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, identifying 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 \( s \rightarrow v \) cut separating \( s \) from each other vertex \( v \in V \).

False intuition. Global min-cut is harder than min \( s \rightarrow v \) 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 \( u \) and \( v \).
- Return the cut (all nodes that were contracted to form \( v \)).

Randomization

Algorithmic design patterns.
- Greedy.
- Divide-and-conquer.
- Dynamic programming.
- Network flow.
- Randomization.

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.

Reference: Thore Husfeldt
Contraction algorithm

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

Proof. Consider a global min-cut $(P, P')$ of $G$.
* Let $E$ be edges with one endpoint in $P$ and the other in $P'$.
* Let $k = |E|$ be size of min-cut.
* Suppose no edge in $E$ has been contracted. The min-cut in $G'$ is still $k$.
* Thus, algorithm contracts an edge in $E$ with probability $\geq 2/n^2$.

Let $E_i$ event that an edge in $E$ is not contracted in iteration $i$.

$$\Pr[E_i \cap E_j] = \Pr[E_i] \times \Pr[E_j] \times \ldots \times \Pr[E_k] \cap \cap E_k]$$

$$\geq \left(\frac{2}{n^2}\right)^k$$

$$= \frac{2^k}{n^{2k}}$$

So probability of failure is at most:

$$\left(\frac{1}{2}\right)^n$$

$$\leq \frac{1}{n^{2k}}$$

Contraction algorithm: example execution

Global min cut: context

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

Improvement. [Karger-Stein 1996] $O(n \log^3 n)$.
* Early iterations are more likely than later ones: probability of contracting an edge in min cut hits 50% when $\leq 2$ nodes remain.
* Run contraction algorithm until $\leq 2$ 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] $O(n \log^4 n)$.

Maximum 3-satisfiability

Claim. Given a 3-SAT formula with $n$ variables, the expected number of clauses satisfied by a random assignment is $\frac{3n}{8}$.

Proof. Consider random variable $X_i$.

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

Let $Z$ weight of clauses satisfied by assignment $X$.

$$E[Z] = \frac{1}{2^m} \sum_{i=1}^{m} X_i$$

$$= \frac{1}{2^m} \sum_{i=1}^{m} \Pr[C_i \text{ is satisfied}]$$

$$\leq \frac{3n}{8}$$
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 Erdős] Prove the existence of a non-obvious property by showing that a random construction produces it with positive probability!

Maximum 3-satisfiability: analysis

Johnson’s algorithm. Repeatedly generate random truth assignments until one of them satisfies at least 7/8 clauses.

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

Pf. By previous lemma, each iteration succeeds with probability ≥ 1/2. By the waiting-time bound, the expected number of trials to find the satisfying assignment is at most 16. •

Maximum 3-satisfiability: analysis

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.

Maximum satisfiability

Extensions.

• Allow one, two, or more literals per clause.
• Find max weighted set of satisfied clauses.


Theorem. [Karloff-Zwick 1997, Zwick-computer 2002] There exists a 7-approximation algorithm for version of Max-3-Sat where each clause has at most 3 literals.

Theorem. [Hastad 1997] Unless P = NP, no p-approximation algorithm for Max-3-Sat (and hence Max-Sat) for any ρ > 7/8.

RP and ZPP


One-sided error:

• If the correct answer is yes, always return yes.
• If the correct answer is no, return yes with probability ≤ 1/2.

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

Theorem. P ⊆ ZPP ⊆ RP ⊆ 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
    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) + PartitionTime(n)
      = T(n−1) + Θ(n)
      = Σi=1 to n Θ(k)
      = Θ(n²)

Best-case Partitioning

- Each subproblem size ≤ n/2.
- Recurrence for running time
  - T(n) = 2T(n/2) + Θ(n)
  - T(n) = Θ(n log n)

Unbalanced Partition Analysis

What happens if we get poorly-balanced partitions,
e.g., something like: T(n) = T(n/10) + T(n/10) + Θ(n)?
Still get Θ(n log n)!! (As long as the split is of constant proportionality.)

Intuition: Can divide n by c > 1 only Θ(log n) times before getting 1.

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?

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.

Fixes

- Each subproblem size ≤ n/2.
- Recurrence for running time
  - T(n) = 2T(n/2) + Θ(n)
  - T(n) = Θ(n log n)

Intuition: Can divide n by c > 1 only Θ(log n) times before getting 1.

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?

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.

Fixes

- Each subproblem size ≤ n/2.
- Recurrence for running time
  - T(n) = 2T(n/2) + Θ(n)
  - T(n) = Θ(n log n)
Intuition for the average case

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 \( O(n \log n) \), 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.

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
```

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.