## Basic Results

**Union Bound**. Let $E_1, E_2, \ldots, E_n$ be a collection of events. Then $Pr\left(\bigcup_{i=1}^{n} E_i\right) \leq \sum_{i=1}^{n} Pr(E_i)$.

Useful to bound the probability of at least 1 bad events happening.

**Markov's Inequality**. $P(|X| \geq a) \leq \frac{E[|X|^n]}{a^n}$ for $a > 0$ and some $n \in \mathbb{N}$

Well known. $\frac{1}{e^2} \leq \left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}$. The first inequality is correct for $x > 2$.

**Hoeffding Bound.** If $Y_1, Y_2, \ldots, Y_s$ are **independent** RV in the **range** $[0, 1]$, define $Z = \sum_{j=1}^{s} Y_j$, then $Pr[|Z - E[Z]| \geq \delta] \leq 2e^{-2\delta^2/s}$

Note: make sure $Y_i$-s and $Z$ satisfy the requirements before using this technique

**Chernoff Bound.** If $Y_1, Y_2, \ldots, Y_s$ are **independent** RV in the **range** $[0, a]$ for some constant $s$, define $Z = \sum_{j=1}^{s} Y_j$ and $\mu = E[Z]$ then for all $0 \leq \delta \leq 1$,

1. $Pr[Z \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/(3a)}$
2. $Pr[Z \leq (1 - \delta)\mu] \leq e^{-\mu\delta^2/(2a)}$

**Chebyshev.** Let $X$ be an RV. For any real number $k > 0$, $Pr(|X - E[X]| \geq k) \leq \frac{Var[X]}{k^2}$

# Sublinear Sampling Algorithms (1 - 3)

Approximate solutions:

1. Relative error: $MST(G)(1 - \epsilon) \leq ALG(G) \leq MST(G)(1 + \epsilon)$
2. Absolute error: $MST(G) - \epsilon \leq ALG(G) \leq MST(G) + \epsilon$
3. Gap error:
   - If G is connected, return TRUE.
   - If G is $\epsilon$-far from connected, return FALSE.
   - Otherwise, don't care

Key Idea:

- find a local cost (or anything you can sample locally) to learn about large parts of the graph.

## All-Zeros?

```
AllZeros(A, ε):
    Repeat s = 2/ε times:
        Choose random i in [1, n]
        If A[i] == 1: return FALSE
    return TRUE
```

Using $\left(1-\frac{1}{x}\right)^x \le \frac{1}{e}$, we can prove Gap Error, i.e.

1. If the array is all zeros, we always return TRUE
2. If the array has $\ge \epsilon n$ ones, we will return FALSE w.p. $\ge 2/3$

## How many ones?

```
FractionOnes(A, ε):
    sum = 0
    Repeat s = 1/ε^2 times:
        Choose random i in [1, n]
        sum = sum + A[i]
    return sum / s
```

Let $f$ be the fraction of ones in the array. Using Hoeffding Bound, we can prove that $|\text{sum}/s - f| \le \epsilon$ w.p. $\ge 2/3$.

## Is the graph connected?

Graph $G$ has $n$ nodes, $m$ edges. Each node has a maximum degree of $d$. A good algorithm for sparse undirected graphs runs in $O(1/\epsilon^2 d)$ time to output Gap-Error w.p. $> 2/3$

```
Connected(G, n, d, ε)
    Repeat 16/εd times:
        Choose random node u.
        Do a BFS from u, stopping after 8/εd nodes are found.
        If CC of u has ≤ 8/εd nodes: return FALSE.
    Return TRUE
```

Key Ideas:

1. If G is $\epsilon$-far, then there are $\ge \epsilon dn/4$ connected components
2. If G is $\epsilon$-far, there are at $\ge \epsilon dn/8$ CC with size $\le 8/\epsilon d$. (In simple words, there are many small CC if G is $\epsilon$-far)
3. Stop early with the BFS. The BFS runs in $O(d \times 8/\epsilon d) = O(1/\epsilon)$

Insights:

> Why is the definition of $\epsilon$-close based on $\epsilon nd$ entries?

Intuition: we want to count how many edges do we need to add s.t. the graph is connected. Observe that $nd$ is the total degree of the graph (twice the edges) and so the maximum number of edges we can add is $nd/2$. So, it is natural that we use $\epsilon nd$ as our definition

> The running time is $O(1/(\epsilon^2 d))$. It seems that if $d$ is larger, the running time will be smaller?

If $d$ is larger, say $d \to n$, then for the problem to be interesting, we should have $\epsilon < 1/n$ (why? See our definition of $\epsilon$-far). This implies that our running time goes to $O(n)$ as $d$ goes to $n$ (not desirable).

## Number of connected components

Running time : $O(d(\ln(1/\delta))/\epsilon^3)$ to output $C$ s.t. $|CC(G) - C| \leq \epsilon n$ w.p. $\geq 1 - \delta$

```
sum = 0
for j = 1 to s = 4/ε^2:
    Choose u uniformly at random.
    Perform a BFS from u; stop after seeing 2/ε nodes.
    if BFS found > 2/ε nodes:
        sum = sum + ε/2
    else if BFS found n(u) nodes:
        sum = sum + 1/n(u)
return n * (sum/s)
```

Key Ideas:

1. For all nodes $u$, define $cost(u) = 1/n(u)$ where $n(u)$ is the number of nodes in the CC containing node $u$. → $\sum cost(u)$ is the total number of CC in the graph
2. Sampling: Use Hoeffding to get $Pr\left(|sum - E[sum]| \geq \epsilon s/2\right) \leq 1/3$
3. Approximating $cost(u)$ → by stopping early if CC is large.

Insights

> Why is the definition of $\epsilon$-close based on $\epsilon n$?

Intuition: The maximum number of CC we can have is $n$.

## MST Weight

Problem setting: undirected connected weighted graph with $n$ nodes, $m$ edges, maximum degree $d$ and max weight $W$. Output $M$ s.t. $M = MST(G)(1 \pm \epsilon)$ in $O((dW^4 \log W)/\epsilon^3)$

```
sum = n - W
for j = 1 to W - 1:
    sum += ApproxCC(G_j, d, ε', δ) where ε' = ε/W and δ = 1/(3W)
return sum
```

Key Ideas

1. $MST(G) = n - W + \sum_{j=1}^{W-1} C_j$ where $C_j$ is the number of connected components in $G_j$, graph containing edges with weight $\leq j$.
2. $MST(G) \geq n - 1 > n/2$ to change additive approximation to multiplicative approximation.

## Maximal Matching

We know that maximal matching gives a 2-approximation of the maximum matching.

Algorithm (runs in $O(e^d/\epsilon^2)$ time):

Choose a random permutation for the edges, e.g. assign a hash value to each edge

```
def query(e):
    for all neighbors e' of e:
        if hash(e') < hash(e) && query(e'):
            return FALSE
    return TRUE

sum = 0
for j = 1 to s:
    Choose an node u uniformly at random.
    if query(e) = True for all adjacent edge e of u:
        sum += 1
return (1/2) * n * (sum / s)
```

Query takes expected time: $\sum_{i=1}^{\infty} 2d^k/k! = O(e^d)$

The algorithm returns 0.5 because each edge in $M$ have 2 endpoints.

## Yao's Min-Max Principle for Lower Bounds

The expected cost of a randomized algorithm on its worst-case input is no better than the expected cost for a worst-case probability distribution as the inputs of the deterministic algorithm that performs best against that distribution.

Recipe

1. Choose a distribution $\gamma$
2. Show that the expected cost of every deterministic algorithm on input from $\gamma$ is slow
3. Conclude that every randomized algorithm has at least one input with expected cost just as slow.

Property Testing Version: replace expected cost with probability algorithm is wrong, i.e. if there exists a distribution $\gamma$ of inputs s.t. the probability of $A(x)$ is wrong $> 1/3$ for every deterministic algorithm $A$ with query complexity $q$ , then for every randomized algorithm $B$ of query complexity $q$, there exist an input $x$ s.t. the probability of $B(x)$ is wrong $> 1/3$.

# Sketching and Sampling (4 - 6)

The space in streaming algorithms counts the number of bits to store the entire DS. e.g. $n \in \mathbb{N}$ is stored in $\log n$ bits.

## Item Frequencies / Heavy Hitters

Given a stream of items $s_1, s_2, \dots s_m$, find in small space:

1. count(x) : $|N(x) - \text{count}(x)| \leq \epsilon m$
2. heavy hitters: return
   1. every item that appears $\geq 2\epsilon m$ times.
   2. no item that appears $< \epsilon m$ times.

**Misra-Gries Algorithm**

```
Set P of <item, count> pairs
For each u in stream S
    if <u, c> in P: increment c
    else: add <u, 1> to set P

    if | P | > k, decrement c for each item in P
    remove all items from P with c = 0
```

```
Count(x) = c if <x, c> in P else 0
```

Space: $O(k \log m)$
Correctness: $N(x) \geq \text{count}(x) \geq N(x) - m/k$

Taking $k = 1/\epsilon$ and life is good.

**Heavy hitters** : return all item that appears $\geq 2\epsilon m$ times, but no item that appears $< \epsilon m$ times

- Solution: return x if count(x) $\geq \epsilon m$

Insight:

1. Misra-Gries cannot differentiate elements that appear a small number of times, i.e. an element that appear zero times / once looks the same to MG.

# Counting distinct elements

Given a stream of items $s_1, s_2, \dots s_m$, find in small space:

1. distinct($\epsilon$): $(1 \pm \epsilon)$ approximation with probability at least $(1 - \delta)$

**Flajolet-Martin (FM) Algorithm**

```
Let x = 1, and h(u) in [0, 1]
For each u in stream S:
    if h(u) < x: x = h(u)
Return -1 + 1/x
```

Tricks:

1. Use a hash function: to randomize the items
2. Use the minimum of a set of RV.

We can prove that $E[X] = \frac{1}{t+1}$ and $Var(X) \leq \frac{1}{(t+1)^2}$ using $X$ as a continuous RV.

If we apply Chebyshev directly, we get
$Pr\left[\left|X - \frac{1}{t+1}\right| \geq \epsilon\left(\frac{1}{t+1}\right)\right] \leq Var(X)\frac{(t+1)^2}{\epsilon^2} \leq \frac{1}{e^2}$

However the right hand side is still too big (Note that $\frac{1}{e^2} > 1$).

To make it smaller, we can repeat FM $a$ times and **take the average**. (reducing variance by $1/a$)

**FM+ Algorithm**

1. Run $a$ copies of FM-subroutine. get $X_1, X_2, \ldots, X_a$
2. Compute average $Z = \frac{1}{a}\sum_{j=1}^{a} X_j$
3. Return $-1 + 1/Z$.

Using basic probability, we can prove that $E[Z] = \frac{1}{t+1}$ and $Var(Z) \leq \frac{1}{a(t+1)^2}$

Now we have $Pr\left[\left|Z - \frac{1}{t+1}\right| \geq \epsilon\left(\frac{1}{t+1}\right)\right] \leq \frac{1}{ae^2}$. Taking $a = \frac{4}{e^2}$, the RHS $\leq 1/4$

Using the fact that for $0 < x < 1/2$, $\frac{1}{1-x} \leq 1 + 2x$ and $\frac{1}{1+x} \geq 1 - x$, we have $-1 + 1/Z \in t(1 \pm 4\epsilon)$ w.p. $\geq 3/4$.

To make the probability bigger, we can repeat FM+ $b$ times and **take its median.** (amplifying probability)

**FM++ Algorithm**

1. Run $b$ copies of FM+ subroutine. Get $Y_1, Y_2, \ldots, Y_b$
2. Return median$(Y_1, Y_2, \ldots, Y_b)$

Key Idea: If $> 1/2$ of the $Y_j$'s are within $t(1 \pm 4\epsilon)$, then its median are also within $t(1 \pm 4\epsilon)$

Using Chernoff Bound, and $b = 36 \ln(2/\delta)$, we can get the desired result.

# Connectivity

Maintain spanning forest of the graph

```
F: forest, initially empty
for each edge e in stream:
    if F U e has no cycles then <- Union Find
        add e to F
n = # of components in F
return n
```

Space: $O(n \log n)$ ← there are at most $n - 1$ edges denoted by the 2 endpoints, each taking $O(\log n)$ bits to store.

Update cost: $O(\alpha(n, n))$

## Is the graph bipartite?

Bipartite = 2 way coloring / no odd cycle

Maintain spanning forest of the graph

```
F: forest, initially empty
for each edge e in stream:
    if F U e has no cycles then <- Union Find
        add e to F
    if F U e has odd-length cycles then <- maintain 2-coloring
        return NO
return YES
```

Space: $O(n \log n)$
Update cost: $O(\alpha(n, n))$

## Approximating shortest paths

Idea: Find a "good" subgraph $H \subseteq G$ (called spanner) s.t.

1. $H$ is sparse
2. $\forall (u, v) \in V^2$, $d_G(u, v) \leq d_H(u, v) \leq \alpha d_G(u, v)$ where $\alpha = \max \left\{ \frac{d_H(u,v)}{d_G(u,v)} \mid (u, v) \in E \right\}$

```
H: subgraph, initially empty
for each edge e = (u,v) in stream:
    if d_H(u,v) >= 2k:
        add e to H
return H
```

**Claim**: $\alpha < 2k$.
**Claim**: $H$ has no cycle with size $\leq 2k$

Define girth as the size of the smallest cycle
**Theorem**: If graph $G$ has girth $> 2k$, then it has $O(n^{1+1/k})$ edges.
Proof Sketch: Suppose $|H| \geq 10n^{1+1/k}$. Kill all nodes with degree $\leq 2n^{1/k}$. Using the fact that all cycles are of size $\geq 2k+1$, obtain a contradiction on the number of node $n$.

So, to obtain a 3-spanner, we need $O(n^{3/2} \log n)$ space and to obtain a $\log n$-spanner, we can do it in $O(n \log n)$ space.

## (Weighted) Matching

```
M: matching, initially empty
for each edge e = (u,v) in stream:
    let C be edges adjacent to nodes u and v in M
    if w(e) > (1 + \gamma) w(C):
```

```
        remove C from M
        add e to M.
```

Define:

1. edge $e$ is born when added to $M$
2. edge $e$ is killed by $e'$ if $e$ is removed when $e'$ is born
3. edge $e$ is a survivor if it is born and never killed.

For $e \in M$, define tree of the dead $T(e) = T_1(e) \cup T_2(e) \cup \ldots$ where $T_0(e) = \{e\}$ and $T_{j+1}(e)$ is the set of edges killed by $T_j(e)$. Note we have $(1+\gamma)W(T_{j+1}(e)) < W(T_j(e))$. We can then prove $\gamma W(T(e)) < W(e)$

**Charging argument:**
Let $e$ be some edge in $M^*$ (optimal maximum matching).

- If $e \in M$ or $e \in T(M)$, we charge $w(e)$ to $e$.
- Otherwise $e$ is never born. So, for edge $e'$ adjacent to $e$, we split $w(e)$ proportionally s.t. charge to $e'$ is $< (1+\gamma)W(e')$ (Note that $e' \in M$ or $e' \in T(M)$)

Total charges = $W(M^*)$
Now, for all $e \in M$ or $e \in T(M)$, they are either:

1. charged once for being a survivor / being killed once
2. charged at most twice by unborn edges $e_1, e_2 \in M^*$

Since each charge (to an edge $e$) is $< (1+\gamma)W(e)$, we have
$W(M^*) < 2(1+\gamma)(W(M) + W(T(M)) \leq 8W(M)$. So we have an 8-approximation.

# k-median clustering

Problem: find $k$ centres that minimize the average distance to a center.

Given points $P = \{p_1, p_2, \ldots, p_n\}$, find points $C = \{c_1, c_2, \ldots, c_k\} \subset P$ that minimize
$D(P, C) = \sum_{i=1}^{n} \min_{c_j \in C} |p_i - c_j|$

### LP Approximation Algorithm

Goal: find $2k$ centers that give a 4-approximation of the optimal clustering. This is called a (2, 4)-approximation.

Integer LP (NP-hard)

$$\begin{aligned} \min \sum_{i,j} x_{i,j} d(p_i, p_j) & \\ \textstyle\sum_j x_{i,j} = 1 & \quad \forall i \\ \textstyle\sum_j y_j \leq k & \\ x_{i,j} \leq y_j & \quad \forall i, j \\ x_{i,j}, y_j \in \{0, 1\} & \quad \forall i, j \end{aligned}$$

Intuition:

- $x_{i,j}$ denotes if $p_i$ is assigned to centre $p_j$
- $y_j$ denotes if $p_j$ is a chosen centre

We can relax by replacing the integral constraints with continuous constraints $0 \leq x_{i,j}, y_j \leq 1$. We now have LP which is easily solvable.

Note that the solution to the LP $C$ is at least as optimal as the ILP's $C^*$, i.e. $D(C, P) \leq D(C^*, P)$, but might not be valid as it can be fractional. So, how to round?

Define the cost of $p_i$ as $C_i = \sum_j x_{i,j} d(p_i, p_j)$. Our goal after rounding is to construct $C'$ s.t. $C'_j \leq 4C_j$

1. Sort the points by cost
2. Add $p_j$ with the smallest cost $C_j$ to our set of centres $S$. (why? because smaller cost is more sensitive to bad roundups)
3. Delete all points in $V(j) = \{p_i \mid \exists q \in P, d(p_i, q) \leq 2C_i, d(q, p_j) \leq 2C_j\}$
4. Repeat steps 2 and 3 until all points are deleted. Return $S$

**Claim**: For all $j$, $C'_j \leq 4C_j$
**Claim**: $|S| \leq 2k$

Note: $V(j)$ is not disjoint, so for to prove the following lemma, we define $V'(i) = V(i) \cap \{p_j \mid d(p_i, p_j) \leq 2C_i\}$. We can show that $V'(j)$ is disjoint.

**Lemma**: Let $p_i \in S$, then $\sum_{j\,:\,d(p_i,p_j) \leq 2C_i} y_j \geq 1/2$

The lemma implies there are at most $|S| \leq 2k$ as $\sum_j y_j \leq k$.

Observation 1: $\sum_{j\,:\,d(p_i,p_j) \leq 2C_i} y_j \geq \sum_{j\,:\,d(p_i,p_j) \leq 2C_i} x_{i,j}$
Observation 2: $C_i = \sum_j x_{i,j} d(p_i, p_j)$ is the avg distance from a point $p_i$ to a center.

Let $Z$ be an RV that equals $d(p_i, p_j)$ with probability $x_{i,j}$. Note that $E[Z] = C_i$. By Markov, we have $P(Z \geq 2C_i) \leq 1/2$. Thus, we have $\sum_{j\,:\,d(p_i,p_j) \leq 2C_i} x_{i,j} = P(Z \leq 2C_i) \geq 1/2$

## Streaming k-Median

$O(\sqrt{nk})$ memory for a (2, O(1))-approximation

Points arrive in stream, and we'd like to output $k$ cluster centers at the end.

#### Core-Set Algorithm

```
C = {}
Repeat sqrt(n / k) times:
    Let P = next sqrt(nk) points and find its (2,4)-approximate clustering.
    Add the 2k new cluster centers to C.
    Each center is weighted according to the number of points attached to it.
Return (2,4)-approximate weighted clustering on C
```

Define substream $S_i$ as the i-th segment of stream $S$. Let $T_i$ be the $2k$ centers output by `ApproxCluster` on $S_i$. Let $S_w$ be the weighted points used for the final `ApproxCluster`, and $T$ be its final output.

Useful fact: $\min_{T' \subseteq S'} d(S', T') \leq 2 \min_{T' \subseteq A} d(S', T')$ for arbitrary $A$ s.t. $S' \subseteq A$. In other words, to cluster $S'$, we can focus on points in $S'$ and only lose a factor of 2 compared to the global optimum.

Let $C^*$ be the optimal clustering, we have

1. $d(S, T) \leq \sum_{i=1}^{t} d(S_i, T_i) + d(S_w, T)$ : Triangle Ineq
2. $\sum_{i=1}^{t} d(S_i, T_i) \leq 8d(S, C^*)$ and $d(S_w, T) \leq 8d(S_w, C^*)$ : Useful fact + (2,4)-`ApproxCluster`
3. $d(S_w, C^*) \leq \sum_{i=1}^{t} d(S_i, T_i) + d(S, C^*)$

Hence, we conclude that $d(S, T) \leq 80d(S, C^*)$

#### Hierarchical Construction

We can extend this core-set algorithm to more depth. E.g. instead of processing every $\sqrt{nk}$ points, we process every $n^\epsilon$ points → we get $1/\epsilon$ height. We can store at most $m$ sets of centres for each level of the tree, ending up with $O(2kn^\epsilon/\epsilon)$ space and approximation factor $(8^{1/\epsilon})$

## k-center

Problem: find $k$ centres that minimize the maximum distance to a center.

2-Approximation algorithm

```
T = {x} for any x in P
Repeat until |T| = k:
    T += { the point in P that maximizes d(z, T) }
Return T
```

# Cache-Efficient Search Structures (7, 9, 10)

#### External Memory Model

A single ideal cache of size $M$ with block size $B$ fronting disk with infinite capacity.

**Cost**: Number of lines read from or written to memory.

Tall Cache assumption, i.e. $M = \Omega(B^{1+\epsilon})$, is sometimes used.

#### Basic Results

- Scans: LinkedList: $O(N)$, Array: $O(\lceil N/B \rceil)$
- Search: LinkedList: $O(N)$, Red-Black Tree: $O(\lg n)$, Array binary search: $O(\log N/B)$, B-Tree: $O(\log_B N)$

- Sort:
  - External Sort ($M/B$-way merge sort[1]) : $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$
  - Funnel Sort (cache-oblivious, similar to van Emde Boas layout) [2] : $N^{1/3}$-way merge sort using $K$-funnel which merges $K$ sorted list of total size $\Theta(K^3)$ in $O(\frac{K^3}{B}\log_{M/B}\frac{K}{B}+K)$
- Graphs $G(V,E)$: Priority Queue: $O(\frac{1}{B}\log_{M/B}\frac{V}{B})$, Unweighted shortest paths: $O(V+\frac{E}{B}\log_{M/B}\frac{E}{B})$, Dijkstra's: $O(V+\frac{E}{B}\log\frac{E}{M})$, Unweighted APSP: $O(\frac{VE}{B}\log_{M/B}\frac{E}{B})$

## B-tree

(a, b)-trees with $a \geq 2, b \geq 2a$ and take $a, b \sim O(B)$

- We can do a lazy split / optimistic splitting.
- Insert: insert to leaf, do split if an internal node is $\geq b$
- Delete: delete from leaf, "borrow" elements if total elements in siblings are $\geq b$, otherwise do a merge with sibling.
- Insert / Delete is $O(1/B\log_B N)$ amortized cost with $a = B, b = 5B$
- Search is $O(\log_B N)$
- If we store a parent pointer for each node, the cost would be $O(\log_B N)$ amortized as we need to update $\Theta(B)$ pointers for node in each level.

## Buffer Tree

Define: Leaf parameter as the max number of keys in the leaf, and branching parameter as the max number of keys in internal nodes. leaf parameter might be different from branching parameter.

Write-optimized data structure: use delayed queries and batched updates.

**Idea**: B-tree with branching parameter $5M/B$ and leaf parameter $5B$ and add a buffer of size $M$ to each *internal* node to temporarily store queries.

- Insert/Delete: Add op to root buffer (if it is not cancelling a previous op), and buffer `flush` if size of buffer $\geq B$
- `flush` to an internal node: sort the buffer, move every op to its proper child, clean child buffer (e.g. removing duplicates), recursively flush child buffer if necessary
- `flush` to a leaf node: sort the buffer, perform delete ops, then insert ops, do splits as needed (whenever we do a split, its buffer is empty), do merges.
- Flush cost is cost of load + distribution: $O(M/B)$ amortized
- height is $O(\log_{M/B}\frac{N}{B})$
- Insert / Delete costs: $O(\frac{\text{flush cost}}{M}\log_{M/B}\frac{N}{B})$ amortized.

Note: If we use a (2,4)-tree as underlying DS, we obtain flush cost: $O(1/B)$ amortized. If we use a $(\sqrt{B}, 2\sqrt{B})$-tree as underlying DS, we obtain flush cost: $O(\sqrt{B}/B)$ amortized.

## Van Emde Boas Layout

This is under Cache-Oblivious Model, which is similar to the External Memory Model except that the algorithm do not know about the value of $B$ nor $M$.

1. Start with a (perfectly) balanced binary search tree
2. Divide it in half, from top to bottom
3. Recursively layout each of the $\sqrt{n} + 1$ subtrees, starting from the root, then followed by the $\sqrt{n}$ children in order.

Analysis:

1. Look at a level of detail where each subtree has the largest size $< B$
2. Notice that each subtree is stored in at most 2 memory blocks and each subtree has height $\in [1/2 \log B, \log B)$
3. Hence search is at most $2 \times \log N / (1/2 \log B) = O(\log_B N)$

## Cache-Efficient BFS

Setup: undirected graph $G(V, E)$, each adjacency list stored as an array

1. Set $L_{i+1}$ as the neighbors of all nodes in $L_i$
2. Sort $L_{i+1}$ and remove duplicates
3. Remove item in $L_{i+1}$ that exist in $L_i$ or $L_{i-1}$

Complexity: $O(V + \frac{E}{B} \log_{M/B} \frac{E}{B})$

Unlikely to improve in dense graph where $|E| > B|V|$ and if adjacency lists are stored separately.

## Cache-Efficient Connected Components

Setup: Graph G consist of an array which stores all edges once.

1. Divide $E$ into two parts: $E_1$ and $E_2$
2. Recursively transform $E_2$ → depth-1 trees
3. Contract $E_1$ : move all edges to $E_2$ to the root node of $E_2$
4. Recursively transform $E_1$ → depth-1 trees
5. Merge $E_2$ into $E_1$.

```
// (a,b) is a directed edge a -> b
contract:
    for each (x,y) in E1:
        if (a,x) is in E2: replace (x,y) with (a,y)
        if (a,y) is in E2: replace (x,y) with (a,x)
merge:
    for each (a,b) in E2:
        if (x,a) in E1: add (x,b) to E1
        else: add (a,b) to E1
```

Both `contract` and `merge` do not change the number of connected components.

To make `contract` and `merge` cache-efficient, we first sort $E_1, E_2$ either by first / second component such that we could do a linear scan to decide whether to replace or not. This incurs $O(sort(E) + E/B) = O(\frac{E}{B} \log_{M/B} \frac{E}{B})$

So the algorithm satisfies $T(E) = 2T(E/2) + O(sort(E))$ → $O(sort(E)\log E)$

## Cache-Efficient Minimum Spanning Tree

Similar to CC (above)

1. Divide $E$ into two parts: small $E_1$ and big $E_2$ based on median weight $w$
2. Recursively find MST $T_1$ of $E_1$ as every edge in $T_1$ is in the MST of $G$.
3. Contract $E_1$ :
   1. Make a copy of $T_1$
   2. transform $T_1$ → depth-1 trees,
   3. If 2 nodes in $E_2$ are in the same connected, replace the edge in a similar manner to CC's `contract` but swap $E_1$ and $E_2$.
   4. Tag every edge that is replaced with info about its "ORIGIN"
4. Recursively find MST $T_2$ of $E_2$
5. Expand $E_2$: revert all replaced edge with its "ORIGIN".
6. Return $T_1 \cup T_2$

Time: $O(sort(E)\log^2 \frac{E}{M})$

# Parallel Algorithms (11, 13)

## Fork-Join Model and Bounds

#### PRAM Model

Assume that each processor is connected to some memory modules. Algorithms are designed for a specific number of processor.

#### Fork-Join Model

We rely an (almost) optimal scheduler that can assign work to $p$ processors evenly.

Some example of such scheduler:

1. Greedy Scheduler: centralized, tries to execute as many tasks as possible at any time.
2. Work-Stealing Scheduler: each process keeps a queue of tasks to work on, if queue is empty, try to steal from another process's queue at random.

Metric: Work & Span

Let $T_j$ be the amount of (wall-clock) time algo takes on $j$ processor

**Brent's Theorem.** $\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$

Proof:

- Model work as DAG where each node is a unit of computation and draw a directed arc $u \rightarrow v$ if computation $u$ is required as an input of $v$.
- Operations in different layers of a DAG cannot be computed in parallel.
- Total work : $T_1$. Span is $T_\infty$. Parallelism: $T_1/T_\infty$

Example:

- Sequential sum: $(((a_1 + a_2) + a_3) + a_4 + \ldots$ → $O(n)$
- Parallel sum: $(((a_1 + a_2) + (a_3 + a_4)) + \ldots$ → $O(\frac{n}{p} + \log n)$

## Parallel Sort

```
pMergeSort(A)
    if (n = 1) return
    x = fork pMergesort(A[1..n/2])
    y = fork pMergesort(A[n/2+1, n])
    sync;
    pMerge(X,Y);

pMerge(A[1..k], B[1..n-k], C[1..n]) // assume k > n/2
    // handle base case
    binary search for j s.t. B[j] <= A[k/2] <= B[j+1]
    fork pMerge(A[1..k/2], B[1..j], C[1..k/2+j])
    fork pMerge(A[k/2+1..k], B[j+1..n-k], C[k/2+j+1..n]
    sync;
```

pMerge → Work: $O(n)$, Span: $O(\log^2 n)$
pMergeSort → Work : $O(n \log n)$, Span: $O(\log^3 n)$

## Parallel Set

Support: insert, delete, divide(equal split), union, subtraction, set difference

Backing DS: (2,4) tree that supports `split(T,k) -> (T1, T2, k or null)`, `join(T1, T2)` where all elts in T1 < T2, `root(T)`, `insert(T,x)`

Work: $O(\log n + \log m)$, Span: $O(\log n + \log m)$

### union / subtraction / set difference

Algo :

- Split T2 based on root(T1)
- Recursively solve both left and right subtrees in parallel
- Join + add root(T1) if needed

Work: $O(n \log m)$, Span: $O(\log^2 n)$ where $|T_1| = n; |T_2| = m; n > m$.

## Parallel BFS

```
parBFS(G, start):
    F = {start}, D = {}
    while F not empty:
        D = Union(D, F)
```

```
        F = ProcessFrontier(F) // divide and conquer
        F = SetSubtraction(F, D)
```

Work: $O(m\log^2 n)$, Span: $O(D\log^3 m)$ where $m$ = number of edges and $n$ number of nodes.

## Map-Reduce

Model: data in KV pairs, stored in distributed file system, relies on a scheduler to assign Map and Reduce processes to nodes.

Round:

1. Map: process each KV pair, stateless
2. Shuffle : Group items by key
3. Reduce : process items with same key together

Metric: Number of rounds

Bottleneck: communication cost / shuffling data around

**Efficient Map-Reduce**. Each map/reduce functions must satisfy:

- run in polynomial time.
- use sublinear memory in the size of the problem, e.g. $< O(\sqrt{n})$ memory
- process sublinear number of KV pairs and each pair should be $O(\mathrm{polylog} n)$

Example:

1. Array, compute square of odd & even-indexed elements
   - Map: $(i, A[i]) \to (i \mod 2, A[i]^2)$
   - Reduce: return (key, sum of values)
2. Word Count:
   - Map: (idx, word) → (word, 1)
   - Reduce: return (word, sum of values)
   - Problem: reduce might take in too many values. Solution: since reduce function is **associative**, scheduler can call reduce function on fewer keys at a time.
3. Semijoin
   - Input: $A = [(k_A, v_A), \ldots], B = [k_B, \ldots]$, select all $v_A$ with $k_A \in B$.
   - Map: $(A, (k_A, v_A)) \to (k_A, v_A)$, $(B, k_B) \to (k_B, \mathrm{BVALUE})$
   - Reduce: return (key, value) if BVALUE amongst values except BVALUEs
   - Problem: Reduce not associative. Solution: process values in a stream and make sure BVALUE appears first.
4. Sorting / Bucket Sorting
   - Map: $(k, v) \to (j, v)$ where $j$ is the bucket index. (for regular sorting, just use $k$ instead)
   - Reduce: return (key * number of buckets + idx in bucket, value)
   - Reasonable if values are well distributed or number of buckets is large, say $> O(\sqrt{n})$

## Bellman-Ford

Parallelize each round of relaxation

- Input: (nodeID, (nodeID, est, nbrIDs, nbrWeights)). Note we could also store the graph as a list of edges instead.
- Map: (nodeID, info) → return (nodeID, info) and (nbrID[i], info.estimate + nbrWeight[i])
- Reduce: (nodeID, info) + (nodeID, estimates) → (nodeID, info) with the estimates updated.
- Reasonable if degree is not too large. Not associative. Process edges in streaming fashion.

Running time: $N$ map-reduce rounds (without early termination) or $2D$ map-reduce rounds (with early termination).

## Page Rank

- Input: (nodeID, (nodeID, est, nbrIDs)). Each est is initialized to $1/n$. est is the probability that a random walk will end up at the node after $t$ steps.
- Map: (nodeID, info) → return (nodeID, info) and (nbrID[i], info.estimate / nbrID.len)
- Reduce: (nodeID, info) + (nodeID, estimates) → (nodeID, info) with the estimates updated.

Depends on the mixing time of the graph. For random graphs / cliques: $O(\log n)$. Worst case is $O(n^3)$

---

1. Berkeley CS186 Notes > External Sort↩
2. MIT 6.851, Funnelsort - Wikipedia↩