CS174 Fall 98: Lecture Note 11

Alistair Sinclair, November 1998

Random Walks

We investigate a new algorithmic paradigm called random walk and look at some of its computational applications.

Let $G = (V, E)$ be a connected, undirected graph. A random walk on G, starting from vertex $s \in V$, is the random process defined as follows:

> $u := s$
> **repeat** for T steps
>   choose a neighbor v of u u.a.r.
> $u := v$

Given G, this is obviously trivial to implement on a computer.

For our applications, we will need the following key quantities associated with random walk:

Definitions
H_{uv}: the (expected) hitting time from u to v, i.e., the expected number of steps taken by a random walk starting from u to visit v for the first time.
C_u: the (expected) cover time from u, i.e., the expected number of steps taken by a random walk starting from u to visit all vertices of G.
$C_G = \max_{u \in V} C_u$: the cover time of G.

Application 1: Graph Searching

It is well known that a graph can be searched using, e.g., depth-first search (DFS) or breadth-first search (BFS) in $O(m)$ time and $O(n)$ space, where n, m are the numbers of vertices and edges respectively in G. What if we use random walk to search G? The space here is only $O(1)$, since the above algorithm just needs to remember the current vertex x; i.e., there is no need to keep a large data structure such as a stack or a queue in main memory. What about time? The problem is that the random walk doesn't know when it has searched the whole graph! However, we do know from Markov's inequality and the definition of cover time that

$$\Pr[\text{random walk has not visited all vertices after } 2C_G \text{ steps}] \leq \tfrac{1}{2}.$$

So, if we know an upper bound C on the cover time C_G, we can set $T = 2C$ in the random walk algorithm. When the algorithm terminates, we know that we've searched the whole graph with probability $\geq \frac{1}{2}$.

Ex: Show that setting $T = 200C$ reduces the failure probability from $\frac{1}{2}$ to 2^{-100}. [Note: Markov's inequality does not imply this; you need to argue in terms of repeated trials of the above algorithm.]
$\square$

As we shall see shortly, it turns out that $C_G \leq 2m(n-1)$ for any graph G. Therefore, putting $T = \text{constant} \times mn$ in the random walk algorithm gives us a search procedure that runs in time $O(mn)$ and space $O(1)$ (and searches the whole graph with high probability). This is potentially a useful approach for large graphs when memory is limited. It is also an example of a time-space tradeoff when compared to DFS/BFS.

Application 2: 2SAT

2SAT is the following problem:

Input : A Boolean formula ϕ in conjunctive normal form with exactly two literals in every clause.
E.g., $\phi = (x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2)$.

Question : Is ϕ satisfiable?

There does exist a linear-time deterministic algorithm for 2SAT based on strongly-connected components (as you may have seen in CS170). However, this algorithm is quite complicated. Here is a very simple randomized algorithm, first analyzed by Papadimitriou (recall Note 0):

> start with $x_i = \text{TRUE} \quad \forall i$
> **while** ϕ contains an unsatisfied clause and #steps $< 2n^2$ **do**
>   pick an unsatisfied clause C arbitrarily
>   pick one of the two literals in C u.a.r. and flip its truth value
>   **if** ϕ is now satisfied **then** output "yes" and stop
> output "no"

Here n is the number of variables in ϕ. Note that the choice of clause in line 3 is arbitrary, but the choice of literal in line 4 must be made u.a.r. (This is the only place where randomness enters.)

Obviously this algorithm will always produce the correct answer "no" when ϕ is not satisfiable. What if ϕ is satisfiable? We have to show that, in this case, the algorithm will find an assignment within $2n^2$ random steps with high probability.

We can reduce this to a question about random walks as follows. Suppose ϕ is satisfiable, and let $\mathcal{A}^*$ be any satisfying assignment of ϕ. For any assignment $\mathcal{A}$, define $f(\mathcal{A})$ to be the number of variables of ϕ whose value in $\mathcal{A}$ is the same as their value in $\mathcal{A}^*$. We will represent the state of the algorithm at any time by the value $f(\mathcal{A})$, where $\mathcal{A}$ is the current assignment of the algorithm. The following facts are easy to check:

1. $f(\mathcal{A})$ takes values in the set $\{0, 1, \ldots, n\}$.
2. If $f(\mathcal{A}) = n$ then $\mathcal{A}$ must be $\mathcal{A}^*$.
3. At each step of the algorithm, $f(\mathcal{A})$ either increases by 1 or decreases by 1.
4. At each step, $\Pr[f(\mathcal{A})$ increases by $1] \geq \frac{1}{2}$.

Facts 1 and 2 are trivial. Fact 3 follows because, at each step, we flip the value of exactly one variable. To see Fact 4, note that at least one literal in the chosen clause C must have a different value in $\mathcal{A}$ from its value in $\mathcal{A}^*$ (because in $\mathcal{A}$ both literals in C are false, whereas in $\mathcal{A}^*$ at least one must be true). So if we pick this literal in line 4 of the algorithm (which happens with probability $\frac{1}{2}$), we'll increase $f(\mathcal{A})$ by 1.

The above four facts mean that, until the algorithm stops, $f(\mathcal{A})$ behaves like a random walk on the line graph L_{n+1} with $n+1$ vertices $\{0, 1, \ldots, n\}$ and an edge connecting each pair $\{i, i+1\}$ for $i = 0, 1, \ldots, n-1$. The only difference is that, whereas the random walk moves left or right (i.e., from i to $i+1$ or to $i-1$) with probability exactly $\frac{1}{2}$ each, our process moves right (from i to $i+1$) with probability at least $\frac{1}{2}$ (and left with the remaining probability). This means that our process moves to the right at least as fast as the random walk. The process stops at the vertex n because then the algorithm has found the satisfying assignment $\mathcal{A}^*$. (It may stop earlier if it happens to find another satisfying assignment.)

From this it should be clear that the number of steps required for the algorithm to find a satisfying assignment is stochastically dominated by the hitting time from 0 to n on the line graph L_{n+1}. (Check you believe this.) So the expected number of steps is $\leq H_{0n}$. We shall see shortly that $H_{0n} = n^2$. Therefore, using Markov's inequality, if we run the algorithm for $2n^2$ steps we will fail to find a satisfying assignment with probability $\leq \frac{1}{2}$. This ensures our algorithm is correct with probability at least $\frac{1}{2}$ when ϕ is satisfiable. The number of random walk steps is $O(n^2)$ in the worst case, and often much less in practice when the formula is satisfiable (why?).

Ex: Explain how, in theory, you could compute the probability that the algorithm moves right from any given assignment. $\square$

Ex: Recall from Note 6 that the expected distance of a symmetric random walk from its starting point after n steps is $\sim c\sqrt{n}$, where c is a constant. Is this in line with the fact that $H_{0n} = n^2$? $\square$

Analysis of Random Walks

The above algorithms, and many others, depend on our being able to figure out hitting times and cover times of random walks. We show how to do this using a surprising connection with electrical networks.

View G as an electrical network, with each edge representing a wire of unit resistance. Current can be injected into and removed from nodes by an external source of potential difference (a "battery"). Current flows in the circuit according to:

Kirchoff's Law: the total current flowing into any node is equal to the total current flowing out of it.

Ohm's Law: the potential difference across any wire is equal to the current along it times its resistance.

From these, we can in principle figure out all current flows in the network.

Definition: The effective resistance R_{uv} between nodes u and v is the potential difference between u and v when one unit of current is injected at u and removed at v.

The above laws lead to the familiar formulae for combining effective resistances in series ($R = R_1 + R_2$) and in parallel ($R = \frac{R_1 R_2}{R_1 + R_2}$).

Ex: Show that, for any pair of nodes u, v that are adjacent in G, we have $R_{uv} \leq 1$. When is $R_{uv} = 1$? $\square$

Theorem 1 *Suppose we inject $d(u)$ units of current into each node u, where $d(u)$ is the degree of u, and remove all $\sum_{u \in V} d(u) = 2m$ units of current from v. For each u, let ϕ_{uv} denote the potential difference between u and v in this scenario. Then $\phi_{uv} = H_{uv}$ for all u.*

Proof: We write down two systems of equations, one for the ϕ_{uv} and one for the H_{uv}, and observe that they are identical.

First, considering currents at node u we have

$$d(u) = \sum_{(u,x) \in E} (\text{current from } u \text{ to } x) = \sum_{(u,x) \in E} \phi_{ux} = \sum_{(u,x) \in E} (\phi_{uv} - \phi_{xv}).$$

The first step here is Kirchoff's law, the second is Ohm's law, and the third is a basic property of potential differences. We can rearrange the last expression above, using the fact that $\sum_{(u,x) \in E} \phi_{uv} = d(u)\phi_{uv}$, to get

$$\phi_{uv} = 1 + \sum_{(u,x) \in E} \frac{\phi_{xv}}{d(u)}. \tag{1}$$

Note that (1) is actually a family of equations, one for each vertex u. Together, these equations determine the values of ϕ_{uv} for all u.
Secondly, by considering the first step of a random walk from u we have

$$H_{uv} = 1 + \sum_{(u,x)\in E} \frac{H_{xv}}{d(u)}. \tag{2}$$

(Check you understand this.) Again, there is one such equation for each u, and together they determine all H_{uv}.
Now look at the systems of equations (1) and (2): they are identical! Hence we must have $\phi_{uv} = H_{uv}$ for all u, as claimed. $\square$

The next theorem is a symmetrized version of Theorem 1, which is much cleaner.

Theorem 2 *For all pairs of vertices u, v, we have $H_{uv} + H_{vu} = 2mR_{uv}$.*

Proof: The proof relies heavily on Theorem 1. Call the scenario described there Scenario A. Theorem 1 says that $H_{uv} = \phi_{uv}$, the potential difference between u and v in this scenario.

Now consider Scenario B, which is the same as Scenario A except that the $2m$ units of current are removed from u, not from v. By Theorem 1 again, we see that $H_{vu} = \phi'_{vu}$, the potential difference between $\underline{v \text{ and } u}$ in Scenario B. Next, reverse the directions of all currents in Scenario B, arriving at Scenario C. The potential difference between $\underline{u \text{ and } v}$ in this scenario is $\phi''_{uv} = \phi'_{vu} = H_{vu}$.

Finally, sum the currents in Scenarios A and C. The result is Scenario D, in which $2m$ units of current are injected at u and removed at v (and all other external currents cancel). Since potential differences also sum (because they are a linear function of currents), the potential difference between u and v in Scenario D is $\phi_{uv} + \phi''_{uv} = H_{uv} + H_{vu}$. But by definition of R_{uv} this potential difference is exactly $2mR_{uv}$, since we are driving $2m$ units of current from u to v. $\square$

Examples

1. In the line graph L_n with n vertices $\{0, \ldots, n-1\}$, for $i > j$ we have $R_{ij} = i - j$. (Why?) So $H_{ij} + H_{ji} = 2(n-1)(i-j)$. In particular, for the situation in Application 2 (2SAT) we have, in the graph L_{n+1}, $H_{0n} + H_{n0} = 2n^2$. But by symmetry $H_{0n} = H_{n0} = n^2$, as we promised earlier.

2. The n-vertex lollipop LP_n consists of a chain of length $\frac{n}{2}$ connected at one end to a clique of size $\frac{n}{2}$. Call the vertex of the clique where the chain is attached v, and the vertex at the other end of the tail u. Then obviously $R_{uv} = \frac{n}{2}$, and $m = \frac{n}{2} - 1 + \binom{n/2}{2} = \frac{1}{8}(n-2)(n+4) = \Theta(n^2)$. So $H_{uv} + H_{vu} = 2mR_{uv} = nm = \Theta(n^3)$. But here $H_{uv} = (\frac{n}{2})^2 = \frac{n^2}{4}$, from example 1, because a random walk starting from u looks just like a random walk on $L_{n/2}$ until it hits v. So we see that H_{vu} must be $\Theta(n^3)$.

Ex: Compute H_{ij} on the line L_n using Theorem 1. $\square$

Ex: For random walk on L_{n+1}, it is clear by symmetry that $L_{0n} = L_{n0}$. Is it true that $H_{ij} = H_{ji}$ for all i, j? Justify your answer. $\square$

Ex: Give a direct argument to show that, in the complete graph K_n, $H_{uv} = n - 1$ for all u, v. What can you deduce about R_{uv}? $\square$

Finally, we turn to the question of the cover time.

Theorem 3 *For any graph G, we have $C_G \le 2m(n-1)$.*

Proof: Let T be any spanning tree of G. Starting at some arbitrary vertex u, trace around the outside of T, finishing up back at u again. (Really, you are tracing out the progress of a DFS of T.) As you go, write down the sequence of vertices you encounter along the path: $u = v_0, v_1, v_2, \ldots, v_{2n-3}, v_{2n-2} = u$. Since T has $n-1$ edges and each is traversed twice (once in each direction), the path has length $2(n-1)$. Note also that the number of times a vertex appears on the path is equal to its degree in T (except that u appears one extra time).
Now I claim that

$$C_u \le H_{v_0v_1} + H_{v_1v_2} + \ldots + H_{v_{2n-3}v_{2n-2}}. \tag{3}$$

This is because the time to visit all vertices starting from $u = v_0$ is certainly no more than the time to visit first v_1, then v_2, then v_3 and so on, and finally finish up back at $u = v_{2n-2}$. Now since every edge e of T is traced once in each direction on our path, (3) can be rewritten as

$$C_u \le \sum_{(x,y)\in T} (H_{xy} + H_{yx}) = \sum_{(x,y)\in T} 2mR_{xy} \le 2m(n-1).$$

The second step here follows from Theorem 2, and the final step from the fact that $R_{xy} \le 1$ for adjacent vertices x, y (see Exercise just before Theorem 1). Since the above holds for any vertex u, we have $C_G \le 2m(n-1)$ as claimed. □

Examples

1. For the line L_n, $C_{L_n} \le 2(n-1)^2 = \Theta(n^2)$.
2. For the complete graph K_n, $C_{K_n} \le 2\binom{n}{2}(n-1) = n(n-1)^2 = \Theta(n^3)$.
3. For the lollipop LP_n, $C_{LP_n} = \Theta(n^3)$.

Ex: Show, using our results about hitting times in L_n and LP_n, that the above bounds for C_{L_n} and C_{LP_n} are both tight up to constant factors. □

Ex: Show that the above bound for C_{K_n} is very weak by proving directly that $C_{K_n} = \Theta(n \log n)$. [Hint: use coupon collecting.] □

Ex: Show that both of the following statements are false: (i) adding edges to a graph always decreases its cover time; (ii) adding edges to a graph always increases its cover time. [Hint: think about the three examples above.] □

Memory loss in random walks

So far we've concentrated on the hitting time and the cover time. But random walks have another property that is often very useful in computational applications: if we run the walk for sufficiently many steps, the probability of finding it at any given vertex will converge to a fixed value regardless of where the walk started.
To explain this, we generalize the framework slightly. We allow the edges of our graph G to be directed, and we allow any fixed probability distribution for choosing a neighbor at each step (not just the uniform distribution as we had earlier). More precisely, for each vertex u of G we have a probability P_{uv} of going to each other vertex v (including staying at u), so that $\sum_{v\in V} P_{uv} = 1$. The directed edges of G are just those pairs (u, v) for which $P_{uv} > 0$. The random process is defined as follows:

$u := s$
repeat for T steps
  choose a vertex v with probability P_{uv}
  $u := v$

This process is called a Markov chain with transition probabilities P_{uv}. Note that our earlier random walk is just a special case in which $P_{uv} = \frac{1}{d(u)}$ when $\{u,v\} \in E$ and $P_{uv} = 0$ otherwise.

Now let the vector $p^{(t)}$ be the probability distribution of the Markov chain after t steps: i.e., $p_u^{(t)}$ is defined to be the probability that the process is at vertex u after t steps. Since the process starts at some specific vertex s, $p^{(0)}$ is very simple: $p_s^{(0)} = 1$ and $p_u^{(0)} = 0$ for all $u \neq s$. How does $p^{(t)}$ evolve with time? Well, it is not hard to see that

$$p_v^{(t+1)} = \sum_u p_u^{(t)} P_{uv},$$

i.e., the vector $p^{(t)}$ gets multiplied by the matrix P. (Why?) We are now ready for:

Fundamental Theorem of Markov Chains *Under mild assumptions,*[1] *we have*

$$p^{(t)} \to \pi \quad \text{as } t \to \infty,$$

where π is a fixed probability distribution independent of the starting distribution $p^{(0)}$. Moreover, π is the unique vector satisfying $\pi P = \pi$ and $\sum_u \pi_u = 1$. □

We will not prove this theorem here. The important point is this: if we let a Markov chain run for long enough, then it will eventually lose all memory of where it started and reach some fixed distribution π over the vertices of G. This distribution is called the stationary distribution.

Examples

1. Random walk. Here $\pi_u = \frac{d(u)}{2m}$.
2. In any Markov chain with $P_{uv} = P_{vu}$ for all u, v, we have π uniform (i.e., $\pi_u = \frac{1}{n}$ for all u).

Ex: Check the above examples by showing that the given π satisfies $\pi P = \pi$. □

This brings us to another computational application of random walks (and more general Markov chains): Running the walk for a while and observing the final state gives us a way of randomly sampling from a probability distribution π. This is very useful, for example, in computational physics, where the vertices of our graph are the states of some physical system and we want to sample a "typical" state. There is no way to do this directly because the number of states is huge; but by taking a (hopefully small) number of random Markov chain steps in this space we can achieve the desired sampling. Such computations, often known as Monte Carlo experiments, consume vast numbers of supercomputer cycles every year.

To give the flavor of how such algorithms work, we'll look at a simpler application.

Card shuffling

We are given a deck of k cards (usually $k = 52$). We want to shuffle them — i.e., we want to sample from the uniform distribution over the space of all $k!$ permutations of the deck.[2] What we do is set

[1] The main condition is that the underlying graph G is strongly connected, i.e., there is a path from every vertex to every other vertex.

[2] Of course, in this simple example we happen to know a simple linear time algorithm for doing this (recall HW3). The point here is to analyze a different kind of algorithm that is (a) similar to more realistic examples in physics etc.; and (b) close to what humans do in practice when shuffling cards.

up a Markov chain whose vertices are the $n = k!$ permutations, whose edges correspond to natural shuffling moves, and whose stationary distribution π is uniform. Here are three examples:

1. Random transpositions. Pick two cards at random and swap them. (This is the same as one of the algorithms proposed on page 3 of Note 2.)

2. Top-in-at-random. Take the top card and insert it into the deck at any of the k possible positions u.a.r.

3. Riffle shuffle. See HW8.

Ex: Describe the transition probabilities P_{uv} for the first two examples, as functions of k. $\square$

It is quite easy to check that in all three cases the stationary distribution π is uniform, i.e., $\pi_u = \frac{1}{k!}$ for all permutations u. By the Fundamental Theorem, we therefore know that performing sufficiently many random shuffle steps (in each of the three examples) will give us a random permutation.

In computational applications, the key question is: how many steps are necessary to get us close to π? This quantity is called the mixing time of the Markov chain (or random walk). Notice that the size of the graph in these examples is $n = k!$, which is huge (e.g., $52! \approx 10^{68}$). So for a good algorithm we want that the mixing time is tiny compared to n. (Contrast this with the cover time, which clearly must be at least n, and is usually quite a bit larger.)

We sketch here an argument that the mixing time for top-in-at-random shuffle is $O(k \log k)$, which is indeed tiny compared to $n = k!$.

Claim: The mixing time of the top-in-at-random shuffle is $O(k \log k)$.

Informal Proof: We follow the progress of the card that starts off on the bottom of the deck: call it B. Note that B gradually rises during the shuffle as cards are inserted below it.

I claim the following: at any time during the shuffle, the cards below B are in a completely random order. Why is this? Well, all of these cards must have been inserted there during a shuffle step (why?), and when this was done their position relative to the others was chosen randomly.

Now let the r.v. T be the number of steps until B reaches the top of the deck. Then after $T+1$ steps, the deck is surely random. This follows because after T steps all the other cards in the deck are below B and hence in random order, and at the next step B is inserted into a random position. So what we will do is calculate the expectation $\mathrm{E}(T)$. Intuitively, this should tell us what the mixing time is. (We won't bother to translate this expectation formally into a statement about the mixing time: for this, we'd have to define formally what we mean by "close to π." But with a suitable definition, it turns out that the mixing time is $\leq$ constant $\times\, \mathrm{E}(T)$.)

What is $\mathrm{E}(T)$? Well, let's write $T = T_1 + T_2 + \ldots + T_{k-1}$, where the r.v. T_i is the number of steps taken for B to move up from $i-1$ places above the bottom to i places above the bottom. What is the distribution of T_i? Well, I claim that T_i is just the number of flips of a coin with heads probability $\frac{i}{k}$ until the first heads appears. (Why?) This means that $\mathrm{E}(T_i) = \frac{k}{i}$, and therefore

$$\mathrm{E}(T) = \sum_{i=1}^{k-1} \mathrm{E}(T_i) = \sum_{i=1}^{k-1} \frac{k}{i} \sim k \ln k. \qquad \square$$

The above argument shows that only $O(k \log k)$ top-in-at-random steps are enough to get a well shuffled deck. Similar (rather more complicated) arguments show that the mixing time is also $O(k \log k)$ for random transpositions, and is only $O(\log k)$ for the riffle shuffle.