CSE 103: Probability and statistics **Winter 2010**

Topic 4 — Randomized algorithms, II

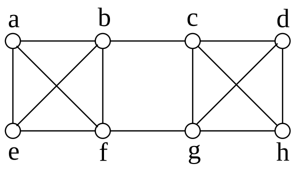
4.1 Karger's minimum cut algorithm

4.1.1 Clustering via graph cuts

Suppose a mail order company has the resources to prepare two different versions of its catalog, and it wishes to target each version towards a particular sector of its customer base. The data it has is a list of its regular customers, along with their purchase histories. How should this set of customers be partitioned into two coherent groups?

One way to do this is to create a graph with a node for each of the regular customers, and an edge between any two customers whose purchase patterns are similar. The goal is then to divide the nodes into two pieces which have very few edges between them.

More formally, the *minimum cut* of an undirected graph $G = (V, E)$ is a partition of the nodes into two groups V_1 and V_2 (that is, $V = V_1 \cup V_2$ and, $V_1 \cap V_2 = \emptyset$), so that the number of edges between V_1 and V_2 is minimized. In the graph below, for instance, the minimum cut has size two and partitions the nodes into $V_1 = \{a, b, e, f\}$ and $V_2 = \{c, d, g, h\}$.



4.1.2 Karger's algorithm

Here's a randomized algorithm for finding the minimum cut:

- Repeat until just two nodes remain:
  - Pick an edge of G at random and collapse its two endpoints into a single node
- For the two remaining nodes u_1 and u_2, set $V_1 = \{\text{nodes that went into } u_1\}$ and $V_2 = \{\text{nodes in } u_2\}$

An example is shown in Figure 4.1. Notice how some nodes end up having multiple edges between them.

4.1.3 Analysis

Karger's algorithm returns the minimum cut with a certain probability. To analyze it, let's go through a succession of key facts.

Fact 1. *If degree(u) denotes the number of edges touching node u, then*

$$\sum_{u \in V} \textit{degree}(u) = 2|E|.$$

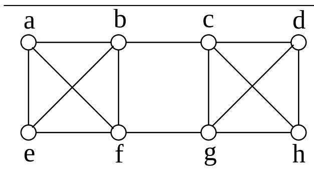
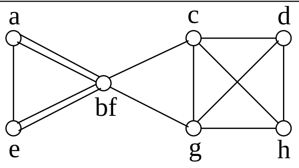
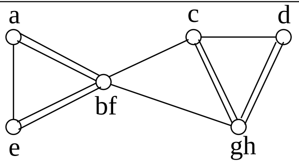
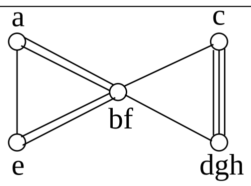
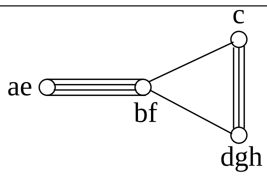
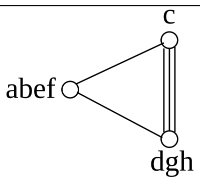
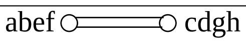
	14 edges to choose from Pick $b - f$ (probability 1/14)
	13 edges to choose from Pick $g - h$ (probability 1/13)
	12 edges to choose from Pick $d - gh$ (probability 1/6)
	10 edges to choose from Pick $a - e$ (probability 1/10)
	9 edges to choose from Pick $ab - ef$ (probability 4/9)
	5 edges to choose from Pick $c - dgh$ (probability 3/5)
	Done: just two nodes remain

Figure 4.1. Karger's algorithm at work.

To see this, imagine the following experiment: for each node, list all the edges touching it. The number of edges in this list is exactly the left-hand sum. But each edge appears exactly twice in it, once for each endpoint.

Fact 2. *If there are n nodes, then the average degree of a node is $2|E|/n$.*

This is a straightforward calculation: when you pick a node X at random,

$$\mathbb{E}[\text{degree}(X)] \;=\; \sum_{u \in V} \Pr(X = u)\text{degree}(u) \;=\; \frac{1}{n}\sum_u \text{degree}(u) \;=\; \frac{2|E|}{n}$$

where the last step uses the first Fact.

Fact 3. *The size of the minimum cut is at most $2|E|/n$.*

Consider the partition of V into two pieces, one containing a single node u, and the other containing the remaining $n-1$ nodes. The size of this cut is degree(u). Since this is a valid cut, the minimum cut cannot be bigger than this. In other words, for all nodes u,

$$(\text{size of minimum cut}) \leq \text{degree}(u).$$

This means that the size of the minimum cut is also $\leq$ the average degree, which we've seen is $2|E|/n$.

Fact 4. *If an edge is picked at random, the probability that it lies across the minimum cut is at most $2/n$.*

This is because there are $|E|$ edges to choose from, and at most $2|E|/n$ of them are in the minimum cut.

Now we have all the information we need to analyze Karger's algorithm. It returns the right answer *as long as it never picks an edge across the minimum cut.* If it always picks a non-cut edge, then this edge will connect two nodes on the same side of the cut, and so it is okay to collapse them together.

Each time an edge is collapsed, the number of nodes decreases by 1. Therefore,

$$\begin{aligned}
\Pr(\text{final cut is the minimum cut}) \;&=\; \Pr(\text{first selected edge is not in mincut}) \times \\
&\qquad \Pr(\text{second selected edge is not in mincut}) \times \cdots \\
&\geq\; \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) \\
&=\; \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdots\frac{2}{4}\cdot\frac{1}{3} \\
&=\; \frac{2}{n(n-1)}.
\end{aligned}$$

The last equation comes from noticing that almost every numerator cancels with the denominator two fractions down the line.

Karger's algorithm succeeds with probabililty $p \geq 2/n^2$. Therefore, it should be run $\Omega(n^2)$ times, after which the smallest cut found should be chosen.

Those who are familiar with minimum spanning tree algorithms might be curious to hear that another way to implement Karger's algorithm is the following:

- Assign each edge a random weight
- Run Kruskal's algorithm to get the minimum spanning tree
- Break the largest edge in the tree to get the two clusters

(Do you see why?) Over the decades, the running time of Kruskal's algorithm has been thoroughly optimized via special data structures. Now this same technology can be put to work for cuts!

4.2 Two types of randomized algorithms

Our algorithms for finding percentiles and for sorting are guaranteed to return the correct answer. But if you run them multiple times on the same input, their running times will fluctuate, even though the answer will be the same every time. Therefore we are interested in their *expected* running time. We call these *Las Vegas algorithms.*

But in the case of minimum cut we saw another type of algorithm – called a *Monte Carlo algorithm* – that always has the same running time on any given input, but is not guaranteed to return the correct answer. It merely guarantees that it has some probability $p > 0$ of being correct. Therefore, if you run it multiple times on an input, you'll get many different answers, of which roughly a p fraction will be correct. In many cases, it is possible to look through these answers and figure out which one(s) are right.

In a Monte Carlo algorithm, how much does the probability of success increase if you run it multiple times, say k times?

$$\Pr(\text{wrong every time}) \; = \; (1-p)^k \; \leq \; e^{-pk}.$$

To make this less than some δ, it is enough to run the algorithm $(1/p)\log(1/\delta)$ times. For instance, to make the failure probability less than 1 in a million, just run the algorithm $20/p$ times (a million is roughly 2^{20}).

4.3 Hashing

In many situations, such as a dictionary application, we need to store a vast collection of items in such a way that we can look up any item instantaneously. The way to do this is by *hashing.*

4.3.1 The hashing framework

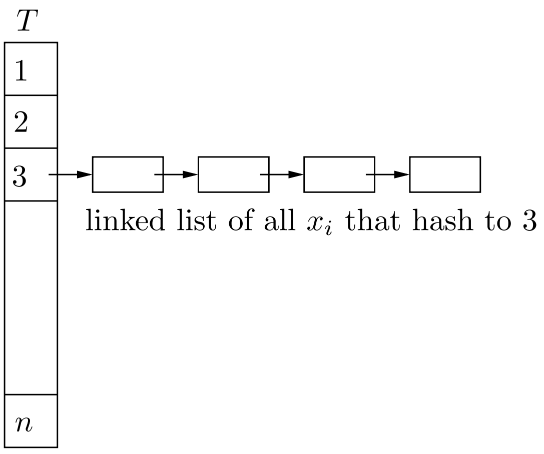
Suppose you have a large collection of items $x_1, \ldots, x_n$ that you want to store (for instance, all English words), where these items are drawn from some set $\mathcal{U}$ (for instance, the set of all conceivable words). The requirements are:

1. The total storage space used should be $O(n)$.
2. Given a query $q \in \mathcal{U}$, it should be possible to *very rapidly* determine whether q is one of the stored items x_i.

4.3.2 A simple solution using randomization

1. Pick a completely random function $h : \mathcal{U} \to \{1, 2, \ldots, n\}$.
   This is the *hash function.*
2. Create a table T of size n, each of whose entries is a pointer to a linked list, initialized to null.
3. Store each x_i in the linked list at $T[h(x_i)]$.
   We say x_i *hashes to* location $h(x_i)$.
4. Given a query q, look through the linked list at $T[h(q)]$ to see if it's there.

Here's a picture of the data structure.



The storage used is $O(n)$. What about the query time?

4.3.3 Average query time

Suppose query q is picked at random, so that it is equally likely to hash to any of the locations $1, 2, \ldots, n$. What is the expected query time?

$$\begin{aligned}
\text{Expected query time} &= \sum_{i=1}^{n} \Pr(q \text{ hashes to location } i) \cdot (\text{length of list at } T[i]) \\
&= \frac{1}{n} \sum_{i} (\text{length of list at } T[i]) \\
&= \frac{1}{n} \cdot n \;=\; 1
\end{aligned}$$

So the average query time is constant!

4.3.4 Worst case query time, and a balls-in-bins problem

What is the worst case query time; that is, what is the length of the longest linked list in T? Equivalently, when you throw n balls in n bins, what is the size of the largest bin? We'll see that with very high probability, no bin gets $\geq \log n$ balls.

For any bin i, let E_i be the event that it gets $\geq \log n$ balls.

$$\Pr(E_i) \;\leq\; \binom{n}{\log n} \left(\frac{1}{n}\right)^{\log n}.$$

(Do you see why?) It turns out, using all sorts of calculations, that this is at most $1/n^2$.

Therefore,

$$\Pr(\text{some bin gets} \geq \log n \text{ balls}) \;=\; \Pr(E_1 \cup E_2 \cup \cdots \cup E_n) \;\leq\; \Pr(E_1) + \cdots + \Pr(E_n) \;\leq\; \frac{1}{n}.$$

For instance, if you throw a million balls into a million bins, then the chance that there is a bin with ≥ 20 balls is at most 1 in a million.

Getting back to hashing, this means that the worst case query time is (with high probability) $O(\log n)$.

4.3.5 The power of two choices

Here's a variant on the balls and bins setup. As usual, you have before you a row of n bins, along with a collection of n identical balls. But now, when throwing each ball, *you pick two bins at random and you put the ball in whichever of them is less full.*

It turns out, using an analysis that is too complicated to get into here, that under this small change, the maximum bin size will be just $O(\log \log n)$ instead of $O(\log n)$.

This inspires an alternative hashing scheme:

1. Pick *two* completely random functions $h_1, h_2 : \mathcal{U} \rightarrow \{1, 2, \ldots, n\}$.
2. Create a table T of size n, each of whose entries is a pointer to a linked list, initialized to null.
3. For each x_i, store it in either the linked list at $T[h_1(x_i)]$ or $T[h_2(x_i)]$, whichever is shorter.
4. Given a query q, look through *both* the linked list at $T[h_1(q)]$ and at $T[h_2(q)]$ to see if it's there.

The storage requirement is still $O(n)$, the average query time is still $O(1)$, but now the worst case query time drops to $O(\log \log n)$.