Syllabus for "A Random Walk Through North America" Summer 2022

Clark Alexander
email: the author

June 13, 2022

1 Main Objectives

We wish to "solve" the traveling salesman problem for visiting all the states and provinces in the United States, Canada, and Mexico which are in continental North America. We will also talk about random walks on graphs and how to implement them in multiple computing langauges.

2 Daily Goals

2.1 Before we Meet

Get Setup with Julia. Best method is to download Julia 1.7 from the website and we'll be running in either JuPyter notebooks with iJulia, and for the more sophisticated coder, VSCode. Recently Atom has foregone the development of Juno REPL and VSCode has taken it over.

2.2 Monday

Morning

- Basics of Monte Carlo Methods
- How to draw a sample from a distribution
- Calculating π
- Markov Chains
- Predicting weather for the rest of the week

Afternoon

- Random Walks on Graphs
- How to perform a random walk on a graph

2.3 Tuesday

Morning

- The Simulated Annealing Algorithm
- Solving a random TSP

Afternoon

- How to Set up the North America TSP
- Taking our first shot at solving the big one

2.4 Wednesday

Morning

- How to get a better solution
- solving smaller problems
- Competing Models
- Coupled Models (Instructor's Own algorithm)

Afternoon

- Running our solvers
- Intro to Quantum Walks on graphs
- Starting our presentation for Coffee talks

2.5 Thursday

Morning

- Computing properties of the North America Graph Walk via Monte Carlo Methods
- Quantum Walking Through North America

Afternoon

- Quantum Walking
- Preparing our presentation Saturday

2.6 Friday

- Random Topics
- Working on Presentation

3 Monday

3.1 Monte Carlo Methods

A little history to begin. Monaco is known for a few things. It's one of the world's tiniest countries, the insanely lavish races culminating in the Formula One Grand Prix, and gambling. As much as I'd like to spend the rest of the week talking about the Grimaldi Family and their breakaway from the Republic of Genoa in the 12^{th} century, and even more I'd love to talk about the history of F1 in Monaco, but this is math and computer science course and so we're going to start with the gambling.

Monte Carlo is more-or-less a neighborhood in Monaco, but it's really just the casino district. The world's high rollers and those who live extravagant lifestyles lose obscene immoral amounts of money there. But the question we should be asking ourselves is why the "house always wins?." If you ever go to a casino in the United States (and most places in Europe) there will be a sign posted about the house's expected payout. Here in Chicago it's something like 91%. That is, for every $100 wagered the house collects about $9. Now there are two questions.

1. Can anyone actually win anything?
2. How do they know the payout so precisely?

The first answer is yes. And no. If one is willing to take a few turns at the roulette wheel and one gets a good spin, then one can walk away a happy person. I once witnessed this with my own mother. She decided it would be fun to wager $20 on the roulette wheel. She bet on a few random numbers she likes. One of them hit, she collected her winnings, and we went for a nice dinner. (Of course the house got the money back in an overpriced meal plus tips and wages, but that's another story). So yes, if one is lucky and extremely cautious simultaneously, one can win. However, the house doesn't care about random mothers and grandmothers winning a single turn of the roulette wheel. In fact, they quite prefer it. Why is that?

Our second question can best be answered by a set of techniques we call Monte Carlo methods based on techniques the famous casinos used to make sure the house always wins.

Monte Carlo methods in a nutshell are as simple as can be. Generate a bunch of random guesses, compute the average. The more guesses you generate, the closer your average is to the true average. In the case of a casino, roulette and craps have set probabilities. So we could generate millions of scenarios randomly and calculate what we would pay out and what the house would keep. In modern times this is feasible on a laptop as we shall soon see.

3.2 How to Draw a Sample from a Distribution

For this course we'll mostly be using Julia 1.6+ as our computing language. So we'll talk about drawing from distributions in this way, but we will also give

some tips for python and Octave/MatLab.

Let's start with what a probability distribution is. In order to do this we need to step just a couple paces back. We need to know what a "random variable" is. To make things as simple as possible without the highly technical details of σ-algebras and Borel spaces, etc, We define a random variable as a function, but the outcome is not deterministic, it's "random." A probability distribution is another function that determines the structure of the randomness of the outcome. A couple of well-known example are coin flipping and rolling dice. If we roll a single "fair" die we have the outcomes $\{1, 2, 3, 4, 5, 6\}$ each with probability 1/6. So the distribution is called a uniform distribution since each outcome is equally likely. With two dice we have the outcomes $\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ but the probabilities are not uniform. They are $\{1/18, 1/12, 1/9, 5/36, 1/6, 5/36, 1/9, 1/12, 1/18\}$

Some more common distributions we use and will use for this set of notes are the normal distribution and the uniform distribution on and interval. We will also need to draw samples from unnamed distributions, but those arising from experiments which we'll perform.

In Julia we use the packages Distributions and Random. using Random, Distributions

To draw a random normal variable with mean 0, deviation 1. we type a = randn()

for an entire vector of 100 random normal variables vec = randn(100)

3.2.1 Julia

3.2.2 Python

3.2.3 Octave

3.3 Example: Calculating π

Now let's go through a fun example. Let's calculate the value of π using a Monte Carlo simulation. Our approach will be to simply pick random points in the unit square and see how many have length less than one. Put simply, we'll count how many land inside the quarter circle versus how many random points we've chosen.

The code is easy 1

Now we compute count/numTrials which is approximately $\pi/4$ so we have

$$\pi \approx 4 * \text{count/numTrials}$$

The more samples we take, the closer we are. What happens when we combine the whole class's answers collectively?

3.4 Markov Chains

A Markov chain is a sequence of events where each subsequent event depends on the previous one. A little more precisely when we have a sequence $\{x_n\}_{n=0}^{N}$ the

Algorithm 1 Monte Carlo π calculation

```
1: procedure CALCULATE π(numberTrials)
2:     points = rand(numberTrials,2)
3:     count = 0
4:     for p = 1 :numberTrials do
5:         if norm(points[p,:]) < 1 then
6:             count + = 1
7:         end if
8:     end for
9: end procedure
```

probability that x_{N+1} will produce a certain value is conditioned on the value of X_N

$$Pr(x_{N+1} = \alpha | x_N = \beta)$$

If we follow this inductively, this gives us a chain of events.

$$Pr(x_{N+1} = \alpha | x_N = \beta, x_{N-1} = \gamma, ...x_0 = \zeta)$$

The secret is that the events of two previous points in the sequence are "baked in."

There are two relatively distinct types of Markov Chains, discrete and continuous. For the moment we'll focus on the discrete Markov chain. Let's take an example. Let's say we've been observing the weather daily for the last year and we wrote down only the whether or not we had precipitation. At the end of year we have 365 observations 75 of which had precipitation; which gives us 290 days without. Naively we say the probability of rain is 70/365 for the next day. However, if tomorrow it rains the next day's probability will be 71/366 and if not 70/366. However, weather is a little more complicated than counting raw numbers. We recognize, from having lived, that two days of sun in a row are more likely than two days of rain. But a day of rain is likely to follow another rainy day.

So for our 365 days we have 364 "next day" items to consider. Now look at this matrix

today \ tomorrow	precipitation	none
precipitation	35	40
none	40	250

This gives us a better picture of what to expect tomorrow. That is to say, if today is rainy, we have a likelihood 35/75 to see more precipitation tomorrow. If, on the other hand, it's nice and sunny today, our chances of seeing rain have been substantially reduced. This gives us a prime example of a discrete Markov chain.

More formally, if we have a probability start (a vector which sums to 1) the next state is determined by a transition matrix which is what we have constructed above.

$$p_{n+1} = Mp_n$$

Our example is starting in the state $p_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ (no precipitation)

$$p_1 = \begin{bmatrix} 0.47 & 0.13 \\ 0.53 & 0.87 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.13 \\ 0.87 \end{bmatrix}$$

However two days from now we have

$$p_2 = M^2 p_0 = \begin{bmatrix} 0.175 \\ 0.825 \end{bmatrix}$$

This tells us, it's slightly more likely to rain in two days than it is tomorrow. In weather forecasting we will collect many, many, many more weather types, in addition to precipitation and not, we collect temperatures, barometric pressure, inches of precipitation, etc. So instead of a 2×2 matrix, we'll have a 100000×100000 matrix. In order to predict the rest of the week's weather we consider the transition matrix to the 5^{th} power.

Remark. The continuous Markov chains follow exactly the form you'd expect with the difference replaced by a derivative

$$\frac{dp}{dt} = Mp(t)$$

We'll see this show up in a big way later in our course.

3.5 Predicting this Week's Weather

One this that is important in weather prediction is that we're beginning with frequentist statistics and from them we make Bayesian predictions. The transition matrices are based purely on observation.

So let's give this a try. Consider the following matrix:

$$M = something$$

Let's get a quick prediction for tomorrow, Wednesday, Thursday, Friday, and Saturday.

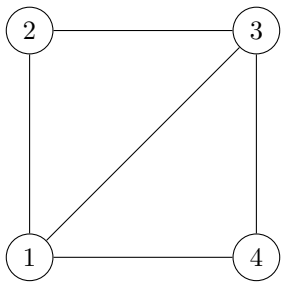
We will set today's vector appropriately and call it p_0. Notice this should be a measured data point. So we again use the frequentist statistics as a starting point. Let's run a million simulations (as a class) and see what we come up with.

3.6 Random Walks on Graphs

Before we go to a general graph let's start with an easy random walk. Let's imagine we're on a side walk with roughly square sections. Instead of moving along linearly, let's walk along randomly according to a coin flip. Let's flip a coin and if it's heads we step forward, tails we step back. My question to you is, after 100 flips where are we? The answer is, we don't know, but we're roughly close to where we started. In fact with high likelihood we're within 10 $\sqrt{100}$

steps of where we started. On a graph, we may have a slightly more complicated relationship.

Consider the following graph



If we start at vertex 1, we have 3 possibilities. So with probability 1/3 we move to each of the vertices 2,3,4. If we move to vertex 2, then we only have two possibilities.

For historical reasons, we tend to think of the random walk as a row vector as opposed to the more common (in textbooks) column vector. So our equation becomes

$$p_{n+1} = p_n W$$

In this case W is the matrix which encodes both the graph structure and the probabilities.

$$W = D^{-1}A = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 1/2 & 0 \\ 1/3 & 1/3 & 0 & 1/3 \\ 1/2 & 0 & 1/2 & 0 \end{bmatrix}$$

The matrix D is the *degree* matrix which encodes the degree of each vertex. In this way we can encode the probabilities of moving to another vertex. The matrix A is the *adjacency* matrix which encodes which vertices are connected via a single edge.

3.7 How to Perform a Random Walk on a Graph

Now we want to perform a random walk on an actual graph. Let's keep with our 4 vertex graph from above. In order to perform a random walk, we need a starting point. Let's start at vertex 1.

$$p_0 = [1, 0, 0, 0]$$

Now we compute p_1 and draw the next vertex from the distribution $[0, 1/3, 1/3, 1/3]$. This gives us vertex two. So our walk is $v1, v2$ now we choose from 1 and 3 with probability 1/2. We continue this as long as we want.

I tried this for 5 steps and got $v_1, v_2, v_1, v_4, v_3, v_1$.

3.7.1 Some Interesting Statistics about Random Walks on Graphs

Here we present some definitions, and we'll leave the harder exercises for your later classes.

Definition 1. The *hitting time* $H(u, v)$ for a random walk from vertex u to vertex v is the minimum expected time t so that the value of the walk at time t is v where the value at time 0 is u. Formally

$$H(u, v) = \mathbb{E}\left[\{t \in \mathbb{N} | X_t = v\} | X_0 = u\right]$$

Definition 2. The *cover time* of a graph from a vertex u is the minimum amount of expected time in which a random walk visits every vertex in a (connected) graph. The cover time of a graph is the maximum of all the cover times of individual vertices.

We'll try to get a hold on these via Monte Carlo methods for a big graph later on. Probably Thursday.

The main thing I wanted to talk about in this particular sub section is the stationary vector and how we get there. For historical reasons we call this vector π. It's defined by

$$\pi = \pi W$$

where W is the walking matrix we've defined above. You'll notice from the definition that this is the eigenvector with eigenvalue 1 for the matrix W. We know that such a vector exists since we have defined our matrix W the have every row sum to 1. In fact, we know this is the largest eigenvalue because we've normalized every row, so that the resulting vectors are probability distributions.

If you've every worked with an algorithm computing the largest eigenvalue you'll recognize:

$$\pi = \lim_{n \to \infty} p_n = \lim_{n \to \infty} p_0 W^n$$

The interesting thing (to me) is that we don't have to go to ∞.

Theorem 3. *Given a connected graph G and associated walking matrix W and an initial random walk vector p_0*

$$\|\pi - p_0 W^t\| \in O(\lambda_2^t)$$

Where λ_2 is the second largest (absolute) eigenvalue of W.

The difference between $\lambda_1 = 1$ and λ_2 is called the *spectral gap*. The larger the spectral gap, the fewer steps to get to the stationary vector. For graphs in which the spectral gap is particularly large, these graphs are called expander graphs and in extreme cases are called *Ramanujan* graphs. Generally, we expect Ramanujan graphs to be regular graphs, but we can still calculate the spectral gap on a normal graph, especially for the purposes of finding how long it will take to reach the vector π

4 Tuesday

Today we're going to turn our knowledge gained yesterday into an ability to optimize difficult problems. The goal, of course, is to allow randomness to do a lot of the hard work for us and not go through the brute force calculations. In particular when we have a potential solution space which grows exponentially or faster, we cannot check all the possibilities. We'll run out of time. Just some quick "back-of-the-envelope" calculations show us how long it would take to achieve the exact solution. Let's start with two quick examples.

4.0.1 Choosing a Stock Portfolio

A bunch of empirical studies (hidden Monte Carlo methods in action) show us that a good amount of stocks to invest in at a time is 20. On the Bolsa de Valores there are 144 stocks listed. How many portfolios of 20 stocks are there?

$$\binom{144}{20} \approx 1.51 \times 10^{24}$$

That's a lot! Consider a modern laptop can operate at roughly 3 GHz (3×10^9) this means it would take us 4×10^{14} seconds to calculate each portfolio if the individual calculations only cost us one flop. That's about 16 million years. I guarantee your laptop won't last that long. So this is obviously not feasible.

If you go to the Nasdaq where there are more than 3000 stocks available, well, that becomes infinite (to first approximation). There must be a better way.

4.0.2 The Traveling Salesman Prblem

This is one of the classics in computing theory. We want to start our journey at "home" and visit some fixed number of sites (usually cities or businesses) and return home having traveled the shortest distance. This grows as $n!$ where n is the number of places we have to visit (including home). For 19 additional cities this is already 2.4×10^{18} possibilities. We're going to try this week to "solve" a problem with 95 cities which lands us on

$$95! \approx 1.03 \times 10^{148}$$

No supercomputer or cluster of supercomputers nor quantum computer no nothing, can handle that size problem.

4.1 The Simulated Annealing Algorithm

Let's talk about annealing first. Yes, metallurgy. What is annealing in the metallurgical sense? The idea is that a raw piece of metal which we get by mining or sifting through dirt or picking out of a riverbed has a lot of impurities.

To really make a good point about the impurities of metals consider this fun story.

It is said that the king of Siam (now Thailand) in the early 1800s was the richest man in the world. During a meeting with Napolean Bonaparte this wealth was on display. At a formal dinner, Napolean's men ate with silver utensils, while Napolean himself ate with gold utensils. The king of Siam, however, ate with aluminum utensils. Now you're thinking, what the ****!? Aluminum is super cheap! Not in the early 1800s it wasn't! Aluminum has so many impurities in it that finding a sample pure enough from which to make utensils made aluminum the most valuable metal in the world at that time.

Now let's come to the present. Aluminum is super cheap, and that's because it turns out, it's very easy to remove impurities by electrolysis. The older technique of annealing became prevalent in sword making. Annealing is the process of taking a metal with impurities and heating it up to some extreme temperature where all the impurities burn off and as it cools, one can bend and shape the new metal into a new shape which will be much stronger. This is what happens in the process of making iron into steel. In sword making, if one hammers out a piece of iron into a sword, then heats it up to extreme temperatures and quickly cools it (while emblazening the master maker's mark) we get extraordinary swords. One can also see this technique at work in high quality kitchen knoves these days.

Now, our goal is to turn this optimization process of metals into an optimization process for combinatorial problems. As in basic Monte Carlo methods, we let nature guide us, rather than trying to be too clever at the beginning. So simulated annealing involves a temperature parameter which we start "hot" and reduce to "freezing" at some rate. There are so many ways to do this, but the basic process is the following:

1. Guess a solution
2. Using the first solution guess a second solution, called a neighbor (This is a Markov Chain Monte Carlo step)
3. score both solutions
4. if the "new solution" is better than the "old solution" accept it immediately and make this the"old solution"
5. If the "new solution" is worse, we still might accept it, and that is based on how much worse it is, and the temperature.
6. Lower the temperature
7. Repeat until "freezing"

Now you're looking at this and you got stuck at step 5, and probably blurted out some expletive and said, this guy is totally mad, why!? NO! STOP! That's not how to find an optimal solution! But dear friends, this is a highly effective method and let me explain why.

If we only use neighbors to find better solutions, this amounts to a simple guess and check method, and is likely to get us stuck in a local minimum. Here's a good analogy. We want to find the highest peak in a mountain range, but we don't know where to start. So we climb the hill closest to our house. We never accept going down hill. Once we reach the top of our local hill, we look around and see any direction we step is downward. Thus we have reached the highest peak, right? Obviously no. We look to the next hill over and realize if we had started there, we would have landed at a higher elevation. So what went wrong? Our strategy was too rigid. Since our solution has so many possibilities, the likelihood that we started in the "right place" for a strict uphill climb to the top is very low. Thus we allow ourselves to jump around to neighoring hills once in a while to give ourselves a better shot at reaching the highest peak.

In simulated annealing, we generally look for the minimum, but this can easily be made into a maximization problem by looking for the lowest value of $-f$ and wiping away to negative sign at the end.

So here's our pseudo-code

Algorithm 2 Simulated Annealing

```
1: procedure MINIMIZE "SCORE"(x_0, T_0, T_f, trials_per_degree)
2:     oldSolution = x_0
3:     oldScore = score(x_0)
4:     bestolution = neighbor(x_0)
5:     bestScore = score(x_0)
6:     T = T_0
7:     while T > T_f do
8:         for k = 1:trials_per_degree do
9:             newSolution = neighbor(oldSolution)
10:            newScore = score(newSolution)
11:            if newScore < bestScore then
12:                bestScore = newScore
13:            end if
14:            if acceptanceProbability(oldScore, newScore, T) > U(0,1) then
15:                newSolution = oldSolution
16:                newScore = oldScore
17:            end if
18:        end for
19:        Lower T
20:    end while
21: end procedure
```

You can see here there are several auxiliary functions we need to define, and that's the beauty of simulated annealing. It can be tuned to any kind of problem. We need to define how to score a solution, how to pick a neightbor, how much tolerance we have for selecting a worse solution, and at what rate we wish to lower the temperature. These particulars must be tuned for each

problem, but luckily I have enough practice runs at this to help guide you in picking these functions.

4.2 Solving a Random Traveling Salesman Problem

Now let's take this toy out for a spin. We're going to minimize the route we need to take around the unit square and visit 50 different points along the way. First we'll generate our points randomly, and then I'll provide a set of pre-selected points. Let's see who can get a really small solution.

Some suggestions here. For the time being, distance will be the standard Euclidean distance in 2D.

$$d(x, y) = \|x - y\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

The score will be the sum of all distances traveled including returning home. So we'll have

$$\text{score} = \sum_{n=0}^{50} d(x_n, x_{n+1}) \text{with} x_0 = x_{50}$$

The acceptance probability is fairly common

$$\exp((oldScore - newScore)/T)$$

Notice two things here. If the new score is smaller, the probability is automatically greater than 1, and so will be immediately accepted. Second, as the temperature cools, we get more strict about how much worse the solution is before we accept it. For example if $T = 100$ and new score $= 2$ with old score $= 1$

$$\exp((1 - 2)/100)) \approx 0.99$$

So we're very likely to accept that. Later when our temperature gets down to say $T = 0.1$

$$\exp(1 - 2)/(0.1) \approx 1 \times 10^{-10} \approx 0$$

We're extremely unlikely to allow this to happen.

In the department of lowering temperature, there are two common ways: linear and geometric. I generally prefer the geometric

$$T* = \alpha$$

where $\alpha < 1$ but reasonably close. Something like 0.95. There are not set rules on this. The second common way is just decrementing by some fixed linear amount

$$T- = 0.01$$

However, there are lots of ways to do this, in fact, one can pick any montonically decreasing function, so that the temperature can slow down at points to allow more explanation and speed up when solutions get better. For example,

Algorithm 3 Custom Temperature Decrement for Simulated Annealing

1: **procedure** SLOW AT $T = \beta$(numTemperatures)
2: Section1 = $(\beta + 0.001) + (1 - 0.001 - \beta) * rand(n/3)$
3: Section2 = $(\beta - 0.001) + (0.002) * rand(n/3)$
4: Section3 = $(\beta - 0.001) * rand(n/3)$
5: Temperatures = sort([Section1;Section2;Section3], reverse = true)
6: **end procedure**

consider that we want to slow down our temperature around 0.4 We can pick a sequence of temperatures as follows

In our annealing algorithm we replace the while loop with a for loop T in Temperatures.

The final piece, and potentially the most crucial is selecting a "neighboring" solution from the current one. In the traveling salesman problem, if we pick completely randomly, and plot our path it looks like a big tangle. A "solved" problem has a few "obvious" characteristics; namely, no loops and no crossings. If you ask a random person to draw you a solution, they will use some sort of heuristics like that. So our goal as a machine teacher is to tell the computer how to "untangle" the mess. Again, there are multiple suggestions, but the simplest (for a small problem) is pick a section at random and reverse the order in which it is traversed. This is equivalent to telling the computer to try to untangle a section at a time. It also allows us to see why we might accept a slightly worse answer if there are a lot of tangles still remaining.

One can pick any number of variations on that theme. Unravel two separate sections at a time, switch two points only, move through a cycle of 3,4,5,6,7,etc points. Randomly exchange ten points, switch two adjacent points, etc.

4.3 How to Set up the North America TSP

For our main project this week we're going to thoroughly examine the graph of (connected) North America. That is, 13 Canadian Provinces, 49 states and Washington D.C. in America, and 31 states and the Federal District of Mexico which brings us to 95. Quick aside, while Alaska is not connected to the other 48 states, it is connected to Canada, so we're including it here as part of the connected component. Additionaly, Prince Edward Island and Nova Scotia in Canada are not connected by a physical land border, but they are connected by bridges, so we include them here as well, since they are lovely places, and we don't want to cheat and give ourselves any advantages. We really want to travel through North America.

We will have two sets of distances. The first is the flying distance between capital cities of these regions. That will be computed using Earth's great circle calculation (and taking into account that Earth is squished a little). I'll provide those calculations for you. That will be a 95×95 matrix. Additionally, we'll use driving miles which I'll also provide, but looked up through google maps.

Our task is to start in Guadalajara and traverse the other 94 cities and return here in as little distance as possible. This calculation in this case will be done by simple look-ups rather than calculating the Euclidean distance for each pair.

4.4 Taking Our First Shot at the Big One

Now that we have set up our distance calculations and our goal of minimizing said distance, let's take a shot at solving both of these problems. What I want is for people to split up into teams of 3 and try out different strategies for picking neighbors. Let's see who comes up with the shortest distance by day's end!

5 Wednesday

5.1 How to get a better Solution

We've now taken a shot at solving the North American Traveling Salesman problem with a variety of neighbor solutions. Hopefully we've also used different neighboring techniques. Let's see who has the best solution so far. There is no guarantee that this is the globally optimal solution, but it's possible. Remember, in order to verfiy this we'd need to check all 95! possible paths. However, let's again take our cue from the physical world and see what makes a great sword or a great knife. A great knife is made by having an already decent knife to begin with, heating it up extremely hot,no hotter than that... still hotter, hotter, there. Then "quenching" which means cooling it very quickly. Basically dousing it in a liquid. It's a funny fact, that the dousing liquid's composition can drastically change the final outcome too, but we aren't going to that level of detail in our simulated annealing just yet.

So here are some suggestions to get a better solution, Try doing a partial anneal, and leave an ok, but obviously wrong "solution" in play. This solution should still have some tangles and crossings etc, but mostly good. Now, slow down the annealing and play around with a different neighboring method.

You can break this up into several cooling phases and neighbor selections. Usually this will get you over a hump.

A second suggestion is to break up the solution into smaller, tractable problems. For example, 19 points is a fairly easy problem for a simulated annealing algorithm to "solve." So we may want to cluster our cities into groups of 19, solve the 5 individual groups and than add the connections between groups. This can be done in the most naive way of finding the two closest points in each cluster and adding the connections there.

5.2 Competing Algorithms

Hopefully we've tried a couple of these solution techniques and we've gotten slightly different plausible solutions. One of the several tried and true techniques I use in machine learning in general is to make algorithms compete. Where they

agree on the solution, that's what I use and take to investors and CEOs and other business types who don't really care about the machine learning, but are much more focused on the business strategy. So instead of stopping the anneal in the middle, and switching up the neighbors and temperature selections, why not run them all together in parallel? We can also run a full algorithm really fast and use the "solution" as a seed for the other. In essence, run two annealing algorithms with different parameter selections and then use their respective solutions as seeds for the other.

- Run Algorithm 1
- Run Algorithm 2
- Send Solution 1 to Algorithm 2
- Send Solution 2 to Algorithm 1

There are as many variations on this as you want. You can even run entirely different algorithms (not just annealing) and play this game. For example, algorithm 2 could be a genetic algorithm instead of an annealing algorithm. Or the second algorithm could be reverse annealing (something which I don't really like, based on empirical results.)

5.3 Coupled Models

In the statistical sciences Markov Chain Monte Carlo methods have shown great success in learning probability distributions and statistics of different studies. They are extremely useful for giving good election forecasts and settling high dimensional problems which are hard to visualize, but easy to compute. We've already talked about the weather, but this shows up in the social sciences and biological sciences a lot: Elections, polls, pharmaceutical studies, medical studies,chemical reactions, flight simulations, stock predictions, pricing of commodities, etc.

In 1991 Geyer found that as well as MCMC algorithms perform (in this particular case he used the Metropolis Hasting algorithm), they tend to take a very long time to converge when the distribution is multimodal. (give a picture here) Thus we proposed the idea of a Chained MCMC model, which for the sake of naming has become a Metropolis Chained Markov Chain Monte Carlo or MCMCMC.

Remark. Nicholas Metropolis was a Greek-American statistician and had easily the greatest name ever. If I had a name as cool as Metropolis I'd be inventing algorithms left and right just to get my name of them.

What we'vebeen focusing on this week is using MCMC algorithms in the form of simulated annealing to make them optimization algorithms. So with that in mind, last summer I wrote the first (to my knowledge) Metropolis Coupled Simulated Annealing algorithm (MCSA) where I run multiple chains of

simulated annealers for one temperature each with a different type of neighboring solution and then rank them by how well they've done. Then I change the temperature decrements and neighbor selections with a little randomness (an additional Monte Carlo step) and carry it out until one chain is frozen.

5.4 Intro to Quantum Walks on Graphs

Maybe you haven't studied quantum mechanics, and you don't it just yet, but a few basic may help. For the sake of simplicity we'll definitely stick to continuous quantum walks on graphs.

So here are some basics: The main object we're concerned with if a function called the *wave* function. It's generally denoted ψ or Ψ or $|\psi\rangle$ more on that funny notation later. This function is subject to a very famous equation called the *Schödinger* equation which you may recognize as

$$\frac{\partial \Psi}{\partial t} = \left(-i\hbar\nabla^2 + V(\vec{x}, t)\right)\Psi$$

We can reduce this in a lot of different ways, almost none of which I'll write for you here. The main message to take away is that this is simply a partial differential equation which at it's heart looks like

$$\frac{\partial \Psi}{\partial t} = U(\vec{x}, t, \Psi)$$

And a crucially important point is that U is a unitary operator. In the wide world of advanced mathematics, that has a very specific meaning, but in the world of linear algebra, specifically Matrix algebra a unitary matrix is one in which

$$U^\dagger = \bar{U}^T = U^{-1}$$

That is, the complex conjugate transpose is the inverse. If you've never seen something like this before, you're probably cursing me right now, and hopefully in Spanish. However, while finding matrices like this out in the wild isn't so easy, we have a really easy construction for them.

Theorem 4. *Let A be a square symmetric matrix, that is $A^T = A$ then for any $t \in \mathbb{R}$ the matrix*

$$e^{itA}$$

is unitary.

The proof is quite simple. since $A^T = A$. We get $\left(e^{tA}\right)^T = e^{tA}$ for free. And the complex conjugate is really easy.

$$e^{\bar{i}tA} = e^{-itA}$$

Finally for the inverse $e^{-itA}e^{itA} = I$. That's becauase A natturally comutes with itself and so we can (if we really like to punish ourselves) expand the power series and collect terms, we'll find, only the identity is leftover.

$$\left(e^{itA}\right)^{\dagger} = e^{-itA} = \left(e^{itA}\right)^{-1}$$

Well, it turns out we know plenty of such matrices which are symmatric: The adjacency matrices of undirected graphs. We have a small problem. The walking matrix from before $W = D^{-1}A$ is not symmetric unless the graph is regular. So we take care of this in a difference way. The matrix we actually consider is aclled the matrix *Laplacian* (yes, I know, a lot of old dead dudes with their names on stuff....)

$$L = D - A$$

Since the degrees are on the diagonal, the Laplacian is symmetric.

Now back to quantum mechanics. We had this funny other symbol $|\psi\rangle$. This is Dirac's (again another dead dude) notation for a vector. At the level of freshman and sophomore level physics this seems totally ridiculous, but trust me, this grew out of a much more organic process in need a notation that is highly versatile and really works. One of the very few rewritten Schödinger equations I'll actually show you is this.

$$\frac{\partial \psi}{\partial t} = -iL\psi$$

(Does this look familiar? Maybe a continuous Markov Chain!?!?!?)

Which we may have seen in basic differential equations as having the solution

$$\psi(t) = e^{-itL}\psi(0)$$

If we make our life really easy (notationally) we can just rewrite this whole thing in our final Schödinger rewrite.

$$|\psi(t)\rangle = U(t)|\psi(0)\rangle$$

ooh! Ah! wow!

When we're thinking about a graph, and a random walk on it, the vector $|\psi(t)\rangle$ represents our quantum walker at time t. Since this is quantum mechanics we use amplitudes instead of probabilities and that means our vector has length 1 instead of sum 1. Hence the importance of the unitary matrix which in reality means it doesn't change a vector's length. That's the whole point!

In the classical scenario, we measure our probability of being at a specific vertex by reading the corresponding item in the vector.

For example

$$v = 0.1, 0.2, 0.3, 0.4$$

Means we have a 0.1 probability of being at vertex 1, 0.2 probability of being at vertex 2, etc.

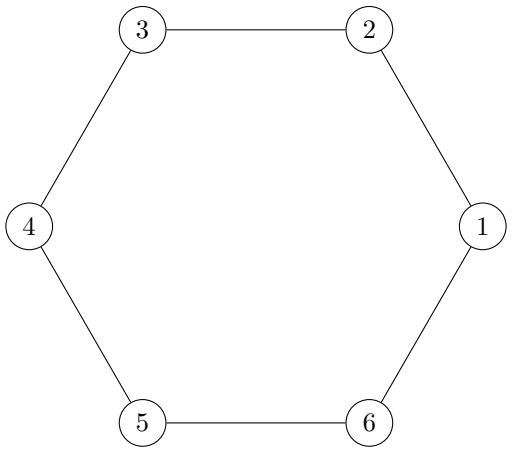
In the quantum world it's slightly more complicated, but not much. Recall that our vectors need to have unit legnth and not unit sum. So these are what

we call amplitudes. The probability of having our quantum walker at vertex n after time t is

$$Pr(|\psi(t)\rangle \text{ at } |n\rangle) = \langle n|U(t)|psi(0)\rangle = \langle n|\psi(t)\rangle$$

That is to say we take the inner product. In this case the backward looking thing $\langle\psi|$ is called a *bra* and the forward looking thing is called the *ket* together making "bracket" notation. The ket is a normal column vector like we've always used, but the bra is the conjugate transpose vector in keeping with complex valued norms.

Let's give an example before we go into any more gross details and justifications. Let's consider the graph C_6 which looks like a hexagon.



We have the associated matrices

1. $A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

2. $D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix} = 2I$ Because this is a regular graph.

3. $L = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ -1 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$

4. And for posteriety, to tie the week together if you want to work through this: $W = \begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 1/2 \\ 1/2 & 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}$

Now let's look at a classical walk. If we start at vertex 1 in one step we can get to vertex 2 or 6. After two steps we can get to 1(probability $\frac{1}{2}$),3 or 5 probability $\frac{1}{4}$ each) and after 3 steps we could finally get to vertex 4 (probability $\frac{1}{8}$)

So let's do something crazy... Let's quantum walk for time $0.1, 0.5, 0.9, 1.1, 1.5, 1.9, 2.1, 2.5, 2.9,$ and finally 3.0 We can really see the big difference in walking quantum mechanically versus classically. Then we can show some plots.

time	v_1	v_2	v_3	v_4	v_5	v_6
0.1	0.990025	0.0995009	0.0049792	0.000332501	0.0049792	0.0995009
0.5	0.765156	0.440299	0.112427	0.0391267	0.112427	0.440299
0.9	0.338673	0.585725	0.282937	0.197602	0.282937	0.585725
1.0	0.221486	0.583589	0.318816	0.257882	0.318816	0.583589
1.1	0.10623	0.566568	0.347366	0.324639	0.347366	0.566568
1.5	0.282839	0.379538	0.353577	0.617957	0.353577	0.379538
1.9	0.479182	0.111481	0.155893	0.834819	0.155893	0.111481
2.0	0.495312	0.0508316	0.0791656	0.858466	0.0791656	0.0508316
2.1	0.499984	0.0027888	0.00486176	0.865998	0.00486176	0.0027888
2.5	0.439542	0.120151	0.361602	0.718623	0.361602	0.120151
2.9	0.352132	0.0751176	0.618826	0.314367	0.618826	0.0751176
3.0	0.339938	0.0460985	0.650054	0.187219	0.650054	0.0460985

There are some really exciting results here. Most noatbly the probability of hitting vertex 4 after 1 unit of time, reaching a maximum (in our experiment at 2.1 units of time). I ran a Monte Carlo simulation of this (rather than computing the full derivative) and found the maximum value occurs at ≈ 2.0944 units of time.

6 Thursday

6.1 Properties of the North America Graph via Monte Carlo

In random walks on graphs there are some interesting properties we want to think about. For example back in §3.7.1 we talked about *hitting time, cover time* and there is another called *commute time* which is simply

$$\kappa(u, v) = H(u, v) + H(v, u)$$

which is interesting because this satisfies all the criteria to be a metric on a graph. But as simple as these definitions are, on graphs which are reasonably complex or bigger than a roughly trivial graph or a complete graph, actual calculating these things are quite difficult. So let's look at how a Monte Carlo Analysis might shed some light on this. Let's try this explicitly. On the North America graph, let's see if we can calculate the hitting time between Jalisco and Illinois (and also in the reverse direction). You will notice that while it is possible to get from one to the other in a minimum of 6 steps, that is extremely unlikely to happen. Here are a couple of paths

1. Jalisco, Zacatecas, Coahuila, Texas, Arkansas, Missouri, Illinois
2. Jalisco, Zacatecas, Coahuila, Texas, Oklahoma, Missouri, Illinois

However, these are extremely unlikely to happen in a random walk of 6 steps. For example, the first walk will happen with the following probabilities:

- Jalisco to Zacatecas 1/4
- Zacatecas to Coahuila 1/6
- Coahuila to Texas 1/5
- Texas to Arkansas 1/8
- Arkansas to Missouri 1/6
- Missouri to Illinois 1/8
- Total 1/46080 (← not good odds)

The other path has exactly the same odds, so for either of these to happen, we're looking at 1/23040 still not great odds. There are a lot of paths in which we can travel in 7 steps, but again, these are unlikely. But rather than actually counting all the possible paths let's simulate the walks and count how long to hit Illinois starting in Jalisco.

The algorithm will look something like this.

This procedure gets a single "minimal" hitting time. As we see in the definition, hitting time is the expectation (average) of minimal hitting times. So

Algorithm 4 Monte Carlo Hitting Time

```
1: procedure GET HITTING TIME(Start Vertex, End Vertex)
2:     v_0 ← start vertex
3:     set counter = 0
4:     while v_1 ≠ end vertex do
5:         counter += 1
6:         Perform one step in Random Walk v1 ← new vertex.
7:     end while
8: end procedure
```

we perform this algorithm many many times (A Monte Carlo simulation) and take the average. The more times we perform this simple procedure, the more likely we are to get a precise number.

If we have a good computer then we can consider computing the following sequence

$$p_n = v_0^t W^n v_f$$

That is, what is the probability of landing on our final vertex (v_f) in n steps after having begun at the initial vertex (v_0). This is roughly equivalent to computing the matrix W^n for some large n (meaning that we are within ε of the steady state vector.) This is something we get from the spectral gap. Then we consider

$$v_0^t W^n v_f \approx \pi^t v_f$$

Then expected number of steps is the reciprocal of this $1/\pi^t v_f$

6.2 A Quantum Walk Through North America