Coupling:

Method

- Run two copies of Markov chain X_t, Y_t
- Each considered in isolation is a copy of MC (that is, both have MC distribution)
- **but** they are not independent: they make dependent choices at each step
- in fact, after a while they are almost certainly the **same**
- Start Y_t in stationary distribution, X_t anywhere
- Coupling argument:

$$\begin{aligned}\Pr[X_t = j] &= \Pr[X_t = j \mid X_t = Y_t] \Pr[X_t = Y_t] + \Pr[X_t = j \mid X_t \neq Y_t] \Pr[X_t \neq Y_t] \\ &= \Pr[Y_t = j] \Pr[X_t = Y_t] + \epsilon \Pr[X_t = j \mid X_t \neq Y_t]\end{aligned}$$

  So just need to make ϵ (which is r.p.d.) small enough.

n-bit Hypercube walk: at each step, flip random bit to random value

- At step t, pick a random bit b, random value v
- both chains set but b to value v
- after $O(n \log n)$ steps, probably all bits matched.

Counting k colorings when $k > 2\Delta + 1$

- The reduction from (approximate) uniform generation
  - compute ratio of coloring of G to coloring of $G - e$
  - Recurse counting $G - e$ colorings
  - Base case k^n colorings of empty graph
- Bounding the ratio:
  - note $G - e$ colorings outnumber G colorings
  - By how much? Let L colorings in difference (u and v same color)
  - to make an L coloring a G coloring, change u to one of $k - \Delta = \Delta + 1$ legal colors
  - Each G-coloring arises at most one way from this
  - So each L coloring has at least $\Delta + 1$ neighbors unique to them
  - So L is $1/(\Delta + 1)$ fraction of G.
  - So can estimate ratio with few samples
- The chain:

- Pick random vertex, random color, try to recolor
- loops, so aperiodic
- Chain is time-reversible, so uniform distribution.

- Coupling:
  - choose random vertex v (same for both)
  - based on X_t and Y_t, choose bijection of colors
  - choose random color c
  - apply c to v in X_t (if can), $g(c)$ to v in Y_t (if can).
  - What bijection?
    * Let A be vertices that agree in color, D that disagree.
    * if $v \in D$, let g be identity
    * if $v \in A$, let N be neighbors of v
    * let C_X be colors that N has in X but not Y (X can't use them at v)
    * let C_Y similar, wlog larger than C_X
    * g should swap each C_X with some C_Y, leave other colors fixed. **Result:** if X doesn't change, Y doesn't

- Convergence:
  - Let $d'(v)$ be number of neighbors of v in opposite set, so

$$\sum_{v \in A} d'(v) = \sum_{v \in D} d'(v) = m'$$

  - Let $\delta = |D|$
  - Note at each step, δ changes by $0, \pm 1$
  - When does it increase?
    * v must be in A, but move to D
    * happens if only one MC accepts new color
    * If c not in C_X or C_Y, then $g(c) = c$ and both change
    * If $c \in C_X$, then $g(c) \in C_Y$ so neither moves
    * So must have $c \in C_Y$
    * But $|C_Y| \le d'(v)$, so probability this happens is

$$\sum_{v \in A} \frac{1}{n} \cdot \frac{d'(v)}{k} = \frac{m'}{kn}$$

  - When does it decrease?
    * must have $v \in D$, only one moves

    * sufficient that pick color not in either neighborhood of v,
    * total neighborhood size 2Δ, but that counts the $d'(v)$ elements of A twice.
    * so Prob.

$$\sum_{v \in D} \frac{1}{n} \cdot \frac{k - (2\Delta - d'(v))}{k} = \frac{k - 2\Delta}{kn}\delta + \frac{m'}{kn}$$

  - Deduce that expected *change* in δ is difference of above, namely

$$-\frac{k - 2\Delta}{kn}\delta = -a\delta.$$

  - So after t steps, $E[\delta_t] \le (1-a)^t \delta_0 \le (1-a)^t n$.
  - Thus, probability $\delta > 0$ at most $(1-a)^t n$.
  - But now note $a > 1/n^2$, so $n^2 \log n$ steps reduce to one over polynomial chance.

Note: couple depends on state, but who cares

- From worm's eye view, each chain is random walk
- so, all arguments hold

Counting vs. generating:

- we showed that by generating, can count
- by counting, can generate:

Parallel Algorithms

PRAM

- P processors, each with a RAM, local registers
- global memory of M locations
- each processor can in one step do a RAM op or read/write to one global memory location
- synchronous parallel steps
- various conflict resolutions (CREW, EREW, CRCW)
- not realistic, but explores "degree of parallelism"

Randomization in parallel:

- load balancing
- symmetry breaking
- isolating solutions

Classes:

- NC: poly processor, polylog steps
- RNC: with randomization. polylog runtime, monte carlo
- ZNC: las vegas NC
- immune to choice of conflict resolution

Practical observations:

- very little can be done in $o(\log n)$ with poly processors
- lots can be done in $\Theta(\log n)$
- often concerned about *work* which is processors times time
- algorithm is "optimal" if work equals best sequential

Basic operations

- and, or
- counting ones

Sorting

Quicksort in parallel:

- n processors
- each takes one item, compares to splitter
- count number of predecessors less than splitter
- determines location of item in split
- total time $O(\log n)$
- combine: $O(\log n)$ per layer with n processors
- problem: $\Omega(\log^2 n)$ time bound
- problem: $n \log^2 n$ work

Parallel recursion:

- paradigm: reduce problem size from n to $\sqrt{n}$ in $O(\log n)$ time.
- total time $O(\log n + \log \sqrt{n} + \cdots) = O(\log n)$

More processors:

- n^2 processors
- do all comparisons
- count number of items smaller than me: $O(\log n)$
- put into place
- **result:** $O(\log n)$ time with n^2 processors
- or, $O(n)$ time with n processors

BoxSort:

- n processors
- Choose $\sqrt{n}$ random splitters
- sort in $O(\log n)$ time
- insert items in splitters: $O(\log n)$ time
- solve each piece separately, recursively

Intuition:

- expected subproblem size $O(\sqrt{n})$
- so expected time spent on a branch is $O(\log n)$ as above
- problem: many branches: need high probability result.
- solution: analyze each path, show $O(\log n)$ time whp
- thus max path is $O(\log n)$

High probability:

- consider item x
- claim splitter within $\alpha\sqrt{n}$ on each side
- since prob. not at most $(1 - \alpha\sqrt{n}/n)^{\sqrt{n}} \leq e^{-\alpha}$
- fix $\gamma, d < 1/\gamma$
- define $\tau_k = d^k$
- define $\rho_k = n^{\gamma^k}$
- note size ρ_k problem takes $\gamma^k \log n$ time
- argue at most d^k size-ρ_k problems whp

- deduce runtime $\sum d^k \gamma_k = \sum (d\gamma)^k \log n = O(\log n)$
- note: as problem shrinks, allowing more divergence in quantity for whp result
- minor detail: "whp" dies for small problems
- OK: if problem size $\log n$, finish in $\log n$ time with $\log n$ processors