Monte Carlo Optimization

Two uses of computer-generated random variables to solve optimization problems.

The first use is to produce stochastic search techniques

To reach the maximum (or minimum) of a function

Avoid being trapped in local maxima (or minima)

Are sufficiently attracted by the global maximum (or minimum).

The second use of simulation is to approximate the function to be optimized.

Introduction

Optimization problems can mostly be seen as one of two kinds

Find the extrema of a function $h(\theta)$ over a domain Θ

Find the solution(s) to an implicit equation $g(\theta) = 0$ over a domain Θ.

The problems are exchangeable

The second one is a minimization problem for a function like $h(\theta) = g^2(\theta)$

while the first one is equivalent to solving $\partial h(\theta)/\partial\theta = 0$

We only focus on the maximization problem

Deterministic or Stochastic

Similar to integration, optimization can be deterministic or stochastic.

Deterministic: performance dependent on properties of the function

such as convexity, boundedness, and smoothness

Stochastic (simulation)

Properties of h play a lesser role in simulation-based approaches.

Therefore, if h is complex or Θ is irregular, chose the stochastic approach.

Numerical Optimization

R has several embedded functions to solve optimization problems

The simplest one is **`optimize`** (one dimensional)

We saw use of this already when finding M in importance sampling in relation to the ratio of the target function to the importance function

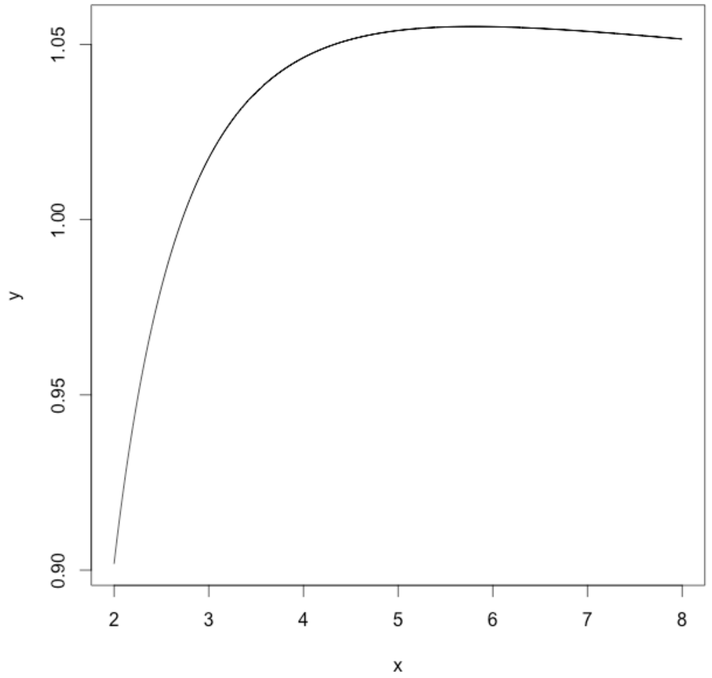
Numerical Optimization

Just as a refresher lets maximize the function

$$f(x) = \frac{log(1 + log(x))}{log(1 + x)}$$

with respect to x.

The graph of f(x) shows the maximum occurs



Numerical Optimization

R code that applies the optimize function on the interval (4, 8). The default is to minimize the function. The maximum is the x value where the maximum occurs while objective is the maximum of the function.

```
> x <- seq(2, 8, .001)
  >     y <- log(x + log(x))/(log(1+x))
  >     plot(x, y, type = "l")
  >
  >     f <- function(x)
  +         log(x + log(x))/log(1+x)
  >
  >     optimize(f, lower = 4, upper = 8, maximum = TRUE)
  $maximum
  [1] 5.792299


  $objective
  [1] 1.055122
```

Newton-Raphson

Similarly, **`nlm`** is a generic R function using the Newton–Raphson method

Based on the recurrence relation such as convexity, boundedness, and smoothness

$$\theta_{i+1} = \theta_i - \left[\frac{\partial^2 h}{\partial\theta\partial\theta^{\mathrm{T}}}(\theta_i)\right]^{-1} \frac{\partial h}{\partial\theta}(\theta_i)$$

Where the matrix of the second derivatives is called the Hessian

The vector of the first derivatives is the gradient

This method is perfect when h is quadratic

But may also deteriorate when h is highly nonlinear

It also obviously depends on the starting point θ_0 when h has several minima.

Stochastic - A Basic Solution

A natural, if rudimentary way of using simulation to find $\max_{\theta} h(\theta)$

Simulate points over Θ according to an arbitrary distribution f positive on Θ until a high value of $h(\theta)$ is observed

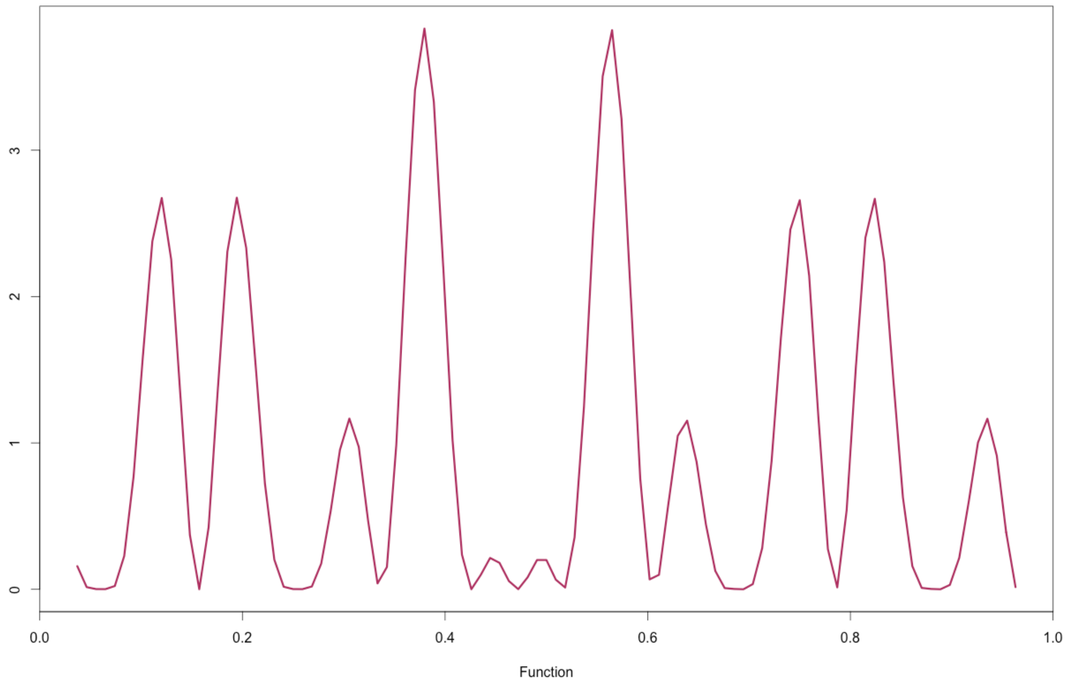
Solution may be inefficient if f is not chosen in connection with h

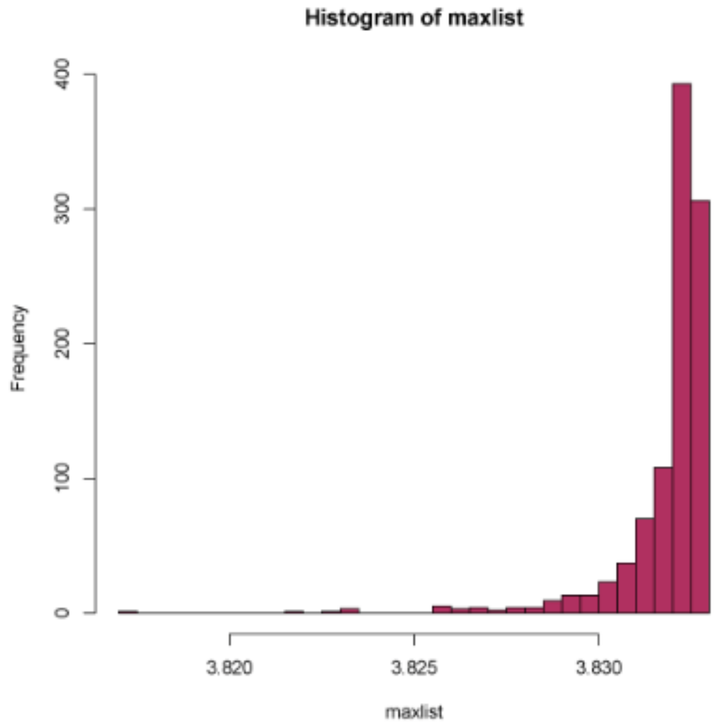
Given an infinite number of simulations and some regularity requirements on the problem (including compactness on the domain) it is bound to converge

For instance if Θ is bounded, simulate from a uniform distribution on Θ and use $\max(h(u_1) \ldots h(u_m))$ as the approximate solution

Stochastic - A Basic Solution

Lets use h(x) = [cos(50x) + sin(20x)]^2



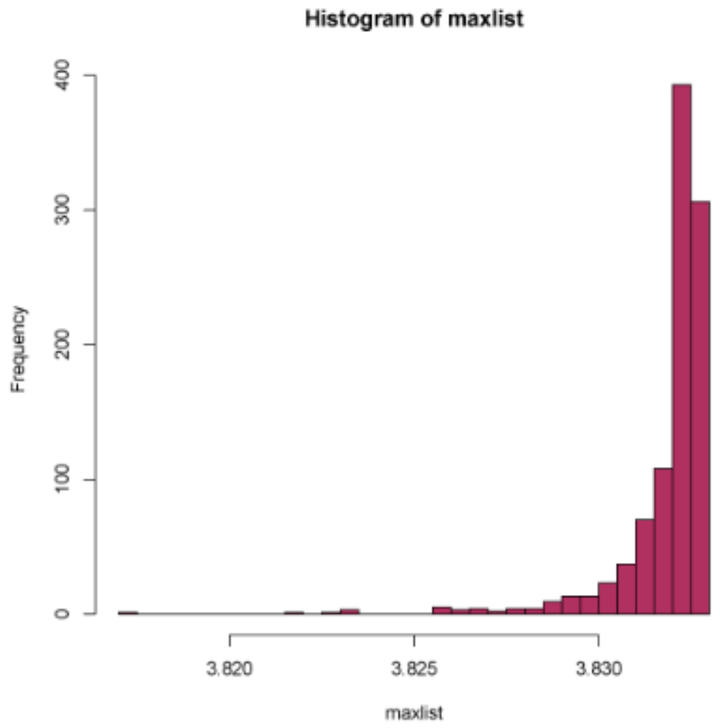


Stochastic - A Basic Solution

Recall h(x) = [cos(50x) + sin(20x)]^2

True Max = 3.8325

```
h=function(x){(cos(50*x)+sin(20*x))^2*(x>0)*(x<1)}

rangom=h(matrix(runif(10^6),ncol=10^3))

max(rangom)

[1] 3.832544
```



Stochastic Search

Obviously this is blind solution (since it doesn't take h into account) quickly gets impractical as the dimension or the complexity of the problem increases

It becomes more useful to design the simulation experiment in connection with h as well as with the domain

Intuitively, it makes sense to increase the probability in regions where h is large and to decrease the probability where it is small

Stochastic - Stochastic Gradient Methods

Generating direct simulations from the target can be difficult.

Different stochastic approach to maximization

Explore the surface in a local manner.

A Markov Chain

Can use $\theta_{j+1} = \theta_j + \epsilon_j$

The random component ϵ_j can be arbitrary

Can also use features of the function: Newton-Raphson Variation

Where $\nabla h(\theta_j)$ is the gradient

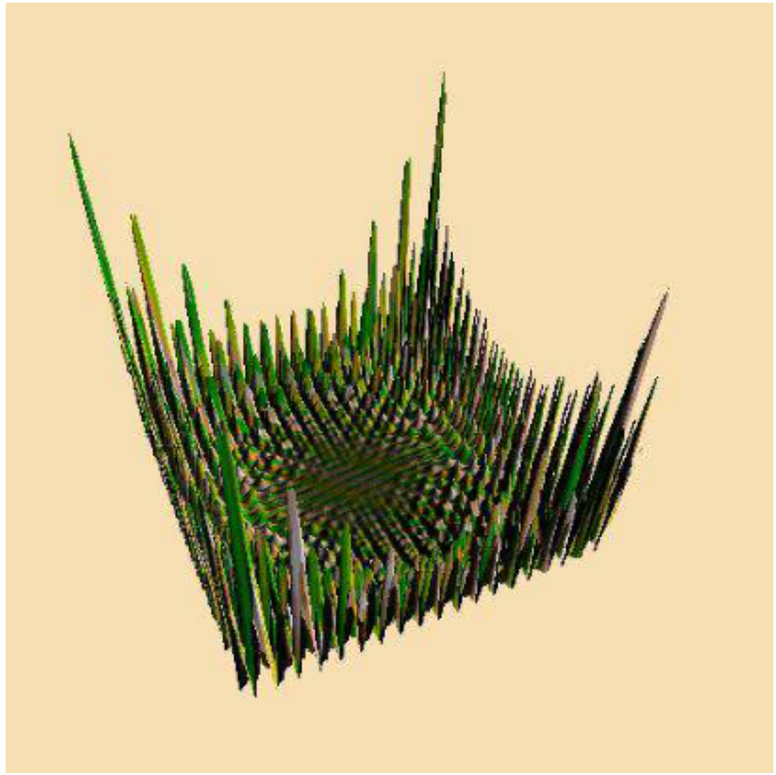
α_j the step size

A Difficult Minimization

Many local minima

Global Min at (0, 0)

In difficult problems, the gradient sequence will most likely get stuck in a local extremum of h



Simulated Annealing

This name is borrowed from Metallurgy:

A metal manufactured by a slow decrease of temperature (annealing)

Is stronger than a metal manufactured by a fast decrease of temperature.

The fundamental idea of simulated annealing methods

A change of scale, or temperature
◅ Allows for faster moves on the surface of the function h to maximize. ◅ Rescaling partially avoids the trapping attraction of local maxima.

As T decreases toward 0, the values simulated from this distribution become concentrated in a narrower and narrower neighborhood of the local maxima of h

Simulated Annealing - Metropolis Algorithm

- Simulation method proposed by Metropolis *et al.* (1953)
- Starting from θ_0, ζ is generated from

$$\zeta \sim \text{Uniform in a neighborhood of } \theta_0.$$

- The new value of θ is generated as

$$\theta_1 = \begin{cases} \zeta & \text{with probability } \rho = \exp(\Delta h/T) \wedge 1 \\ \theta_0 & \text{with probability } 1 - \rho, \end{cases}$$

  - $\Delta h = h(\zeta) - h(\theta_0)$
  - If $h(\zeta) \geq h(\theta_0)$, ζ is accepted
  - If $h(\zeta) < h(\theta_0)$, ζ may still be accepted
  - This allows escape from local maxima

Simulated Annealing

Simulated annealing typically modifies the temperature T at each iteration

It has the form

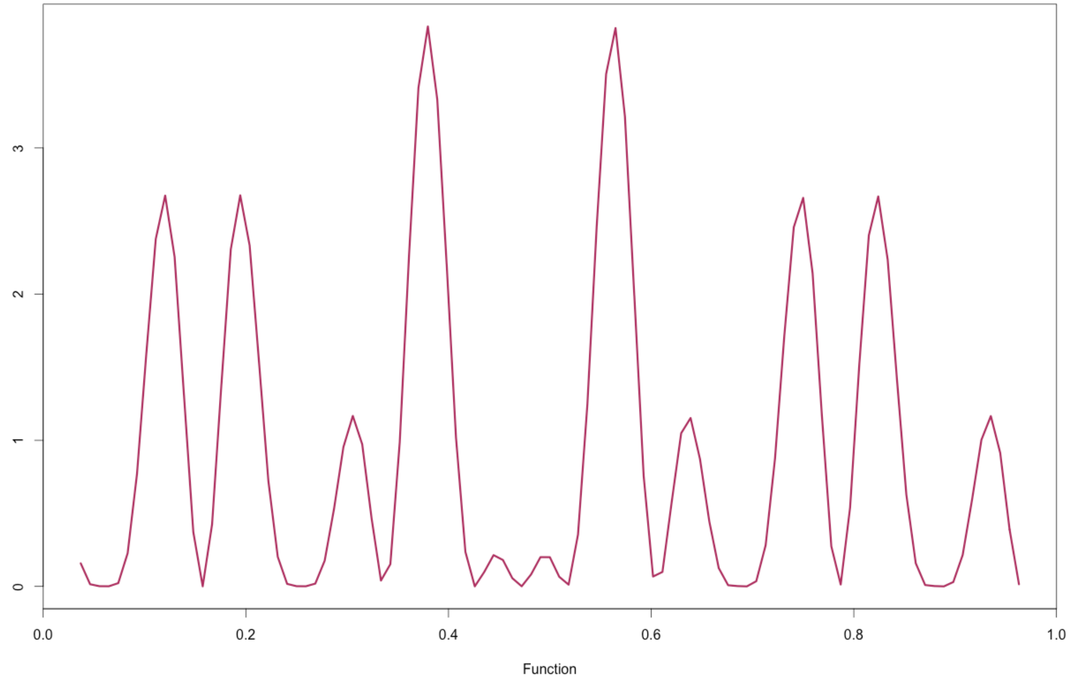
1. Simulate ζ from an instrumental distribution with density $g(|\zeta - \theta_i|)$;
2. Accept $\theta_{i+1} = \zeta$ with probability

$$\rho_i = \exp\{\Delta h_i / T_i\} \wedge 1;$$

   take $\theta_{i+1} = \theta_i$ otherwise.
3. Update T_i to T_{i+1}.

Simulated Annealing

Lets use a our function, h(x) = [cos(50x) + sin(20x)]^2

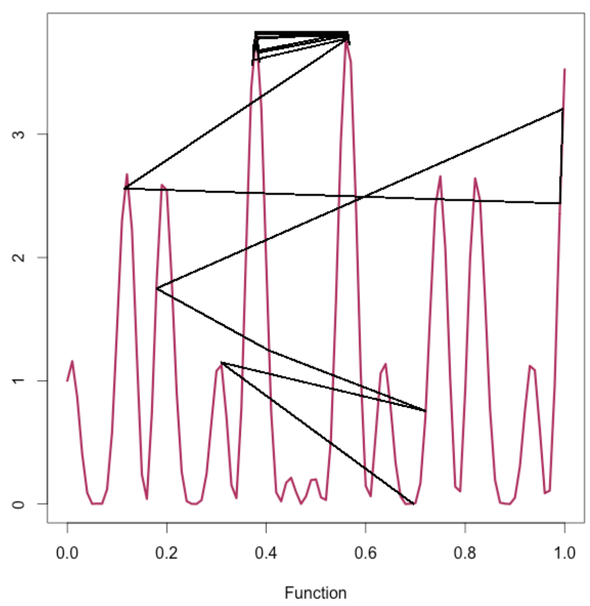


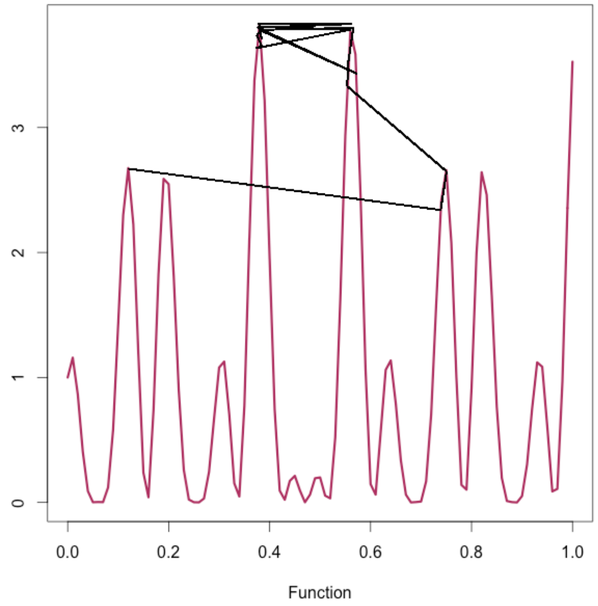
Simulated Annealing

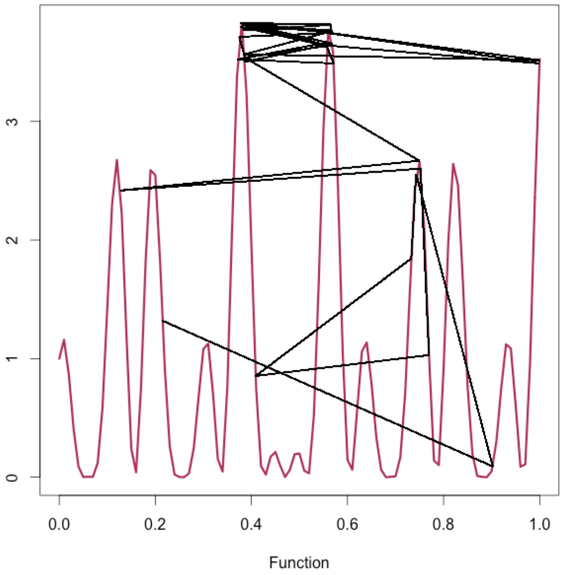
Set a temperature schedule of 1/log(1+x)

```
curve(h,xlab="Function",ylab="",lwd=3,col="maroon")

h=function(x){(cos(50*x)+sin(20*x))^2}

x=runif(1)

hval=hcur=h(x)

temp=1/log(1+1:10000)

diff=iter=1

while(diff>10^(-4)){

  prop=x[iter]+runif(1,-1,1)

  if((prop>1) || (prop <0) ||(log(runif(1))*temp[iter]>h(prop)-hcur))prop=x[iter]

  x=c(x,prop)

  hcur=h(prop)

  hval=c(hval,hcur)

  if ((iter>1000)&&(length(unique(x[(iter/2):iter]))>1))diff=max(hval)-max(hval[1:(iter/2)])

  iter=iter+1

  points(x,hval,type="l")

}
```

Simulated Annealing







Simulated Annealing Sudoku

We can use Simulated Annealing as a method to complete sudoku puzzles of any difficulty

Sudoku puzzles are logic based, combinatorial number-placement puzzles

Objective to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contains all of the digits from 1 to 9

Refresher: Sudoku puzzles fall under what type of experimental design?

Simulated Annealing Sudoku

A simple sudoku and its solution

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Simulated Annealing Steps for Sudoku

Randomly assign integers remaining integer values to small sub-squares to have a consistent sub-solution (all numbers 1- 9)

Score a puzzle by giving -1 for every unique element in each row or each column. Best solution has a score of -162.

Candidate for new puzzle is created by randomly selecting sub-square, then randomly flipping two of its entries, evaluating the new score. The delta_S is the difference between the scores.

Let T be the global temperature of our system, with a geometric schedule for decreasing (perhaps by T <- .99999 T).

If U is drawn uniformly from [0,1], and exp((delta_S/T)) > U,then we accept the candidate solution as our new state.

Scoring Schedule

Example score = -61 - 56 = - 117

1	5	6	9	4	7	1	7	5
7	2	3	5	8	3	2	3	9
8	4	9	1	2	6	6	4	8
6	9	2	9	5	1	4	7	1
8	1	3	4	7	3	9	6	5
4	7	5	8	2	6	3	2	8
6	1	5	8	1	5	2	9	3
7	4	2	3	6	9	1	7	6
8	3	9	4	2	7	8	4	5

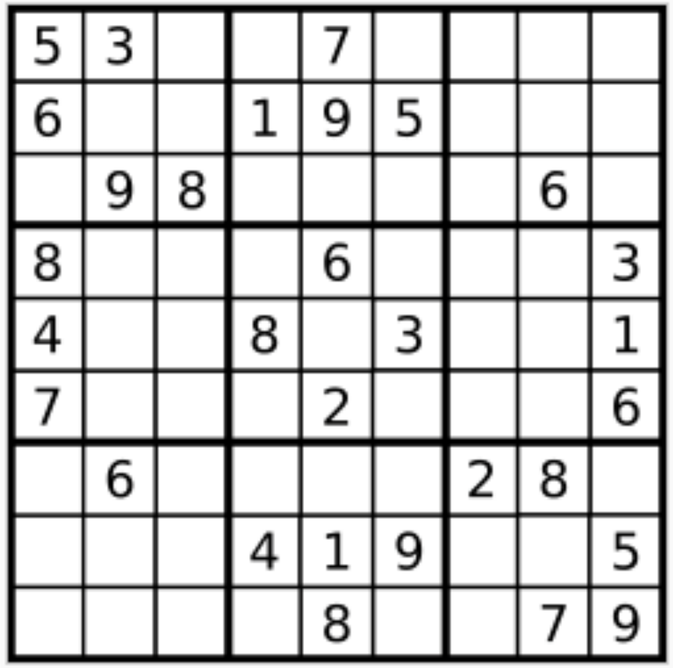
Scoring Schedule

Example score = -62 - 56 = - 118

1	5	6	9	4	7	1	7	5
7	2	3	5	8	3	2	3	9
8	4	9	1	2	6	6	4	8
6	9	2	3	5	1	4	7	1
8	1	3	4	7	9	9	6	5
4	7	5	8	2	6	3	2	8
6	1	5	8	1	5	2	9	3
7	4	2	3	6	9	1	7	6
8	3	9	4	2	7	8	4	5

Simulated Annealing Sudoku

First enter the sudoku



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
s=matrix(0,ncol=9,nrow=9)
s[1,c(1,2,5)]=c(5,3,7)
s[2,c(1,4:6)]=c(6,1,9,5)
s[3,c(2,3,8)]=c(9,8,6)
s[4,c(1,5,9)]=c(8,6,3)
s[5,c(1,4,6,9)]=c(4,8,3,1)
s[6,c(1,5,9)]=c(7,2,6)
s[7,c(2,7,8)]=c(6,2,8)
s[8,c(4:6,9)]=c(4,1,9,5)
s[9,c(5,8,9)]=c(8,7,9)
```

Simple Sudoku Solution

Need a scoring function

```
scor<-function(matr){

 scr=-1*(sum(apply(matr,1,function(x) length(unique(x))))+sum(apply(matr,2,function(x) length(unique(x)))));return(scr)}
```

We can also set the initial temperature

```
temp <- 0.5
```

Simple Sudoku Solution

I set the seed to set.seed(21) for reproducibility

```
[1] "iteration" 1000 "score" -154

[1] "iteration" 2000 "score" -149

[1] "iteration" 3000 "score" -155

[1] "iteration" 4000 "score" -158

[1] "iteration" 5000 "score" -158

[1] "puzzle solved"
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]
[1,]	5	3	4	6	7	8	9	1	2
[2,]	6	7	2	1	9	5	3	4	8
[3,]	1	9	8	3	4	2	5	6	7
[4,]	8	5	9	7	6	1	4	2	3
[5,]	4	2	6	8	5	3	7	9	1
[6,]	7	1	3	9	2	4	8	5	6

Medium Difficulty Sudoku

We can up the difficulty of the sudoku (taken from sudokusaviour.com on April 19, 2015. Rated 6/10)

				7		1		
7	9			4				
2		4	3					
		7					5	
8				9				4
	1		7			8		
					1	5		8
				8			2	1
		6		3				

Medium Difficulty Sudoku

I set the seed to set.seed(47) for reproducibility and temperature = 2

```
[1] "puzzle solved"
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]
[1,]	3	6	8	2	7	9	1	4	5
[2,]	7	9	1	8	4	5	2	6	3
[3,]	2	5	4	3	1	6	9	8	7
[4,]	9	4	7	1	2	8	3	5	6
[5,]	8	2	5	6	9	3	7	1	4
[6,]	6	1	3	7	5	4	8	9	2
[7,]	4	7	2	9	6	1	5	3	8
[8,]	5	3	9	4	8	7	6	2	1
[9,]	1	8	6	5	3	2	4	7	9

```
[1] 181543
```

Difficult Sudoku

For very, very difficult Sudokus this is not the most efficient way to complete

We may want to add some deterministic pruning at the outset to limit what randomly selected digits can be placed into each square

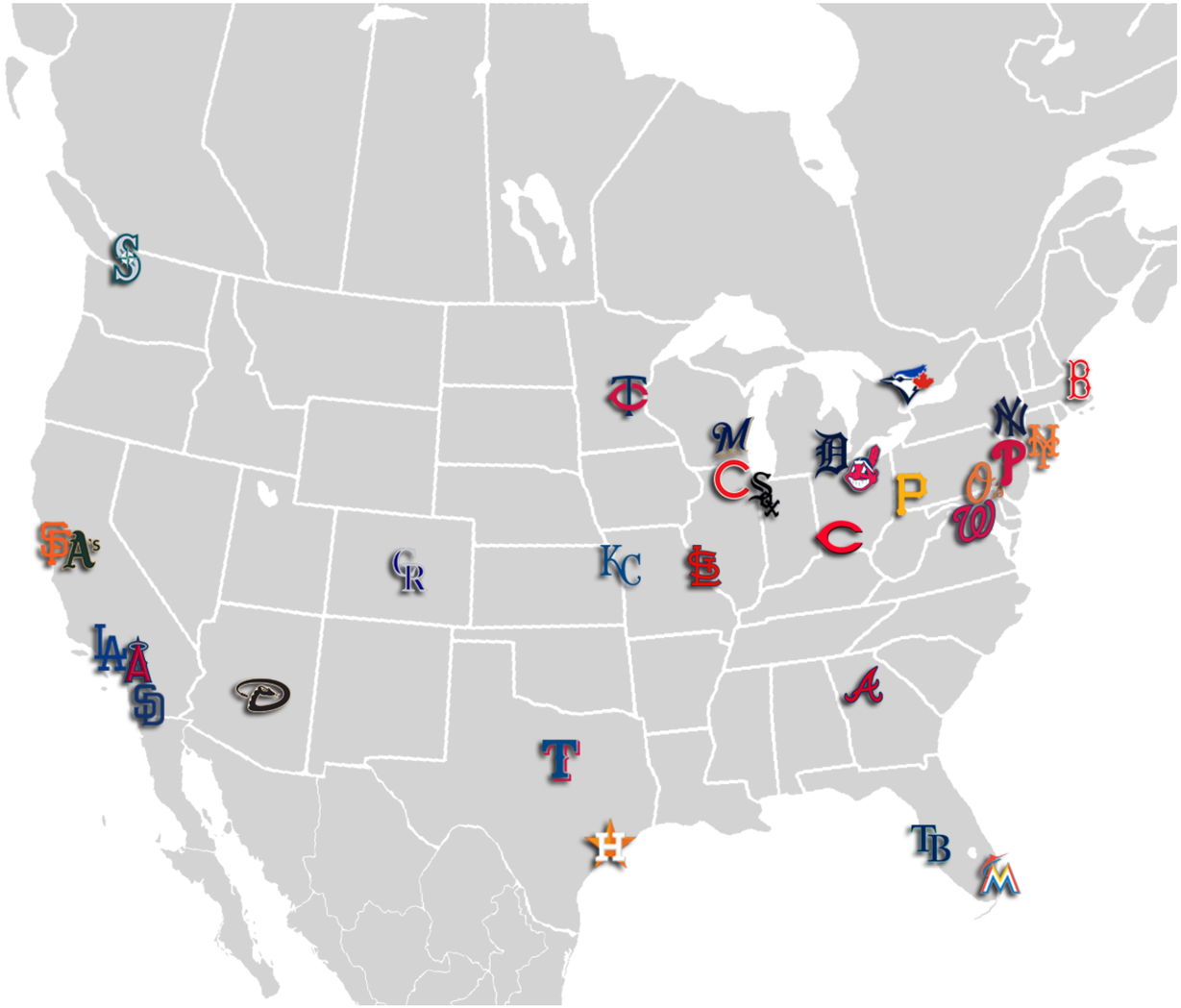
Simulated Annealing

Let's continue our study of Simulated Annealing through way of two of America's favorite past times

Road Trips

Baseball

Visit All 30 Stadiums



Traveling Salesman Problem

This can be thought of as the well-known "Traveling Salesman Problem"

The problem:

> Given n cities terms of their geographical coordinates C_i, i = 1,...,n, a tour of the cities is a path starting at C_i and leading from one city to the next in some order and finally back to C_i. Each city is to be visited exactly once (other than C_i). A tour is uniquely identified by a permutation x of the set {2,3,...,n}

Traveling Salesman Problem

The length of a tour is the length of its path. For the tour,

$$E(x) = \sum_{k=1}^{n-1} d(C_{i_{k-1}}, C_{i_k}) + d(C_{i_{n-1}}, C_{i_0}),$$

where $d(C_i, C_j)$ is the distance from city C_i to C_j. Take this length as the "energy" or objective E(x)

The traveling salesman problem (TSP) is that of finding the tour having the smallest length. Of course, the reverse of a given tour has the same length and for our purposes, can count as the same tour.

Traveling Salesman Solutions

For n cities there are $(n - 1)!$ permutations of the set $\{2, 3, \ldots, n\}$, and half that when accounting for reverse tours; this can be a large number for even small values of n. Hence finding the minimal tour is a very hard problem.

https://www.youtube.com/watch?v=SC5CX8drAtU

TSP Simulated Annealing

Start with an arbitrary initial tour from the set of all possible tours

Pick a new candidate tour at random from all neighbors of existing tour which may be better or worse than the existing (shorter or longer)

Neighboring tours can be defined through a partial path reversal scheme

For example suppose 3, 5, 2, 8, 4, 6, 7 is a tour for an 8-city TSP and suppose the terms $i_{k1} = 5$ and $i_{k2} = 4$ are selected. Then the resulting PPR tour is 3, 4, 8, 2, 5, 6, 7.

A visual representation of this is

3,**4,2,8,5,**6,7

changes to

3,**5,8,2,4**,6,7.

TSP Simulated Annealing

If a candidate tour is better than the existing tour accept it as the new tour

If the candidate tour is worse than the existing tour, possibly still accept it according to some probability

- The prob of accepting an inferior tour is a function of how much longer the candidate is compared to the current tour and the temp of the annealing process

- At higher temps we are more likely to accept inferior tours

TSP Simulated Annealing

Return to the random selection of a neighboring tour and repeat many times.

Each step reduce the temperature a bit until arrival at in this case the minimum (hopefully global, possibly local) tour.

Can change to a different temperature cooling schedule if necessary

TSP Simulated Annealing

Important step is the considering a tour that is worse than that tour we already have

Sometimes accepting the worse tour temporarily because it might be the step that gets us out of a local minimum and ultimately closer to the global min

“Sometimes things really have to get worse before they can get better”

Simulated Annealing Shiny App

Fortunately, someone has implemented this algorithm for us in Shiny

Shiny is a great way to make interactive html pages with R code

Uses great circle distances rather than the (ideal) driving distances

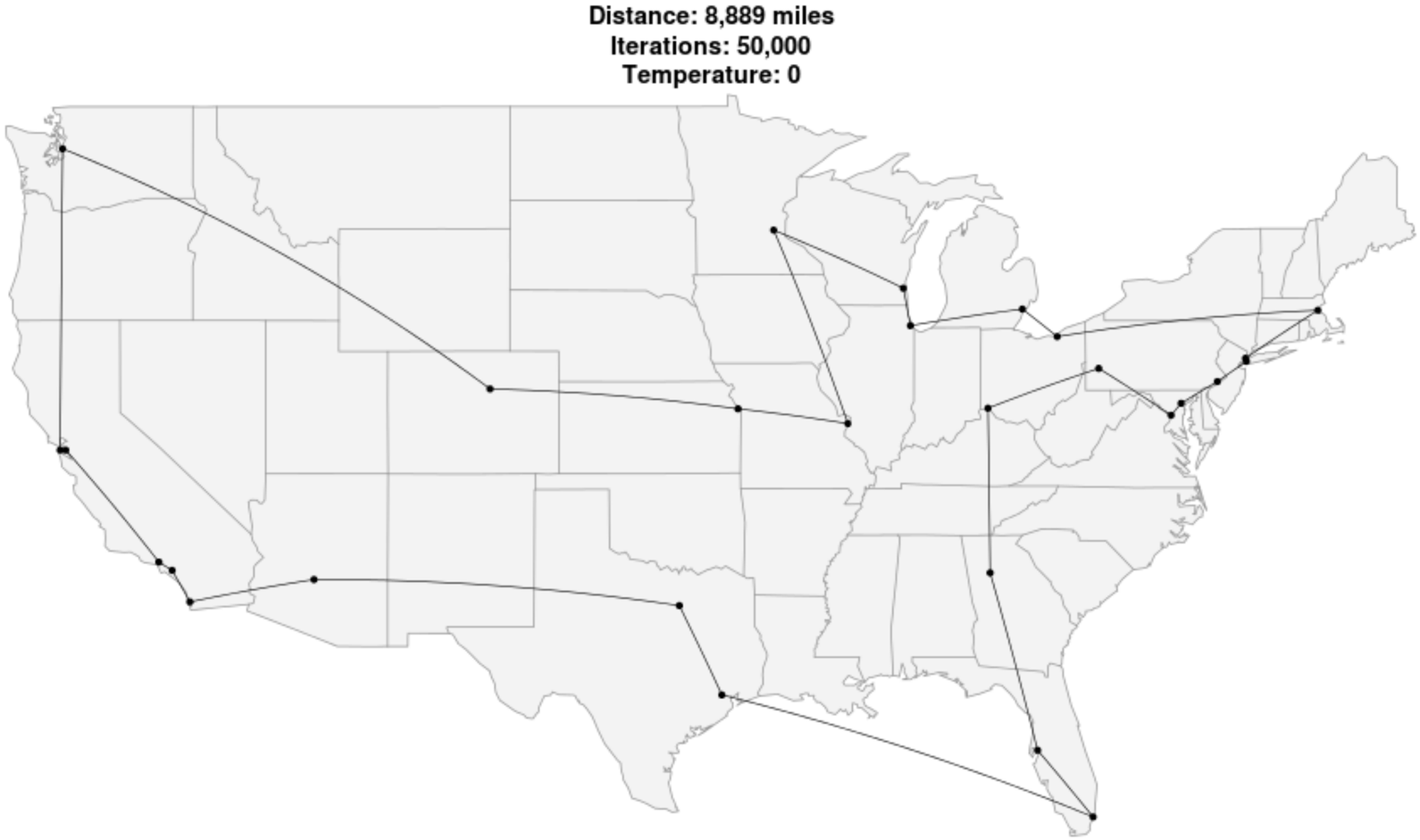
Uses an s-curve cooling schedule 1/(1+e^((t-center)/width))

Doesn't include Canada so we will leave out Toronto for our stadium tour

Baseball Roadtrip

Show Shiny App

Baseball Roadtrip



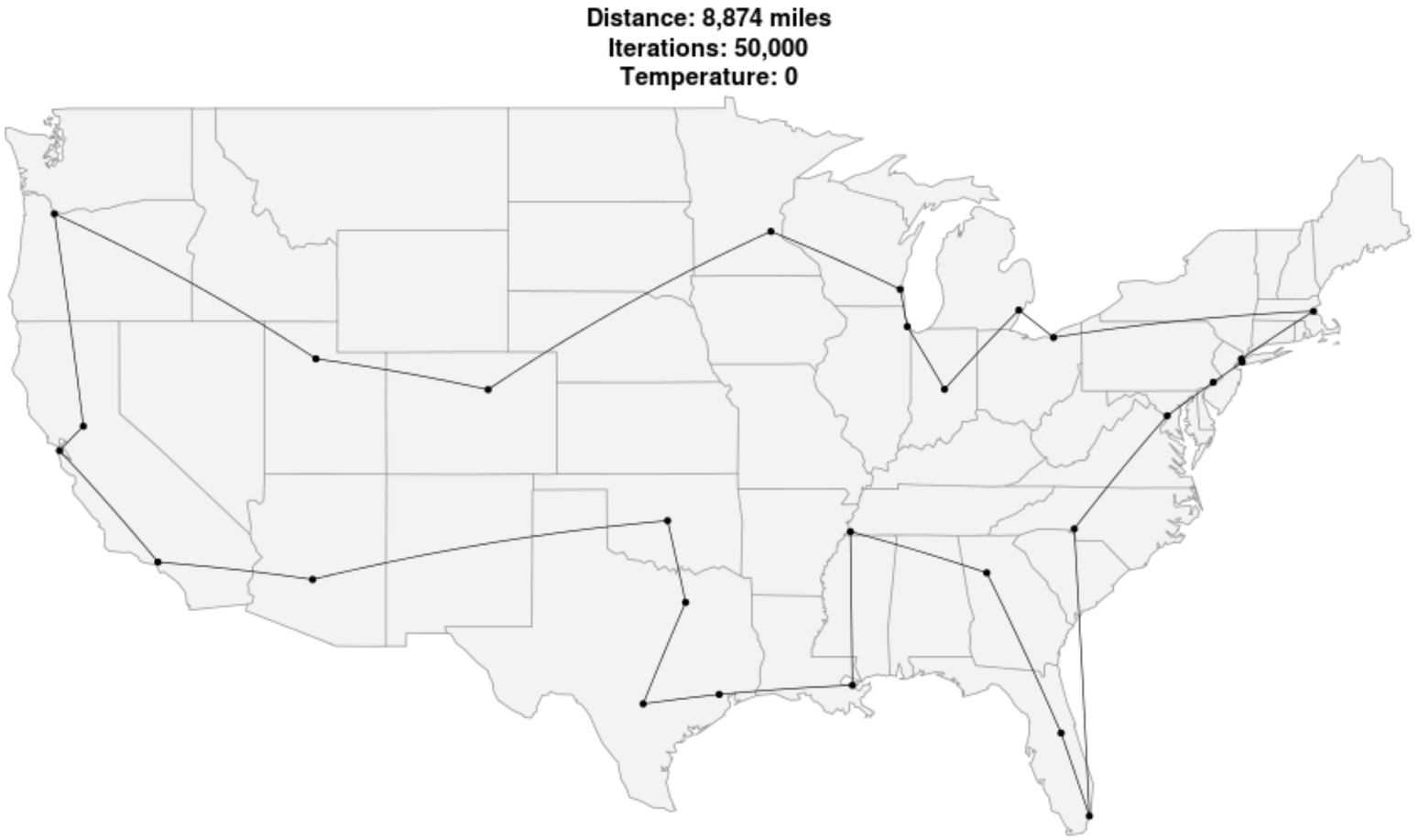
Simulated Annealing Google Map

Just for fun I entered this route in to google maps to get an estimate driving time and distance keeping the order of the cities the same

Total Mileage = 10624 miles and Expected Driving Time = 158 hours

NBA Roadtrip

NBA Roadtrip



Why the Annealing step?

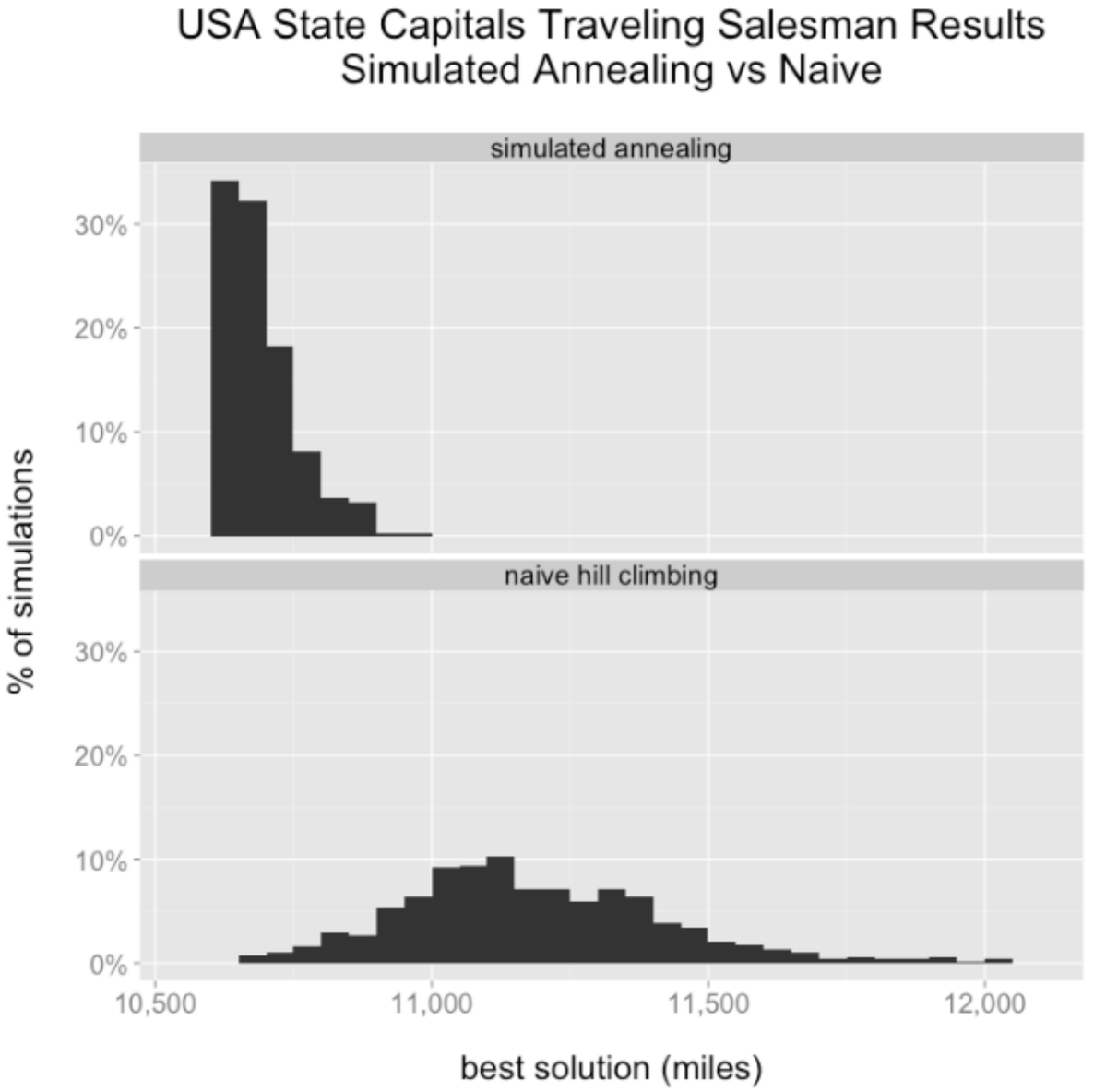
Why not do the same process with 0 temperature?

> 0 temperature means accepting the new tour if and only if it's better the the existing tour

Following this naive strategy we are far more likely to get stuck in a local minimum

1000 trials with and

without the annealing step



Choice of Temperature Schedule

So far we have considered 3 temperature schedules

T = 0.99999 * T (sudoku)

T = 1/(1+e^((t-center)/width)) (TSP)

T = 1/log(1 + T) (cos & sin function)

There exist theoretical results about temp schedules that guarantee convergence, but they have little practical value because they depend on problem related calibration

General recommendation for the temp decrease is that is should be logarithmic rather than geometric

Choice of Temperature Schedule

Adaptive strategies that update the temperature after learning episodes of several iterations that evaluate acceptance rates and maximum increase are recommended

Validation of these are mostly empirical

Choice of Temperature Schedule

Using three different temperature schedule we can show that the faster it decreases to zero, the less likely the sequence is to leave the current mode

We return to h(x) = [cos(50x) + sin(20x)]^2 and the temperature functions

T = .99999*T

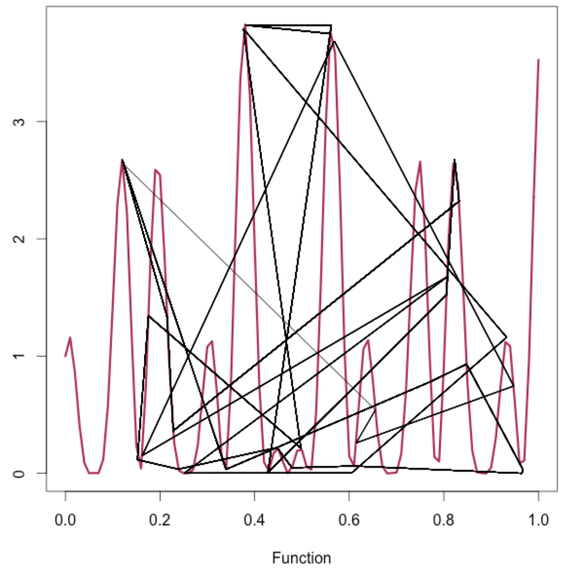
T = 1/log(1 + T)

T = 1/log(1 + T)^2

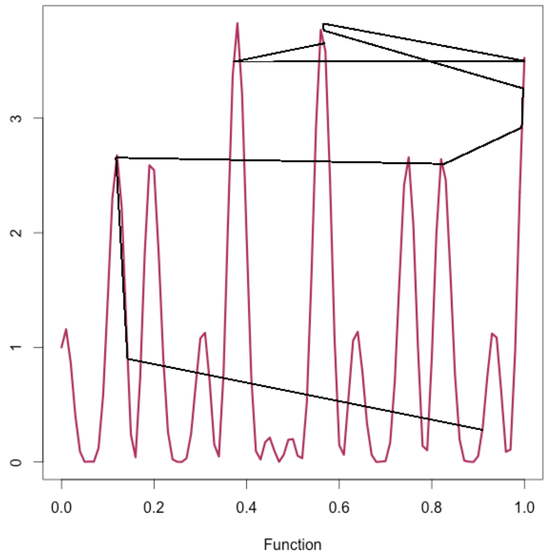
Choice of Temperature Schedule

Running 100 simulated annealing sequences for each of the 3 temperature schedules with only 100 iterations

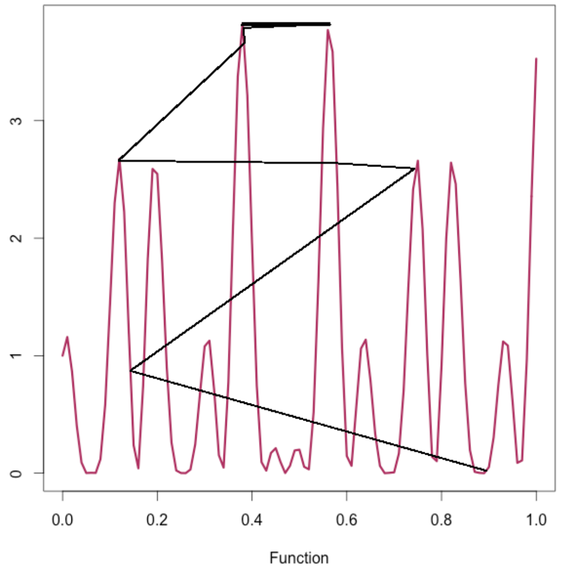
$$T = 0.99999 * T$$



$$T = \frac{1}{log(1+T)}$$



$$T = \frac{1}{log(1+T)^2}$$



Genetic Algorithms

A search heuristic

Generate a vast number of possible model solutions used to evolve towards an approximation of the best solution of the model

Mimics evolution in nature

Inspired by Darwin's theory about evolution

Simply said, solution to a problem solved by genetic algorithms is evolved

Chromosomes

All living organisms consist of cells

In each cell there is the same set of **chromosomes**.

A chromosome consist of **genes**, blocks of DNA. Each gene encodes a particular protein.

Basically can be said, that each gene encodes a **trait**, for example color of eyes.

Possible settings for a trait (e.g. blue, brown) are called **alleles**.

Each gene has its own position in the chromosome called the **locus**

Reproduction

During reproduction, first occurs **recombination** (or **crossover**). Genes from parents form in some way the whole new chromosome.

The new created offspring can then be mutated. **Mutation** means, that the elements of DNA are a bit changed. This changes are mainly caused by errors in copying genes from parents.

The **fitness** of an organism is measured by success of the organism in its life.

Genetic Algorithms

Generates a population where the individuals in this population (chromosomes) have a given state

Once the population is generated, the state of these individuals is evaluated and graded on their value (fitness).

The best individuals are then taken and crossed-over – in order to hopefully generate 'better' offspring – to form the new population.

In some cases the best individuals in the population are preserved in order to guarantee 'good individuals' in the new generation (this is called *elitism*).

Genetic Algorithms Outline

1. **[Start]** Generate random population of n chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness f(x) of each chromosome x in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
    1. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
    2. **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
    3. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
    4. **[Accepting]** Place new offspring in a new population
4. **[Replace]** Use new generated population for a further run of algorithm
5. **[Test]** If the end condition is satisfied, stop, and return the best solution in current population
6. **[Loop]** Go to step 2

Crossover

Most widely used technique is one-point crossover

The initial sequence of k bits of one parent is concatenated with the bits beyond the kth position of the second parent to produce offspring

Parent 1 11011 00100110110
Parent 2 11011 11000011110

Offspring 1 11011 11000011110
Offspring 2 11011 00100110110

There are other ways how to make crossover, for example we can choose more crossover points. Crossover can be rather complicated and very depends on encoding of the encoding of chromosome. Specific crossover made for a specific problem can improve performance of the genetic algorithm.

Other Crossovers

Two point crossover - two crossover point are selected, binary string from beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent

Uniform crossover - bits are randomly copied from the first or from the second parent

Arithmetic crossover - some arithmetic operation is performed to make a new offspring

Mutation

After crossover, mutation takes place

This is to prevent falling all solutions in population into a local optimum of solved problem. Mutation changes randomly the new offspring. For binary encoding we can switch a few randomly chosen bits from 1 to 0 or from 0 to 1

Original offspring 1 1101111000011110
Mutated offspring 1 1100111000011110

Original offspring 2 1101100100110110
Mutated offspring 2 1101101100110110

Crossover Probability

Determines how often crossover will be performed

if **100%** then all offspring is made by crossover

if **0%** whole new generation is made from exact copies of chromosomes from old population

If no crossover, offspring is exact copy of parents

If crossover, offspring is made from parts of parents' chromosome

done in hope that new chromosomes will have good parts of old chromosomes and maybe the new chromosomes will be better

also good to leave some part of population survive to next generation

Mutation Probability

Determines how often parts of chromosomes be mutated

if **100%**, whole chromosome is changedno mutation, offspring is taken after crossover (or copy) without any change

if **0%**, nothing is changed

mutation is performed, part of chromosome is changed

prevents GA from falling into local extreme, but it should not occur very often, because then GA will in fact change to **random search**.

Population Size

Determines how many chromosomes in the population (in one generation)

Too few chromosomes, there are only a few possibilities to perform crossover and only a small part of search space is explored

Too many chromosomes, GA slows down

Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to increase population size, because it does not make solving the problem faster

Roulette Wheel Selection

Parents are selected according to their fitness

The better the chromosomes are, the more chances to be selected they have

Imagine a **roulette wheel** where all chromosomes in the population have its size accordingly to its fitness function

Then a marble is thrown in the wheel and where the marble stops will select the chromosome

Obviously, chromosome with bigger fitness will be selected more times

Rank Selection

If fitnesses differ greatly then roulette wheel selection will pick same parents too often

For example, if the best chromosome fitness is 90% of all the roulette wheel then the other chromosomes will have very few chances to be selected

Rank selection first ranks the population and then every chromosome receives fitness from this ranking

The worst will have fitness ***1***, second worst ***2*** etc. and the best will have fitness ***N*** (number of chromosomes in population)

Elitism

When creating new population by crossover and mutation, we have a big chance, that we will loose the best chromosome.

Elitism first copies the best chromosome (or a few best chromosomes) to new population

Then we proceed with the classical way.

This can very rapidly increase performance of GA, because it prevents losing the best found solution

Recommendations for GA

Crossover rate generally should be high, about **80%-95%**

Mutation rate should be very low. Best rates reported are about **0.5%-1%**

Surprisingly, very big population size usually does not improve performance. Good population size have been reported between 20 - 100, but can also depend on size of encoded string

Basic roulette wheel selection can be used, but sometimes rank selection can be better. Definitely use elitism.

Encoding, crossover, and mutation type all are dependent on the problem type

Knapsack Problem (from wikipedia)

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography and applied mathematics.

Knapsack Problem

The knapsack problem occurs commonly in role-playing games, where the player character is constrained by their encumbrance threshold when carrying items and treasure, which regularly forces the player to evaluate the items' value-to-weight ratio in order to bring only the most valuable items to a merchant.

The Elder Scrolls series

The *Dungeons and Dragons* game

Also in the book Cryptonomincon used in a example of a way to distribute family heirlooms

Our Knapsack Problem

You are going to spend a month in the wilderness. You're taking a backpack with you, however, the maximum weight it can carry is 20 kilograms. You have a number of survival items available, each with its own number of "survival points." Your objective is to maximize the number of survival points.

Available Choices

The following table shows the items you can choose from for your knapsack

Item	Survival	Weight
pocketknife	10	1
beans	20	5
potatoes	15	10
onions	2	1
sleeping bag	30	7
rope	10	5
compass	30	1

Available Choices

Defining the dataset and weight constraint

```
dataset <- data.frame(item = c("pocketknife", "beans",
"potatoes", "onions",
    "sleeping bag", "rope", "compass"), survivalpoints = c(10,
20, 15, 2, 30,
    10, 30), weight = c(1, 5, 10, 1, 7, 5, 1))
weightlimit <- 20
```

Evaluation (fitness) Function

The evaluation function will evaluate the different individuals (chromosomes) of the population on the value (fitness) of their gene configuration.

An individual can for example have the following gene configuration: **1001100**

Each number in this binary string represents whether or not to take an item with you. A value of 1 refers to putting the specific item in the knapsack while a 0 refers to leave the item at home.

Evaluation (fitness) Function

Given the example gene configuration: **1001100**

we would take the following items:

```
chromosome = c(1, 0, 0, 1, 1, 0, 0)
dataset[chromosome == 1, ]
##            item survivalpoints weight
## 1  pocketknife              10      1
## 4        unions               2      1
## 5 sleeping bag              30      7
```

Evaluation (fitness) Function

The evaluation function will provide a value to the gene configuration of a given chromosome

We can check to what amount of survival points this configuration sums up to.

```
cat(chromosome %*% dataset$survivalpoints)
## 42
```

The GA code we will use in R wants to optimize to a minimum value so we will calculate the value above and multiply by -1

Also, a gene configuration that exceeds the weight constraint returns a value of 0

Evaluation (fitness) Function

The evaluation function

```
evalFunc <- function(x){
    current_solution_survivalpoints <- x %*% dataset$survivalpoints
    current_solution_weight <- x %*% dataset$weight
    if (current_solution_weight > weightlimit)
    return(0) else return(-current_solution_survivalpoints)
}
```

GA implementation in R

The evaluation function

```
iter = 100

GAmodel <- rbga.bin(size = 7, popSize = 100, iters =
iter, mutationChance = 0.01, elitism = T, evalFunc =
evalFunc)

cat(summary.rbga(GAmodel))
```

GA function

R Based Genetic Algorithm (binary chromosome)

Description

A R based genetic algorithm that optimizes, using a user set evaluation function, a binary chromosome which can be used for variable selection. The optimum is the chromosome for which the evaluation value is minimal.

It requires a `evalFunc` method to be supplied that takes as argument the binary chromosome, a vector of zeros and ones. Additionally, the GA optimization can be monitored by setting a `monitorFunc` that takes a `rbga` object as argument.

Results can be visualized with `plot.rbga` and summarized with `summary.rbga`.

Usage

```
rbga.bin(size=10,
         suggestions=NULL,
         popSize=200, iters=100,
         mutationChance=NA,
         elitism=NA, zeroToOneRatio=10,
         monitorFunc=NULL, evalFunc=NULL,
         showSettings=FALSE, verbose=FALSE)
```

Arguments

`size`	the number of genes in the chromosome.
`popSize`	the population size.
`iters`	the number of iterations.
`mutationChance`	the chance that a gene in the chromosome mutates. By default 1/(size+1). It affects the convergence rate and the probing of search space: a low chance results in quicker convergence, while a high chance increases the span of the search space.
`elitism`	the number of chromosomes that are kept into the next generation. By default is about 20% of the population size.
`zeroToOneRatio`	the change for a zero for mutations and initialization. This option is used to control the number of set genes in the chromosome. For example, when doing variable selectionm this parameter should be set high to
`monitorFunc`	Method run after each generation to allow monitoring of the optimization
`evalFunc`	User supplied method to calculate the evaluation function for the given chromosome
`showSettings`	if true the settings will be printed to screen. By default False.
`verbose`	if true the algorithm will be more verbose. By default False.
`suggestions`	optional list of suggested chromosomes

GA Knapsack Results

Our results

```
GA Settings

  Type                  = binary chromosome

  Population size       = 100

  Number of Generations = 100

  Elitism               = TRUE

  Mutation Chance       = 0.01


Search Domain

  Var 1 = [,]

  Var 0 = [,]


GA Results

  Best Solution : 1 1 0 1 1 1 1
```

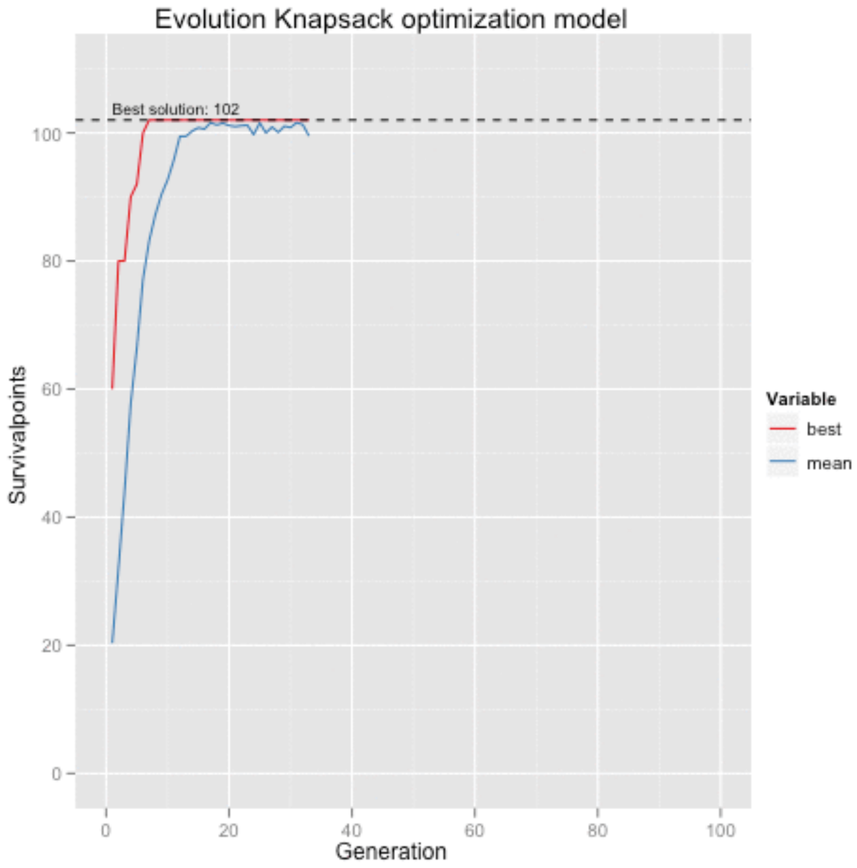
GA Knapsack Results

The resulting knapsack and total number of survival points

```
> solution = c(1, 1, 0, 1, 1, 1, 1)

> dataset[solution == 1, ]

          item survivalpoints weight

1  pocketknife             10      1

2        beans             20      5

4       onions              2      1

5 sleeping bag             30      7

6         rope             10      5

7      compass             30      1


> cat(paste(solution %*% dataset$survivalpoints, "/", sum(dataset$survivalpoints)))

102 / 117
```

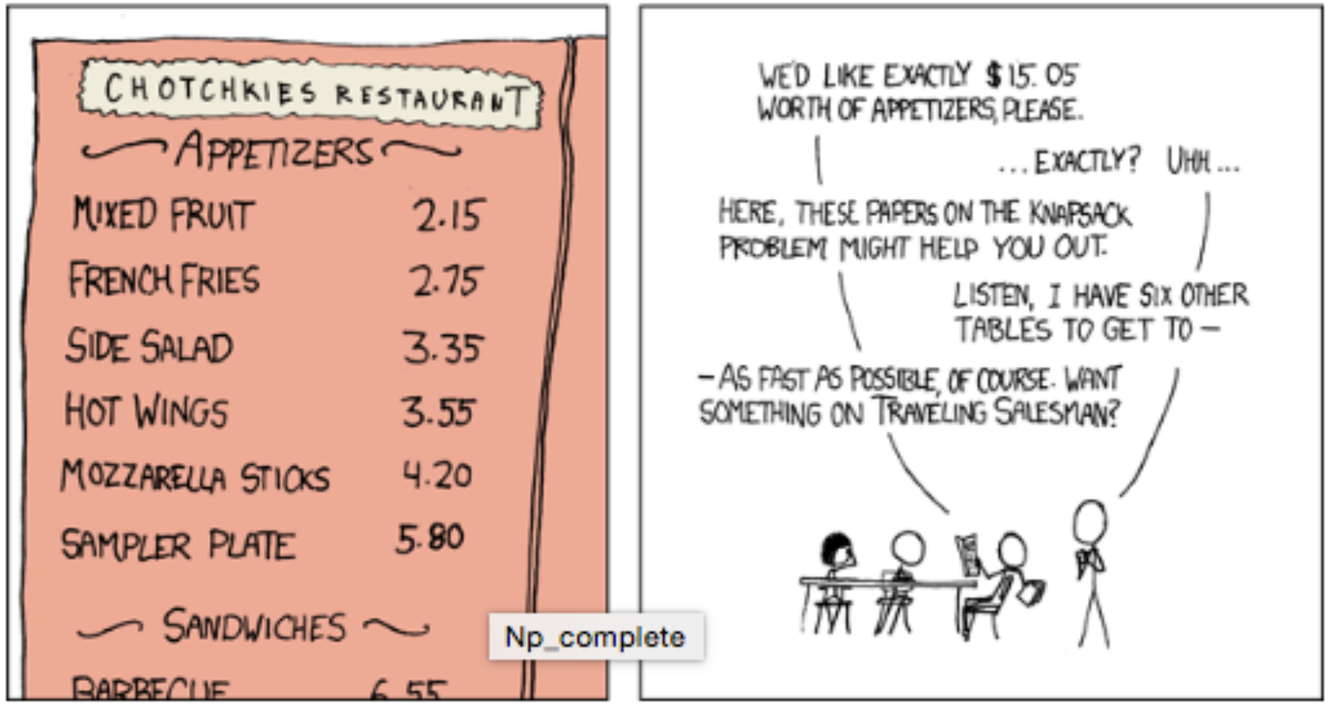
GA Visualization

The x-axis denotes the different generatic olution of the entire population of that generatio solution of that generation. As you can se erations to hit the best solution, after that it is just the population of subsequent generations evc



MY HOBBY:

EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



Returning to the TSP

Permutation encoding can be used in ordering problems, such as Traveling Salesman Problem or task ordering problem.
In permutation encoding, every chromosome is a string of numbers, which represents number in a sequence.

Chromosome A
1 5 3 2 6 4 7 9 8
Chromosome B
8 5 6 7 2 3 1 4 9

Permutation encoding is only useful for ordering problems. Even for this problems for some types of crossover and mutation corrections must be made to leave the chromosome consistent (i.e. have real sequence in it).

Example of Problem: Traveling salesman problem (TSP)

The problem: There are cities and given distances between them. Traveling salesman has to visit all of them, but he does not to travel very much. Find a sequence of cities to minimize travelled distance.

Encoding: Chromosome says order of cities, in which salesman will visit them.