アルゴリズム論
Theory of Algorithms

第12回講義
スケーリング法

アルゴリズム論
Theory of Algorithms

Lecture #12
Scaling Algorithms

スケーリング法

整数性を利用して問題を効率よく解くための方法.

問題P29:（最短経路問題）
辺に正の整数重みをもつグラフGと1点sが与えられたとき,
始点sからGの他のすべての頂点への最短経路を求めよ.

最短経路問題についてはダイクストラのアルゴリズムが有名.
頂点数をn, 辺数をmとするとき, フィボナッチヒープを用いると,
ダイクストラ法はO(m + n log n)時間で実行できる.

辺の重みがすべて整数であるとき, 効率を改善できるか？
すべての辺の重みを半分にした問題を再帰的に解いて,
その解を用いて元の問題を効率よく解く.

Scaling Algorithms

Algorithms for speeding up using integral property

Problem P29:（Shortest-path problem）
Given a positively weighted graph G and one vertex s, find all shortest paths from s to all other vertices in G.

The Dijkstra's algorithm is famous for the shortest-path problem. For a graph with n vertices and m edges, the Dijkstra's algorithm can be implemented in O(m + n log n) time using Fibonacci Heap.

Is any speed up possible if every weight is integer?

Solve the problem after halving every weight recursively, and using the solution solve the original problem efficiently.

最短経路を求めるダイクストラ法

```
(0) すべての頂点vについてdist[v]=∞とする.
(1) dist[s]=0とする（sは始点）.
(2) すべての頂点をプールPに蓄える.
(3) while(Pが空でない){
(4)    Pからdist[]の値が最小の頂点uを取り出す.
(5)    uにマークをつける.
(6)    for(マークがついていないuの隣接頂点v)
(7)        if( dist[u] + leng(u,v) < dist[v] )
(8)            then dist[v] = dist[u] + leng(u,v)
(9) }
```

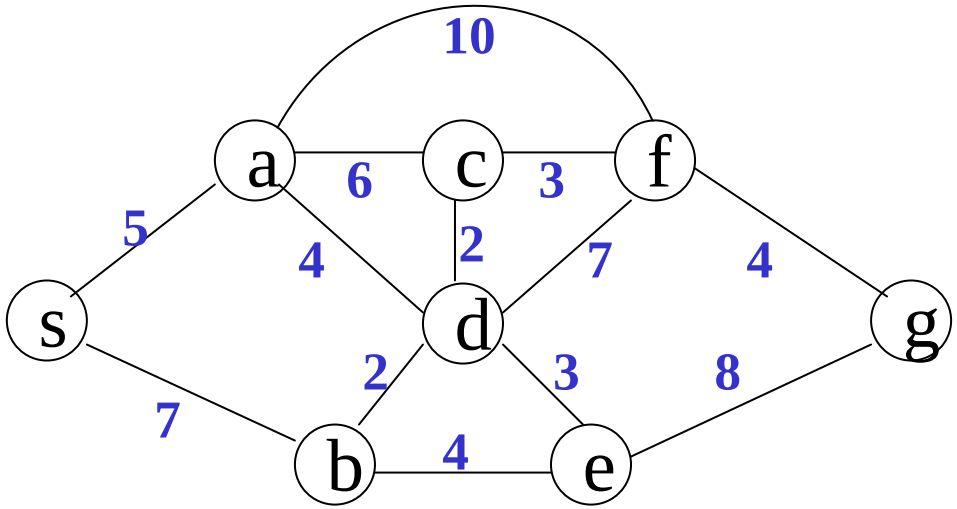
dist[u]: 始点sから頂点uまでの最短経路の長さを蓄える配列.
leng(u,v): ２頂点u, v間の辺の重み（長さ）.
プールP: 頂点の集合を蓄えるためのデータ構造.
dist[]の値が最小の頂点の取り出し,
任意の要素vについてdist[v]の値を減らす.

Dijkstra's algorithm for finding a shortest path

```
(0) For each vertex v, let dist[v]=∞.
(1) Set dist[s]=0（s is the source）.
(2) Store all the vertices in a pool P.
(3) while(P is not empty){
(4)    Take a vertex u with the smallest dist[] out of P.
(5)    Mark u.
(6)    for each unmarked vertex v adjacent to u
(7)        if( dist[u] + leng(u,v) < dist[v] )
(8)            then dist[v] = dist[u] + leng(u,v)
(9) }
```

dist[u]: array to keep the length of a shortest path from s to u.
leng(u,v): weight (length) of an edge between u and v.
Pool P: data structure to maintain a set of vertices.
extract a vertex with the smallest dsit[] value, and
decrease dist[v] value for any element v.

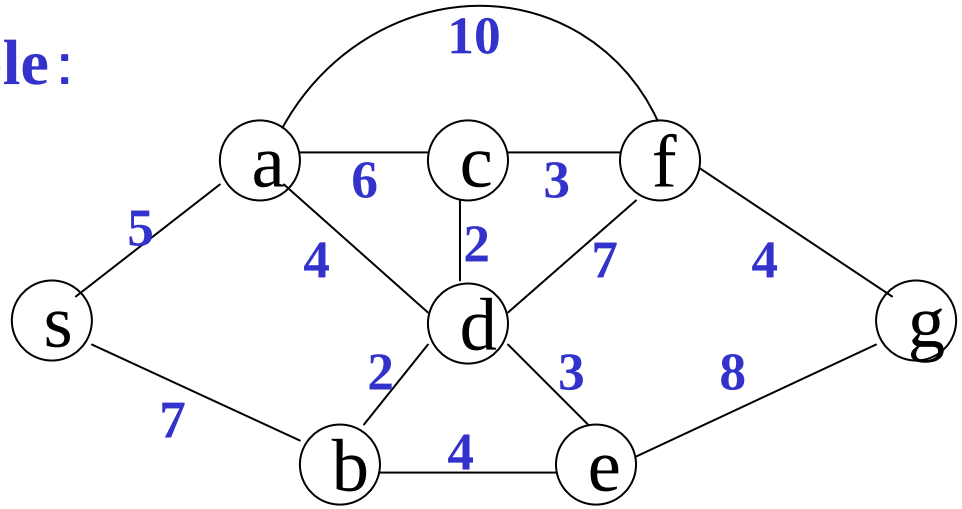
例題:



アルゴリズムの動作:

(1) sをマーク: dist[a]=5, dist[b]=7
(2) aをマーク: dist[c]=11, dist[d]=9, dist[f]=15
(3) bをマーク: dist[d]不変, dist[e]=11
(4) dをマーク: dist[c]不変, dist[e]不変, dist[f]不変
(5) cをマーク: dist[f]=14
(6) eをマーク: dist[g]=19
(7) fをマーク: dist[g]=18
(8) gをマーク: 終了.

Example:



Behavior of the algorithm:

(1) Mark s : dist[a]=5, dist[b]=7

(2) Mark a : dist[c]=11, dist[d]=9, dist[f]=15

(3) Mark b : dist[d] no change, dist[e]=11

(4) Mark d : dist[c]no change, dist[e] no change, dist[f] no change

(5) Mark c : dist[f]=14

(6) Mark e : dist[g]=19

(7) Mark f : dist[g]=18

(8) Mark g : stop.

ダイクストラ法の効率

ダイクストラ法では，すべての辺を一度しか調べない．
（無向グラフの場合には，各辺をそれぞれの方向に一度）

プールPを実現するデータ構造による計算時間の違い

プールPに要求される操作

(A) dist[]の値が最小の頂点の取り出し，　T_A時間

(B)任意の要素vについてdist[v]の値を減らす．　T_B時間

(A)の操作をn回，(B)の操作をm回繰り返すから，全体では
$O(nT_A+mT_B)$時間

(1) プールPを単純な配列で実現する場合

プールの中からdist[]の値が最小の頂点を選ぶのにO(n)の時間がかかるが，dist[v]の値を変更するのはO(1)時間．
したがって，全体では$O(nT_A+mT_B)=O(n^2+m)$時間

Efficiency of the Dijkstra's algorithm

In the Dijkstra's algorithm every edge is checked only once.
(each edge is checked only for each direction for undirected graph)

Difference of computing time for data structure for pool P

Operation required for the pool P

(A) extract a vertex with smallest dist[], T_A time

(B)decrease dist[v] for any element v. T_B time

Repeating operation (A) n times and (B) m times, it amounts to $O(nT_A+mT_B)$ time.

(1) Simple array for pool P

It takes O(n) time to choose one with smallest dist[] in the pool, but it takes only O(1) time to decrease dist[v].
Thus, in total it takes time $O(nT_A+mT_B)=O(n^2+m)$.

(2) プールPを平衡2分探索木で実現する場合

(A) dist[]の値が最小の頂点を選ぶのにO(log n)の時間
(B) dist[v]の値を変更するのもO(log n)時間.
したがって，全体では$O(nT_A+mT_B)=O(n\log n+m\log n)$
$=O((n+m)\log n)$時間.

(3) プールPをフィボナッチヒープで実現する場合

(A) dist[]の値が最小の頂点を選ぶのにO(log n)の時間
(B) dist[v]の値を減らす操作は1回当たりO(1)時間で実行可能.
したがって，全体では$O(nT_A+mT_B)=O(n\log n+m)$時間.
（ただし，計算時間の解析はならし解析に基づく．）
フィボナッチヒープ：Fredman and Tarjan, 1984.

結局，オーダー的に最も良いのはフィボナッチヒープを使う場合のO(m + n log n)時間．これを改善できるか？

(2) Balanced Binary Search Tree for Pool P

(A) O(log n) time to choose a vertex with smallest dist[]
(B) O(log n) time to decrease dist[v].
In total, it takes $O(nT_A+mT_B)=O(n\log n+m\log n)$
$=O((n+m)\log n)$ time.

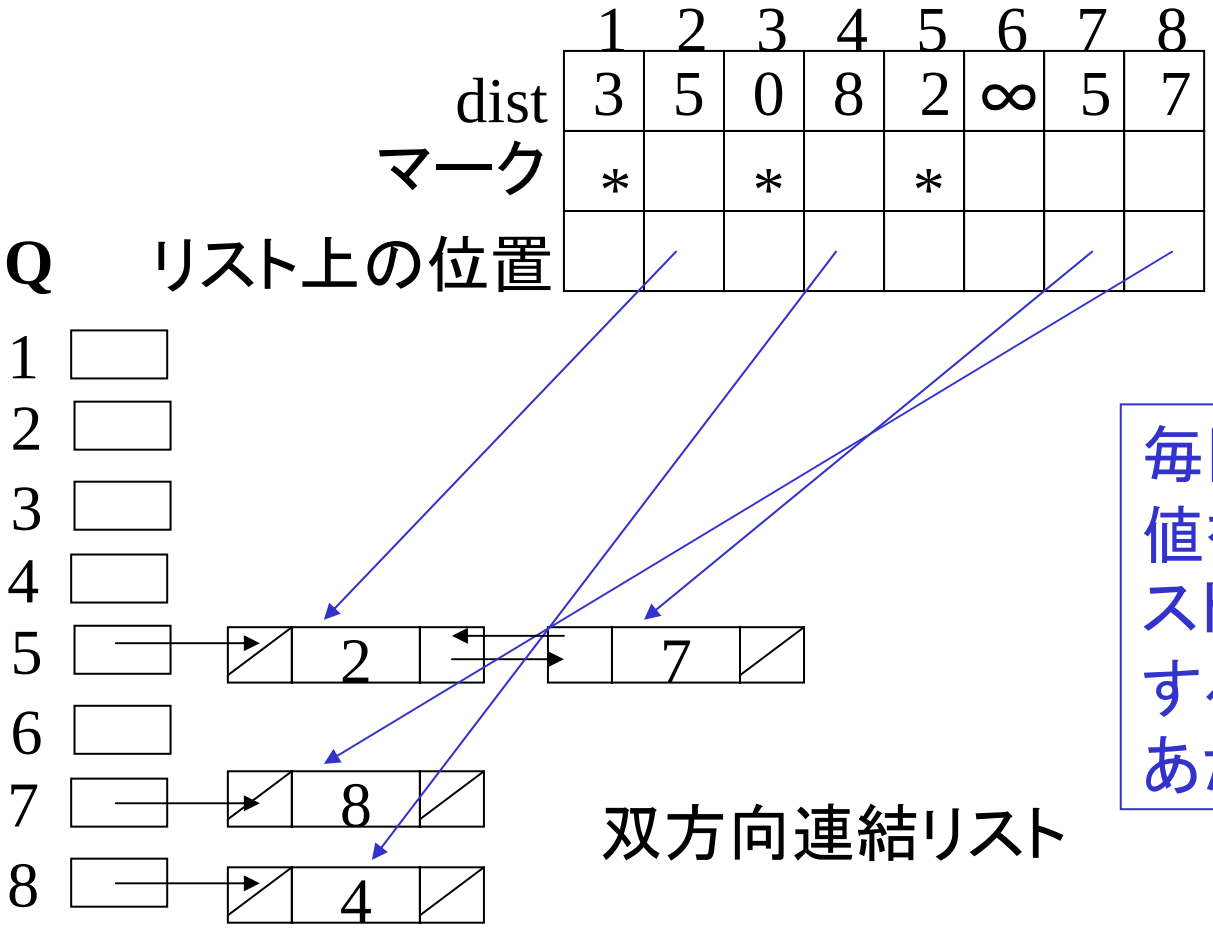
(3) Fibonacci Heap for pool P

(A) O(log n) time to choose a vertex with smallest dist[]
(B) O(1) time in average per one operation to decrease dist[v].
In total, it takes $O(nT_A+mT_B)=O(n\log n+m)$ time.
(the analysis is based on amortization)
Fibonacci Heap : Fredman and Tarjan, 1984.

The best data structure in the order is the one using Fibonacci heap.
It takes O(m + n log n) time. Can we improve it?

辺の重みがm/nの定数倍以下の整数であれば高速化可能

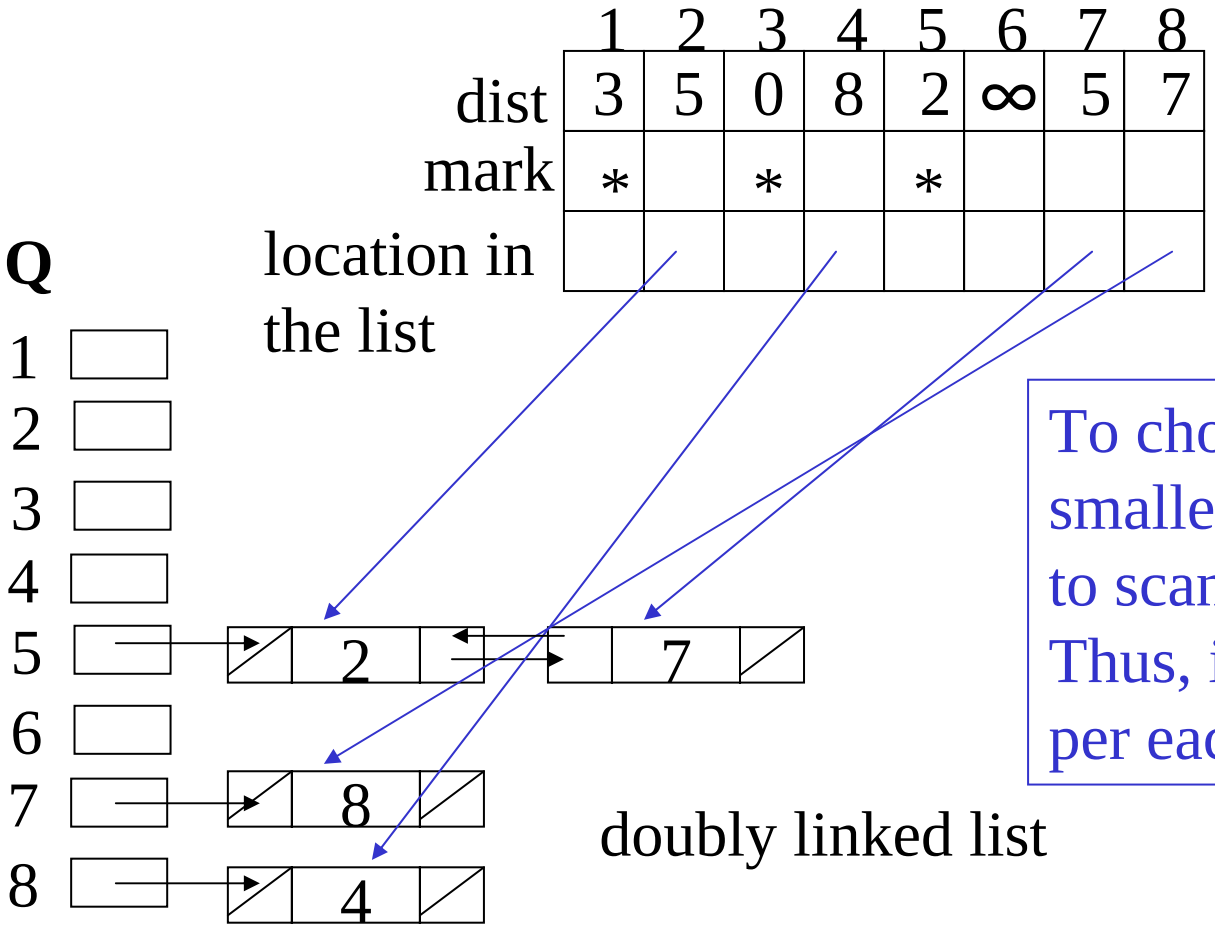
サイズM=O(m)の配列Qを用いて頂点のプールを実現.
すなわち，配列要素Q[k]は，dist[v]=kである頂点への
ポインタを蓄えたリストの先頭を指している.



毎回，dist[]が最小の値を選ぶ操作は，リストQを単調に走査するだけだから，1回あたり定数時間.

Speed up is possible if edge weights are integers of O(m/n)

We implement a pool of vertices using an array Q of size M=O(m). That is, an element Q[k] points to the head of the list of pointers to vertices such that dist[v]=k.



To choose a vertex with smallest dist[] it suffices to scan the list Q in order. Thus, it takes O(1) time per each operation.

dist[]の値ごとに頂点を蓄えるデータ構造

1. dist[v]=kであるすべての頂点vが配列要素Q[k]で先頭を指定されたリストに蓄えられている.
2. リストは双方向リストなので，位置さえわかれば，挿入も削除も定数時間でできる.
3. まだマークされていない各頂点vについて，頂点vからQで管理されるリスト上へのポインタを持っておく.

(A) dist[]の値が最小の頂点を選ぶ操作は毎回定数時間.
（単純にQの上を単調に移動し，最初にnullでないポインタを見つけるだけでよい．また，dist[]の最小値は単調に増加していくから，後戻りもない．）

(B) dist[v]の値を変更するのも定数時間.
（頂点vからのポインタを辿ってQで管理するリスト上でdist[v]の値を削除し，dist[v]の更新された値を適当な場所に挿入）

したがって，**全体では$O(M+nT_A+mT_B)=O(M+n+m)$時間**.

Data Structure to keep vertices for dist[] values

1. Every vertex v such that dist[v]=k is stored in the list whose head is pointed by the array element Q[k].
2. Since the list is doubly linked list, insertion and deletion are done in constant time once their positions are specified.
3. For each vertex v which has not been marked yet, a pointer from v to its position in the list maintained by Q is stored.

(A) Constant time required to choose a vertex with smallest dist[].
（It suffices to scan Q monotonically until we find the first non-null pointer. Also, there is no backtrack because dist[] monotonically increases.)

(B) Constant time required to decrease dist[v] value.
（It suffices to delete the value of dist[v] following the pointer from the vertex v and insert the updated value of dist[v] at an appropriate place in the list.)

Thus, in total it takes **$O(M+nT_A+mT_B)=O(M+n+m)$ time**.

定理12-1: n個の頂点とm本の重みつき辺からなるグラフと1つの頂点sが与えられたとき，辺の重みがすべてm/nの定数倍以内の正整数ならば，先のデータ構造を用いると，sからの最短経路問題はO(n+m)時間で解くことができる.

証明：始点sから任意の頂点までの最短経路は高々n-1本の辺しか通らない. ⇒辺の重みはO(m/n)だから，最短経路の長さはO(m). したがって，先のデータ構造が適用でき，配列Qのサイズも M=O(m)だから，全体の計算時間はO(n+m). 証明終

m/nの定数倍より大きな重みをもつ辺が存在しても，最短経路の長さがどの頂点についてもM=O(m)以内なら，アルゴリズムは適用可能.

辺の重みが辺数mに比べて非常に大きかったり，最短経路の長さがM=O(m)を超える場合はどうする？

スケーリングアルゴリズムの利用

Theorem 12-1: Given a weighted graph with n vertices and m edges and a vertex s, if every edge weight is an integer within a constant factor of m/n, the shortest path problem with respect to s can be solved in O(n+m) time using the data structure described before.

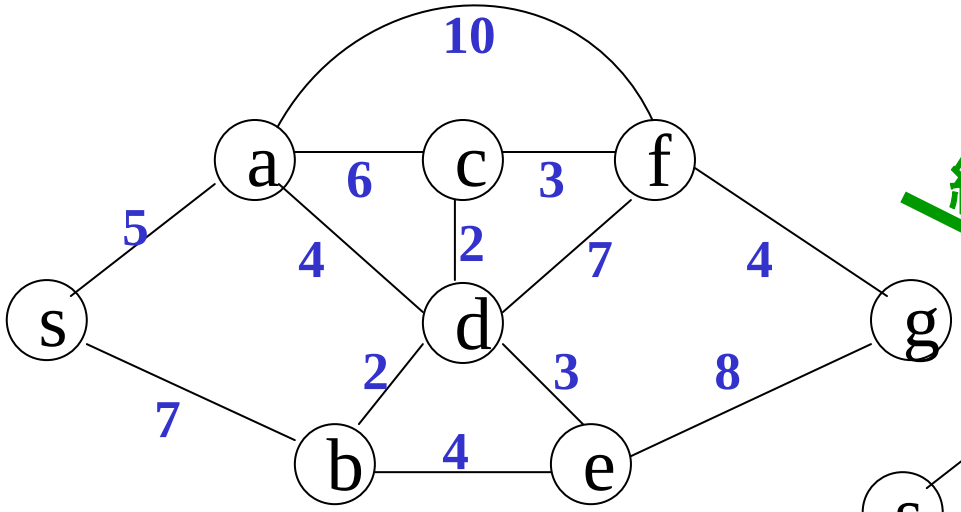
Proof : Any shortest path from s to any vertex passes through at most n-1 edges. ⇒edge weight is O(m/n), so lengths of shortest paths are O(m). Hence, applying our data structure, the total time is O(n+m) since the size of the array Q is also M. Q.E.D.

Even if there is an edge of weight larger than a constant factor of m/n, the algorithm can be applied if the length of any shortest path is bonded by M=O(m).

What about the case in which edge weight is much larger than the number m of edges or length of a shortest path exceeds M=O(m)? Use of Scaling Algorithm

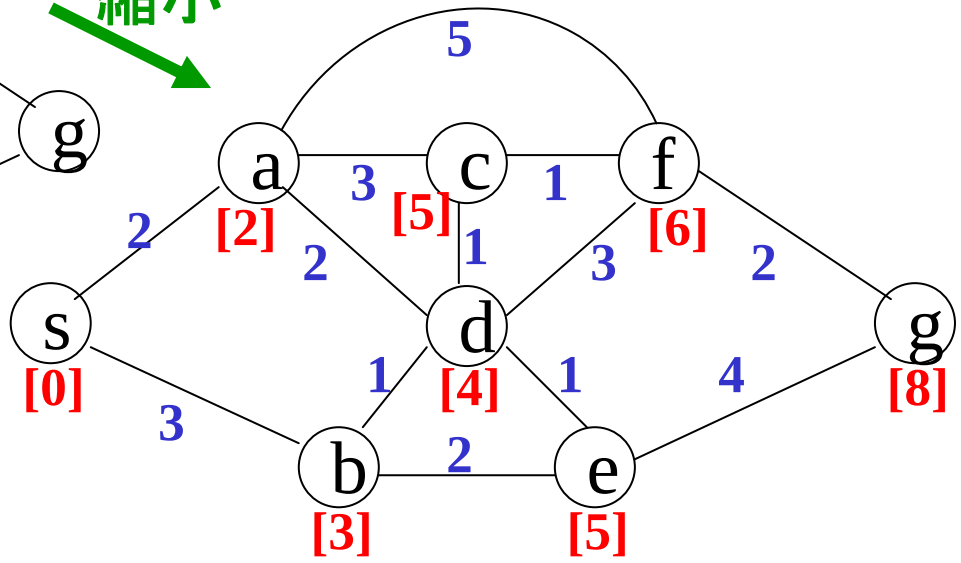
スケーリング法に基づく最短経路発見

ステップ1：各辺の重みを2で割って切り捨てて整数化することによって縮小された最短経路問題を再帰的に解く.
各頂点vまでの最短経路の長さを2倍したものをdist[v]とする.



元のグラフ

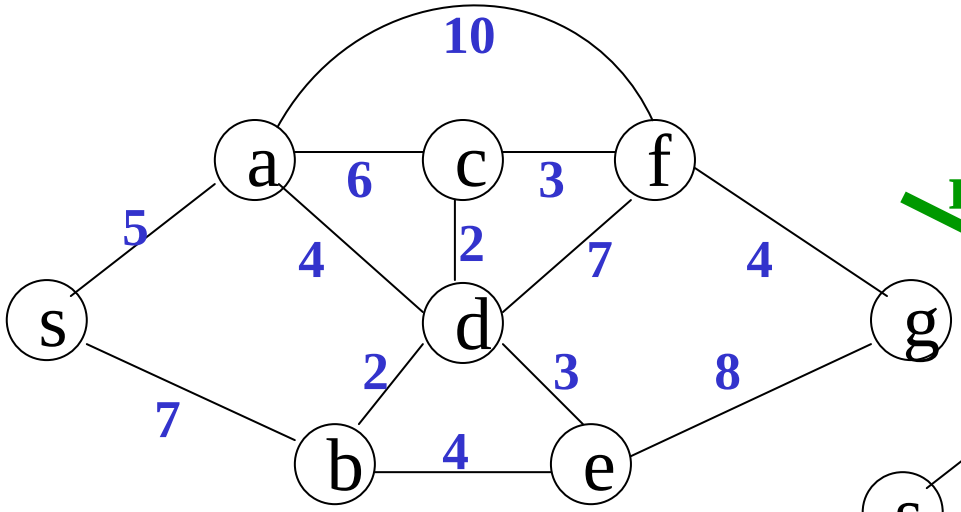
縮小



縮小された問題に対する解
2倍して近似解として使う

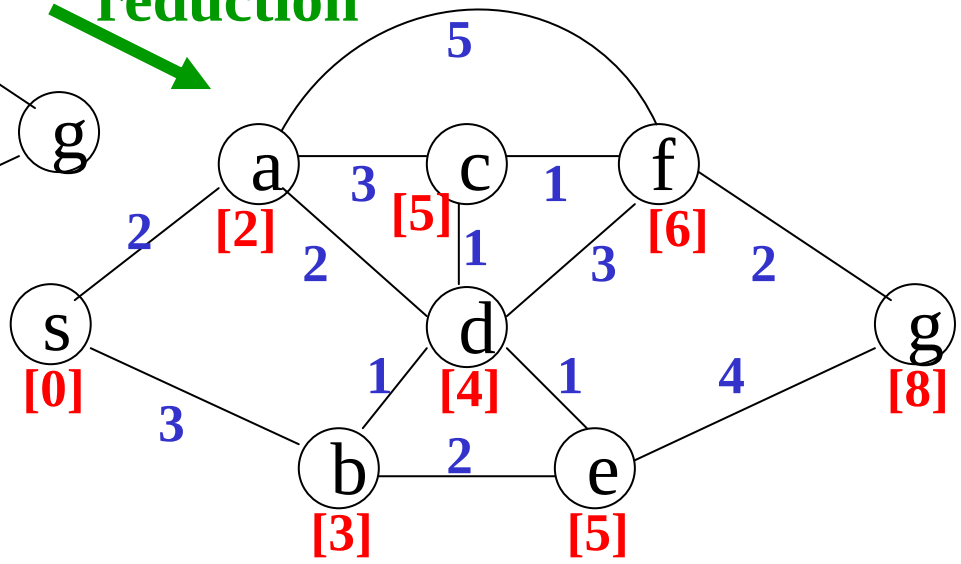
Finding Shortest Paths Using Scaling Algorithm

Step 1: Recursively solve the shortest path problem reduced by dividing each edge weight by 2 and rounding it off.
Let the doubled length of a shortest path to each v be dist[v].

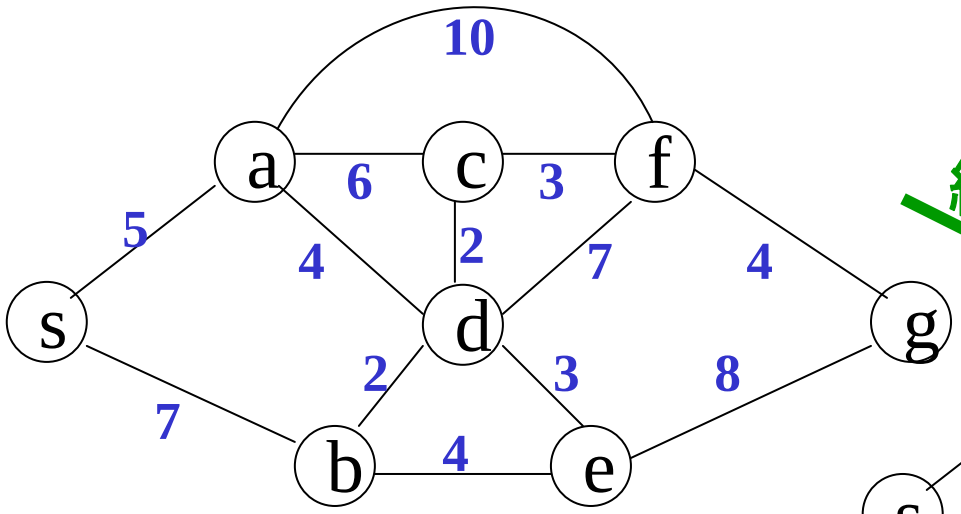


original graph

reduction

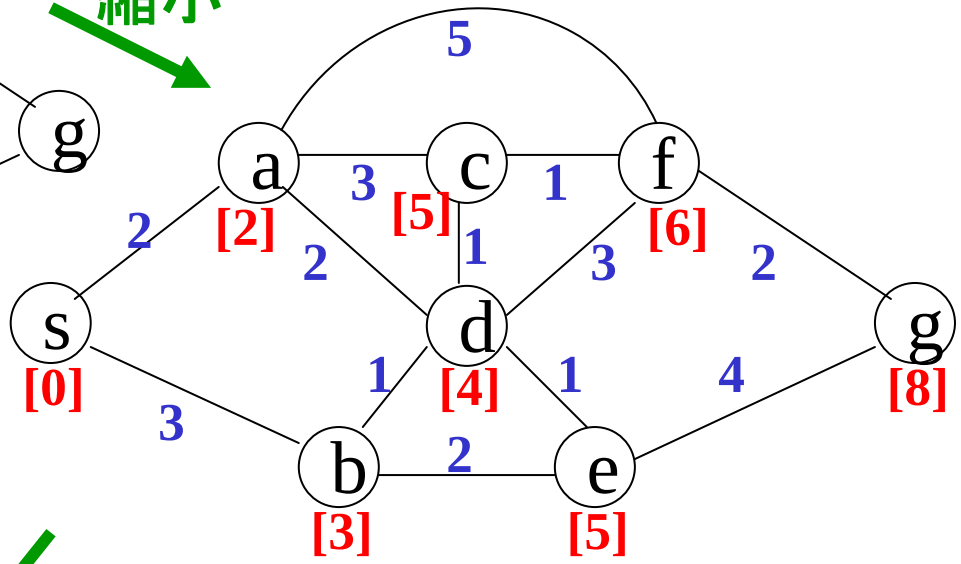


Use the solution to the reduced problem after multiplying it by 2 as an approximation solution

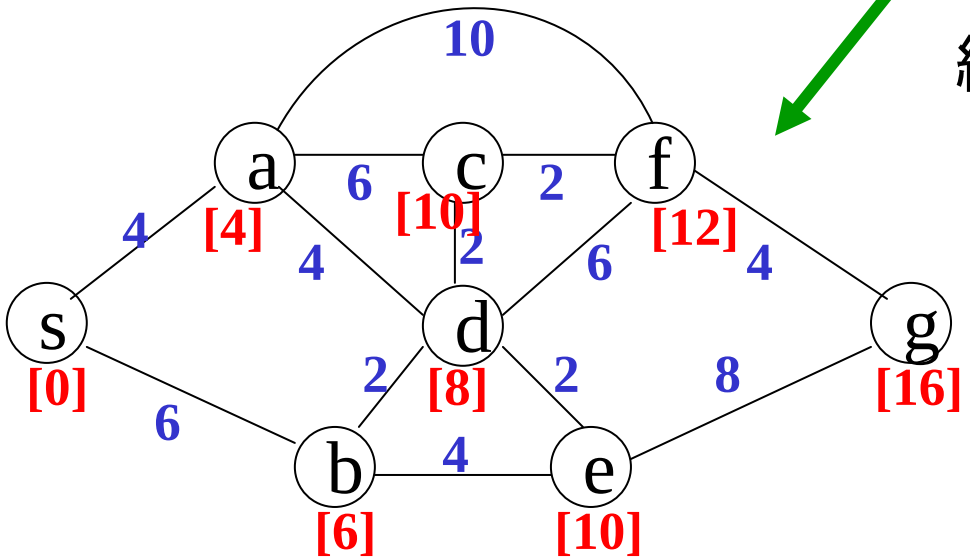


元のグラフ

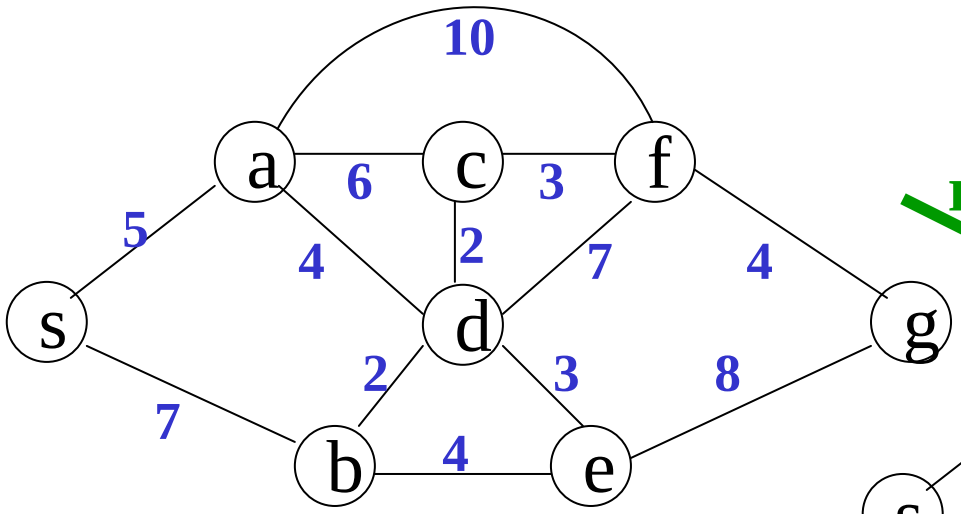
縮小



縮小された問題に対する解
２倍して近似解として使う

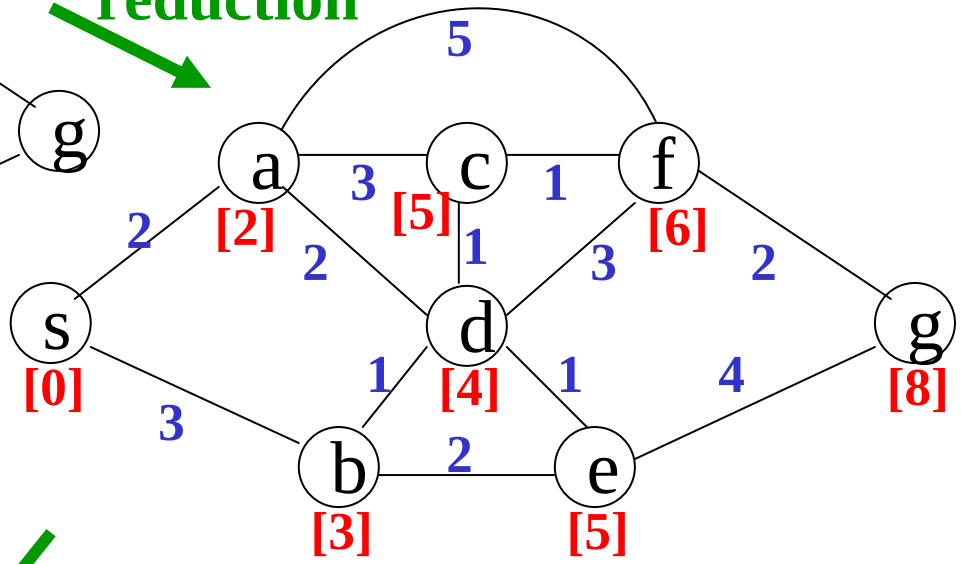


ここで得られた距離を
dist[]とする.

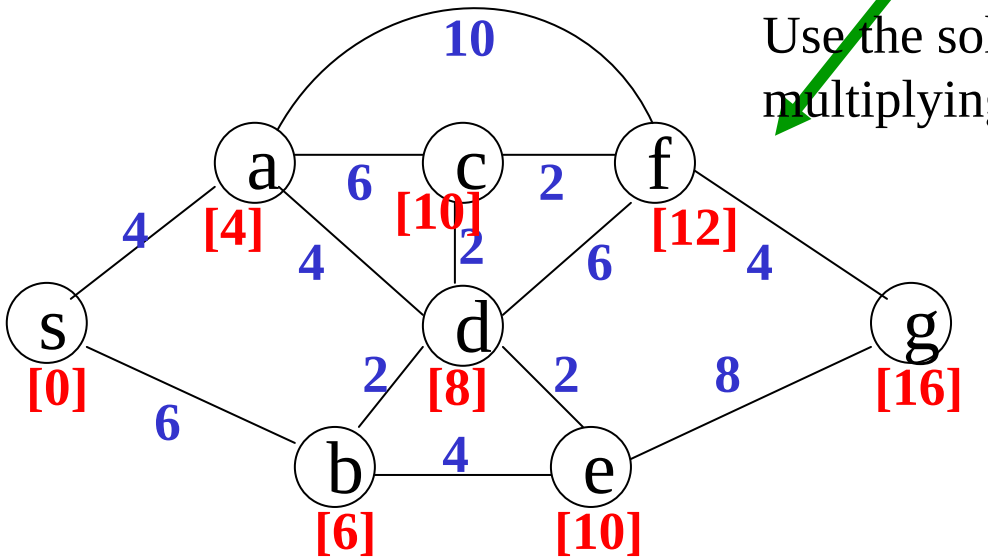


original graph

reduction

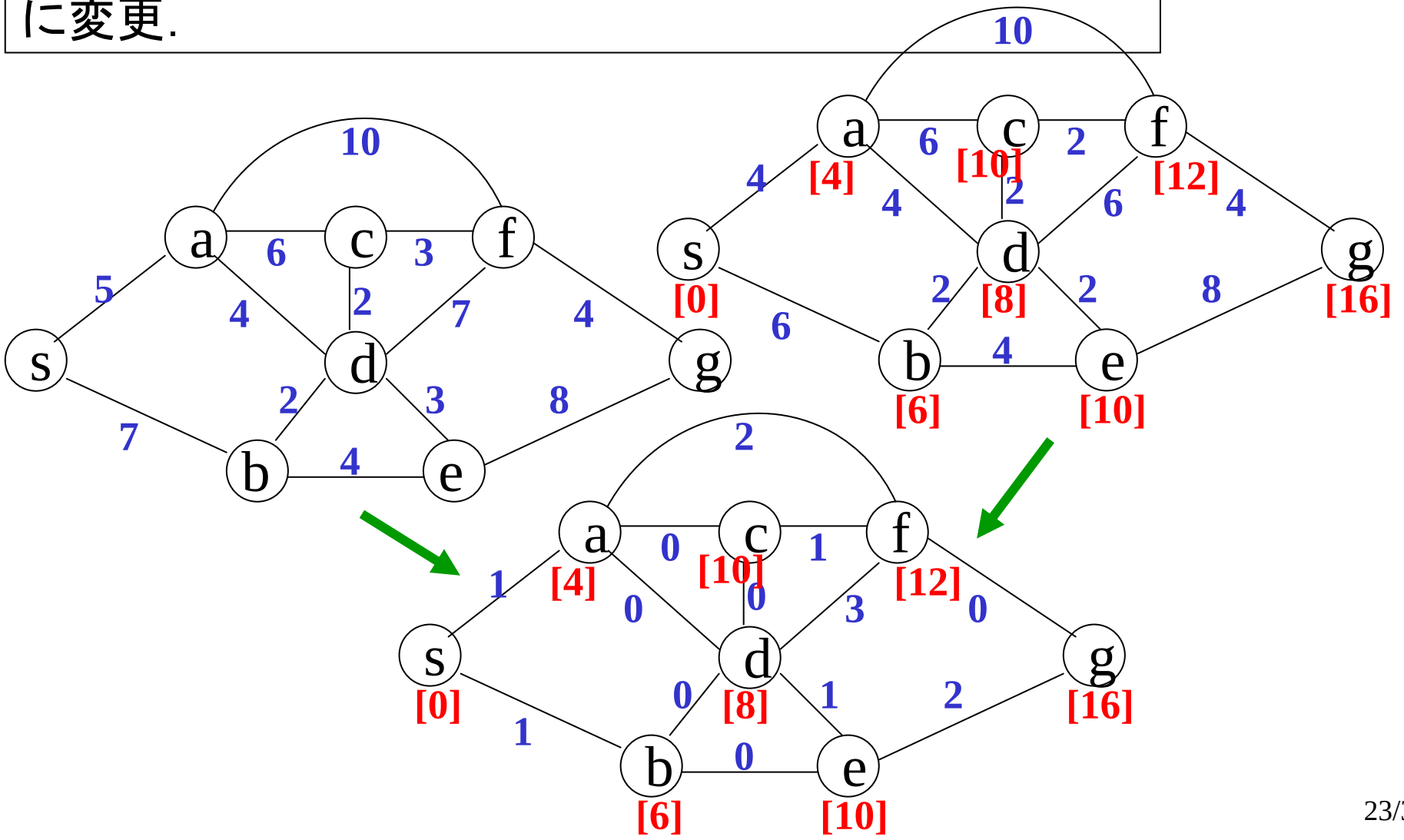


Use the solution to the reduced problem after multiplying it by 2 as an approximation solution

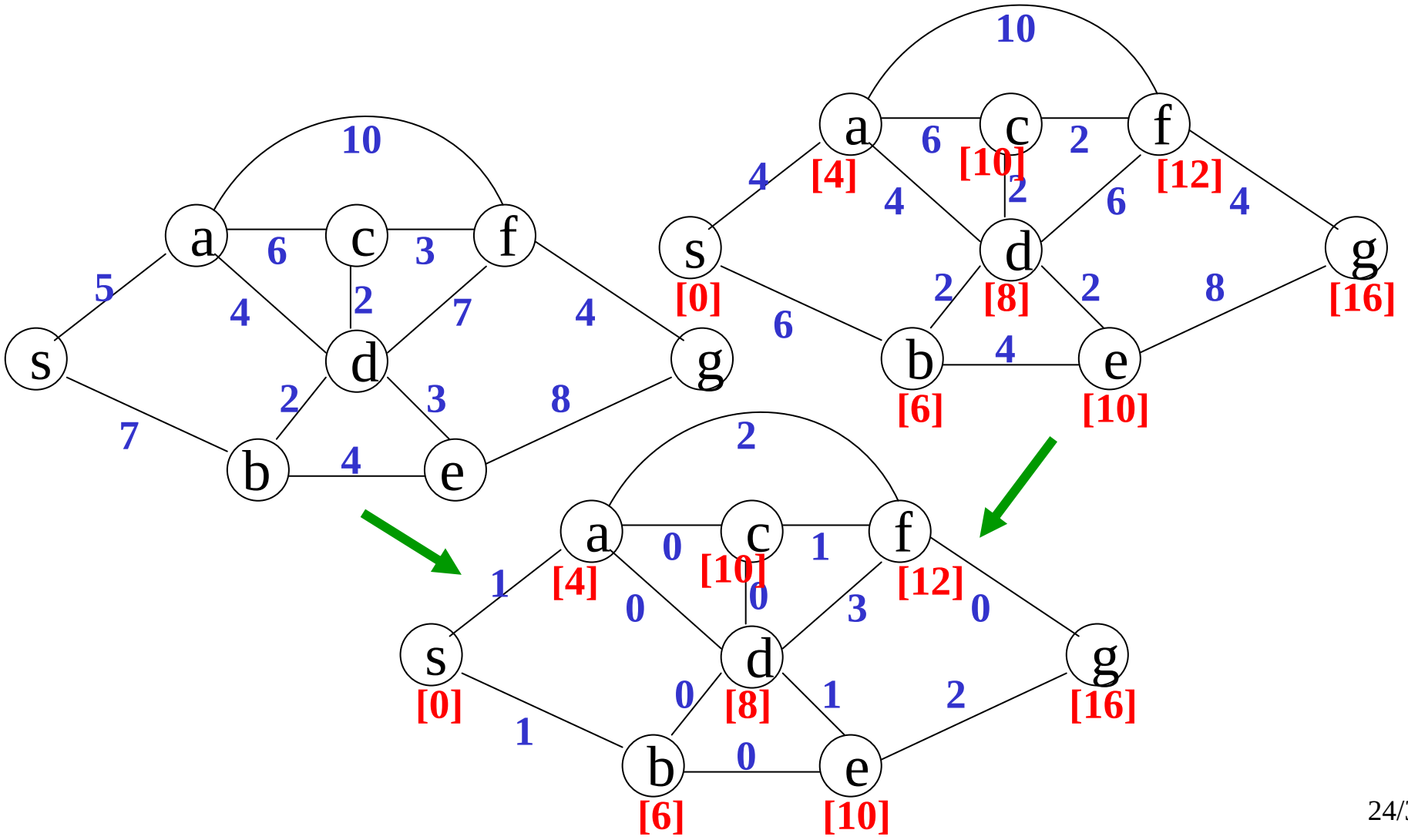


The distance obtained here is dist[].

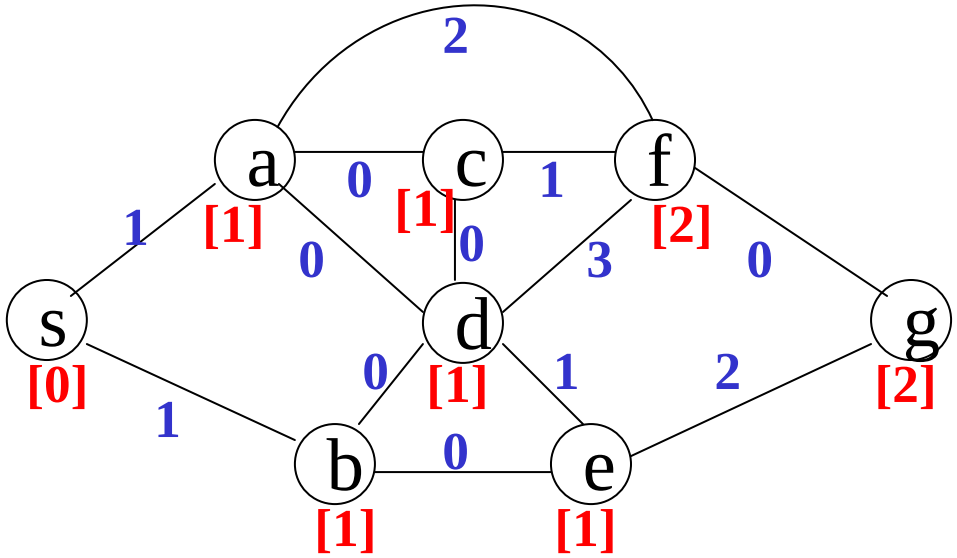
ステップ2：各辺(u,v)の重みleng(u,v)を次のように変更：
dist[u]<dist[v]ならば，その重みを
leng'(u,v)=leng(u,v)+dist[u]－dist[v]
に変更.



Step 2: Modify the weight leng(u,v) of each edge (u,v) as follows:
if dist[u]<dist[v] then change its weight to
leng'(u,v)=leng(u,v)+dist[u]－dist[v].



ステップ3：辺の重みを変更された問題を解き，各頂点までの最短経路の長さをdist'[]とする．

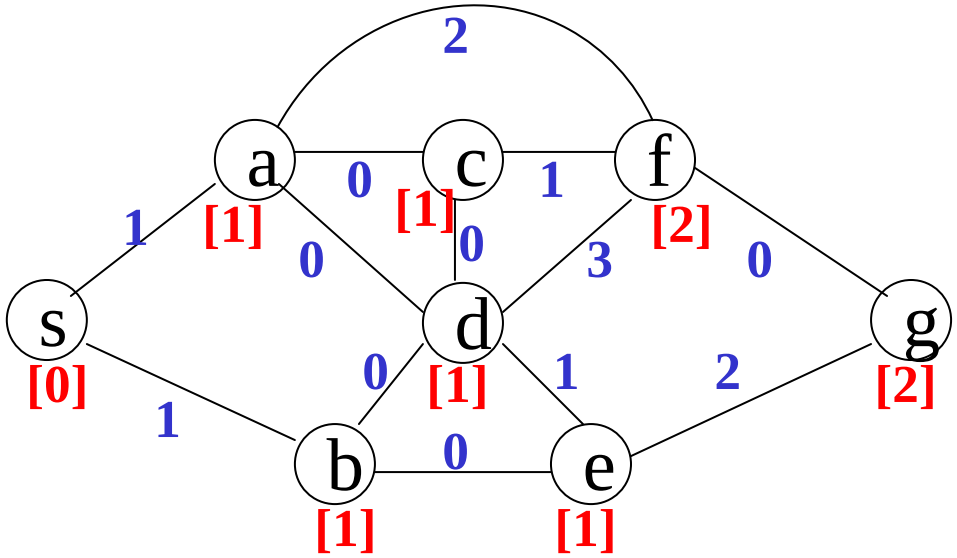


この距離は整数化に伴う誤差分に相当

このグラフの重みは辺の重みを2で割って切り捨てたときの丸め誤差であるから，このグラフでの最短経路の長さは辺数を超えない．=>先の線形時間アルゴリズムが使える．

ステップ4：各頂点についてdist[]の値とdist'[]の値を加えたものをdist[]とし，これを最終的な距離として出力．

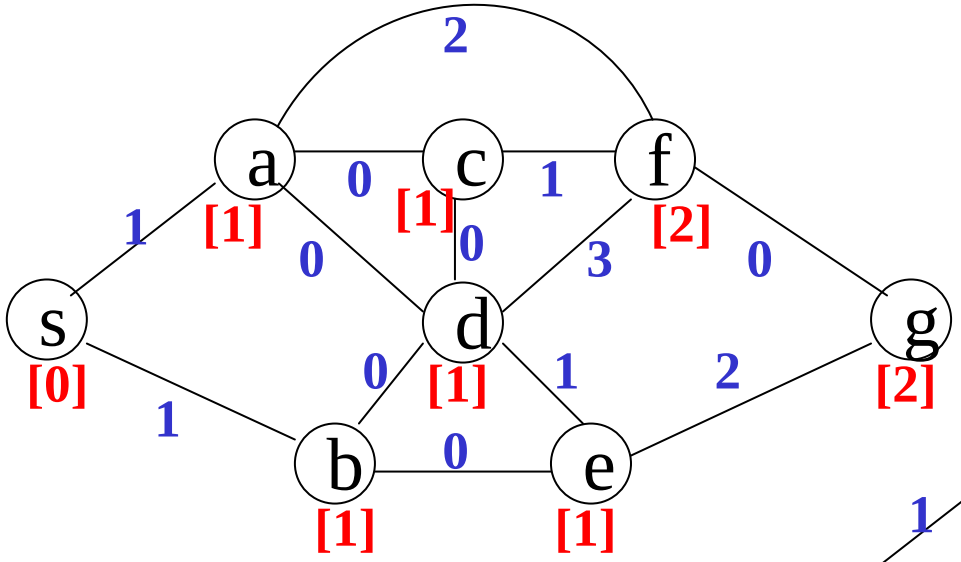
Step 3: Solve the problem resulting by changing weights. Then, let the length of a shortest path to each vertex v be dist'[v].

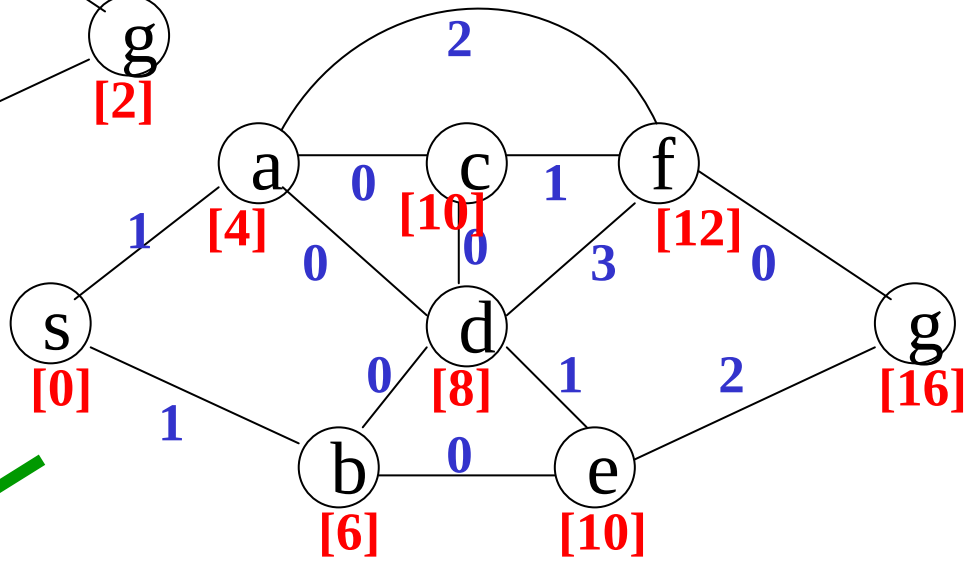


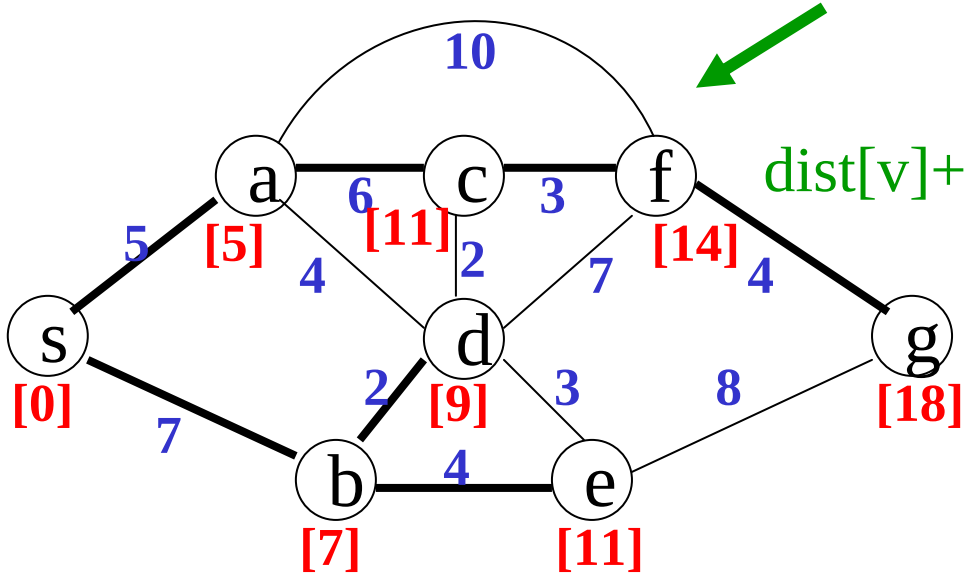
This distance corresponds to rounding error.

Since the weight of the graph is rounding error in dividing weight by 2, the length of a shortest path in this graph does not exceed the number of edges. =>We can apply our algorithm.

Step 4: For each vertex, let the sum of dist[] and dist'[] be dist[]. This value is reported as the final distance.

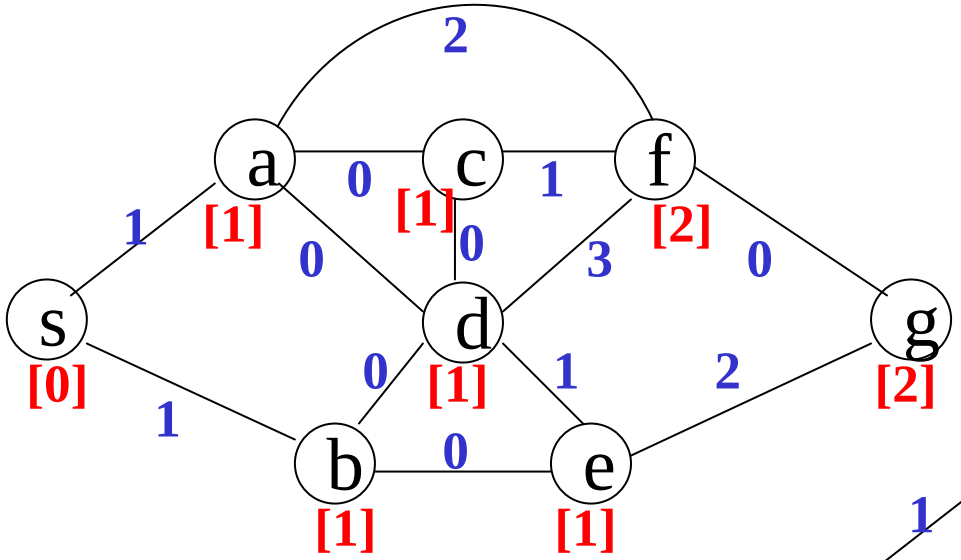


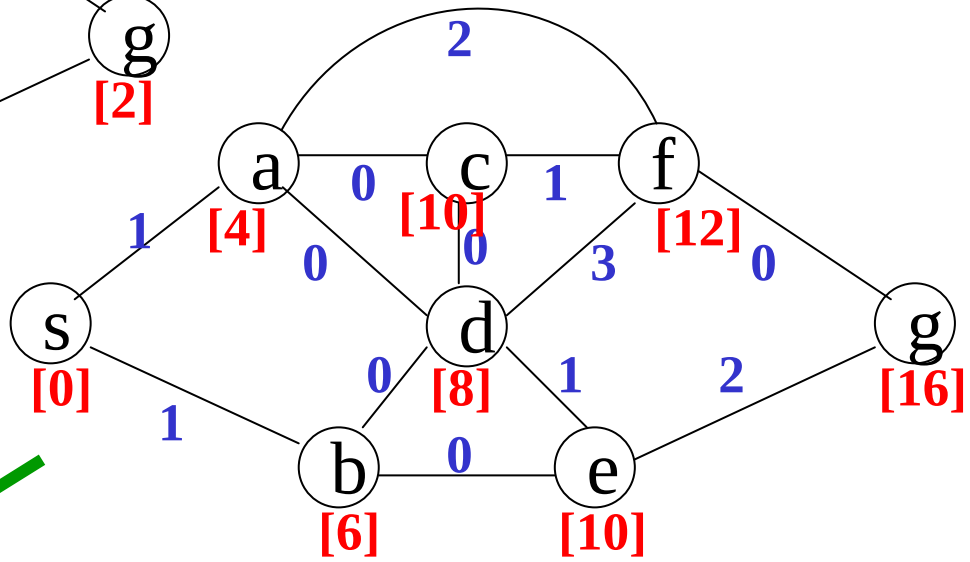


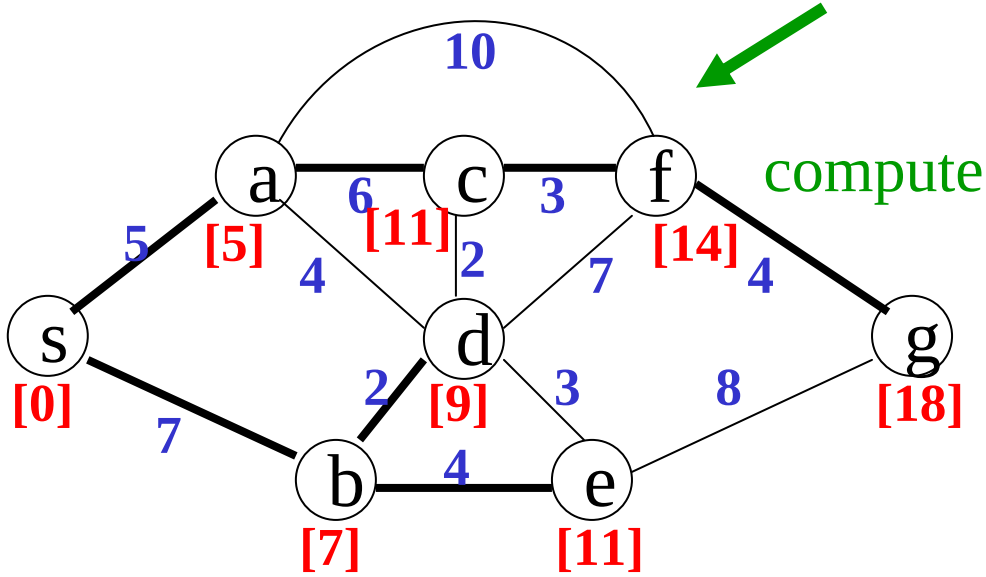


dist[v]+dist'[v]を求める

最終結果と
最短経路木







compute dist[v]+dist'[v]

Final result and
Shortest path tree

アルゴリズムP29-A0: (スケーリングアルゴリズム)
(1)各辺の重みを2で割って切り捨てて整数化することに
よって縮小された最短経路問題を再帰的に解く.
各頂点vまでの最短経路の長さを2倍したものをdist[v]とする.
(2) 各辺(u,v)の重みleng(u,v)を次のように変更:
dist[u]<dist[v]ならば, その重みを
leng'(u,v)=leng(u,v)+dist[u]－dist[v]
に変更.
(3) 辺の重みを変更された問題を解き, 各頂点までの最短
経路の長さをdist'[]とする.
(4) 各頂点についてdist[]の値とdist'[]の値を加えたものを
dist[]とし, これを最終的な距離として出力.

辺の重みの最大値をNとすると, 繰り返し回数は$O(\log_2 N)$回.
(2)-(4)の操作はO(m)時間でできるから, 全体では
$O(m \log_2 N)$時間となる.

Algorithm P29-A0: (Scaling Algorithm)
(1) Recursively solve the shortest path problem reduced by dividing each edge weight by 2 and rounding it off.
Let the doubled length of a shortest path to each v be dist[v].
(2) Modify the weight leng(u,v) of each edge (u,v) as follows:
  if dist[u]<dist[v] then change its weight to
  leng'(u,v)=leng(u,v)+dist[u]－dist[v].
(3) Solve the problem resulting by changing weights. Then, let the length of a shortest path to each vertex v be dist'[v].
(4) For each vertex, let the sum of dist[] and dist'[] be dist[]. This value is reported as the final distance.

Let N be the largest edge weight. It is iterated $O(\log_2 N)$ time.
Since the operations (2)-(4) are done in O(m) time, it takes
  $O(m \log_2 N)$ time in total.

演習問題P12-1: 配列Qを用いたデータ構造を用いてダイクストラのアルゴリズムを実装せよ.

演習問題P12-2: 10頂点以上の重みつきグラフについてアルゴリズムの動作を確かめよ.

演習問題P12-3: ここで説明したスケーリングアルゴリズムによって正しく解が求まることを証明せよ.

Exercise P12-1: Implement the Dijkstra's algorithm using the data structure using an array Q.

Exercise P12-2: Verify behavior of the algorithm for a graph with at least 10 vertices.

Exercise P12-3: Prove that the scaling algorithm described here certainly finds a correct solution.