Randomized Graph Algorithms

Michael T. Goodrich
University of California, Irvine

Las Vegas and Monte Carlo Algorithms

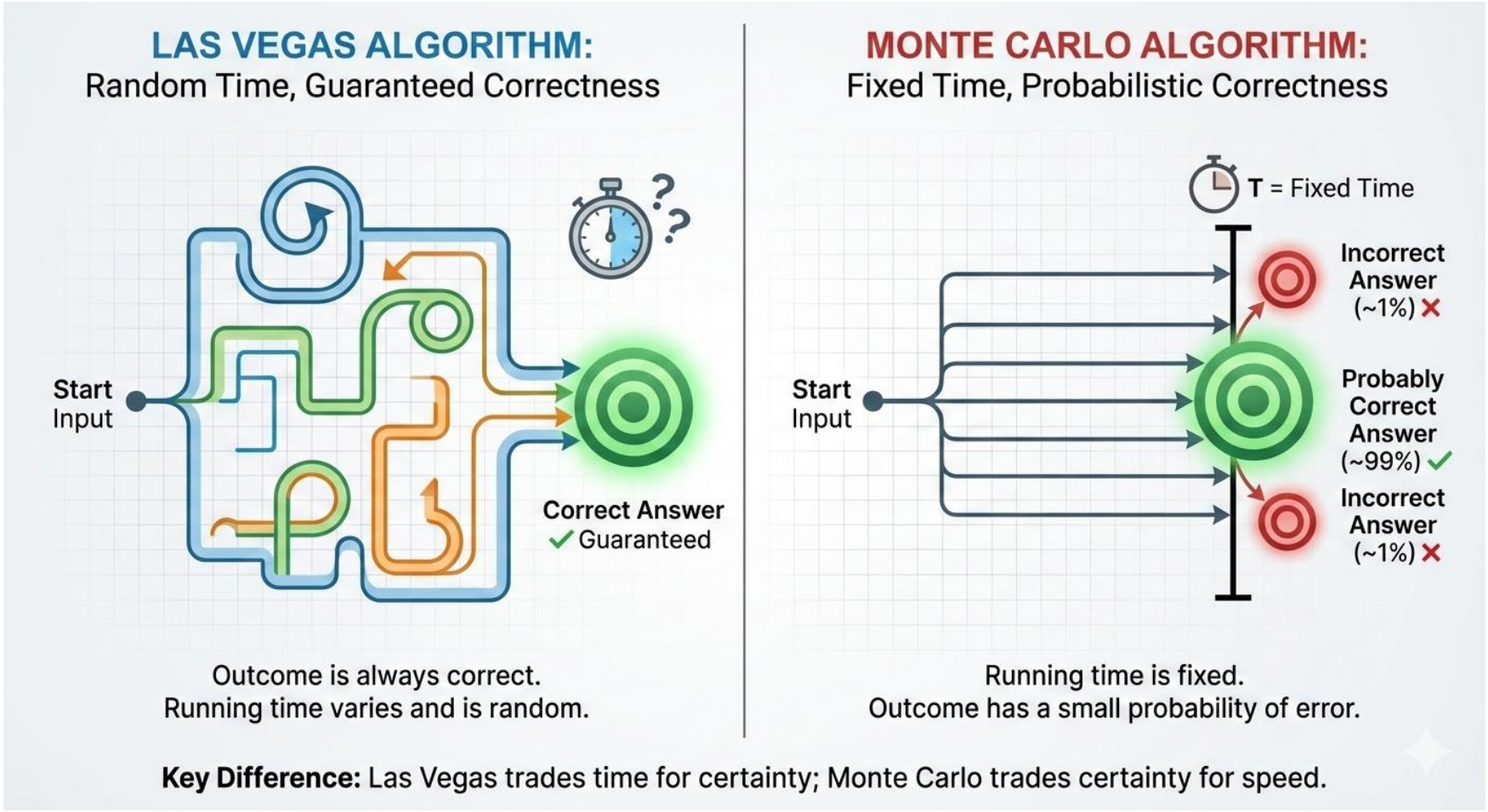
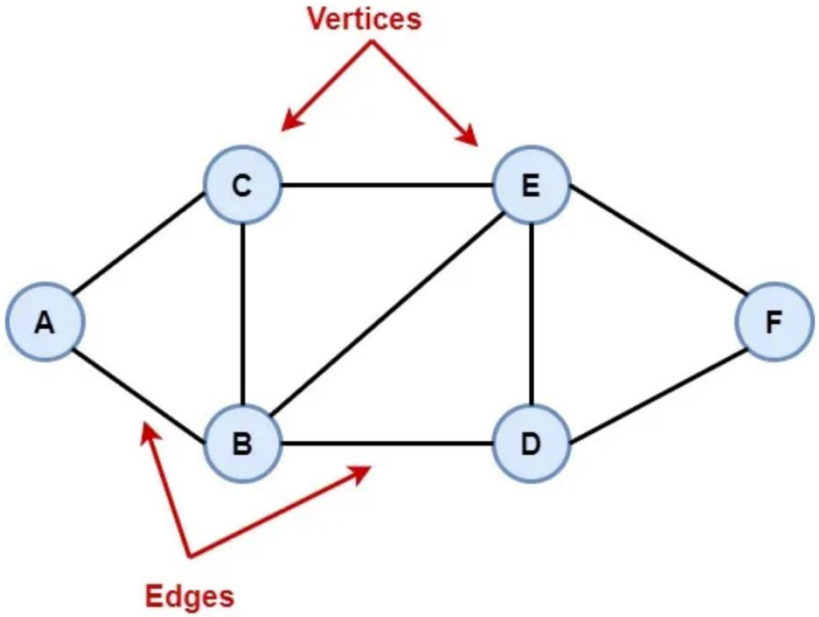


Image generated using Nano Banana Pro.

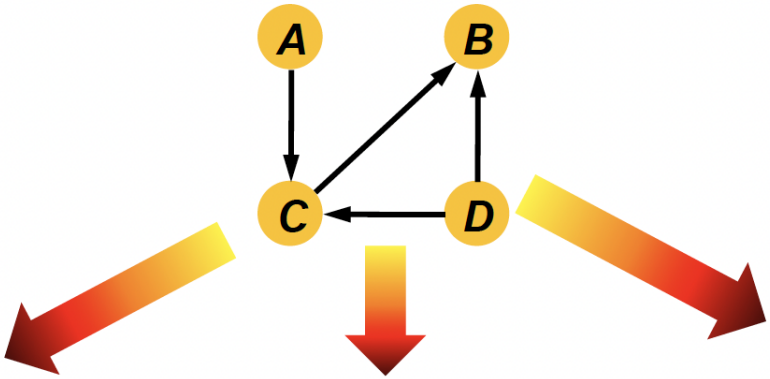
Graphs

- A **graph**, G=(V,E) is defined by a set of **vertices**, V, and a set of **edges**, E, which is a subset of V x V.
- The following graph is an **undirected** graph, in which the set of vertices is V = {A, B, C, D, E, F} and the set of edges is

E = {AB, AC, BC, BE, BD, CE, DE, DF, EF}.



Traditional Graph Representations



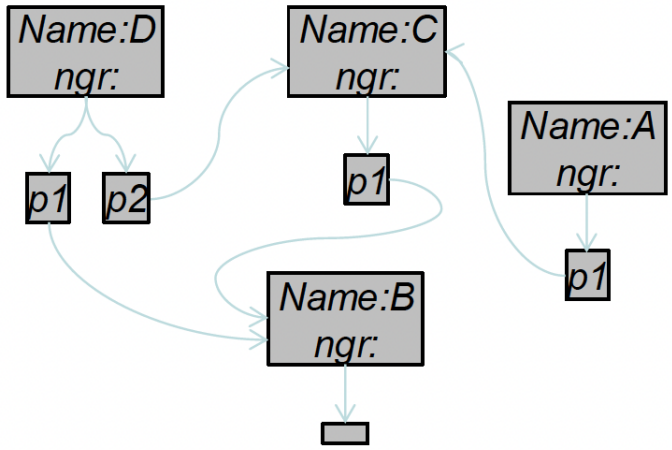
List or hash table of edges: (ordered) pairs of nodes

{ (A,C) , (C,B) , (D,B) , (D,C) }

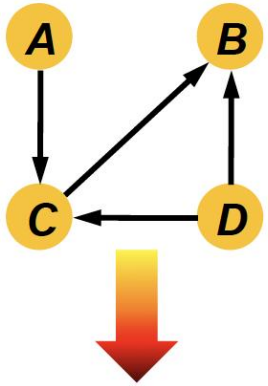
Adjacency Matrix

	A	B	C	D
A	0	0	1	0
B	0	0	0	0
C	0	1	0	0
D	0	1	1	0

Adjacency List



A Unified Graph Representation



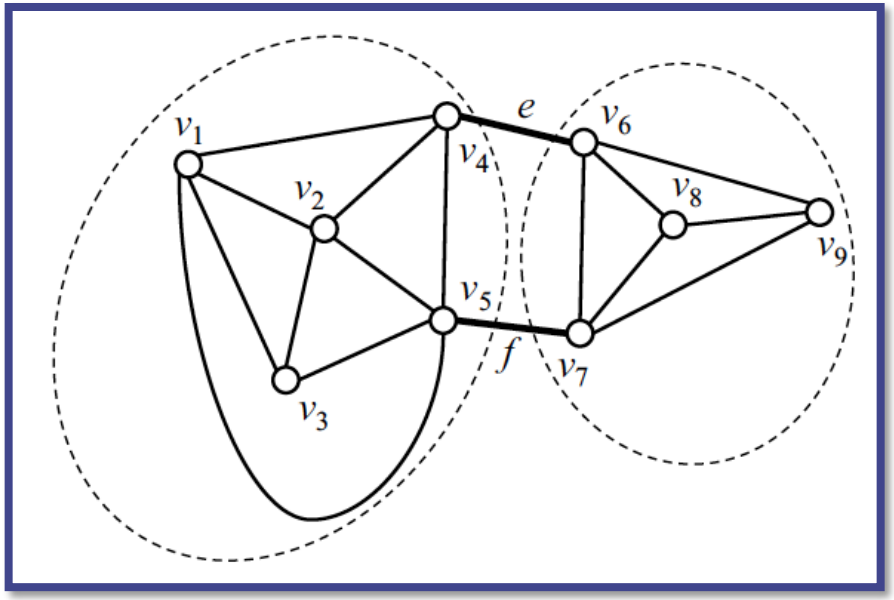
Adjacency Set

A: Adjacencies: {C}, Edges: {(A,C)}
B: Adjacencies: {}, Edges: {}
C: Adjacencies: {B}, Edges: {(C,B)}
D: Adjacencies: {B,C}, Edges: {(D,B), (D,C)}

- **Time for listAdjacent(v) is O(degree(v))**
- **Time for areAdjacent(v,w) is O(1) if sets have hash tables, like in Python**

Definition of Minimum Cut

- A cut, C, of a connected graph, G, is a subset of the edges of G whose removal disconnects G.
- That is, after removing all the edges of C, we can partition the vertices of G into two subsets, A, and B such that there are no edges between a vertex in A and a vertex in B.
- A **minimum cut** of G is a cut of smallest size among all cuts of G.

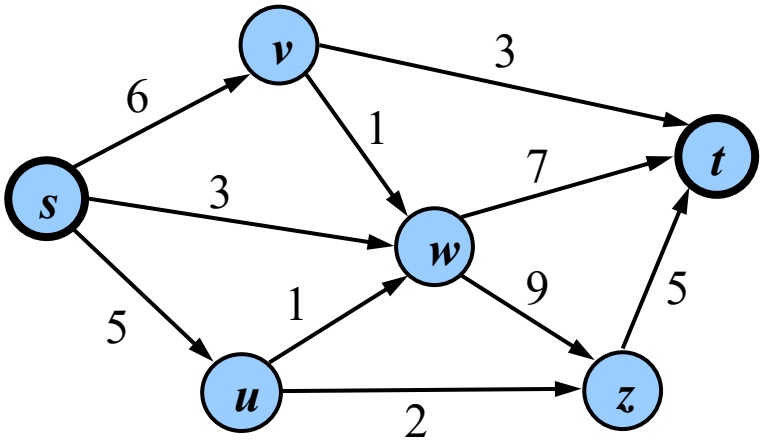


Applications

- In several applications, it is important to determine the size of a smallest cut of a graph.
  - For example, in a communications network, the failures of the edges of a cut prevents the communication between the nodes on the two sides of a cut.
  - Thus, the size of a minimum cut and the number of such cuts give an idea of the vulnerability of the network to edge failures.
- Small cuts are also important for the automatic classification of web content.
  - Namely, consider a collection of web pages and model them as a graph, where vertices correspond to pages and edges to links between pages.
  - The size of a minimum cut provides a measure of how much groups of pages have related content. Also, we can use minimum cuts to recursively partition the collection into clusters of related documents.

Flow Network

- A flow network (or just network) N consists of
  - A weighted digraph G with nonnegative integer edge weights, where the weight of an edge e is called the capacity $c(e)$ of e
  - Two distinguished vertices, s and t of G, called the source and sink, respectively, such that s has no incoming edges and t has no outgoing edges.
- Example:



Flow

- A flow f for a network N is is an assignment of an integer value $f(e)$ to each edge e that satisfies the following properties:

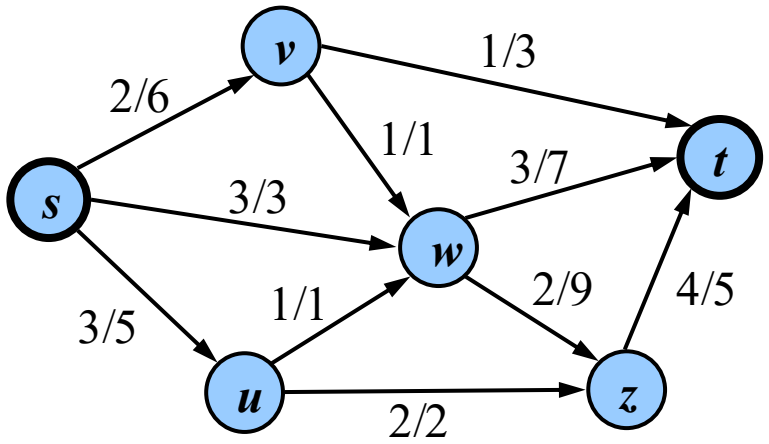
  Capacity Rule: For each edge e, $0 \le f(e) \le c(e)$

  Conservation Rule: For each vertex $v \neq s,t$ $\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e)$

  where $E^-(v)$ and $E^+(v)$ are the incoming and outgoing edges of v, resp.

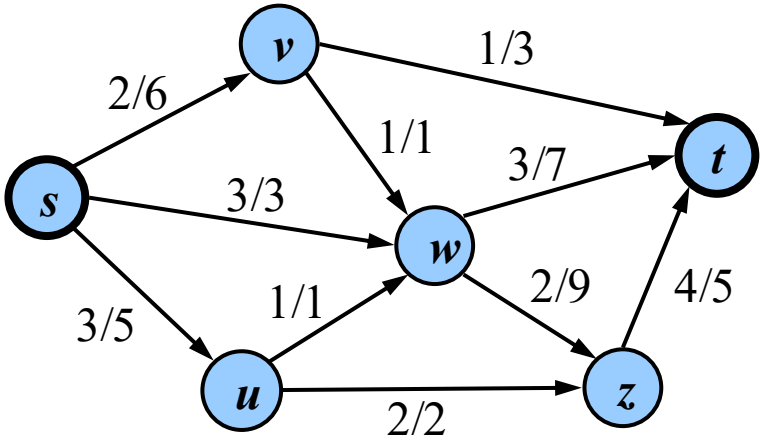
- The value of a flow f, denoted $|f|$, is the total flow from the source, which is the same as the total flow into the sink

- Example:

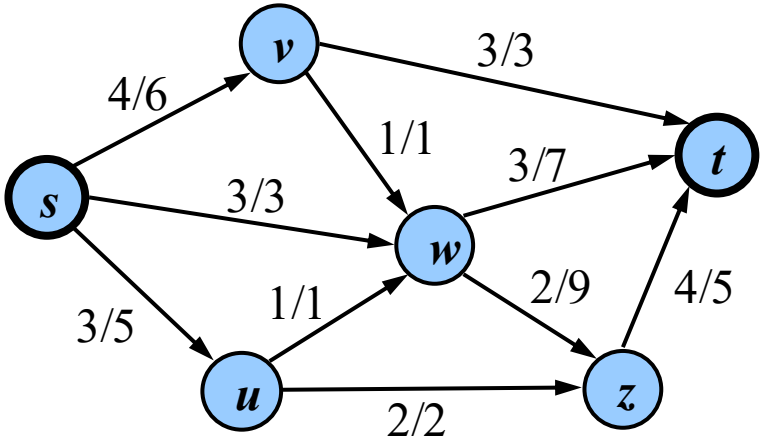


Maximum Flow

- A flow for a network N is said to be maximum if its value is the largest of all flows for N
- The maximum flow problem consists of finding a maximum flow for a given network N
- Applications
  - Hydraulic systems
  - Electrical circuits
  - Traffic movements
  - Freight transportation



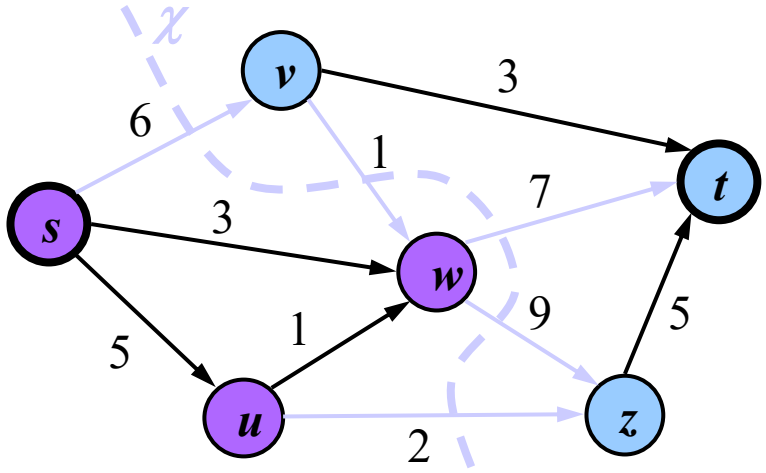
Flow of value 8 = 2 + 3 + 3 = 1 + 3 + 4

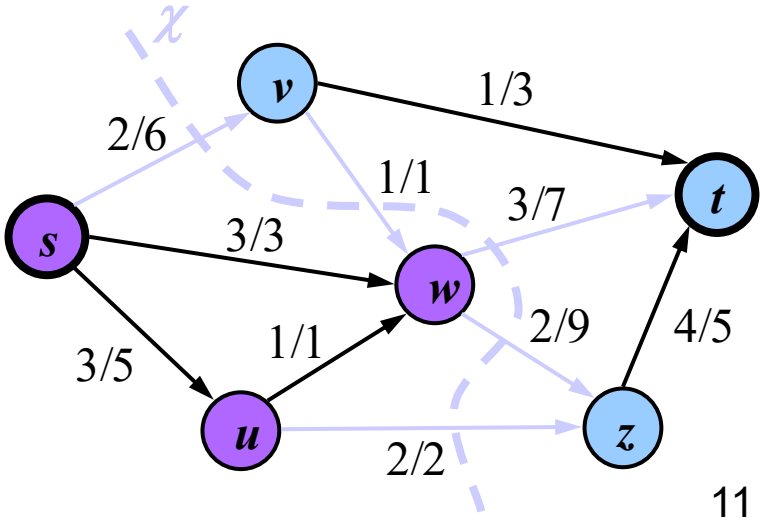


Maximum flow of value 10 = 4 + 3 + 3 = 3 + 3 + 4

Cut

- A cut of a network N with source s and sink t is a partition $\chi = (V_s, V_t)$ of the vertices of N such that $s \in V_s$ and $t \in V_t$
  - Forward edge of cut χ: origin in V_s and destination in V_t
  - Backward edge of cut χ: origin in V_t and destination in V_s
- Flow $f(\chi)$ across a cut χ: total flow of forward edges minus total flow of backward edges
- Capacity $c(\chi)$ of a cut χ: total capacity of forward edges
- Example:
  - $c(\chi) = 24$
  - $f(\chi) = 8$





Flows and Cuts

Lemma:

The flow $f(\chi)$ across any cut χ is equal to the flow value $|f|$

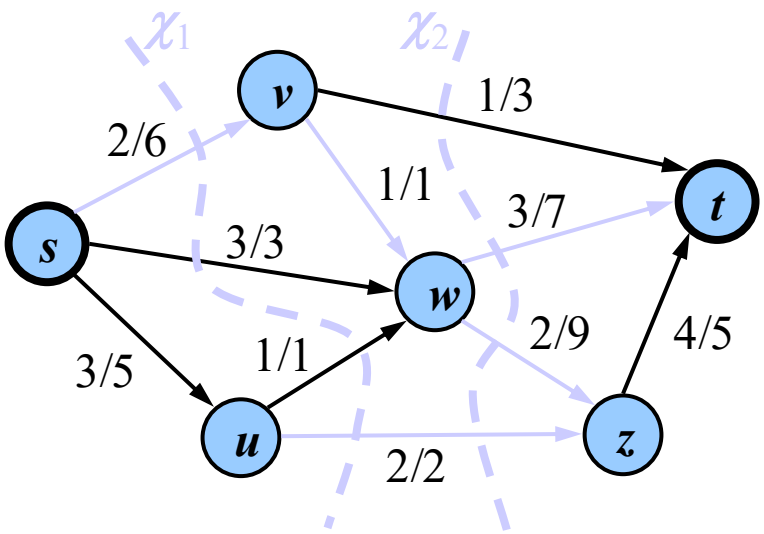
Lemma:

The flow $f(\chi)$ across a cut χ is less than or equal to the capacity $c(\chi)$ of the cut

Theorem:

The value of any flow is less than or equal to the capacity of any cut, i.e., for any flow f and any cut χ, we have

$$|f| \leq c(\chi)$$



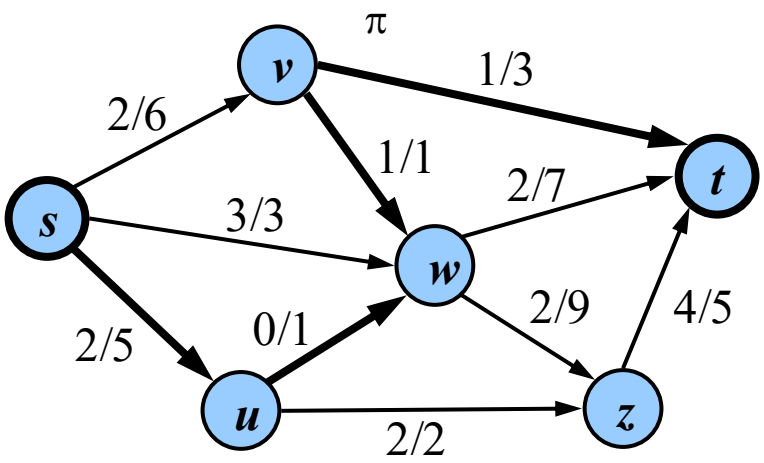
$c(\chi_1) = 12 = 6 + 3 + 1 + 2$

$c(\chi_2) = 21 = 3 + 7 + 9 + 2$

$|f| = 8$

Augmenting Path

- Consider a flow f for a network N
- Let e be an edge from u to v:
  - Residual capacity of e from u to v: $\Delta_f(u, v) = c(e) - f(e)$
  - Residual capacity of e from v to u: $\Delta_f(v, u) = f(e)$
- Let π be a path from s to t
  - The residual capacity $\Delta_f(\pi)$ of π is the smallest of the residual capacities of the edges of π in the direction from s to t
- A path π from s to t is an augmenting path if $\Delta_f(\pi) > 0$



$\Delta_f(s,u) = 3$
$\Delta_f(u,w) = 1$
$\Delta_f(w,v) = 1$
$\Delta_f(v,t) = 2$
$\Delta_f(\pi) = 1$
$|f| = 7$

Flow Augmentation

Lemma:

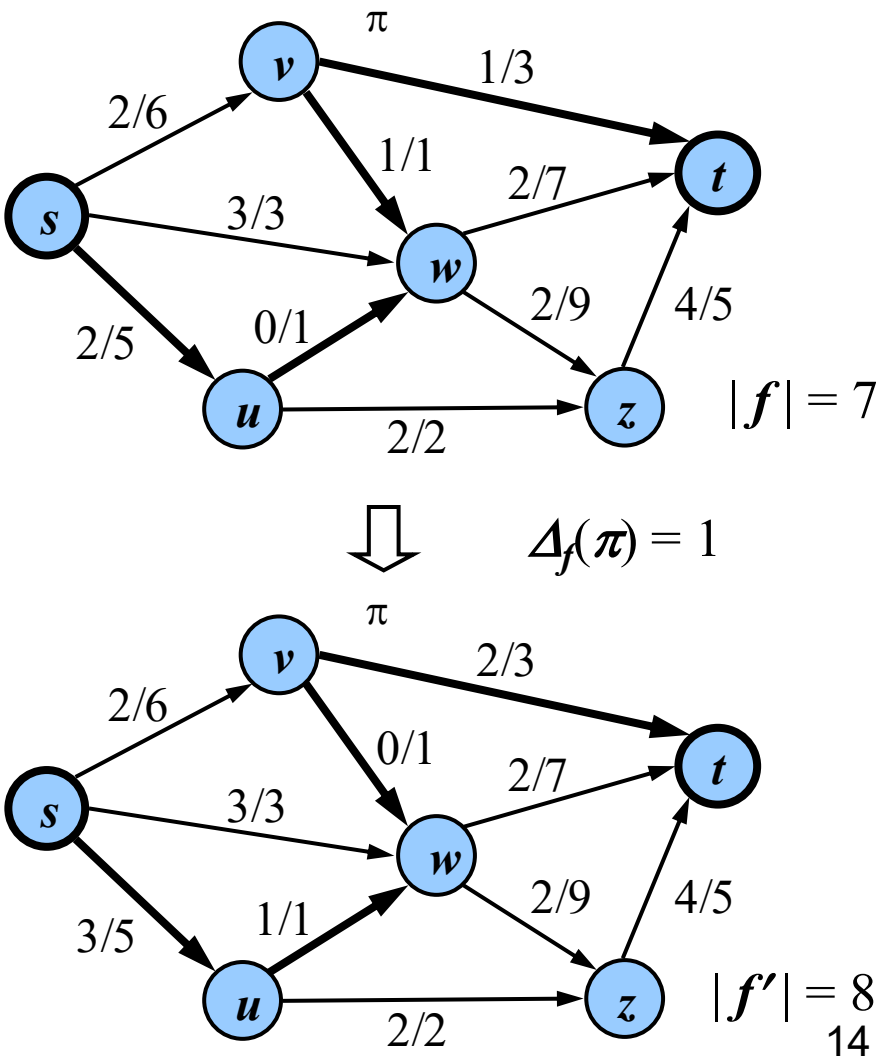
Let π be an augmenting path for flow f in network N. There exists a flow f' for N of value

$$|f'| = |f| + \Delta_f(\pi)$$

Proof:

We compute flow f' by modifying the flow on the edges of π

- Forward edge:
$$f'(e) = f(e) + \Delta_f(\pi)$$
- Backward edge:
$$f'(e) = f(e) - \Delta_f(\pi)$$



The Ford-Fulkerson Algorithm

- Initially, $f(e) = 0$ for each edge e
- Repeatedly
  - Search for an augmenting path π
  - Augment by $\Delta_f(\pi)$ the flow along the edges of π
- A specialization of DFS (or BFS) searches for an augmenting path
  - An edge e is traversed from u to v provided $\Delta_f(u, v) > 0$

Algorithm MaxFlowFordFulkerson(N):
Input: Flow network $N = (G, c, s, t)$
Output: A maximum flow f for N

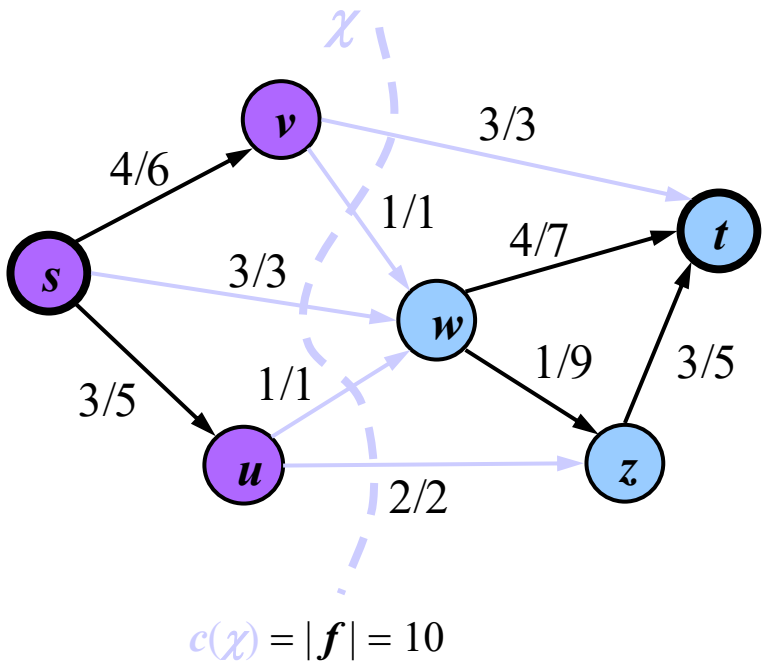
```
for each edge e ∈ N do
    f(e) ← 0
stop ← false
repeat
    traverse G starting at s to find an augmenting path for f
    if an augmenting path π exists then
        // Compute the residual capacity Δ_f(π) of π
        Δ ← +∞
        for each edge e ∈ π do
            if Δ_f(e) < Δ then
                Δ ← Δ_f(e)
        for each edge e ∈ π do   // push Δ = Δ_f(π) units along π
            if e is a forward edge then
                f(e) ← f(e) + Δ
            else
                f(e) ← f(e) − Δ        // e is a backward edge
    else
        stop ← true        // f is a maximum flow
until stop
```

Max-Flow and Min-Cut

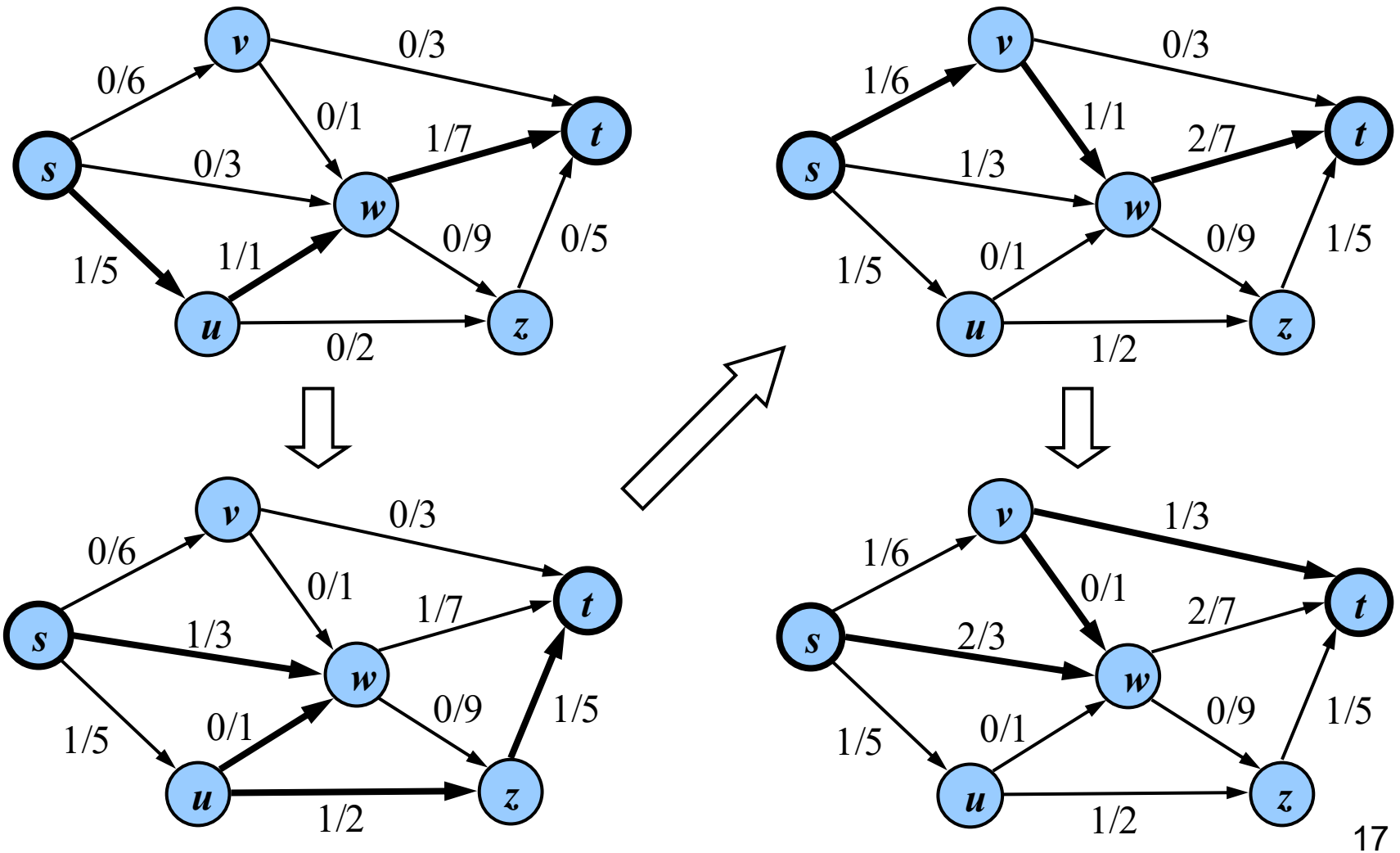
- Termination of Ford-Fulkerson's algorithm
  - There is no augmenting path from s to t with respect to the current flow f
- Define
  - V_s set of vertices reachable from s by augmenting paths
  - V_t set of remaining vertices
- Cut $\chi = (V_s, V_t)$ has capacity $c(\chi) = |f|$
  - Forward edge: $f(e) = c(e)$
  - Backward edge: $f(e) = 0$
- Thus, flow f has maximum value and cut χ has minimum capacity

Theorem:

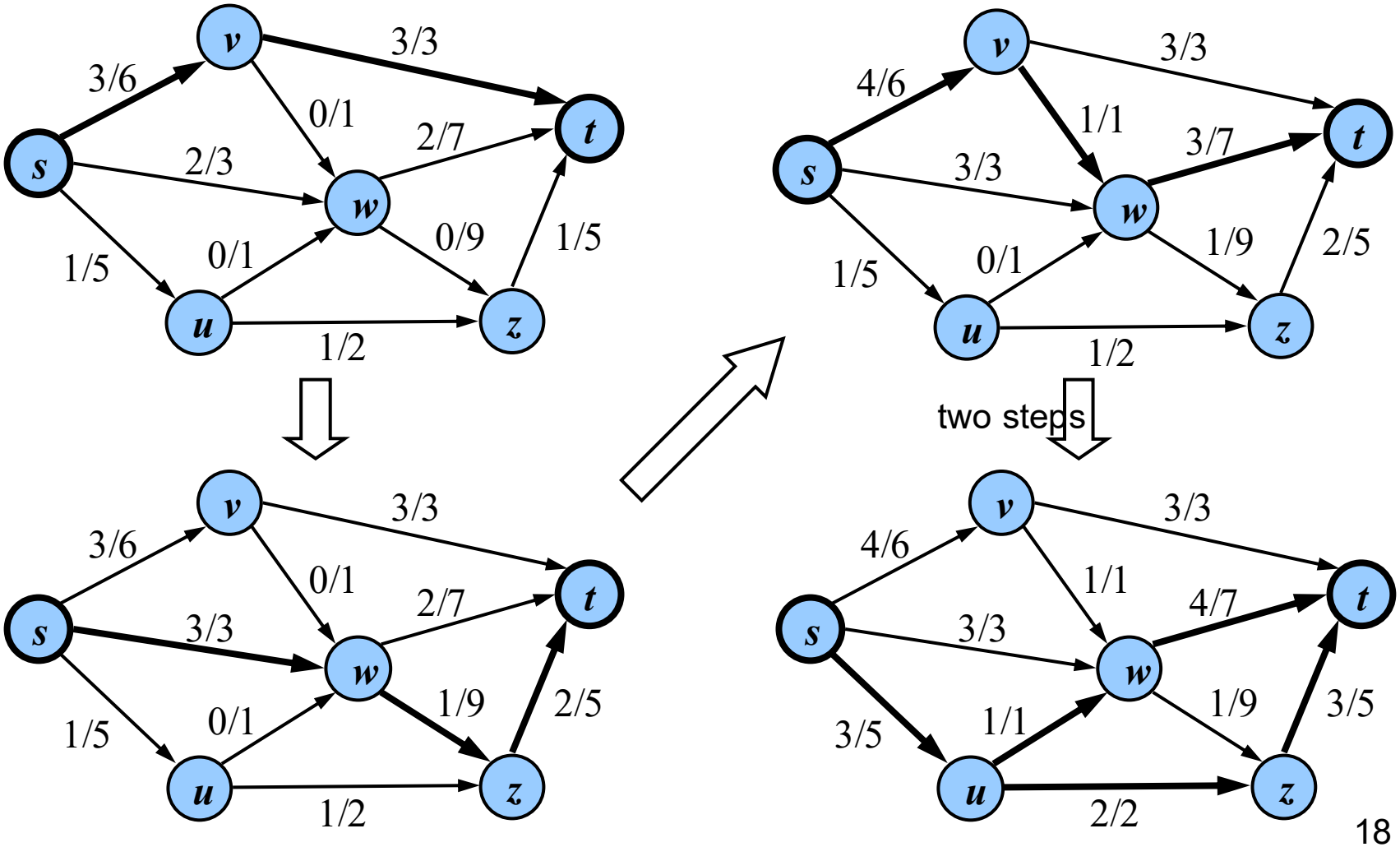
The value of a maximum flow is equal to the capacity of a minimum cut



Example (1)

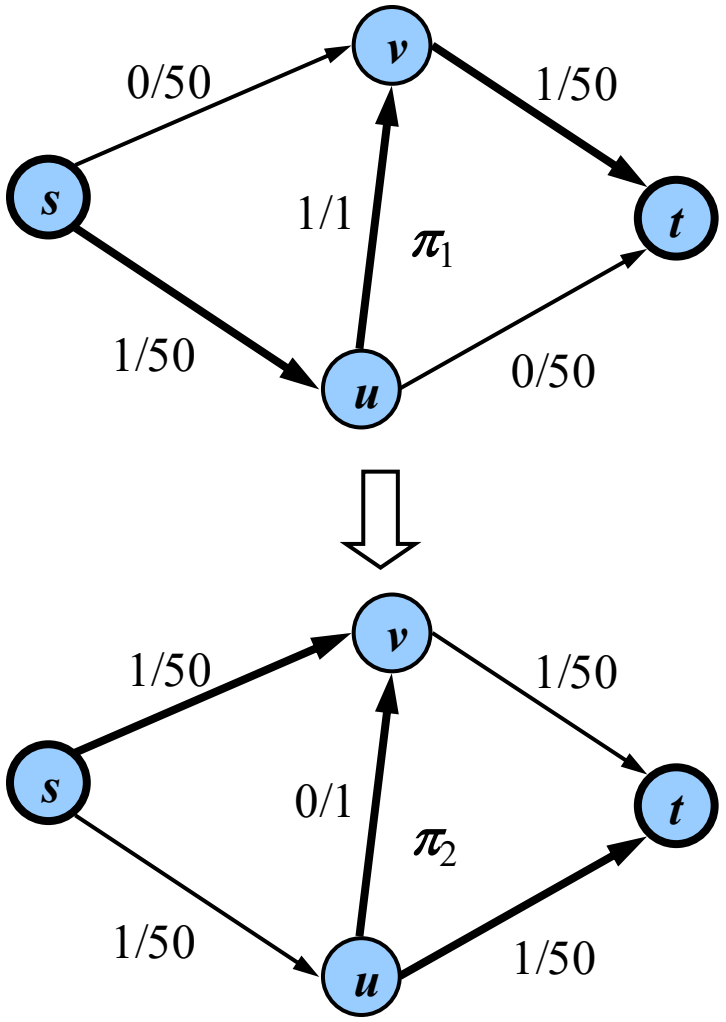

v
0/3
0/6
0/1
t
1/7
s
0/3
w
1/5
1/1
0/9
0/5
u
z
0/2
v
0/3
1/6
1/1
t
2/7
s
1/3
w
1/5
0/1
0/9
1/5
u
z
1/2
v
0/3
0/6
0/1
t
1/7
s
1/3
w
1/5
0/1
0/9
1/5
u
z
1/2
v
1/3
1/6
0/1
t
2/7
s
2/3
w
1/5
0/1
0/9
1/5
u
z
1/2

Example (2)


v
3/3
3/6
0/1
2/7
t
s
2/3
w
1/5
0/1
0/9
1/5
u
z
1/2
4/6
1/1
3/7
3/3
1/9
2/5
two steps
3/5
4/7
1/1
3/5
2/2

Analysis

- In the worst case, Ford-Fulkerson's algorithm performs $|f^*|$ flow augmentations, where f^* is a maximum flow
- Example
  - The augmenting paths found alternate between π_1 and π_2
  - The algorithm performs 100 augmentations
- Finding an augmenting path and augmenting the flow takes $O(n + m)$ time
- The running time of Ford-Fulkerson's algorithm is $O(|f^*|(n + m))$



Min-Cut Relationship to Max Flow

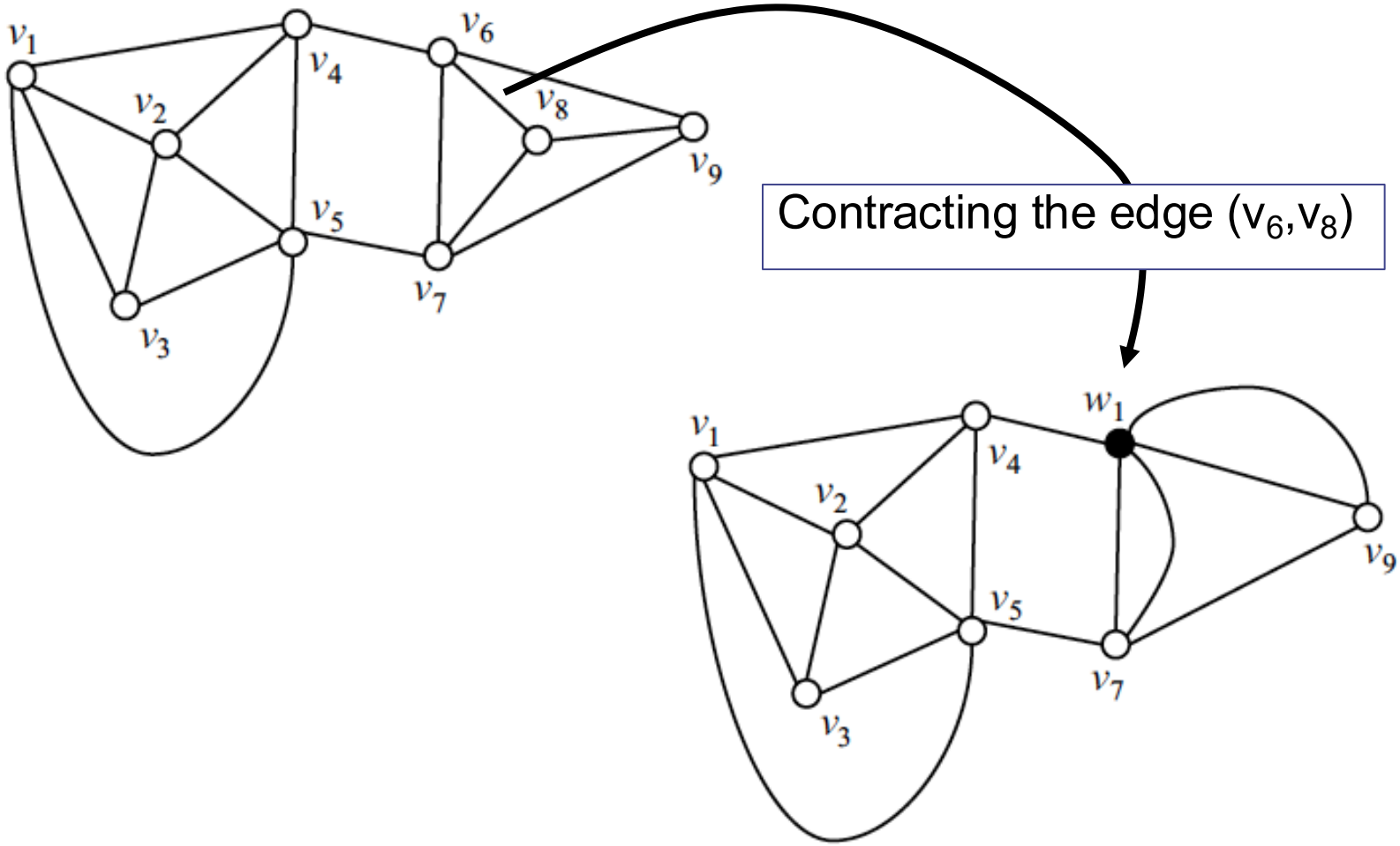
- The min-cut/max-flow theorem states that, given a pair of vertices, s and t, we can compute a minimum cut in polynomial time such that s is on one side of the cut and t is on the other.
- In this case, however, we want the minimum cut over all possible cuts.
- Nevertheless, we can compute such an overall minimum cut by O(n) calls to an (s,t)-min-cut-max-flow algorithm.
  - How?
- Here, we show how to design a simple, efficient Monte Carlo randomized algorithm that succeeds with high probability without using min-cut-max-flow.

Contracting Edges

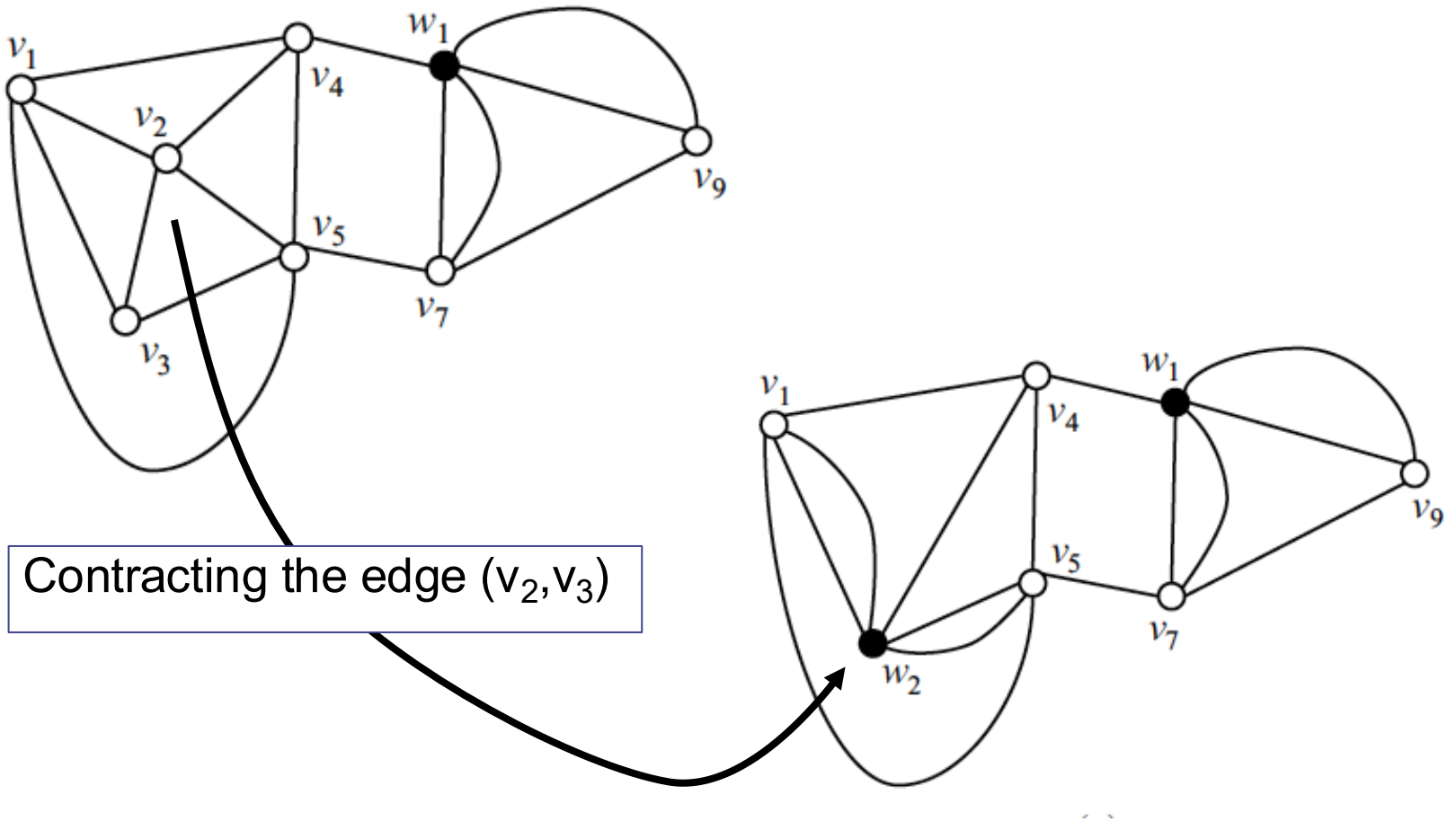
- The simple randomized algorithm algorithm repeatedly performs contraction operations on the graph.
- Let G be a graph with n vertices, where we allow G to have parallel edges. We denote with (v,w) any edge with endpoints v and w.
- The contraction of an edge e of G with endpoints u and v consists of the following steps that yield a new graph with n − 1 vertices, denoted G/e:

1. Remove edge e and any other edge between its endpoints, u and v.
2. Create a new vertex, w.
3. For every edge, f, incident on u, detach f from u and attach it to w. Formally speaking, let z be the other endpoint of f. Change the endpoints of f to be z and w.
4. For every edge, f, incident on v, detach f from v and attach it to w.

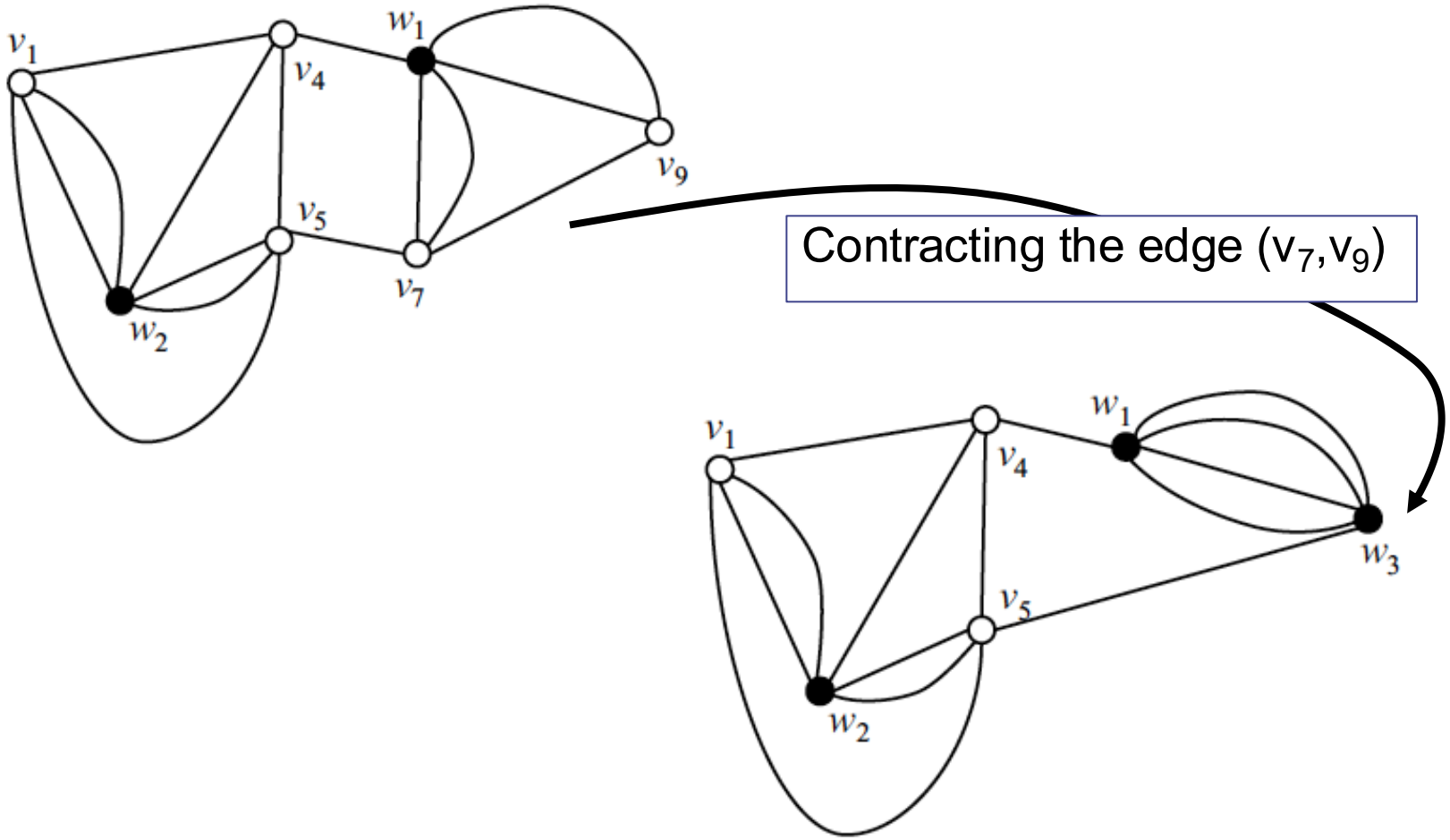
Contracting Edges, An Example



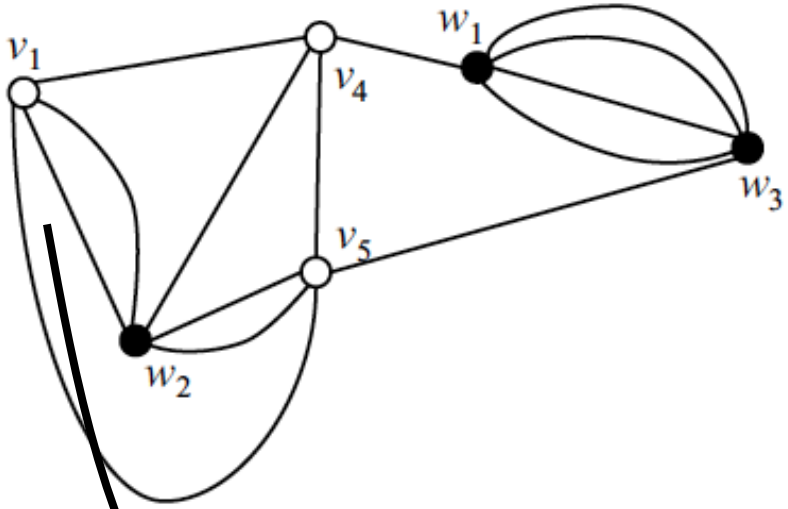
Contracting Edges, An Example



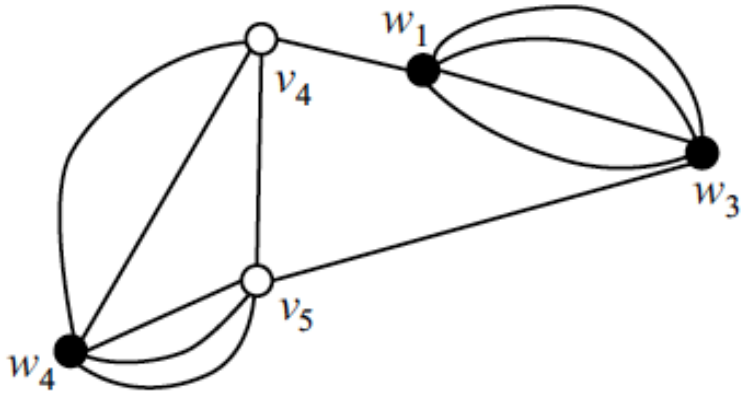
Contracting Edges, An Example



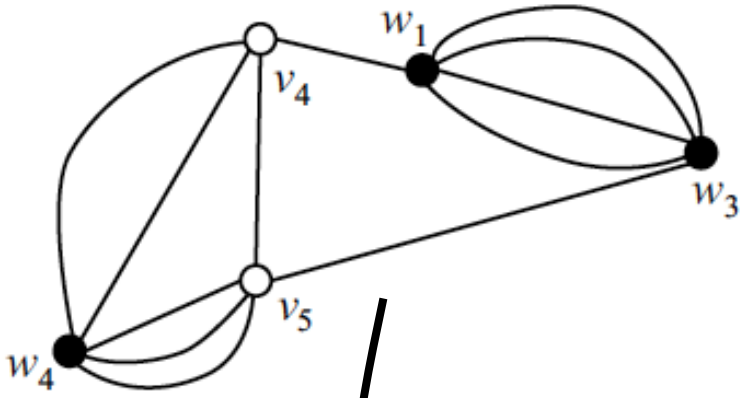
Contracting Edges, An Example



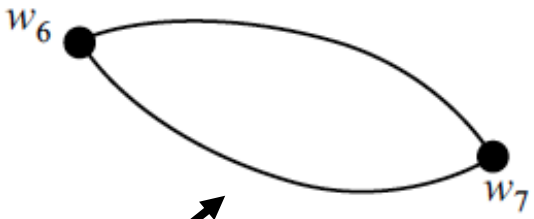
Contracting the edge (v_1, w_2)



Contracting Edges, An Example



Contracting the edge (v_4,v_5), yielding w_5, contracting (w_4,w_5), yielding w_6, and contracting (w_1,w_3), yielding w_7.



This final graph has the same minimum cut as the original graph.

Karger's Algorithm

- A simple min-cut algorithm, which succeeds with high probability is to repeat the following procedure multiple times, keeping track of the smallest cut that it ever finds:

Algorithm ContractGraph(G):

Input: An undirected graph, G, with n vertices

Output: A cut of G that has minimum size with probability at least $\frac{2}{n(n-1)}$

while G has more than 2 edges **do**

pick a random edge, e, of G

contract edge e

$G \leftarrow G/e$

return the edges of G

Analysis

- Let G be a graph with n vertices and m edges, and let C be a given minimum cut of G. We will evaluate the probability that the algorithm returns the cut C.
- Since G may have other minimum cuts, this probability is a lower bound on the success probability of the algorithm.
- Let G_i be the graph obtained after i contractions performed by the algorithm and let m_i be the number of edges of G_i. Assume that G_{i-1} contains all the edges of C. The probability that G_i also contains all the edges of C is equal to $1 - k/m_{i-1}$ since we contract any given edge of C with probability $1/m_{i-1}$ and C has k edges.
- Thus, the probability, P, that the algorithm returns cut C is

$$P = \prod_{i=0,\ldots,n-3} \left(1 - \frac{k}{m_i}\right).$$

Analysis, part 2

- Since k is the size of the minimum cut of each graph G_i, we have that each vertex of G_i has degree at least k.
- Thus, we obtain the following lower bound on m_i, the number of edges of G_i:

$$m_i \geq \frac{k(n-i)}{2}, \quad \text{for } i = 0, 1, \ldots, n-3 .$$

- We can then use these bounds to derive a lower bound for P.

Analysis, part 3

- The following bound implies that P is at least proportional to $1/n^2$:

$$
\begin{aligned}
P &= \prod_{i=0}^{n-3}\left(1-\frac{k}{m_i}\right) \\
&\geq \prod_{i=0}^{n-3}\left(1-\frac{2k}{k(n-i)}\right) \\
&= \prod_{i=0}^{n-3}\left(\frac{n-i-2}{n-i}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\left(\frac{n-5}{n-3}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)} \\
&= \frac{1}{\binom{n}{2}}
\end{aligned}
$$

Analysis, part 4

- We can boost the probability by running the algorithm multiple times. In particular, if we run the
- algorithm for t*n-choose-2 rounds, where t is a positive integer, we have that at least one round returns cut C with probability

$$P(t) = 1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^{t\binom{n}{2}}.$$

- By a well-known property of the mathematical constant **e**, the base of the natural logarithm, ln, we obtain $P(t) > 1 - 1/\mathbf{e}^t$.
- If we choose $t = c \ln n$, where c is a constant, then the success probability is at least $1 - 1/n^c$.

Running Time Analysis

- A contraction operation can be executed in O(n) time, if the graph is represented using an adjacency list.
- Thus, the running time of one round is $O(n^2)$.
- We repeat the algorithm $O(n^2 \log n)$ times.
- Thus, the total running time is $O(n^4 \log n)$.
- This can be improved to $O(n^2 \log^3 n)$…

Improved Algorithm

Algorithm RecursiveContractGraph(G, n):

Input: An undirected graph, G, with n vertices

Output: A cut of G that has minimum size with probability at least $\frac{1}{\log n}$

if G has at most 6 vertices **then**

  return ContractGraph$(G, 2)$

else

  $G_1 \leftarrow$ ContractGraph $\left(G, \frac{n}{\sqrt{2}}\right)$

  $C_1 \leftarrow$ RecursiveContractGraph $\left(G_1, \frac{n}{\sqrt{2}}\right)$

  $G_2 \leftarrow$ ContractGraph $\left(G, \frac{n}{\sqrt{2}}\right)$

  $C_2 \leftarrow$ RecursiveContractGraph $\left(G_2, \frac{n}{\sqrt{2}}\right)$

  if $|C_1| \leq |C_2|$ **then**

    return C_1

  else

    return C_2

- ContractGraph(G, r) is a variation of ContractGraph(G) that performs n − r contractions and returns the resulting graph, which has r vertices

Time Analysis of Improved Algorithm

- RecursiveContractGraph makes two recursive calls.
- After each call, the size of the input graph is a constant fraction, $1/\sqrt{2}$, of the size before the call.
- Thus, the execution of the algorithm can be modeled by a binary recursion tree with depth 2 log n.
- The number of contractions performed in a call, excluding those within its recursive calls, is proportional to the number of vertices of the graph.
- Since each contraction takes linear time in the number of vertices, the work done within a call, excluding its recursive calls, is quadratic in the number of vertices.
- Thus, the total work done at level i in the recursion tree is

$$2^i \left(\frac{n}{\sqrt{2}^i} \right)^2 = n^2.$$

(So total time is $O(n^2 \log n)$

Probability Analysis

- Referring to the recursion tree, we say that a node is **safe** if its associated graph contains cut C.
- The recursive call at a safe node succeeds in returning C if one of the following events occurs:

- graph G_1 obtained from ContractGraph contains cut C and the recursive call returns C;
- graph G_2 obtained from ContractGraph contains cut C and the recursive call returns C.

Probability Analysis, Part 2

- Define P(i) as the probability that a safe node at height i of the recursion tree has a safe leaf in its subtree. Thus, P(2 log n) denotes the probability that algorithm RecursiveContractGraph succeeds in returning C.
- Each of graphs G1 and G2 returned by ContractGraph contains C with probability 1/2. Thus,

$$P(i+1) \geq \frac{1}{2}P(i) + \frac{1}{2}P(i) - \left(\frac{1}{2}P(i)\right)^2 = P(i) - \frac{1}{4}P(i)^2.$$

The base case of the recurrence is for a leaf, which has height 0 and is associated with a graph with at most 6 vertices:

$$P(0) \geq \frac{1}{\binom{6}{2}} = \frac{1}{15}.$$

- This implies P(i) is c/log n, for come constant c; hence, to get high probability takes $O(n^2 \log^3 n)$ time.