SEPT 09 - LEC 01

- read the syllabus
- TOPICS
  1. algorithm design basics
     greedy, dynamic programming, linear programming,
  2. Complexity theory & NP completeness
  3. approximation algorithms

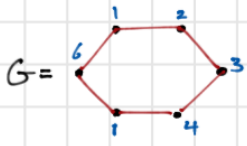
SEPT 11 - LEC 02

purpose of this course

1. design efficient algorithms
2. know when there is not an efficient algorithm (or when it is unlikely)
   an efficient algorithm runs in polynomial time ↖ poly. time
   algorithm runs in polynomial space ↙ ⇒ poly. space
   algorithm is the fastest is fastest known solution
   note polynomial refers to n^5, not n^{10000}

problem: some problems easy to check if a solution is correct but hard to find a solution

- ex. graph $G = (V, E)$ if given a min. spanning tree,
  easy to check: 1. it is a tree
  2. it has all the $v \in V$
  3. its the minimum most
- ex. given a graph $G = (V, E)$ and a number k
  find k vertices with no edges between them



- algorithms to find the k vertices are complicated
- check, given the k vertices is easy

- ex given n, find d, $1 < d < 120$ such that d is a factor of n
  - easy to check if a given d is a factor of $n = 2^m$
  - hard to find a d
    for $i = 1$ to $\sqrt{n}$
    if $i/n \rightarrow$ return i

  runtime $\sqrt{2^m} = 2^{\frac{m}{2}} = 2^{\log n/2}$

Types of algorithms

1. back tracking / brute force
   - check all possible answers, takes exponential time
   - usually not efficient
2. divide & conquer - ex. merge sort
3. greedy - ex. MST
   - when the solution for a subproblem is found, then the solution for the larges problem is found
     - ex. MST for G with v_1 removed for both is still MST
   - better than algorithms that generate all possible solution because a greedy algorithm reduces the number of possibilities that needs to be tracked
     1. an optimal solution for a large problem is also the optimal solution for a smaller subset of the problem
     2. no backtracking / check multiple solutions needed b/c all decisions made by the algorithm are final

SEPT 13 CSC 373

Interval Scheduling Problem (ISP)

INPUT: a list of non-sorted intervals I

$I_1, \ldots, I_n$ where $I_i = (s_i, f_i)$

s = start time f = finish time

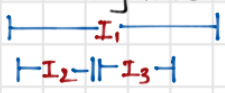
OUTPUT: return a list of intervals do not intersect

if intervals intersect at the end points, its valid

want to maximize the number of intervals

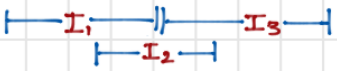
IDEAS —— 1. earliest starting or according to the input list

counter example:



2. the shortest intervals to allow more intervals

Counter example:

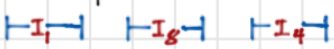


3. the interval with the minimum # of intervals it intersections by peeking first

counter example:

optimal solution:

algorithm output:



4. sort by earliest finishing intervals (correct)

ALGORITHM

Intuition: pick the interval w/ min. finishing time, reduce the size of the problem from end of that interval on.

⇒ Can fit the most amount of intervals past the point of the previously selected intervals

$S = \emptyset$

for $i = 1$ to n; do

add the next interval with the earliest finishing time that does not intersect previously selected interval

return S

- can put all intervals in a min-heap by finish time or initially sort the list of intervals

- greedy algorithm: pick a solution depend on local conditions, and at each step of the algorithm:
  - the optimal solution is also the optimal solution for a subset of the input
  - reduce the problem to 1 smaller problem

PROOF OF CORRECTNESS —— exchange argument

let S be the solution that the algorithm returned

let S' be an optimal solution

Want —— argue S has the same or more than S'

IH —— for all i, the set we have selected is one of "the best thing" we could and we can extend it to become an optimal solution. (start = empty S.)

It is true before the loop & assume that this is true before loop value i

Suppose selected S_{i-1} = list of intervals select before the i^{th} intervation of the loop

By IH: S_{i-1} can be extended to an optimal solution

note that S_{i-1} is optimal at I_{i-1}

$$S_i = \begin{cases} S_{i-1} & \text{if } I_i \text{ intersects one of the previous intervals in } S_{i-1} \\ S_{i-1} \cup \{I_i\} & \text{otherwise} \end{cases}$$

S_i' extends S_{i-1} and is an optimal solution

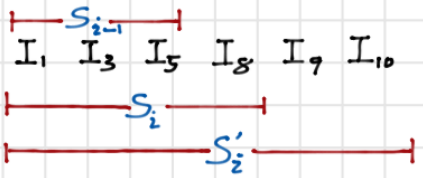
if I_i intersects S_{i-1} then $S_i = S_{i-1}$ and $S_{i-1} \subseteq S_i'$ therefore S_i can be extended into an optimal solution by IH

else if I_i does not intersect S_{i-1} then $S_i = S_{i-1} \cup \{I_i\}$

if $I_i \in S_i'$ then $S_i \subseteq S_i'$

else $I_i \notin S_i'$ then in this case we can replace the next interval in S_i with I_i because I_i finishes earlier than/or at the same time than to that interval, so I_i will not intersect any other interval in S'

- ex. I_1 I_3 I_5 I_8 I_9 I_{10}



- ex.

Steps —— select I_1 $S = \{I_1\}$
ignore I_2 b/c overlap w/ I_1
select I_3 $S = \{I_1, I_3\}$
ignore I_4, I_5 b/c overlap w/ I_3
Select I_6 $S = \{I_1, I_3, I_6\}$
ignore I_7 b/c overlap w/ I_6
Select I_8 $S = \{I_1, I_3, I_6, I_7\}$
ignore I_9 b/c overlap w/ I_8

final $S = \{I_1, I_3, I_6, I_7\}$

SEPT 12 — TUTOR

1. Asymptotic Notation

$f(n)$, $g(n)$ are functions

if $f(x) = O(g(x))$ then
$\exists c > 0, x_0$ s.t. $f(x) \le c \cdot g(x), \forall x \ge x_0$

if $f(x) = \Omega(g(x))$ then
$\exists c > 0, x_0$ s.t. $f(x) \ge c \cdot g(x), \forall x \ge x_0$

if $f(x) = o(g(x))$ then $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$

if $f(x) = \omega(g(x))$ then $\lim_{x \to \infty} \frac{g(x)}{f(x)} = 0$

if $f(x) = \Theta(g(x))$ then
$\exists c_0, c_1, x$ s.t. $c_0 g(x) \le f(x) < c_1 g(x), \forall x \ge x_0$

- ex $x^2 = o(x^3)$ $\log_2 n = \Theta(\ln n)$
$e^x = \omega(x^2)$ $\frac{1}{n} = o(1)$ since $\lim_{n \to \infty} \frac{1/n}{1} = 0$
$2^n = o(3^n)$

2. Recurrence Relations —— ex. $T(n) = T(n/2) + 1$

masters theorem:
if $T(n) = a \cdot T(n/b) + O(n^d)$ ← polynomial w/ degree d

then $= \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log_b a) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$

the recursive dominates — the polynomial dominates

- ex $T(n) = 10 \cdot T(n/5) + \Theta(n)$
then $a = 10$, $b = 5$, $d = 1$
$\log_5 10 > 1$ then $\Rightarrow T(n) = O(n^{\log_5 10})$

- ex. $T(n) = T(n-1) + \log n$, $T(2) = 1$
$= T(n-2) + \log(n-1) + \log n$
$= T(n-2) + \log(n-2) + \cdots + \log n$
$= \sum_{i=2}^{n} \log i = \log(n \cdot (n-1) \cdots 2)$
$= \log(n!) < \log(n^n) = n \log n$

can't use masters theorem

$\Rightarrow T(n) \in O(n \log n)$

3. Induction Proof — ex. $T(n) = O(\log n)$
   1. base $n = 1, 2, 0$
   2. IH: suppose claim holds for $k = n-1$
      or if using complete induction $k < n$
   3. prove correctness for n

4. Basic Data Structures

graph: $G = (V, E)$ — E: set of edges, $|E| = m$; V: set of vertices, $|V| = n$

Sept 16 Lecture 4

Review: Interval Scheduling Problem (ISP)

algorithm: $S = \emptyset$

sort intervals according to their finishing time

for $i = 1$ to $n \leftarrow$ # intervals; do

if I_i does not intersect S; then

$S \leftarrow S \cup \{I_i\}$

return S

Exchange Argument

Induction on $P(i)$ = the S before ith iteration of the loop (S_i) can be extended into an optimal solution

S' = an optimal solution extending S_i

Cases for $S_{i+1} \longrightarrow S_{i+1} = S_i$ (I_{i+1} intersects S_i)

$\longrightarrow S_{i+1} = S_i \cup \{I_{i+1}\}$

$I_{i+1} \notin S' \longleftarrow$ $\longrightarrow I_{i+1} \in S'$

then show:

S_{i+1}: S_i I_{i+1}

S': S_i I_i'

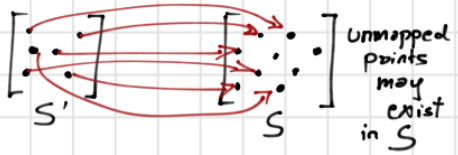
Recharging Argument — second type of argument to prove a greedy algorithm

Let S be the solution returned by the greedy algorithm

Let S' be an optimal solution

Want: show that $|S| \geq |S'|$

define a function $f: S' \longrightarrow S$ (one-to-one) to show $|S| \geq |S'|$



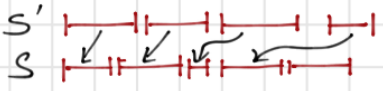
$S' = I_1', \ldots, I_{k'}'$ $\quad S = I_1, \ldots, I_k$

$f(I_i')$ = the interval in S which has the earliest finishing time and doesn't intersect previously selected intervals of S

need to prove: f is a function

f is one-to-one

note that: the intervals mapped so far should not be later than the optimal solution



prove f is a function ——— omitted

prove f is one-to-one

for every $j \leq k'$,

finishing time of $f(I_j') \leq$ finishing time of I_j

- only need to show we can map the next interval I_{j+1} to an interval in S. But there is something in S after $f(I_j')$.
  - the algorithm finds the first interval that does not intersect with $I_1, \ldots, I_{j-1}$ and I_j' does not intersect then
  - so finishing time of $I_{j-1} \leq \cdots I_{j-1}'$

SEPT 18 — LEC 05

Problem: List of jobs, each job has length & price
Want: maximize profit (total price of selected jobs) in a given amount of time
$job_i = (len_i, price_i)$ — ex. job1 = (5, 3), job2 = (7, 1), job3 = (3, 5)
- greedy algorithm can't be used here (can't prove it)
- can be efficiently solved by dynamic programming

greedy algorithm running time for ISP
Sort according to finishing time $O(n \log n)$
for i = 1 to n; do $O(n)$ ×
  if I_i.start ≥ f
    $S = S \cup \{I_i\}$
    $f = I_i$.finish
  } $O(1)$
return S

total running time = $O(n \log n) + O(n) + O(1)$
$\in O(n \log n)$

Brute force algorithms
Template: Best = Some solution
for all possible solutions S
  poly time: check if S statisfies the restrictions
    ↳ i.e. - ex. intervals in S do not intersect
  check the value of S
  if it is more optimal than Best; then
    update Best

Problem: too many possible solution (exponentially many)
- ex. interval scheduling: the ※ possible solutions = ※ subset of jobs
⇒ n jobs then 2^n subset

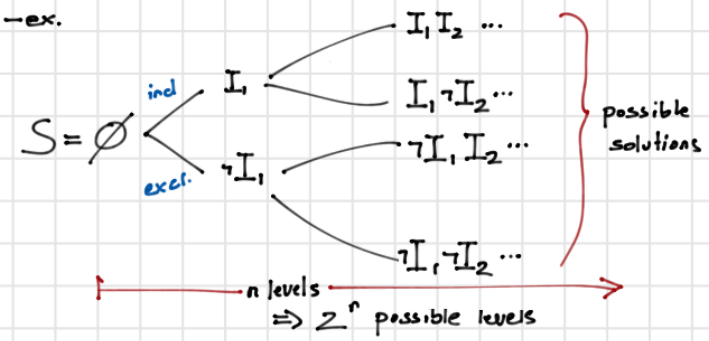
Note: Bruteforce does not build solutions like greedy

Improvements over Brute Force Algorithms
depends on the fact that each possible solution has clear "parts"
the decision about each interval can be made seperately
- ex. not an example: given a numbes n, find a nontrivial factor of n — no clear parts

in greedy — make a decision which extends to an optimal solution. Don't need to check past decisions

in brute force with backtracking — build all solutions recursively and evaluate them to find an optimal solution

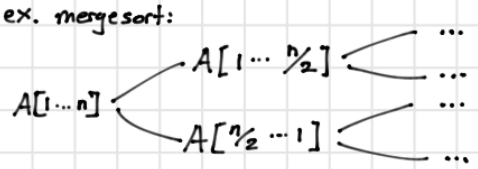
- ex.



- can improve backtracking by eliminating subtrees early
  - ex. I2 subtree doesn't have to satisfy some solution requirements ⇒ doesn't have to explore

in divide & conquer — divide a solution into two or more seperate parts and combine into final solution later

- ex. mergesort:



SEPT 20 — LEC 02

SEPT 19 — TUT 02

Interval Coloring Problem (ICP)

INPUT: ⇒ list of intervals $L = [I_1, \dots, I_n]$

OUTPUT: minimize # of colors needed to color them such that no two intersecting intervals have the same color

IDEAS ——— find the max # of non intersecting intervals

Counter-example:

ALGORITHM

sort interval according to their starting time $O(n \log n)$

colorHeap = {1, ..., n}, maxColor = 0

for i = 1 to n; do

$O(n) \times$ while Active.Min().finish ≤ I_i.start; do } check active jobs that's done

c = Active.ExtractMin().color

$O(\log n) \rightarrow$ colorHeap.Insert(c)

I_i.c = colorHeap.ExtractMin() ← color I_i

Active.Insert(I_i.c, I_i.finish) ← minheap

(val, key)

if max < I.c; then

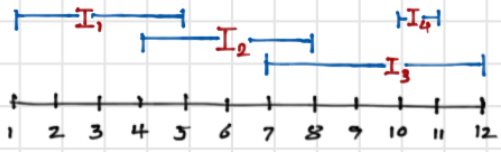
max = c

Total running time = $O(n \log n)$ + while + $O(n \log n)$

while loop through the entire execution of the problem will only run n times and extract n times

$= O(n \log n)$

-ex.



colorHeap	Active	Max	i
{1,2,3,4}	∅	0	0
{2,3,4}	I_1	1	1
{3,4}	I_1, I_2	2	2
{3,4}	I_2, I_3	2	3
{3,4}	I_3, I_4	2	4

Proof Idea:

the colors after the i^{th} iteration can be extended to an optimal coloring at step i ——— Let colors assigned after i^{th} iteration = $J_i \subseteq J'$

$J_{i+1} = J_i \cup \{(I_{i+1}, c)\} \subseteq J''$ ← another optimal

Second Proof: max intersections found

Interval Scheduling Problem on m machines (m-ISP): Schedule a set of intervals $\{I_1, I_2, \cdots, I_n\}$ on m machines such that no two intervals scheduled on the same machine intersect. Note that each interval I_i has a start time s_i and a finish time f_i. This problem is an extension of the standard Interval Scheduling Problem discussed in the lecture.

An optimal algorithm

Algorithm 1: Best Fit EFT (an extension of the standard EFT algorithm)

Sort intervals such that $f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n$
for $k = 1$ *to* m **do**
  $e_k = 0$ `// e_k is the latest finish time of intervals on machine k.`
for $i = 1$ *to* n **do**
  Let $k = \begin{cases} \arg\min_l(s_i - e_l \geqslant 0) & \text{if such } l \text{ exists} \\ 0 & \text{if such } l \text{ does not exist} \end{cases}$
  $\sigma(i) = k$ `// σ(i) specifies on which machine Interval I_i is scheduled. σ(i) = 0 means that I_i is not scheduled.`
  $e_k = f_i$

Proof of optimality: The exchange proof method.
Idea: Let $S_0, S_1, ..., S_n$ be the partial solutions constructed by the algorithm at the end of each iteration. The solution S_i contains the scheduling for intervals $I_1, \cdots, I_i$.
Prove each S_i can be *completed (extended)* to reach an optimal solution (just by scheduling $I_{i+1}, \cdots, I_n$). Call that optimal solution S'_i. The scheduling for all intervals $I_1, \cdots, I_i$ are the same in both S_i and S'_i.
If S'_i exists, we say S_i is *promising*.
Note: S'_i may not be unique (there may be more than one way to achieve optimal).
Prove that S_i *is promising* by induction in i (number of iterations).

Proof:

- Base case: $S_0 = \{\}$: any optimal solution S'_0 extends S_0 just by scheduling the intervals in $\{I_1, ..., I_n\}$.
- Ind. Hyp.: Suppose $i \geqslant 0$ and optimal S'_i extends S_i by scheduling only the intervals in $\{I_{i+1}, ..., I_n\}$.
- Ind. Step (To prove): S_{i+1} is promising w.r.t. $\{I_{i+2}, ..., I_n\}$.

  Let's see what happens in iteration $i + 1$. There are two cases.

  1. The algorithm sets $\sigma(i + 1) = 0$

     It means that I_{i+1} conflicts with all machines according to the S_i scheduling. Thus, in S'_i we should have $\sigma_{S'_i}(i + 1) = 0$ (otherwise, S'_i has a conflict and it is not a solution). Set $S'_{i+1} = S'_i$. Thus, S_{i+1} is promising.
     Note: $\sigma_{S'_i}(i + 1)$ is the scheduling for interval I_{i+1} in S'_i.
  2. The algorithm sets $\sigma(i + 1) = k$ $(k \neq 0)$

     Three cases may happen:

     (a) $\sigma_{S'_i}(i + 1) = k$
     Set $S'_{i+1} = S'_i$. Thus, S_{i+1} is promising.

     (b) $\sigma_{S'_i}(i + 1) = 0$
     It means that there is an interval I_j scheduled by S'_i on machine k that conflicts with I_{i+1}; otherwise we can change $\sigma_{S'_i}(i + 1)$ to k (schedule I_{i+1} on machine k) and get a better solution. It means

that S'_i is not optimal that is a contradiction!
Moreover, $j > i+1$ and also I_j is unique. Why? If there are two intervals I_{j_1} and I_{j_2}, since $f_{i+1} \leqslant f_{j_1}$ and $f_{i+1} \leqslant f_{j_2}$, they should conflict. Hence they cannot be part of a solution.
Therefore if we set $\sigma_{S'_i}(i+1) = k$ and $\sigma_{S'_i}(j) = 0$, the updated scheduling S'_i still extends S_i and is optimal.
Set S'_{i+1} to this updated S'_i. Hence, S_{i+1} is promising.

(c) $\sigma_{S'_i}(i+1) = k'$ ($k' \neq k$, $k' \neq 0$)
Look at machines k and k'. First we know that $s_{i+1} - e_k \geqslant 0$. Thus, $s_{i+1} \geqslant e_k$.
Second, $s_{i+1} - e_k$ has the minimum positive value among all machines. Thus, $e_{k'} \leqslant e_k$.
Substitute all jobs after e_k on machine k with all jobs after $e_{k'}$ on machine k'. Note that the number of scheduled intervals remain the same and there is no conflict. why?
In the new scheduling, I_{i+1} is scheduled on machine k. This scheduling can be utilized to extend S_{i+1}. Hence, S_{i+1} is promising.

Thus, S_n is promising. It means that S_n is optimal.

Problem 1. Fractional Knapsack (greedy algorithm). There are n items $I_1, \cdots, I_n$. Item I_i has weight w_i and worth v_i. All these items can be broken into smaller pieces. We have a knapsack with capacity S and want to pack this knapsack with the maximum value. So we may decide to carry only a fraction x_i of item I_i where $0 \leqslant x_i \leqslant 1$. Example: if S = 10, w1 = 5, v1 = 2, w2 = 6, v1 = 1, w3 = 4, v1 = 3, then the optimal value we can get occurs when we take 4 units of the item 3 (worth 3), 5 units of item 1 (worth 2), and 1 unit of item 2 (worth 1/6) for a total value of 31/6.
Solution. Sort according to decreasing unit value: $\frac{v_1}{w_1} \geqslant \frac{v_2}{w_2} \geqslant \cdots \geqslant \frac{v_n}{w_n}$
Optimality: The optimality is trivial. There are two factors we should consider.

- There is an optimal solution that begins with the item that has the maximum unit value. To prove this, let the item I' has the maximum ratio $\frac{v'}{w'}$. Thus for all items I we have $\frac{v'}{w'} \geqslant \frac{v}{w}$. Thus, $v' \geqslant \frac{v \times w'}{w}$. Now assume there is an optimal solution that does not use the full w' weight of I'. Then if we replace any amount of any other item I in the solution with I', the total value will not decrease.
- Let O be the optimal solution for the original problem P. Moreover, let I be the first greedy choice we make in O. Then $O - I$ is the optimal solution for the subproblem P' after the first greedy choice has been made.

Problem 2. Knapsack with repetition (dynamic programming): page 167, the DPV textbook (in Section 6.4). Note the difference with the classic knapsack problem (knapsack without repetition).

Problem 3. The traveling salesman problem (dynamic programming): pages 173-175, the DPV textbook (in Section 6.6).

SEPT 23 LEC 09

Minimum Spanning Tree

INPUT: weighted connected Graph

OUTPUT: minimum weight spanning tree → contains no cycles
→ contains all vertices

-ex.

— MST

cost

Kruskal's algorith for finding a MST

IDEA: grow a forest w/ minimum total weight
↳ graph w/ no cycles;
tree = forest + connected

Algorithm:

$S = \emptyset$

Sort the edges according to their weight

for vertex $v \in V$ do

makeset(v)

for edges u,v do

if adding w to S does not create a cycle

$S \leftarrow S \cup \{uv\}$ — Union(u, v)

return S

check if u & v are not in the same disjoint sets by:

if FindSet(u) != FindSet(v)

Proof of Correctness

S_i = the solution after the i^{th} iteration of the loop

S_i can be extended into the optimal solution

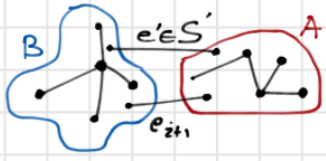
let $S_i \subseteq S'$ be an optimal solution

Consider S_{i+1} → $S_{i+1} = S_i \Rightarrow$ DONE

→ $S_{i+1} = S_i \cup \{e_{i+1}\}$

$e_{i+1} \notin S'$ then each of A, B contains one end point of e_{i+1} and e'

$e_{i+1} \in S' \Rightarrow$ DONE



$A, B \subseteq V$

$A \cup B = V$

$A \cap B = \emptyset$

then show: $S'' = S' \cup \{e_{i+1}\} - \{e'\}$

$w(S'') = w(S') + w(e_{i+1}) - w(e')$

b/c way algorithm works: $w(e_{i+1}) \geq w(e')$

$\Rightarrow \quad w(e_{i+1}) - w(e') \geq 0$

Need to argue: S'' is a tree and a MST (textbook)

Inverse of Kruskal:

Sort edge from high to low

for all edges in the sorted list

remove largest edge from list

if removing that edges does not disconnect the graph

remove it from the graph

return left overgraph

SEPT 25 — Lec 08

Huffman Encoding — for compressing documents

- given a document containing symbols $S_1, \ldots, S_n$ encode the document by encoding each symbol w/ a binary string
- encoded string must be prefix free
  - the code for symbols are not prefixes of each other
  - prevent this problem: A = 001, B = 00, C = 100 — what is 00100 decoded?
- want to minimize total # bits in the encoded document

Huffman encoding vs. n-bit encoding

- saves space b/c will encode more freq. symbols w/ less bits
- ex.

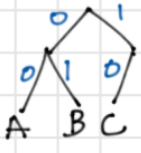
Symbol	freq	dynamic encoding	2-bit
A	70	0	00
B	3	100	01
C	20	101	10
D	37	11	11
total =		231	260

INPUT: Symbols $S_1, \ldots, S_n$
$f(S_j)$ = # times that S_j appears in the document

GOAL: $w(S_i)$ = # bits used to encode the symbol S_i such that the total # bits in the encoded output document $T = \sum f(S_i)\, w(S_i)$ is minimized

IDEA: find a full BST — each symbol has a unique path from root to a leaf ⇒ no symbol is on the path from root to another symbol (ensures prefix free code)

- ex.

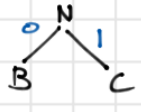


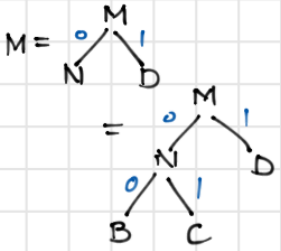
A = 00, B = 01, C = 10

A B B C
00 01 01 10

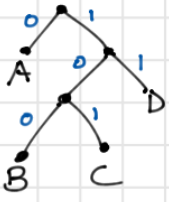
To build the tree: sort all symbols by freq (non-descending) then combine the 2 lowest freq recursively

- ex. A, B, C, D —sort→ B, C, D, A → N, M, T

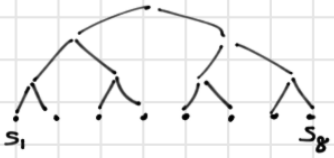




T = (final tree)



Note: that if all freq equal, then will produce a full tree



ALGORITHM

```
for i from 1 to n, do
    H.insert(s_i, f_i)        } buildheap(... )
                  keys
for i from 1 to n-1, do        → items in H, decrease by 1 every time
    S = H.ExtractMin()
    S' = H.ExtractMin()
    n = newTree(S, S')
    H.insert(n, freq(S) + freq(S'))

return H.ExtractMin()
```

PROOF

the solution after the i^{th} decision can be extended into an optimal solution

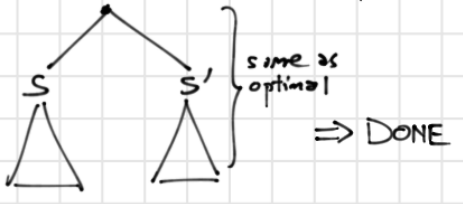
Base Case $i = 0$ easy

Induction Step $i > 0$

we can combine the trees obtained upto step i into an optimal tree

Assume that we can combine the trees at step i to obtain an optimal tree then at $i+1$ we can combine two trees into one

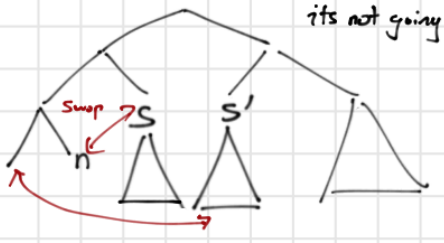
Case1 Combined tree same as optimal



⇒ DONE

Case2: diff from optimal — then swap a symbol to the bottom its not going to be worse

Let n be the tree placed next by freq below S



then swap S and S' to the bottom

$f(n) \geq f(s)$ and $f(n) - f(s) \geq 0$ ⇒ $f(n)\,w(n) + f(s)\,w(s) \geq f(n)(w(n)-1) + f(s)(w(s)+1)$

SEPT 28 TUTS

MINIMUM SPANNING TREES (MSTs)

-recap: the following statements are equivalent definitions of a tree $G=(V,E)$, $|V|=n$, $|E|=m$, $T \subseteq E$

1. Acyclic, $|T|=n-1$
2. Connected, $|T|=n-1$
3. Acyclic, connected
4. There is unique path between every two nodes

Input: A weighted undirected graph $G=(V,E,w)$

Minimum Spanning Tree: a spanning tree of minimum weight

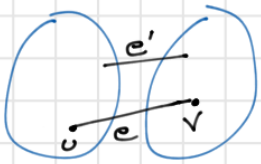
Algorithms: Prim, Kruskal

Q: given a graph $G=(V,E,w)$ with a unique edge $e \in E$ of min weight is e contained $e \in E$ in every MST

A: Yes. Suppose, for contradiction, T is an MST such that $e \notin T$

Consider $T \cup \{e\} \Rightarrow e$ is continous in a cycle C

remove an edge $e' \in C$ such that $e' \neq e$ $w(e') > w(e)$



$T' \leftarrow T \setminus \{e'\} \cup \{e\}$

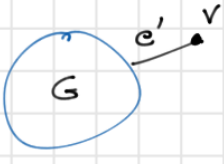
T' is such that: $n-1$ edges and connected

$w(T') = w(T) - w(e') + w(e) < w(T)$

$\Rightarrow$ contradiction

Q: same graph as before but now G contains a unique edge of max weight e' is it true that e' isn't contained in any MST?

A: No. Counter example G is a tree



Q: Same as before but now $\exists$ cycle C that contains e'. Is the claim true?

A: Yes. Suppose for contradiction $\exists$ MST T such that $e' \in T$

$\exists e'' \in C$ s.t $w(e') < w(e'')$, $e'' \notin T$

$T' \leftarrow T \setminus \{e'\} \cup \{e''\}$ and T' is MST

$w(T') < w(T)$

Q For any graph G w/ distinct weights the two edges w/ the smallest weights are contained in every MST

A True. write e_1 — min weight
e_2 — 2nd lightest weight
already know: e_1 is in any MST. suppose for contradiction $e_2 \in T \leftarrow MS$

P is a path between u and v
P is of length ≥ 2 ($e_2 \notin P$)
$\Rightarrow$ P contains an edge $e' \neq e_1$

$T' \leftarrow T \setminus \{e'\} \cup \{e_2\}$
T' connected, n-1 edges } spanning Tree
$w(T') = w(T) + w(e_2) - w(e')$
$< w(T)$
$w(e') > w(e_2)$ $\Rightarrow$ contradiction

Q Graph $G = (V, E, w)$, $A \subseteq E$ free edges
find an algorithm that finds the min cost Spanning Tree T such that T minimize $w(T \setminus A)$

A: Solution = update weight.

$$w'(e') = \begin{cases} 0 & \text{if } e \in A \\ w & \text{otherwise} \end{cases}$$

then run Prim / Kruskal

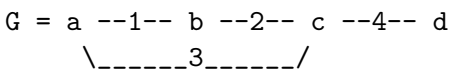
Q input $G = (V, E, w)$
$T = MST$, $e \notin T$ if $w(e)$ is reduced see if it goes into the tree
update: $w(e) = w(e) - x$, $x > 0$
update T

Problem 1. If graph G is connected and contains more than $n-1$ edges (where $n = |V|$, as usual), and if there is a unique edge e with minimum cost, then is e guaranteed to be in every MST of G? If so, give a convincing argument. If not, provide a counter-example. In this case, what other conditions can you put on G to guarantee that e will be in every MST of G?

Solution. Proof. For a contradiction, suppose T is a MST that does not contain e. Then T contains some path between the endpoints of e. Pick some edge e' on this path. Then $T' = T \cup \{e\} - \{e'\}$ is a spanning tree. But $c(T') = c(T) + c(e) - c(e') < c(T)$ because $c(e') > c(e)$ (by assumption, $c(e)$ is minimum and unique). This contradicts the fact that T is a MST.

Problem 2. If graph G is connected and contains more than $n-1$ edges (where $n = |V|$, as usual), and if there is a unique edge e with maximum cost, then is e guaranteed *not* to be in any MST of G? If so, give a convincing argument. If not, provide a counter-example. In this case, what other conditions can you put on G to guarantee that e will be in no MST of G?

Solution. Counter-example:



Additional condition:

If e belongs to some cycle C in G, then e belongs to no MST.

Proof. For a contradiction, suppose e belongs to a MST T. Consider $T - \{e\}$. This is made up of two connected components. Because e belongs to some cycle C, there is a way to get from one endpoint of e to the other along this cycle. So there is at least one edge e' of C that connects both components. Then $T' = T \cup \{e'\} - \{e\}$ is a spanning tree, and $c(T') = c(T) + c(e') - c(e) < c(T)$ because $c(e') < c(e)$ (by assumption, $c(e)$ is maximum and unique). This contradicts the fact that T is a MST.

Problem 3. For every graph G whose edge weights are all distinct, every MST of G contains the two edges e_1, e_2 with the two smallest weights. If this is true, give a convincing argument. If not, provide a counter-example. In this case, what other conditions can you put on G to guarantee that e_1, e_2 will be in every MST of G?

Solution. TRUE. Suppose G is a graph whose edge weights are all distinct. Let e_1, e_2 be the two edges with smallest weights ($c(e_1) < c(e_2) <$ cost of every other edge).
For a contradiction, suppose T is a MST that does not contain both e_1 and e_2. WLOG, suppose T does not contain e_2. Consider the endpoints (u, v) of e_2. They are connected by a path P in T. This path contains at least two edges (it cannot contain just one as this would just be e_2 itself). Since e_1, e_2 have the two smallest edge costs, there is at least one edge e' on P with $c(e') > c(e_2)$. But then, $T' = T \cup \{e_2\} - \{e'\}$ is a spanning tree and $c(T') < c(T)$.
This contradicts the fact that T is a MST. Hence, every MST of G contains both e_1 and e_2.