Intro to Optimization

Practical AI for Practical Problems

About the Speaker

Thomas Nield

Business Consultant, Operations Research for Southwest Airlines

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

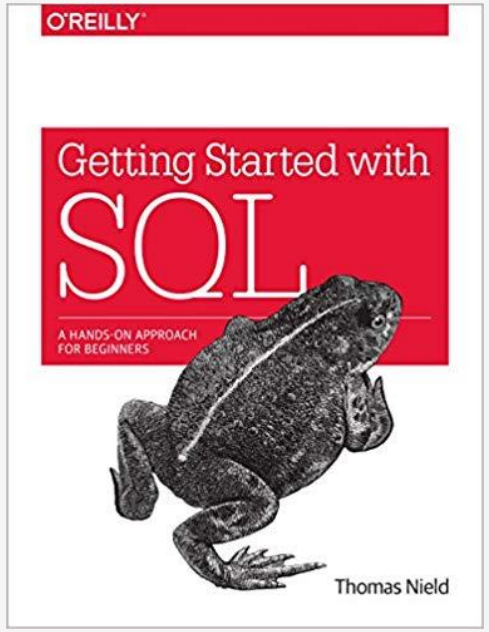
My other online trainings at O'Reilly:

SQL Fundamentals for Data

Intermediate SQL for Data Analytics

Intro to Mathematical Optimization

Machine Learning from Scratch



thomasnield9727

https://github.com/thomasnield

Agenda

Here is what we will do for the next 3 hours:

1. **Introduction** and what to expect
2. Local search algorithms using **Metaheuristics**
3. Search algorithms using **Tree Search**, **Constraint Programming**, and **Branch and Bound**
4. Convex optimization using **Linear, Integer, and Mixed Programming.**
5. **Gradient descent** and its role in machine learning

What Problems Will We Solve?

Some fun problems we will talk about!

1. **Linear regressions** from scratch
2. Finding shortest tour with **Traveling Salesman Problem**
3. Solving a **Sudoku** puzzle and **scheduling** classes
4. Manufacturing and **blending** problems
5. **Neural networks** and **gradient descent**

Section I
Intro and Overview

Demo: The Traveling Salesman

Why Learn Optimization?

Machine learning is quite the rage these days, so much that it's easy to overlook the fact there are other algorithms in the "artificial intelligence" space.

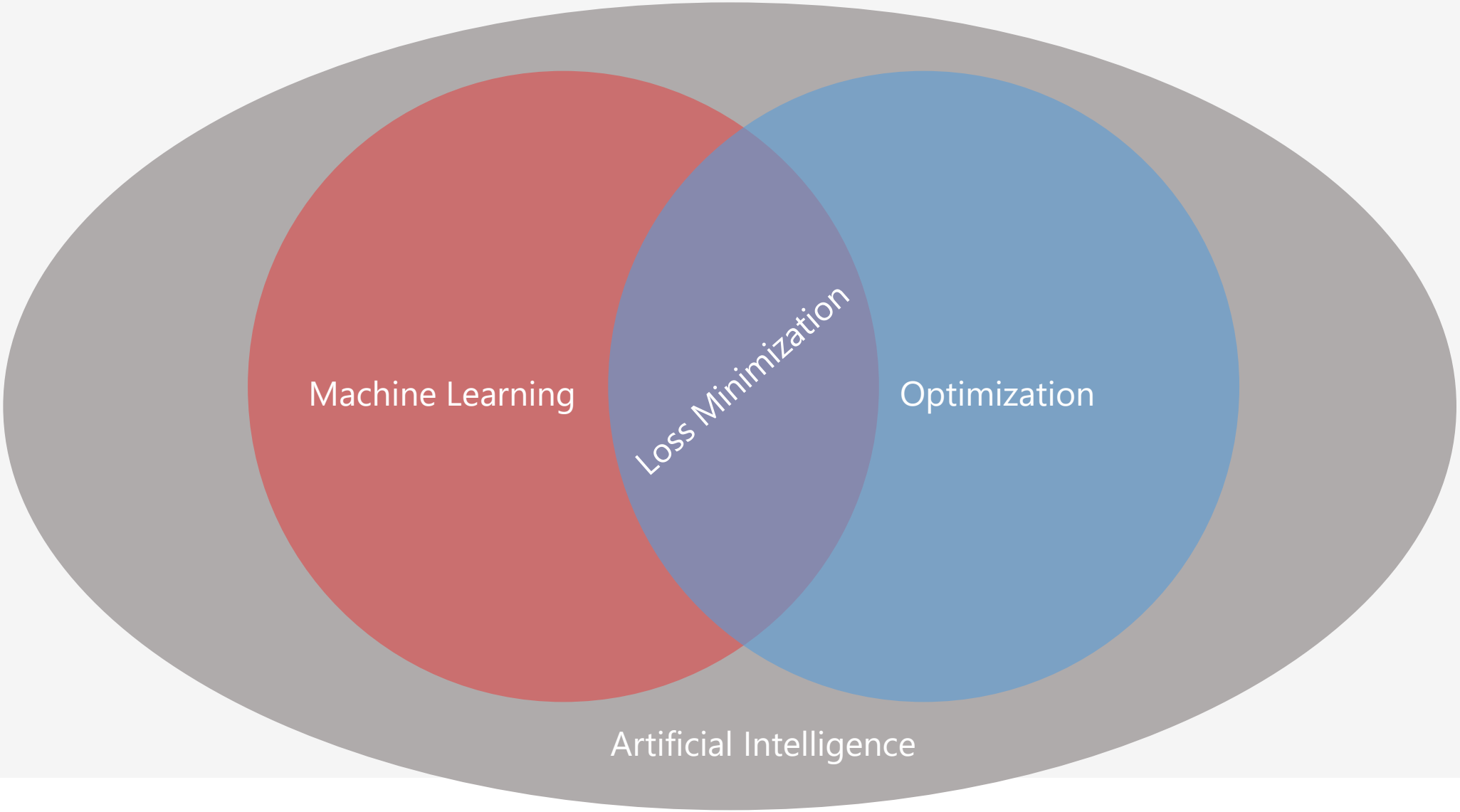
Our world economies and everyday lives are optimized algorithmically.

- Transportation network design and on-time performance
- Scheduling constrained resources like staff, transportation, server jobs, classrooms, events, etc.
- Optimizing cost/revenue/profit objectives by mixing the right decision variables.
- Solving move-based games like Sudoku and Chess

Machine learning itself is an optimization problem, as you are trying to minimize a loss function and can use techniques beyond just gradient descent.

Learn More: *https://en.wikipedia.org/wiki/Mathematical_optimization*

Machine Learning vs Optimization vs AI



Discrete versus Continuous Variables

Before we get started, it is critical we define discrete versus continuous variables.

- **Discrete Variables** – Deals with whole numbers, sets, and “countable” entities
- **Continuous Variables** – Deals with decimals, fractions, and a range of infinitely many values

Discrete optimization is dominant in real-world optimization, and is easier to reason with but harder to compute.

Continuous optimization is harder to reason with but easier to compute, while requiring more mathematical knowledge (linear algebra, calculus)

What is Discrete Optimization?

Discrete optimization is a space of algorithms that tries to find a feasible or optimal solution to a constrained problem with a "discrete" nature.

- 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.

What is Continuous Optimization?

Continuous optimization is a space of algorithms that tries to find a feasible or optimal solution to a constrained problem with a "continuous" nature.

- Finding an optimal mix of investments in a portfolio
- Fitting a percentage of on-time performance to historical transportation data
- Mixing food ingredients while minimizing costs with various suppliers.
- Minimizing loss by calculating the optimal weights for a neural network

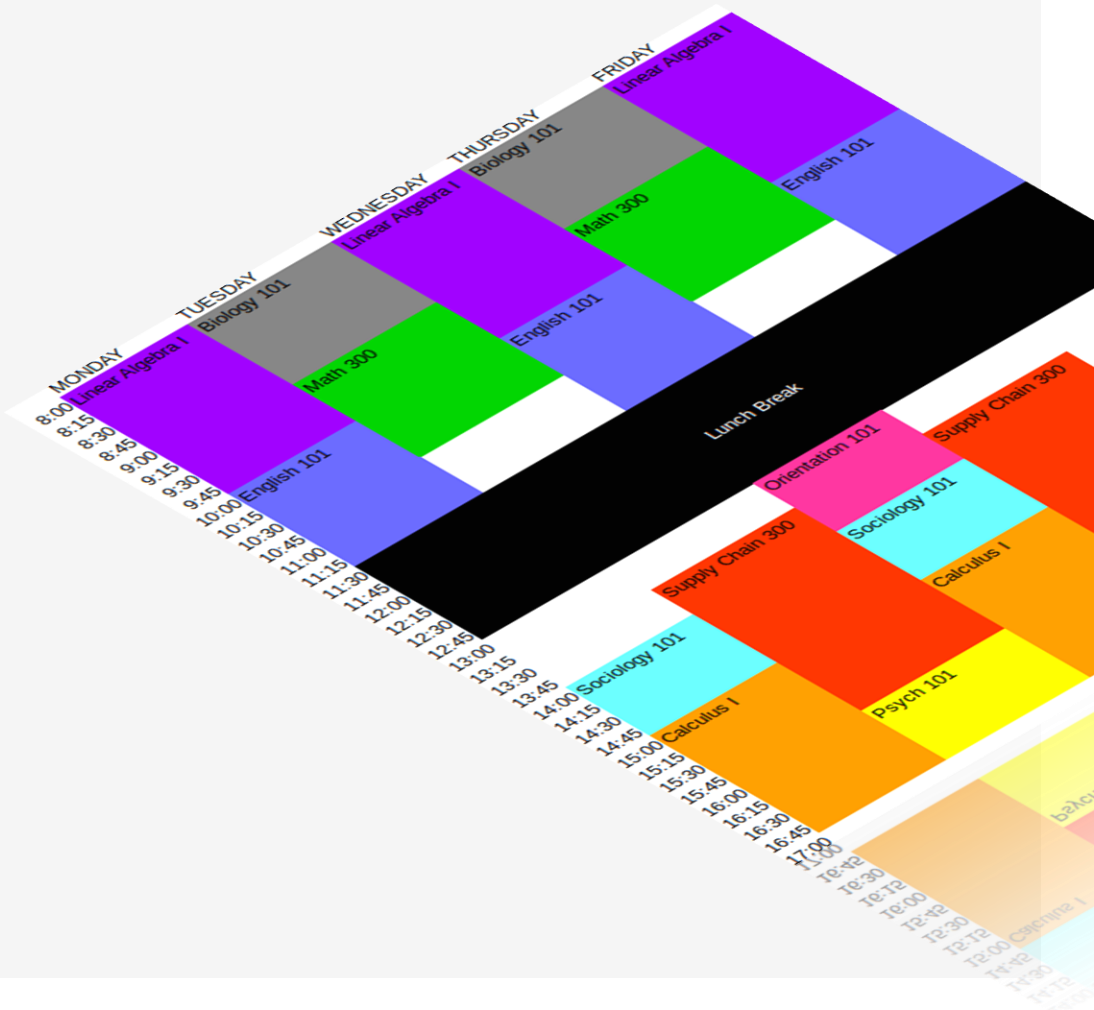
Continuous optimization is ideally solved with linear programming or calculus techniques (e.g. gradient descent), but can also employ random-based algorithms.

Section II
Metaheuristics and Random Search

What Are Search Algorithms?

Search algorithms take a problem with variables and try to find the right values to achieve a specific outcome.

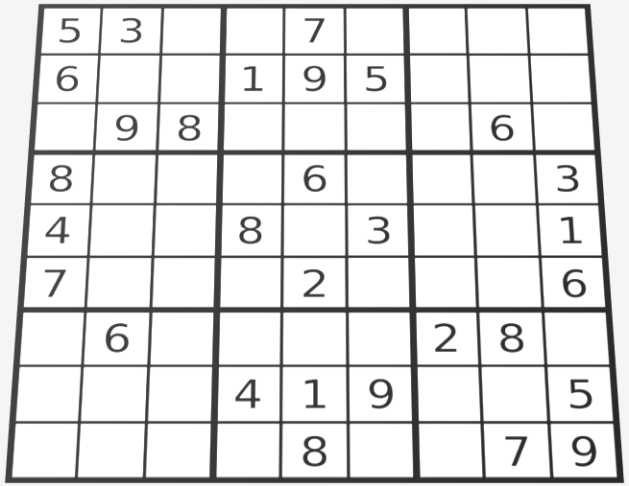
- What's the shortest round-trip tour to visit several cities?
- What time should I schedule worker shifts while respecting their hourly limits and vacation requests?
- Which numbers will solve my Sudoku?
- Given factory capacity, how much of **Product X** versus **Product Y** should I make to maximize profit?
- What weight values will minimize the loss function of my neural network?



What Are Search Algorithms?

The most naive way to solve these problems is using iterative **brute-force**, but this will never be practical because it will effectively take forever to compute outside of trivial problems.

We will need some more clever techniques!



What Are Metaheuristics?

Metaheuristics are a powerful tool to find a solution to an optimization problem.

- Highly flexible, can be relatively easy to code from scratch
- Algorithms leverage controlled randomness, which is called **stochastic programming**

A heuristic is a clever strategy to guide a search algorithm to a more likely solution.

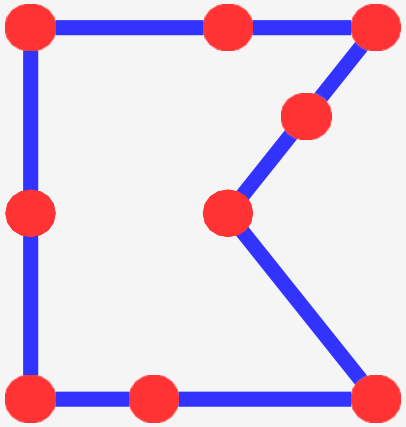
- Example: if maximizing revenue with limited factory capacity, prefer high-value items first in your search
- More generally, heuristics will start searches where a solution is more likely, and deprioritize less likely solutions.

Metaheuristics will apply a variety of heuristics, essentially using a heuristic that manages a heuristics

https://en.wikipedia.org/wiki/Metaheuristic
https://en.wikipedia.org/wiki/Stochastic_optimization
https://en.wikipedia.org/wiki/Heuristic

Traveling Salesman Problem

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



Also serves as a benchmark for transportation and routing problems.

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

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

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

Metaheuristic – Hill Climbing

1. Start with a random solution, even if it is poor quality.

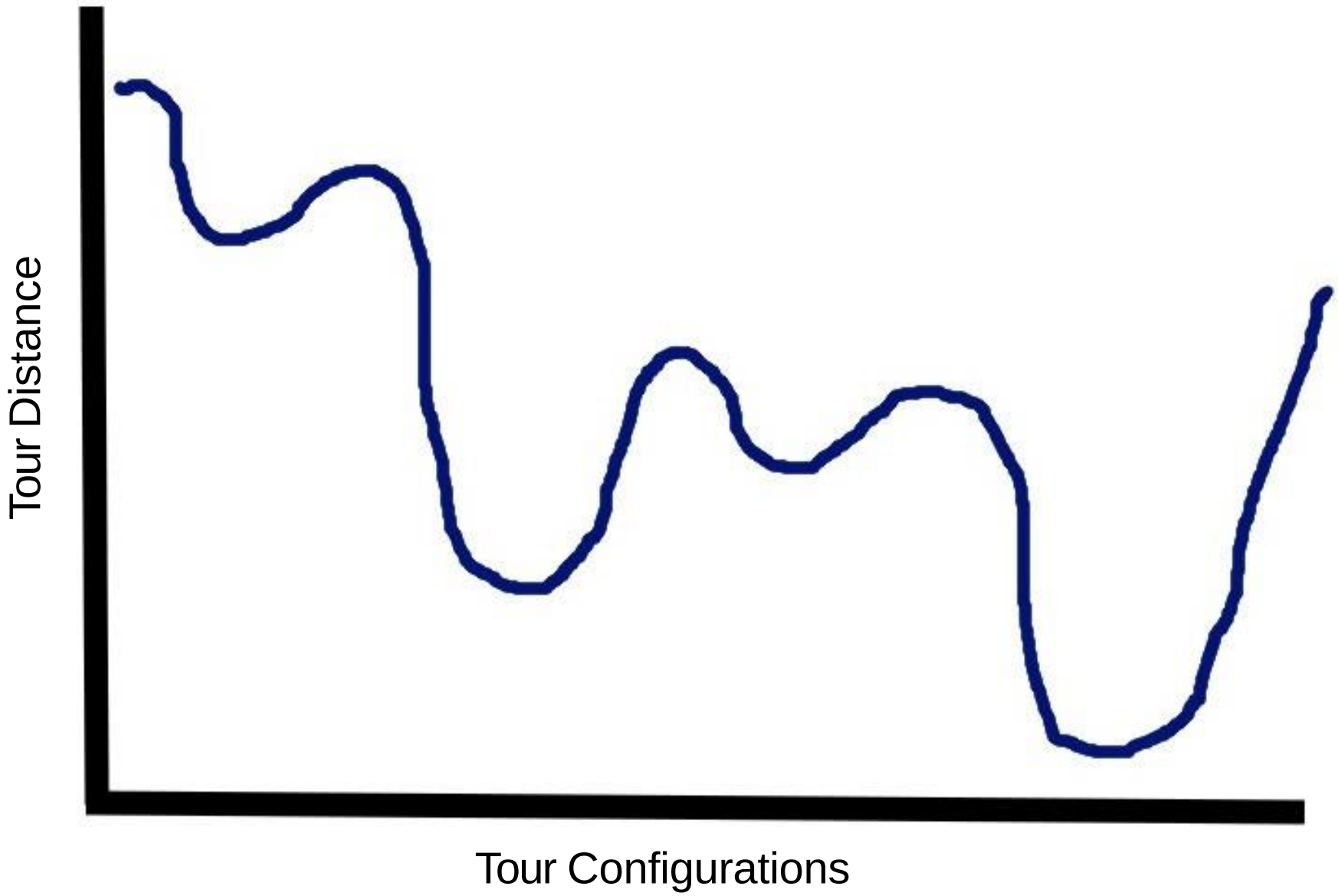
2. Repeat the following steps for a number of iterations and/or until the solution cannot improve anymore.

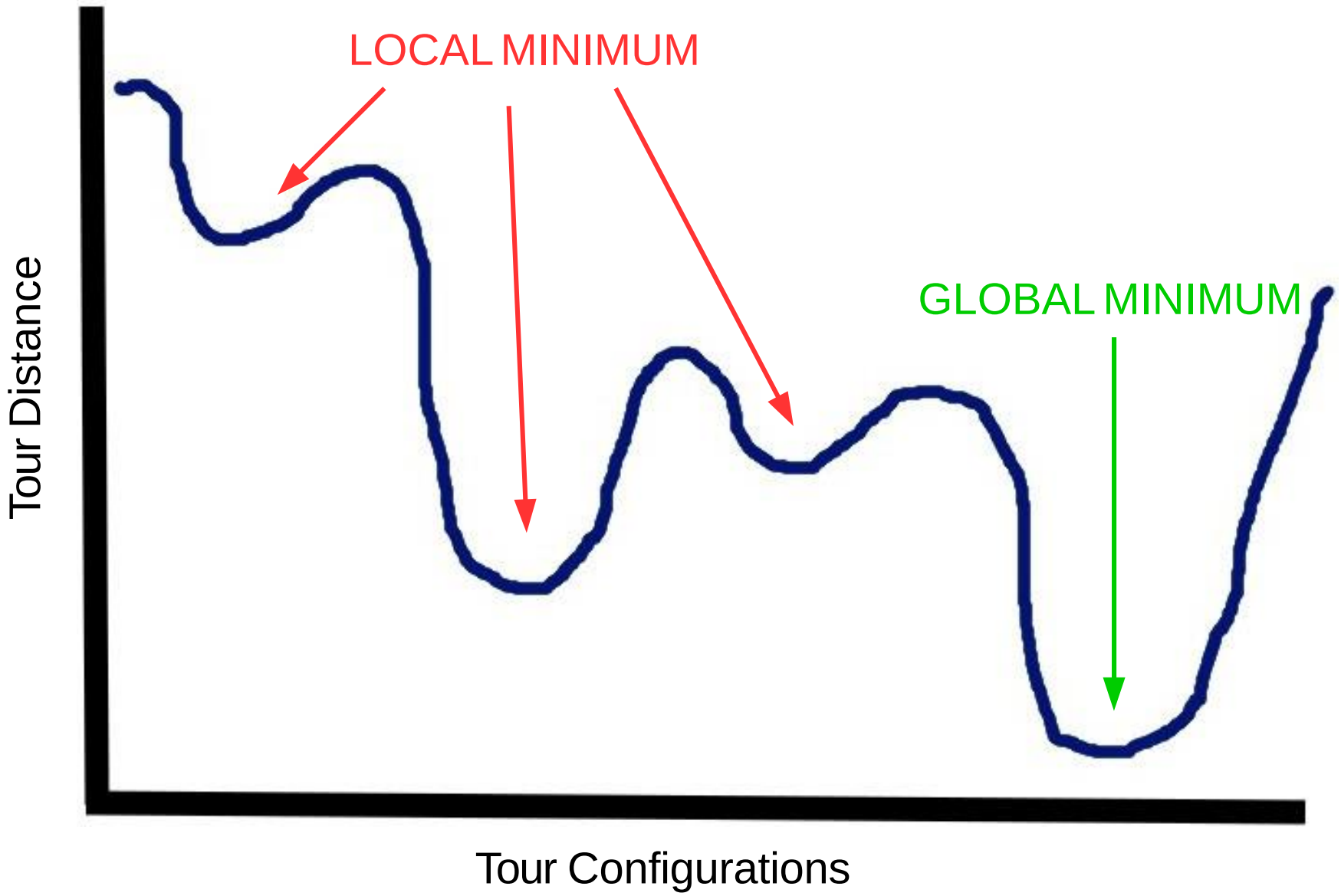
    1. Select a random part of the solution and change it.

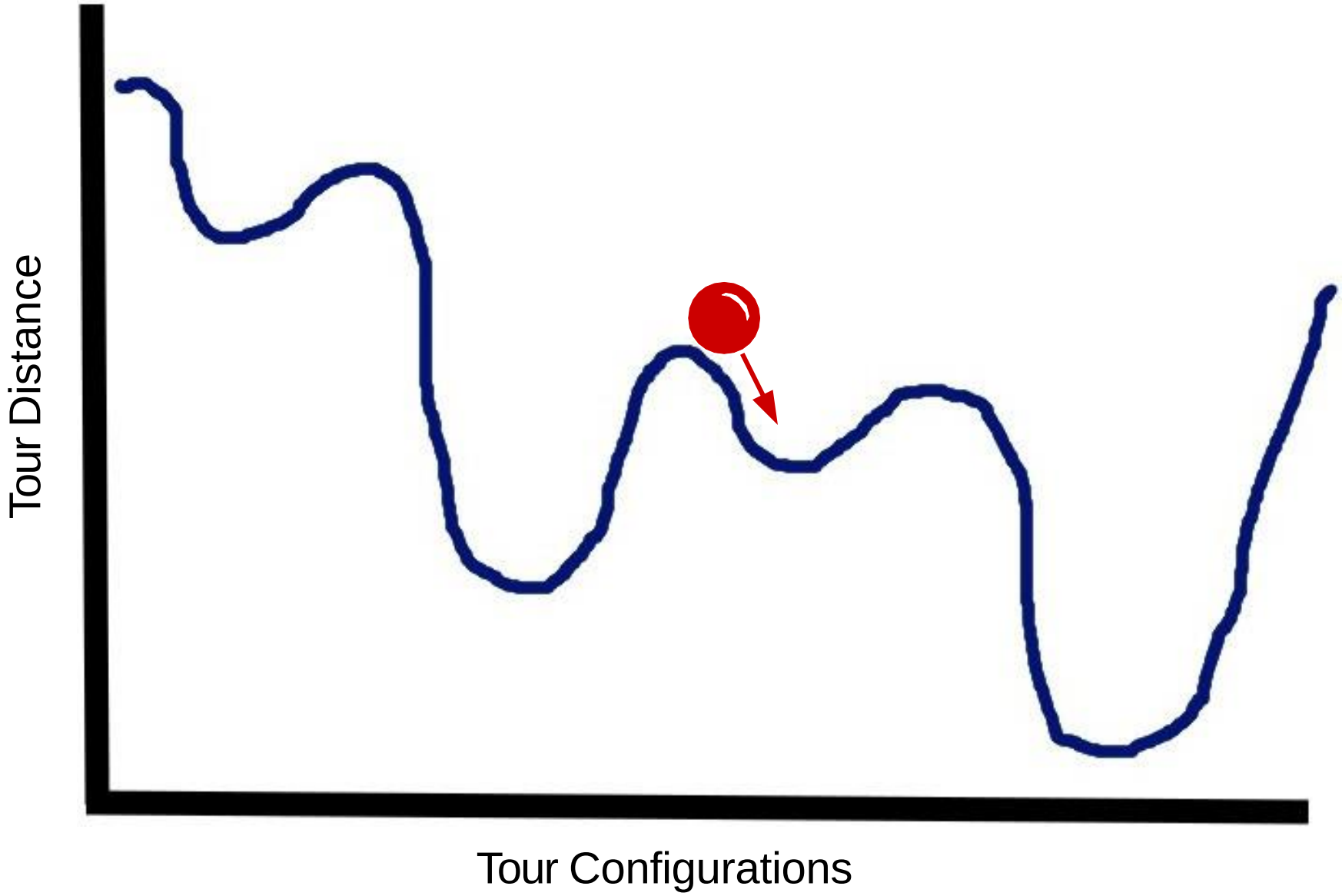
    2. If that results in an improvement, keep it.

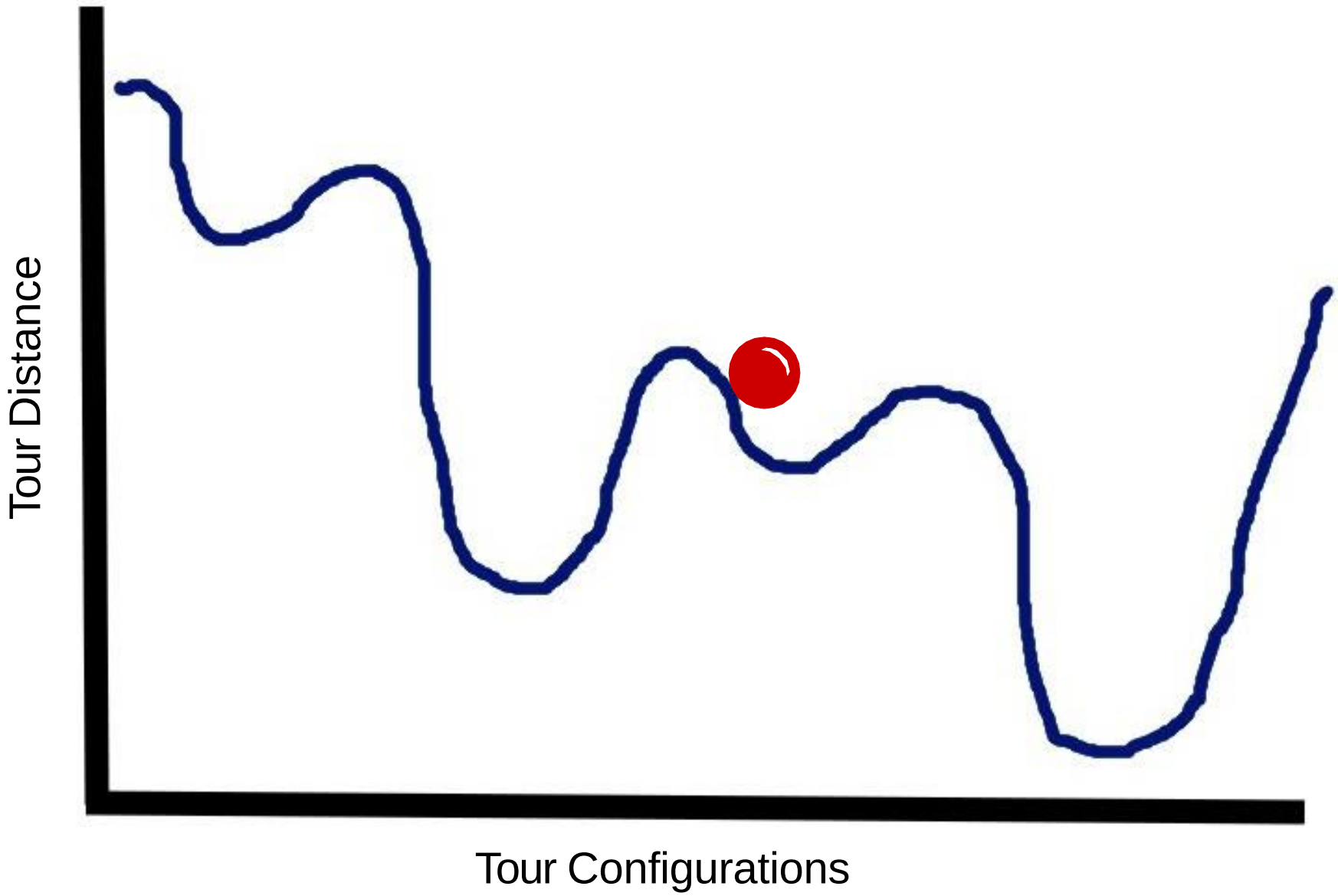
Easy, right? But there is a problem…

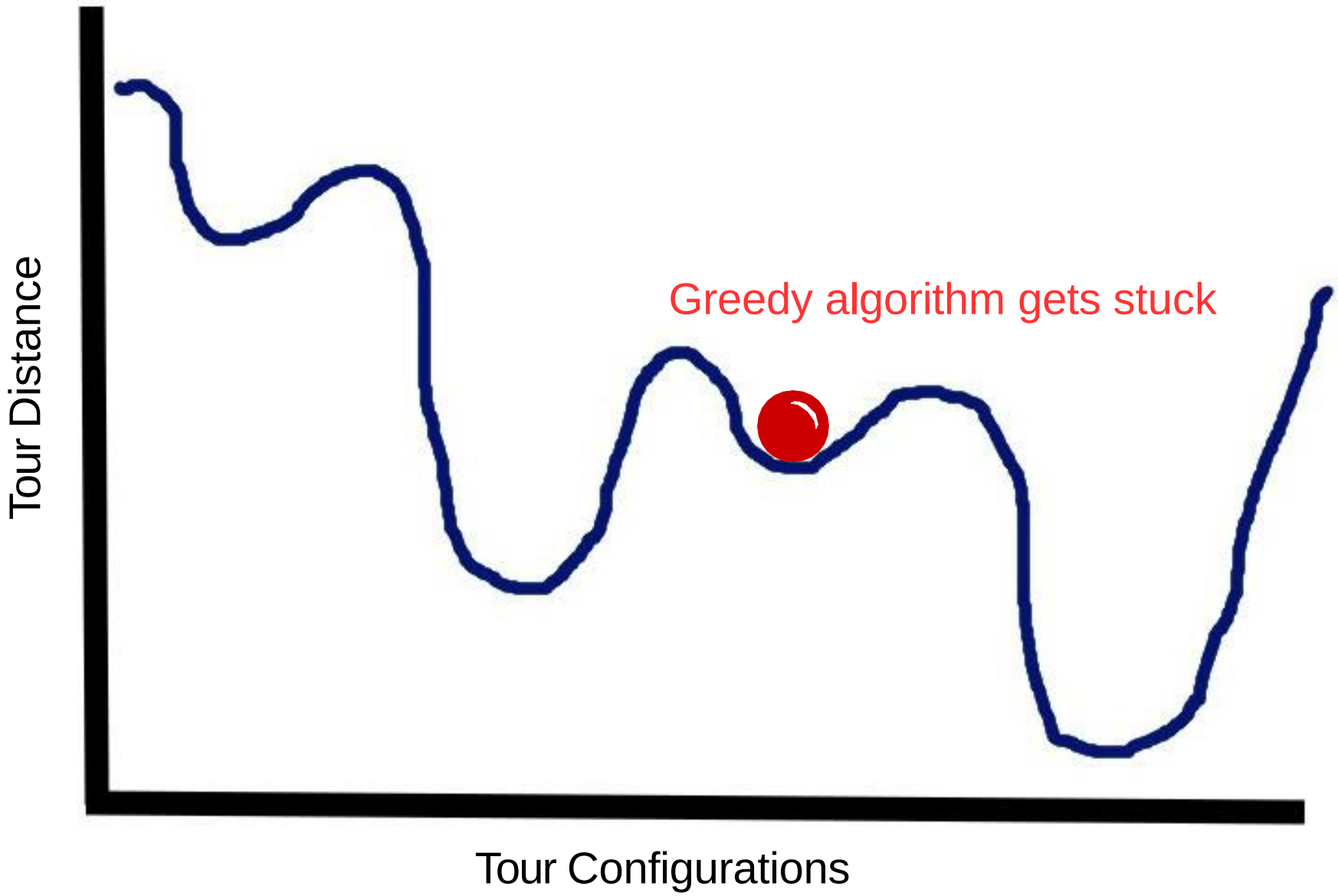
Demo: Walking Through Hill Climbing

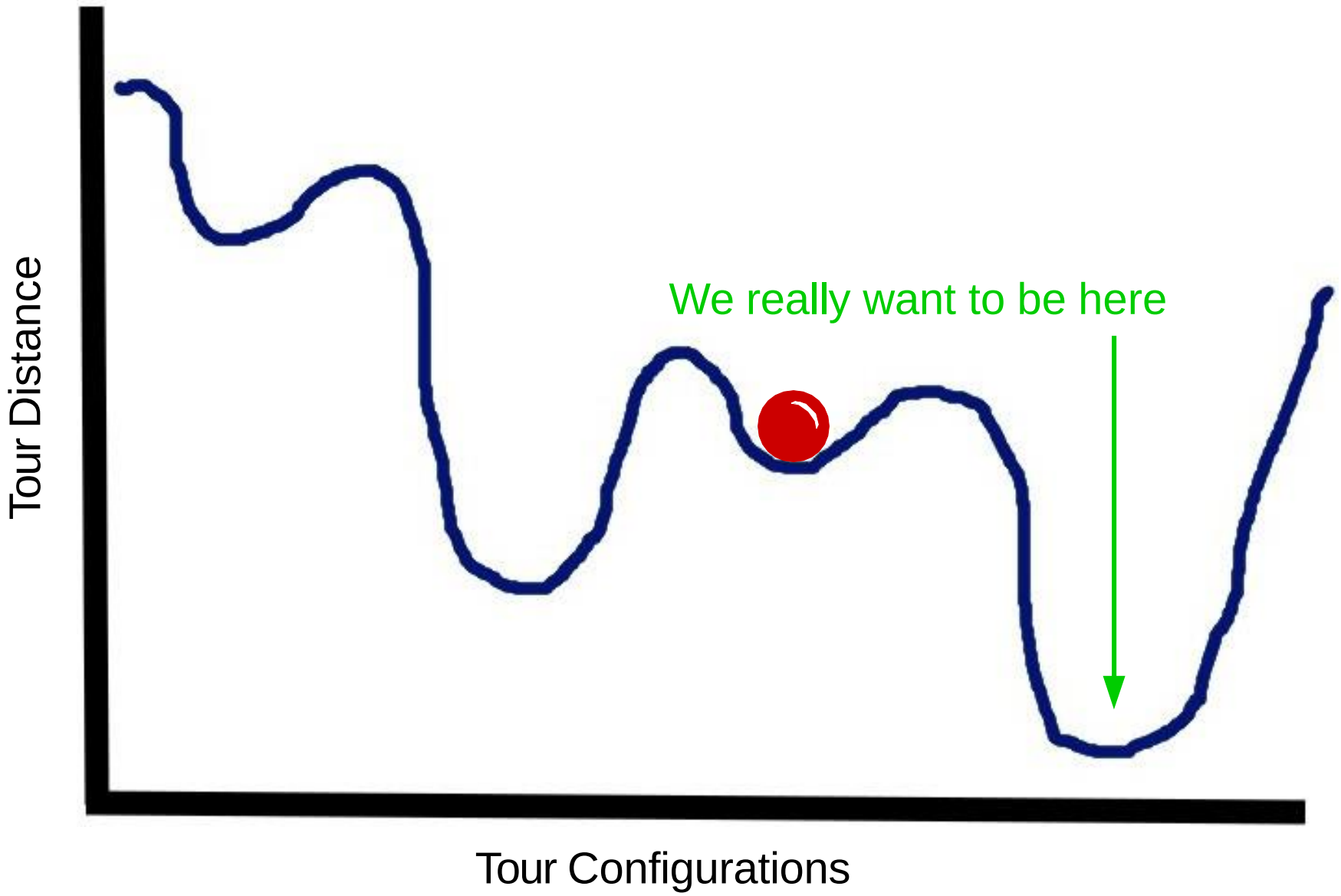

Tour Distance
Tour Configurations

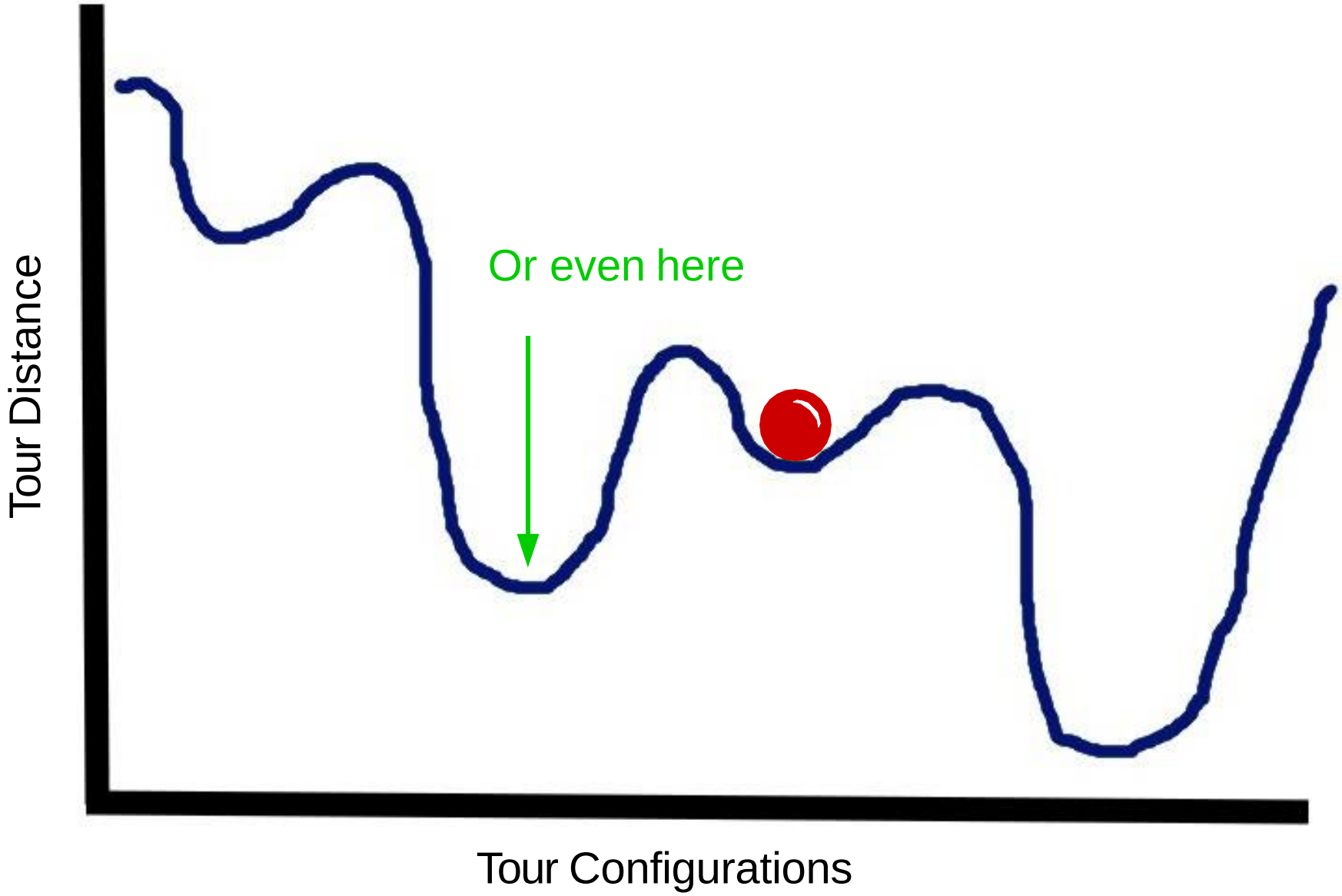

LOCAL MINIMUM
GLOBAL MINIMUM
Tour Distance
Tour Configurations

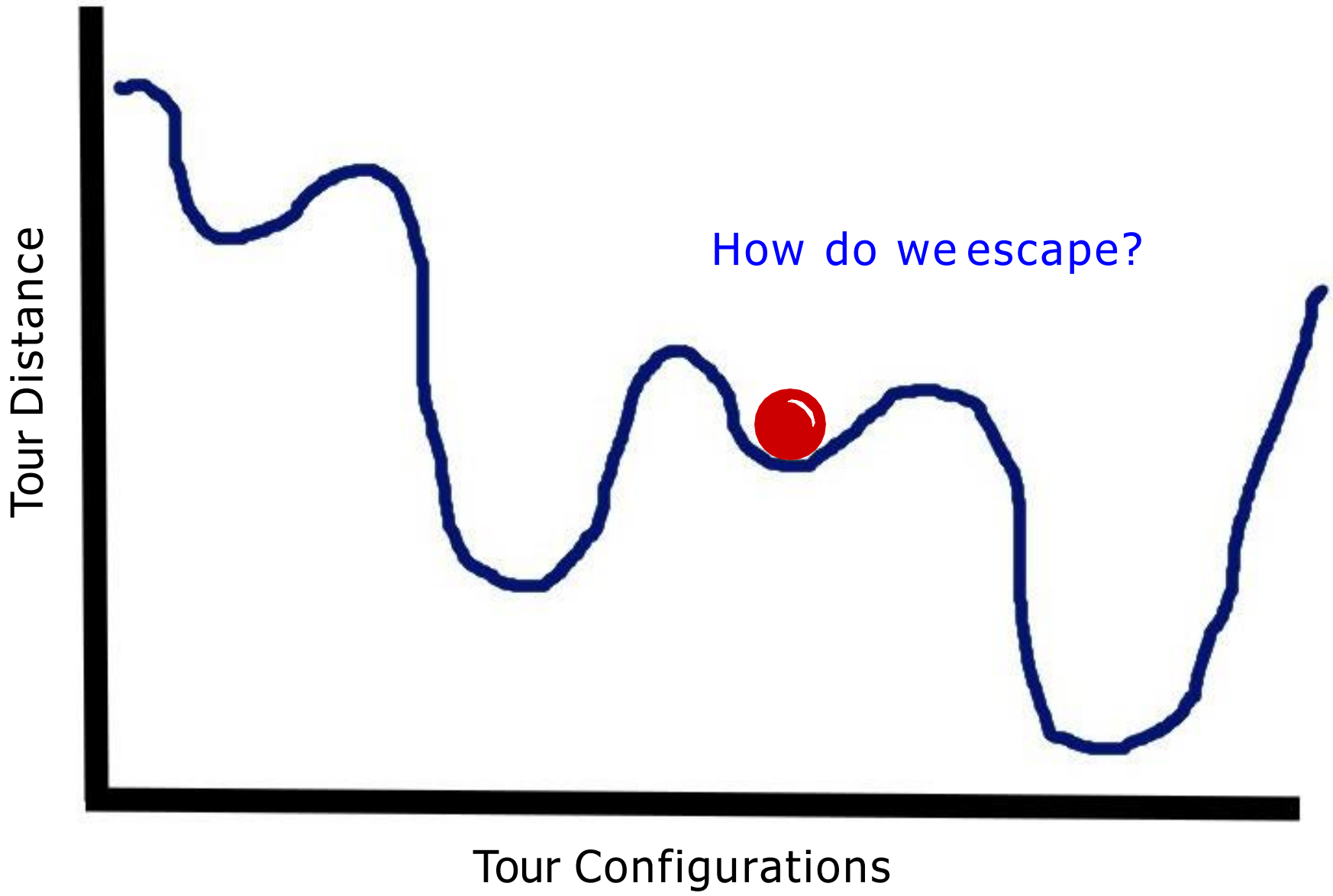

Tour Distance
Tour Configurations

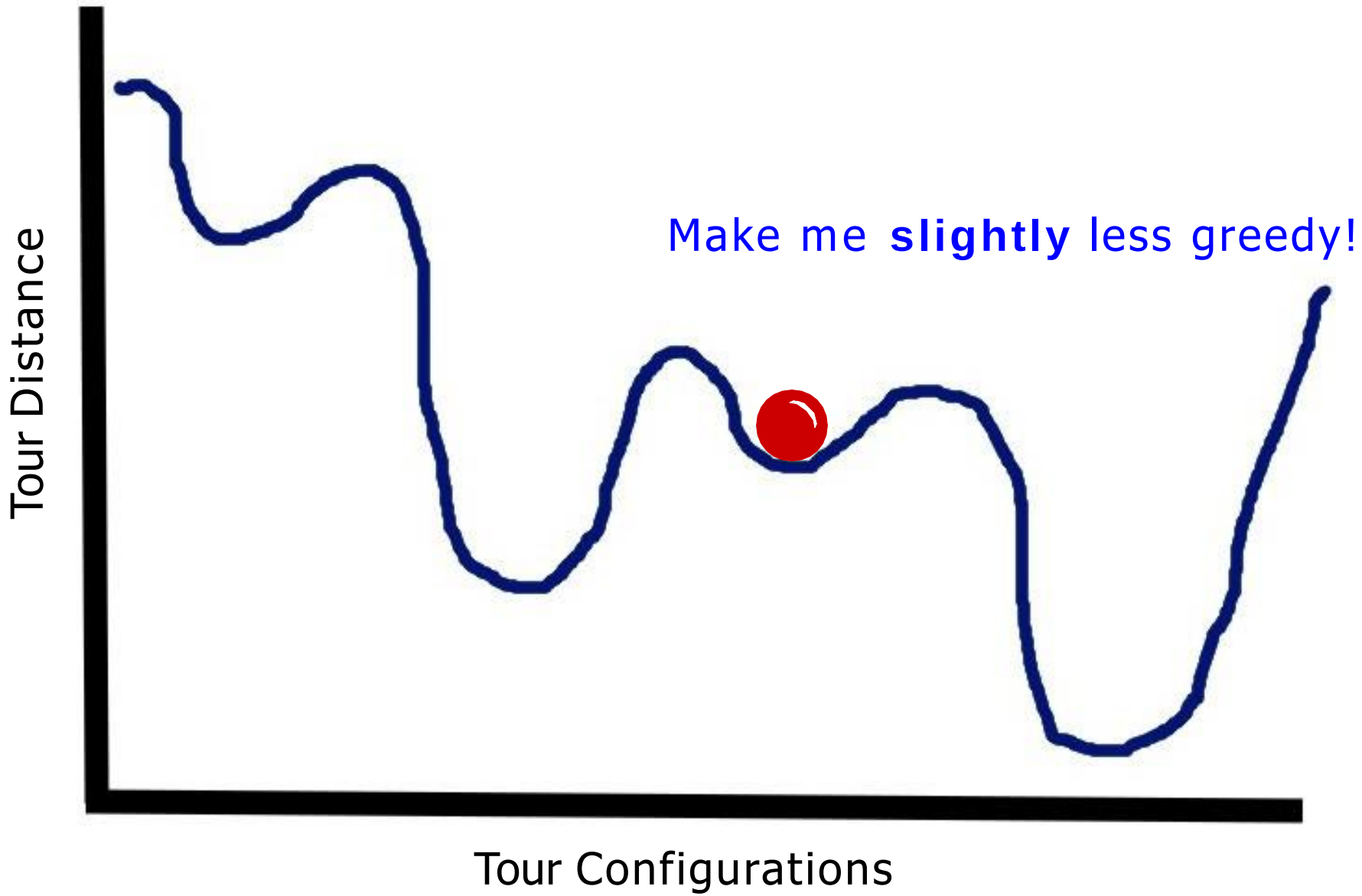

Tour Distance
Tour Configurations

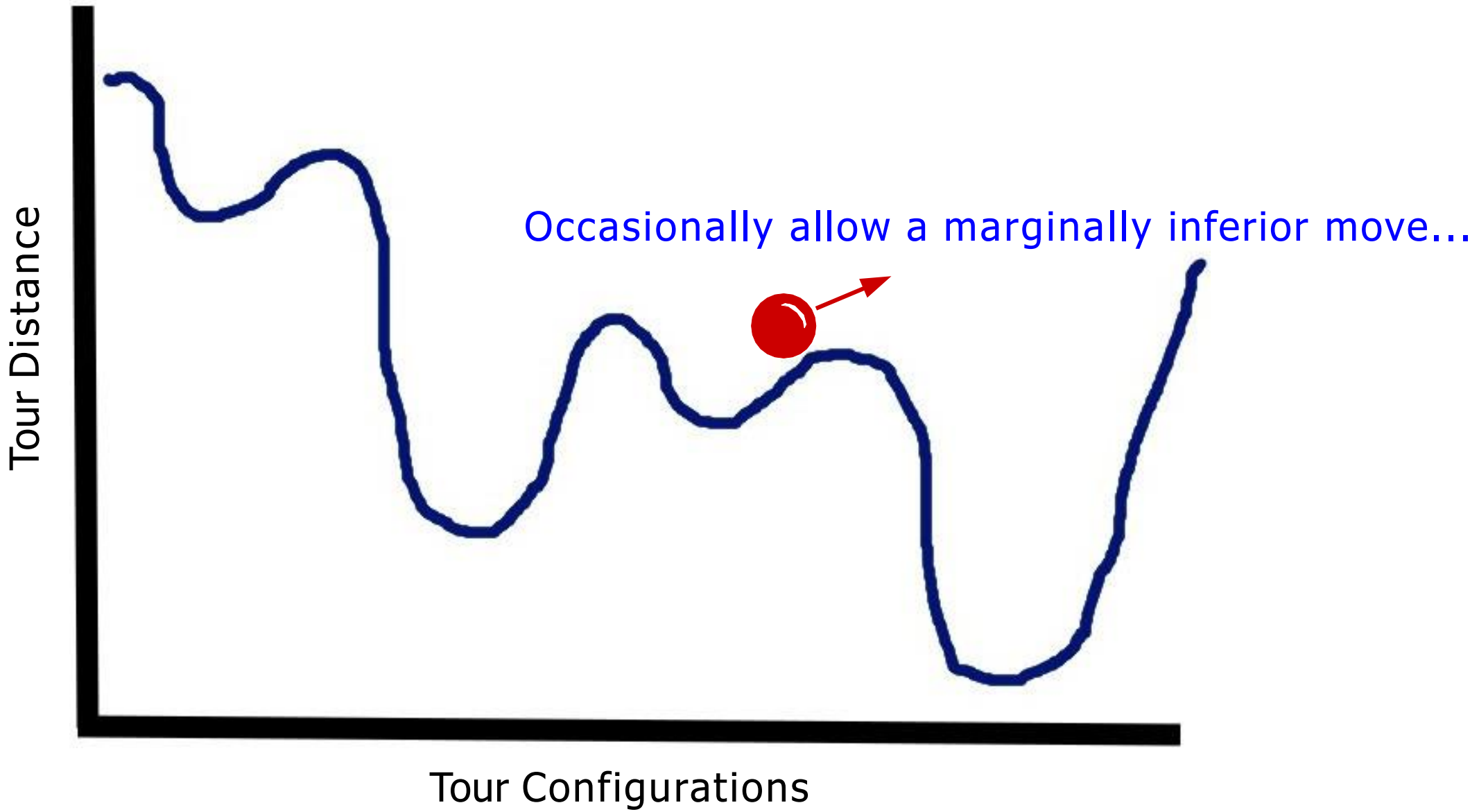

Tour Distance
Greedy algorithm gets stuck
Tour Configurations

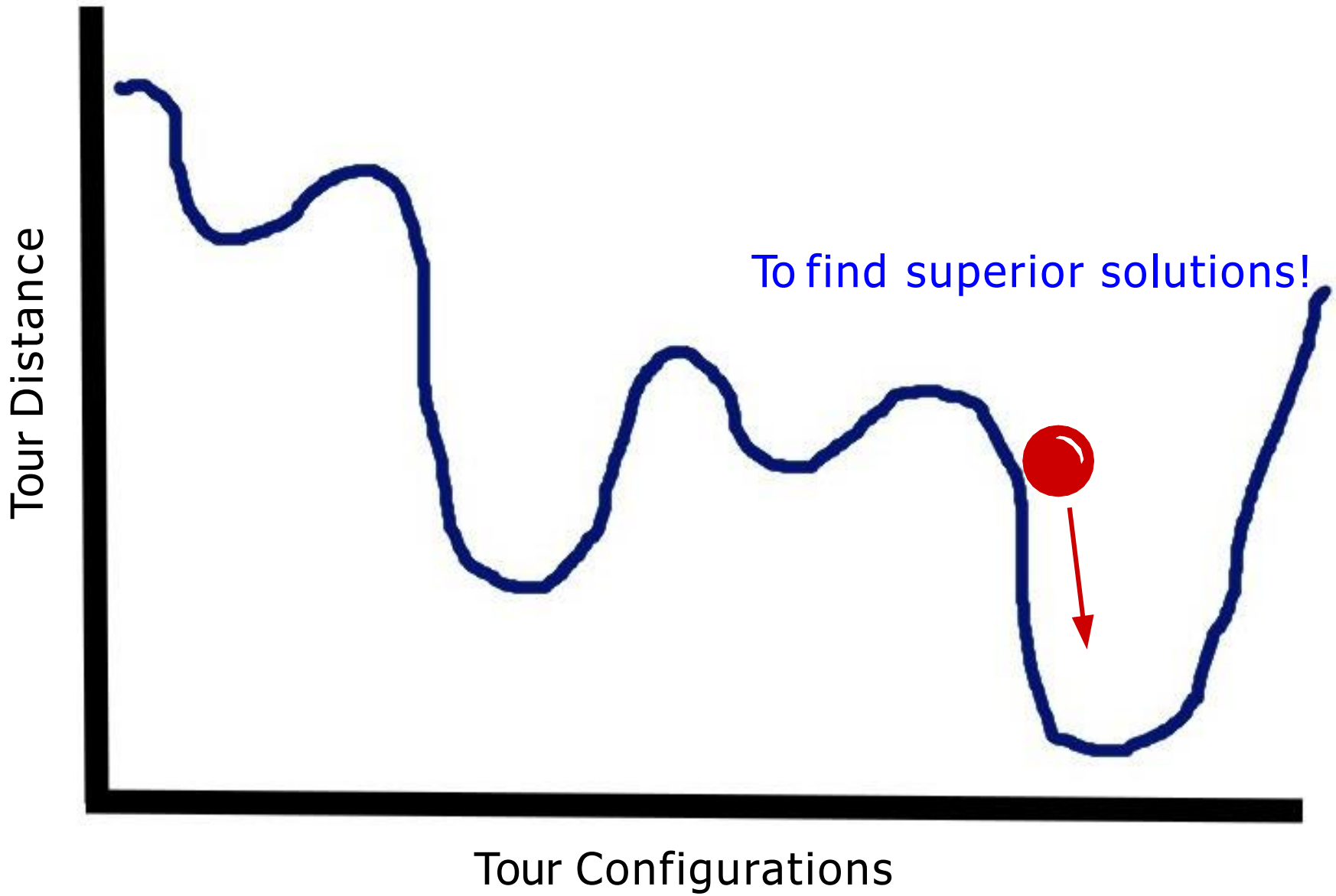

Tour Distance
We really want to be here
Tour Configurations

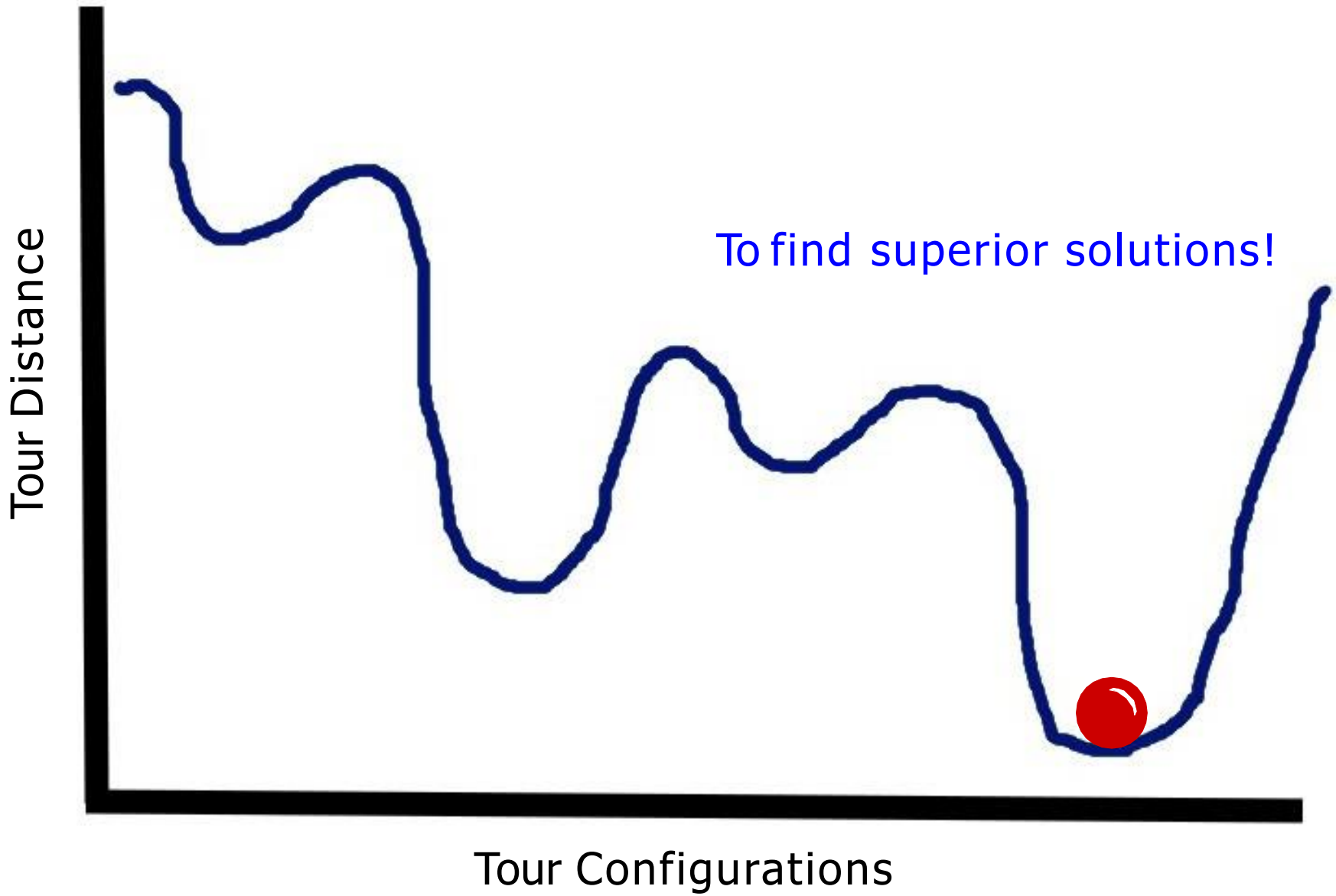

Tour Distance
Or even here
Tour Configurations

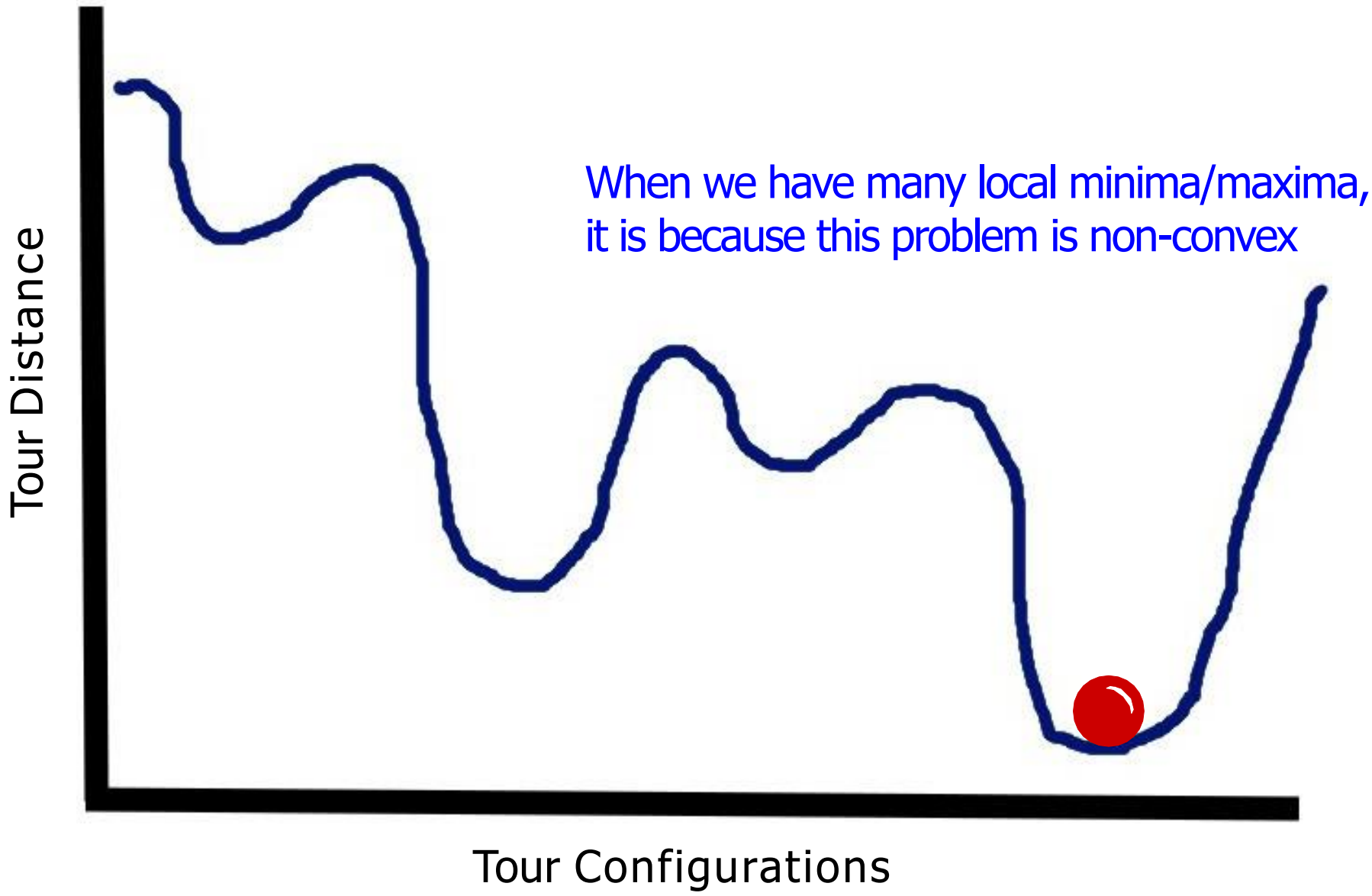

Tour Distance
How do we escape?
Tour Configurations


Tour Distance
Make me **slightly** less greedy!
Tour Configurations


Occasionally allow a marginally inferior move...
Tour Distance
Tour Configurations


Tour Distance
To find superior solutions!
Tour Configurations

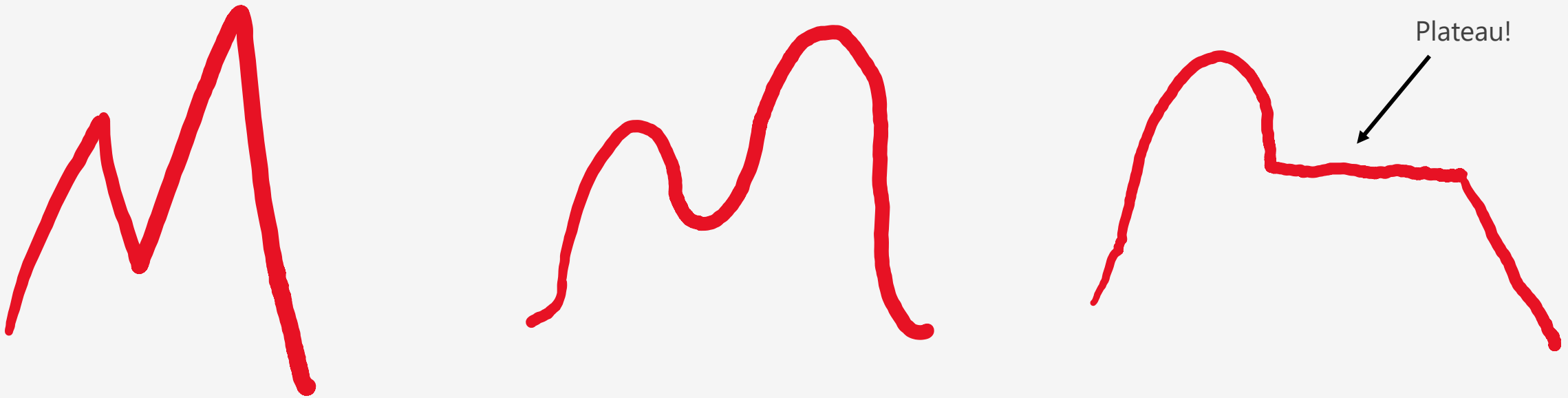

Tour Distance
To find superior solutions!
Tour Configurations


When we have many local minima/maxima,
it is because this problem is non-convex
Tour Distance
Tour Configurations

Convex versus Non-Convex

Convex

Convex versus Non-Convex



Non-Convex

Convex versus Non-Convex

If all problems were convex, optimization and machine learning would be easy because there would only be one global minimum or maximum.

Unfortunately, this is often not the case and it is easy to get stuck in a local minimum or maximum.

This is why stochastic (randomized) searches are a common solution to finding a better minimum/maximum when many exist.

Simulated Annealing

Simulated annealing is another metaheuristic similar to hill-climbing, but it periodically allows inferior moves hoping it will lead to better solutions.

It can be surprisingly versatile, and used to optimize problems ranging from the Traveling Salesman to tuning neural networks.

So how do you determine whether to keep an inferior move?

- A **temperature schedule** throttles periodically between greediness (don't accept an inferior move) and randomness (accept an inferior move).
- A given temperature on the given iteration will result in a weighted coin flip, and that coin flip will determine whether the inferior move is accepted or not.

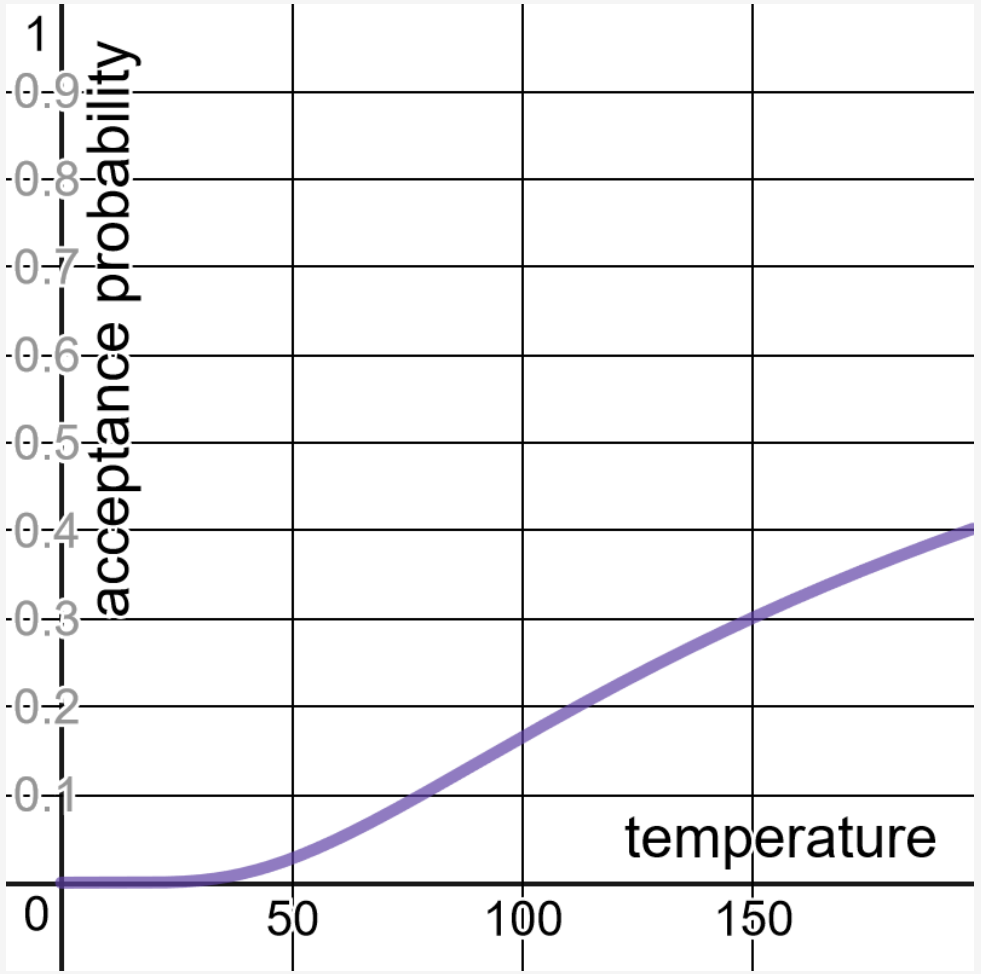
Calculating the Weighted Coin Flip Probability

*Where **x** is a given temperature, here is how to convert it into a probability of accepting an inferior move.*

***d*_{degrading}** *and* ***d*_{current}** *reflect the inferior distance and current distance of the move respectively.*

$$\exp\left(\frac{-\left(d_{degrading}-d_{current}\right)}{x}\right)$$

https://www.desmos.com/calculator/rpfpfiq7ce



Metaheuristic - Simulated Annealing

1. Start with a random solution, even if it is poor quality.

2. At each given temperature/iteration, do the following steps:

    1. Select a random variable in the solution and change it.

    2. If that results in an improvement, keep it.

    3. If the result is inferior, *flip a weighted coin* based on the temperature and degradation of the move to determine whether to keep it.

3. If your solution is not acceptable, modify your temperature schedule and try again.

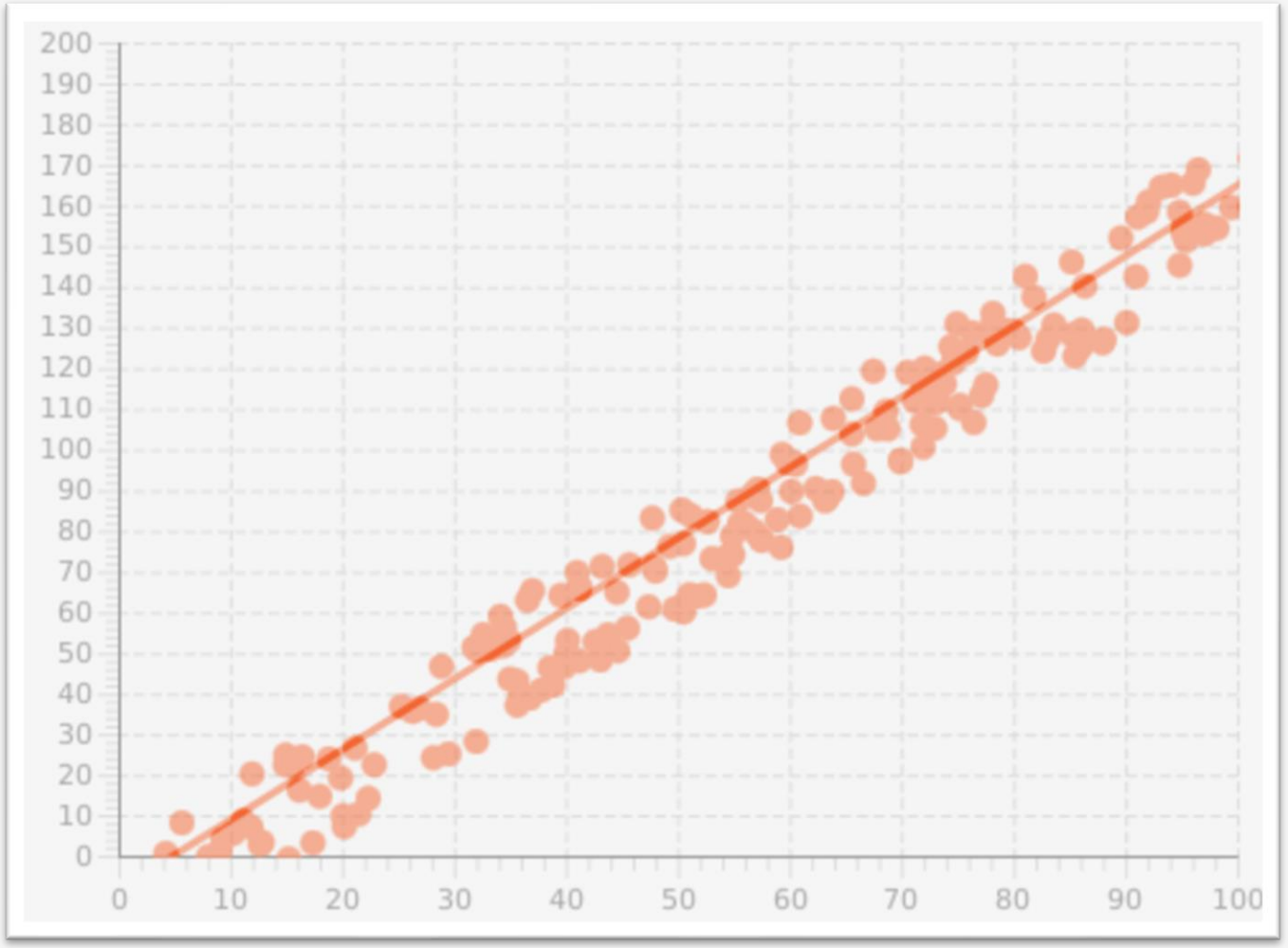
Demo: Walking Through Simulated Annealing

Metaheuristics for Linear Regression

Let's use hill climbing/simulated annealing for a simple problem with continuous variables: linear regression.

A simple linear regression fits a line of the form **y = mx + b**, where we find the best **m** and **b** values that fit the points, minimizing the sum of least squared differences.

Note: For an intuitive graph demonstration sum of least squares fitting: https://www.desmos.com/calculator/lywhybeztt



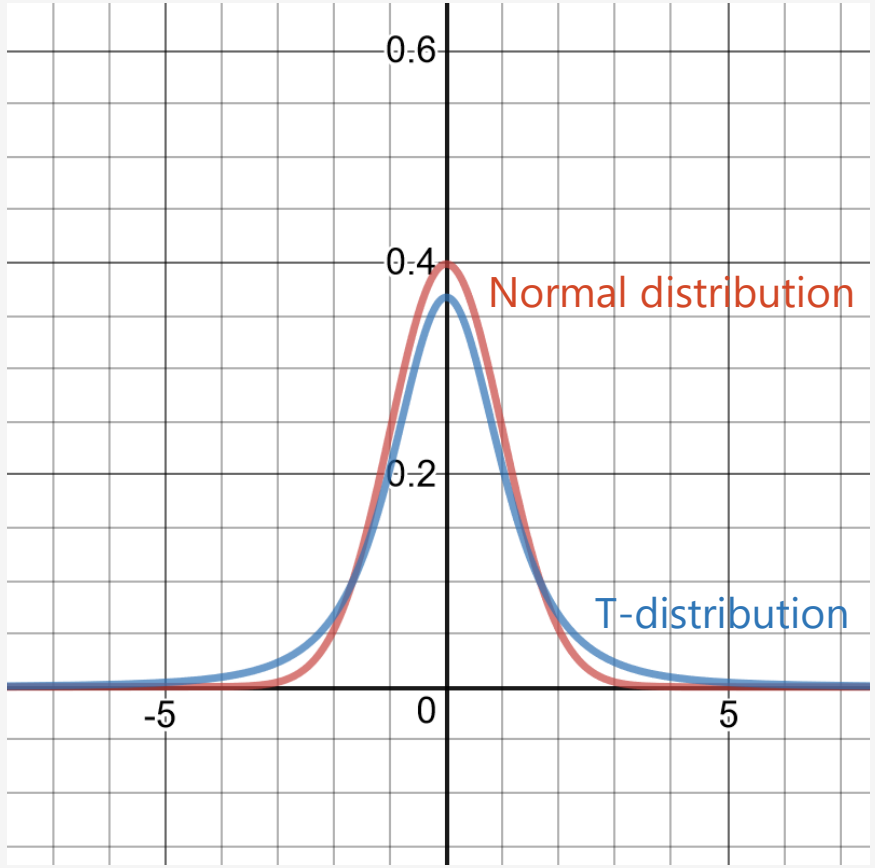
Metaheuristics for Linear Regression

Since they are continuous and not discrete, how do we randomly change values for **m** or **b**?

- We randomly increase/decrease **m** and **b** with random values from a standard normal distribution, or even better a T-Distribution which has fatter tails.

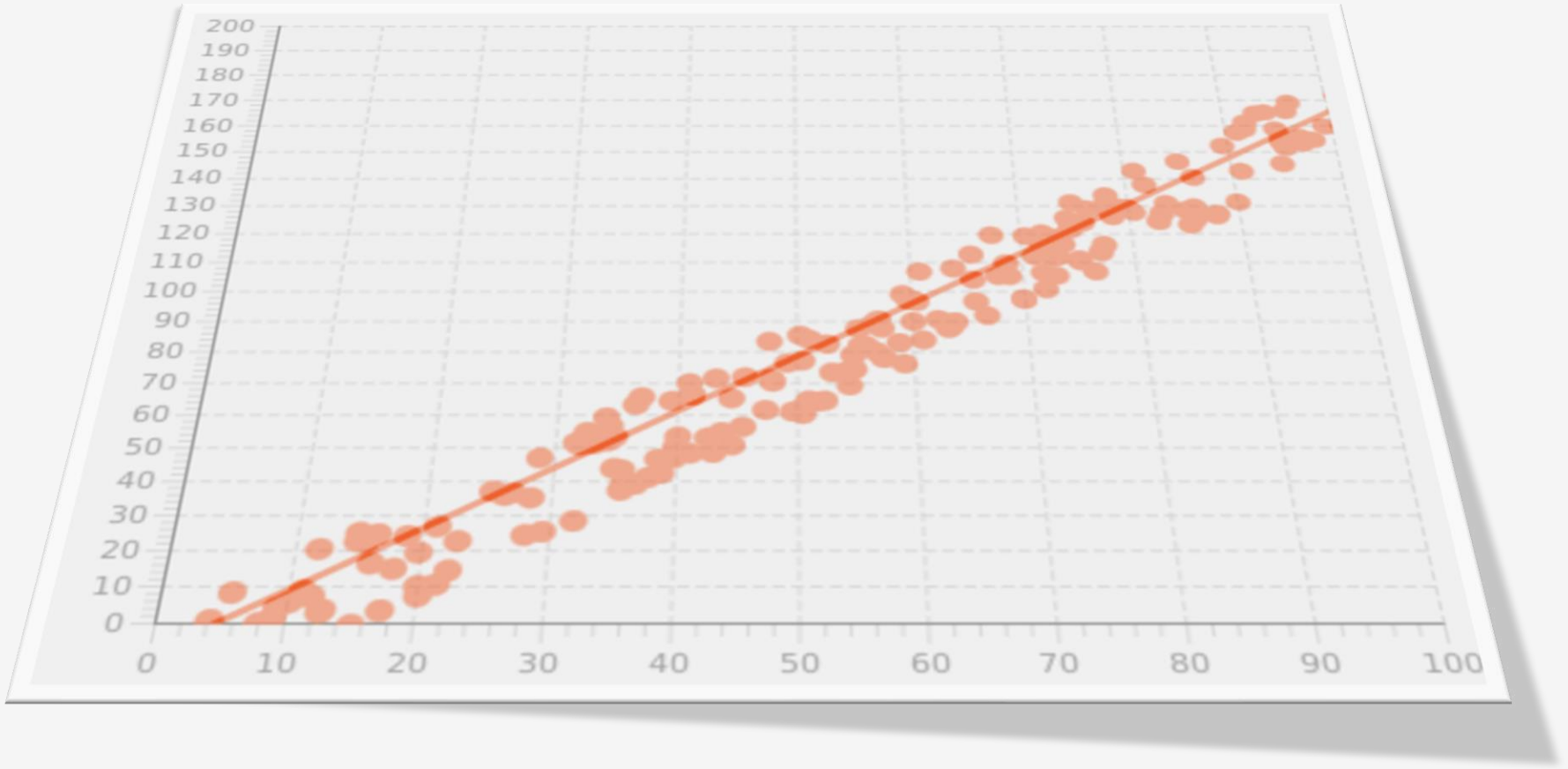
- Fatter tails = more larger adjustments = more diverse moves.

We can use these random adjustments to **m** and **b** as our moves in hill climbing and simulated annealing.



https://www.desmos.com/calculator/xm56tvvalh

Demo: Linear Regression

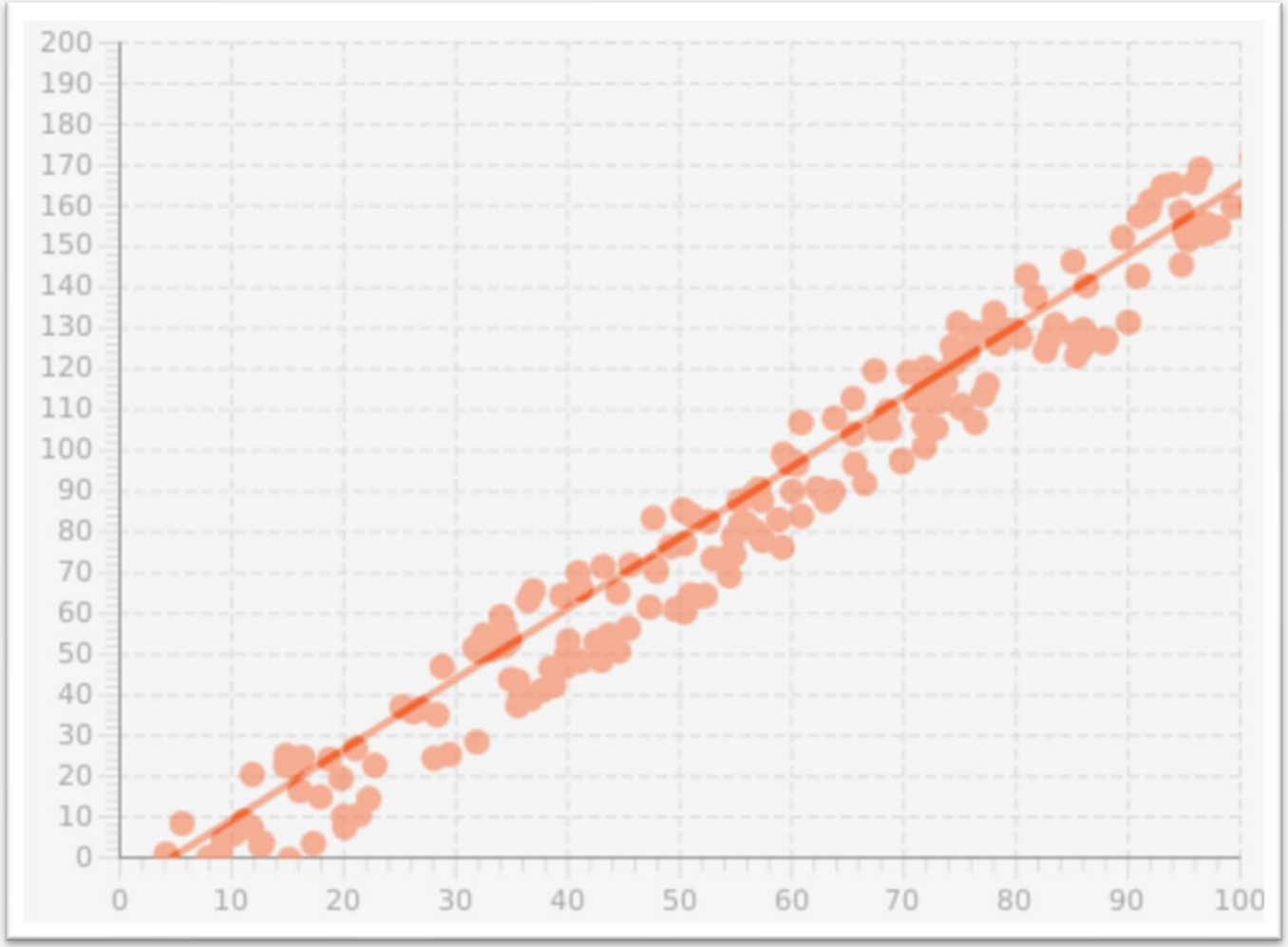


Metaheuristics for Quantile Regression

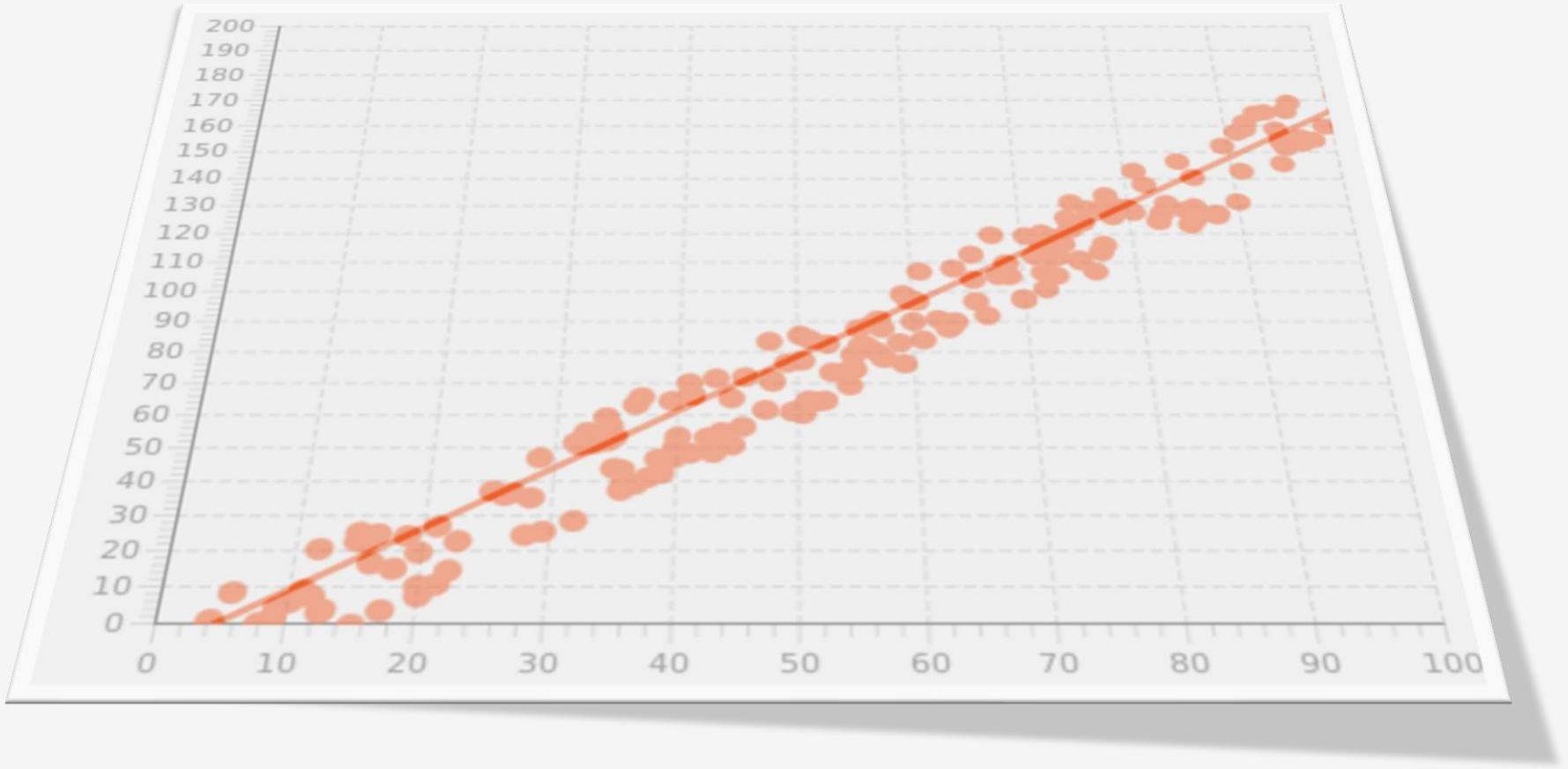
What's cool about building search algorithms is we can pursue weird and niche objectives even if we have incomplete information or no mathematical model, which happens often in the real world.

This may not yield the most efficient or precise approach, but it can be effective nonetheless. For example, quantile regression is just like linear regression, but it puts a percentage of points under the line.

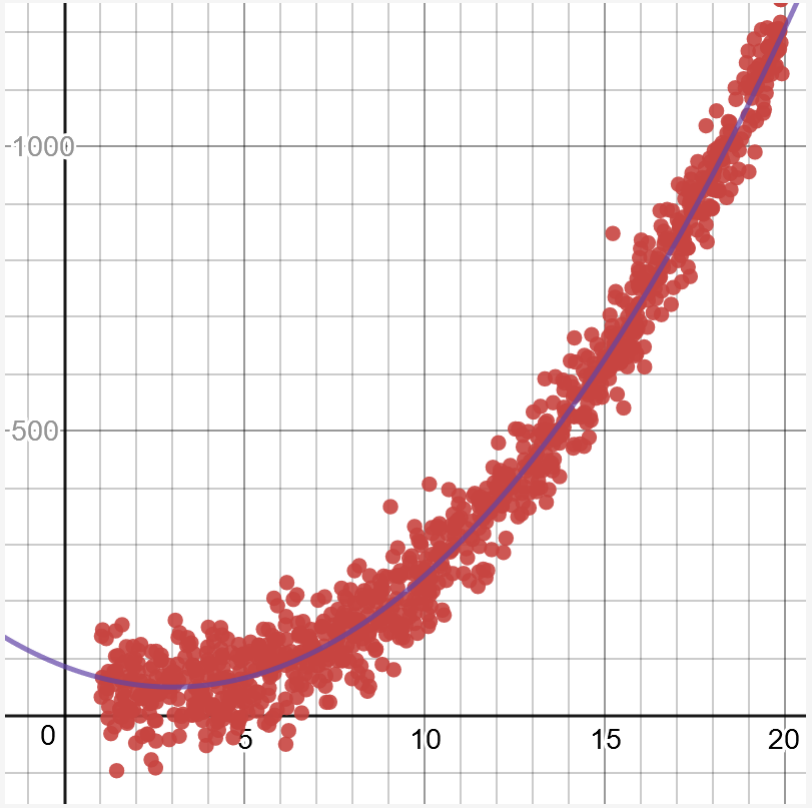
If you want 80% of the point to fall under the line, you can quickly build a search algorithm seeking that objective rather than having to deep dive into niche Calculus concepts.



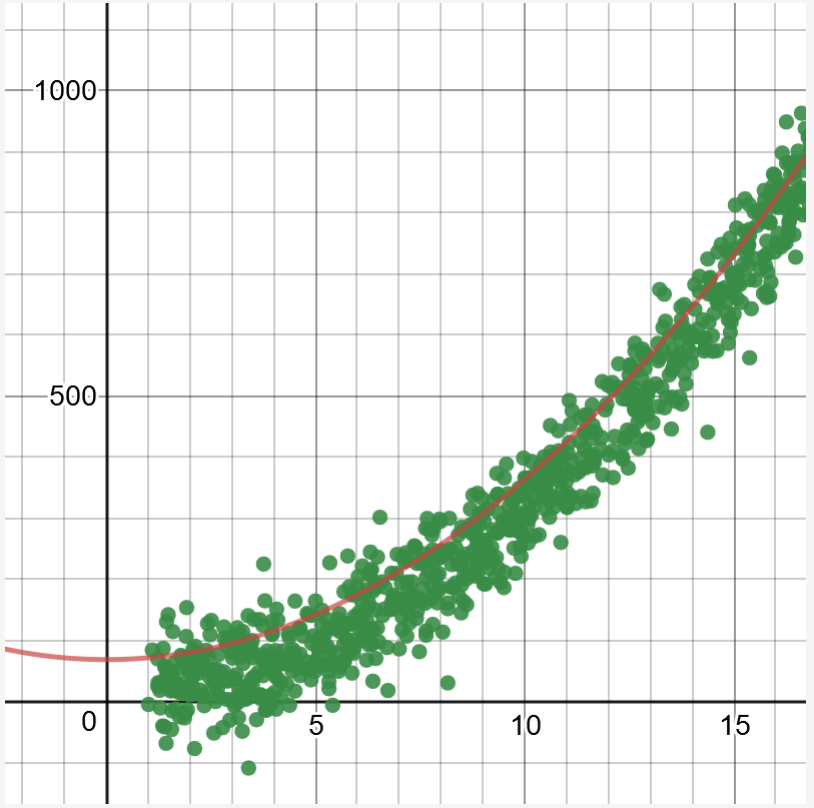
Demo: Quantile Regression



Use Metaheuristics for Nonlinear Regressions Too!



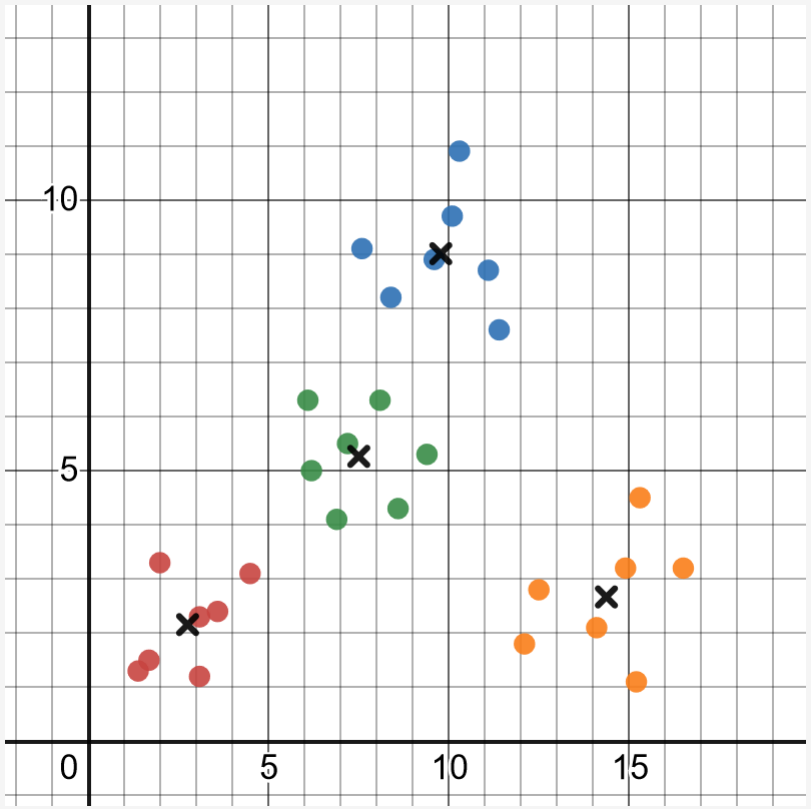
Regression for $f(x) = ax^2 + b$
https://www.desmos.com/calculator/zdlsp6erjq



80% quantile regression for $f(x) = ax^2 + b$
https://www.desmos.com/calculator/4yp5m3lj5h

Hill Climbing and K-Means Clustering

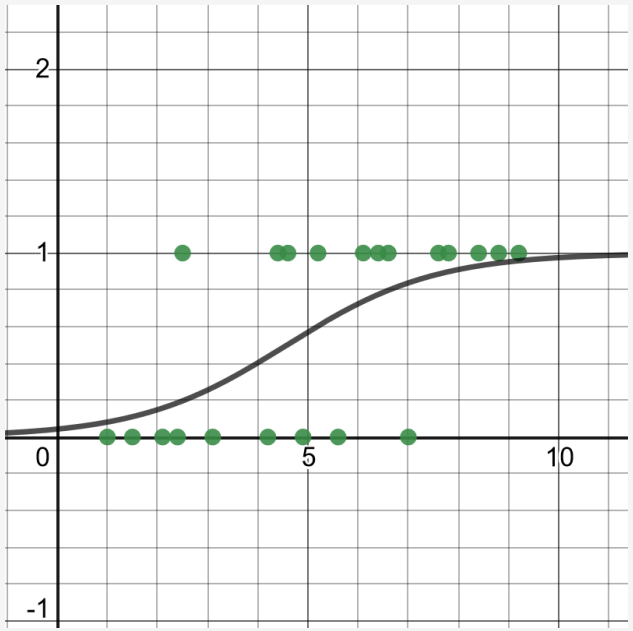
```
import pandas as pd
import numpy as np
import random

# Desmos graph: https://www.desmos.com/calculator/pb4ewmqdvy

class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def __str__(self):
      return "{0},{1}".format(self.x, self.y)


def distance_between(point1, point2):
   return pow(pow(point2.x - point1.x, 2) + pow(point2.y - point1.y, 2), .5)


k = 4
points = [(Point(row.x,row.y)) for index, row in
pd.read_csv("https://tinyurl.com/y25lvxug").iterrows() ]
centroids = [Point(0, 0) for i in range(k)]

best_loss = 1_000_000_000.0

for i in range(100_000):
   random_centroid = random.choice(centroids)

   random_x_adjust = np.random.standard_normal()
   random_y_adjust = np.random.standard_normal()

   random_centroid.x += random_x_adjust
   random_centroid.y += random_y_adjust

   new_loss = 0.0

   for p in points:
      new_loss += pow(min([distance_between(p,c) for c in centroids]), 2)
```



https://tinyurl.com/yy8lzvtt

Hill Climbing and Logistic Regression

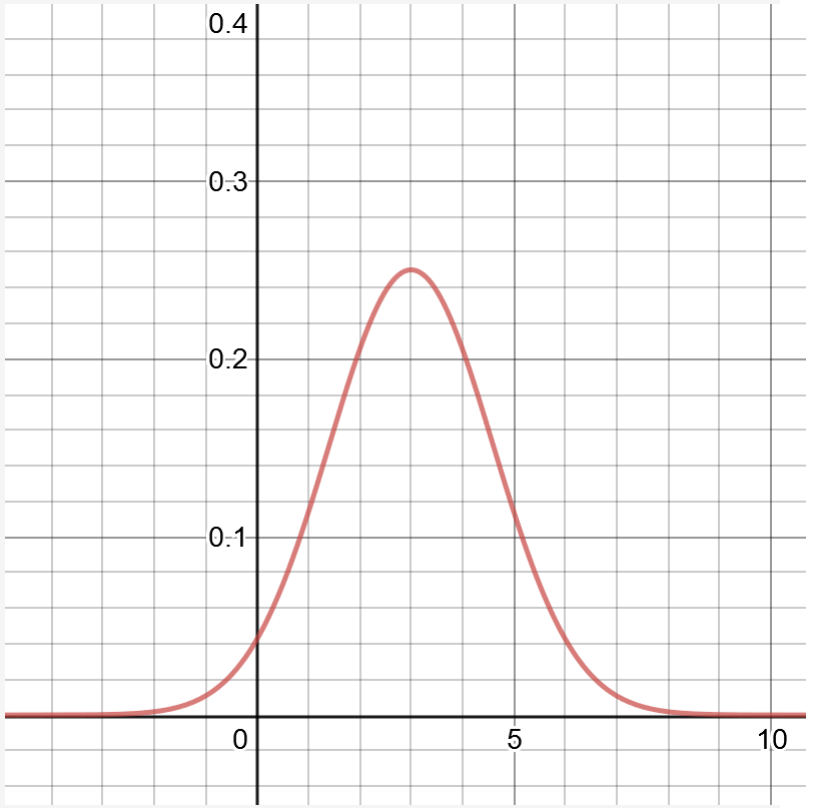
```
import random

import numpy as np
import pandas as pd
import math


# Desmos graph: https://www.desmos.com/calculator/6cb10atg3l

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "{0},{1}".format(self.x, self.y)


points = [(Point(row.x, row.y)) for index, row in pd.read_csv("https://tinyurl.com/y2cocoo7").iterrows()]

best_likelihood = -10_000_000
b0 = .01
b1 = .01


# calculate maximum likelihood

def predict_probability(x):
    p = 1.0 / (1.0 + math.exp(-(b0 + b1 * x)))
    return p


for i in range(1_000_000):

    # Select b0 or b1 randomly, and adjust it randomly
    random_b = random.choice(range(2))

    random_adjust = np.random.normal()
```



https://tinyurl.com/y6ye8vzr

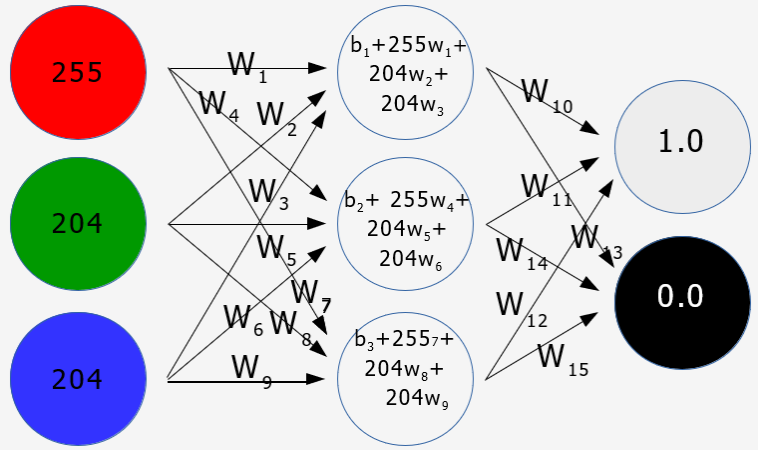
Hill Climbing and Fitting a Probability Distribution

```python
import random, math

observations = [1.0, 1.0, 1.0, 2.0, 2.0, 3.0, 4.0, 4.0, 5.0, 5.0, 5.0]


def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) * math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev ** 2)))


best_likelihood = 0.0
std_dev = 1.0
mean = 1.0

for i in range(100_000):
    adj = random.normalvariate(0, 1)
    selected_variable = random.randint(0, 1)

    if selected_variable == 0:
        mean += adj
    elif selected_variable == 1:
        std_dev += adj

    likelihood = math.exp(sum([math.log(.000000001 + normal_pdf(x, mean, std_dev)) for x in observations]))

    if likelihood > best_likelihood:
        best_likelihood = likelihood
    elif selected_variable == 0:
        mean -= adj
    elif selected_variable == 1:
        std_dev -= adj

print("mean={0}, std_dev={1}".format(mean, std_dev))
```



Simulated Annealing and Neural Networks

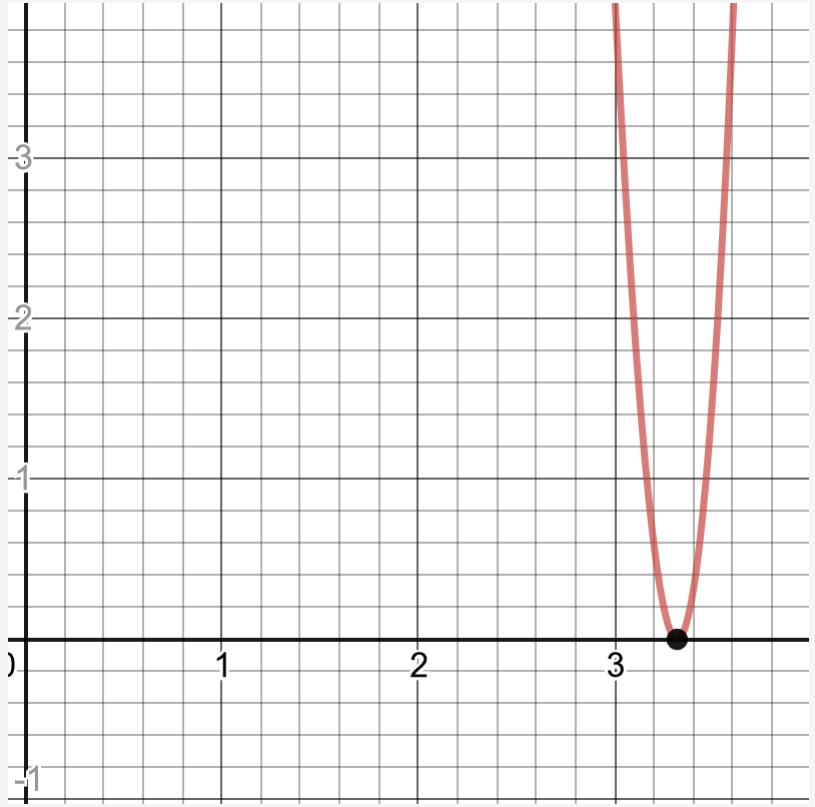
```
import numpy as np
import pandas as pd
from scipy import special

training_data = pd.read_csv("https://tinyurl.com/y2qmhfsr")
training_data_count = len(training_data.index)

# Learning rate controls how slowly we approach a solution
# Make it too small, it will take too long to run.
# Make it too big, it will likely overshoot and miss the solution.
learning_rate = 0.1

# Extract the input columns, scale down by 255
training_inputs = training_data.iloc[:, 0:3].values.transpose() / 255

# Extract output column, and generate an opposite column where 1 is 0 and 0 is 1.
actual_outputs = np.vstack(
    (training_data.iloc[:, -1].values.transpose(), -1 * (training_data.iloc[:, -1].values.transpose() - 1)))

# Build neural network with weights and biases
middle_weights = np.random.rand(3, 3)
output_weights = np.random.rand(2, 3)

middle_bias = np.random.rand(3, 1)
output_bias = np.random.rand(2, 1)

best_middle_weights = None
best_output_weights = None
best_middle_bias = None
best_output_bias = None


# Activation functions

def relu(x):
    return np.maximum(x, 0)

def softmax(x):
    return special.softmax(x, axis=0)
```



https://tinyurl.com/yxojew66

Hill Climbing to Find Square and N-Roots

```python
import numpy as np

# Use hill climbing to estimate a square root

value = 11.0
sqrt_candidate = value / 2.0

for i in range(100000):
    adjust = np.random.standard_t(3, 1)[0]
    new_candidate = sqrt_candidate + adjust

    if (value - sqrt_candidate ** 2) ** 2 > (value - new_candidate ** 2) ** 2:
        sqrt_candidate = new_candidate

print(sqrt_candidate) # prints 3.316622346827584
```



Other Metaheuristics Algorithms

Hill Climbing and Simulated Annealing are practical workhorses for a lot of optimization problems

There are many other metaheuristics algorithms, some of which you might have heard of:

- Genetic algorithms
- Tabu search
- Ant colony optimization
- Evolutionary algorithms
- K-Opt

Go down the rabbit hole: https://en.wikipedia.org/wiki/Metaheuristic

Optimization Algorithms Ranking

Algorithm	Scalable	Optimal	Easy to Use	Tweakable
Brute Force	0/5	5/5	5/5	0/5
Branch and Bound	0/5	5/5	4/5	2/5
Hill Climbing	5/5	2/5	4/5	3/5
Tabu Search	5/5	4/5	3/5	5/5
Simulated Annealing	5/5	4/5	2/5	5/5
Late Acceptance	5/5	4/5	3/5	5/5
Step Counting Hill Climbing	5/5	4/5	3/5	5/5
Variable Neighborhood Descent	3/5	3/5	2/5	5/5
Evolutionary Strategies	3/5	3/5	2/5	5/5
Genetic Strategies	3/5	3/5	2/5	5/5

SOURCE: https://docs.optaplanner.org/7.11.0.Final/optaplanner-docs/html_single/index.html#searchSpaceSize

Optimization Algorithms Ranking – Most Practical

Algorithm	Scalable	Optimal	Easy to Use	Tweakable
~~Brute Force~~	~~0/5~~	~~5/5~~	~~5/5~~	~~0/5~~
Branch and Bound	**0/5**	**5/5**	**4/5**	**2/5**
Hill Climbing	**5/5**	**2/5**	**4/5**	**3/5**
Tabu Search	**5/5**	**4/5**	**3/5**	**5/5**
Simulated Annealing	**5/5**	**4/5**	**2/5**	**5/5**
Late Acceptance	**5/5**	**4/5**	**3/5**	**5/5**
Step Counting Hill Climbing	**5/5**	**4/5**	**3/5**	**5/5**
~~Variable Neighborhood Descent~~	~~3/5~~	~~3/5~~	~~2/5~~	~~5/5~~
~~Evolutionary Strategies~~	~~3/5~~	~~3/5~~	~~2/5~~	~~5/5~~
~~Genetic Strategies~~	~~3/5~~	~~3/5~~	~~2/5~~	~~5/5~~

SOURCE: https://docs.optaplanner.org/7.11.0.Final/optaplanner-docs/html_single/index.html#searchSpaceSize

Pros and Cons of Metaheuristics

Pros	Cons
• Performs relatively quickly for large problems • Works with limited computational resources • Relatively easy to implement and explain, even to nontechnical people • Can be used when information about the model is incomplete or imperfect • If you're better at programming than you are at math, metaheuristics are for you!	• Does not guarantee an optimal solution, only an "acceptable" one. • Does not guarantee a deterministic solution, and can produce different results with each execution. • Like most models, it will require tinkering and experimentation with hyperparameters before it provides an acceptable answer *(no free lunch theorem*).

No Free Lunch Theorem

In machine learning and optimization, an unfortunate reality is you will need to customize models to every problem.

- A given temperature schedule can work well with one Traveling Salesman Problem, but not as well with another.

- A set of weights for one deep learning problem recognizing cats will not work with another recognizing dogs.

This is know as the no free lunch theorem, and because of it you should rarely expect a silver bullet implementation of any algorithm. You will always need to experiment with parameters and manually adapt to new situations.

https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization

P Versus NP Problem

Another problem to be aware of in optimization and machine learning is the P versus NP problem.

- If P = NP, that would mean solutions to hard problems (like optimization and machine learning) can be calculated just as quickly as they are verified.
- Think of Sudokus: it's quick to verify if a Sudoku solution is valid, but it takes much longer to solve for a solution.

While P = NP has been neither proven or disproven in the past 50 years, more scientists are coming to believe P does not equal NP.

This is a major limitation to the capabilities of machine learning and optimization, as it means complexity will always limit what we can do.

YouTube – The P versus NP Problem


The Traveling Salesman (2012 Thriller Film)

“Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line.”

— Benoît Mandelbrot

Quiz Time!

Metaheuristics algorithms (like hill climbing and simulated annealing) guarantee the most optimal solution.

A True

B False

Quiz Time!

Metaheuristics algorithms (like hill climbing and simulated annealing) guarantee the most optimal solution.

A True

B False

Metaheuristics algorithms are random-based searches, and can only guarantee an approximation if executed correctly.

Quiz Time!

You used Hill Climbing to find a solution to the Traveling Salesman Problem. However, you noticed that when it finished, many overlaps still exist indicating a poor solution. Which of these is the more likely problem?

A There were **too many** iterations

B There were **not enough** iterations

Quiz Time!

You used Hill Climbing to find a solution to the Traveling Salesman Problem. However, you noticed that when it finished, many overlaps still exist indicating a poor solution. Which of these is the more likely problem?

A There were **too many** iterations

B There were **not enough** iterations

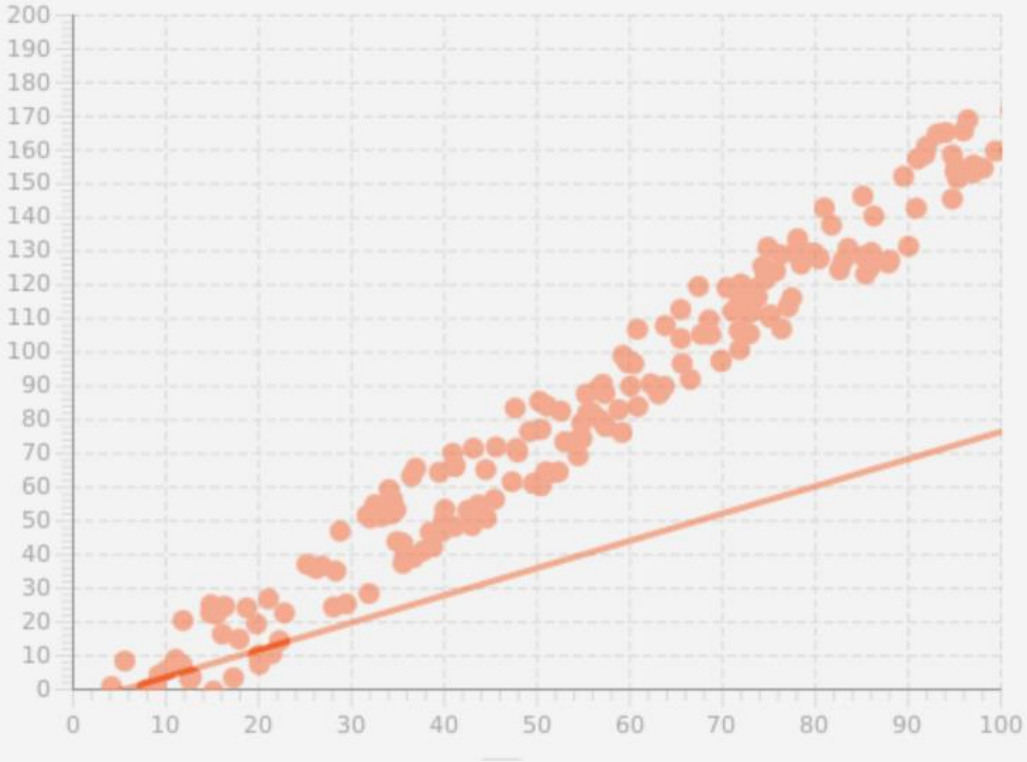
Hill-climbing should converge on a local minimum where overlaps do not exist anymore. If there are still overlaps, it means the algorithm did not do enough random samples and swaps. Therefore it needs more iterations.

Quiz Time!

Hill climbing was used for a linear regression. However, starting from a flat position, the line has stopped well below the points and clearly did not fit to them. Which of these could be the more likely cause?

A There were **too many** iterations

B There were **not enough** iterations

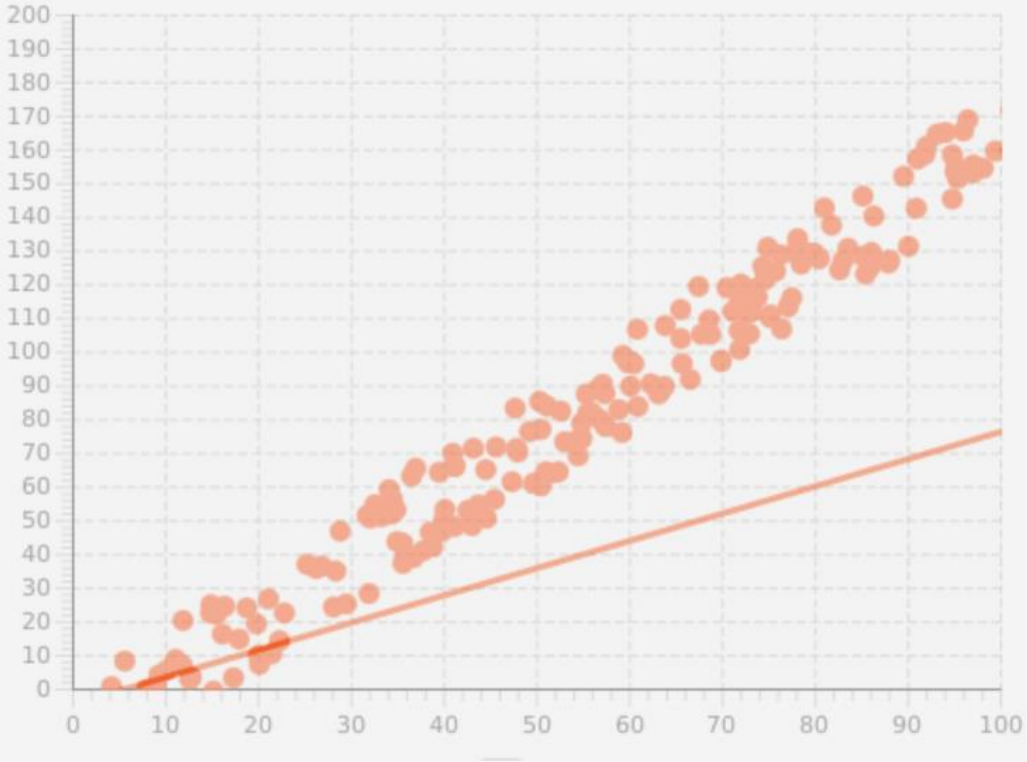


Quiz Time!

Hill climbing was used for a linear regression. However, starting from a flat position, the line has stopped well below the points and clearly did not fit to them. Which of these could be the more likely cause?

A There were **too many** iterations

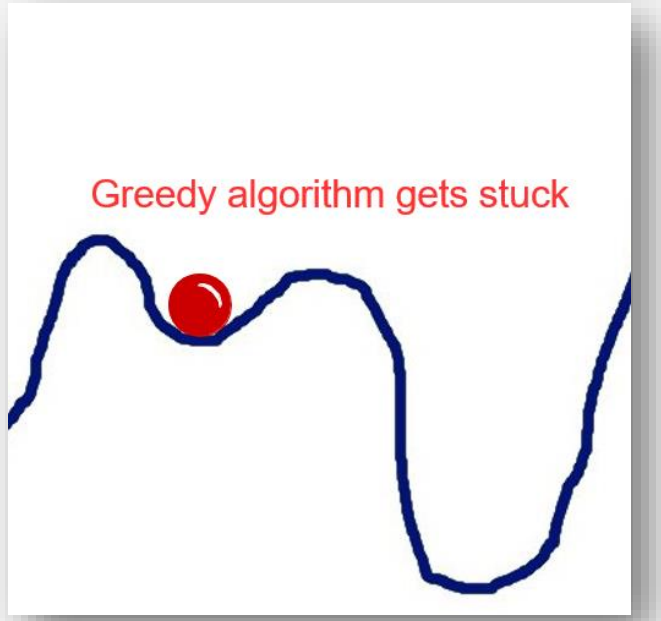
B There were **not enough** iterations



Again, hill climbing should converge on a local minimum. If the line started flat and prematurely stopped moving towards the points, it is likely it did not have enough iterations.

Quiz Time!

You are using simulated annealing to minimize cost of a factory line schedule. You know it is possible to achieve a cost of $20K or less, but your search algorithm consistently gets stuck around $21K. Which of these should you likely try next?



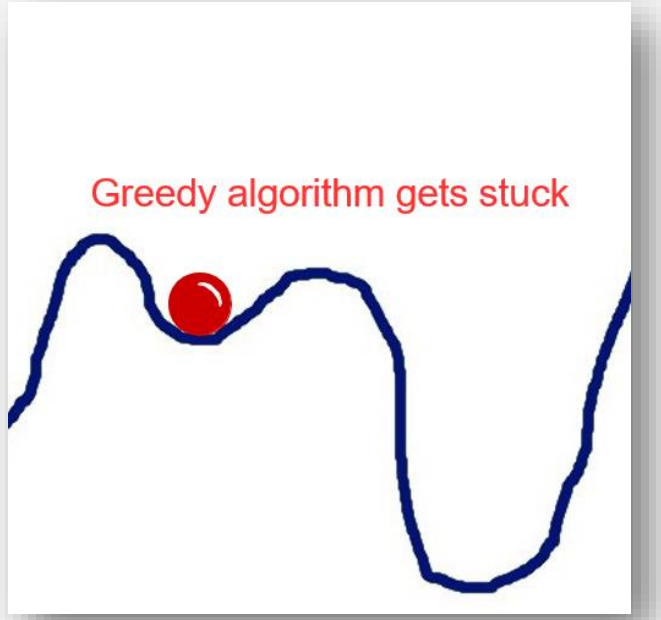
A. Switch from a Simulated Annealing to a Hill Climbing algorithm

B. Use a temperature schedule that periodically allows more randomness

C. Increase the number of iterations

Quiz Time!

You are using simulated annealing to minimize cost of a factory line schedule. You know it is possible to achieve a cost of $20K or less, but your search algorithm consistently gets stuck around $21K. Which of these should you likely try next?



A. Switch from a Simulated Annealing to a Hill Climbing algorithm

B. Use a temperature schedule that periodically allows more randomness

C. Increase the number of iterations

We are already using simulated annealing, but are stuck in a local minimum so our temperature schedule needs to be hotter. You need to introduce some more randomness, and hopefully that will “shake the ball” out of the local minimum it keeps getting stuck in.

Source Code

Traveling Salesman Demo

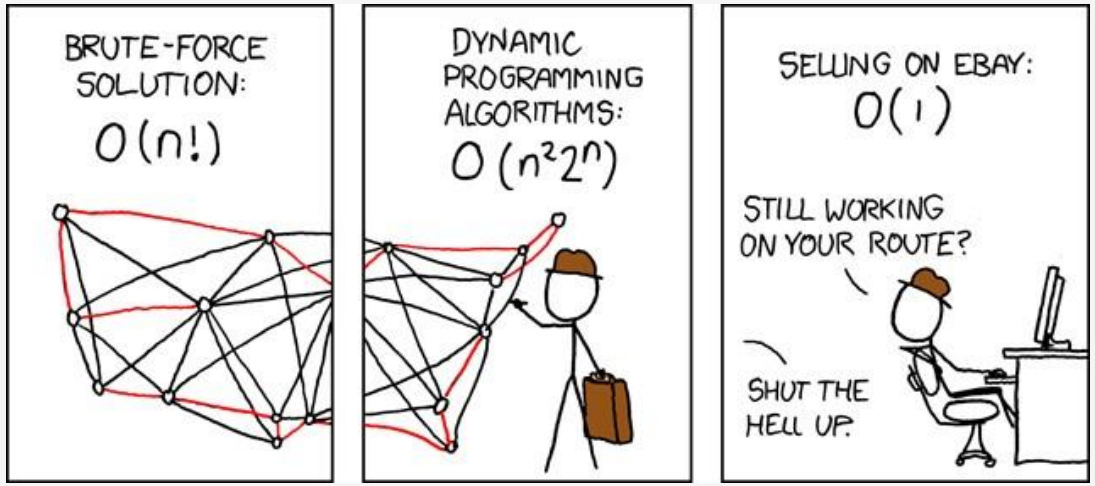
https://github.com/thomasnield/traveling_salesman_demo

Traveling Salesman Plotter

https://github.com/thomasnield/traveling_salesman_plotter

Linear Regression App

https://github.com/thomasnield/kotlin_linear_regression/



SOURCE: xkcd.com

Section III
Tree Search

Introducing Tree Search

Stochastic search algorithms, which rely on controlled randomness, can be effective and useful although they have drawbacks.

- They are not deterministic and will likely provide an approximate and inconsistent solution every time it is executed.
- This compromise is why they perform fast and with little computing resources.

Introducing Tree Search

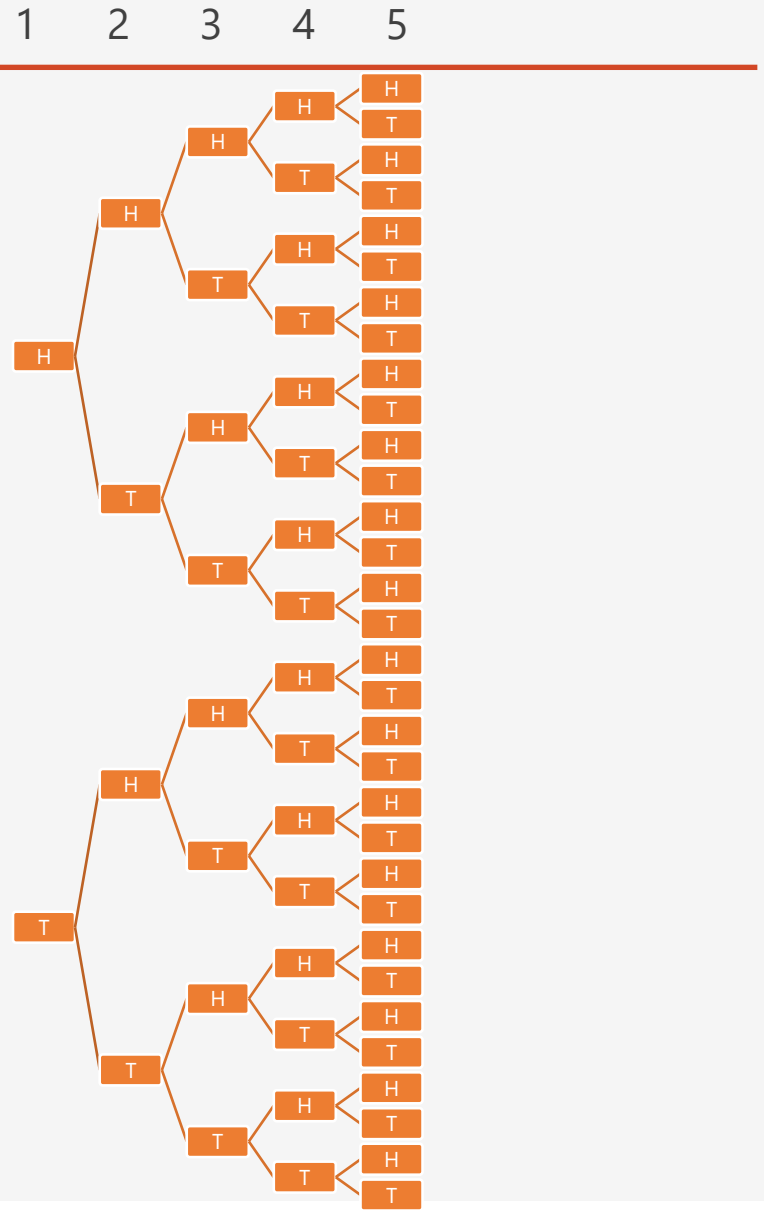
Tree search is another search algorithm that is consistent, deterministic, and effectively exhaustive of all possible solutions but also has drawbacks.

- The algorithm will explore every possible combination of solutions, minus the ones it knows will be infeasible.
- This clever avoidance of infeasible solutions is done with a combination of heuristics, bounding, and pruning.
- However, they can be demanding on computing resources and, in extreme cases, be downright impractical.

A Crash Course on Permutations

Suppose you had 5 coin flips, and you want to know all ordered combinations that yield 3 heads and 2 tails, such as HHHTT, HHTHT, TTHHH, and so on.

You can use a recursive function to generate a tree of all possible combinations, and keep the ones that include 3 heads and two tails.



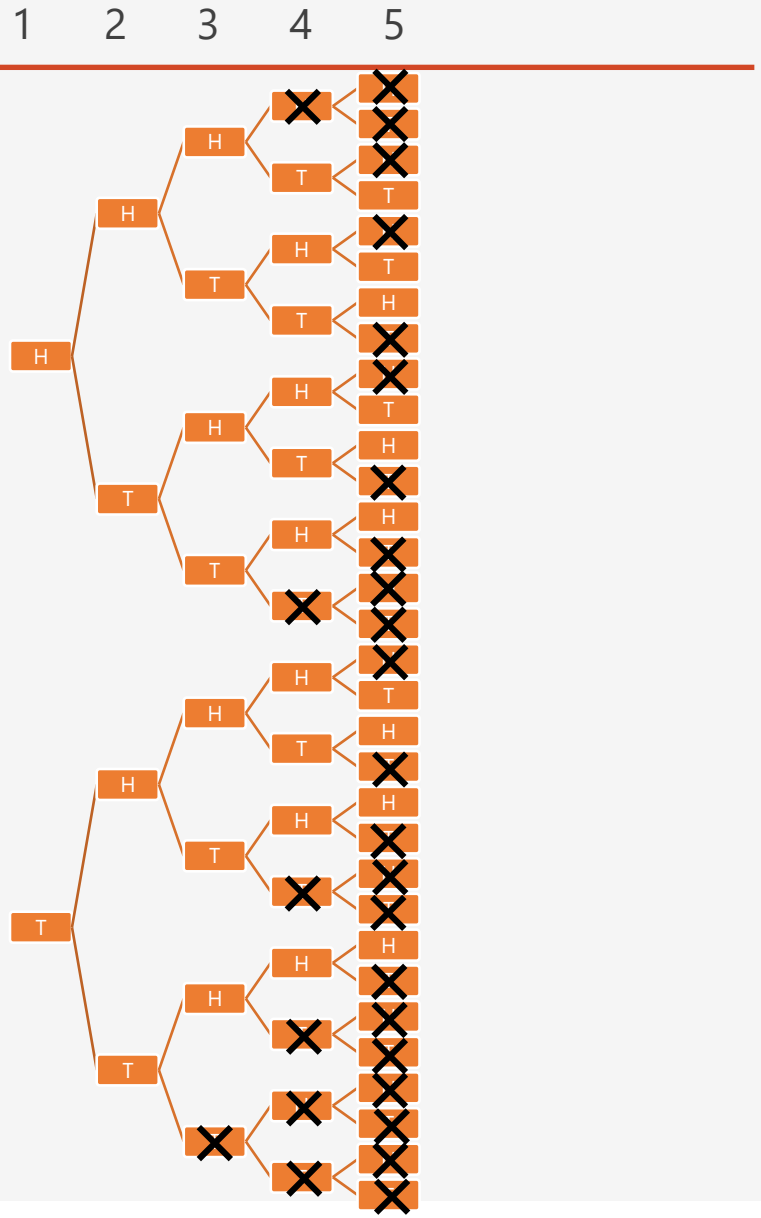
A Crash Course on Permutations

Suppose you had 5 coin flips, and you want to know all ordered combinations that yield 3 heads and 2 tails, such as HHHTT, HHTHT, TTHH, and so on.

You can use a recursive function to generate a tree of all possible combinations, and keep the ones that include 3 heads and two tails.

However, if we have rules dictating specific outcomes, we can search this tree faster by avoiding combinations we know will not work.

Fun Tangent: The Binomial Distribution
https://www.youtube.com/watch?v=ConmIDAzRqI



Consider the Sudoku

The Sudoku is hopefully a familiar game puzzle.

OBJECTIVE: Fill in the blank cells with numbers 1-9 so that every row, column, and square has the numbers 1-9.

We could use simulated annealing to solve this: *start with a random solution and randomly swap numbers, minimizing the count of "infractions" until it is 0.*

However, a tree search is going to make more sense here since Sudoku constraints can be leveraged to reduce the search space quickly.

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

Consider the Sudoku

[4,4] →5
[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
[2,6] →1, 7
[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

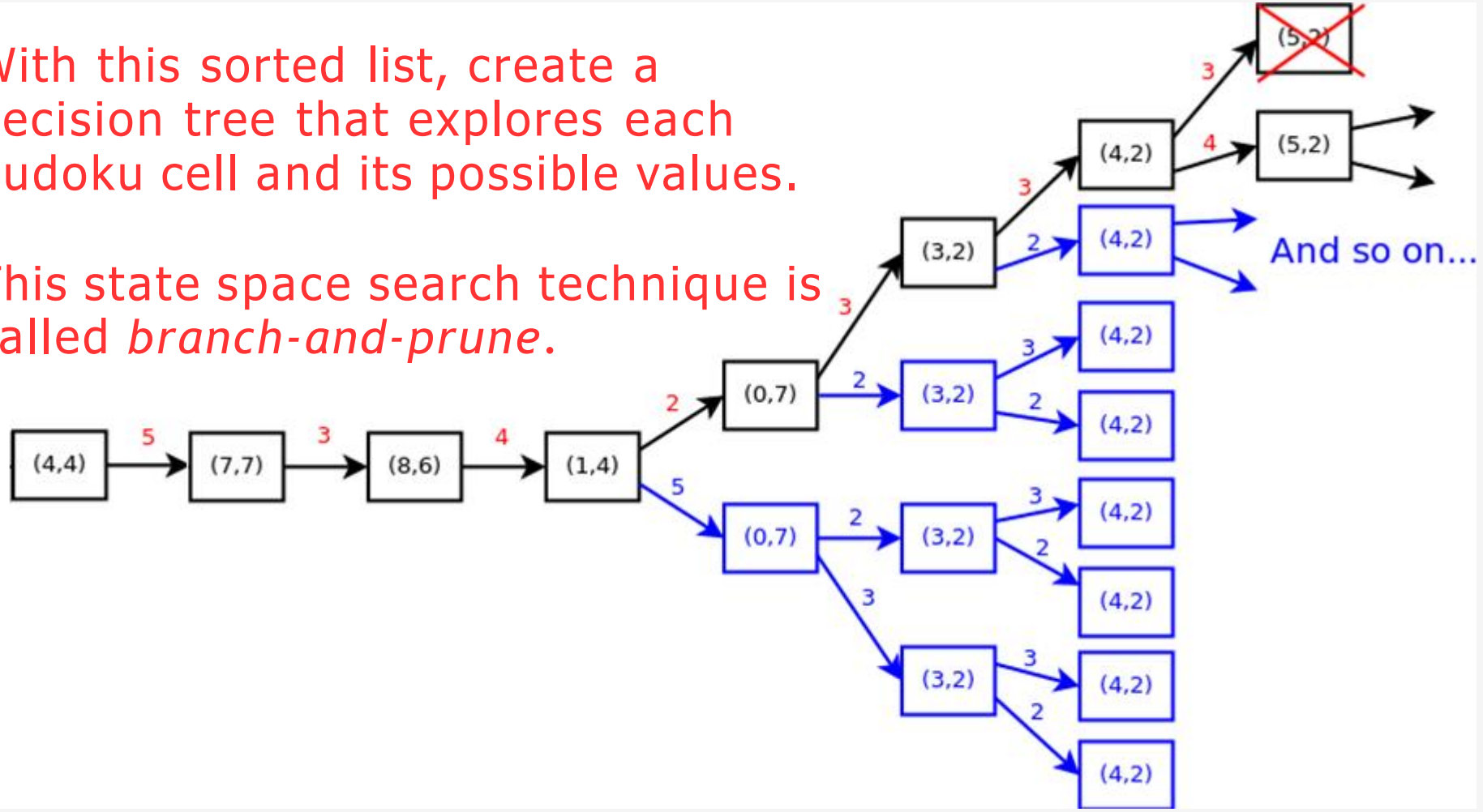
	0	1	2	3	4	5	6	7	8
0	5	3			7				
1	6			1	9	5			
2		9	8					6	
3	8				6				3
4	4			8		3			1
5	7				2				6
6		6					2	8	
7				4	1	9			5
8					8			7	9

Solving the Sudoku

[4,4] →5
[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
[2,6] →1, 7
[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 state space search technique is called *branch-and-prune*.



Solving the 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

Building a Tree Search Algorithm

You will need to get comfortable with recursive functions!

1. Sort variables by a good heuristic (e.g. Sudoku cells by their candidate value count)

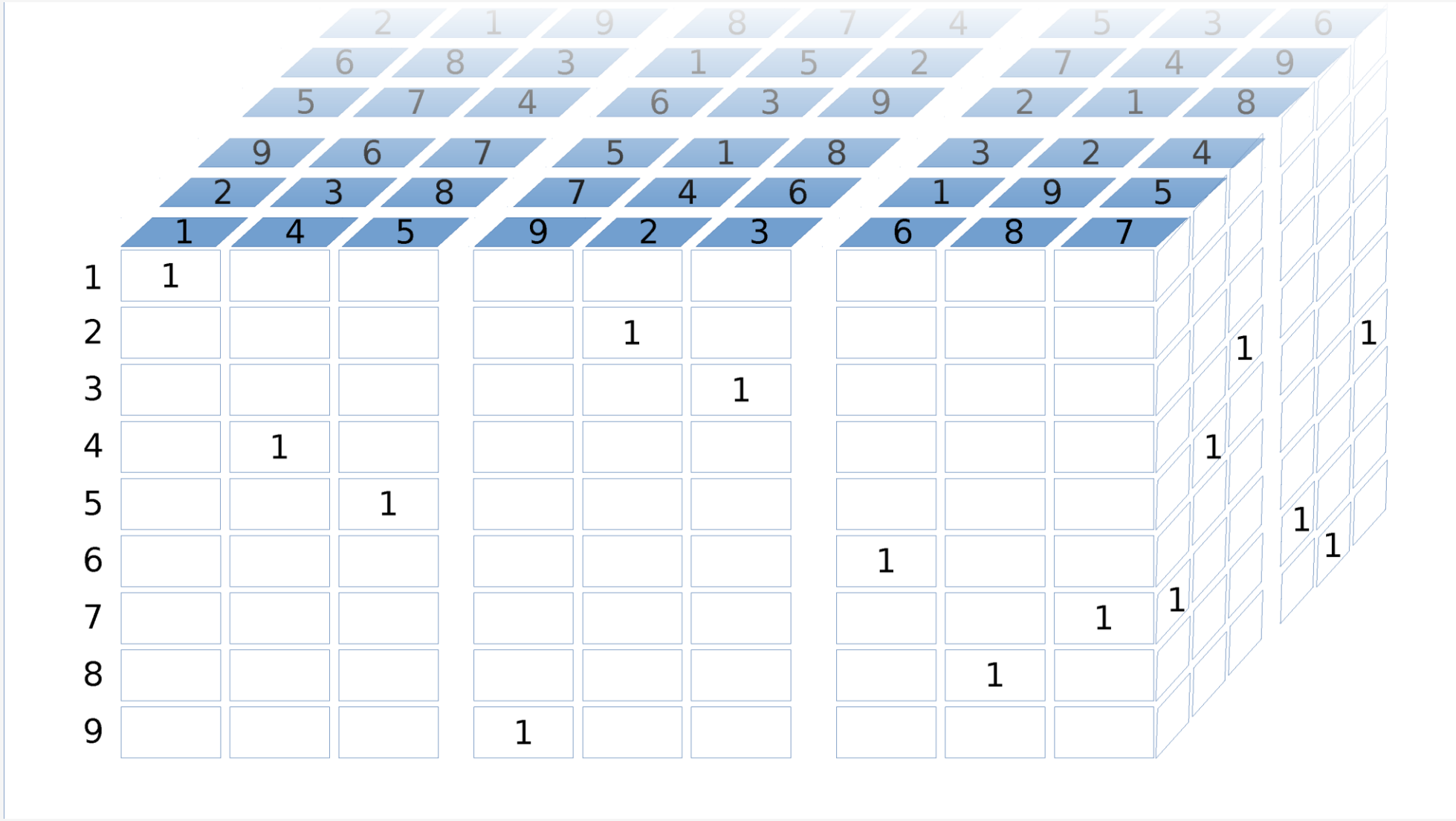
2. Start at the first variable, and explore every combination of values for all the other variables in a tree-like fashion.

   A. On each “node”, check to see if the current branch is still feasible and not breaking problem constraints.

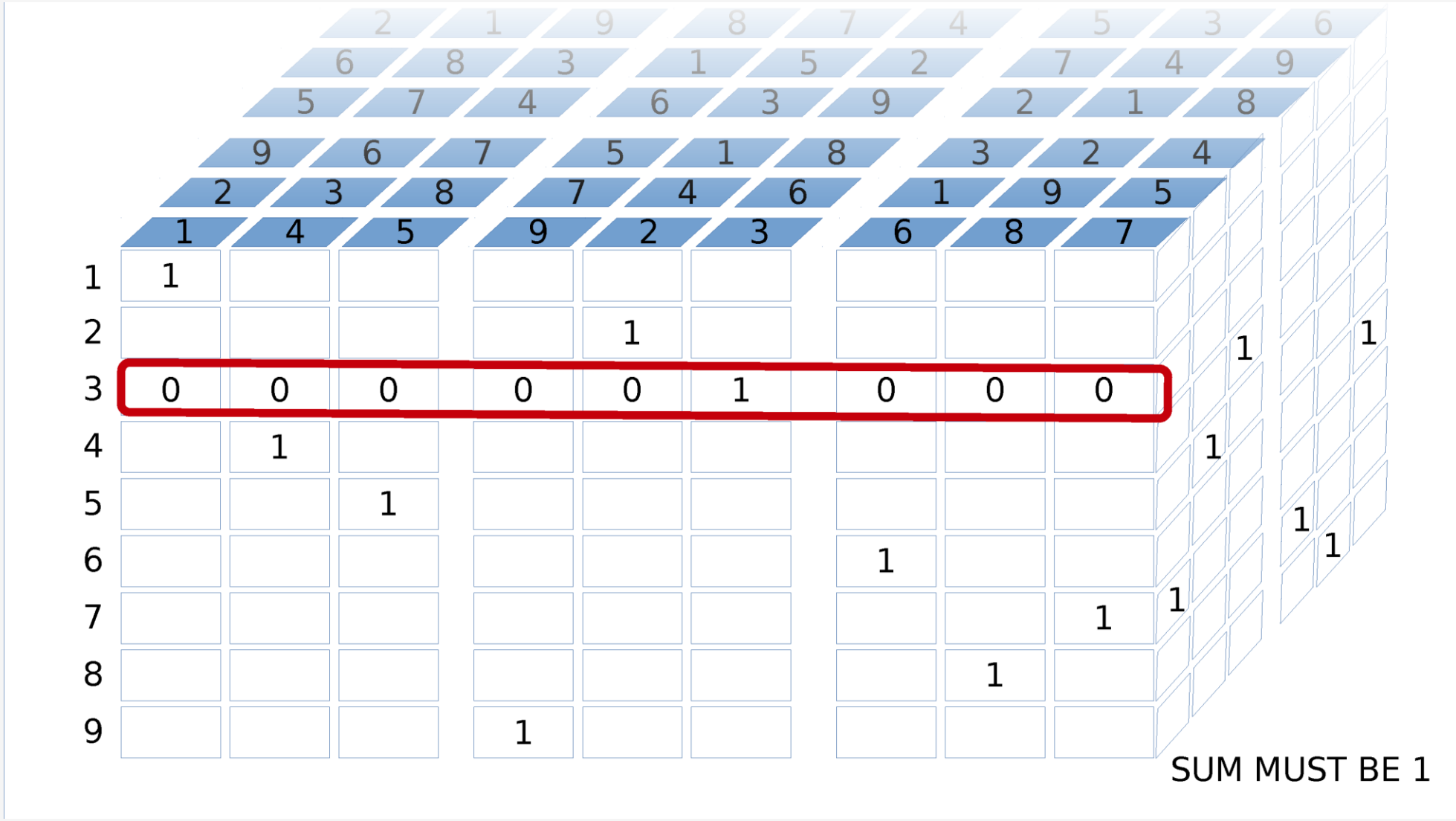
   B. If an infeasibility is detected, “prune” and cancel that branch and move onto the next one.

3. Proceed until a feasible solution is found, then stop the search. If no solution is found, report that no solution exists.

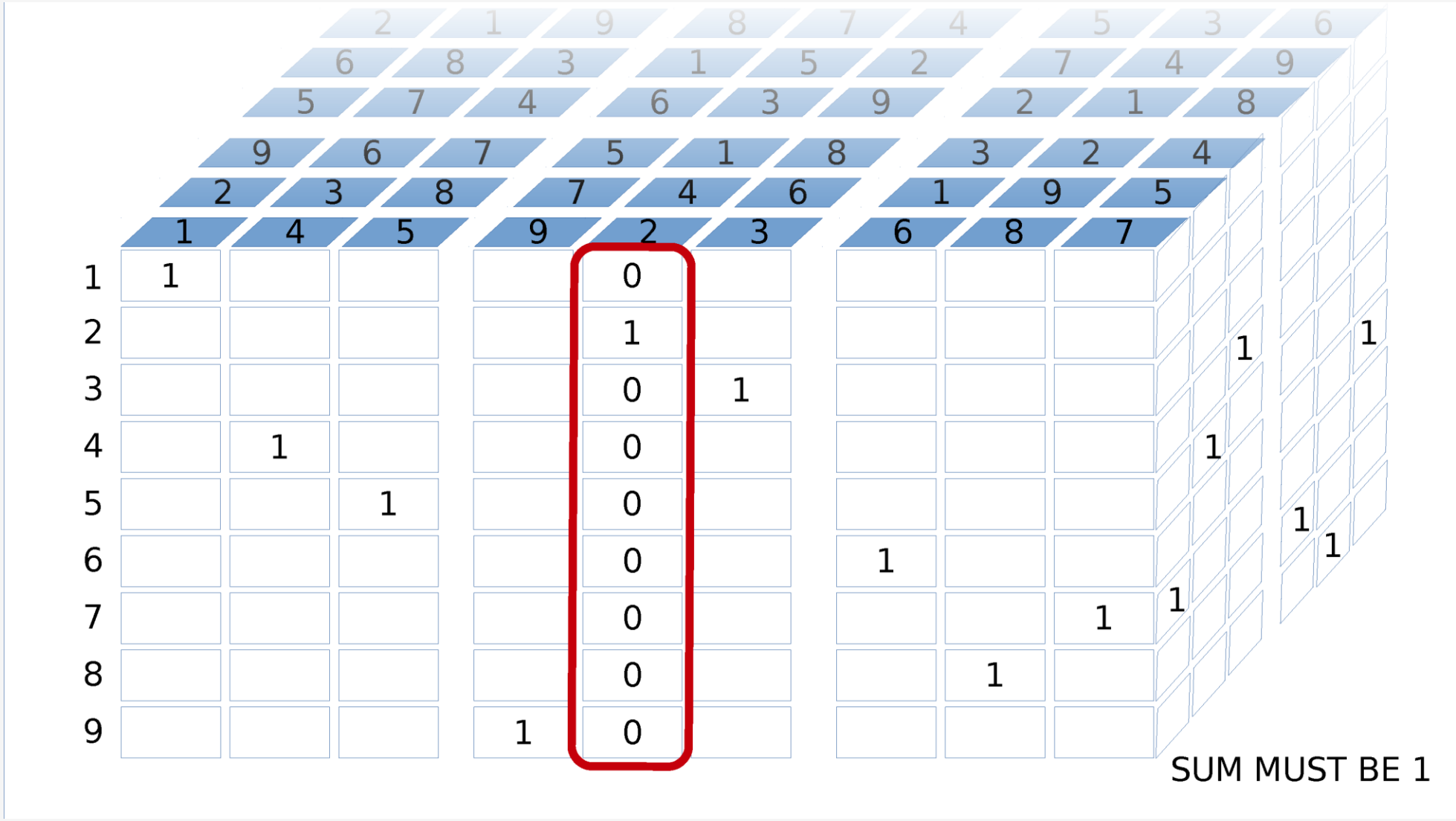
ALTERNATIVE: Binary Model for Sudokus



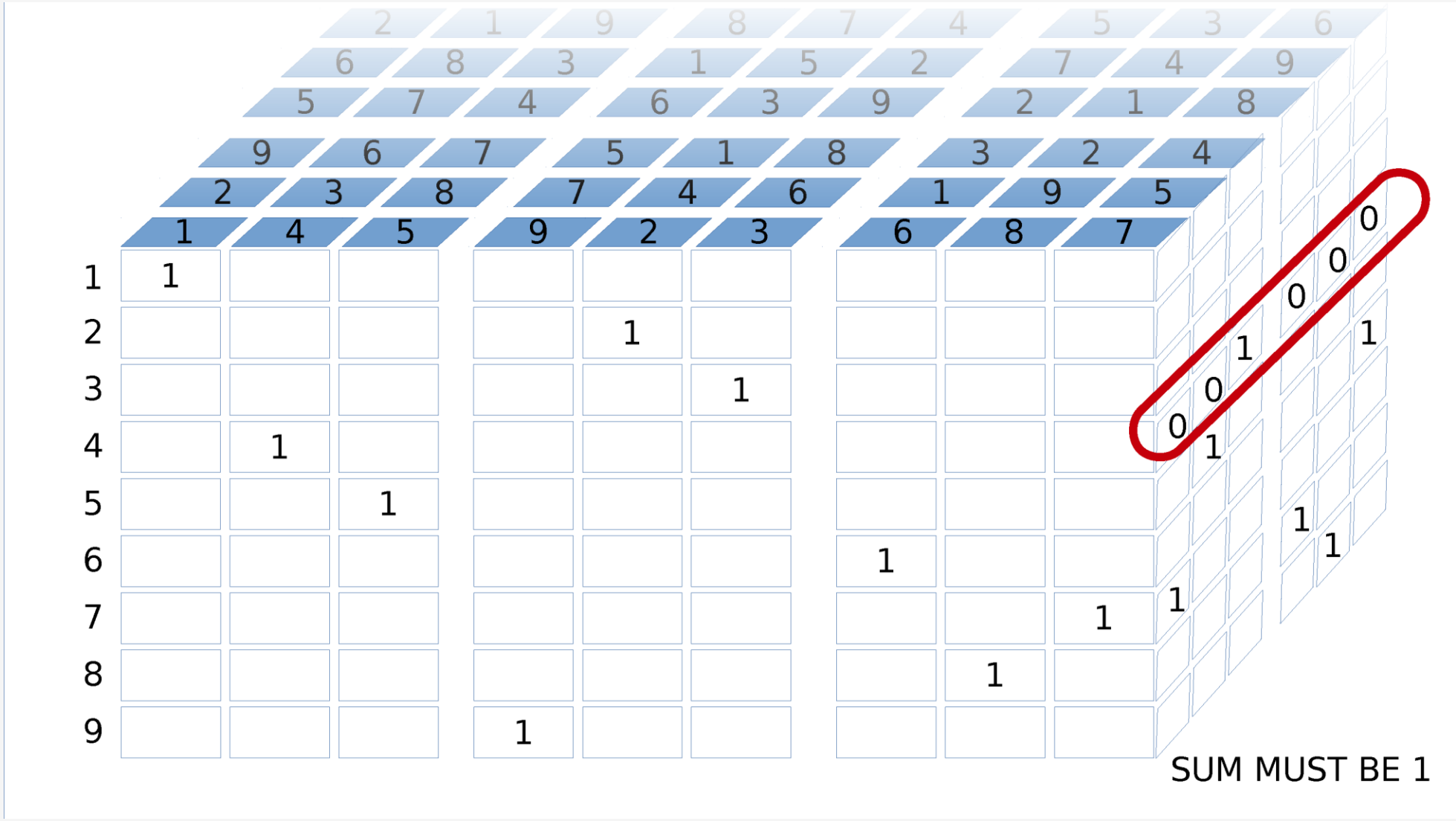
ALTERNATIVE: Binary Model for Sudokus



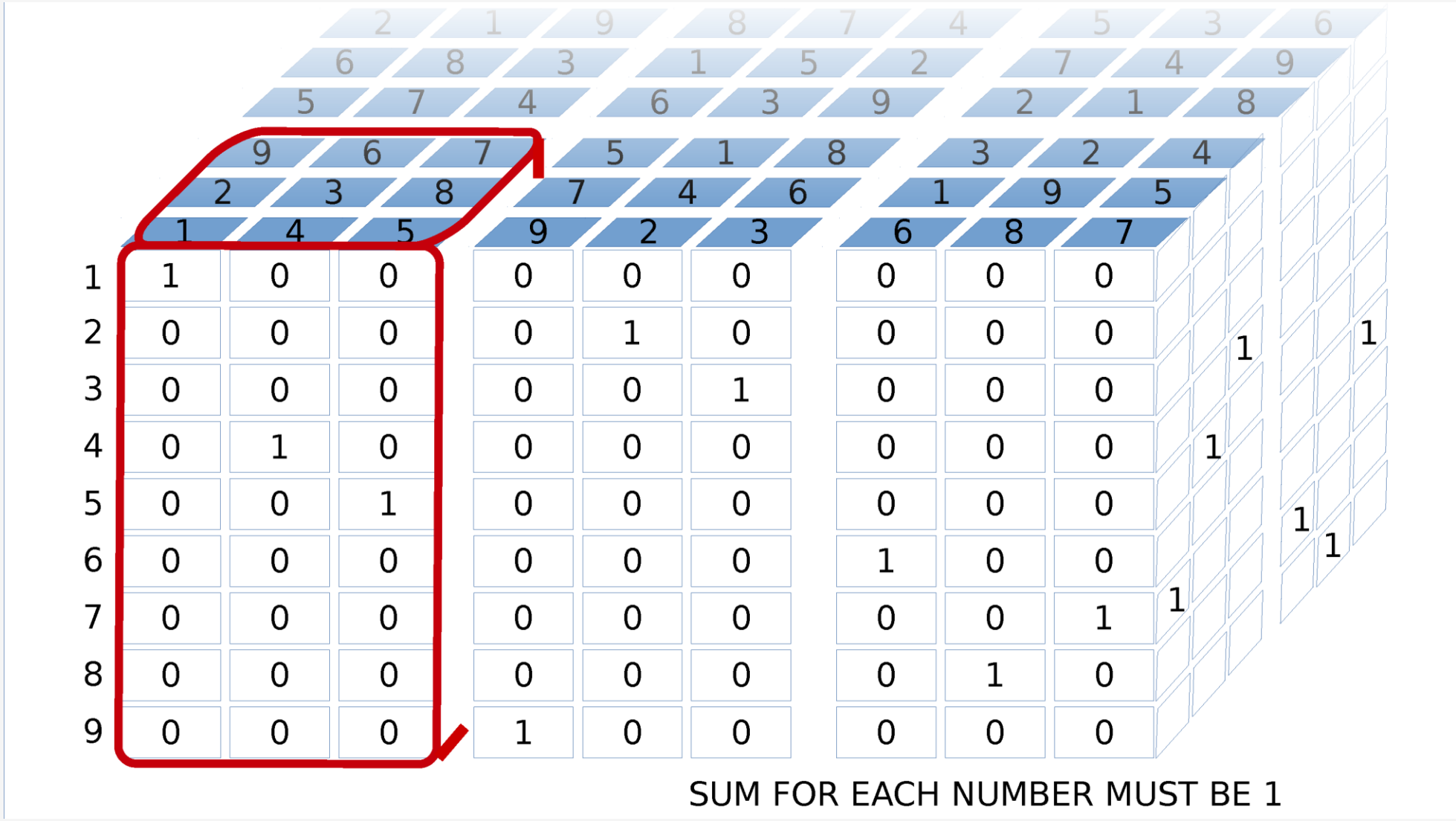
ALTERNATIVE: Binary Model for Sudokus



ALTERNATIVE: Binary Model for Sudokus



ALTERNATIVE: Binary Model for Sudokus



ALTERNATIVE: Binary Model for Sudokus

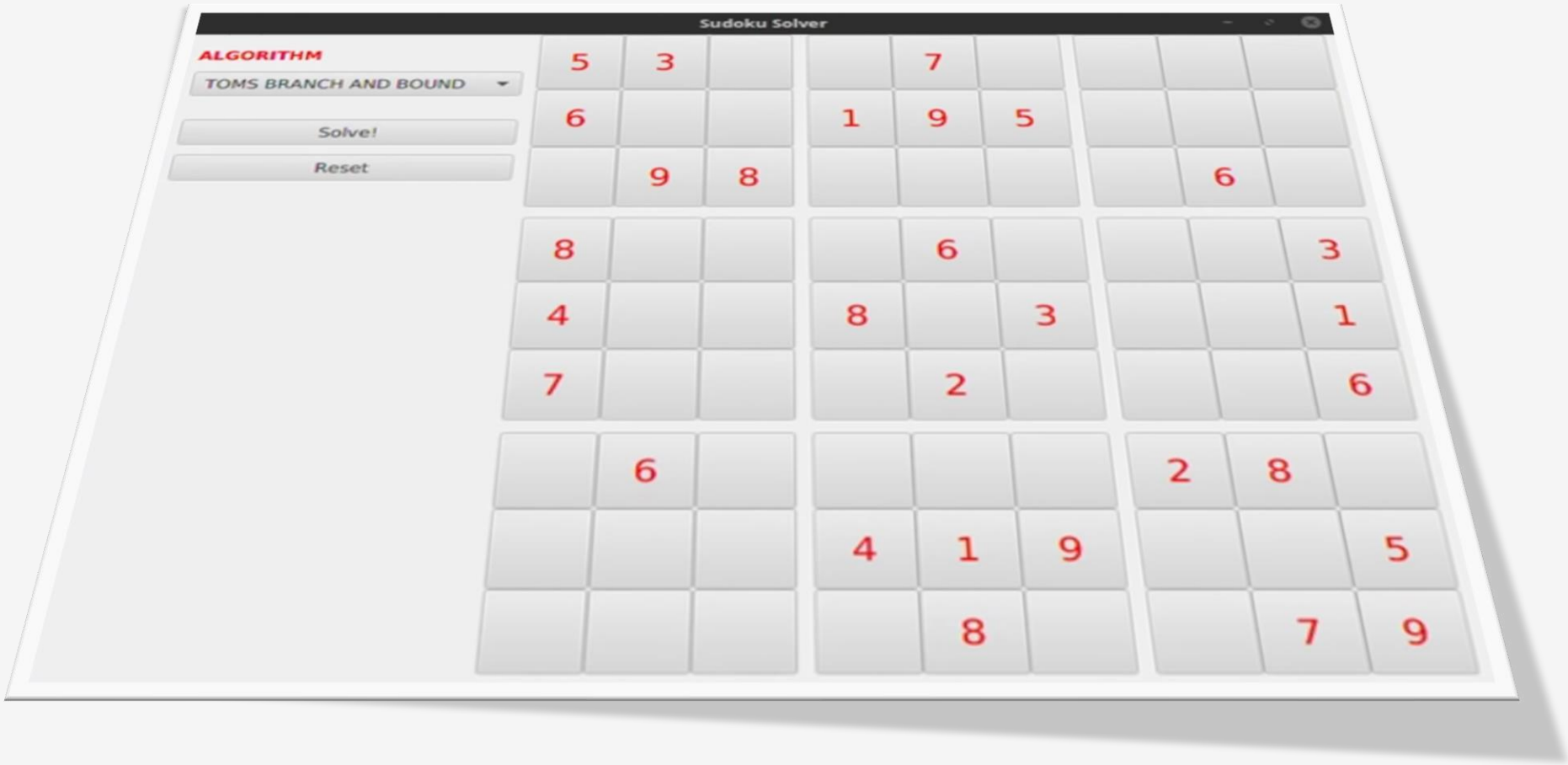
We could create a solver from scratch, but a solver library like ojAlgo or OptaPlanner will make life easier.

- We just model our problem and define the constraints.
- The solver library will find the values for the variables.

The Sudoku model is a great example of binary variables constraining occupation to one resource.

- Only one “number” can occupy a square.
- This concept can apply to real-life, such as only one class can occupy a classroom at a given time.

Demo: Solving a Sudoku



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		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		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 <=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		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 <=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		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 <=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		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	MON	MON	MON	MON	MON	MON	MON	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
English 101	...	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
Psych 300	...	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
Linear Algebra I	...	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
Biology 101	...	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
Orientation 101	...	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.

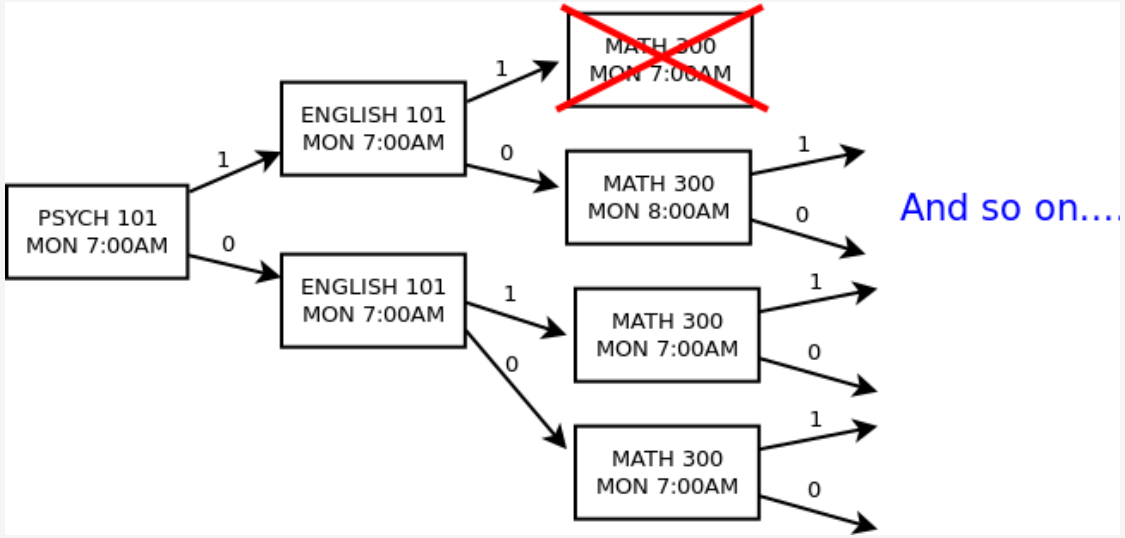
View image here.

Branch-and-Prune for Scheduling

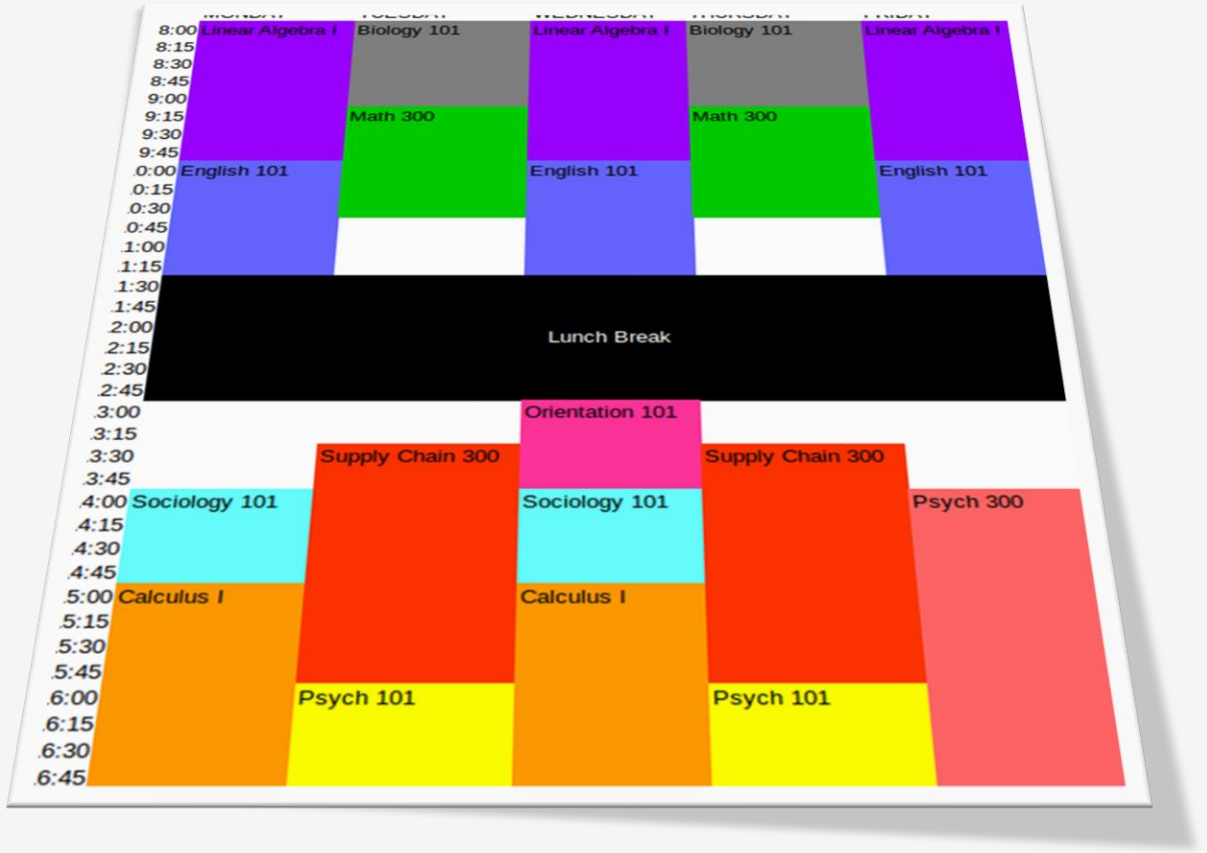
You could solve the scheduling problem from scratch with branch-and-prune.

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.



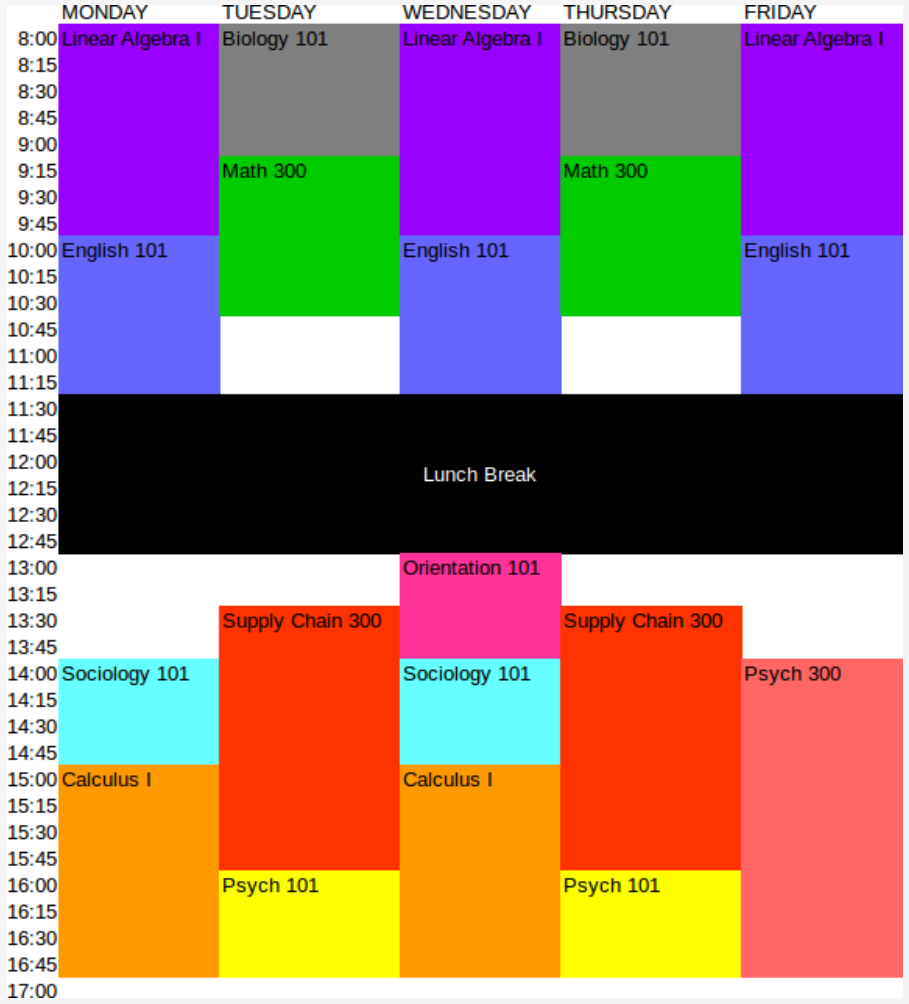
Demo: Generating a Classroom Schedule



Generating a Schedule

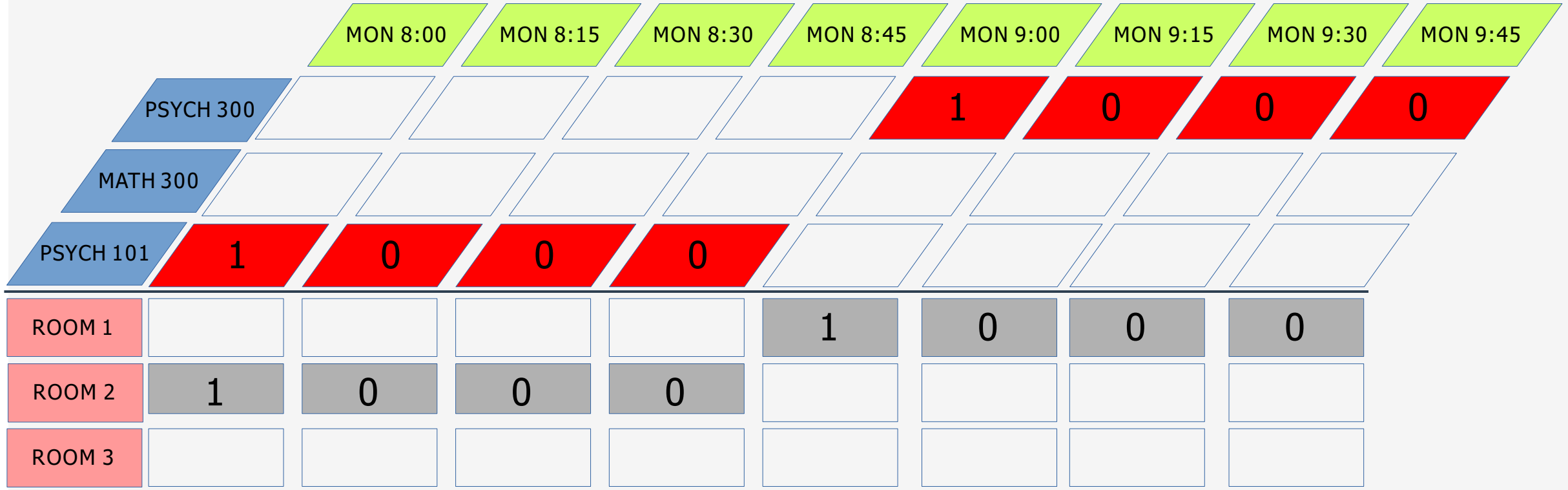
Plug these variables and *feasible* constraints into the optimizer or a tree search algorithm (which I'll show next), and you will get a solution.

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.



Linear Relaxation with Branch and Bound

When you are doing a minimization/maximization objective with tree search, you might consider using branch-and-bound to prune your search tree more aggressively.

Say you have a shipping container that has a capacity of 24, and you have the following items you want to ship:

- Order A – Requires a capacity of 5, and produces revenue of $10K.
- Order B – Requires a capacity of 10, produces revenue of $50K
- Order C – Requires a capacity of 3, produces revenue of $30K
- Order D – Requires a capacity of 12, produces revenue of $40K

What is the maximum revenue we can achieve with only so much capacity?

Linear Relaxation with Branch and Bound

This type of problem is known as the knapsack problem, and you have to make a decision whether to take an item (1) or don't take it (0) towards an optimization goal.

Let's declare four binary variables, representing whether (1) or not (0) to take that item:

x_A = Order A

x_B = Order B

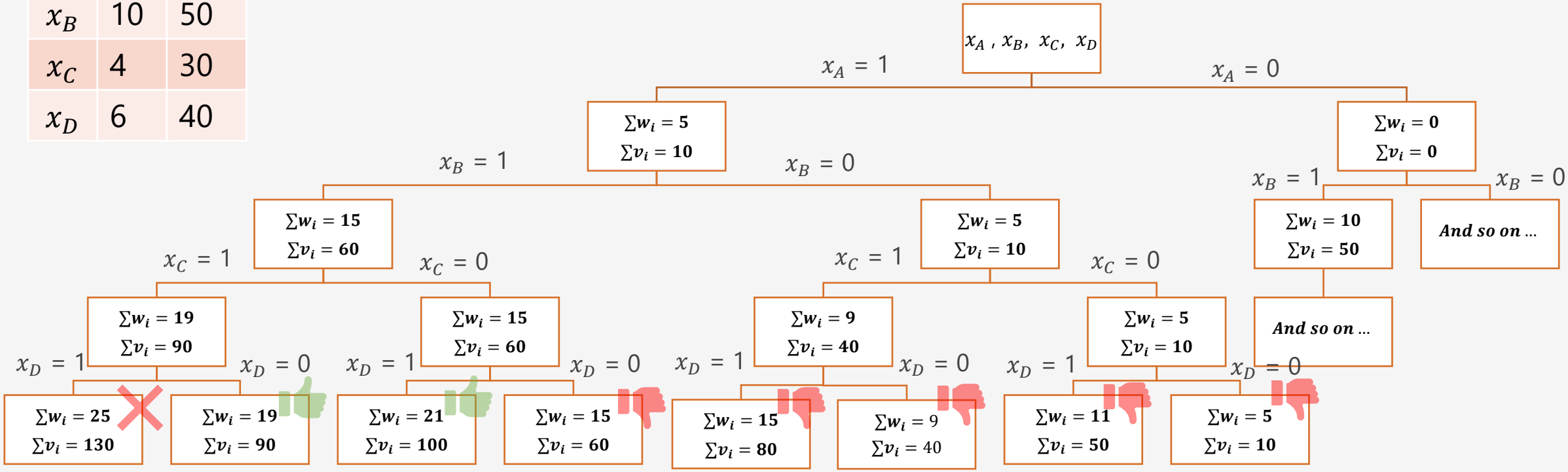
x_C = Order C

x_D = Order D

Knapsack Basic Search

x_i	w_i	v_i
x_A	5	10
x_B	10	50
x_C	4	30
x_D	6	40

$k = 24$

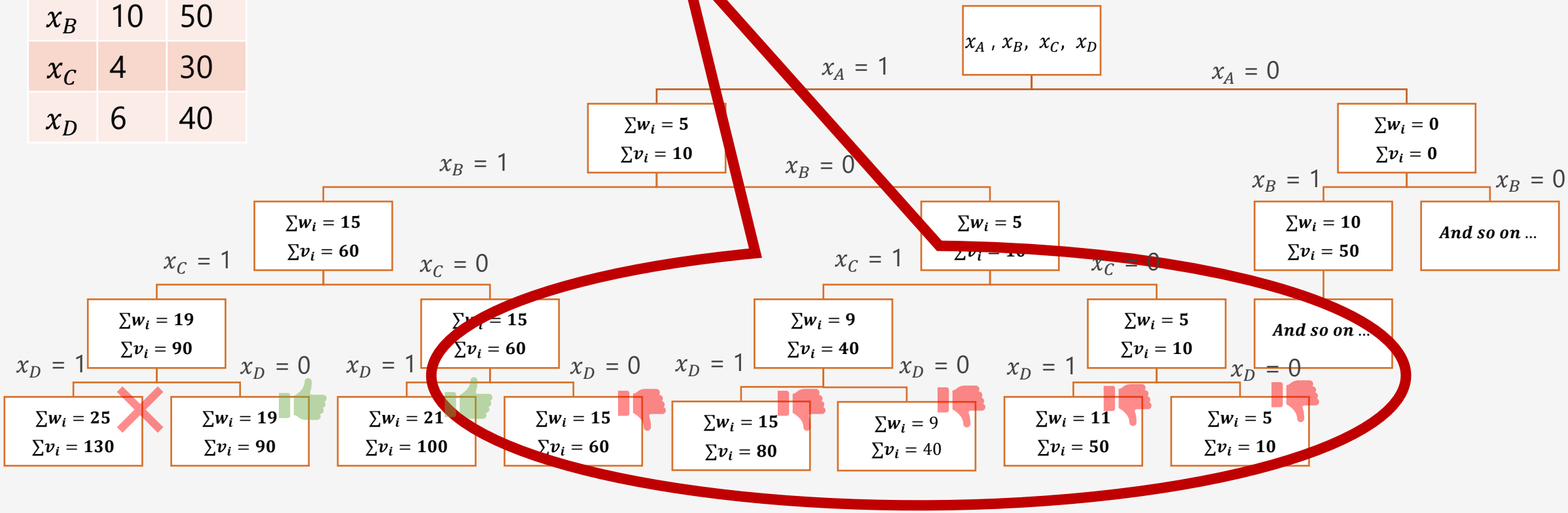


Knapsack Basic Search

x_i	w_i	v_i
x_A	5	10
x_B	10	50
x_C	4	30
x_D	6	40

$k = 24$

This is so inefficient. We found a solution and then keep exploring bad solutions. Is there a way to prune and avoid all this?



Linear Relaxation with Branch and Bound

Yes! We can prune our search tree further with **linear relaxation**, a.k.a. **bounding**.

Bounding means we calculate the **minimum achievable value** at each node, and prune branches where that value is inferior to the best found value.

To implement, let's first use a heuristic that ratios the value to weight of each item, and sorts them starting with highest value items first.

Item	Weight	Value (in thousands $)	Value/Weight Ratio
Order C	4	$30	$30/4 = $7.50
Order D	6	$40	$40/6 = $6.66
Order B	10	$50	$50/10 = $5
Order A	5	$10	$10/5 = $2

Linear Relaxation with Branch and Bound

Next, let's see how many items we can fit with our capacity of 24, starting with the highest value/weight ratio item first.

Then we take a fraction of the last item that exceeds the capacity so it matches capacity, and multiply that fraction against that item's value to get a relaxed achievable value.

Item	Weight	Value (in thousands $)	Value/Weight Ratio	Relaxed Weight	Relaxed Value
Order C	4	$30	$30/4 = $7.50	4	$30
Order D	6	$40	$40/6 = $6.66	4 + 6 = 10	$30 + $40 = $70
Order B	10	$50	$50/10 = $5	4 + 6 + 10 = 20	$30 + $40 + $50 = $120
Order A	5	$10	$10/5 = $2	~~4 + 6 + 10 + 5 = 25~~ 4 + 6 + 10 + (24-20) = 24	~~$30 + $40 + $50 + $10 = $130~~ $30 + $40 + $50 + $\left(\frac{24-20}{5}\right)$ ($10) = $128

This is the max we can ever achieve

Knapsack Branch-and-Bound Search

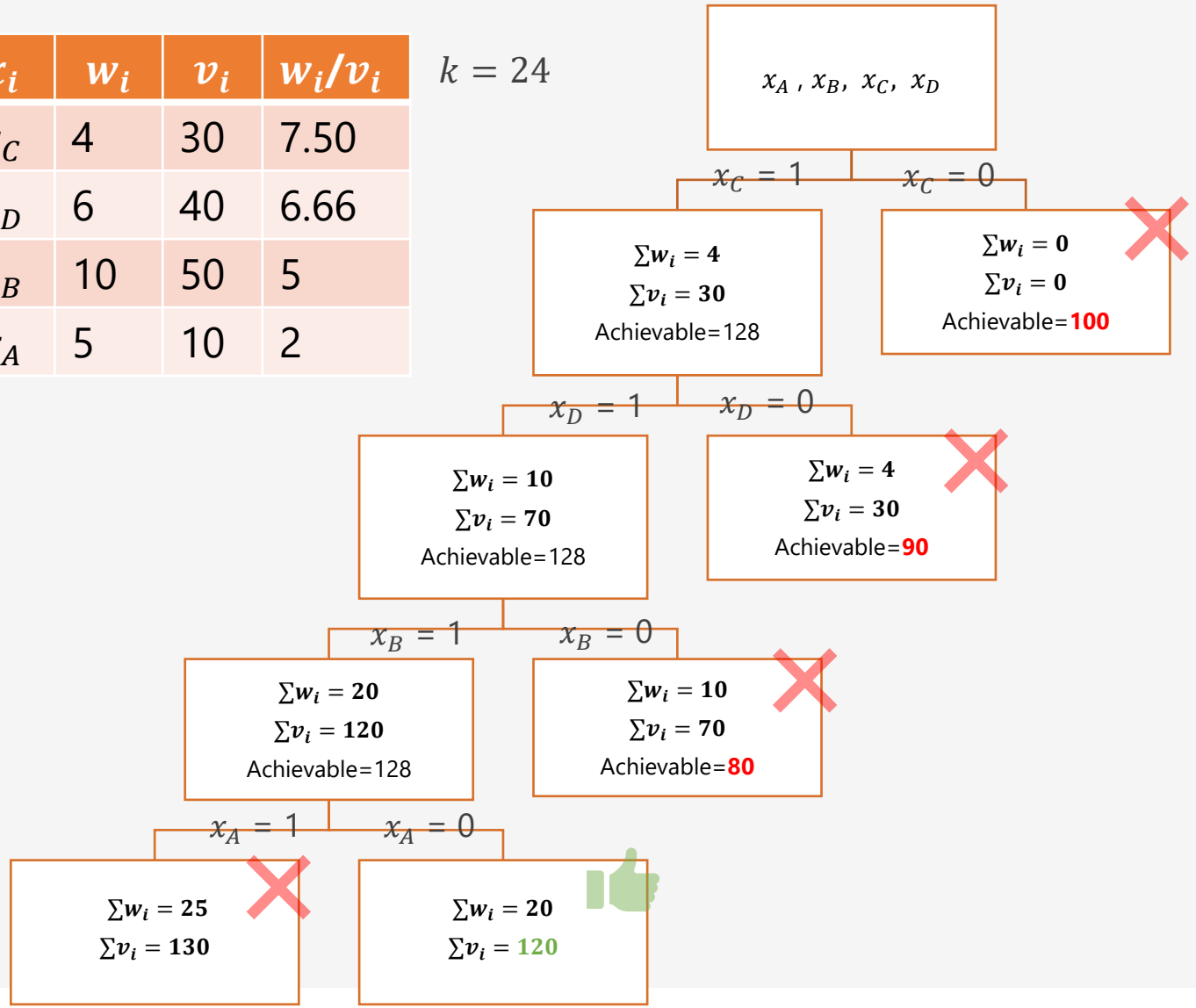
Finally, let's create a new search tree exploring items in this heuristic order.

Along the way, we will calculate the **relaxed bounded value** at each node to see what is achievable. If that value is inferior the best solution found, we prune and terminate that branch.

Using this technique, look how much more efficiently our search turned out on the right!

x_i	w_i	v_i	w_i/v_i
x_C	4	30	7.50
x_D	6	40	6.66
x_B	10	50	5
x_A	5	10	2

$k = 24$



Solving with PuLP

You can create this branch-and-bound search tree from scratch, but most people will use libraries like PuLP, PyOmo, or ojAlgo.

The Python code to the right solves this knapsack problem using PuLP.

These libraries also allow dropping in licensed solvers that can improve performance for larger problems, like Gurobi or IBM's CPLEX which costs over $10K.

Keep in mind these solvers are largely limited to linear problems, so no exponential constraints, sine functions, etc!

We will learn more about linear programming in the next section.

```
from pulp import *


# declare your variables
x_a = LpVariable("x_a", cat=LpBinary)
x_b = LpVariable("x_b", cat=LpBinary)
x_c = LpVariable("x_c", cat=LpBinary)
x_d = LpVariable("x_d", cat=LpBinary)

# defines the problem
prob = LpProblem("Knapsack Problem", LpMaximize)

# defines the weight constraint where items can't exceed 24
prob += 5*x_a + 10*x_b + 4*x_c + 6*x_d <= 24

# defines the objective function to maximize
prob += 10*x_a + 50*x_b + 30*x_c + 40*x_d

# solve the problem
status = prob.solve()
print(LpStatus[status])

# print the results x_a=0.0, x_b=1.0, x_c=1.0, x_d=1.0
print(value(x_a))
print(value(x_b))
print(value(x_c))
print(value(x_d))
```

Pros and Cons of Tree Search

Pros	Cons
• Precise, deterministic, and consistent • Does a thorough search while avoiding infeasible search space • Relatively easy to reason with and explain to others, even non-technical people • Flexible and adaptable to many discrete problems, supporting a wide array of constraints and optimization objectives	• Can be really slow if applied to improperly pruned or extremely large problems, heuristic and bounding is often necessary to prune search space feasibly. • Not necessarily designed for problems with continuous variables (unless you use decision trees/random forests-like paradigms)

The Many Applications of Search Trees

Tree search algorithms are used heavily in many areas of applied mathematics and "artificial intelligence".

- Permutations and combinations
- Minimax (used for Chess, game theory)
- Alpha-Beta Pruning (for Chess, adversarial search)
- Branch-and-bound algorithms
- Mixed Integer Programming

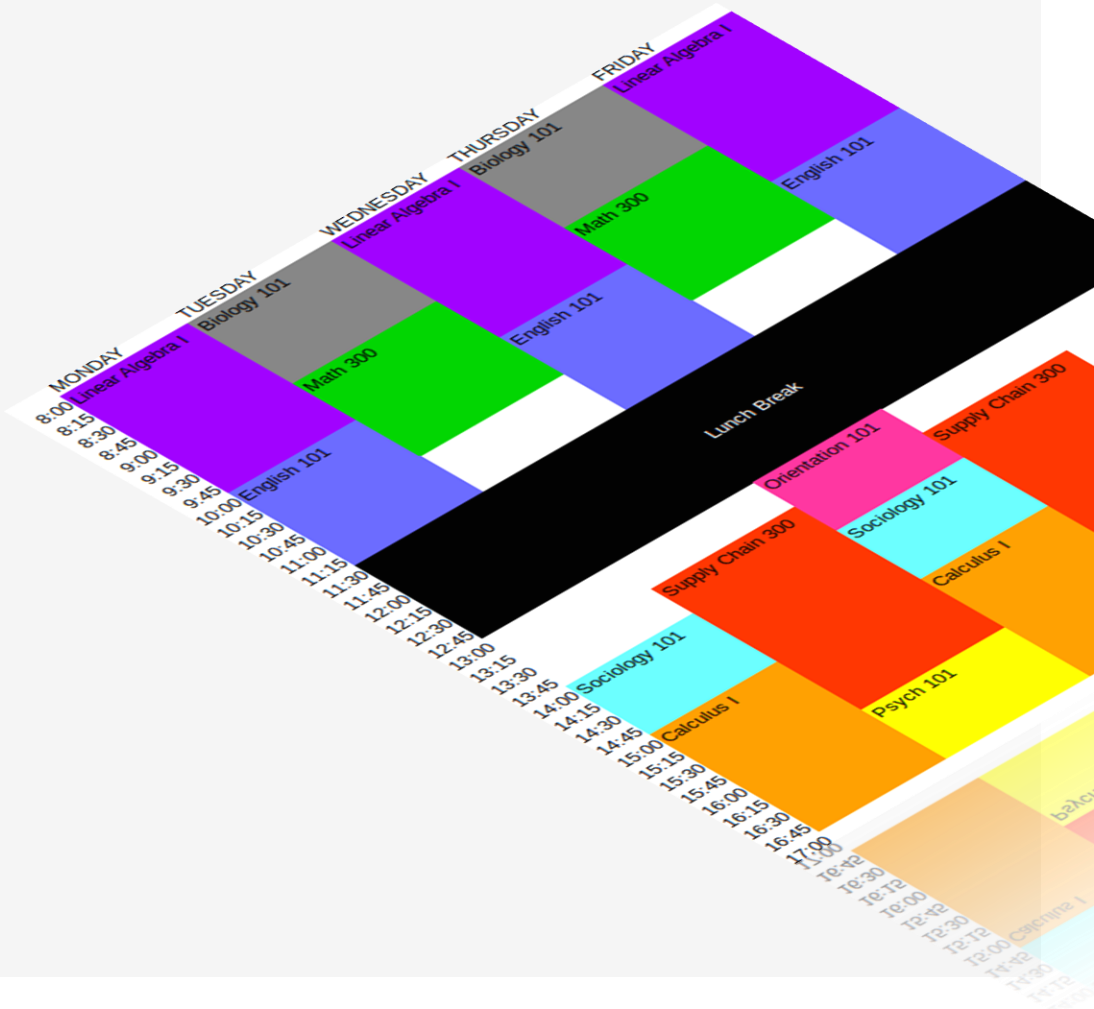
The breakthrough "MAC Hack VI" chess program in 1965.

Quiz Time!

Branch and Bound algorithms will guarantee an optimal solution.

A True

B False



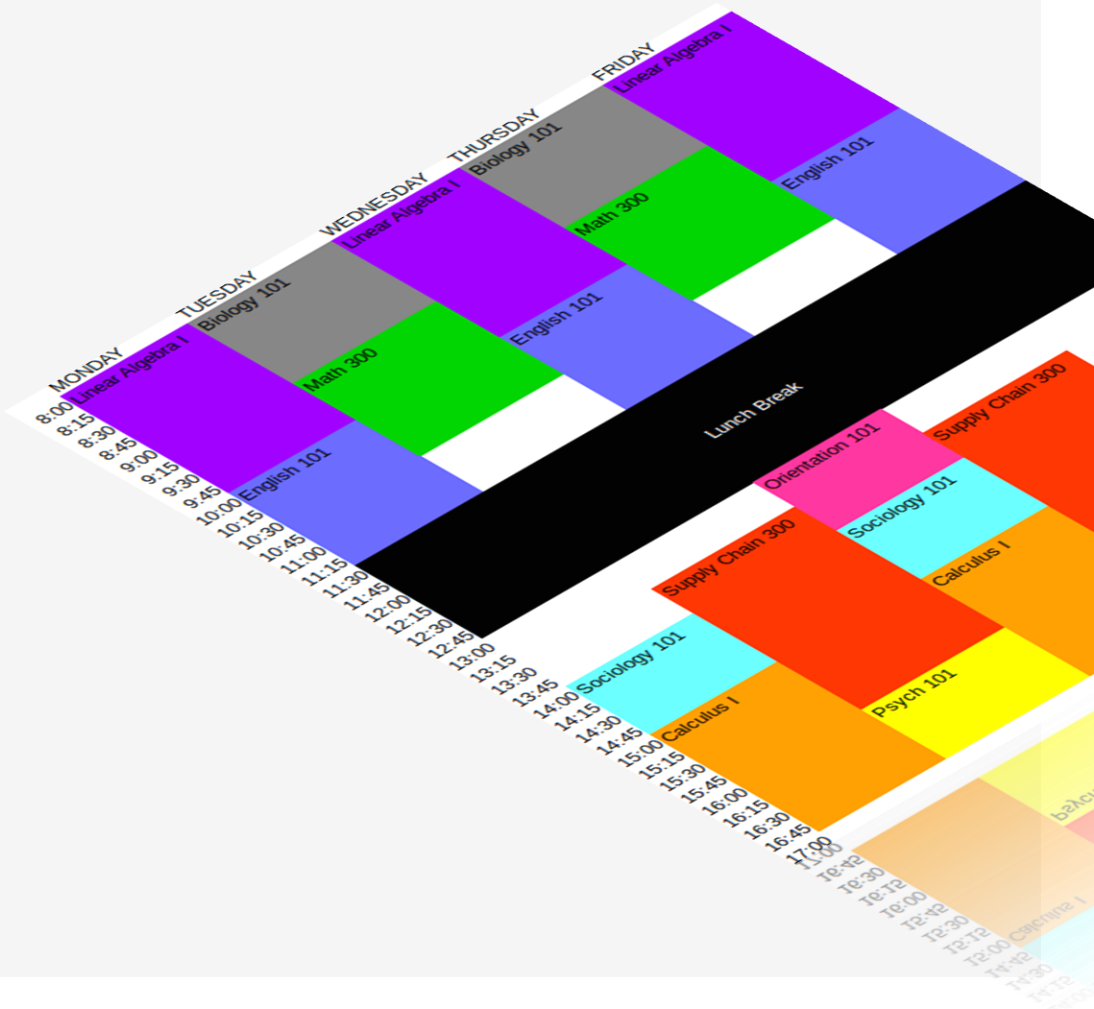
Quiz Time!

Branch and Bound algorithms will guarantee an optimal solution.

A True

B False

Unlike stochastic (random) algorithms, search trees usually cover an entire search space while pruning obvious infeasible/inferior solutions, and ultimately will find the optimal solution given enough time and correct heuristics.



Quiz Time!

You have 175 late shipments and you need to prioritize which ones to put on a truck. Your boss says to prioritize by shipping the most delayed orders first, but take into account the size of the orders too.

Which heuristic should you likely sort your items in your tree search?

A Sort on $\frac{\#\ days\ delay}{weight}$, starting with the highest first

B Sort on $\frac{\#\ days\ delay}{weight}$, starting with the lowest first

C Sort on *# days delay,* starting with the highest first

Quiz Time!

You have 175 late shipments and you need to prioritize which ones to put on a truck. Your boss says to prioritize by shipping the most delayed orders first, but take into account the size of the orders too.

Which heuristic should you likely sort your items in your tree search?

A Sort on $\frac{\#\ days\ delay}{weight}$, starting with the highest first

B Sort on $\frac{\#\ days\ delay}{weight}$, starting with the lowest first

C Sort on *# days delay,* starting with the highest first

Think of "lateness" as the currency driving value for each shipment. To effectively do a branching search, we need to sort the items by a ratio dividing their value (lateness) by the weight. This will guide the heuristic to select the most critical items first.

Section IV
Linear Programming

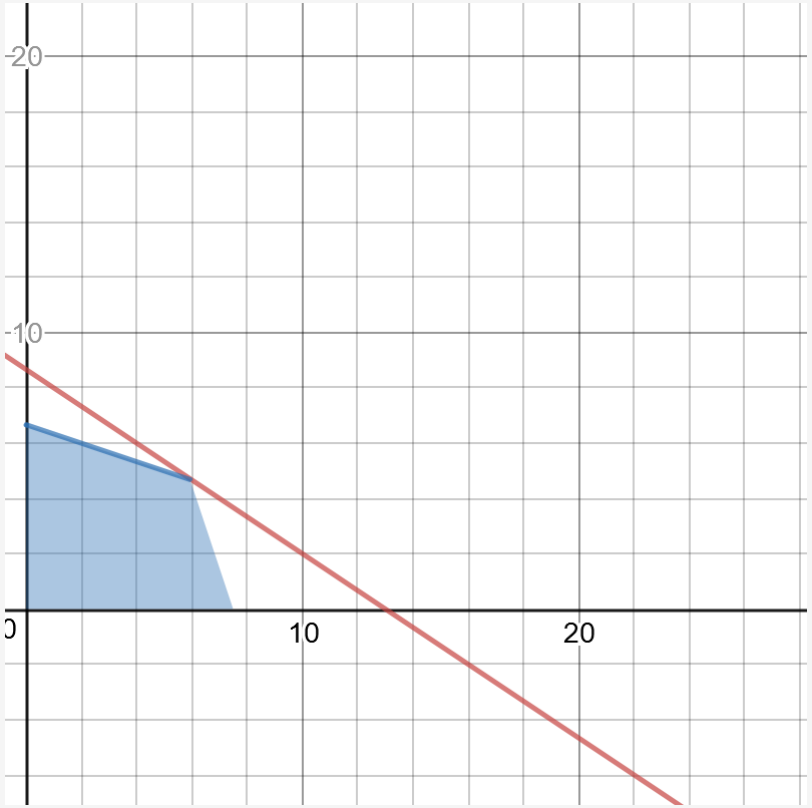
What Is Linear Programming?

Linear Programming is the workhorse of operations research algorithms, and models variables, objectives, and constraints as linear relationships

- LP models convert a problem into a system of linear equations.
- Linear algebra techniques will express these variables and constraints in matrix format, and use Gaussian elimination to solve for the variables.
- When discrete/integer variables are involved, tree search algorithms can use linear programming to reduce the search space

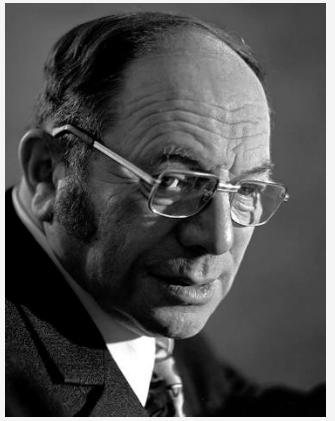
Linear programming and the simplex method can be quite a rabbit hole, so we will do an intuitive graphical approach and encourage using libraries to do the linear algebra work for you.

https://en.wikipedia.org/wiki/Linear_programming

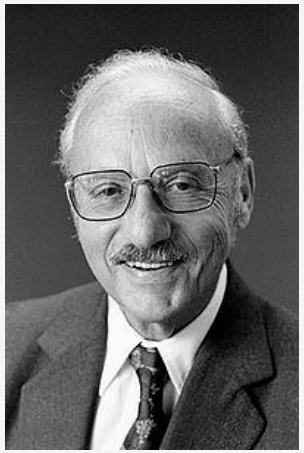


A Brief History of Linear Programming

- Pioneered by Soviet economist Leonid Kantorovich in 1939 to reduce costs of the Soviet Army.
- Further developed by T.C. Koopman (Dutch American) and Frank Lauren Hitchcock (American).
- The Simplex Method was then invented by George B. Dantzig in 1947 for the U.S. Air Force, and this is the breakthrough algorithm used today.



Leonid Kantorovich



George B. Dantzig

George B. Dantzig being awarded National Medal of Science by by U.S. President Gerald Ford in 1976.

Problem 1 – Factory Optimization

You have two lines of products: the **iPac** and **iPac Ultra**.

The **iPac** makes $200 of profit while the **iPac Ultra** makes $300 of profit.

However, the assembly line can only work for **20 hours**, and it takes 1 hour to produce the **iPac** and 3 hours to produce an iPac Ultra.

Only 45 kits can be provided in a day, and an **iPac** requires 6 kits while **iPac Ultra** requires 2 kits.

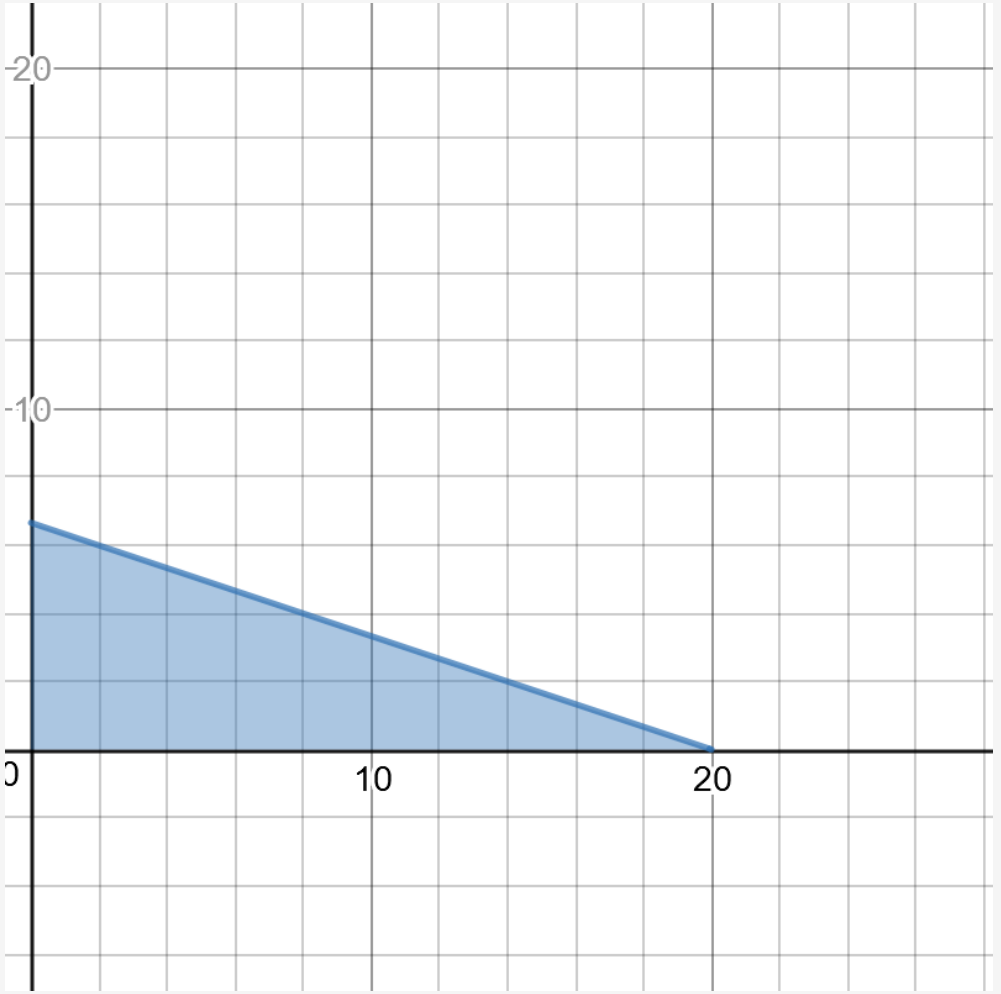
Assuming all supply will be sold, how many of the **iPac** and **iPac Ultra** should we sell to maximize profit?

Problem 1 – Factory Optimization

First Constraint:

"A single assembly line can only work for **20 hours**, and it takes 1 hour to produce the **iPac** and 3 hours to produce an iPac Ultra."

$x + 3y \leq 20 \ (x \geq 0, y \geq 0)$

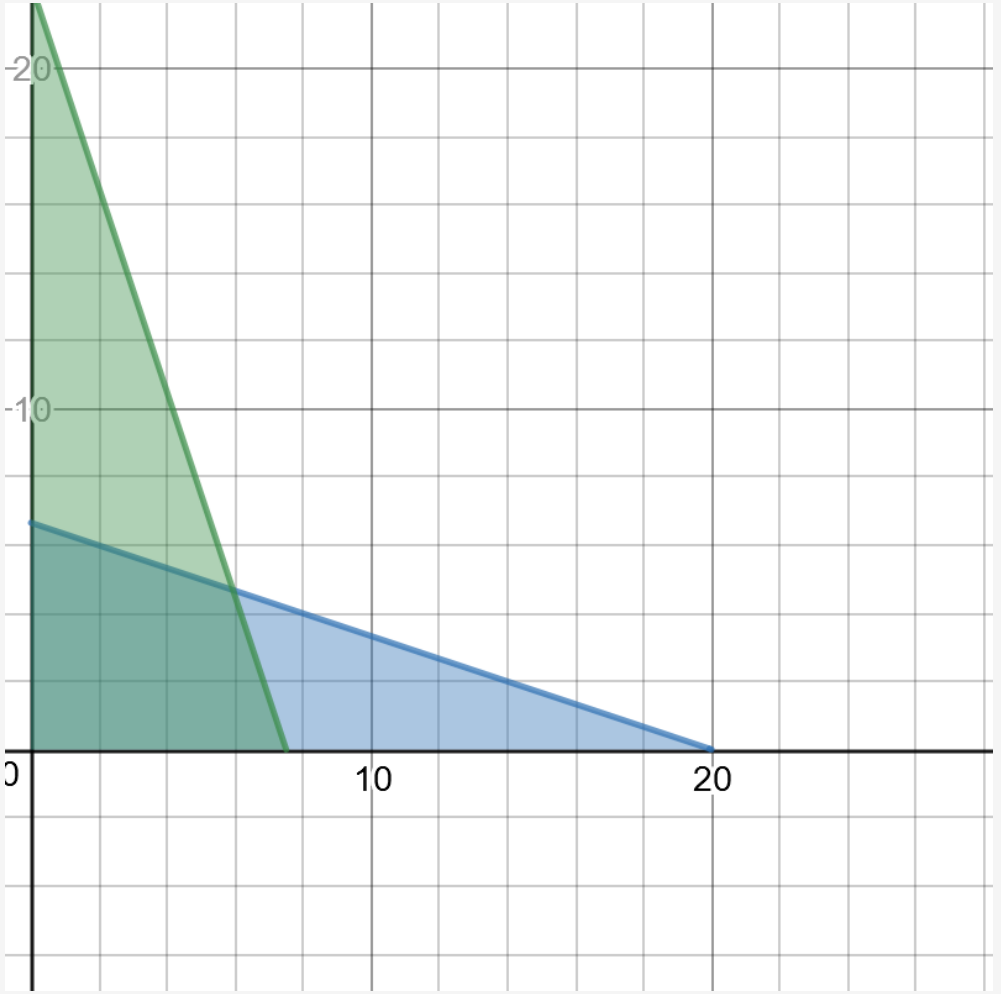


Problem 1 – Factory Optimization

Second Constraint:

"**Only 45 kits** can be provided in a day, and an **iPac** requires 6 kits while **iPac Ultra** requires 2 kits. "

$6x + 2y \leq 45 \ (x \geq 0, y \geq 0)$



Problem 1 – Factory Optimization

The region where all the constraints overlap is the **feasible** region.

At one of the corners (called vertices) is the solution.

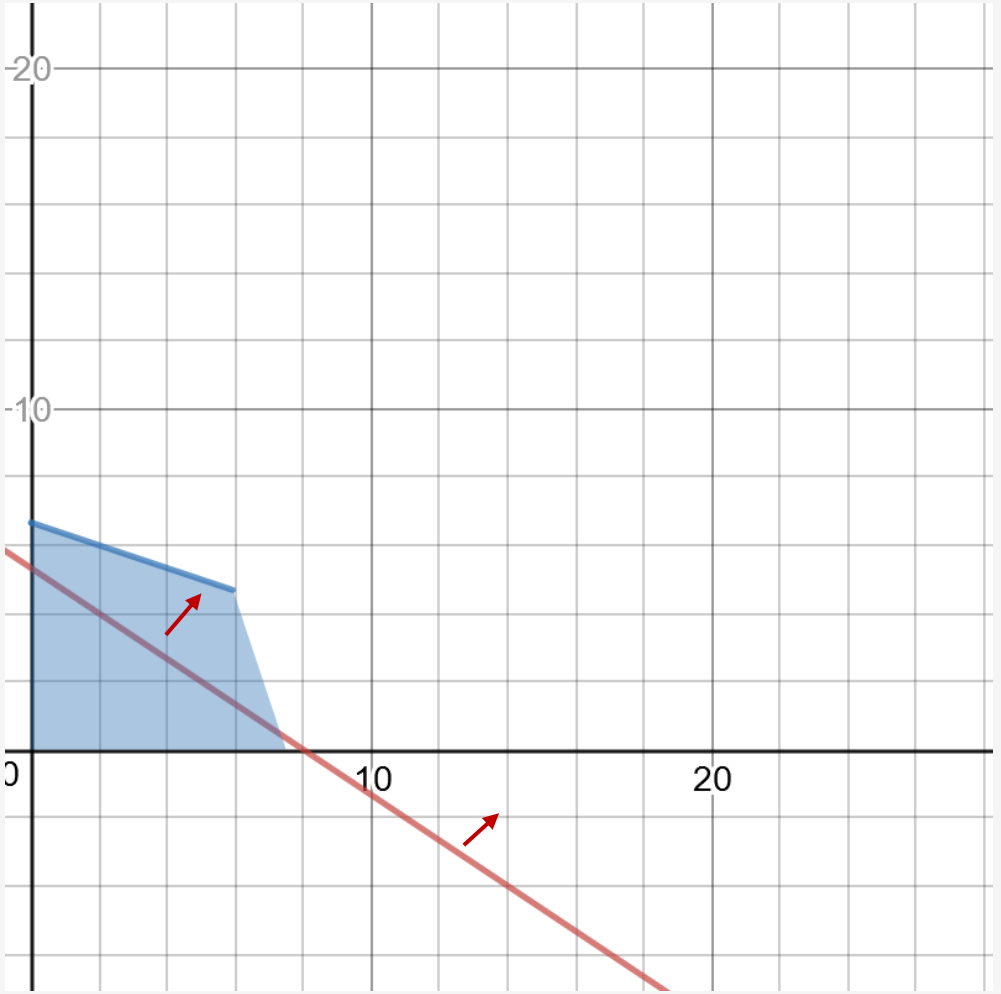
Problem 1 – Factory Optimization

"The **iPac** makes $200 of profit while the **iPac Ultra** makes $300 of profit."

The objective function (profit) is defined by:

$$Z = 200x + 300y$$

Since this is a maximization problem, we will increase **Z** and move the objective line upwards and the last convex it touches is the solution.



https://www.desmos.com/calculator/iildqi2vt7

Problem 1 – Factory Optimization

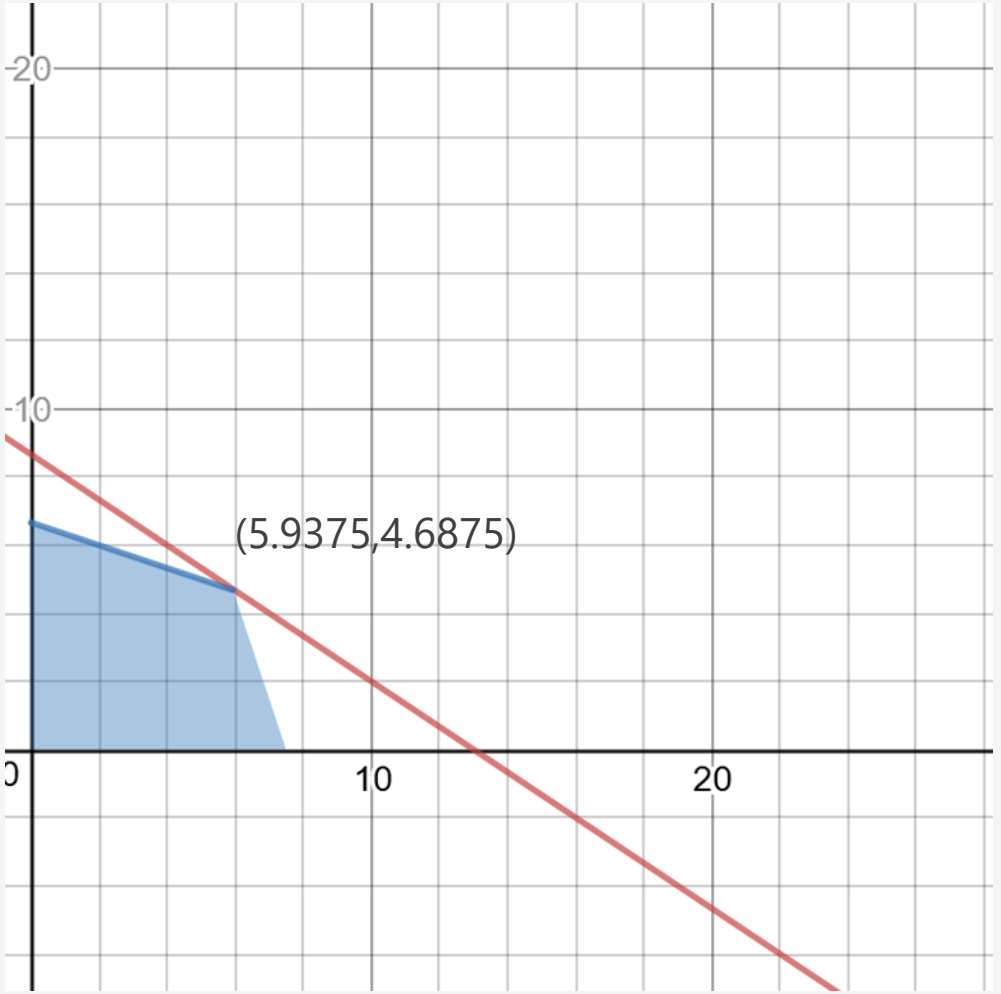
"The **iPac** makes $200 of profit while the **iPac Ultra** makes $300 of profit."

The objective function (profit) is defined by:

$$Z = 200x + 300y$$

Since this is a maximization problem, we will increase **Z** and move the objective line upwards and the last convex it touches is the solution.

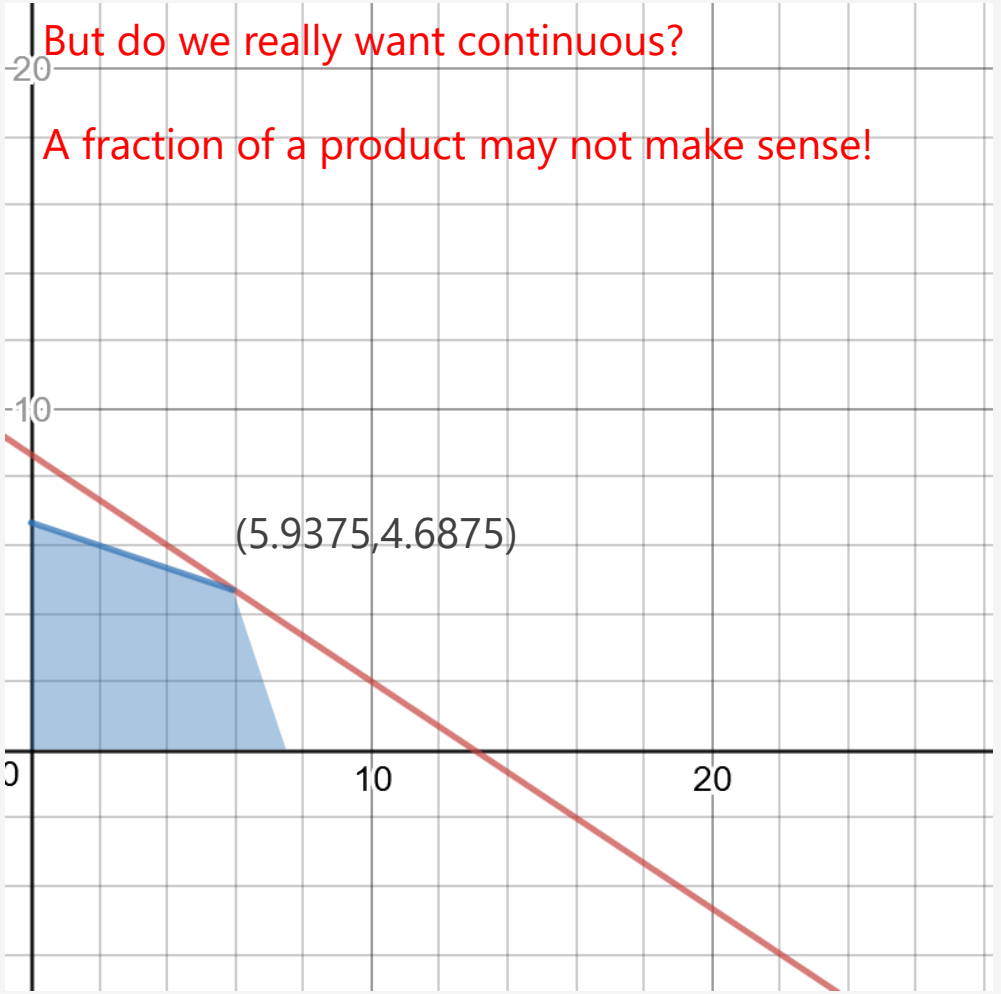
On the next slide, we will see how to solve this using Python PuLP.



https://www.desmos.com/calculator/iildqi2vt7

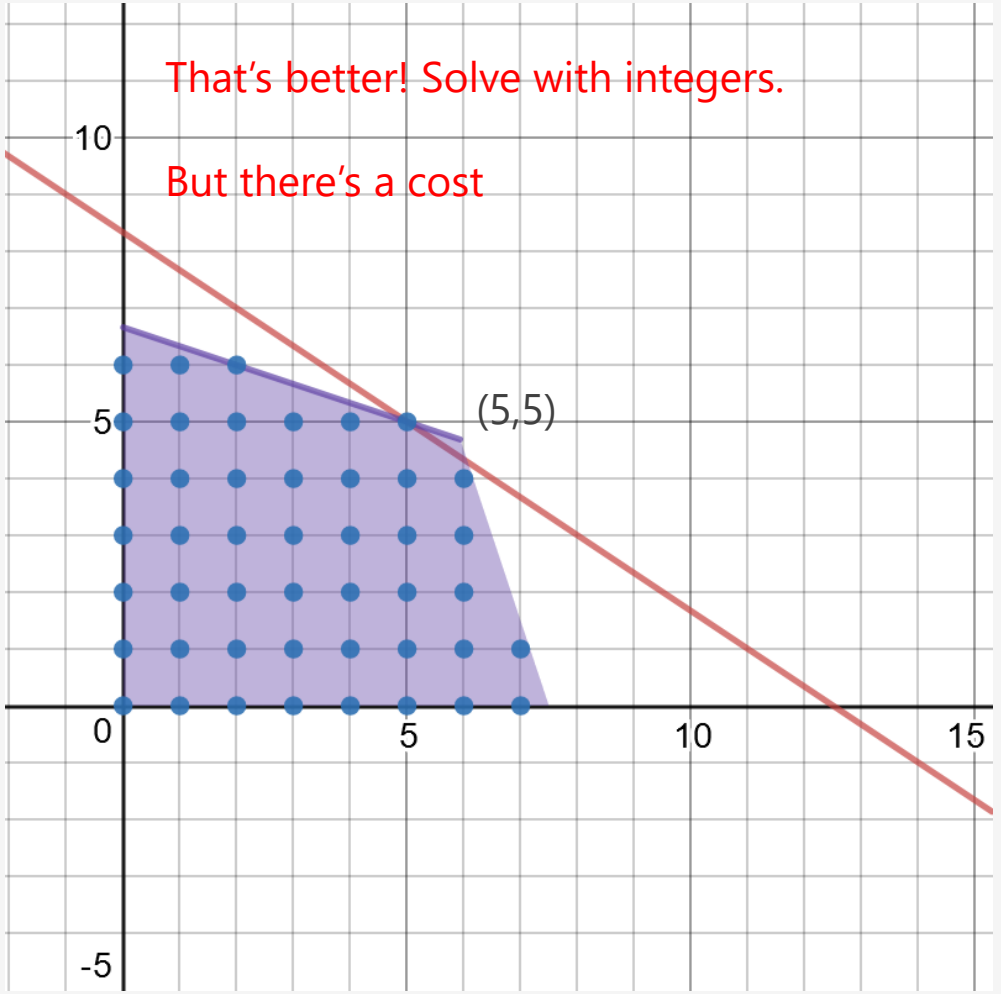
Problem 1 – Factory Optimization

```
from pulp import *

# declare your variables
x = LpVariable("x", 0)    # 0<= x
y = LpVariable("y", 0)   # 0<= x

# defines the problem
prob = LpProblem("factory_problem", LpMaximize)

# defines the constraints
prob += x + 3*y <= 20
prob += 6*x +2*y <= 45



# defines the objective function to maximize
prob += 200*x + 300*y

# solve the problem
status = prob.solve()
print(LpStatus[status])

# print the results x = 5.9375, y = 4.6875
print(value(x))
print(value(y))
```



Problem 1 – Factory Optimization (Integer Version)

```python
from pulp import *

# declare your variables
x = LpVariable("x", lowBound=0, cat=LpInteger)    # 0<=x
y = LpVariable("y", lowBound=0, cat=LpInteger) # 0<=y

# defines the problem
prob = LpProblem("factory_problem", LpMaximize)

# defines the constraints
prob += x + 3*y <= 20
prob += 6*x +2*y <= 45

# defines the objective function to maximize
prob += 200*x + 300*y

# solve the problem
status = prob.solve()
print(LpStatus[status])

# print the results x = 5, y = 5
print(value(x))
print(value(y))
```



https://www.desmos.com/calculator/vu6gxi1tgn

Linear versus Integer Programming

Linear programming via the Simplex algorithm is usually quite lean, and does some matrix work (called the tableau) to find a solution.

However, as shown in our last example, we do not always want our variables to be continuous but rather discrete integers.

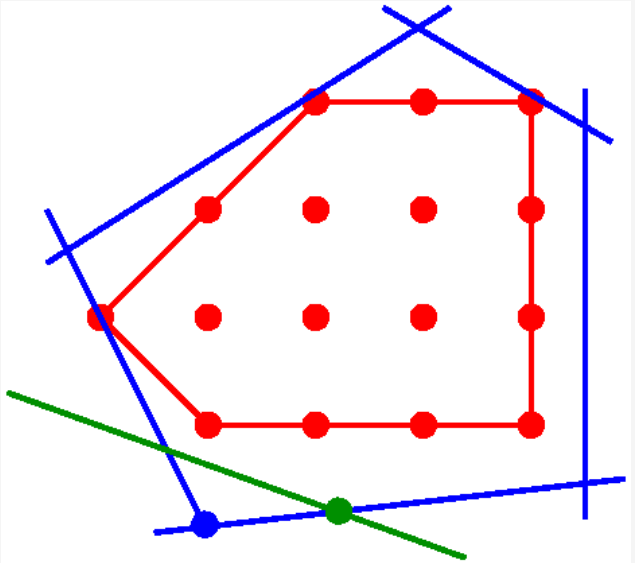
- This has a steep cost because our solver now has to use a tree search algorithm to traverse a search space.
- For this reason, you may consider using linear programming and just rounding the result.
- But if integer-based solutions are critical, linear programming can still be leveraged to reduce the search space and guide it towards the solution (using techniques like polyhedral cutting, bounding, etc.)

Linear versus Integer Programming

If you are comfortable with linear algebra, then doing linear/integer programming from scratch can be quite rewarding.

However linear programming, the simplex algorithm, and cutting-plane methods are a bit involved.

Unless you are super comfortable with linear algebra, just stick with using libraries like PuLP, PyOmo, and ojAlgo.



Problem 2: Minimization

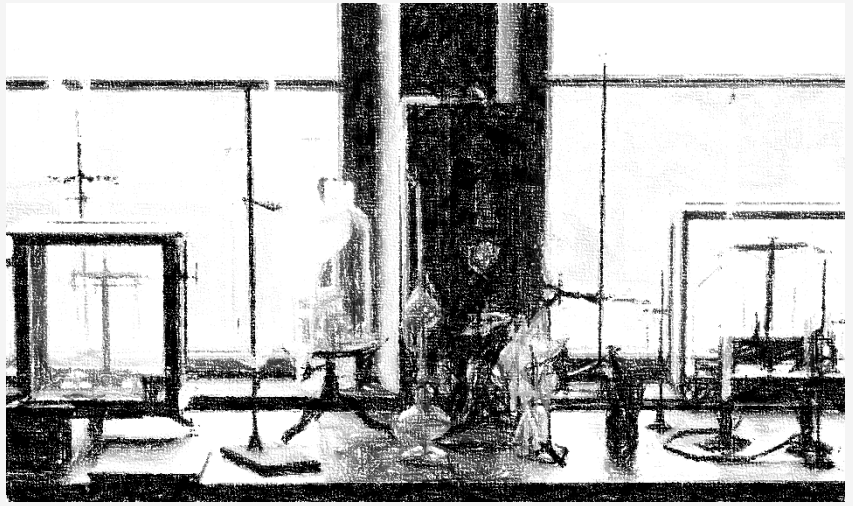
You need **at least 50 oz** of oil**.**

It costs $3/oz for oil from Supplier A, and $1/oz from Supplier B.

An ounce of Supplier A takes 1 unit of storage and Supplier B takes 2 units of storage. You only have **120 units of storage.**

To maintain quality, you need at least 1oz of Supplier A oil per 2oz of Supplier B oil.

How much of Supplier A and Supplier B oil should we buy to minimize cost?



Problem 2: Minimization

You need **at least 50 oz** of oil.

$$x + y \geq 50$$

It costs $3/oz for oil from Supplier A, and $1/oz from Supplier B.

$$3x + y = Z$$

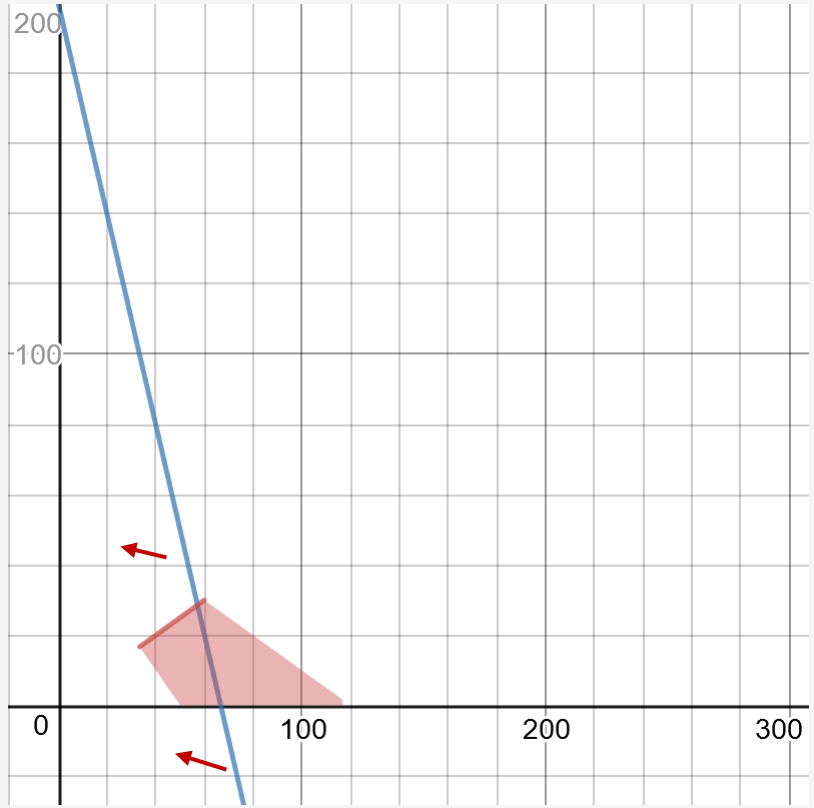
An ounce of Supplier A takes 1 unit of storage and Supplier B takes 2 units of storage. You only have **120 units of storage.**

$$1x + 2y \leq 120$$

To maintain quality, you need at least 1oz of Supplier A oil per 2oz of Supplier B oil.

$$x \geq 2y$$

How much of Supplier A and Supplier B oil should we buy to minimize cost?



Since we are minimizing, move the objective line so that Z decreases until we hit the last convex point.

https://www.desmos.com/calculator/jxjytnpad3

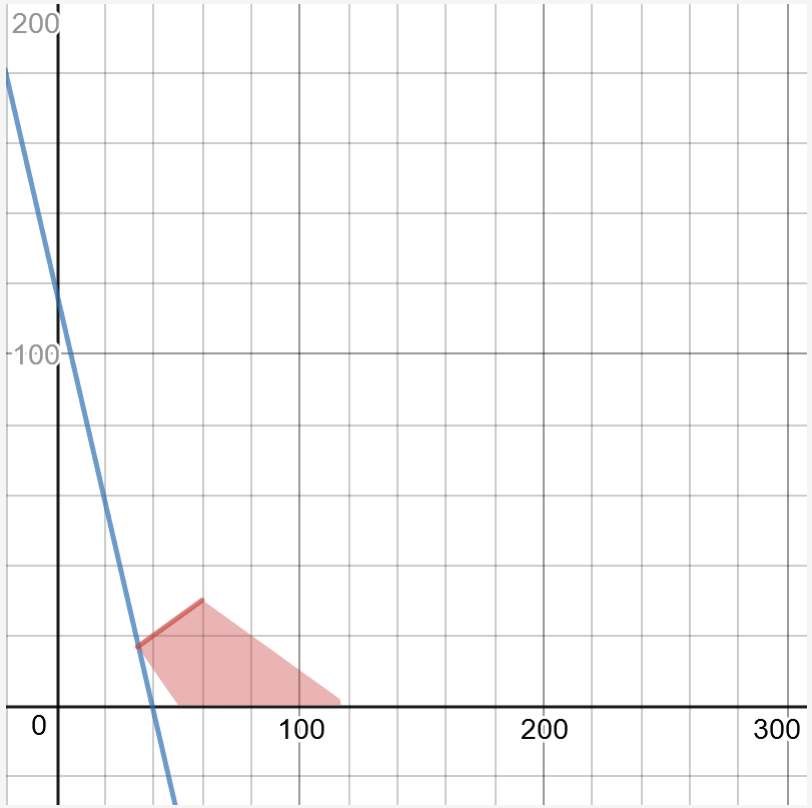
Problem 2: Minimization

ANSWER:

x=33.333333

y=16.666667

```
from pulp import *

x = LpVariable("x", lowBound=0, cat=LpContinuous)
y = LpVariable("y", lowBound=0, cat=LpContinuous)

# Defines the problem
prob = LpProblem("Oil Supplier Problem", LpMinimize)

# Objective
prob += 3*x + y

# Constraints
prob += x + 2*y <= 120
prob += x + y >= 50
prob += x >= 2*y

# Solve!
status = prob.solve()
print(LpStatus[status])

print(value(x)) # 33.333333
print(value(y)) # 16.666667
```



Since we are minimizing, move the objective line so that Z decreases until we hit the last convex point.

https://www.desmos.com/calculator/jxjytnpad3

Problem 3 – Portfolio Optimization

You have $100,000 to put into 5 investment opportunities:

Investment	Projected Return
Stock A	15%
Stock B	10%
Stock C	20%
Real Estate	10%
Unconventional	40%

Find the right mix of investments to maximize projected return for the entire portfolio, while respecting these constraints:

- Stocks cannot account for more than 60% of total investment
- No single stock investment can exceed 40% of total stock investment
- Real estate investment must be at least 50% of the amount invested in stocks
- Unconventional investment cannot exceed 15% of total investment

Problem 3 – Portfolio Optimization

You have $100,000 to put into 5 investment opportunities:

Investment	Projected Return	Variable
Stock A	15%	x_1
Stock B	10%	x_2
Stock C	20%	x_3
Real Estate	10%	x_4
Unconventional	40%	x_5

Find the right mix of investments to maximize projected return for the entire portfolio, while respecting these constraints:

- Stocks cannot account for more than 60% of total investment: $\boldsymbol{x_1 + x_2 + x_3 \leq (.60)(100,000)}$
- No single stock investment can exceed 40% of total stock investment: $.40(\boldsymbol{x_1 + x_2 + x_3}) \leq \boldsymbol{x_{1\ through\ 3}}$
- Real estate investment must be at least 50% of the amount invested in stocks: $(.\mathbf{50})(\boldsymbol{x_1 + x_2 + x_3}) \leq \boldsymbol{x_4}$
- Unconventional investment cannot exceed 15% of total investment: $(.\mathbf{15})(\mathbf{100,000}) >= \boldsymbol{x_5}$

Problem 3 – Portfolio Optimization

To the right is how we use Python PuLP to solve this portfolio problem.

We should get the following results:

$x_1 = 22666.667$

$x_2 = 11333.333$

$x_3 = 22666.667$

$x_4 = 28333.333$

$x_5 = 15000.0$

```
from pulp import *

# declare your variables
x_1 = LpVariable("x_1", lowBound=0, cat=LpContinuous)
x_2 = LpVariable("x_2", lowBound=0, cat=LpContinuous)
x_3 = LpVariable("x_3", lowBound=0, cat=LpContinuous)
x_4 = LpVariable("x_4", lowBound=0, cat=LpContinuous)
x_5 = LpVariable("x_5", lowBound=0, cat=LpContinuous)

# defines the problem
prob = LpProblem("Portfolio Problem", LpMaximize)

# total investment must be 100,000
prob += x_1 + x_2 + x_3 + x_4 + x_5 == 100000

# No single stock can be more than 40% of all stock investment
prob += x_1 <= .40*(x_1 + x_2 + x_3)
prob += x_2 <= .40*(x_1 + x_2 + x_3)
prob += x_3 <= .40*(x_1 + x_2 + x_3)

# stock investments cannot exceed 60% of total investment
prob += x_1 + x_2 + x_3 <= (.60 * 100000)

# Real estate must be at least 50% of the amount invested in stocks
prob += .5*(x_1 + x_2 + x_3) <= x_4

# Unconventional investment cannot exceed 15% of total investment
prob += (.15 * 100000) >= x_5

# defines the objective function to maximize
prob += .15*x_1 + .10*x_2 + .20*x_3 + .10*x_4 + .40*x_5

# solve the problem
status = prob.solve()
print(LpStatus[status])

# print the results
print(value(x_1))
print(value(x_2))
print(value(x_3))
print(value(x_4))
print(value(x_5))
```

Problem 4 – Mixed Integer Problem

You have three drivers who charge the following rates:

- Driver 1: $10 / hr
- Driver 2: $12 / hr
- Driver 3: $15 / hr

From 6:00 to 22:00, schedule one driver at a time to provide coverage, and minimize cost.

Each driver must work 4-6 hours a day. Driver 2 cannot work after 11:00.

Stay Calm

Variables and Constants

S_i = Shift start time of each i driver

E_i = Shift end time for each i driver

R_i = Hourly rate for each i driver

δ_{ij} = Binary (1,0) between two ij drivers

M = Length of planning window

Minimize

$$\sum_{i=1}^{3} R_i(E_i - S_i)$$

Constraints

$$4 <= E_i - S_i <= 6$$

$$16 = \sum_{i=1}^{3} E_i - S_i$$

$$E_2 <= 11$$

$$S_i >= E_j - M\delta_{ij}$$

$$S_j >= E_i - M(1 - \delta_{ij})$$

Problem 4 – Mixed Integer Problem

Here is how we solve this problem mixing continuous and binary variables in PuLP.

ANSWER:
Optimal: $190

Driver 1: 11.0-17.0
Driver 2: 6.0-11.0
Driver 3: 17.0-22.0

```
from pulp import *


# Declare your variables
s_1 = LpVariable("s_1", lowBound=6, upBound=22, cat=LpContinuous)
e_1 = LpVariable("e_1", lowBound=6, upBound=22, cat=LpContinuous)

s_2 = LpVariable("s_2", lowBound=6, upBound=22, cat=LpContinuous)
e_2 = LpVariable("e_2", lowBound=6, upBound=22, cat=LpContinuous)

s_3 = LpVariable("s_3", lowBound=6, upBound=22, cat=LpContinuous)
e_3 = LpVariable("e_3", lowBound=6, upBound=22, cat=LpContinuous)

# Defines the problem
prob = LpProblem("Driver Shift Problem", LpMinimize)

# Objective
prob += 10*(e_1-s_1) + 12*(e_2-s_2) + 15*(e_3-s_3)

# Keep each driver between 4 and 6 hours
prob += 4 <= e_1 - s_1 <= 6
prob += 4 <= e_2 - s_2 <= 6
prob += 4 <= e_3 - s_3 <= 6

# Sum of all shifts must be 16
prob += (e_1 - s_1) + (e_2 - s_2) + (e_3 - s_3) == 16

# Driver 2 must be done by 11am
prob += e_2 <= 11

# Prevent Overlaps
b_12 = LpVariable("b_12", cat=LpBinary)
prob += s_1 >= e_2 - 24*b_12
prob += s_2 >= e_1 - 24*(1-b_12)

b_23 = LpVariable("b_23", cat=LpBinary)
prob += s_2 >= e_3 - 24*b_23
prob += s_3 >= e_2 - 24*(1-b_23)

b_31 = LpVariable("b_31", cat=LpBinary)
prob += s_3 >= e_1 - 24*b_31
prob += s_1 >= e_3 - 24*(1-b_31)

# Solve!
status = prob.solve()
print(LpStatus[status])

print("Driver 1: {}-{}".format(value(s_1), value(e_1)))
print("Driver 2: {}-{}".format(value(s_2), value(e_2)))
print("Driver 3: {}-{}".format(value(s_3), value(e_3)))
```

Problem 4 – Mixed Integer Problem

If you would like to iterate variables and constraints dynamically instead of hard-coding, use lists and loops and even some object-oriented programming.

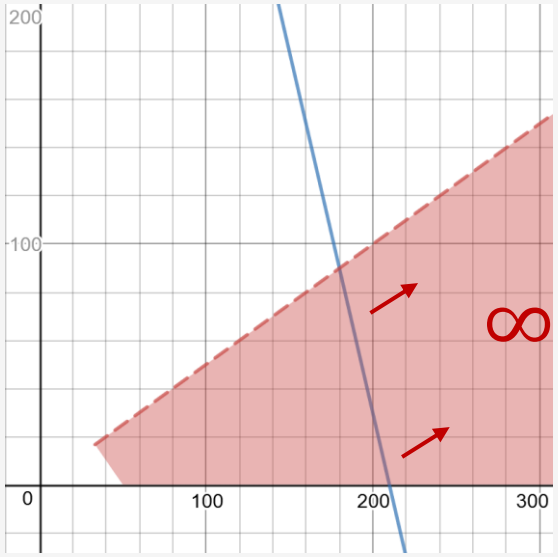
This is a bit more production-friendly if you need to import data and constraints from a database or other sources.

ANSWER:
Optimal: $190

Driver 1: 11.0-17.0
Driver 2: 6.0-11.0
Driver 3: 17.0-22.0

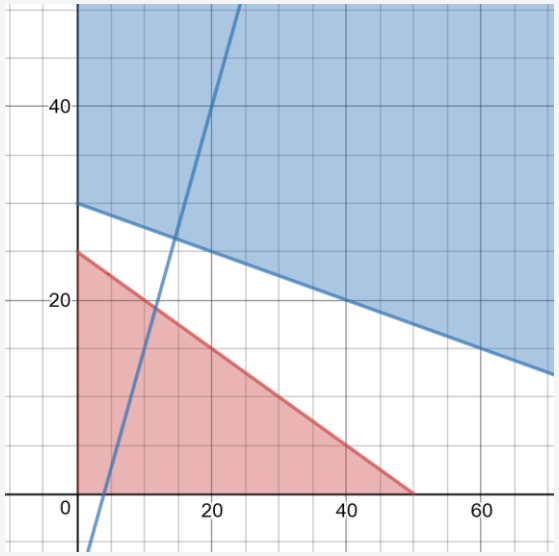
```
from pulp import *


class Driver:

    def __init__(self, index, rate):
        self.index = index
        self.rate = rate

        # Declare variables
        self.start = LpVariable("s_{}".format(index), lowBound=6, upBound=22, cat=LpContinuous)
        self.end = LpVariable("e_{}".format(index), lowBound=6, upBound=22, cat=LpContinuous)


drivers = [Driver(1, 10), Driver(2, 12), Driver(3, 15)]

# Defines the problem
prob = LpProblem("Driver Shift Problem", LpMinimize)

# Objective
prob += lpSum(d.rate * (d.end - d.start) for d in drivers)

# Keep each driver between 4 and 6 hours
for d in drivers:
    prob += 4 <= d.end - d.start <= 6

# Sum of all shifts must be 16
prob += lpSum(d.end - d.start for d in drivers) == 16

# Driver 2 must be done by 11am
prob += drivers[1].end <= 11

# Prevent Overlaps
for d1 in drivers:
    for d2 in drivers:
        if d2 != d1:
            b = LpVariable("b_{}{}".format(d1.index, d2.index), cat=LpBinary)
            prob += d1.start >= d2.end - 24 * b
            prob += d2.start >= d1.end - 24 * (1 - b)

# Solve!
status = prob.solve()
print(LpStatus[status])

for d in drivers:
    print("Driver {}: {}-{}".format(d.index, value(d.start), value(d.end)))
```

Special Cases



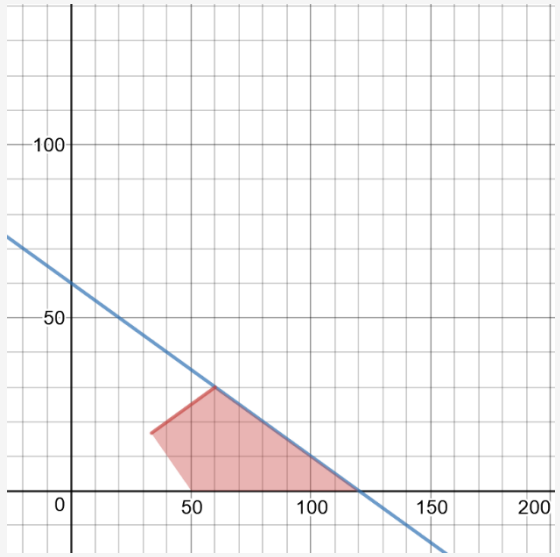
Unbounded – There's no limit to a maximization or minimization, usually due to missing or broken constraints.

https://www.desmos.com/calculator/grz7ewogfd



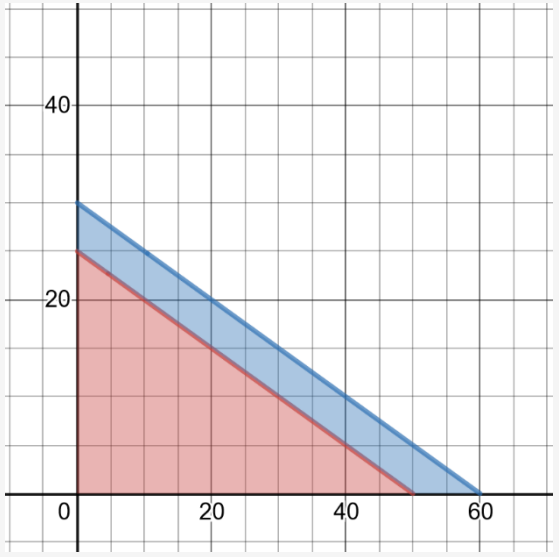
Infeasible – There is no overlap with all the constrained regions.

https://www.desmos.com/calculator/8cuwz0qvm1



Alternate Solutions – The objective line is parallel to the bounding constraint, presenting many solutions along the line.

https://www.desmos.com/calculator/oevm6pww8r



Redundancy – Two parallel constraints bounding the same direction can be consolidated into one.

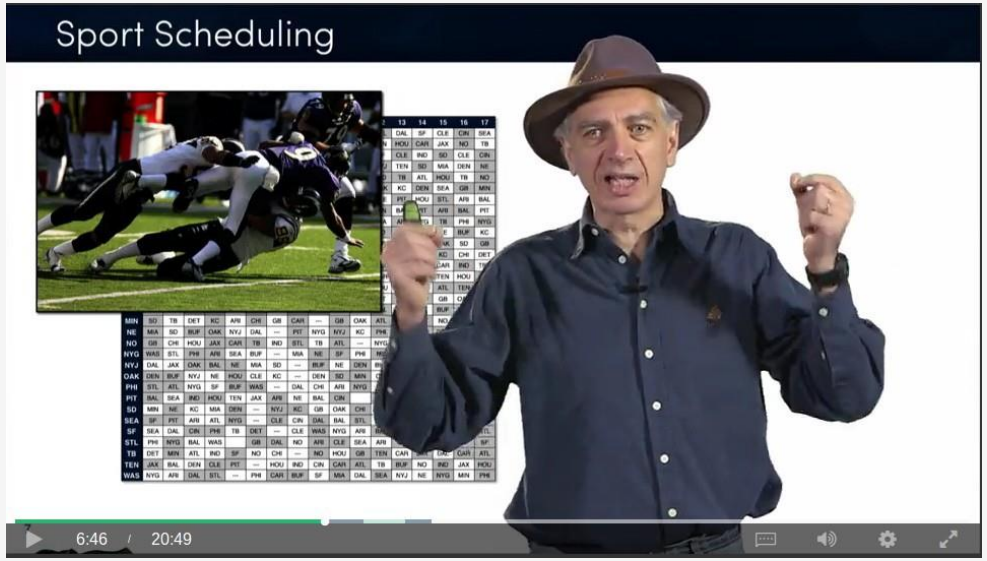
https://www.desmos.com/calculator/jlig5pjwsw

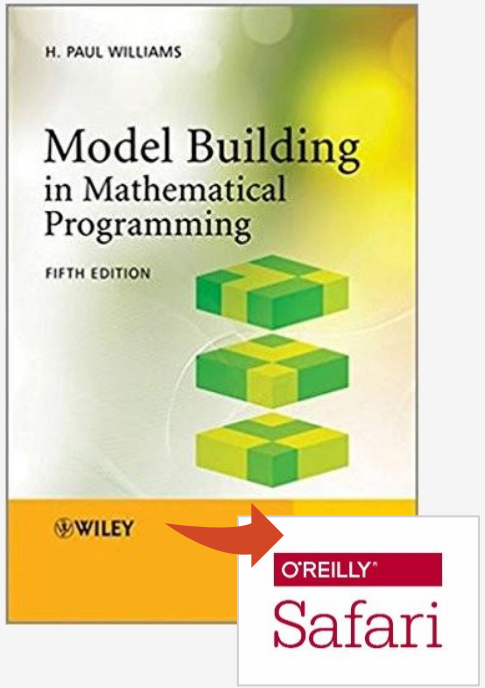
Pros and Cons of Linear/Integer/Mixed Programming

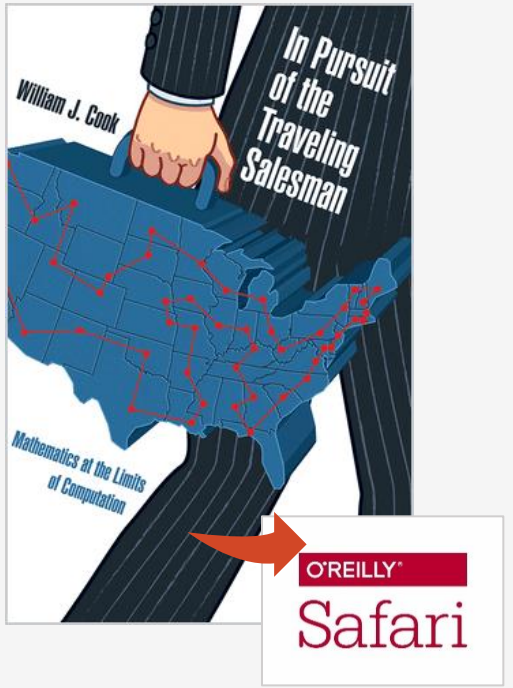
Pros	Cons
• Efficient and powerful, extremely fast especially with Simplex algorithm • Can be applied to a wide array of problems with a linear nature including Sudokus, manufacturing problems, finance, and operational planning. • Modeling constraints as linear inequalities is convenient and provides a nice abstraction to describe problems. • Many libraries are available to do the heavy-lifting for you	• **Strictly only supports linear variables and constraints** • Very mathy and requires esoteric linear algebra knowledge to build from scratch.

Learn More

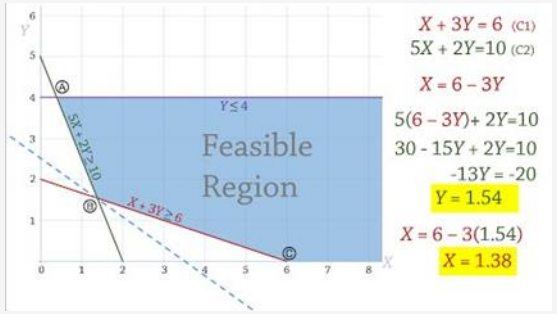
coursera *Discrete Optimization*



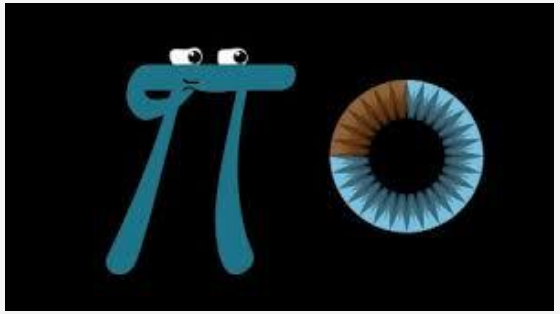




PatrickJMT on YouTube



Joshua Emmanuel on YouTube



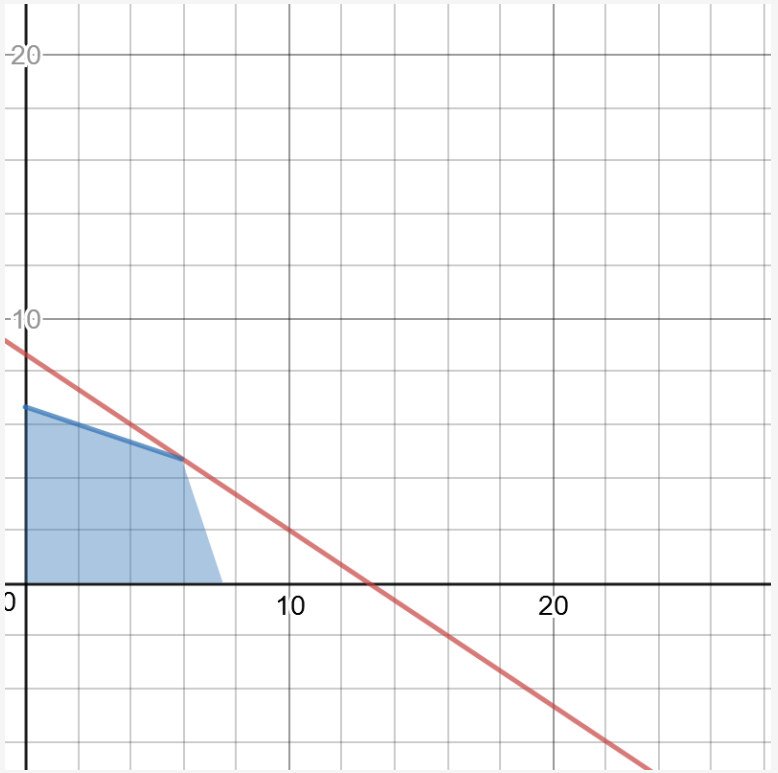
3Blue1Brown on YouTube

Quiz Time!

Linear programming guarantees an optimal solution, assuming one exists.

A True

B False



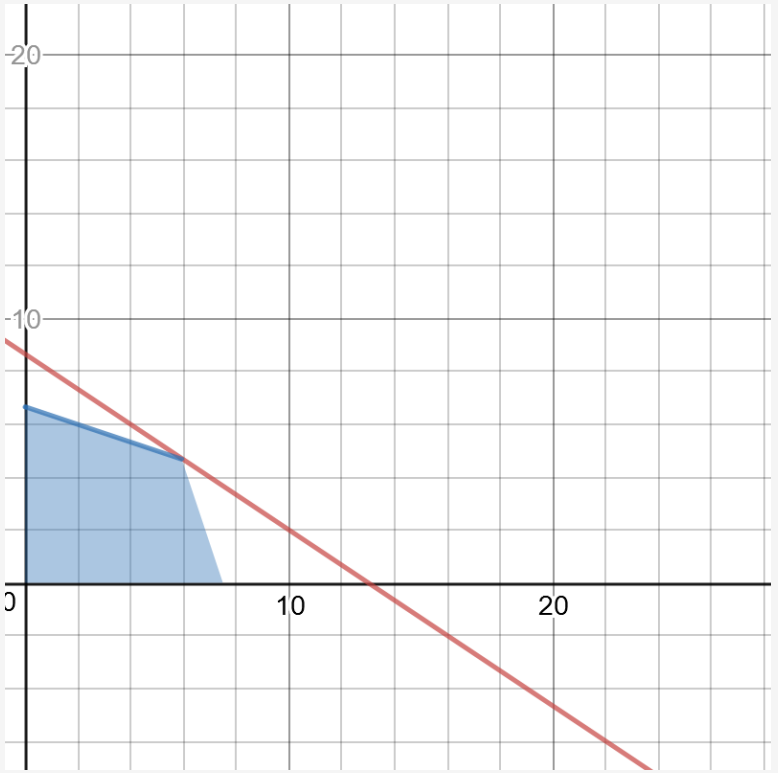
Quiz Time!

Linear programming guarantees an optimal solution, assuming one exists.

A True

B False

Linear programming systems guarantee an optimal solution and converge on a single global minimum/maximum. This is made possible by the convex nature of linear systems and efficient linear algebra methodologies.

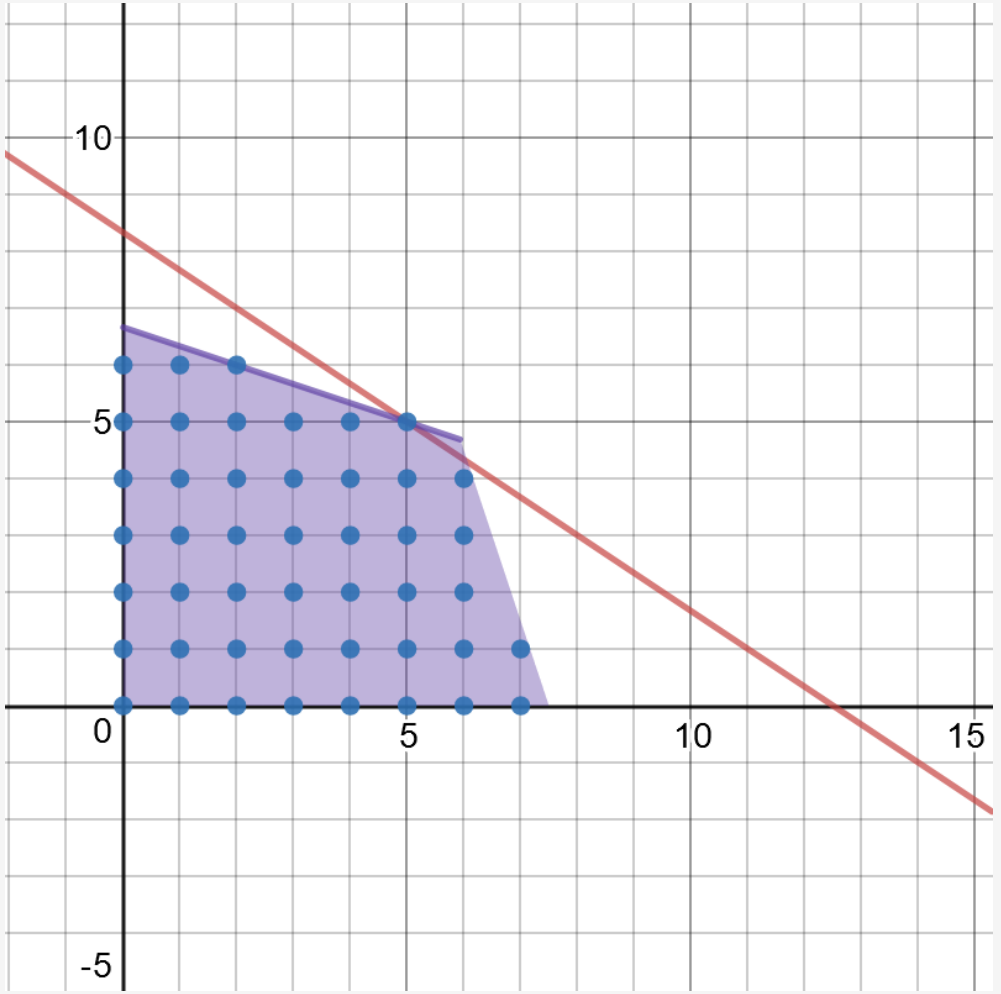


Quiz Time!

A linear program that solves for integers is much easier to compute than one that solves for decimals.

A True

B False



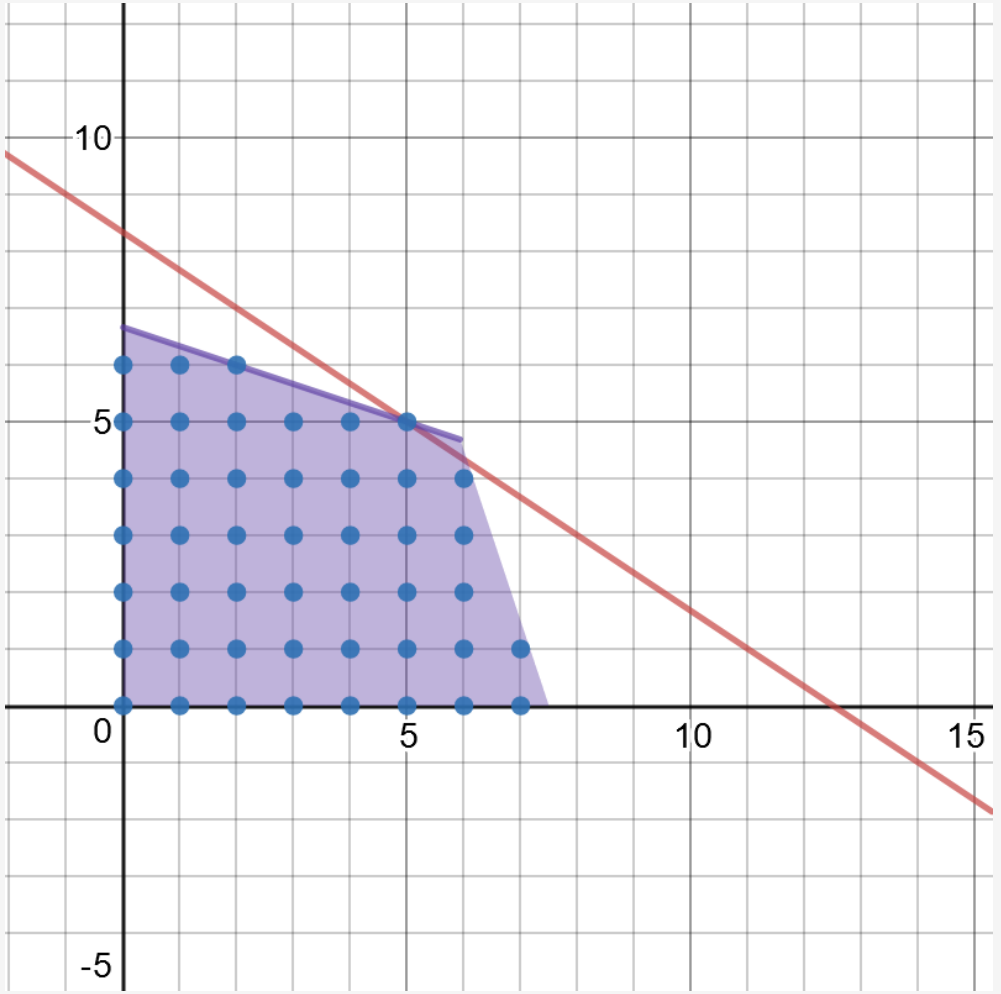
Quiz Time!

A linear program that solves for integers is much easier to compute than one that solves for decimals.

A True

B False

Pure continuous linear systems are easy to solve and can be done with some efficient matrix work. However, integer and discrete variables require search trees and pruning methodologies which can have a much higher performance cost.

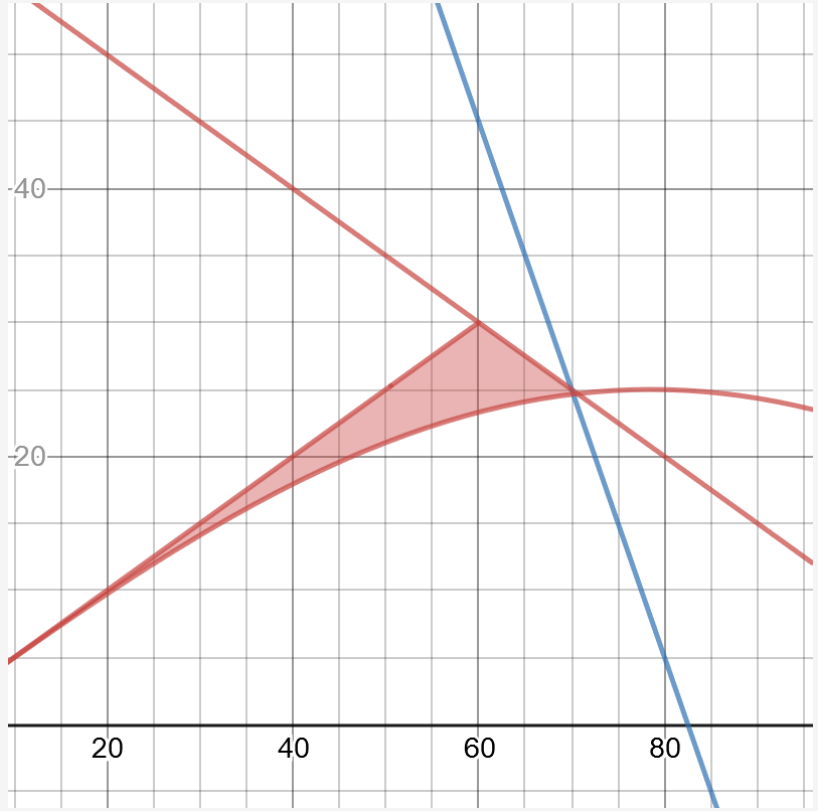


Quiz Time!

A linear program can model variables into constraints that involve squares, square roots, trigonometric functions, and other nonlinear functions.

A True

B False

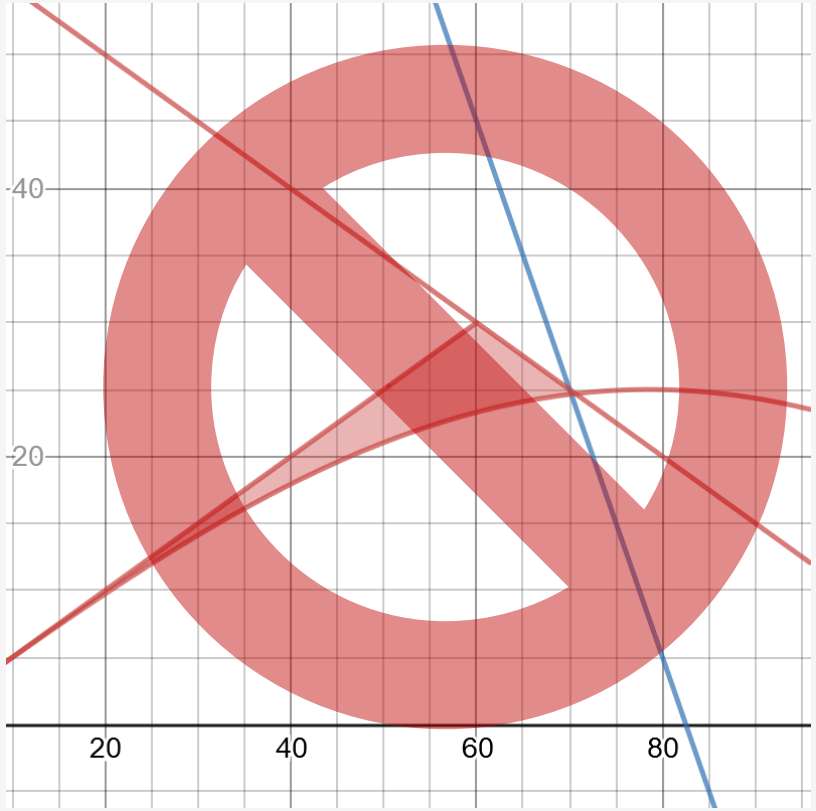


Quiz Time!

A linear program can model variables into constraints that involve squares, square roots, trigonometric functions, and other nonlinear functions.

A True

B False



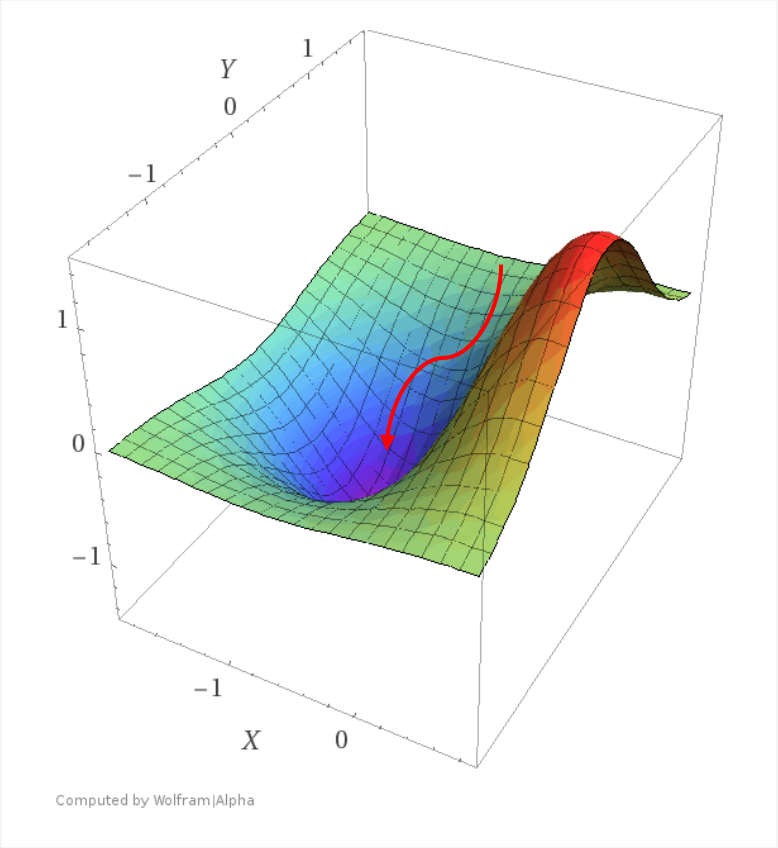
For the most part, solver libraries only support linear constraints and variables. So no squares, square roots, trig functions, etc.

You will need to use metaheuristics, gradient descent, specialized tree search, and other approaches to solve nonlinear systems.

Section V
Gradient Descent and Neural Networks

Gradient Descent - Making Calculus Useful

- **Gradient descent** is a continuous/nonlinear optimization method that uses Calculus derivatives to find a local minimum/maximum.
- This methodology has become highly popular due to the need to optimize thousands and even millions of machine learning variables, like neural networks and their node weight values.
- While you can use simulated annealing to do these ML optimizations, gradient descent offers a more guided methodology that scales well with extremely large numbers of variables, at the cost of finding a better optimum.
- The main drawback of gradient descent is it is very mathy and easily gets stuck in local minima, which is why stochastic approaches are used to add randomness.



A Crash Course in Calculus Derivatives

A derivative tells the slope of a function (rate of change) at a given value for variable *x*.

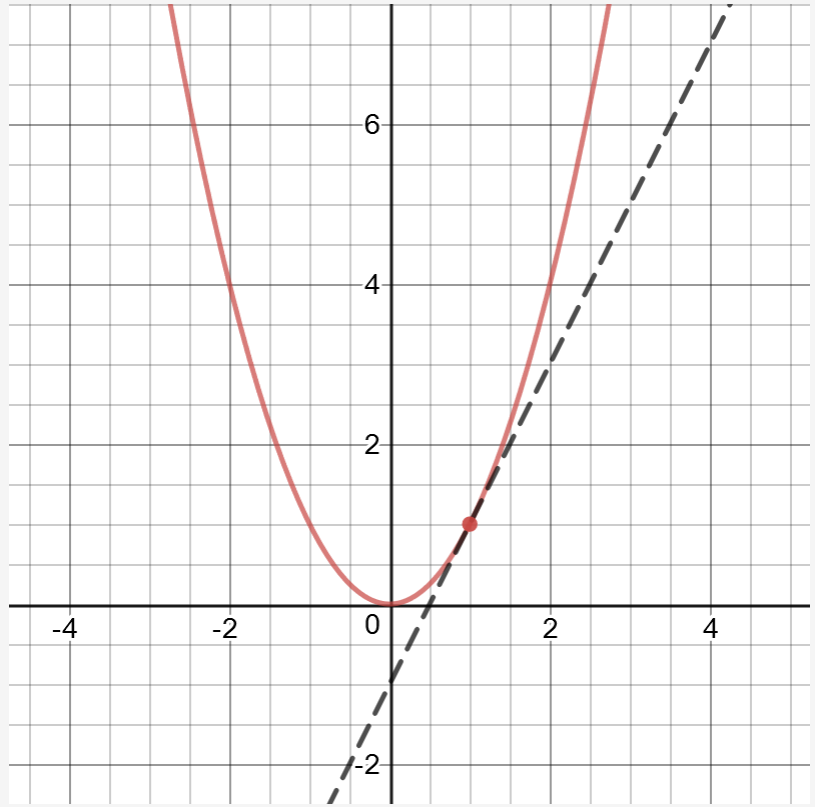
For example, the derivative of x^2 is $2x$.

$$\frac{\mathrm{d}}{\mathrm{d}x}x^2 = 2x$$

This means at $x = 1$, the slope of the function is 2.

- At $x = 2$, the slope is 4.
- At x = 3, it is 6.
- And so on…

Derivatives can be very useful for optimization. Can you imagine why? (Hint: where are we if the slope is 0?)



https://www.desmos.com/calculator/ruvtlp9zk6

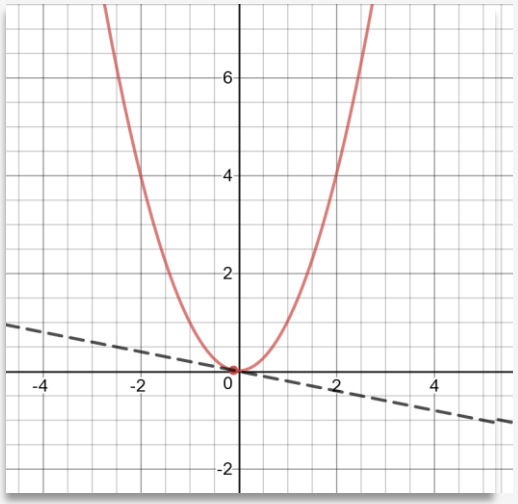
The Essence of Gradient Descent

Notice how when the slope is zero, we are at a local minimum/maximum!

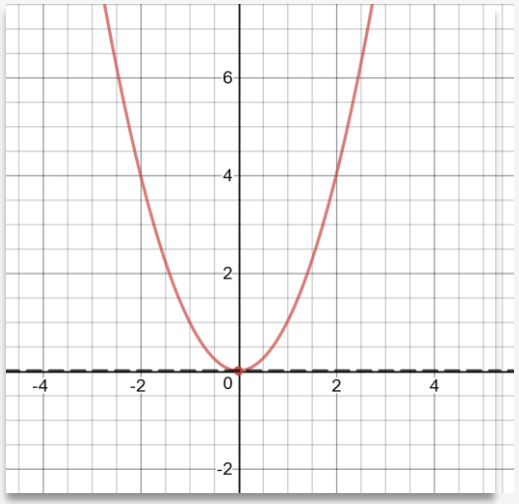
As the slope gets closer to zero, the closer we are to the local minimum /maximum.

Oppositely, the larger/steeper the slope the farther we are from the local minimum.

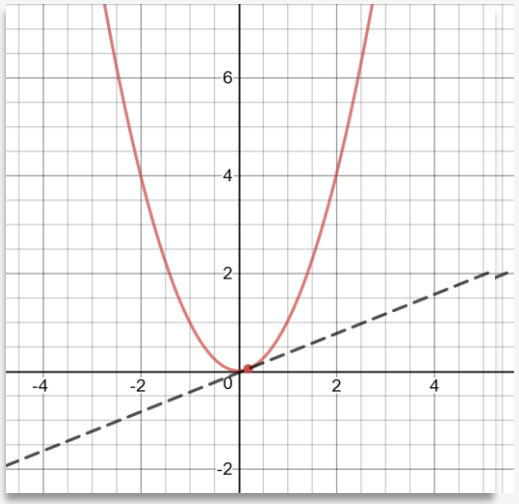
m =-0.4



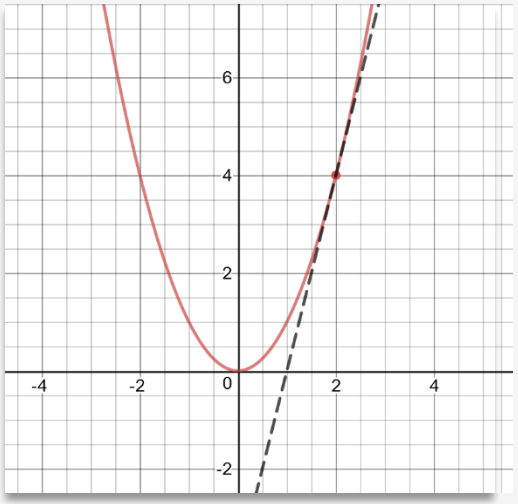
m = 0



m = 0.4



m = 4.0



https://www.desmos.com/calculator/ruvtlp9zk6

The Essence of Gradient Descent

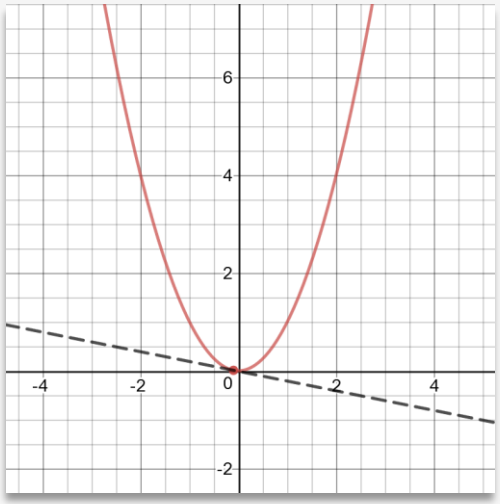
Starting our search in a random or arbitrary location, we want it to *step* towards the local minimum.

The slope is like a compass providing direction to the local minimum/maximum. We step in directions that make the slope smaller, leading it to 0.

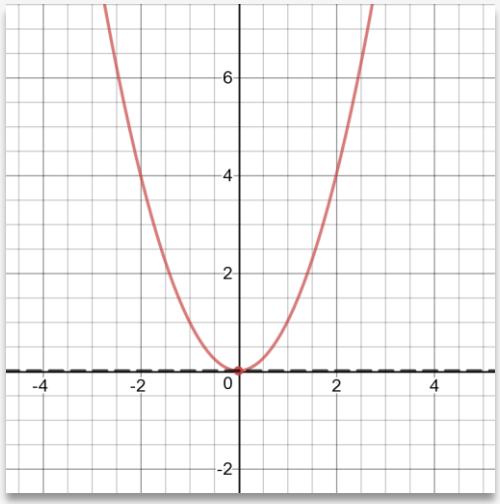
To make this process quicker, we can take bigger steps for bigger slopes, and smaller steps for smaller slopes.

This is the basic idea behind gradient descent.

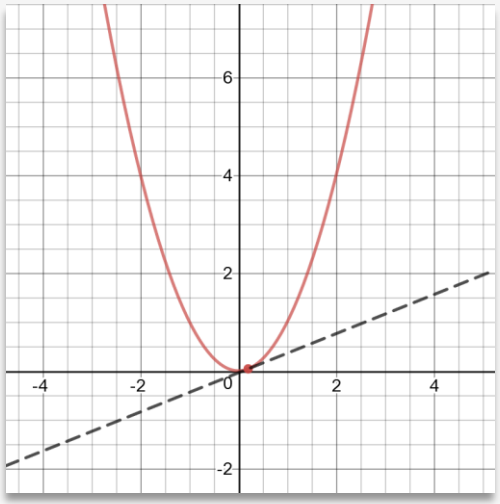
m =-0.4



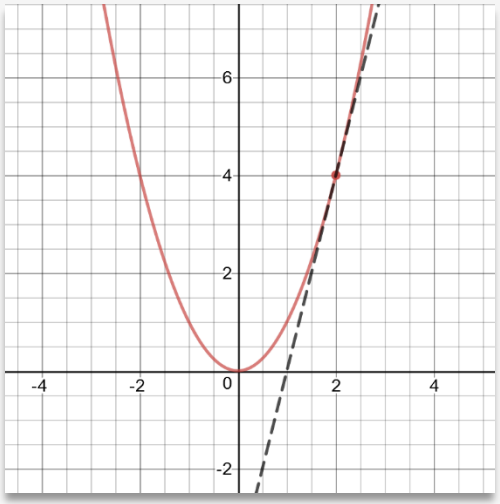
m = 0



m = 0.4



m = 4.0

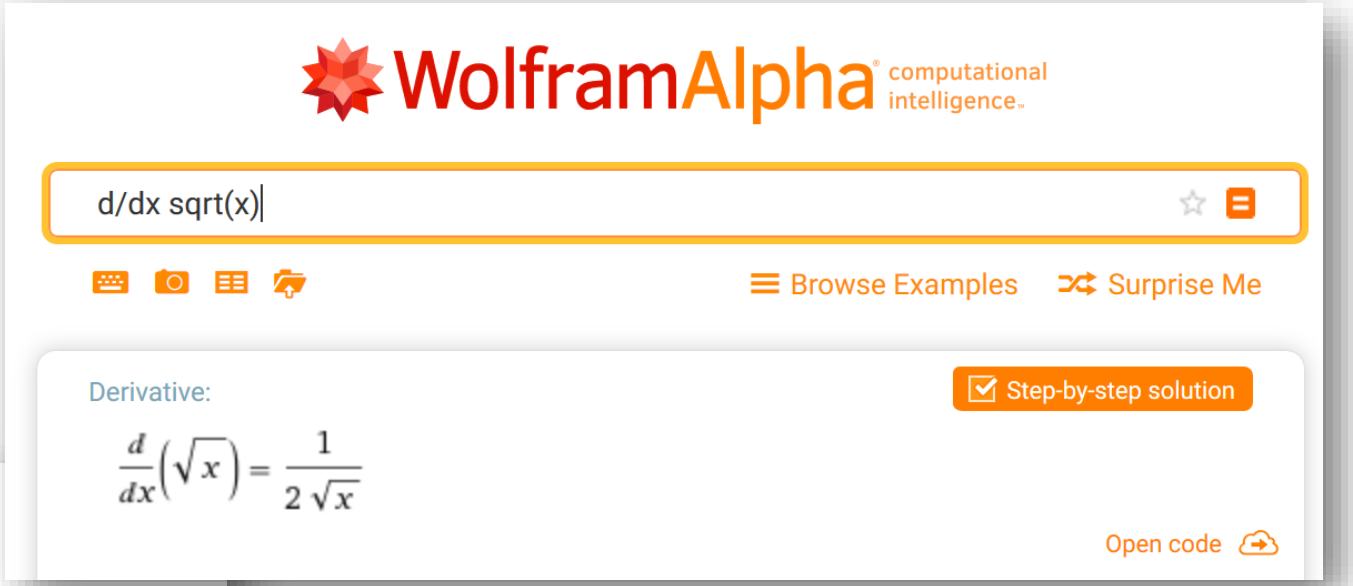


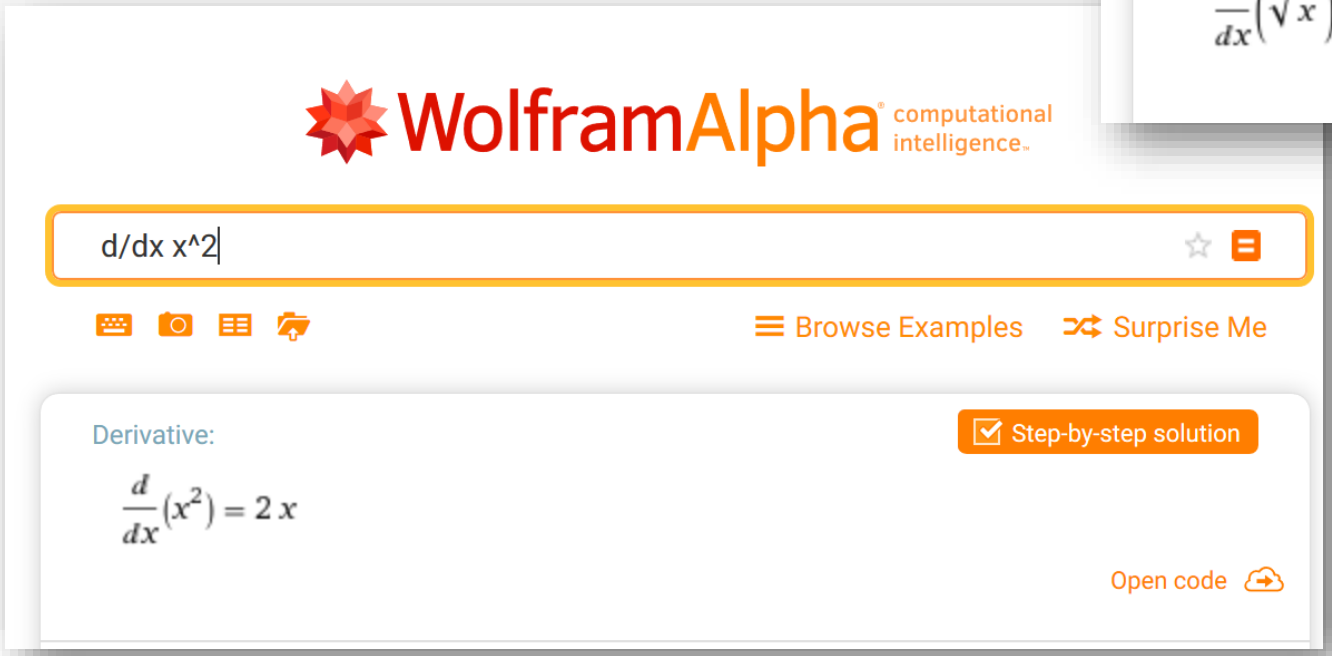
https://www.desmos.com/calculator/ruvtlp9zk6

A Crash Course in Calculus Derivatives

Calculus and algebra is tedious to do by hand…

So just use https://www.wolframalpha.com to calculate derivatives.





Gradient Descent with One Variable

Let's try it!

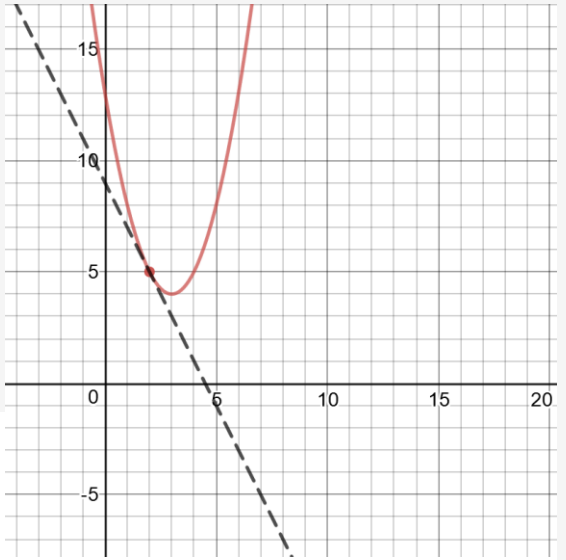
$$f(x) = (x-3)^2 + 4$$

$$\frac{d}{dx} f(x) = 2(x-3)$$

While we could algebraically find the local minimum by setting $2(x-3) = 0$, let's try it with gradient descent instead.

Using gradient descent, we roughly get x=3, y=4 for the local minimum.

https://www.desmos.com/calculator/io531xdcrn

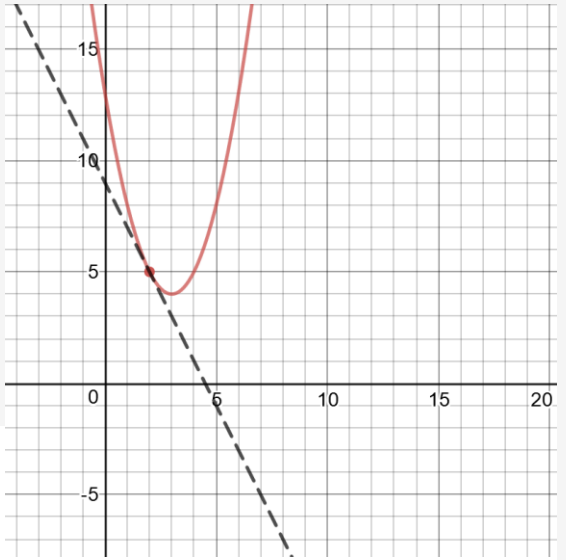



```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001  # The learning rate
epochs = 100000  # The number of iterations to perform gradient descent

x = random.randint(-15,15)   # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x  # update x by subtracting the (learning rate) *
(slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

Gradient Descent with One Variable

Let's walk this through:

We declare our function and derivative of our function.



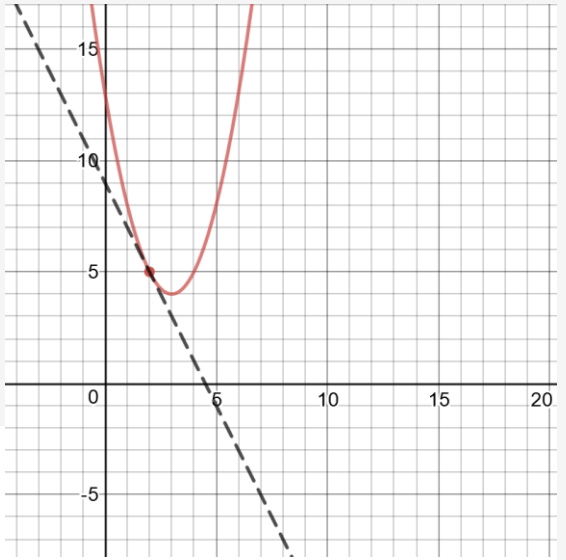

```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001  # The learning rate
epochs = 100000  # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x  # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

Gradient Descent with One Variable

Let's walk this through:

L is our **learning rate**, which scales how small our steps are. Smaller value means smaller steps. Having this too small can make this slow and require more iterations. Having it too large will cause it to keep stepping over the local minimum.

Epochs is a fancy name for number of iterations. You need enough so the search doesn't stop prematurely.




```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001  # The learning rate
epochs = 100000  # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x  # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

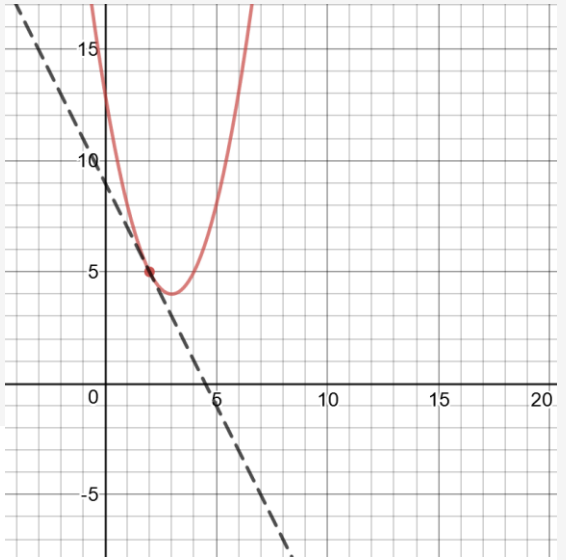
Gradient Descent with One Variable

Let's walk this through:

We start x at a random or arbitrary location.

If there are several local minimums, this could have an impact on our answer.

Fortunately, this problem only has one local minimum.



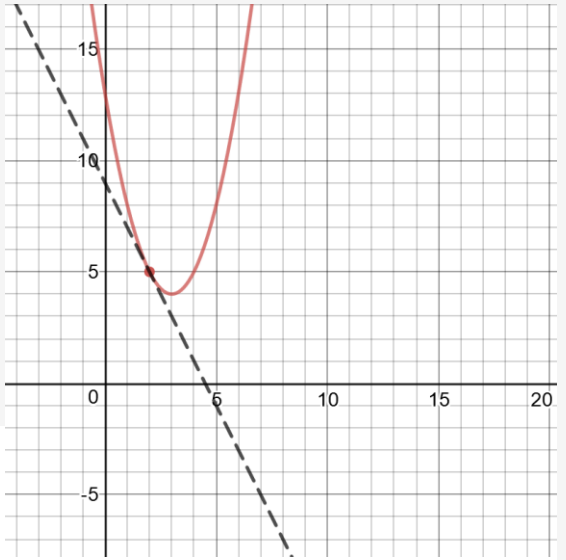

```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001  # The learning rate
epochs = 100000  # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x  # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

Gradient Descent with One Variable

Let's walk this through:

We use the derivative to calculate the slope and figure out which direction to step *x* towards and by how much.

This means repeatedly subtracting the (slope * learning rate) from *x* to progress towards the minimum.



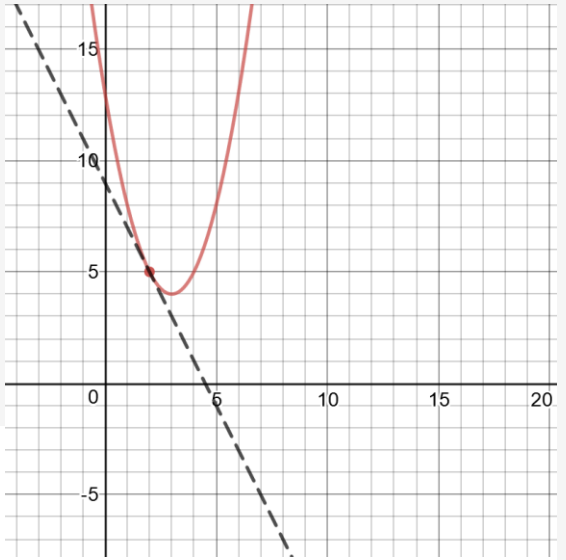

```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001  # The learning rate
epochs = 100000  # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x  # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

Gradient Descent with One Variable

Let's walk this through:

Finally, if our parameters are right we reach an acceptable local minimum. In this case we can easily eyeball from the graph it's *x=3, y=4*.

Our gradient descent algorithm got close enough at *x=2.999999999999889, y=4.0*




```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001  # The learning rate
epochs = 100000  # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x  # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables. So we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

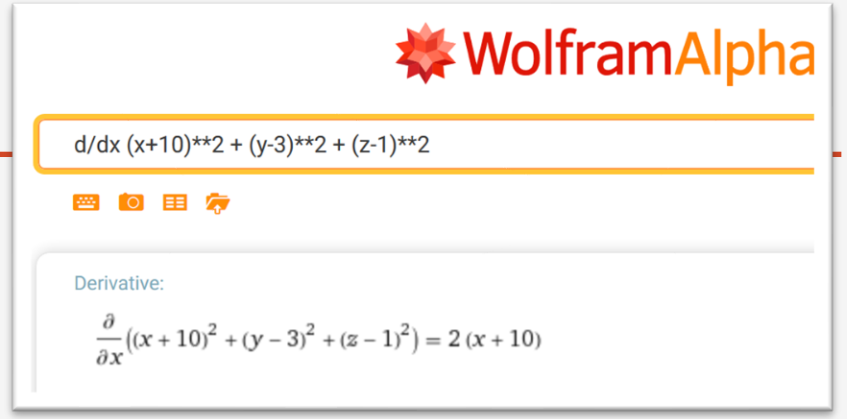
$$f(x, y, z) = (x + 10)^2 + (y - 3)^2 + (1 - z)^2$$

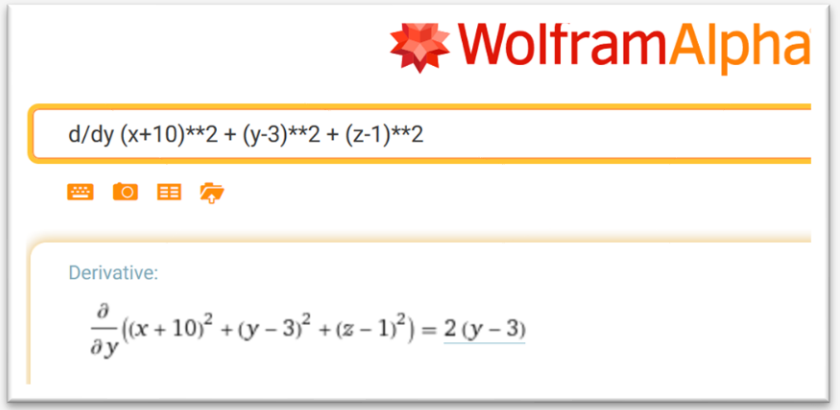
This function has several variables, and we will want to isolate just one of the variables and find its slope.

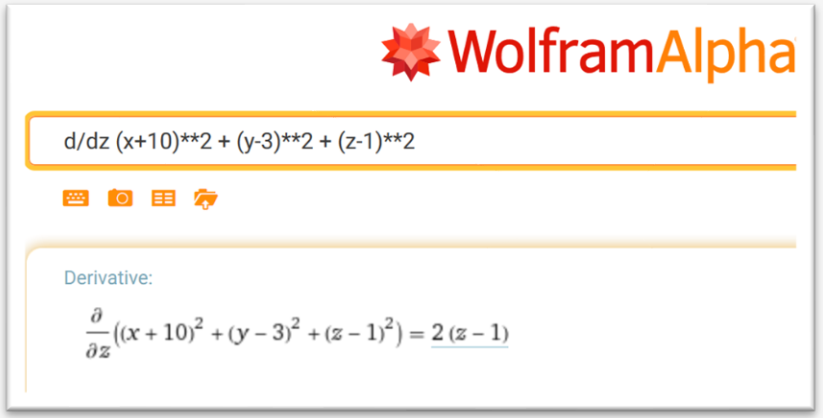
$$\frac{\mathrm{d}}{\mathrm{d}x} f(x, y, z) = 2(x + 10)$$

This is called a partial derivative, and it helps identify the rate of change for just that variable.

On WolframAlpha, we can calculate partial derivatives for the each variable using $\frac{\mathrm{d}}{\mathrm{d}x}$, $\frac{\mathrm{d}}{\mathrm{d}y}$, and $\frac{\mathrm{d}}{\mathrm{d}z}$ on the function as shown on the right.







Multivariable Gradient Descent

We now manage three derivatives and not just one, and use them to adjust the three variables respectively.

Using gradient descent, we roughly minimize (x,y,z) to numbers close to (-10, 3, 1) and closely reach a minimum of 0.

Congrats! You are doing multivariable calculus.

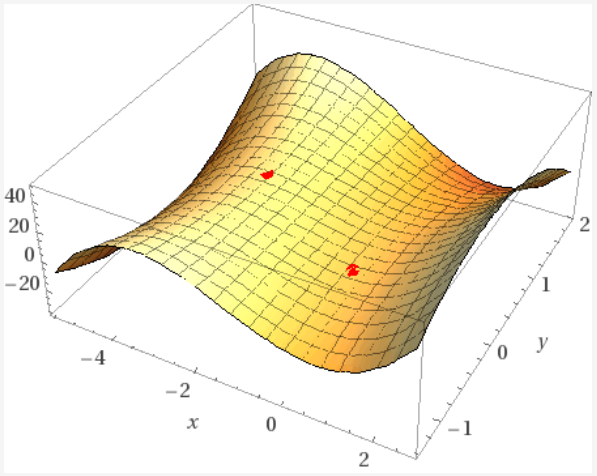
```
def f(x,y,z):
    return (x+10)**2 + (y-3)**2 + (z-1)**2

def dx_f(x):
    return 2*(x+10)

def dy_f(y):
    return 2*(y-3)

def dz_f(z):
    return 2*(z-1)

L = 0.0001  # The learning rate
epochs = 1000000  # The number of iterations to perform gradient descent

x = 0   # we will find the x for the minimum
y = 0   # we will find the y for the minimum
z = 0   # we will find the z for the minimum

for i in range(epochs):
    d_x = dx_f(x) # get x slope
    d_y = dy_f(y) # get y slope
    d_z = dz_f(z) # get z slope

    x = x - L * d_x  # update x
    y = y - L * d_y  # update y
    z = z - L * d_z  # update z

# -9.999999999995559 2.9999999999988898 0.9999999999997224 2.1031154992708616e-23
print(x, y, z, f(x,y,z))
```

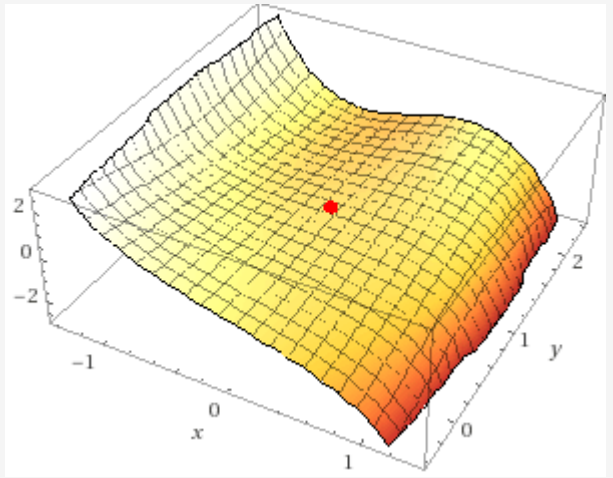
When Slopes of 0 are Not Minimums/Maximums

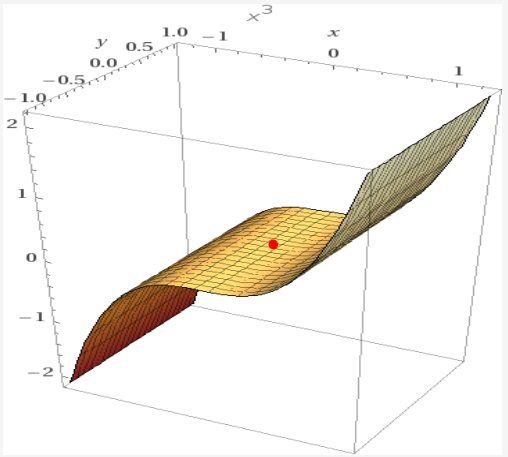
There are some awkward special cases when all variables reach a slope of 0 but are not the local minimum or maximum.

These are called saddle points.

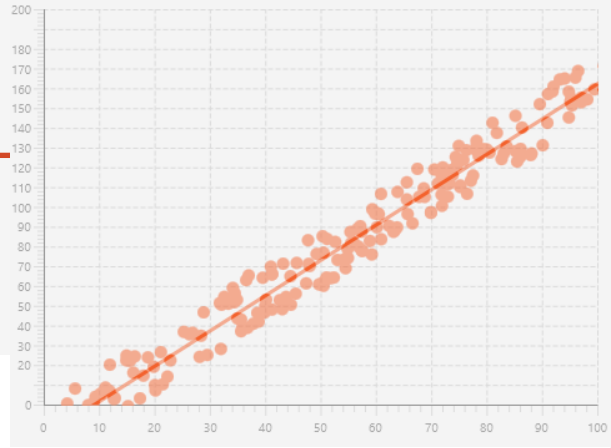
There are methods to overcome these, the simplest being to do multiple random starting points or second partial derivatives, but in the interest of time we will not go here.







Gradient Descent for Linear Regression



Let's step it up a notch. For a simple linear regression fitting to $y = mx + b$, let's use the following mean of squares loss function:

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2$$

The derivative with respect to m is as follows:

$$D_m = \frac{-2}{n}\sum_{1=0}^{n} x_i(y_i - \bar{y}_i)$$

The derivative with respect to b is:

$$D_b = \frac{-2}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)$$

To the right is how we use these two partial derivatives to do gradient descent on a simple linear regression.

```
import pandas as pd

# Input data
data = pd.read_csv('https://tinyurl.com/yaxgfjzt', header=None)
X = data.iloc[:, 0]
Y = data.iloc[:, 1]

# Building the model
m = 0.0
b = 0.0

L = .00001  # The learning Rate
epochs = 1000000  # The number of iterations to perform gradient descent

n = float(len(X))  # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    Y_pred = m * X + b  # The current predicted value of Y
    D_m = (-2 / n) * sum(X * (Y - Y_pred))  # d/dm derivative of loss function
    D_b = (-2 / n) * sum(Y - Y_pred)  # d/dc derivative of loss function
    m = m - L * D_m  # Update m
    b = b - L * D_b  # Update b


print(m, b) # 1.786000956883716 -16.422581112886167
```

Stochastic Gradient Descent

In practice, it can be expensive to process an entire data set at every iteration during training with gradient descent.

Fitting the entire training data on each iteration is also more likely to get stuck in a local minimum/maximum.

This is why stochastic gradient descent is how gradient descent is often used in practice (with neural networks, logistic regression, support vector machines, etc), as it randomly selects only a few training samples at every iteration, adding some randomized benefits to find a better local minimum as well as avoid saddles.

There are interesting variants of stochastic gradient descent, some with simulated annealing-like qualities such as a fluctuating learning rate (similar to temperature).

Learn more:
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Pros and Cons of Gradient Descent

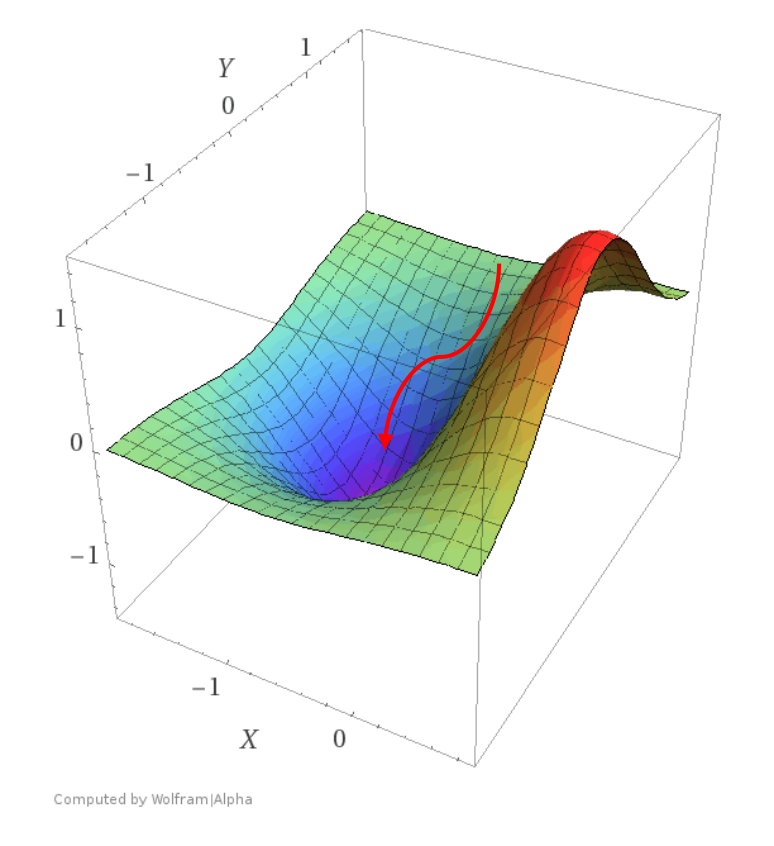
Pros	Cons
• Is able to handle an extremely large number of variables • Can handle nonlinear functions • More guided approach than simulated annealing, relying on mathematical direction towards solution rather than metaheuristic randomness.	• Extremely greedy, can easily get stuck in local minimums without some stochastic behaviors • Requires mathematical functions to describe your model as well as their derivatives. • Processing a lot of data can get expensive, so stochastic sampling may be necessary.

Quiz Time!

Gradient descent guarantees an optimal solution.

A True

B False

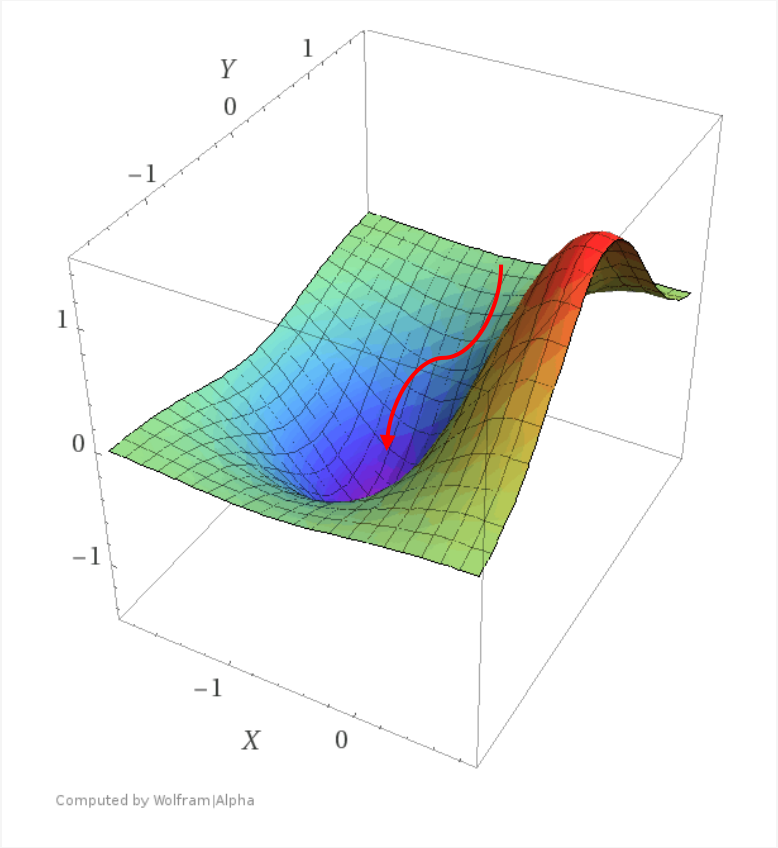


Quiz Time!

Gradient descent guarantees an optimal solution.

A True

B False



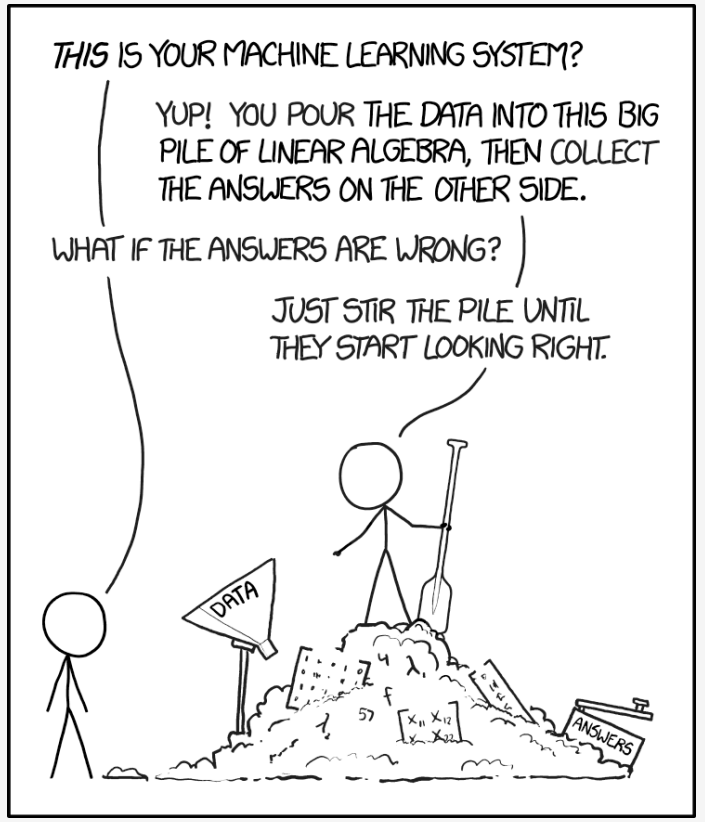
Gradient descent naturally gravitates towards the first local minimum/maximum it detects. It is for this reason stochastic gradient descent is used to add some randomness, and have similar qualities to metaheuristics algorithms.

Quiz Time!

Gradient descent is the only way to optimize a neural network.

A True

B False



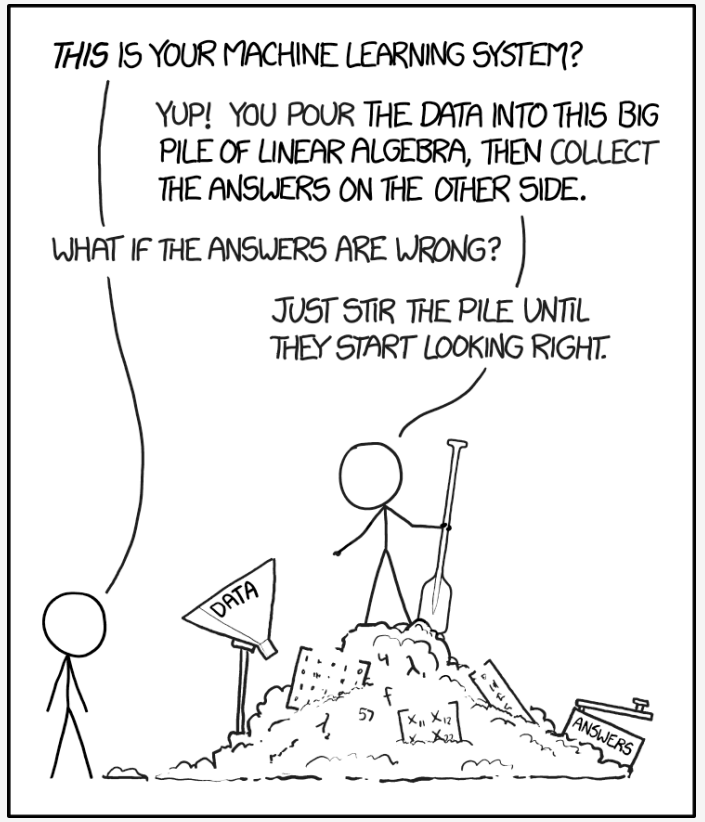
Source: https://xkcd.com/1838/

Quiz Time!

Gradient descent is the only way to optimize a neural network.

A True

B False



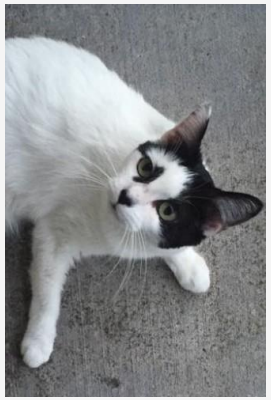
Source: https://xkcd.com/1838/

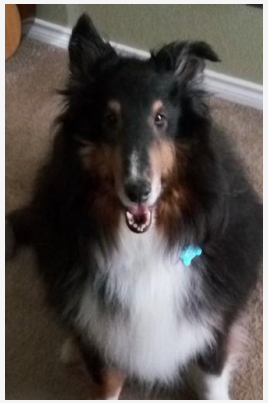
While gradient descent is the most mainstream way to optimize a neural network, metaheuristics algorithms like simulated annealing can also be used with comparable performance. Sometimes simulated annealing can yield a better optimization at the cost of more computation time.

Going Forward

Use the Right “AI” for the Job

Neural Networks



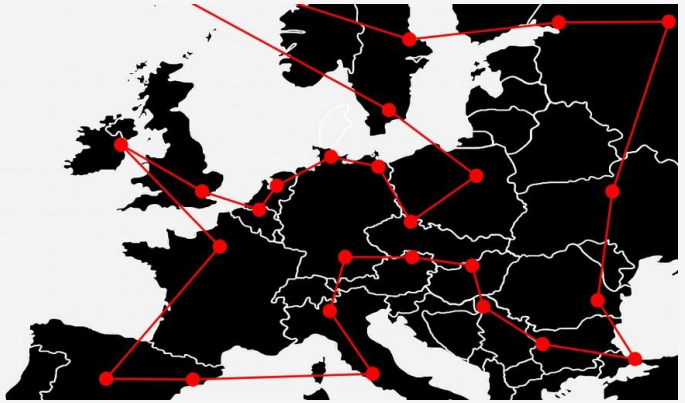


- **Image/Audio/Video Recognition**
  “Cat” and “Dog” photo classifier
- **Natural language processing**
- **Image/Audio generation**

Bayesian Inference

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

Discrete Optimization



- **Routing and Scheduling**
  Staff, transportation, classrooms, sports tournaments, server jobs
- **On-Time Optimization**
  Transportation, manufacturing
- **Industry**
  Manufacturing, farming, nutrition, energy, engineering, finance

Other Online Trainings by Thomas Nield

My other online trainings at O'Reilly:

SQL Fundamentals for Data

Intermediate SQL for Data Analytics

Intro to Mathematical Optimization

Machine Learning from Scratch

Homework Assignments (Answers Are In “code” Folder)

Homework #1

You are trying to create a better guinea pig food blend. For a day's worth of food, you would like it to contain a minimum of 25 grams (g) of fat, 32 g of carbohydrates, and 5 g of protein. But the guinea pig can only have 4 oz of food a day.

You have two existing foods: Food A and Food B, and you want to blend them to create your balanced diet. Food A contains 6g of fat, 13g of carbohydrates, and 3g of protein per ounce, and costs $0.40 per ounce. Food B contains 11g of fat, 14g of carbohydrates, and 2g of protein per ounce, at a cost of $0.30 per ounce.

How many ounces of each food should you mix each day, while minimizing cost?

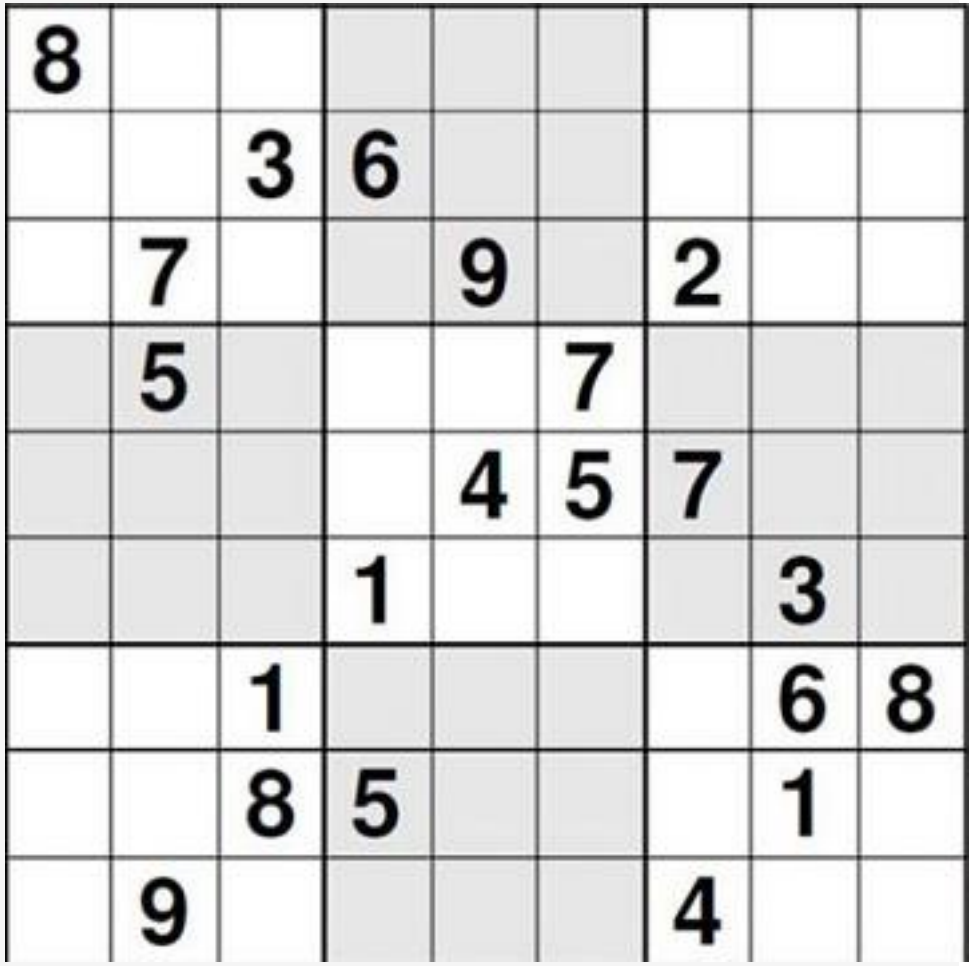
Homework #2

Write a tree search or mixed linear program to crack the world's hardest Sudoku, as described in this article:

https://www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html

You should be able to easily verify the solution by checking each row, column, and square to have the numbers 1-9.

HINT: Using just constrained-based tree search, this particular puzzle may take some time to crack. My Java-based algorithm takes about 2 minutes. You can also use a solver like PuLP if you structure variables in a binary way, which will solve much more quickly.



Homework #3

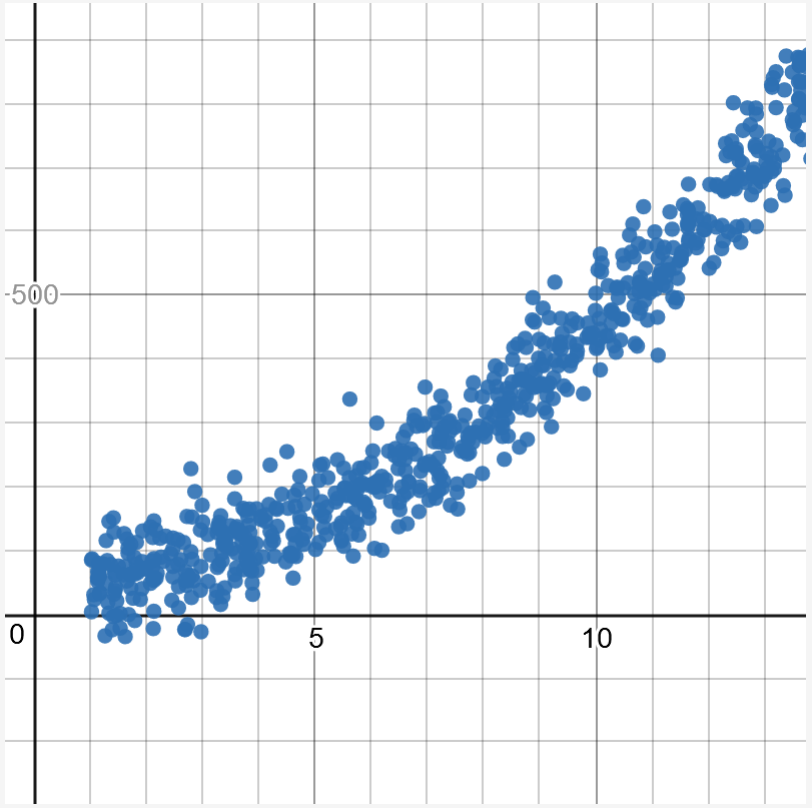
You notice some data (accessible at https://bit.ly/2UBhrMG) has a familiar shape on a scatterplot, and you believe that the best regression will fit to a function

$$y = ax^2 + b$$

where a and b are some constants. Find the a and b constants for the best fit regression based on mean squared error.

HINT: You can use a metaheuristics or gradient descent approach, and there should only be one local minimum for the loss function.

BONUS: Can you also find a $y = ax^2 + b$ curve that not only finds the best fit, but does with 90% of the points are under the curve?



https://www.desmos.com/calculator/canygmx67n

Appendix

Pop Culture

Traveling Salesman **(2012 Movie)**
http://a.co/d/76UYvXd

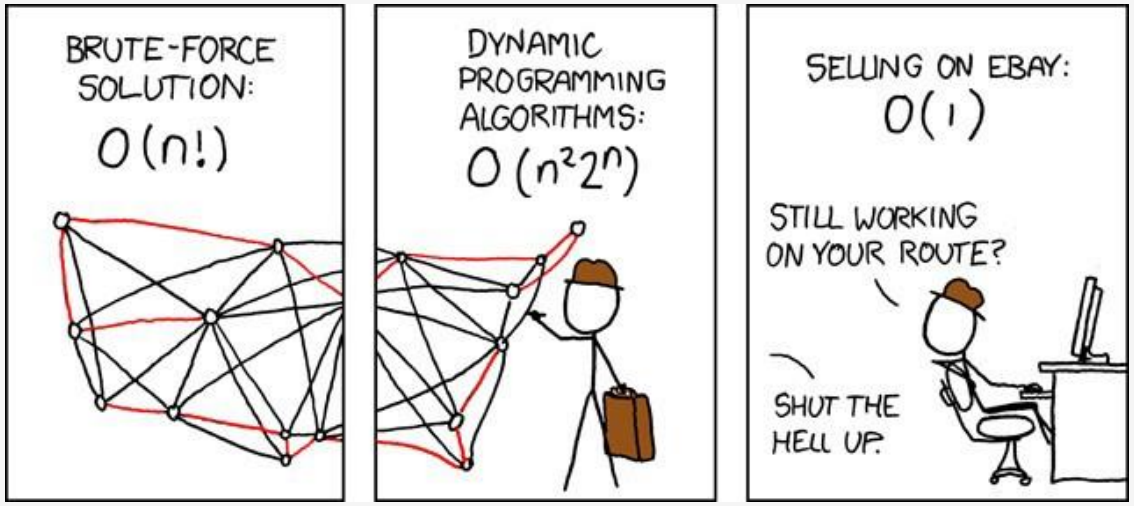
Silicon Valley (HBO) **–The "Not Hotdog" App**
https://youtu.be/vIci3C4JkL0

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

YouTube Channels and Videos

Thomas Nield
https://youtu.be/F6RiAN1A8n0

Brandon Rohrer
https://www.youtube.com/c/BrandonRohrer

3Blue1Brown
https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw

YouTube –P vs NP and the Computational Complexity Zoo
https://youtu.be/YX40hbAHx3s

The Traveling Salesman Problem Visualization
https://youtu.be/SC5CX8drAtU

The Traveling Salesman w/ 1000 Cities (Video)
https://youtu.be/W-aAjd8_bUc

Interesting Articles

Does A.I. Include Constraint Solvers?

https://www.optaplanner.org/blog/2017/09/07/DoesAIIncludeConstraintSolvers.html

Can You Make Swiss Trains Even More Punctual?

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

The SkyNet Salesman

https://multithreaded.stitchfix.com/blog/2016/07/21/skynet-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

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

The Traveling Salesman: A Neural Network Perspective

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