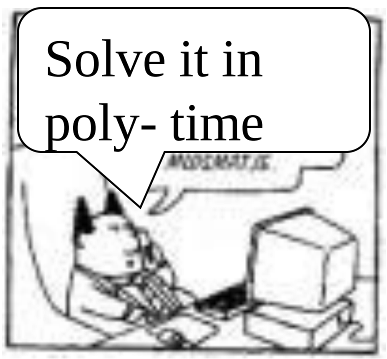
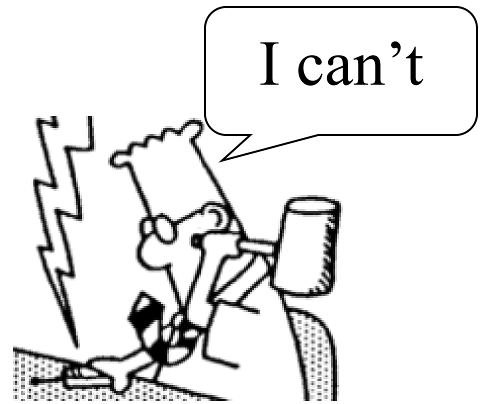
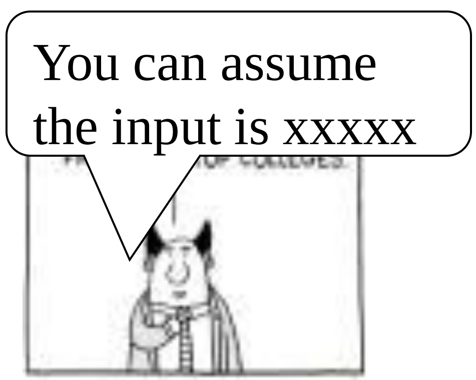
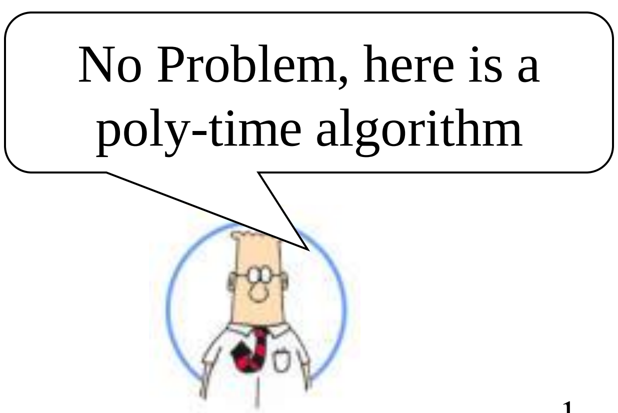
Solving NP-hard Problems on Special Instances









Solving NP-hard Problems on Special Instances

We are going to see that some problems that are NP-hard on general instances, can be solved efficiently when the instance has some special characteristics.

Similarly, some problems that are hard to approximate, can be approximated with better ratio for some instances.

Solving NP-hard Problems on Special Instances

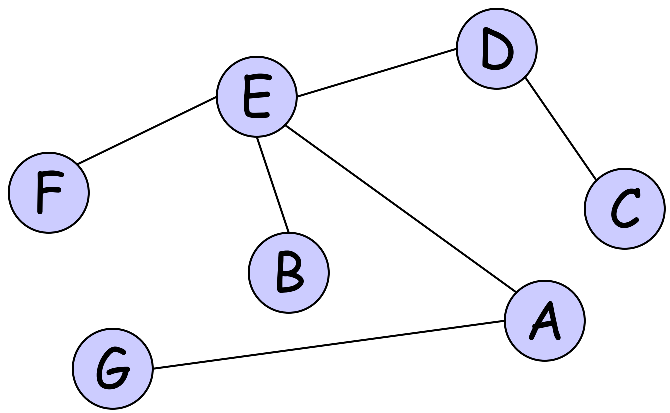
Special instance =>

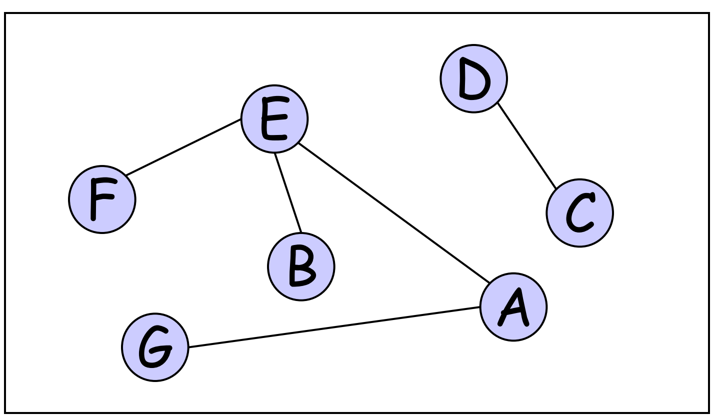
Structural properties =>

Can be exploited to solve the problem

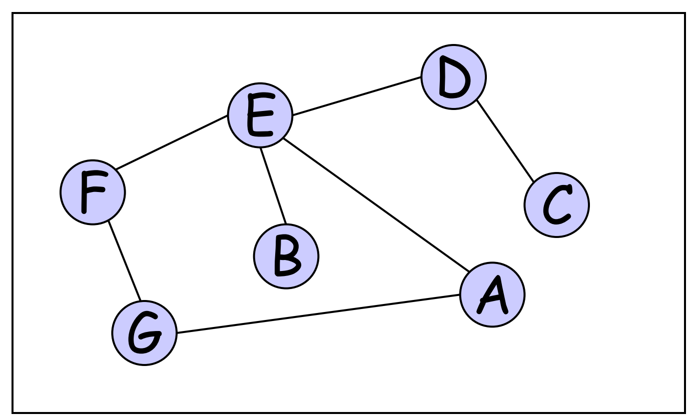
Trees

- An undirected graph is a tree if it is connected and contains no cycles.



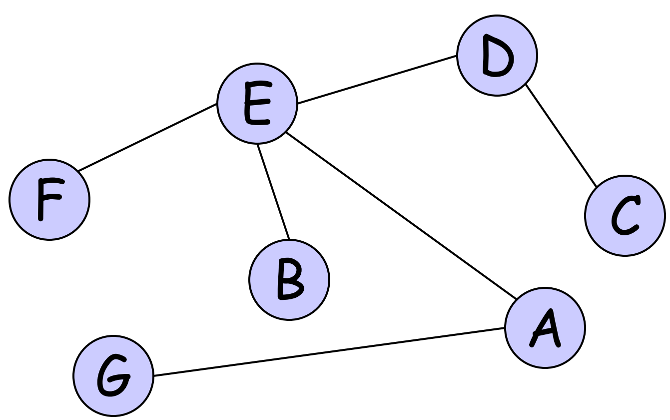


Not trees



Alternative Definitions of Undirected Trees

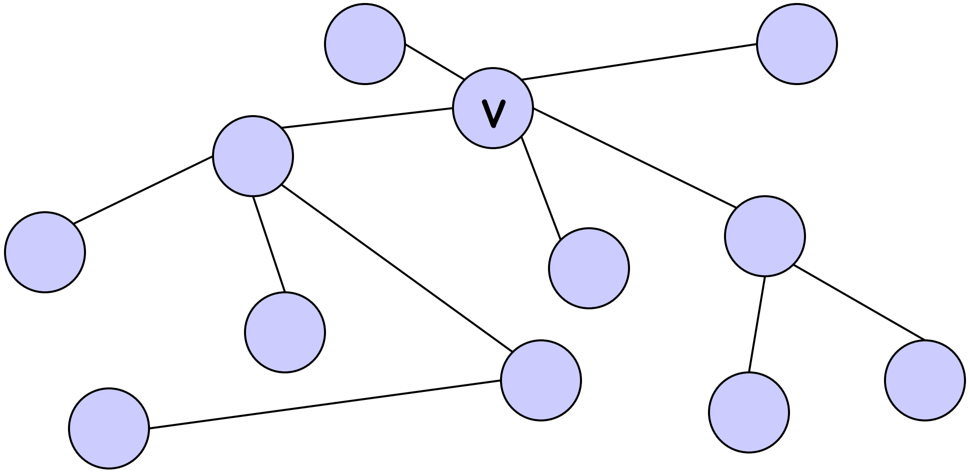
a. G is a tree (connected and contains no cycles).
b. G is cycles-free, but if any new edge is added to G, a circuit is formed.
c. For every two vertices there is a unique simple path connecting them.
d. G is connected, but if any edge is deleted from G, G becomes diconnected.



Solving NP-hard Problems on Trees

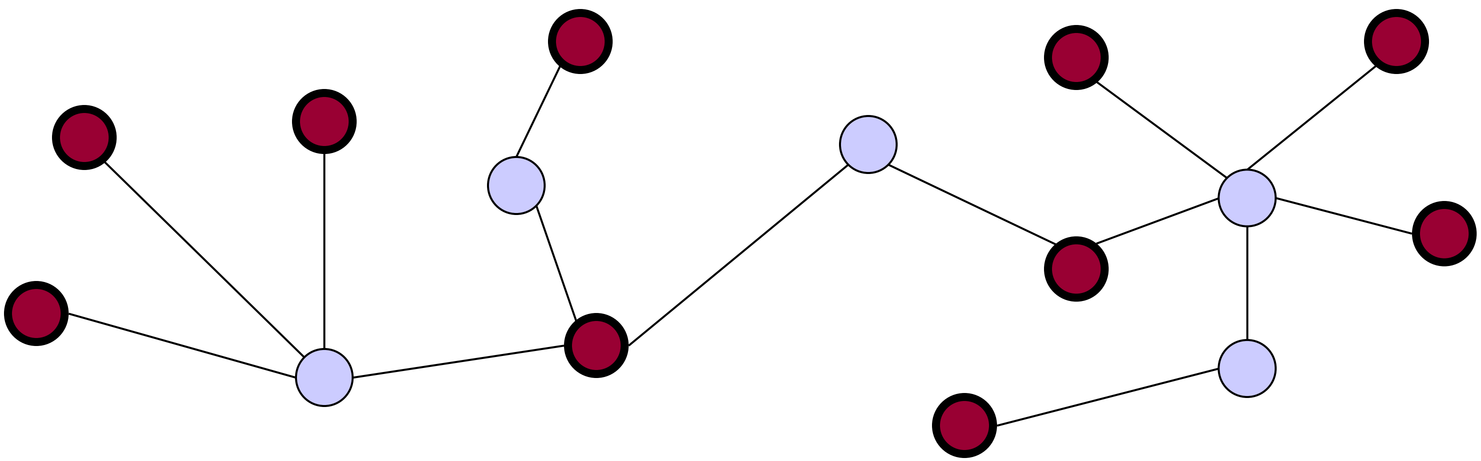
Some NP-hard problems can be solved in linear time on trees.

Intuition: if we consider a subtree of the input, rooted at v, the solution to the problem restricted to this subtree only interacts with the rest of the graph through v.



Solving Maximum Independent Set on Trees

- Input: A **tree** $T=(V,E)$
- Problem: What is the maximum size subset $S \subseteq V$ such that no pair of vertices in S is connected by an edge.



For general graphs, this is an NP-hard problem.

Solving MIS on Trees

- Idea: Consider an edge e=(u,v) in G. In any independent set S of G, at most one of u and v is in S. In trees, for some edges, it will be easy to determine which of the two endpoints will be placed in the IS.

- A leaf in a tree is a node with degree 1.
- Property: Every tree has at least one leaf. (why?)

Structural Property of MIS on Trees

- Claim: If $T=(V,E)$ is a tree and v is a leaf of the tree, then there exists a maximum-size independent set that contains v.
- Proof: In Class.

- The algorithm is based on that claim: Repeatedly identify a leaf, add it to the IS, remove it and the vertex adjacent to it (+ incident edges) from the tree (in fact, it might become a forest).

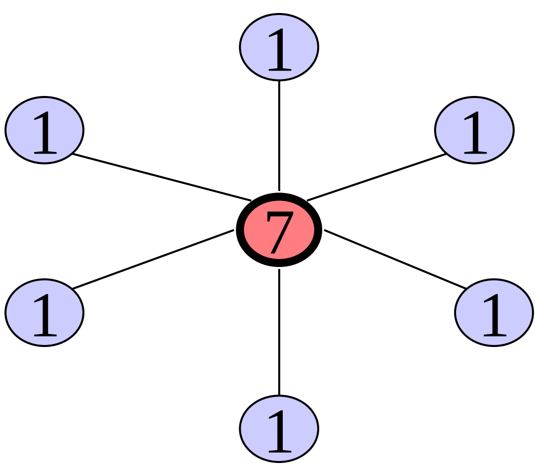
Maximum **Weighted** IS on Trees.

Assume each vertex has a positive weight w_v

The goal is to find an independent set S such that the total weight $\Sigma_{v \in S} w_v$ is maximized.

When for all v, $w_v=1$, we get the regular MIS problem.

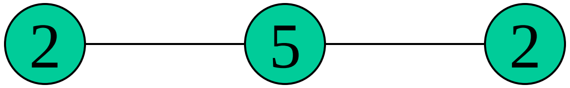
For arbitrary weights this is a different problem.



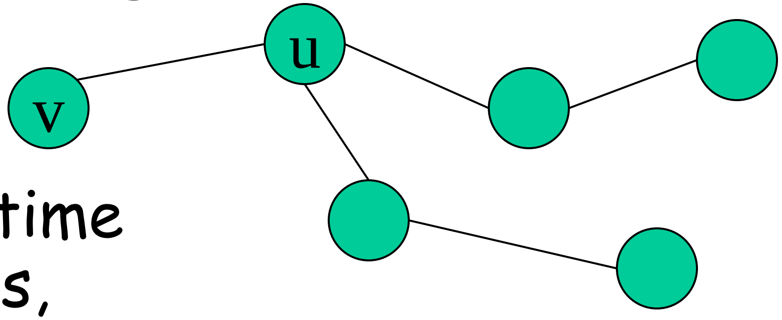
Picking the center is optimal.

Maximum Weighted IS on Trees.

In particular, it is not 'safe' anymore to include a leaf in the solution.



Let $e=(u,v)$ be an edge such that v is a leaf. If $w_v \geq w_u$, then it is safe to include it, but if $w_v < w_u$ then by including u we gain more weight but we block other vertices (neighbors of u) from entering the MIS.



We will see a polynomial time algorithm for trees, based on dynamic programming.

Dynamic Programming

- A strategy for designing algorithms.
- A technique, not an algorithm.
- The word "programming" is historical and predates computer programming.
- Use when problem breaks down into recurring small sub-problems.

Recursive Solutions

- Divide a problem into smaller subproblems
- Recursively solve subproblems
- Combine solutions of subproblems to get solution to original problem.

- In some cases, the same subproblems are repeated, (as subproblems of more than one bigger problem).

Recursive Solutions

- Occasionally, straightforward recursive solution takes too much time
- Solving the same subproblems over and over again
- Example: Fibonacci Numbers

$F(0) = 1$; $F(1) = 1$

$F(n) = F(n - 1) + F(n - 2)$

```
fib(n)
    if (n < 2)  return 1
     return fib(n-1) + fib(n-2)
```

How much time does this take? Exponential!

Recursive Solutions

- But how many different subproblems are there, for finding fib(n) ? Only n-1
- The recursion takes so much time because we are recalculating solutions to subproblems again and again.
- What if we store solutions to subproblems in a table, and only recalculated if the values are not in the table?

1	1	2	3	5	8	…

```
Fibonacci(n)
A[0] = 1; A[1] = 1
for i = 2 to n do A[i] = A[i-1] + A[i-2]
return A[n]
```

Dynamic Programming

* Simple, recursive solution to a problem.
* Straightforward implementation of recursion leads to exponential behavior, because of repeated subproblems.
* Create a table of solutions to subproblems.
* Fill in the table, in an order that guarantees that each time you need to fill in an entry, the values of the required subproblems have already been filled in.

Example: Most Profitable Tour

- Assume that you need to travel from the bottom row of a chessboard to the top row. You can select your initial and final locations (anywhere on the bottom and top rows)
- On each square (i,j) there are c(i,j) dollar-coins.

stop anywhere here →

4	3	12	7	1
7	4	1	3	8
3	18	4	7	13
8	4	1	2	5
6	9	5	3	4

Start anywhere here →

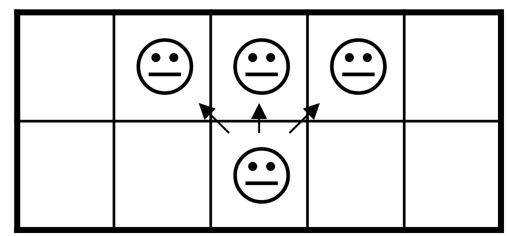
c(row, col)

c(2,4)=2

c(5,2)=3

Example: Most Profitable Tour

- Whenever you visit a square you can pick up the money on it.
- The amounts c(i,j) are known in advance.
- From each square you can advance to the next row in all three directions (diagonally left, diagonally right, or straight forward)
- You want to maximize your profit.



A possible tour.
Profit = 40

4	3	12	7	1
7	4	1	3	8
3	18	4	7	13
8	4	1	2	5
6	9	5	3	4

Most Profitable Tour

- Define $q(i,j)$ as the maximum possible profit to reach square (i,j).
- For any column j, $q(1,j)=c(1,j)$.
- For any column j and $i>1$,
  $q(i,j) = c(i,j)+\max\{q(i-1,j-1), q(i-1,j), q(i-1,j+1)\}$
- Make sure you don't leave the board:
  - if $j<1$ or $j>n$ then $q(i,j)= 0$.
- The goal: find $\max_j q(n,j)$

Most Profitable Tour - Recursive solution:

```
main()
    for j =1 to n
    q[j]= maxProfit(n, j)
return max_j q[j].

maxProfit(i, j)
    if j < 1 or j > n return 0
    if i = 1 return c(1, j)
    return max(maxProfit(i-1, j-1), maxProfit(i-1, j),
      maxProfit(i-1, j+1) ) + c(i,j) .
```

- Time complexity: Exponential.

Most Profitable Tour: DP solution

Input:

4	3	12	7	1
7	4	1	3	8
3	18	4	7	13
8	4	1	2	5
6	9	5	3	4

Output:

46	45	51	43	31
42	39	36	25	30
20	35	17	17	22
17	13	10	7	9
6	9	5	3	4

Time complexity: O(board size)

Dynamic Programming works!

```
function maxProfit( )  //for the whole table!
for j= 1 to n
    q[1, j] = c(1, j)
for i=1 to n
   q[i, 0] = 0
   q[i, n + 1] = 0
for i=2 to n
   for j= 1 to n
      m = max(q[i-1, j-1], q[i-1, j], q[i-1, j+1])
      q[i, j] = m + c(i, j)
```

```
main()
   maxProfit()
return max_j q[n,j].
```

Most Profitable Tour: DP solution

Finding the actual path:

- For each table (i,j) cell, remember which of the 3 cells (i-1,j-1), (i-1,j) , (i-1,j+1) contributed the maximum value

46	45	51	43	31
42	39	36	25	30
20	35	17	17	22
17	13	10	7	9
6	9	5	3	4

4	3	12	7	1
7	4	1	3	8
3	18	4	7	13
8	4	1	2	5
6	9	5	3	4

Example: Knapsack with bounded item values

- Define $A[i,p]$ = minimum weight of a subset of items $1,\ldots,i$ whose total value is exactly p. ($A[i,p] = \infty$ if no such subset)
  $i=1,\ldots,n$; $p=1, \ldots, nB$

- Dynamic programming solution:
  - $A[1,p]$ is easy to compute for all p.
  - $A[i+1,p]$ = minimum of $A[i,p]$ and $w_{i+1} + A[i,p-b_{i+1}]$
- OPT = maximum p for which $A[n,p] \leq W$
- Running time?
  Number of cells in table A $O(n^2B)$

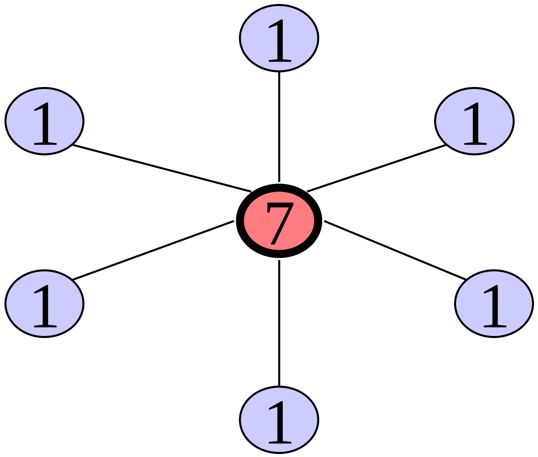
Maximum **Weighted** IS on Trees.

Assume each vertex has a positive weight w_v

The goal is to find an independent set S such that the total weight $\Sigma_{v \in S} w_v$ is maximized.

When for all v, $w_v=1$, we get the regular MIS problem.

For arbitrary weights this is a different problem.



Picking the center is optimal.

Maximum Weighted IS on Trees.

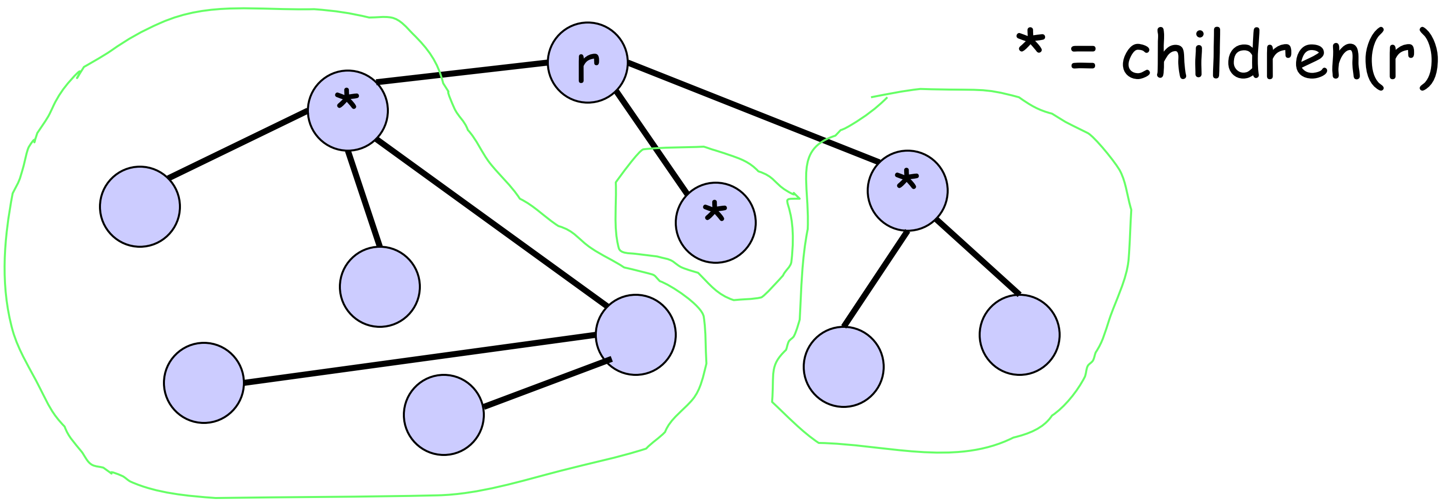
We will see a polynomial time algorithm for finding a MWIS on trees, based on dynamic programming.

- What are the subproblems?

We will construct subproblems by rooting the tree T at an arbitrary node r

For a root r and any $u \neq r$, parent(u) is the vertex preceding u on the path from r to u. The other neighbors of u are its children.

Maximum Weighted IS on Trees.



The subproblems will be the problems on each of the subtrees rooted at children(r).

Let T_u be the subtree of T rooted at u.

The tree T_r is our original problem.

If $u \neq r$ is a leaf then T_u consists of a single vertex.

Maximum Weighted IS on Trees.

For each vertex u, we keep two values:

$M_{out}[u]$: The maximum weight of an IS that does not include u in the subtree T_u.

$M_{in}[u]$: The maximum weight of an IS that includes u in the subtree T_u.

Base case: For a leaf u, the subtree rooted at u contains the single vertex u, therefore:

$M_{out}[u] = 0$

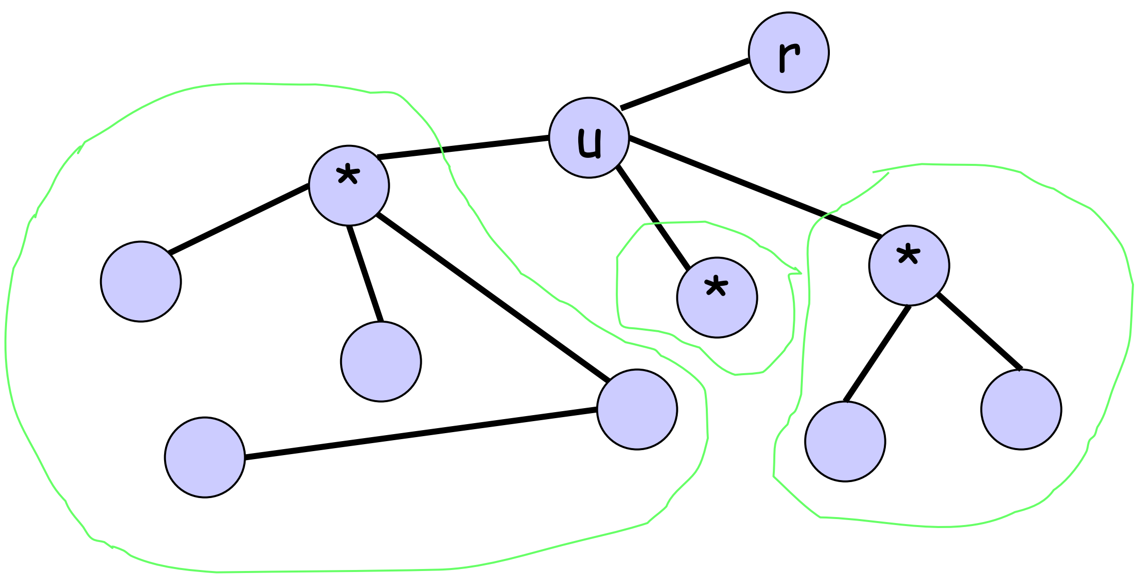
$M_{in}[u] = w_u$

Maximum Weighted IS on Trees.

For each vertex u that has children, the following recurrence defines the values of $M_{out}[u]$ and $M_{in}[u]$:

$M_{out}[u] = \Sigma_{v \in children(u)} \max(M_{out}[v], M_{in}[v])$;

$M_{in}[u] = w_u + \Sigma_{v \in children(u)} M_{out}[v]$;



If u is out then the *'s can be in or out. If u is in, all *'s must be out.

Maximum Weighted IS on Trees.

<u>The complete algorithm:</u>

Root the tree at a vertex r.

For all vertices u of T in *<u>post-order</u>*

If u is a leaf:

$M_{out}[u] = 0$

$M_{in}[u] = w_u$

else

$M_{out}[u] = \Sigma_{v \in children(u)} \max(M_{out}[v], M_{in}[v]);$

$M_{in}[u] = w_u + \Sigma_{v \in children(u)} M_{out}[v];$

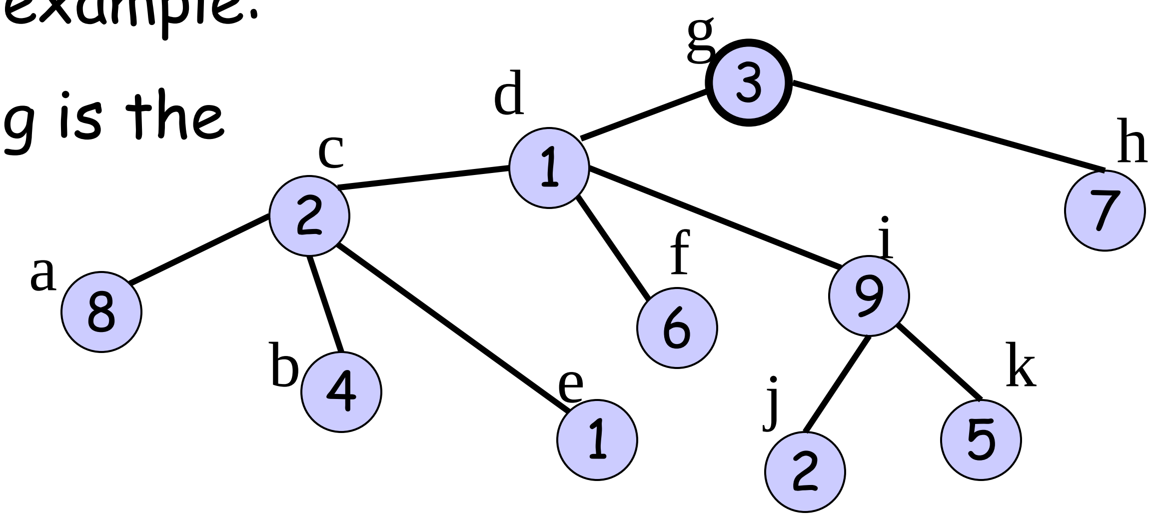
Return $\max(M_{out}[r], M_{in}[r]);$

In post-order, a node is processed after all its children.

Maximum Weighted IS on Trees.

Running example:

Assume g is the root



	a	b	c	d	e	f	g	h	i	j	k
order											
M_{in}											
M_{out}											

Facility Location

The location of a set of facilities should be determined. These facilities serve clients and we want them to be as close as possible to the clients.

facilities can be…

- factories, warehouse, retailers, servers, antennas.

    objective: min sum (or average) of distances.

- hospitals, police stations, fire-stations

    objective: min maximal distance.

Facility Location

Various questions:

- Where should a facility be?
- How many facilities should we build?
- How should demand be allocated?

Problems can be more complex (adding constraints)

- warehouse capacities
- each client can be allocated to only one warehouse
- different costs (transportation, holding, operating, set-up)
- distance / service time

FL Network Problems

1. Covering: how many facilities should be built so that each customer is within a given distance from its nearest facility?

   Example: fire stations.

2. Center Models (k-center problem)

   Where to build k facilities so as to minimize the max distance between facilities and a customer (between a customer and its nearest facility).

3. Median Models: (k-median problem)

   Minimize the sum of distances between customers and their nearest facility.

   Example: warehouse problem

Covering a Network

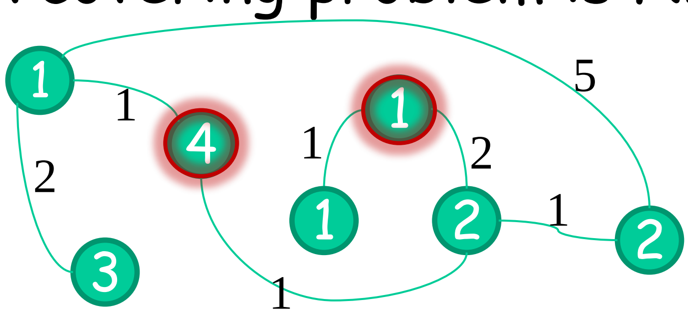
Covering: how many facilities should be built so that each customer is within a given distance from its nearest facility?

Possible problems:

- Each client has its own requirement, or all clients have the same requirement.
- Facilities can be located only on vertices or any point in the network.

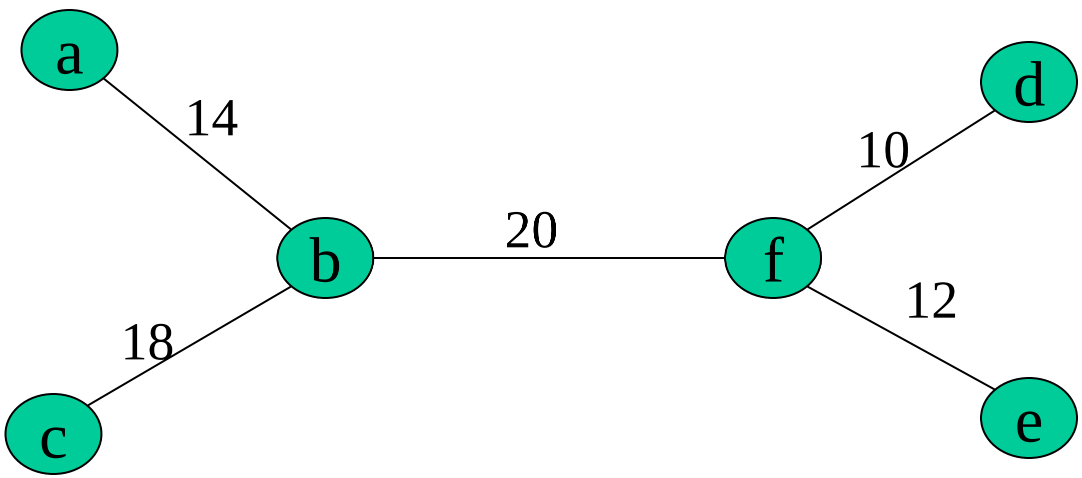
Theorem: The network covering problem is NP-hard.

Proof: In class.



Covering a tree using a minimal number of facilities

When the network is a tree there is a simple algorithm to find an optimal solution to the covering problem.

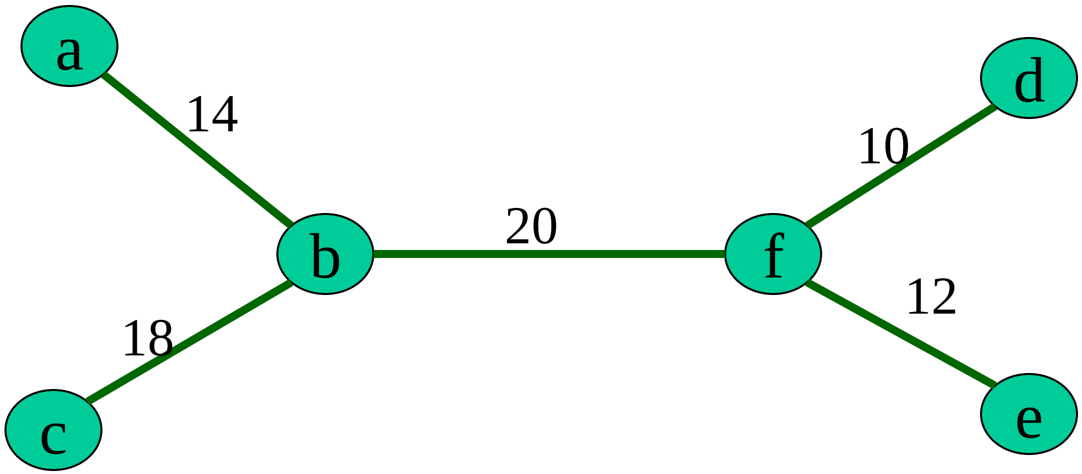


Input: A weighted tree, each vertex i needs to be within some distance s_i from a center. $s_a=10$; $s_b=5$; $s_c=3$; $s_d=14$; $s_e=15$; $s_f=8$

Covering a tree.

Output: location of centers. Centers can be opened anywhere on the tree (also on edges).

Goal: A cover with minimal number of centers.

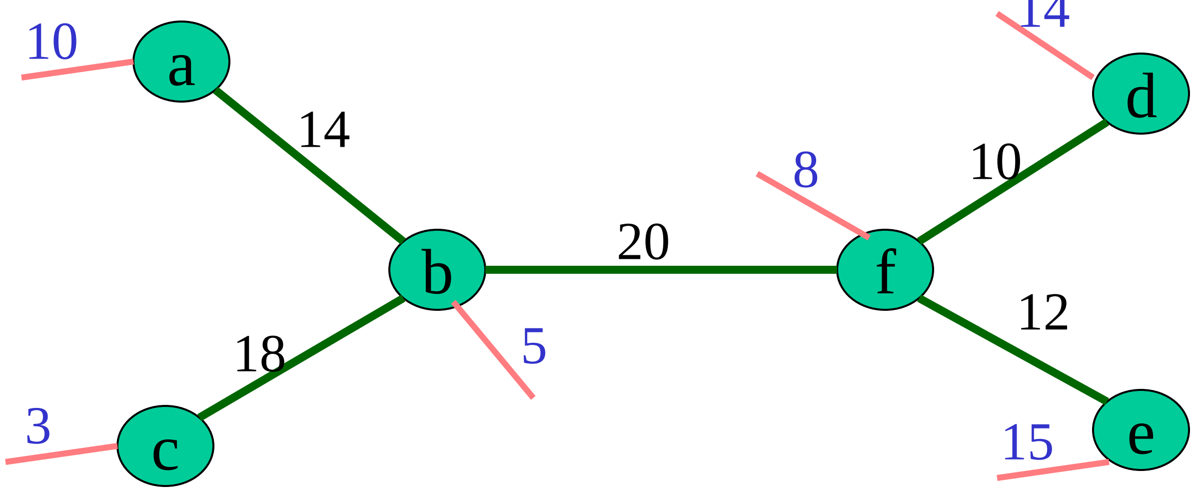


$s_a=10$
$s_b=5$
$s_c=3$
$s_d=14$
$s_e=15$
$s_f=8$

Covering a tree.

Output: location of centers. Centers can be opened anywhere on the tree (also on edges).

Goal: A cover with minimal number of centers.



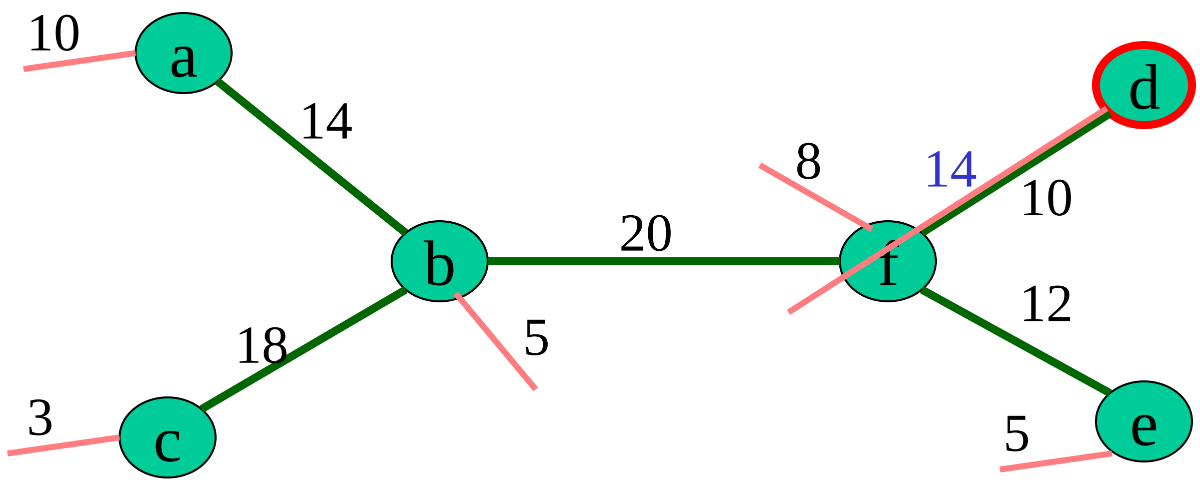
s_a=10
s_b=5
s_c=3
s_d=14
s_e=15
s_f=8

Step 1: attach a "string" of length s_i to vertex i.

Mark all the vertices as non-processed (green).

Covering a tree.

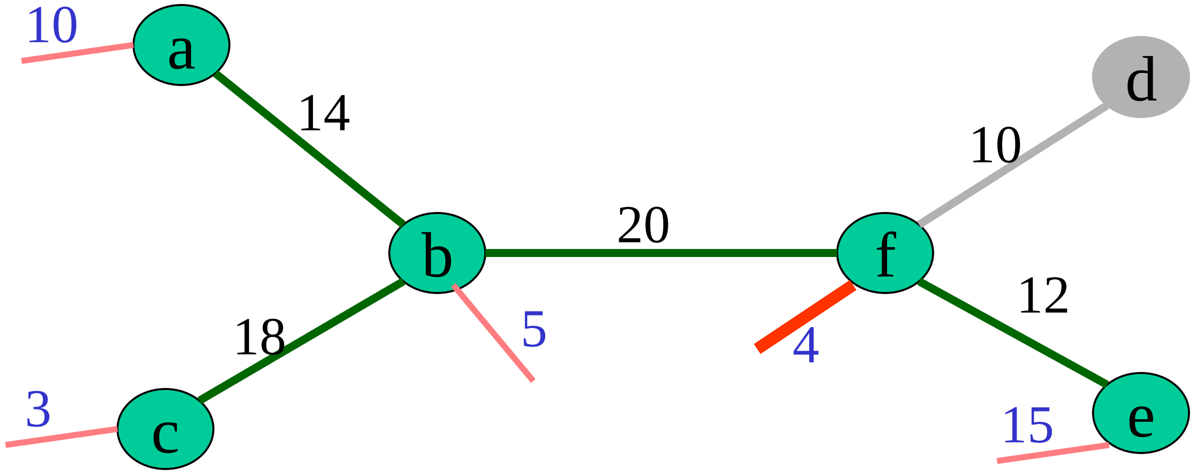
Step 2: pick an arbitrary leaf v, 'stretch' its string towards its neighboring vertex u. If it reaches u, s_u = min (s_u, excess). If it doesn't reach u, add a facility.



s_a=10
s_b=5
s_c=3
s_d=14
s_e=15
s_f=8

Example: select d for active leaf. Stretch the string towards f. Excess=4, update s_f =14-10=4.

Covering a tree.

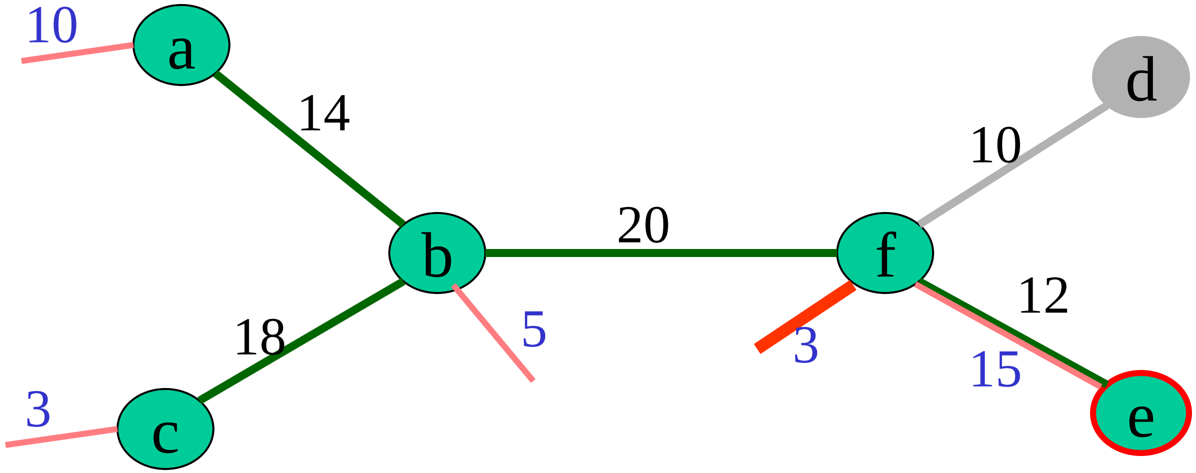


Step 3: remove v and the edge (u,v) from the graph (color them gray).

If the graph is not empty, go to step 2.

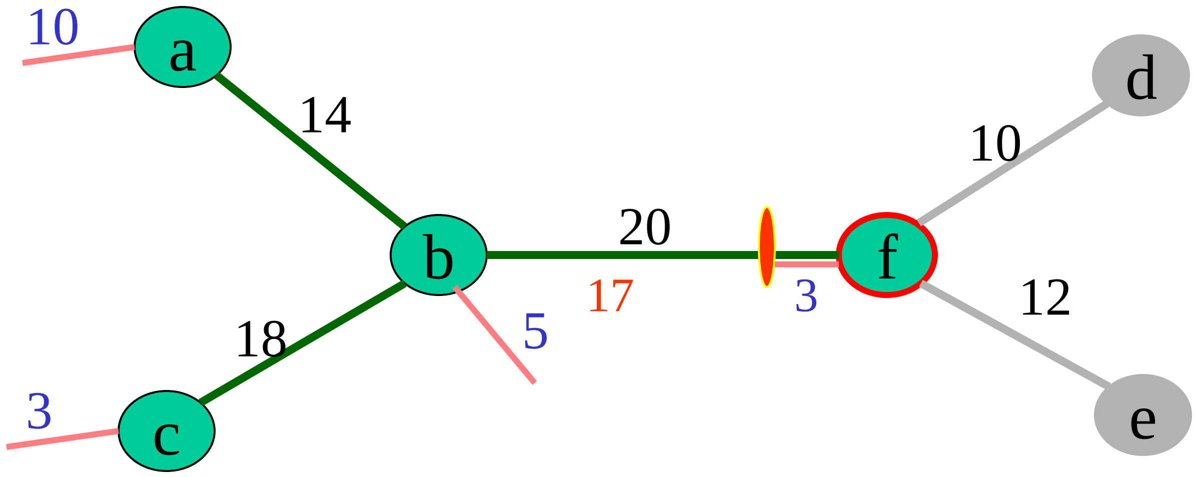
Covering a tree.

v=e, s_e=15, Excess=3



s_f is reduced from 4 to 3

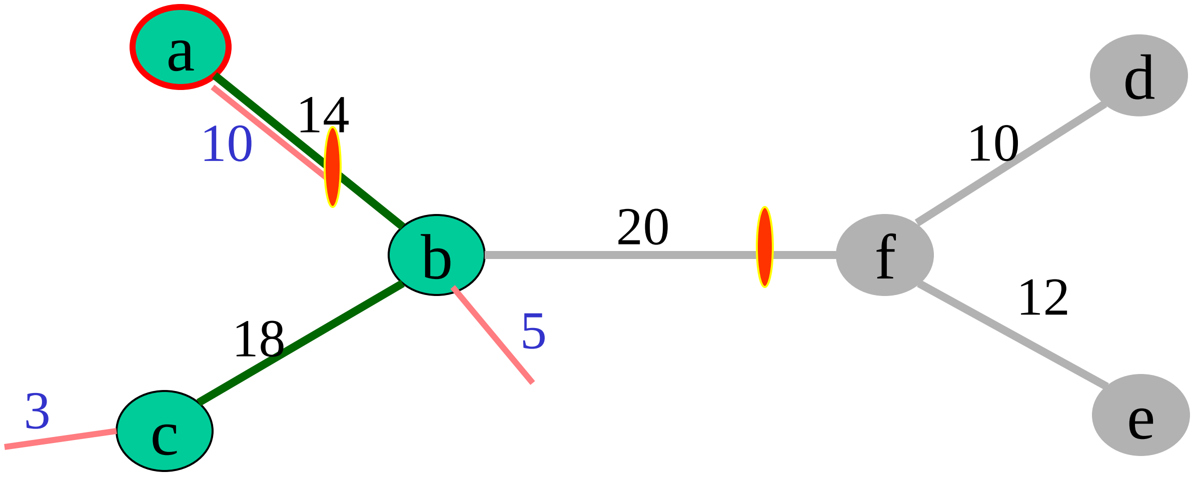
Covering a tree.



v=f. s_f=3, No Excess.

Place a center along f-b. 3 units from f

Covering a tree.

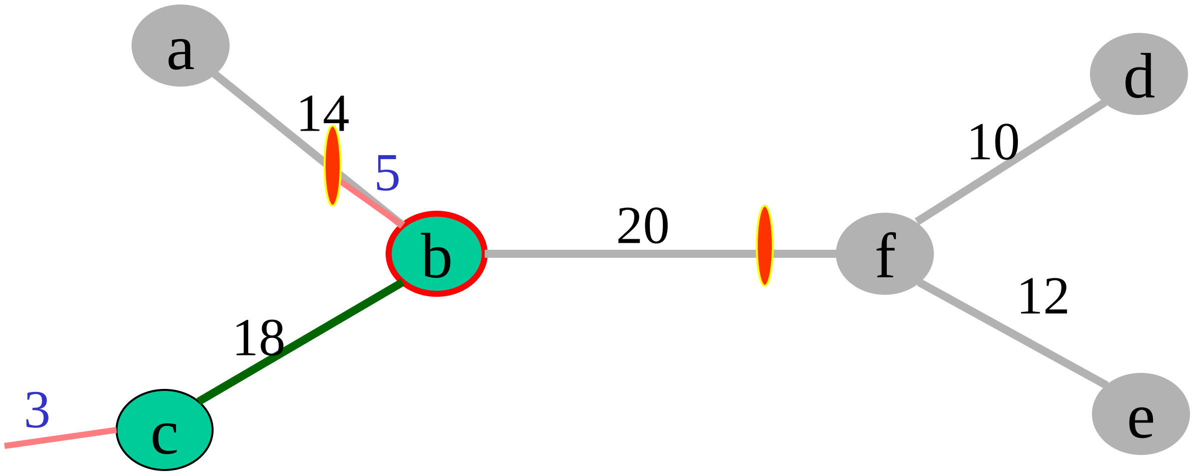


v=a. s_a=10, No Excess.

Check if a is already covered by any center (no)

Place a center along a-b.

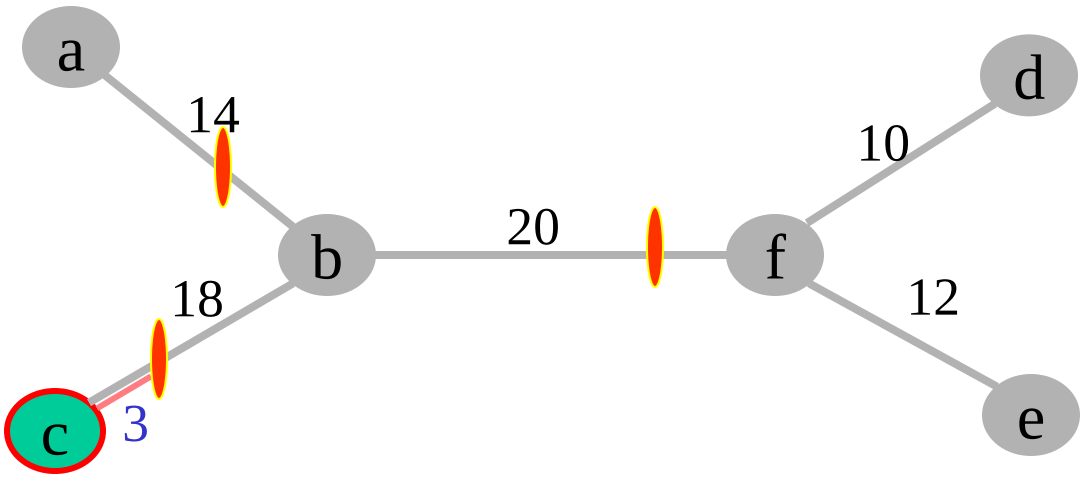
Covering a tree.



v=b. s_b=5, No Excess.

Check if b is already covered by any center (yes!)

Covering a tree.

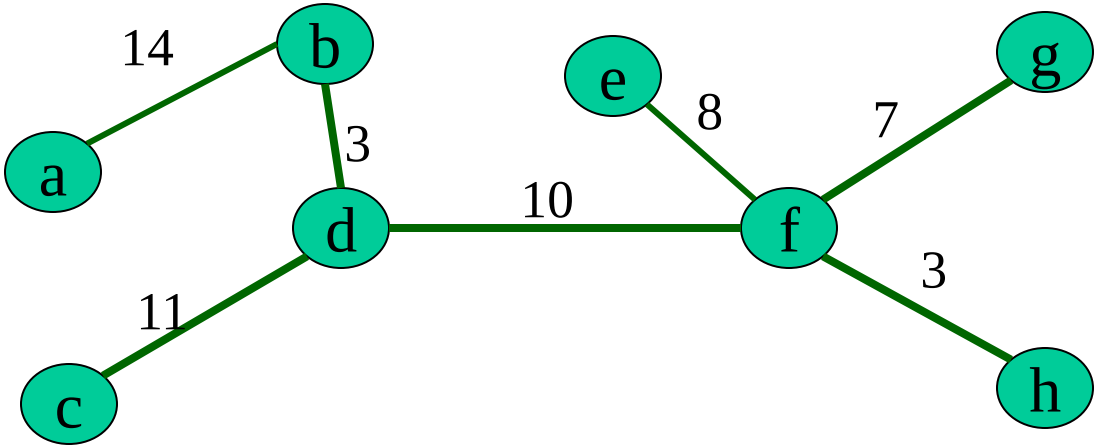


v=c. s_c=3, No active neighbor
Check if c is already covered by any center (no)

can place a center anywhere along (c-b) within distance 3 from c

The whole graph is covered (gray) using 3 centers.

In class exercise: find an optimal covering.



s_a=18, s_b=5, s_c=10, s_d=2,
s_e=5, s_f=4, s_g=10, s_h=6

Covering a tree.

Theorem: The algorithm produces an optimal solution. I.e., it uses the minimal possible number of centers.

Proof: In class.

Partition Problems

The partition problem:

Input: a set of n numbers, $A = \{a_1, a_2, \ldots, a_n\}$, such that $\sum_{a \in A} a = 2B$.

Output: Is there a subset S′ of A such that $\sum_{a \in A'} a = B$?

Example: $A = \{5, 5, 7, 3, 1, 9, 10\}$; $B = 20$

A possible partition:

$A' = \{10, 5, 5\}$, $A - A' = \{7, 3, 1, 9\}$

The Partition Problem is NP-hard.

But what if the numbers are powers of 2?

Solving Partition for power-of 2 Instances.

Input: a set of n numbers, all are of the form 2^c, for some integer c, such that $\sum_{a \in A} a = 2B$.

Output: Is there a subset S' of A such that $\sum_{a \in A'} a = B$?

Example: A={32, 16, 16,8,4,2,2}; B=40
A possible partition:
A'={32,8}, A-A'={16,16,4,2,2}

Solving Partition for power-of 2 Instances.

An Algorithm:

Sort the items such that $a_1 \geq a_2 \geq \ldots \geq a_n$

$S_1 = S_2 = \emptyset$;

$s_1 = s_2 = 0$;

for $i = 1$ to n

if $s_1 > s_2$ add a_i to S_2, $s_2 += a_i$

else add a_i to S_1, $s_1 += a_i$.

if $s_1 = s_2$ output "Partition exists"

else output "No Partition".

Solving Partition for power-of 2 Instances.

Example:

64,32,16,16,4,2,1 – No partition

64,32,16,16,4,2,1,1 – Partition.

Just to make sure, the same method doesn't work for arbitrary instances:

62,34,32,32,16,16,1 – Partition (but not by the algorithm).

Time Complexity: O(n log n) – for sorting

Solving Partition for Power-of 2 Instances- Correctness Proof

Theorem: There is a partition if and only if the algorithm finds one.

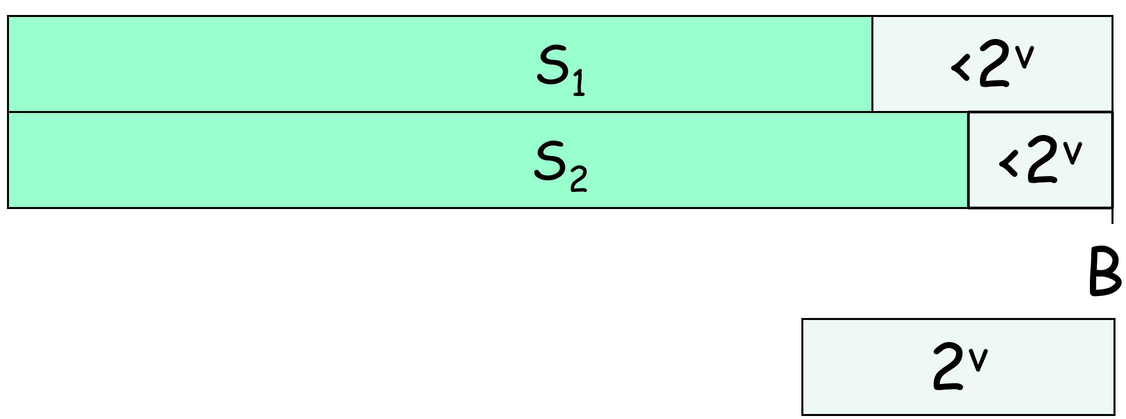
Proof:

1. Clearly, if the algorithm produces a partition, it exists.
2. We prove that if the algorithm does not produce a partition, then a partition does not exist.

Claim (simple property): Let A_1, A_2 be two sets of power-2 integers, such that each integer is $\geq 2^v$. Then $\sum_{a \in A1} a - \sum_{a \in A2} a$ is a multiple of 2^v.

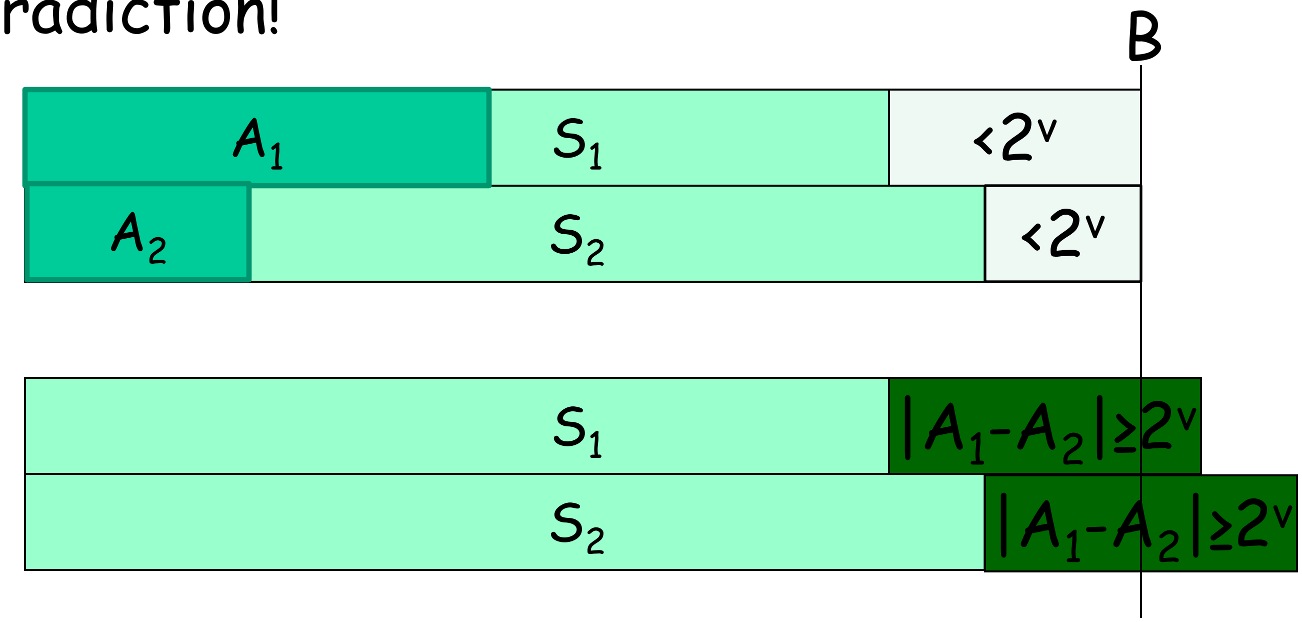
Solving Partition for Power-of 2 Instances- Correctness Proof

Let $\sum_{a \in A} a = 2B$.
Assume that the algorithm does not find a partition.
Then at some point, one set has volume at least B.
Consider the time when a set is about to become larger than B. At this time, some item, of size 2^v, is considered, and the remaining volume in both bins is less than 2^v.



Solving Partition for Power-of 2 Instances- Correctness Proof

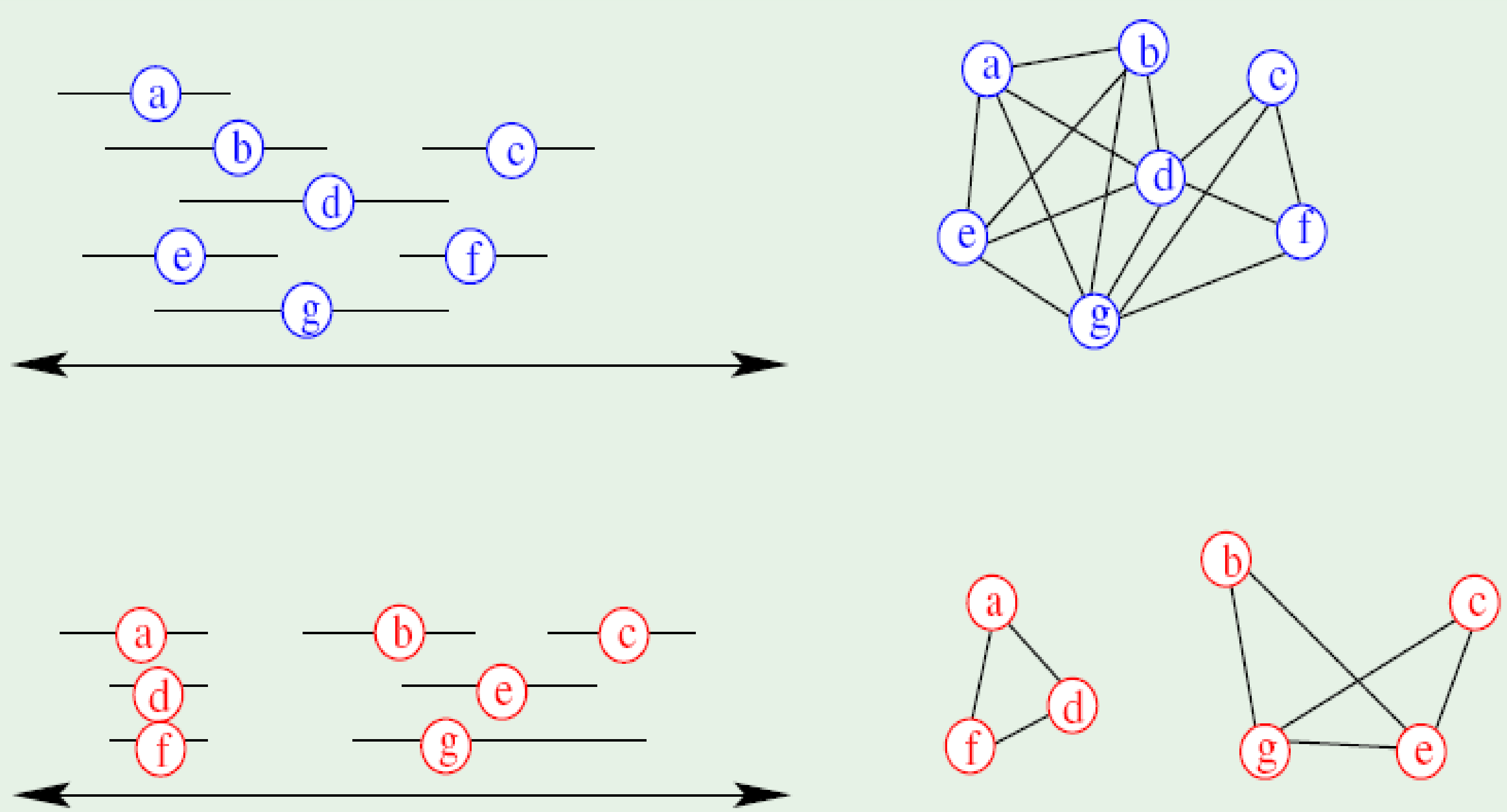
Assume that a partition exists. Then we can exchange subsets $A_1 \subseteq S_1$, $A_2 \subseteq S_2$ to fix the partition produced by the algorithm. Since all integers so far are $\geq 2^v$, The difference $|A_1 - A_2|$ is at least 2^v (it is a non-zero multiple of 2^v). Therefore at least one of the sets overflows.
A contradiction!



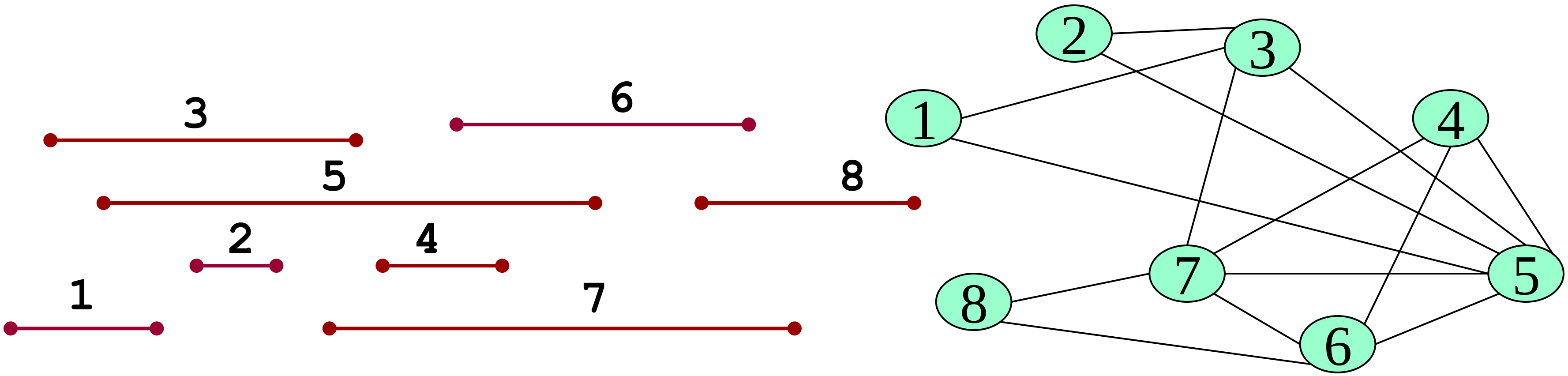
Interval Graphs

- An Interval Graph is the intersection graph of a set of intervals on the real line.

Example Interval Graphs



Interval Graphs



Vertices: Intervals

Edges: between intersecting intervals

Many resource- allocation problems can be modeled as theoretical interval graph problems.

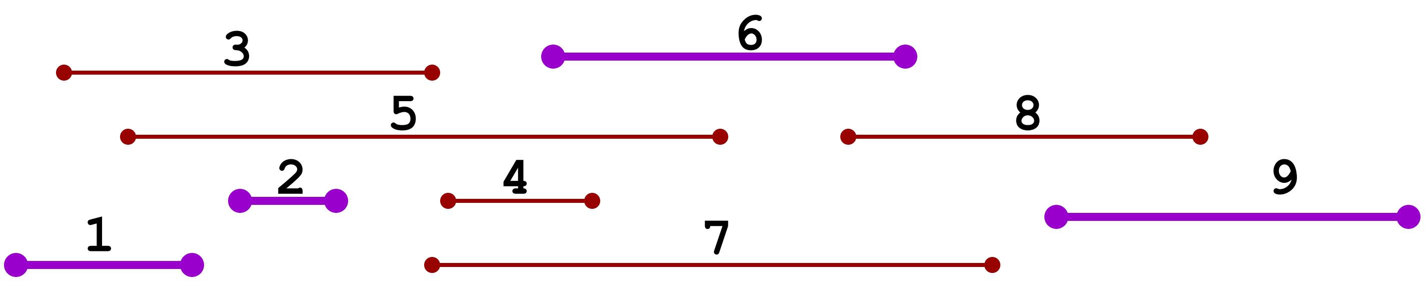
Some Problems that are NP-hard on general graphs can be solved efficiently on interval graphs.

Maximum Independent Set

- Problem: get your money's worth out of a amusement park
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Activity-Selection

- Formally:
  - Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities
    s_i = start time of activity i
    f_i = finish time of activity i
  - Find max-size subset A of non-conflicting activities



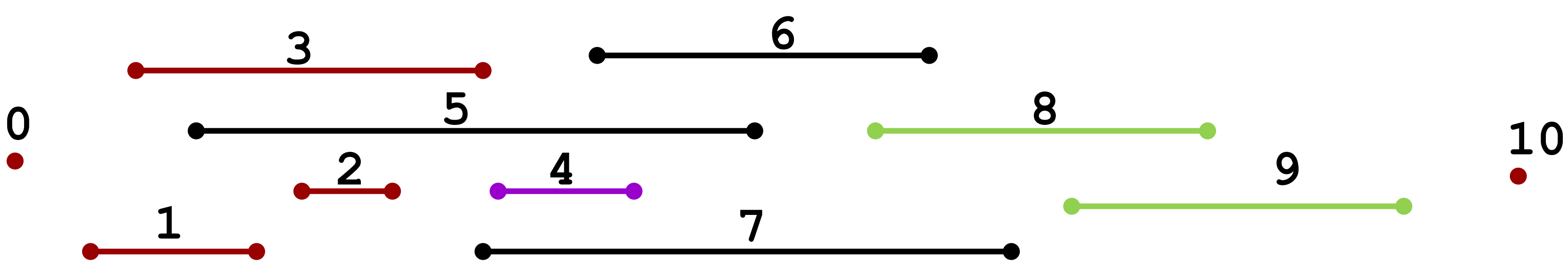
- Assume (w.l.o.g) that $f_1 \leq f_2 \leq \ldots \leq f_n$

Activity-Selection – A DP solution

Try each possible activity k.

Recursively find activities ending before k starts and after k ends.

Turn this into a DP



Activity-Selection – A DP solution

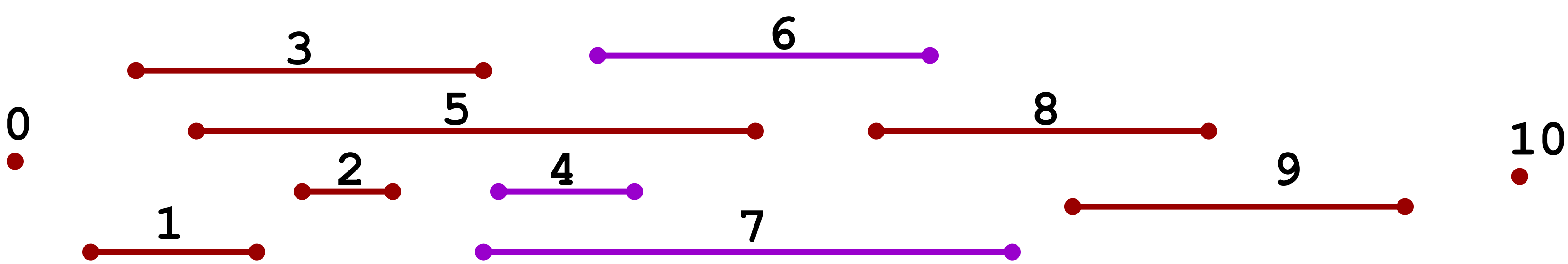
Define:

$S_{ij} = \{k : f_i \leq s_k < f_k \leq s_j\}$

The subset of activities that can start after a_i finishes and finish before a_j starts.

Remark: we add 'dummy activities' a_0 with $f_0=0$

And a_{n+1} with $s_{n+1}=\infty$



Examples: $S_{2,9} = \{4,6,7\}$; $S_{1,8} = \{2,4\}$; $S_{0,10} = S$

Activity-Selection – A DP solution

Define:

C[i,j]= maximum number of activities from S_{ij} that can be selected.

$$C[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

In words: if S_{ij} is not empty, then for any activity k in S_{ij} we check what is the best we can do if k is selected.

Based on this formula we can write a DP whose time complexity is $O(n^3)$

Greedy Choice Property

- The activity selection problem exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal solution
- Theorem: if S is an activity selection instance sorted by finish time, then there exists an optimal solution $A \subseteq S$ such that $\{a_1\} \in A$
- Proof: Given an optimal solution B that does not contain a_1, replace the first activity in B with a_1. The resulting solution is feasible (why?), it has the same number of activities as B, and it includes a_1.

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Time complexity: O(n log n)
- Intuition is even more simple:
  - Always pick the earliest to finish ride available at the time.

Back to MIS in Interval Graphs

Property: Any Interval graph has an interval representation in which all interval endpoints are distinct integers and this representation is computable in poly-time.

Proof: Not Here

Therefore: Activity selection = MIS: Given an instance of MIS in an interval graph:

1. convert it into an interval representation
2. solve the activity selection problem

Note: An independent set in the graph is equivalent to a feasible set of activities.

Graph families

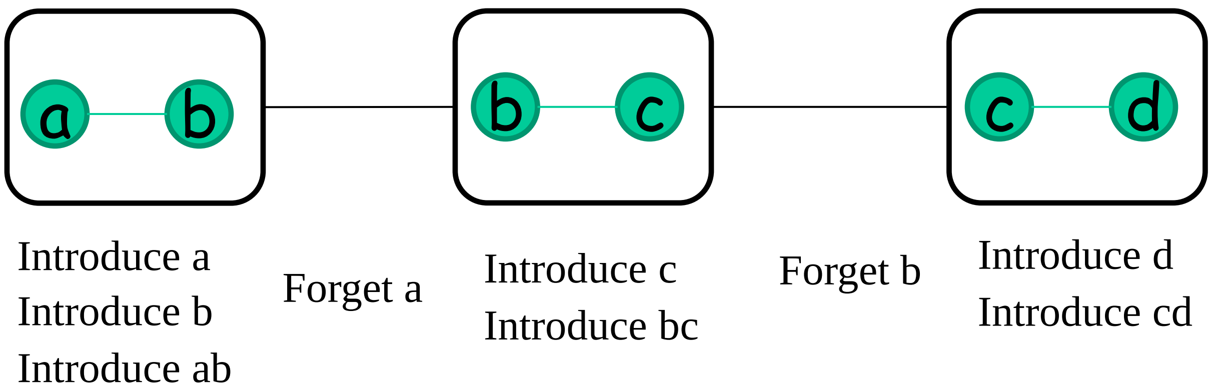
- Trees
- Intersection graphs
- Chordal graphs
- Planar graphs, surface embedded graphs
- Random graphs
- Serial-parallel
- Many many more.....

Generalization - graphs similar to trees

- What does it mean for a graph to be similar to a tree?
- Easier: what does it mean for a graph to be similar to a path?
- Many possible answers. Here is one.

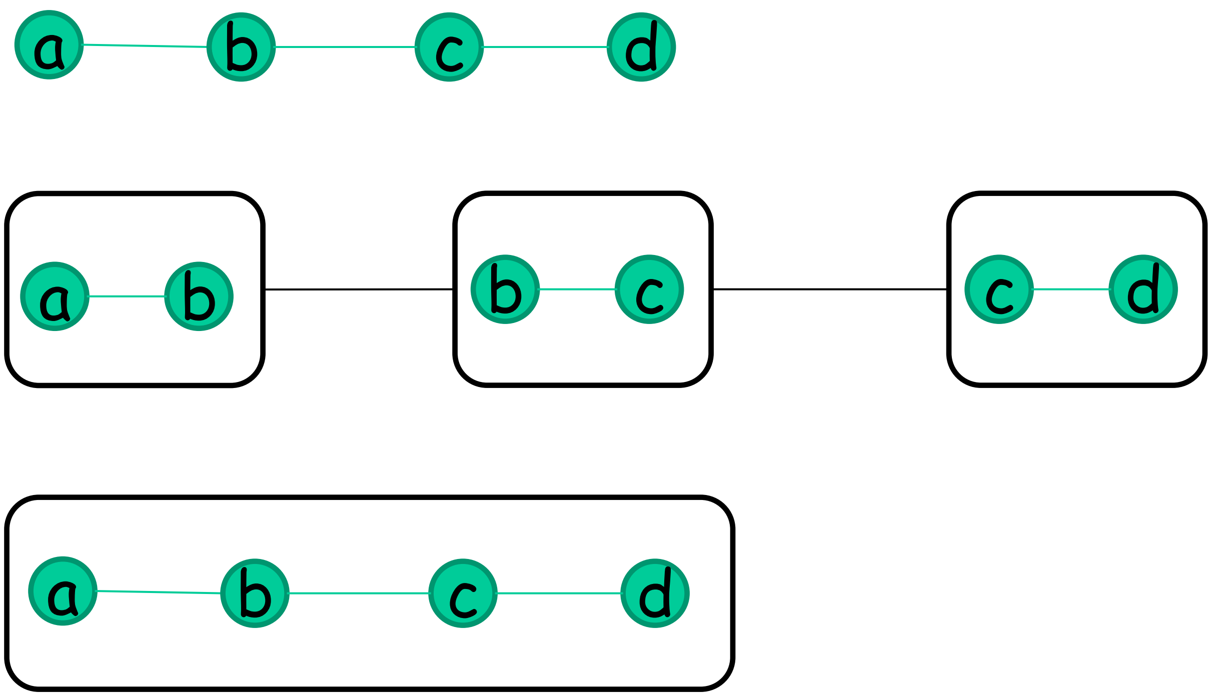
Path decomposition

- We can build a path using the following operations:
- Start with an empty graph
- Introduce a vertex
- Introduce an edge
- Forget a vertex (forever)



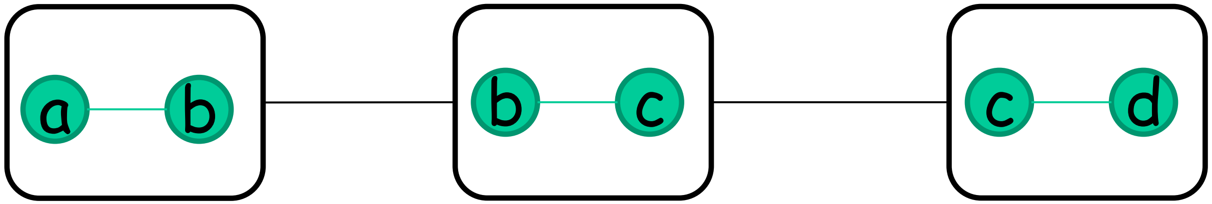
Path decomposition

- This is called a path decomposition
- Two decompositions of the same path:

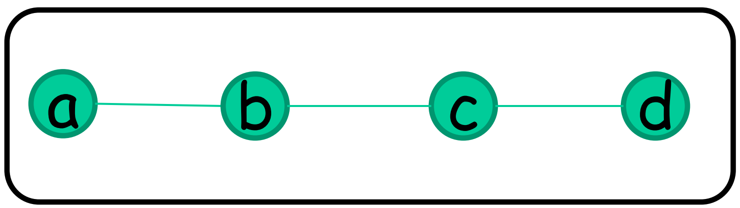


Pathwidth

- The width of the decomposition is defined as one less than the size of the largest bin.



Width 1

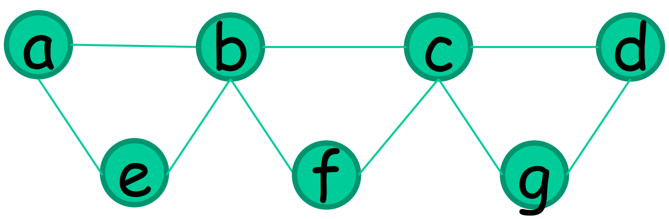


Width 3

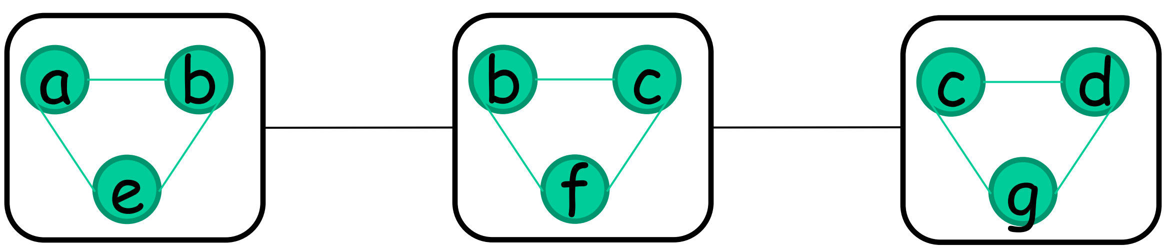
Pathwidth

- The pathwidth of a graph G is the minimum width of a path decomposition of G.

pathwidth 1

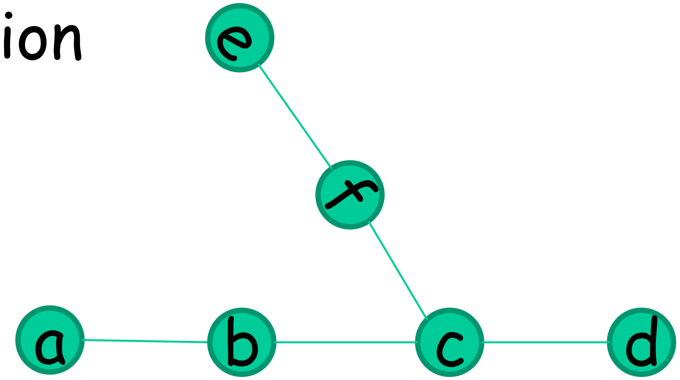


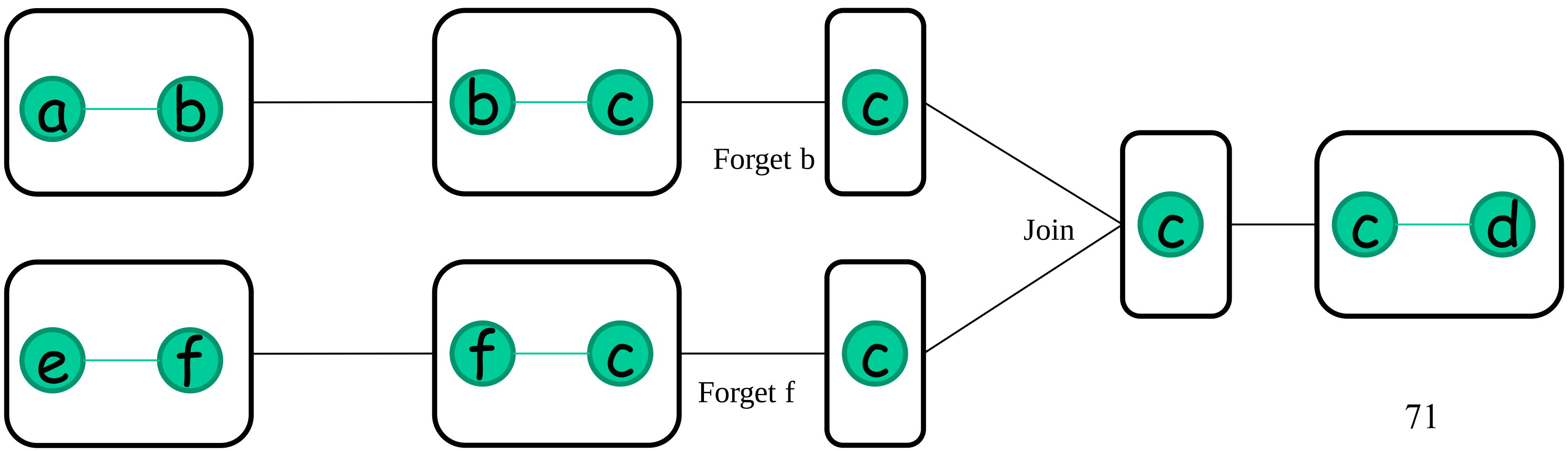
pathwidth 2



Tree decomposition

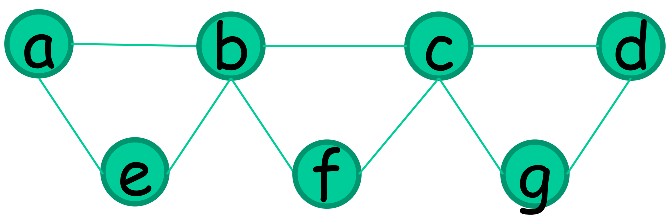
- Same as path decomposition
  - Start with an empty graph
  - Introduce a vertex
  - introduce an edge
  - Forget a vertex
- Also allow:
  - Join two bags together



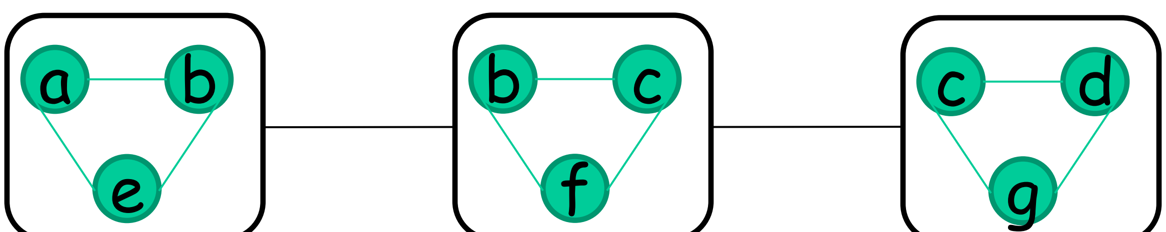


Alternative Definition

- Path decomposition P of G : a path of bags s.t.:
  - Every vertex of G is in some bag.
  - Every edge of G is in some bag.
  - For every vertex v of G, the bags containing v are connected in P.



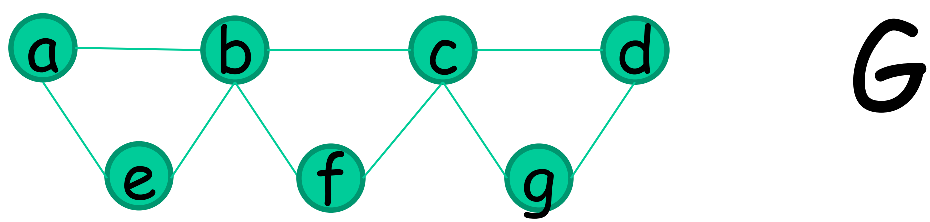
G

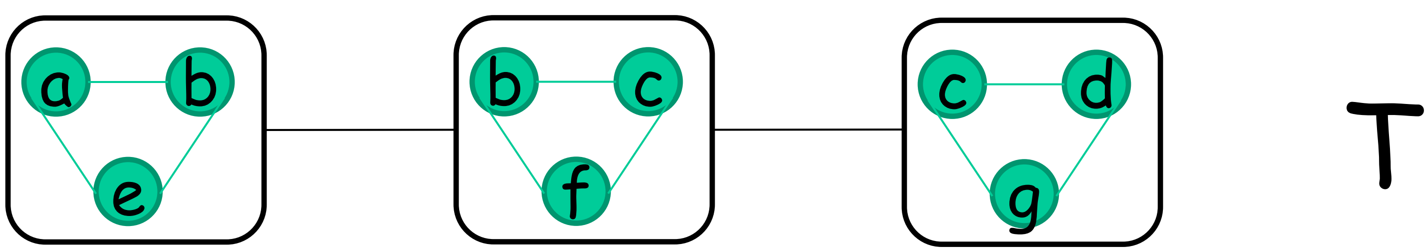


P

Alternative Definition

- Tree decomposition T of G : a tree of bags s.t.:
  - Every vertex of G is in some bag.
  - Every edge of G is in some bag.
  - For every vertex v of G, the bags containing v are connected in P.





Smooth Decompositions

- A width-k tree decomposition T of G is smooth if:
  - Every bag has k+1 vertices
  - The intersection of every pair of adjacent bags has size k

- G has a tree decomposition of width k if and only if G has a smooth tree decomposition of width k (easy proof, not here)

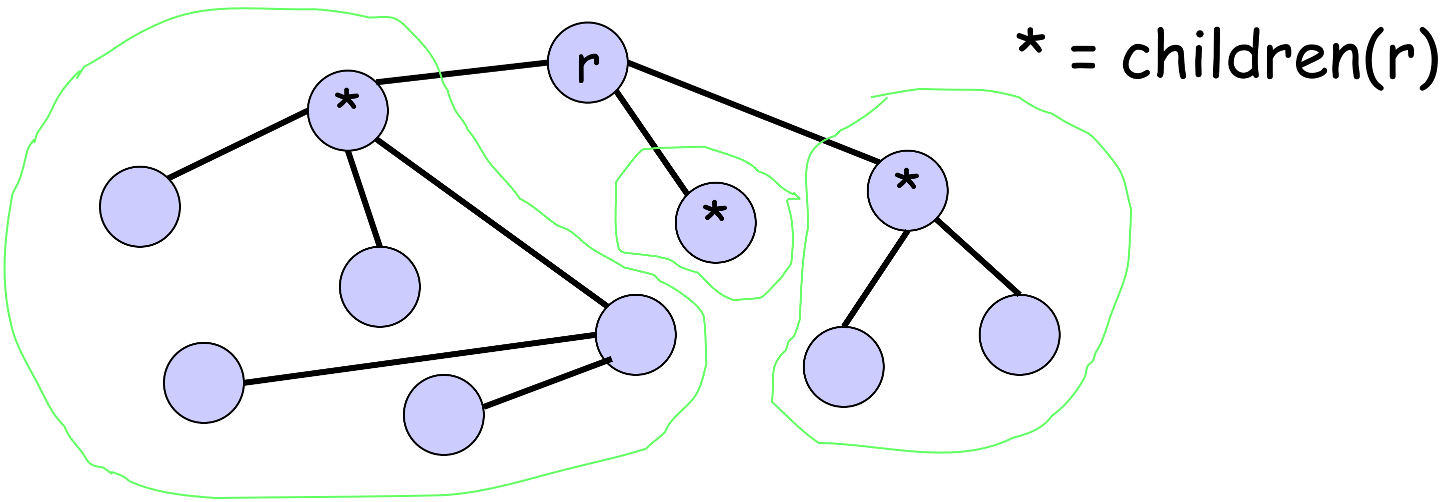
- Easier to work with smooth decompositions

Treewidth

- The treewidth of G is the smallest width of a tree decomposition of G.

- What is the treewidth of a tree?
- What is the treewidth of a clique of size k?

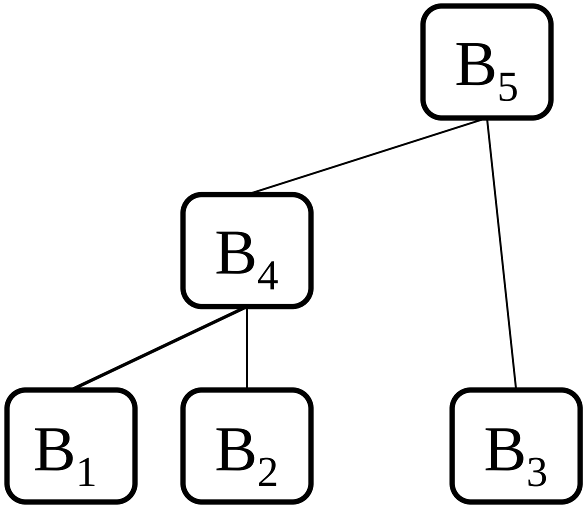
Remember?
Maximum Weighted IS on Trees.



$M_{out}[u]$: The maximum weight of an IS that does not include u in the subtree T_u.

$M_{in}[u]$: The maximum weight of an IS that includes u in the subtree T_u.

Maximum Weighted IS on a tree decomposition



For every bag B and every subset U of the vertices of B:

M[B,U] = size of max. IS in the subgraph induced by all vertices in all bags in T_B such that all vertices of U are in the IS, and vertices in B-U are not in the IS.

Maximum Weighted IS on a graph with small treewidth

To compute M[B,U]:

- For a leaf B:
  M[B,U] = w(U) if U is an IS in B. (-∞ otherwise)

- For internal node B:
- If U is not an IS in B, -∞
- $w(U) + \sum_{B_i \, child \, of \, B} \max_{Y}\{M[B_i, Y]\} - w(U \cap Y)$

where Y is a subset of B_i's vertices that agrees with U (i.e., $Y \cap B = U \cap B_i$)

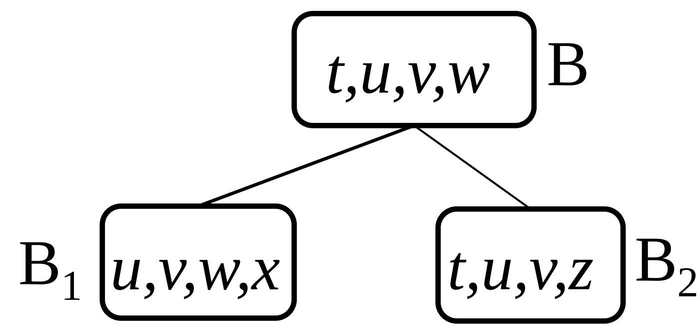
Running time on graph with treewidth k: $O(n \cdot k^2 \cdot 4^k)$

Maximum Weighted IS on a graph with small treewidth

- If U is not an IS in B, $-\infty$
- $w(U) + \sum_{B_i \text{ child of } B} \max_{Y}\{M[B_i, Y]\} - w(U \cap Y)$

  where Y is a subset of B_i's vertices that agrees with U (i.e., $Y \cap B = U \cap B_i$)

Example:



$M[B,\{t,u,w\}] = w(t) + w(u) + w(w)$
$\quad + \max\{M[B_1,\{u,w\}], M[B_1,\{u,w,x\}]\} - w(u) - w(w)$
$\quad + \max\{M[B_2,\{t,u\}], M[B_1,\{t,u,z\}]\} - w(t) - w(u)$

Treewidth

- Many NP-hard problems can be solved by DP on a tree decomposition in polynomial time in n, but exponential in treewidth.
- Computing the treewidth of a graph is NP-copmlete (also pathwidth).
- $O(\sqrt{\log n})$-approximation exists.
- Some hard problems are still hard on graphs with small treewidth.
- There are many similar notions of width: branch-width, carving-width, clique-width.

Parametrized Complexity

If L is NP-hard then there is no algorithm which solves all instances of L in polynomial time.

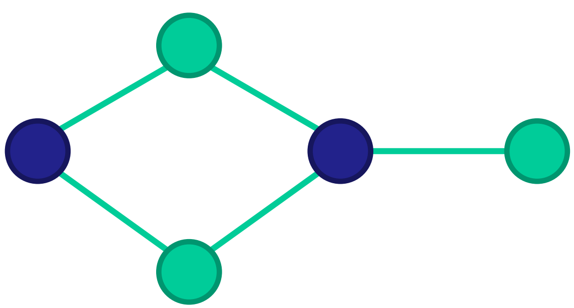
What about the easy instances?

How do we capture easy?

Based on slides by Daniel Lokshtanov

Example: Vertex Cover

Input: G, k
Question: $\exists S \subseteq V(G), |S| \leq k$ such that every edge in G has an endpoint in S?



Algorithms for Vertex Cover

How fast can you solve VC?

Naive: $O(n^k m)$.

Can we do it in linear time for $k=10$?
Or linear time for any fixed integer k?

Pre-processing for Vertex Cover

If any vertex v has degree $\geq$ k+1 it must be a part of any vertex cover of size $\leq$ k

→ Pick it in to solution.

→ Remove v and decrease k by 1.

Pre-processing

If no vertices of degree $\geq k+1$ and $\geq k^2$ edges left say NO.

k^2 edges left. Remove vertices of degree 0, then there are $2k^2$ vertices left.

In linear time, we made $n \leq 2k^2$ and $m \leq k^2$.

Brute force now takes time $O((2k^2)^k k^2)$

Running time

Total running time is: $O(n+m + (2k^2)^k k)$

Linear for any fixed k ☺

Pretty slow even for k = 10 ☹

Parameterized Complexity

Every instance comes with a parameter k.

Often k is solution size, but could be many other things

The problem is fixed parameter tractable (FPT) if exists algorithm with running time $f(k)n^c$.

So Vertex Cover parameterized by solution size is fixed parameter tractable.

Kernelization

For vertex cover we first reduced the instance to size $O(k^2)$ in polynomial time, then we solved the instance.

Let's give this approach a name – kernelization.

Kernels

A f(k)-kernel is a polynomial time algorithm that takes an instance I with parameter k and outputs an equivalent instance I' with parameter k' such that:

$|I'| \leq f(k)$

$k' \leq f(k)$ (but typically $k' \leq k$)

Kernelizable = FPT

A problem Π is solvable in $f(k)n^c$ time for some f.

$\Leftrightarrow$

Π is decidable and has a $g(k)$ kernel for some g.

$\leftarrow$ Kernelize in n^c and solve in time that depends only on k.

$\rightarrow$ If $n \le f(k)$, done (problem size already $\le f(k)$).
If $n \ge f(k)$ solve in time $f(k)n^c = O(n^{c+1})$ and output a fixed size equivalent instance.

Kernel: Point-line cover

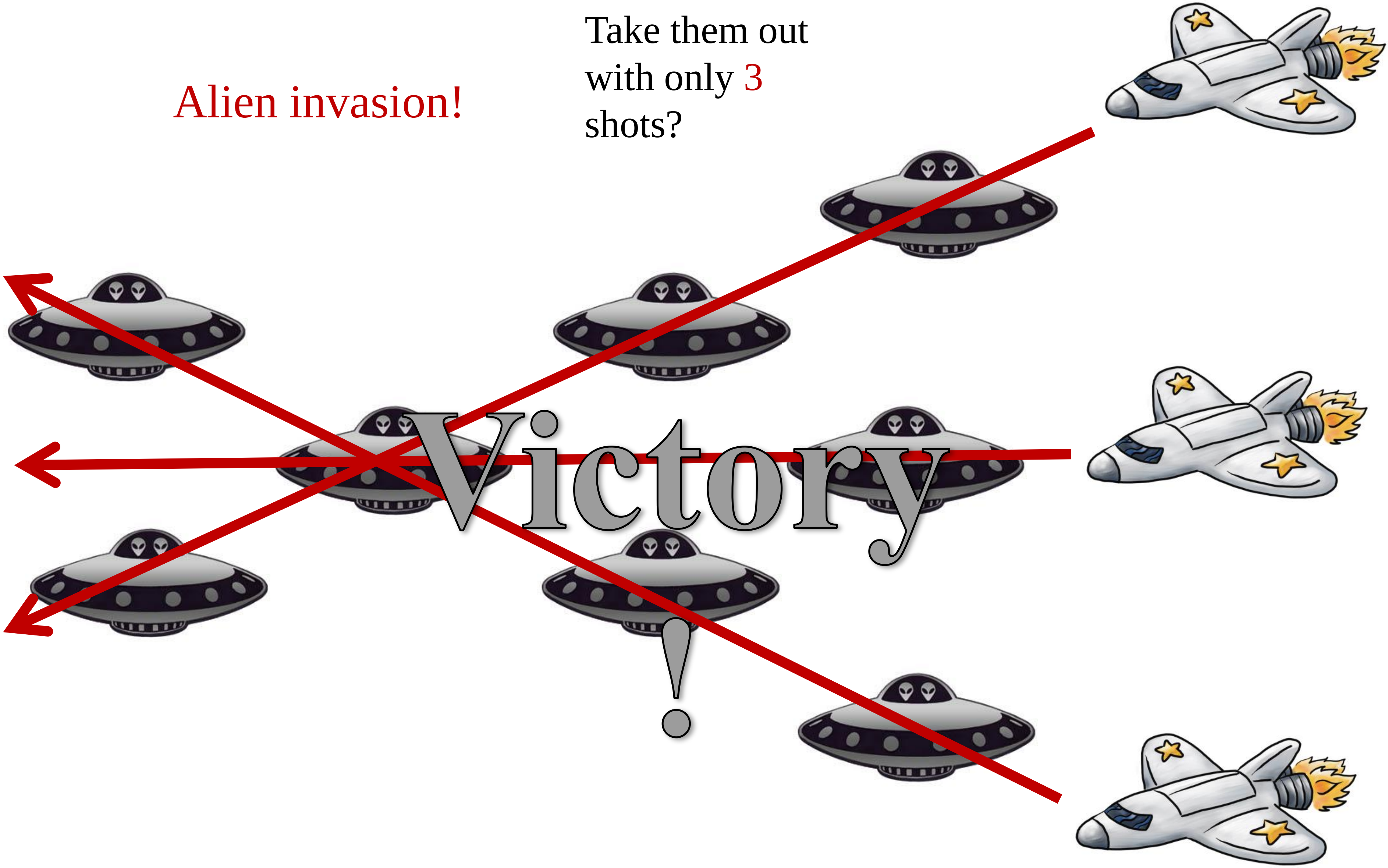
Input: n points in the plane, integer k

Question: Can you hit all the points with k straight lines?

Fact: Point-Line cover is NP-Complete.

Alien invasion!

Take them out with only 3 shots?



Reduction rules

R1: If some line covers $k+1$ points use it (and reduce k by one). (why?)

R2: If no line covers n/k points, say NO.

If neither R1 nor R2 can be applied then $n \leq k^2$.

Kernel with k^2 points!

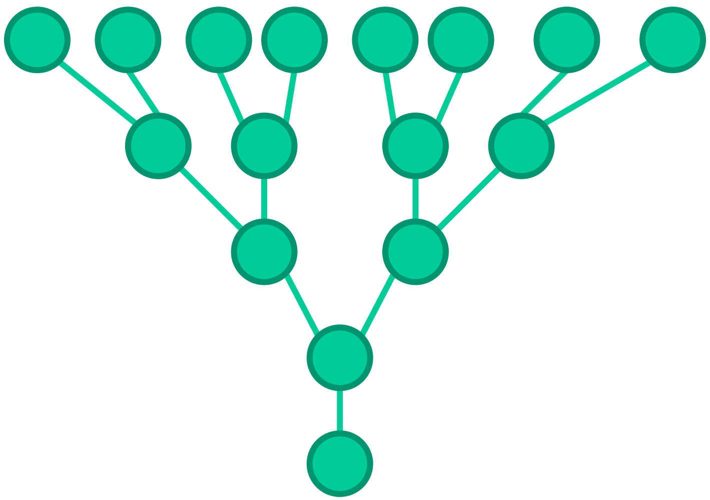
Kernelization

Initially thought of as a technique for designing FPT algorithms.

Interesting in its own right, because it allows us to analyze polynomial time pre-processing.

Branching

A simple and powerful technique for designing FPT algorithms.



Vertex Cover (again)

Let $uv \in E(G)$.

At least one of u and v must be in the solution.

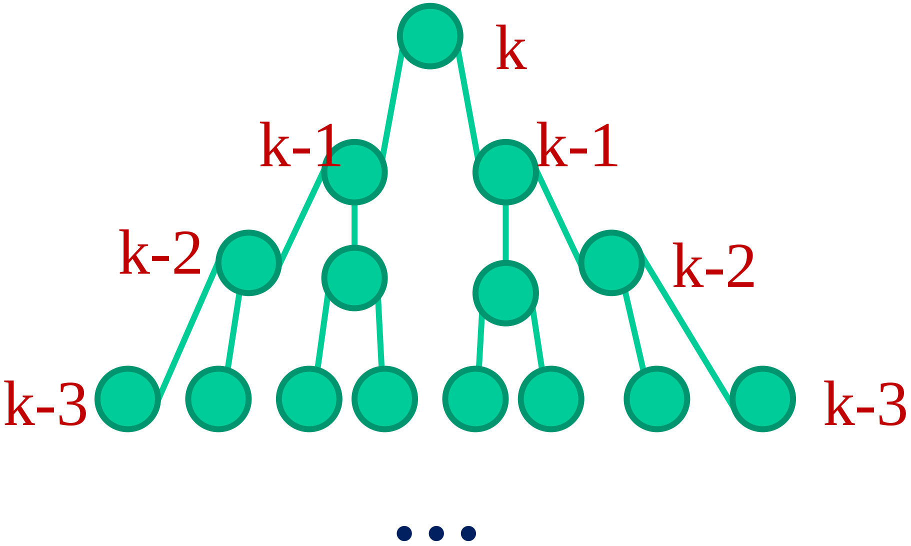
G has a vertex cover of size $\leq k$

$\leftrightarrow$

$G \backslash u$ has a vertex cover of size $\leq k$-1 OR $G \backslash v$ has a vertex cover of size $\leq k$-1

Recursive algorithm!

Running time



Total running time is $O(2^k n^c)$

$O(n + m + 2^k k^{2c})$ if we run kernel first.

3-Hitting Set

Input: Family $S_1 \ldots S_m$ of sets of size 3 over a universe $U = v_1 \ldots v_n$, integer k.

Question: Is there a set $X \subseteq U$ such that $|X| \leq k$ and every set S_i intersects with X?

Parameter: k

Branching for 3-Hitting Set

Pick a set $S_i = \{v_a, v_b, v_c\}$.

At least one of them must be in the solution X.

Branch on which one, decrease k by one.

Remove all sets that are hit.

Total running time: $O(3^k \cdot (n+m))$

Even Better Branching for Vertex Cover

(Going below 2^k)

If all vertices have degree ≤ 2 then

G is a set of paths and cycles,

so we can solve Vertex Cover in polynomial time.

Even Better Branching

Let $v \in V(G)$, degree(v) ≥ 3.

If v is not in the solution, then N(v) is.

G has a vertex cover of size $\leq$ k

$\leftrightarrow$

G\v has a vertex cover of size $\leq$ k-1

OR

G\N(v) has a vertex cover of size $\leq$ k-degree(v)

Recursive algorithm!

Running time

$T(n, k) =$ Running time on a graph on at most n vertices and parameter at most k.

$N(k) =$ Number of *nodes* in a recursion tree if parameter is at most k.

$L(k) =$ Number of *leaves* in a recursion tree if parameter is at most k.

$$T(n, k) = O(N(k) \cdot (n+m)) = O(L(k) \cdot (n+m))$$

Recurrence

$$L(k) \leq \begin{cases} L(k-1) + L(k-3) & \text{If exists vertex of degree } \geq 3. \\ 1 & \text{otherwise.} \end{cases}$$

Will prove $L(k) \leq 1.47^k$ by induction.

$$\begin{aligned} L(k) &\leq L(k-1) + L(k-3) && \text{(recurrence)} \\ &\leq 1.47^{k-1} + 1.47^{k-3} && \text{(induction hypothesis)} \\ &\leq 1.47^k \cdot (1.47^{-1} + 1.47^{-3}) \\ &\leq 1.47^k && \text{(choice of 1.47)} \end{aligned}$$

Running time analysis

Number of leaves in the recursion tree is at most 1.47^k, so total running time is $O(1.47^k(n+m))$.

Fastest known algorithm for Vertex Cover has running time $\approx 1.27^k$ [Chen, Kanj, Xia, 2010].

Graphs with k=400 can be solved in practice using FPT branching techniques [Cheetham et al., 2003]

Alternative Parameters

So far we have only seen the solution size as the parameter.

Often other parameters also make sense, or even make more sense than solution size.

k-Coloring

A valid k-coloring is a funcion $f : V(G) \rightarrow \{1...k\}$ such that no edge has same colored endpoints.

Input: G, k

Question: Does G have a valid k-coloring?

Parameter: k

Cannot have FPT algorithm – NP-hard for $k=3$!

k-Coloring parameterized by VC

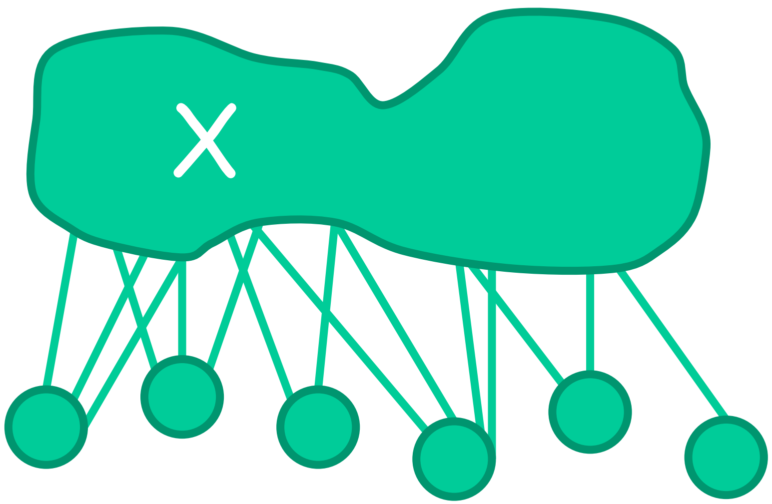
Input: G, integer k, set $X \subseteq V(G)$ such that X is a vertex cover of G, integer $x = |X|$.

Question: Does G have a proper k-coloring?

Parameter: x

FPT now means $f(x)n^{O(1)}$.

k-Coloring parameterized by VC



$I = V(G) \setminus X$

If $x+1 \leq k$ say YES

Thus, assume $k \leq x$.

Branch on k^x colorings of X.

For each guess, color I greedily.

Total running time: $O(k^x \cdot (n+m)) = O(x^x \cdot (n+m))$.

Dynamic Programming

Steiner Tree

Input: Graph G, vertex set Q, integer k.

Question: Is there a set S of size at most k such that $Q \subseteq S$ and $G[S]$ is connected?

Parameter: $|Q|$

Will see $3^{|Q|}n^{O(1)}$ time algorithm.

DP for Steiner Tree

Vertex

$\leq k$, solution size

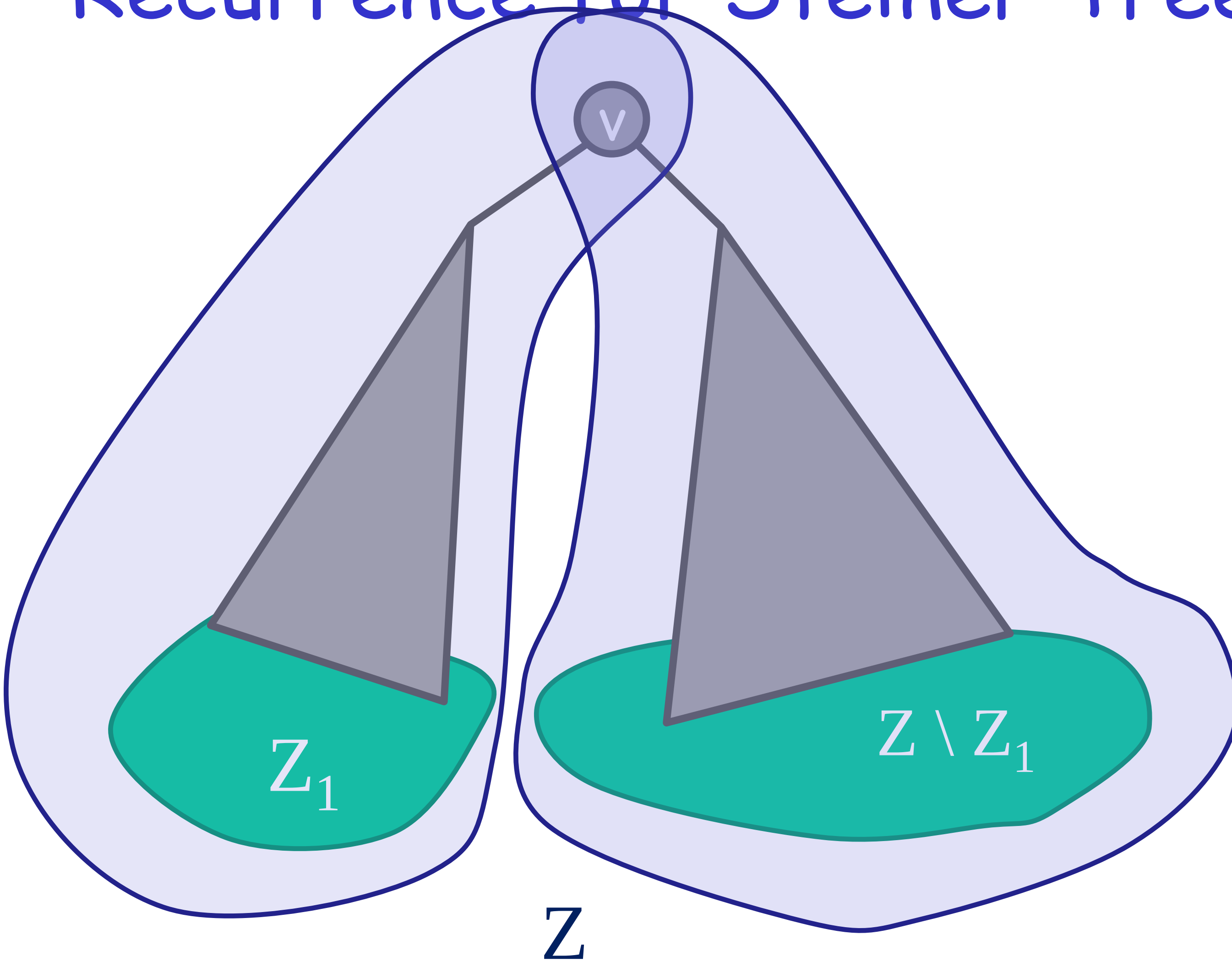
$T[v,p,Z] =$ **True** if there exists a set S of size at most p such that that $Z \cup \{v\} \subseteq S$ and $G[S]$ is connected.

$\subseteq Q$

Table size is $2^{|Q|}kn$

We want to know the minimum p such that $T[v,p,Q] = \text{true}$, for some $v \in V(G)$

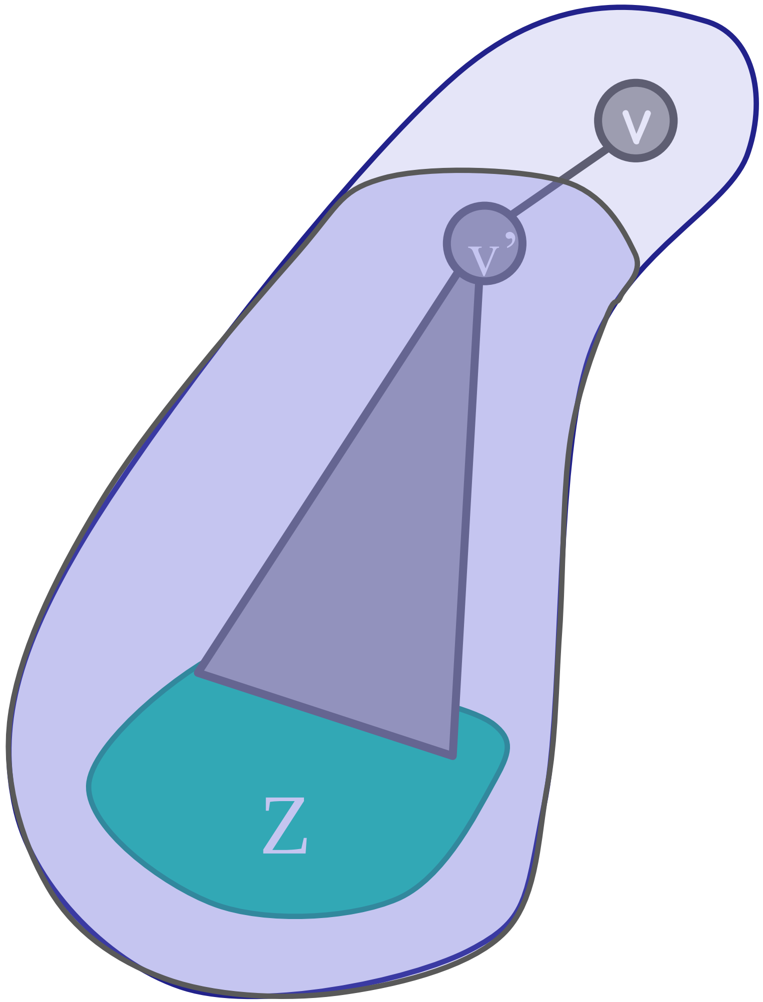
Recurrence for Steiner Tree



Recurrence for Steiner Tree

$$T[v,p,Z] = \bigvee_{1<p_1<p} \bigvee_{\emptyset\subset Z_1\subset Z} \begin{array}{c} T[v,p_1,Z_1] \\ + T[v,p-p_1+1,Z \setminus Z_1] \end{array}$$

Recurrence for Steiner Tree



Recurrence for Steiner Tree

$$T[v,p,Z] = \bigvee_{1<p_1<p} \bigvee_{\emptyset \subset Z_1 \subset Z} \begin{array}{c} T[v,p_1,Z_1] \\ + T[v,p-p_1+1,Z \setminus Z_1] \end{array}$$

$$\bigvee_{u \in N(v)} T[u,p-1,Z]$$

Steiner Tree, Analysis

Table size: $2^{|Q|}nk$

Time to fill one entry: $O(k2^{|Q|} + n)$

Total time: $O(4^{|Q|}nk^2 + 2^{|Q|}n^2k)$

Independent Set

Is Independent Set FPT? With what parameter?

Yes, we saw a $O(4^k \cdot n)$ time algorithm, k = treewidth.

Many other problems are FPT in treewidth.