Problem analysis of Greedy Algorithm(2)

YAO ZHAO

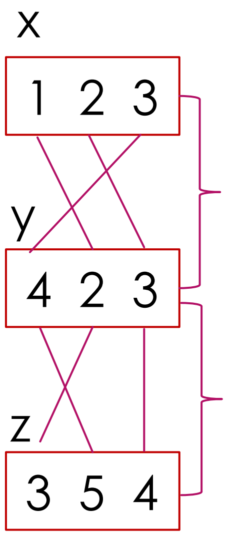
Nesting boxes

- A $\boldsymbol{d}$**-dimensional** box with dimensions $(x_1, x_2, \dots, x_d)$ ***nests*** within another box with dimensions $(y_1, y_2, \dots, y_d)$ if there exists a permutation π on $\{1, 2, \dots, d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d$.
- (1) Argue that the nesting relation is transitive.
- (2) Describe an efficient method to determine whether one $\boldsymbol{d}$**-dimensional** box nests inside another.
- (3) Suppose that you are given a set of $\boldsymbol{n}$ $\boldsymbol{d}$**-dimensional** boxes $\{B_1, B_2, \dots B_n\}$. Give an efficient algorithm to find the longest sequence $\langle B_{i1}, B_{i2}, \dots B_{ik} \rangle$ of boxes such that B_{ij} nests within B_{ij+1} for $j = 1,2, \dots k-1$. Express the running time of your algorithm in terms of n and d.

Question 1: Argue that the nesting relation is transitive

- Suppose that box $x = (x_1, x_2, \dots, x_d)$ nests with box $y = (y_1, y_2, \dots, y_d)$ and box y nests with box $z = (z_1, z_2, \dots, z_d)$ Then there exist permutations π and σ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d$ and $y_{\sigma(1)} < z_1, y_{\sigma(2)} < z_2, \dots, y_{\sigma(d)} < z_d$. This implies $x_{\pi(\sigma(1))} < z_1, x_{\pi(\sigma(2))} < z_2, \dots, x_{\pi(\sigma(d))} < z_d$, so x nests with z and the nesting relation is transitive.

Sample:



exist permutations π : $\pi(1)$=3 $\pi(2)$=1 $\pi(3)$=2

$x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, x_{\pi(3)} < y_3$

$x_3 < y_1, x_1 < y_2, x_2 < y_3$

exist permutations σ : σ (1)=2 σ (2)=1 σ (3)=3

$y_{\sigma(1)} < z_1, y_{\sigma(2)} < z_2, y_{\sigma(3)} < z_3$

$y_2 < z_1, y_1 < z_2, y_3 < z_3$

exist permutations $\pi\sigma$:

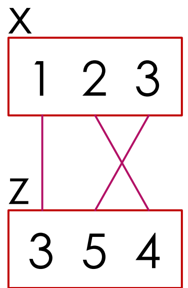
$\pi(\sigma(1))$= $\pi(2)$=1

$\pi(\sigma(2))$= $\pi(1)$=3

$\pi(\sigma(3))$= $\pi(3)$=2

$x_{\pi(\sigma(1))} < z_1, x_{\pi(\sigma(2))} < z_2, x_{\pi(\sigma(3))} < z_3$

$x_1 < z_1, x_3 < z_2, x_2 < z_3$



Question 2: Describe an efficient method to determine whether one d-dimensional box nests inside another

- Box x nests inside box y if and only if the increasing sequence of dimensions of x is component-wise strictly less than the increasing sequence of dimensions of y. Thus, it will suffice to sort both sequences of dimensions and compare them. Sorting both length d sequences is done in $O(d \log d)$, and comparing their elements is done in $O(d)$, so the total time is $O(d \log d)$.

Question 2:proof

- **Pf.** Box x nests inside box y if and only if the increasing sequence of dimensions of x is component-wise strictly less than the increasing sequence of dimensions of y.

the increasing sequence of dimensions of x is component-wise strictly less than the increasing sequence of dimensions of y.

By the definition of Nesting

Box x nests inside box y

Box x nests inside box y

?

the increasing sequence of dimensions of x **must be** component-wise strictly less than the increasing sequence of dimensions of y.

- **Claim.** the increasing sequence of dimensions of x must be component-wise strictly less than the increasing sequence of dimensions of y if Box x nests inside box y .

- **Pf. (by induction)**
  - Base：let dimension = 1, if Box x nests inside box y , $x_1 < y_1$
  - Induction：Suppose that dimension = d, if Box x nests inside box y, there is an increasing permutations π for x and an increasing permutations σ for y, satisfy $x_{\pi(1)} < y_{\sigma(1)}, x_{\pi(2)} < y_{\sigma(2)}, \dots, x_{\pi(d)} < y_{\sigma(d)}$
  - When dimension =d+1, if Box x nests inside box y, according the definition, there exist permutations λ :

- $x_{\lambda(1)} < y_1, x_{\lambda(2)} < y_2, \dots, x_{\lambda(d)} < y_d, x_{\lambda(d+1)} < y_{d+1}$ -------(1)

Observe(1), the first d terms, $x_{\lambda(1)} < y_1, x_{\lambda(2)} < y_2, \dots, x_{\lambda(d)} < y_d$, The first d terms satisfy the nested relationship of d dimensions, so for the first d terms, there will exist permutations π, σ

- $x_{\pi(1)} \le x_{\pi(2)} \le \dots \le x_{\pi(\mathrm{d})}$ --------(2)
- $y_{\sigma(1)} \le y_{\sigma(2)} \le \dots \le y_{\sigma(d)}$ --------(3)
- $x_{\pi(1)} < y_{\sigma(1)}, x_{\pi(2)} < y_{\sigma(2)}, \dots, x_{\pi(d)} < y_{\sigma(d)}$ --------(4)

- **Pf. (continue)**

$x_{\lambda(d+1)}$ is inserted into Formula (2) (let the insertion position be j), thereby having a new permutation π' such that the x sequence remains increasing.

Then insert y_{d+1} into Formula (3) (let the insertion position be k), a new permutation σ' is created so that y sequence remains increasing.

$x_{\lambda(d+1)} = x_{\pi'(j)} < y_{d+1} = y_{\sigma'(k)}$ (5)

There are 3 cases:

- $j < k$: $x_{\pi'(1)} < y_{\sigma'(1)}, x_{\pi'(2)} < y_{\sigma'(2)}, \ldots x_{\pi'(j)} < x_{\pi'(j+1)} < y_{\sigma'(j)}, x_{\pi'(j+1)} < x_{\pi'(j+2)} < y_{\sigma'(j+1)},$
  $\ldots x_{\pi'(k-1)} < x_{\pi'(k)} < y_{\sigma'(k)}, \quad x_{\pi'(k)} < y_{\pi'(k-1)} < y_{\sigma'(k)}, x_{\pi'(k+1)} < x_{\pi'(k+1)}, x_{\pi'(d+1)} < y_{\sigma'(d+1)}$

  So, for any $1 \le i \le d+1$, $x_{\pi'(i)} < y_{\sigma'(i)}$, the claim is true.

- $j = k$:

the insertion position: we have $x_{\lambda(d+1)} < y_{d+1}$ (see formula(1)), so $x_{\lambda(d+1)} = x_{\pi'(j)} < y_{d+1} = y_{\sigma'(k)}$(Formula(5)) = $y_{\sigma'(j)}$ (j=k)

The other positions hasn't changed. for any $1 \le i \le d+1$, $x_{\pi'(i)} < y_{\sigma'(i)}$, the claim is true.

- $j > k$: $x_{\pi'(1)} < y_{\sigma'(1)}, x_{\pi'(2)} < y_{\sigma'(2)}, \ldots x_{\pi'(k)} < x_{\pi'(j)} < y_{\sigma'(k)}$ (see formula(5)), $x_{\pi'(k+1)} < x_{\pi'(j)} < y_{\sigma'(k)} < y_{\sigma'(k+1)}, \ldots. x_{\pi'(j)} < y_{\sigma'(k)} < y_{\sigma'(j)}, x_{\pi'(j+1)} < x_{\pi'(j+1)}, x_{\pi'(d+1)} < y_{\sigma'(d+1)}$

So, for any $1 \le i \le d+1$, $x_{\pi'(i)} < y_{\sigma'(i)}$, the claim is true.

Sample:

	1	2	3	4
x	21	23	25	27
y	31	33	35	37

22
36

J=2 < k=4

	1	2	3	4	5
x	21	22	23	25	27
y	31	33	35	36	37

	1	2	3	4
x	21	23	25	27
y	31	33	35	37

22
32

j=k=2

	1	2	3	4	5
x	21	22	23	25	27
y	31	32	33	35	37

	1	2	3	4
x	21	23	25	27
y	31	33	35	37

26
32

J=4 > k=2

	1	2	3	4	5
x	21	23	25	26	27
y	31	32	33	35	37

Question 3: Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \dots B_n\}$. Give an efficient algorithm to find the longest sequence $\langle B_{i1}, B_{i2}, \dots B_{ik} \rangle$ of boxes such that B_{ij} nests within B_{ij+1} for $j = 1, 2, \dots k-1$. Express the running time of your algorithm in terms of n and d.

- We will create a nesting-graph G with vertices $B_1, B_2, \dots B_n$ as follows. For each pair of boxes B_i, B_j, we decide if one nests inside the other. If B_i nests in B_j, draw an arrow from B_i to B_j. If B_j nests in B_i, draw an arrow from B_j to B_i. If neither nests, draw no arrow. To determine the arrows efficiently, after sorting each list of dimensions in $O(nd \log d)$ we compare all pairs of boxes using the algorithm from part (2) in $O(n^2 d)$. By part (1), the resulted graph is acyclic, which allows us to easily find the longest chain in it in $O(n^2)$ in a bottom-up manner. This chain is our answer. Thus, the total time is $O(nd * max(\log d, n))$.

Sample:

8 boxes

5	2	20	1	30	10
23	15	7	9	11	3
40	50	34	24	14	4
9	10	11	12	13	14
31	4	18	8	27	17
44	32	13	19	41	19
1	2	3	4	5	6
80	37	47	18	21	9

sorting each list: →

Node 1

1 2 5 10 20 30

Node 2

3 7 9 11 15 23

Node 3

4 14 24 34 40 50

Node 4

9 10 11 12 13 14

Node 5

4 8 17 18 27 31

Node 6

13 19 19 32 41 44

Node 7

1 2 3 4 5 6

Node 8

9 18 21 37 47 80

Create a nesting-graph

Node 1

1 2 5 10 20 30

Node 2

3 7 9 11 15 23

Node 3

4 14 24 34 40 50

Node 4

9 10 11 12 13 14

Node 5

4 8 17 18 27 31

Node 6

13 19 19 32 41 44

Node 7

1 2 3 4 5 6

Node 8

9 18 21 37 47 80

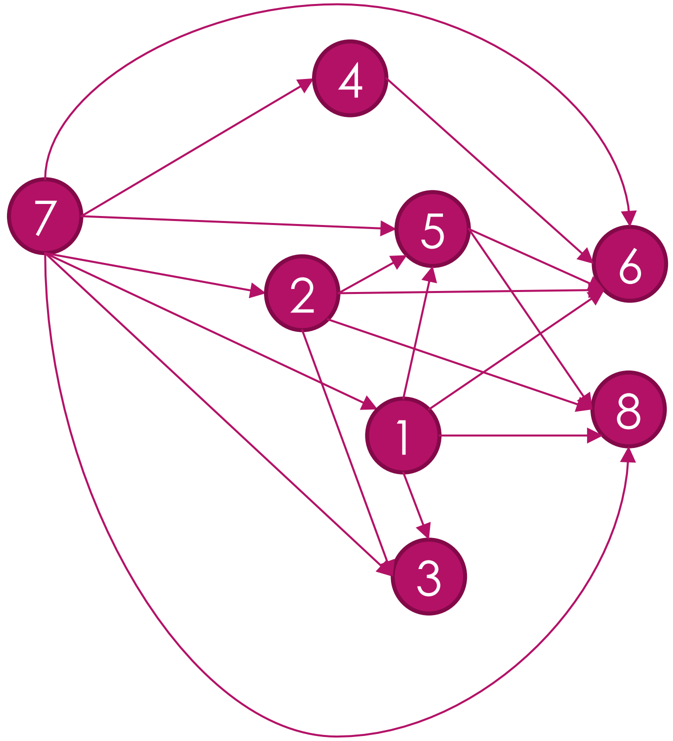
compare all pairs of boxes using the algorithm from part (2)

	1	2	3	4	5	6	7	8
1	0	0	1	0	1	1	0	1
2	0	0	1	0	1	1	0	1
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	0	0
7	1	1	1	1	1	1	0	1
8	0	0	0	0	0	0	0	0

We will create a nesting-graph G with vertices $\boldsymbol{B_1}, \boldsymbol{B_2}, \ldots \boldsymbol{B_n}$ as follows. For each pair of boxes $\boldsymbol{B_i}, \boldsymbol{B_j}$, we decide if one nests inside the other. If $\boldsymbol{B_i}$ nests in $\boldsymbol{B_j}$, draw an arrow from $\boldsymbol{B_i}$ to $\boldsymbol{B_j}$. If $\boldsymbol{B_j}$ nests in $\boldsymbol{B_i}$, draw an arrow from $\boldsymbol{B_j}$ to $\boldsymbol{B_i}$. If neither nests, draw no arrow.

Find the longest chain in the DAG in $O(n^2)$ in a bottom-up manner

	1	2	3	4	5	6	7	8
1	0	0	1	0	1	1	0	1
2	0	0	1	0	1	1	0	1
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	0	0
7	1	1	1	1	1	1	0	1
8	0	0	0	0	0	0	0	0



the longest chain :4

The following are the possible chain:
7 → 1 → 5 → 8
or
7 → 1 → 5 → 6
or
7 → 2 → 5 → 6
or
7 → 2 → 5→ 8

A Task-scheduling Problem

- Given n unit-time tasks **with start times, end times and profits for a single processor**, we obtain a profit of v_i if the i^{th} task is finished between s_i and t_i, and we obtain no profit if a task misses its end time or starts before its start time.
- Return the maximum profit you can take while ensuring there are no two tasks scheduled in the same timeslot.
- n≤5000, $1 \le s_i \le t_i \le 10^8$, $1 \le v_i \le 10^8$

Sample Input	Sample Output	Explanation
4 1 1 2 2 2 2 1 2 3 1 3 1	6	Time slot: 1 2 3 1 3 4 Profit: 2 3 1

End time
weight

Start time
End time
weight

Difference?

Judgment: Greedy algorithm can be used to solve this problem?

Claim 1 and proof

Claim 1: Sort tasks by profit so that $v_1 \geq v_2 \geq \ldots \geq v_n$, consider each task in turn. Let the previously selected tasks set be S. If the current task a_i can be added to S to ensure that every task matches to a timeslot successfully, then select a_i , otherwise do not select a_i. This greedy strategy is bound to result in the maximum profit.

Proof:

1. Let the i^{th} task be a_i and the optimal task set of the previous i-1 tasks be S.
2. If a_i can be added to S to ensure that every task in S matches a timeslot, then the i^{th} task must be selected, otherwise the total profit will not be optimal.
3. If after joining a_i , some tasks in S incur conflicts, that means at least one task in S can't be selected. As $v_1 \geq v_2 \geq \ldots \geq v_i$, a_i has the least profit, and the original tasks in S has no conflict, so do not select a_i.
4. Every selection can guarantee that the obtained set S is optimal, so the result is optimal.

which timeslot would you place the elected tasks in?

$$1 \leq s_i \leq t_i \leq 10^8$$

How to choose the timeslot since the time range is too large?

Algorithm 1: Brute force

- Since each task can only one timeslot in the specified interval, the tasks are sorted according to their profits. For each task a_i , which is judged whether it conflicts with the previously selected tasks. If no conflict, enumeration its no-conflict timeslots. Let **Max {Ti}-min{Si}+1=L**, there are at most **L** choices for each task, and at least **1** choice. After enumerating all possible options, get the maximum profit.
- $O(L^{n-1})$
- L Possible range: $\leq 10^8$,
- **low efficiency**

A huge number of timeslots in the interval are redundant

How to identify and remove redundant timeslots?

Active Timeslots

Initially, all time slots are marked black. Then, for each task $a_i\ (s_i, t_i, v_i)$, find the smallest k, where $k \geq s_i$ and the time slot k is black, and then mark the time slot k as white. Finally, we get exactly n time slots marked white, and these n time slots are useful time slots. Call them "active timeslots" .

Algorithm of Finding all Active Timeslots of all tasks

```
S ← ø
Sort the tasks by start time so that s_1 ≤ s_2 ≤ s_3 ... ≤ s_n
x ← 0
For i=1 to n {
    x ← max(x+1, s_i);
    add x to S
}
```

Claim 2 and proof

Claim 2. Suppose an active timeslots set T and a task set S, if each task in S can schedule no conflict on T, there must be such a greedy strategy: scan each timeslot in T from front to rear, for each timeslot, if there are some tasks whose s_i ≤ the timeslot ≤ t_i and haven't be scheduled, select the task with the smallest t_i.

Proof: Suppose that there is no conflict match M between S and T. Now scan the first active timeslot ts_1 in T:

(1) If ts_1 is idle and if there are some tasks whose $s_i \leq ts_1 \leq t_i$, we can make the task with the smallest t matches ts_1 . The operation will not affect the other match relations.

(2) If ts_1 is idle and no tasks whose $s_i \leq ts_1 \leq t_i$, make no change.

(3) If ts_1 is occupied by a task a_x but a_y with the smallest t for ts_1 , swap the task a_x and a_y , assume the timeslot of a_y is ts_y,

We have:

$s_y \leq ts_y \leq t_y \quad s_x \leq ts_1 \leq t_x \quad t_y < t_x$

So, $s_x \leq$ ts1 $< ts_y \leq t_y < t_x$, the task a_x can be arrange to timeslot ts_y.

Repeat the swap operations on the remaining active time slots, the greedy strategy can generate a no-conflict match between$S$ and T.

Claim 3 and proof

Claim 3. The maximum profit of scheduling the tasks only on the "active timeslots" is the same as that scheduling the tasks on all timeslots.

Proof: by induction

- Base： let $n = 1$, for the task (s_1, t_1, v_1)， according the algorithm of p.9, the active timeslot is s_1, can finish the task at s_1, obtain the maximum profit v_1 .
- Induction： Suppose that $n = m$, for the first m tasks, according the algorithm of p.9, get m "active timeslots"(let the m "active timeslots" set T), let tasks set S be the optimal task set on T, and the S also the optimal task set on all timeslots.

When $n = m + 1$, now consider the task a_{m+1} , there are 3 cases:

- The optimal task set of the first $m + 1$ tasks doesn't include the task a_{m+1}
- Need to delete a task in S to free its timeslot for the task a_{m+1}
- Add the task a_{m+1} to the optimal task set S without deleting the previously selected tasks

Claim 3 proof: case 1

- Case 1: The optimal task set of the first $m+1$ tasks doesn‘t include the task a_{m+1}

Since there are $m+1$ tasks, only one newly added active timeslot, no previous active timeslots will be affected. The optimal solution of the first $m+1$ tasks is equal to the optimal solution of the first m tasks. So, the optimal solution of the first $m+1$ tasks can only be related to the first $m+1$ active timeslots.

Claim 3 proof: case 2

- Need to delete a task in S to free its timeslot for the task a_{m+1}

the task a_{m+1} oy replaces the task in the **previous active timeslot**, **the newly added active timeslot does not affect the previous active timeslot**. Therefore, The selected active timeslots of the first $m+1$ tasks is equal to that of the first m tasks. So, the optimal solution of the first $m+1$ tasks is only related to the first $m+1$ active timeslots.

Claim 3 proof: case 3

- Add the task a_{m+1} to the optimal task set S without deleting the previously selected tasks

1. We known that tasks set S is the optimal task set on the m active timeslots set T . According to the greedy strategy of claim 2, the only matching relation R of the tasks in S and the first m active points can be obtained.
2. Under the current matching relation R, add the $m+1$ **active timeslot**.
3. Try to add the task a_{m+1}, according to the greedy strategy of claim 2, try to find a timeslot for the task a_{m+1}.

Enumerate each timeslot ts_i from the s_{m+1} to ∞{

If ts_i is idle, a_{m+1} can match the ts_i, return ts_i

Else if ts_i←→ a_k{

if $T_k > T_{m+1}$ swap a_{m+1} and a_k

}

}

From above finding algorithm, the returned ts must be the s_{m+1} or the first idle timeslot $s_{m+1}+j$(j depends on the number of conflicts).

The returned ts must be either an active timeslot in T, or the $m+1$ **active timeslot**, So, the optimal solution of the first $m+1$ tasks is only related to the first $m+1$ active timeslots.

Algorithm 2: Complete Bipartite Graph + KM

- According **Claim 3**, create a N*N complete bipartite G:
  - add vertices $x_1, x_2, \dots x_n$ for each task;
  - add vertices $y_1, y_2, \dots y_n$ for each active timeslot;
  - $i, j \in [1, n]$, add edge (x_i, y_j) , if $\mathrm{s_i} \le y_j \le t_i$, the weight of edge (x_i, y_j) is v_i, otherwise, the weight is 0.
- In a complete bipartite graph G, using Hungarian algorithm(also called the Kuhn-Munkres algorithm) to find the maximum-weight matching.
- $O(n^3)$

See:https://brilliant.org/wiki/hungarian-matching/

Algorithm 3: Bipartite Graph + Greedy+ Augmenting path

- According **Claim 3**, create a bipartite G:
  - add vertices $x_1, x_2, \dots x_n$ for each task;
  - add vertices $y_1, y_2, \dots y_n$ for each active timeslot;
  - $i, j \in [1, n]$, if $s_i \le y_j \le t_i$, add a edge (x_i, y_j), the weight of edge (x_i, y_j) is v_i.
- Sort tasks by v_i so that $v_1 \ge v_2 \ge v_3 \dots \ge v_n$

$S \leftarrow \emptyset$

for each task a_i {

 if (AugmentingPath(x_i, S,G) = true){

  add x_i to S

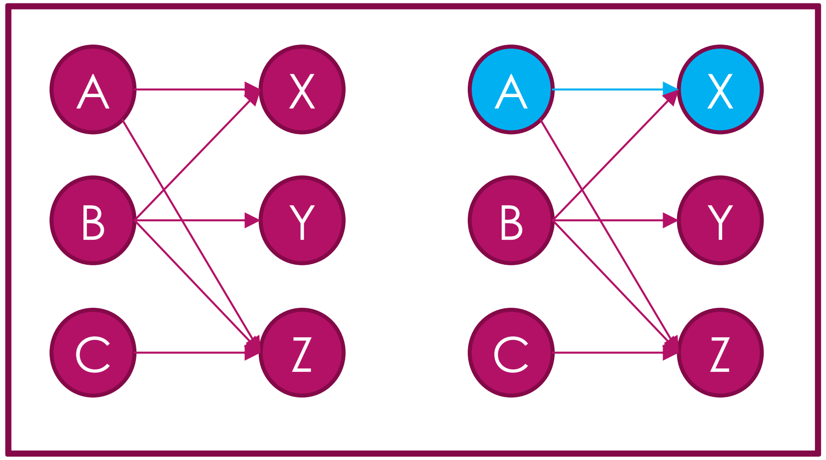
 }//else don't select the task a_i ,see Claim1

}

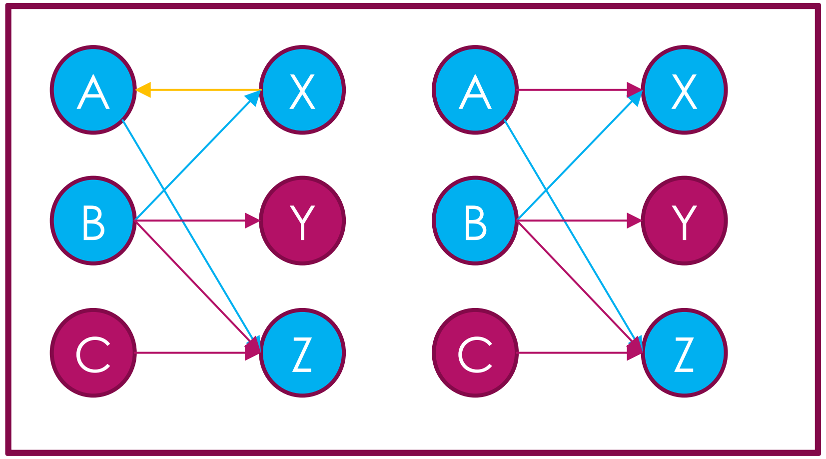
- $O(n^3)$

Matching Algorithms for Bipartite Graphs.pdf

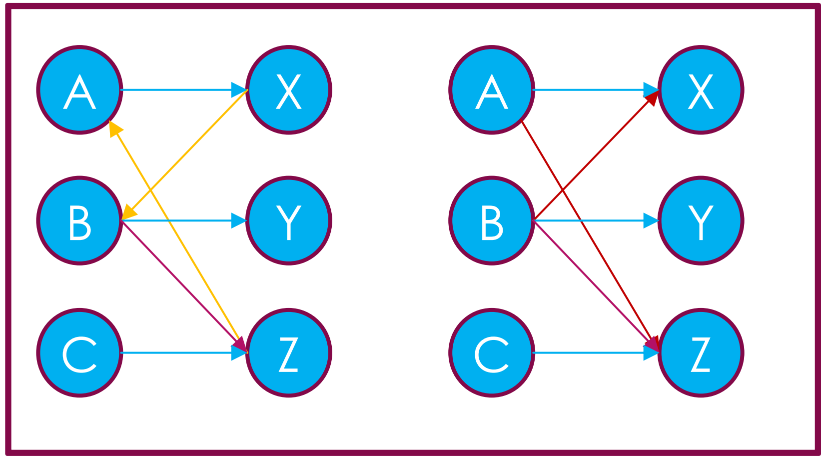
Finding Augmenting path Demo



First：DFS from A, find X
Result: A→X



Second：reverse the edge (A, X) ，DFS from B, find a augmenting path: X →A→Z
Result: B →X A→Z



Third：reverse the edge (A,Z) and(B,X),DFS from C, find a augmenting path: C→Z→A→X→B→Y
Result: A →X B→Y C→Z

Algorithm 4: Greedy + Heap

- According **Claim 1 and Claim 2**, consider the following algorithm:
- Sort tasks by v_i so that $v_1 \geq v_2 \geq v_3 \ldots \geq v_n$

```
S ← ∅
T ← n active timeslots
for each task a_i {
    add a_i to S
    flag = CheckByClaim2(S, T)
    if (flag == false){
        remove a_i from S
    }
}
```

- When realizing CheckByClaim2(S, T), can using Heap to select the smallest t_i task of all tasks satisfy $s_i \leq ts_j \leq t_i$ ($ts_j \in T$)

Algorithm 5: Greedy + Linear Match

- According **Claim 1 and Claim 2**, consider the following algorithm:
- Sort tasks by v_i so that $v1 \geq v2 \geq v3 \dots \geq vn$

S←∅

T← n active timeslots and $ts_1 \leq ts_2 \leq ts_3 \dots \leq ts_n$

for each task a_i {

    if (LinearMatch(a_i,s_i) == true){

        add a_i to S

    }

}

Algorithm 5: How to Linear Match

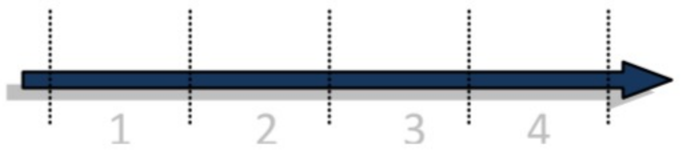
```
LinearMatch(a_i, x){
    if (x>t_i) return false
    if x is idle{
        x matches a_i
        return true
    }
    a_j = getTaskOf(x)
    if (Ti > Tj){
        return LinearMatch(a_i, next active timeslot of x)
    }else{
        if LinearMatch(a_j, next active timeslot of x){
            x matches a_i
            return true
        }
    }
    return false

}
```

Algorithm 5 Demo

Sample Input	Sample Output
4 1 1 3 1 2 2 1 3 1 1 2 4	8

Sort tasks by v_i
1 2 4
1 1 3
1 2 2
1 3 1

active timeslots:
1
2
3
4



Algorithm 5 Sample: Task 1

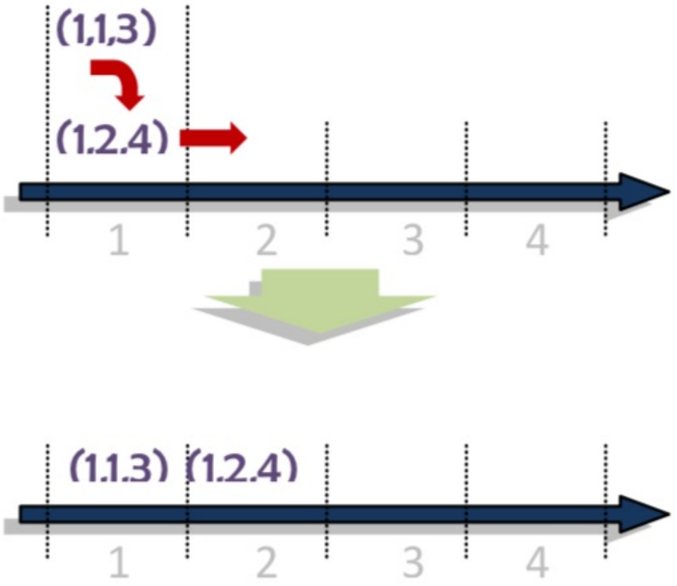
Task 1, matches active timeslot 1.

Result:
Task1(1,2,4)←→ active timeslot 1

Algorithm 5 Sample: Task 2

Task2(1,1,3) , try active timeslot 1, but 1 matches task1, conflict.
Observe $t_2 < t_1$
Then try to place Task1 on timeslot2, $t_2 \leq 2$ and timeslot2 is idle, successful.

Result:
Task1(1,2,4)←→ active timeslot 2
Task1(1,1,3)←→ active timeslot 1



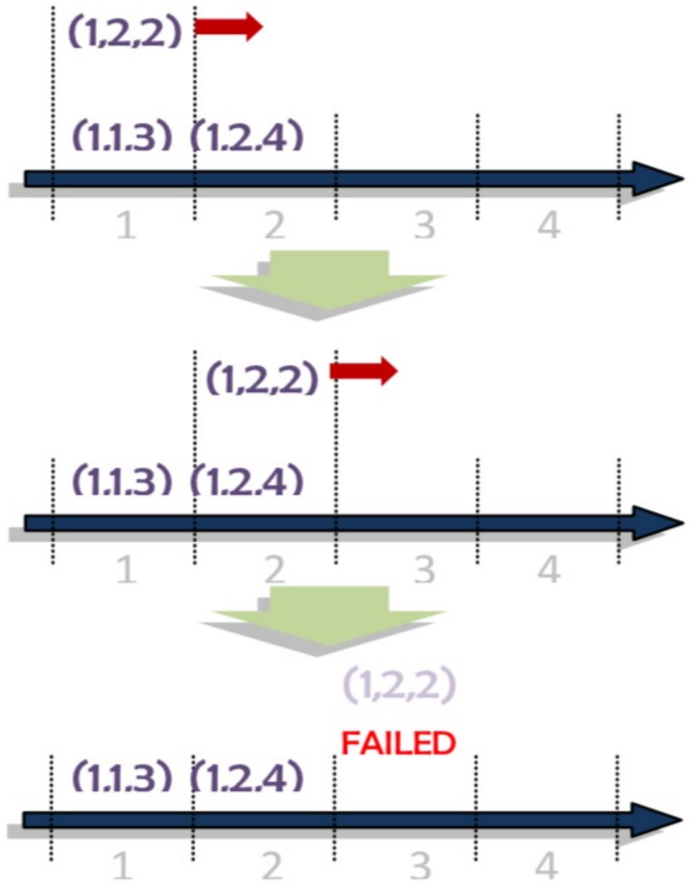
Algorithm 5 Sample: Task 3

Task3(1,2,2) , try active timeslot 1, but 1 matched task2, conflict.
Observe $t_3 > t_1$
Then try to place Task3 to next active timeslot

Try active timeslot 2, but 2 matched task1, conflict again.
Observe $t_3 \geq t_2$

Try active timeslot 3, observe $t_3 < 3$, failed.
Don't select task 3

Result:
Task1(1,2,4)←→ active timeslot 2
Task1(1,1,3)←→ active timeslot 1



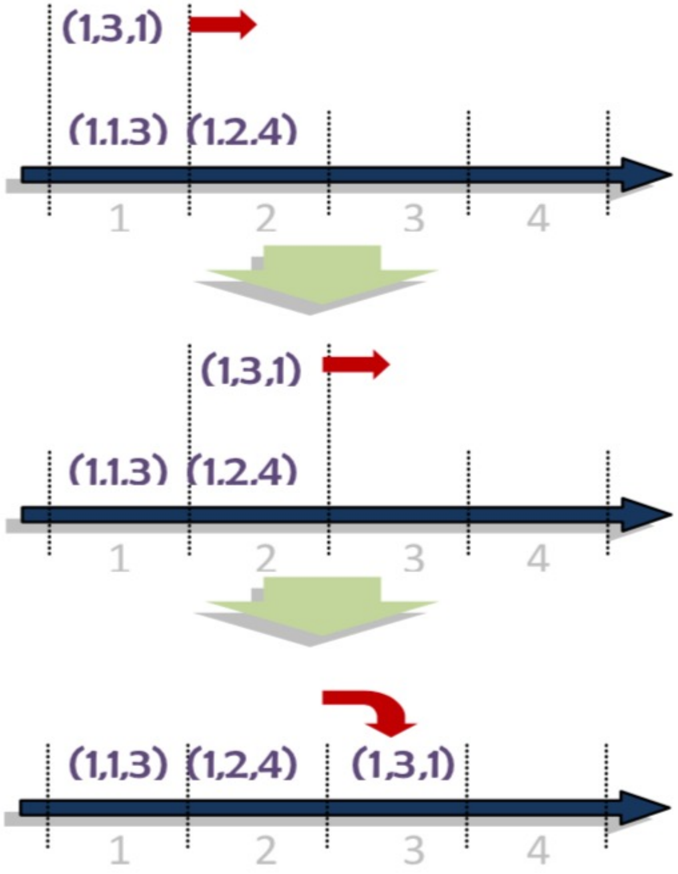
Algorithm 5 Sample: Task 4

Task4(1,3,1) , try active timeslot 1, but 1 matched task2, conflict.

$t_4 = 3 > t_1 = 1$

Try active timeslot 2， but 2 matched task1, conflict.

$t_4 = 3 > t_2 = 2$



Try active timeslot 3，3 is idle, 3 can match task 4

Result:
Task1(1,2,4)←→ active timeslot 2
Task2(1,1,3)←→ active timeslot 1
Task4(1,3,1)←→ active timeslot 3