Deep of Enumeration Algorithms

2. Amortized Analysis on Enumeration Algorithm

- Mechanism of amortization
- Basic (toy) case (elimination ordering)
- Local amortization (path)
- Biased (general) case (matching, k-subtree)

2-1 Better Analysis

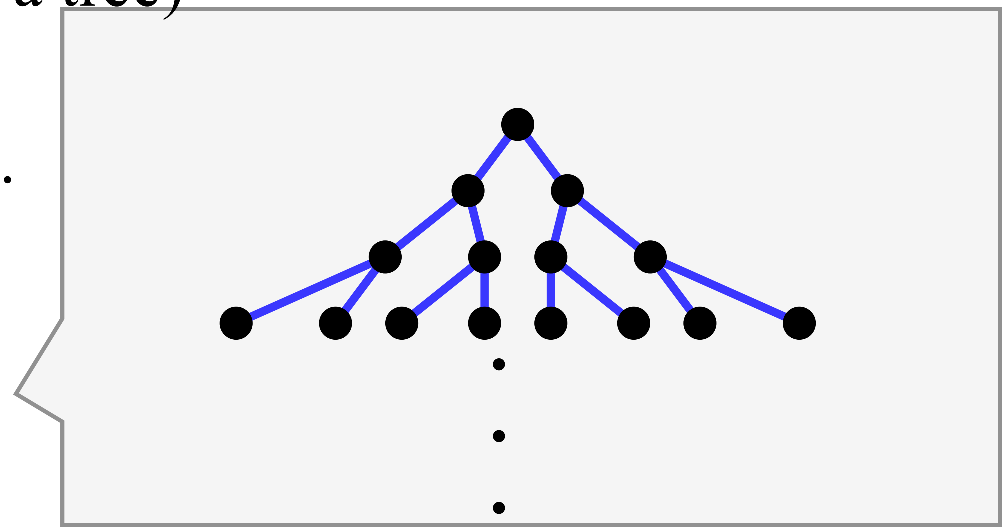
There is an Algorithm

- Suppose that there is an enumeration algorithm **UNO**.
  we want to know time complexity of **UNO** (output polynomiality)

- What is needed?
- What will we obtain as a result?

- We assume that **UNO** is a tree-shaped recursion algorithm
  (the structure of the recursion is a tree)

and the problem is combinatorial.
(has at most **2n** solutions)

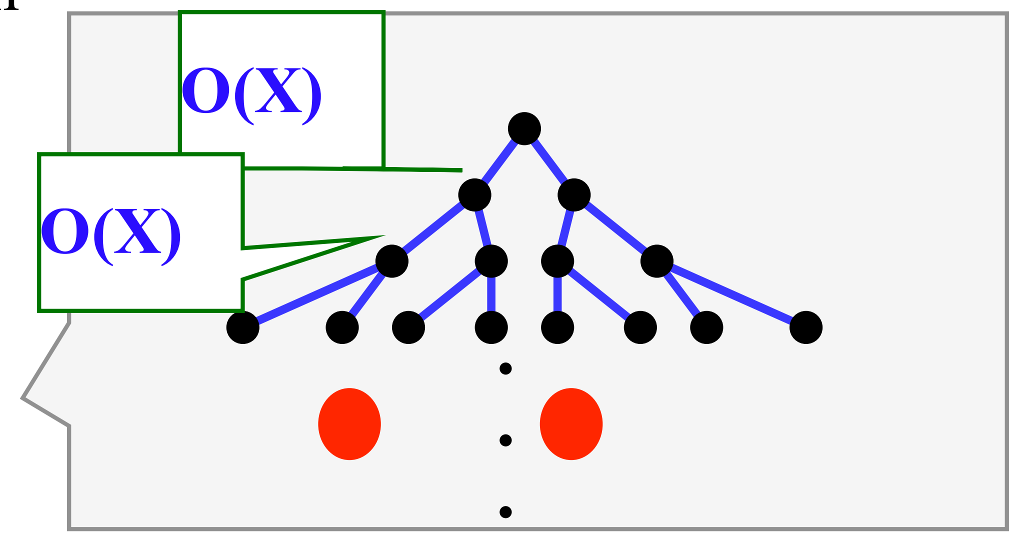


Iteration = O(X)

- We now know that each iteration takes **O(X)** time.
  Can we do something?

□ No. Possibility for
<u>"exponentially many iterations, with few solutions"</u>

ex) feasible solutions for SAT,
with branch-and-bound algorithm

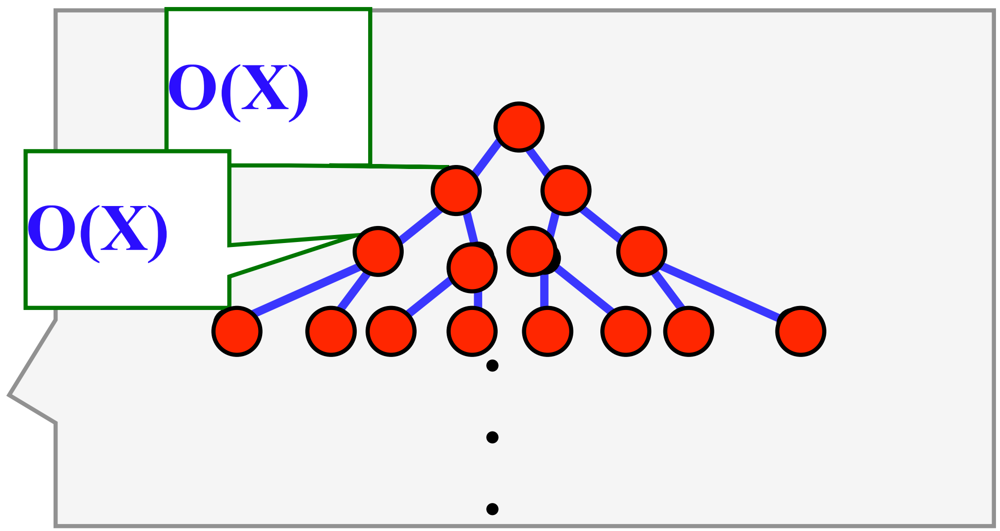


Solution for Each

- We now know that each iteration outputs a solution.
  Can we do something?

□ Yes! **#solutions = #iterations**

"O(X) time for each solution"

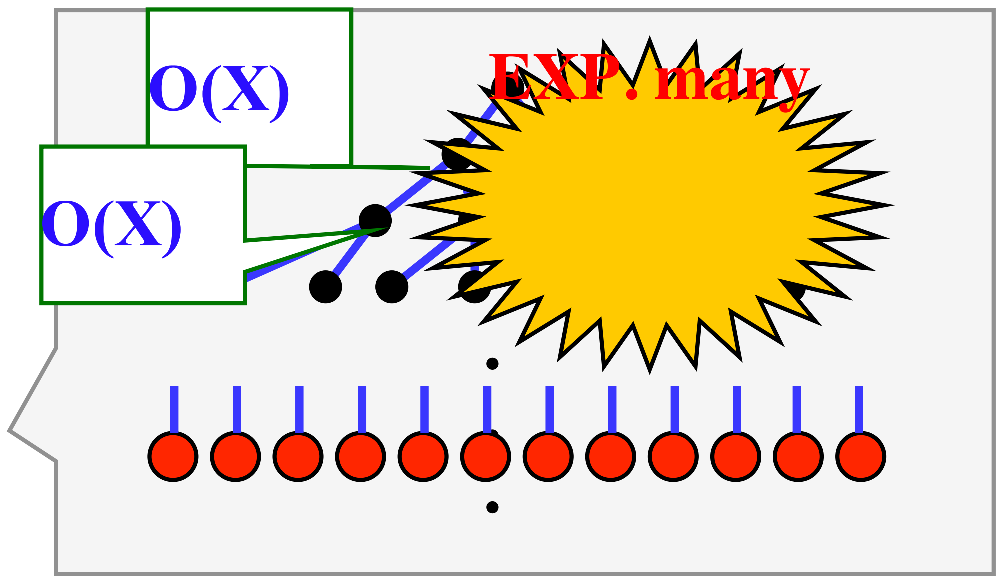


Solutions at Leaves

- We now know that at each leaf, a solution is output.
  Can we do something?

ex) s-t paths, spanning trees, …

□ No. Possibility for
<u>"exponentially many inner iterations, with few leaves"</u>



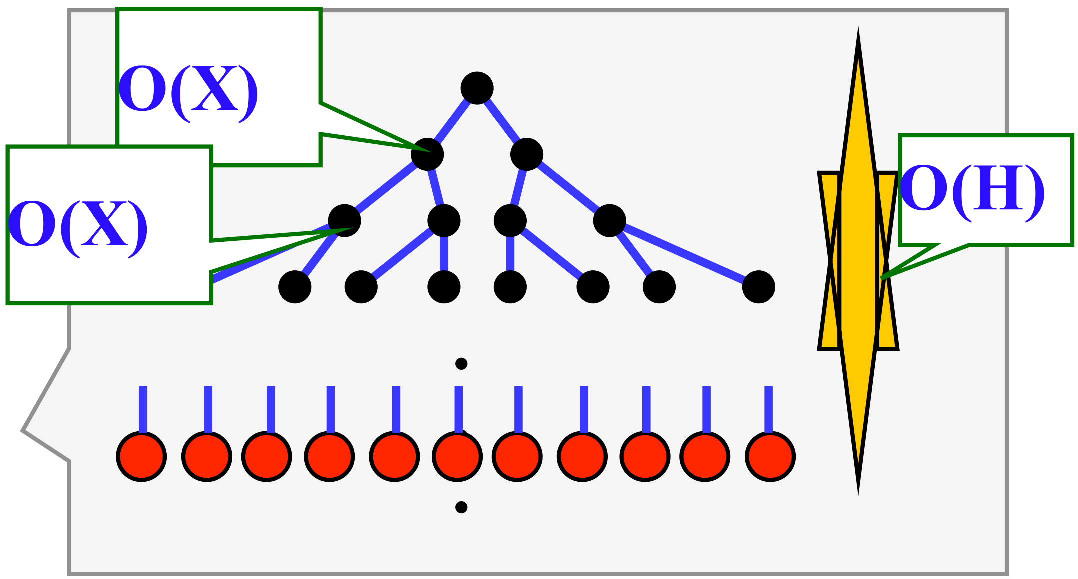
Bounded Depth

- We now know that the height of recursion tree is at most **H**

Can we do something?

□ YES! **[#iterations]** < **[#solutions]** × [height]

"O(X・H) time for each solution"



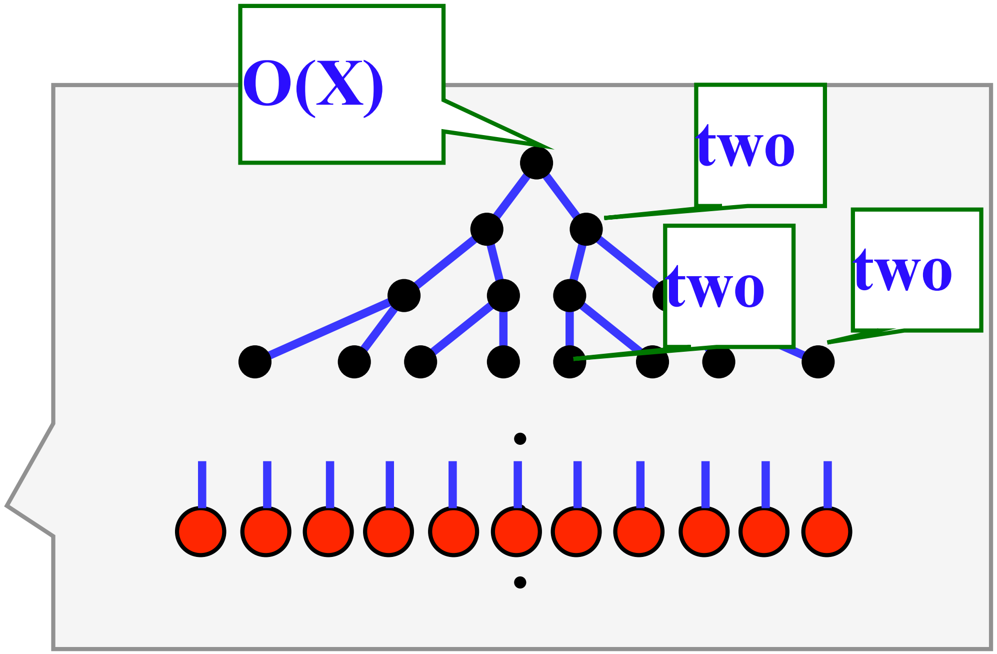
At least Two Children

- Instead of the height, we now know that each non-leaf iteration has at least two children

Can we do something?

□ YES! [#iterations] < 2 × [#solutions]

"O(X) time for each solution"



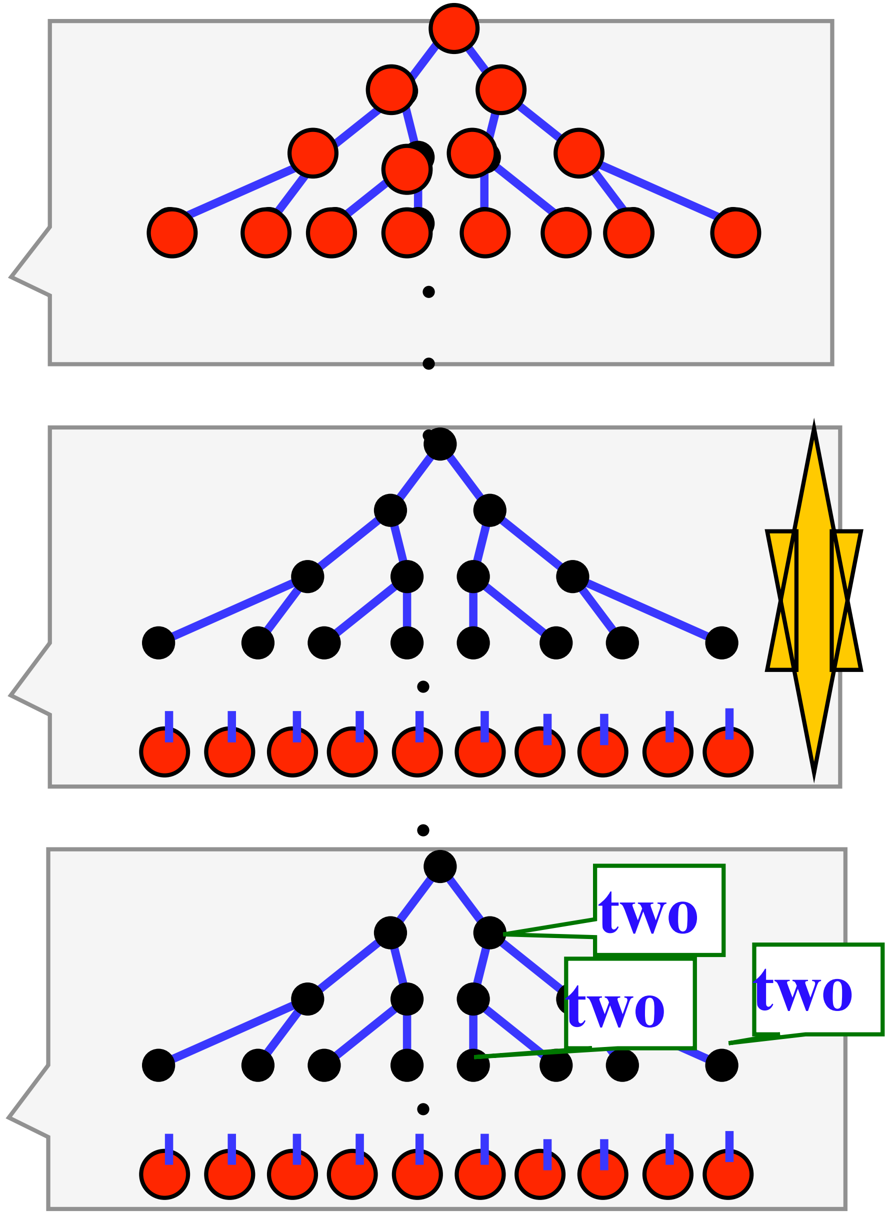
Good Three Cases

- These three cases are typical in which we can bound the time complexity efficiently

- In each case, the time complexity for an iteration depends on maximum computation time on an iteration

- If we want to do better,

we have to use amortized analysis (average computation time of an iteration)



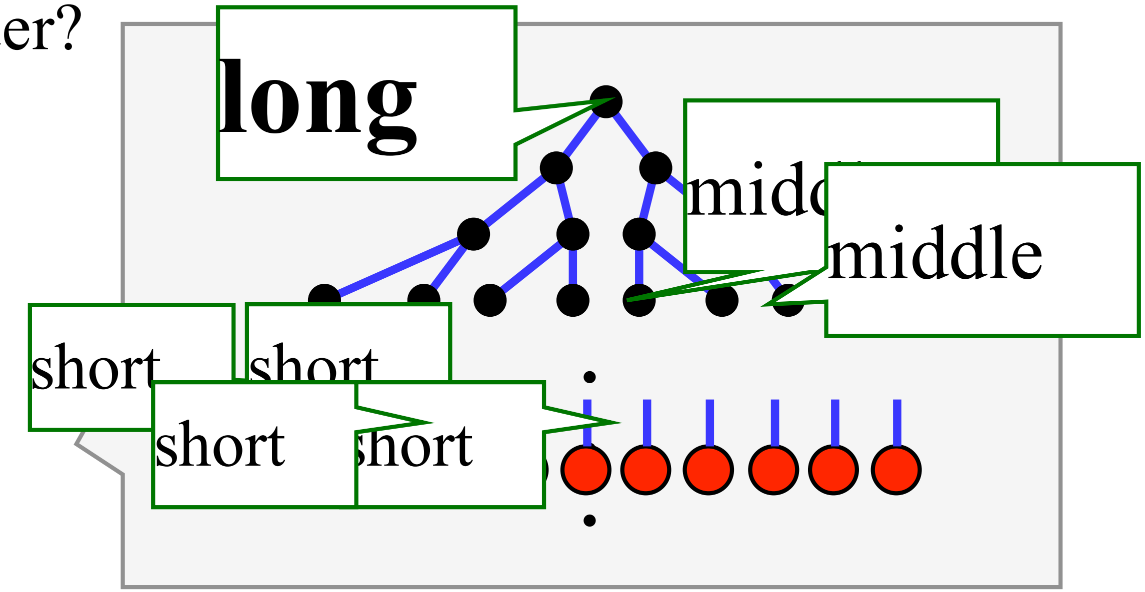
2-2 Basic Analysis

Bottom-wideness

- An iteration of enumeration algorithm generates recursive calls for solving “**subproblems**”
- Subproblems are usually smaller than the original problem

- Many bottom level iterations take short time, few iterations take long time (we call **bottom-wideness**)

- Can we do something better?



Bad Case

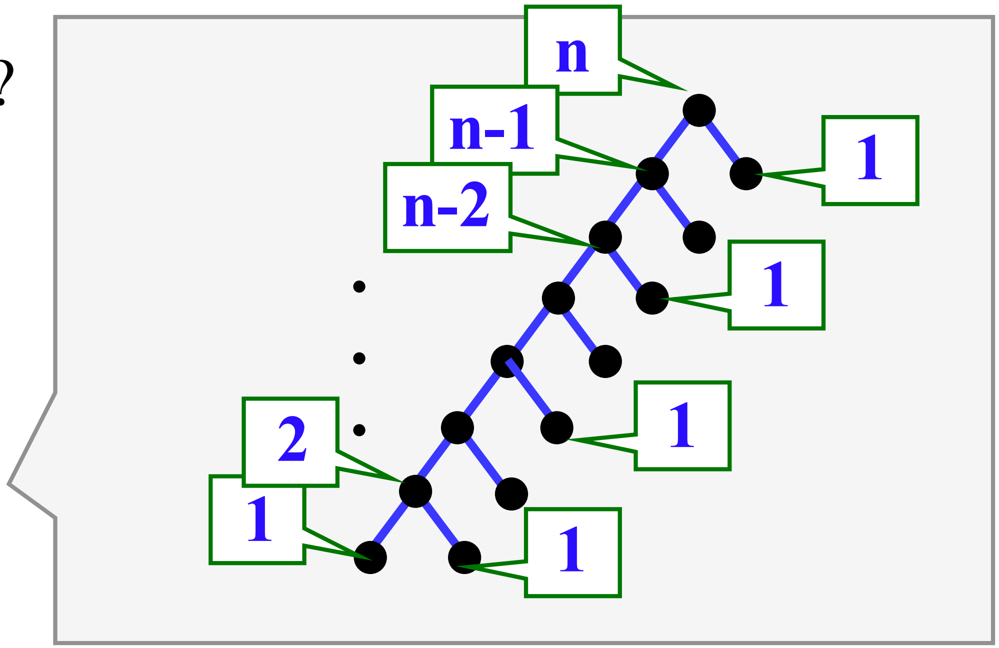
- An iteration of enumeration algorithm generates recursive calls for solving "**subproblems**"
- Subproblems are usually smaller than the original problem

□ Many bottom level iterations take short time, few iterations take long time

- Can we do something better?

□ No. not sufficient

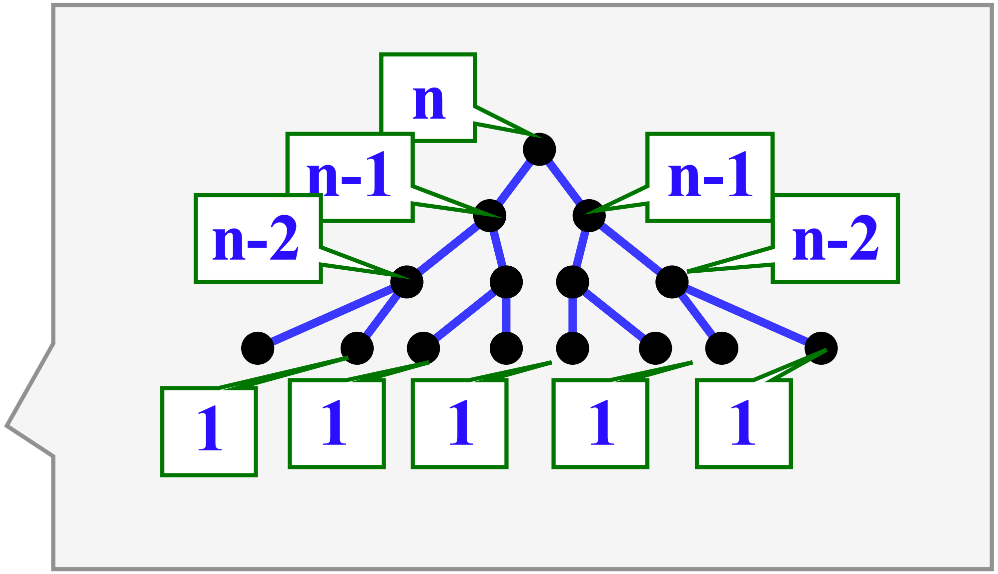
in the right case, an iteration takes **O(n)** time on average



Balanced

- The recursion tree was biased. If balanced?

□ No. not sufficient
in the right case, an iteration
takes **O(n)** time on average



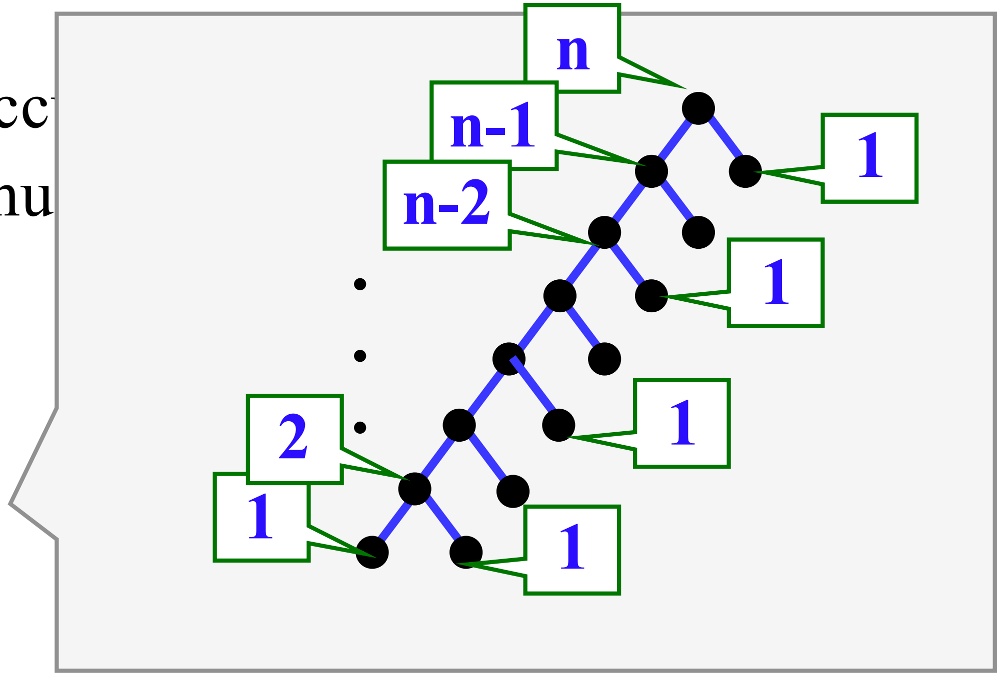
What is sufficient to reduce the amortized time complexity?

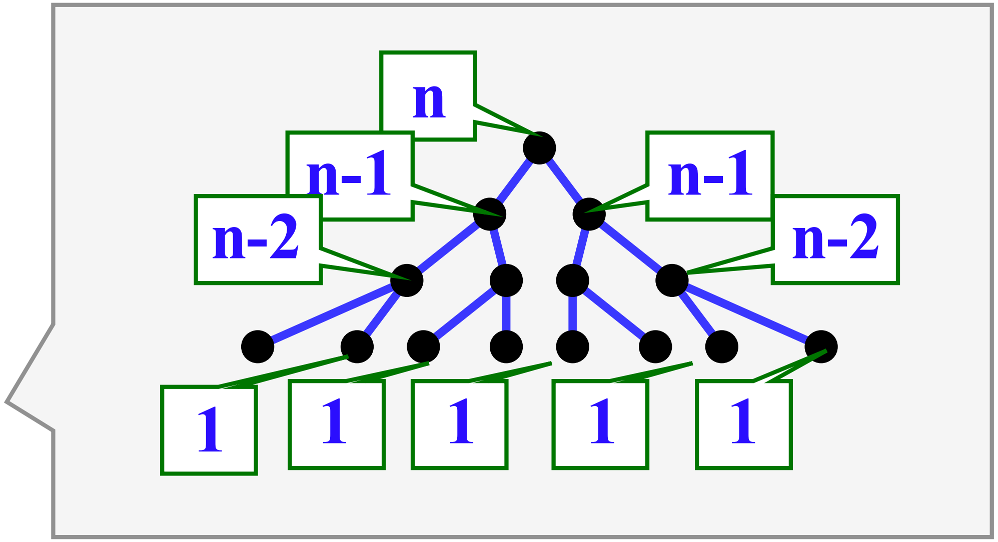
Sudden Decrease

- In the cases, sudden decrease occ
time for parent and child differ mu

□ We shall clarify good characterization for "no sudden decrease"

- Then, what is good?





Toy Case

- If any iteration has two children, and the computation time decreases constantly, the amortized computation time will be reduced

Ex)

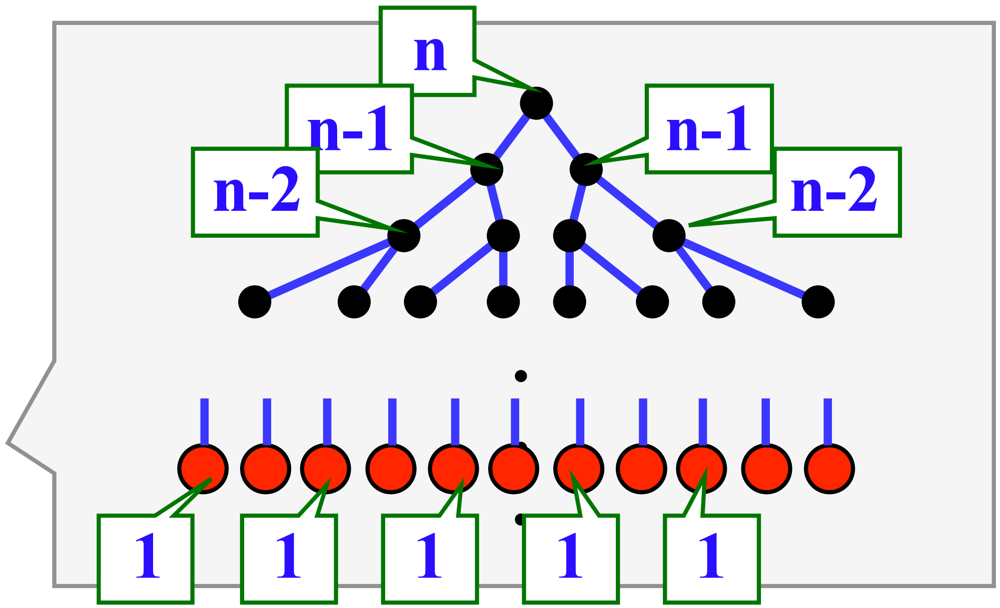
$$\frac{n + 2(n-1) + 4(n-2) + \ldots + 2n-1 \cdot 2 + 2n \cdot 1}{2 \cdot 2n} = O(1)$$

computation time

#iterations

- This holds for any polynomial

$$\{\Sigma\ 2n-i\ poly(i)\ \} / 2 \cdot 2n = O(1)$$



Analysis

- This holds for any polynomial of the form **poly(i) = i2, i3** ,…

  $\{ \Sigma\, 2n\text{-}i \text{ poly}(i) \} / 2 \cdot 2n = O(1)$

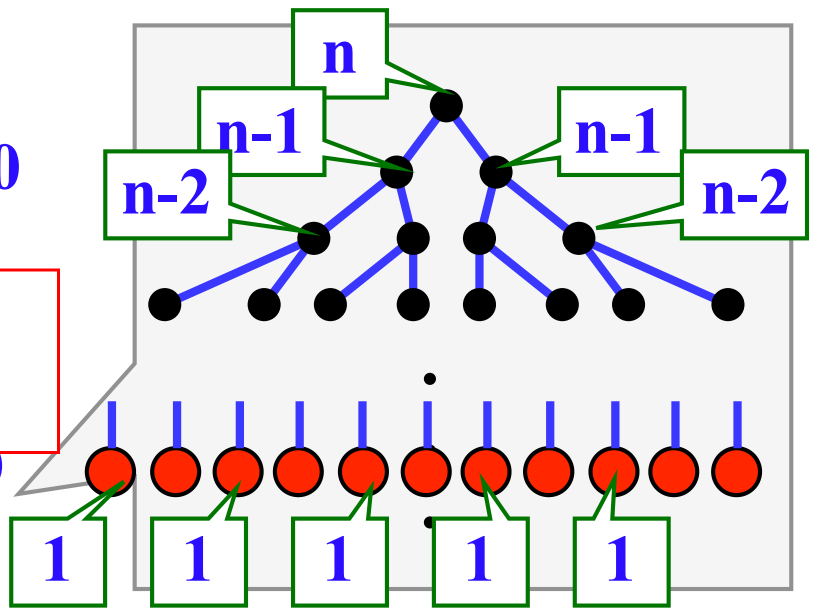
- Compare the computation time on adjacent levels

  $2n\text{-}(i+1) \text{ poly}(i+1) \;/\; 2n\text{-}i \text{ poly}(i) \;=\; \text{poly}(i+1) \;/\; 2 \cdot \text{poly}(i)$

- There are constants $\alpha < 1$ s.t.

  $\text{poly}(i+1) \;/\; 2 \cdot \text{poly}(i) < \alpha$ for any $i > 0$

If bottom level iteration takes 1 unit time,
total computation time $< 2n\,(1 / (1-\alpha))$



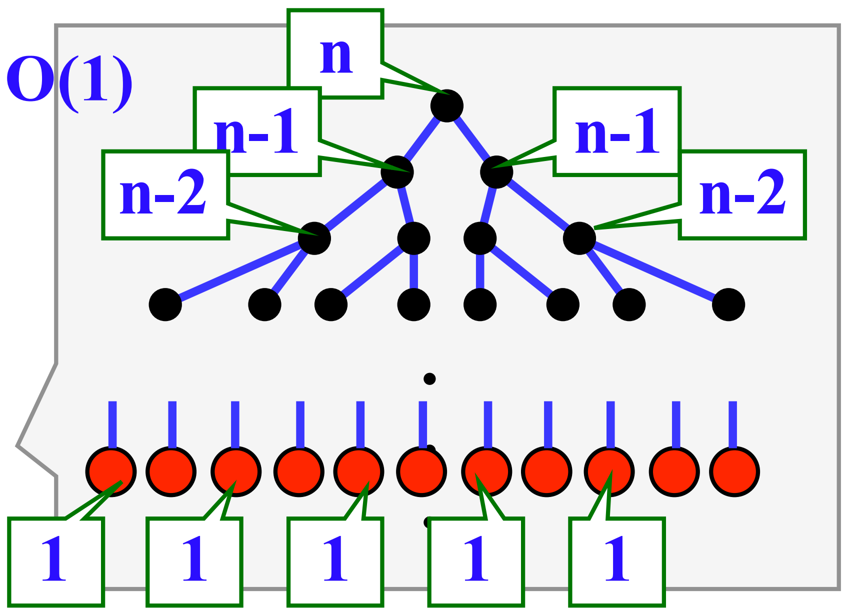
Generalization of the Toy Case

- Assume that **poly(i)** is an arbitrary polynomial
- There are constant **δ** and **α < 1** s.t.

  poly(i+1) / 2•poly(i) < α for any **i > δ**

- When **i < δ**, **poly(i)** is constant, thus any iteration on level below **δ** takes constant time
- For **i ≥ δ** ,

  { Σi ≥δ 2n-i poly(i) } / 2•2n(n-δ) = O(1)



Therefore, amortized computation time for one iteration is **O(1)**

More Than Two Children

- Consider cases that an iteration may generate more than two recursive calls (so, iterations have three or more children)

- Let **N(i)** be the number of iterations in level **i**

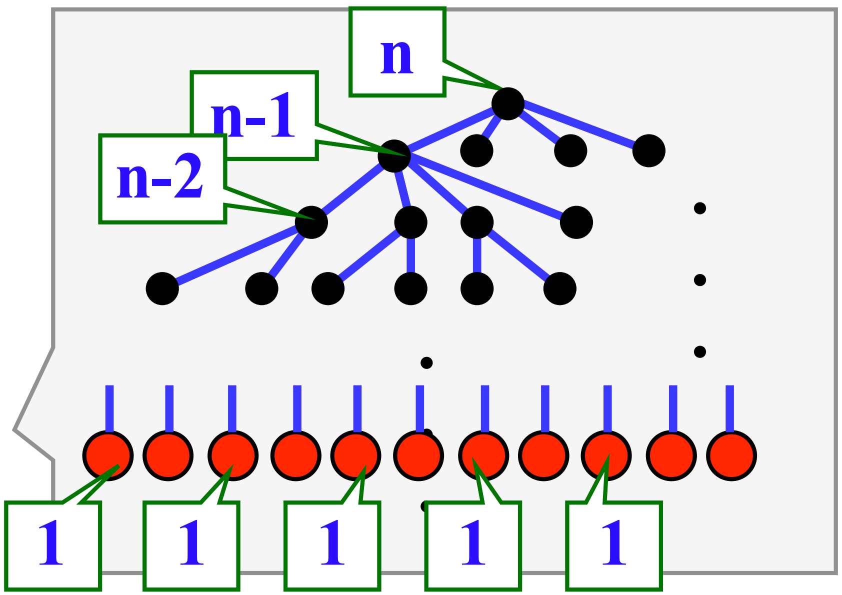
  □ computation time on level **i** is bounded by **Σ N(i) poly(i)**

- Compare adjacent levels

N(i+1) poly(i+1) / N(i) poly(i)

≤ poly(i+1) / 2•poly(i)

Thus, in the same way, we can show amortized computation time for one iteration is **O(1)**

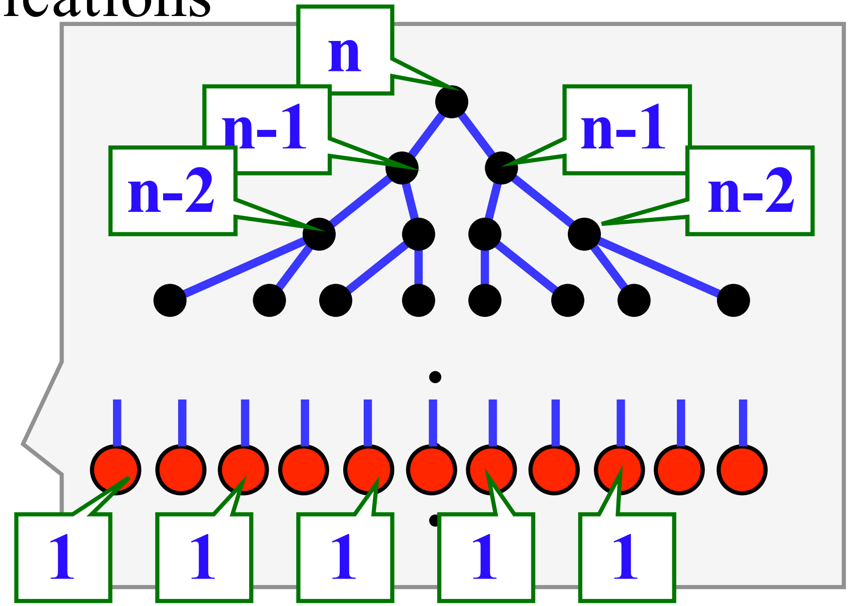


Application

- You may think this is too much trivial to enumeration algorithms

- However, surprisingly, there are applications

- Consider the enumeration of elimination orderings



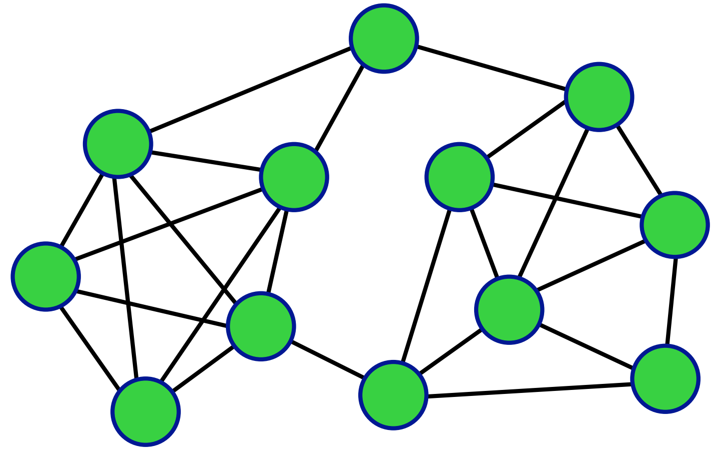
Elimination ordering

for given a structure (graph, set, etc.)

a way of removing its elements one by one until the structure will be empty, with keeping a given property

Elimination Ordering for Connectivity

Ex) For given a connected graph **G=(V,E)**, remove vertices one by one with keeping the connectivity



- We can enumerate this elimination ordering by simple backtracking

- Each iteration takes **O(|V|2)** time

Iter (**G**, **X**)
1. **if** **G** is empty **then output** **X**
2. **for** each vertex **v** in **G**,
    if **G-v** is connected **then call** **Iter** (**G-v**, **X+v**)

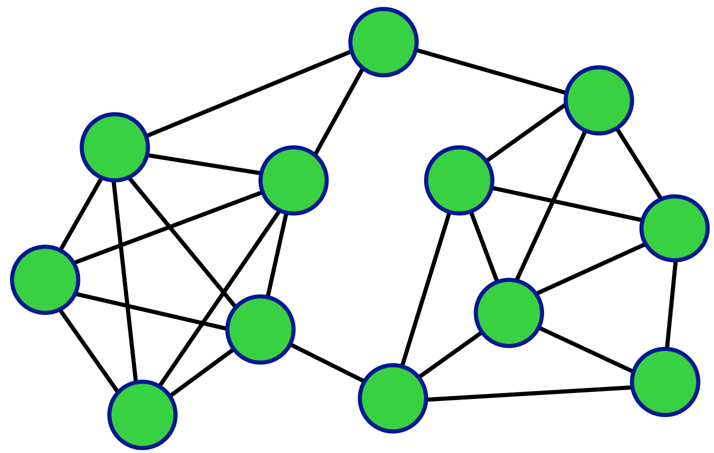
Necessary Condition

(1) For any connected graph, there are at least two vertices whose removals are connected

- Each (internal) iteration has at least two children

(2) Computation time on an iteration in level **i** is **O(i2)**

- Amortized computation time of
  and iteration is **O(1)** time



Iter (**G**, **X**)
1. **if** **G** is empty **then output** **X**
2. **for** each vertex **v** in **G**
   if **G-v** is connected **then**
   call **Iter** (**G-v**, **X+v**)

Small Pit Falls

Q How to output **X** in **O(1)** time?
 output **X** by the difference from the previously output one

Since the number of additions and deletions is linear in the number of iterations, the amortized output time is **O(1)**

Q How to give **G** and **X** to the recursive call in **O(1)** time?
 always update them, and give them by pointers to **G** and **X**

Before the recursive call, we remove **v** and adjacent edges from **G**, and add **v** to **X**

After the recursive call, we add **v** and adjacent edges to **G**, and remove **v** from **X**. This doesn't increase the time/space complexity

Other Elimination Ordering

- There are many kinds of elimination orderings
  + perfect elimination ordering (chordal graphs)
  + strongly perfect elimination ordering (chordal graphs)
  + vertex on the surface of the convex hull (points)

    …

  + edge coloring of bipartite graph can be also solved

- At least, there proposed constant time algorithms for the first two (technical, to achieve amortized constant time for each iteration)

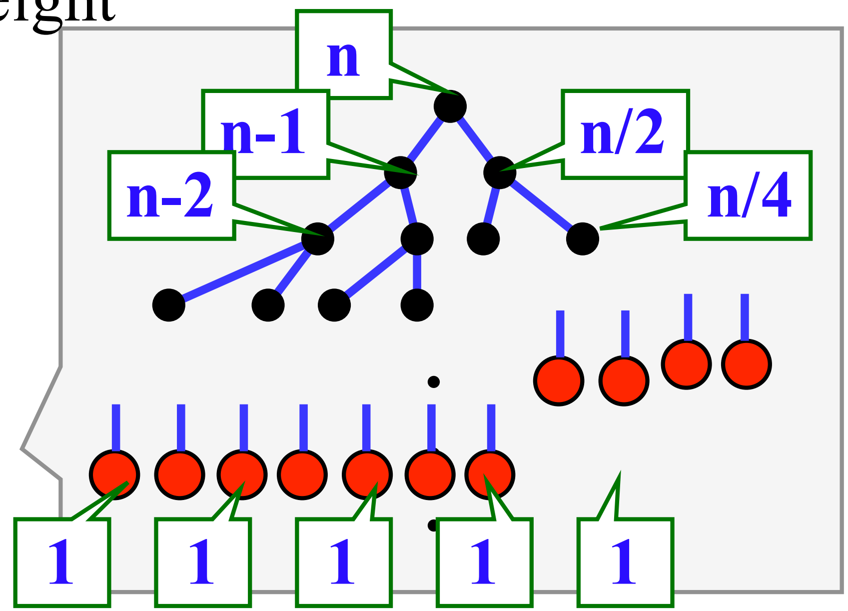
- We can have the same results with very very simple algorithms

2-3 Amortize by Children

Biased Recursion Trees

- The previous cases are something perfect
  + the height (depth) is equal at everywhere
  + computation time depends on the height

- We want to have stronger tools that can be applied to biased cases



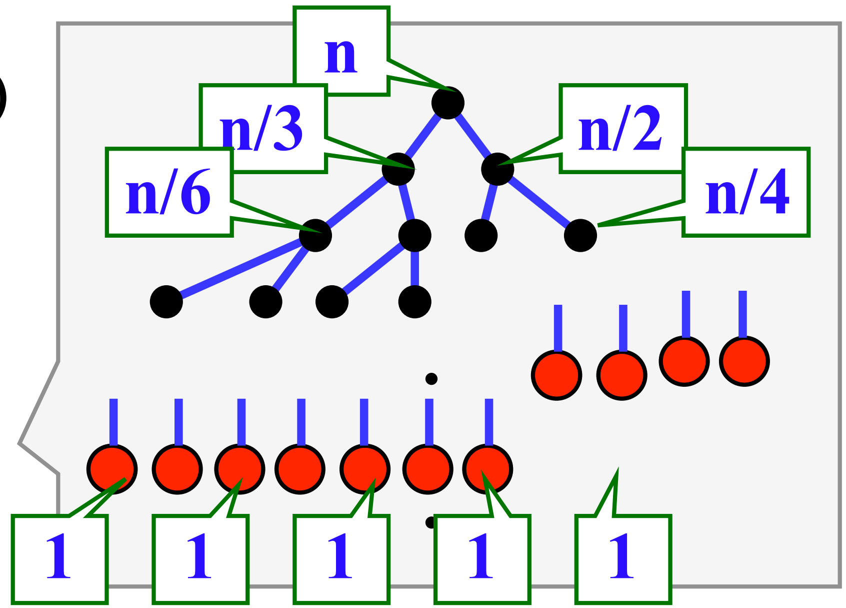
Well-known Case

- Let **T(x)** be the computation time on iteration **x**

- If every child takes at most **T(x) / α,** for some **α > 1**
  the height of the tree is **O(log n)**

+ useful in complexity analysis
+ however, **#iterations** is bounded by
  polynomial (not fit for enumeration)

We need another idea



Local Amortization

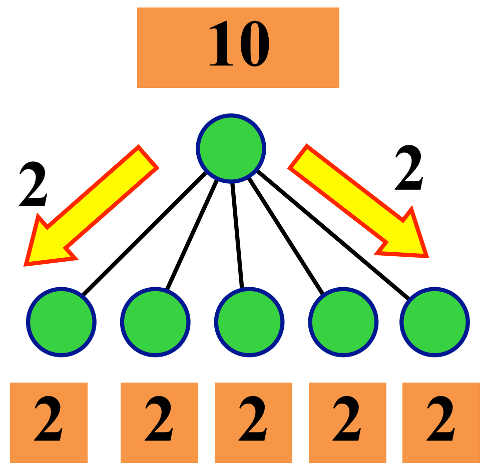
- If **#children** is large, amortized time complexity will be small even though sudden decrease occurs

- Let **|Chd(x)|** be **#children** of iteration **x**,
and assign computation time **T(x)** to its children

 each child receives **T(x) / |Chd(x)|**

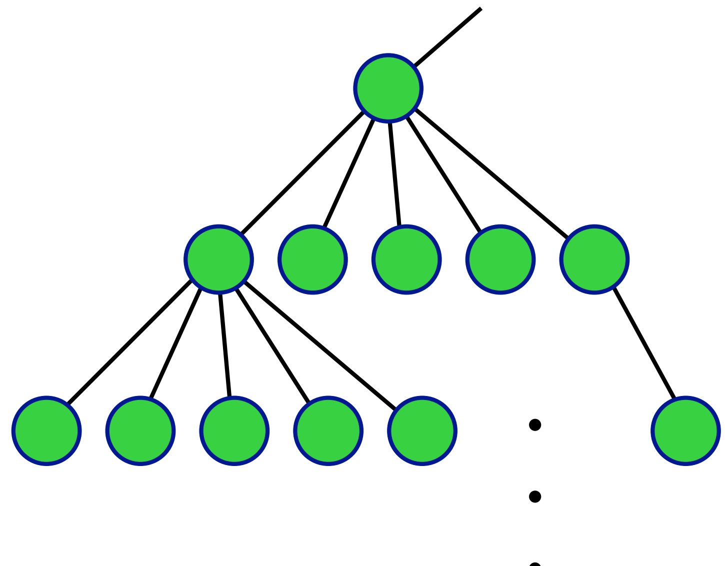
- The time complexity of an iteration is
O(max x { T(x) / (|Chd(x)| + 1) })

- We can use **#grandchildren** instead of **#children**



Estimating #(Grand)Children

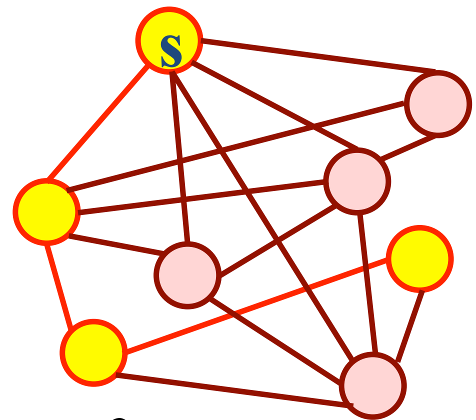
- This analysis needs to estimate **#children** (and **#grandchildren**) this will be a technical part of the proof
  - + estimate by the degree of the pivot vertex
  - + #edges in a cycle
  - + #edges in a cut…



Enumeration of s?-path

Problem: given a graph **G=(V,E)**, and a vertex **s**, enumerate all simple paths one of whose end is **s**

- Simply, by back tracking, we can solve



Iter (**G=(V,E)**, **t**, **X**)
1. **output X**
2. **for** each vertex **v** in **G** adjacent to **s**
    call Iter (**G-t**, **v**, **X+v**)

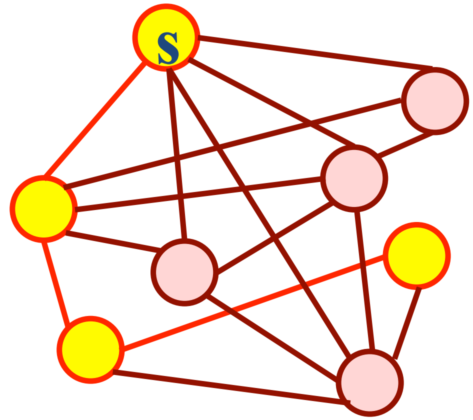
- Each iteration takes **O(d(t))** where **d(t)** is the degree of **t**

  □ the time complexity of an iteration is **O(|V|)**

Amortization

+ Each iteration takes **O(d(t))** time
+ Each iteration generates **d(t)** recursive calls

- Thus, **max x { T(x) / (|Chd(x)| + 1) } = O(1)**, and the amortized time complexity of an iteration is **O(1)**




```
Iter (G=(V,E), s, X)
1.  output X
2. for each vertex v in G adjacent to s
      call Iter (G-s, v, X+v)
```

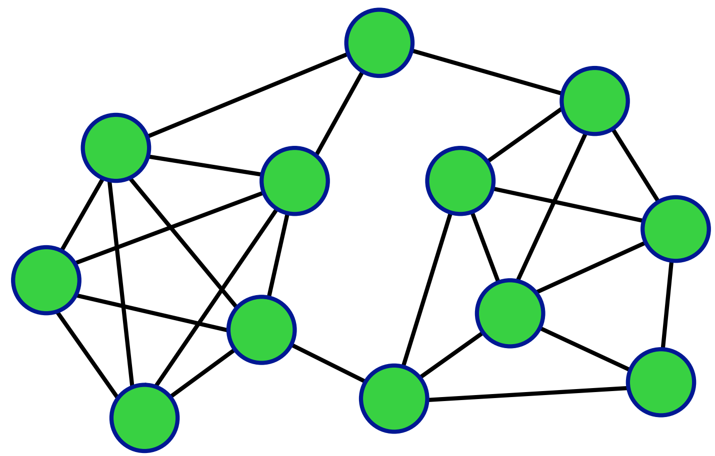
Other Problems

- By using **#grandchildren**, the complexity on the enumeration algorithms for the following structures are established

\+ spanning trees of a given graph **O(|V|) □ O(1)**

\+ trees of size k in a given graph **O(|E|) □ O(k)**

etc…



2-4 Push out Amortization

Computation time “Increases”

- In the “toy” cases, the key property was that
  “the total computation time on each level increases with a
  constant factor, by going to a deeper level “

$$2n-(i+1)\ \mathrm{poly}(i+1) \quad / \quad 2n-i\ \mathrm{poly}(i) \quad = \quad \mathrm{poly}(i+1) \quad / \quad 2\cdot\mathrm{poly}(i)$$

- It seems that “increase of computation time is good for us”
  (it implicitly forbids “sudden decrease”)

- Since the tree is biased, apply this idea locally
  --- parent and child (or descendants)

Local Increase

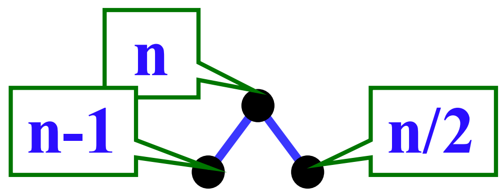
- In the "toy" cases, we compare the total computation time on a level and that on the neighboring level

- Instead of that, we compare the computation time of a parent, and the total time on its children

□ the condition of the toy case is implemented as follows

$$\Sigma_{\text{child } z \text{ of } x} T(z) \geq \alpha T(x) \qquad \text{for some } \alpha > 1$$

We will characterize good cases by this condition



Formula

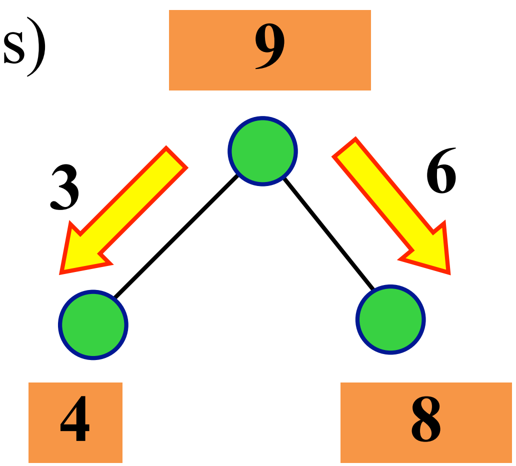
- **T*** : the maximum computation time on a leaf
- **C(x)** : sum of the computation time of children of **x**
- a constant **α > 1** s.t., for any (inner) iteration **x**, **C(x) ≥ αT(x)**

Theorem: Amortized computation time of each iteration is **O(T*)**

Proof: give one's computation time to its children, so that each child **z** receives **T(x) • { T(z) / C(x) }**

(just move, for analysis)

Each child gives the received computation time and its own computation time to its children (so, grandchildren)



Induction

- Each child gives the received computation time and its own computation time to its children (so, grandchildren)

Claim: under this rule, any iteration **z** receives

at most **$T(z) / (\alpha-1)$** from its parent

(intuitively, **$T(z) / \alpha$** from parent, **$T(z) / \alpha 2$** from grandparent, ...)

- Suppose that an iteration **x** satisfies the condition

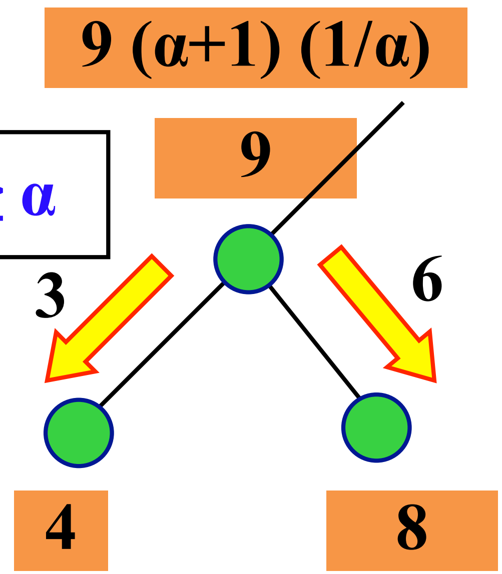
□ Then, its child **z** receives at most

$\{ T(z) / C(x) \} \cdot \{ T(x) + T(x) / (\alpha-1) \}$

$= \{ T(z) / C(x) \} \cdot \{ T(x) \alpha / (\alpha-1) \}$

$\leq \{ T(z) / \alpha \} \cdot \{ \alpha / (\alpha-1) \} = T(z) / (\alpha-1)$

$T(x)/ C(x) \geq \alpha$



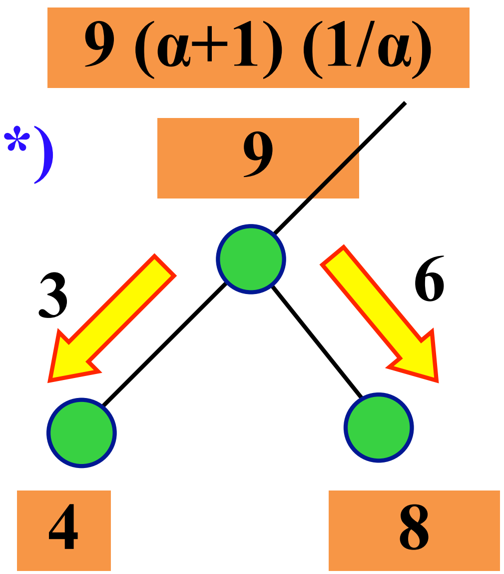
Amortized Time on Leaf

Claim: under this rule, any iteration **z** receives at most **T(z) / (α-1)** from its parent

□ Each leaf receives **T* / (α-1)** time from its parent

- After this move, only leaves have computation time
- Each leaf has **T* + T* / (α-1) = O(T*)** time

□ amortized time complexity of an iteration is **O(T*)**

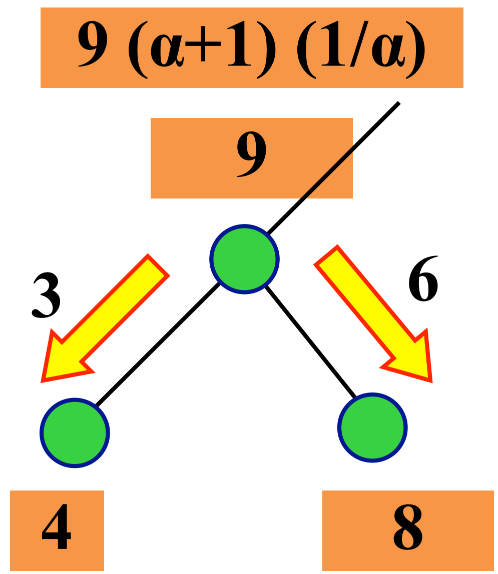


Formulation

- If a recursion algorithm satisfies that

+ the maximum computation time on a leaf is **T^***
+ there exists a constant **$\alpha > 1$** such that any internal iteration **x** satisfies **Σ child z of x $T(z) \geq \alpha T(x)$**

then, the amortized time complexity of an iteration is **$O(T^*)$**



Modification: Formulation

- For a recursion algorithm and a constant **α > 1**, if any its iteration **x** satisfies either

(1) its computation time is **O(T*)** (leaf, one-child iteration, etc)

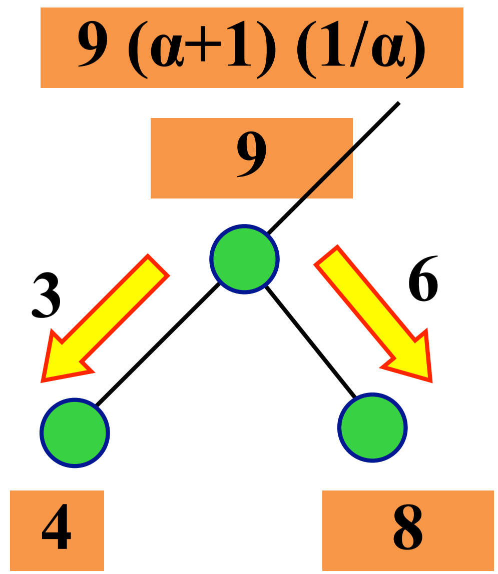
(2) satisfies **Σ child z of x T(z) ≥ αT(x)**

then, the amortized time complexity of an iteration is **O(T*)**

- We can further add the following condition

… or, **(3)** **x** has **Ω(T(x) / T*)** children

(assign **O(T*)** for each child, that will not be given to grand children)

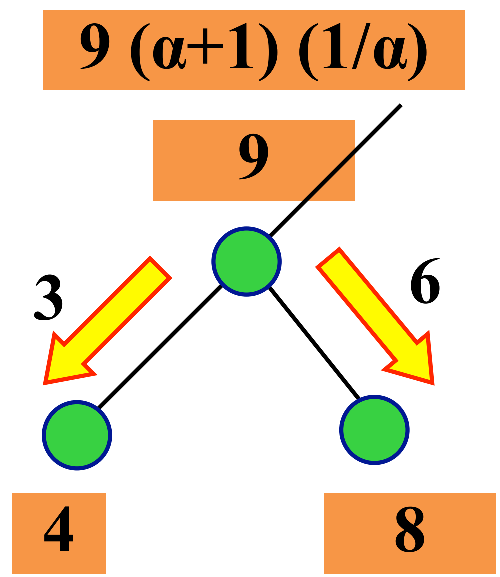


Once More: Formulation

- For a recursion algorithm and a constant **$\alpha > 1$**,
  if any its iteration **x** satisfies either

(1) T(x) = (T*),
(2) Σ child z of x T(z) ≥ αT(x), or **, or**
(3) x has **Ω(T(x) / T*)** children,
or output **Ω(T(x) / T*)** solutions

then, the amortized time complexity of an iteration is **O(T*)**

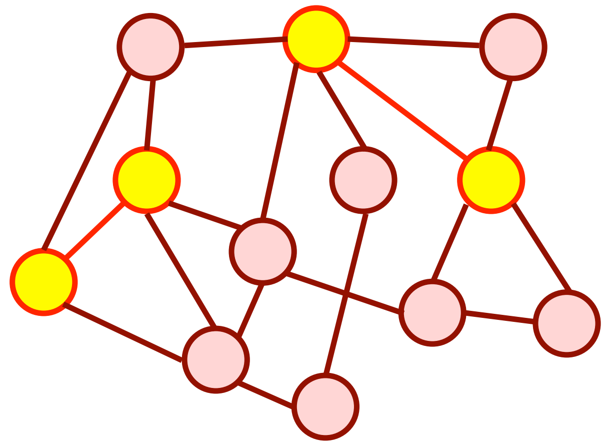


2-5 Matching Enumeration

Example: Enumeration of Matchings

Problem: for given a graph **G = (V, E)**, output all matchings of **G**

matching: an edge subset s.t.
no two edges are adjacent



terms

d(v): the degree of **v**

G-e: the graph obtained from **G** by removing **e**

G+(e): the graph obtained from **G** by removing edge **e** and edges adjacent to **e**

G-u: the graph obtained from **G** by removing vertex **u** and edges incident to **u**

Basic Algorithm

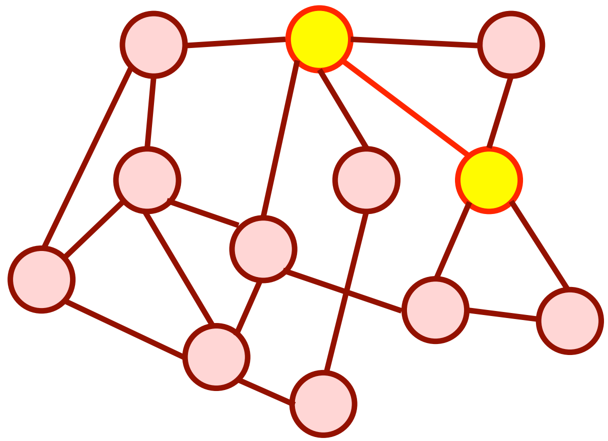
Iter (**G=(V,E), M**)

1. **if E =φ then output M; return**
2. choose an edge **e** of **G**
3. **call Iter** (**G-e**, **M**) // enumerate those not including **e**
4. **call Iter** (**G+(e)**, **M∪e**) // enumerate those including **e**

- Clearly, correct
- An iteration takes **O(|V|)** time
- Leaf iterations output solutions
- Any iteration generates two recursive calls,

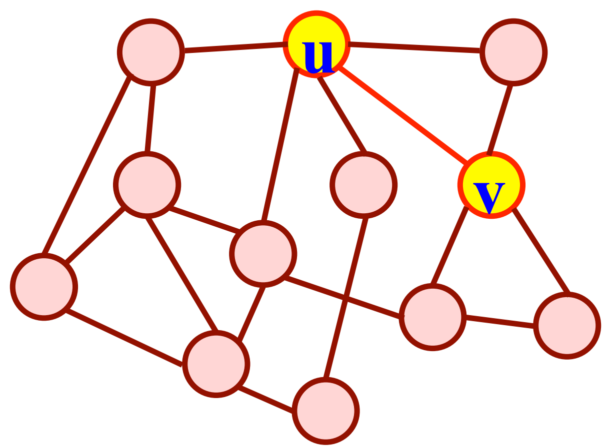
  thus **#iterations** / 2 $\leq$ **#matchings**

Therefore, **O(|V|)** time for each matching



Observation

- An iteration takes **O(d(u)+d(v))** time, in detail (where **e=(u,v)**)
- **#edges** in the input graph of children is at least **|E|-1**, **|E| - d(u) - d(v)**, respectively
- Hereafter, for the sake of clear analysis, we estimate the computation time of an iteration by **c (|E| +1) = O(|E|)**

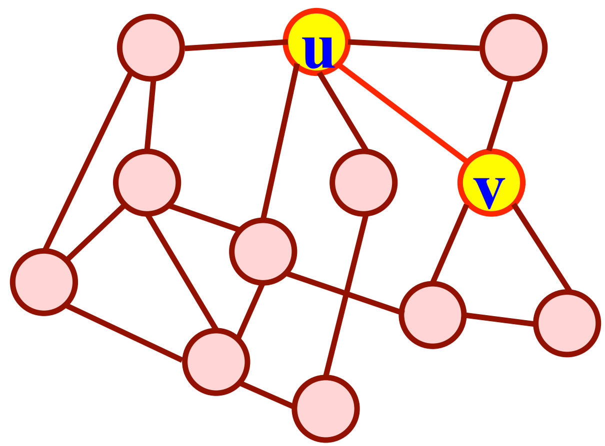


Other Recursion

- For an iteration **x**, if an edge **e=(u,v)** satisfies **d(u)+d(v) < |E| /2**,
  Σchild z of x T(z) ≥ 1.5 T(x) - O(T*) □ **(2) is satisfied**

- Otherwise, there is a vertex **u** s.t. **d(u) ≧ |E| /4**
- We generate recursive calls for all edges incident to **u**

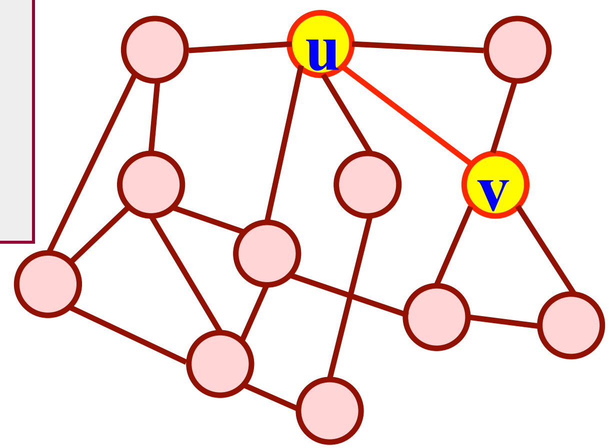
A. choose **u** s.t. **d(u) ≧ |E| /4**
B. **for** each **e=(u,v)**, **call iter** (**G+(e)**, **M∪e**)
C. **call iter** (**G-u**, **M**)



In this case, **|E| /4** recursive calls are generated,
□ **#children** is at least **|E| /4** □ **(3) is satisfied**

Overall Algorithm

```
Iter (G=(V,E), M)
if E =ϕ then output M; return
if an edge e = (u,v) s.t. d(u)+d(v) < |E| /2
   call Iter (G-e, M)
   call Iter (G+(e), M∪e )
else
   choose u s.t. d(u) ≧ |E| /4
   call iter (G-u, M)
   for each e=(u,v), call Iter (G+(e), M∪e)
end if
```



Case Analysis

- For an iteration **x**

+ if **x** is a leaf, then **T(x) = O(T*)** □ **(1) is satisfied**

+ otherwise, if an edge **e=(u,v)** satisfies **d(u)+d(v) < |E| /2**,
  Σchild x of X T(z) ≥ 1.5 T(x) - O(T*) □ **(2) is satisfied**

+ otherwise, **|E| /4** recursive calls are generated,
  □ **#children** is at least **|E| /4** □ **(3) is satisfied**

In any case, iteration **x** satisfies either **(1)**, **(2)**, or **(3)**
□ amortized time complexity of iteration is **O(T*) = O(1)**

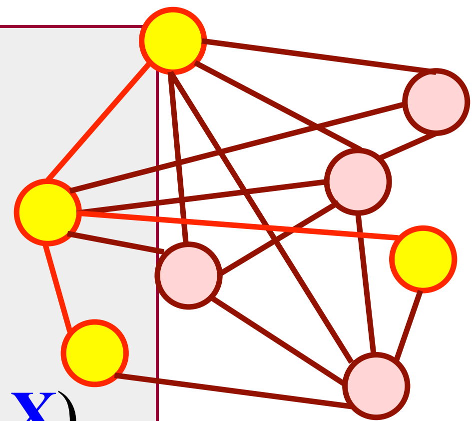
2-6 k-subtree Enumeration

k-subtree

Problem: given a graph **G=(V,E)**, vertex **r** and **k**, enumerate all subtrees of **G** having exactly **k** edges

Iter (**G=(V,E), r, X**)

1. **if |X| = k then output X**; **return**
2. choose an edge **e** incident to **r**
3. **if** the connected component of **G-e** including **r** has at least **k-|X|+1** vertices **then call Iter** (**G-e, r, X**)
4. **call Iter** (**G', r, X∪e**) where **G'** is obtained by contracting **e** and removing selfloops from **G**



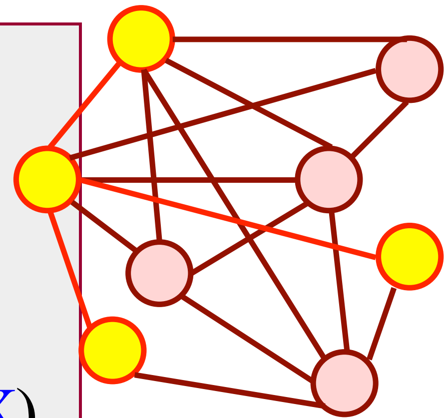
- Correctness is OK. Computation time of an iteration is **O(d(r)+d(v)+ k2)**, thus **O(k (d(r)+d(v) + k2))** per solution

Time and Input

- Rewrite the algorithm

Iter (**G=(V,E), k, r, X**)

1. **if k = 0 then output X**; **return**
2. choose and edge **e** incident to **r**
3. **if** the connected component of **G-e** including **r** has at least **k+1** vertices **then call Iter** (**G-e, k, r, X**)
4. **call Iter** (**G', k-1, r, X**) where **G'** is obtained by contracting **e** and removing selfloops from **G**



- Check in 3 takes **O(|V|+|E|)** time, but can be bounded by **O(k2)**
  - Actually, some parts of **G** is unnecessary
- Sometime, only one recursive call is generated (if **G** is a path)

Speed up by Trimming

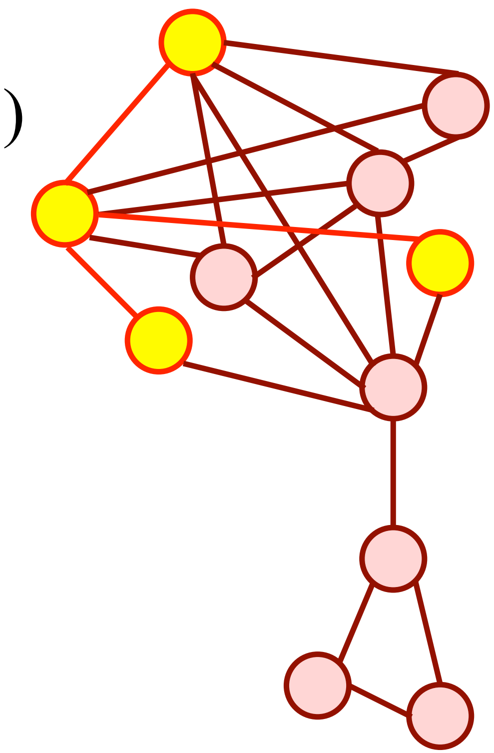
- If the input is small, the computation time will be short
  □ remove unnecessary parts from **G**

- Edges included in no k-subtree is unnecessary

□ edges whose distances to **r** is more than **k** (**k-|X|**)

- Edges included in all k-subtree is redundant

□ Such edges **e** are bridges, and **c(G, e, r) < k+1**, where **c(G, e, r)** is the #vertices in the connected component of **G-e** that includes **r**

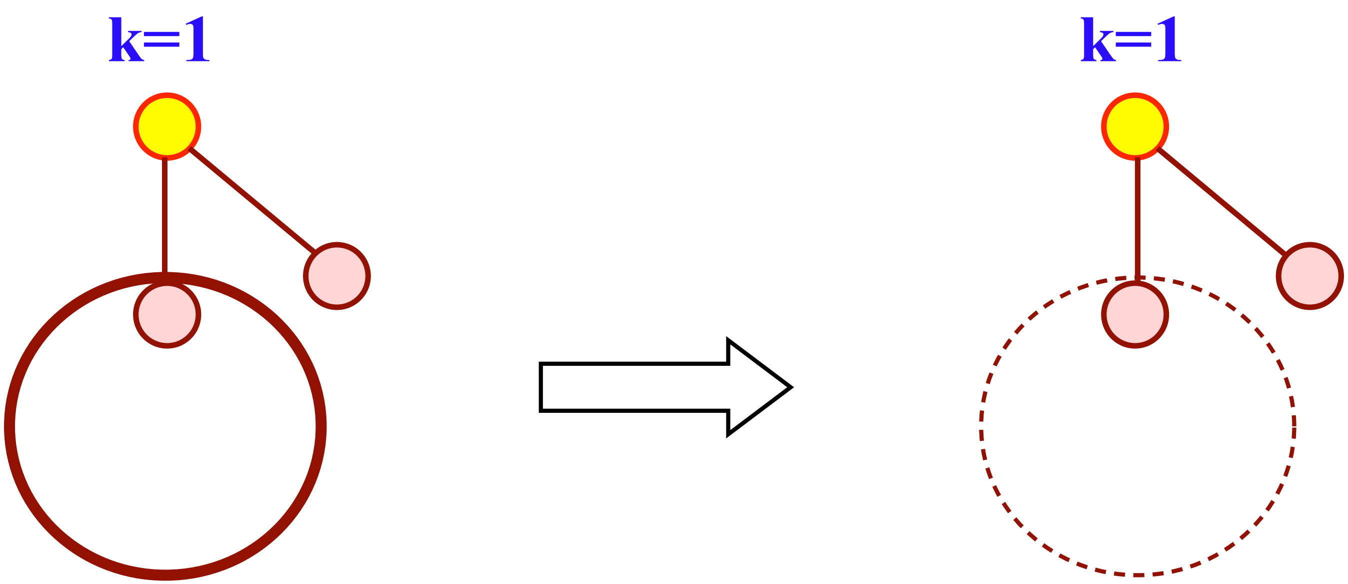


All edges of both types can be found in **O(|V|+|E|)** time

Trimming before Recursive Call

- When we generate a recursive call, we generate its input graph, trim it, and then pass it to the recursive call

- Intuitively, by this trimming, sudden decreases do not occur.
  In precise, there is no case that both children are so small

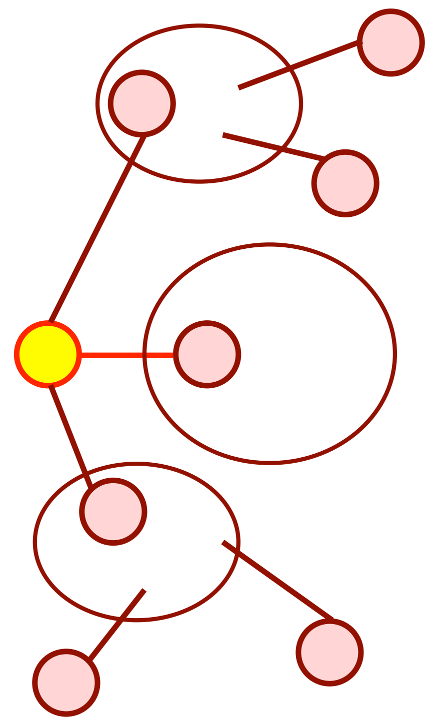


Small Children

- Consider the cases in which child inputs a small graph

- Recursive call for **k**-subtrees not including **e**
  - **(a)** some edges over **e** will be unnecessary
  - **(b)** if **e** is a bridge, some edges not over **e** would be redundant

- Recursive call for **k**-subtrees including **e**
  - **(c)** some edges of distance **k** to **r** becomes unnecessary
  - **(d)** edges parallel to **e** becomes selfloops, so unnecessary



Small Children

- Recursive call for **k**-subtrees not including **e**

 (a) some edges over **e** may never be used

 (b) if **e** is a bridge, some edges not over **e** would be always used

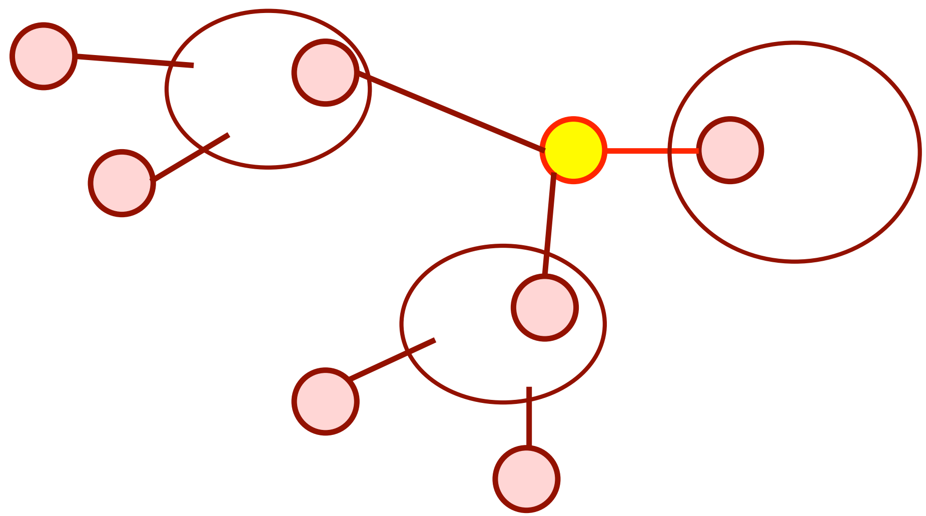
- Recursive call for **k**-subtrees including **e**

 (c) the edges not over **e** of distance **k** to **r** are never used

 (d) edges parallel to **e** becomes selfloops, so are never used

- If there are many these edges, condition **(2)** doesn't hold

- We need a modification so that condition **(3)** will be satisfied



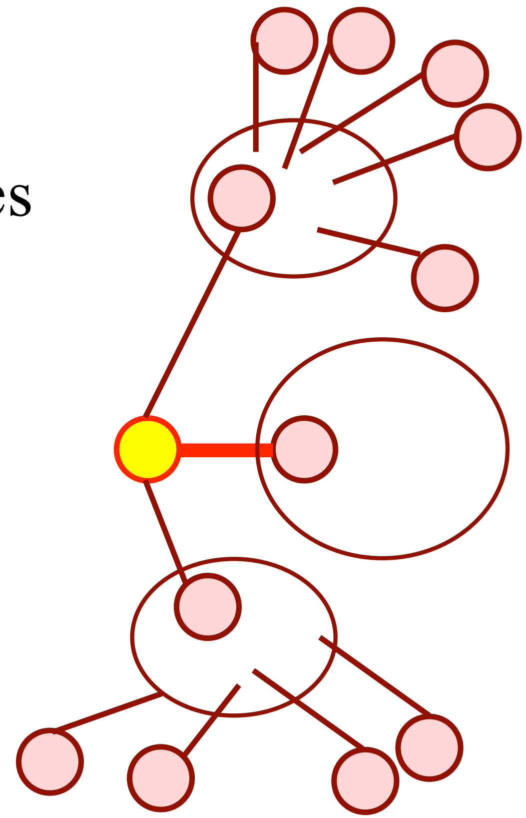
Case (c)

(c) some edges of distance **k** to **r** are never used

+ **k-**subtrees including such edges are all paths,
  whose ends are the edges

+ We enumerate all these paths,
  starting **r** and ending at the unnecessary edges

+ This can be done in **O(1)** time for each path



□ **(3) is satisfied**

Case (d)

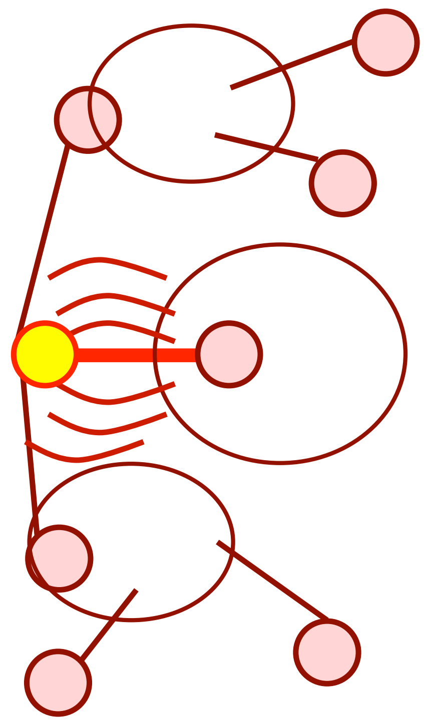
(d) edges parallel to **e** becomes selfloops, so unnecessary

+ We generate all subproblem of
  enumerating **k**-subtrees of including each parallel edge

+ The input graph for each subproblem is equivalent
  to that of **e**

+ Each recursive call is generated in **O(1)** time
  for each

□ **(3) is satisfied**



Cases (a) and (b)

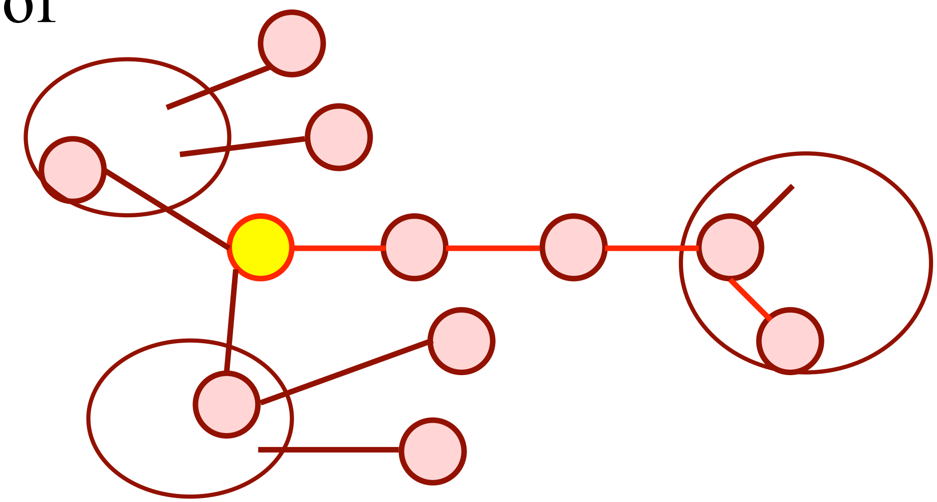
(b) occur only when **e** is a bridge

- We trace **e**, and go further until we meet a 2-connected component, or a vertex of degree 1 (if **e** is in a 2-connectted component, **e = eh**)
- After the meet, we trace one more edge. Obtained path is **e1,e2, …,eh**

- Make subproblems of **k**-subtrees of
  + not including **e1**
  + including **e1** but not **e2**

  . . .

  + including **e1,…,eh-1** but not **eh**
  + including all **e1,…,eh**

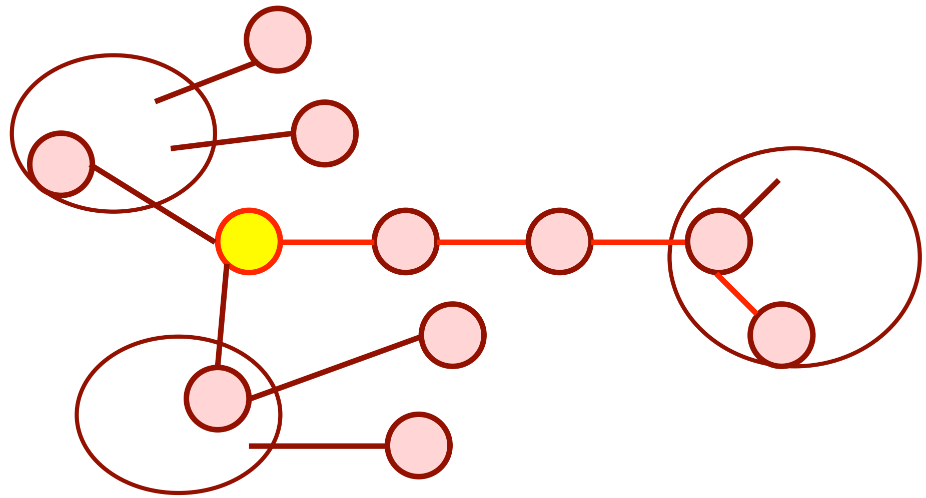


Generating Subproblems

+ not including **e1**

. . .

+ including **e1,…,eh-1** but not **eh**
+ including all **e1,…,eh**



- For last two problems, we spend **O(|V|+|E|)** time for the last two
- We iteratively make the remaining, in total **O(|V|+|E|)** time
  + no parallel edge to **ei**
  + all edges over **ei** are unnecessary

If **h > |E|/10**, **(3)** is satisfied

Generating Subproblems (2)

- for the subproblem of including **e1,…,ei-1** and not including **ei**,

(a) remove all edges over **ei**

(b) contract all bridges **f** not over **ei** s.t. **c(G-ei, f, r) < k+1**

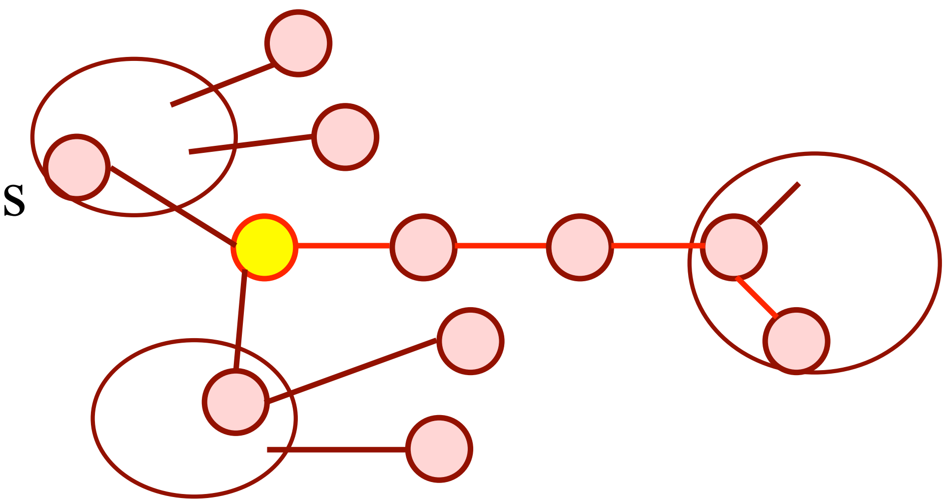
c(G, ei, r) - (|V| - c(G, f, r)) < k-1

(c) remove all edges whose distance to **r** is **k-i**

(d) there is no parallel edge to **ei**, no need to care

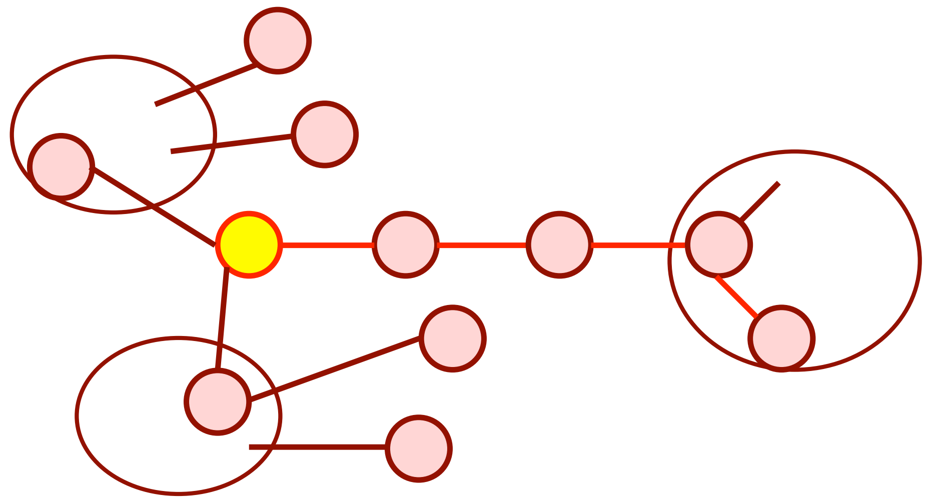
- When we generate subproblems for **e1,…,eh-1**, the edges monotonically increases / decreases

 Total time is **O(|V|+|E|)**



Satisfying the Conditions

- Consider the case that
  the last vertex is of degree 1



[1] not including **e1**

• • •

[h] including **e1,…,eh-1** but not **eh**

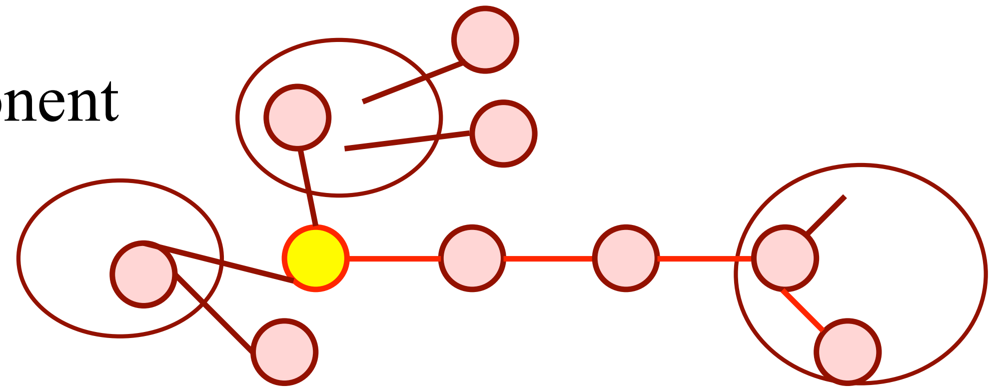
[h+1] including all **e1,…,eh**

in **[h]** and **[h+1]**, + **(a)** and **(d)** never occur

+ **(b)** may occur, but for at most one edge
+ if there are (**|E|/10**) edges of condition **(c)**,
  according to the previous cases, **(3)** will be satisfied
+ if not, at least **9|E|/10 – h-2** edges remain □ **(2)** is satisfied

Satisfying the Conditions (2)

- The case of 2-connected component



[1] not including **e1**

• • •

[h] including **e1,…,eh-1** but not **eh**

[h+1] including all **e1,…,eh**

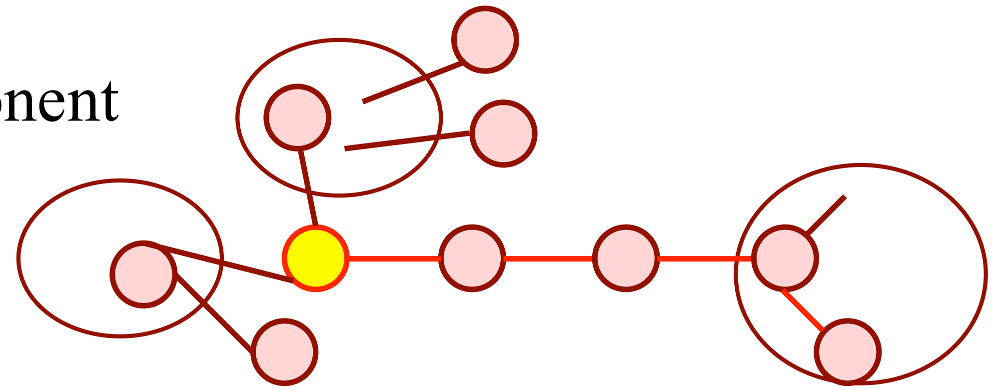
in **[h]**, **(b), (d)** never occur

+ if there are (**|E|/10**) edges of condition **(c)** ,
  according to the previous cases, **(3)** will be satisfied

+ if there are more than **(9|E|/10 – h-2) / 2** edges of condition **(a)**
  choose another edge from the component as **eh**
  □ at most **(9|E|/10 – h-2) / 2** edges satisfy condition **(a)**

Satisfying the Conditions (2)

- The case of 2-connected component



[1] not including **e1**

. . .

[h] including **e1,…,eh-1** but not **eh** (> **(9|E|/10 – h-2)/2** edges)

[h+1] including all **e1,…,eh**

in **[h+1]**, **(a)** and **(b)** never occur

+ if there are many (**|E|/10**) edges of condition **(c)** or **(d)**, according to the previous cases, **(3)** will be satisfied

+ if not, at least **8|E|/10 – h-2** edges remain

Satisfying the Conditions

[1] not including **e1**

. . .

[h] including **e1,…,eh-1** but not **eh** (> **(9|E|/10 – h-2)/2** edges)

[h+1] including all **e1,…,eh** (> **8|E|/10 – h-2** edges)

In the case that **h < |E|/10** holds, the sum of the sizes (#edges) of **[h]** and **[h+1]** is at least

$$(9|E|/10 - h-2)/2 \;+\; 8|E|/10 - h-2$$

$$\geq\; (8|E|/10 - 2)/2 \;+\; 7|E|/10 - 2$$

$$\geq\; 4|E|/10 - 1 \;+\; 7|E|/10 - 2$$

$$=\; 11|E|/10 - 3$$

□ **(2)** is satisfied!!

Thus, an iteration = **O(1)** time on average

Conclusion

- Mechanism of amortization
  - enumeration algorithm spends much time on bottom level
- Basic (toy) case   (elimination ordering)
  - even toy cases are interesting!
- Local amortization (path enumeration)
  - cost for a parent is assigned to children and grandchildren
- Biased (general) case (matching enumeration)
  - just modify the algorithm so that the conditions are satisfied

References

Matching

T. Uno, Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs, ISAAC97, LNCS 1350, 92-101 (1997)

T. Uno, A Fast Algorithm for Enumerating Bipartite Perfect Matchings, ISAAC2001, LNCS 2223, 367-379 (2001)

T. Uno, A Fast Algorithm for Enumerating Non-Bipartite Maximal Matchings, J. National Institute of Informatics 3, 89-97 (2001)

k-subtree

R. Ferreira, R. Grossi, R. Rizzi, Output-Sensitive Listing of Bounded-Size Trees in Undirected Graphs, ESA2011, LNCS 6942, 275-286 (2011)

K. Wasa, Y. Kaneta, T. Uno, H. Arimura, Constant Time Enumeration of Bounded-Size Subtrees in Trees and Its Application, COCOON2012, LNCS 7434, 347-359 (2012)

References

Spanning Trees

H. N. Kapoor and H. Ramesh, Algorithms for Generating All Spanning Trees of Undirected, Directed and Weighted Graphs, LNCS 519, 461-472 (1992)

A. Shioura, A. Tamura and T. Uno, An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs, SIAM J. Comp. 26, 678-692 (1997)

T. Uno, An Algorithm for Enumerating All Directed Spanning Trees in a Directed Graph, ISAAC96, LNCS 1178, 166-173 (1996)

T. Uno, A New Approach for Speeding Up Enumeration Algorithms, ISAAC98, LNCS 1533, 287-296 (1998)

T. Uno, A New Approach for Speeding Up Enumeration Algorithms and Its Application for Matroid Bases, COCOON 99, LNCS 1627, 349-359 (1999)

Exercise 2

Elimination Ordering

2-1. For given a point set in a plane, consider an elimination ordering obtained by iteratively removing the points in its convex hull. Construct an enumeration algorithm for this elimination ordering that runs in **$O(1)$** time for each solution.

2-2. A regular bipartite graph **$G=(V,E)$** of degree **Δ** always has an edge colorings of **Δ** colors. Construct an algorithm for enumerating such edge colorings of **G** in **$O(|V|)$** time for each.

Elimination Ordering

2-3. A graph is chordal if it has no chordless cycle of length greater than 3, equivalently, if it has a clique tree. The vertices of a clique tree are maximal cliques of G, and if clique **Y** is in the path between cliques **Y** and **Z**, **Y∩Z** is included in **X**.

A chordal graph always has a simplicial vertex, whose neighbors compose a clique. A perfect elimination ordering is obtained by iteratively removing simplicial vertices.

Construct an algorithm for enumerating perfect elimination ordering in **O(1)** time for each.

Elimination Ordering

2-4. For given a digraph (acyclic directed graph **G**), topological ordering is an ordering of vertices such that each arc satisfies that its head precedes its tail, in the ordering.

Construct an algorithm for enumerating topological ordering in

O(1) time for each. If it is difficult, explain why it is difficult.

Algorithms

2-5. Construct an algorithm for enumerating vertex subsets **S** in the given graph such that **S** induces a connected graph (induced graph is a subgraph of vertices of **S** and edges connecting two vertices in **S**)

2-6. A path is chordless if no edge not included in the path connects two vertices of the path. Construct an algorithm for enumerating chordless paths in a given graph, such that one of their ends are a given specified vertex **s**, whose amortized time complexity is **O(1)**

Exercises

2-7. Construct an algorithm for enumerating spanning trees of a given graph, in **O(1)** time for each.

2-8. Construct an algorithm for the following problem with time complexity **O(1)** time for each.

For given a point set in a plane, enumerate all convex polygons obtained by connecting the points.

2-9. A zig-zag sequence of a string of numbers is a subsequence **(a1,…,ak)** so that $a1 < a2 > a3 < a4 > a5 \ldots$holds. Construct an algorithm for their enumeration running **O(1)** time for each.