Tackle NP-Hard Problems (Ch 10)

- **Strategies for tackling NP-hard problems**
- **Special cases: trees**
- **Special cases: planarity**
- **Exponential time algorithms**
- **Fixed parameter algorithms**
- **Approximation algorithms for metric TSP**
- **Graph decompositions and algorithms**

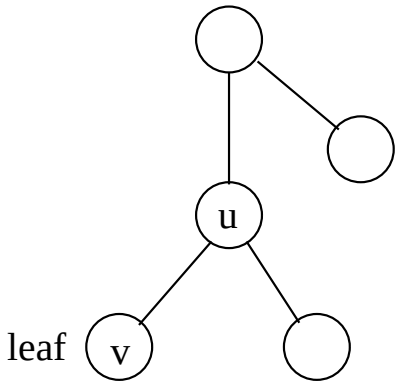
Strategies for Tackling NP-Hard Problems

- **To solve an NP-hard problem, we need to sacrifice one of following:**
  - **Find an optimal solution of the problem.**
  - **Find a solution for every instance of the problem.**
  - **Find a solution of the problem in polynomial time.**
- **Strategies for coping NP-hard problems**
  - **Design algorithms to find approximate solutions.**
  - **Design algorithms for special cases of the problems.**
  - **Design algorithms which may take exponential time.**

Special Cases: Trees

- **Maximum independent set (Max-IS): find a maximum cardinality subset of nodes in a graph that no two nodes in the subset are adjacent.**
- **Max-IS in trees: Given a tree T, find a Max-IS of T.**
- **For any leaf node v of T, there is a Max-IS containing v.**

Proof. Let S be a Max-IS of T. If $v \in S$, then the proof is done. Otherwise, let $\{u, v\}$ be the edge incident to v. If $u \notin S$, then $S \cup \{v\}$ is an independent set, implying S is not maximum, contradiction. For $u \in S$, $(S \setminus \{u\}) \cup \{v\}$ is a Max-IS. □



A greedy algorithm for Max-IS in trees

Max-IS-Tree(T)

Input: A tree T.

Output: A Max-IS S in T.

$S = \emptyset$;

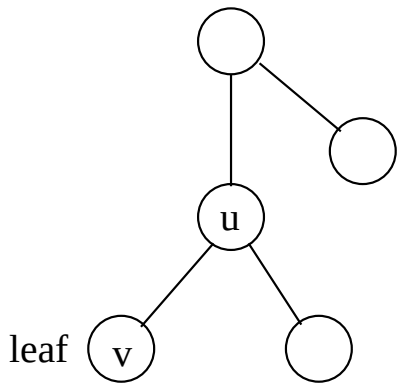
while T has an edge do

Let v be a leaf node of T and $\{u, v\}$ be the edge incident to v;

$S = S \cup \{v\}$; **Remove nodes u, v from T;**

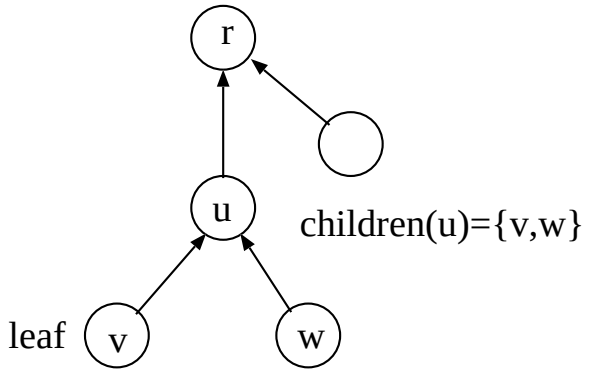
end while

Return $S \cup \{$nodes remaining in $T\}$; Max-IS-Tree algorithm finds a Max-IS of T in $O(n)$ time.



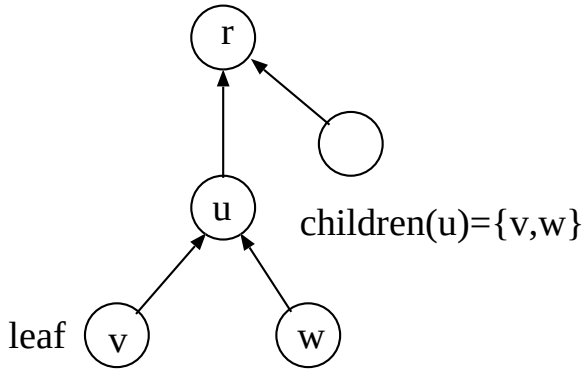
- **Max-weight IS: Given a node weighted graph G (each node v has a weight $w(v)$), find an independent set S of G s.t. $\sum_{v \in S} w(v)$ is maixmized.**
- **Max-weight IS on trees: find a Max-weight IS of a node weighted tree T.**
- **A dynamic programming algorithm: Root T at a node r.**

  Optimal solution structure: for each node u,
  - $\mathrm{opt}_{in}(u)$**: weight of max-weight IS in subtree rooted at u, containing u.** $\mathrm{opt}_{in}(u)$ **has $w(u)$ and $\mathrm{opt}_{out}(v)$ for every child v of u.**
  - $\mathrm{opt}_{out}(u)$**: weight of max-weight IS in subtree rooted at u, not containing u.** $\mathrm{opt}_{out}(u)$ **has $\max\{\mathrm{opt}_{in}(v), \mathrm{opt}_{out}(v)\}$ for every child v of u.**
  - **Goal: maximize $\{\mathrm{opt}_{in}(u), \mathrm{opt}_{out}(u)\}$.**



- **Bellman equations:**

$$\begin{aligned}
\mathrm{opt}_{in}(u) &= w(u) + \sum_{v \textbf{ a child of } u} \mathrm{opt}_{out}(v) \\
\mathrm{opt}_{out}(u) &= \sum_{v \textbf{ a child of } u} \max\{\mathrm{opt}_{in}(v), \mathrm{opt}_{out}(v)\}
\end{aligned}$$



A dynamic programming algorithm for Max-weight IS in trees.

Max-Weight-IS-Tree(T)

Input: A node weighted tree T.

Output: Weight of Max-weight IS S in T.

for each node u of T in postorder do

if u is a leaf then $M_{in}[u] = w(u)$; $M_{out}[u] = 0$

else $M_{in}[u] = w(u) + \sum_v$ a child of u $\mathrm{opt}_{out}(v)$;

$M_{out}[u] = \sum_v$ **a child of** u $\max\{\mathrm{opt}_{in}(v), \mathrm{opt}_{out}(v)\}$

end if

end for

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

Max-Weight-IS-Tree algorithm finds the weight of a Max-weight IS of T in $O(n)$ time.

Solving NP-hard problems in graphs

- **A well used approach for solving a problem P in graph G is**

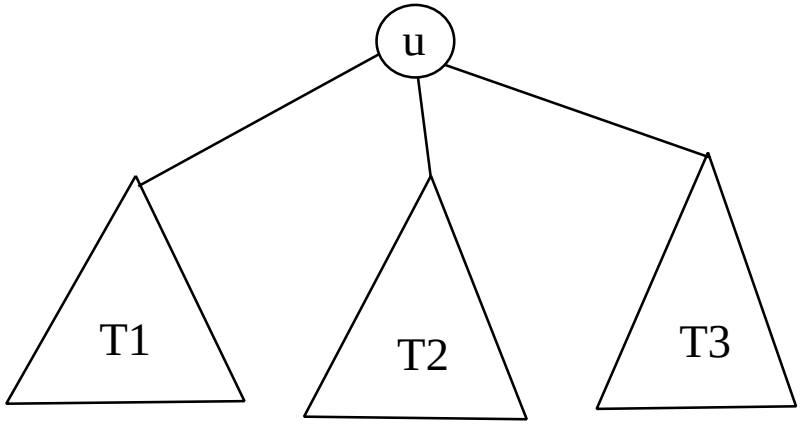
  (1) divide G into subgraphs by a node/edge cut set C,

  (2) solve subproblems in subgraphs and

  (3) combine solution to subproblems into a solution to P.

- **For an NP-hard problem P, Step (3) may run in $O(2^{\mathrm{Poly}(|C|)})$ time.**

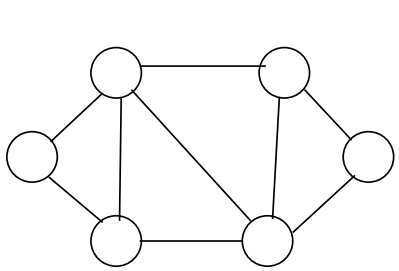
- **For a tree, a cut set C with $|C| = 1$ may be easily found and the above approach gives an efficient algorithm.**



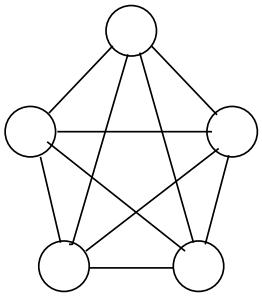
Special Cases: Planarity

- **Planar graph: a graph G can be drawn on a plane s.t. no two edges cross with each other.**
- **Graphs K_5 (complete graph of 5 nodes) and $K_{3,3}$ (complete bipartite graph with 3 nodes in each of the bipatition sets) are nonplanar.**
- **If G has a subgraph of K_5 or $K_{3,3}$, then G is nonplanar.**

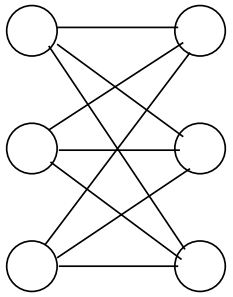
  G is planar iff G does not have a minor of K_5 or $K_{3,3}$.
- **Whether G is planar or not can be decided in $O(n)$ time.**



Planar



K_5



$K_{3,3}$

- **Many problems in graphs can be solved faster in planar graphs**

  Examples, shortest paths, max-flow, MST, ...

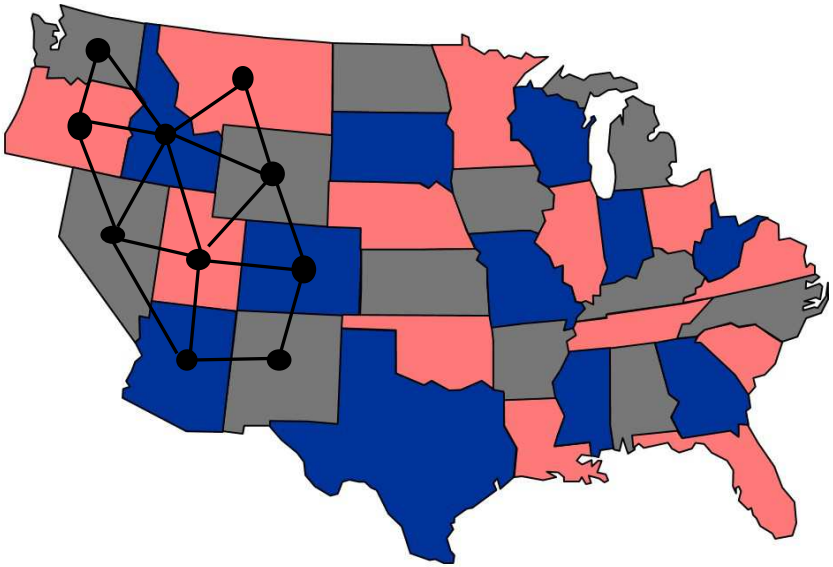
- **Some NP-complete problems become tracktable in planar graphs.**

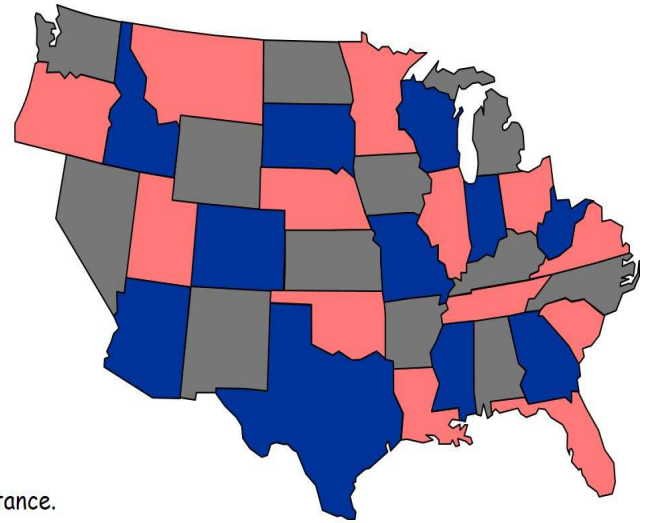
  Examples, 4-COLORABILITY,..

- **More efficient algorithms are known for some NP-complete problems in planar graphs**

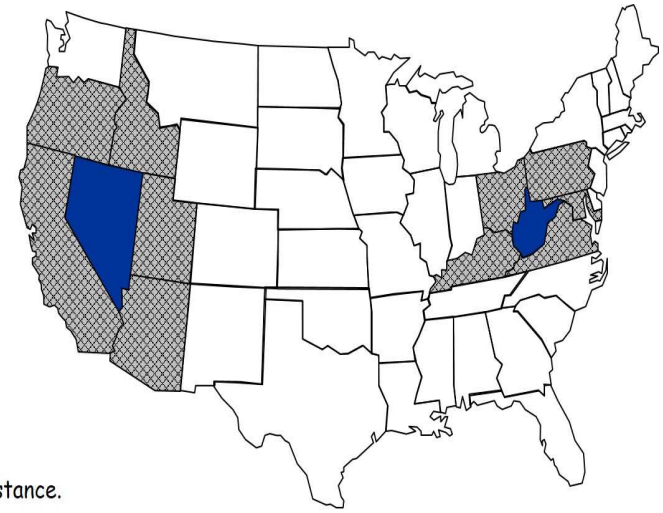
  Examples, Indepedent set, vertex cover, TSP,...

- **Planar-3-Color: if a planar graph can be colored by 3 colors s.t. no two adjacent nodes are given a same color?**
- **Planar-Map-3-Color: If a planar map can be colored by 3 colors s.t. no two adjacent regions are given a same color?**
- **Planar-3-Color$\equiv_P$Planar-Map-3-Color.**





YES instance.



NO instance.

- **Every planar map is 4-colorable [Appel-Haken 1976]**

  Solved a long standing open problem.

  First major theorem proved using a computer.

- **Appel-Haken's proof gives an $O(n^4)$ algorithm to 4-color of a planar map.**

  Best known: $O(n^2)$ for 4-color, $O(n)$ for 5 color.

- **Planar-3-Color is NP-complete.**

  Planar-3-Color is in NP and 3-Color$\leq_P$Planar-3-Color.

Exponential Time Algorithms

- **Brute-force algorithm for 3-SAT**

  Given 3-CNF Φ of n variables and m clauses, check whether each of $\{0,1\}^n$ truth assignments saisfies Φ, takes $O((m+n)2^n)$ time.

- **Recursive framework for improvement**

  Let $(l_1 \vee l_2 \vee l_3)$ be a clause of Φ. Then

$$\begin{aligned}\Phi &= (l_1 \vee l_2 \vee l_3) \wedge \Phi' = (l_1 \wedge \Phi') \vee (l_2 \wedge \Phi') \vee (l_3 \wedge \Phi') \\ &= (\Phi'|l_1 = 1) \vee (\Phi'|l_2 = 1) \vee (\Phi'|l_3 = 1),\end{aligned}$$

  where $l_i = 1$ denotes setting literal l_i to true.

- **Example:**

$$\begin{aligned}\Phi &= (x_1 \vee x_2 \vee \bar{x}_3)(x_1 \vee \bar{x}_2 \vee x_3)(x_4 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee x_3) \\ \Phi' &= (x_1 \vee \bar{x}_2 \vee x_3)(x_4 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee x_3) \\ (\Phi'|x_1 = 1) &= (x_4 \vee x_2 \vee \bar{x}_3)(x_2 \vee x_3).\end{aligned}$$

Recursive algorithm for 3-SAT

3-SAT(Φ)

Input: A 3-CNF Φ.

Output: T if Φ is satisfiable, otherwise F.

if Φ is empty **then** return T;

Compute $(l_1 \vee l_2 \vee l_3) \wedge \Phi'$ from Φ;

if 3-SAT($\Phi'|l_1 = 1$)=T **then** return T;

if 3-SAT($\Phi'|l_2 = 1$)=T **then** return T;

if 3-SAT($\Phi'|l_3 = 1$)=T **then** return T;

else Return F

3-SAT algorithm solves 3-SAT problem in $O(\mathrm{Poly}(n)3^n)$ time.

Proof. Let $T(n)$ be the running time. $T(n) \leq 3T(n-1) + O(m+n)$. □

Improve 3-SAT algorithm

- $\Phi'|l_1 = 1$, $\Phi'|l_2 = 1$ **and** $\Phi'|l_3 = 1$ **are not mutually exclusive, each satisfiable assignment containing** $(l_1 \vee l_2 \vee l_3)$ **is one of the cases:**
  - l_1 **is true**
  - l_1 **is false and** l_2 **is true**
  - l_1 **is false and** l_2 **is false and** l_3 **is true**

3-SAT1(Φ)

Input: A 3-CNF Φ.

Output: T if Φ is satisfiable, otherwise F.

if Φ is empty **then** return T;

Compute $(l_1 \vee l_2 \vee l_3) \wedge \Phi'$ from Φ;

if 3-SAT($\Phi'|l_1 = 1$)=T **then** return T;

if 3-SAT($\Phi'|l_1 = 0, l_2 = 1$)=T **then** return T;

if 3-SAT($\Phi'|l_1 = l_2 = 0, l_3 = 1$)=T **then** return T;

Return F

- **3-SAT1 algorithm solves 3-SAT problem in $O(\mathrm{Poly}(n)1.84^n)$ time.**

  Proof. Let $T(n)$ be the running time.

  $$T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(m+n).$$

  Guess $T(n) = \alpha^n$. Then

  $$\alpha^n = \alpha^{n-1} + \alpha^{n-2} + \alpha^{n-3}.$$

  Largest root of $\alpha^3 = \alpha^2 + \alpha + 1$: $\alpha \approx 1.84$. □

- **There is an $O(1.33334^n)$ ($(4/3)^n\mathrm{Poly}(n)$) time algorithm for 3-SAT [Moser and Scheder 2010].**

DPLL algorithm for SAT [Davis-Putman-Logemann-Loveland 1960 1962]

- **Davis-Putman-Logemann-Loveland algorithm**
  - **Splitting: assign truth value to a literal and solve both possibilities.**

    Example, assign x_i 0 **and** 1 **to reduce** $\Phi(x_1, .., x_i, .., x_n)$ **to** $\Phi(x_1, .., 0, .., x_n)$ **and** $\Phi(x_1, ..1, .., x_n)$**.**
  - **Unit propagation: clause contains only a single unassigned literal.**
  - **Pure literal elimination: if a variable appears only unnegated or negated.**
- **Base for many SAT solver, worst case running time** $O(2^n)$**, space** $O(n)$**.**

Fixed Parameter Algorithms

- **Vertex-Cover: Given a graph G and integer k, is there a subset of nodes $S \subseteq V(G)$ s.t. $|S| \leq k$ and for each edge $\{u, v\} \in E(G)$, either u or v is in S.**
- **Vertex-Cover is NP-complete. A brute-force algorithm takes $O(kn^{k+1})$ time:**
  - **take each of $\binom{n}{k} = O(n^k)$ subsets of size k, and**
  - **check if the selected subset is a vertex cover in $O(kn)$ time.**
- **Is it possible to have an algorithm with $(f(k)\mathrm{Poly}(n))$ time, $f(k)$ is a function depending only on k (e.g., $f(k) = 2^k$)?**
- **If k is a constant then $O(f(k)\mathrm{Poly}(n))$ is poly-time.**

- **A decision problem X can be expressed as a parameterized decision problem (X, K) based on their inherent difficulty w.r.t. some parameter k of the input or output.**
- **A parameterized problem (X, K) is fixed-parameter tractable (fpt) if given a problem instance (x, k), whether $(x, k) \in (X, K)$ can be decided in $O(f(k)\mathrm{Poly}(n))$ time, f is a function depending on k only.**
- **An algorithm which solves problem (X, K) in $O(f(k)\mathrm{Poly}(n))$ is called an fpt-algorithm for the problem.**
- **Example, Vertex-Cover admits an fpt-algorithm.**

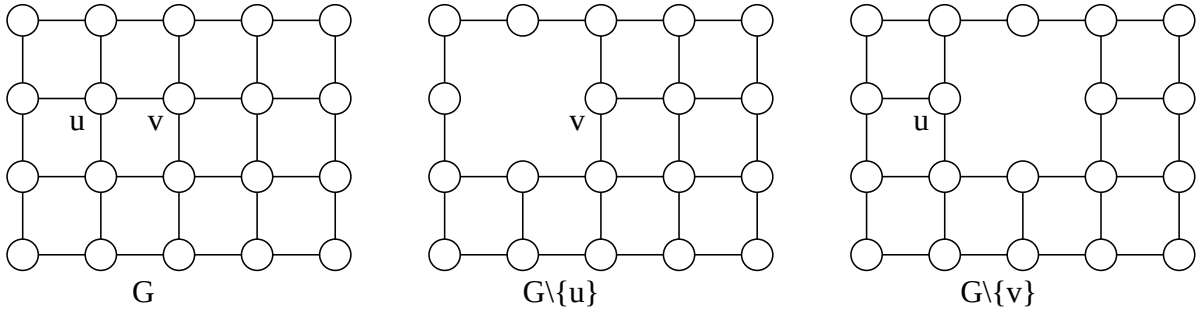
Fpt-algorithm for Vertex-Cover

- **For any edge $\{u, v\}$ of G, G has a vertex cover of size$\leq k$ iff one of $G \setminus \{u\}$ and $G \setminus \{v\}$ has a vertex cover of size$\leq k-1$.**

  Proof. Assume G has a vertex cover S of $|S| \leq k$. Then S has u or v. Assume S has u. Then $S \setminus \{u\}$ is a vertex cover of $G \setminus \{u\}$.

  Assume S is a vertex cover of $G \setminus \{u\}$ with $|S| \leq k-1$. Then $S \cup \{u\}$ is a vertex cover of G. □

- **If G has a vertex cover of size k, G has at most $k(n-1)$ edges.**



VC(G, k)

Input: A graph G and integer k.

Output: T if G has a vertex cover of size$\leq k$, otherwise F.

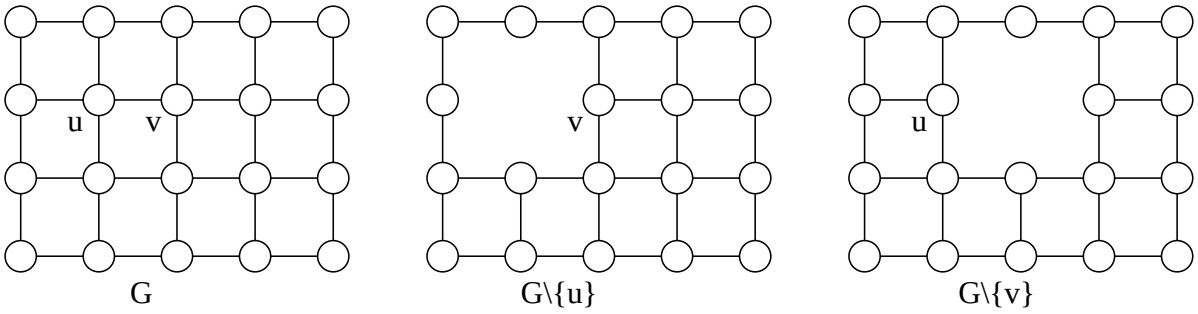
if G has no edge then return T;

if G has$\geq kn$ edges then return F;

Let $\{u, v\}$ be any edge of G;

$a =$**VC(**$G \setminus \{u\}, k-1$**);** $b =$**VC(**$G \setminus \{v\}, k-1$**);**

if $a =$F and $b =$F then return F else return T



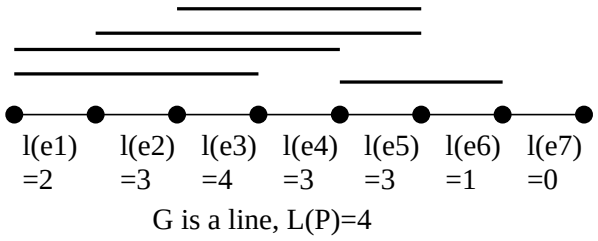
- **VC algorithm solves Vertex-Cover in $O(2^k kn)$ time.**

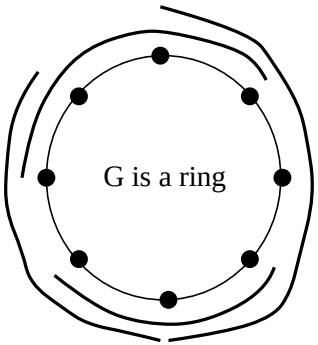
  Proof. The depth of the recursion is at most k, implying at most 2^{k+1} calls of the algorithm. Each call takes $O(kn)$ time. □

- **VC algorithm is an-fpt algorithm for Vertex-Cover.**
- **Vertex-Cover is fixed parameter tractable.**

Circular arc coloring problem

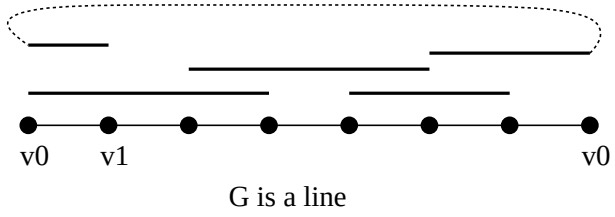
- **Given a graph G and a set P of paths on G, the load $l(e)$ of edge e is the number of paths containing e. The load of P is defined as $L(P) = \max_{e \in E(G)} l(e)$.**
- **Path-Color: given G, P and k colors, if each path of P can be colored by one of the k colors s.t. any two paths sharing a common edge are given distinct colors.**
  - **$k \geq L(P)$.**
  - **Circular-Arc-Color: Path-Color when G is a ring; NP-complete.**
  - **Path-Color can be solved in Poly-time if G is a line.**
- **For G of n nodes and P of m paths, brute force algorithm solves Circular-Arc-Color in $O(k^m \mathrm{Poly}(m, n))$ time.**

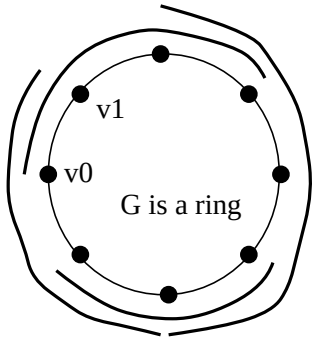




Fixed parameter algorithm for Circular-Arc-Color

- **Dynamic programming algorithm**
  - **Let G be a ring of n nodes $v_0, .., v_{n-1}$. Cut the ring at any node, say v_0, into a line $\{v_0, v_1\}\{v_1, v_2\}..\{v_{n-1}v_0\}$.**
  - **Assign distinct colors for the paths containing edge $\{v_0, v_1\}$.**
  - **For paths containing an edge of $\{v_0, v_1\}, .., \{v_{i-1}, v_i\}$ ($i > 1$), enumerate all k-colorings that are consistent with the k-colorings for paths containing an edge of $\{v_0, v_1\}, .., \{v_{i-2}, v_{i-1}\}$ to find all possible colorings s.t. the paths sharing a common edge are given distinct colors.**





- **The algorithm solves Circular-Arc-Color in $O(k!n)$ time.**

  Proof. The algorithm has n phases, each phase has $L(P) \leq k$ paths to consider, implying at most $k!$ different colorings to enumerate. □

- **Circular-Arc-Color is fixed parameter tractable.**

Approximation Algorithms for Metric TSP

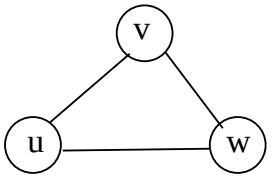
- **TSP problem: Given a complete (di)graph G with each edge assigned a non-negative weight, find a simple cycle containing very node of G s.t. the total weight of edges in the cycle is minimized.**

  For any $\alpha > 1$, an α-approximation algorithm for TSP implies P=NP.

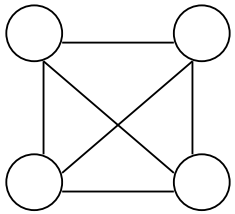
- **Metric TSP: TSP on G that for any three nodes u, v, w of G, $d(u, w) \leq d(u, v) + d(v, w)$, where $d(x, y)$ is the weight of edge (x, y).**

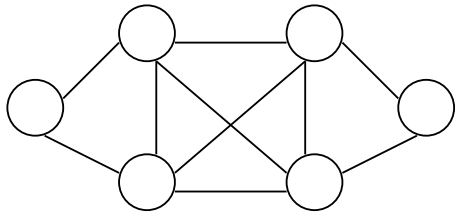
- **Eulerian cycle G, a cycle of G contains each edge exactly once.**

  An undirected graph G has an Eulerian cycle iff each node has an even degree.



d(u,w)≤d(u,v)+d(v,w)


Graph does not have an Eulerian cycle

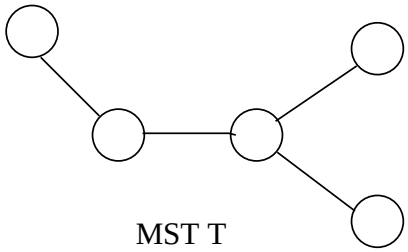

Graph has an Eulerian cycle

- **For a subgraph G' of G, denote $d(G')$ be the sum of all edge weights in G'.**
- **A 2-approximation algorithm for (symmetric) metric TSP on undirected graph (double tree algorithm).**

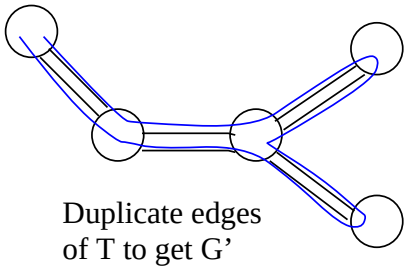
  Find a minimum spanning tree (MST) T of G.

  Duplicate each edge of T to get graph G', find an Eulerian cycle of G'.

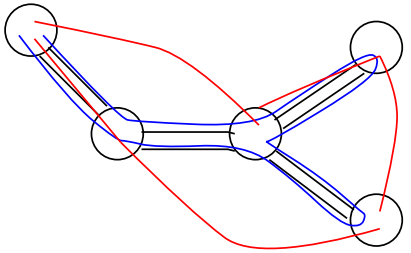
  Shortcut C: Initalize $C =$ the Eulerian cycle. If a node v of G appears in C more than once, replace edges $\{u, v\}, \{v, w\}$ in C by edge $\{v, w\}$ until each node of G appears in C exacly once.



MST T



Duplicate edges
of T to get G'
Eulerian cycle



Shortcut Eulerian cycle to get
cycle C

- **There is a 2-approximation algorithm for the symmetric metric TSP.**

  Proof. Let C be the solution by the double tree algorithm. For a node v of G appears in cycle C $k \geq 1$ times, v has degree $2k$ in C. For $k > 1$, edges $\{u, v\}, \{v, w\}$ are replaced by edge $\{u, w\}$, degree of v becomes $2k - 2$, degrees of u and w unchanged. So, when the algoithm terminates, each node of G appears in C exacly once.

  Let C^* be an optimal solution for the metric TSP and T be an MST of G. Since removing one edge from C^* gives a spanning tree of G, $d(T) \leq d(C^*)$. Initially, $d(C) = 2d(T)$, when edges $\{u, v\}, \{v, w\}$ are replaced by $\{u, w\}$, $d(u, w) \leq d(u, v) + d(v, w)$. So, $d(C) \leq 2d(T) \leq 2d(C^*)$. The running time of the algorithm is $O(t(n) + n)$, where $t(n)$ is the time to find an MST. □

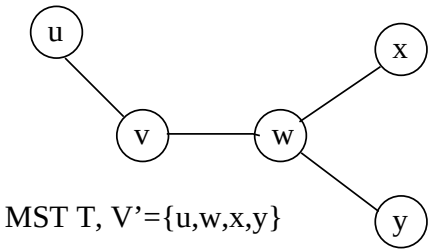
- **A 1.5-approximation algorithm for symmetric metric TSP (Christofides' algorithm)**

  Find an MST T of G. Let V' be the set of nodes in T of odd degree.

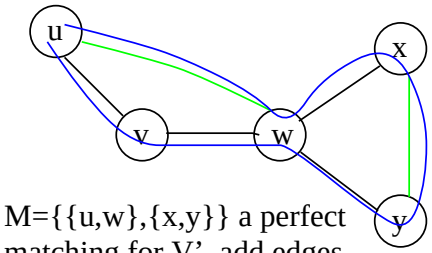
  Find a minimum weight perfect matching M of edges in G for V'.

  Add edges of M to T to get graph G', find an Eulerian cycle of G'.

  Shortcut C: Initalize $C =$ the Eulerian cycle. If a node v of G appears in C more than once, replace edges $\{u, v\}, \{v, w\}$ in C by edge $\{u, w\}$ until each node of G appears in C exacly once.

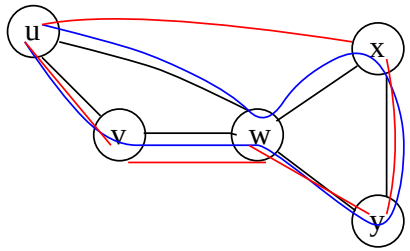


MST T, V'={u,w,x,y}



M={{u,w},{x,y}} a perfect matching for V', add edges of M to T to get graph G'

Find a Eulerian cycle of G'

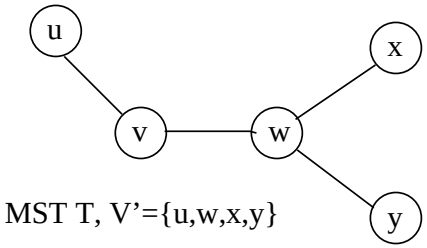


Shortcut Eulerian cycle to get cycle C

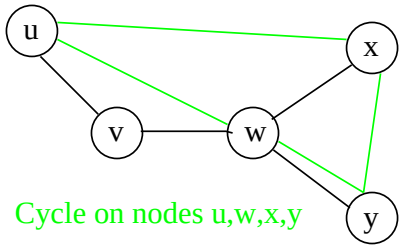
- **There is a 1.5-approximation algorithm for the symmetric metric TSP**

Proof. The number of nodes of odd degree in a graph is even. So, $|V'|$ is even and there is a perfect matching for V'. Add edges of M to T to get G', every node of G' has even degree. Let C be the solution by Christofides' algorithm. As shown before, each node of G appears in C exactly once.

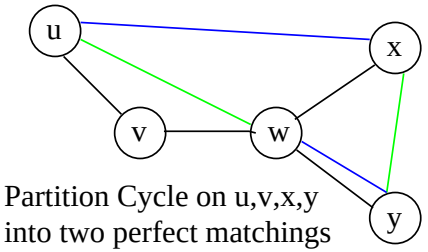
Let C^* be an optimal solution for the metric TSP and M be a minimum perfect matching for V'. We first show $d(M) \leq d(C^*)/2$.
Initialize $C' = C^*$. For each node $v \notin V'$, replace edges $\{u, v\}, \{v, w\}$ in C' by edge $\{u, w\}$ until C' contains every node of V' only. Since $d(u, w) \leq d(u, v) + d(v, w)$, $d(C') \leq d(C^*)$. C' can be decomposed into two perfect matchings M_1 and M_2 for V'.
So, $d(M) \leq \min\{d(M_1), d(M_2)\} \leq d(C')/2 \leq d(C^*)/2$.
For solution C by the algorithm, $d(C) \leq d(G') \leq d(T) + d(M) \leq 1.5d(C^*)$. A minimum cost perfect matching for V' can be computed in poly-time, algorithm takes poly-time. □



MST T, V'={u,w,x,y}



Cycle on nodes u,w,x,y



Partition Cycle on u,v,x,y into two perfect matchings

Tree-decomposition based Algorithms

- **Graph decomposition approach for problems in graphs**
  - **Partition a graph G into subgraphs by a separator (vertex cut, edge cut, subgraph).**
  - **Problem in G becomes subproblems on subgraphs, subproblems may be independent or only related via the separator.**
  - **Combine solutions to subproblems to a solution to the problem. This step may take exponential time in the size of separator for NP-hard problems.**
- **When G is a tree, G can be paritioned into subtrees by a separator of a single node (size 1) and many NP-hard problems in graph G become tractable.**
- **Whether some NP-hard problems become tractable for G close to a tree?**
- **Tree-decomposition, a formal way to decribe how G is close to a tree.**

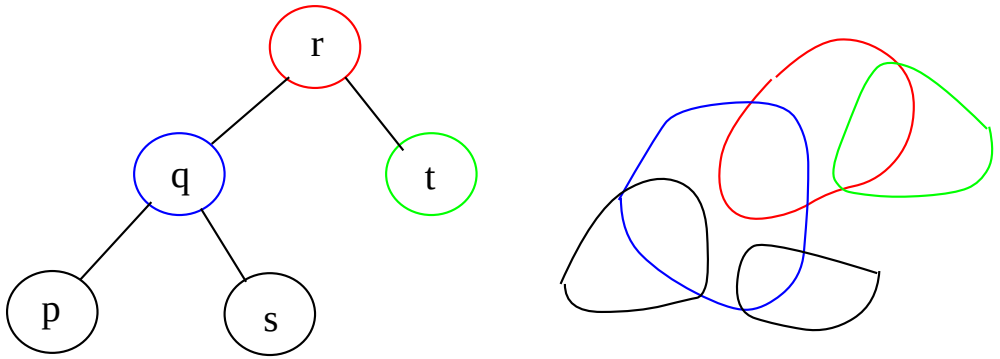
- **Tree-decomposition [Robertson and Seymour, 1986]**

  Decompose a graph G into subgraphs, view each subgraph as a node (bag) and connect the nodes into a tree T subject to some conditions.

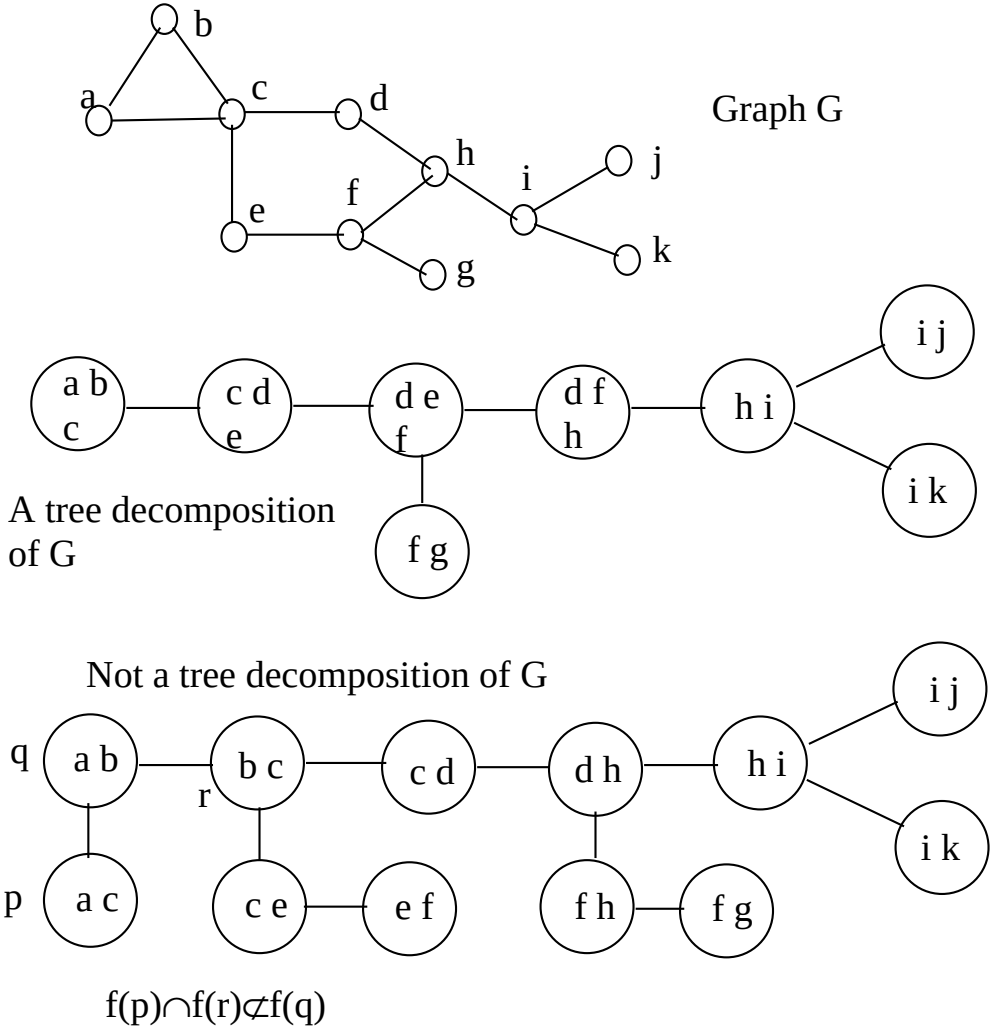
  Formally, a tree-decomposition of graph G is a pair (f, T), T is a tree and $\forall p \in V(T), f(p) \subseteq V(G)$ s.t.

  1. $\cup_{p \in V(T)} f(p) = V(G)$
  2. $\forall (u, v) \in E(G), \exists p \in V(T)$ **s.t.** $u, v, \in f(p)$
  3. $\forall p, q, r \in V(T)$**, if** q **is on the path from** p **to** r **in** T**,** $f(p) \cap f(r) \subseteq f(q)$**.**

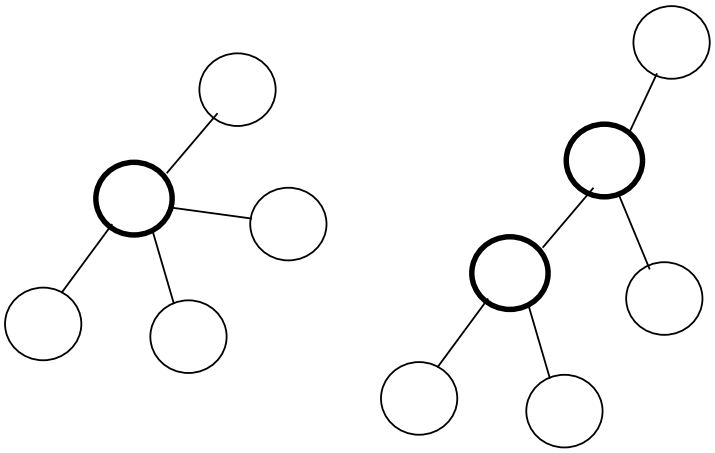
- **Treewidth of** (f, T) **is** $\max_{p \in V(T)} |f(p)| - 1$
- **Treewidth** $\mathrm{tw}(G)$ **of graph** G **is the minimum treewidth of all** (f, T)**.**



An Example of Tree Decomposition



- **Maximum independent set problem (Max-IS): given $G(V, E)$, find a maximum $W \subseteq V(G)$ s.t. $\forall u, v \in W, \{u, v\} \notin E(G)$.**
- **Tree-decomposition based algorithm for the Max-IS.**
  1. **Find a tree-decomposition (f, T) and change T to a rooted binary tree.**
  2. **$\forall p \in V(T)$, compute by dynamic programming method in bottom-up manner the IS of the subgraph induced by $\cup_q$ a descendant of ${}_pf(q)$.**



Simple example, dynamic programming for Max-IS on a rooted binary tree T

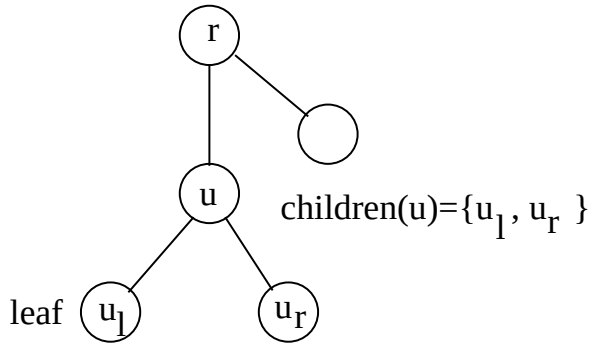
- **Optimal solution structure: For each node u of T,**
  - $\mathrm{opt}_{in}(u)$**=max-IS in subtree rooted at u, containing u.**
  - $\mathrm{opt}_{out}(u)$**=max-IS in subtree rooted at u, not containing u.**
  - **Goal: maximize $\{\mathrm{opt}_{in}(u), \mathrm{opt}_{out}(u)\}$.**

  $\{u\}$ **has two subsets, $\emptyset$ and $\{u\}$; $\emptyset \subseteq \mathrm{opt}_{out}(u)$, $\{u\} \subseteq \mathrm{opt}_{in}(u)$.**

- **Bellman equations:**

$$\begin{aligned} \mathrm{opt}_{in}(u) &= \{u\} \cup \mathrm{opt}_{out}(u_l) \cup \mathrm{opt}_{out}(u_r) \\ \mathrm{opt}_{out}(u) &= \max\{\mathrm{opt}_{in}(u_l), \mathrm{opt}_{out}(u_l)\} \cup \max\{\mathrm{opt}_{in}(u_r), \mathrm{opt}_{out}(u_r)\} \end{aligned}$$

u_l **and** u_r **are child nodes of** u**.**

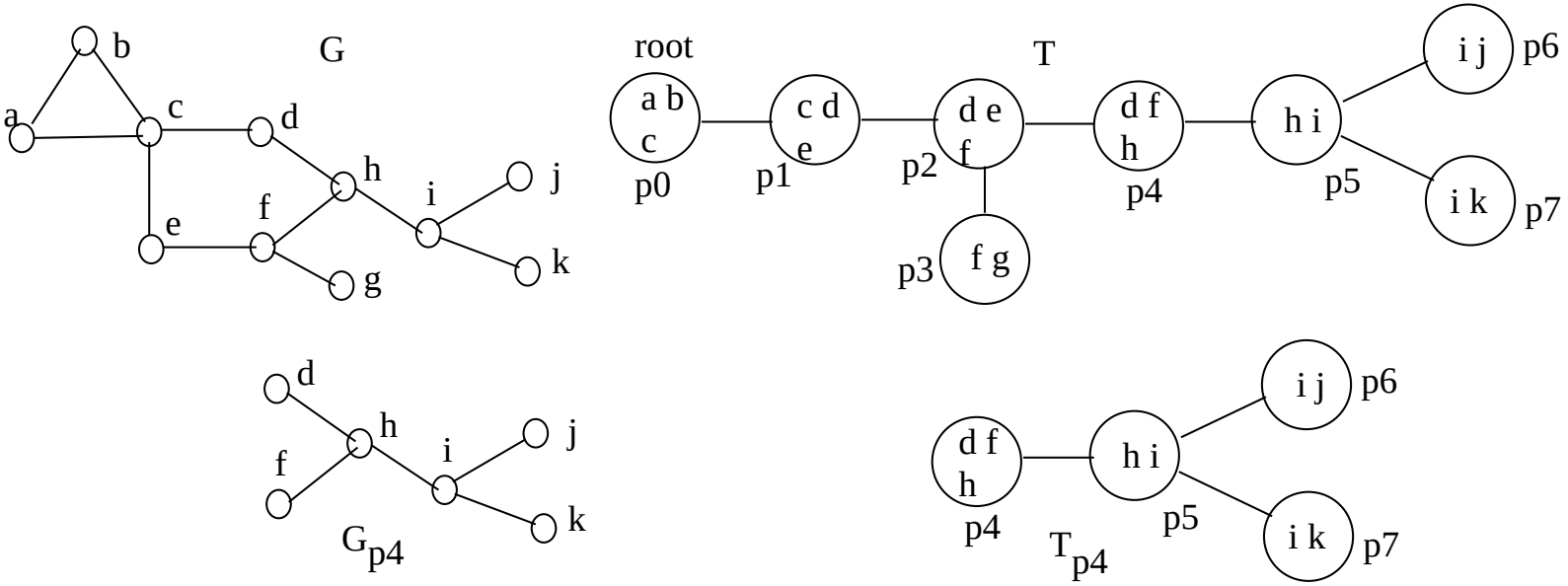


Tree-decomposition based dynamic programming for Max-IS

- **Optimal solution structure: Given a tree-decomposition (f, T) of graph G, for each bag p of T (rooted binary tree), let T_p be the subtree of T rooted at p and G_p be the subgraph of G induced by the nodes in the bags of T_p.**

  For each subset $S_p \subseteq f(p)$,

  - $\mathrm{opt}(S_p) =$**max-IS W of G_p with $S_p \subseteq W$.**
  - **Goal: maximize $\{\mathrm{opt}(S_p) | S_p \subseteq f(p)\}$.**



- **Bellman equations:**

  Let $\mathcal{W}_p = \{W | W$ **maximal IS of** $G_p\}$**.**

  For each bag p **of** T **and each subset** $S_p \subseteq f(p)$**, define**

  $\mathcal{W}_{S_p} = \{W | W$ **maximal IS of** G_P **with** $S_p \subseteq W\}$**.**

  Then $\mathcal{W}_{S_p} = \{W = S_p \cup W_l \cup W_r | W_l \in \mathcal{W}_{p_l}, W_r \in \mathcal{W}_{p_r}, W$ **is IS of** $G_p\}$**,**
  p_l **and** p_r **are child bags of** p**.**

  Let $\mathrm{opt}(S_p) = \arg\max_{W \in \mathcal{W}_{S_p}} |W|$**. Then**

  $\mathcal{W}_p = \{W | W = \mathrm{opt}(S_p), S_P \subseteq f(p), W$ **maximal**$\}$**.**

- **(1) Compute** $\mathcal{W}_p$ **for each leaf node** $p \in V(T)$**.**

  (2) Compute $\mathcal{W}_p$ **for each non-leaf** $p \in V(T)$ **in bottom-up manner in** T**.**

  (3) A maximum set in $\mathcal{W}_p$**,** p **is the root of** T**, is a solution to Max-IS.**

- **For leaf bag** p**, time for** $\mathcal{W}_p$ **is** $O(2^{|f(p)|})$**. For non-leaf bag** p**, time for** $\mathcal{W}_p$ **is** $O(2^{|f(p) \cup f(p_l) \cup f(p_r)|})$**.**

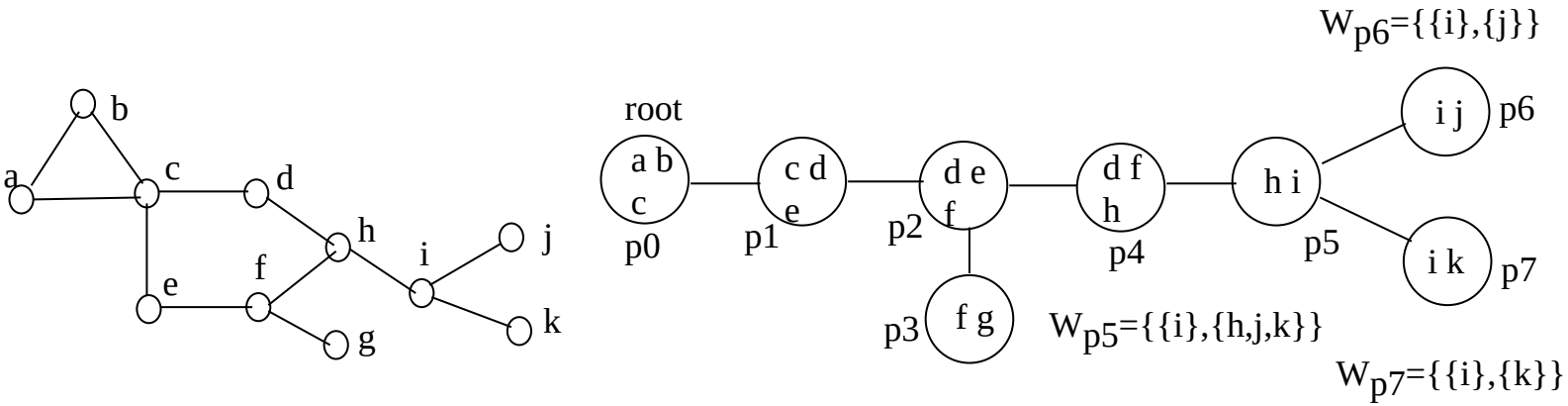
- **An example**

$\mathcal{W}_{S_p} = \{W = S_p \cup W_l \cup W_r | W_l \in \mathcal{W}_{p_l}, W_r \in \mathcal{W}_{p_r}, W$ **is IS of** $G_p\}$**,**

p_l **and** p_r **are child bags of** p**.**

$\text{opt}(S_p) = \arg\max_{W \in \mathcal{W}_{S_p}} |W|$**,**

$\mathcal{W}_p = \{W | W = \text{opt}(S_p), S_P \subseteq f(p), W$ **maximal**$\}$**.**

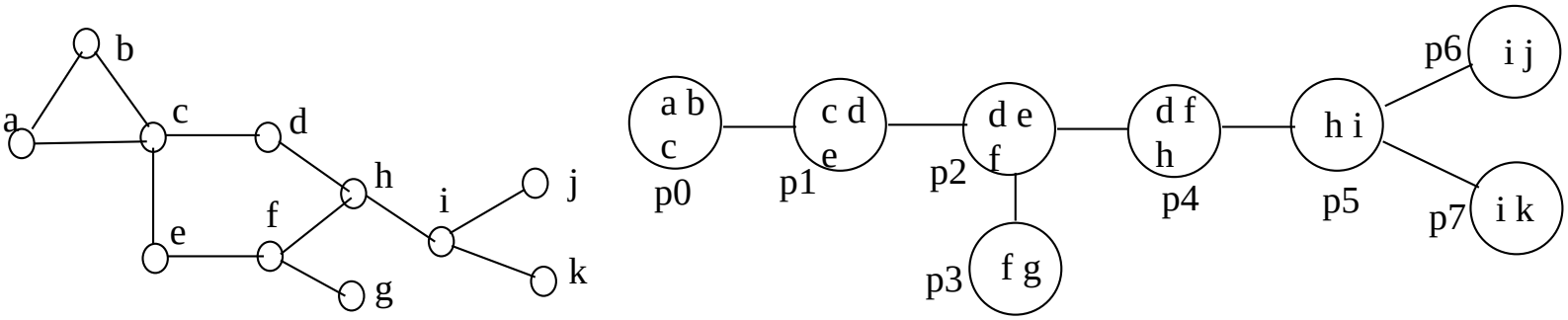


$f(p_5) = \{h, i\}, S_{p_5} = \emptyset, S_{p_5} = \{h\}, S_{p_5} = \{i\}.$

$\text{opt}(\emptyset) = \max\{\emptyset \cup \{i\} \cup \{i\} = \{i\}, \emptyset \cup \{k\} \cup \{j\}\} = \{j, k\}.$

$\text{opt}(\{h\}) = \max\{\{h\} \cup \{k\} \cup \{j\}\} = \{h, j, k\}.$

$\text{opt}(\{i\}) = \max\{\{i\} \cup \{i\} \cup \{i\}\} = \{i\}. \mathcal{W}_{p_5} = \{\{h, j, k\}, \{i\}\}.$



$\mathcal{W}_{p_3} = \{\{f\}, \{g\}\}, \mathcal{W}_{p_6} = \{\{i\}, \{j\}\}, \mathcal{W}_{p_7} = \{\{i\}, \{k\}\}.$

$\mathcal{W}_{p_5} = \{\{h, j, k\}, \{i\}\}, \mathcal{W}_{p_4} = \{\{h, j, k\}, \{d, f, i\}, \{d, f, j, k\}\}.$

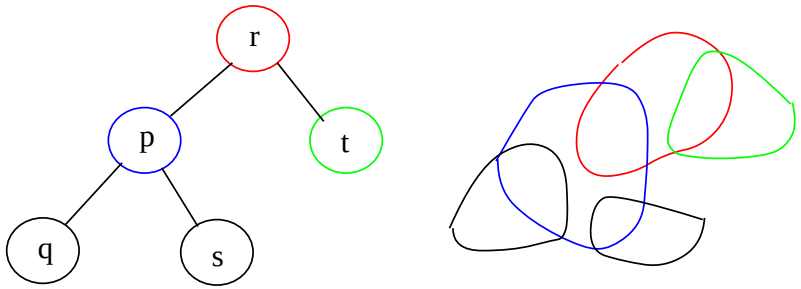
$\mathcal{W}_{p_2} = \{\{e, g, h, j, k\}, \{d, f, i\}, \{d, f, j, k\}, \{d, e, g, i\}, \{d, e, g, j, k\}\}.$

$\mathcal{W}_{p_1} = \{\{c, g, h, j, k\}, \{e, g, h, j, k\}, \{c, f, i\}, \{d, f, i\}, \{c, f, j, k\}, \{d, f, j, k\},$
$\{c, g, i\}, \{d, e, g, i\}, \{c, g, j, k\}, \{d, e, g, j, k\}\}.$

$\mathcal{W}_{p_0} = \{\{c, g, h, j, k\}, \{a, e, g, h, j, k\}, \{b, e, g, h, j, k\}, \{c, f, i\}, \{a, d, f, i\},$
$\{b, d, f, i\}, \{c, f, j, k\}, \{a, d, f, j, k\}, \{b, d, f, j, k\}, \{c, g, i\}, \{a, d, e, g, i\},$
$\{b, d, e, g, i\}, \{c, g, j, k\}, \{a, d, e, g, j, k\}, \{b, d, e, g, j, k\}\}.$

Intuition of tree-decomposition based algorithm

- T can be viewed as a binary tree with root r.
- The vertex set $f(r)$ separates G into two subgraphs G_1 and G_2.
- For a problem on G, if the solution on G_1 (or G_2) only depends on G_1 (or G_2) and the subgraph induced by $f(r)$ then the problem can be decomposed into two independent problems on G_1 and G_2.
- For every internal bag p of T, $f(p)$ separates the subgraph G_P induced by the nodes of bags in the subtree T_p rooted at p into two subgraphs.
- Dynamic programming algorithm using the tree-decomposition (f, T).



Construction of tree-decomposition

- **For arbitrary graphs**

  It is NP-hard to find an optimal tree-decomposition [Arnborg et al. 1987].

  An $O(\log n)$-approximation algorithm is known [Bodlaender et al. 1992].

  $O(n)$ time for an optimal tree-decomposition of G with constant $\mathrm{tw}(G)$ [Bodlaender 1996].

- **For planar graphs**

  A 1.5-approximation algorithm is known [Robertson-Seymour 1991, Seymour-Thomas 1994].

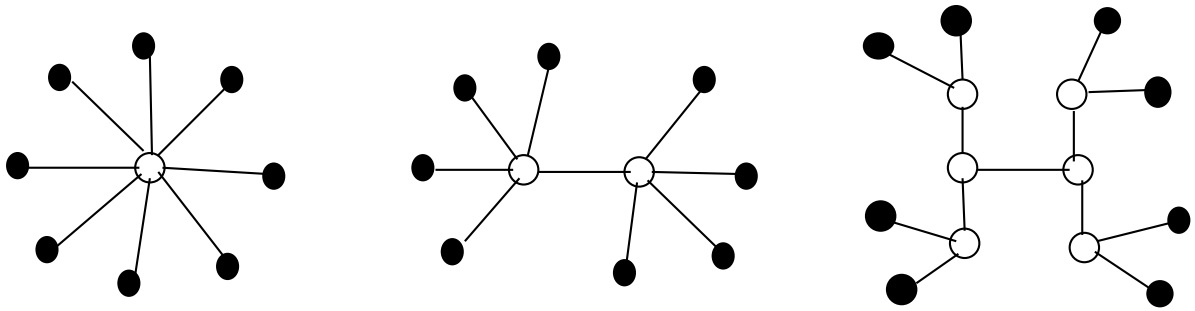
  It is open if finding an optimal tree-decomposition is NP-hard.

- **A number of heuristics are known.**

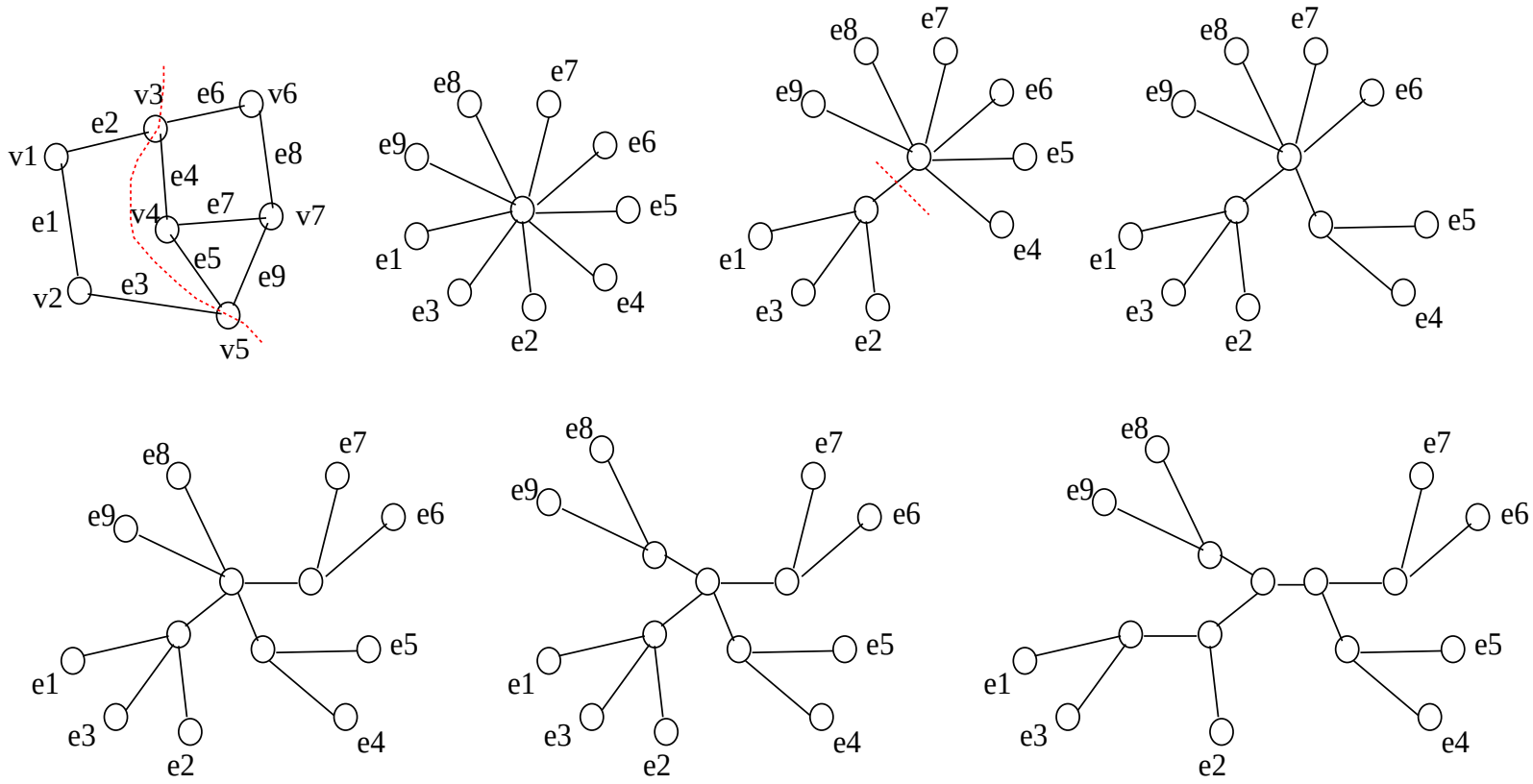
A simple heuristic for finding a tree-decomposition (1)

1. **Given G, initialize a tree T_B with one internal node and $|E(G)|$ leaf nodes, each leaf node is assigned a distinct edge of $E(G)$..**
2. **If $\exists p \in V(T_B)$ with degree greater than 3 then partition the leaves adjacent to p into two subsets V_1 and V_2, create two nodes p_1 and p_2, connect the leaves of V_1 (V_2) to p_1 (p_2), and connect p_1 and p_2 to p; otherwise output T_B and terminate.**
3. **Repeat Step 2**

T_B is known as a branch-decomposition of G. Then turn T_B into a tree-decomposition (f, T).



An example of simple heuristic to construct a branch-decomposition T_B

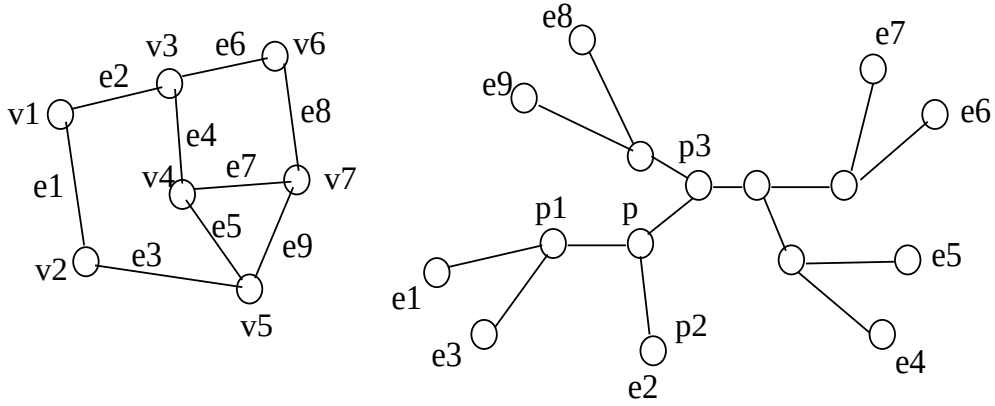


A simple heuristic for finding a tree-decomposition (2)

- **Given a branch-decomposition T_B of G, we construct a tree-decomposition (f, T).**
- **Let $T = T_B$. For each link $\{p, q\}$ of T, let $L_T(p, q)$ is the subset of leaves of T reachable from p not passing through q. (e.g., $L_T(p, p_3) = \{e_1, e_2, e_3\}$ in figure).**
- **For each leaf node p of T assigned edge $e = \{u, v\}$ of G, $f(p) = \{u, v\}$.**
- **For each internal node p of T, let $\{p, p_1\}, \{p, p_2\}, \{p, p_3\}$ be the three links of T incident to p and**

$$f(p) = \{u | u \in V(L_T(p_i, p)) \cap V(L_T(p_j, p)), i, j = 1, 2, 3, i \neq j\}$$

- **Example: in the figure, $V(L_T(p_1, p)) = \{v_1, v_2, v_5\}$, $V(L_T(p_2, p)) = \{v_1, v_3\}$, $V(L_T(p_3, p)) = \{v_3, v_4, v_5, v_6, v_7\}$, $f(p) = \{v_1, v_3, v_5\}$.**



A simple heuristic for finding a tree-decomposition (3)

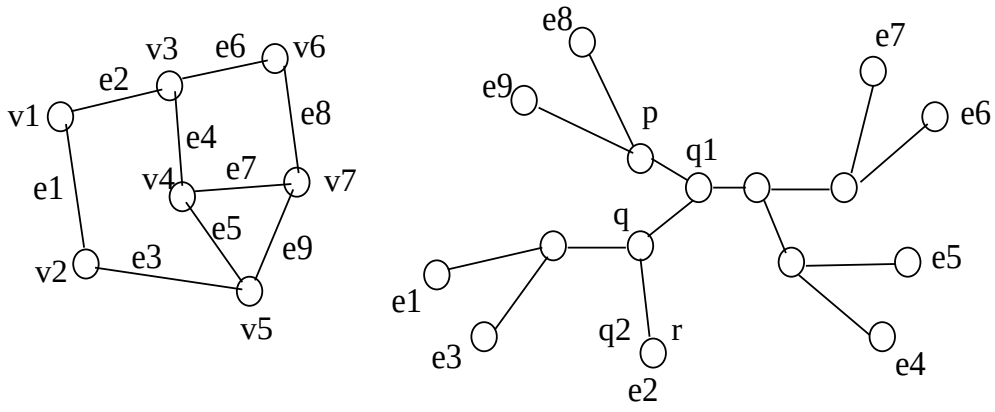
- (f, T) **constructed in previous slide is a tree-decomposition of** G**.**

  (1) $\cup_{p \in V(T)} f(p) = V(G)$**.**

  (2) For each edge $\{u, v\}$ **of** G**, there is a leaf node** p **of** T **s.t.** $u, v \in f(p)$**.**

  (3) For any nodes p, q, r **of** T**, assume** q **is on the path between** p **and** r**. Let** $\{q_1, q\}$ **be the edge in the path between** p **and** q **and** $\{q, q_2\}$ **be the edge in the path between** q **and** r**. Then**

$$f(p) \cap f(r) \subseteq V(L_T(q_1, q)) \cap V(L_T(q_2, q)) \subseteq f(q).$$

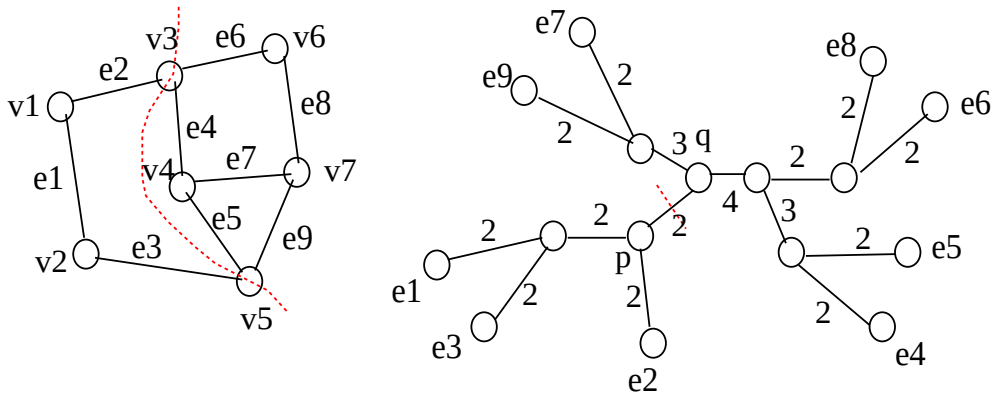


Branch-decomposition [Robertson and Seymour 1991]

- **A branch-decomposition of $G(V, E)$ is a tree T_B s.t. every internal node of T_B has node degree three and the set of leaves of T_B is $E(G)$.**
- **Removing an link $e = \{p, q\}$ of T_B separates T_B into two subtrees. Let E' and E'' be the sets of leaves of the subtrees, and $S_e = V(L_T(p, q)) \cap V(L_T(q, p))$ be the set of vertices incident to an edge of E' and an edge of E''.**
- **The width of link e is $|S_e|$ and the width of T_B is $\max_{e \in E(T)} |S_e|$.**

  The branchwidth $\mathrm{bw}(G)$ of G is the minimum width of all T_B.

Example: $e = \{p, q\}$, $L_T(p, q) = \{e_1, e_2, e_3\}$, $V(L_T(p, q)) = \{v_1, v_2, v_3, v_5\}$, $L_T(q, p) = \{e_4, e_5, e_6, e_7, e_8, e_9\}$, $V(L_T(q, p)) = \{v_3, v_4, v_5, v_6, v_7\}$, $S_e = \{v_3, v_5\}$.



Relation between treewidth and branchwidth [Robertson-Seymour 1991]
$\max\{2, \mathrm{bw}(G)\} \leq \mathrm{tw}(G) + 1 \leq \max\{2, \lfloor 3\mathrm{bw}(G)/2 \rfloor\}$.

- **A proof for** $\mathrm{tw}(G) + 1 \leq \lfloor 3\mathrm{bw}(G)/2 \rfloor$.

  Let T_B **be an optimal branch-decomposition of** G **and** (f, T) **be a tree-decomposition of** G **constructed by the simple heuristic. For any link** $e = \{p, q\}$ **in** T_B**,** $S_e = V(L_T(p, q)) \cap V(L_T(q, p))$**. For each node** p **of** T**,**

  $$f(p) = \{u | u \in V(L_T(p_i, p)) \cap V(L_T(p_j, p)), i, j = 1, 2, 3, i \neq j\}.$$

  For each node p **incident to links** $e_1 = \{p_1, p\}, e_2 = \{p_2, p\}, e_e = \{p_3, p\}$**, each vertex in** $f(p)$ **appears in at least two of** $S_{e_i} = V(L_T(p_i, p)) \cap V(L_T(p, p_i))$**,** $i = 1, 2, 3$**. Hence,** $2|f(p)| \leq \sum_{i=1}^{3} |S_{e_i}| \leq 3\mathrm{bw}(G)$**, implying** $\mathrm{tw}(G) + 1 \leq \lfloor 3\mathrm{bw}(G)/2 \rfloor$.

- **A proof for** $\max\{2, \mathrm{bw}(G)\} \leq \mathrm{tw}(G) + 1$**.**

  Let (f, T) **be a tree-decomposition of** G**. Each edge** $e = \{u, v\}$ **of** G **is in some bag** p **of** T**. If** p **has more than one edge or is an internal bag of** T **then create a new bag** p' **and with** $f(p') = \{u, v\}$**, remove edge** e **from** p**, and connect** p' **to** p **by a link. Repeat this process until each leaf of** T **has exactly one edge.**

  If there is a bag p **in** T **with degree greater than 3 then partition the neighbors of** p **into subsets** V_1 **and** V_2 **of same size, replace** p **by link** $\{p_1, p_2\}$**, connect bags of** V_1 **to** p_1 **and bags of** V_2 **to** p_2**. Repeat this process,** T **is turned into a tree with degree at most 3. For each bag** p **of degree 2, replace links** $\{p', p\}$ **and** $\{p, p"\}$ **by link** $\{p', p"\}$**, we get a branch-decomposition** T'**. The branchwidth of** T' **is at most** $\max\{f(p) | p \in V(T)\} = \mathrm{tw}(G) + 1$**.**

Intuition of branch-decomposition of a graph G**: define a collection of (vertex) separators of** G **that partitions** G **into subgraphs with a single edge a minimum subgraph.**

Branchwidth

- **Given an integer β, it is NP-complete to decide if $\mathrm{bw}(G) \geq \beta$ for general graph G [Seymour and Thomas, 1994]**
- **An optimal branch-decomposition of G with constant $\mathrm{bw}(G)$ can be found in linear time [Bodlaender and Thilikos, 1997]**
- **A number of heuristics are known.**
- **For planar graph G, if $\mathrm{bw}(G) \geq \beta$ can be decided in $O(n^2)$ time. [Rat-catching algorithm, Seymour-Thomas 1994]**

  An optimal branch-decomposition of G can be computed in $O(n^4)$ time [Edge-contraction algorithm, Seymour-Thomas 1994], implying 1.5-approximation algorithm for optimal tree-decomposition of planar graphs.

  An improved edge-contraction of $O(n^3)$ time is known [Gu-Tamaki 2005]

  An $\min\{O(n \log^3 n \log k), O(nk^2 \log k)\}$, $k = \mathrm{bw}(G)$, time $(2+\epsilon)$-approximation algorithm is known [Gu-Xu 2019]

More information on fixed parameter algorithms and tree-/branch-decomposition based algorithms can be found in:

- **Invitation to Fixed Parameter Algorithms, by Rolf Niedermeier, Oxford Univ Press, 2006.**
- **N. Robertson and P.D. Seymour. Graph minors II. Algorithmic aspects of tree-width. Journal of Algorithms, 7:309–322, 1986.**
- **N. Robertson and P.D. Seymour. Graph minors X. Obstructions to tree-decomposition. Journal of Combinatorial Theory, Series B 52(2):153–190, 1991.**
- **P.D. Seymour and R. Thomas. Call routing and the ratcatcher. Combinatorica, 14(2):217–241, 1994.**
- **Q. Gu and H. Tamaki. Optimal branch-decomposition of planar graphs in $O(n^3)$ time, ACM Trans. on Algorithms, 4(3):30.1-30.13, 2008**
- **I. Hicks, A. Koster, and E. Kolotoglu. Branch and tree-decompostion techniques for discrete optimization, Tutorials in Operations Research, Informs 2005.**