by

# 1

# Randomized Graph Algorithms: Techniques and Analysis

Surender Baswana
*Department of Computer Science and Engineering, IIT Kanpur, 208016, India.*

Sandeep Sen
*Department of Computer Science and Engineering, IIT Delhi, New Delhi 110016, India.*



## 1.1 Introduction

Although the first significant application of randomized techniques in algorithm design happened in mid 1970s in the celebrated primality testing algorithms of Miller-Rabin and Solovay-Strassen, its impact in graph algorithms was felt more than a decade later. Arguably, the first non-trivial applications were in the area of parallel graph algorithms like connectivity (Gazit [18]), Depth-First Search (Aggarwal et al. [2]) and Matching (Mulmuley-Vazirani-Vazirani [30]). Luby's work [28] on parallel maximal matching inspired a new line of work in derandomization. Seidel's work ([36] in shortest paths was soon followed by some very innovative approaches to global mincuts and minimum spanning trees by Karger et al. ([24, 26]) that consolidated the use of randomization in the area of mainstream graph algorithms. Subsequently, randomization has been used very effectively in dynamic graph algorithms like connectivity (Henzinger-King [20]), transitive closure and shortest path maintenance problems, some of which have no matching deterministic counterparts.

The above line of work must be clearly distinguished from a related but a fundamentally different model, viz., the random graph model of Erdos-Renyi that has a very long and rich history. However, these results are obtained by analysing the expected performance of an algorithm on a *random* instance of a graph in the $G_{n,p}$, a class of graphs with $n$ vertices where each of the $\binom{n}{2}$ edges occur with probability $p$. Our focus is on algorithms that use *random choices* and work for any input instance. Traditionally randomzed algorithms come in two flavors, namely *Monte Carlo* and *Las Vegas*. The former denotes a

0-8493-8597-0/01/$0.00+$1.50

class of algorithms whose output may contain errors (with probability bounded below 1/2) whereas the Las-Vegas algorithms always output the correct result. While Monte Carlo algorithms terminate in some predictable number of steps, the running time of a Las Vegas is a random variable whose expected value must be analyzed. For efficiency consideration, we are interested in polynomial time randomized algorithms (in case of Las Vegas algorithms the expected running time is bounded by a polynomial in size of the input). The class of polynomial time Monte Carlo algorithms are called $RP$ (1-sided error* Randomized Polynomial) whereas the (expected) polynomial time Las Vegas algorithms are known as $ZPP$ (zero error probabilistic polynomial). As the reader might have supected, the Las Vegas algorithms is the more desirable kind and it is known that the complexity classes are related as $ZPP \subseteq RP$, but not known in the reverse direction.

In this chapter we will focus on Las Vegas randomized algorithms that always produce the correct answer, but manifest some variation in the running times. The most common measure is the expected running time of the algorithm, where the expectation is over the choice of random bits in the algorithm and not the the input distribution. The natural concern is what is an *acceptable* probability of deviating from the expected running time by a certain amount - say a small constant factor. Roughly speaking, the expected running time implies that the probability of exceeding twice the expected time is 1/2. If the hardware itself is subjected to a similar scrutiny, there is empirical evidence that it has a failure probability as high as $\frac{1}{2^k}$ where $k$ is a constant. Therefore it is not unreasonable to aim for a success probability for randomized algorithms that is about an order of magnitude smaller than this. Over the past decade, the notion of *high probability* bound has gained wide acceptance. An algorithm is said to have a running time $f(n)$ with high probability (whp), if the following holds

$$\Pr(T(n) > c\alpha f(n)) \leq \frac{1}{n^\alpha}$$

for some constant $c$ and any $\alpha$, where $T(n)$ is the running time for an input of size $n$. This is also referred to as *inverse polynomial* probability of success. It must be noted that there is an implicit trade-off between the success probability and the running time. An inverse-*exponential* probability is even better, namely,

$$\Pr(T(n) > cf(n)) \leq \frac{1}{2^n}.$$

However, randomized algorithms that succeed with high probability are considered as reliable as any deterministic algorithm.

In this chapter, we shall discuss five problems on graph algorithms and present randomized algorithms for them. The particular choice of these problems has been due to the following reasons. These problems are simple as well some of the well researched problems. Moreover, they employ simple randomization ideas in a powerful way to achieve efficiency in the time complexity.

*There is a more general class called $BPP$ that allows 2 sided errors

## 1.2 A linear time Minimum Spanning Tree

The basic idea for obtaining a faster algorithm for MST is very intuitive. We first randomly sample a set of edges $S \subset E$ and construct a Minimum Spanning Forest $F$ (may not be connected). Using $F$, we can filter some of the edges using the *red rule*, i.e., the heaviest edge in any cycle cannot belong to the final MST. However, the technical lemma that achieves a provable bound on the number of edges that can be eliminated on the basis of the sampling is quite subtle.

**LEMMA 1.1** Given an undirected graph $G = (V, E)$, with a weight function $w : E \rightarrow \mathcal{R}$, let $G(p) = (V, E(p))$ denote the subgraph where each edge of $G$ is included in $G(p)$ independently with probability $p$. If $F(p)$ is a minimum spanning forest of $G(p)$,we define an edge $(u, v) \in E - E(p)$ to be **heavy** if each edge on the path from $u$ to $v$ in $F(p)$ has weights smaller than $w(u, v)$. Since the heavy edges cannot be part of the MST of $G$, we retain only the remaining edges, that are called **light** and this set is denoted by $L$. Then $E[L] = O(|V|/p)$.

The proof is based on an useful observation called principle of of deferred decision. We pretend that the sampling is happening online on a predetermined sequence using independent Bernoulli trials (with success probability $p$). This principle states that the sequence in which we do the Bernoulli trials is indistinguishable from independent offline sampling and therefore the bounds that we prove using some convenient online ordering will hold for the offline sampling.

We first sort the edges in increasing order of their weights and divide the sampling into $|V| - 1$ phases - in each phase we discover an additional edge of $F$. Note that $F$ may not be connected and may have less than $|V| - 1$ edges. We will prove that in each phase, the expected number of *light* edges is $O(1/p)$. We argue using induction. Suppose we are currently in the $i$-th phase, i.e., the currect size of $F$ is $i-1$ and $E_i$ is the remaining sequence for which we have not yet sampled. The (partial) $F$ categorizes $E_i$ into two distinct classes - those edges that are light (on the basis of the current $F$) and those that are not, call them *heavy*. The outcome of the sampling on the *heavy* edges of $E_i$ is not relevant, since they will be discarded. So we only focus on the subsequence $E_i^\ell$ of the *light* edges. The first edge that is selected from $E_i^\ell$ terminates the $i$-th phase. What is the expected number of edges in $E_i^\ell$ that are not sampled before the first one ? This is upperbounded by a geometric random variable that has expectation $1/p$. In fact, we observe that the number of light edges in each phase is independent and so we can obtain concentration bounds (that holds with probability approaching 1) on the total number of light edges over all the phases.

*The Algorithm*

The algorithm first applies two rounds of Boruvka's algorithm, i.e., each vertex $v$ chooses its nearest neighbour $u = N(v)$ and includes $(u, v)$ in the final MST *. Now contract these edges, and the resultant graph has no more than $|V|/2$ vertices (after contraction, every merged group has at least two vertices). Therefore after two rounds, the contracted graph, which is denoted by $G_2$ has at most $|V|/4$ vertices and each round takes $O(|E|)$ time. In $G_2$ we sample every edge with probability $1/2$ and in the resultant graph $G_2(1/2)$, we construct the MST of $G_2(1/2)$ *recursively*, denote this by $F(G_2(1/2))$. Using $F(G_2(1/2))$ to filter out

*This is a well known result.

those edges that cannot contribute to $F(G_2)$, we are left with a set of edges $E'$ where the expected size of $E'$ is $\leq 2 \cdot |V|/4 = |V|/2$. We again run the algorithm recursively on this graph $G' = (V', E)$ and output the edges in $F(G')$.

The running time of the algorithm depends on the procedure by which we detect the *light* edges. Let the time complexity of this procedure be $L(m, n)$. The entire algorithm can be viewed in terms of the (binary) recursion tree where the left-nodes correspond to the sampling and filtering step and the right node corresponds to the second recursive. Let us first bound the cost of applying Boruvka's algorithm to all nodes of the recursion tree. For this, it sufficies to count the total number of vertices and edges in all the subproblems. Since the number of vertices decrease by a factor of 4 at each level, the total number of nodes can be bounded by $\sum_{d=0}^{\infty} \frac{n}{4^d} \cdot 2^d \leq \sum_d \frac{n}{2^d} \leq 2n$. The sum total of edges can be bound by looking at the (disjoint) union of the edges belonging to all the maximal length left paths from each node. Since the expected number of edges in successive left nodes decrease by a factor of 2, the expected total number of edges is no more than a factor of two of the number of edges in the starting node. At the root level, there is only one starting node that contributes $2m$ edges. In level $d$, there are $2^{d-1}$ starting nodes (that are right children), each with $2\frac{n}{4^d}$ edges, that yields a total of $\sum_d \frac{n}{2^d}$ edges in these nodes and therefore the maximal left paths starting from these have a total of twice this quantity, i.e., $2n$. So the overall (expected) number of edges in all subproblems combined is $O(m + n)$ which is also the asymptotic cost of Boruvka's iterations. Moreover if $L(m, n)$ is $O(m + n)$*, then, the expected running time of the algorithm is $O(m + n)$.

## 1.3 Global mincut

A *cut* of a given (connected) graph $G = (V, E)$ is set of edges which when removed disconnects the graph. An $s - t$ cut must have the property that the designated vertices $s$ and $t$ should be in separate components. A *mincut* is the minimum number of edges that disconnects a graph and is sometimes referred to as *global* mincut to distinguish is from $s - t$ mincut. The weighted version of the mincut problem is the natural analogue when the edges have non-negative associated weights. A cut can also be represented by a set of vertices $S$ where the cut-edges are the edges connecting $S$ and $V - S$.

It was believed for a long time that the mincut is a harder problem to solve than the $s-t$ mincut - in fact the earlier algorithms for mincuts determined the $s-t$ mincuts for all pairs $s, t \in V$. The $s - t$ mincut can be determined from the $s - t$ maxflow flow algorithms and over the years, there have been improved reductions of the global mincut problem to the $s - t$ flow problem, such that it can now be solved in one computation of $s - t$ flow.

In a remarkable departure from this line of work, first Karger and Stein [26], followed by Karger [22] developed faster algorithms (than maxflow) to compute the mincut with *high probability*. The algorithms produce a cut that is very likely the mincut, i.e., these are Monte Carlo algorithms. Unfortunately, there is yet no known matching verification algorithms. We will describe an algorithm that runs in time $O(n^2 polylog(n)$, $(n = |V|)$ which is nearly best possible for dense graphs* This algorithm exploits some properties of branching processes.

*The existing linear time algoritms for verification of MST and detecting light edges are quite complex and is a good topic of research

*A more recent algorithm of Karger improves this to $O(|E|polylog(n)$ using a more sophisticated Monte Carlo algorithm.

**The contraction algorithm**

The basis of the algorithm is the procedure contraction described below. The fundamental operation $contract(v_1, v_2)$ replaces vertices $v_1$ and $v_2$ by a new vertex $v$ and assigns the set of edges incident on $v$ by the union of the edges incident on $v_1$ and $v_2$. We do not merge edges from $v_1$ and $v_2$ with the same end-point but retain them as multiple edges. Notice that by definition, the edges between $v_1$ and $v_2$ disappear.

**Procedure Contraction(t)**
Input: A multigraph $G = (V, E)$
Output: A $t$ partition of $V$

**Repeat** until $t$ vertices remain

choose an edge $(v_1, v_2)$ at random
**contract**$(v_1, v_2)$

Procedure $Contraction(2)$ produces a cut. Using the observation that, in an $n$-vertex graph with a mincut value $k$, the minimum degree of a vertex is $k$, the following can be shown quite easily.

**LEMMA 1.2** The probability that a specific mincut $\mathcal{C}$ survives at the end of Contraction(t) is at least $\frac{t(t-1)}{n(n-1)}$.

Therefore $Contraction(2)$ produces a mincut with probability $\Omega(\frac{1}{n^2})$.

**LEMMA 1.3** A single iteration of the Procedure Contraction can be carried out in $O(n)$ steps.

This is done by using an adjacency graph representation (see Karger and Stein [26] for details). Therefore using the Procedure Contract to produce mincut is somewhat expensive since we need to repeat it about $n^2$ times. Instead, we run Procedure $Contraction(\sqrt{n}/2)$ twice independently and repeat it recursively on the contracted graphs. The Algorithm is described below.

**Algorithm Fastmincut**
Input: A multigraph $G = (V, E)$
Output: A cut $\mathcal{C}$

1. Let $n := |V|$.
2. If $n \leq 6$ then compute mincut of $G$ directly else

2.1 $t := \lceil 1 + n/\sqrt{2} \rceil$.
2.2 Call $Contraction(t)$ twice (independently) to produce to graphs $H_1$ and $H_2$.
2.3 Let $C_1 =$ **Fastmincut** $(H_1)$ and $C_2 =$ **Fastmincut** $(H_2)$.
2.4 $\mathcal{C} = \min\{C_1, C_2\}$

The running time of algorithm **Fastmincut** satisfies the following recurrence

$$T(n) = 2T\left(\lceil 1 + n/\sqrt{2}\rceil\right) + O(n^2)$$

which yields $T(n) = O(n^2 \log n)$. Perhaps a more interesting question is to ascertain the probability with which **Fastmincut** returns a mincut. The probability that a mincut survives in $H_1$ after Step 2.2 is

$$\frac{(\lceil 1 + n/\sqrt{2}\rceil)(\lceil 1 + n/\sqrt{2}\rceil - 1)}{n(n-1)} \geq \frac{1}{2}$$

from Lemma 1.2. The same argument applies to $H_2$ independently. Therefore we can view the recursive algorithm as a branching process where any node can have zero, one or two children depending on if the mincut survived in zero, one or both children. The distribution function at each node can be approximated by a binomial distribution with two trials, each with success probability greater than 1/2 (i.e. mean $\mu \geq 1$). Since the algorithm has roughly $2 \log n$ levels of recursion, we can restate the survival probability of the mincut as the complement of the extinction probability at the $2 \log n$ generation.

The extinction probability of a branching process* with mean $\mu > 1$ converges to the solution of $x = P(x)$ where $P$ is the generating function of the probability distribution. Here, we can approximate $P(s)$ by $\frac{1}{4} + \frac{1}{2}s^2 + \frac{1}{4}s^2$. Solving for $x$ yields $x = 1$ which is an asymptotic solution but does not give us much information about the rate of convergence. For this, we need to solve the recurrence

$$x_n = P(x_{n-1})$$

where $x_i$ is the extinction probability of the $i$-th generation. Substituting our generating function and simplifying yields

$$x_n = \frac{1}{4}(1 + x_{n-1})^2$$

The solution to this recurrence is $x_n = \Theta(\frac{1}{n})$. So the survival probability of the mincut (after $2 \log n$) levels of recursion is $\Omega(\frac{1}{\log n})$. Repeating the procedure (with independently chosen random bits) $m \log n$ times increases the probability of finding the mincut to $1 - \exp^{-m}$.

## 1.4 Estimating the Size of Transitive Closure

Let $G = (V, E)$ be a directed graph on $n = |V|$ vertices and $m = |E|$ edges. For each vertex $v$, let $\tau(v)$ be the number of vertices reachable from $v$ in the given graph. Consider the problem of computing a very accurate estimate of $\tau(v)$ for each $v \in V$. We can compute $\tau$ trivially by executing a breadth first or depth first traversal from each vertex, and this will take $O(mn)$ time. We can also use repeated squaring of the adjacency matrix and this approach will achieve $O(n^\omega \log n))$ time, where $\omega$ is the exponent of the best known algorithm for multiplying two $n \times n$ integer matrices. Currently best bound on $\omega$ is 2.317 due to Coppersmith and Winograd [14]. These are the only two deterministic algorithms known for this problem. We shall now discuss an $O(m \log n)$ time randomized Monte Carlo

*The reader is referred to standard textbooks, for example, Feller[] for theoretical analysis of Branching Process

algorithm by Cohen [10] which, for any given constants $c_1, c_2 > 1$, computes $\hat{\tau}(v)$ for each vertex $v \in V$ satisfying

$$\frac{\tau(v)}{c_1} < \hat{\tau}(v) < c_2\tau(v)$$

with high probability. Here $c_1, c_2$ can be chosen arbitrarily close to 1. Notice that the constant in $O(m \log n)$ running time will depend upon the values $c_1, c_2$ and the desired probability of success.

In addition to being a problem of independent theoretical interest, this problem has applications in data bases. Before answering a data base query, one would like to estimate the size of the *query-answer set* to be reported. This prior knowledge may sometimes help in optimizing query processing time.

### 1.4.1 The idea of randomization

Suppose we select $k$ numbers uniformly independently from the interval $[0, 1]$. Let $X$ be the random variable defined as the smallest of these $k$ numbers.

**LEMMA 1.4** Expected value of $X$ is $1/(k + 1)$.

**PROOF 1.1** Notice that selecting $k$ numbers splits the interval $[0, 1]$ into $k+1$ intervals. Taking this viewpoint, $X$ is equal to the length of the leftmost of the $k+1$ intervals formed. To calculate $\mathbf{E}[X]$, we shall pursue this viewpoint. Consider a circle of circumference 1. Suppose we select $k+1$ points randomly uniformly and independently from the circumference of this circle. This will split the circumference into $k+1$ intervals. Exploiting the uniformity in sampling the points and the symmetry of the circle conclude that the lengths of these $k + 1$ intervals have identical probability distribution (though they are not independent). This implies that the expected length of any interval would be $1/(k + 1)$. Let us straighten the circle to form a line interval $[0, 1]$ by cutting the circle at any of the $k+1$ selected point. This creates an instance of our original experiment of selecting $k$ points from interval $[0, 1]$. Hence $\mathbf{E}[X] = 1/(k + 1)$.

The fact that the expected value of the smallest number among the $k$ numbers selected uniformly independently from $[0, 1]$ is $1/(k+1)$ conveys the following important observation: the number of random variables $k$ is related to $\mathbf{E}[X]$ very closely. We can use $X$ to infer the number of random variables which define it. In particular, if $X$ takes value $a$, we may return $1/a - 1$ as the estimate of the number of random variables. This randomization idea is the underlying idea of the algorithm for estimating the value $\tau(v)$ for each $v \in V$. However, to improve the accuracy of our estimate and the associated probability, we shall use the idea of multiple sampling.

### 1.4.2 Monte Carlo Algorithm

Though there is no deterministic algorithm to compute $\tau(v)$ for all $v \in V$ in $O(m)$ time, there are many problems on directed graphs which can be solved in $O(m)$ time (for example, computing the strongly connected components of the graph). One such problem is the following. Let each vertex stores some key which is a real number and the aim is to compute, for each $v \in V$, the key of the smallest key vertex reachable from $v$. There is a deterministic $O(m)$ time algorithm for this problem based on DFS (depth first search) of

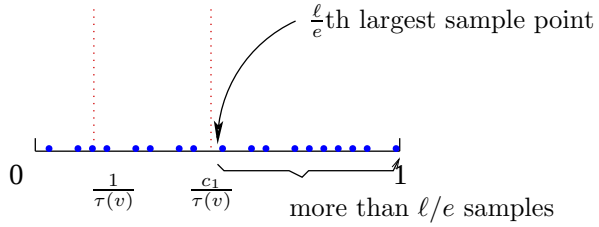


**FIGURE 1.1** The event when $(\ell/e)$th largest element happens to be greater than $c_1/\tau(v)$

the graph. We shall use this algorithm and the randomization idea described above to solve the problem of estimating $\tau(v)$ for each $v \in V$.

The algorithm will perform $\ell$ iterations. In $i$th iteration, we assign a key to each vertex in the graph by selecting a number uniformly randomly from the interval $[0, 1]$. After this we execute $O(m)$ time algorithm which computes, for each $v \in V$, $k_i[v]$ : the key of the smallest key vertex reachable from $v$. At the end of $\ell$ iterations, we shall have a set $\{k_1[v], \ldots, k_\ell[v]\}$ of $\ell$ such labels for each $v \in V$. It follows from Lemma 1.4 that the expected value of any element of the set is $\frac{1}{\tau(v)}$. (Actually, it is $\frac{1}{\tau(v)+1}$, but for simplicity and clarity of exposition we ignore the additive term of 1 from the denominator). In order to accurately estimate $\tau(v)$, we should select that element from $\{k_1[v], \ldots, k_\ell[v]\}$ which is going to be closest to $1/\tau(v)$ most likely. For this purpose, we state the following Lemma whose proof is elementary.

**LEMMA 1.5** If we select $j$ numbers uniformly independently from interval $[0, 1]$, the probability that the smallest number is greater than $c$ is $(1 - c)^j$ for any $0 < c < 1$.

It follows from Lemma 1.5 that the probability that $k_i[v]$, for any $i \leq \ell$, takes value greater than $\frac{1}{\tau(v)}$ is

$$\mathbf{P}[k_i[v] > \frac{1}{\tau(v)}] = \left(1 - \frac{1}{\tau(v)}\right)^{\tau(v)} \approx 1/e \tag{1.1}$$

Equation 1.1 implies that out of $\ell$ iterations of the algorithm, roughly $\ell/e$ times the sample will be greater than $1/\tau(v)$. Therefore, from the set $\{k_1(v), \ldots, k_\ell(v)\}$ the $(\ell/e)$th largest element is most likely to be closest to $1/\tau(v)$. Let this element be denoted as $k_v^*$. So the algorithm finally reports $1/k_v^*$ as the estimate $\hat{\tau}(v)$ of $\tau(v)$.

### 1.4.3 How accurate the estimate $\hat{\tau}(v)$ is to $\tau(v)$?

We shall try to get a bound on the probability for the event "$\tau(v)/c_1 < \hat{\tau}(v) < c_2\tau(v)$" for any given constants $c_1, c_2 > 1$. For this purpose we need to calculate a bound on the probability for the event "$\hat{\tau}(v) < \tau(v)/c_1$" and a bound on the probability for the event "$\hat{\tau}(v) > c_2\tau(v)$" separately. Let us focus on bounding the probability of "$\hat{\tau}(v) < \tau(v)/c_1$".

$\hat{\tau}(v) < \tau(v)/c_1$ means that the $(\ell/e)$th largest element from $\{k_1(v), k_2(v), ..., k_\ell(v)\}$ happened to be greater than $c_1/\tau(v)$. This means that at least $\ell/e$ of the elements from $\{k_1(v), k_2(v), ..., k_t(v)\}$ turned out to be greater than even $c_1/\tau(v)$. See Figure 1.4.3.

Let us introduce random variables $X_i, i \leq t$ at this moment. $X_i$ takes value 1 if $k_i[v]$ is greater than $c_1/\tau(v)$ and zero otherwise. Note that each of $X_i$, $i \leq \ell$ chooses its value independent of other $X_j$'s $j \neq i$ since every iteration of the algorithm assigns labels to vertices independent of other iterations. Let $X = \sum_i X_i$. So we can see that the event "$X > \ell/e$" is necessary for happening of the event "$\hat{\tau}(v) < \tau(v)/c_1$". So we shall now

calculate the probability of event "$X > \ell/e$".

Using lemma 1.5 and linearity of expectation, it is easy to observe that $\mathbf{E}[X]$ is $(1 - c_1/\tau(v))^{\tau(v)}\ell$, which is at most $e^{-c_1}\ell$ since $1 + \alpha < e^{\alpha}$, $\forall\alpha > 0$. So the situation is the following. Compared to its expected value of $\ell/e^{c_1}$, the random variable $X$ took value greater than $\ell/e$ which is quite large (depending upon $c_1$).

Let us compute the probability for this event using Chernoff bound. Convince yourself that $X$ fulfills all requirements for applying Chernoff bound. Let us, for the sake of clarity of exposition, assume the value of $c_1$ such that $e^{-c_1} = \frac{1}{2e}$ (the value of $c_1$ is close to 1.7). So the expected value of $X$ is $\ell/(2e)$. Since the value taken by $X$ is at least $\ell/e$, it implies that $\delta > 1$. Hence applying the Chernoff bound

$$\mathbf{P}[X > \ell/e] < e^{-\frac{\ell}{2e}/4} = e^{-\ell/8e}$$

If we choose $\ell = 24e \ln n$, that is, if we repeat the main iteration of the algorithm more than $24e \ln n$ times, the probability that $\tau(v)$ takes value less than $\tau(v)/(1.7)$ is less than $1/n^3$ . Using Boole's inequality (union theorem) we can thus state the following Lemma.

**LEMMA 1.6** If we repeat the main iteration of the algorithm at least $24e \ln n$ times, the probability that $\tau(v)$ is less than $\tau(v)/1.7$ for any $v$ is less than $1/n^2$.

In a similar fashion, we can calculate the probability of the event "$\hat{\tau}(v) > c_2\tau(v)$". We can thus conclude with the following theorem.

***THEOREM 1.1*** *Given a directed graph $G$ on $n$ vertices and $m$ edges, and any two constants $c_1, c_2 > 1$, there is a Monte Carlo randomized algorithm which takes $O(m \log n)$ time to compute $\hat{\tau}(v)$ for each $v$ such that with probability exceeding $1 - 1/n^2$,*

$$\tau(v)/c_1 \leq \hat{\tau}(v) \leq c_2\tau(v)$$

## 1.5 Decremental algorithm for maintaining strongly connected components

Given a directed graph $G = (V, E)$, two vertices $u, v \in V$ are are said to be strongly connected if there is a path from $u$ to $v$ as well as a path from $v$ to $u$. A strongly connected component (SCC) in a graph is a maximal subset of strongly connected vertices. The problem of maintaining SCCs under deletion of edges can be stated formally as follows:

There is an online sequence of edge deletions interspersed with the queries *"are $u$ and $v$ strongly connected"* for any $u, v \in V$. The aim is to maintain a data structure which can answer each query in $O(1)$ time and can be updated efficiently upon any edge deletion.

In addition to being a problem of independent interest, an efficient algorithm for this problem provides efficient decremental algorithm for maintaining all-pairs reachability.

Let us start with a simple-minded solution for this problem. This solution will use the well known static $O(m)$ time algorithm to compute SCCs of a given graph. In order to answer any query in $O(1)$ time, we may keep an array $A$ such that $A[u] = A[v]$ if $u$ and $v$ belong to the same SCC. For this purpose, we may select a unique vertex $\textsc{rep}(c)$ called *representative* vertex of $c$. For each $v \in c$, $A[v]$ store $\textsc{rep}(c)$. This ensures $O(1)$ query time.

In order to handle any edge deletion, we may execute the static $O(m)$ time algorithm to recompute SCC, and hence update $A$, after each edge deletion. This is a trivial decremental algorithm for maintaining SCCs while ensuring $O(1)$ query time. However, this algorithm will take a total of $O(m^2)$ update time to process any sequence of edge deletions.

We shall now discuss a very simple randomized Las Vegas algorithm by Roditty and Zwick [34] for maintaining SCCs under deletion of edges. The expected time taken by this algorithm to process any arbitrary sequence of edge deletions will be $O(mn \log n)$ only. Let us introduce a couple of notations here. For a SCC $c$, we shall use $G(c)$ to denote the subgraph of $G$ induced by $c$ and $E(c)$ to denote the edges of $G(c)$. Let $G^r$ denote the graph obtained by reversing all the edge directions in $G$.

We basically need an efficient mechanism to maintain the array $A$ under deletion of edges which bypasses the need of recomputing SCCs after each edge deletion. In particular, whenever an edge is deleted, we need to determine if it has indeed split an SCC into multiple SCCs. For this objective, for each SCC $c$, the following data structure is maintained :

- $T_{out}(c)$ : the BFS tree rooted at $\textsc{rep}(c)$ in graph $G(c)$.
- $T_{in}(c)$ : the BFS tree rooted at $\textsc{rep}(c)$ in graph $G^r(c)$.

We now provide an overview of a simple algorithm for maintaining a BFS tree rooted at a vertex under deletion of edges. This algorithm maintains level of each vertex. As the edges are being deleted, vertices may fall from their level to lower levels. Upon deletion of an edge, this algorithm takes $O(1)$ time to determine if it has caused fall of one or more vertices. The algorithm computes new levels of each vertex which falls. In doing so, for each vertex $x$ which has fallen from its level $i$ to level $j$, the algorithm incurs $O((j-i)\deg(x))$ computation cost in processing $x$. For the sake of clarity of our analysis, we *charge* the total computation performed during any sequence of edge deletions to the respective vertices which the algorithm processes. Since a vertex can only fall during edge deletions, and the lowest level is $n$, the total computation cost charged to $x$ during any sequence of edge deletion will be $O(n\deg(x))$. This implies a total of $O(mn)$ update time for maintaining a rooted BFS tree in a graph. Next, we describe the procedure for handling deletion of an edge.

### 1.5.1 Handling deletion of edges

Consider deletion of an edge $(u, v)$. If $u$ and $v$ belong to different SCCs, nothing needs to be done except deletion of $(u, v)$ from the graph. But, if $u$ and $v$ belong to the same SCC, say $c$, we update the data structures associated with $c$. If the deletion leads to any vertex leaving $T_{in}(c)$ or $T_{out}(c)$, it implies that SCC $c$ has split. We execute $O(|E(c)|)$ time algorithm to determine the new SCCs. For all vertices which leave the old SCC $c$, we delete them and all their edges from $T_{in}(c)$ and $T_{out}(c)$. For each new SCC $c'$, we select the representative $\textsc{rep}(c')$, and build BFS trees $T_{in}(c')$ and $T_{out}(c')$. We now state an important observation.

**Observation 1.5.1** *Suppose deletion of an edge splits a SCC $c$ into SCCs $c_1, \ldots, c_k$ with $\textsc{rep}(c)$ belonging to some SCC, say $c_1$. Each vertex of $c_1$ will not be charged any additional computation cost. But any vertex $x \in c_i, i > 1$ will be charged a computation cost of $O(|c_i|\deg(x))$ in building and maintaining the data structure associated with $c_i$.*

The algorithm described above is deterministic. and its worst case running time can still be $O(m^2)$. To achieve efficiency, we now add the following simple randomization ingredient to this algorithm. Each SCC $c$, at the moment of its creation, selects its $\textsc{rep}(c)$ randomly uniformly from $c$. It will turn out that this simple randomization leads to an efficient decremental algorithm for maintaining SCC. Let us analyze its running time.

There are two major computational tasks which are performed by the algorithm for any sequence of edge deletions. The first task is the execution of the static algorithm for determining new SCC. This task is executed whenever some SCC gets split and hence will be executed at most $n-1$ times during any sequence of edge deletions. A single execution of this algorithm takes $O(m)$ time, so overall $O(mn)$ computation time is spent in this task. Another computation task is associated with maintaining the data structures for various SCCs which get created during a sequence of edge deletions. To Analise this major task, we shall take vertex centric approach.

Consider any arbitrary sequence of edge deletions and focus on any vertex $v$. Let vertex $v$ changed its *SCC* a total of $t$ times and let $c_1, c_2, \ldots, c_t$ be the respective connected components. Notice that $c_i \subset c_{i-1}$ for all $1 < i \leq t$. Let $n_i = |c_i|$ and $n_1 = n$. We shall show that the expected computation cost charged to $v$ while maintaining the data structures associated with $c_1, \ldots, c_t$ will be $O(n \deg(v))$ only. Recall Observation 1.5.1. Let $X_i$ be the random variable which takes value 1 if during transition from $c_i$ to $c_{i+1}$, vertex $v$ is charged $O(|n_i| \deg(v))$ computation cost, and zero otherwise.

Note that $X_i = 1$ if $\textsc{rep}(c_i)$ belonged to $c_i \backslash c_{i+1}$. Since $\textsc{rep}(c_i)$ was selected randomly uniformly from $c_i$, hence

$$\mathbf{P}[X_i = 1] = \frac{n_i - n_{i+1}}{n_i}$$

So the expected computation cost charged to $v$ during any sequence of edge deletion is of the order of

$$\begin{aligned}\sum_{i=1}^{t-1} \mathbf{P}[X_i = 1] n_i \deg(v) &= \sum_{i=1}^{t-1} \frac{n_i - n_{i+1}}{n_i} n_i \deg(v) \\ &= \deg(v) \sum_{i=1}^{t-1} (n_i - n_{i+1}) = \deg(v)(n-1) = O(n \deg(v))\end{aligned}$$

So the expected total computation performed by the decremental algorithm of SCC for processing any sequence of edge deletions is $O(mn)$.

***THEOREM 1.2*** *For any directed graph $G = (V, E)$, there is a randomized decremental algorithm for maintaining SCCs with $O(1)$ query time and expected $O(mn)$ total update time.*

## 1.6 Approximate Distance Oracles

The all-pairs shortest paths problem is one of the most fundamental algorithmic graph problem. This problem is commonly phrased as follows : *Given a graph on $n$ vertices and $m$ edges, compute shortest-paths/distances between each pair of vertices.*

In many applications the aim is not to compute *all* distances, but to have a mechanism (data structure) through which we can extract distance/shortest-path for any pair of vertices efficiently. Therefore, the following is a useful alternate formulation of the APSP problem.

*Preprocess a given graph efficiently to build a data structure that can answer a shortest-path query or a distance query for any pair of vertices.*

The objective is to construct a data structure for this problem such that it is efficient both in terms of the space and the preprocessing time. There is a lower bound of $\Omega(n^2)$ on the space requirement of any data structure for APSP problem, and space requirement of

all the existing algorithms for APSP match this bound. However, this quadratic bound on the space is too large for many graphs which appear in various large scale applications. In most of these graphs, it is usual to have $m \ll n^2$, hence a table of $\Theta(n^2)$ size is too large to be kept. This has motivated researchers to design a subquadratic space data structures which may report approximate instead of exact distance between any two vertices. A path from $u$ to $v$ in a graph is said to be $t$-approximate if its length is at most $t$ times the length of the shortest path from $u$ to $v$. Thorup and Zwick [38] presented a novel data structure for all-pairs approximate shortest paths, called approximate distance oracles. They showed that any given weighted undirected graph can be preprocessed in sub-cubic time to build a data structure of sub-quadratic size for answering a distance query with stretch 3 or more. Note that 3 is also the least stretch for which we can achieve sub-quadratic space for APASP (see [12]). There are two very impressive features of their data structure. First, the trade-off between stretch and the size of data structure is essentially optimal assuming a 1963 girth lower bound conjecture of Erdős [16] and second, in spite of its sub-quadratic size their data structure can answer any distance query in *constant* time, hence the name "oracle". In precise words, Thorup and Zwick achieved the following result.

***THEOREM 1.3*** *[38] For any integer $k \geq 1$, an undirected weighted graph on $n$ vertices and $m$ edges can be preprocessed in expected $O(kmn^{1/k})$ time to build a data structure of size $O(kn^{1+1/k})$ that can answer any $(2k-1)$-approximate distance query in $O(k)$ time.*

We provide below description of 3-approximate distance oracle.

### 3-approximate distance oracle

In order to achieve subquadratic space, the 3-approximate distance oracle is based on the following idea.

*From each vertex $v$, we store distance and shortest paths to a small set of vertices lying with in* small *vicinity of $v$. This will take care of distance queries from $v$ to its nearby vertices. For querying distance to a vertex, say $w$, lying outside the vicinity of $v$, we may do the following. We may have a small set $S$ of* special *vertices and we store distance between each special vertex and every vertex of the graph. In order to report (approximate) distance between $v$ and $w$, we may report the sum of the distances from some (carefully defined) special vertex to $v$ and $w$.*

The above idea, though looks intuitively appealing, appears difficult to materialize. In particular, how do we define *vicinity* around a vertex and ensure a finite stretch, while having only a small set of special vertices ? Randomization in selecting $S$ plays a crucial role to achieve all these goals simultaneously. The following terminology captures the notion of vicinity of a vertex in terms of the special vertices.

**DEFINITION 1.1** Given a graph $G = (V, E)$, a vertex $v \in V$, and any subset $S \subset V$ of vertices, we define $Ball(v, V, S)$ as a set in the following way.

$$Ball(v, V, S) = \{x \in V | \delta(v, x) < \delta(v, Y)\}$$

Let $p(v, S)$ denote the vertex from set $S$ which is nearest to $v$. In simple words, $Ball(v, V, S)$ consists of all those vertices of the graph whose distance from $v$ is less than the distance of $p(v, S)$ from $v$. The 3-approximate distance oracle is built as follows.

1. Let $S \subseteq V$ be formed by selecting each vertex uniformly and independently with probability $1/\sqrt{n}$.

2. From each vertex $S$, store distance to all the vertices.
3. From each vertex $v \in V \backslash S$, compute $p(v, S)$ and build a hash table which stores all the vertices of $Ball(v, V, S)$ and their distance from $v$.

See Figure 1.6 to get a better picture of the 3-approximate distance oracle.

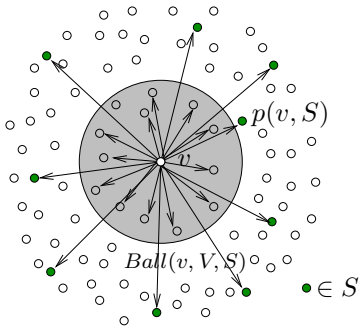


**FIGURE 1.2** The oracle keeps distance information between $v$ and all the vertices pointed by arrows.

**Expected size of the data structure :**
The expected size of the data structure computed as described above depends upon the expected size of $Ball(v, V, S)$ which turns out to be $O(\sqrt{n})$ as follows. Consider the sequence $\langle (v =)x_0, x_1, x_2, \cdots \rangle$ of vertices $V$ arranged in non-decreasing order of their distances from $v$. The vertex $x_i$ will belong to $Ball(v, V, S)$ only if none of $x_1, \cdots, x_i$ are selected in the set $S$. Therefore, $x_i \in Ball(v, V, S)$ with probability at most $(1 - 1/\sqrt{n})^i$. Hence using linearity of expectation, the expected number of vertices in $Ball(v, V, S)$ is at most $\sum_i (1 - 1/\sqrt{n})^i < \sqrt{n}$. Hence the expected size of the data structure will be $O(n\sqrt{n})$.

**Answering a distance query :**
Let $u, v \in V$ be any two vertices whose approximate distance is to be computed. First it is determined whether or not $u \in Ball(v, V, S)$, and if so, the exact distance $\delta(u, v)$ is reported. Note that $u \notin Ball(v, V, S)$ would imply $\delta(v, p(v, S)) \leq \delta(v, u)$. In this case, report the distance $\delta(v, p(v, S)) + \delta(u, p(v, S))$, which is at least $\delta(u, v)$ (using the triangle inequality) and upper-bounded by $3\delta(u, v)$ as shown below.

$$\begin{aligned} \delta(v, p(v, S)) + \delta(u, p(v, S)) &\leq \delta(v, p(v, S)) + (\delta(u, v) + \delta(v, p(v, S))) && (1.2) \\ &= 2\delta(v, p(v, S)) + \delta(u, v) && (1.3) \\ &\leq 2\delta(u, v) + \delta(u, v) = 3\delta(u, v) && (1.4) \end{aligned}$$

## Recent and related work on approximate distance oracles

Thorup and Zwick [38] presented an expected $O(kmn^{1/k})$ time algorithm for computing a $(2k - 1)$ spanner. There has been a lot of work [6, 4] in designing faster algorithms for computing a $(2k - 1)$-approximate distance oracle. Currently, the best bounds for constructing these oracles is $O(n^2)$. Interestingly, these fast algorithms also employ simple randomization ideas.

An equally interesting question is to explore the possibility of approximate distance oracles for stretch better than 3. Recently Patrascu and Roditty [32] presented a positive

answer to this question. They designed a 2-approximate distance oracle of size $O(n^{5/3})$ for unweighted graphs. They also generalize it for weighted graphs, though the size bounds depend on the number of edges of the graph: For graphs with edges $n^2/\alpha$, they design a 2-approximate distance oracle with $O(n^2/\sqrt[3]{\alpha})$.

A related structure in graph theory, similar to the approximate distance oracles, is a graph spanner. A spanner is a subgraph which is sparse and yet preserves all-pairs distances approximately: Given an undirected graph $G = (V, E)$ and positive integer $k$, a $(2k-1)$-spanner is a subgraph $(V, E_S)$, such that the distance between any pair of vertices $u, v$ in the subgraph is at most $(2k-1)\delta(u, v)$. An interesting byproduct of the $(2k-1)$-approximate distance oracle of Thorup and Zwick [38] is a $(2k-1)$-spanner with $O(kn^{1+1/k})$ edges. In fact 3-approximate distance oracle described above leads to 3-spanner as follows. From each special vertex $v \in S$, include all the edges of the shortest path tree rooted at $v$ in the spanner. For every non-special vertex $v \in V \backslash S$, just store the edges of the shortest path tree from $v$ to only those vertices that belong to $Ball(v, V, S)$. It follows that the subgraph defined by these edges is indeed a 3-spanner. Interestingly, there has been a Las Vegas randomized algorithm [8] to compute a $(2k-1)$-spanner with $O(kn^{1+1/k})$ edges in expected $O(km)$ time.

## References


2. A. Aggarwal, R. J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.
4. S. Baswana and T. Kavitha. Faster algorithms for all-pairs approximate shortest paths in undirected graphs. *SIAM J. Comput.*, 39(7):2865–2896, 2010.
6. S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in expected $O(n^2)$ time. *ACM Transactions on Algorithms*, 2(4):557–577, 2006.
8. S. Baswana and S. Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.
10. E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.
12. E. Cohen and U. Zwick. All-pairs small-stretch paths. *J. Algorithms*, 38(2):335–353, 2001.
14. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.
16. P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.
18. H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.
20. M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.
22. D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.
24. D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.
26. D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.
28. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

30. K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.
32. M. Patrascu and L. Roditty. Distance oracles beyond the thorup-zwick bound. In *FOCS*, pages 815–823, 2010.
34. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.
36. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.
38. M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.