COMP251: Randomized Algorithms

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Kleinberg & Tardos, 2006)
Randomization

Algorithmic design patterns.
- Greedy.
- Divide-and-conquer.
- Dynamic programming.
- Network flow.
- Randomization. in practice, access to a pseudo-random number generator

Randomization. Allow fair coin flip in unit time.

Why randomize? Can lead to simplest, fastest, or only known algorithm for a particular problem.

Ex. Symmetry breaking protocols, graph algorithms, quicksort, hashing, load balancing, Monte Carlo integration, cryptography.
Global minimum cut

**Global min cut.** Given a connected, undirected graph \( G = (V, E) \), find a cut \((A, B)\) of minimum cardinality.

**Applications.** Partitioning items in a database, identify clusters of related documents, network reliability, network design, circuit design, TSP solvers.

**Network flow solution.**
- Replace every edge \((u, v)\) with two antiparallel edges \((u, v)\) and \((v, u)\).
- Pick some vertex \(s\) and compute min \(s\)-\(v\) cut separating \(s\) from each other vertex \(v \in V\).

**False intuition.** Global min-cut is harder than min \(s\)-\(t\) cut.
**Contraction algorithm**

**Contraction algorithm.** [Karger 1995]

- Pick an edge \( e = (u, v) \) uniformly at random.
- **Contract** edge \( e \).
  - replace \( u \) and \( v \) by single new super-node \( w \)
  - preserve edges, updating endpoints of \( u \) and \( v \) to \( w \)
  - keep parallel edges, but delete self-loops
- Repeat until graph has just two nodes \( v_1 \) and \( v_1 \).
- Return the cut (all nodes that were contracted to form \( v_1 \)).

![Diagram](image-url)
Contraction algorithm

**Contraction algorithm.** [Karger 1995]

- Pick an edge $e = (u, v)$ uniformly at random.
- **Contract** edge $e$.
  - replace $u$ and $v$ by single new super-node $w$
  - preserve edges, updating endpoints of $u$ and $v$ to $w$
  - keep parallel edges, but delete self-loops
- Repeat until graph has just two nodes $v_1$ and $v_1$.
- Return the cut (all nodes that were contracted to form $v_1$).

Reference: Thore Husfeldt
Contraction algorithm

Claim. The contraction algorithm returns a min cut with prob $\geq 2 / n^2$.

Pf. Consider a global min-cut $(A^*, B^*)$ of $G$.

- Let $F^*$ be edges with one endpoint in $A^*$ and the other in $B^*$.
- Let $k = |F^*| = \text{size of min cut}$.
- In first step, algorithm contracts an edge in $F^*$ probability $k / |E|$.
- Every node has degree $\geq k$ since otherwise $(A^*, B^*)$ would not be a min-cut $\Rightarrow |E| \geq \frac{1}{2} k n$.
- Thus, algorithm contracts an edge in $F^*$ with probability $\leq 2 / n$. 

![Diagram of min-cut](image-url)
**Contraction algorithm**

**Claim.** The contraction algorithm returns a min cut with prob $\geq 2 / n^2$.

**Pf.** Consider a global min-cut $(A^*, B^*)$ of $G$.

- Let $F^*$ be edges with one endpoint in $A^*$ and the other in $B^*$.
- Let $k = |F^*| = \text{size of min cut}$. 
- Let $G'$ be graph after $j$ iterations. There are $n' = n - j$ supernodes.
- Suppose no edge in $F^*$ has been contracted. The min-cut in $G'$ is still $k$.
- Since value of min-cut is $k$, $|E'| \geq \frac{1}{2} kn'$.
- Thus, algorithm contracts an edge in $F^*$ with probability $\leq 2 / n'$.
- Let $E_j$ = event that an edge in $F^*$ is not contracted in iteration $j$.

\[
\Pr[E_1 \cap E_2 \cdots \cap E_{n-2}] = \Pr[E_1] \times \Pr[E_2 \mid E_1] \times \cdots \times \Pr[E_{n-2} \mid E_1 \cap E_2 \cdots \cap E_{n-3}] \\
\geq (1 - \frac{2}{n}) \left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{4}\right) \left(1 - \frac{2}{3}\right) \\
= \left(\frac{n-2}{n}\right) \left(\frac{n-3}{n-1}\right) \cdots \left(\frac{2}{4}\right) \left(\frac{1}{3}\right) \\
= \frac{2^{n-2}}{n(n-1)} \cdots \left(\frac{2}{4}\right) \left(\frac{1}{3}\right) \\
\geq \frac{2}{n^2}
\]
**Contraction algorithm**

**Amplification.** To amplify the probability of success, run the contraction algorithm many times.

**Claim.** If we repeat the contraction algorithm $n^2 \ln n$ times, then the probability of failing to find the global min-cut is $\leq 1 / n^2$.

**Pf.** By independence, the probability of failure is at most

$$
\left(1 - \frac{2}{n^2}\right)^{n^2 \ln n} = \left[\left(1 - \frac{2}{n^2}\right)^{\frac{1}{n^2}}\right]^{2 \ln n} \leq \left(e^{-1}\right)^{2 \ln n} = \frac{1}{n^2}
$$

with independent random choices,

$(1 - 1/x)^x \leq 1/e$
Contraction algorithm: example execution

trial 1

trial 2

trial 3

trial 4

trial 5 (finds min cut)

trial 6

Reference: Thore Husfeldt
Global min cut: context

**Remark.** Overall running time is slow since we perform $\Theta(n^2 \log n)$ iterations and each takes $\Omega(m)$ time.

**Improvement.** [Karger-Stein 1996] $O(n^2 \log^3 n)$.
- Early iterations are less risky than later ones: probability of contracting an edge in min cut hits 50% when $n/\sqrt{2}$ nodes remain.
- Run contraction algorithm until $n/\sqrt{2}$ nodes remain.
- Run contraction algorithm **twice** on resulting graph and return best of two cuts.

**Extensions.** Naturally generalizes to handle positive weights.

**Best known.** [Karger 2000] $O(m \log^3 n)$.

faster than best known max flow algorithm or deterministic global min cut algorithm
Maximum 3-satisfiability

Maximum 3-satisfiability. Given a 3-SAT formula, find a truth assignment that satisfies as many clauses as possible.

\[ C_1 = x_2 \lor \overline{x}_3 \lor \overline{x}_4 \]
\[ C_2 = x_2 \lor x_3 \lor \overline{x}_4 \]
\[ C_3 = \overline{x}_1 \lor x_2 \lor x_4 \]
\[ C_4 = \overline{x}_1 \lor \overline{x}_2 \lor x_3 \]
\[ C_5 = x_1 \lor x_2 \lor \overline{x}_4 \]

Remark. \textbf{NP}-hard search problem.

Simple idea. Flip a coin, and set each variable true with probability \( \frac{1}{2} \), independently for each variable.
Maximum 3-satisfiability: analysis

**Claim.** Given a 3-SAT formula with $k$ clauses, the expected number of clauses satisfied by a random assignment is $7k/8$.

**Pf.** Consider random variable 

$$Z_j = \begin{cases} 
1 & \text{if clause } C_j \text{ is satisfied} \\
0 & \text{otherwise.} 
\end{cases}$$

- Let $Z$ = weight of clauses satisfied by assignment $Z_j$.

$$E[Z] = \sum_{j=1}^{k} E[Z_j]$$

(linearity of expectation)

$$= \sum_{j=1}^{k} \Pr[\text{clause } C_j \text{ is satisfied}]$$

$$= \frac{7}{8} k$$
The Probabilistic Method

Corollary. For any instance of 3-SAT, there exists a truth assignment that satisfies at least a $7/8$ fraction of all clauses.

Pf. Random variable is at least its expectation some of the time. □

Probabilistic method. [Paul Erdös] Prove the existence of a non-obvious property by showing that a random construction produces it with positive probability!
**Maximum 3-satisfiability: analysis**

**Q.** Can we turn this idea into a 7/8-approximation algorithm?

**A.** Yes (but a random variable can almost always be below its mean).

**Lemma.** The probability that a random assignment satisfies \( \geq 7k / 8 \) clauses is at least \( 1 / (8k) \).

**Pf.** Let \( p_j \) be probability that exactly \( j \) clauses are satisfied; let \( p \) be probability that \( \geq 7k / 8 \) clauses are satisfied.

\[
\frac{7}{8} k = E[Z] = \sum_{j=0} j p_j
\]

\[
= \sum_{j < 7k/8} j p_j + \sum_{j \geq 7k/8} j p_j
\]

\[
\leq \left( \frac{7}{8} k - \frac{1}{8} \right) \sum_{j < 7k/8} p_j + k \sum_{j \geq 7k/8} p_j
\]

\[
\leq \left( \frac{7}{8} k - \frac{1}{8} \right) \cdot 1 + k p
\]

Rearranging terms yields \( p \geq 1 / (8k) \).
Maximum 3-satisfiability: analysis

Johnson's algorithm. Repeatedly generate random truth assignments until one of them satisfies \( \geq 7k/8 \) clauses.

Theorem. Johnson's algorithm is a 7/8-approximation algorithm.

Pf. By previous lemma, each iteration succeeds with probability \( \geq 1/(8k) \). By the waiting-time bound, the expected number of trials to find the satisfying assignment is at most \( 8k \). □
Maximum satisfiability

Extensions.

- Allow one, two, or more literals per clause.
- Find max \textit{weighted} set of satisfied clauses.

\textbf{Theorem.} [Asano-Williamson 2000] There exists a 0.784-approximation algorithm for 3-SAT.

\textbf{Theorem.} [Karloff-Zwick 1997, Zwick+computer 2002] There exists a 7/8-approximation algorithm for version of MAX-3-SAT where each clause has at most 3 literals.

\textbf{Theorem.} [Håstad 1997] Unless P = NP, no $\rho$-approximation algorithm for MAX-3-SAT (and hence MAX-SAT) for any $\rho > 7/8$.

very unlikely to improve over simple randomized algorithm for MAX-3SAT
Monte Carlo vs. Las Vegas algorithms

**Monte Carlo.** Guaranteed to run in poly-time, likely to find correct answer.  
*Ex:* Contraction algorithm for global min cut.

**Las Vegas.** Guaranteed to find correct answer, likely to run in poly-time.  
*Ex:* Randomized quicksort, Johnson's MAX-3-SAT algorithm.

**Remark.** Can always convert a Las Vegas algorithm into Monte Carlo, but no known method (in general) to convert the other way.
RP and ZPP

**RP.** [Monte Carlo] Decision problems solvable with **one-sided error** in poly-time.

**One-sided error.**
- If the correct answer is *no*, always return *no*.
- If the correct answer is *yes*, return *yes* with probability ≥ ½.

**ZPP.** [Las Vegas] Decision problems solvable in **expected** poly-time.

**Theorem.** \( P \subseteq ZPP \subseteq RP \subseteq NP \).

**Fundamental open questions.** To what extent does randomization help?
Does \( P = ZPP \)? Does \( ZPP = RP \)? Does \( RP = NP \)?
QuickSort: Review

Quicksort(A, p, r)
   if p < r then
      q := Partition(A, p, r);
      Quicksort(A, p, q - 1);
      Quicksort(A, q + 1, r)
   fi

Partition(A, p, r)
   x, i := A[r], p - 1;
   for j := p to r - 1 do
      if A[j] ≤ x then
         i := i + 1;
      fi
   od;
   A[i + 1] ↔ A[r];
   return i + 1
Worst-case Partition Analysis

Recursion tree for worst-case partition

Split off a single element at each level:

\[ T(n) = T(n - 1) + T(0) + \text{PartitionTime}(n) \]

\[ = T(n - 1) + \Theta(n) \]

\[ = \sum_{k=1}^{n} \Theta(k) \]

\[ = \Theta\left(\sum_{k=1}^{n} k\right) \]

\[ = \Theta(n^2) \]
Best-case Partitioning

- Each subproblem size $\leq n/2$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$
  - $= 2T(n/2) + \Theta(n)$
- $T(n) = \Theta(n \lg n)$
Variations

• Quicksort is not very efficient on small lists.

• This is a problem because Quicksort will be called on lots of small lists.

• **Fix 1:** Use Insertion Sort on small problems.

• **Fix 2:** Leave small problems unsorted. Fix with one final Insertion Sort at end.
  
  – **Note:** Insertion Sort is very fast on almost-sorted lists.
Unbalanced Partition Analysis

What happens if we get poorly-balanced partitions, e.g., something like: \( T(n) \leq T(9n/10) + T(n/10) + \Theta(n) \)?

Still get \( \Theta(n \lg n) \)!! (As long as the split is of constant proportionality.)

**Intuition:** Can divide \( n \) by \( c > 1 \) only \( \Theta(\lg n) \) times before getting 1.

\[
\begin{align*}
    n & \downarrow \\
    \frac{n}{c} & \downarrow \\
    \frac{n}{c^2} & \downarrow \\
    \vdots & \downarrow \\
    \frac{n}{c \log_c n} & = 1
\end{align*}
\]

Roughly \( \log_c n \) levels; Cost per level is \( O(n) \).

(\textbf{Remember:} Different base logs are related by a constant.)
Intuition for the Average Case

• Partitioning is unlikely to happen in the same way at every level.
  – Split ratio is different for different levels.
    (Contrary to our assumption in the previous slide.)
• Partition produces a mix of “good” and “bad” splits, distributed randomly in the recursion tree.
• What is the running time likely to be in such a case?
Intuition for the average case

Bad split followed by a good split:
Produces subarrays of sizes 0, \((n - 1)/2 - 1\), and \((n - 1)/2\).
Cost of partitioning:
\[ \Theta(n) + \Theta(n-1) = \Theta(n). \]

Good split at the first level:
Produces two subarrays of size \((n - 1)/2\).
Cost of partitioning:
\[ \Theta(n). \]

Situation at the end of case 1 is not worse than that at the end of case 2.
When splits alternate between good and bad, the cost of bad split can be absorbed into the cost of good split.
Thus, running time is \(O(n \lg n)\), though with larger hidden constants.
Randomized Quicksort

- Want to make running time independent of input ordering.

- **How can we do that?**
  - Make the algorithm randomized.
  - Make every possible input equally likely.
    - Can randomly shuffle to permute the entire array.
    - For quicksort, it is sufficient if we can ensure that every element is equally likely to be the *pivot*.
    - So, we choose an element in $A[p..r]$ and exchange it with $A[r]$.
    - Because the *pivot* is randomly chosen, we expect the partitioning to be well balanced on average.
Variations (Continued)

• Input distribution may not be uniformly random.

• **Fix 1:** Use “randomly” selected pivot.
  – We’ll analyze this in detail.

• **Fix 2:** Median-of-three Quicksort.
  – Use median of three fixed elements (say, the first, middle, and last) as the pivot.
  – To get $O(n^2)$ behavior, we must continually be unlucky to see that two out of the three elements examined are among the largest or smallest of their sets.
Randomized Version

Want to make running time independent of input ordering.

Randomized-Partition(A, p, r)
  i := Random(p, r);
  A[r] ↔ A[i];
  Partition(A, p, r)

Randomized-Quicksort(A, p, r)
  if p < r then
    q := Randomized-Partition(A, p, r);
    Randomized-Quicksort(A, p, q – 1);
    Randomized-Quicksort(A, q + 1, r)
  fi