CS612 Algorithm Design and Analysis

Lecture 13. Extending the limits of tractability [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1] The slides are made based on Chapter 10 of Algorithm design, and lectures by D. P. Williamson.

Outline

- Introduction to FPT(Fixed parameter tractability) problems;
- Solving special cases of NP-Hard problems:
  1. Solving a problem when parameters are small: VertexCover;
  2. Solving a problem when connection among sub-problems is small:
     - NP-Hard problems might be easy when input is a tree: IndependentSet, and WeightedIndependentSet;
     - Extending tree to tree-like: tree-decomposition of graph;

Note:

- A particular practical instance of a NP-Hard problem might has special structure to make it easier than the worst cases.
- If we have an efficient algorithm for a tree, it is instructive to consider whether the algorithm can be extended to deal with a general graph with small width.

How to deal with hard problems? Trade-off "quality" and "time"

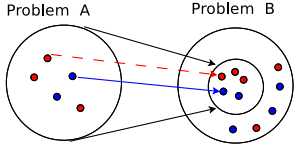
We have a couple of options:

1. Give up polynomial-time restriction: hope that our algorithms run fast on the practical instances. (e.g. branch-and-bound, branch-and-cut, and branch-and-pricing algorithms are used to solve a TSP instance with over 24978 Swedish Cities. See http://www.tsp.gatech.edu/history/pictorial/sw24978.html)
2. Give up optimum restriction: from "optimal" solution to "nearly optimal" solution in the hope that "nearly optimal" is easy to find. e.g., approximation algorithm (with theoretical guarantee), heuristics, local search (without theoretical guarantee);
3. Give up deterministic restriction: the expectation of running time of a randomized algorithm might be polynomial;
4. Give up worst-case restriction: algorithm might be fast on special and limited cases;

Extending the limit of tractability

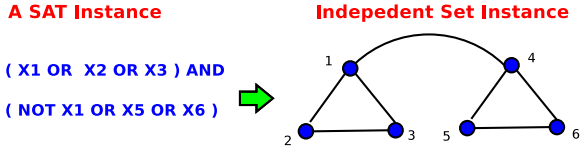
Recall the reduction procedure

- Polynomial-time reduction: a procedure to transform ANY instance α of problem A to some instance $\beta = f(\alpha)$ of problem B with the following characteristics:
  1. (Transformation) The transformation takes polynomial time;
  2. (Equivalence) The answers are the same: the answer for α is "YES" iff the answer to $\beta = f(\alpha)$ is also "YES".
- Denoted as $A \leq_P B$, read as " *A is reducible to B* ".



Recall the NP-Hardness proof of INDEPENDENT SET

- For a given SAT instance ϕ with k clauses, constructing an INDEPENDENT SET instance (G, k') as follows:
  1. G consists of k triangles: each triangle corresponds to a clause C_i; the nodes are labeled with the literals; connecting x_i and $\neg x_i$ with an edge;
  2. Set $k' = k$;
- Example:
  $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_5 \vee x_6)$



- Intuition: edge represents "conflicts"; we should identify k nodes (each node from a triangle) without connections (no conflict);

A way to deal with NP-Hard problems

- NP-Completeness just means that the the worst-case instances of the problem are very difficult and unlikely to be solvable in polynomial time.
- However, on a particular practial instance, it is possible that we are not really in the worst case—the instance we're looking at might have some special problem structure that makes our task easier.
- What special problem structure can we utilize?
  1. Special parameter: The instance is easier when parameters are small;
  2. Special input structure: The instance is easier if we require the input to be a tree(a special graph);
  3. Extension: "tree-like" graph, a special class of graph with small *tree-width*;

Sovling NP-Hard problems when parameters are small: the iteration number is limited

Small parameter: Parameterized complexity and FPT problems

- In computer science, parameterized complexity (also called Fixed-Parameter Tractability) is a measure of complexity for problems with multiple input parameters [R. Downey, M Fellows, 99].
- Motivation: Some hard problems can be solved by algorithms that are exponential only in the size of a fixed parameter k while polynomial in the size of the input size n, say, $T(n,k) = O(2^k ploy(n))$.
- Hence, if k is fixed at a small value, such problems can still be considered 'tractable' despite their traditional classification as 'intractable'.

Two examples

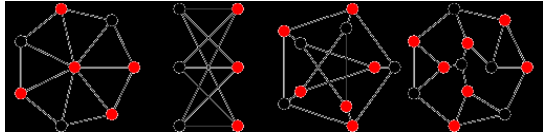
1. For the VertexCover problem, the parameter can be the number of vertices in the cover.
2. When modeling error correction, one can assume the error (denoted as k) to be "small" compared to the total input size. Then it is interesting to see whether we can find an algorithm which is exponential only in k, and not in the input size.

Example: finding small vertex covers

Intuition: Given a graph $G =< V, E >$, how many guards should be deployed on nodes to surveille ALL the paths?

Formalized Definition:

Input: Given a graph $G =< V, E >$, and an integer k,
Output: is there a set of nodes $S \subseteq V$, $|S| \leq k$, such that each edge has at least one of its endpoints in S?



Trial 1: a brute-force algorithm

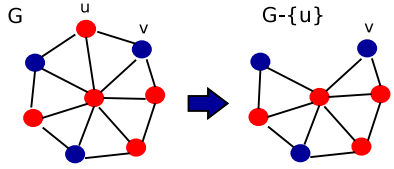
- Brute-force algorithm: enumerating all possible subsets of V of size k.
- Time-complexity: $O(kn\binom{n}{k}) = O(kn^{k+1})$. (There are $\binom{n}{k}$ subsets, and it takes $O(kn)$ time to check whether a subset is a vertex cover.)
- Note: The brute-force algorithm is a polynomial time algorithm when k is a fixed constant (e.g., $k = 2$ or $k = 3$).

Trial 1: a brute-force algorithm cont'd

- However, even for moderately small values of k, (say $n = 1000$ and $k = 10$), a running time of $O(kn^{k+1})$ is quite impractical.
- In contrast, an algorithm with the running-time of $O(k2^k n)$ is appealing since:
  1. The algorithm will be practical even when $n = 1000$ and $k = 10$;
  2. The expoential dependence on k has been moved out of the exponent on n and into a separate function.
- Question: can we break out the interaction of n and k?

Trial 2: Limited enumeration

- Basic idea: Perform limited enumeration. Enumerate all 2^k possibilities for k arbitrarily chosen edges;
  1. Consider any edge $e = (u, v)$. In *any* k-node vertex cover of G, either u or v should belong to vertex cover S.
  2. Suppose that u belongs to S. Then if we delete node u and all incident edges, we obtain a new graph $G - \{u\}$, and there should be a vertex cover of $G' = G - \{u\}$ of size at most $k - 1$.



Limited enumeration: correctness

Theorem

Let $e = (u, v)$ be any edge of G. G has a vertex cover of size at most k iff $G - \{u\}$ or $G - \{v\}$ has a vertex cover of size at most $k - 1$.

Proof.

- $\Leftarrow$

  Obvious.

- $\Rightarrow$

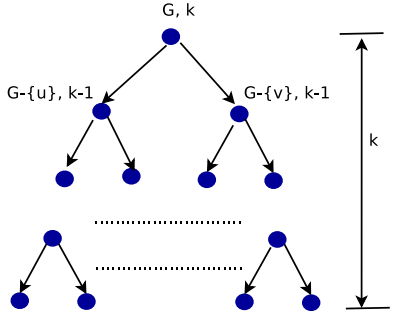
  - Suppose S is a vertex cover with size at most k, and $u \in S$.
  - Then $S - \{u\}$ must cover all edges except the edges incident to u.

Algorithm $VertexCover(G, k)$

1: **if** $G = \{\}$ **then**
2: **return** $\{\}$;
3: **end if**;
4: **if** $|G| > k|V|$ **then**
5: **return** "Infeasible";
6: **end if**
7: Arbitrarily select an edge $e =< u, v >$;
8: $S_u = VertexCover(G - \{u\}, k - 1)$;
9: $S_v = VertexCover(G - \{v\}, k - 1)$;
10: **if** $S_u \neq Infeasible$ **then**
11: **return** $S_u \cup \{u\}$;
12: **else if** $S_v \neq Infeasible$ **then**
13: **return** $S_v \cup \{v\}$;
14: **else**
15: **return** Infeasible;
16: **end if**

Note: If $G = (V, E)$ has n nodes and a vertex cover S of size k, then G has at most $k(n - 1)$ edges. (Reason: each node in S covers at most $n - 1$ edges since the maximum degree is $n - 1$.)

Analysis



Theorem

The recursive algorithm cost $O(k2^k n)$ time.

Analysis: Induction proof

Proof.

- Let $T(n,k)$ be the running time of searching a vertex cover with size of k on a graph with n nodes.
- We have:

$$\begin{aligned} T(n,1) &\leq cn && (1)\\ T(n,k) &\leq 2T(n-1,k-1)+ckn && (2)\end{aligned}$$

- Suppose $T(n,k-1) \leq ck2^k n$. Then we have:

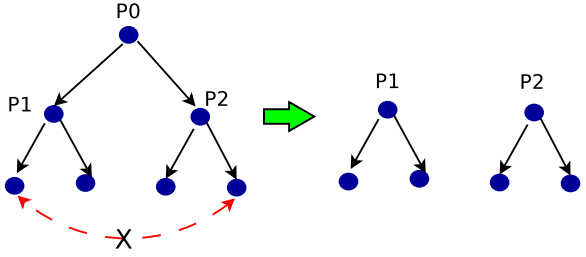
$$\begin{aligned} T(n,k) &\leq 2T(n-1,k-1)+ckn && (3)\\ &\leq 2c(k-1)2^{k-1}n+ckn && (4)\\ &= c(k-1)2^k n+ckn && (5)\\ &\leq ck2^k n && (6)\end{aligned}$$

□

Solving NP-Hard problems on trees: no communications among sub-problems

Solving NP-Hard problem on trees

- Another favor of specical structure: not when the natural "size" parameters are small, but when the problem structure is "simple";
- Tree is a simple graph: on a tree, a problem can be decomposed into "independent" sub-problems. Thus, the communication between sub-problems are broken.
- In fact, it has been found that many NP-Hard graph problems can be solved efficiently when the underlying graph is tree.

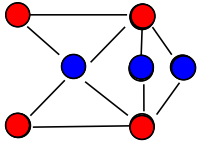


An example: INDEPENDENT SET Problem

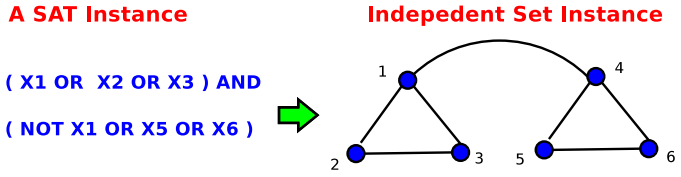
Practical Problem:
Suppose you have n friends, and some pairs of them don't get along. How to invite at least k of them to dinner if you don't want any interpersonal tension?

Formalized Definition:

Input: Given a graph $G =< V, E >$. Each node $u \in V$ has a weight $w(u) \geq 0$.
Output: to find a set of nodes $S \subseteq V$ such that no two nodes in S are joined by an edge and $\sum_{v \in S} w(v)$ is maximized.



Recall the NP-Hardness proof of Independent Set problem

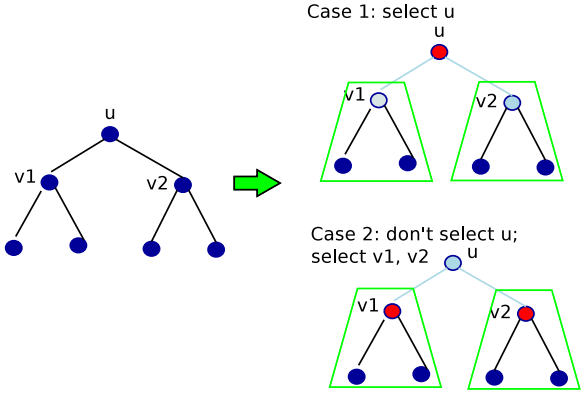


- Note: the generated graph is very special
- Question: is the problem still hard when the underlying graph is a tree?

Independent Set problem is easy when G is a tree.

1. Solution: a set of nodes. Imagine the solving process as a series of decisions; each decision is to select a node.
2. Suppose we have already worked out the optimal solution O, where the first decision is made for root node u. There are a total of 2 options for this step:
   1. Select u: This decision decomposes the original problem into several independent sub-problems, i.e., solving smaller sub-problems for each sub-trees.
   2. Do not select u: This decision also decomposes the original problem into several independent sub-problems.
3. Thus, we can design the general form of sub-problems as: selecting independent nodes when the root is seleted or not selected.

Independent Set problem is easy when G is a tree



A dynamic programming algorithm

- General form of sub-problems: to find the maximum weighted independent set $S(u)$ in sub-tree $T(u)$. Here, $T(u)$ refers to the sub-tree with node u as its root.
- There are only two cases: $u \in S(u)$, and $u \notin S(u)$. In the case $u \notin S(u)$, we can include all children of u in $S(u)$.
- Optimal substructure: Let $OPT_{in}(u)$ be the maximum weight when $u \in S(u)$, and $OPT_{out}(u)$ be the maximum weight when $u \notin S(u)$. We have the following recursions:

$$OPT_{in}(u) = w_u + \sum_{\text{v is a child of u}} OPT_{out}(v) \quad (7)$$

$$OPT_{out}(u) = \sum_{\text{v is a child of u}} \max\{OPT_{out}(v), OPT_{in}(v)\} \quad (8)$$

and

$$OPT_{in}(u) = w_u \quad \text{if } T(u) = \{\} \quad (9)$$

$$OPT_{out}(u) = 0 \quad \text{if } T(u) = \{\} \quad (10)$$

Dynamic programming algorithm

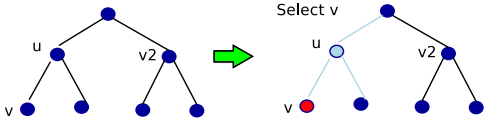
$MaximumIndependentSetDP(T)$

for all nodes in T in posterior order **do**
  if u is a leaf **then**
    $M_{out}[u] = 0$;;
    $M_{in}[u] = w(u)$;;
  else
    $M_{out}[u] = \sum_{v \in son(u)} \max\{M_{out}[v], M_{in}[v]\}$;;
    $M_{in}[u] = w_u + \sum_{v \in son(u)} M_{out}[v]$;;
  end if
end for
return $\max\{M_{in}[root], M_{out}[root]\}$;

Time-complexity: $O(n)$. (Reason: we calculate M_{in} and M_{out} in a bottom-up manner; at each node u, only $O(d)$ time is needed, where d denotes the number of children of u.)

From DP to Greedy when $w(u) = 1, \forall u \in V$.

- Greedy selection: Consider an edge $e = (u, v)$, where v is a leaf. Then there exists a maximum independent set containing v (Why? exchange argument).




```
Greedy_Independent_Set ( F )
S = \Phi;
while F contains at least an edge
do
   Let e=(u,v) be an edge with v as a leaf;
   S = S + {v};
   delete u and v from F;
endwhile
Adding all remaining nodes to S;
return S;
```

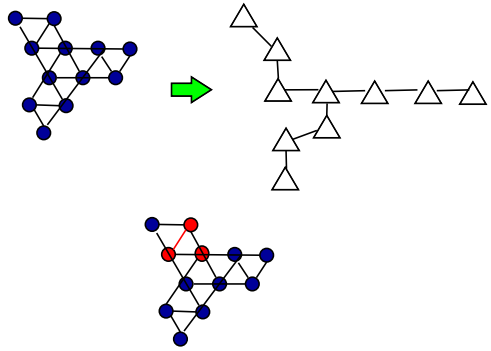
- Note: removing node u may change a tree to a forest.
- Time complexity: $O(n)$ again.

Solving NP-Hard problem on tree-like graph: the connection among sub-problems is small

Extension of tree: "tree-like" graph

- Why the IndependentSet problem becomes tractable when the underlying graph is a tree?
- Reason: the communication between sub-problems is broken by removing ONE node, i.e., the sub-problem are independent.
- A weak question: If the removal of ONE node cannot completely cut all communication between sub-problems, can we achieve this goal through removing a SMALL SET of nodes?

Tree-like graph



- Observations:
  1. There might be many cycles in NODE level; however, it is a tree in TRI-ANGLE level.
  2. Removal of a NODE cannot generate independent sub-problems; however, removal of a TRI-ANGLE will decompose the graph into parts.

How to change a graph into tree? Tree-decomposition

Definition (Tree decomposition)

- A tree-decomposition of a graph $G = (V, E)$ is a tree T, where each node t of T corresponds to a subset $V_t \subset V$ (called "pieces" of G). T and V_t must satisfy the following properties:
  1. (Node coverage) Every node of G belongs to at least one piece V_t.
  2. (Edge coverage) The two end nodes of an edge should be covered by a piece V_t.
  3. (Coherence) In tree T, if t_2 is in a path from t_1 to t_3, then $V_{t_1} \cap V_{t_3} \subseteq V_{t_2}$.

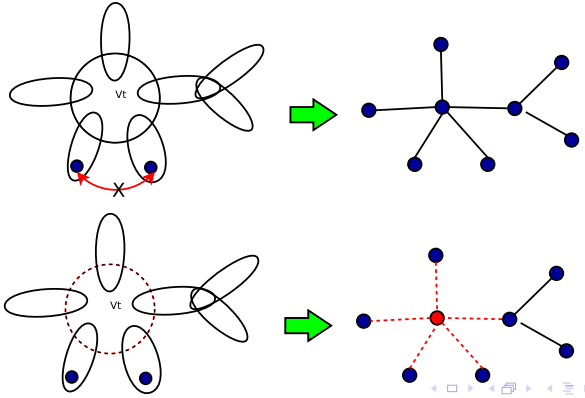
Properties of tree-decomposition

- The important property of tree that makes things easier:
  1. Removal an edge: usually decompose the tree into two INDEPENDENT parts;
  2. Removal a node: usally decompose the tree into a forest containing INDEPENDENT trees;
- The coherence property ensure these two properties for the tree-decomposition T of a graph G.

Removing a node: generating independent sub-problems

Theorem

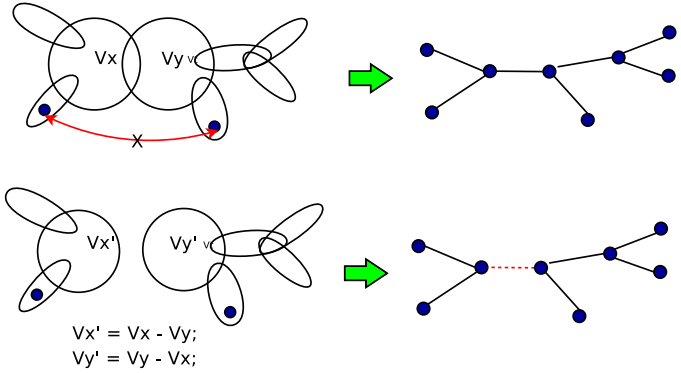
(Removing a node) Suppose $T - t$ (a forest) consists of trees $T_1, T_2, ..., T_d$. Then the subgraph $G_{T_1} - V_t$, $G_{T_2} - V_t$, ..., $G_{T_d} - V_t$ share no common nodes, and there is no edge connecting any two subgraphs.



Removing an edge: generating independent sub-problems

Theorem

(Removing an edge) Suppose $T - \{e\}$ has two branches, namely X and Y. Correspondingly, $G - \{V_x \cap V_y\}$ has two subgraphs: $G_X - \{V_X \cap V_y\}$, and $G_Y - \{V_X \cap V_Y\}$. The two subgraphs share no common nodes and no edge can connect them.



Tree-width: an intrinsic property of a graph

- Every graph G has a tree-composition. (A trivial tree-decomposition: T has only one node t, and $V_t = V$. (See an extra slide)
- We are interested in the tree-composition with small pieces. (Intuition: removal of a piece will break the graph into INDEPENDENT parts.)

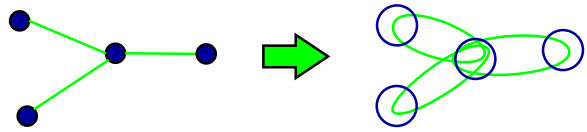
Definition (Tree-Width)

$TreeWidth(G) = \max_t |V_t| - 1.$

Note: Why -1? Just to make the tree-width of a tree be 1.

An example: tree-decomposition of a tree

- Tree-decomposition of a tree $G = (V, E)$:
  - For each node $u \in V$, create a piece V_u, and for each edge e, create a piece V_e.
  - Connecting two pieces if they share a common node.
  - Verify the three requirements and $TreeWidth(G) = 1$.



Theorem

A connected graph G has a width of 1 iff G is a tree.

Proof.

- Suppose G is not a tree but $width(G) = 1$.
- There is a cycle C in G. Consider two edges $e = (u, v)$ and $e' = (u', v')$. The corresponding pieces are denoted as V_e and $V_{e'}$.
- There should be a path connecting V_e and $V_{e'}$ in T. (by the connection property of tree.)
- Consider a path (x, y) in the path, and $V_x \neq V_y$. Thus $V_x \cap V_y \leq 1$.
- Thus the removal of $V_x \cap V_y$ will decompose C into two INDEPENDENT parts. (by the previous Theorem)
- However, we cannot decompose a cycle into two INDEPENDENT part through removing ONLY one edge.

□

Fact: Suppose H is a subgraph of G. We have $width(H) \leq width(G)$. (Argument: constructing T_H based on T_G: $V_t^H = V_t^G \cap H$. Verify the three requirements.)

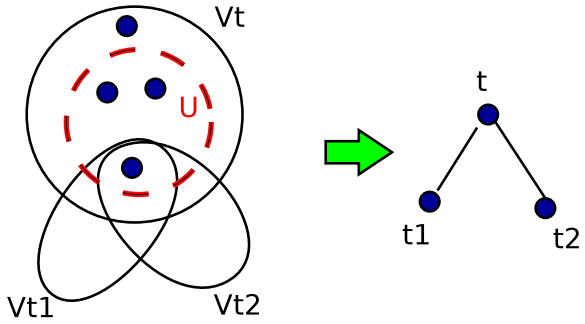
F

inding maximum independent set in a general graph

Finding maximum independent set in a general graph I

- Basic idea:
  - For a general graph $G = (V, E)$, we construct its tree-decomposition T first.
  - Let's travel T in the bottom-up manner
  - Consider node t of tree T. The maximum independent set intersects the piece V_t onto a subset U.
  - However, we have no idea what nodes that U has.
  - Solution: enumerate all possibility of U within V_t. This operation costs at most 2^{w+1} time.
  - The properties of T ensure that the sub-problems corresponding to sub-trees of t can be INDEPENDENTLY solved as soon as U is determined.

Finding maximum independent set in a general graph II



- Key observation: (sub-problem) For each sub-tree T_t, we define a FAMILY of sub-problems: for each subset $U \in V_t$, we use $f_t(U)$ to denote the value of the maximum independent set in G_t, where U is an independent set.

Finding maximum independent set in a general graph III

- Optimal substructure:
  $f_t(U) = w(U) + \sum_{i=1}^{d} \max\{f_{t_i}(U_i) - w(U_i \cap U)\}$, where $U_i \cap V_t = U \cap V_{t_i}$ and $U_i \subset V_{t_i}$ is an independent set.

```
Tree_Decomposition_DP ( G, T )

for all node t of T in post-order
do
    if t is a leaf
    then
        for each independent set U of Vt
        do
            f_t(U) = w(U);
        endfor
    else
        for each indepdendent set U of Vt
        do
            f_t(U) = w(U) + \sum_{i=1}^d ( f_ti(Ui) - w(U_i and U );
            // Ui and Vt = U and Vti
        endfor
     fi
endfor
```

Finding maximum independent set in a general graph IV

- Time-complexity: $O(2^w wn)$.
- Note: the practical graph, say network and residue contact map, usually have small width.