Deep of Enumeration Algorithms

1. Fundamentals

- Motivation – advantage and disadvantage
- Difficulty
- Basic enumeration scheme

1-1 Essence of Enumeration
- when it works -

Definition

Enumeration

output all the solution to the given problem (exactly one for each)

Ex.)

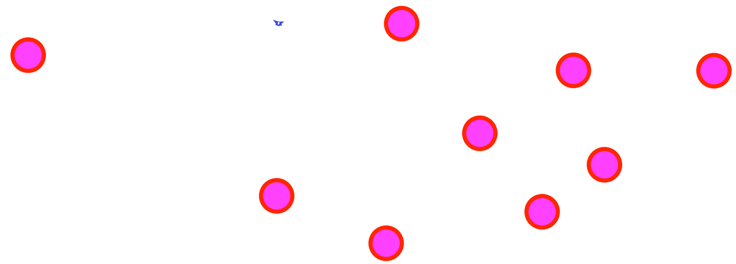
+ Enumerate all paths from vertex **s** to vertex **t** in a graph
+ enumerate all feasible solutions to a knapsack problem

An algorithm for solving an enumeration problem

□ enumeration algorithm

Why Enumerate?

- Optimization finds only one best solution (find an extreme case)
  - □ Enumeration finds all parts of the problem



However,

+ if data is incomplete, or the objective is not clear, the best solution would not be the best (sometimes, bad)
+ in sampling, or search, solutions should be many; we should completely find all solutions
+ if we want to capture not global, but local structures of the data, we should enumerate all remarkable local structures
+ on the other hand, not good if a solution is output many times

Often, enumeration is important

Typical Cases in Practice

- Such motivations often arise in data analysis

  + cluster mining (find (all) densely connected/related structures)
  + similarity analysis (where is similar, and where is not)
  + enumerate explanations / rules (how to explain the data)
  + pattern mining (local features are important)
  + substructure mining (component structures of the data)

- Enumeration is efficient for these tasks

- They have many solutions, we have to have efficient algorithms

When We Can Use

- Enumeration finds all solutions, therefore,

  - Bad for those having huge solutions (≠ large scale data)

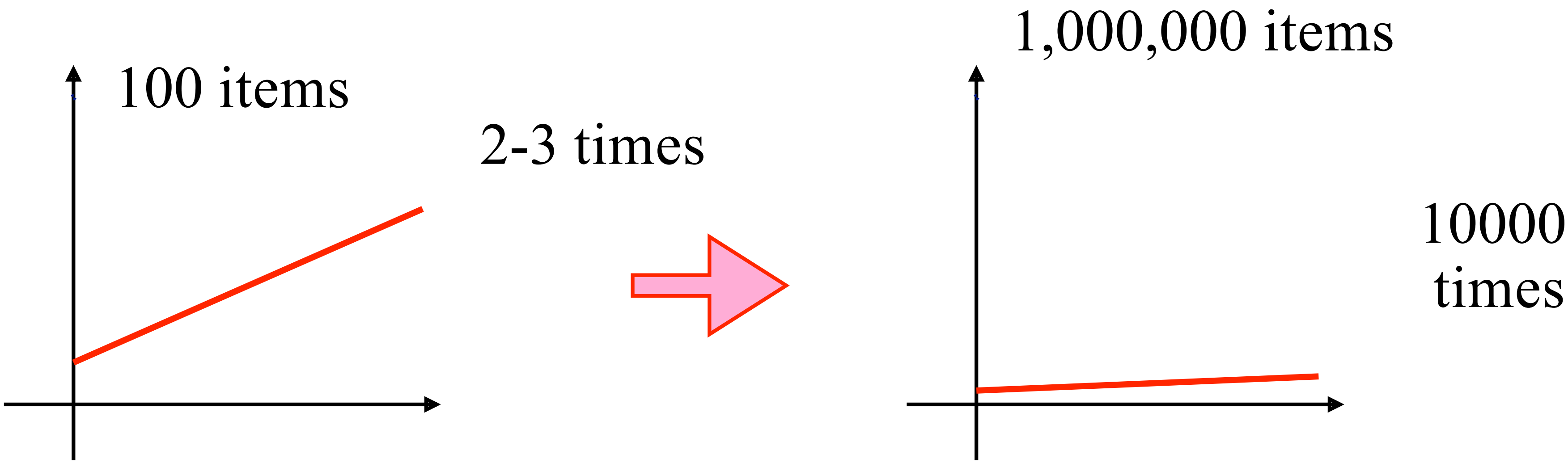
    (implies badly modeled)　having relatively few solutions is fine
  - small problem, few variables □　few solutions
  - good if we can control #solutions by parameters

    (such as, solution size, frequency, weights)
  - unifying similar solutions into one is also good

- Simple structures are easy to enumerate
- Even difficult problems, brute force works for small problems
- Tools with simple implementations would help research/business

Advantage of Algorithm Theory

- Approach from algorithm theory is efficient for large scale data
  - theoretically supported speed up bounds the increase of computation time against the increase of the problem size
  - The results of the computation are the same



Acceleration increase as the increase of problem size

Theoretical Aspect of Enumeration

- Problems of fundamental structures are almost solved
  - path, subtree, subgraph, connected component, clique,…
- Classes of poly-time isomorphism check are often solved
  - sequence, necklace, tree, maximal planer graph,…
- But, problems having tastes of applications are usually not solved
  - patterns included in positive data but not in negative data
  - reduction of complexity / practical computational costs
- Problems of slight complicated structures are also not solved
  - graph classes such as chordal graphs / interval graphs
  - ambiguous constraints
  - heterogeneous constraints (class + weights + constraints)

Advantages for Application

- Enumeration introduces completeness of solutions
- Ranking/comparison can evaluate the quality of methods exactly
  - We can evaluate the solution of small problems by a method

- Various structures / solutions are obtained in short time

- Enumeration + candidate squeeze is a robust method against the changes of models / constraints
  - Good tools can be re-used in many applications
  - Trial of an analysis, and extension of research become easy

- Fast computation can handle huge data

The Coming Research on Theory

- Basic schemes for enumeration are of high-quality
- Producing a totally novel algorithm would be very difficult
  - In some sense, approach is fixed (always have to "search")

- The next level of the research for search route, duplication, canonical form,… are important topics
  - maximal/minimal, geometric patterns, hypothesis, …

- Applications to exact algorithm, random sampling, counting are also interesting
  - Enumeration may become the core of research
  - Apply techniques of enumeration to usual algorithms

1-2 Difficulty of Enumeration
- how to brute force -

Can We Enumerate?

- Once we meet a problem, we have to outlook for the solvability of the problem

- … and, how much will be the cost (time, and workload)

- For the purpose, we have to know

    ”what are the difficulties of the enumeration”,
    “what kind of techniques and methods we can use”
    “how to enumerate in simple and straightforward ways”

Difficulty of Enumeration

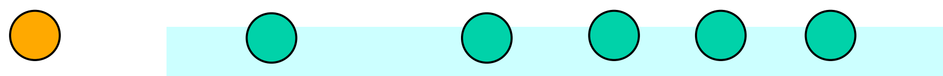
- Designing enumeration algorithms involves some difficulty
  - + how to avoid duplication
  - + how to find all
  - + how to identify isomorphism
  - + how to compute quickly

…

Difficulty on Duplication

- Assume that we can find all
- Even though, it is not easy to avoid duplication, (not to perform the search on the same solution)

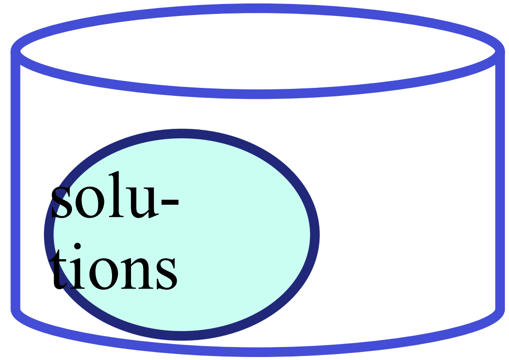
- Simply, we can store all solutions in memory, to check solutions found are already output or not
  - memory inefficient…
  - dynamic memory allocation & efficient search (hash)



- Output/ not output: deciding without past solutions is better

Its generalization yields “reverse search”

For Real-World Problems

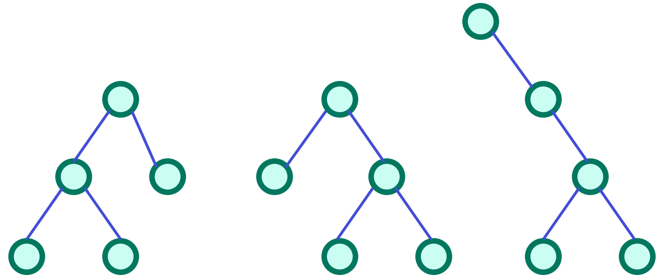


- …Memory inefficient for huge solutions
- If this is not a problem, i.e.,
  - Non-huge solutions, or we have sufficiently much memory
  - We have hash/binary trees that are easy to use
- Brute-force is simple Good in term of engineering
- No problem arises in the sense of complexity
  - theoretically, OK

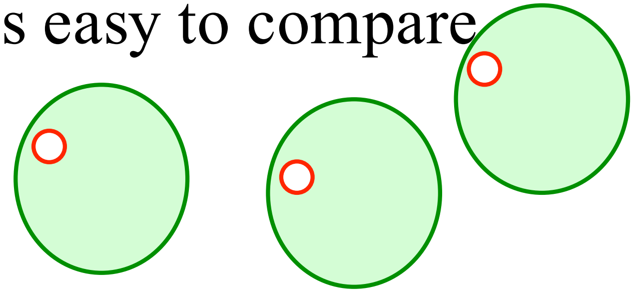
In the application (we want to obtain solutions), brute force is a best way if it doesn't lose efficiency

Difficulty on Isomorphism

- Non-linear structures, such as graphs, are hard to identify the isomorphism



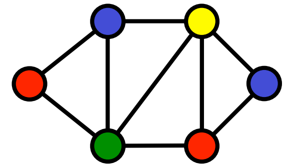
- An idea is to define "canonical form", that is easy to compare
  - □ has to be one-to-one mapping
  - □ no drastic increase of size



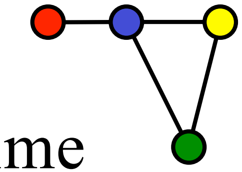
- bit-encoding ordered tree □ un-ordered tree □ transforming series-parallel graphs and co-graphs to trees

Enumeration of canonical form corresponding to enumeration of the original structures

When Isomorphism is Hard



- How to define canonical form if isomorphism is hard?
  - graph, sequence data, matrix, geometric data…



- Even though, isomorphism can be checked in exponential time (so, we can define canonical form, which takes exp. time to comp.)

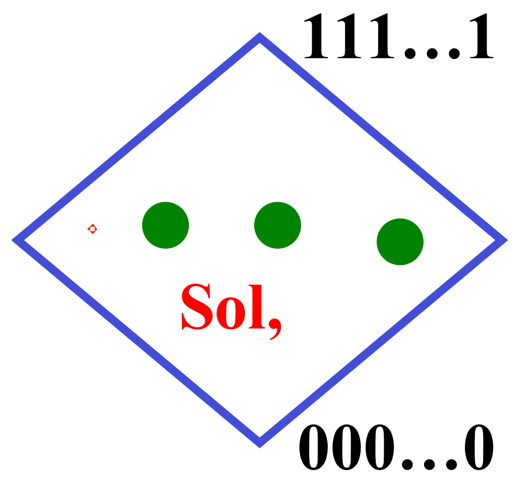
- If solutions are few, (ex. graph mining), brute-force works
  - usually not exponential time
  - embedding is, basically, the bottle-neck computation

- From complexity theory, algorithms taking exponential time only few iterations are really interesting, (but still open).

Difficulty on Search

- Cliques and paths are easy to enumerate
  - Cliques can be obtained by iteratively adding vertices
  - Path sets can be partitioned clearly

- However, not all structures are so
  - maximal XXX, minimal OOO
  - XXX with constraints (a), (b), (c), and…
  - Solutions are not neighboring each other



- Roughly, there are two difficult cases
  - Easy to find a solution, but … (maximal clique)
  - Even finding a solution is hard (SAT, Hamilton cycle…)

“Finding One Solution” is Hard

- For hard problems such as NP-complete problems, even finding one solution is hard

  (maximum clique, SAT, Hamilton cycle, …)

- Even though we want to find all, thus hopeless
  - Each iteration would involve NP-complete problems
  - We should give up usual complexity results

- However, similar to the isomorphism, if one of

  \+ **“usually easy”** such as SAT

  \+ **”non-huge solutions”** maximal/minimal

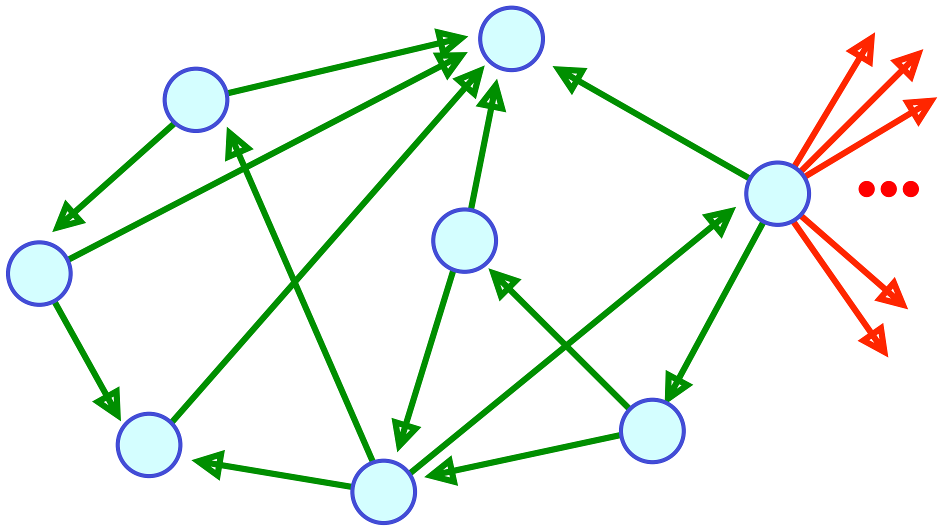
  \+ **”bounded solution space”** size is bounded,

  is satisfied, we can solve the problem in a straightforward way

"Finding One Solution" is Easy

- Even if any solution can be found by a (poly-time) algorithm,
  we have to consider how to make other solutions from a solution
  + if succeeded, we can visit all solutions by moving iteratively
  + if the move spans all the solutions, enumeratable

- But, if "making other solutions" takes exponential time, or exponentially many solutions are made, time inefficient



Move Efficiently

- For maximal solutions,

+ remove some elements and add others to be maximal

  □ can move iteratively to any solution

  + but, #neighboring solutions is exponential,

    enumeration would take exponential time for each

- Restrict the move to reduce the neighbors

  + add a key element and remove unnecessary elements

  □ exponential choices for removal results exponential time

- For maximal solutions, pruning works well

  □ no other solution above a maximal solution

    & easy to check the existence of maximal solution

Fast Computation

- Standard techniques for speeding up are also applicable
- Fasten the computation of each iteration
  + data structures such as binary trees and hashes
  + use sparseness by using array lists of adjacency matrix
  + avoid redundant computation
  + fitting to cache, polish up codes

- In enumeration, we usually change the current solution, dynamic data structures are often efficient
  - input graph, maintain vertex degree, weight sum, frequency…

- Using "bottom-wideness" of recursion is especially efficient

Brute-force Algorithm

- Brute force is acceptable if the problem is easy (small)
  - how do we do “brute force”?

    + enumerate all candidates and output the solutions among them
    + enlarge the solutions one by one and remove isomorphic ones
    + scan the candidates from the smaller ones

- We should have “simple algorithms” for solving easy problems

- … and also “simple implementing ways”, such as, only making a subroutine (oracle) for computing a function is necessary

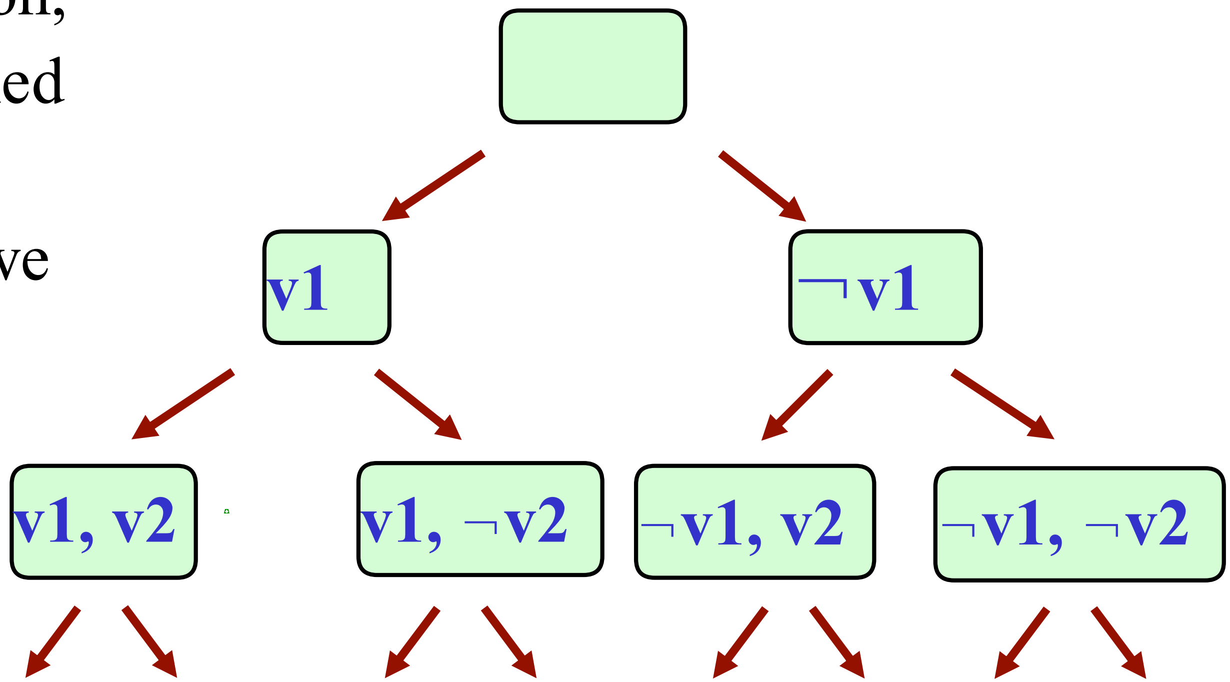
Divide and Conquer

- Determine the value of variables one by one
- Recursive call for each value

- At the bottom of recursion, an assignment is determined

- Prune the brunch when we confirm non-existence of solution in descendants

□ accuracy and speed are the key.



Exact ∩ polytime, then polytime delay

(1) Enumerate Combinations

- Enum. all combinations □ determine variable values recursively

Enum1 (**X** :set of all determined values, **i**: index)
1. **if** no solution includes **X** **then return**
2. **if** **i** > maximum index **then**
3.   **if** **X** is a solution **then output X**
4. **else**
5.   **for** each **e** **in** values which **xi** can take
6.   **call Enum1** (**X∪(xi =e)**, **i+1**)
7. **end for**

- Only "3. check of being a solution" is needed. 1. is not necessary
- Fast if check in 1 is of high accuracy

(2): Enumerate Patterns

- To avoid isomorphic solutions, incremental generation (for graphs, matrix, sequences…)

Global variable: database **D := ϕ**

Enum2 (**X**: pattern)

1. insert **X** to **D**
2. **if** no solution includes **X** **then return**
3. **if** **X** is a solution **then output** **X**
4. **for** each **X’** obtained by adding an element to **X**
5. **if** none of **D** is isomorphic to **X’** **then call** **Enum2** (**X’**)

- Only designs of 3. and 4. are necessary
- Efficient if check in 2. is fast and of high accuracy

1-3 Basic Algorithms

Evaluate the Complexity

- Enumeration has exponentially many solutions, thus is natural to take exponential time
- Indeed, desired to terminate shortly if few solutions

- Actually, usual complexity does not work for enumeration

+ An enumeration algorithm may output **2n** solutions
+ An algorithm testing all **2n** combinations is optimal!

- Essentially, this is impossible to improve the algorithm in the sense of usual time complexity, in this case

Complexity on Enumeration

- Enum. algorithm is desired to terminate shortly if few solutions

- For given an instance, #solutions **N** is determined (so, is an invariant), low degree order in **N** is good

An algorithm is **output polynomial time** if it terminates in time polynomial to input size **n** and the number of solutions **N**

An algorithm is **polynomial (time) delay** if the maximum time interval between two solutions is polynomial to input size **n**

Basic Enumeration Algorithms

- Since fundamental, construction scheme is also simple
- On the other hand, not so many variations

\+ **Backtracking**

depth-first search with lexicographic ordering

\+ **binary partition**

branch & bound like recursive partition algorithm

\+ **reverse search**

search on traversal tree defined by parent-child relation

Backtracking

- Mainly used for independent (monotone) sets (maximals)

Independent set system F : **X∈F** □ for any **X'⊆X** , **X'∈F**

(**X∈F** □ any subset of **X** is a member of **F**)

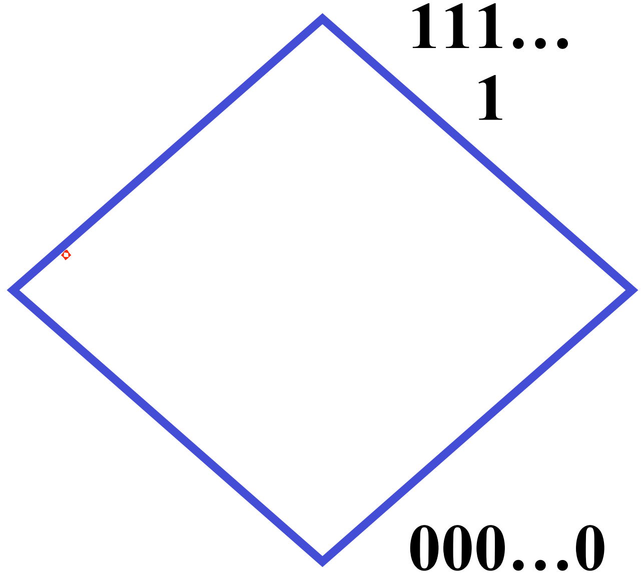
Ex)

\+ cliques of a graph, matchings, combinations of numbers whose sum is less than **b**, frequent itemsets…

× **Not**

\+ trees of a graph, paths, cycles, …

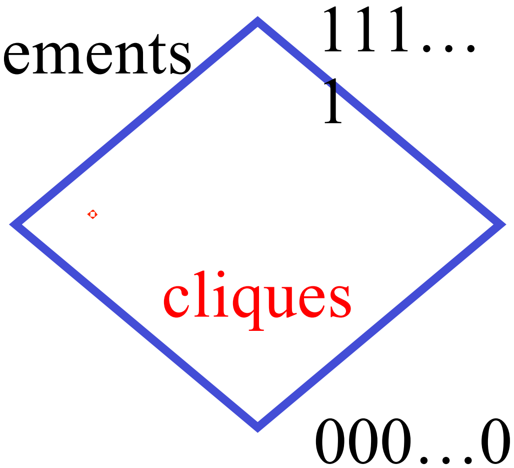
111…1



000…0

Framework of Backtracking

- Start from the empty set, and recursively add elements

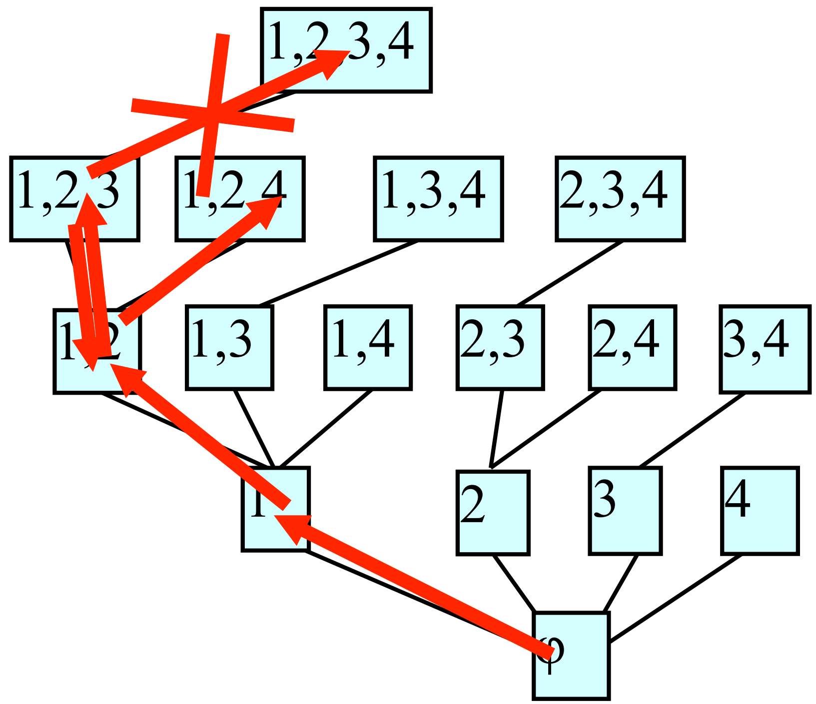


- In each iteration, add only elements larger than the current maximum element

(an iteration does not include those in its recursive calls)

- Recursive call with the result of addition, if it is a solution

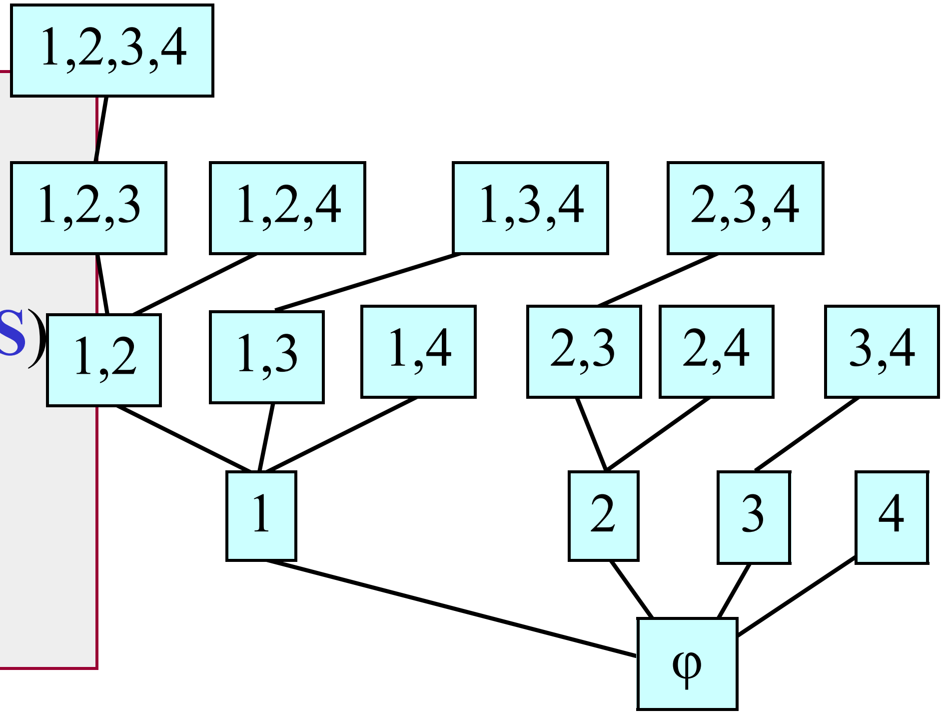
- Go back after all examinations



Pseudo Code for Backtracking

- Start from the empty set, and recursively add elements;
add only elements larger than the current maximum element

```
Backtrack (S)
1.  output S
2.  for each e > tail of S
              (the max. element in S)
3.     if  SU{e} is a solution then
            call Backtrack (SU{e})
4.  end for
```



- simple, and polynomial space
- polynomial delay (output polynomial time)

Feasible Solutions to Knapsack Problem

Problem: enumerate all subsets of **a1,…,an** whose sum is less than **b**

Backtrack (S)
1. **output** S
2. **for** each **i** > tail of S (maximum element in S)
3. **if** ∑S + **ai** < **b** **then call Backtrack** (S∪{**ai**})
4. **end for**

Computation time:

each iteration output a solution, and take **O(n)** time

□ time per solution is **O(n)**

- Sort **a1,…,an**, then each recursive call can be generate in **O(1)** time

Code for Knapsack

- Print all combinations of **a[0],…,a[n]** with summation less than **b**

```
int a[n], flag[n];

sub (int i, int s){
  int j;
  for (j=0 ; j<n ; j++)
      if (flag[j] = = 1) printf (“%d\n”, a[j]);   // print a solution
  for (j=i+1 ; j<n ; j++)
     if (s+a[j] <= b){     // check the feasibility
        flag[j] = 1;
        sub (i, s+a[j]);
        flag[j] = 0;
     }
  }
}
```

Maximal Solutions

- #solutions increases exponentially when **n** or the sizes of solutions are large
- If #solutions is large, post-process is also hard

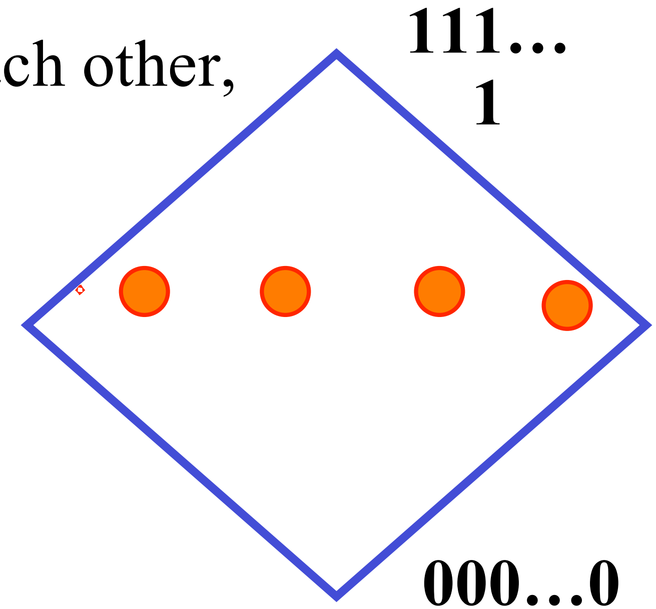
□ enumerate maximal so that the solution set is irredundant

X∈F is maximal in **F** □ for any **X⊆X'**, **X'∈F** does not hold

- Maximal solutions are not neighboring to each other, search is difficult

(exception; spanning trees, matroid bases)

- If there is a good pruning method, it's OK



Enumerating Maximals

Problem: enumerate all maximal subsets of **a1,…,an** whose sum is no greater than **b**

- Put indices to **a1,…,an** in decreasing order

Backtrack (S)
1. **output S**
2. **for** each **i >** tail of **S**
   and ∑S + ai +…+ an > b – ai-1
3. **if ∑S + ai ≦ b then call Backtrack (S∪{ai})**
4. **end for**

Pruning that with only non-maximal solutions

Computation time:

An iteration takes **O(n)** □ **O(n)** time per solution

Maximal: Shift a Solution to the End

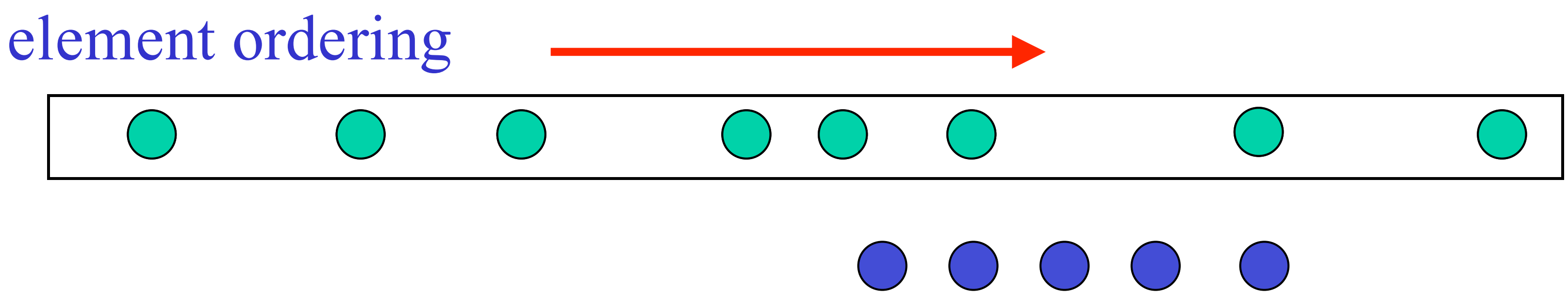
- Maximal enumeration admits a simple pruning algorithm

(1) prune if meets a non-member

(2) no brunch needed if addition of all remaining members is a member

- Even if **(1)** is complete, exhaust search for all members is inefficient

- Find a maximal solution, shift all its element to the bottom, then no need of recursive calls for the shifted elements
  - because **(2)** works for the elements!

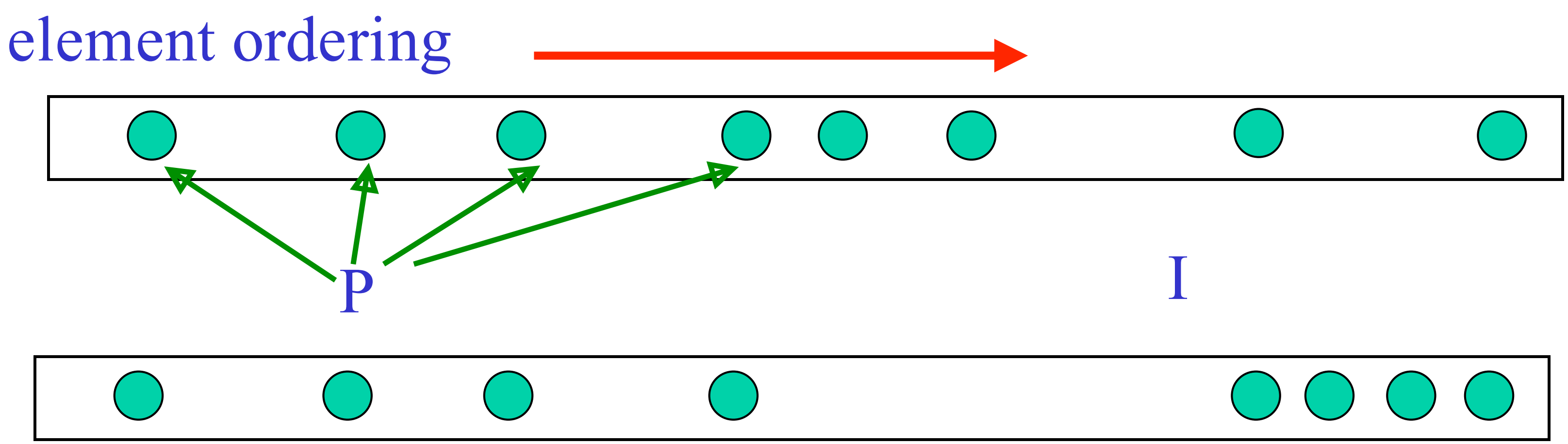


For small maximal solution sizes (up to 30), practically efficient

Pseudo Code

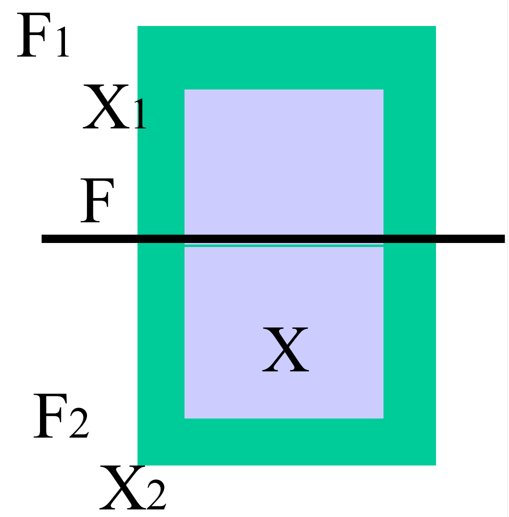
- Describe the algorithm by a pseudo code

EnumMax (**P**:current solution, **I**: undetermined elements)
1. find maximal set **S** among those including **P** and included in **P∪I**
2. **if S** is a maximal solution of the problem **then output S**
3. **for each** **e∈I＼S**
   I := I＼{e}; **call EnumMax (P∪e, I)**



Binary Partition

- **X** is a set of solutions, that is a subset (subsequence, etc.) of **F** satisfying a property **P**
- Binary partition outputs the solution if solution in **X** is unique
- Otherwise, it partitions **F** into two (or several) sets so that **X** is partitioned into non-empty sets
- Do this recursively, until the solution is unique



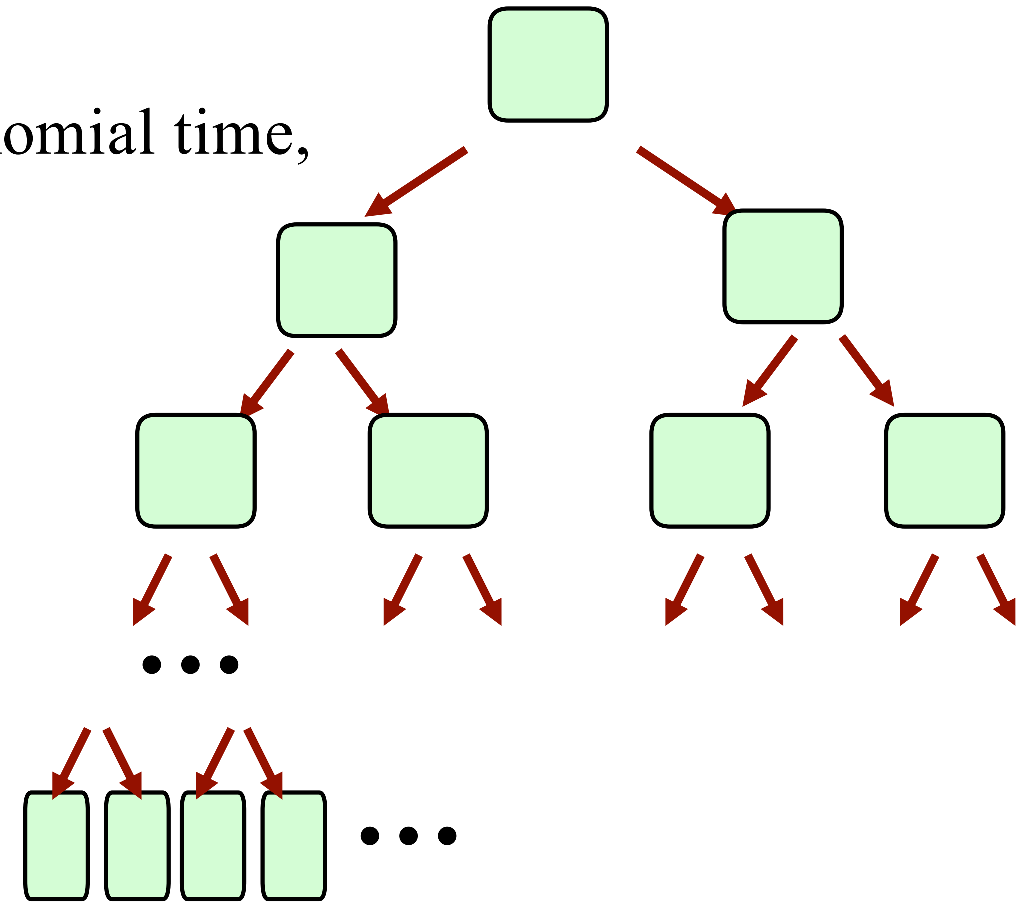
Ex.)

+ paths of a graph connecting vertex **s** and vertex **t** (st-paths)
+ perfect matchings of a bipartite graph
+ spanning trees of a graph
+ connected components of a graph

Time Complexity

- Binary partition always partitions a problem or outputs a solution
  - #iteration is bounded by **2N**

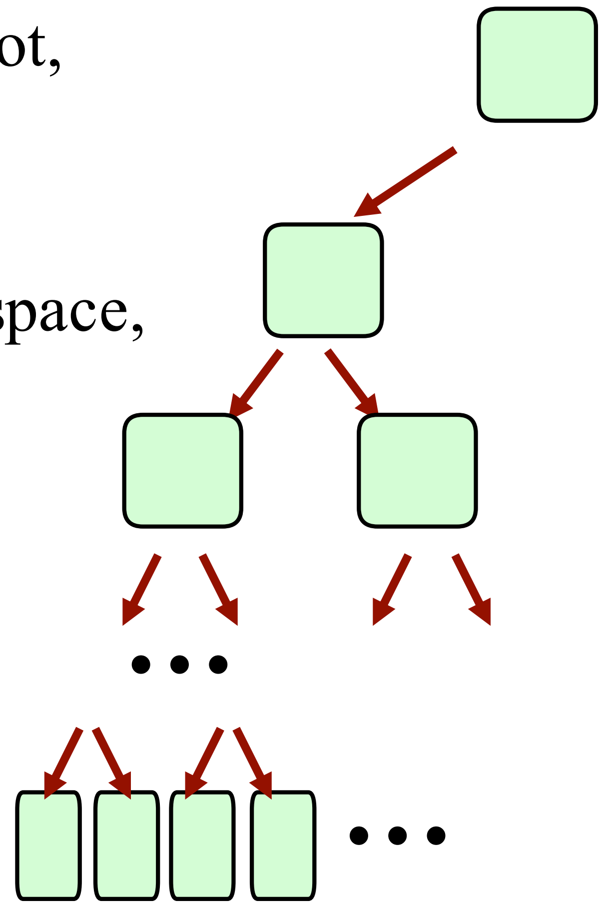
- The partition process is polynomial time,
  (determine how to partition,
  and check empty or not)
- the algorithm is
  output polynomial time



Time Complexity

- If the height of the tree is polynomial in **n**, it is polynomial delay (to go up (go back) from the leaf to the root, **O(height)** time is needed)

- If the partition process needs polynomial space, the algorithm is polynomial space



Binary Partition of st-paths

Problem: enumerate all **st**-paths in **G=(V,E)**

Partition: choose an edge **e** incident to **s**, and partition into

+ enumeration of **st**-paths including **e**
+ enumeration of **st**-paths not including **e**

so that both problems are non-empty

Child Problems:

st-paths including **e**: remove all edges incident to **s** except **e**

st-paths not including **e**: remove **e**

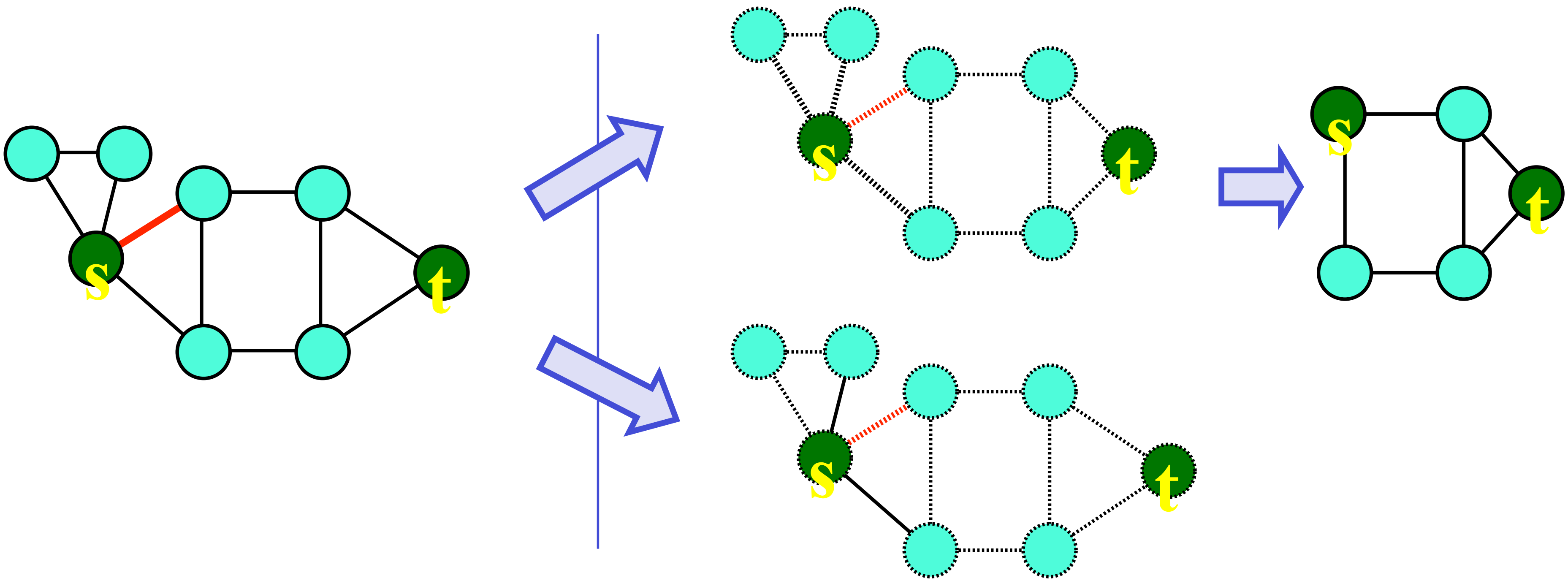
Child Problems on st-paths

Child Problems:

st-paths including **e**: remove all edges incident to **s**

(and move **s** to the next vertex) □ denote **G-s**

st-paths not including **e**: remove **e** □ denote **G-e**



Computation time:

one iteration **= O(|E|)**

Choosing Valid Edge

- If we choose a bad edge, the subproblems will be empty;

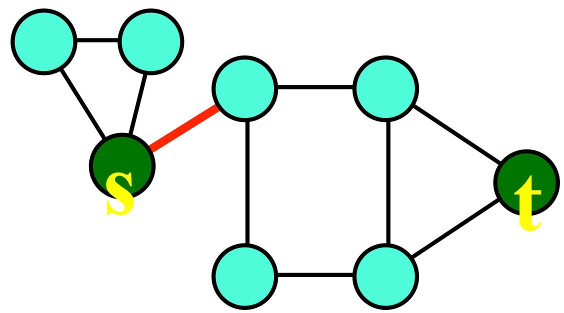
\+ **“including e”** is empty, if **t** is not reachable via **e**

 remove the component including **e**

\+ **“not including e”** is empty, if **e** is the only edge reachable to **t**

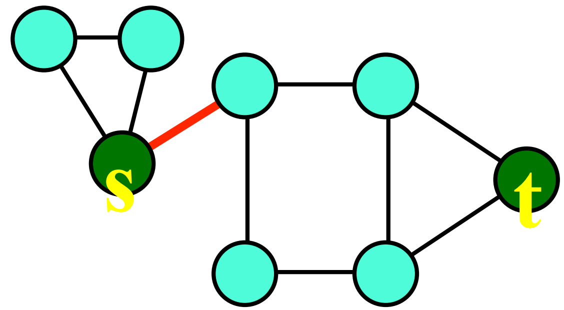
 move **s** to the next vertex , and remove **e**

- After at most **|E|** repetitions, we can always find a valid edge



Time Complexity

- Test of the validity of the edge takes **O(|V|)** time
  at most **O(|E|)** repetitions



- An iteration takes **O(|E||E|)** time

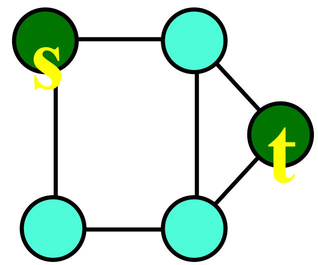
- Since #iterations **< 2N**, time per solution is **O(|E||E|)**

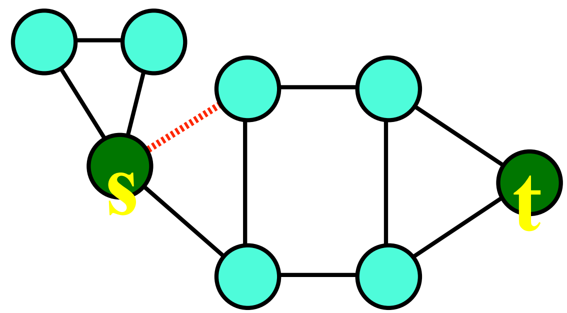
- Since the height of the recursion tree is **O(|V|)**,
  the delay **O(|V||E|2)**

Pseudo Code for st-paths

Enum_st-path (**G, s, t, S**)

1. **if s = t then output S, return**
2. choose an edge **e=(s,v)**
3. **if** no **vt**-path in **G-s then**
   remove **e**, **go to 1.**
4. **if** no **st**-path in **G-e then**
   remove **e**, **S := S+s**, **s := v**, **go to 1.**
5. **call Enum_st-path** (**G-s, v, t, S**)
6. **call Enum_st-path** (**G-e, s, t, S**)





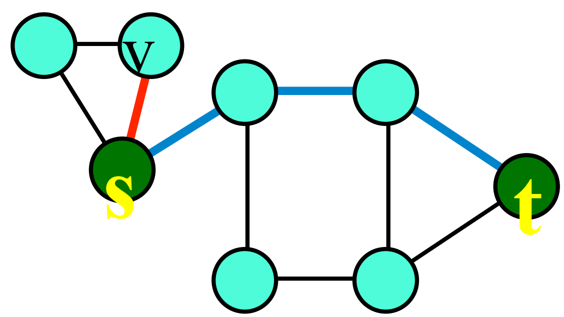
Better Algorithm

- How long does it take (graph reform) to find a valid edge?

- Find a path **P** from **s** to **t**

- Choose an edge **e = (s,v)** incident to **s** and not in **P**
  + **t** is not reachable via **e** □ delete the visited edges
    □ **O(**#delete edges**)**

  + only one edge (in **P**) is incident to **s** □ move **s** to **v**, and remove **e**
    □ **O(1)**



- Computation time is **O(**#delete edges**)**, until we find a valid edge, i.e., **O(|E|)**

Pseudo Program Code

- **flag[] :=0** in initialization, **path** is the current solution

```
int mark[m], path[n];

enum_path (int s, int i){
 if (s = t){ output path[0],…,path[i]; return }
 • find an st-path, f   (=(s,v)) := the edges in the path incident to s
 • mark[f]:= 1 (put mark)
 while (1){
    • choose an edge e=(s, v) s.t. mark[e] = 0
    • mark[e] := 1
    • if (no such edge e exist){
        path[i] := v; i++; s := v
        if (s = t){ output path[0],…,path[i]; return }
     } else if ( t is reachable from v via only unmarked edges and not through s ){
break }
  }
  call enum_path (s, i);
   path[i] := v; call enum_path (v, i+1);
   • set mark[e]:= 0 for edges e marked in this iteration
}
```

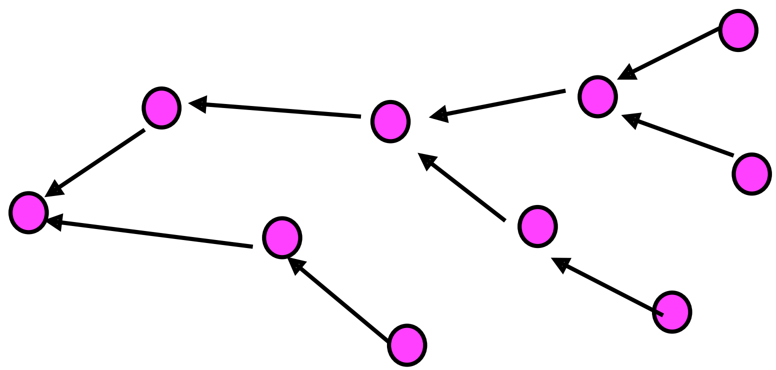
1-4. Reverse Search and Maximal Clique Enumeration

Reverse Search

- For every solution except for several, define its parent, so that
  ★ any solution is not its proper ancestor (acyclic)

- The parent-child relation induces a tree (or a forest)

- Traverse the tree by depth-first search



- #iteration is equal to #solutions
- Computation time per solution is that per iteration

Realization

- Depth-first search on induced tree (called, family tree)

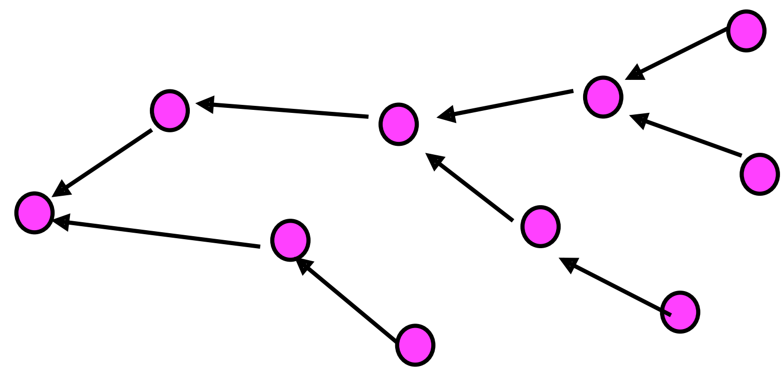
  □ no need to store the tree in the memory (or disk)

- Algorithm for finding all children of a parent is sufficient

- Particularly, it is better to have an algorithm that finds the **(i+1)**-th child by giving **i**-th child

```
Reverse_Search (S)
1.  output S
2. for each child S' of S
3.       call Reverse_Search (S')

4. end for
```

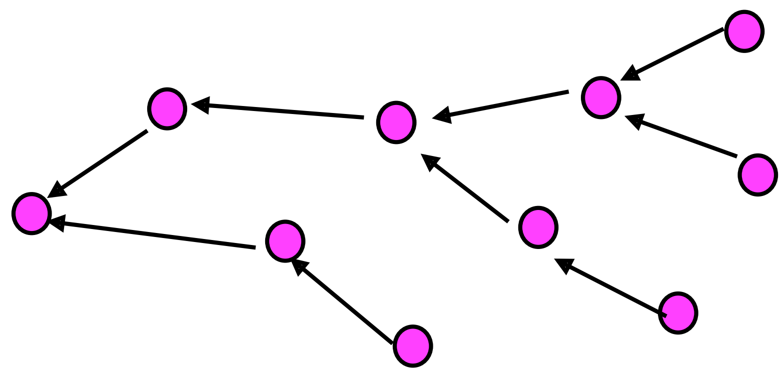


Complexity

- each iteration = each solution (#iterations = #solutions)

- If finding a (next) child takes **O(X)** time, the computation time per iteration is **O(X)**
(finding one child = children enumeration time / #children)
   □ the computation time per solution is **O(X)**

- Output polynomial if **X** is polynomial
- Space = memory usage of iterations and height of the family tree
- Using (find **(i+1)**-th child),
  height is eliminated



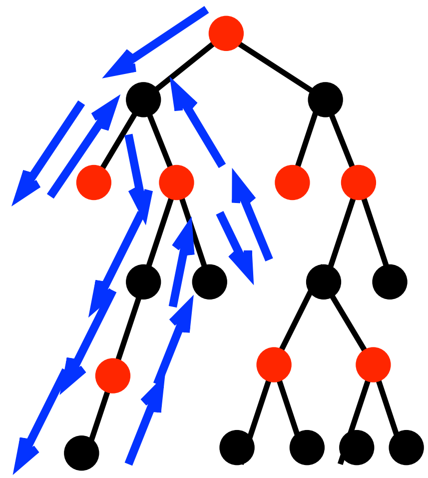
- **O(X)** delay by alternative output

Alternative Output

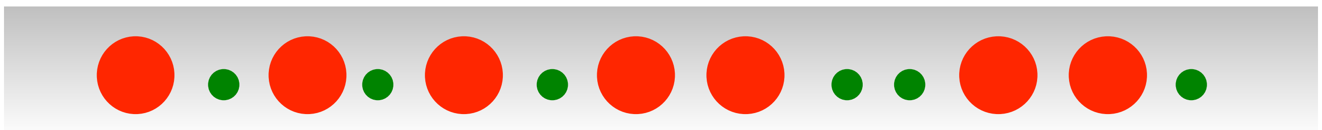
- Alternative output is a technique for reducing the delay (avoid long path (going up) with no output)

- Suppose that an enumeration algorithm takes **O(X)** time in each iteration, and always outputs a solution

AlternativeOutput (S)
1. **if** depth is even **output** S
2. **for** each child S' of S
   call **AlternativeOutput** (S')
3. **if** depth is odd **output** S

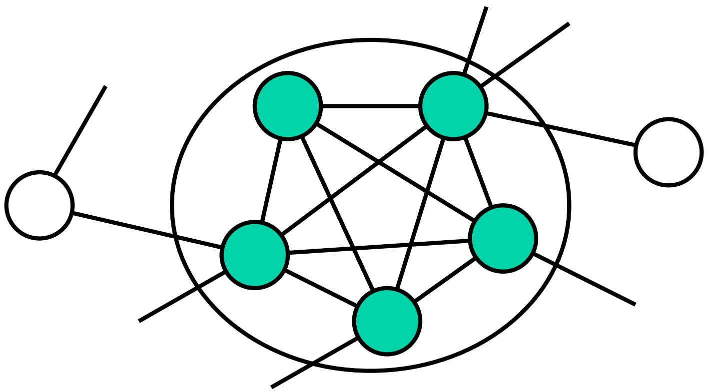


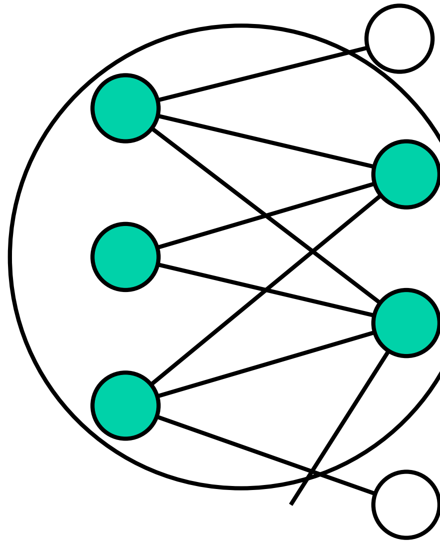
- Delay is **O(X)**



Clique Enumeration

Clique: a subgraph that is a complete graph (any two vertices are connected





- Finding a maximum size is NP-complete
- Bipartite clique enumeration is converted to clique enumeration
- Finding a maximal clique is easy (**O(|E|)** time)
- Many researches and many applications, with many models

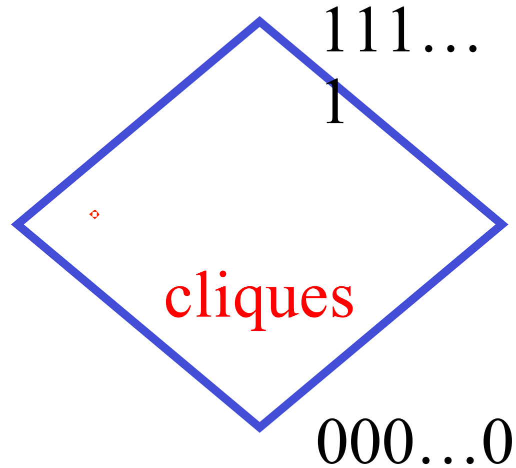
Monotone

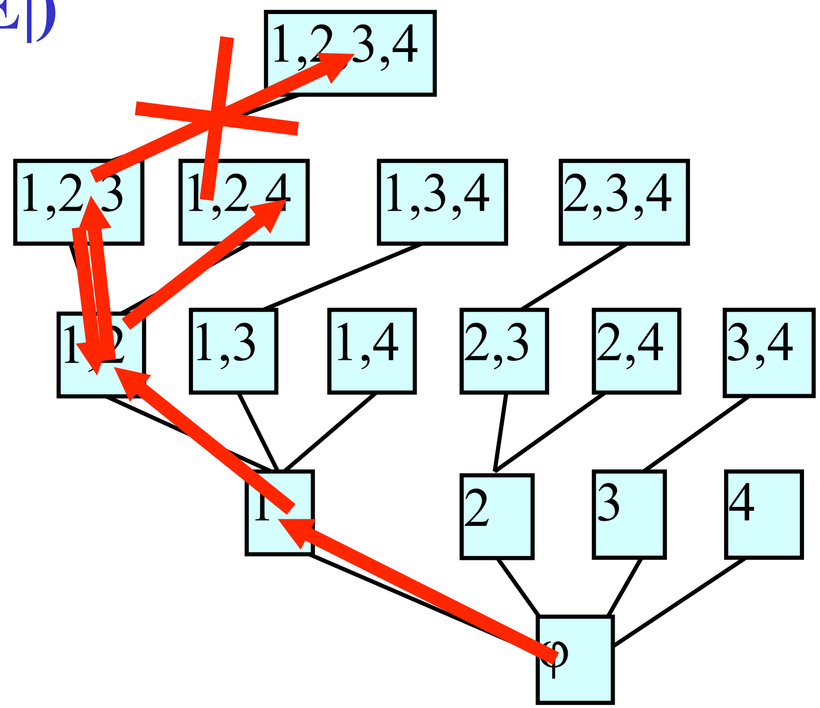
- Set of cliques is monotone, since any subset of a clique is also a clique

□ Backtracking works

- The check being a clique takes **O(|E|)** time, and at most |V| recursive calls

□ **O(|V| |E|)** per clique





Motivations

- Real-world graphs are usually sparse, thus clique sizes are small
- On the other hand, large cliques also exist

□ #cliques explodes

- Enumeration of maximal ones looks better

Clique

+ the number reduces to 1/10～1/1000
+ no information loss (any clique is included in some maximal)
+ maximal cliques are complete in some sense, and non-maximals are incomplete, thus good for modeling

Difficulty on the Search

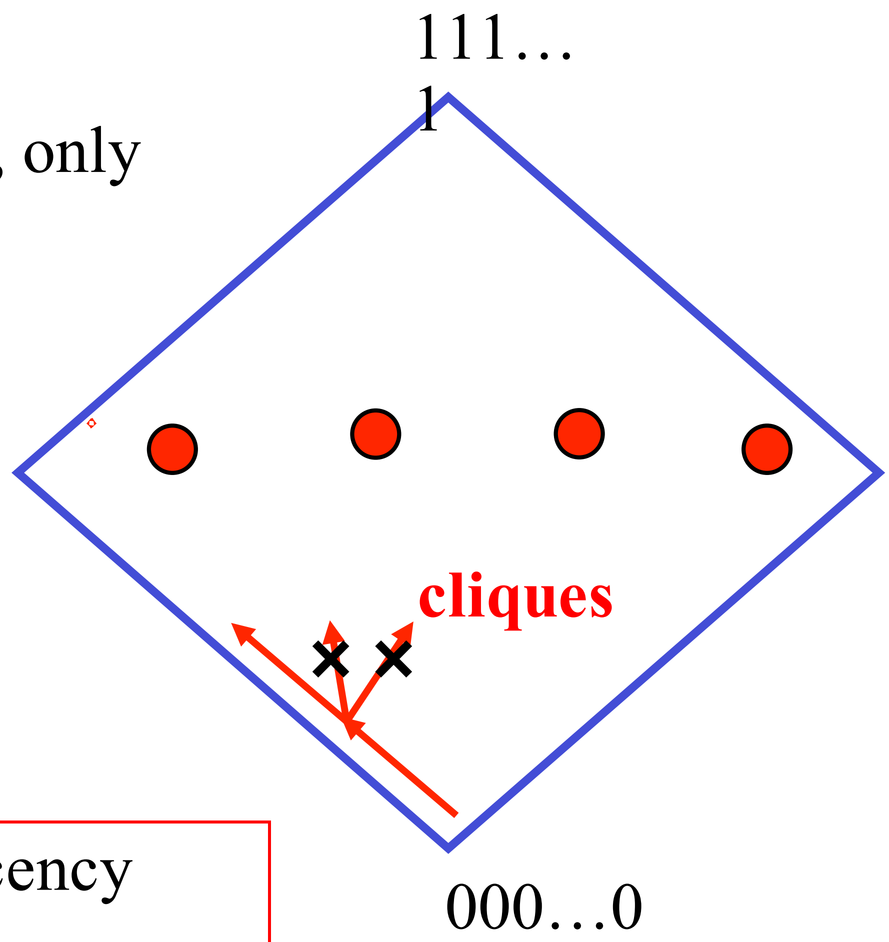
- Maximal cliques are tops of the mountains

□ Impossible to move to each other, only with simple operation

- No maximal near by start

- … Backtrack doesn't work…

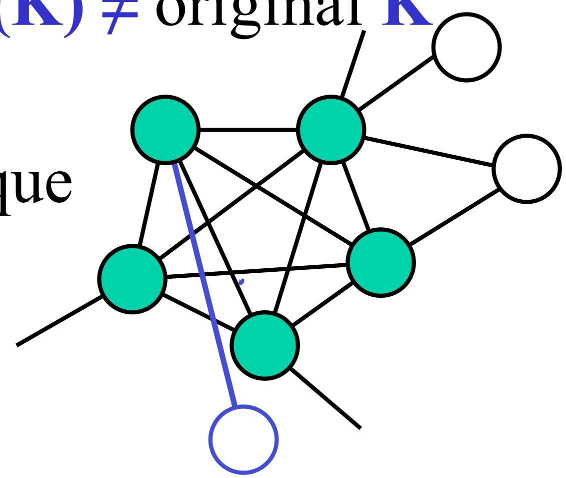
Introduce more sophisticated adjacency on maximal cliques



Adjacency on Maximal Cliques

- **C(K) :=** lexicographically smallest maximal clique including **K** (greedily add vertices from the smallest index)

- For maximal clique **K**, remove vertices iteratively, from largest index
- At the beginning **C(K) = K**, but at some point **C(K)** ≠ original **K**



- Define the parent **P(K)** of **K** by the maximal clique (uniquely defined) .
- The lexicographically smallest maximal clique (= root) has no parent

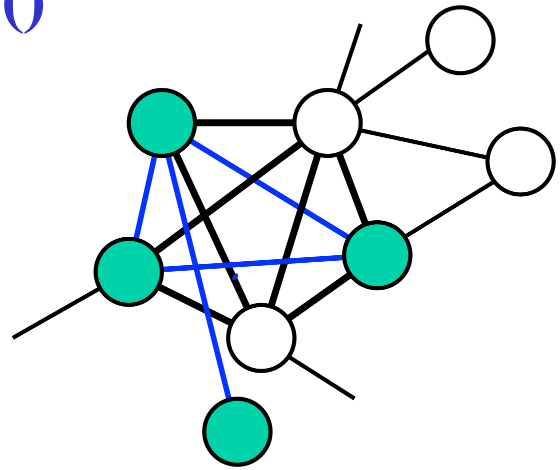
- **P(K)** is always lexicographically smaller than **K**

□ the parent-child relation is acyclic, thereby induces tree

Finding Children

- **K[v]** : The maximal clique obtained by adding vertex **v** to **K**, remove vertices not adjacent to **v**, and take **C()**

  □ $\mathbf{K[v] := C(K \cap N(v) \cup \{v\})}$



- **K'** is a child of **K** □ **K' = K[v]** for some **v**

  K[v] for all **v** are sufficient to check

- For each **K[v]**, we compute **P(K[v])**

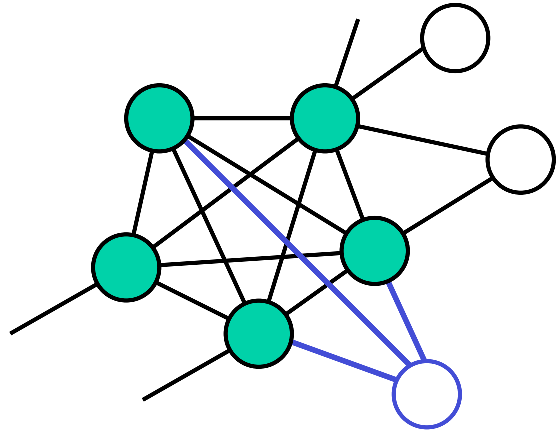
  If it is equal to **K** to, **K[v]** is a child of **K**

All children of **K** can be found by at most **|V|** checks, thus an iteration takes **O(|V| |E|)** time □ **O(|V| |E|)** per maximal clique

- Note that **C(K)** and **P(K)** can be computed in **O(|E|)** time

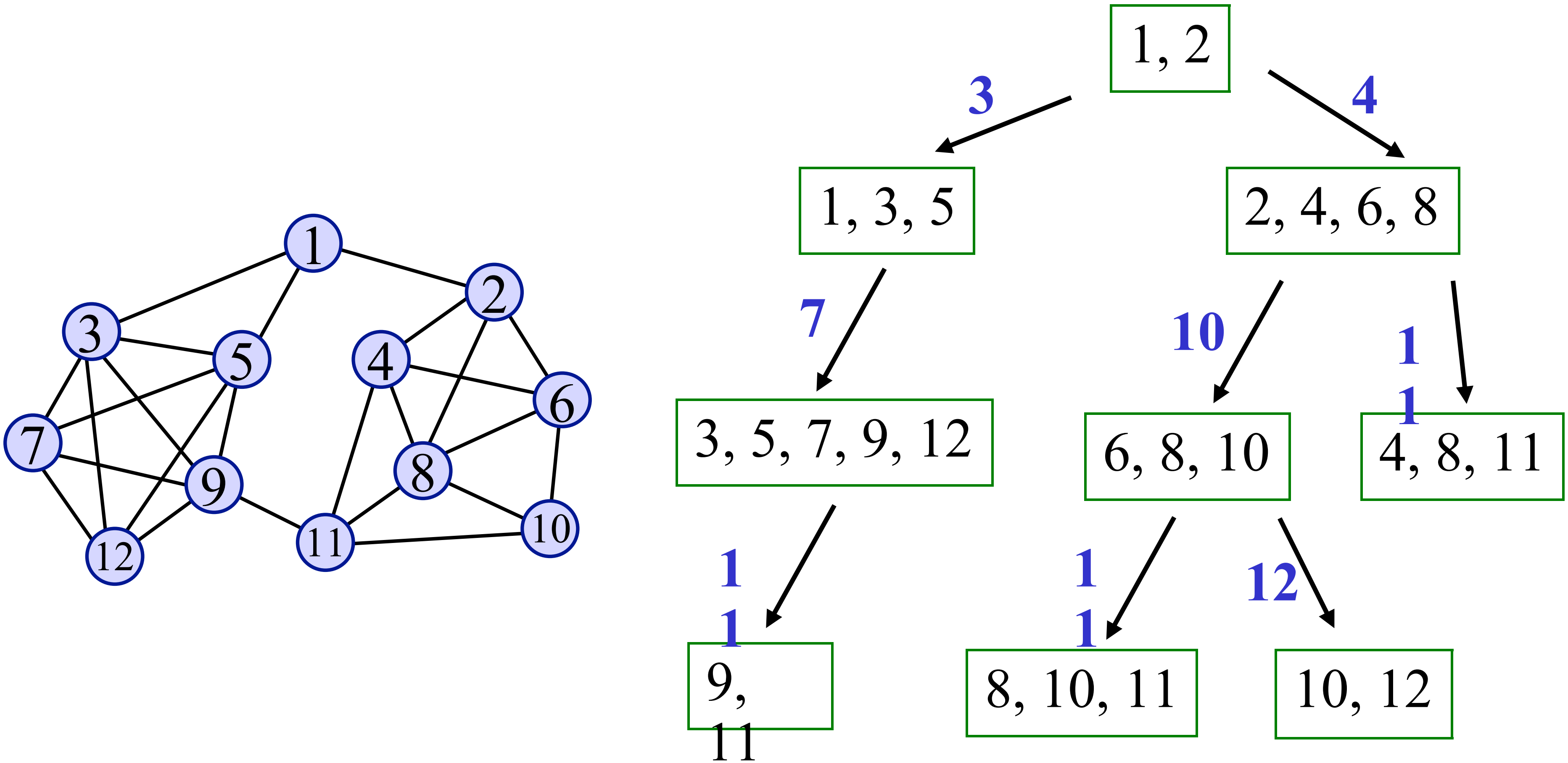
Pseudo Code for Maximal Clique

EnumMaxcliq (**K**)
1. **output K**
2. **for** each vertex **v** not in **K**
3. **K' := K[v]** (**= C(K∩N(v)∪v)**)
4. **if P(K') = K then call EnumMaxcliq (K)**
5. **end for**



Example

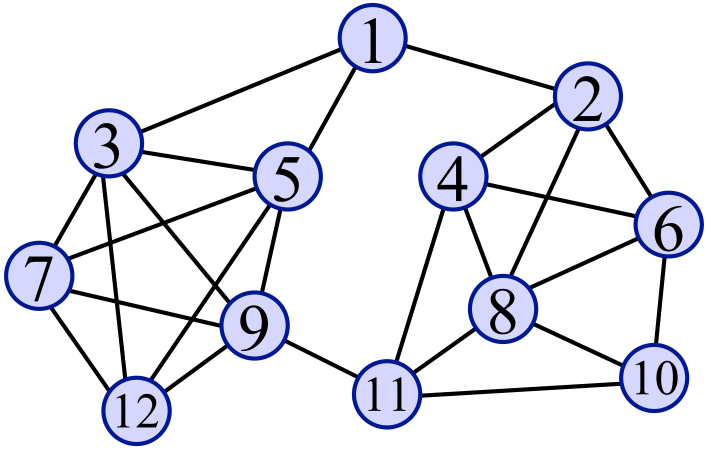
- The parent-child relation on the left graph

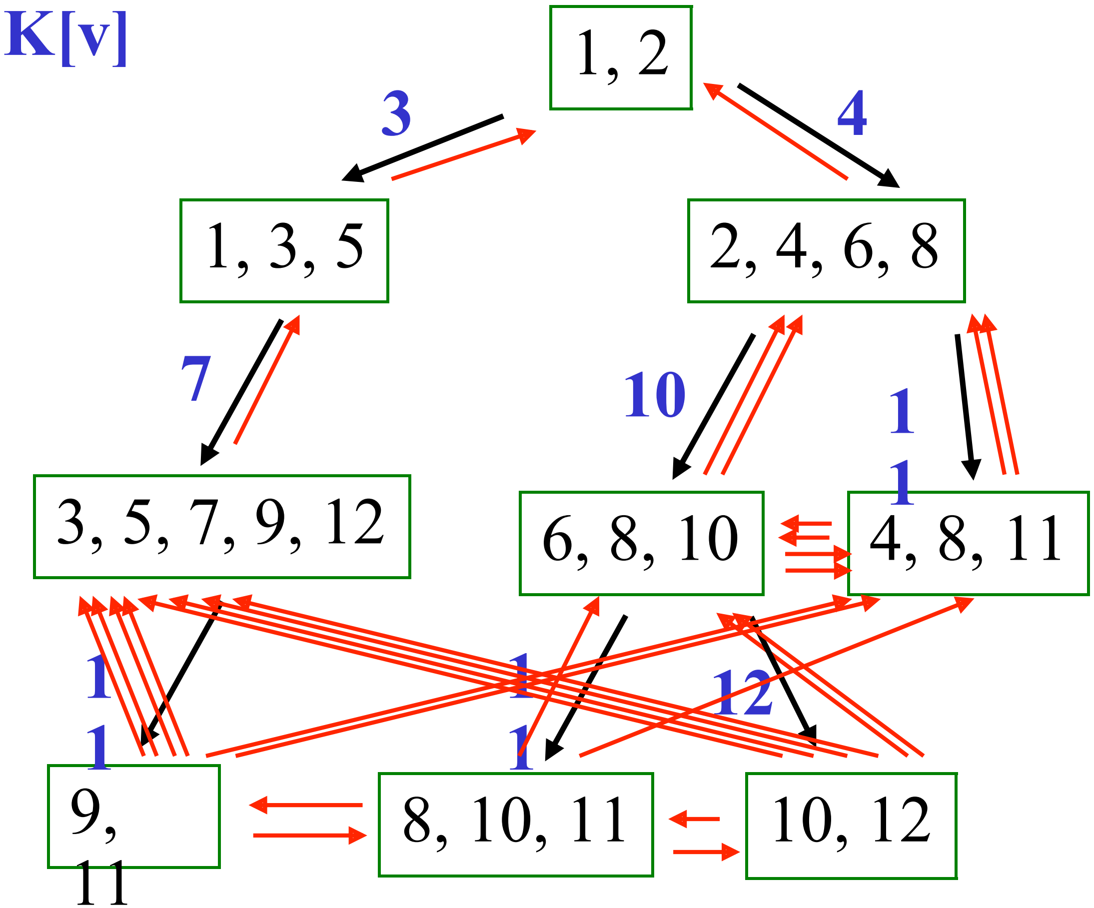


Example

- The parent-child relation on the left graph

- The red-lines are moves by **K[v]**





1-5 Reverse Search for Non-Isomorphic Tree Enumeration

Tree Enumeration

- Previous enumeration problems aim to enumerate **“substructures”** of the given instances (ex. paths in a graph)

- On the other hand, there is a problem of finding “all structures” in the given specified class (ex, matrices)

- For some classes, the problem is trivial
  + **paths, cycles**: lengths of 1, 2, …
  + **cliques**: sizes of 1, 2, …
  + **permutations** of size **n**

- For some classes, the problem is non-trivial
  + **trees, crossing lines** (in plane)**, matriods**, **01-matrices**…

Isomorphism

- On non-trivial structures, we have to take care of "**isomorphism**"

Isomorphism: a structure is isomorphic to another if there is one-to-one correspondence between the elements with keeping some condition

+ a ring sequence (necklace) is isomorphic to another iff it can be transformed to another by rotation

+ a matrix is isomorphic to another iff it can be transformed to the other by swapping rows, and swapping columns

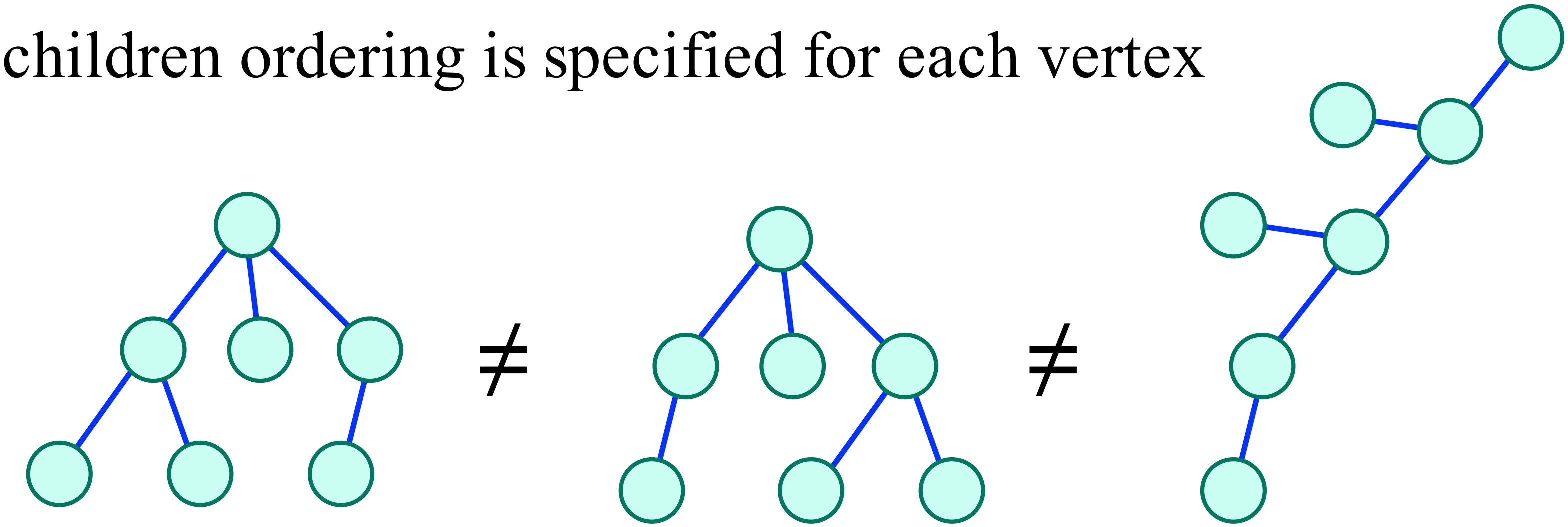
+ a graph is isomorphic to another iff there is a one to one mapping between vertices preserving the adjacency

Enumerate all structures so that no two are isomorphic

Ordered Tree

- Consider enumeration of trees
- Tree has many classes
  among them, we first consider ordered trees

Ordered tree: a rooted tree
s.t. a children ordering is specified for each vertex



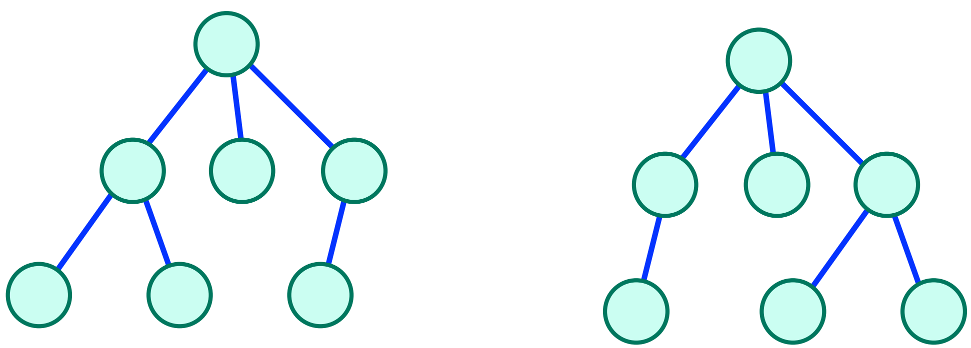
They are isomorphic in the sense of tree (graph),
but the orders of children, and the roots are different

Ambiguity on Representation

- Trees (graphs) are represented by combination of sets, thus we need to put indices to vertices (in the case of data structure, same)

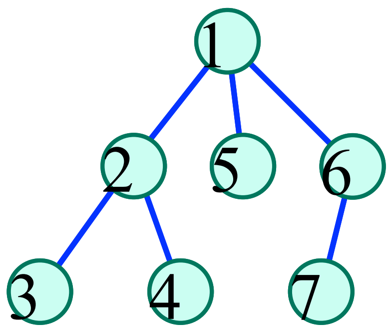
- It results ambiguity on the representation
  - there are many ways to put indices

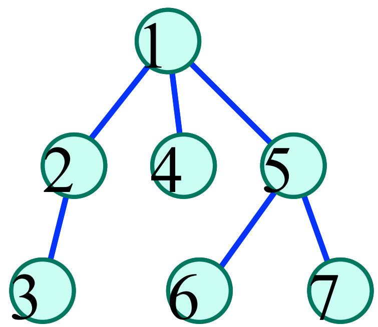
- By putting the indices in a unique way, or representing by other objects, we can avoid the ambiguity



Left-first DFS

- Put indices to vertices by visiting order of depth-first search that visits the leftmost child first, and the remaining from left to right
  - indices are put uniquely
  - an ordered tree is isomorphic another if any its edge is included in the other (and #edges are equal)

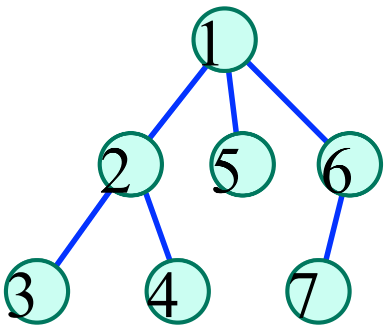




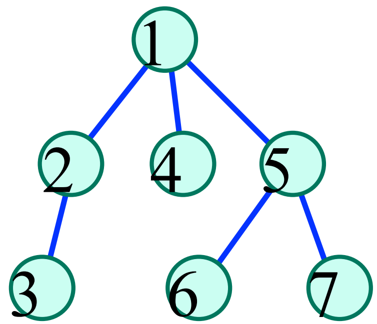
Isomorphism can be checked by comparing edge sets

Depth Sequence

- The left-first DFS can be used to encode ordered trees
- The movement of the DFS is encoded by the sequence of the depth of the visiting vertices (**depth sequence**)
  - the sequence of depths of the vertices ordered by the indices



0122112



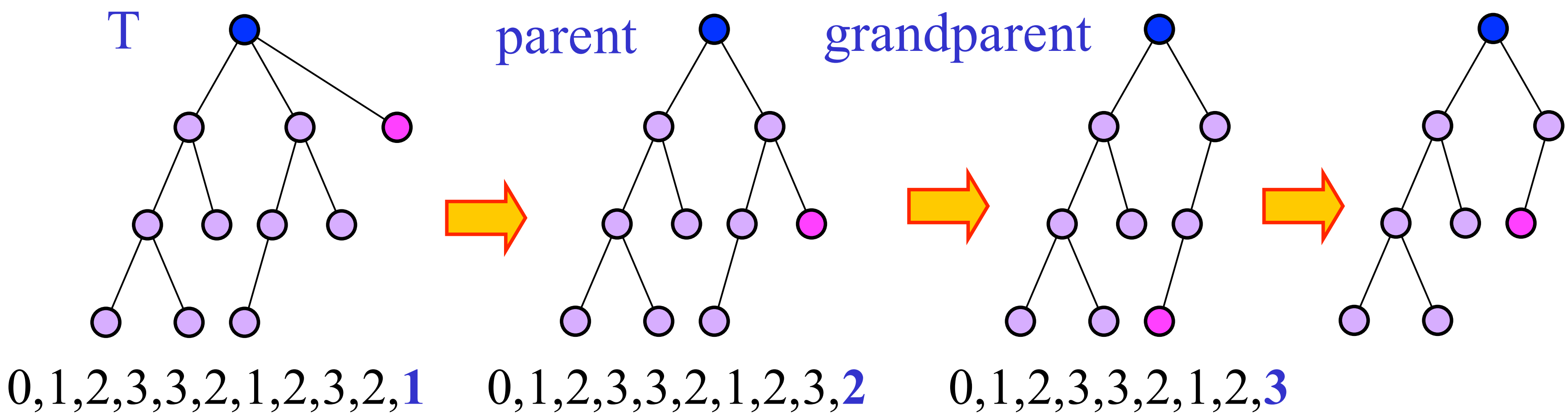
012**1**122

Isomorphism can be checked by comparing the sequences

Parent-Child Relation for Ordered Trees

- Based on the idea of these representations, we define the parent of each ordered tree

- The parent of an ordered tree is defined by the tree, obtained by removing the vertex having the largest index

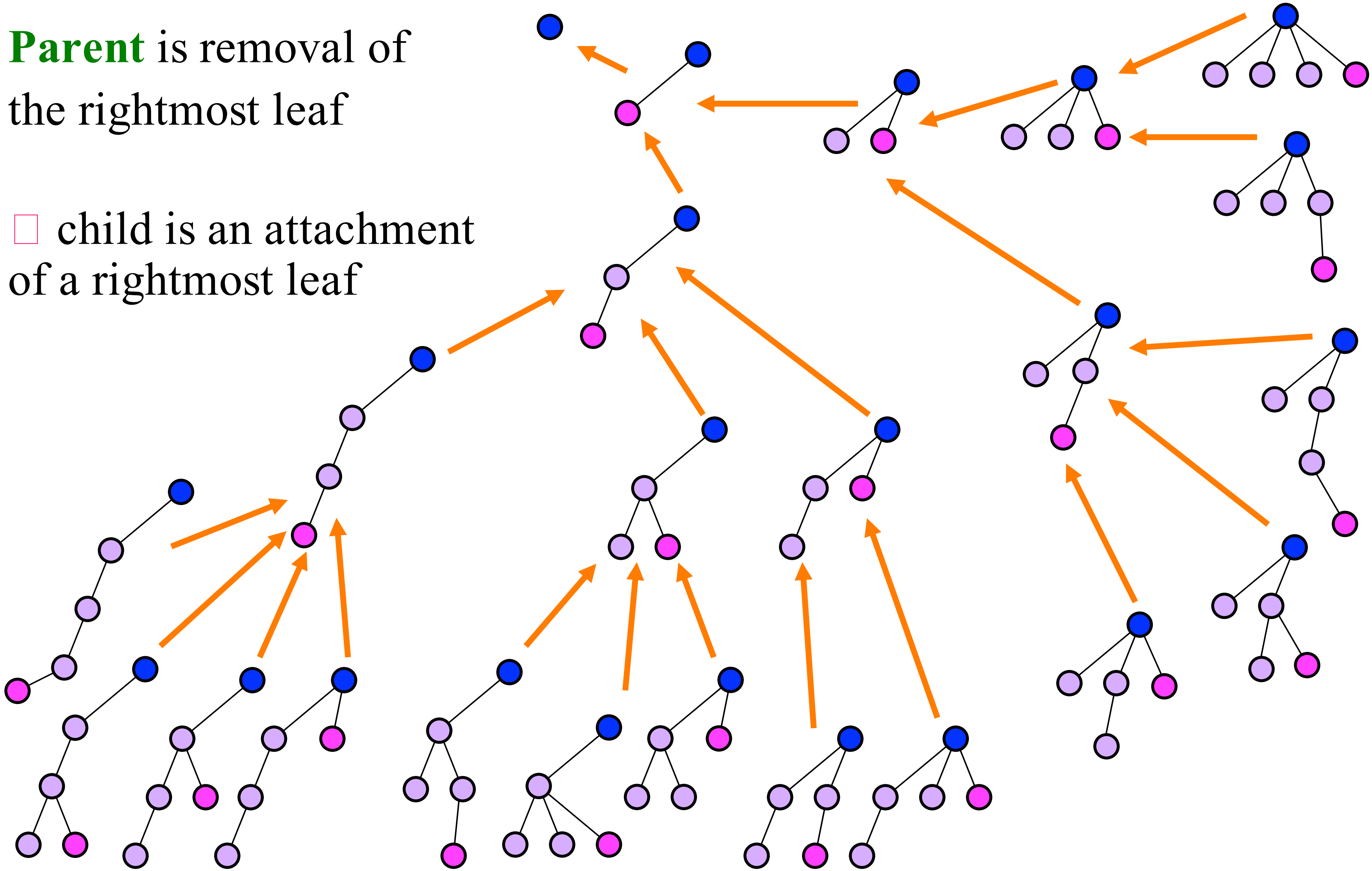


size decreases by going to the parent
□ acyclic & spans all ordered trees

Family Tree of Ordered Trees

Parent is removal of the rightmost leaf

□ child is an attachment of a rightmost leaf

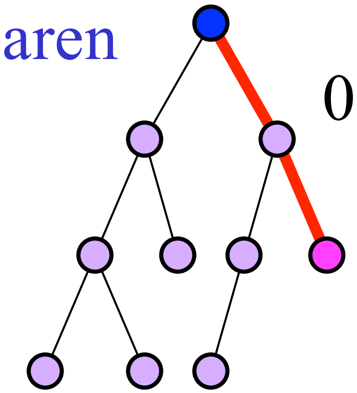


Finding Children

- For an ordered tree T, we can obtain its children by adding a vertex so that the vertex has the largest index

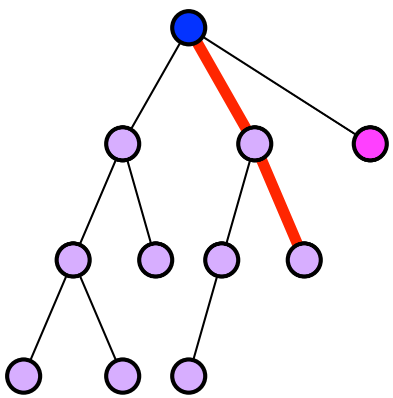
   add to right-hand of the rightmost path

paren
t

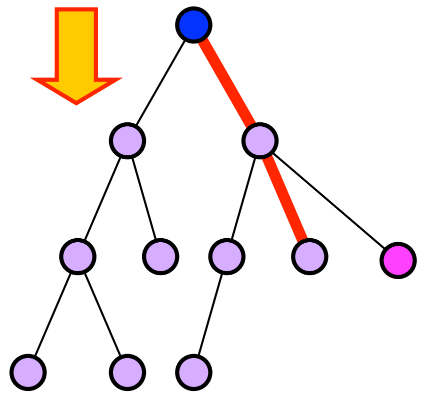


0,1,2,3,3,2,1,2,3,**2**

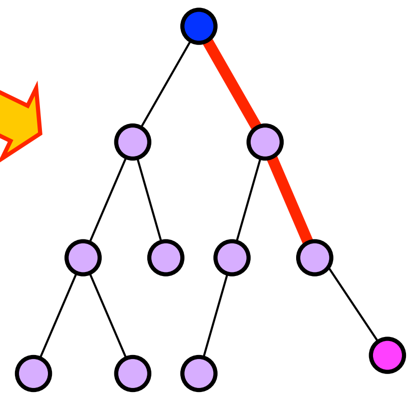
addition always yields a child



0,1,2,3,3,2,1,2,3,2,**1**



0,1,2,3,3,2,1,2,3,2,**2**



0,1,2,3,3,2,1,2,3,2,**3**

Pseudo Code

- By giving the size limitation, we can enumerate all ordered trees of size less than the specified number **k**

EnumOrderedTree (T)
1. **output** T
2. **if** size of T = k **then return**
3. **for** each vertex **v** in the right most path
4. add a rightmost child to **v**
5. **call EnumOrderedTree** (T)
6. remove the child added in **4**
7. **end for**

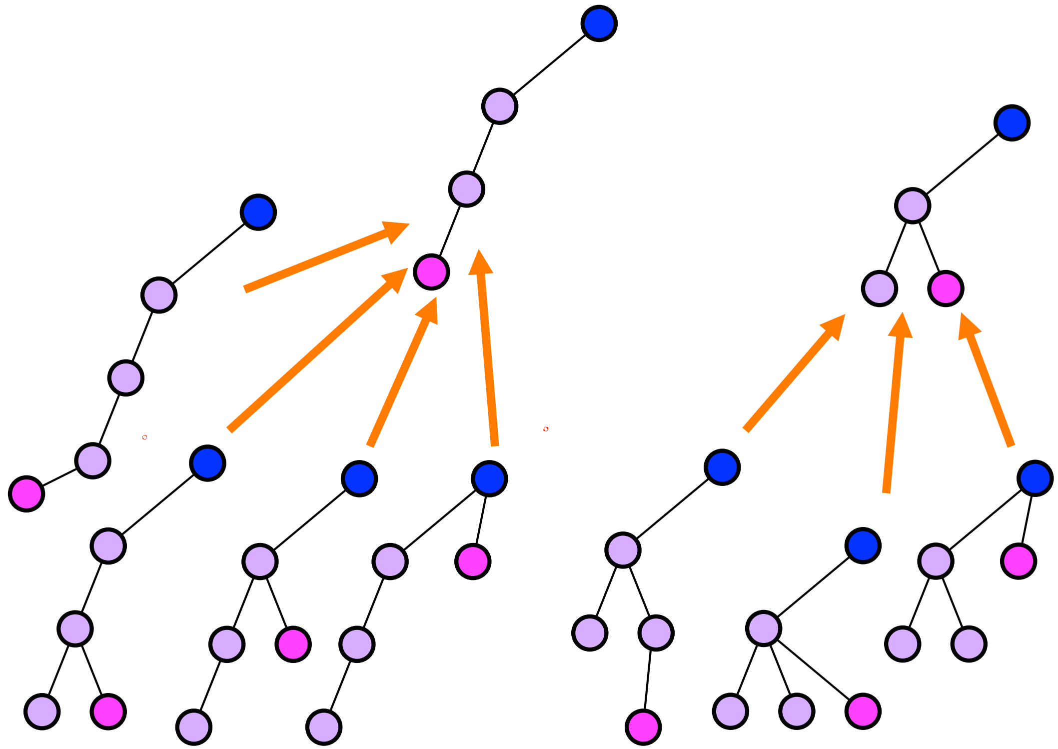
The inside of the for loop takes constant time, thus time complexity is **O(1)** for each (output by difference from the previous)

Ordered Trees □ Un-ordered Trees

- There are many ordered trees isomorphic to an ordinary un-ordered tree (rooted tree)

- If we enumerate un-ordered trees in the same way, many duplications occur

Use canonical form

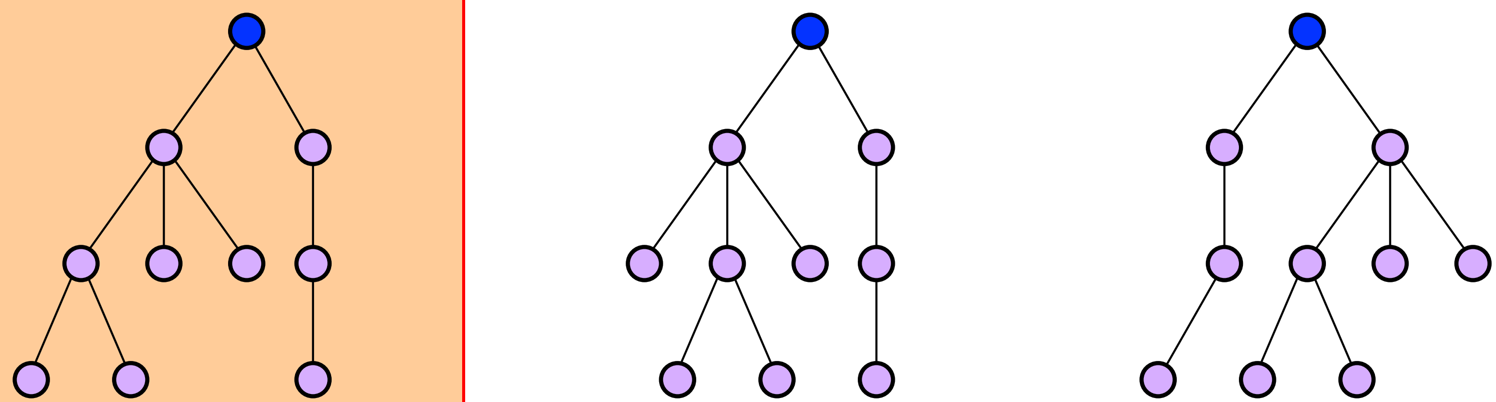


Canonical Form

- Ordered trees are isomorphic □ depth sequences are the same
- **left heavy embedding** of a rooted tree **T**

  the lexicographically maximum depth sequence, among all ordered trees obtained from **T** (by giving children orderings)

- Rooted trees are isomorphic □ left heavy embeddings are the same

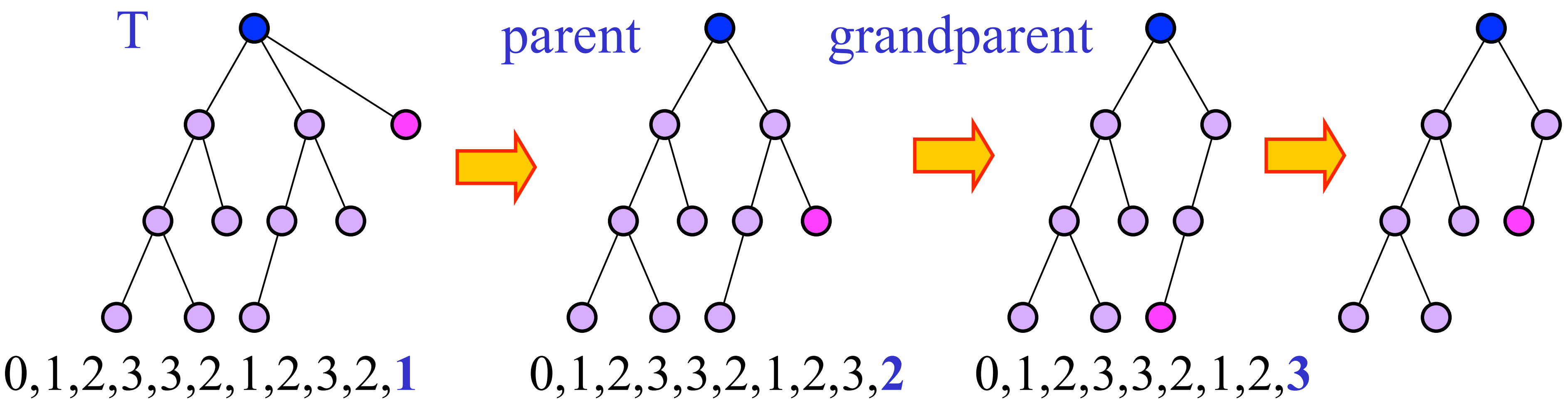


0,1,2,3,3,2,2,1,2,3   0,1,2,2,3,3,2,1,2,3   0,1,2,3,1,2,3,3,2,2

Parent-Child Relation for Canonical Forms

• The parent of left-heavy embedding **T** is the removal of the rightmost leaf (same as ordered trees)

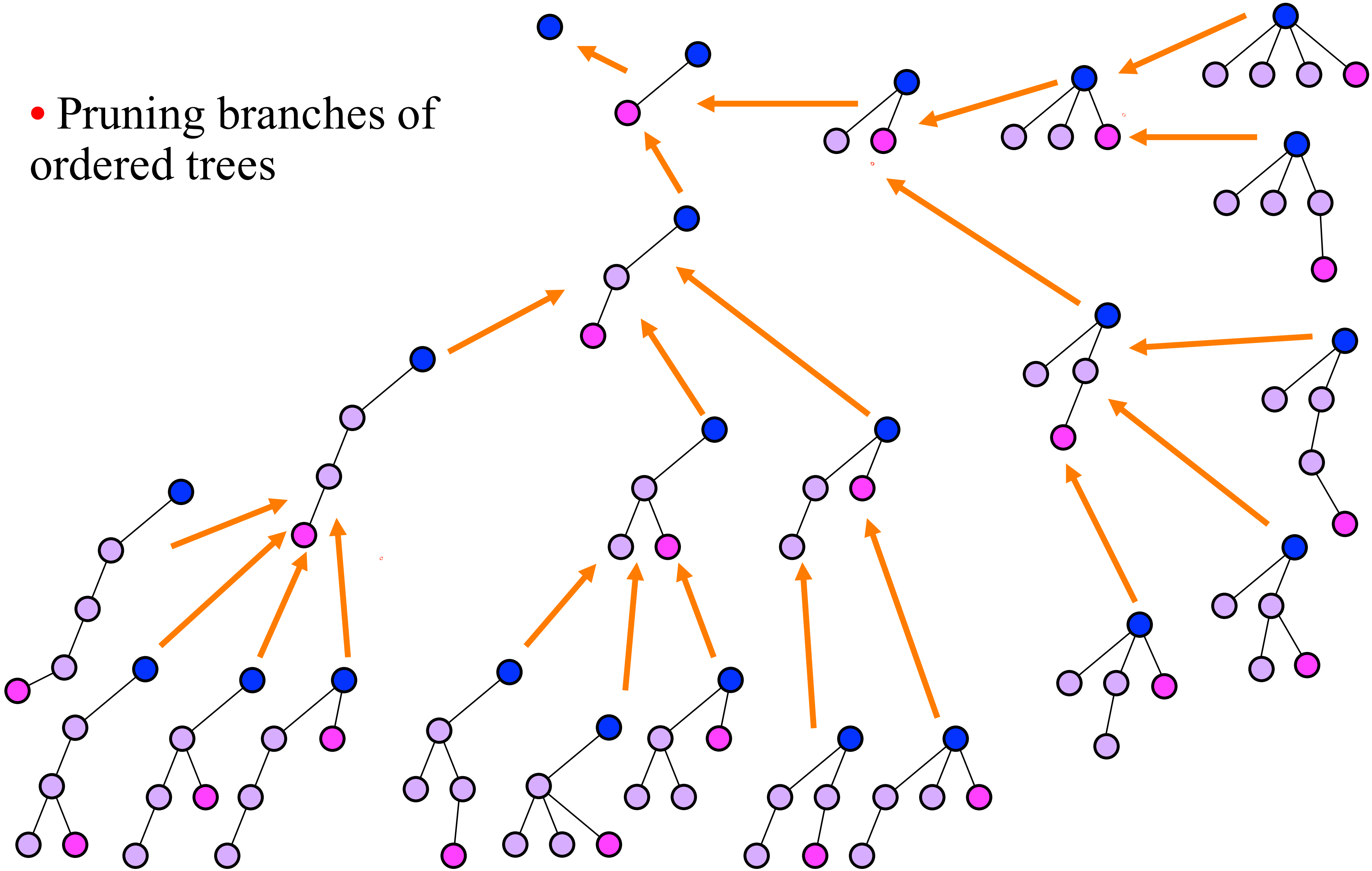
 the parent is also a left-heavy embedding, since the rightmost subtree becomes lexicographically smaller by the removal



The relation is acyclic and spanning all

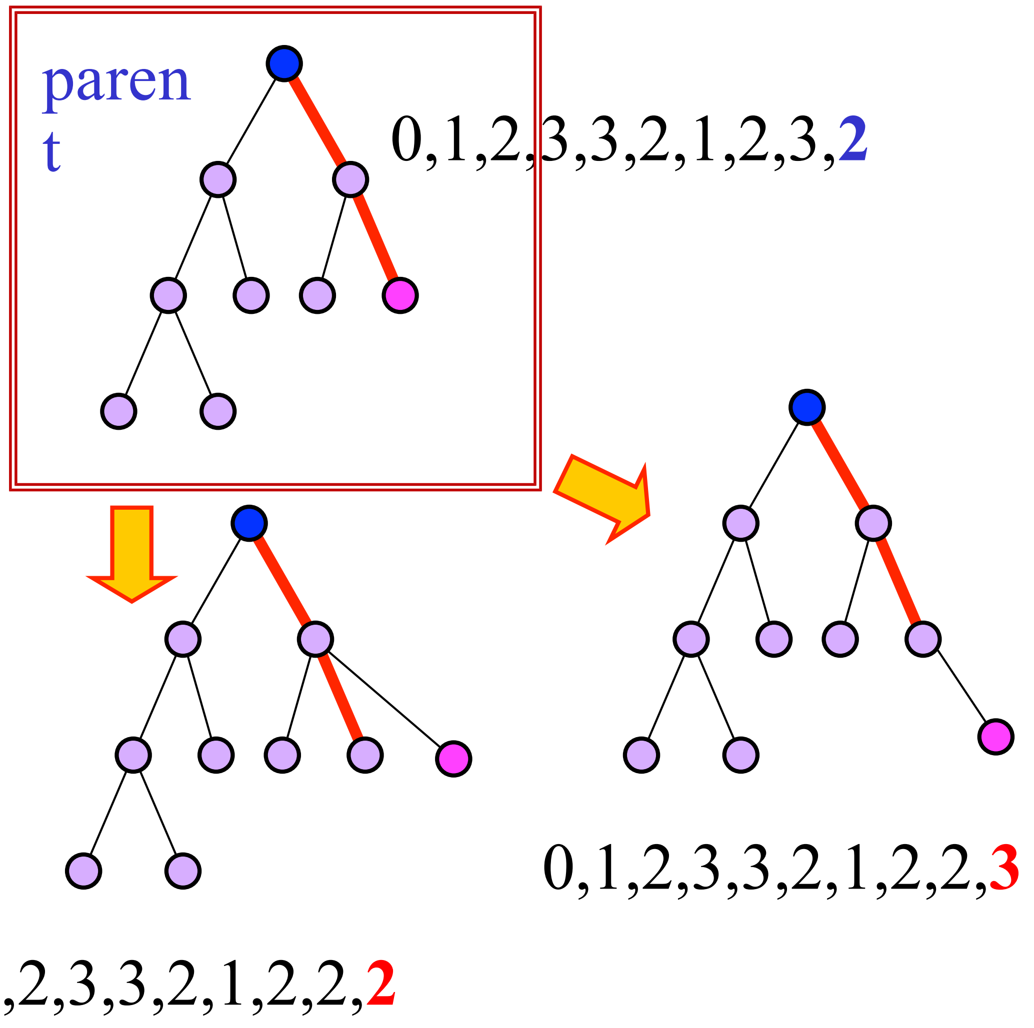
Family Tree of Un-ordered Trees

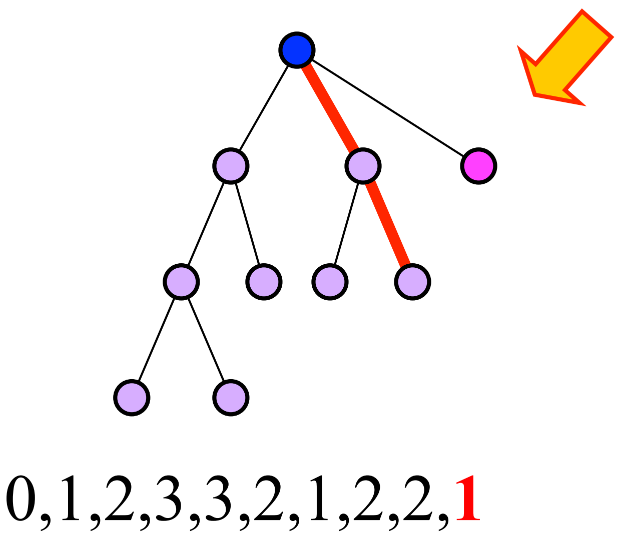
- Pruning branches of ordered trees



Finding Children

- Any child of a rooted tree (parent) is obtained by adding a vertex so that it is the rightmost leaf
- However, some additions do not yield a child





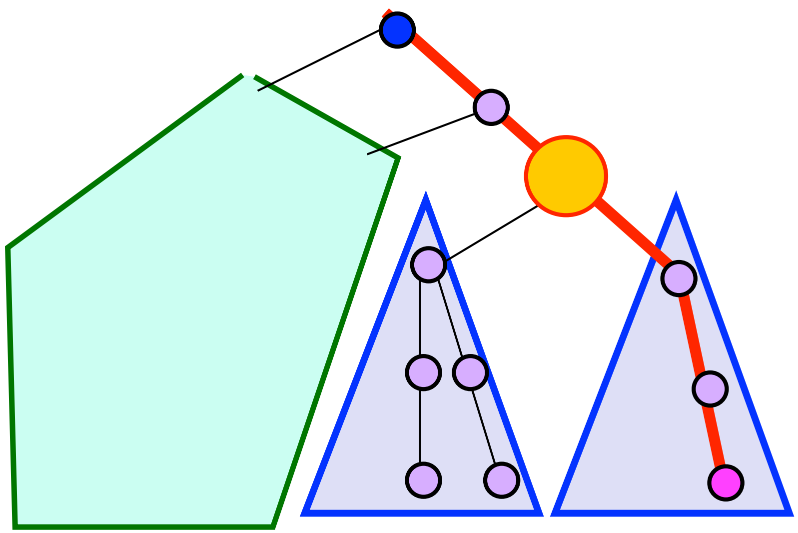
Finding Children

- Addition is not a child
  - □ at some level, right subtree becomes larger than the left

- It happens only when the depth sequence of the right is a prefix of that of the left

- Below the next depth of the left, no addition yields a child
- For all above that, yields a child

- We have to take care only the upmost such vertex (being prefix)

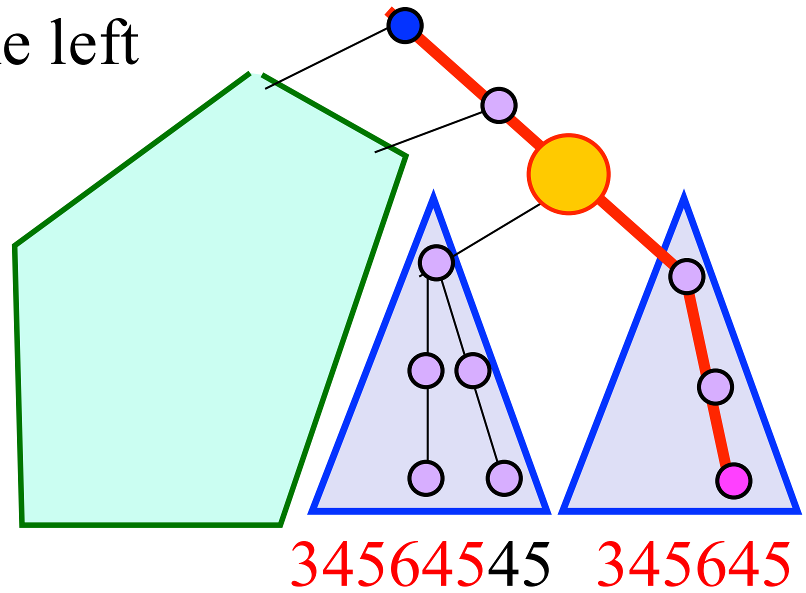


□ violate only lower prefix □ corresponding prefix on the left too

Copy Vertex

- **Copy vertex**
  - the upmost vertex s.t. the right subtree is a prefix of the left

- Copy vertex changes by the addition of the rightmost leaf. It
  + does not change
    if the addition is the same level to the left
  + becomes to **u**, if the level is above
    (**u** is the parent of the added leaf)



We can compute copy depth in constant time for each child

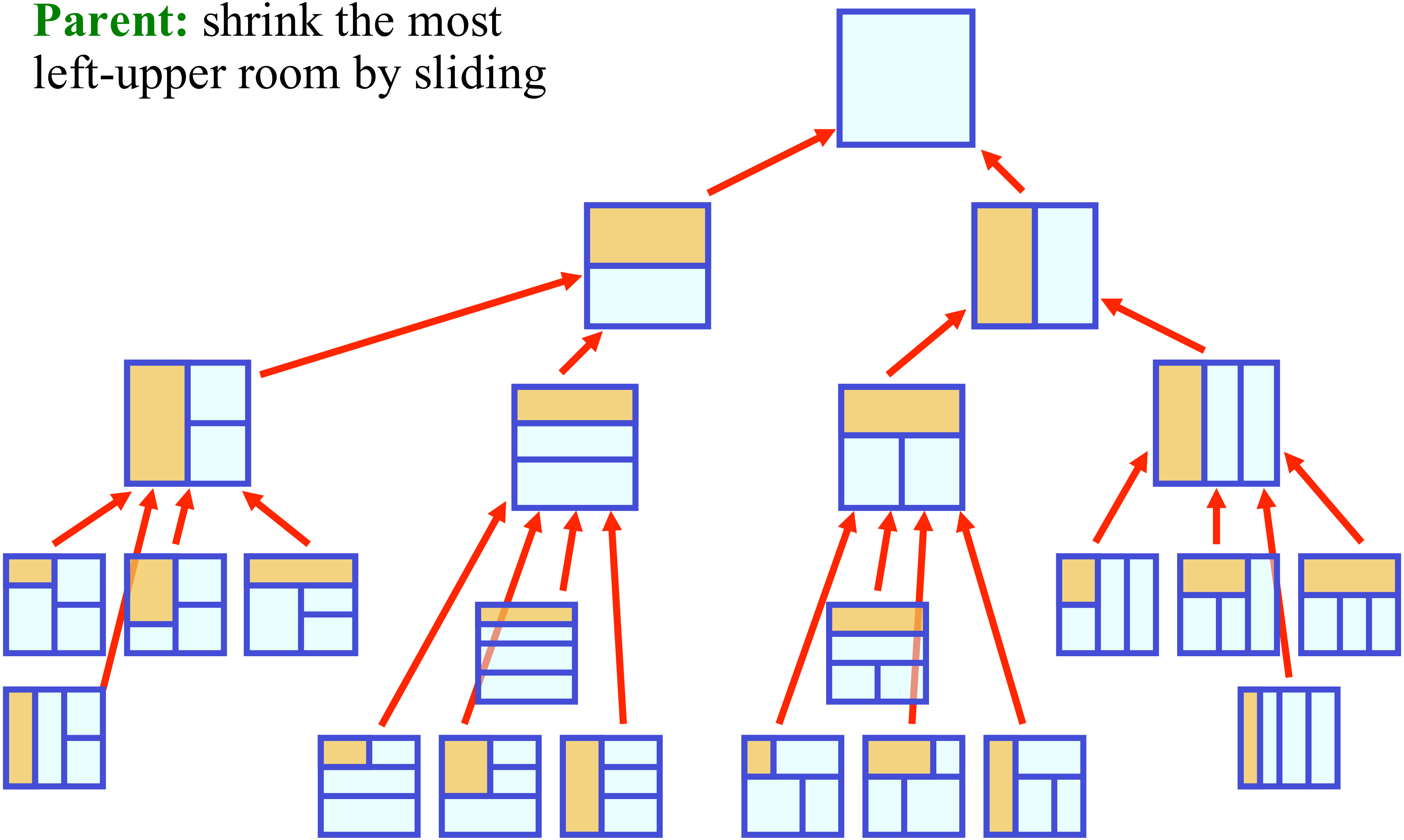
Pseudo Code

```
EnumRootedTree (T, x)
output T
 if size of T = k then return
y := the vertex next to x in the depth sequence
 for each v in right most path, in increasing order of the depth
    c := the rightmost child of v
    add a rightmost child to v
    if depth of v = depth of y then
            call EnumRootedTree (T, y); break
    call EnumRootedTree (T, c)
    remove the rightmost child of v
end for
```

The inside of the for loop takes **O(1)** time, thus the time complexity is **O(1)** for each (output by difference from the previous)

Other Family Tree: Floor Plan

Parent: shrink the most left-upper room by sliding



Conclusion

- Definition of enumeration algorithm
- Motivations and applications
- Difficulty
- Basic schemes
  + **Backtracking:** feasible solutions to knapsack problem
  + **Binary partition:** st-paths of a graph
  + **Reverse search:** maximal cliques, ordered tree, rooted tree

References

D. Avis and K. Fukuda, Reverse Search for Enumeration, Discrete Appl. Math. 65, 21-46 (1996)

st-path, cycle

D. Eppstein, Finding the k Shortest Paths, FOCS94, 154-165 (1994)

D. B. Johnson, Finding All the Elementary Circuits of a Directed Graph, SIAM J. Comp. 4, 77-84 (1975)

R. C. Read and R. E. Tarjan, Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees, Networks 5, 237-252 (1975)

Ordered Trees & Rooted Trees

T. Asai, H. Arimura, T. Uno, S. Nakano, Discovering Frequent Substructures in Large Unordered Trees, DS2003, LNAI 2843, 47-61 (2003)

S. Nakano, T. Uno, Constant Time Generation of Trees with Specified Diameter, WG2004, LNCS 3353, 33-45 (2004)

S. Nakano, T. Uno, Generating Colored Trees, WG2005, LNCS 3787, 249-260 (2005)

S. Nakano, T Uno, Efficient Generation of Rooted Trees, Tech. Rep. NII , 2003

References

Spanning tree

D. Eppstein, Finding the k Smallest Spanning Trees, SWAT '90, LNCS 447, 38-47 (1990)

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

N. Katou, T. Ibaraki and H. Mine, An Algorithm for Finding K Minimum Spanning Trees, SIAM J. Comp.10, 247-255 (1981)

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)

References

Clique

E. A. Akkoyunlu, The Enumeration of Maximal Cliques of Large Graphs, SIAM J. Comp. 2,1-6 (1973)

D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, On Generating All Maximal Independent Sets, Info. Proc. Lett. 27, 119-123 (1988)

T. Kashiwabara, S. Masuda, K. Nakajima and T. Fujisawa, Generation of Maximum Independent Sets of a Bipartite Graph and Maximum Cliques of a Circular-Arc Graph, J. Algo. 13, 161-174 (1992)

S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, A New Algorithm for Generating All the Maximum Independent Sets, SIAM J. Comp. 6, 505-517 (1977)

K. Makino, T. Uno, New Algorithms for Enumerating All Maximal Cliques, SWAT2004, LNCS 3111, 260-272 (2004)

E. Tomita, A. Tanaka, H. Takahashi, The Worst-case Time Complexity for Generating all Maximal Cliques and computational experiments", Theoretical Computer Science 363, 28-42 (2006)

References

Matching

K. Fukuda and T. Matsui, Finding All the Perfect Matchings in Bipartite Graphs, Appl. Math. Lett. 7, 15-18 (1994).

K. Fukuda and T. Matsui, Finding All the Minimum Cost Perfect Matchings in Bipartite Graphs, Networks 22, 461-468 (1992)

C. R. Chegireddy and H. W. Hamacher, Algorithms for Finding K-best Perfect Matchings, Discrete Appl. Math. 18, 155-165 (1987)

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)

Exercises 1

Brute Force

1-1. Design an algorithm to enumerate all combinations of integers **a1,...,a10** ranging 1 to 10 such that **a2 + a4 + a6 = 20, a1 + a3 + a5 = 10**, **a1 + a7 + a9 = 10**, **a2 + a8 + a10 = 20**. Brute force is OK.

1-2. Design an algorithm to enumerate all graphs of **n** vertices and **m** edges so that no two isomorphic graphs will be enumerated. We are given a function to check isomorphism of given two graphs **G1** and **G2**. Brute force is OK.

1-3. Design an algorithm to enumerate all ways to put marks (from **1** to **k**) on vertices in a cycle of **n** vertices, so that no two solutions will be the same by a rotation.

Lv. 1: Brute Force 2

1-4. Using algorithm 1-2, we want to design an algorithm for enumerating all graphs including no clique of size 4. Design an efficient pruning method.

1-5. Using algorithm 1-2, we want to design an algorithm for enumerating all graphs such that there is a vertex **v** such that distance from the given vertex **r** to **v** is at least **k**. Design an efficient pruning method or algorithm.

1-6. Using algorithm 1-2, we want to design an algorithm for enumerating all graphs such that the degree of any vertex is at most **k**. Design an efficient pruning method.

Backtrack

1-7. Design a backtracking algorithm for the following problem:

For given a sequence of numbers **a1,…,an**, enumerate all its subsequence such that any two consecutive numbers **ai** and **aj** satisfies **ai < aj**.

1-8. Design a backtracking algorithm for the following problem:

For given a set of points in a plane, a non-crossing graph is a graph whose vertex set is the point set, and its edge set is a set of segments whose ends are on the points, such that no two segments intersects except for their ends. Enumerate all non-crossing trees for given a point set

Backtrack

1-9. Design a backtracking algorithm for the following problem:

For given a set of vectors **x1,…,xn** of composed of positive integers, enumerate all sets of vectors such that their sum is no greater than given vector **b**. (subset **X** s.t., $\Sigma x \in X \leq b$)

1-10. Design a backtracking algorithm for the following problem:

For given a set of rectangles and a square, enumerate all possible locations of the subset of the rectangles s.t. no two rectangles overlap. The left-up corner of each rectangle has to be placed at an integer grid point, and the edges of rectangles has to axis parallel (so, 90 degree rotation is allowed).

Backtrack

1-11. Design a backtrack algorithm for the following problem, and analyze its time complexity

For given a graph, enumerate all matchings of the graph

1-12. Design a backtrack algorithm for the following, and analyze its time complexity

For given a sequence of letters, enumerate all its subsequences that form palindromes, i.e., forming a,b,c,…,d,d,…,c,b,a

Binary Partition

1-13. Design a binary partition algorithm for the following problem, and analyze its time complexity

For given a set of points in a plane, enumerate all non-crossing spanning trees

1-14. Design a binary partition algorithm for the following problem, and analyze its time complexity

For given a set of points in a plane, and two points **s** and **t**, enumerate all non-crossing **s**-**t** paths (simple paths whose ends are **s** and **t**)

Binary Partition

1-15. For two perfect matchings **M** and **M'** of a bipartite graph **G**, the symmetric difference between **M** and **M'** is composed of disjoint cycles. Further, the symmetric difference between **M** and an alternating cycle in which edges of **M** and edges not in **M** appear alternatively results a perfect matching different from **M**. Design a binary partition algorithm by using this fact.

1-16. Design a binary partition algorithm for the following problem, and analyze its time complexity

For a given partial order, a chain is a sequence of elements **e1,…,em** s.t., **ei < ei+1** holds for any **i**. Enumerate all maximal chains that are included in no other chain

Exercises

1-17. Design a binary partition algorithm for the following, and analyze its time complexity

For a connected graph, a minimal cut is a partition of vertices such that the subgraphs induced by each group is connected. For given two vertices **s** and **t**, enumerate all minimal cuts s.t. one component includes **s** and not **t**

1-18. Design a binary partition algorithm for the following problem, and analyze its time complexity

For given a graph such that each edge is colored, enumerate all matchings of the graph s.t. any two edges have different colors

Reverse Search

1-19. Design a reverse search algorithm for the following, and analyze its time complexity

For given a sequence of numbers **a1,…,an**, enumerate all its subsequence such that any two consecutive numbers **ai** and **aj** satisfies **ai < aj**

1-20. Design a reverse search algorithm for the following problem, and analyze its time complexity

For a given graph **G=(V,E)**, enumerate all vertex sets that induces a subgraph of edge density at least **θ** (edge density of graph induced by a vertex set **K** = #edges in **K / (|K|(|K|-1)**)

Exercises

1-20. Design a reverse search algorithm for the following problem, and analyze its time complexity

For a given directed graph **G=(V,E)**, enumerate all acyclic subgraphs

1-21. Design a reverse search algorithm for the following, and analyze its time complexity

For given red points and black points in **Rd**, enumerate all sets of red points such that it can be included in a hyper rectangle without including any black point