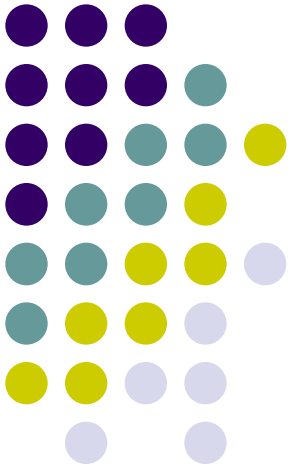
情報システム基盤学 基礎1
アルゴリズムとデータ構造

Elements of Information Systems Fundamentals1



第4回：動的計画法とグラフの最短経路問題

目次

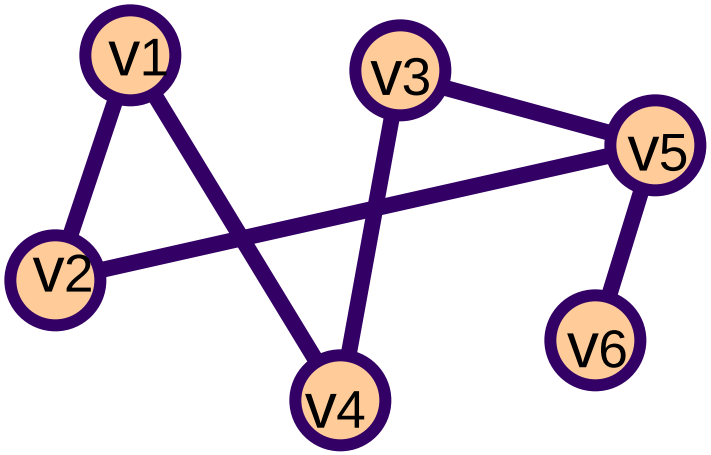
- グラフとは
- 最短経路問題
  - Dijkstraのアルゴリズム
  - Floyd-Warshallのアルゴリズム（動的計画法）
- 付録：グラフの探索（深さ優先，幅優先）

グラフの定義（直観）

グラフ

頂点(vertex, node)と辺(edge)からなる構造

○ : 頂点（点）　／ : 辺



グラフの定義（形式的）

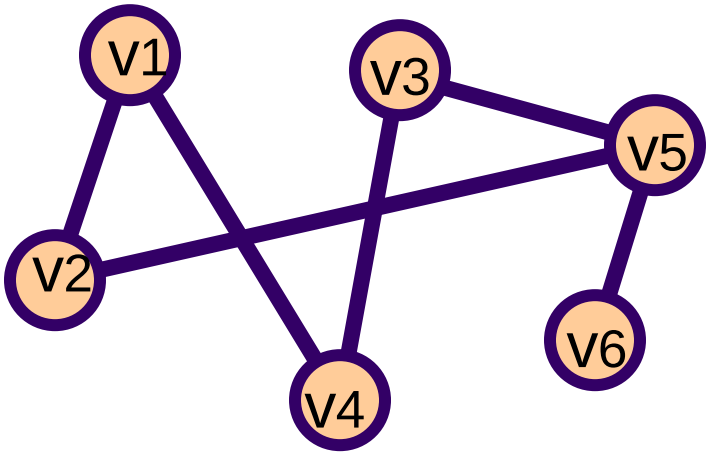
（集合の記述を使ってグラフを定義します）

グラフ

頂点集合 V と 辺集合 E の対　$G=(V, E)$ をグラフと呼ぶ

ここで，$E = \{ (x, y) \mid x \in V, y \in V \}$

頂点数　$|V| = n$,　辺の数　$|E| = m$



$V = \{ v1, v2, v3, v4, v5, v6 \}$

（V : 頂点集合）

$E = \{ (v1, v2), (v1, v4), (v2, v5), (v3, v4), (v3, v5), (v5, v6) \}$

（E : 辺集合）

無向グラフと有向グラフ

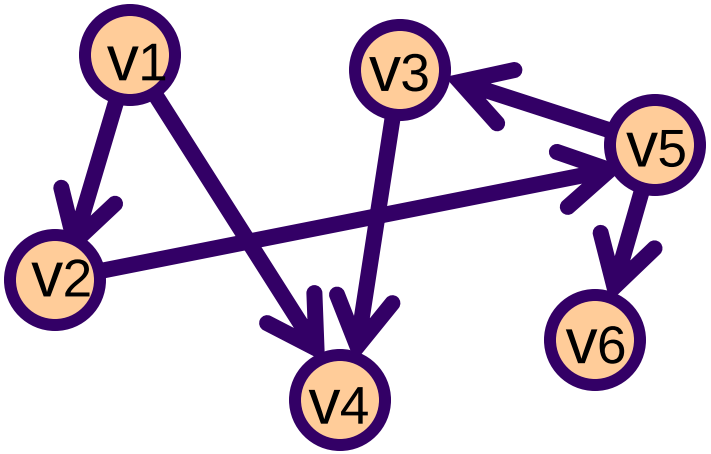
無向グラフ: 辺に向きがない

有向グラフ: 辺に向きがある

（辺集合が頂点の順序対からなるようなグラフ）

(*v1*, *v2*)と(*v2*, *v1*)は異なる

（始点，終点）



V = { v1, *v2*, *v3*, *v4*, *v5*,*v6* }

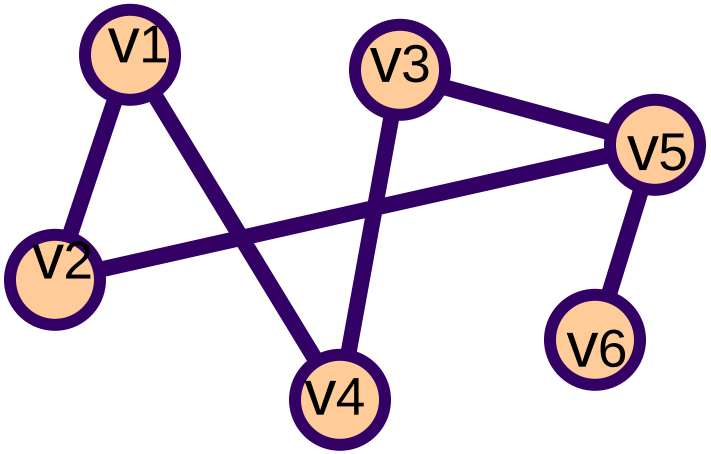
（*V* : 頂点集合）

E = { (*v1*, *v2*), (*v1*, *v4*), (*v2*, *v5*),
(*v3*, *v4*), (*v3*, *v5*), (*v5*, *v6*) }

（*E* : 辺集合）

グラフの表現方法：隣接行列

- 利点：枝の有無の判定が高速
  - 2点v_i, v_j間の枝の有無⇒A_{ij}より$O(1)$でわかる
- 欠点：
  - メモリ使用量 $O(n^2)$
  - 頂点vと隣接する頂点集合の取り出し$O(n)$



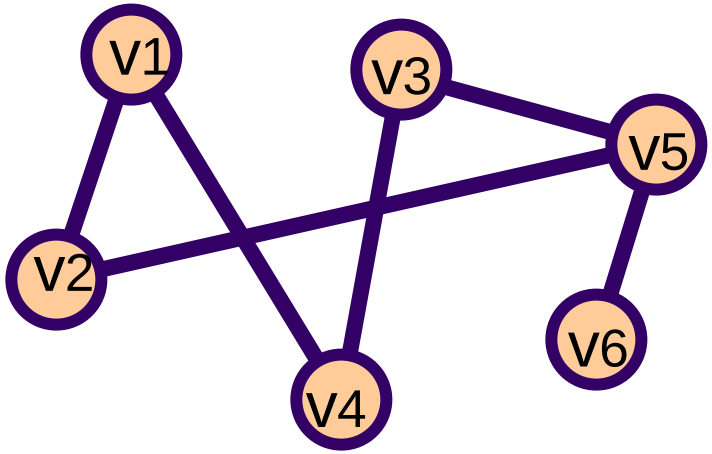
$$A = \begin{array}{c} \\ v1 \\ v2 \\ v3 \\ v4 \\ v5 \\ v6 \end{array} \begin{array}{c} \begin{array}{cccccc} v1 & v2 & v3 & v4 & v5 & v6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

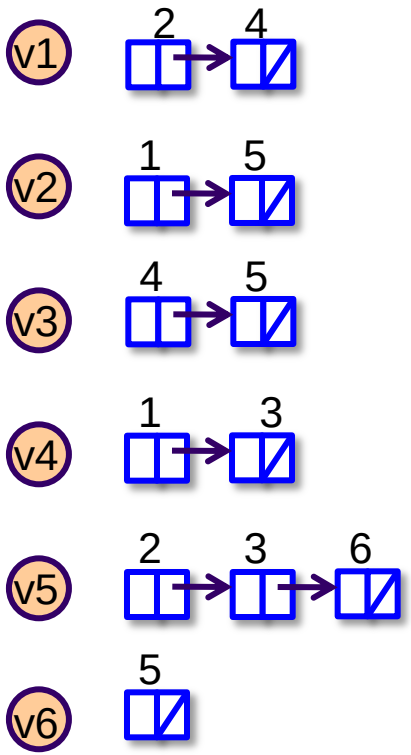
グラフの表現方法：隣接リスト

各頂点と隣接する頂点集合をリストで表現

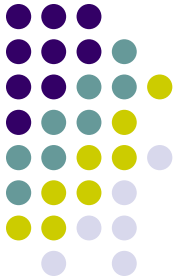
- 利点：
  - メモリ使用量 $O(m)$
  - 頂点vと隣接する頂点集合の取り出し　O(degree(v))
- 欠点：
  - 頂点v_i, v_j間の枝の有無⇒v_iまたはv_jに対するリスト走査

vから出る枝数
・次数





目次



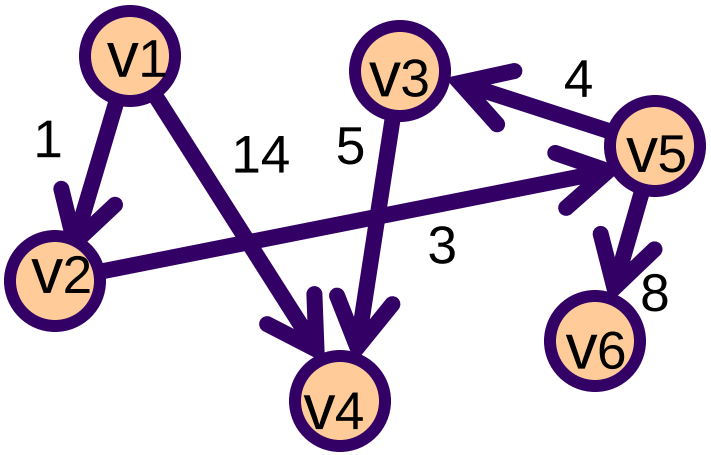
- グラフとは
- 最短経路問題
  - Dijkstraのアルゴリズム
  - Floyd-Warshallのアルゴリズム（動的計画法）
- おまけ：グラフの探索（深さ優先，幅優先）

重み付き有向グラフ

各辺に数値（重み）が付与された有向グラフ

（重み関数 $w: E \rightarrow R$ が与えられている有向グラフ）

（R は実数集合）



$V = \{ v1, v2, v3, v4, v5, v6 \}$

（V : 頂点集合）

$E = \{ (v1, v2), (v1, v4), (v2, v5), (v3, v4), (v3, v5), (v5, v6) \}$

（E : 辺集合）

$w(v1, v2) = 1, w(v1, v4) = 14, \ldots$

（w : 重み関数）

最短経路問題

- 入力:辺に正の重みが付いたグラフG
- 出力
  - 2点v_i, v_j間の最短経路
    - 経路p: v_i, v_j間を結ぶパス
      - 長さ$l(p)$: 経路を構成する辺の重みの和 $\sum_{e \in p} w(e)$

最適化問題:実行可能解の中で目的関数を最小(最大)化する解を見つける問題

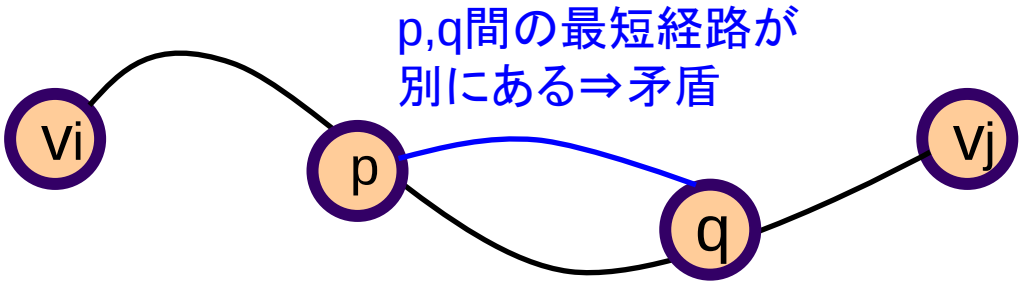
1. 単一始点最短路問題: 入力G,v
2. 全点間最短路問題: 入力G

最適化問題を解く手筋

- 与えられた問題への最適解と部分問題への最適解の関連性に注目

- 部分構造最適性:
  - 問題の最適解が部分問題の最適化を含む
- 最短経路問題では
  - p,q: v_i, v_j間の最短経路上に存在する2点

「v_i, v_j間の最短経路はp, q間の最短経路を含む」



最短経路問題

- 入力
  - 辺に正の重みが付いた重み付きグラフG
- 出力
  - 2点v_i, v_j間の最短経路
    - 経路P: v_i, v_j間を結ぶパス
      - 長さ$l(P)$: 経路を構成する辺の重みの和 $\sum_{e \in p} w(e)$

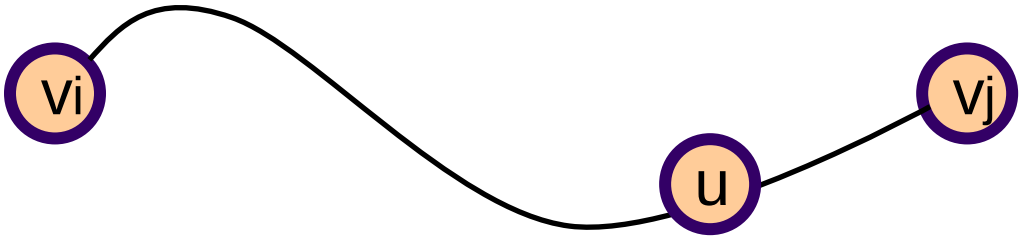
最適化問題:実行可能解の中で目的関数を最小(最大)化する解を見つける問題

1. 単一始点最短路問題: 入力G, 始点s
2. 全点間最短路問題: 入力G

Dijkastraアルゴリズムのアイデア

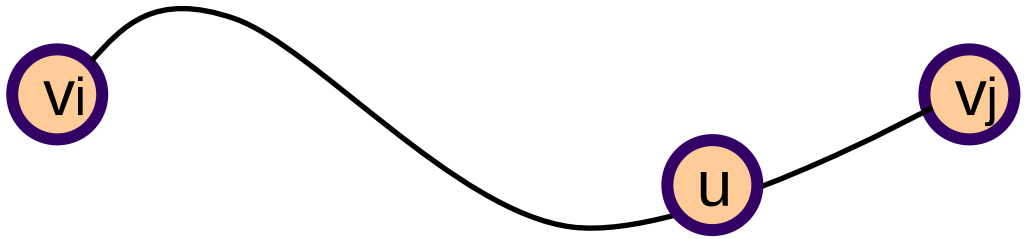
- なるべく大きな部分問題との関係に着目
- 最短経路問題では
  - u: v_i, v_j間の最短経路上でv_jの手前に存在する点

「 v_i, v_j間の最短経路はv_i, u間の最短経路を含む」



Dijkastraアルゴリズムのアイデア

「 v_i, v_j間の最短経路はv_i, u間の最短経路を含む」



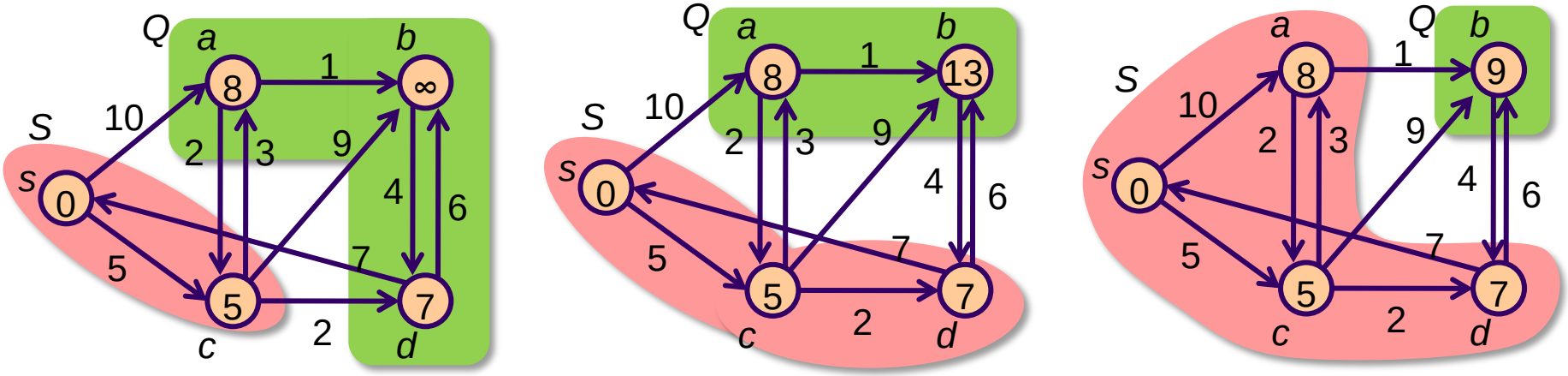
- 既に確定した最短経路を利用して、より離れた点までの最短経路を計算
- 始点(sとする)からの最短経路が短い順に1点ずつ最短経路を確定

Dijkstraのアルゴリズム: 概要

グラフ G の全頂点を次の2つのグループに分けておいて

- 始点 s からの最短経路が決まった頂点（S と書く）
- そうでない頂点（Q と書く）

始点 s からの最短経路が決まった頂点を1つずつ増やす
（*Note*：インクリメンタルな考え方になってます）



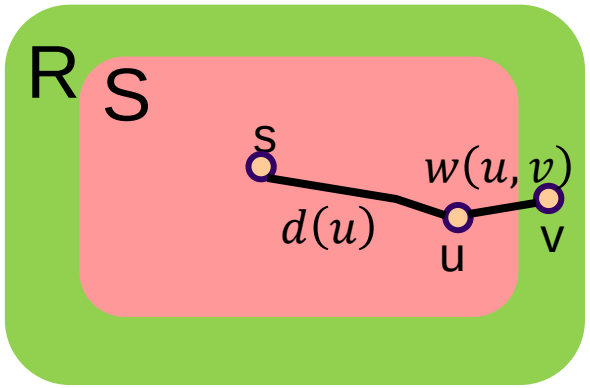
（※重みは全て正であると仮定してください）

Sの拡大方法

- R: Qの中でSと隣接する点集合
  - Sの点から枝を1本だけ経由して到達できる点集合
- Rから以下を満足する点v,uを決定。vをSに追加

$$\underset{v \in R, u \in S}{\operatorname{argmin}}\, d(u) + w(u, v)$$

  - $d(u)$: uまでの最短経路の長さ



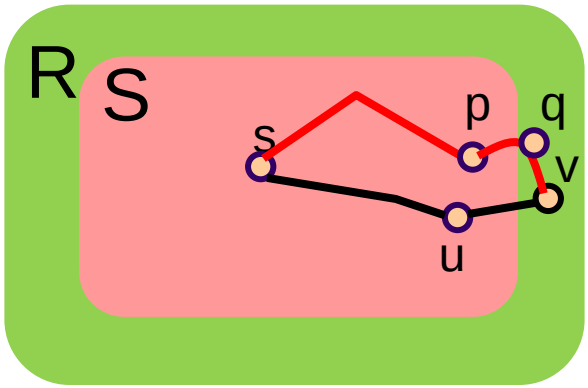
vまでの最短経路長= $d(u) + w(u, v)$

アルゴリズムが正しいことの証明

- $d(u)+w(u,v)$がvまでの最短経路でないとする

- vまでの最短経路が
  - Sにおいてノードpを最後に経由
    - $q \in R$: vまでの最短経路においてpの次のノード

$d(p)+w(p,q) \leq$ 最短経路長 $< d(u)+w(u,v)$⇒矛盾



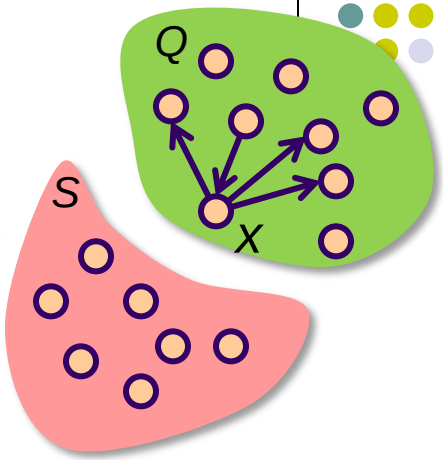
Dijkstraのアルゴリズム

s: 始点

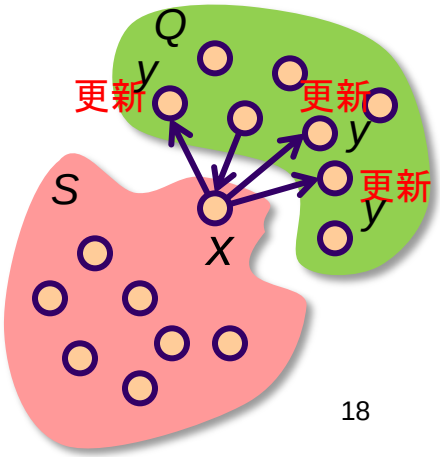
$d(v)$: Sの点のみを経由した時のs から v への経路長

$p(v)$: 現段階の v のpredecessor

- 経路長決定後にpredecessorを辿ると実際の経路が得られる



（ほとんどの辺を省略してあります）



アルゴリズム

$d(s) \leftarrow 0$, $S \leftarrow s$, $Q \leftarrow V - \{s\}$

for 各頂点 $v \in Q$ について, $d(v) \leftarrow \infty$ // 初期化

for 各 s の隣接点 u について $d(u) \leftarrow w(s,u)$ // 初期化

while $Q \neq \Phi$ do

　　Q の中から$d(x)$ が最小の頂点 x を取り出す

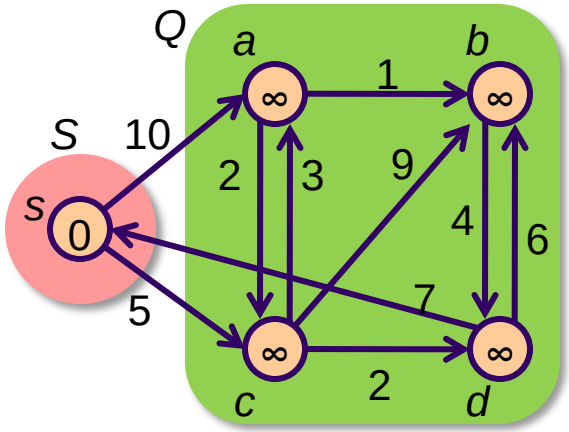
　　S に x を追加

　　for 各 x の隣接点 y について,

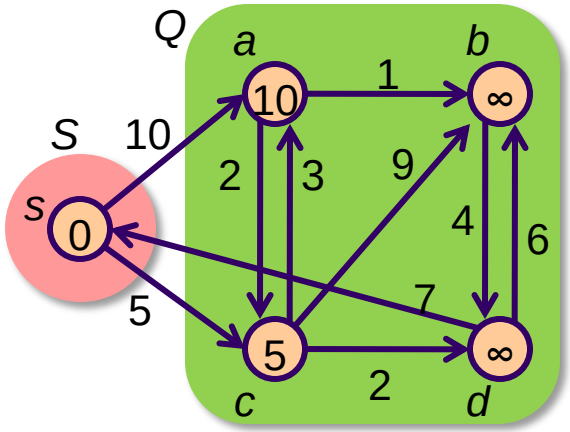
　　　if $d(y) > d(x) + w(x,y)$

　　　　then $d(y) \leftarrow d(x) + w(x,y)$, $p(y) \leftarrow x$

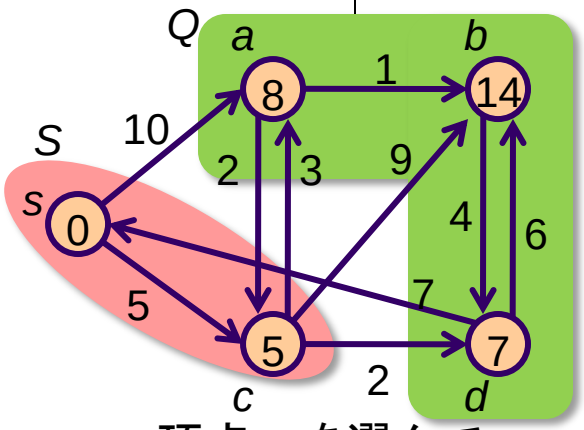
Dijkstraのアルゴリズム: 動作例



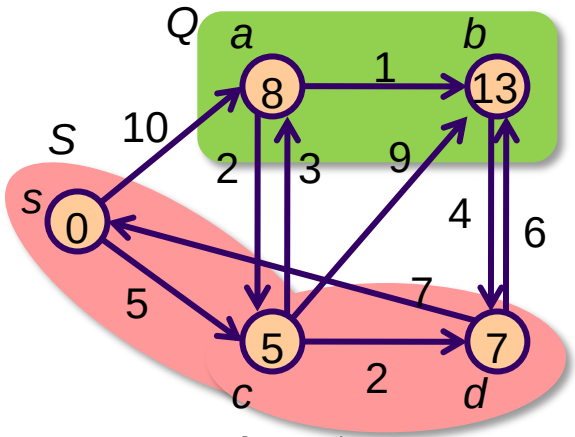
初期化その1



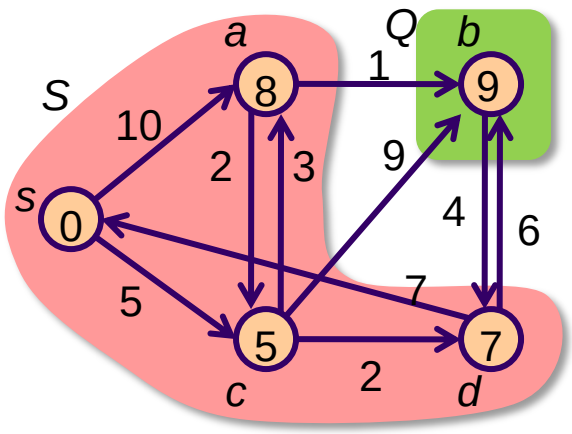
初期化その2



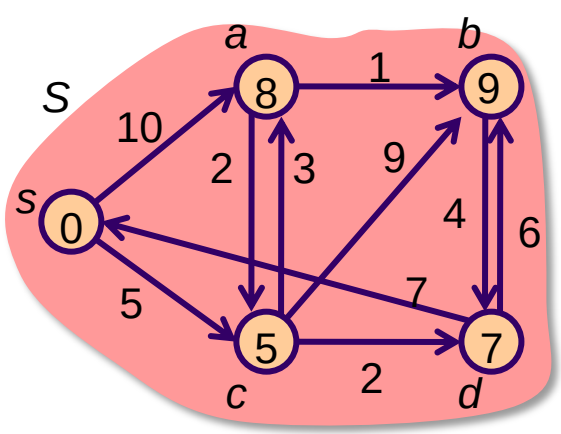
頂点 c を選んで,
c の周りを更新



頂点 d を選んで,
d の周りを更新



頂点 a を選んで,
a の周りを更新



頂点 b を選んで終了

計算量

アルゴリズム

$d(s) \leftarrow 0$, $S \leftarrow s$, $Q \leftarrow V - \{s\}$

for 各頂点 $v \in Q$ について, $d(v) \leftarrow \infty$ // 初期化

for 各 s の隣接点 u について $d(u) \leftarrow w(s,u)$ // 初期化

while $Q \neq \Phi$ do ← |V|回

　Q の中から$d(x)$ が最小の頂点 x を取り出す ← |Q|回

　S に x を追加

　for 各 x の隣接点 y について, ← 合計で|E|回

　　if $d(y) > d(x) + w(x,y)$

　　then $d(y) \leftarrow d(x) + w(x,y)$, $p(y) \leftarrow x$

計算時間: $O(|V|^2 + |E|) \rightarrow O((|V| + |E|)\log|V|)$ (ヒープ使用)

- 疎なグラフ$(|E| = O(|V|))$ならヒープを使用した方が高速
- 密なグラフなら単純な実装でも遜色なし

(バイナリ)ヒープ

部分順序付き木を配列に表現したもの
- 親子間に順序が定義される

木のノードは以下の2つによって定義される
- インデックス：木の中のノードの位置
- 値：実数

配列$A[\,]$

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

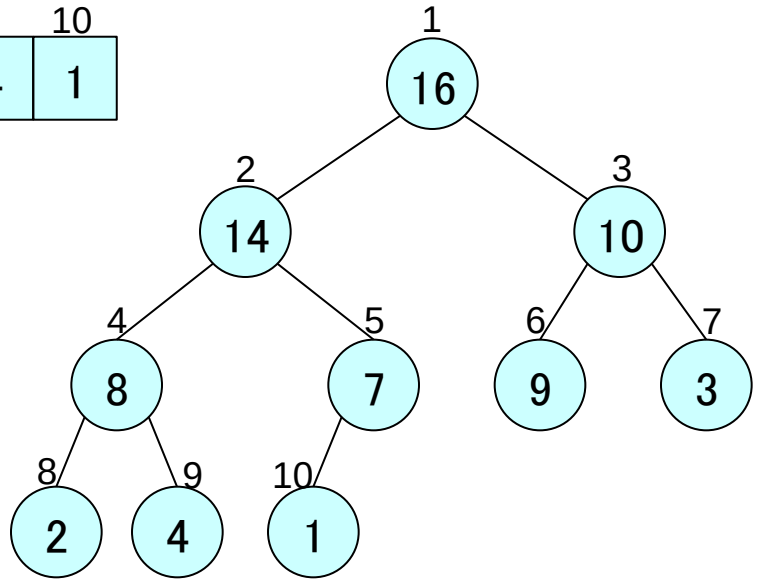
添字iが与えられたとき，

その親 $PARENT(i) = \lfloor i/2 \rfloor$

左の子 $LEFT(i) = 2i$

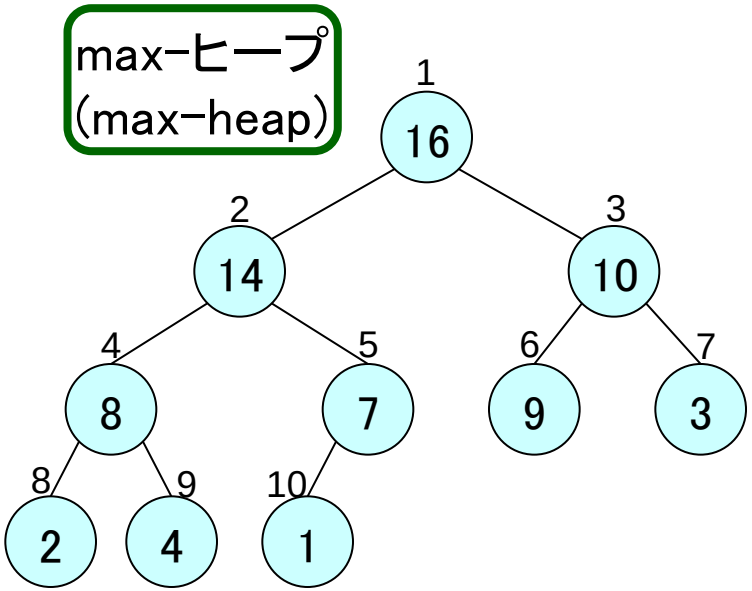
右の子 $RIGHT(i) = 2i + 1$

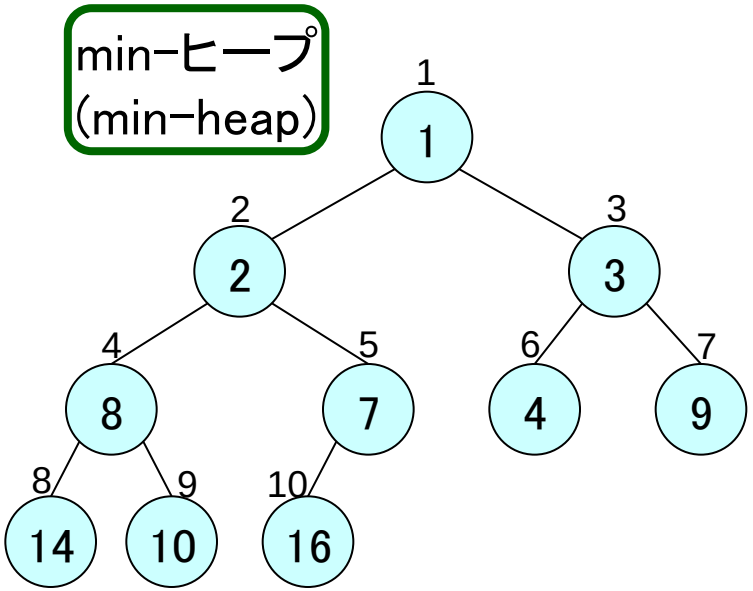
と簡単に計算可能



2分木ヒープの種類

2分木ヒープは，max-ヒープとmin-ヒープの2種類

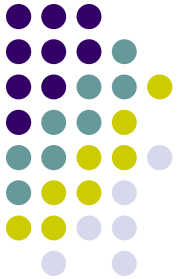




- max-ヒープ条件(max-heap property)
  根以外の接点iが
  $A[PARENT(i)] \geq A[i]$
- 最大の要素が根に格納される

- min-ヒープ条件(min-heap property)
  根以外の接点iが
  $A[PARENT(i)] \leq A[i]$
- 最小の要素が根に格納される

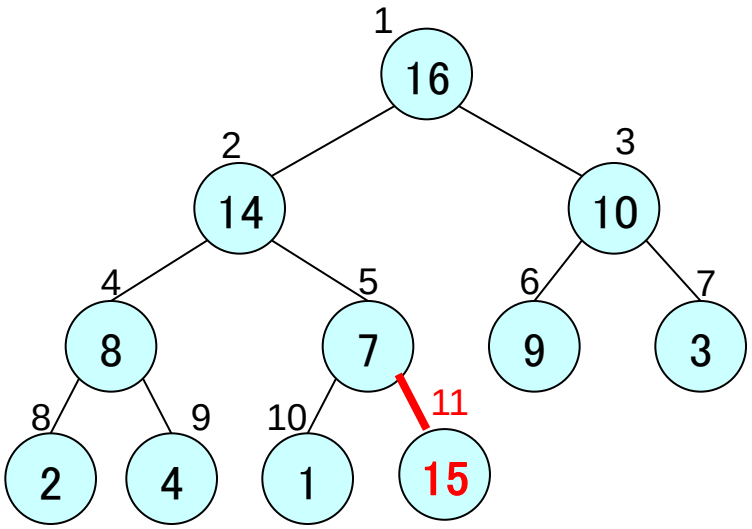
ヒープに対する操作



- insert(val): 新しい値valを挿入
- delete（i）: ノードiを削除
- changekey（i,val）: ノードiの値をvalに変更

操作を行った後、木を再構成してヒープ条件維持

insert(val)

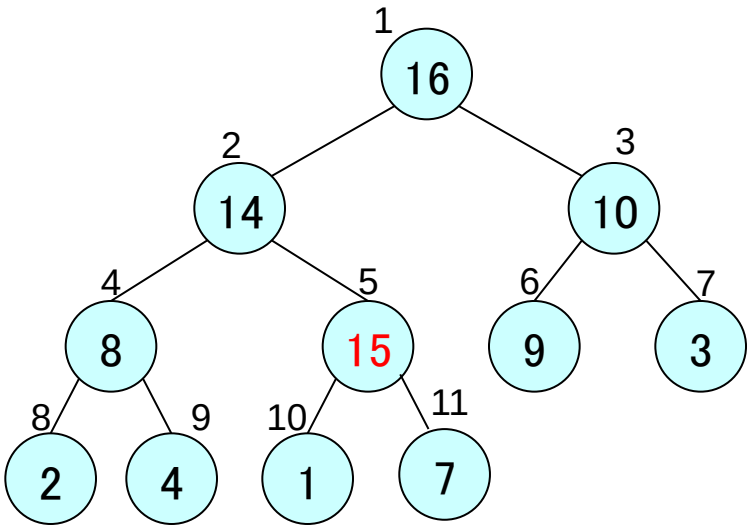


insert(値 val)
1 heapsizeを1増やす
2 A[heapsize]=val
3 upheap(heapsize)

upheap(インデックス i)
1 j= $\lfloor i/2 \rfloor$
2 if(a[i]>a[j])
3 a[i]⇔a[j]
4 upheap(j)

1. 新しい値を配列の最後に挿入
2. ヒープの条件が満足されていない場合に親子の入れ替え

insert(val)

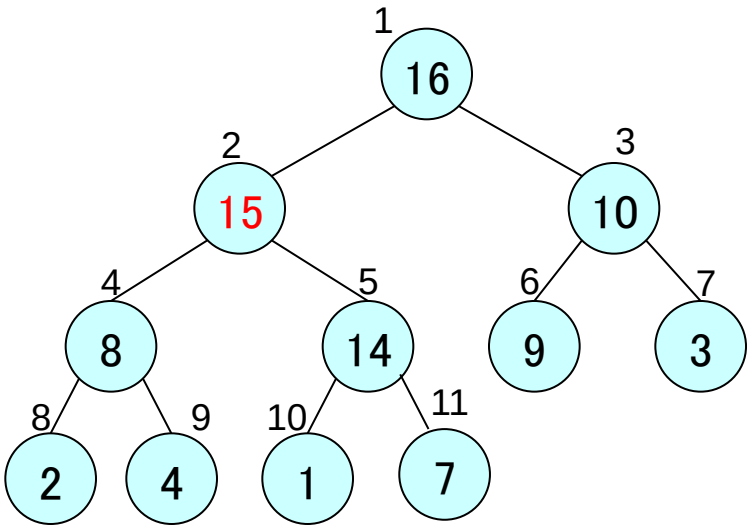


insert(値 val)
1 heapsizeを1増やす
2 A[heapsize]=val
3 upheap(heapsize)

upheap(インデックス i)
1 j= $\lfloor i/2 \rfloor$
2 if(a[i]>a[j])
3 a[i]⇔a[j]
4 upheap(j)

1. 新しい値を配列の最後に挿入
2. ヒープの条件が満足されていない場合に親子の入れ替え

insert(val)



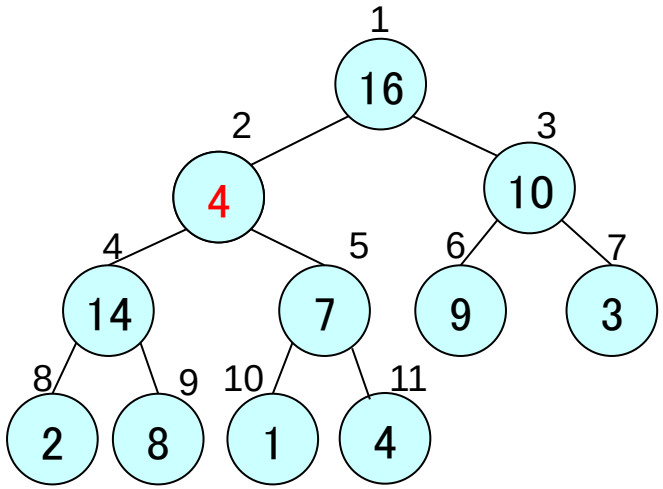
insert(値 val)
1 heapsizeを1増やす
2 A[heapsize]=val
3 upheap(heapsize)

upheap(インデックス i)
1 j= $\lfloor i/2 \rfloor$
2 if(a[i]>a[j])
3 a[i]⇔a[j]
4 upheap(j)

1. 新しい値を配列の最後に挿入
2. ヒープの条件が満足されていない場合に親子の入れ替え

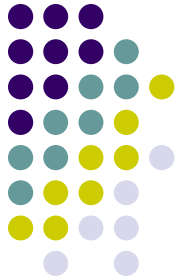
delete(i)

```
delete(インデックス  i)
  if i ≠ heapsize
       old=A[i]
       A[i]=A[heapsize]
       if(old > A[i])  downheap(i)
       else upheap(i)
  ヒープサイズを1減らす
```



1. ノードiを削除
2. 配列の最後のノードを削除位置に移動
3. ヒープの条件が満足されていない場合に親子の入れ替え

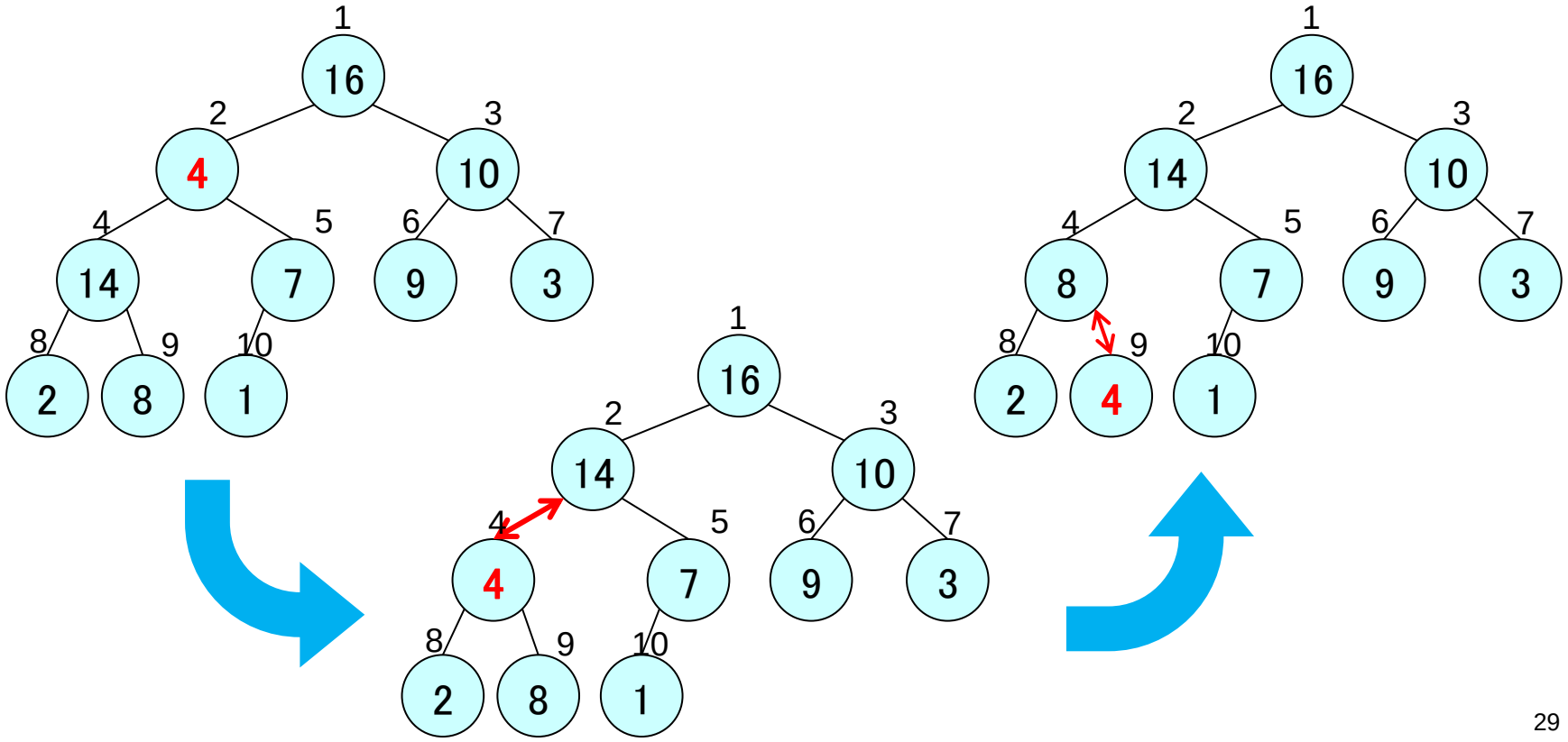
downheapの疑似コード



```
downheap(i)
 1  a ← LEFT(i)
 2  b ← RIGHT(i)
 3  if A[a] > A[i]
 4     then largest ← a
 5     else  largest ← i
 6  if A[b] > A[largest]
 7     then largest ← b
 8  if largest ≠ i
 9     then 値の交換 A[i] ⟷ A[largest]
10          downheap(largest)
```

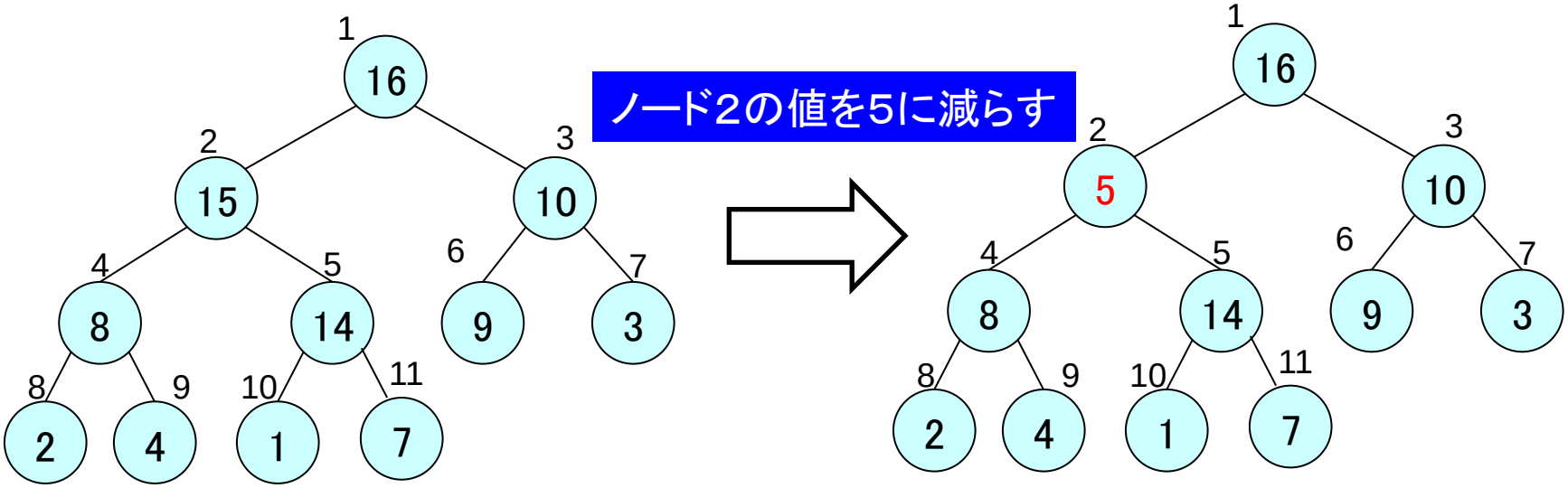
downheapの動き

max-ヒープの条件を満たさない節点A[i]がある場合，
max-ヒープ条件を満たすようにA[i]の値を，「滑り落とす」

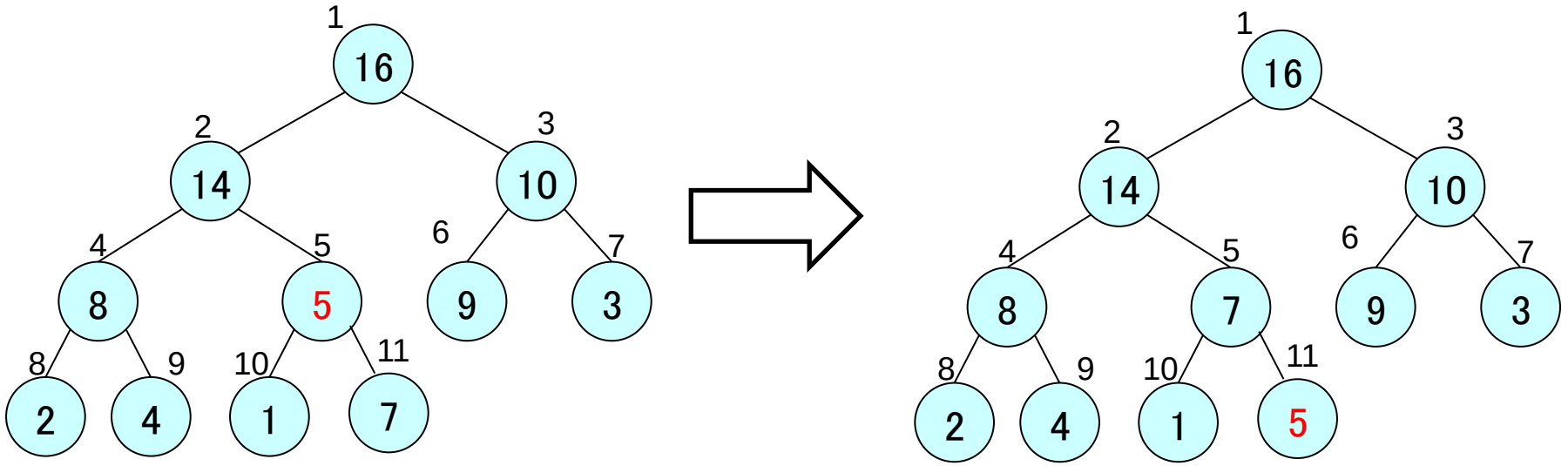


changekey(i,val)

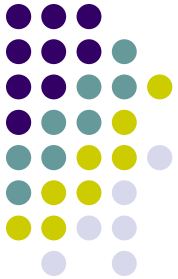
```
changekey(i,val)
    old = A[i]
    A[i] ← val
    if(old>A[i])  downheap(i)
    else upheap(i)
```



```
changekey(i,val)
    old = A[i]
    A[i] ← val
    if(old>A[i])  downheap(i)
    else upheap(i)
```



Dijkstraのアルゴリズム(ヒープ版)



- min-heapを使用
  - d(v)を格納

アルゴリズム

$d(s) \leftarrow 0$, $S \leftarrow s$, $Q \leftarrow V - \{s\}$

for 各頂点 $v \in Q$ について, $d(v) \leftarrow \infty$ //

for 各 s の隣接点 u について $d(u) \leftarrow w(s,u)$ // 初期化

while $Q \neq \Phi$ do

　Q の中から$d(x)$ が最小の頂点 x を取り出す

　S に x を追加

　for 各 x の隣接点 y について,

　　if $d(y) > d(x) + w(x,y)$

　　then $d(y) \leftarrow d(x) + w(x,y)$, $p(y) \leftarrow x$

|V|回のinsert

changekey(index(u),w(s,u))

ヒープのrootにアクセスしてから削除

changekey(index(y),d(x)+w(x,y))

ヒープ版の計算量

$O((|E| + |V|) \log |V|)$

1. changekey: 最大$|E|$回
2. insert: $|V|$回
3. delete:$|V|$回

- 各関数の実行時間＝$O(\log |V|)$

理論的にはFibonattiヒープを使えば$O(|V| \log |V| + |E|)$に減らせることが知られている

最短経路問題

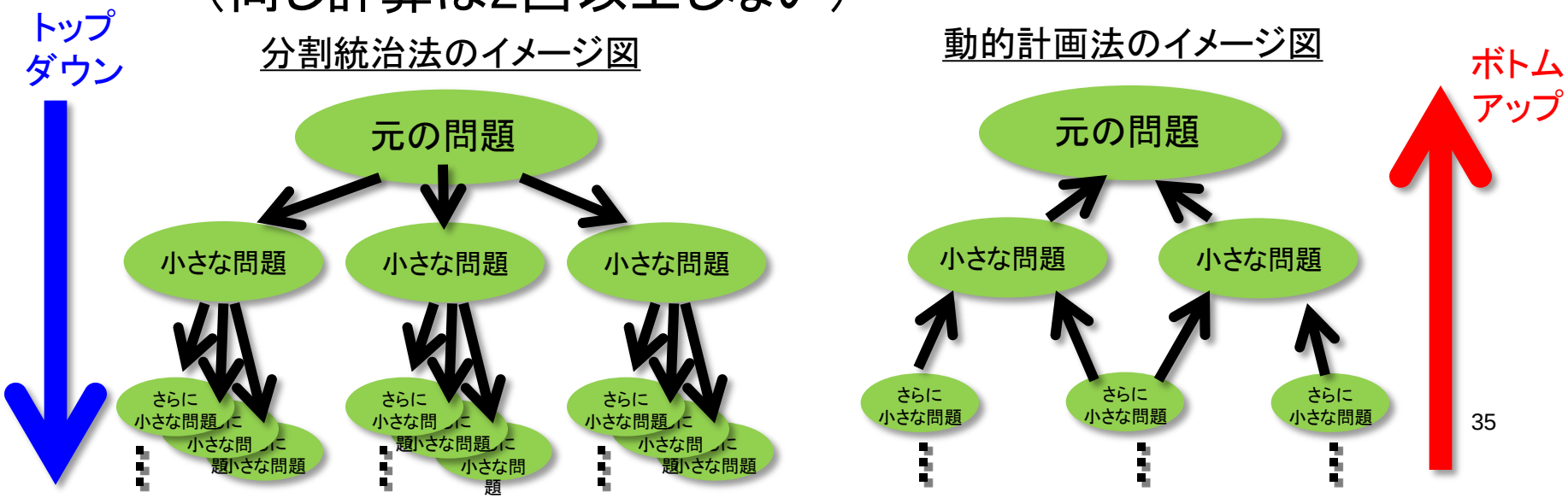
- 入力
  - 辺に正の重みが付いた重み付きグラフG
- 出力
  - 2点v_i, v_j間の最短経路
    - 経路P: v_i, v_j間を結ぶパス
      - 長さ$l(P)$: 経路を構成する辺の重みの和 $\sum_{e \in p} w(e)$

最適化問題:実行可能解の中で目的関数を最小(最大)化する解を見つける問題

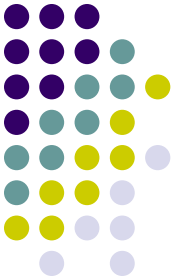
1. 単一始点最短路問題: 入力G, v
2. 全点間最短路問題: 入力G

動的計画法

- 最適化問題に対するアルゴリズム設計技法の1つ
- 大きな問題を部分問題の解を組み合わせて解く（ボトムアップ的）
  1. 小さい問題に対する解を先に計算しておく
  2. 小さい問題の解を利用して大きい問題を効率的に解く
- 1度計算したものは表に記憶して後で利用（同じ計算は2回以上しない）



動的計画法を適用できる基準



- 問題が部分構造最適性(optimal substructure)を持つ
  - 大きい問題の最適解が部分問題の最適解を含む
    - 漸化式で関係性を表現
      - 分割統治法の漸化式とは異なる
- 部分問題重複性(overlapping subproblem)
  - 部分問題数が少ない

参考:フィボナッチ数列

$$f(n) = f(n-1) + n$$

Floyd-Warshallのアルゴリズム

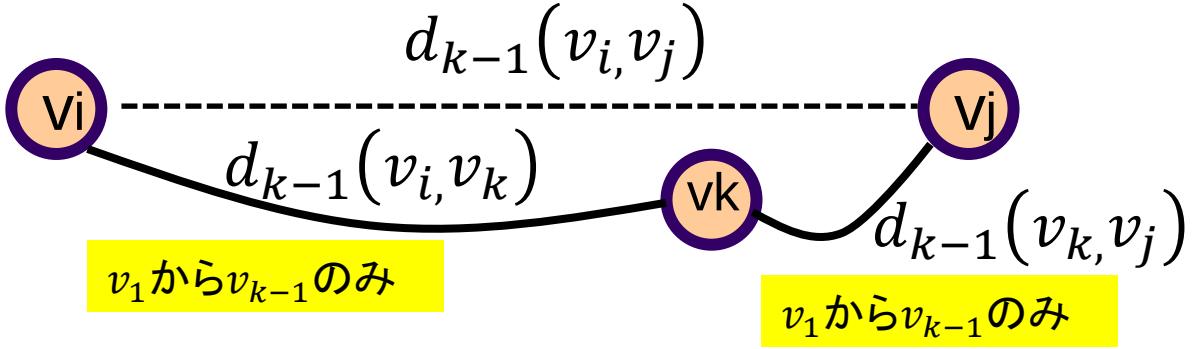
- $d(v_i, v_j)$: v_i, v_j間の最短経路長
- 最適化問題　$d_k(v_i, v_j)$: $\{v_1, v_2 \cdots v_k\}$のみを経由するv_i, v_j間の最短経路長を求める問題を考える

n:頂点数

$$d(v_i, v_j) = d_n(v_i, v_j)$$

- 部分問題　$d_{k-1}(v_i, v_j)$: $\{v_1, v_2 \cdots v_{k-1}\}$のみを経由する$v_i, v_j$間の最短経路長を求める問題

$$d_k(v_i, v_j) = \min\{d_{k-1}(v_i, v_j),\ d_{k-1}(v_i, v_k) + d_{k-1}(v_k, v_j)\}$$



Floyd-Warshallのアルゴリズム

- $d(v_i$
- 最適 v_j間
  の最

部分構造最適性

v_k を通るv_i, v_j間の最短経路は以下を含む

- $\{v_1, v_2 \cdots v_{k-1}\}$のみを通る$v_i, v_k$間の最短経路
- $\{v_1, v_2 \cdots v_{k-1}\}$のみを通る$v_k, v_j$間の最短経路

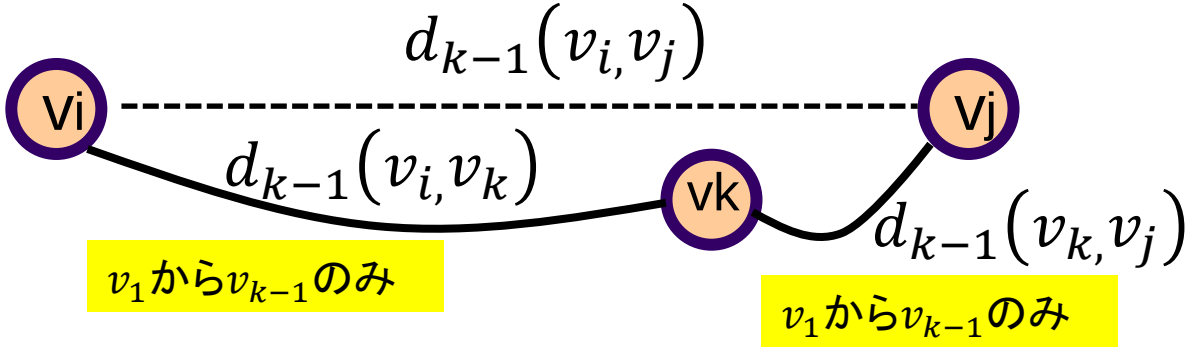
部分問題数： $O(n^3)$

n:頂点数

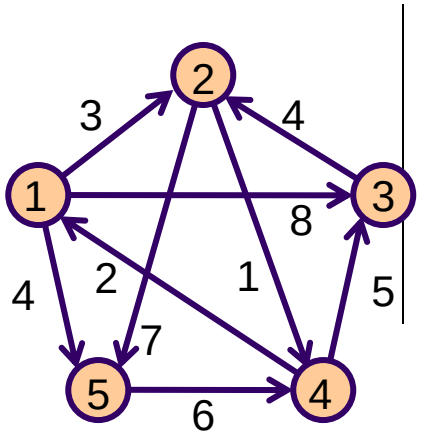
部分

- 部分問題　$d_{k-1}(v_i, v_j)$: $\{v_1, v_2 \cdots v_{k-1}\}$のみを経由する$v_i, v_j$間の最短経路長を求める問題

$$d_k(v_i, v_j) = \min\{d_{k-1}(v_i, v_j),\ d_{k-1}(v_i, v_k) + d_{k-1}(v_k, v_j)\}$$



Floyd-Warshallのアルゴリズム



1. D_0 を $d_0(i,j)$ からなる $n \times n$ 行列とする

   $d_0(i,j)$: 頂点 i から頂点 j への辺の長さ

   $d_0(i,j) = w_1(i,j)$　　辺がなければ∞

2. D_1 を $d_1(i,j)$ からなる $n \times n$ 行列とする

   $d_1(i,j)$: 中継点として 頂点 1 のみを許す, 頂点 i, j 間の最短距離

   $d_1(i,j) = \min\{ d_0(i,j), d_0(i,1) + d_0(1,j) \}$

3. D_2 を $d_2(i,j)$ からなる $n \times n$ 行列とする

   $d_2(i,j)$: 中継点として 頂点 1, 2 のみを許す, 頂点 i, j 間の最短距離

   $d_2(i,j) = \min\{ d_1(i,j), d_1(i,2) + d_1(2,j) \}$

D_0

	1	2	3	4	5
1	0	3	8	∞	4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	5	0	∞
5	∞	∞	∞	6	0

D_1

	1	2	3	4	5
1	0	3	8	∞	4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	5	0	6
5	∞	∞	∞	6	0

D_2

	1	2	3	4	5
1	0	3	8	4	4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	5	0	6
5	∞	∞	∞	6	0

アルゴリズム

アルゴリズム

D_0 を作成

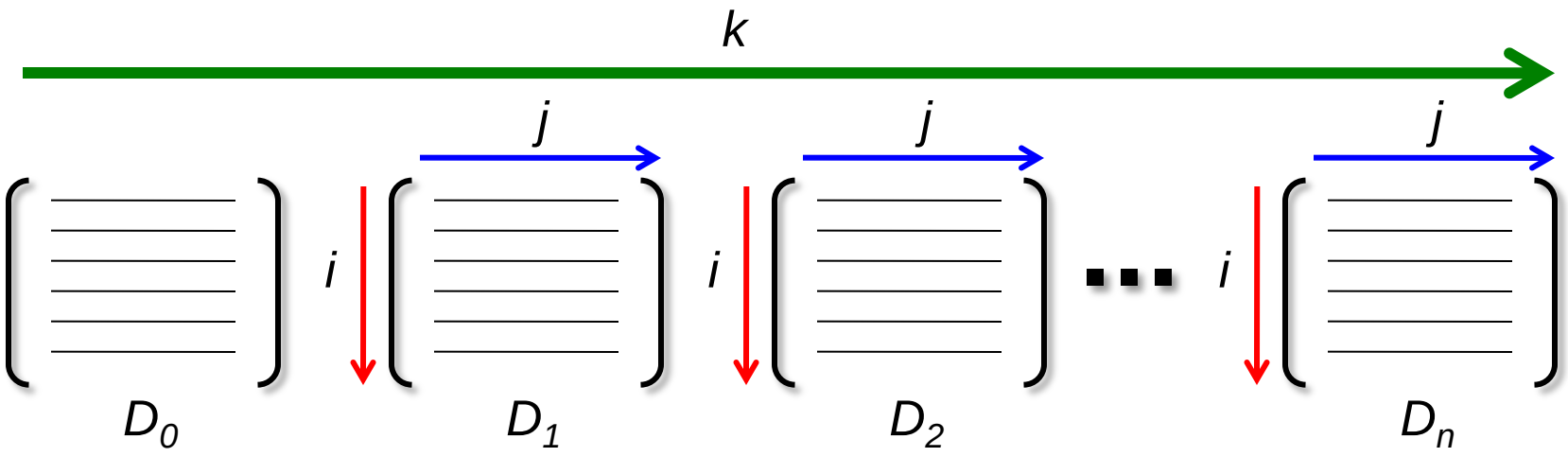
for each k = 1 … n

　for each j = 1 … n

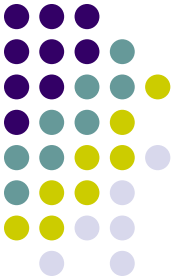
　　for each i = 1 … n

　　$d_k(i,j) = \min\{ d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j) \}$

$O(n^3)$時間



DijkstraとFloyd-Warshall



- Dijkstraアルゴリズム（ヒープ版）を全点に適用すると全点間最短路問題が解ける
  - 計算量: $O(|V|^2 \log|V| + |E||V| \log|V|)$

$$|V| - 1 \leq |E| \leq |V|(|V| - 1)$$

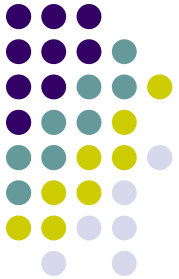
木　　完全グラフ

- Floyd-Warshall
  - 計算量: $O(|V|^3)$

グラフの疎密に応じて適切なアルゴリズムは変化

- **疎なグラフ（辺数が少ない）に対してはDijkstraの方が速い**

目次

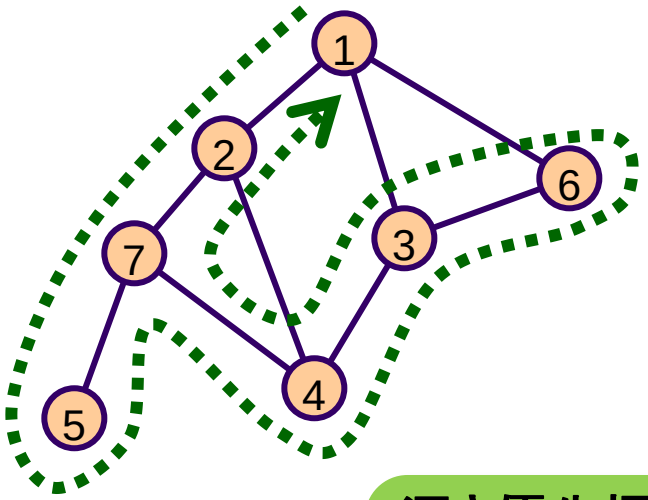


- グラフとは
- 最短経路問題
  - Dijkstraのアルゴリズム
  - Floyd-Warshallのアルゴリズム（動的計画法）
- 付録：グラフの探索（深さ優先，幅優先）

グラフの探索

問題

グラフ中の全ての頂点を訪問したい



もっとも基本的な探索手法

- 深さ優先探索
  （*Depth-First Search*）
- 幅優先探索
  （*Breadth-First Search*）

深さ優先探索の例

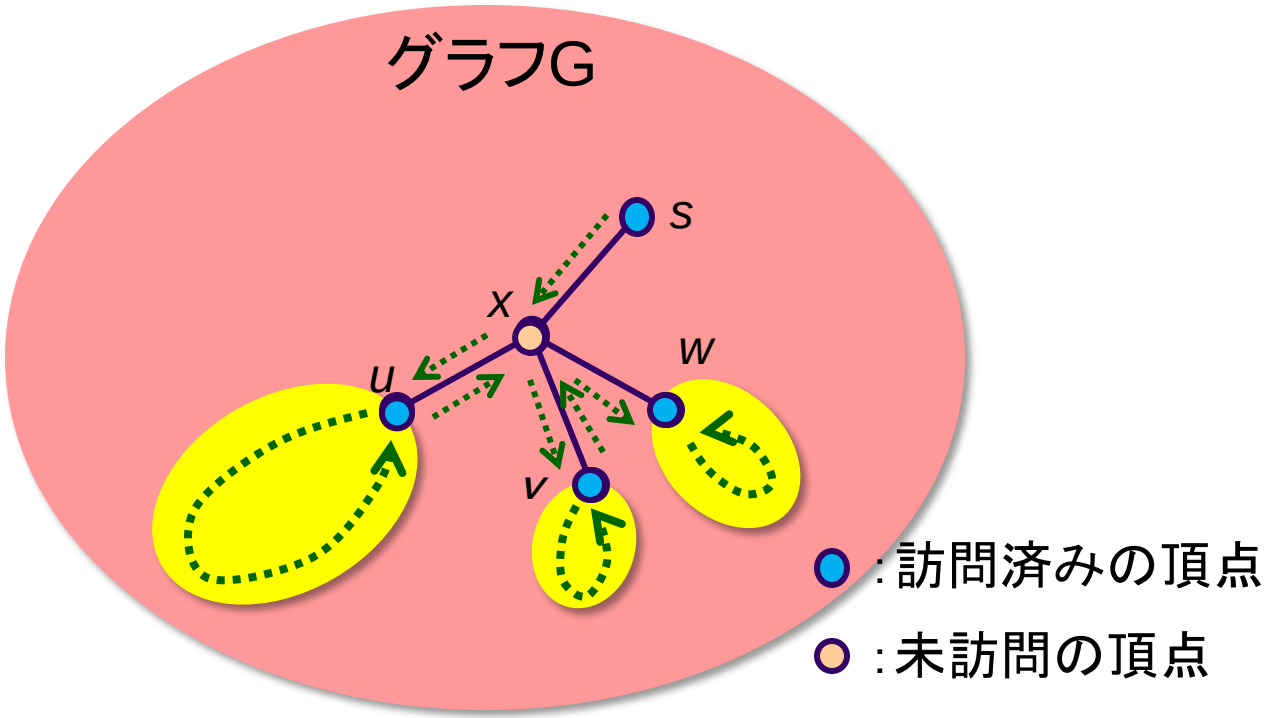
訪問順：1, 2, 7, 5, (7), 4, 3, 6, (3), (4), (7), (2), (1)
（カッコ内は2回目以降の訪問を意味する）

深さ優先探索

方針

まだ訪れていない頂点を優先的に探索
（未訪問の頂点を見つけたら即座に訪問）

（現在、頂点 s にいる状況を考えて下さい！）



深さ優先探索の擬似コード

DFS(*x*)

for each *x* の隣接点 *u* **begin**

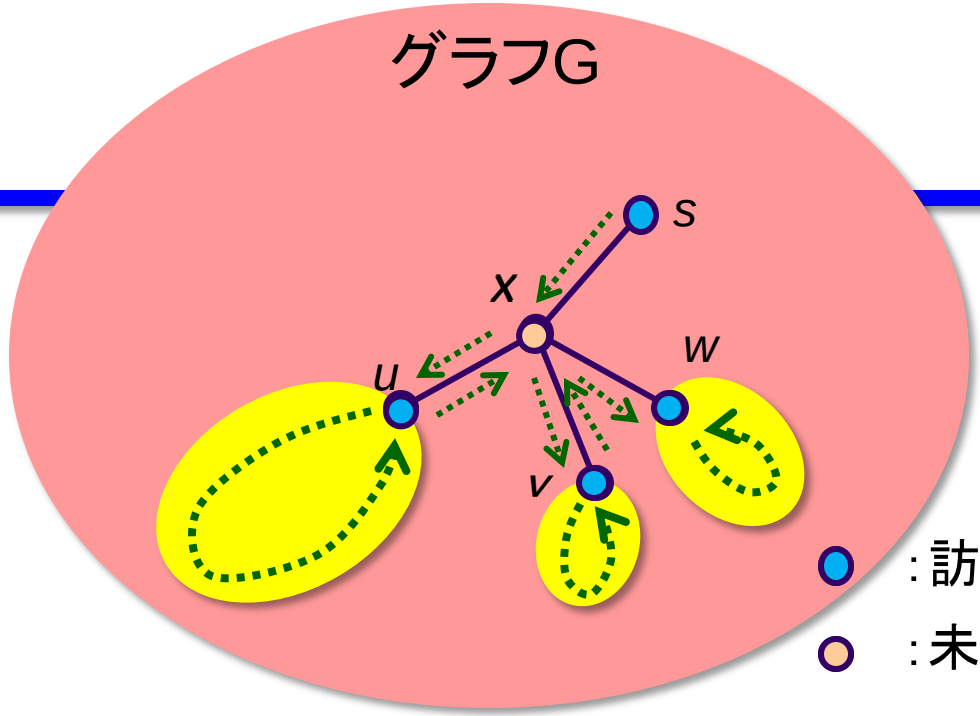
  if *u* は未訪問 **begin**

    u を訪問した旨を出力, *u* を訪問済みの点に変更

    DFS(*u*)　　// 再帰呼出

  end

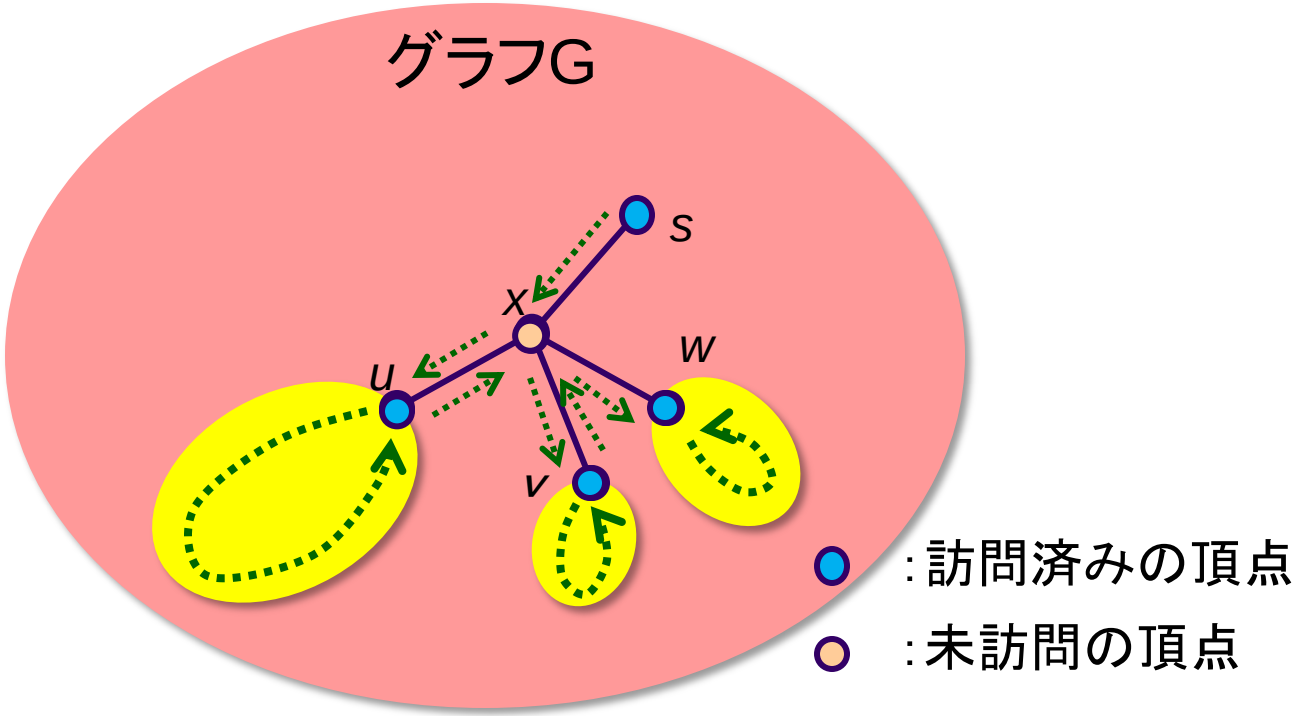
end



幅優先探索

方針

まずは, 現在の頂点に隣接する頂点を全て訪問
終わったら, “次の”頂点へ探索を進める



幅優先探索の擬似コード

BFS(*s*)

始点 *s* をキュー*Q* にエンキュー　// エンキュー: キューへの追加

while *Q* ≠ Φ **begin**

　Q からデキュー　// デキュー: キューから要素の取り出し

　得られた頂点を *x* とする

　for each *x* の隣接点 *u*

　　if *u* は未訪問 **then begin**

　　u を訪問した旨を出力し、訪問済みの頂点とする

　　u をQ にエンキュー

　　end

end