

グラフ探索アルゴリズム とその応用

2011 / 05 / 04

保坂和宏





内容

- グラフの扱い方
 - 深さ優先探索 (DFS)
 - 橋, 関節点
 - 幅優先探索 (BFS)
 - Dijkstra 法, 0-1-BFS
 - 辞書順幅優先探索 (LexBFS)
 - cograph
- 
- 

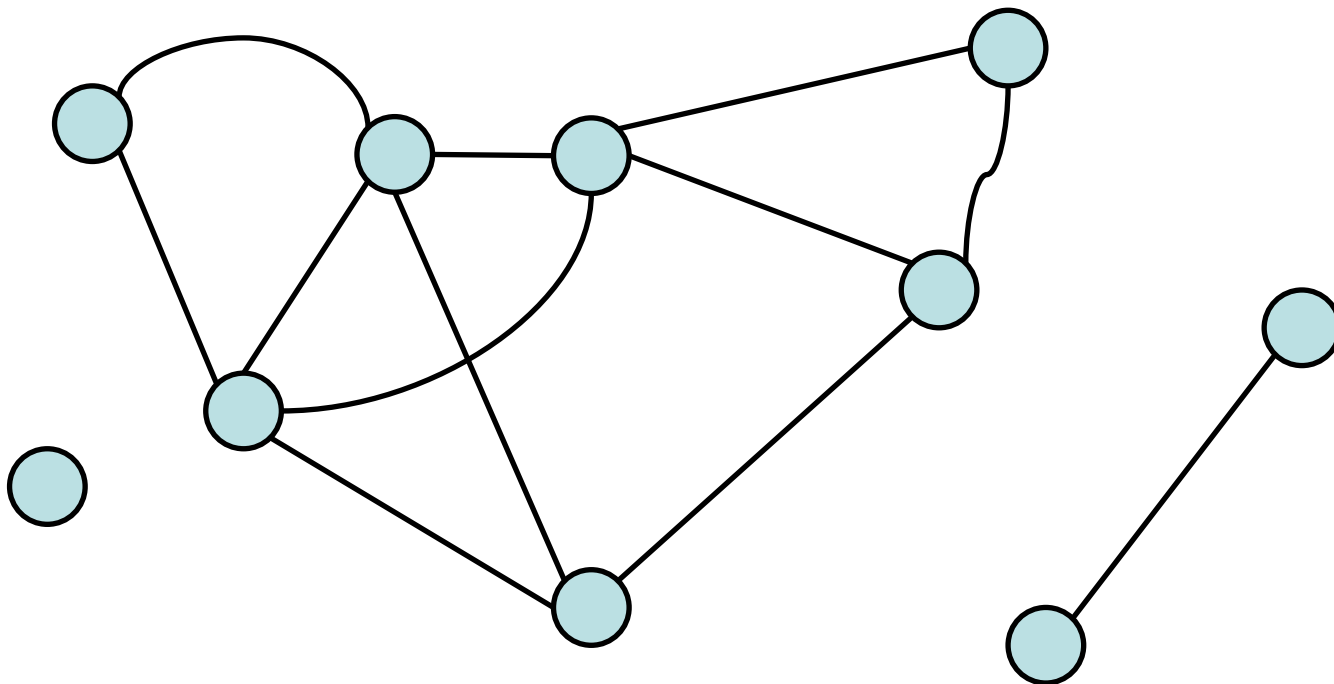


グラフ

- グラフ理論からの出題
 - Islands (IOI 2008)
 - Regions (IOI 2009)
 - Saveit (IOI 2010)
 - Regions (JOI 2010 春合宿)
 - Finals (JOI 2010 春合宿)
 - Joitter (JOI 2011 選考会)
 - Shiritori (JOI 2011 選考会)
 - Report (JOI 2011 選考会)
 - Orienteering (JOI 2011 選考会)
- 
- 

グラフ

- グラフとは？
 - 「点が線で繋がった構造」

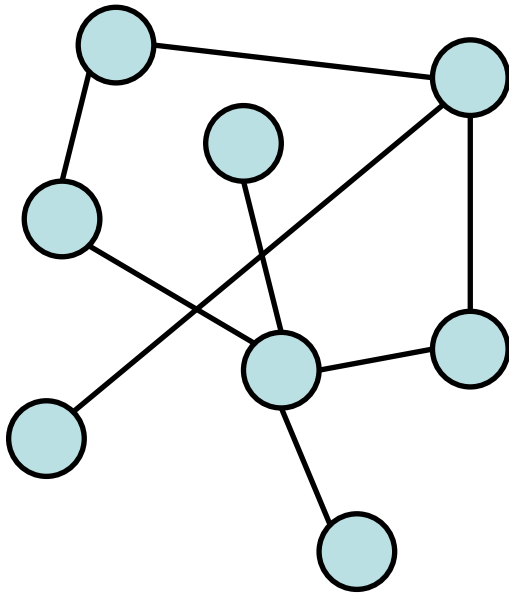


グラフ

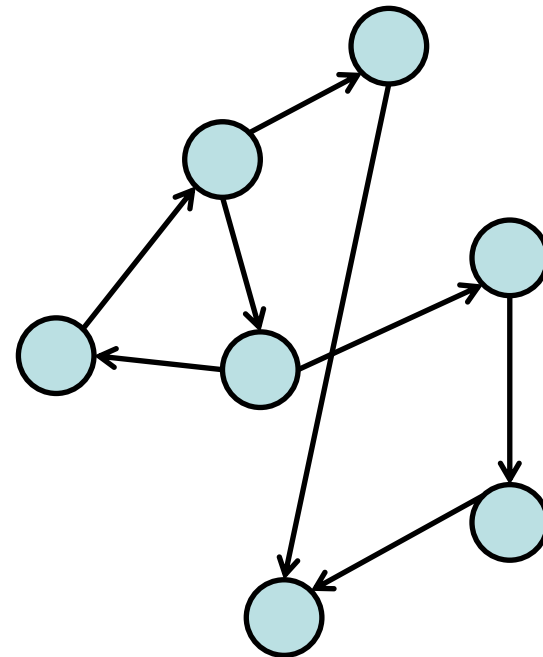
- グラフ (graph) $G = (V, E)$
 - V : 頂点 (vertex) の集合
 - E : 辺 (edge) の集合
 - $n = |V|$ (頂点数), $m = |E|$ (辺数) とおく
- 辺 : 頂点の 2 つ組
 - 無向グラフ : $\{u, v\}$ と $\{v, u\}$ は区別しない
 - 有向グラフ : (u, v) と (v, u) は異なる辺
 - 重み $c: E \rightarrow \mathbb{R}$ が定まっていることも

無向グラフと有向グラフ

無向グラフ

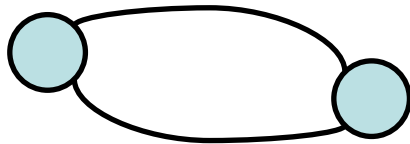


有向グラフ

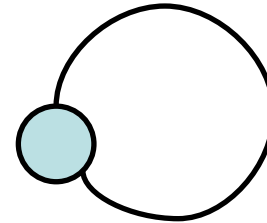


特殊な辺

多重辺



ループ

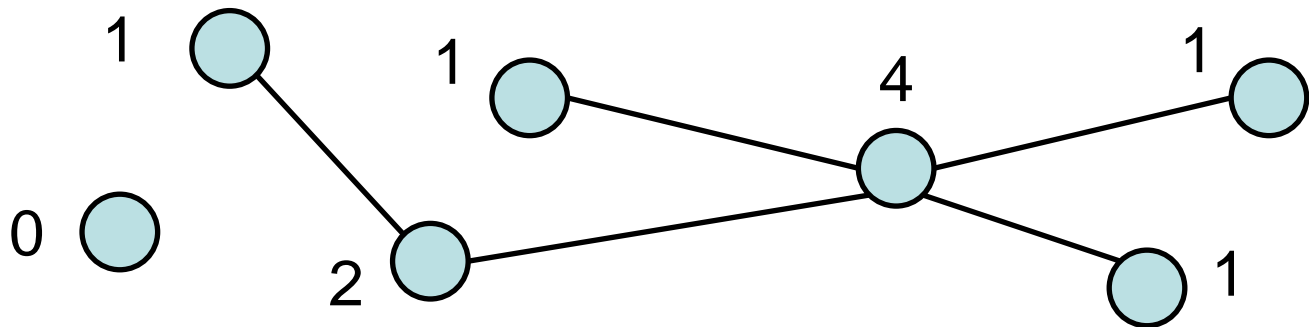


- 単純グラフ
 - 多重辺やループを含まないグラフ
 - 今回よくでてくるのは無向単純グラフ

グラフの用語

- 次数

- 頂点 u の 次数 (degree) とは, u に接続している辺の個数
- 有向グラフに対しては,
 - u を始点とする辺の個数 : 出次数 (out-degree)
 - u を終点とする辺の個数 : 入次数 (in-degree)



グラフの用語

- パス

- 頂点と辺の列 $v_1, e_1, v_2, e_2, \dots, v_k, e_k, v_{k+1}$ であつて, $v_1, v_2, \dots, v_k, v_{k+1}$ と e_1, e_2, \dots, e_k がすべて異なるものをパス (path) という
 - 頂点の重複を許す場合もある

- サイクル

- 頂点と辺の列 $v_1, e_1, v_2, e_2, \dots, v_k, e_k, v_1$ であつて, v_1, v_2, \dots, v_k と e_1, e_2, \dots, e_k がすべて異なるものをサイクル (cycle) あるいは閉路という

グラフの用語

- 連結性

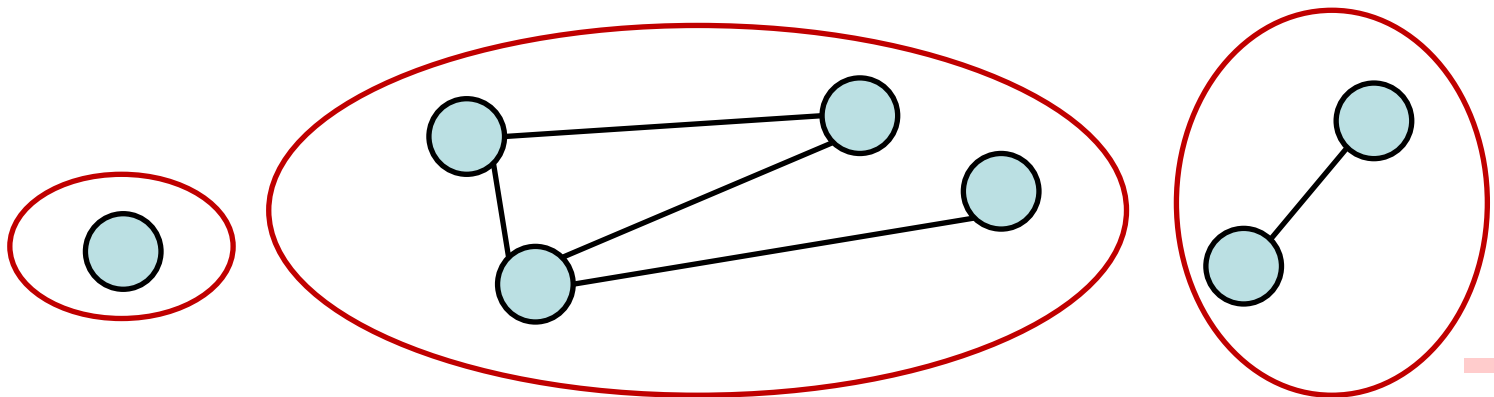
- どの 2 頂点 u, v に対しても, u から v へのパスが存在するとき, グラフは連結 (connected) であるという
 - 有向グラフでは強連結 (strongly connected) という

グラフの用語

- 連結成分

- 「 u から v へのパスも v から u へのパスも存在する」ときに u と v が同じグループに属する, として V をグループ分けしたときの各グループを 連結成分 (connected component) という

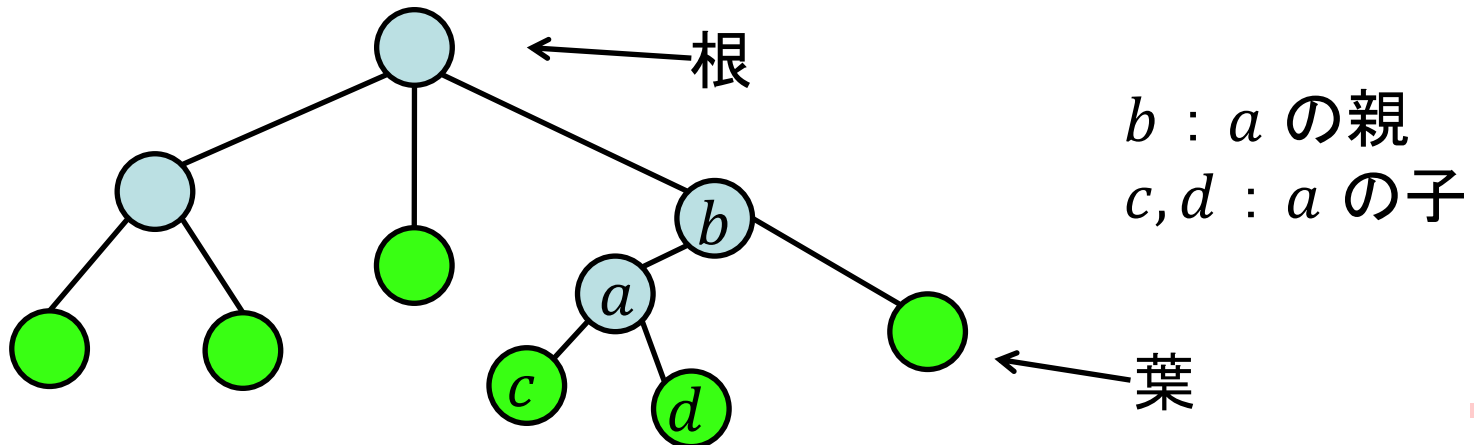
- 有向グラフでは 強連結成分 (strongly connected component) という



グラフの用語

- 木

- 木 (tree) : 連結でサイクルがないグラフ
 - n 頂点の木は辺の数 $m = n - 1$
- 森 (forest) : サイクルがないグラフ
- ある頂点を根 (root) として考えることがある
 - 有向木 : 辺の向きが根から葉の方向





グラフの扱い方



グラフの扱い方

- 隣接行列
 - 実装が単純
 - 頂点の 2 つ組で辺を管理したいとき楽
- 隣接リスト
 - 疎グラフに対して高速・省メモリ
 - 疎グラフ ... $m = O(n)$ くらい
 - 密グラフ ... $m = \Theta(n^2)$ くらい
 - 辺に番号をつけて管理したいとき楽
- グラフを陽に持たない

隣接行列

- 2次元配列 $g[MAX_N][MAX_N]$ を確保
- 辺 uv に対して $g[u][v]$ を更新
 - 無向グラフなら $g[v][u]$ も同じ値に
- 代入する値 (例)

- 重みなし : $g[u][v] = \begin{cases} 1 & (uv \in E) \\ 0 & (uv \notin E) \end{cases}$

- 重みつき : $g[u][v] = \begin{cases} 0 & (u = v) \\ c(uv) & (uv \in E) \\ \infty & (uv \notin E) \end{cases}$

隣接リスト

- 各頂点に対し, その点に接続している辺のリストを管理
 - 有向グラフであれば「その点を始点とする辺」
- 実装方法
 - STL の vector を使う (C++)
 - ポインタを使う
 - 配列でリストを実装
 - おすすめ
 - 配列に格納
 - 次数が小さいとわかっているときに良い

隣接リストの実装

- 配列でリストを実装

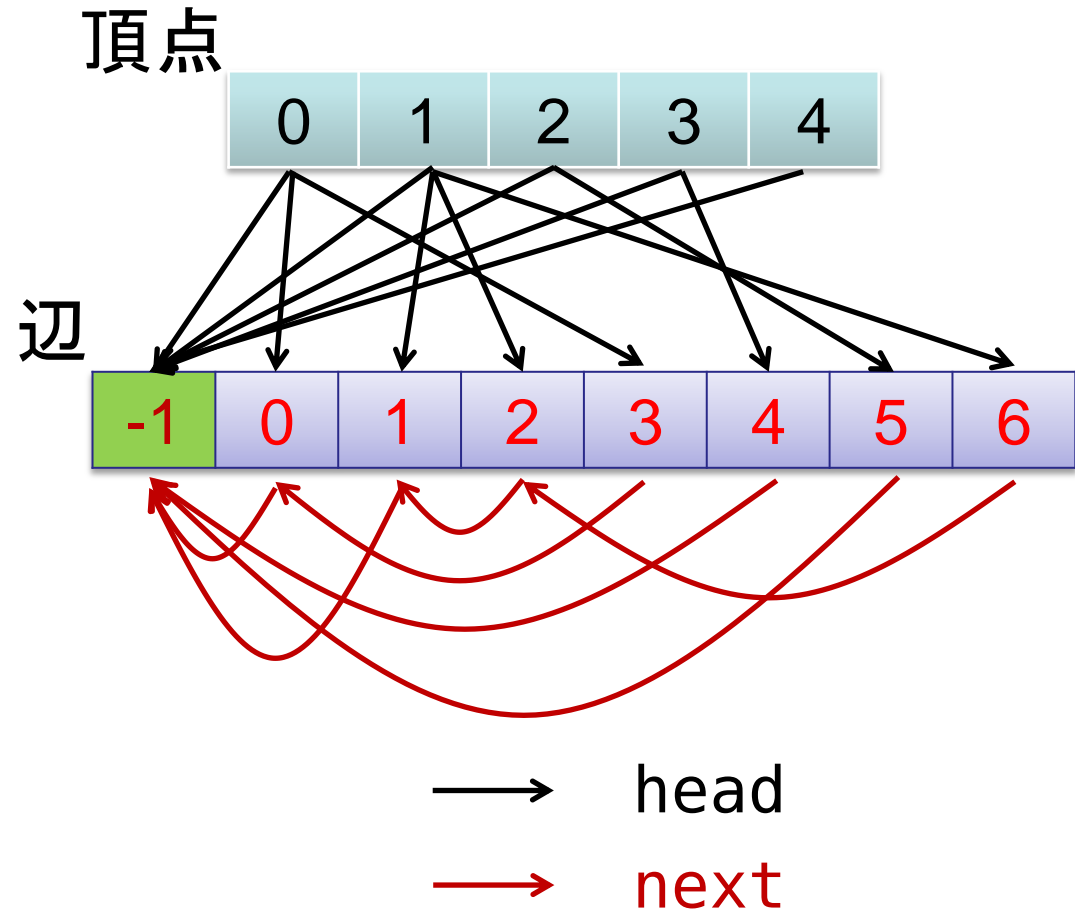
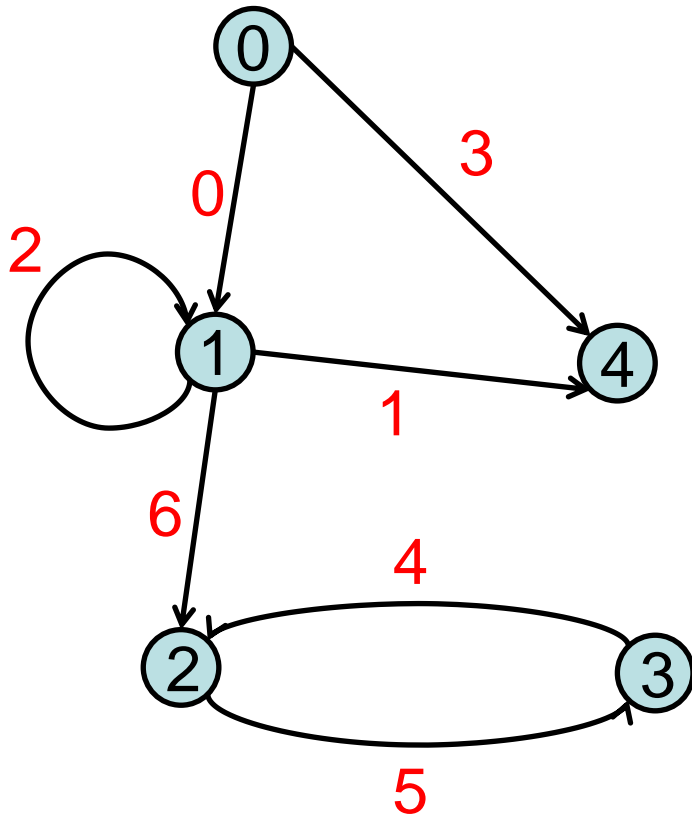
```
int m, head[MAX_N], next[MAX_M], to[MAX_M];

// 初期化
memset(head, -1, sizeof(head));

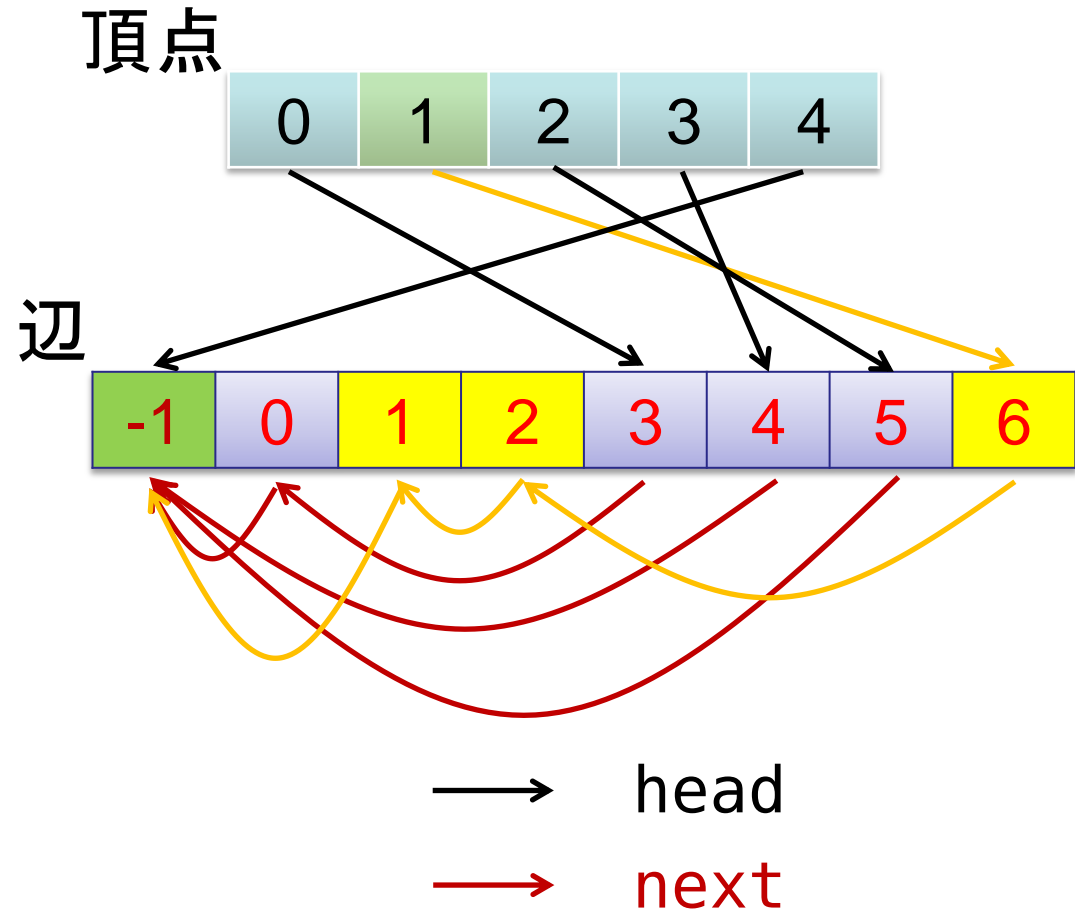
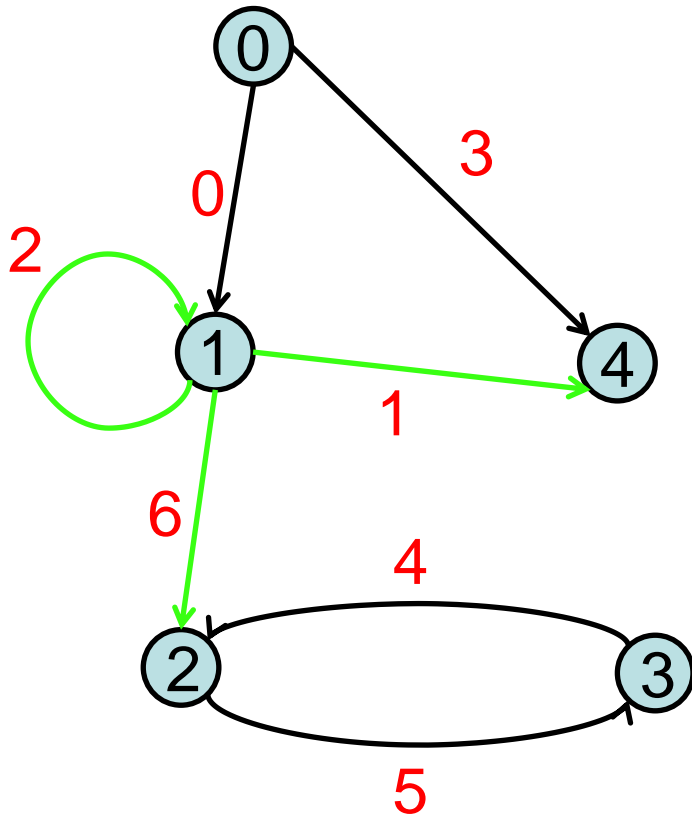
// 辺 uv を加える (無向グラフならば逆向きも加える)
next[m] = head[u]; head[u] = m; to[m] = v; ++m;

// u に接続している辺を調べる (追加と逆順)
for (e = head[u]; e != -1; e = next[e]) {
    // 辺 e = (u, to[e]) に対する処理
}
```

隣接リストの実装



隣接リストの実装



隣接リストの実装

- 配列でリストを実装

```
int m, head[MAX_N], next[MAX_M], to[MAX_M];

// 初期化
memset(head, -1, sizeof(head));

// 辺 uv を加える (無向グラフならば逆向きも加える)
next[m] = head[u]; head[u] = m; to[m] = v; ++m;

// u に接続している辺を調べる (追加と逆順)
for (e = head[u]; e != -1; e = next[e]) {
    // 辺 e = (u, to[e]) に対する処理
}
```

隣接リストの実装

- $N \times (\text{次数の上限})$ がメモリに収まる場合

```
int deg[MAX_N], g[MAX_N][MAX_DEG];

// 初期化
memset(deg, 0, sizeof(deg));

// 辺 uv を加える (無向グラフならば逆向きも加える)
g[u][deg[u]++] = v;

// u に接続している辺を調べる
for (i = 0; i < deg[u]; ++i) {
    // 辺 e = (u, g[u][i]) に対する処理
}
```

隣接リストの実装

- 最初にすべての辺を読み込む方法

```
int deg[MAX_N], p[MAX_N + 1], pp[MAX_N], g[MAX_M];


// すべての辺を読み込んだ後, 次数を配列 deg に保存
for (u = 0; u < N; ++u) {
    p[u + 1] = p[u] + deg[u];
    pp[u] = p[u];
}

// 辺 uv を加える (無向グラフならば逆向きも加える)
g[pp[u]++] = v;

// u に接続している辺を調べる
for (i = p[u]; i < p[u + 1]; ++i) {
    // 辺 e = (u, g[i]) に対する処理
}
```

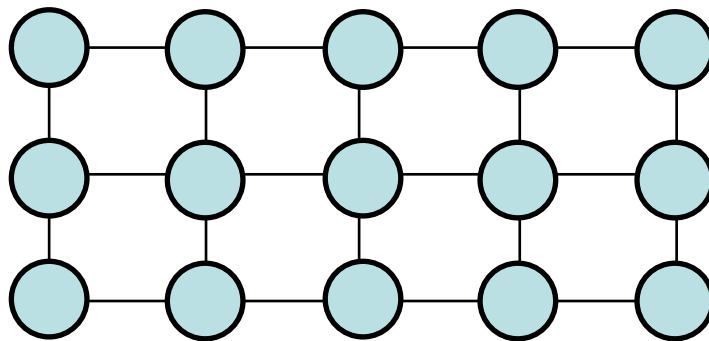


隣接リスト

- 各実装方法の主なデメリット
 - STL の vector を使う (C++)
 - 大きいサイズでは push_back のコストが重い
 - ポインタを使う
 - 自分で構造体を作るのが面倒
 - 配列でリストを実装
 - メモリアクセスの効率が悪い
 - 配列に格納
 - (次数固定) 次数にばらつきがあるとメモリが無駄
 - (1次元) 最初に読み込んで保存する手間がかかる
- 

グラフを陽に持たない

- 「辺 uv があるかどうか」「辺 uv のコスト」
- 「頂点 u に接続している辺」
- これらを予め調べて配列などに保存するのではなく、必要になるたびに計算
- 例



辺集合を直接管理

- 辺 uv を「 u の小さい順 \rightarrow v の小さい順」にすべてまとめてソート (STL の `pair` (C++))
- すべての辺を平衡二分木, ハッシュテーブルなどに入れる
- 隣接リストの代わりになる
- 2 頂点を指定して辺を調べたいときに便利
- 操作に $O(\log m)$ 程度の時間がかかる

深さ優先探索 (DFS)

グラフ探索アルゴリズム

- 以下のアルゴリズムでグラフの頂点を1つずつ取り出す

$R := V.$

while R が空でない:

$u \in R$ を「適切に」選ぶ.

R から u を取り除く.

u に接続している辺を走査して「なにか」する.

- 「適切に」「なにか」は用いるアルゴリズムによる

DFS

- 深さ優先探索 (depth-first search)
 - 「バックトラック法」とも
- 「人間が街で行える探索」
- 再帰, スタック
- 計算量
 - 時間 $O(n + m)$ (隣接行列で持つ場合 $O(n^2)$)
 - 空間 $O(n)$ (再帰の深さ)

DFS

function $DFS(u)$

頂点 u を訪れた, と記録.

for each v (u に隣接している点):

if 頂点 v をまだ訪れていない:

$DFS(v)$.

for each $u \in V$:

if 頂点 u をまだ訪れていない:

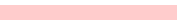
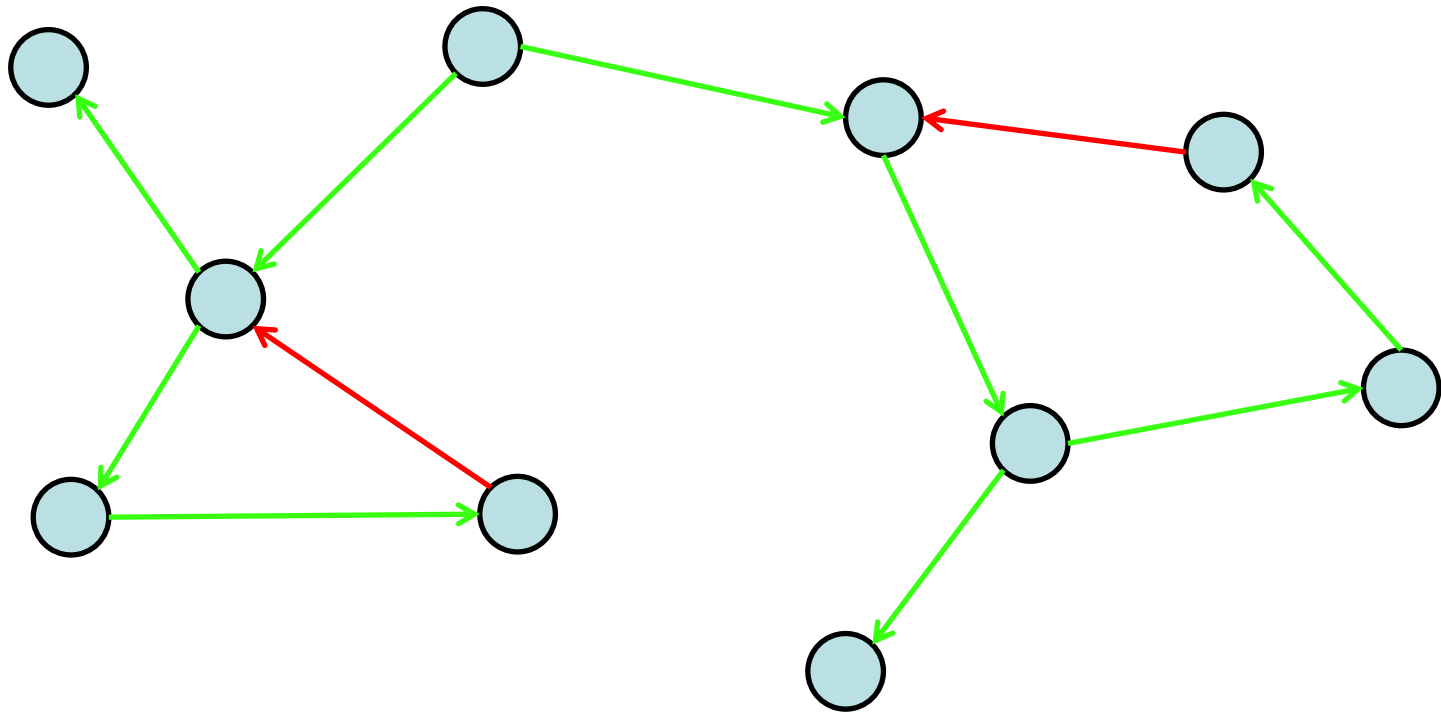
$DFS(u)$.

DFS-tree

- DFS を行うと, 通った辺からなる森ができる
 - 連結グラフに対しては木が得られる
 - 深さ優先探索木 (DFS-tree) あるいは 深さ優先探索森 (DFS-forest)
- 以下, 無向連結グラフで考える
- DFS で通らなかつた辺は?
 - 後退辺 (back edge)



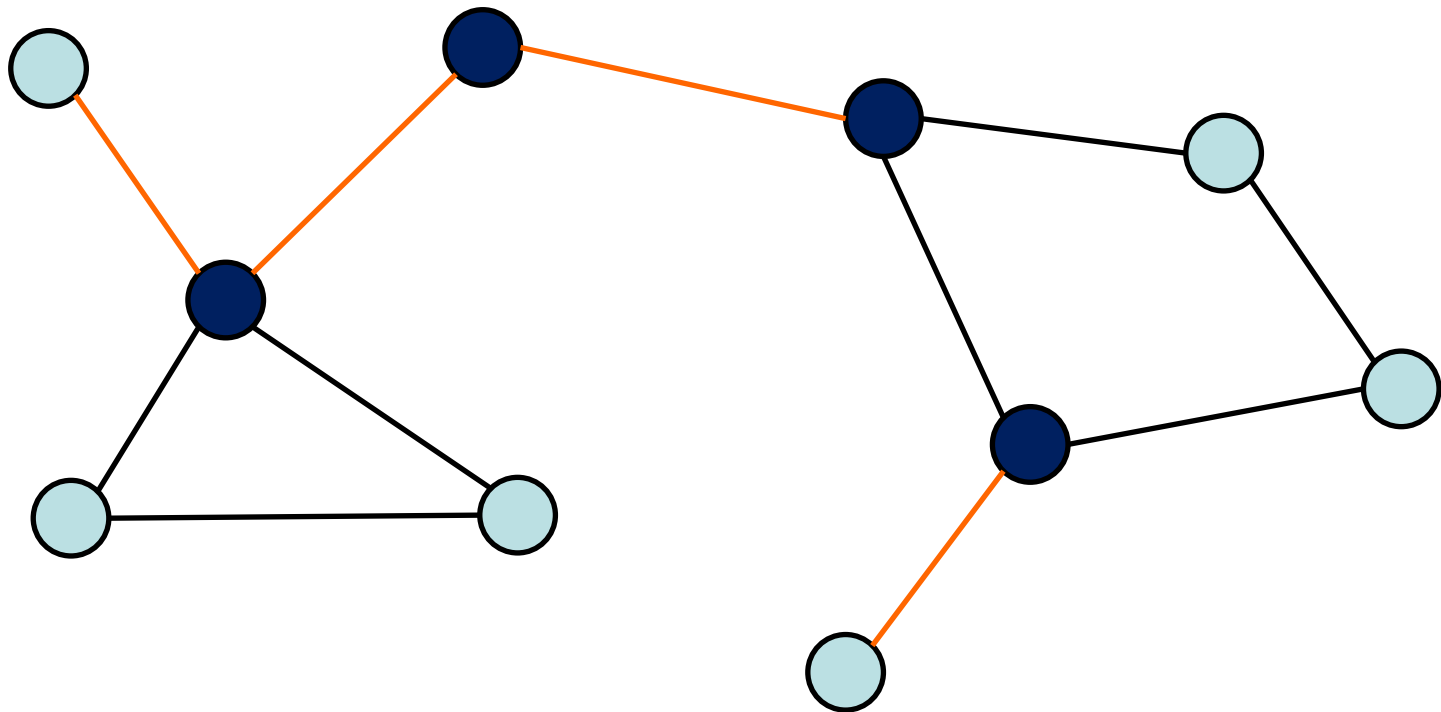
DFS-tree



橋, 関節点

- 橋 (bridge) : 取り除くと連結成分が増える辺
- 関節点 (articulation point) : 取り除くと連結成分が増える頂点
 - 接続している辺も同時に取り除く
- 問題 : 与えられた無向連結グラフの橋, 関節点をすべて求めよ.

橋, 關節点



橋, 関節点

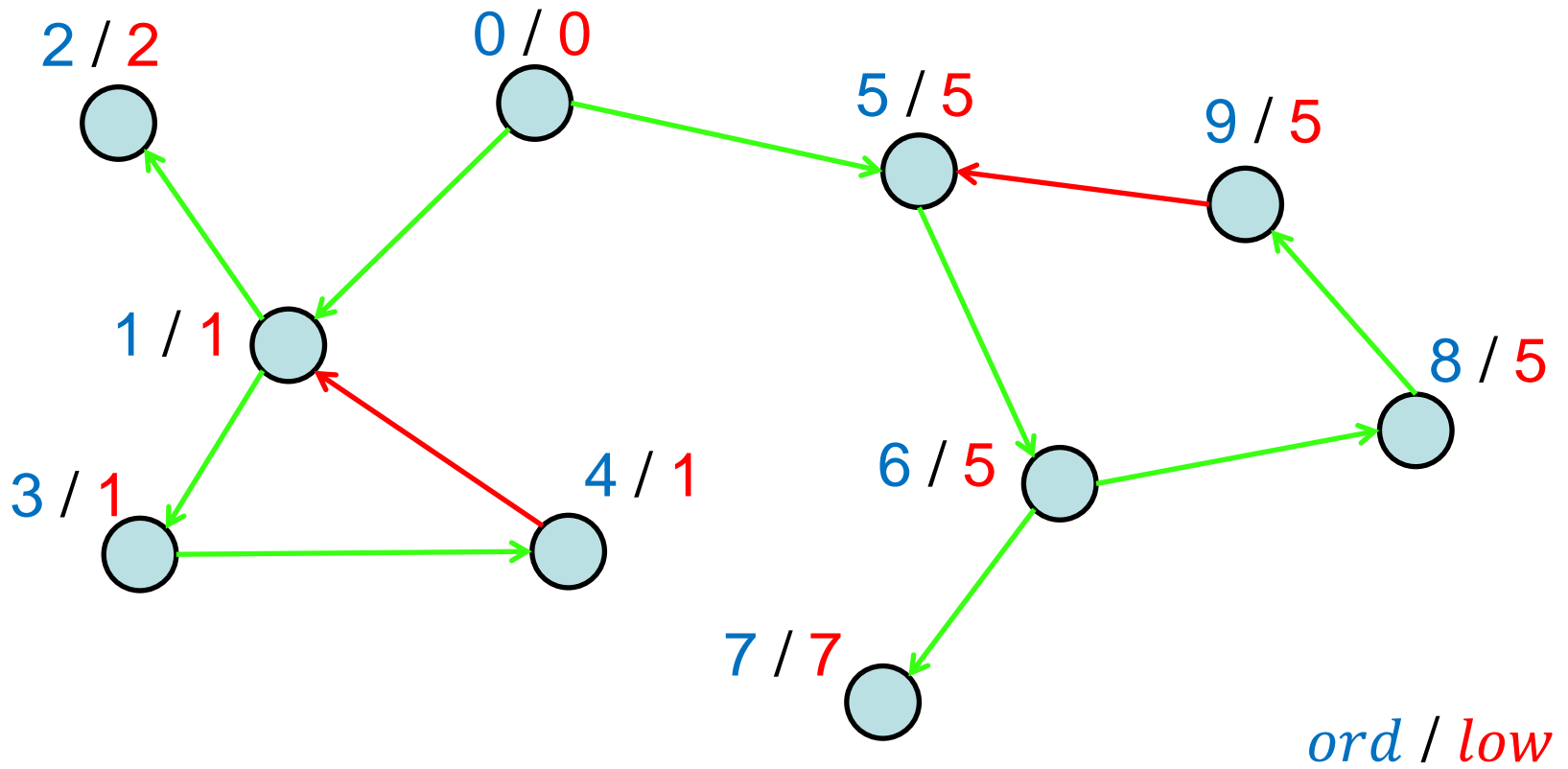
- 橋

- 単純な方法 : 各辺に対し, それを取り除いたときに連結かどうかを DFS などにより調べる
- $O(m(n + m))$
- 少し工夫 : DFS を行ったとき, 後退辺は橋になりえないので, DFS 木の辺 ($n - 1$ 本) のみを調べる
- $O(n(n + m))$

Lowlink

- 頂点を訪れた順に番号 $ord[u]$ をつける
 - 根から葉の向きへ進むと ord は大きくなる
- 各頂点の “Lowlink” $low[u]$ を求める
 - $low[u]$ は, u から以下の方法で辿り着ける頂点での ord の最小値
 1. DFS 木の辺を根から葉の向きへ進む (何度でも)
 2. 後退辺を葉から根の向きへ進む (1 回まで)

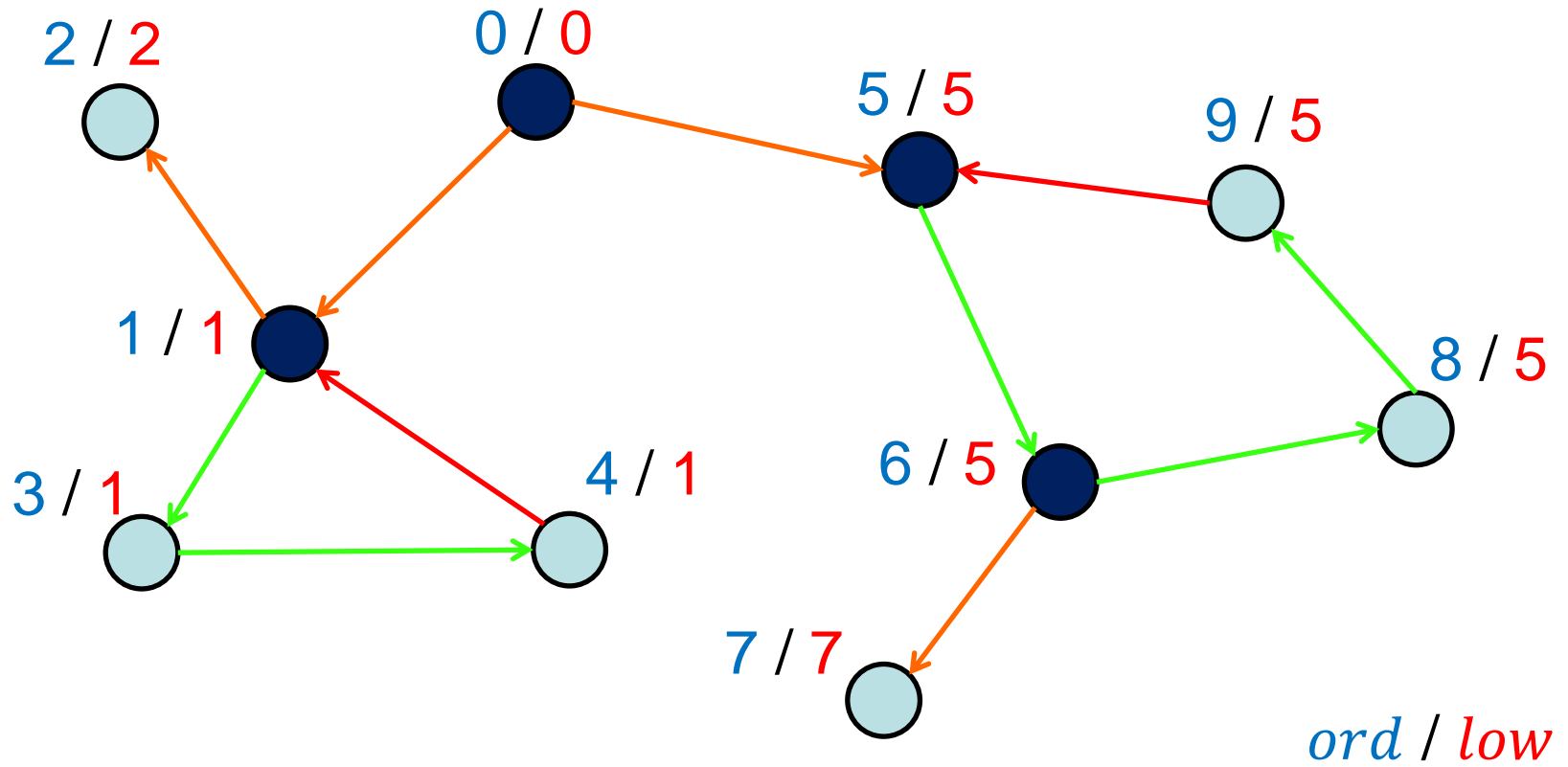
Lowlink



Lowlink

```
function DFS (u)
  ord[u] := k.
  k := k + 1.
  low[u] := ord[u].
  for each v (u に隣接している点):
    if 頂点 v をまだ訪れていない:
      DFS (v).
      low[u] := min{ low[u], low[v] }.
    else if 辺 vu を通っていない:
      /* このとき uv は後退辺 */
      low[u] := min{ low[u], ord[v] }.
```

橋, 關節点



橋, 関節点

- 橋

- DFS 木の辺 uv が橋 $\Leftrightarrow ord[u] < low[v]$

- $O(n + m)$

- 応用例 : 同じ長さの単語がたくさん与えられるので, 辞書順最小のしりとりを求めよ (JOI 2011 選考会 Shiritori).

橋, 関節点

- 関節点

- 単純な方法 : 各頂点に対し, それを取り除いたときに連結かどうかを DFS などにより調べる
- $O(n(n + m))$

- DFS 木の根が関節点 \Leftrightarrow (次数) > 1
- DFS 木の根以外の頂点 u が関節点
 $\Leftrightarrow u$ のすべての子 v に対し $ord[u] \leq low[v]$
- $O(n + m)$

橋, 関節点

- 発展 : 無向連結グラフが与えられる. 以下のようなクエリに答えよ (Croatia OI 2007).
 - 頂点 u, v と辺 e に対し, e を取り除いたとき u から v へ辿り着けるか?
 - 頂点 u, v, w に対し, w を取り除いたとき u から v へ辿り着けるか?

幅優先探索 (BFS)

BFS

- 幅優先探索 (breadth-first search)
- 距離が近いところから順番に見る
 - 最短路が求まる
- キュー (待ち行列)
- 計算量
 - 時間 $O(n + m)$ (隣接行列で持つ場合 $O(n^2)$)
 - 空間 $O(n)$

BFS

for each $u \in V$:

$dist[u] := \infty$.

$dist[start] := 0$.

Q の末尾に $start$ を追加.

while Q が空でない:

$u := Q$ の先頭から取り除く.

for each v (u に隣接している点):

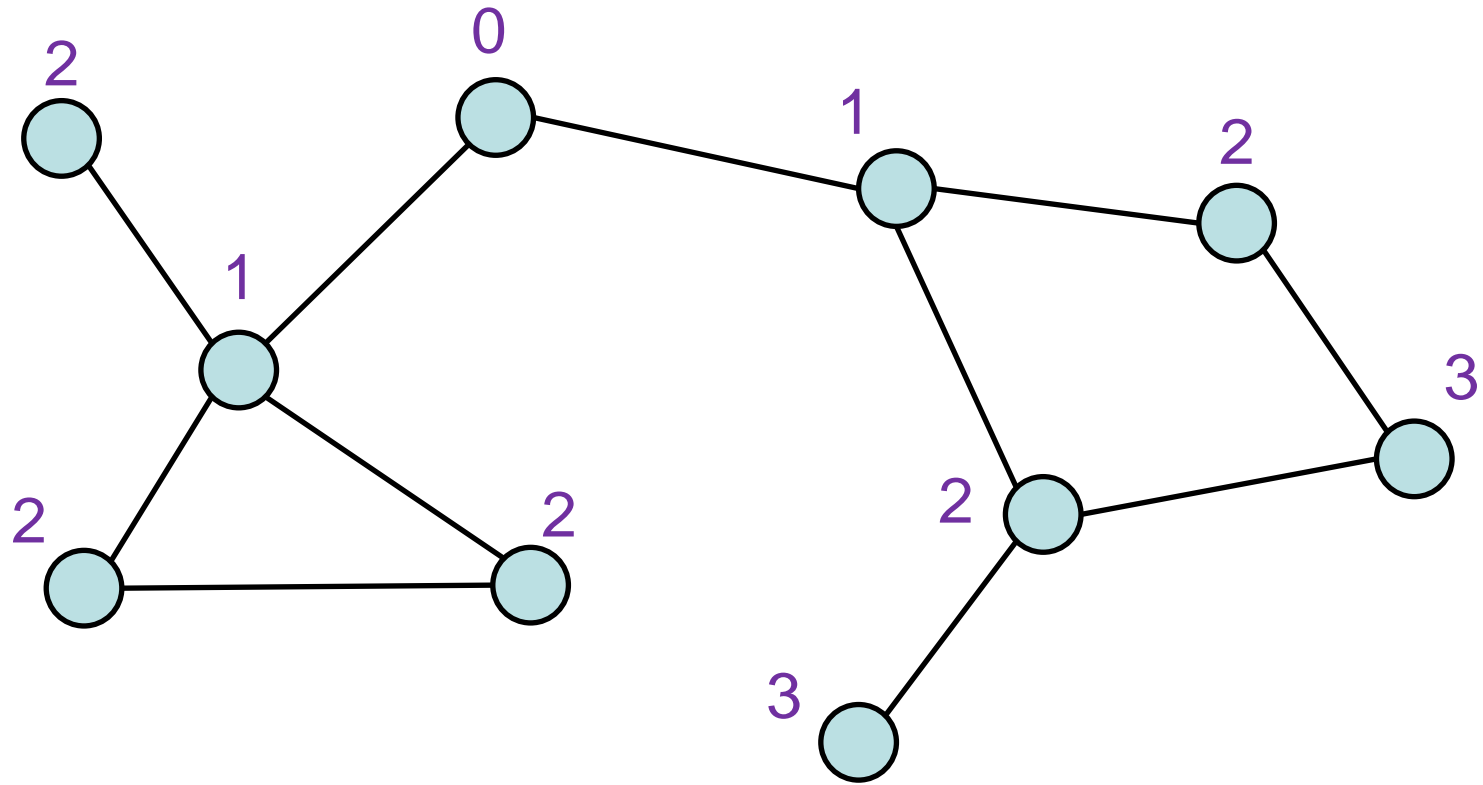
if $dist[v] > dist[u] + 1$:

$dist[v] := dist[u] + 1$.

Q の末尾に v を追加.



BFS



BFS-tree

- BFS を行くと, 通った辺からなる木ができる
 - 幅優先探索木 (BFS-tree)
 - *start* から各頂点への最短路は BFS-tree 上の根からのパス
- BFS で通らなかった辺は？
 - *dist* の差が 0 または 1



Dijkstra 法, 0-1-BFS

- BFS では, 次の頂点を選ぶ機構としてキューを用いた
 - 最初に追加したものが最初に取り出される
- これを変えると重みつきグラフの最短路を求めることができる



Dijkstra 法

- Dijkstra 法
 - 辺の重みは非負に限る
 - $dist[v] := dist[u] + c(uv)$ のように距離を更新
 - 次の頂点を選ぶときに、まだ選ばれていない頂点のうち $dist[u]$ が最小であるものを選ぶ
 - 単純な方法 : $O(n^2)$ (密グラフに対しては高速)
 - ヒープなどのデータ構造を用いる :
 $O((n + m) \log n)$
 - 実用的ではないが $O(n \log n + m)$ となるものもある

0-1-BFS

- 0-1-BFS
 - 辺の重みは 0 または 1 に限る
 - 次の頂点を選ぶときに, まだ選ばれていない頂点のうち $dist[u]$ が最小であるものを選ぶ
 - デック (両端キュー) を用いる :
 $O(n + m)$
 - 先頭および末尾に対する追加・削除ができる

0-1-BFS

Q の末尾に $start$ を追加.

while Q が空でない:

$u := Q$ の先頭から取り除く.

for each v (u に隣接している点):

if $dist[v] > dist[u] + c(uv)$:

$dist[v] := dist[u] + c(uv)$.

if $c(uv) = 0$:

Q の先頭に v を追加.

else if $c(uv) = 1$:

Q の末尾に v を追加.

辞書順幅優先探索 (LexBFS)



LexBFS, Cograph

- LexBFS
- Cograph
- P_4 -free graph
- Read-once function

- グラフはすべて無向単純グラフ



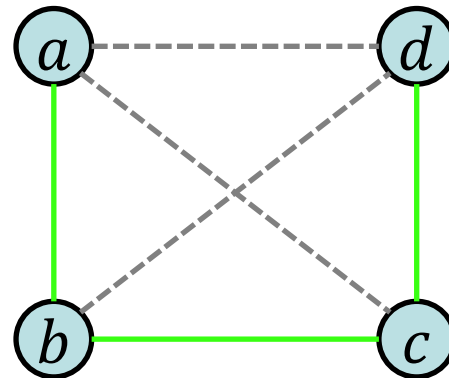
P_4 -free graph

- 問題 1 : グラフ $G = (V, E)$ が与えられるので, 以下の条件をみたす 4 頂点 a, b, c, d が存在するか否か判定せよ.

(PKU Judge Online 3236)

- $ab, bc, cd \in E$
- $ac, ad, bd \notin E$

P_4



Read-once function

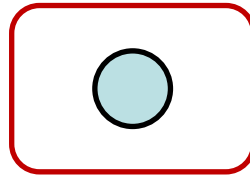
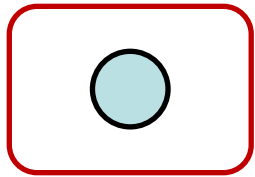
- 問題 2 : 単項式の和の形の式が与えられるので, 現れた文字を 1 回ずつと加算・乗算のみを使って等価な式に変形できるか否か判定せよ.
 - 可能 : $acf + bc + ce h + dg$
 - $(af + b + eh)c + dg$
 - 可能 : $abc + aef + ag + bcd + def + dg$
 - $(a + d)(bc + ef + g)$
 - 不可能 : $abc + abe + ade + bcf + bef + def$

Cograph

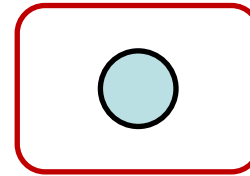
- cograph (complement reducible graph)
 1. 1 頂点からなるグラフは cograph
 2. G_1, G_2 が cograph $\Rightarrow G_1 \cup G_2$ も cograph
 3. G が cograph $\Rightarrow \bar{G}$ も cograph
 4. 以上で定まるものが cograph 全体
 - $G_1 \cup G_2$: G_1, G_2 の union (結ばずに並べる)
 - \bar{G} : G の complement (辺の有無を逆転)
 - complement ... 補グラフ



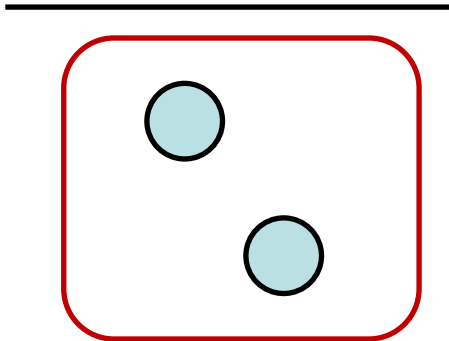
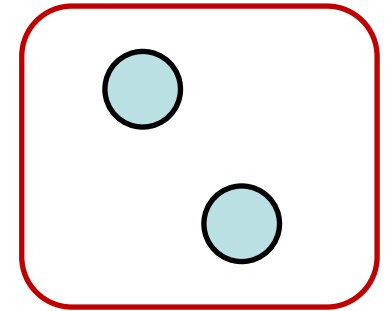
Cograph



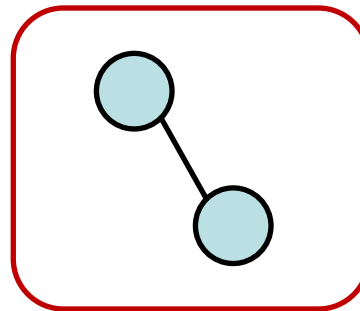
\cup



$=$

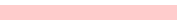
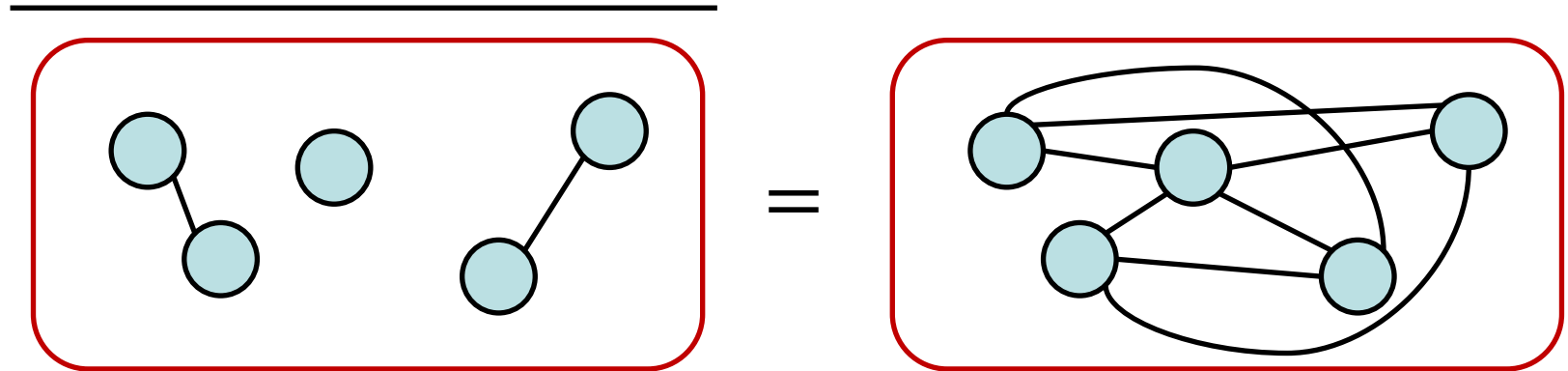
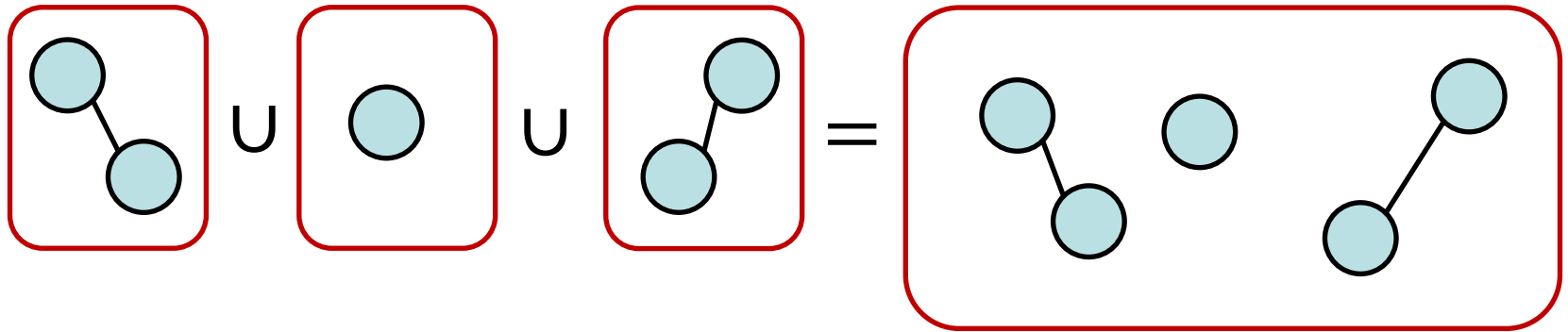


$=$





Cograph



無向グラフの性質

- $G = (V, E)$ に対し, G または \bar{G} の少なくとも一方は連結
 - (証明) G が連結でないとする. $u, v \in V$ に対し,
 - $uv \in E$ のとき : u, v を含まない G の連結成分が存在. そこから頂点 w を選べば, $uw, wv \notin E$ より \bar{G} のパス uwv が存在
 - $uv \notin E$ のとき : \bar{G} のパス uv が存在
- 2 頂点以上の cograph G に対し, G または \bar{G} のちょうど一方が連結

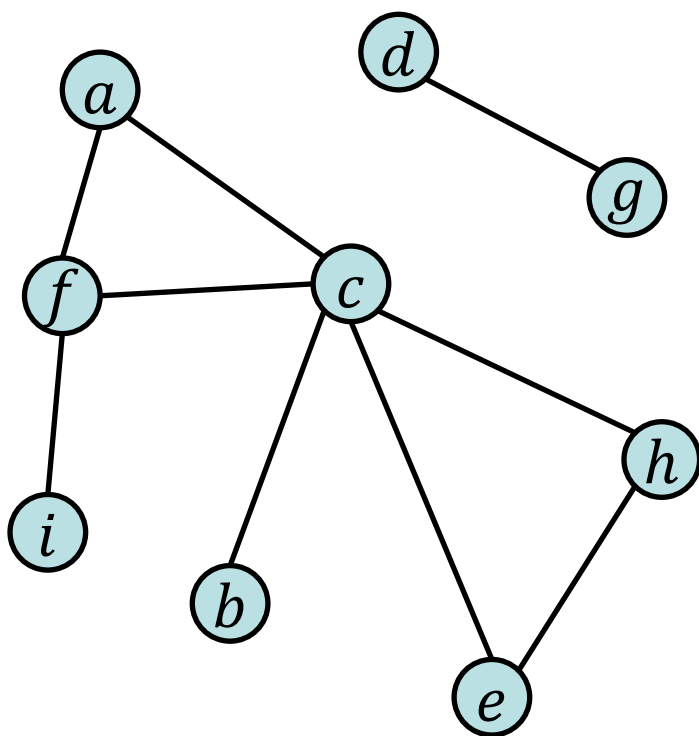
Cograph 判定

- 問題：グラフが与えられたとき, cograph であるか否かを判定せよ.
- 単純な方法： $O(nm)$
 - 1 頂点ならば cograph
 - G も \bar{G} も連結ならば G は cograph でない
 - G または \bar{G} のうち連結でない方を連結成分に分解し, それぞれが cograph であるかどうかを再帰的に調べる

LexBFS

- 辞書順幅優先探索
(LexBFS, lexicographic breadth-first search)
- BFS の一種
 - 最初の方に選んだ頂点に隣接する頂点を優先
- 計算量
 - 時間 $O(n + m)$ (隣接行列で持つ場合 $O(n^2)$)
 - いろいろ手抜くと $O(n^2 \log n)$
 - 空間 $O(n)$

LexBFS



- グラフ探索における「まだ訪れていない頂点」を「リストのリスト」で管理する
- 最初に頂点に適切な順序を与える

LexBFS

- slice の構造

$$\left[a \left[c \left[f \right] \right] \left[b \left[e \left[h \right] \right] \right] \left[i \right] \left[d \left[g \right] \right] \right]$$

- $S(u)$ を分解

$$S^A(a) = cf$$
$$S_1(a) = beh, S_2(a) = i, S_3(a) = dg$$

– u

– $S^A(u)$: $S(u)$ 中の u に隣接している頂点

– $S_1(u), S_2(u), \dots$: LexBFS で $S^A(u)$ の点まで取り出したときのリストの中身

- $S_i(u)$ は一般には slice とは限らない

Cograph 判定

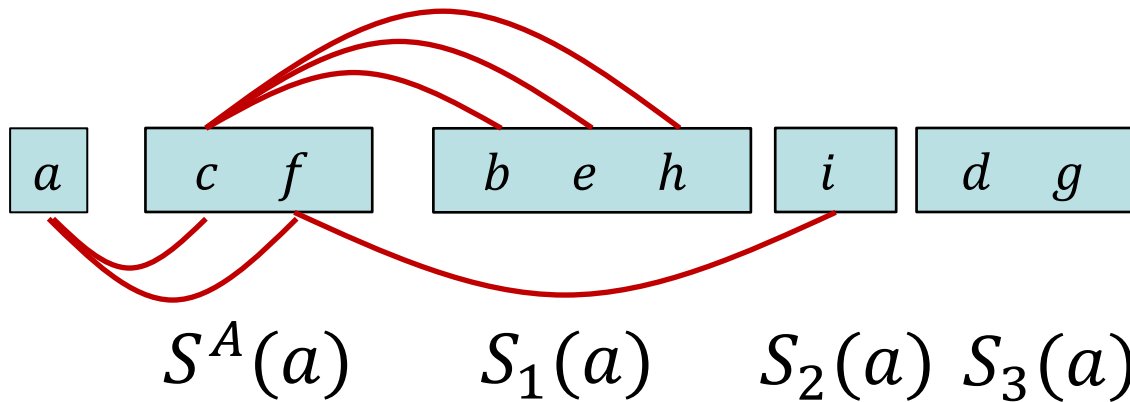
- G が cograph であるかを $O(n + m)$ で判定
 1. G に LexBFS を行う
 2. 1. で頂点を取り出した順序を最初の順序として, \bar{G} に LexBFS を行う
 - G に対する LexBFS と同様に行うが, リストを隣接点とそれ以外に分けるときの, 隣接点を後ろに置く
 3. 1. と 2. で得られる slice が「適切な性質」を持つかどうか調べる

Cograph 判定

- cograph の slice が持つ性質 (証明略)
 - $i < j$ に対し, $S_i(u)$ の頂点と $S_j(u)$ の頂点を結ぶ辺は存在しない
 - $v \in S^A(u)$ と $i < j$ に対し, v が $S_j(u)$ の頂点と隣接しているならば, v は $S_i(u)$ の頂点とも隣接している
 - 各 $S_i(u)$ 内では, $S^A(u)$ のどの頂点と隣接しているかは同じであることに注意
- 計算量 $O(n + m)$

Cograph 判定

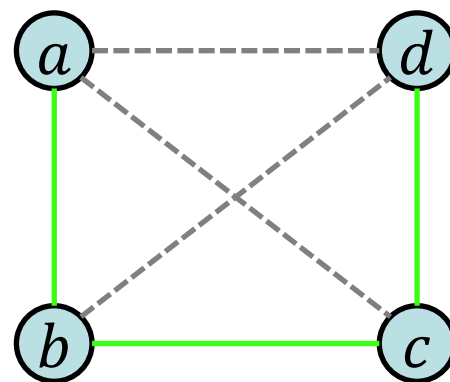
$[a [c [f]] [b [e [h]]] [i] [d [g]]]$



- $S^A(a)$ の点は, $S_1(a)$ に対しては c , $S_2(a)$ に対しては f で隣接
→ cograph でない

Cograph と P_4

- 元のグラフも complement も連結なグラフ
 - 1 頂点からなるグラフ
 - P_4 (complement も P_4)



- 実は, cograph であることは, P_4 を含まないことと同値
 - 「 P_4 を含む」とは, 単に 4 頂点からなるパスが存在するだけでなく, 上図の ac, ad, bd に対応する箇所には辺がないことも要する

Cograph と P_4

- cograph は P_4 を含まない
 - (証明)
 1. 1 頂点からなるグラフは P_4 を含まない
 2. G_1, G_2 が P_4 を含まない $\Rightarrow G_1 \cup G_2$ も P_4 を含まない
 3. G が P_4 を含まない $\Rightarrow \bar{G}$ も P_4 を含まない
 - 以上より帰納的に示される

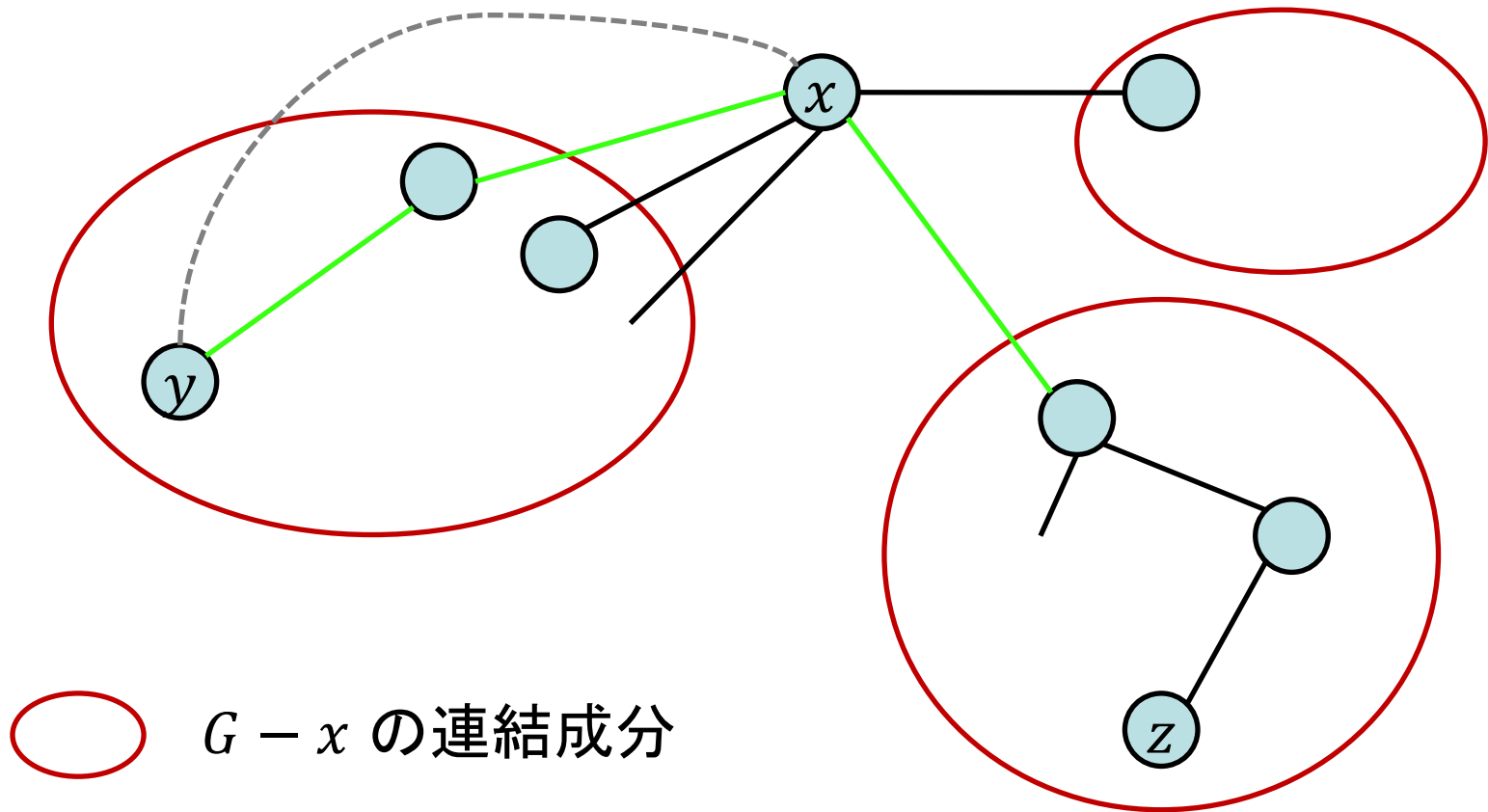
Cograph と P_4

- P_4 を含まないグラフは cograph である
 - (証明)
 - P_4 を含まないかつ cograph でないグラフが存在すると仮定
 - G : そのうち頂点数が最小のものの中の 1 つ
 - x : G の適当な頂点
 - $G - x$ (G から x を取り除いたグラフ) は P_4 を含まないので cograph
 - $G - x$ が連結のときは $\overline{G - x}$ が連結でないので, G の代わりに \bar{G} を考えることで, $G - x$ は連結でないとしてよい

Cograph と P_4

- P_4 を含まないグラフは cograph である
 - (証明つづき)
 - G も \bar{G} も連結
 - 連結でないとすると cograph たちに分かれるので矛盾
 - y : G の頂点で x と隣接しないものがとれる
 - z : $G - x$ で y と異なる連結成分に属する頂点
 - G は連結なので y から z へのパスが存在するが,
 - x を経由しなければならない
 - y から x へは他の頂点を経由しなければならない
 - y から z への辺が最小のパスを考えると、「ショートカット」がないのでこのパスは P_4 を含み矛盾

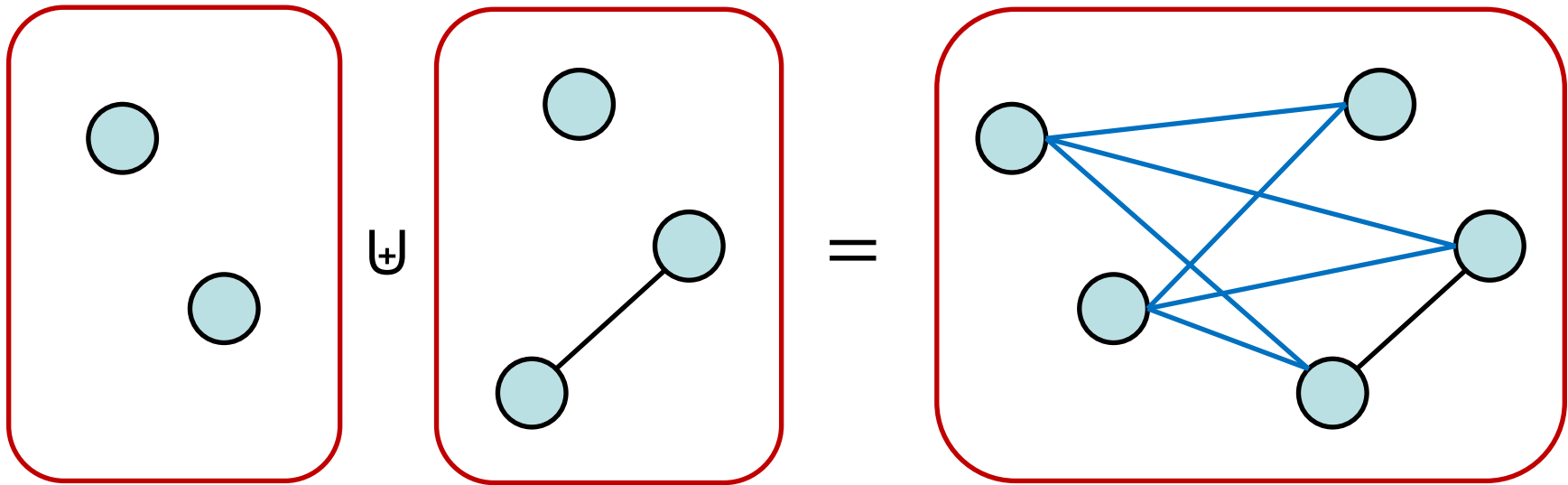
Cograph と P_4




Cograph と Cotree

- cograph の定義は, 次の定義と同値
 1. 1 頂点からなるグラフは cograph
 2. G_1, G_2 が cograph $\Rightarrow G_1 \cup G_2$ も cograph
 3. G_1, G_2 が cograph $\Rightarrow G_1 \uplus G_2$ も cograph
 4. 以上で定まるものが cograph 全体
- $G_1 \uplus G_2$: G_1, G_2 の join (G_1 の頂点と G_2 の頂点を結ぶ辺をすべて加えて並べる)
 - join は complement, union, complement ができる



Cograph & Cotree





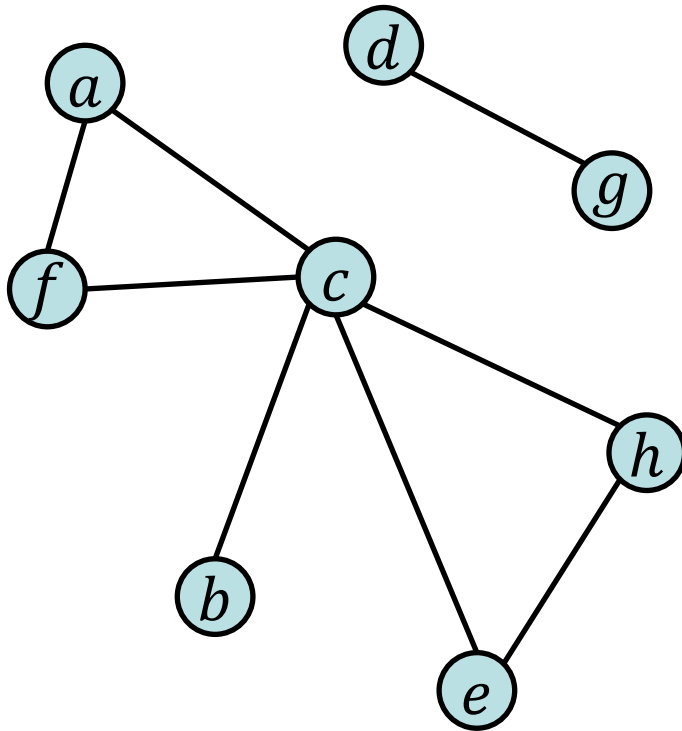
Cograph と Cotree

- cotree

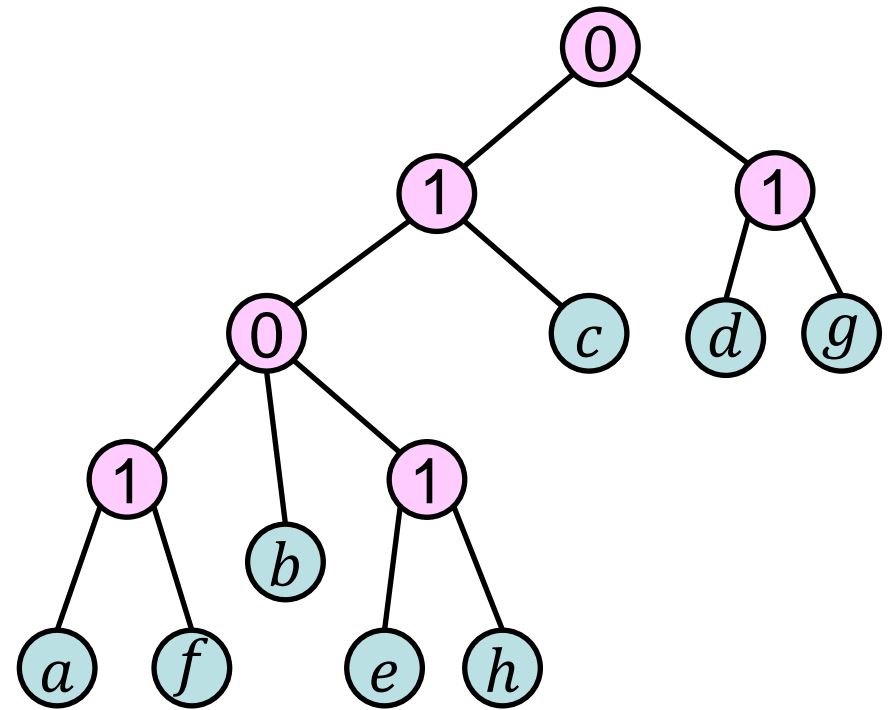
- cograph の頂点を葉とし, その他の頂点には 0 か 1 の印がついている
 - cograph の構成に対応
 - 0 : union
 - 1 : join
 - complement をとると 0 / 1 がすべて入れ替わる
 - cograph の 2 頂点に対して, 最も根から遠い共通の祖先が 0 ならば辺がなく, 1 ならば辺がある
- 
- 

Cograph と Cotree

cograph



対応する cotree



Cograph と Cotree

- LexBFS で cograph 判定をするとき, 以下も計算量 $O(n + m)$ で行える (詳細略)
 - cograph である場合, cotree (後述) を構成
 - cograph でない場合, 含まれる P_4 を検出



Cotree と Read-once function

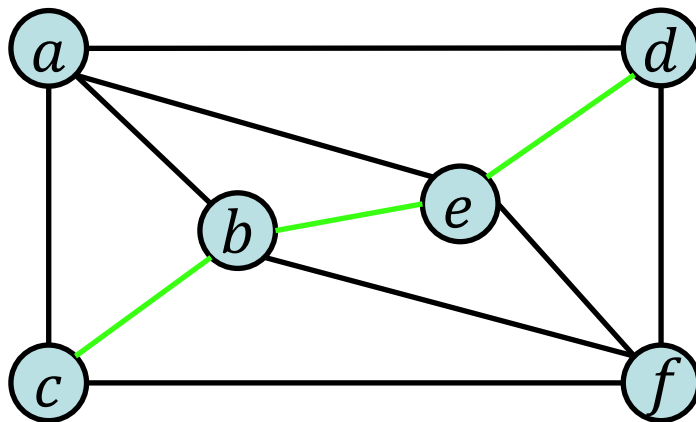
- read-once かどうかの判定
 - $acf + bc + ce + eh + dg$
 - $abc + abe + ade + bcf + bef + def$
- 1. 各変数を頂点とするグラフを考える
- 2. 掛け合わさっている変数どうしを辺で結ぶ
 - $ac, af, cf, bc, ce, ch, eh, dg$
 - ab, ac, bc, \dots
 - 重複は無視する





Cotree と Read-once function

- read-once かどうかの判定 (つづき)
 3. cograph でないならば失敗



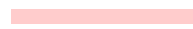
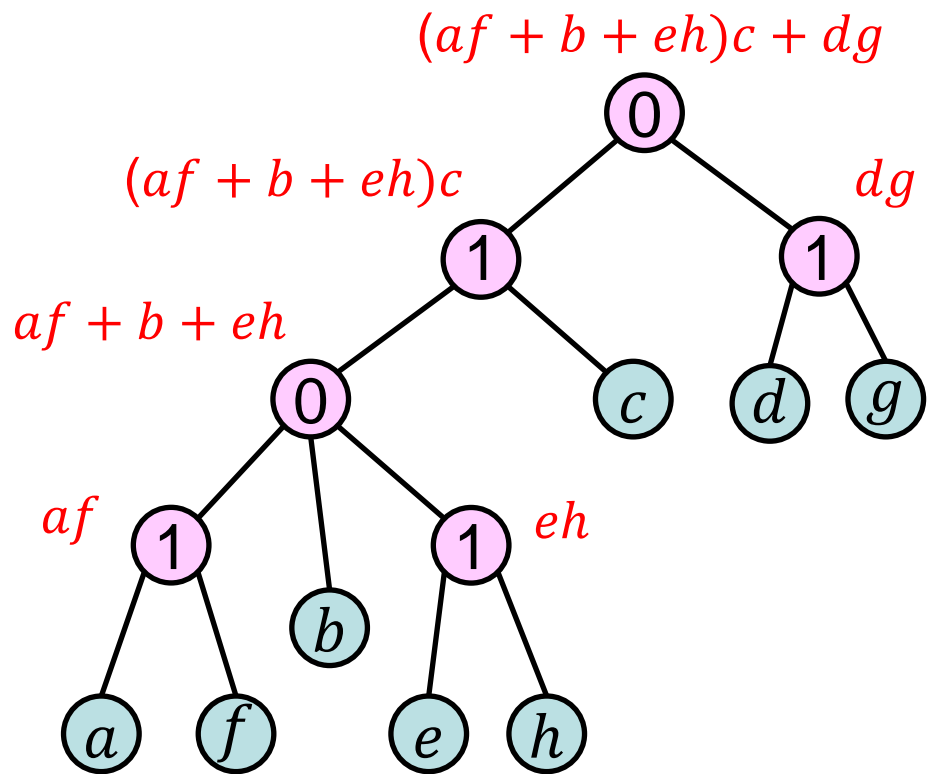
4. cograph ならば, cotree から多項式を復元して元の多項式になれば read-once



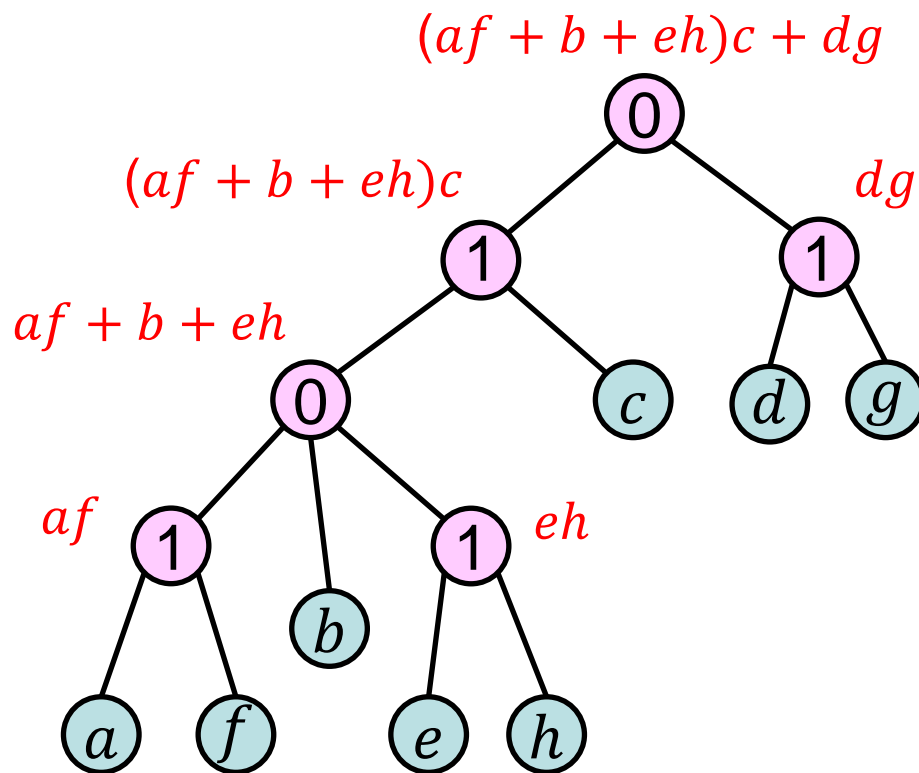


Cotree と Read-once function

- cotree の 0 を +, 1 を \times と考えて多項式を復元



Cograph と Read-once function



- 各項は cograph の極大クリークに対応している
 - クリーク：頂点の集合で、どの2点も辺で結ばれているもの

Cograph の性質

- cotree を構成すると,
 - 最大クリークが求められる
 - 最大安定集合が求められる
 - 安定集合 : 頂点の集合で, どの 2 点も辺で結ばれていないもの
 - 彩色数が求められる
 - 彩色数 : 辺で結ばれた頂点を同じ色で塗らないとき, 何色で塗り分けられるか
 - cograph においては彩色数は最大クリークの大きさに等しい
 - 一般には, 彩色数は最大クリークの大きさ以上

Cograph の性質

- cograph から頂点をいくつか取り除いても cograph
- 任意の極大クリークと任意の極大安定集合はちょうど 1 頂点を共有
などなど
- 文献入手法
 - cograph, LexBFS など検索