Design and Analysis of Algorithms
Massachusetts Institute of Technology — 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor — Handout 8

Recitation 8: Random Walks, MCMC, Streaming Algorithms

1 Random Walks Recap

A *random walk* on a graph $G = (V, E)$ of length t is defined as a random process in which we start at some node s and repeat the process t times of choosing randomly among the neighbors v' of the current vertex v we are at and moving to it. If the graph is weighted, we transition to a neighbor v' from v with probability proportional to the weight of the edge.

In this class, we focus on random walks as distributions, namely the probability distribution across the vertices of the walk for a fixed starting point s and known number of steps t. We can mathematically represent the distribution using the expression p_v^t, which is the probability that the walk visits vertex v at step t. We then have the following recurrence.

$$p_v^0 = \begin{cases} 1 & \text{if } v = s \\ 0 & \text{otherwise} \end{cases}$$

$$p_v^{t+1} = \sum_{e \in E, e=(u,v)} \frac{1}{d(u)} p_u^t$$

where $d(u)$ is the degree of u in G. With these definitions, we can formulate random walks as a series of matrix products.

We define the adjacency matrix A of G as follows: $A_{u,v} = 1$ if $(v, u) \in E$ and $A_{u,v} = 0$ if $(v, u) \notin E$. The degree matrix D is a diagonal matrix and has $D_{u,u} = d(u)$. We can define the *walk* matrix as $W = AD^{-1}$ or

$$W_{u,v} = \begin{cases} \frac{1}{d(v)} & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Notice then that $p^{t+1} = Wp^t = W^{t+1}p^0$.

We also define a *stationary distribution* as a distribution π such that $W\pi = \pi$. This distribution π represents a steady state, one that does not change after a random walk step. Some graphs converge to a stationary distribution regardless of the starting state. Recall the following two definitions:

Definition 1 (Strongly connectedness of a Graph) *A graph is strongly connected if every vertex is reachable from every other vertex.*

Definition 2 (Aperiodicity of a Graph) *A graph is* aperiodic *if there is no integer $k > 1$ where k divides the length of very cycle in the graph. In other words, the greatest common divisor of the length of all cycles in the graph is 1. Furthermore, if we can show that the graph has a self-loop (cycles of length 1), then we can show that it is is aperiodic.*

We now present the following theorem:

Theorem 1 (Unique Stationary Distribution of a Graph) *A directed graph converges to a* unique *stationary distribution if it* strongly connected *and* aperiodic.

2 Stationary Distributions of Top-In Shuffling

Our goal of card shuffling is to achieve a permutation of a deck of cads chosen from a uniformly random distribution. We will identify a deck with the ordering of cards within it. Here is one algorithm for shuffling a deck of k cards using n operations:

```
SHUFFLE-DECK($deck, n$)
    for $i = 1$ to $n$
        TOP-IN($deck$)
    return $deck$
```

```
TOP-IN($deck$)
    Take the top card of $deck$ and insert it at one of the $k$ positions in the deck uniformly at random[a].
```

[a]this includes putting the card back on top at the original position

We'd like to show that SHUFFLE-DECK is a good way to shuffle a deck of cards, i.e. that as $n \to \infty$ the probability distribution associated with our deck is uniform over all possible permutations of the cards.

We can do this by considering this problem as a random walk on a graph. Consider the weighted directed graph $G = (V, E, p : E \to \mathcal{R})$ with

- $V =$ the set of all possible permutations of the deck (so $|V| = k!$)
- $p(u, v) = \Pr[\text{TOP-IN}(u) = v]$
- $E = \{(u, v) \mid p(u, v) > 0\} = \{(u, v) \mid v$ is a possible output of TOP-IN$(u)\}$ [1]

Now, it is evident that SHUFFLE-DECK is a random walk on G of length n. Thus if we can show that the directed graph G is strongly-connected and aperiodic, we can show that it converges to a unique stationary distribution.

[1]Note that the existence of an edge (u, v) does not imply that $(v, u) \in E$ because TOP-IN could (for example) put the top card somewhere in the middle of the deck from whence it could not possibly be extracted with a single application of TOP-IN.

We first prove strongly-connectedness. Give the cards in the deck some arbitrary numbering and consider any two orderings $A = [a_1, a_2, ..., a_k]$ and $B = [b_1, b_2, ..., b_k]$ of the cards deck, where the i^{th} card in ordering A is the card with number a_i. We could go from A to B using applications of TOP-IN as follows:

```
A-TO-B$(A, B)$
    for $i = 1$ to $k$
        Take cards off the top and put them at the $i^{th}$ to last position in the deck...
        until we uncover the card with number $b_{k-i+1}$.
        Put card $b_{k-i+1}$ in its proper position.
```

We are choosing where to insert the top card because TOP-IN could insert the card anywhere in the deck.

Now we show aperiodicity. Because for any $u \in V$, TOP-IN(u) could return the same deck u if the top card is re-inserted on the top of the deck, and so this is a self-loop at u by definition, making G aperiodic.

Thus G has a stationary distribution. Now all that remains to be shown is that this distribution is uniform. Because theorem 1 tells us that the stationary distribution is unique, it will suffice to show that the uniform distribution is stationary.

Suppose our probability distribution over the nodes of G at time t is the uniform distribution

$$x_t(u) = \Pr[\text{we are at vertex } u \text{ at time } t] = \frac{1}{k!} \quad \text{for all } u \in V.$$

Now consider any arbitrary node $u \in V$. Let S be the set of vertices v for which there is an edge in E from v to u. By definition of the random walk the probability of being on node u at time $t+1$ is

$$\sum_{v \in S} x_t(v) p(v, u).$$

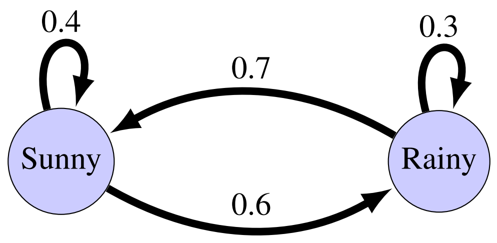
We can consider the operation inverse to TOP-IN as taking a card from anywhere in the deck and placing it on top. Evidently there are k possible outcomes. So $|S| = k$. Furthermore, for all $v \in S$, $p(v, u) = \frac{1}{k}$ because TOP-IN chooses a location uniformly at random. Thus

$$\begin{aligned} x_{t+1}(u) &= \sum_{v \in S} x_t(v) p(v, u) \\ &= \sum_{v \in S} \frac{1}{k!} \frac{1}{k} \\ &= \frac{1}{k!} \end{aligned}$$

and we conclude that the uniform distribution is the unique stationary distribution of G.

3 Markov Chain Monte Carlo Methods

A **Markov Chain** is a process for which the future state of the system depends (probabilistically) on the current state of the system without any dependence on past states.



For example, the diagram above represents a Markov chain depicting transition probabilities of the weather given two states: sunny and rainy. If the weather is sunny today, we think with 60% chance it will rain tomorrow, but whether it rains or not tomorrow isn't affected by the weather from yesterday and before. The Markov chain here is a weighted, directed graph.

The random walks in graphs that we have been discussing are Markov Chains with the vertices representing states and the edges (weights in a weighted graph) representing their relative transition probabilities. Future states of the system can be obtained by operating on the current state (which may be represented as a probability distribution) with the walk matrix W.

Recall that Monte Carlo algorithms run fast and will output a probably correct result. A **Markov Chain Monte Carlo** method allows us to use Markov Chains to *sample from a probability distribution* (in the case when directly sampling from such probability distribution is infeasible or computationally expensive). Because we are sampling a probability distribution, we assign each state i a probability, P_i.

3.1 Metropolis-Hastings Algorithm

To study certain problems, we know the states (vertices) and we know something about the stationary distribution (we know the relative probability of any two states, for instance), but calculating the transition probabilities directly is problematic.

The **Metropolis-Hastings Algorithm** provides a way to construct the transition probabilities "on the fly" so that the random walks on the graph produce the proper stationary distribution. It allows us to obtain a sequence of random samples from a probability distribution, when sampling directly from the probability distribution is hard. The sequence of random samples subsequently can be used to produce the proper stationary distribution.

Note that every random walk on a graph corresponds to a Markov chain, and every Markov chain can be viewed as a random walk in some graph with weights corresponding to transition probabilities.

We define the Metropolis-Hastings Algorithm as follows in algorithm 1.

Algorithm 1 METROPOLIS-HASTINGS

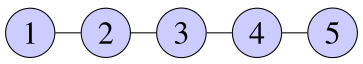
```
1: Start at arbitrary vertex $x_0$
2: for $t = 1, 2 \dots n$ do
3:     Pick neighbor of $x_t$ randomly, define this vertex as $x_{\text{trial}}$
4:     Evaluate $f_{trial} = \frac{P(x_{trial})}{P(x_t)}$
5:     if $f_{trial} \geq 1$ then
6:         “Accept the trial”: Set $x_{t+1} = x_{trial}$
7:     else
8:         choose random number $r_t$ uniformly from $[0, 1]$
9:         if $f_{trial} > r_t$ then
10:            “Accept the trial”: Set $x_{t+1} = x_{trial}$
11:        else
12:            “Reject the trial”: Set $x_{t+1} = x_t$
13:        end if
14:    end if
15: end for
16: return $x_1, x_2, \dots x_n$
```

Intuitively, for every iteration of the algorithm, we first attempt to walk randomly (step 3) to a new vertex x_{trial}. If this new vertex has a higher probability than x_t, we let x_{trial} be a new sample in the sequence. If this new vertex has a lower probability than x_t, then we add the new sample with probability f_{trial}. As we wish to sample from the distribution in regions of higher density, by evaluating f_{trial}, we can attempt to move closer to such region.

Side note: While consecutive samples x_{t+1}, x_t are correlated, the overall distribution of all samples will converge to the stationary distribution. As our graph is strongly connected, while selecting an arbitrary x_0 won't affect the stationary distribution of the overall sequence of samples $x_1, x_2, \dots, x_n$, we may wish to discard the first k samples as they will be closer to x_0. This is known as discarding the “burn-in.”

Example: Diffusion under a potential energy function

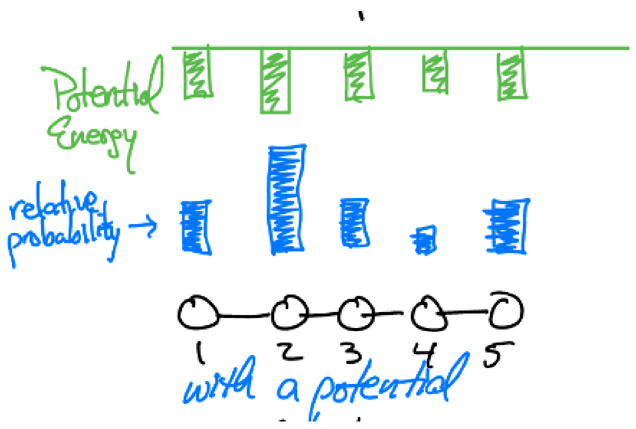
Consider a system of 5 points in the following graph:



For this example, we define the probability $p_i \propto e^{-\beta U_i}$, where $\beta = \frac{1}{k_B T}$ and U_i is the potential energy at state i.

If the potential energy at all 5 points is equal, then the stationary distribution is uniform (which our Metropolis-Hastings algorithm will attain). If the potential energy of the points differ (see figure 1), then the particles (which are our samples) will cluster at the points of lower potential energy.

Figure 1: Potential energy and relative probability



This gives a simple one-dimensional example of the MH algorithm. The true power of the Metropolis-Hastings method is in high dimensions (maybe 100,000-D) where it is not possible to visualize and traverse the graph. Sampling in high dimensions is especially useful in applications such as machine learning, where the number of features in a dataset is of very high dimension. MCMC methods have many applications in Bayesian statistics, computational physics, and computational biology.

4 Streaming Algorithms

In this recitation, we will consider streaming algorithms which operate on a graph $G = (V, E)$. In general, we assume that we know that the set of nodes V ahead of time and that the total amount of memory available is $\Omega(|V|)$ and $o(|V|^2)$: thus, we can store the set of nodes, but we cannot store all the edges of a dense graph. Instead, we process the list of edges as a stream.

When working with large and dense graphs, it would be very convenient if we could somehow reduce the size of the set of edges while preserving certain properties of the graph. We will now examine two streaming algorithms that can find such smaller graphs for two different properties.

Graph Spanners

Our first goal will be to reduce the size of a graph while approximately preserving the distances between every pair of nodes. This produces a *spanner* of the graph.

Definition 3 *Given $G = (V, E)$, a subgraph $H = (V, E_H)$ is a α-spanner of G if $E_H \subseteq E$, and*

$$d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d_G(u, v) \qquad \forall u, v$$

where $d_G(u, v)$ is the length of the shortest path between u and v in G, and $d_H(u, v)$ is the same measure in H.

Algorithm 2 SPANNER

```
1: $E_H = \emptyset$
2: for $(u, v) \in E$ do
3:     if $d_H(u, v) > \alpha$ then
4:         $E_H = E_H \cup \{(u, v)\}$
5:     end if
6: end for
7: return $E_H$
```

Algorithm 2 is constructs a α-spanner for an unweighted graph.

We can show that this procedure yields an α-spanner of G. Consider any pair of vertices u, v. Clearly $d_G(u, v) \leq d_H(u, v)$ since $E_H \subseteq E$. Now, assume the shortest path $P_{u,v}$ from u to v in G is of length k, which we decompose as $P_{u,v} = \{(u, p_1), (p_1, p_2), \ldots, (p_k, v)\}$, where the p_i are intermediate vertices on the path. We can bound $d_H(u, v) \leq \sum_{(l,h)\in P_{u,v}} d_H(l, h)$.

Now, for each edge $(l, h) \in P$, either this edge is in E_H, in which case $d_H(l, h) = 1$, or $d_H(l, h) \leq \alpha$ because otherwise we would have kept the edge (l, h) in E_H. Therefore, $d_H(u, v) \leq \sum_{(l,h)\in Pu,v} \alpha = k\alpha \leq \alpha d_G(u, v)$, and H is an α-spanner of G.

To bound the number of edges in H, we observe that H has no cycles of length less than $\alpha + 2$. It turns out by a pretty complex argument that such a graph has at most $n^{1+\frac{2}{\alpha+1}}$ edges. So, if our original graph was dense and we set $\alpha > 1$, we obtain an asymptotically smaller spanner.

SPANNER makes one pass over the stream of input edges. The total amount of memory used is equal to the memory for the nodes plus the memory for the subset of edges in E_H, which is $O(n^{1+\frac{2}{\alpha+1}})$ total. To process each edge, we have to recalculate the distance between the endpoints of that edge in H, which takes $O(|V| + |E_H|) = O(n^{1+\frac{2}{\alpha+1}})$ time.

Graph Sparsification

Note: This section is a little complicated. Don't get hung up on the definitions or lemmas related to sparsification. The main takeaway from this algorithm is the approach used to divide the input stream into sections and obtain an answer by combining the sections without concurrently keeping most of the sections in memory. If you understand what is happening in Figure 2, you already have the important points.

Now we will see how to decrease the size of our graph when we want to preserve capacity of every cut of a graph. For an undirected weighted graph, the capacity is the sum of the weights of all the edges that cross the cut. A valid sparsifer of a graph contains a (possibly reweighted) subset of the edges of the original graph which approximately preserves the capacity of all cuts.

Definition 4 *For a cut $(A, V \setminus A)$, the capacity* $\lambda_A(G) = \sum_{\{(u,v)\in E|u\in A,v\in V\setminus A\}} w_{(u,v)}$

Definition 5 *A weighted subgraph graph H is a$(1+\epsilon)$-sparsifier (for some $\epsilon < 1$) of G if*

$$(1-\epsilon)\lambda_A(G) \le \lambda_A(H) \le (1+\epsilon)\lambda_A(G), \qquad \forall A \subset V$$

There are nonstreaming algorithms for constructing $(1+\epsilon)$-sparsifiers with at most $\epsilon^{-2}n$ edges that take $O(|E|)$ memory. Unfortunately, if G is large enough that we want to find a sparsifier, $O(|E|)$ memory may be enormous! Instead, we shall use this nonstreaming algorithm as a subroutine to construct a streaming algorithm which never has to consider the entire edge set at once. Let A be any such nonstreaming algorithm which constructs a $(1+\gamma)$-sparsifier with at $O(\gamma^{-2}n)$ edges.

To construct our algorithm, we need two key properties of graph sparsifiers.

Lemma 2 *If G_1 and G_2 are two edge-disjoint graphs over a set of vertices V, and H_1 and H_2 are $(1+\epsilon)$-sparsifiers of G_1 and G_2 respectively, then $H_1 \cup H_2$ is a $(1+\epsilon)$-sparsifier of $G_1 \cup G_2$.*

Intuitively, if we independently find sparsifiers for two parts of a graph, we can put them together to obtain a sparsifier for the entire graph.

Lemma 3 *If G_1 is an α-sparsifier of G and G_2 is an β-sparsifier of G_1, then G_2 is an $\alpha\beta$ sparsifier of G.*

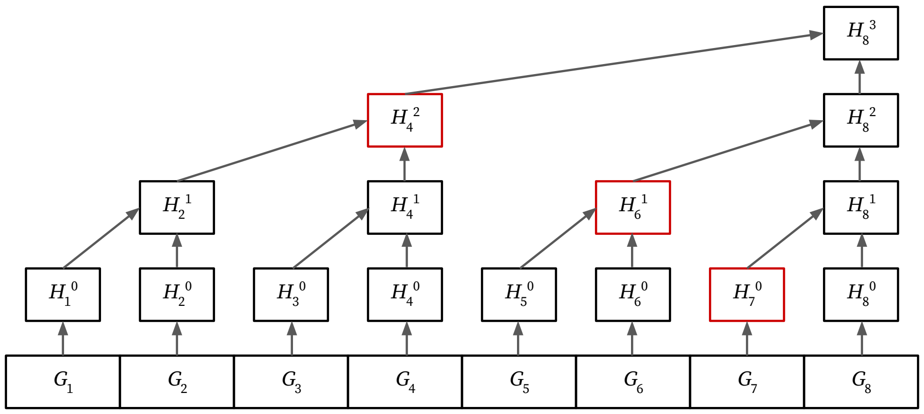
Intuitively, a sparsifier of a sparsifier of a graph is a (slightly less accurate) sparsifier for the original graph.

This algorithm is conceptually very similar to a divide and conquer approach. We will read an input stream of the edges in G in chunks of size $\Theta(n)$. Because $|E| = O(n^2)$, there are at most $O(n)$ such chunks. Let the subgraph defined by the ith chunk of edges be designated G_i. Finding $A(G_i)$ with some fixed γ parameter on each subgraph yields H_i^0, a γ-sparsifier for the given subgraph.

What if we just took the union of all these sparsifiers $\bigcup_i H_i^0$ as our final H? By lemma 2, we know that H is a γ-sparsifier of G, but the number of edges in H might be $\gamma^{-2}n \cdot n$, which is not any better than the original graph! Can we reduce the number of edges in H? By lemma 3, we could calculate $A(H)$ to find a smaller and slightly less accurate sparsifier, but we can't even construct H to pass into A because it would take $O(n^2)$ memory.

Instead, we're going to build up H iteratively while keeping its size manageable. For convenience, we'll assume that the number of chunks t of the edge list is a power of 2. We will have $\log t$ layers of sparsifiers H_i^j, which we will recursively build by sparsifying the union of two H_i^{j-1} from the layer underneath. The computation graph is visualized for $t = 8$ in Figure 2.

Figure 2: Computation graph for $t = 8$



Formally, we define

$$\begin{aligned} H_i^0 &= A(G_i) & i &\in \{1, \ldots, t\} \\ H_i^j &= A(H_{i-2^{j-1}}^{j-1} \cup H_i^{j-1}) & i &\in \{1 \cdot 2^j, 2 \cdot 2^j, 3 \cdot 2^j, \ldots, t\}, j \in \{1, \ldots, \log t\} \end{aligned}$$

By lemmas 2 and 3, H_i^j is a γ^j-sparsifier of the subgraph defined by

$$\bigcup_{k=(i-1)2^j+1}^{i2^j} G_i$$

Thus, $H_1^{\log t}$ is a $\gamma^{\log t}$-sparsifier of $\bigcup_{k=1}^{t} G_k = G$. If we set $\gamma = \epsilon/(2 \log t)$ we obtain a $(1 + \epsilon)$ sparsifier of G with $O(\epsilon^{-2} n \log^2 n)$ edges.

Does this algorithm really take $o(n^2)$ space? It certainly looks like we have to store a lot of intermediate H_i^j. But because of the order in which we merge the sparsifiers, we can actually discard $H_{i-2^{j-1}}^{j-1}$ and H_i^{j-1} after calculating H_i^j. For instance, after we process G_1 we calculate and store H_1^1. After we process G_2 we calculate H_2^1, and then calculate and store H_1^2 and discard H_1^1 and H_2^1. Thus, the maximum number of H_i^j we ever store is exactly $\log t$ right before processing the last chunk of the input stream (the nodes highlighted in red in figure 2), and the total memory usage is $O(\gamma^{-2} n \log t) = O(\epsilon^{-2} n \log^3 n)$.