,1), (2,2), (3,4)\}$

$$\delta = \min \begin{cases} \min\{1,3,2\} \\ \min\{\infty \\ \min\{4-1\} \end{cases} = 1.$$



Iteration 3:

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & (2) \\ 0 & (2) & 1 & 0 \\ (2) & 0 & 0 & 1 \\ 2 & 3 & 1 & 3 \end{bmatrix} \begin{matrix} 3 \\ 3 \\ 3 \end{matrix}$$

$\Delta_2 = \{3\}$, $I_2 = \{1,2,3\}$, $J_2 = \{2,4\}$

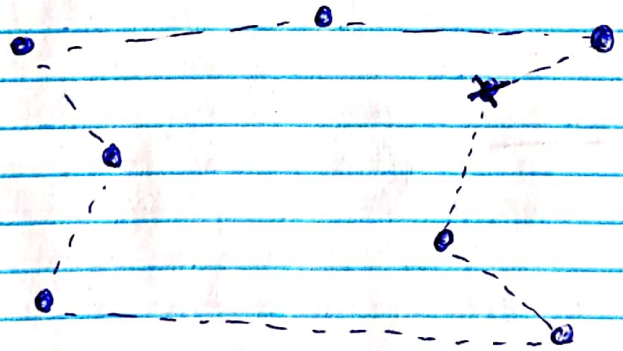
(continuing on $\delta = 2$)

Cont. iteration 4 looks like:

$$A_3 = \begin{bmatrix} 0 & [1] & 0 & 0 \\ 0 & 0 & [1] & 0 \\ 0 & 0 & 0 & [1] \end{bmatrix}, \text{ etc.})$$

Travelling Salesman Problem:

→ We can show that this is equivalent to "jobshop w/ setup times":

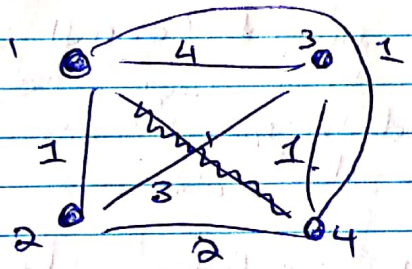


→ Want to travel through all nodes and returning to starting node ⇒ w/ shortest distance/cost.

Defn (Travelling Salesman Problem - TSP)
Given complete graph $G = (V, E)$ with edge costs $c_e \geq 0$ for all $e \in E$. We want to find a (simple) cycle of min. cost that visits all the nodes (called tours in this context)
↓
No repeated vertices.

Ex:
$G = (\{1,2,3,4\}, \{12, 13, 14, 23, 24, 34\})$

	↙	↓	↓	↓	↓	↓
$c_e =$	1	4	1	3	2	1



Q: Is this problem NP-hard?
A: YES.

Q: Can we find a k-approx algorithm?
A: No, not for arbitrary edge weights

(But, we can do this if ~~the edge~~ ~~set~~ we assume the triangle inequality)

Defn
Edge costs c_e for all $e \in E$ satisfy the triangle inequality if:

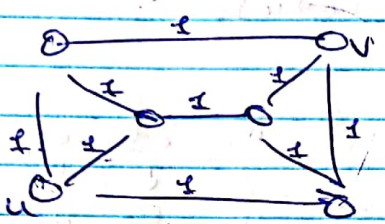
$$c_{uv} \leq c_{uw} + c_{wv} \quad \text{for } u, v \in V.$$

Hilroy

Def'n
We call TSP instance metric if the edge costs satisfy the triangle inequality

Ex: Metric TSP instance

G':



If $e \in G'$, then c_e = length of the shortest uv-path ($e = uv$)

↳ i.e. uv is not in this graph, the cost of uv would be 2. in order to preserve the metric.

Claim: If edge costs satisfy the triangle inequality, then $\forall u, v \in V$, $c_{uv} \leq c(P)$ for any u,v-path P.

(If $P = u = w_0 w_1 \ldots w_k = v$ then $c(P) = \sum_{i=0}^{k} c_{w_i w_{i+1}} = \sum_{e \in P} c_e$.).

[Proof]

$c(P) = c_{w_0 w_1} + c_{w_1 w_2} + \ldots + c_{w_{k-1} w_k}$

$\geq c_{w_0 w_2}$

$\geq c_{w_0 w_3}$

→ Do induction on the length of the path. □

July 5th 2018

12 Lecture 19

COMS4

Recall:

Defn (TSP)

GIVEN: Complete graph $G=(V,E)$ and edge costs $c_e \geq 0$ for all $e \in E$

FIND: A simple cycle T (called TSP Tour) that visits every vertex of min. total cost

i.e. $c(T) = \sum_{e \in T} c_e$

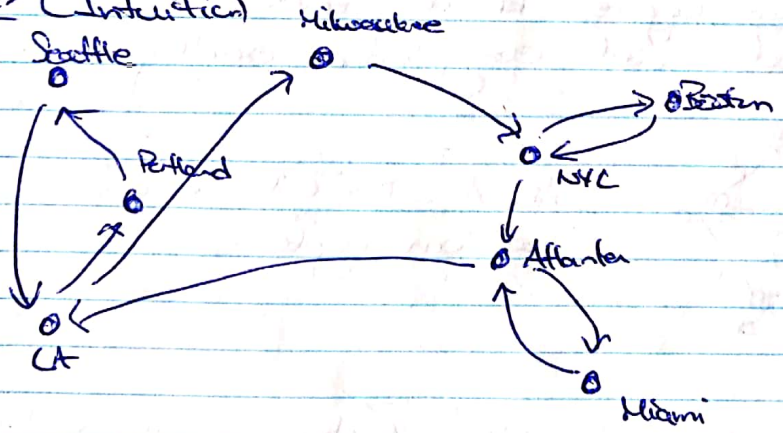
TODAY

GOAL Today: Design approx. algorithm for metric instances. (i.e. edge costs satisfy triangle inequality $\forall u,v,w \in V$, $c_{uw} \leq c_{uv} + c_{vw}$

Claim 1: If edge costs satisfy triangle inequality for all $u,v \in V$ and uv-path P

$c_{uv} \leq c(P)$

Ex: (Intuition)



Greedy Right goes via the above

Problem: This is not a TSP tour

How can we fix this? → Remove duplicates

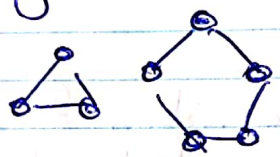
i.e. Approximate in order of visited time!

Graph Theory

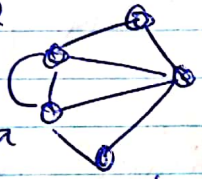
Defn (Eulerian)
A (multi)-graph is called Eulerian if all the nodes have even degree

Ex:

1) All Cycles are Eulerian:



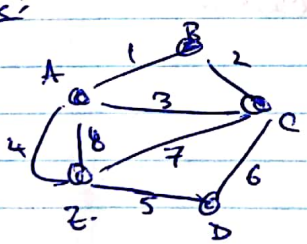
2)



Edge is duplicated, this is a multigraph example

Defn (Eulerian Cycle)
An Eulerian Cycle in a graph is a closed walk that visits every edge exactly once.

Ex:



ABCAEDCEA is an Eulerian cycle.

Lemma: Every connected Eulerian graph has an Eulerian cycle.
(Note: - Eulerian Cycle will visit all vertices.
- We can find this in poly time!
Proof:
Exercise - Do DFS. ("Henry's"); Glue cycles together

July 5th, 2018

Strategy:

1) Find an Eulerian subgraph H of G s.t. $C(H) \leq 2 \cdot OPT_{TSP}$.
2) From H, obtain a tour T of cost $C(T) \leq C(H)$

Then, we have:

$$C(T) \leq C(H) \leq 2 \cdot OPT_{TSP}$$

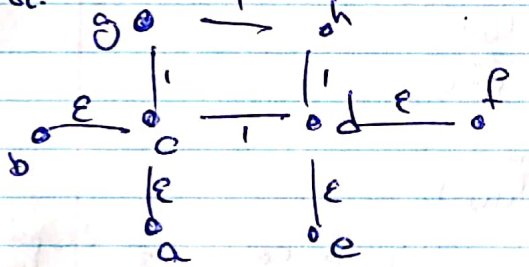
E' can take multiple copies of edges of E

Claim 2:

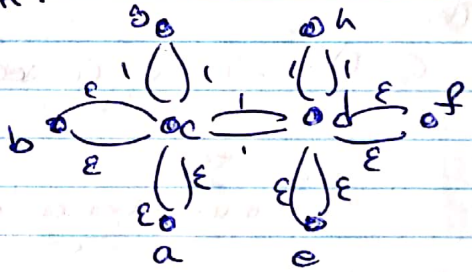
From a connected Eulerian "subgraph" $H = (V, E')$ of G we can find a TSP tour T of with $C(T) \leq C(H) = \sum_{e \in E'} c_e$.

Idea behind proof: (Edges not drawn take on weights of the shortest path)

G:

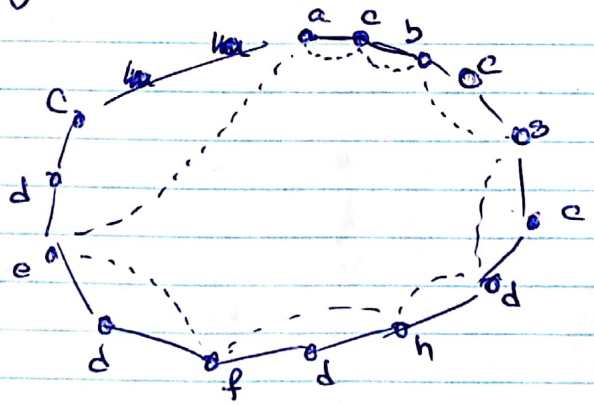


H:



Step 1: Find an Eulerian cycle

acbcgcdhdfdedca is such a cycle.



Step 2: Go around the cycle and skip vertices if we have already seen them (dotted lines)

Tour T: acbgdhfea.

[Proof] (Sketch)
From Eulerian H., obtain Eulerian cycle $\tilde{\tau}$ by DFS. Obtain tour τ from $\tilde{\tau}$ by "Shortcutting" edges.
i.e. Let τ be obtained from $\tilde{\tau}$ by taking vertices in order of first appearance on $\tilde{\tau}$ starting from an arbitrary vertex r.

If u, v consecutive on τ, then
$c_{uv} \leq$ [cost of portion of $\tilde{\tau}$ from first appearance of u to the first appearance of v] $= \tilde{\tau}_{uv}$.

Then,
$$c(\tau) = \sum_{u,v \text{ consecutive on } \tau} c_{uv}$$
$$\leq \sum_{u,v \text{ consecutive on } \tau} c(\tilde{\tau}_{uv})$$
Since $\tilde{\tau} = \bigcup_{u,v \text{ consecutive on } \tau} \tilde{\tau}_{u,v}$ (disjoint unions)
$$= c(\tilde{\tau})$$
$$= c(H)$$

∎

This completes the proof of step 2.

Recall:
Step 1: Finding "cheap" Eulerian subgraphs.
GOAL:
Find H, connected Eulerian, s.t

$$c(H) \leq 2 \cdot OPT_{TSP}$$

Observation:

(Spanning tree bound)
Let T^* be an optimal TSP tour.
Then,

$$OPT_{TSP} = c(T^*) \geq c(T^* - uv) \text{ for any edge } uv \in T^*$$

Since $T^* - uv$ is a spanning tree $\geq MST(G, c)$ ← min cost spanning tree in G under edge weight c.

$$\Rightarrow 2 \cdot OPT_{TSP} \geq 2 \cdot MST(G, c)$$

Algorithm: (2-Approx.)
1. Find MST $F \subseteq E$ of G.
2. Take 2 copies of every edge in F to obtain H.
3. Apply claim 2 to obtain tour T.

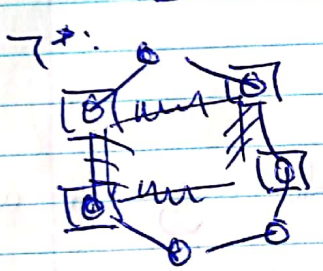
Claim: H is Eulerian
[Proof] Exercise.

3/2-Approx:

Idea: Fix odd degree vertices by matching them with each other

Claim: Let $U \subseteq V$ with $|U|$ even. Let $P_{min}(U)$ be the min cost of a perfect matching in $G[U]$. Then, $OPT_{TSP} \geq 2 P_{min}(U)$.

Picture:



Proof:
Take optimal tour T^* and shortcut to obtain T_U^* containing only the nodes of U.
This cycle can be decomposed into two U perfect matchings M_1 and M_2.
Then,

$$OPT_{TSP} = c(T^*) \geq c(T_U^*) = c(M_1) + c(M_2)$$
$$\geq 2 P_{min}(U)$$

□

Christofides Algorithm

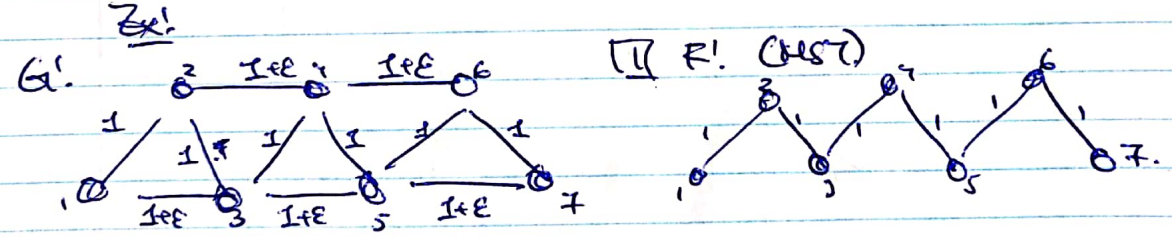
1) Find MST F
2) Let $U = \{$odd degree vertices in $F\}$
   Remark: $|U|$ is even - Handshake lemma
3) Find min-cost perfect matching
   $M^* \subseteq E$ in $G[U]$
4) Let $H = F \cup M^*$
   Remark: H is connected and Eulerian.
   [Proof] - Exercise
5) Apply claim 2 to obtain T.

Then,

$$c(T) \le c(H) = c(F) + c(M^*)$$
$$\le OPT_{TSP} + \tfrac{1}{2} OPT_{TSP}$$
$$= \tfrac{3}{2} \cdot OPT_{TSP}.$$

ϕ

Ex!



1. $\{1, 7\}$ are vertices w odd degree!
2. $\{17\} = M^*_{\{1,7\}}$
3. $H = F \cup \{17\}$.
4. We're done!

Rmk: The min tour is 1246753 with cost 10+4ε.

July 10th 2018

13 Lecture 20

COMSC(

R||Cmax:
Consider the following LP-relaxation, which is a variant of the LP for R|pmtn|Cmax.

(LP)

min Cmax
s.t.

(1) $\sum_{i=1}^{m} x_{ij} = 1 \quad \forall j = 1, \ldots, n$ (Ideally, $x_{ij}=1$ if $j \to i$, and 0 otherwise)

(2) $\sum_{j=1}^{n} p_{ij} x_{ij} \le C_{max} \quad \forall i = 1, \ldots, m.$

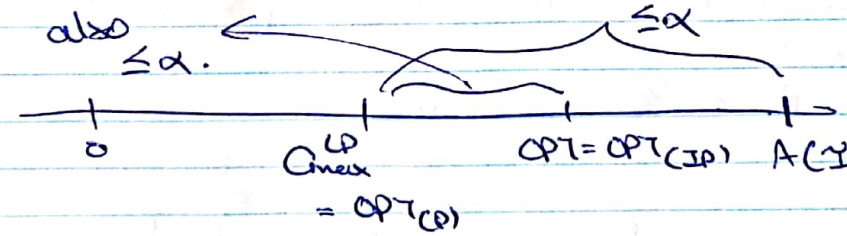
(3) $x, C_{max} \ge 0$

Observations:
(i) Any integral sol'ns to (P) corresponds to a schedule for R||Cmax. So:

OPT = {min C_{max} s.t. (1), (2), (3) and x_{ij} integral $\forall i,j$}

(IP)

(ii) (P) is the LP-relaxation of (IP)



Suppose we have an α-approx. alg A that for every instance I returns a solution of value $A(I) \le \alpha \cdot C_{max}^{LP}(I)$
i.e. It proves an α-approx. using the LP-optimum $C_{max}^{LP}(I)$ as the lower bound on $OPT(I)$.

(cont)
Want to understand: What is an inherent lower bound on α?

$$OPT(I) \le A(I) \le \alpha \cdot C_{max}^{LP}(I)$$

$$\Rightarrow \frac{OPT(I)}{C_{max}^{LP}(I)} \le \alpha.$$

$$\Rightarrow \alpha \ge \max_{\text{instances } I} \frac{OPT(I)}{C_{max}^{LP}(I)} \longrightarrow \text{let's call this } \beta.$$

Integrality gap of (P)

So, any algorithm that uses C_{max}^{LP} as a lower bound to prove an approx. guarantee must have approximation factor at least β.

Claim: β is "large" for (P)

Proof:
Consider an instance of $P||C_{max}$ with m jobs with $p_{ij} = 1 = p_{ij}$ $\forall i = 1, \ldots, m$
Clearly $OPT(I) = 1$. But, there is a feasible LP sol'n with $x_{ij} = 1/m$ $\forall i = 1, \ldots, m$ and $C_{max} = 1/m$.
So, $\beta \ge \frac{1}{1/m} = m$.

□

(Summary: This LP sucks, and doesn't provide anything meaningful.)

How can we fix (P)?

- Would like to add to (P) the constraint:

$$x_{ij} > 0 \Rightarrow C_{max} \geq p_{ij} \quad (*)$$

But this is not a linear constraint.
We can move to the decision version of $R||C_{max}$
i.e.

Given target makespan D, decide whether there is a schedule for $R||C_{max}$ of makespan $\leq D$?

Now we can incorporate (*) as follows:
Define $F = \{(i,j) : p_{ij} > D\}$.
Consider the feasibility LP:
(LP_D)

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \forall j = 1, \ldots, n$$

$$\sum_{j=1}^{n} p_{ij} x_{ij} \leq D \quad \forall i = 1, \ldots, m.$$

$$x \geq 0$$

$$x_{ij} = 0 \quad \forall (i,j) \in F. \leftarrow \text{This is new!}$$

Observation:
If $D \geq OPT(I)$, (LP_D) is feasible
(More generally, if the answer to the decision problem is YES, (LP_D) is feasible).

Theorem 1:
If (LP_D) is feasible, we can efficiently find a schedule of makespan $\leq 2D$.

Go on!

Theorem 1 leads to 2-approx for R||Cmax:
(Assume all P_{ij}'s are integers, so OPT is also an integer)
- Use binary search to find smallest integer D s.t. (LP_D) feasible
(Observation $\Rightarrow D \leq OPT$)
- Use theorem 1 to obtain a schedule of makespan $\leq 2 \cdot D^* \leq 2 \cdot OPT$.

← Call this D^*

Running time:

(time taken for binary search) + (time to implement Theorem 1)

= ((poly * iterations of binary search) * (time to solve (LP_D)))

+ polytime

= polytime.

[Proof] (of Theorem 1)
Let $x = (x_{ij})_{\substack{i=1,\dots,m \\ j=1,\dots,n}}$: feasible solution to (LP_D)

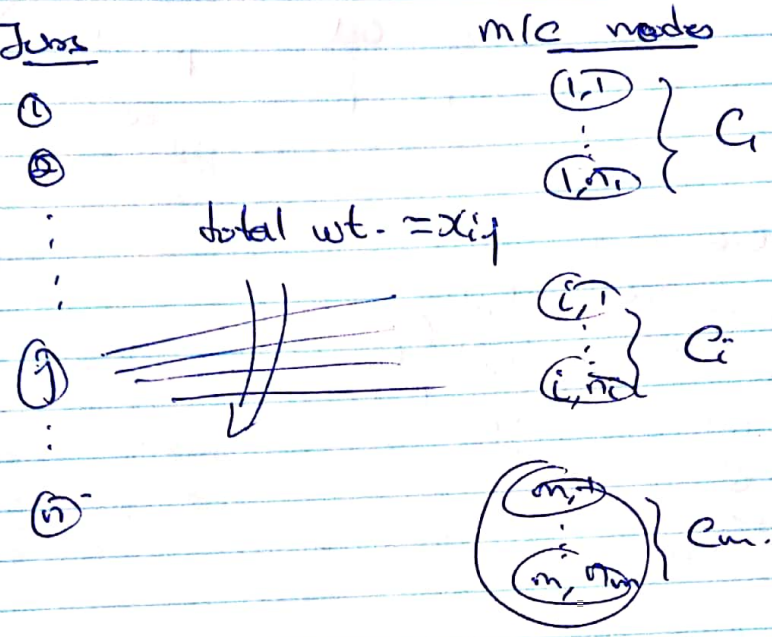
We will build a bipartite graph with job nodes J and some nodes for m/c's s.t.
1) x translates to a fractional J-perfect matching, and.
2) Any J-perfect matching gives a schedule of makespan $\leq 2D$.

Define $n_i = \lceil \sum_{j=1}^{n} x_{ij} \rceil$ = # sol'n x assigns some # $\in [n_{i-1}, n_i]$ jobs on m/c i.

Create for each m/c i, n_i nodes $(i,1), \ldots, (i,n_i)$ = copies
c_i

Jobs

m/c nodes



Will have edges b/w j and same copies of m/c i and assign same wts. $w_{j,(i,c)}$ on these edges

Actual bipartite graph ≡ edges $(j, (i,c))$ s.t $w_{j,(i,c)} > 0$

wts. assigned to $(j, (i,c))$ edges $\forall c = 1, \ldots, n_i$ will have the property that:

$$\sum_{c=1}^{n_i} w_{j,(i,c)} = x_{ij} \quad \forall j, i$$

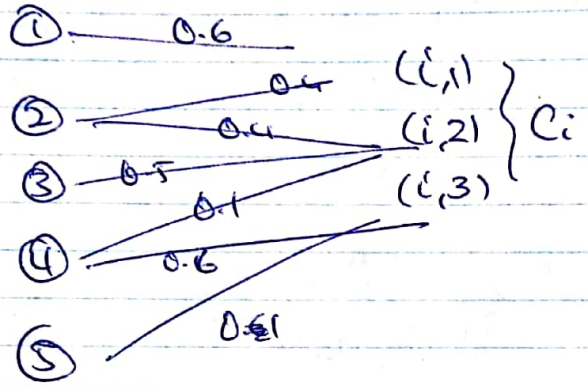
Defining $w_{j,(i,c)}$ wts for a fixed m/c i, but for all $j=1,\ldots,n$, $\forall c = 1, \ldots, n_i$.

- Order the jobs $J_i = \{j : x_{ij} > 0\}$ in ↓ order of p_{ij}
- "Pack the x_{ij}'s on the n_i copies of m/c i" by using McNaughton's wrapround rule and considering jobs in J_i in above sorted order

Ex:

	1	2	3	4	5
x_{ij}	0.6	0.8	0.5	0.7	0.1
p_{ij}	8	4	3	1	1

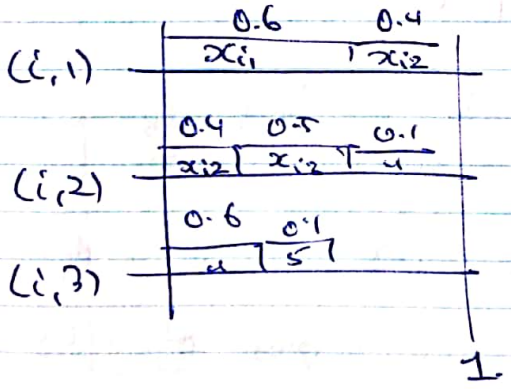
$\sum_{j=1}^{5} x_{ij} = 2.7$, so $n_i = 3$



- Think of each copy $C(i,q)$ as an mk
- Each $j \in J_i$ creates a "job" with length x_{ij}.

Pack x_{ij}'s on n_i copies → each copy can take ≤ 1 job.

Here sorted order: 1, 2, 3, 4, 5



$W_{j,(i,c)}$ = amount of j assigned to any c under packing given by McNaughton's wraparound rule

Observations:

1) $\sum_{c=1}^{n_i} w_{j,(i,c)} > 0 \Rightarrow x_{ij} > 0$

2) $\sum_{c=1}^{n_i} w_{j,(i,c)} = x_{ij}$

3) $w_{j,(i,c)} > 0$ for at most 2 (consecutive) copies c

4) For every copy $c > 1$, every j with $w_{j,(i,c)} > 0$ has $p_{ij} \le p_{ik}$ for all jobs k with $w_{k,(i,c-1)} > 0$

Also $w_{j,(i,c)} \ge 0$ we have $\sum_k w_{k,(i,c-1)} = 1$,

so, $p_{ij} \le \sum_k p_{ik} w_{k,(i,c-1)}$.

July 12th, 2021

14 Lecture 21

454

Recall:

R||Cmax:

- Given target makespan D, solve

(LP_D)

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad \forall j = 1, \ldots, n$$

$$\sum_{j=1}^{n} p_{ij} x_{ij} \leq D \qquad \forall i = 1, \ldots, m$$

$$x \geq 0, \quad x_{ij} = 0 \text{ if } p_{ij} > D$$

Let x be a feasible solution to (LP_D)

Show: x can be rounded to obtain schedule of makespan $\leq 2D$.

Rounding Algo: → we can think about these n_i copies as placeholders for jobs (slots)

Define $n_i = \lceil \sum_{j=1}^{n} x_{ij} \rceil \quad \forall i = 1, \ldots, m$

Create bipartite graph w a node for each job, nodes (i,c) for all $c = 1, \ldots, m$ and $c = 1, \ldots, n_i$.

Let:

- J = job nodes, and
- $C_i = \{(i,c) : c = 1, \ldots, n_i\}$.

Determine edges btw J and C_i by defining wts:

$$w_{j,(i,c)} \quad \forall j = 1, \ldots, n \quad \forall c = 1, \ldots, n_i$$

and create an edge $(j, (i,c))$ if $w_{j,(i,c)} > 0$

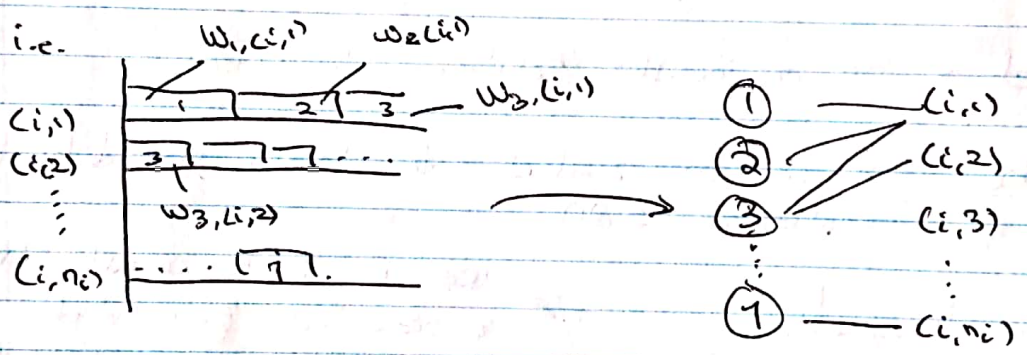
Claim

How do we define $w_{j,(i,c)}$?

Fix an m/c i. Order jobs $J_i = \{j : x_{ij} > 0\}$ in decreasing p_{ij} order. "Pack" the x_{ij}'s on the n_i-copies $(i,1), \ldots, (i,n_i)$ by considering jobs in this order and packing them on a copy up to boundary T. (McNaughton's Wraparound Rule).

Define $w_{j,(i,c)}$ = amount of j assigned to copy c.

i.e.



Algorithm (cont'd)

Find a J-perfect matching in the bipartite graph that we create. If we include edge $(j,(i,c))$ in our matching, that corresponds to scheduling j on m/c i.

Show: Schedule has makespan $\leq 2D$

(We will use $w_{j,(i,c)}$ to show this)

July 12th, 2020

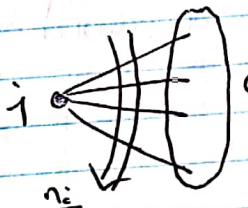
Properties of w_{ij}'s:

Fix i, $\{w_{ij}, (i,c)\}_{\substack{j=1,\dots,n \\ c=1,\dots,n_i}}$ satisfy following properties:

1) $\forall j$, $\sum_{c=1}^{n_i} w_{ij(i,c)} = x_{ij}$

$\Rightarrow \{w_{j,(i,c)}\}_{\substack{j=1,\dots,n \\ i=1,\dots,m \\ c=1,\dots,n_i}}$ gives a fractional J-perfect matching in our bipartite graph

i.e.



$\Rightarrow$ each job node sees a total weight of 1

$\sum_{c=1}^{n_i} w_{ij(i,c)} = x_{ij}$

And each machine copy (i,c) sees a total weight $\sum_j w_{j,(i,c)} \leq J$, since we pack x_{ij}'s up to a bound of 1.

2) By one of our facts on bipartite matching our bipartite graph has a J-perfect matching M.

3) Consider $c>1$. Consider any j with $w_{j,(i,c)} > 0$ and any k with $w_{k,(i,c-1)} > 0$.
Then $P_{ij} \leq P_{ik}$.
Also if $w_{j,(i,c)} > 0$, then $\sum_k w_{k,(i,c)} = 1$.
So, for any $c>1$, any j with $w_{j,(i,c)} > 0$:

$$P_{ij} \leq \sum_k w_{k,(i,c-1)} P_{ik}. \quad (*)$$

Finishing up the proof: ...

Claim: For any $i=1,\dots,m$, $\sum_{j:j\to i \text{ under matching } M} p_{ij} \le 2D$. → Recall M is a perfect matching

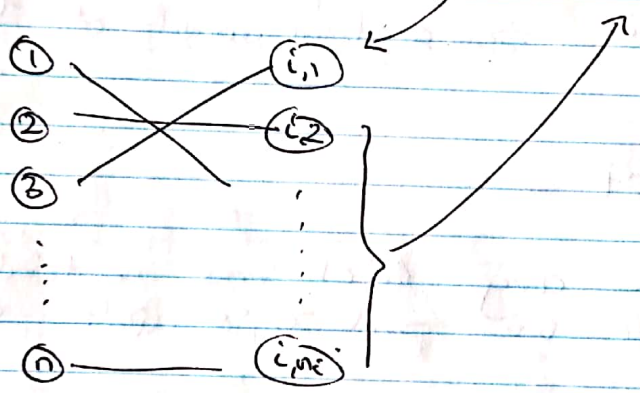
i.e. M contains an edge $(j,(i,c))$ for some $c=1,\dots,n_i$.

Proof:

Total load on $i = \sum_{c=1}^{n_i} \sum_{j:(j,(i,c))\in M} p_{ij}$.

$= \sum_{(j,(i,1))\in M} p_{ij} + \sum_{c=2}^{n_i} \sum_{j:(j,(i,c))\in M} p_{ij}$

i.e.



We want to bound:

$\sum_{(j,(i,1))\in M} p_{ij} + \sum_{c=2}^{n_i} \sum_{j:(j,(i,c))\in M} p_{ij}$

$\le$ max p_{ij} over j with $(j,(i,1))$ is an edge

iff $w_{j,(i,1)} > 0$

$\Rightarrow x_{ij} > 0$

$\Rightarrow p_{ij} \le D$

$\Rightarrow \le D + \sum_{c=2}^{n_i} \sum_{j:(j,(i,c))\in M} p_{ij}$

Cont'd

July 12th, 2013

Proof: (cont'd)

$$\leq D + \sum_{c=2}^{n_i} \sum_{j:(j,i,c) \in M} p_{ij}$$

Only 1 p_{ij} exists since we have a matching

Using (*), we get:

$$\leq D + \sum_{c=2}^{n_i} \left(\sum_k w_{k,(i,c-1)} p_{ik} \right)$$

$$= \sum_k \sum_{c=2}^{n_i} p_{ik} w_{k,(i,c-1)}$$

$$= \sum_k p_{ik} \underbrace{\sum_{c=1}^{n_i - 1} w_{k,(i,c)}}_{\leq x_{ik}}$$

Since $\sum_{c=1}^{n_i} w_{k,(i,c)} = x_{ik}$,

but we are excluding a copy.

$$\leq D + \underbrace{\sum_k p_{ik} x_{ik}}_{\leq D, \text{ one of the constraints in } (LP_0)}$$

$$\leq 2D.$$

□.

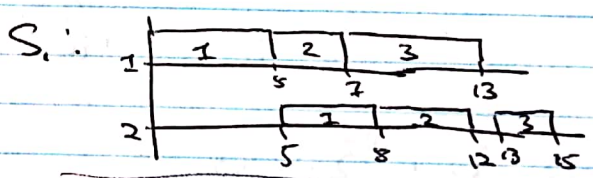
Shop Scheduling:

Flow Shop (FI·1·1)

Each job j consists of m operations where the ith operation has to be processed on m/c i, and takes P_{ij} time. There is a fixed ordering $1, \ldots, m$ of m/c's s.t $\forall$ jobs j, operation 1 must be completed on m/c 1, operation 2 —"— —"— 2, and so on.

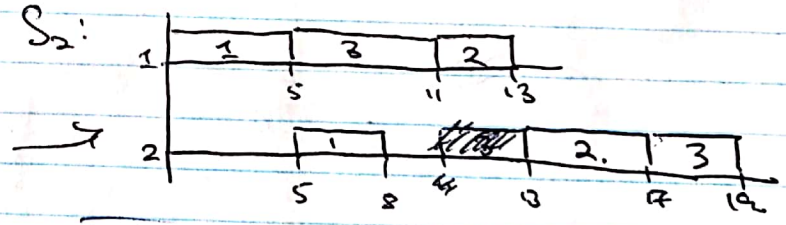
Ex:

	Jobs		
m/c's	1	2	3
1	5	2	6
2	3	4	2



$C_{max} = 15$

On different m/c's, Ordering of jobs can be different



$C_{max} = 19$

Both S_1, S_2 are feasible.

S_1 is called a permutation schedule. (We follow the same sequence of jobs across different m/c's)

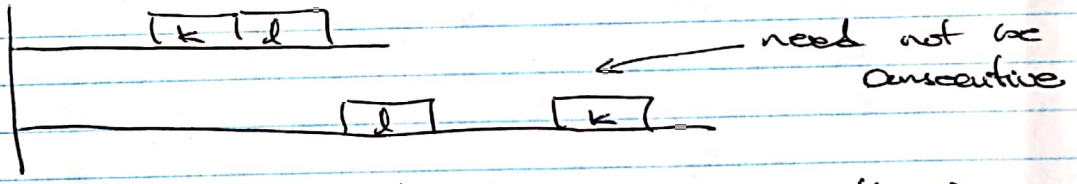
Defn (Permutation Schedule)
All jobs are processed in same order on all m/c's

F2||Cmax:

Lemma: There is always an optimal schedule that is a permutation schedule.

Proof:
This follows from an interchange argument
Let S^*: optimal schedule that is not a permutation schedule.
Then we can find 2 jobs k, l, s.t.
- k comes immediately before l on m/c I.
- l is processed before k (but need not be consecutive)

i.e.

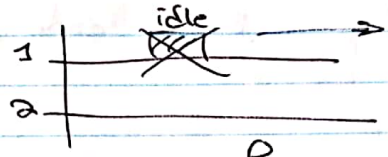


Interchanging l, k on m/c I still gives us a feasible schedule with no greater makespan. So, after a finite # of interchanges, this yields an optimal schedule that is a permutation schedule.

□

O(n!).

On mlc 1, we can always push all idle time towards the end.



→ no reason to add such idle time

(and similarly for mlc 2, we can push idle time towards the beginning) → not as important.

July 11th, 2018

15 Lecture 22

4541

Recall:

$R||C_{max}$: R - flowshop - Each job j consists of m operations. operation 1 has to be completed on m/c 1, operation 2 has to be completed on m/c 2, etc.

Fall C_{max}: There is always an optimal schedule that is a permutation schedule

(Defn - Permutation schedule - All jobs are processed in the same order on all m/c's)

We want to develop an algorithm for Fall C_{max}:

Observations:

1) If we decrease all P_{ij}'s by Δ, where $P_{ij} - \Delta \geq 0 \ \forall i,j$, then the set of optimal permutation schedules is unchanged.

Why?:

- Let S be a permutation schedule corresponding to permutation $1, \ldots, n$. Define $C_{i,j}^S$ = completion time of ith operation of job j in S, where S does not have any unforced idle time

So, $C_{1,j}^S = \sum_{k=1}^{j} P_{1k} \ \forall j$ and in general:

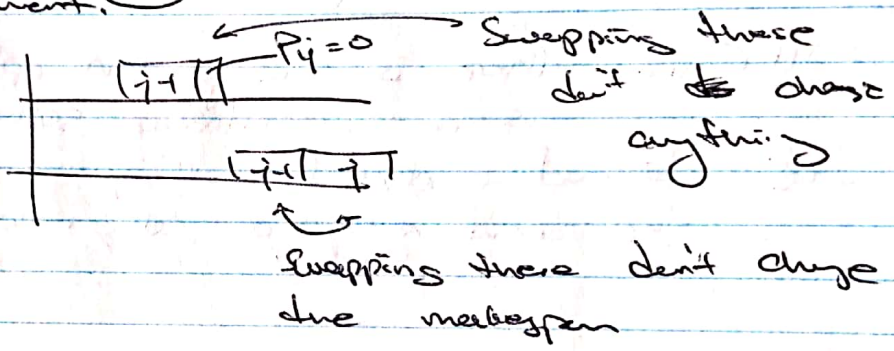
$$C_{i,j}^S = \max\left(C_{i,j-1}^S, C_{i-1,j}^S\right) + P_{ij} \quad \forall i,j$$

$\Rightarrow$ can't

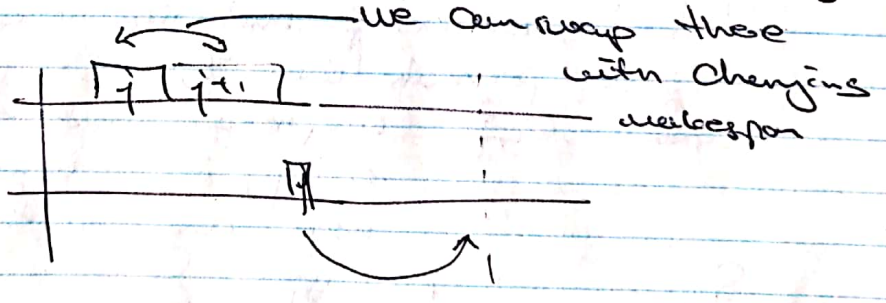
Observations (cont'd):

2) If $P_{ij} = 0$, then there is always an optimal permutation schedule where j is scheduled as the first job.

(this is easy to see w/ an interchange argument:



3) Analogously, if $P_{ji} = 0$ for some j, then there is always an optimal permutation schedule where j is scheduled last. Again, we can see this via an interchange argument.



Algorithm:

1) Decrease all p_{ij}'s by $\Delta := \min_{i,j} p_{ij}$. to get $\hat{p}_{i,j}$'s ($\hat{p}_{ij} = p_{ij} - \Delta \; \forall i,j$)

2a) If some $\hat{p}_{1j} = 0$, then schedule j first and recurse on remaining jobs

2b) If some $\hat{p}_{2j} = 0$, then schedule j last and recurse on remaining jobs.

Algorithm Restated) ⇒ Johnson's Algorithm

Let $J_1 = \{j : p_{1j} \leq p_{2j}\}$ → jobs for which 2a will apply

$J_2 = \{j : p_{2j} < p_{1j}\}$ → jobs for which 2b will apply

We schedule J_1 first in increasing p_{1j} order as this is the order in which $\hat{p}_{1j}$ hits 0. Then we schedule J_2 in decreasing p_{2j} order, as for these jobs $\hat{p}_{2j}$ hits 0 in opposite order.

Ex:

	Jobs	1	2	3	4	5	6
m/c	1	3	2	1	3	2	1
	2	2	4	5	3	1	3

$J_1 = \{2, 3, 4, 6\}$

$J_2 = \{1, 5\}$

So, our permutation is $\underbrace{3, 6, 2, 4}_{J_1}, \underbrace{1, 5}_{J_2}$.

We still need to prove observation 1.

Lemma 1:
Let I be an F2||Cmax instance with P_{ij} operation lengths. Let $\hat{I}$ be the instance obtained from I with operation lengths $\hat{P}_{ij} = P_{ij} - \Delta \geq 0 \quad \forall i,j$. Then S is an optimal permutation schedule for I iff it is an optimal permutation schedule for $\hat{I}$. Moreover, if S is optimal, then $C_{max}^S = opt(I) = OPT(\hat{I}) + (n+1)\Delta$. (where n is the # of jobs)

[Proof].
Let S be an optimal permutation schedule for I corresponding to permutation $1, \ldots, n$.
Let C_{1j}^S, C_{2j}^S be the completion times of operation 1 and operation 2 of j under S for instance I, where S does not have any unforced idle time.

So, $C_{max}^S = OPT(I) = C_{2,n}^S$.
Now schedule jobs in order $1, \ldots, n$ for instance $\hat{I}$ as follows:

- On m/c 1, schedule j so that it completes at time $\hat{C}_{1j}^S = \sum_{k=1}^{j} \hat{P}_{1k} = \sum_{k=1}^{j} (P_{1k} - \Delta) = C_{1j}^S - j\Delta$.

- On m/c 2, schedule j so that it completes at time: $\hat{C}_{2j}^S = (j+1)\Delta$. (i.e. Start j on m/c 2 at time $C_{2j}^S - (j+1)\Delta - \hat{P}_{2j}$)

Proof (cont'd)

We claim: $\hat{C}_{ij}$'s form completion times of a valid flow shop schedule. (i.e. At every time, at most one operation of a job is processed; on any m/c, at any time $\le I$ operation is processed)

We need to check that: (a) $\hat{C}^S_{2j} - \hat{C}^S_{2j-1} \ge \hat{P}_{2j}$
(b) $\hat{C}^S_{2,1} - \hat{C}^S_{1,j} \ge P_{2,1}$

(a) $\hat{C}^S_{2,1} - \hat{C}^S_{2,j-1} = C^S_{2j} - C^S_{2,1} - \Delta$

$\ge P_{2j}$ (since $C^S_{2,j}$'s form valid completion times for a valid schedule of I)

$\ge P_{2j} - \Delta = \hat{P}_{2j}$

(b) $\hat{C}^S_{2,1} - \hat{C}^S_{1,j} = (C^S_{2,j} - (j+1)\Delta) - (C^S_{1,j} - j\Delta)$

$= C^S_{2,j} - C^S_{1,j} - \Delta$

$\ge P_{2j}$ (same reasoning as above)

$\ge P_{2j} - \Delta = \hat{P}_{2,j}$

$\Rightarrow$ S gives a permutation schedule for $\hat{I}$ of makespan $\hat{C}^S_{2,n} = C^S_{2,n} - (n+1)\Delta$

$= OPT(I) - (n+1)\Delta$ (*)

$\Rightarrow OPT(\hat{I}) \le OPT(I) - (n+1)\Delta$

[Proof] (cont.)

Conversely, if S is an optimal permutation schedule for $\hat{I}$, then S' leads to a permutation schedule for I of makespan $\leq OPT(\hat{I}) + (n+1)\Delta$ (Exercise)
(&) (**)

(*) and (**) show
- $OPT(I) = OPT(\hat{I}) + (n+1)\Delta$
- S is an opt. perm. schedule for I iff ——— || ——— $\hat{I}$.

□

$R||C_{max}$:

Goal: Devise a polytime permutation algorithm with guarantee: It returns a schedule S with makespan $\leq OPT + (m-1) \cdot p_{max}$ where $p_{max} := \max_{i,j} p_{ij}$.

Notation:
Let $\pi_i = \sum_{j=1}^{n} p_{ij}$ and $\pi_{max} = \max_{i \in \{1,\dots,m\}} \pi_i$.

Clearly, $OPT \geq \pi_{max}$.
Suppose we have a permutation schedule S that schedules jobs in the order: $\sigma(1), \sigma(2), \dots, \sigma(n)$, where $\sigma: \{1,\dots,n\} \to \{1,\dots,n\}$ is a permutation, s.t. there is no unforced idle time.

As usual, $C_{ij}^S :=$ Completion time of ith operation of job j under schedule S.
And, we know $C_{ij}^S =$

And,
$$C^S_{1,O(j)} = \sum_{k=1}^{j} P_{1,O(k)}, \text{ and}$$

$$C^S_{i,O(j)} = \max\{C^S_{i,O(j-1)}, C^S_{i-1,O(j)}\} + P_{i,O(j)}$$

So, $C^S_{max} = C^S_{m,O(n)} = \sum_{j=1}^{n} P_{m,j} + (\text{Total idle time on m/c } m \text{ in } [0, C^S_{max}])$.

July 10th, 2018

16 Lecture 23

454

Recall:

FlCmax: Let $T_i := \sum_{j=1}^{n} P_{ij}$ and $T_{max} := \max_{i=1,\dots,m} T_i$

Suppose we have a permutation schedule S corresponding to permutation $\sigma(1), \sigma(2), \dots, \sigma(n)$ with no unforced idle time. So

(b) $\longrightarrow$ $C^S_{i,\sigma(j)} = \max\left(C^S_{i,\sigma(j-1)}, C^S_{i-1,\sigma(j)}\right) + P_{i,\sigma(j)}$

$C^S_{max} = T_m +$ (idle time on mc m in $[0, C^S_{max}]$)

Let $I^S_{i,j}$ = total idle on mc i up to completion time of $\sigma(j)$ on mc i.

So

(a) $\longrightarrow$ $I^S_{i,j} = C^S_{i,\sigma(j)} - \sum_{k=1}^{j} P_{i,\sigma(k)}$ Note that: $I^S_{1,j} = 0 \ \forall j = 1, \dots, n.$

So, $C^S_{max} = T_m + I^S_{m,n}$

We want: Relate $I^S_{i,j}$ to $I^S_{i+1,j}$ and $I^S_{i,j-1}$

2 cases:

(1) $C^S_{i,\sigma(j)} = C^S_{i,\sigma(j-1)} + P_{i,\sigma(j)}$:

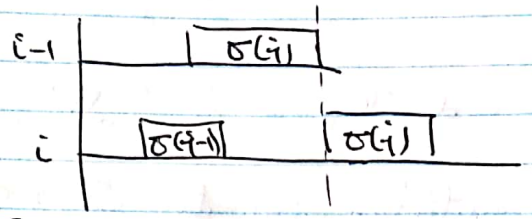
Then, $I^S_{i,j} = C^S_{i,\sigma(j)} - \sum_{k=1}^{j} P_{i,\sigma(k)}$

$= \left(C^S_{i,\sigma(j-1)} - \sum_{k=1}^{j-1} P_{i,\sigma(k)}\right) + P_{i,\sigma(j)} - P_{i,\sigma(j)}$

$= I^S_{i,j-1}$

Cont

(2) $C^s_{i,\sigma(j)} = C^s_{i-1,\sigma(j)} + \square + P_{i,\sigma(j)}$



So,

$$T^s_{i,j} = C^s_{i,\sigma(j)} - \sum_{k=1}^{j} P_{i,\sigma(k)}$$

$$= C^s_{i-1,\sigma(j)} + P_{i,\sigma(j)} - \sum_{k=1}^{j} P_{i,\sigma(k)}$$

$$= \left(\sum_{k=1}^{j} P_{i-1,\sigma(k)} + T^s_{i-1,j}\right) + P_{i,\sigma(j)} - \sum_{k=1}^{j} P_{i,\sigma(k)}$$

$$= T^s_{i-1,j} + P_{i,\sigma(j)} - \sum_{k=1}^{j} \left(P_{i-1,\sigma(k)} - P_{i,\sigma(k)}\right)$$

Suppose we find a permutation σ s.t.

$$(*) \longrightarrow \sum_{k=1}^{j} \left(P_{i-1,\sigma(k)} - P_{i,\sigma(k)}\right) \le (m-1) P_{max} \quad \forall j = 1, \ldots, n, \ \forall i = 2, \ldots, m.$$

Then,

- In case (1), $T^s_{i,j} = T^s_{i,j-1}$ is unchanged
- In case (2),

$$T^s_{i,j} = T^s_{i-1,j} + P_{i,\sigma(j)} + \sum_{k=1}^{j} \left(P_{i-1,\sigma(k)} - P_{i,\sigma(k)}\right)$$

$$\le T^s_{i,j} + P_{max} + (m-1) P_{max}$$

$$= T^s_{i-1,j} + m P_{max}$$

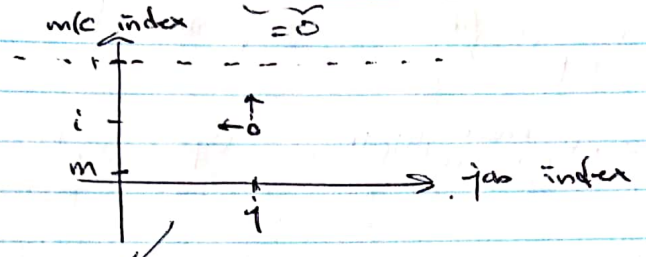
So, $T^s_{i,j} \le \max\left\{T^s_{i,j-1},\ T^s_{i-1,j} + m P_{max}\right\} \longrightarrow (1)$

$\forall i = 2, \ldots, m, \ \forall j = 1, \ldots, n \longrightarrow (2)$

and $T^s_{i,1} = 0 \quad \forall j = 1, \ldots, n$

From (1) and (2), we infer

$\tau^s_{i,j} \le \tau^s_{i,j} + (i-1)m\,p_{max}$. ← can prove by induction on i



From (1), we knew that $\tau^s_{i,j} \le \max(\tau^s_{i,j-1}, \tau^s_{i-1,j} + m p_{max})$

horizontal movement (no cost)

vertical movement towards dotted line. (Cost of mp_{max})

How At most how many vertical steps can we make? ←

So, $\tau^s_{m,n} \le (m-1)m\,p_{max}$.

And, $C^s_{max} \le \tau_m + (m-1)m\,p_{max}$.

⇒ Get an additive approximation of $(m-1)m\,p_{max}$.

Theorem 1: If we find a permutation σ satisfying (*), then the corresponding permutation schedule S has makespan $C^s_{max} \le OPT + (m-1)m\,p_{max}$.

Q: How do we find a permutation σ satisfying (*)?

Notation:

Given a vector $v \in \mathbb{R}^d$, define

$\|v\|_\infty = \max_{i=1,\dots,d} |v_i|$

ℓ_∞-norm of v

Lemma (Steinitz / Sevast'janov)
Given vectors $v_1, \ldots, v_n \in \mathbb{R}$ s.t. $\sum_{i=1}^{n} v_i = \vec{0}$,

we can obtain, in polytime, a permutation σ of $1, \ldots, n$ s.t.

$$\left\| \sum_{k=1}^{j} v_{\sigma(k)} \right\|_\infty \leq d \cdot \max_{k=1,\ldots,n} \|v_k\|_\infty \quad \forall j = 1, \ldots, n$$

[Proof]
In a bit, let's see how we can use this fact!

Recall:
We wanted to use:

$$(*) \quad \sum_{k=1}^{j} (p_{i-1,\sigma(k)} - p_{i,\sigma(k)}) \leq (m-1) p_{max} \quad \forall i = 2, \ldots, m \quad \forall j = 1, \ldots, n.$$

To use Steinitz / Sevast'janov (SS) lemma here:
We will start off by ensuring that

$$\Pi_i = \Pi_{max} \quad \text{for all machines} \quad i = 1, \ldots, m.$$

We can ensure this by repeatedly increasing p_{ij} arbitrarily up to p_{max} until $\Pi_i = \Pi_{max}$ $\forall i = 1, \ldots, m$

Ex:

Jobs

	1	2	3	4	5	Π_i
1	2	(1)	2	(4)	1	11 → (12) → (13)
2	5	2	1	3	2	13 → Π_{max}, $p_{max} = 5$
3	4	(1)	1	1	3	10 → (13)

↳ 4

A permutation schedule for modified p_{ij}'s give a permutation schedule for original p_{ij}'s with no greater makespan.

July 16th

Define $\forall j = 1, \dots, n$.

$v_j = (p_{1,j} - p_{2,j}, p_{2,j} - p_{3,j}, \dots, p_{m-1,j} - p_{m,j})$

Now, verify that $\sum_{j=1}^{n} v_j = \vec{0}$.

The ith component of the sum is:

$$\left(\sum_{j=1}^{n} v_j\right)_i = \sum_{j=1}^{n} (p_{i,j} - p_{i+1,j})$$

$\rightarrow$ since $\pi_i = \pi_{max}$ $\forall i$.

$$= \pi_i - \pi_{i+1} = 0.$$

So, $\sum v_j = \vec{0}$.

So, $v_j \in \mathbb{R}^{m-1}$, and $\|v_j\|_\infty \le p_{max}$. $\forall j = 1, \dots, n$

Then, SS lemma gives a permutation σ s.t.

$$\left\| \sum_{k=1}^{i} v_{\sigma(k)} \right\|_\infty \le (m-1) p_{max}.$$

$$\Rightarrow \left(\sum_{k=1}^{i} v_{\sigma(k)} \right)_{i-1} = \sum_{k=1}^{i} (p_{i+1,\sigma(k)} - p_{i,\sigma(k)}) \le (m-1) p_{max}.$$

$\forall i = 2, \dots, m$, $\forall j = 1, \dots, n$, which is exactly what we wanted.

July 24th, 2018

17 Lecture 24

454

Recall:
For $v \in \mathbb{R}^d$, $\|v\|_\infty := \max_{i=1,\dots,d} |v_i|$

Steinitz (SS) Lemma:
Given $v_1, \dots, v_n \in \mathbb{R}^d$ s.t. $\sum_{i=1}^n v_i = \vec{0}$, we can efficiently find a permutation σ of $1, \dots, n$ s.t.

$$\left\| \sum_{k=1}^{j} v_{\sigma(k)} \right\|_\infty \le d \cdot \max_{k=1,\dots,n} \|v_k\|_\infty \quad \forall j = 1, \dots, n.$$

Algorithm for ℓ1 Case:
1) Increase P_{ij}'s up to P_{max} (arbitrarily) to ensure that $\pi_i = \pi_{max}$ $\forall i = 1, \dots, m$
2) Apply SS lemma to vectors:
$v_j := (P_{1,j+1} - P_{1,j}, \dots, P_{m,j+1} - P_{m,j})$
$\forall j = 1, \dots, n$,
to get permutation σ.
3) Return permutation schedule corresponding to σ.

What's left: Prove lemma!

Proof (Lemma)
We will construct σ by specifying all its prefixes starting in reverse order,
i.e. We will construct
$V_n := \{v_1, \dots, v_n\} \supsetneq V_{n-1} \supsetneq \dots \supsetneq V_1$.
where $|V_j| = j$ $\forall j = 1, \dots n$ so that $\forall j = 1, \dots, n$ we have:

$$\left\| \sum_{v \in V_j} v \right\|_\infty \le d \cdot M \qquad (*)$$

(let $M = \max_{k=1,\dots,n} \|v_k\|_\infty$)

[Proof] (cont.)

- V_n satisfies (*) (since $\|\sum_{v \in V_n} v\|_\infty = \vec{0}$)

(Note: Triangle inequality: $\|x+y\|_\infty \le \|x\|_\infty + \|y\|_\infty$).

So, if we have constructed V_{d+1} with $|V_{d+1}| = d+1$, then we can:

- Take V_d to be any subset of V_{d+1} with d vectors
- Take V_{d-1} to be any subset of V_d with $d-1$ vectors
- ... and so on

This works since for any set of $j \le d$ vectors V_j, we have:

$$\|\sum_{v \in V_j} v\|_\infty \le j \cdot \max_{v \in V_j} \|v\|_\infty \le d \cdot M.$$

↑ By triangle inequality.

So, our goal is: Construct $V_n \supsetneq V_{n-1} \supsetneq \dots \supsetneq V_{d+1}$ (since once we have constructed V_{d+1} the rest follows from above).

with $|V_j| = j$ $\forall j = d+1, \dots, n$ and (*) holds $\forall j = d+1, \dots, n$.

Idea: Use induction, given V_j satisfying (*), show how to get $V_{j-1} \subsetneq V_j$ satisfying (*). We will strengthen (*) as follows:

Sufficient condition implying (*):

Suppose $j > d$. Suppose we have a (set $S \subseteq V_j$ with $|S| = j-d$. s.t. $\sum_{j \in S} V_j = 0$.

Then, we claim: V_j satisfies (*)

Proof of claim:

$$\left\| \sum_{v \in V_j} v \right\|_\infty = \left\| \sum_{v \in S} v + \sum_{v \in V_j \setminus S} v \right\|_\infty$$

$$= \left\| \sum_{v \in V_j \setminus S} v \right\|_\infty \quad \text{and } |V_j \setminus S| = d$$

$$\leq |V_j \setminus S| \max_{v \in V_j \setminus S} \|v\|_\infty$$

$$\leq d \cdot M$$

We write boxed portion as an IP

(IP_j) variables $\lambda_v^j \quad \forall v \in V_j \equiv \begin{cases} 1 & \text{if } v \in S \\ 0 & \text{o/w} \end{cases}$

$$(IP_j) \quad \sum_{v \in V_j} \lambda_v^j = j - d, \quad \sum_{v \in V_j} \lambda_v^j v = \vec{0} \qquad \lambda_v^j \in \{0,1\} \ \forall v \in V_j$$

(IP_j) is feasible $\Rightarrow V_j$ satisfy (*)

Modification strategy: If $j > d+1$, and we have V_j s.t. (IP_j) is feasible, then we can get $V_{j-1} \subseteq V_j$ s.t. (IP_{j-1}) is feasible

In fact this is too strong an antichain. We will show that the LP-relaxation of (IP_j), which we will denote (LP_j):

$$(LP_j) \quad \sum_{v \in V_j} \lambda_v^j = j - d, \quad \sum_{v \in V_j} \lambda_v^j v = \vec{0}, \quad 0 \leq \lambda_v^j \leq 1, \ \forall v \in V_j$$

is feasible still implies V_j satisfies (*)

Claim: (LP_j) is feasible $\Rightarrow V_j$ satisfies $(*)$.

Proof:

Let $\{\lambda_v^j\}_{v \in V_j}$ be a feasible solution to (LP_j)

$$\left\| \sum_{v \in V_j} v \right\|_\infty = \left\| \sum_{v \in V_j} \lambda_v^j v + \sum_{v \in V_j} (1 - \lambda_v^j) v \right\|_\infty$$

$$= \left\| \sum_{v \in V_j} (1 - \lambda_v^j) v \right\|_\infty \quad \left(\text{since } \sum_{v \in V_j} \lambda_v^j v = 0.\right)$$

$$\leq \sum_{v \in V_j} \|(1 - \lambda_v^j) v\|_\infty \quad \text{(Triangle Inequality)}$$

$$= \sum_{v \in V_j} (1 - \lambda_v^j) \|v\|_\infty$$

$$\leq \sum_{v \in V_j} (1 - \lambda_v^j) M.$$

$$= \left(j - \sum_{v \in V_j} \lambda_v^j\right) M = d \cdot M.$$

Lemma 2: We now want to show:

Suppose $j > d+1$. Suppose we have $V_j \subseteq V_n$ s.t. (LP_j) is feasible. Then, we can efficiently find $V_{j-1} \subsetneq V_j$ with $|V_{j-1}| = j-1$ s.t. (LP_{j-1}) is feasible

Notice that Claim 1 + Lemma 2 will finish the proof. Since:

- (LP_n) is feasible, we set $\lambda_v^n = \frac{n-d}{n}$ $\forall v \in V_n$, this gives a feasible sol'n to (LP_n).
- By repeatedly using Lemma 2, we get sets $V_n \supsetneq V_{n-1} \supsetneq \dots \supsetneq V_{d+1}$ s.t. (LP_j) feasible $\forall j = d+1, \dots, n$.

$\Rightarrow V_j$ satisfies (6) $\forall j = d+1, \dots, n$.

Then, from def, we can arbitrarily choose the remaining sets $V_0, V_{d+1}, \dots, V_1$.

Now, for the proof of Lemma 2:

Recall: (from LP theory - COSC 310 / 210)

Let $P = \{x \in \mathbb{R}^n : Ax \le b\}$

Def$_n$: $\bar{x} \in P$ is an extreme point of P if: $\nexists\, x^{(1)}, x^{(2)} \in P$ distinct, $x^{(1)} \neq x^{(2)}$, $0 < \lambda < 1$ s.t.

$$\bar{x} = \lambda x^{(1)} + (1-\lambda) x^{(2)}.$$

Further:

$\bar{x} \in P$ is an extreme point of P iff

$$\text{rank}\left([A_{ij}]_{\substack{j=1,\dots,n \\ i : (A\bar{x})_i = b_i}} \right) = n.$$

(first attempt)

Finally,

If $\text{rank}(A) = n$ and $P \neq \emptyset$, then P has an extreme point. (and this can be found efficiently).

Consider the system: (P) ← Similar to (P_{j-1}), but we have variables $\forall v \in V_j$.

$$\sum_{v \in V_j} x_v = j - 1 - d \quad (1), \qquad \sum_{v \in V_j} x_v v = \vec{0} \quad (2)$$

$0 \le x_v \le 1$, $\forall v \in V_j$.

(3a) (3b)

↓ Corresponds to 2 inequalities $\le$ and $\ge$.

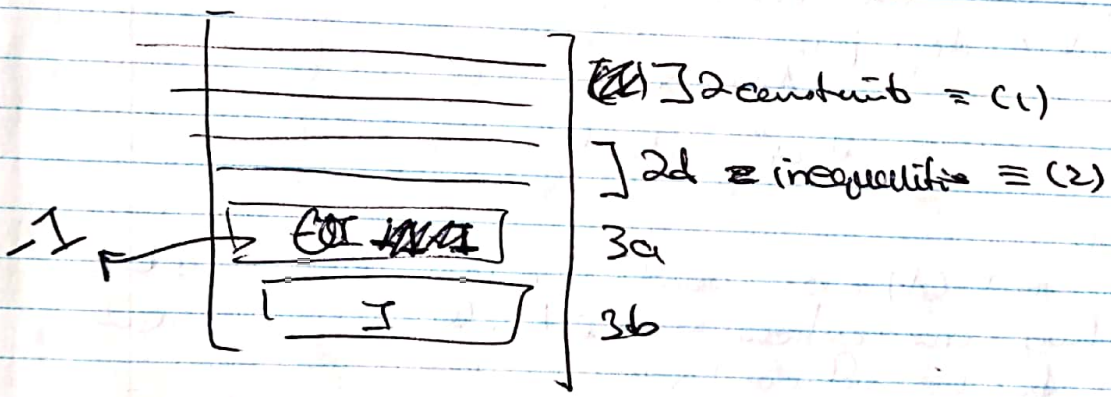
↘ Corresponds to $2d$ $\le$-inequalities.

Will show: (LP_j) feasible $\Rightarrow$ Every extreme point feasible solution to (P) has $x_v = 0$ for some $v \in U_j$ $\Rightarrow$ Taking U_{j-1} to be $U_j \setminus \{v \text{ such that } x_v = 0\}$ then $\lambda^{j-1}_v = x_v$ $\forall v \in U_{j-1}$ is a feasible solution to (LP_{j-1}).

(LP_j) is feasible: Say $\{\lambda^j_v\}_{v \in U_j}$ is feasible to (LP_j)

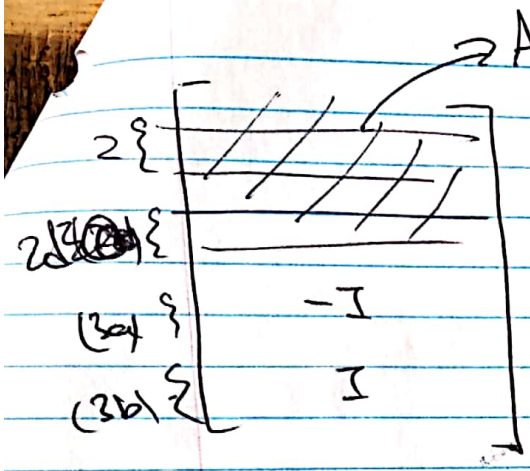
$\Rightarrow x_v = \frac{j-1-d}{j-d} \lambda^j_v$ $\forall v \in U_j$: feasible soln to (P)

Constraints of (P) written as $\leq$ inequalities.



So, the inequalities in matrix form $\leq$ have rank $= j$.

July 24, 07



Suppose $\bar{x}$ is extreme point of feasible region of (P)

We want to show that at least of of the (3c) constraints is tight for $\bar{x}$ (and so one of the values are 0).

Constraint matrix.

Suppose not.

Let $A = [A_{ij}]$ $j = 1, \ldots, n$

$i = a_i \bar{x}_i = b_i$ (tight constraints)

We know $\text{rank}(A') = j$.

If B is a $j \times j$ submatrix of A'.

with $\text{rank}(B) = j$, then B contains $\leq d+1$ rows from A_1 $\Rightarrow$ B must contain $\geq j - d - 1$ rows from (3b) blocks of $A \Rightarrow \geq j - d - 1$ x_i variables equal to I., and all other x_i variables are strictly positive, but then $\bar{x}$ violates constraint (1).

$\square$