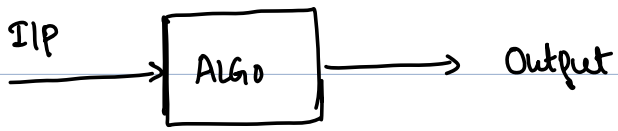
Randomized Algorithms

Deterministic Algorithms :

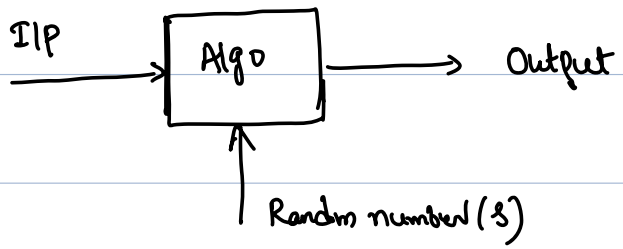


- Algo always behaves the same way given the same input
- The input completely determines the sequence of computations performed by the algorithm.

- Running-time (worst case) :

$$T_{\text{Worst-case}}(n) = \max_{|X|=n} T(X)$$

Randomized Algorithms:



(makes random choice during execution)

↓

It can make decisions based on the output of coinflip.

The same randomized algorithm, given the same input multiple times

- run for a different # of steps
- Produce diff outputs.

Running time: (Worst case expected)

We compute the expected running time for each i/p and then choose the maximum over all inputs of a certain size

$$T_{\text{worst-case-expected}} = \max_{|X|=n} E[T(X)]$$

Examples

```
def f(x):
    y = Bernoulli(0.5)
    If (y==0):
        while (x>0):
            Print("Hello")
            x = x - 1
    return x+y
```

$\Pr[y=0] = \Pr[y=1] = \frac{1}{2}$

Eg: $x = 3$

→ the output can vary (on the same i/p)

→ the running time can vary

(a) @ How should we measure its correctness?

(b) " " " " " running time?

Example

Types of Randomized algorithms

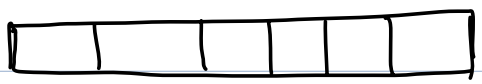
Ip: An array A with n elements, half of the array contains 0's, the other half contains 1's

Qtn:- find an index that contains a 1

n is even here

(a) @ Think of a deterministic algorithm?

(b) What is the worst case running time.



Algo-I

Repeat

k = RandInt(n)

If A[k] == 1

Return k

↑ selects an element from {1, -- n} uniformly

Algo-II

Repeat 300 times

k = RandInt(n)

If A[k] = 1

Return k

return "fail"

Let us analyze these algorithms one by one

Algo - I

```
Repeat
    k = RandInt(n)
    If A[k] == 1
        Return k
```

$Pr[\text{Failure}] = 0$

Worst Case running time : Can not bound

X : # of iterations, Clearly X is a random variable

Can take values from $1, 2, \ldots$

$$E(X) = \sum_{i=1}^{\infty} i \, Pr[X = i]$$

$$= \sum_{i=1}^{\infty} \frac{i}{2^i} = \lim_{n \to \infty} \sum_{i=1}^{n} \frac{i}{2^i} = 2.$$

↳ Constant

This is called a **Las Vegas Algorithm**

(gambles with time but not correctness)

Algo - II

Repeat 300 times

k = RandInt(n)

If $A[k] = 1$

Return k

return "fail"

$Pr[\text{Failure}] = \frac{1}{2^{300}}$, $Pr[\text{Success}] = 1 - \frac{1}{2^{300}}$

Worst Case running time : $O(1)$

This is called a **Monte Carlo algorithm**

(**gambles with Correctness** but not time)

↓

ie, the algorithm does not guarantee success but the **time is bounded**.

Las Vegas Algorithm :

① It is a randomized algorithm that always gives correct answer.

② The running time is a random variable, whose expectation is bounded (by a polynomial)

Eg Randomized Quick Sort (Refer to Coreman for more details)
↓
selects pivot elements uniformly at random.

Monte Carlo algorithms :

① It is a randomized algorithm whose output may be incorrect with certain (small) probability.

② Runtime independent of the randomness.

Eg Mincut (cut of minimum cardinality)

Advantages of Randomized Algorithms

- Simplicity
- Performance

For many problems a randomized algorithm is the simplest & the fastest.

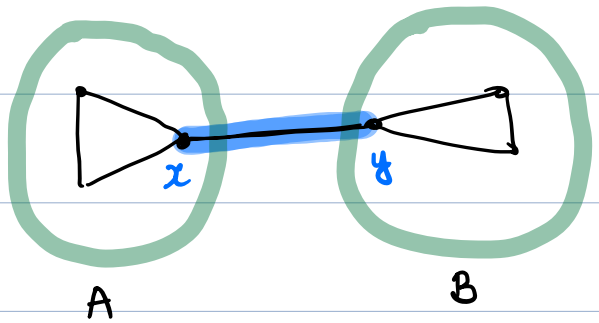
Minimum cut (or) Min cut

[Ref: KT: Algorithm Design]

Given an undirected graph G, a cut of G is a partition of V(G) into two non-empty subsets A and B.

The size of a cut (A,B) is the number of edges with one end in A and the other in B.

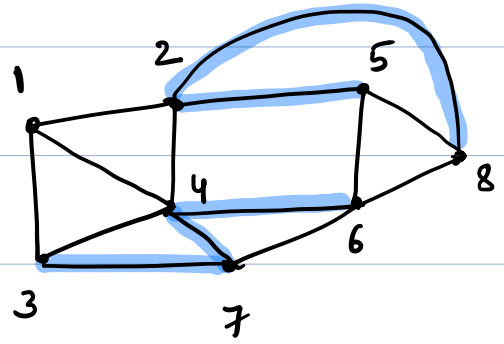
A minimum cut (or min-cut) is a cut of minimum size. [Some times its called global min-cut]



Size of the cut(A,B) is 1

Edge xy is called cut-edge.

Eg:



$A = \{1, 2, 3, 4\}$ $B = \{5, 6, 7, 8\}$

Size of cut (A, B) is 5.

However mincut size $= 3$

True/False : Size of min-cut $\leq \delta(G)$

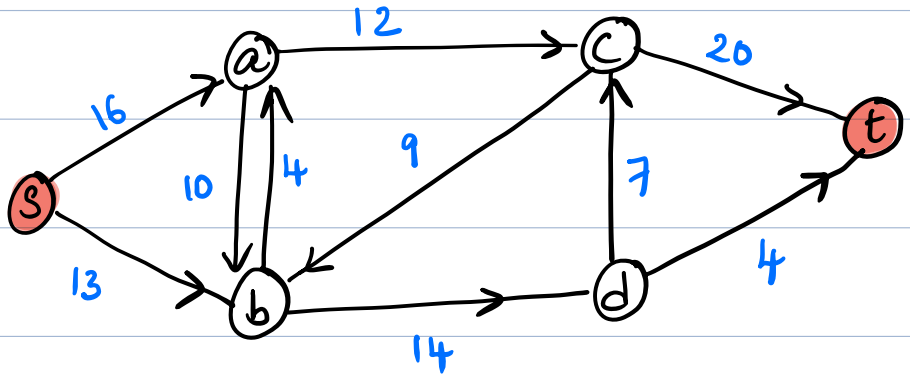
↓

mindegree

When two terminal nodes are given, they are typically referred as the Source and the Sink.

In a directed, weighted flow network, the minimum cut separates Source and Sink Veetices and minimizes the total weight on the edges that are directed from the Source side of the cut to the Sink side of the Cut.

Warmup:



Q: Find minimum s-t cut in the above graph

Ans $A = \{s, a, b, d\}$, $B = \{c, t\}$

(A, B) is a min cut and its capacity is

$12 + 7 + 4 = 23$

↓ ↓ ↓

ac dc dt

Algorithm [Based on flow networks]

We know an algorithm to compute minimum S-t cut in directed graphs using Ford-Fulkerson algorithm.

So given an undirected graph G, we transform it to directed graph with a sink and a source.

Replace every undirected edge $e = uv \in E(G)$ with two directed edges uv and vu each of capacity one. Let G' denote the resulting directed graph.

Pick two arbitrary nodes $s, t \in V(G)$ and find minimum s-t cut in G'

Claim: (A, B) is a min-cut in G' then (A, B) is also a min-cut in in G among all those that seperate s from t.

As we are interested find a min-cut in G,

Any min-cut in G must separate s from Something,

Since both sides A and B are non-empty and s belongs to only one of them.

So we fix any $s \in V(G)$ and compute the minimum s-t cut in G' for every other node $t \in V(G) - \{s\}$.

The best among these $n-1$ min-cuts in G', is the min-cut of G.

Running time: $O(n) *$ time to find min s-t cut in directed graphs

Edge Contraction

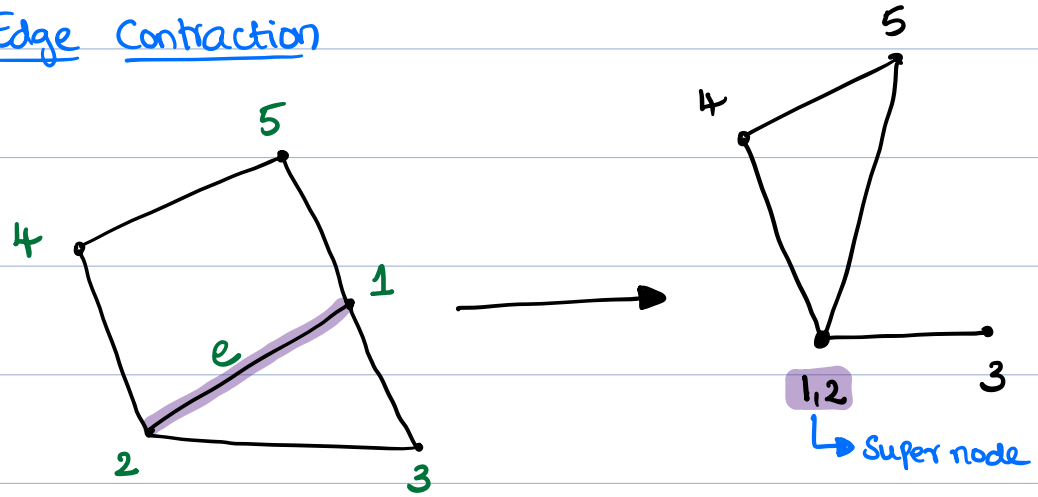


Fig: The effect of contracting edge $e = 12$ is shown

Def:- Let $e = uv$ be an edge in a graph G. Delete the vertices u and v. Add a new vertex w to $G - \{u, v\}$ and join w to all those vertices in $V - \{u, v\}$ to which u or v is adjacent in G. This operation is called the **EDGE CONTRACTION** of e.

Randomized Algorithm for Min-cut

The algorithm we study is the simplest one, and there are algorithms for min-cut Problem having better running time.

Description of the Algorithm

The input graph may have multiple edges b/w the same pair of nodes.

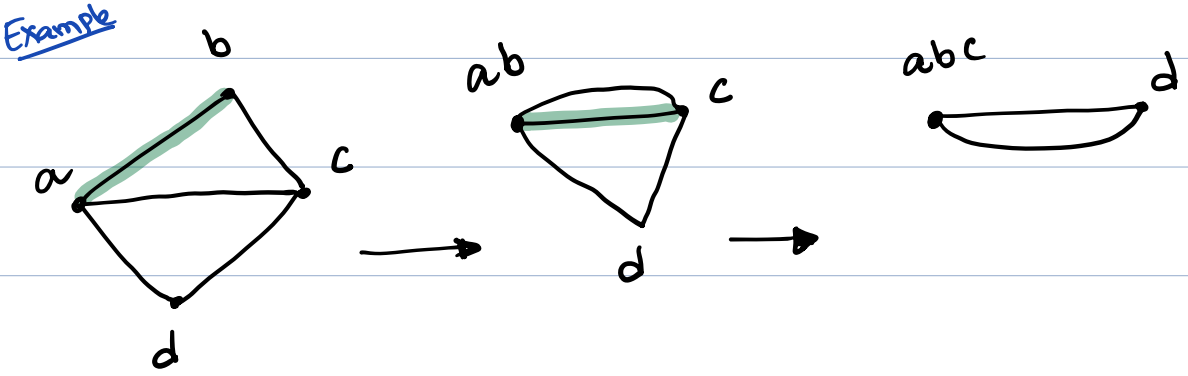
The Algorithm begins by choosing an edge $e = uv$ of G uniformly at random and contracts it. this means we produce a new graph G' in which u & v are identified into a single node w. edges b/w the vertices that are merged are removed. (ie, no self-loops are created).

The contraction algorithm then continues recursively on G', choosing an edge uniformly at random and contracting it.

The algorithm continues the contraction process until only two vertices remain. → Super nodes

At this point, the set of edges blw these two vertices is a cut in G and is output as a candidate min-cut.

Example



Before doing analysis

Q: When do the edge contraction algorithm fails to find min-cut.

A If it choose a cut-edge at any stage during the algorithm execution, then the algorithm is not going to find the mincut.

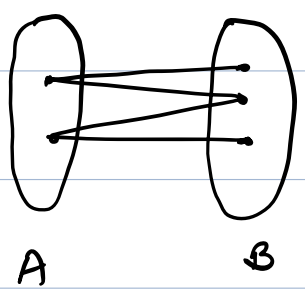
Analysis of the Algorithm

As the algorithm is making random choices, so there is some probability that it will succeed in finding a global min-cut and some probability that it won't.

Now we analyze the success probability.

Let (A, B) be a (global) mincut of G of size k.

Goal: Compute the probability that Algorithm o/p the cut (A, B).



Let F be the set of k edges with one end in A and the other in B.

We give a lower bound on the probability that the Contraction algorithm returns the cut (A, B).

at any time, if the algorithm contracts an edge of F, then (A, B) could not be returned as o/p by the algorithm.

On the other hand if an edge not in F is contracted, then there is still chance that (A, B) could be returned.

Observe that, if any node v, has degree less than k then cut $(\{v\}, V-\{v\})$ would have size less than k, contradicting our assumption that (A, B) is a global mincut.

$\therefore$ Every node in G has degree at least k.

$$\therefore 2|E(G)| = \sum_{v} \deg(v) \geq nk$$

$$|E(G)| \geq \frac{1}{2} kn$$

∴ The Probability that an edge in F is Contracted in the first step is

$$\frac{k}{m} \leq \frac{k}{\frac{kn}{2}} = \frac{2}{n}$$

m : # of edges

Now Consider the Situation after j iterations, when there are $n-j$ Super nodes in the Current graph G', and Suppose that no edge in F has been contracted yet.

Every cut of G' is a cut of G, and so there are at least k edges incident to every Super node of G'.

$$\therefore |E(G')| \geq \frac{1}{2}k(n-j)$$

∴ the Probability that an edge of F is Contracted in the next iteration $j+1$ is at most

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}$$

The cut (A, B) will actually be returned by the algorithm if no edge of F is contracted in any of iterations $1, 2, \ldots n-2$.

Let $\mathcal{E}_j$ denote the event that an edge of F is not contracted in iteration j,

then

$$\Pr[\mathcal{E}_1] \geq 1 - \frac{2}{n}$$

$$\Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \ldots \cap \mathcal{E}_j] \geq 1 - \frac{2}{n-j}$$

We are interested in

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \ldots \mathcal{E}_{n-2}]$$

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-2}]$$

$$= \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 | \mathcal{E}_1] \cdots \Pr[\mathcal{E}_{n-2} | \mathcal{E}_1 \cap \cdots \mathcal{E}_{n-3}]$$

$$\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{3}\right)$$

$$= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right)$$

$$= \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$$

∴ Single run of the algorithm fails to find a global min cut with probability at most $1 - \frac{1}{\binom{n}{2}}$.

We repeatedly run the algorithm, with indep random choices and taking the best cut we find.

if we run the algorithm $\binom{n}{2}$ times then the Pbty that we fail to find a global min-cut in any run is at most

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}} \leq \frac{1}{e}$$

∴ if we run the algorithm $\binom{n}{2} \ln n$ times then the Probability we fail to find a global min-cut is at most

$$e^{-\ln n} = \frac{1}{n}$$

The overall running time required to get a high Probability of success is Polynomial in n.

The running time of our contraction algorithm is fairly large compared with the best network flow techniques, as we perform $\Theta(n^2)$ independent runs and each takes at least $\Omega(m)$ time.

[As I said earlier this not the best, so running time can be improved considerably using some clever optimization tricks]