

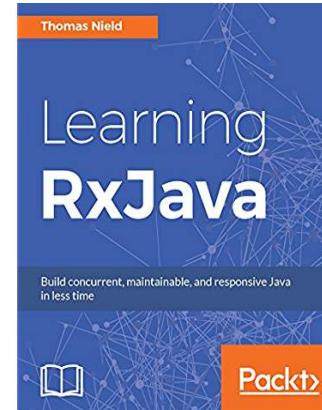
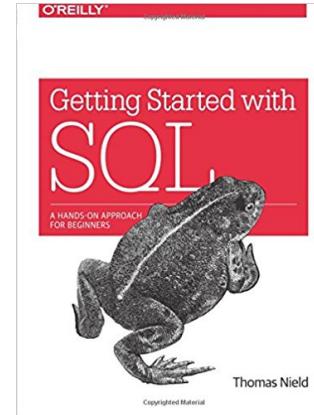
Thomas Nield  
KotlinConf 2018

# Thomas Nield

Business Consultant at Southwest Airlines  
Dallas, Texas

## Author

*Getting Started with SQL* by O'Reilly  
*Learning RxJava* by Packt



## Trainer and content developer at O'Reilly Media

## OSS Maintainer/Collaborator

RxKotlin

TornadoFX

RxJavaFX

Kotlin-Statistics

RxKotlinFX



thomasnield9727



<https://github.com/thomasnield>



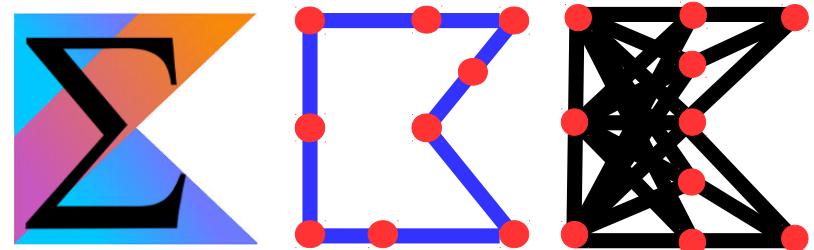
Thomas Nield

# Agenda

## Why Learn Mathematical Modeling?

### Discrete Optimization

- Traveling Salesman
- Classroom Scheduling
- Solving a Sudoku



### Machine Learning

- Naive Bayes
- Neural Networks

# **Part I:** **Why Learn Mathematical Modeling?**



# **Modeling Uncertainty and Approximation**

**As programmers, we thrive in certainty and exactness.**

**But the valuable, high-profile problems today often tackle uncertainty and approximation.**

Users expect “smarter” applications that will predict what they want.

Machine learning and optimization is nondeterministic and inevitably has error.

Many search spaces are too large to exhaustively explore.

# What is Mathematical Modeling?

**Mathematical modeling is a broad discipline that attempts to solve real-world problems using mathematical concepts.**

Applications range broadly, from *biology* and *medicine* to *engineering, business, and economics*.

Mathematical Modeling is used heavily in optimization, machine learning, and data science.

## Real-World Examples

Product recommendations

Staff/resource scheduling

Text categorization

Dynamic pricing

Image/audio recognition

DNA sequencing

Sport event planning

Game "AI" (Sudoku, Chess)

Disaster Management

# Why Learn Mathematical Modeling?

**Become effective at solving difficult, non-deterministic problems that require machine learning and optimization.**

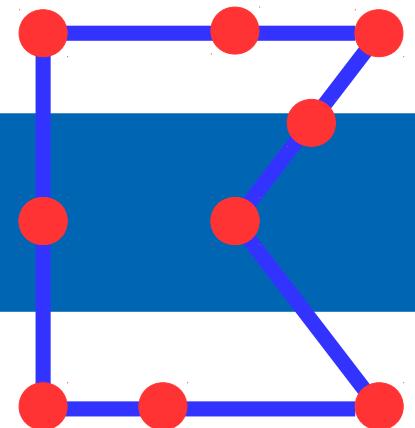
**Technologies, frameworks, and languages come and go... but math never changes.**

Classic mathematical techniques have survived decades (even centuries) and are not prone to obsolescence and deprecation.

Knowing mathematical modeling will greatly improve your professional marketability for decades to come, without constant investment in learning new technologies.

**Math does not get deprecated.**

## Part II: Discrete Optimization



# What Is Discrete Optimization?

**Discrete optimization is a space of algorithms that tries to find a feasible or optimal solution to a constrained problem.**

Scheduling classrooms, staff, transportation, sports teams, and manufacturing

Finding an optimal route for vehicles to visit multiple destinations

Optimizing manufacturing operations

Solving a Sudoku or Chess game

**Discrete optimization is a mixed bag of algorithms and techniques, which can be built from scratch or with the assistance of a library.**

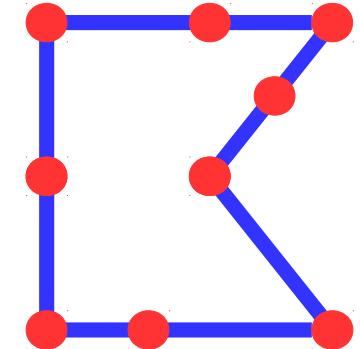
# Traveling Salesman Problem

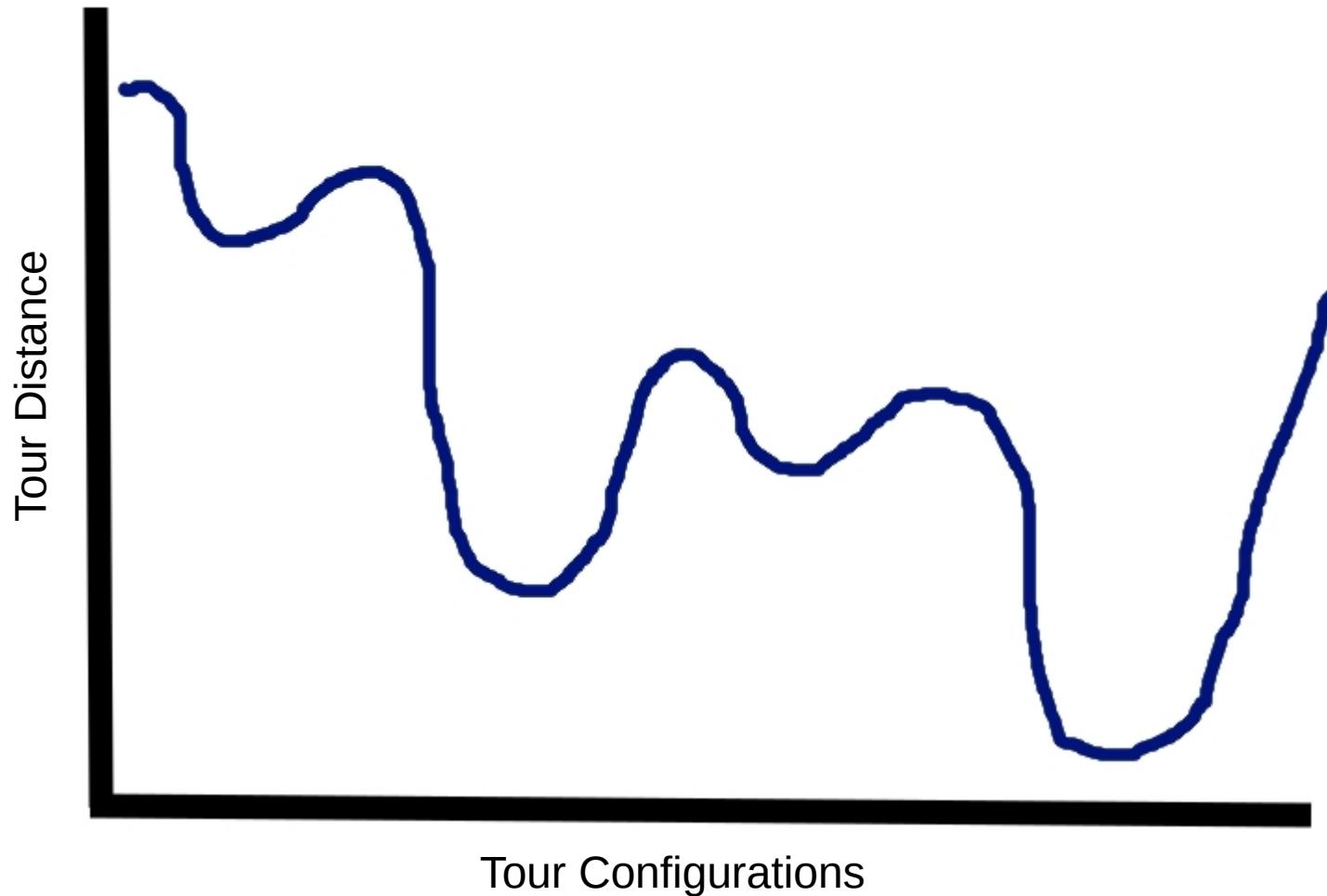
**The Traveling Salesman Problem (TSP)** is one of the most elusive and studied computer science problems since the 1950's.

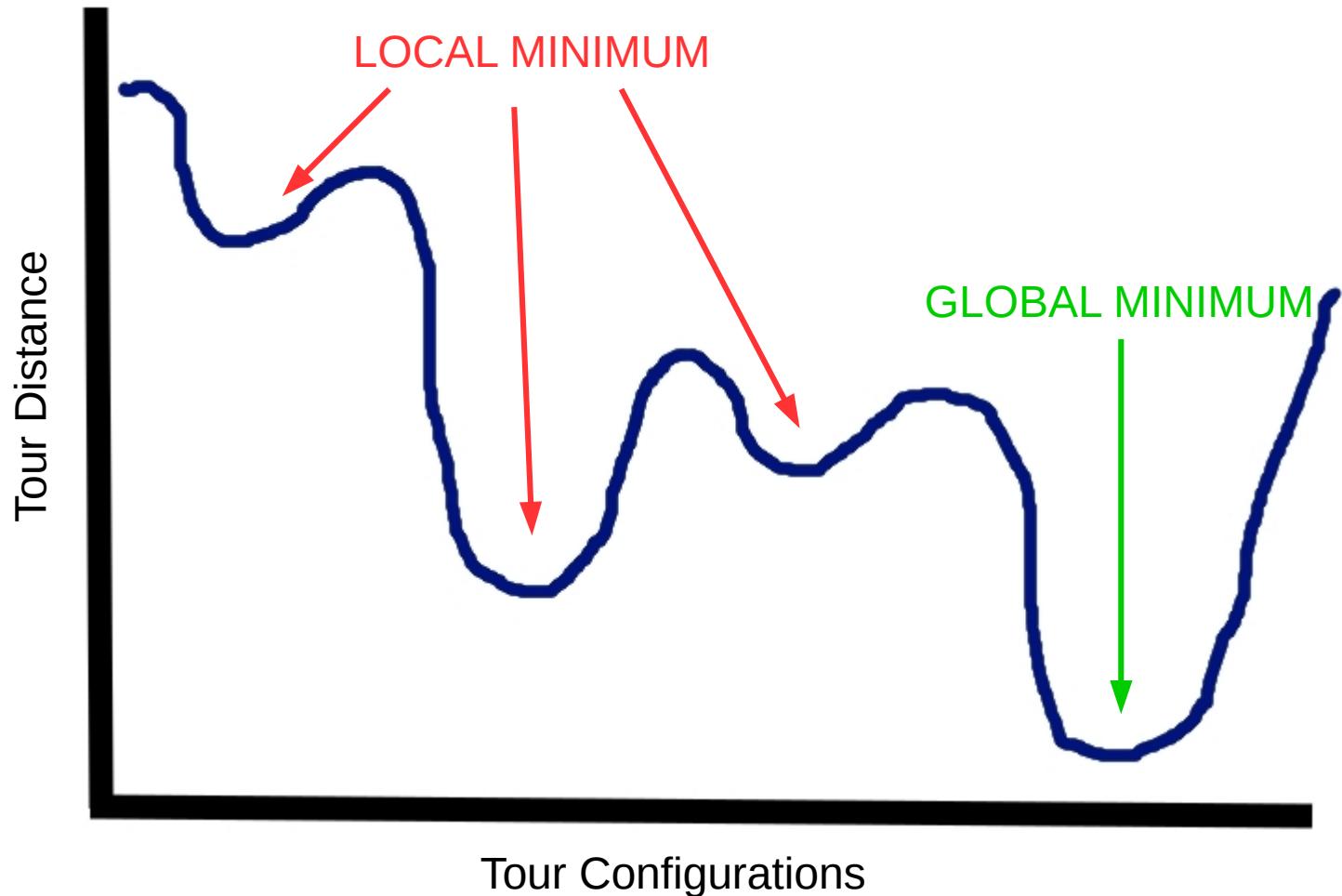
**Objective:** Find the shortest round-trip tour across several geographic points/cities.

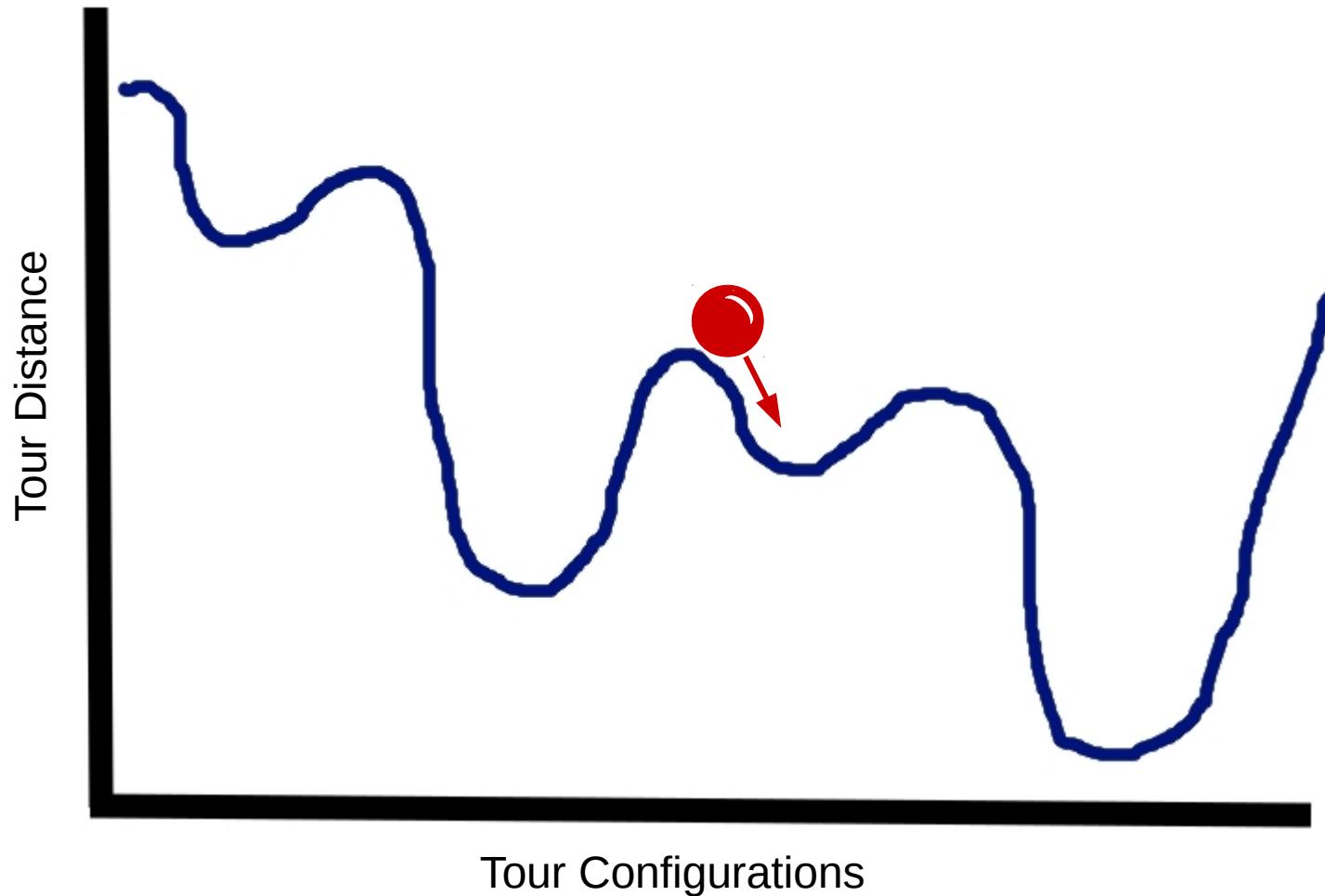
**The Challenge:** Just **60** cities =  **$8.3 \times 10^{81}$**  possible tours

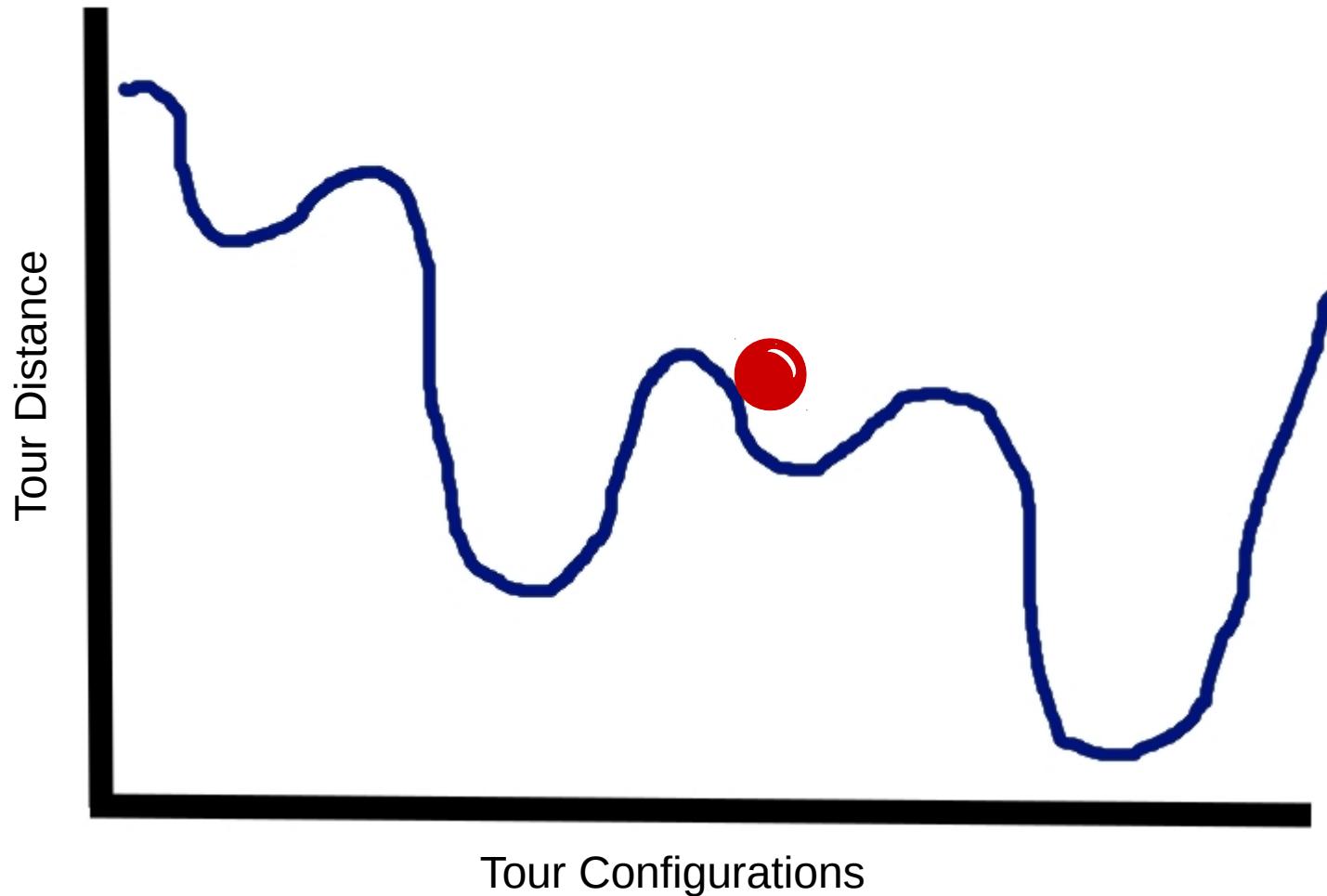
*That's more tour combinations than there are observable atoms in the universe!*

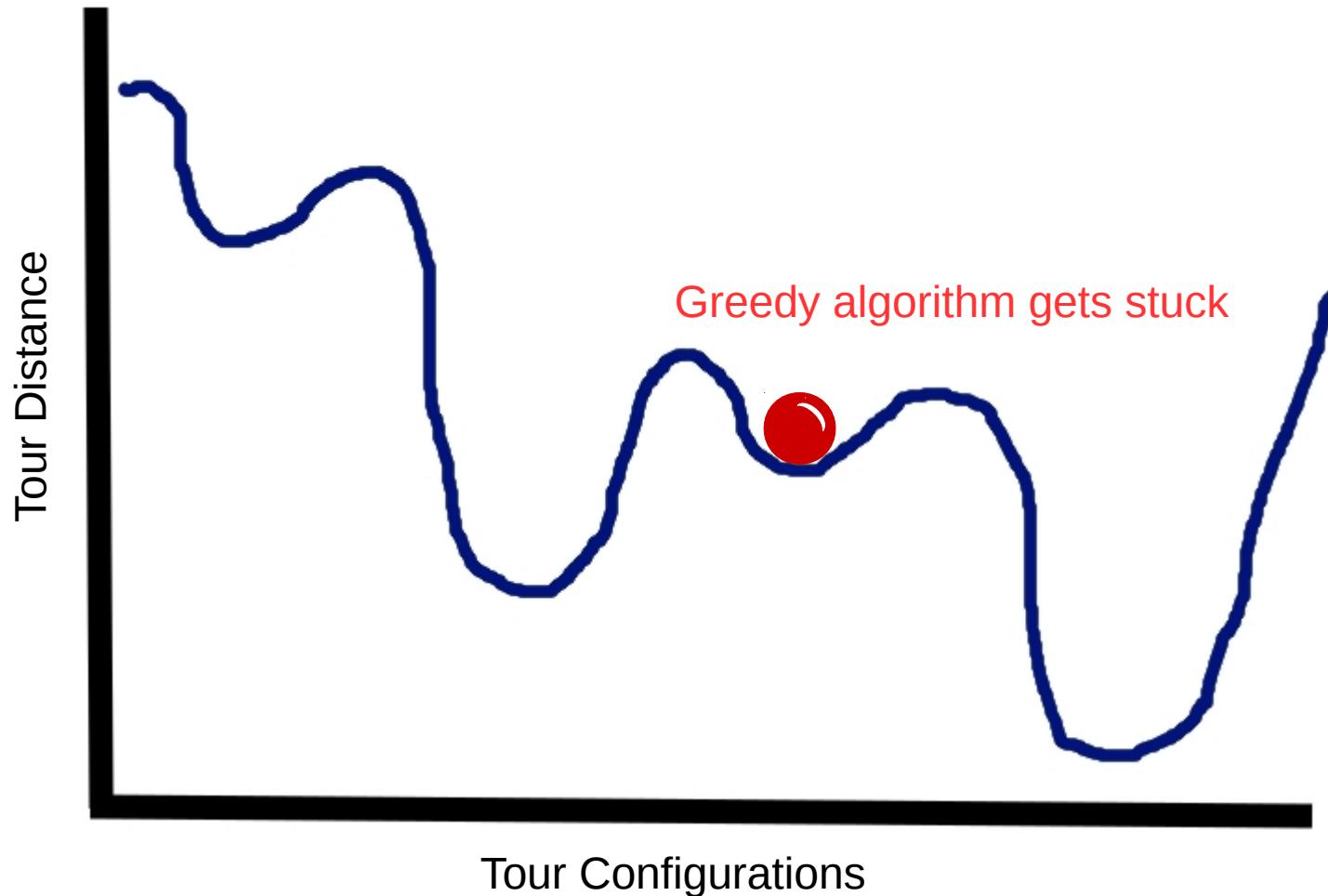


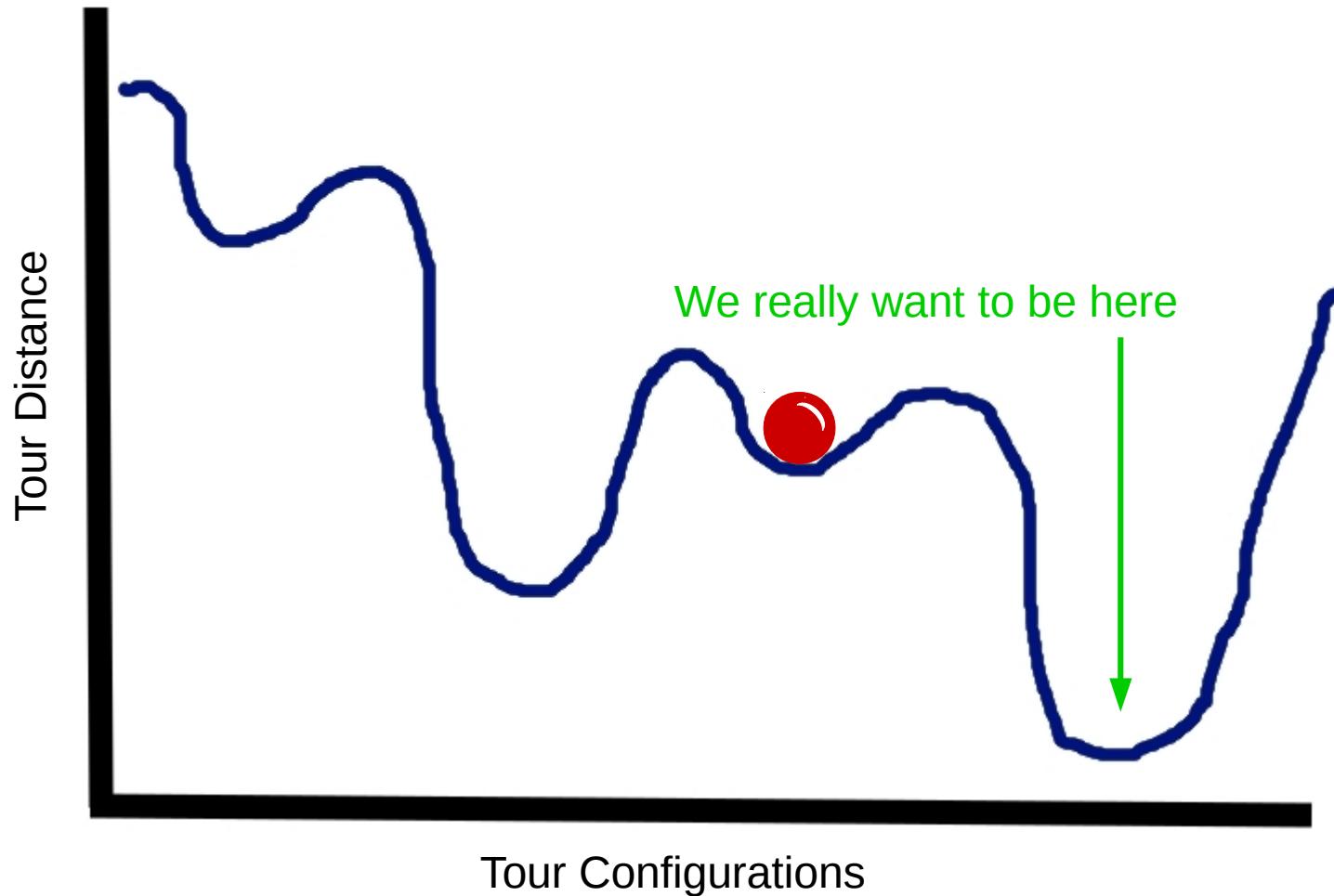


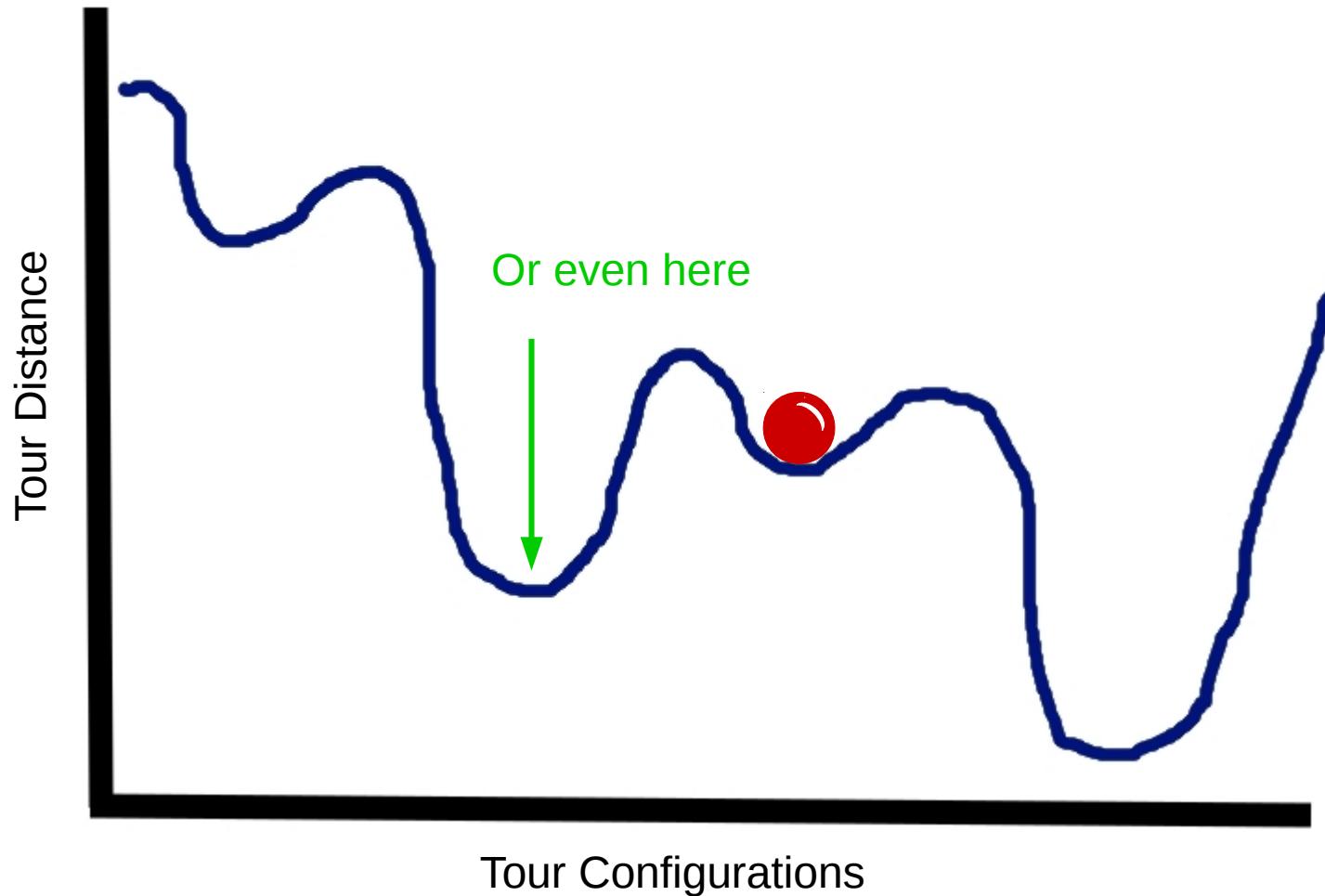


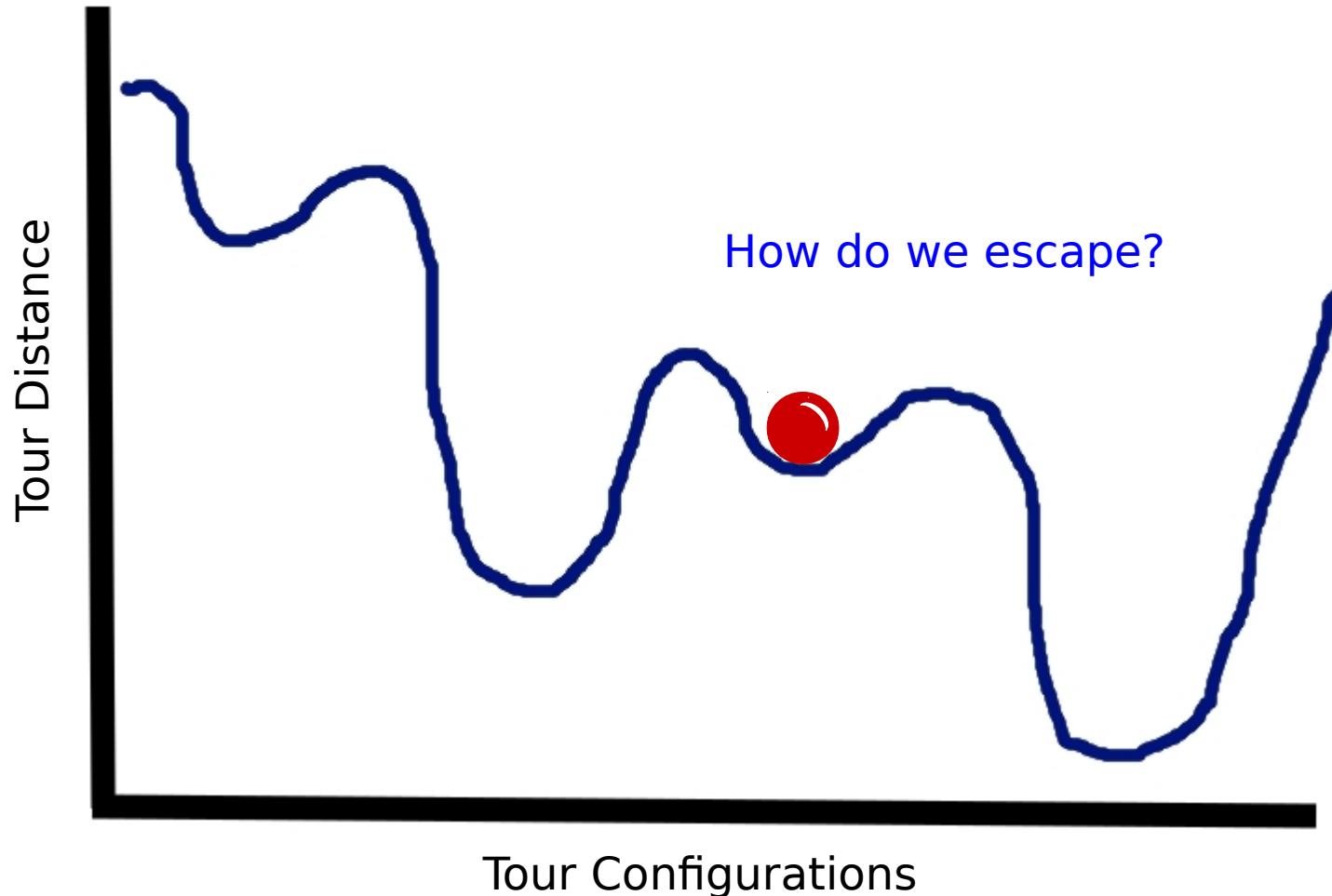


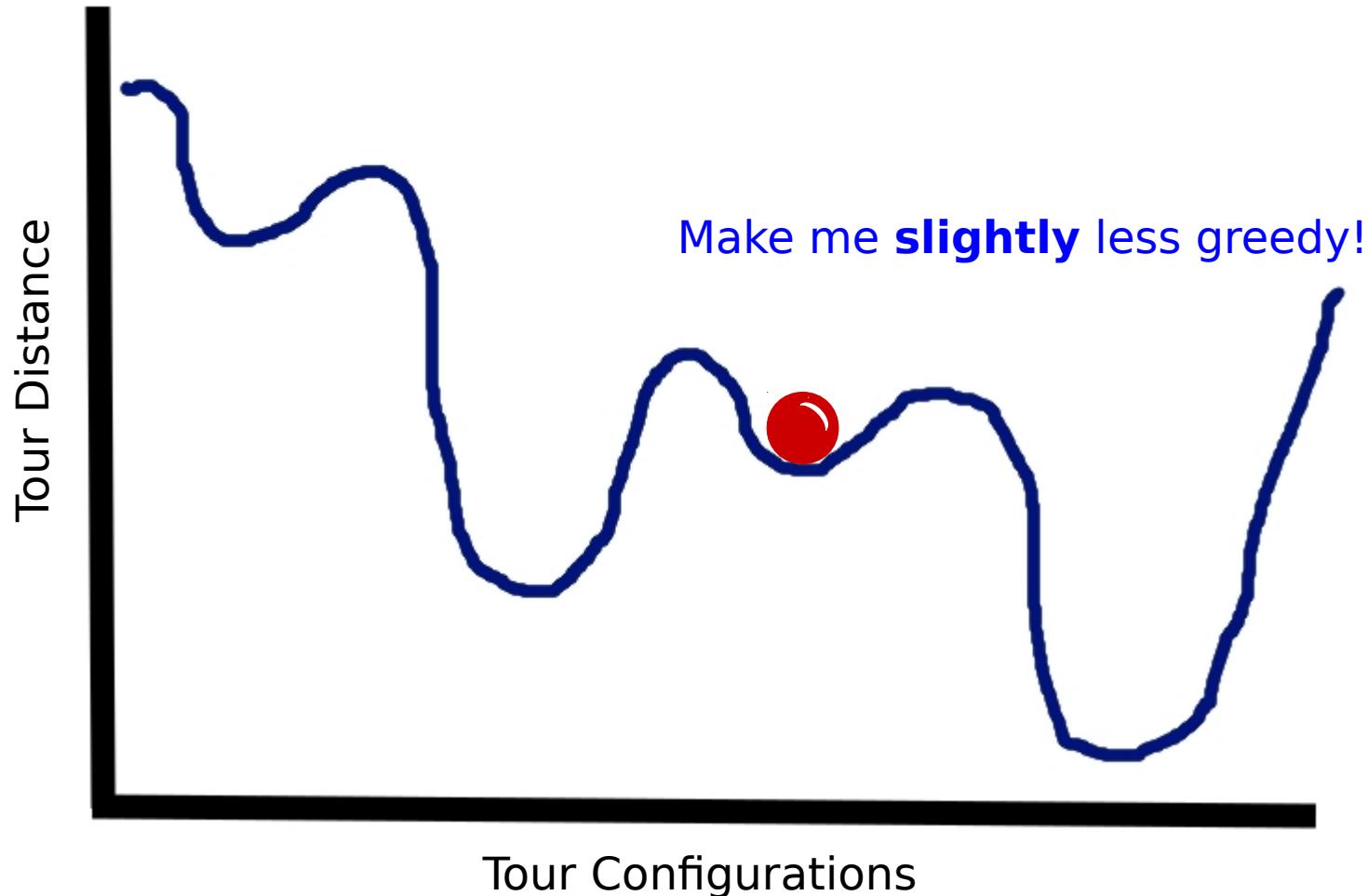


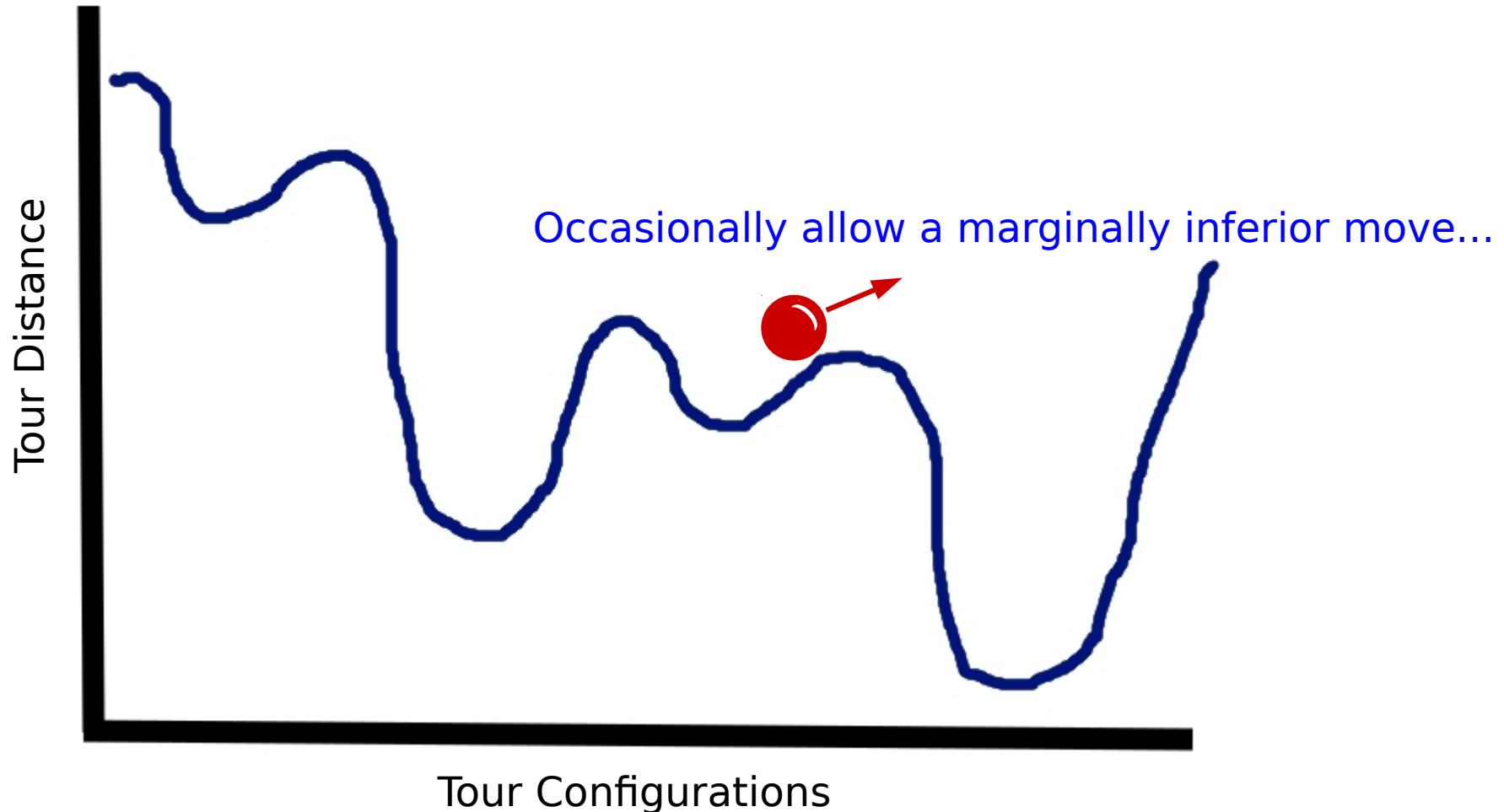


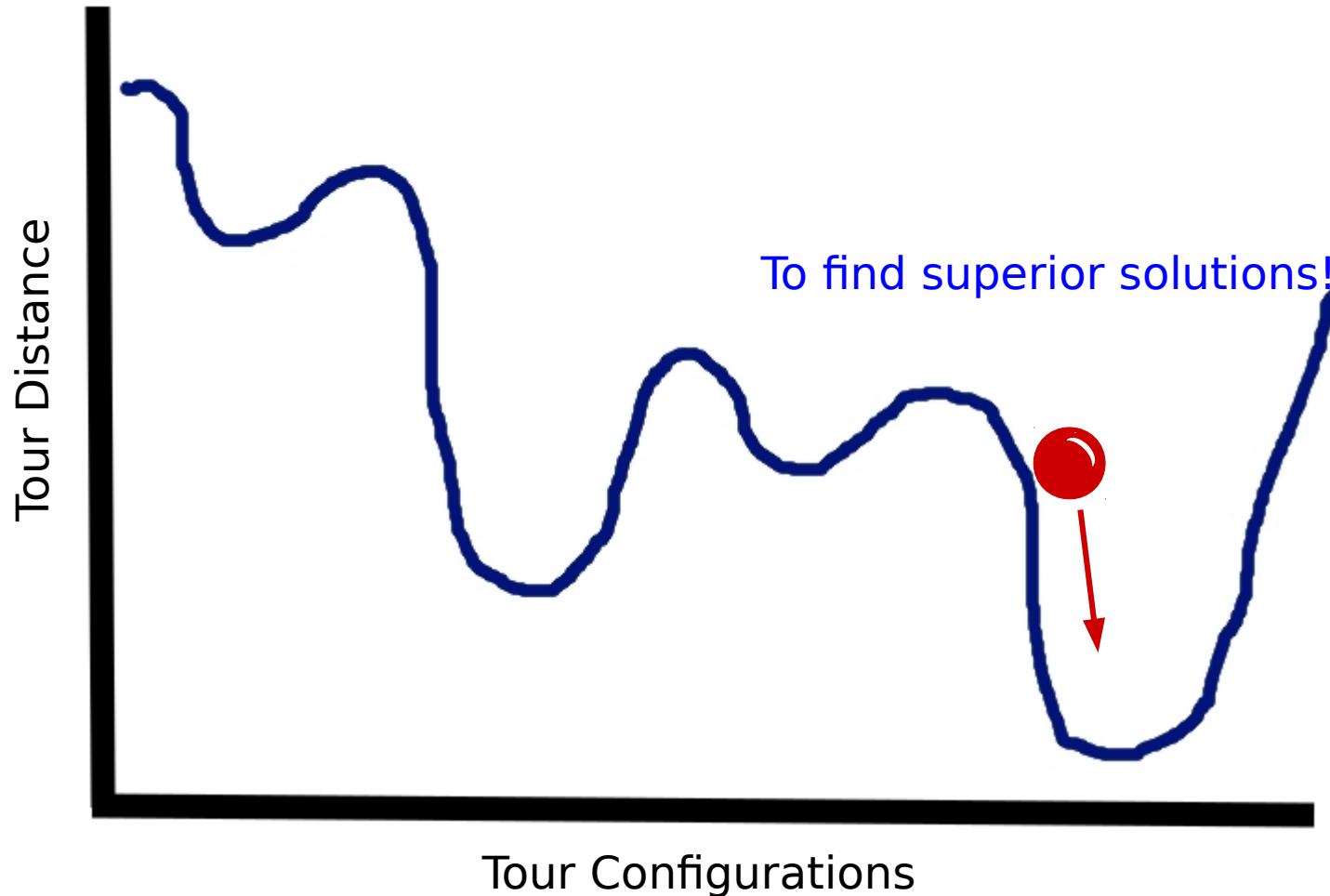


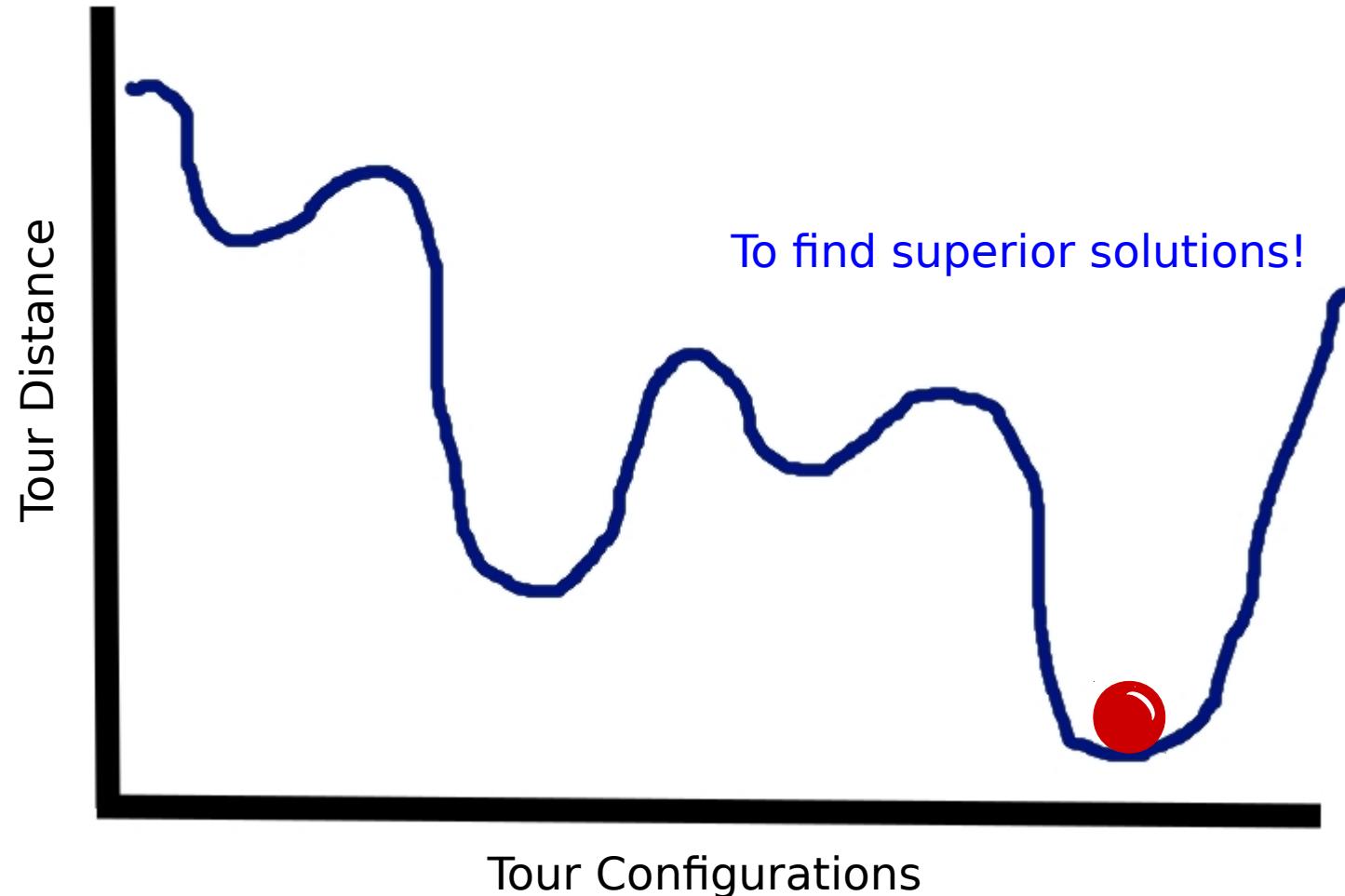












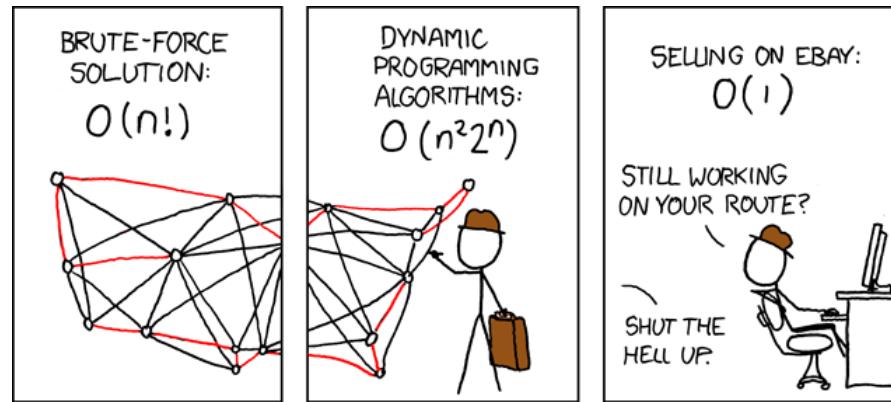
# Source Code

## Traveling Salesman Demo

[https://github.com/thomasnield/traveling\\_salesman\\_demo](https://github.com/thomasnield/traveling_salesman_demo)

## Traveling Salesman Plotter

[https://github.com/thomasnield/traveling\\_salesman\\_plotter](https://github.com/thomasnield/traveling_salesman_plotter)



SOURCE: xkcd.com

# Generating a Schedule

**You need to generate a schedule for a single classroom with the following classes:**

Psych 101 (1 hour, 2 sessions/week)

English 101 (1.5 hours, 2 sessions/week)

Math 300 (1.5 hours, 2 sessions/week)

Psych 300 (3 hours, 1 session/week)

Calculus I (2 hours, 2 sessions/week)

Linear Algebra I (2 hours, 3 sessions/week)

Sociology 101 (1 hour, 2 sessions/week)

Biology 101 (1 hour, 2 sessions/week)

Supply Chain 300 (2.5 hours, 2 sessions/week)

Orientation 101 (1 hour, 1 session/week)

Available scheduling times are **Monday through Friday, 8:00AM-11:30AM, 1:00PM-5:00PM**

Slots are scheduled in **15 minute increments**.

# Generating a Schedule

**Visualize a grid of each 15-minute increment from Monday through Sunday, intersected with each possible class.**

**Each cell will be a 1 or 0 indicating whether that's the start of the first class.**

	MON	MON	MON	MON	MON	MON	MON	MON	MON	SUN
	12:00 AM	12:15 AM	12:30 AM	12:45 AM	1:00 AM	1:15 AM	1:30 AM	1:45 AM	...	11:55 PM
Psych 101	0	0	0	0	0	0	0	0	...	0
English 101	0	0	0	0	0	0	0	0	...	0
Math 300	0	0	0	0	0	0	0	0	...	0
Psych 300	0	0	0	0	0	0	0	0	...	0
Calculus I	0	0	0	0	0	0	0	0	...	0
Linear Algebra I	0	0	0	0	0	0	0	0	...	0
Sociology 101	0	0	0	0	0	0	0	0	...	0
Biology 101	0	0	0	0	0	0	0	0	...	0
Supply Chain 300	0	0	0	0	0	0	0	0	...	0
Orientation 101	0	0	0	0	0	0	0	0	...	0

# Generating a Schedule

Next visualize how overlaps will occur.

Notice how a 9:00AM Psych 101 class will clash with a 9:15AM Sociology 101.

We can sum all blocks that affect the 9:45AM block and ensure they don't exceed 1.

		MON	MON	MON	MON	MON	MON	MON	MON	MON	SUN
	...	9:00 AM	9:15 AM	9:30 AM	9:45 AM	10:00 AM	10:15 AM	10:30 AM	10:45 AM	...	11:55 PM
Psych 101	...	1	0	0	0	0	0	0	0	...	0
English 101	...	0	0	0	0	0	0	0	0	...	0
Math 300	...	0	0	0	0	0	0	0	0	...	0
Psych 300	...	0	0	0	0	0	0	0	0	...	0
Calculus I	...	0	0	0	0	0	0	0	0	...	0
Linear Algebra I	...	0	0	0	0	0	0	0	0	...	0
Sociology 101	...	0	1	0	0	0	0	0	0	...	0
Biology 101	...	0	0	0	0	0	0	0	0	...	0
Supply Chain 300	...	0	0	0	0	0	0	0	0	...	0
Orientation 101	...	0	0	0	0	0	0	0	0	...	0

Sum of affecting slots = 2  
FAIL, sum must be  $\leq 1$

# Generating a Schedule

Next visualize how overlaps will occur.

Notice how a 9:00AM Psych 101 class will clash with a 9:30AM Sociology 101.

We can sum all blocks that affect the 9:45AM block and ensure they don't exceed 1.

		MON	MON	MON	MON	MON	MON	MON	MON	MON	SUN
	...	9:00 AM	9:15 AM	9:30 AM	9:45 AM	10:00 AM	10:15 AM	10:30 AM	10:45 AM	...	11:55 PM
Psych 101	...	1	0	0	0	0	0	0	0	...	0
English 101	...	0	0	0	0	0	0	0	0	...	0
Math 300	...	0	0	0	0	0	0	0	0	...	0
Psych 300	...	0	0	0	0	0	0	0	0	...	0
Calculus I	...	0	0	0	0	0	0	0	0	...	0
Linear Algebra I	...	0	0	0	0	0	0	0	0	...	0
Sociology 101	...	0	0	1	0	0	0	0	0	...	0
Biology 101	...	0	0	0	0	0	0	0	0	...	0
Supply Chain 300	...	0	0	0	0	0	0	0	0	...	0
Orientation 101	...	0	0	0	0	0	0	0	0	...	0

Sum of affecting slots = 2  
FAIL, sum must be  $\leq 1$

# Generating a Schedule

Next visualize how overlaps will occur.

Notice how a 9:00AM Psych 101 class will clash with a 9:45AM Sociology 101.

We can sum all blocks that affect the 9:45AM block and ensure they don't exceed 1.

		MON	MON	MON	MON	MON	MON	MON	MON	MON		SUN
	...	9:00 AM	9:15 AM	9:30 AM	9:45 AM	10:00 AM	10:15 AM	10:30 AM	10:45 AM	...		11:55 PM
Psych 101	...	1	0	0	0	0	0	0	0	...		0
English 101	...	0	0	0	0	0	0	0	0	...		0
Math 300	...	0	0	0	0	0	0	0	0	...		0
Psych 300	...	0	0	0	0	0	0	0	0	...		0
Calculus I	...	0	0	0	0	0	0	0	0	...		0
Linear Algebra I	...	0	0	0	0	0	0	0	0	...		0
Sociology 101	...	0	0	0	1	0	0	0	0	...		0
Biology 101	...	0	0	0	0	0	0	0	0	...		0
Supply Chain 300	...	0	0	0	0	0	0	0	0	...		0
Orientation 101	...	0	0	0	0	0	0	0	0	...		0

Sum of affecting slots = 2  
FAIL, sum must be  $\leq 1$

# Generating a Schedule

If the “sum” of all slots affecting a given block are no more than 1, then we have no conflicts!

		MON	MON	MON	MON	MON	MON	MON	MON	MON	MON	SUN
	...	9:00 AM	9:15 AM	9:30 AM	9:45 AM	10:00 AM	10:15 AM	10:30 AM	10:45 AM	...	11:55 PM	
Psych 101	...	1	0	0	0	0	0	0	0	...	0	
English 101	...	0	0	0	0	0	0	0	0	...	0	
Math 300	...	0	0	0	0	0	0	0	0	...	0	
Psych 300	...	0	0	0	0	0	0	0	0	...	0	
Calculus I	...	0	0	0	0	0	0	0	0	...	0	
Linear Algebra I	...	0	0	0	0	0	0	0	0	...	0	
Sociology 101	...	0	0	0	0	1	0	0	0	...	0	
Biology 101	...	0	0	0	0	0	0	0	0	...	0	
Supply Chain 300	...	0	0	0	0	0	0	0	0	...	0	
Orientation 101	...	0	0	0	0	0	0	0	0	...	0	

Sum of affecting slots = 1  
SUCCESS!

# Generating a Schedule

**For every “block”, we must sum all affecting slots (shaded below) which can be identified from the class durations.**

**This sum must be no more than 1.**

		MON	MON	MON	SUN												
	...	7:00 AM	7:15 AM	7:30 AM	7:45 AM	8:00 AM	8:15 AM	8:30 AM	8:45 AM	9:00 AM	9:15 AM	9:30 AM	9:45 AM	10:00 AM	...	11:55 PM	
Psych 101	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
English 101	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Math 300	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Psych 300	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Calculus I	...	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Linear Algebra I	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sociology 101	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Biology 101	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Supply Chain 300	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Orientation 101	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

# Generating a Schedule

**Taking this concept even further, we can account for all recurrences.**

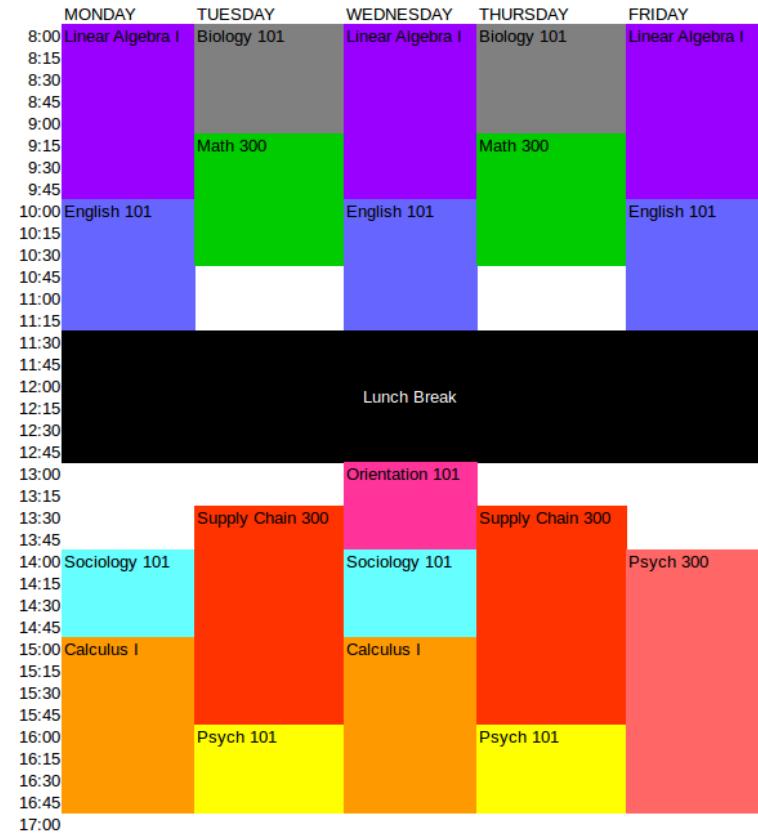
**The “affected slots” for a given block can query for all recurrences for each given class.**

[View image here.](#)

# Generating a Schedule

Plug these variables and *feasible* constraints into the optimizer, and you will get a solution.

Most of the work will be finding the affecting slots for each block.



# Generating a Schedule

If you want to schedule against multiple rooms, plot each variable using three dimensions.

	MON 8:00	MON 8:15	MON 8:30	MON 8:45	MON 9:00	MON 9:15	MON 9:30	MON 9:45
PSYCH 300					1	0	0	0
MATH 300								
PSYCH 101	1	0	0	0				
ROOM 1					1	0	0	0
ROOM 2	1	0	0	0				
ROOM 3								

# Source Code

## Classroom Scheduling Optimizer

<https://github.com/thomasnield/optimized-scheduling-demo>

# Solving a Sudoku

Imagine you are presented a Sudoku.

Rather than do an exhaustive brute-force search, think in terms of constraint programming to reduce the search space.

First, sort the cells by the count of possible values they have left:

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	

# Solving a Sudoku

[4,4] → 5  
[2,6] → 7  
[7,7] → 3  
[8,6] → 4  
[1,4] → 2, 5  
[0,7] → 2, 3  
[3,2] → 2, 3  
[4,2] → 3, 4  
[5,2] → 2, 4  
[3,5] → 5, 9  
[5,5] → 1, 4  
[4,6] → 3, 5  
[5,8] → 2, 6  
[6,7] → 3, 6  
[0,2] → 1, 2, 3  
[1,3] → 1, 2, 5  
...  
[2,6] → 1,3,4,5,7,9

Put cells in a list  
sorted by possible  
candidate count

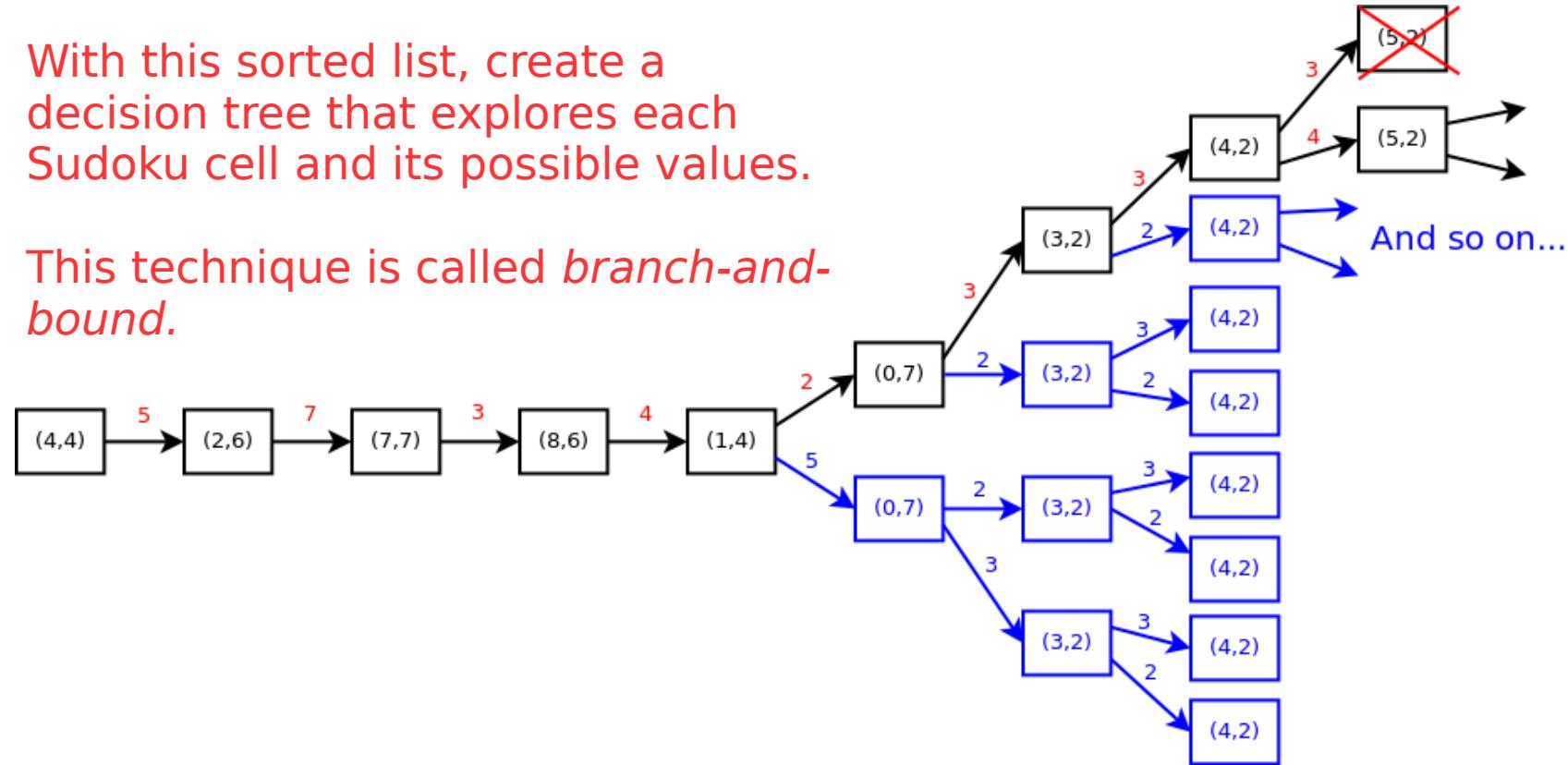
0	1	2	3	4	5	6	7	8
0	5	3		7				
1	6		1	9	5			
2		9	8				6	
3	8			6				3
4	4		8	3				1
5	7			2			6	
6		6				2	8	
7			4	1	9			5
8				8		7	9	

# Solving a Sudoku

[4,4] → 5  
[2,6] → 7  
[7,7] → 3  
[8,6] → 4  
[1,4] → 2, 5  
[0,7] → 2, 3  
[3,2] → 2, 3  
[4,2] → 3, 4  
[5,2] → 2, 4  
[3,5] → 5, 9  
[5,5] → 1, 4  
[4,6] → 3, 5  
[5,8] → 2, 6  
[6,7] → 3, 6  
[0,2] → 1, 2, 3  
[1,3] → 1, 2, 5  
...  
[2,6] → 1,3,4,5,7,9

With this sorted list, create a decision tree that explores each Sudoku cell and its possible values.

This technique is called *branch-and-bound*.



# Solving a Sudoku

A branch should terminate immediately when it finds an infeasible configuration, and then explore the next branch.

After we have a branch that provides a feasible value to every cell, we have solved our Sudoku!

Unlike many optimization problems, Sudokus are trivial to solve because they constrain their search spaces quickly.

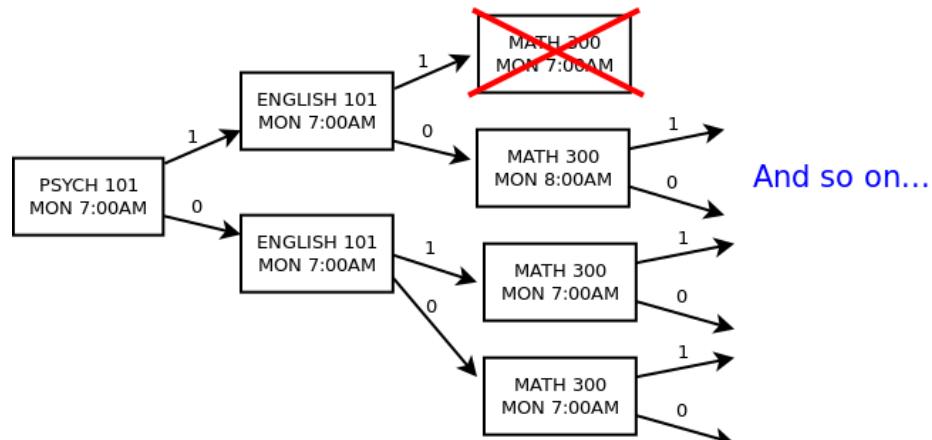
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

# Branch-and-Bound for Scheduling

You could solve the scheduling problem from scratch with branch-and-bound.

Start with the most “constrained” slots first to narrow your search space (e.g. slots fixed to zero first, followed by Monday slots for 3-recurrence classes).

HINT: Proactively prune the tree as you go, eliminating any slots ahead that must be zero due to a “1” decision propagating an occupied state.



# Source Code

## Kotlin Sudoku Solver

<https://github.com/thomasnield/kotlin-sudoku-solver>

# Discrete Optimization Summary

**Discrete Optimization is a best-kept secret well-known in operations research.**

Machine learning itself is an optimization problem, finding the right values for variables to minimize an error function.

Many folks misguidedly turn to neural networks and other machine learning when discrete optimization would be more appropriate.

## **Recommended Java Libraries:**

OjAlgo!

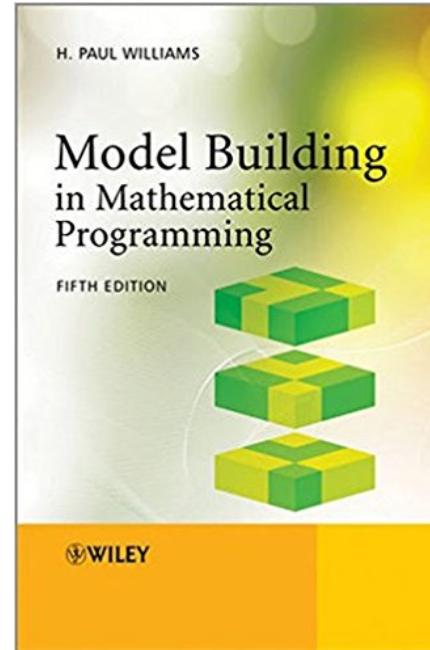
OptaPlanner

# Learn More About Discrete Optimization



*Discrete Optimization*

Sport Scheduling



## **Part III: Classification w/ Naive Bayes**



# Classifying Things

**Probably the most common task in machine learning is classifying data:**

How do I identify images of **dogs** vs **cats**?

What **words** are being said in a piece of audio?

Is this email **spam** or **not spam**?

What attributes define **high-risk**, **medium-risk**, and **low-risk** loan applicants?

How do I predict if a shipment will be **late**, **early**, or **on-time**?

**There are many techniques to classify data, with pros/cons depending on the task:**

Neural Networks

Support Vector Machines

Decision Trees/Random Forests

Naive Bayes

Linear/Non-linear regression

# Naive Bayes

Let's focus on Naive Bayes because it is simple to implement and effective for a common task: *text categorization*.

Naive Bayes is an adaptation of Bayes Theorem that can predict a category **C** for an item **T** with *multiple* features **F**.

A common usage example of Naive Bayes is email spam, where each word is a feature and **spam/not spam** are the possible categories.

# Implementing Naive Bayes

Naive Bayes works by mapping probabilities of each individual feature occurring/not occurring for a given category (e.g. a word occurring in **spam/not spam**).

**A category can be predicted for a new set of features by...**

- 1) For a given category, combine the probabilities of each feature **occurring** and **not occurring** by multiplying them.

$$\text{Occur Product} = P_{f1} * P_{f2} * \dots * P_{fn}$$

$$\text{Not Occur Product} = !P_{f1} * !P_{f2} * \dots * !P_{fn}$$

- 2) Divide the products to get the probability for that category.

$$\text{Combined Probability} = \frac{(\text{Occur Product})}{(\text{Occur Product}) + (\text{Not Occur Product})}$$

# Implementing Naive Bayes

3) Calculate this for every category, and select the one with highest probability.

## Dealing with floating point underflow.

A big problem is multiplying small decimals for a large number of features may cause a floating point underflow.

To remedy this, transform each probability with **log()** or **ln()** and sum them, then call **exp()** to convert the result back!

$$\text{Occur Product} = \exp(\ln(P_{f1}) + \ln(P_{f2}) + \dots + \ln(P_{fn}))$$

$$\text{Not Occur Product} = \exp(\ln(!P_{f1}) + \ln(!P_{f2}) + \dots + \ln(!P_{fn}))$$

# Implementing Naive Bayes

**One last consideration, never let a feature have a 0 probability for any category!**

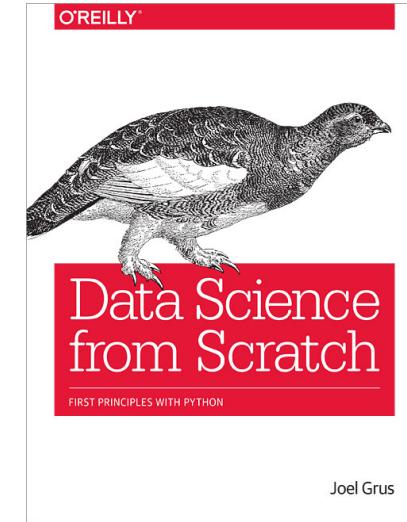
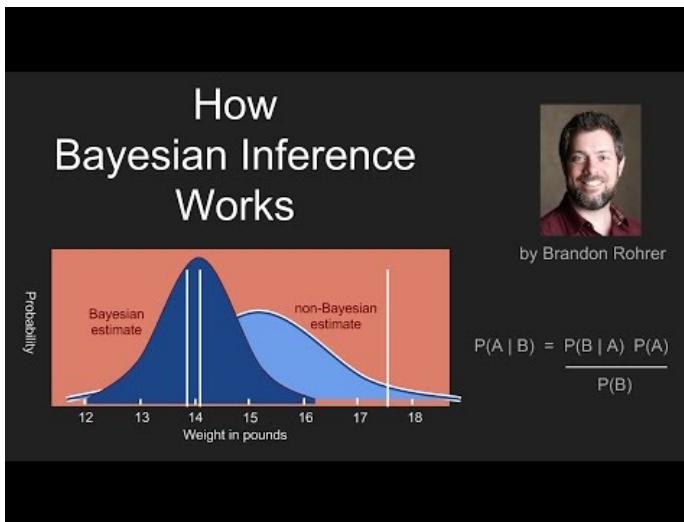
Always leave *a little* possibility it could belong to any category so you don't have *0* multiplication or division mess anything up.

This can be done by adding a small value to each probability's numerator and denominator (e.g. *0.5* and *1.0*).

$$\text{CombinedProbability} = \frac{0.5 + (\text{Occur Product})}{1.0 + (\text{Occur Product}) + (\text{Not Occur Product})}$$

# Learn More About Bayes

Brandon Rohrer - YouTube



# Source Code

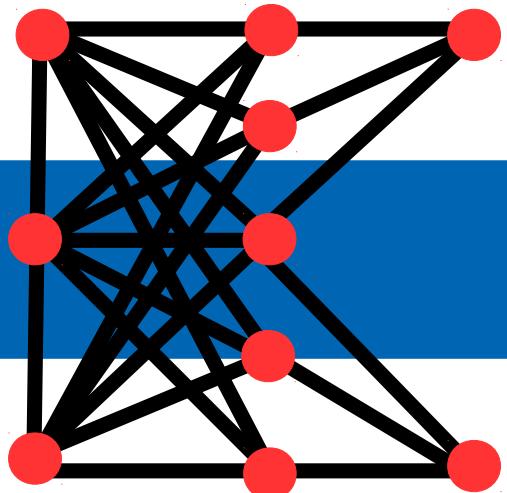
## Bank Transaction Categorizer Demo

[https://github.com/thomasnield/bayes\\_user\\_input\\_prediction](https://github.com/thomasnield/bayes_user_input_prediction)

## Email Spam Classifier Demo

[https://github.com/thomasnield/bayes\\_email\\_spam](https://github.com/thomasnield/bayes_email_spam)

## Part V: Neural Networks

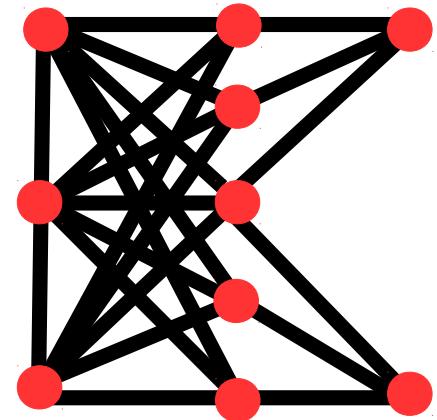


# What Are Neural Networks?

**Neural Networks are a machine learning tool that takes numeric inputs and predicts numeric outputs.**

A series of multiplication, addition, and nonlinear functions are applied to the numeric inputs.

The mathematical operations above are iteratively tweaked until the desired output is met.



# The Problem

**Suppose we wanted to take a background color (in RGB values) and predict a light/dark font for it.**

Hello

Hello

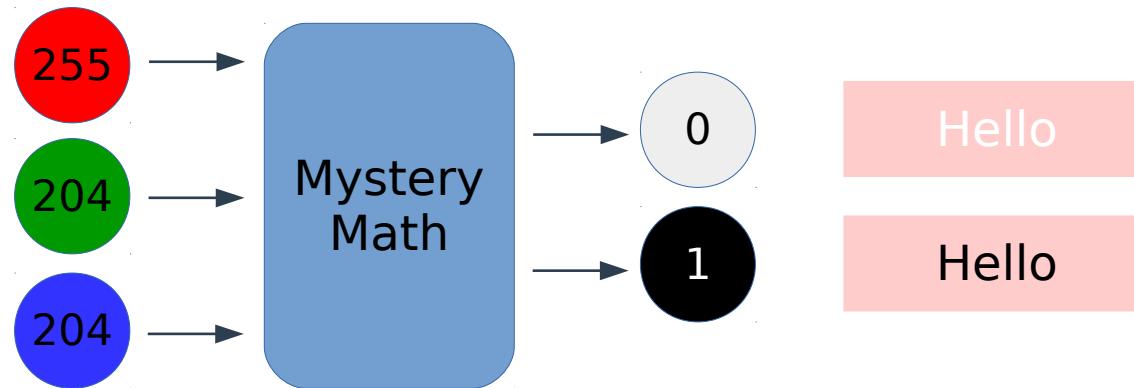
If you search around Stack Overflow, there is a nice formula to do this:

$$L = (.299R + .587G + .114B)/255$$

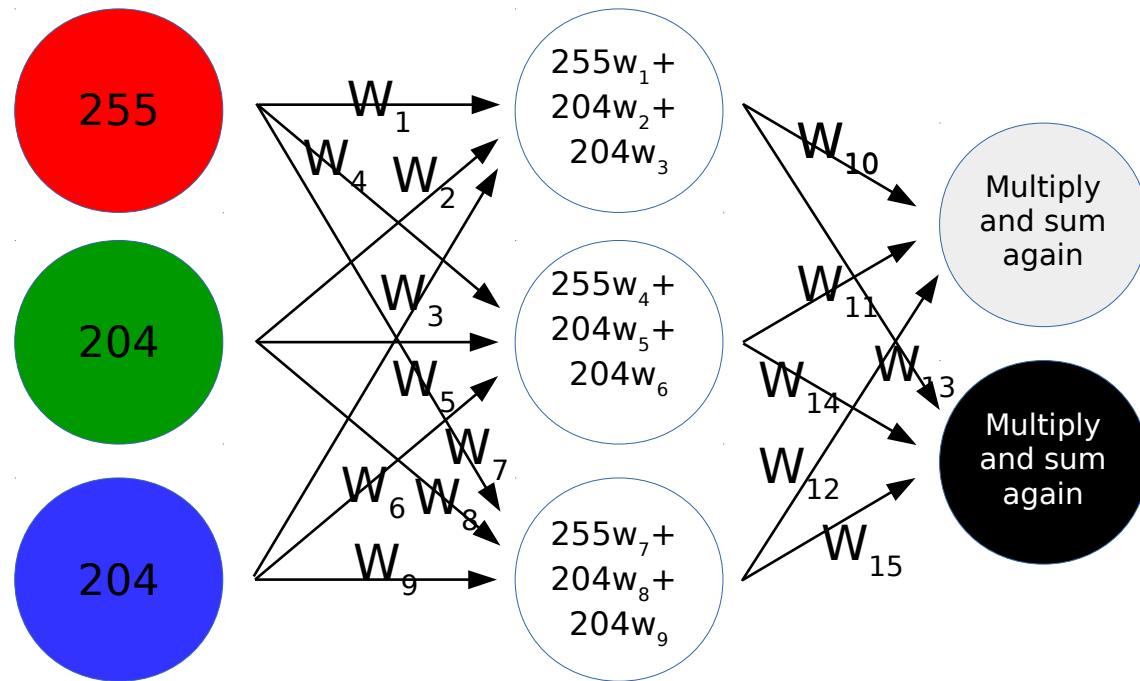
But what if we do not know the formula? Or one hasn't been discovered?

# A Simple Neural Network

**Let's represent background color as 3 numeric RGB inputs, and predict whether a DARK/LIGHT font should be used.**



# A Simple Neural Network

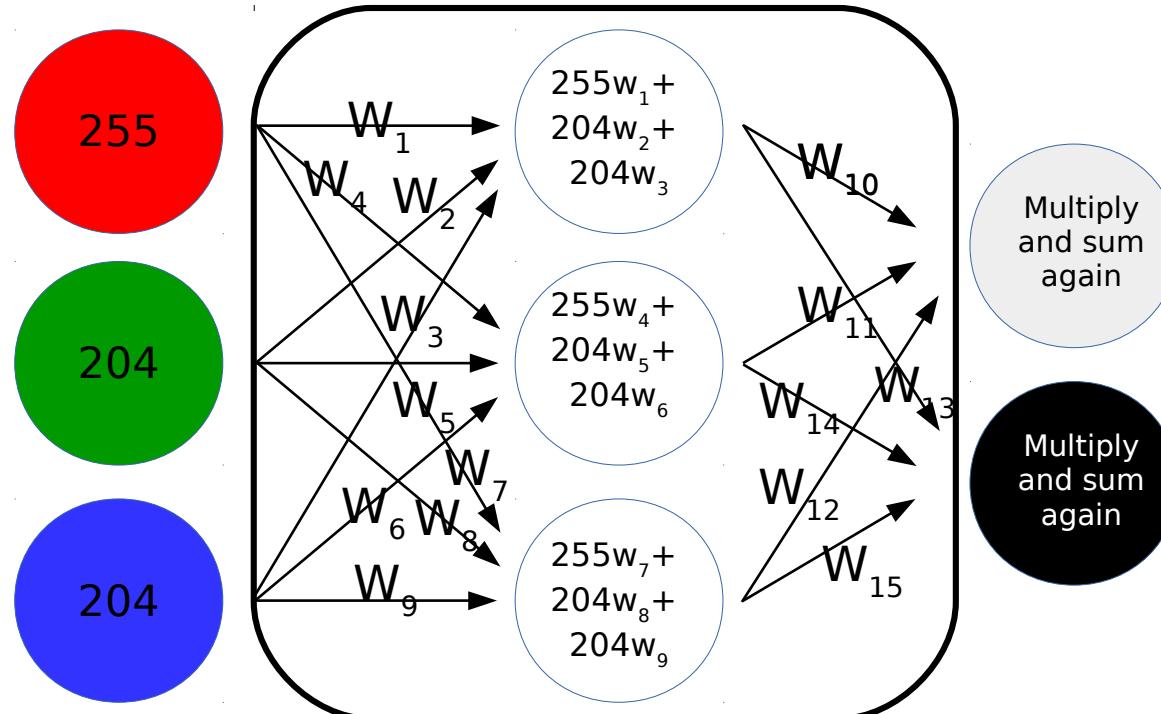


Hello

Hello

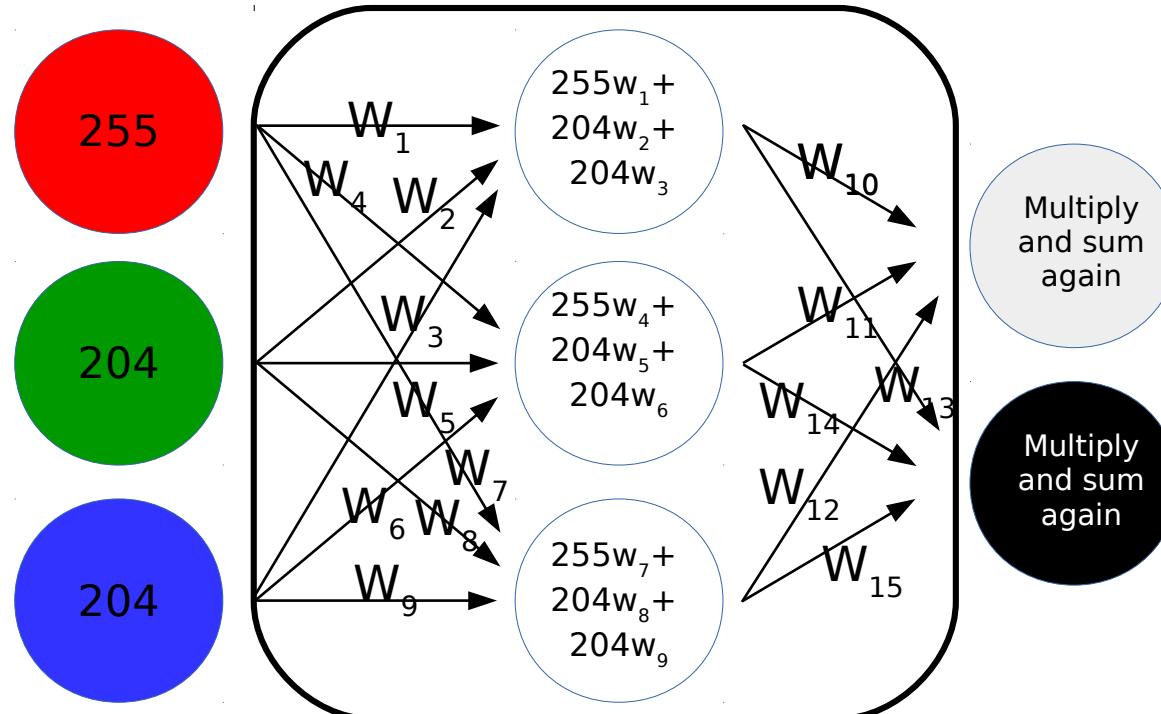
# A Simple Neural Network

This is the “mystery math”



# A Simple Neural Network

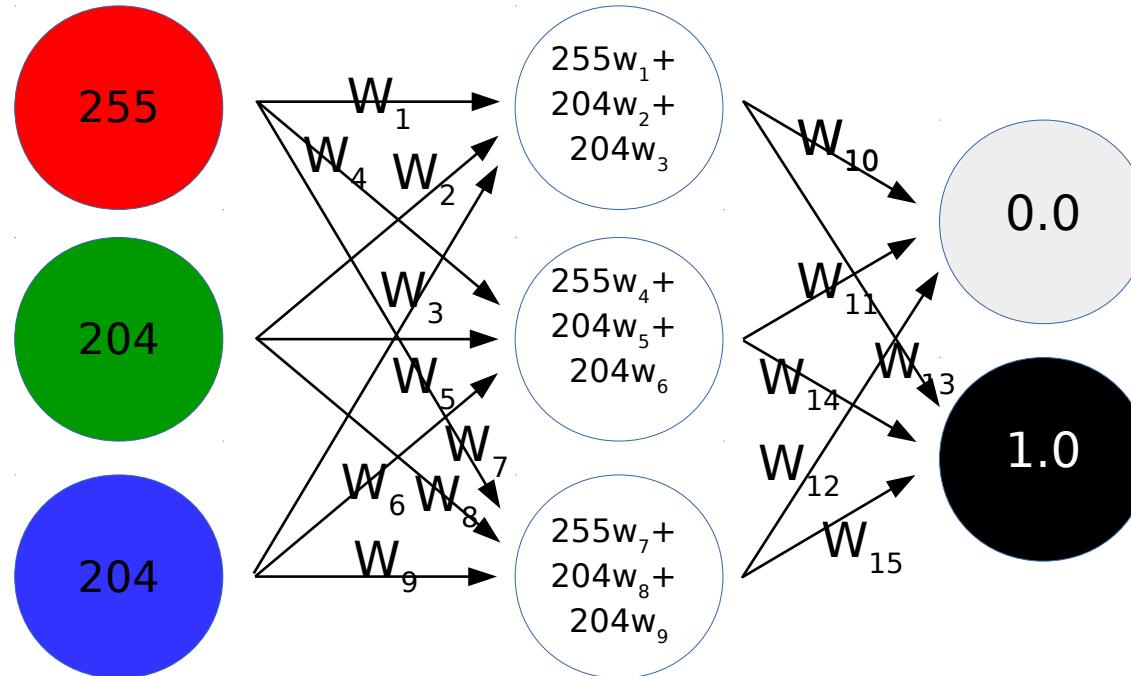
Each weight  $w_x$  value is between -1.0 and 1.0



# A Simple Neural Network

## Million Dollar Question:

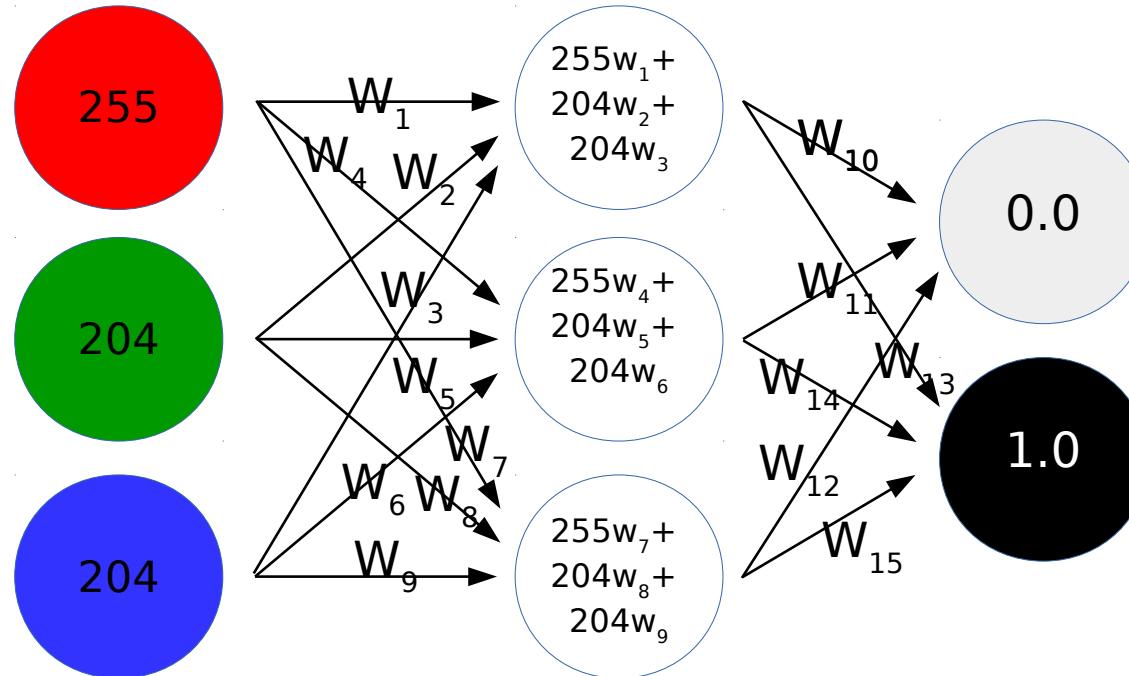
What are the *optimal* weight values to get the desired output?



# A Simple Neural Network

## Million Dollar Question:

What are the *optimal* weight values to get the desired output?



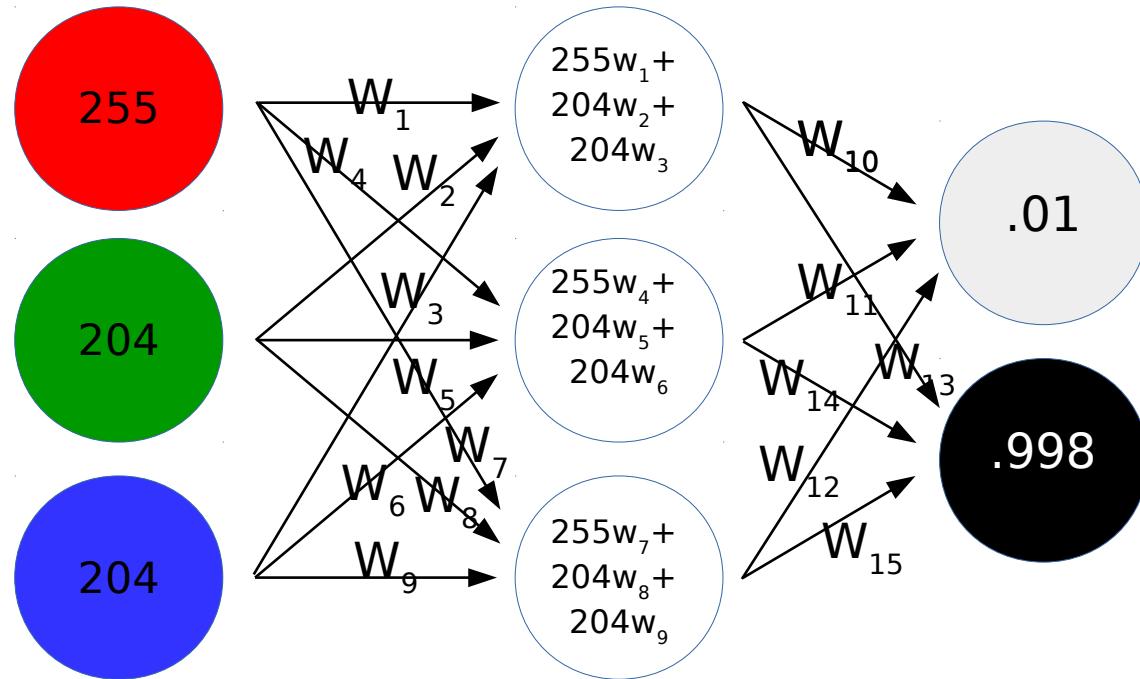
Hello

Hello

# A Simple Neural Network

**Answer:**

This is an optimization problem!

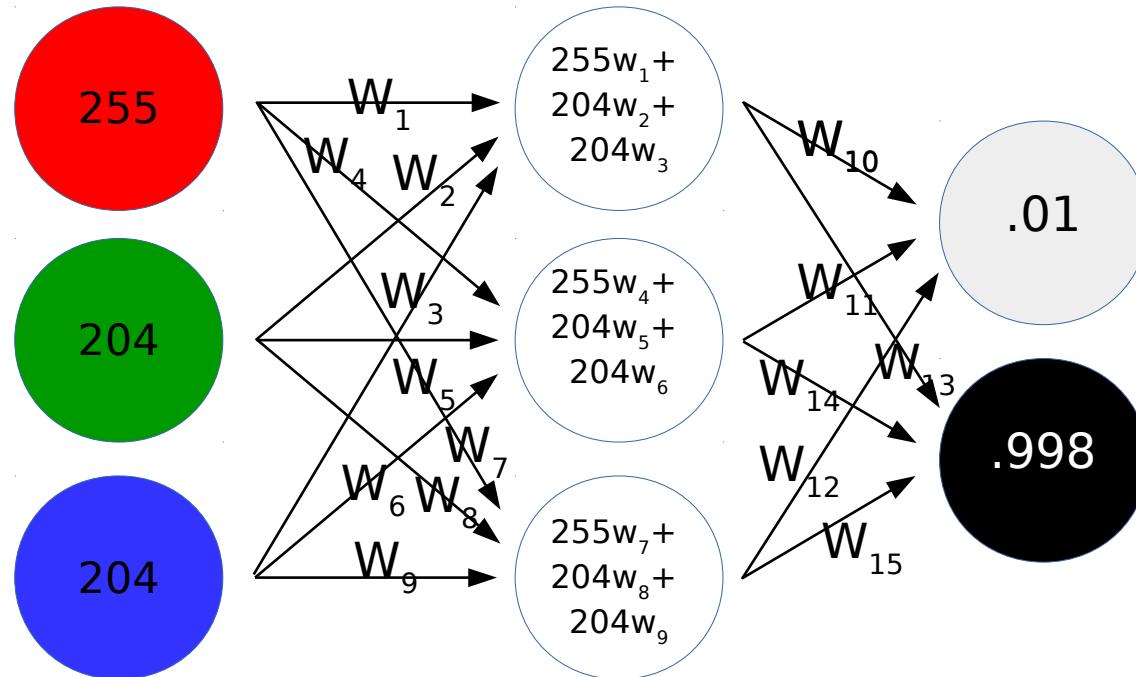


Hello

Hello

# A Simple Neural Network

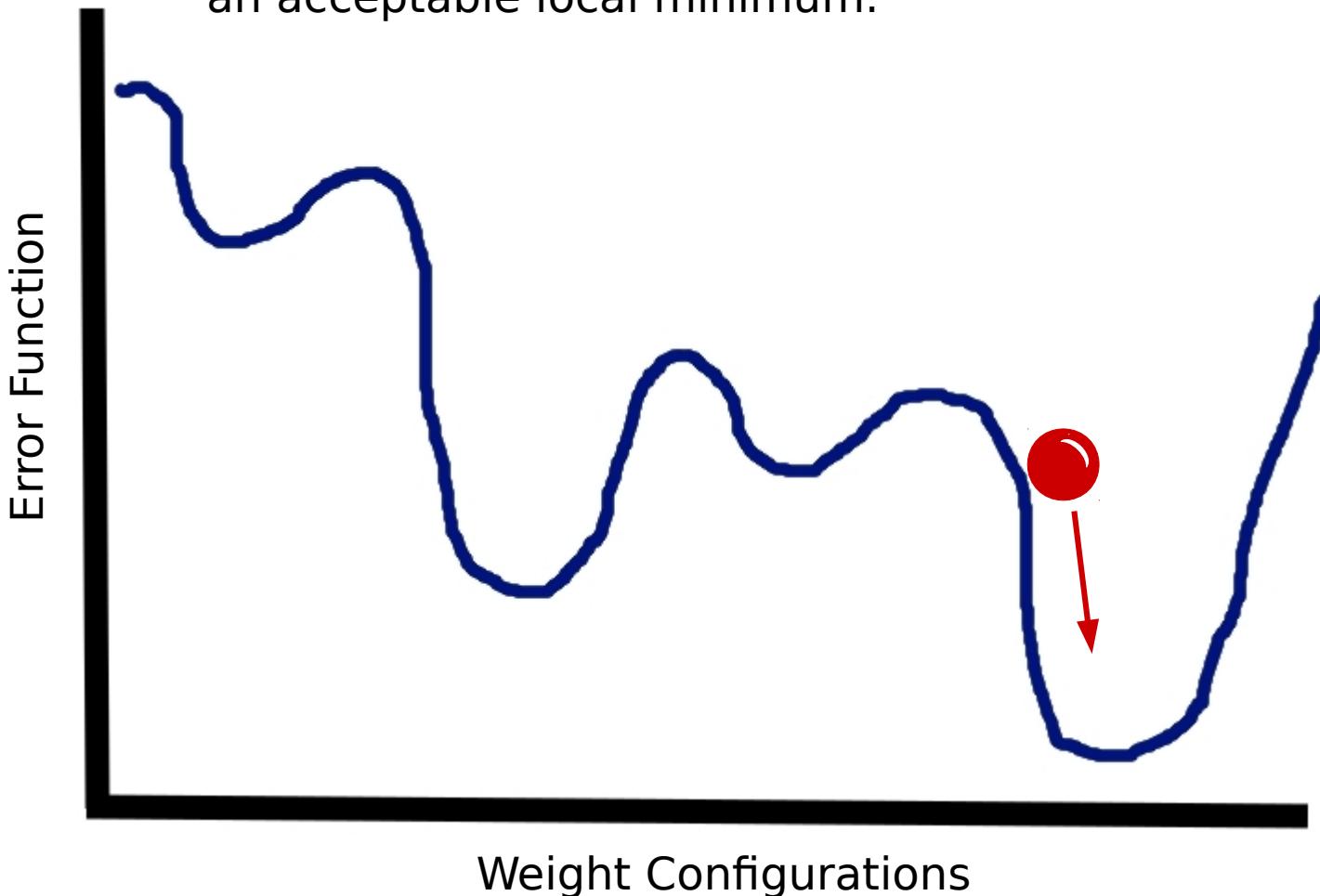
We need to solve for the weight values that gets our training colors as close to their desired outputs as possible.



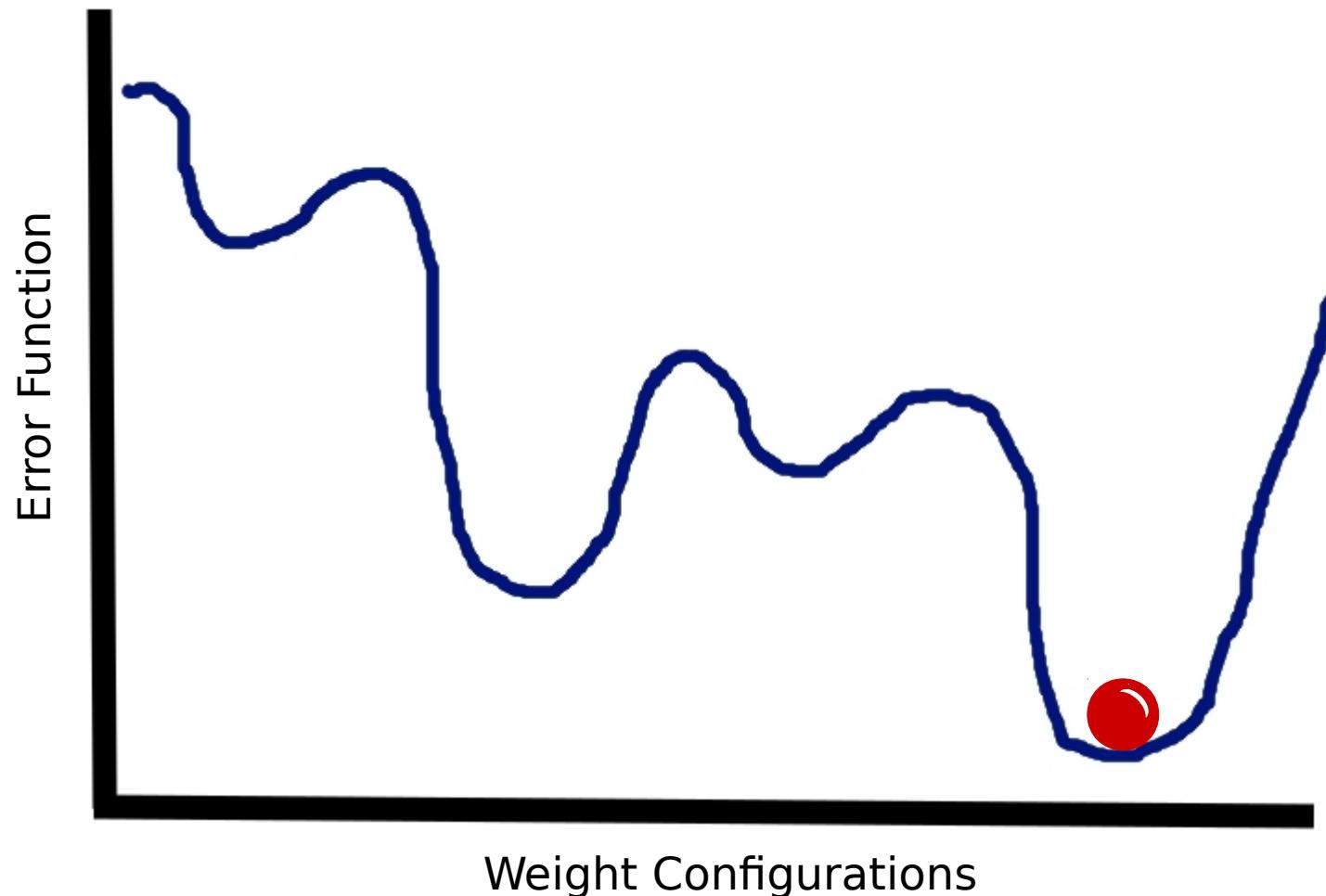
Hello

Hello

Just like the Traveling Salesman Problem,  
you need to explore configurations seeking  
an acceptable local minimum.



*Gradient descent, simulated annealing, and other optimization techniques can help tune a neural network.*

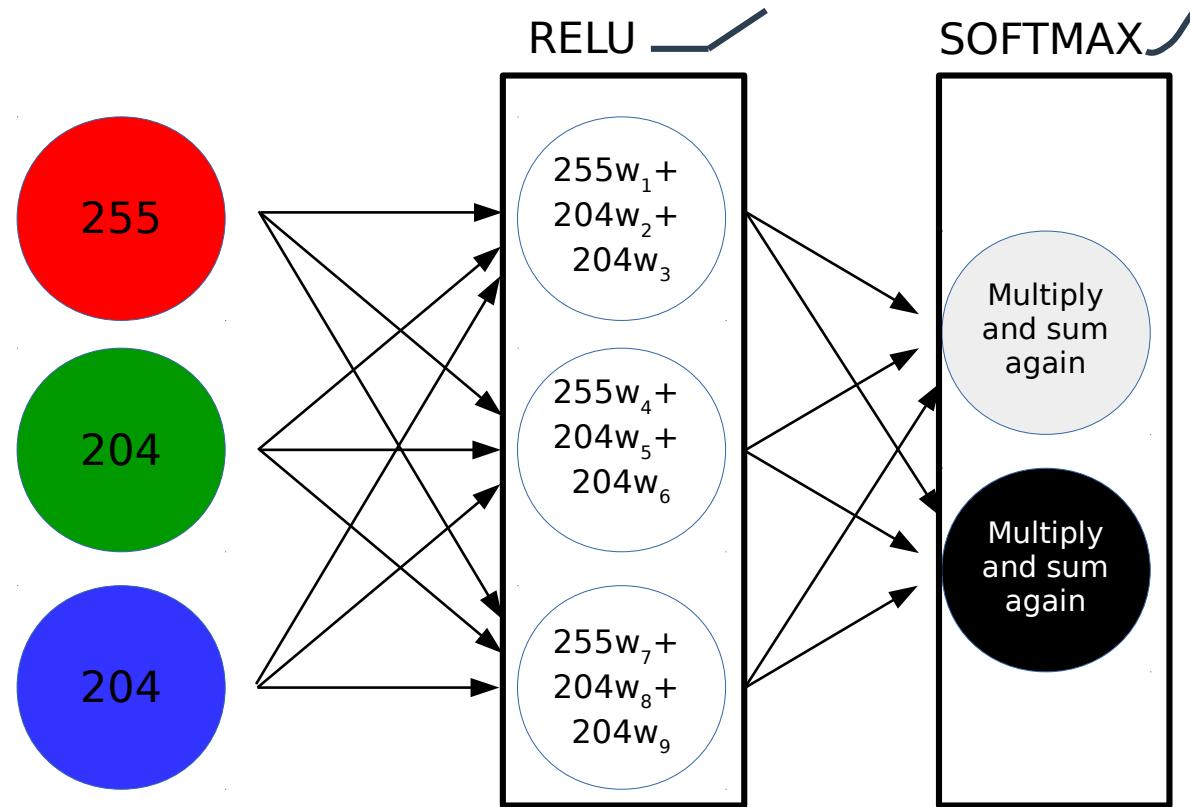


# Activation Functions

You might also consider using **activation functions** on each layer.

These are nonlinear functions that smooth, scale, or compress the resulting sum values.

These make the network operate more naturally and smoothly.



# Activation Functions

Four common neural network activation functions implemented using kotlin-stdlib

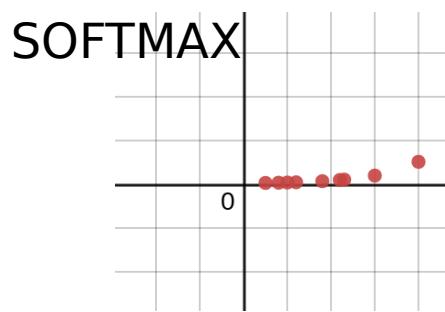
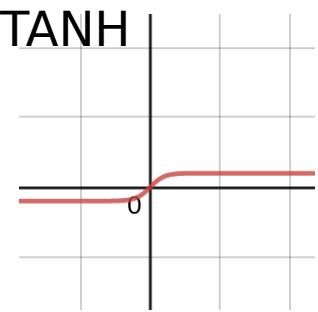
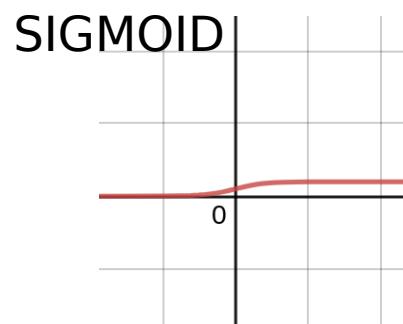
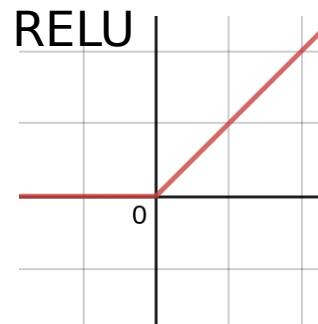
```
import kotlin.math.exp
import kotlin.math.max

fun sigmoid(x: Double) = 1.0 / (1.0 + exp(-x))

fun relu(x: Double) = max(0.0, x)

fun softmax(x: Double, allValues: DoubleArray) =
    exp(x) / allValues.map { exp(it) }.sum()

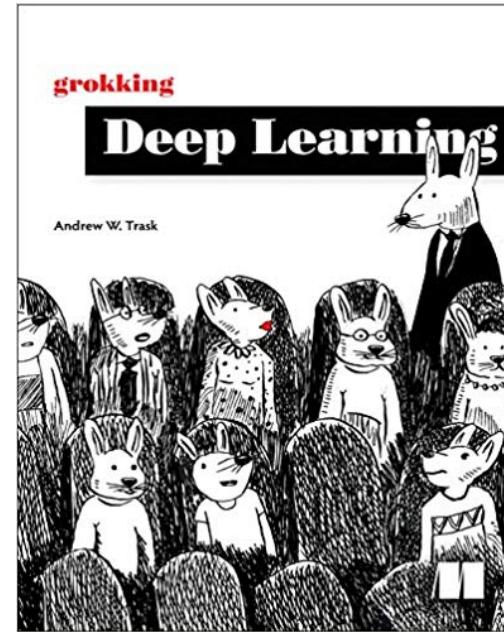
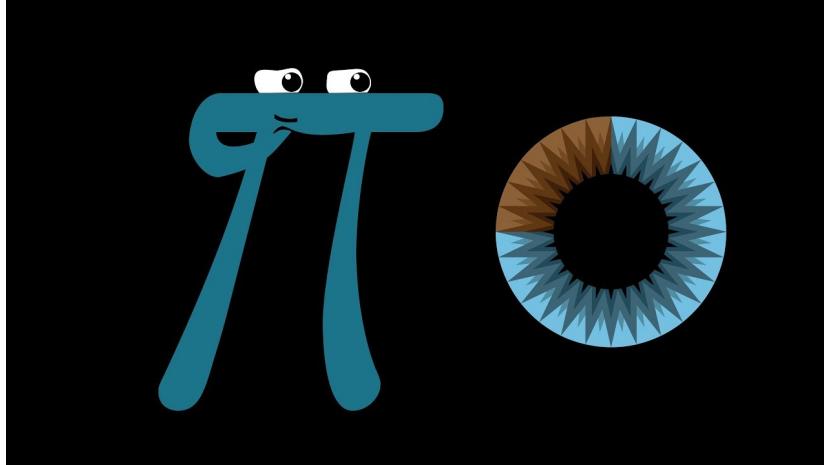
fun tanh(x: Double) = kotlin.math.tanh(x)
```



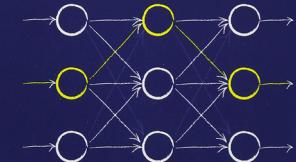
<https://www.desmos.com/calculator/jwjn5rwfy6>

# Learn More About Neural Networks

3Blue1Brown - YouTube



MAKE YOUR  
OWN  
NEURAL NETWORK



*A gentle journey through the mathematics of neural networks, and making your own using the Python computer language.*

TARIQ RASHID

# Source Code

## Kotlin Neural Network Example

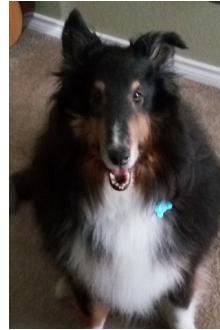
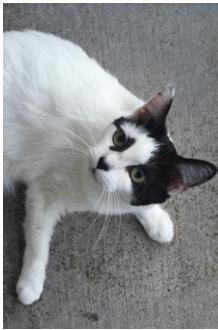
[https://github.com/thomasnield/kotlin\\_simple\\_neural\\_network](https://github.com/thomasnield/kotlin_simple_neural_network)

# Going Forward



# Use the Right “AI” for the Job

## Neural Networks



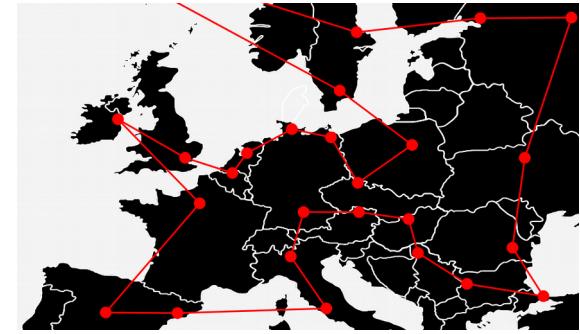
- **Image/Audio/Video Recognition**  
*“Cat” and “Dog” photo classifier*
- **Nonlinear regression**
- **Any fuzzy, difficult problems that have no clear model but lots of data**  
*Self-driving vehicles  
Natural language processing  
Problems w/ mysterious unknowns*

## Bayesian Inference



- **Text classification**  
*Email spam, sentiment analysis, document categorization*
- **Document summarization**
- **Probability inference**  
*Disease diagnosis, updating predictions*

## Discrete Optimization



- **Scheduling**  
*Staff, transportation, classrooms, sports tournaments, server jobs*
- **Route Optimization**  
*Transportation, communications*
- **Industry**  
*Manufacturing, farming, nutrition, energy, engineering, finance*

# Appendix



# Pop Culture

**Traveling Salesman (2012 Movie)**

<http://a.co/d/76UYvXd>

**Silicon Valley (HBO) - The “Not Hotdog” App**

<https://youtu.be/vlci3C4JkL0>

**Silicon Valley (HBO) - Making the “Not Hotdog” App**

<https://tinyurl.com/y97ajsac>

**XKCD - Traveling Salesman Problem**

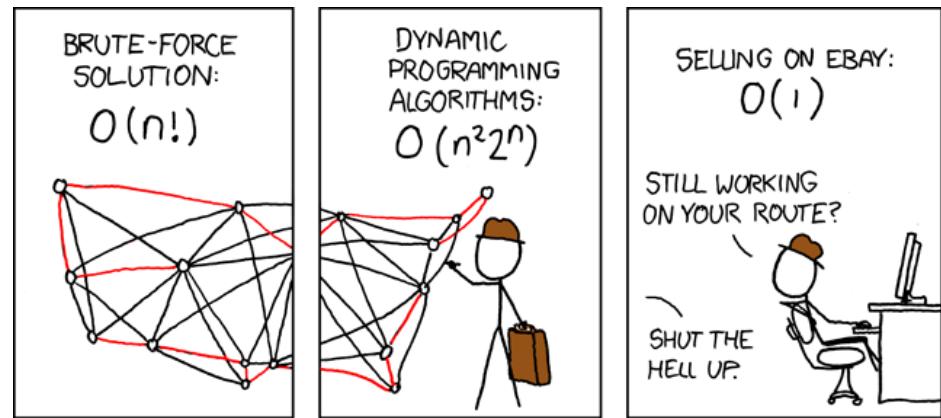
<https://www.xkcd.com/399/>

**XKCD - NP-Complete**

<https://www.xkcd.com/287/>

**XKCD - Machine Learning**

<https://xkcd.com/1838/>



SOURCE: [xkcd.com](http://xkcd.com)

# Areas to Explore

## Machine Learning

Linear Regression  
Nonlinear Regression  
Neural Networks  
Bayes Theorem/Naive Bayes  
Support Vector Machines  
Decision Trees/Random Forests  
K-means (nearest neighbor)  
XGBoost

## Optimization

Discrete Optimization  
Linear/Integer/Mixed Programming  
Dynamic Programming  
Constraint programming  
Metaheuristics

# Java/Kotlin ML and Optimization Libraries

Java/Kotlin Library	Python Equivalent	Description
ND4J / Koma	NumPy	Numerical computation Java library
DeepLearning4J	TensorFlow	Deep learning Java/Scala/Kotlin library
SMILE	scikit-learn	Comprehensive machine learning suite for Java
ojAlgo / okAlgo	PuLP, NumPy, PyTorch	Linear algebra and optimization library for Java
Apache Commons Math	scikit-learn	Math, statistics, and ML for Java
TableSaw / Krangl	Pandas	Data frame libraries for Java/Kotlin
Kotlin-Statistics	scikit-learn	Statistical/probability operators for Kotlin
JavaFX / Vegas / Data2Viz	matplotlib	Charting libraries

# Online Class Resources

## **Coursera - Discrete Optimization**

<https://www.coursera.org/learn/discrete-optimization/home/>

## **Coursera - Machine Learning**

<https://www.coursera.org/learn/machine-learning/home/welcome>

# YouTube Channels and Videos

**Thomas Nield (Channel)**

<https://youtu.be/F6RiAN1A8n0>

**Brandon Rohrer (Channel)**

<https://www.youtube.com/c/BrandonRohrer>

**3Blue1Brown (Channel)**

[https://www.youtube.com/channel/UCYO\\_jab\\_esuFRV4b17AJtAw](https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw)

**YouTube - P vs NP and the Computational Complexity Zoo (Video)**

<https://youtu.be/YX40hbAHx3s>

**The Traveling Salesman Problem Visualization (Video)**

<https://youtu.be/SC5CX8drAtU>

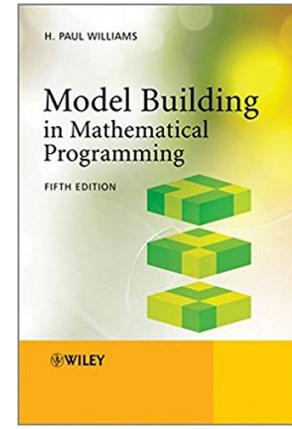
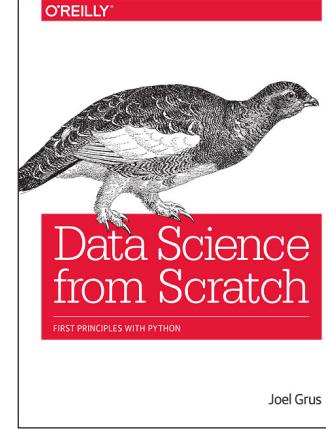
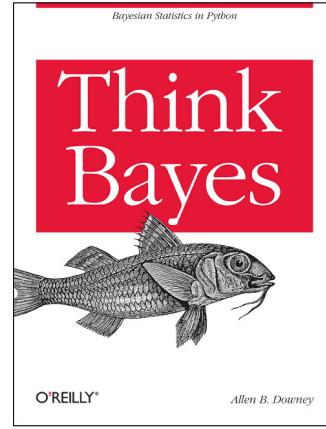
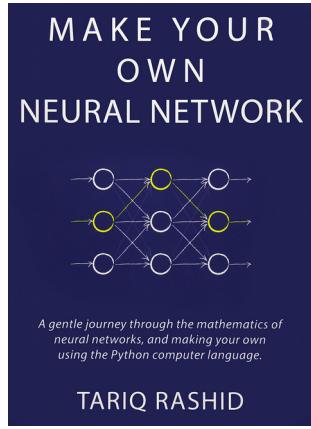
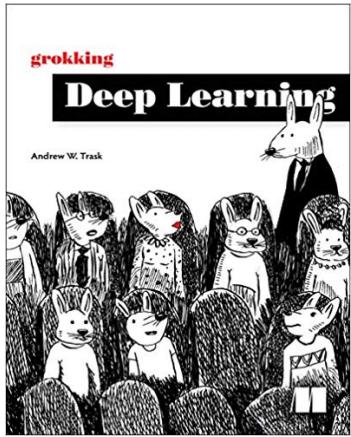
**The Traveling Salesman w/ 1000 Cities (Video)**

[https://youtu.be/W-aAjd8\\_bUc](https://youtu.be/W-aAjd8_bUc)

**Writing My First Machine Learning Game (Video)**

<https://youtu.be/ZX2Hyu5WoFg>

# Books



# Interesting Articles

## **Does A.I. Include Constraint Solvers?**

<https://www.optaplanner.org/blog/2017/09/07/DoesAIIincludeConstraintSolvers.html>

## **Can You Make Swiss Trains Even More Punctual?**

<https://medium.com/crowdai/can-you-make-swiss-trains-even-more-punctual-ec9aa73d6e35>

## **Building a Simple Chess A.I.**

<https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977>

## **The SkyNet Salesman**

<https://multithreaded.stitchfix.com/blog/2016/07/21/skytne-salesman/>

# Interesting Articles

## **Essential Math for Data Science**

<https://towardsdatascience.com/essential-math-for-data-science-why-and-how-e88271367fbd>

## **The Unreasonable Reputation of Neural Networks**

<http://thinkingmachines.mit.edu/blog/unreasonable-reputation-neural-networks>

## **The Hard Thing About Deep Learning**

<https://www.oreilly.com/ideas/the-hard-thing-about-deep-learning>

## **Mario is Hard, and that's Mathematically Official**

<https://www.newscientist.com/article/mg21328565.100-mario-is-hard-and-thats-mathematically-official/>

# Interesting Papers

## **The Lin-Kernighan Traveling Salesman Heuristic**

[http://akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH\\_REPORT.pdf](http://akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf)

## **The Traveling Salesman: A Neural Network Perspective**

[http://www.iro.umontreal.ca/~dift6751/paper\\_potvin\\_nn\\_tsp.pdf](http://www.iro.umontreal.ca/~dift6751/paper_potvin_nn_tsp.pdf)

## **The Interplay of Optimization and Machine Learning Research**

<http://jmlr.org/papers/volume7/MLOPT-intro06a/MLOPT-intro06a.pdf>