

# Lecture 01 – Introduction

## Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Overview

- 1 What is Optimisation?**
- 2 Spherical rocks and special cases**
- 3 A binary guessing game**
- 4 Lots of applications**

# What is Optimisation?



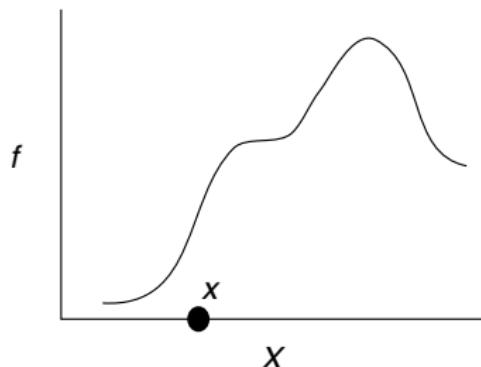
**optimum**: noun, from Latin, *optimus*, “best”

# What is Optimisation?

“The Science of Better” <http://www.scienceofbetter.org/>

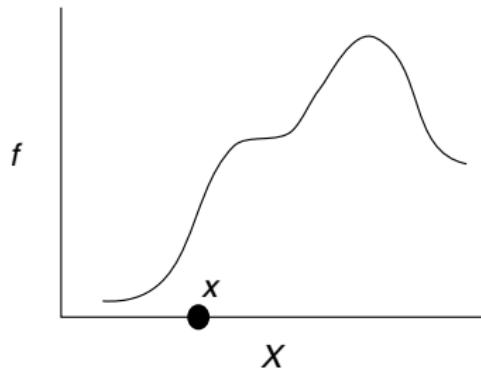
- It's about improving things, reducing costs, maximising profits, saving fuel, reducing wasted time, ...
- Optimisation methods are scientific/mathematical, but the process of formalising a problem can require creativity and intuition.

# What is Optimisation?



We search for  $x \in X$  which maximises the value of  $f : X \rightarrow \mathbb{R}$

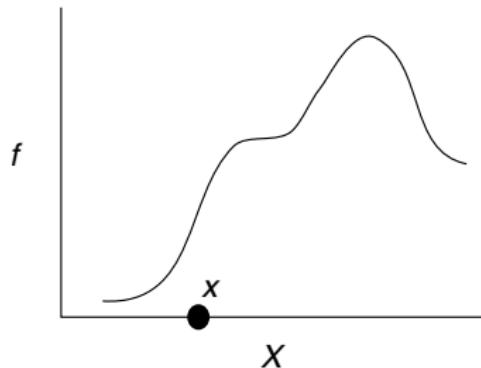
# What is Optimisation?



We search for  $x \in X$  which maximises the value of  $f : X \rightarrow \mathbb{R}$

- $X$  is a **search space**, the set of possibilities.
- $f$  is an **objective function**:  $f(x) \in \mathbb{R}$  measures “how good” is  $x$ .  
E.g., “how much energy is produced by the generator design  $x$ ?”

# What is Optimisation?



We search for  $x \in X$  which maximises the value of  $f : X \rightarrow \mathbb{R}$

- $X$  is a **search space**, the set of possibilities.
- $f$  is an **objective function**:  $f(x) \in \mathbb{R}$  measures “how good” is  $x$ .  
E.g., “how much energy is produced by the generator design  $x$ ?”

In other words, we try to find:  $\arg \max_{x \in X} f(x)$

# What is Optimisation?

**Optimisation = Search**

# What is not Optimisation?

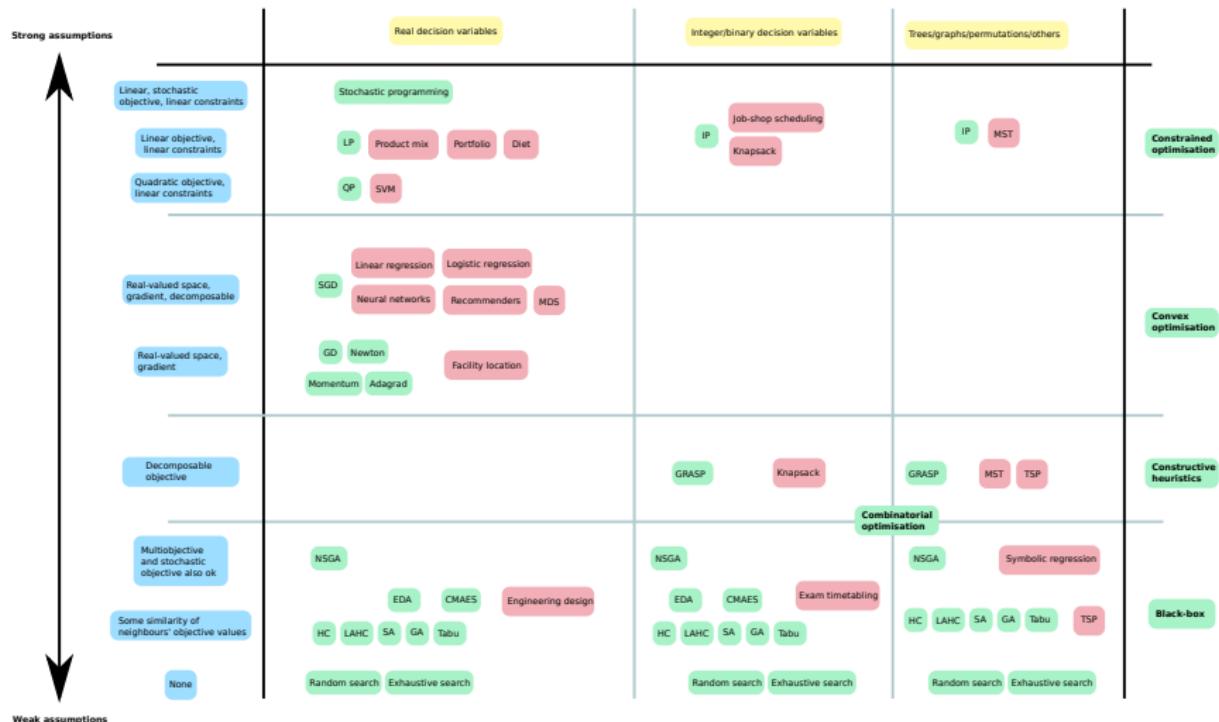
(... for our purposes)

- Rewriting an algorithm to achieve  $O(\log n)$  instead of  $O(n^2)$
- Optimising compilers
- Search in the sense of *pathfinding*, e.g. A\* or breadth-first search.

# Many problems, many algorithms

There are many optimisation problems and many algorithms. The right algorithm may depend on:

- The type of search space (e.g. real vectors, trees, permutations, bitstrings...)
- The properties of the objective (e.g. smooth, has a gradient, convex, multiobjective, ...)
- Constraints (i.e. some solutions are **invalid**)



# Overview

- 1 What is Optimisation?
- 2 **Spherical rocks and special cases**
- 3 A binary guessing game
- 4 Lots of applications

# Example: finding spherical rocks



How come there are lots of jagged rocks, irregular rocks, oval rocks – but no spherical rocks?

# Example: finding spherical rocks



Suppose I wanted to **find** a spherical rock. Suppose I have a robot which can drive around, pick up rocks and scan them with a camera.

How could I use it to find the **most spherical** rock? I want it to do all this automatically, while I have lunch.

# Example: finding spherical rocks

“Scanning” means it produces a data file that describes the 3D shape in detail. If you like, it’s just a cubic 3D array containing a 1 for every filled voxel and a 0 for every empty voxel.

E.g., this is a very small, flat, triangular rock:

```
[[[0,1,0],  
 [0,1,1],  
 [0,0,0]],  
 [[0,1,0],  
 [0,1,1],  
 [0,0,0]],  
 [[0,0,0],  
 [0,0,0],  
 [0,0,0]]]
```

# Example: finding spherical rocks

Suppose we have a function that calculates `sphericalness(x)` for any given rock `x` in this 3D format. We can then just program the robot to try every single rock in the collection `X`, as follows:

```
best = None
bestf = -1
for x in X:
    if sphericalness(x) > bestf:
        best = x
        bestf = sphericalness(best)
```

# Example: finding spherical rocks

Suppose we have a function that calculates `sphericalness(x)` for any given rock `x` in this 3D format. We can then just program the robot to try every single rock in the collection `X`, as follows:

```
best = None
bestf = -1
for x in X:
    if sphericalness(x) > bestf:
        best = x
        bestf = sphericalness(best)
```

Python gives us a shorthand for that common idiom:

```
max(X, key=sphericalness)
```

# Enumerative search

```
max(X) # X a list of numbers
```

# Enumerative search

```
max(X) # X a list of numbers
```

```
max(X, key=f) # pass each x to f
```

# Enumerative search

```
max(X) # X a list of numbers
```

```
max(X, key=f) # pass each x to f
```

This is  $\arg \max_{x \in X} f(x)$ . Enumerative search is **guaranteed** to find the **optimum** – the very best point in  $X$  – because it tries every point in  $X$ .

# An objective function: sphericalness

So the only hard part is to define sphericalness. How would you do it?

# Reflection

- If  $X$  is very large, what happens?
- Is there any way to get a (fairly good) result much faster?

# When would we use enumerative search?

- If we need the optimum, not just a “very good” point
- If we have no “smarter” algorithm
- If time/computational budget allows
- What about computational complexity?

# When would we use enumerative search?

- If we need the optimum, not just a “very good” point
- If we have no “smarter” algorithm
- If time/computational budget allows
- What about computational complexity?



It's a trap!

# Random search

```
def rand_search(X, f, its=10000):
    return max(random.sample(X, its), key=f)
```

its is short for **iterations**.

# Random search

```
def rand_search(X, f, its=10000):
    return max(random.sample(X, its), key=f)
```

its is short for **iterations**.

- Disadvantage: no guarantee of finding the optimum
- Advantage: can be as fast as we want

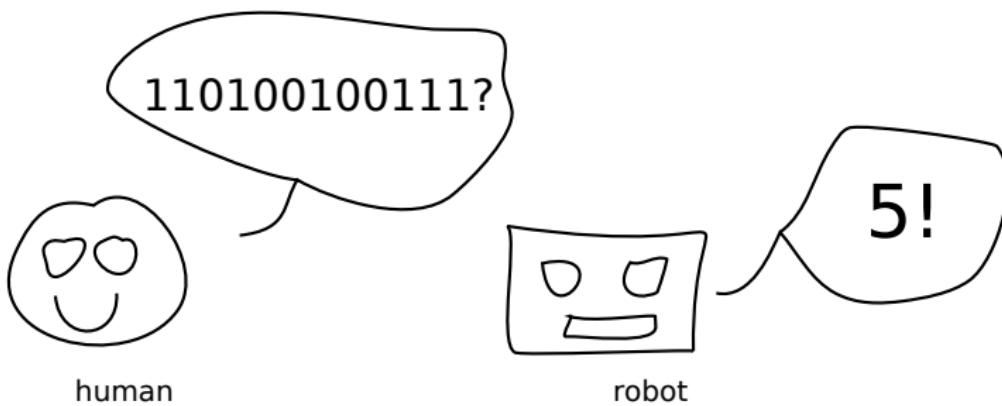
# Reflection

- Random search is not very smart
- When we observe sphericalness ( $x_1$ ) for a particular rock  $x_1$ , what does it tell us about sphericalness ( $x_2$ ) for the next rock  $x_2$ ?

# Overview

- 1 What is Optimisation?
- 2 Spherical rocks and special cases
- 3 A **binary guessing game**
- 4 Lots of applications

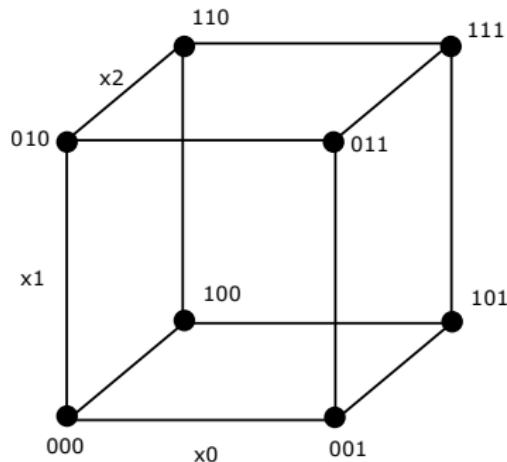
# A guessing game (is an optimisation problem)



- A robot chooses a secret bitstring of length  $n$
- We try to guess the bitstring
- Each time we guess, the robot tells us **how many bits are correct.**
- The bits are called **decision variables**.

# What is the search space?

$X = \mathbb{H}^n$  (Hamming cube in  $n$  dimensions, i.e. space of bitstrings of length  $n$ )



# What is the objective function?

**Maximise:**  $f(x)$  = number of bits correct

# What is the objective function?

**Maximise:**  $f(x)$  = number of bits correct

Or equivalently, **minimise:**  $f(x) = d_H(t, x)$ , the **Hamming distance** between the guess  $x$  and the computer's target  $t$ .

# What is the objective function?

**Maximise:**  $f(x)$  = number of bits correct

Or equivalently, **minimise:**  $f(x) = d_H(t, x)$ , the **Hamming distance** between the guess  $x$  and the computer's target  $t$ .

secret	110100100111
guess	110000110111
<hr/>	
differences	000100010000
Hamming distance	2

# Hill-climbing

```
def hill_climb(init, nbr, f, its=10000):
    x = init()
    for i in range(its):
        xnew = nbr(x) # try a new point...
        if f(xnew) > f(x):
            x = xnew # improvement!
    return x

def init(): # random bitstring
    return [random.randrange(2) for i in range(n)]
def nbr(x): # make a "neighbour" of x
    x = x.copy()
    i = random.randrange(n)
    x[i] = 1 - x[i] # flip a random bit
    return x
```

# Hill-climbing

- Hill-climbing may be the simplest true optimisation algorithm
- Notice that the structure is similar to our `max` code: initialise  $x$ , then every time we find an improvement, replace  $x$ .
- We'll study this properly in a few weeks.

# The curse of dimensionality

The curse of dimensionality: when the number of dimensions grows, every problem (machine learning, optimisation, search, ...) gets harder, and soon becomes intractable.

# Reflection

- 1 In the guessing game, how does the size of the space grow as bitstring length  $n$  grows?

# Reflection

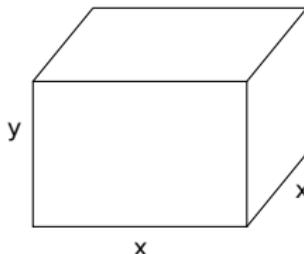
- 1 In the guessing game, how does the size of the space grow as bitstring length  $n$  grows?
- 2 For each guess  $x$ , we received  $f(x)$ , a single number.

What if, instead, we received more information – e.g. exactly **which** bits are incorrect?

# Overview

- 1** What is Optimisation?
- 2** Spherical rocks and special cases
- 3** A binary guessing game
- 4** Lots of applications

# Applications: Design



A box having a square base and an open top is to contain 108 cubic feet. What should its dimensions be to minimise materials? From

[http://www.themathpage.com/  
aCalc/applied.htm](http://www.themathpage.com/aCalc/applied.htm).

We use “good old differentiation”: we formulate  $M$ , the amount of materials, as a function of  $x$  (the side of the base):  $M = x^2 + 432/x$ . We can just set  $dM/dx = 0$  (and also check the second derivative).

# Applications: Travelling salesman problem

$X$  is the set of possible tours of some cities.  $f$  is a measure of **total tour distance**. We might use a **constructive heuristic**.



# Applications: Energy production schedules



Given data on regional electricity requirements and power plants, design a schedule to switch plants on/off to meet demand.  $X$  is all possible schedules.  $f_1$  represents cost of fuel, and  $f_2$  total emissions. (The “unit commitment” problem.) We might use **multiobjective optimisation**.

# Applications: Timetabling

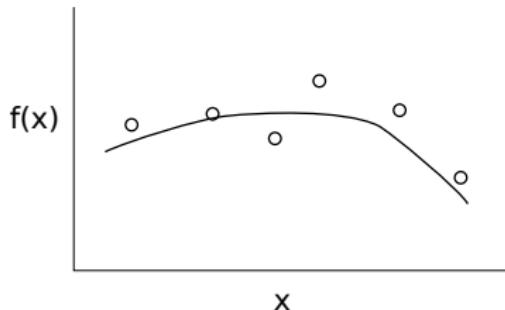
- $X$  is the set of possible exam timetables. We have a **constraint** that any timetable with a class is **invalid**.
- $f(x)$  might measure the number of students who have one exam right after another.
- We might use a **genetic algorithm**.

# Applications: Portfolio allocation

Given a fixed amount of money to be invested and a set of  $n$  stocks to invest in, we want to maximise expected return and minimise risk.

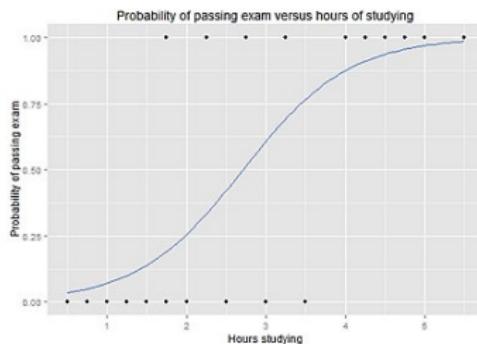
- $X \subset \mathbb{R}^n$  is the set of possible allocations. There are  $n$  decision variables.
- $f$  is a measure of expected return and risk.
- We might use **linear programming**.

# Applications: Curve-fitting



- Given a non-linear regression model  $f(x)$  with fixed form but unknown parameters  $c$
- e.g.  $f(x) = c_0 x^{c_1} \cdot \log(c_2 x)$
- and given training data  $X$  and  $y$
- we try to find the parameters  $c$  which minimise the loss  $\sum_i(f(x_i) - y_i)^2$ . These are the **decision variables** in our optimisation.
- We might use **non-linear least squares**.

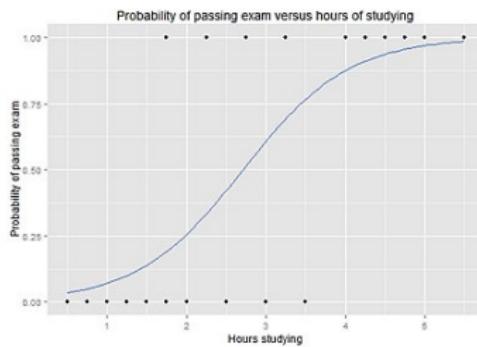
# Applications: Machine Learning



Wiki

- Logistic regression:  $X$  is the set of possible **weight values** for the LR model.  $f$  might be the **loss** on training data. We might use **gradient descent**.

# Applications: Machine Learning



Wiki

- Logistic regression:  $X$  is the set of possible **weight values** for the LR model.  $f$  might be the **loss** on training data. We might use **gradient descent**.
- If we have a neural network, we need **gradient descent with backpropagation**.

# Applications: Program synthesis

- Given a set of input-output examples  $(X, y)$ , we try to find a program  $p \in X$  such that  $p(x_i) = y_i$ .  $f$  might measure how many examples our program  $p$  gets correct. We might use **genetic programming**.

# Applications: Cache expiry algorithms

- A size-limited hardware or software cache uses some algorithm to decide which items to throw out when it becomes full, e.g. a least-recently used (LRU) algorithm.
- $X$  is the set of possible parameters of that algorithm and  $f$  could be a measure of throughput.
- We might use simulation to calculate  $f$ . We might use **covariance matrix adaptation** to search for  $x$ .

# Applications: Menus



From [topsecretrecipes.com](http://topsecretrecipes.com)

- Suppose I'm going to host a party for 10 small kids. They'll eat sausages (EUR5/kg), chips (EUR2/kg), and ice-cream (EUR4/kg). The kids don't care what they get so long as they get 500g of food each. Suppose I don't care about their health. What should I buy?

# Lecture 02 – Linear Programming

## Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Linear programming

In this first section of the module, we'll discuss **linear programming** and its close relative **integer programming**.

# Linear programming

In this first section of the module, we'll discuss **linear programming** and its close relative **integer programming**.

These methods work for real-valued and integer-valued search spaces respectively, and even mixed ones. They require strong assumptions:

- the search space is composed of **decision variables**, either real or integer-value;
- **linearity** of the objective in the decision variables;
- there are **constraints** which mean that large parts of the search space are **infeasible**. Constraints are expressed as **linear** equations and inequalities in the decision variables.

# Overview

- 1 What is linear programming? A motivating problem**
- 2 Graphical solution in 2D**
- 3 More applications**
- 4 A little theory**

# Terminology

An LP is a **mathematical program**, a specific type of optimisation problem:

- a set of decision variables
- a set of **constraints**
- an objective function to be minimised (or maximised).

# Terminology

Terms such as **dynamic programming**, **mathematical programming** and **linear programming** are misleading to modern ears. They **predate** the use of “programming” to mean “writing computer code”.

From [Wikipedia](#):

The term “linear programming” for certain optimization cases was due to George B. Dantzig, although much of the theory had been introduced by Leonid Kantorovich in 1939. (**Programming** in this context does not refer to computer programming, but comes from the use of **program** by the United States military to refer to proposed training and logistics schedules, which were the problems Dantzig studied at that time.) Dantzig published the Simplex algorithm in 1947, and John von Neumann developed the theory of duality in the same year.

# Motivating problem: product mix

# Manufacturing hand sanitizer

Suppose we have a small manufacturing unit for two hand sanitizer products.

- Products have different requirements of a key raw material, of labour, give different profits, etc
- Constraints on labour (time), raw materials, etc
- Want to maximise profits
- How much of each product should we manufacture, ie what **product mix?**

# Manufacturing hand sanitizer

Suppose we have a small manufacturing unit for two hand sanitizer products.

- Products have different requirements of a key raw material, of labour, give different profits, etc
- Constraints on labour (time), raw materials, etc
- Want to maximise profits
- How much of each product should we manufacture, ie what **product mix?**

## Decision variables

- $x_1$  is the quantity of Product 1 to make (in L)
- $x_2$  is the quantity of Product 2 to make (in L)

# Formalising the objective

- Product 1 gives profit €15/L
- Product 2 gives profit €10/L
- Total profit = profit from Product 1 + profit from Product 2
- Maximise  $15x_1 + 10x_2$

# Constraints

If some values for decision variables are not allowed, that is a **constraint**

- Maximum demand for Product 1 is 18L
- Maximum demand for Product 2 is 30L
- Maximum supply of active ingredient is 2.2L
- Each unit of either product requires 0.1L of active ingredient
- Product 1 requires 1.5 hours machine time
- Product 2 requires 2.5 hours machine time
- The machine operator cannot work more than 45 hours per week
- Cannot make a negative amount of either Product

# Formalising the constraints

- Demand for Product 1:  $x_1 \leq 18$
- Demand for Product 2:  $x_2 \leq 30$
- Labour:  $1.5x_1 + 2.5x_2 \leq 45$
- Active ingredient:  $0.1x_1 + 0.1x_2 \leq 2.2$
- Non-negativity:  $x_1 \geq 0$  and  $x_2 \geq 0$

# Formalising the problem

Maximise profit:  $15x_1 + 10x_2$

Subject to:  $0.1x_1 + 0.1x_2 \leq 2.2$  (raw materials)

$1.5x_1 + 2.5x_2 \leq 45$  (labour)

$x_1 \leq 18$  (demand for Product 1)

$x_2 \leq 30$  (demand for Product 2)

$x_1 \geq 0$  (non-negativity)

$x_2 \geq 0$  (non-negativity)

# Example

- A potential solution: 2 units of Product 1 and 2 units of Product 2
- Put the point (2, 2) into the labour constraint:

$$\begin{aligned}1.5x_1 + 2.5x_2 \\= 1.5 \times 2 + 2.5 \times 2 \\= 3 + 5 & \qquad \qquad = 8 \leq 45\end{aligned}$$

- This solution satisfies this constraint...
- ... in fact it satisfies all constraints...
- ... but is not optimal.

# Feasible solutions

- A potential solution which satisfies all constraints is called **feasible**
- The **best** feasible solution is the optimum.

# Overview

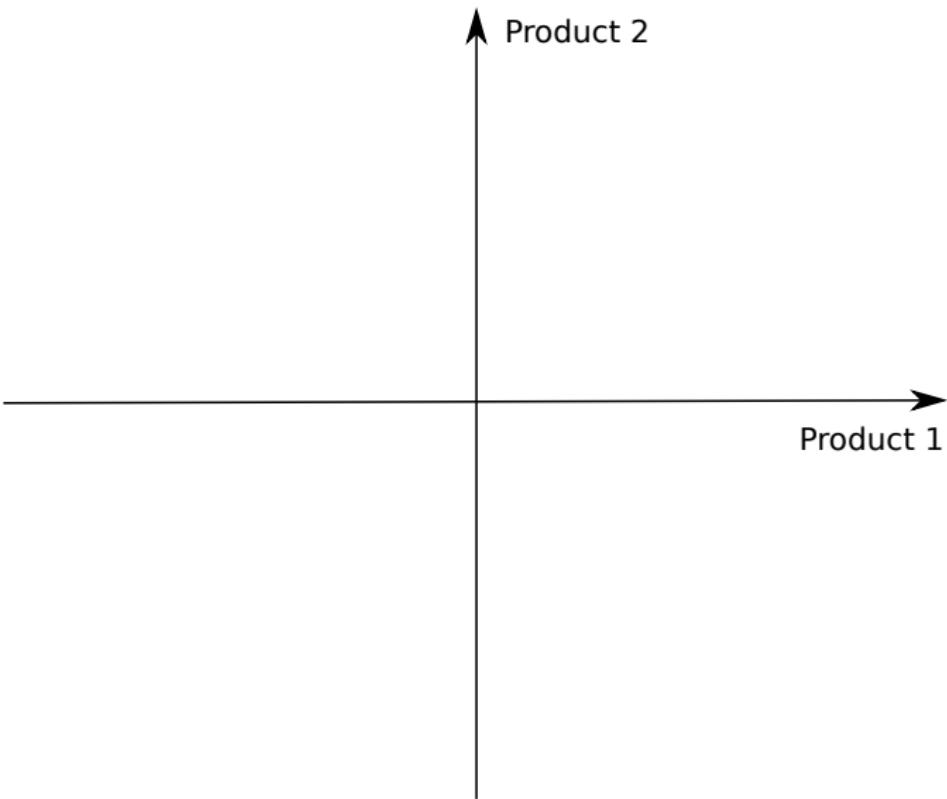
- 1 What is linear programming? A motivating problem
- 2 **Graphical solution in 2D**
- 3 More applications
- 4 A little theory

# Graphical solution in 2D

Of course, this works only with 2 decision variables:

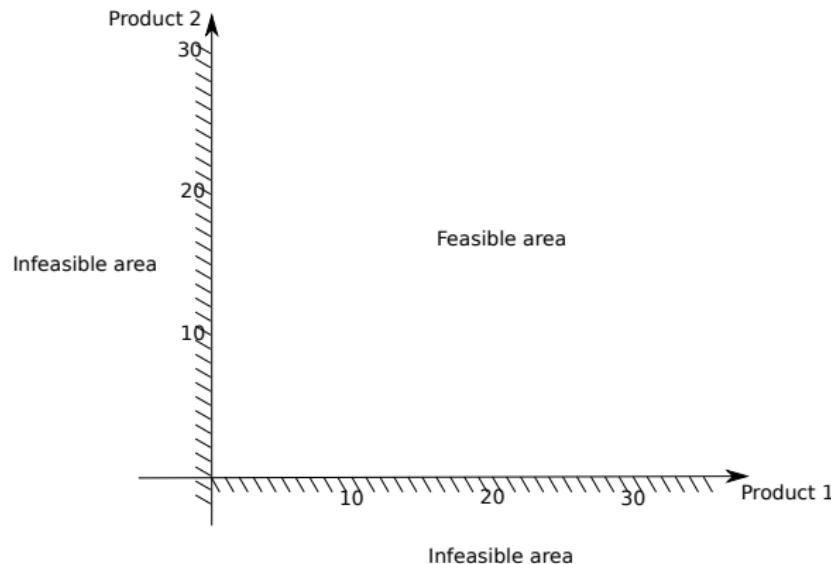
- 1 Draw a 2D graph
- 2 For each constraint, draw it as a line **cutting out half the plane**
- 3 Find the **feasible area**
- 4 Draw some contour lines: **lines of equal profit**
- 5 Identify the best corner point and find value of objective function there

# Step 1: 2D graph



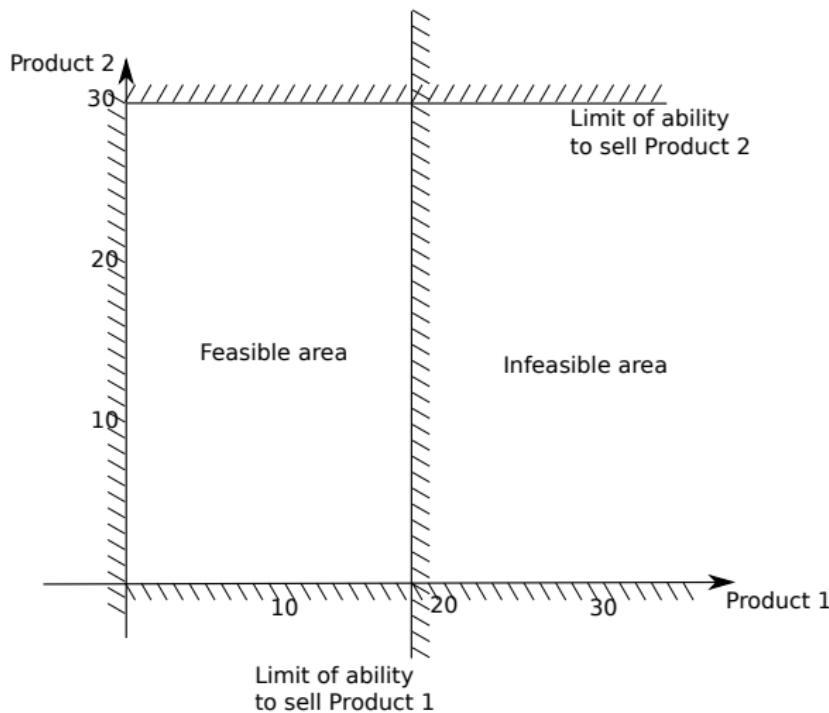
## Step 2: Draw constraints

Non-negativity:  $x_1 \geq 0, x_2 \geq 0$



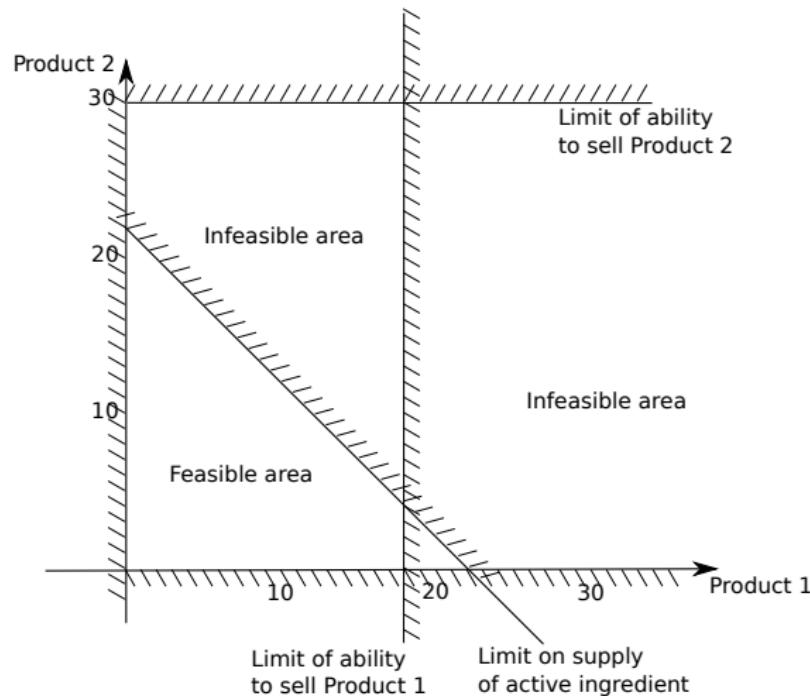
## Step 2: Draw constraints

Demand:  $x_1 \leq 18$ ,  $x_2 \leq 30$



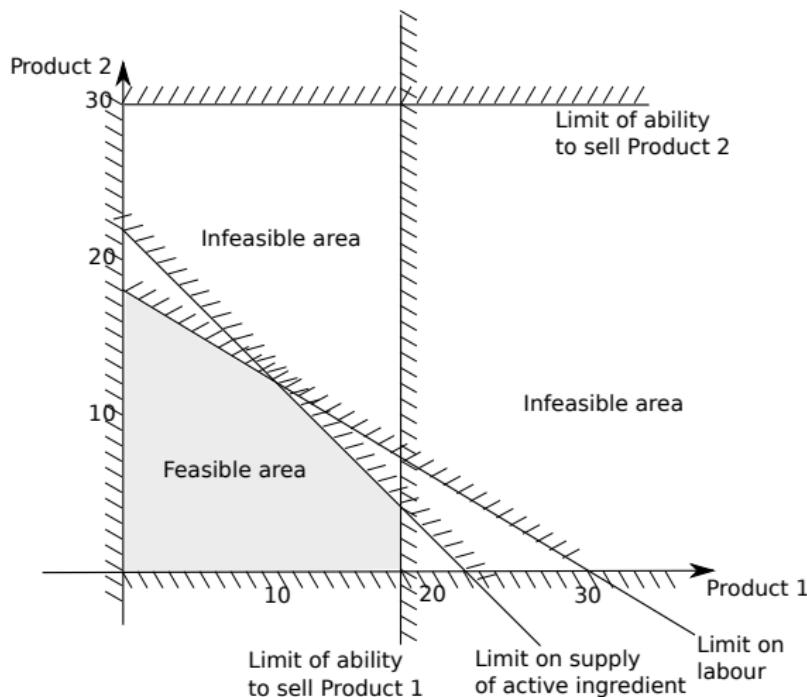
## Step 2: Draw constraints

Supply of active ingredient:  $0.1x_1 + 0.1x_2 \leq 2.2$



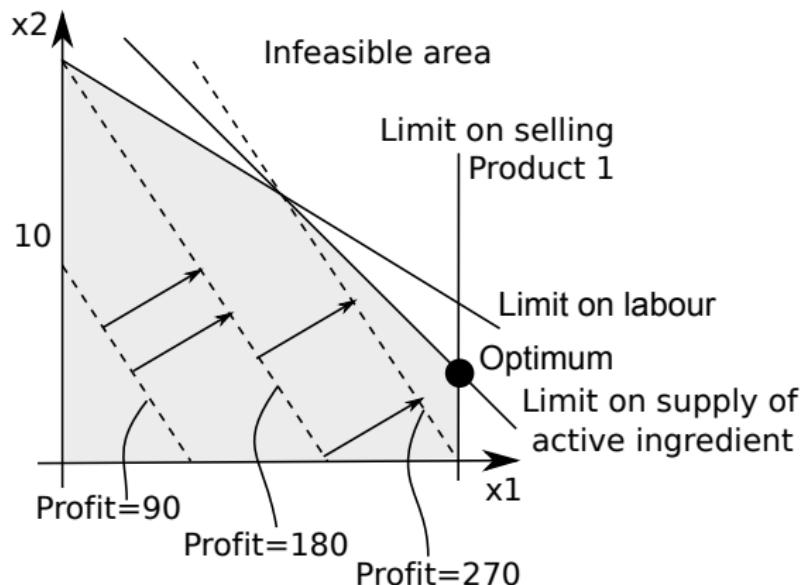
## Step 3: find the feasible area

$$\text{Labour: } 1.5x_1 + 2.5x_2 \leq 45$$



## Step 4: lines of equal profit

- Lines of equal profit (LOEPs) are **contour lines**
- Choose an arbitrary value for profit, e.g. 90
- Objective function:  $15x_1 + 10x_2 = 90$ . **Draw this line**
- Repeat for other values, e.g. profit=180, profit=270

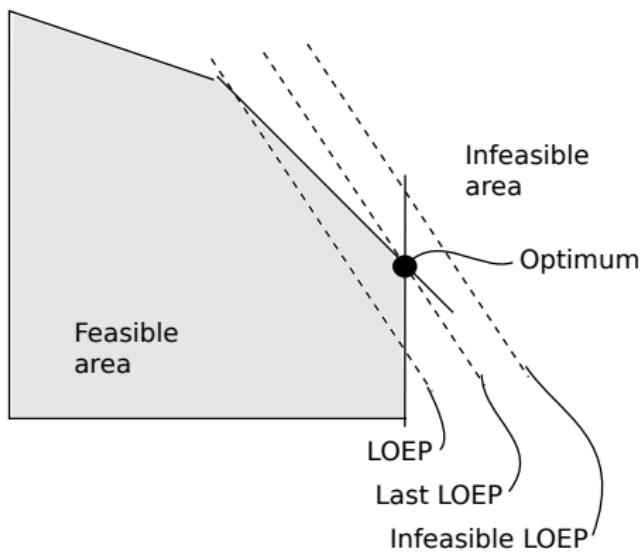


## Step 4: lines of equal profit

- LOEPs are parallel to each other
- Largest profit: rightward and upward
- Each LOEP may have some points in the feasible area, some outside.

## Step 5: find the optimum

- Optimum will always lie at a corner, where two constraints meet
- An LOEP through that point will have no other points in the feasible area
- Any larger LOEP will have no points in the feasible area



## Step 5: find the optimum

- Optimum is at **the corner of two constraint lines**
- Treat them as **equations** (not inequalities)
- Solve them simultaneously

## Step 5: find the optimum

$$0.1x_1 + 0.1x_2 = 2.2 \text{ (raw materials)}$$

$$x_1 = 18 \text{ (demand for P1)}$$

$$\Rightarrow 0.1x_2 = 2.2 - 0.1 \times 18$$

$$\Rightarrow x_2 = 0.4 / 0.1 = 4$$

$\Rightarrow$  Optimum is (18, 4)

## Step 5: find the optimum profit

Optimum: (18, 4)

$$\begin{aligned}\text{Optimum profit: } f(x_1, x_2) &= 15x_1 + 10x_2 \\ &= 15 \times 18 + 10 \times 4 \\ &= \text{EUR}310\end{aligned}$$

## To be careful: test the optimum

- Is it plausible?
- Does it make sense in context of original (verbal) problem?
- Use common sense.

# Overview

- 1 What is linear programming? A motivating problem
- 2 Graphical solution in 2D
- 3 **More applications**
- 4 A little theory

# Exercise: diet problem

In LP we can also **minimize**, e.g. minimize weight:

**Diet problem:** our astronauts need minimum amounts of several nutrients per day: 12mg of calcium, 10mg of zinc, 25mg of iron.

- **Space-biscuits** provide 2mg of calcium, 3mg of zinc, and 5mg of iron per serving. They weigh 100g per serving.
- **Astro-smoothies** provide 9mg of calcium, 5mg of zinc, and 12mg of iron per serving. They weigh 300g per serving.

How many servings of each should be supplied per astronaut per day, to minimize the weight?

Formulate this problem and solve it graphically.

# Diet problem: solution

## Decision variables

- $x_1$  servings of space-biscuits
- $x_2$  servings of astro-smoothies

## Constraints

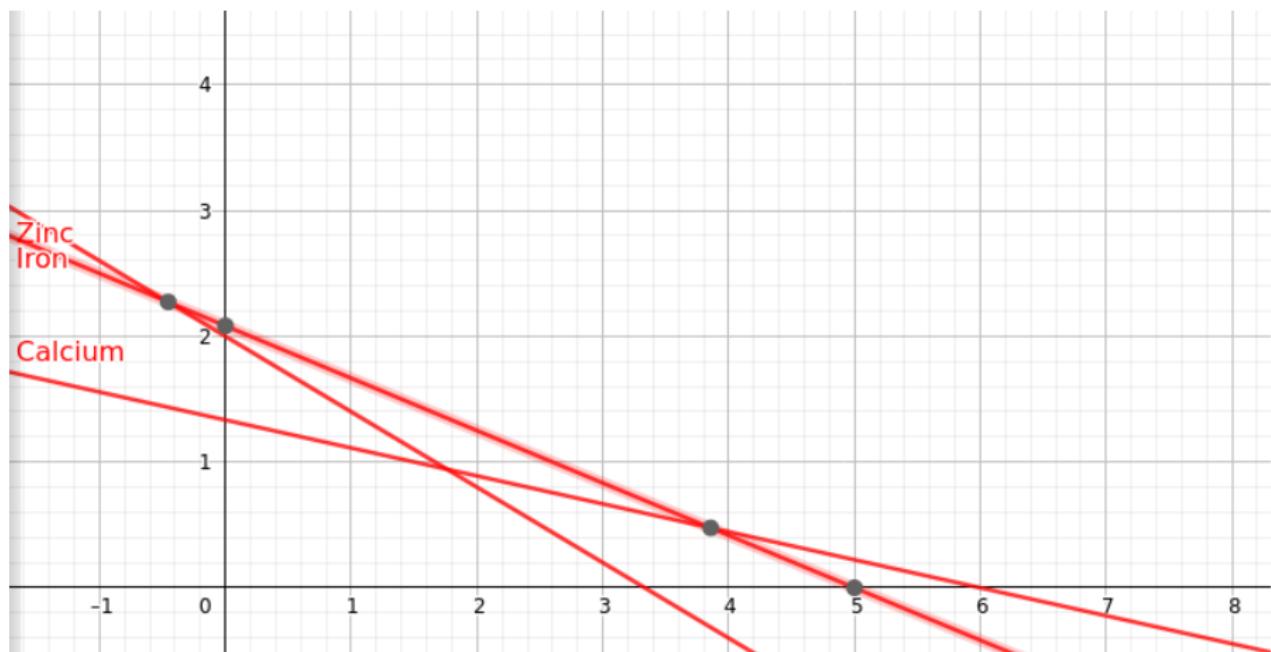
- $2x_1 + 9x_2 \geq 12$  calcium
- $3x_1 + 5x_2 \geq 10$  zinc
- $5x_1 + 12x_2 \geq 25$  iron

## Objective

- Minimize  $100x_1 + 300x_2$  weight

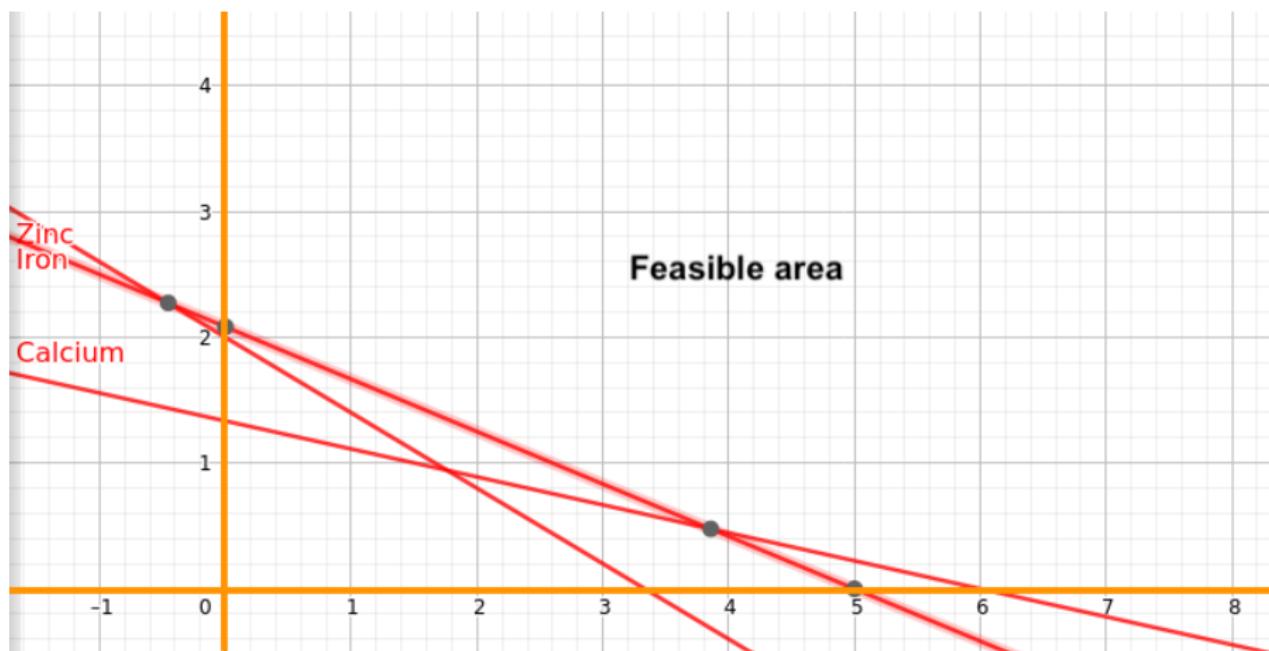
# Diet problem: solution

Constraints:



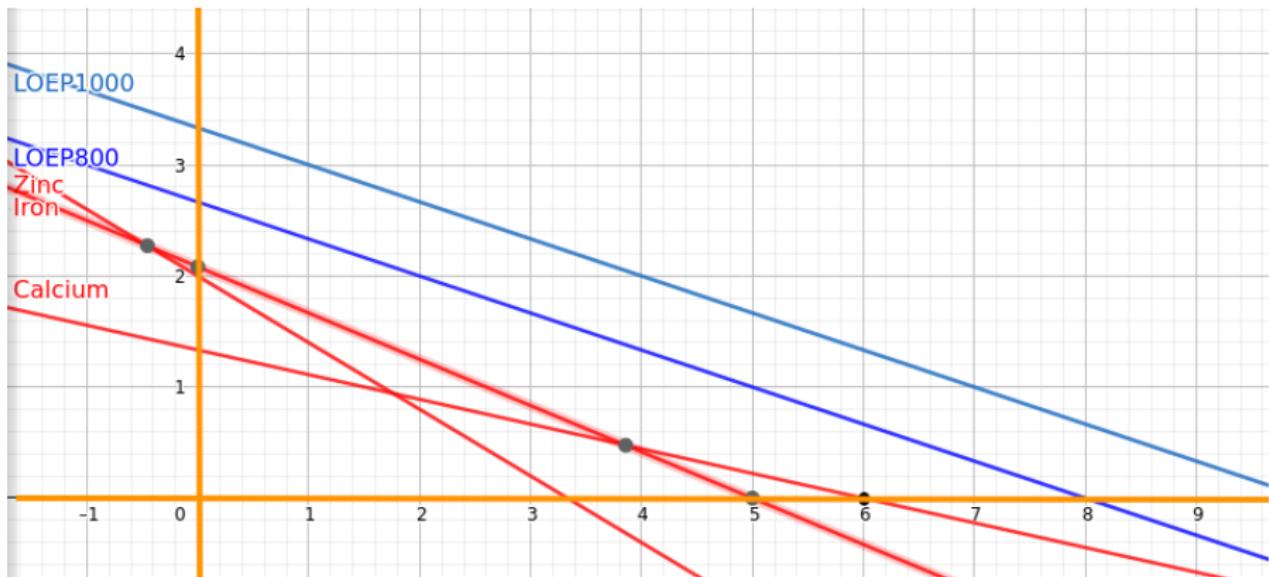
# Diet problem: solution

Feasible area:



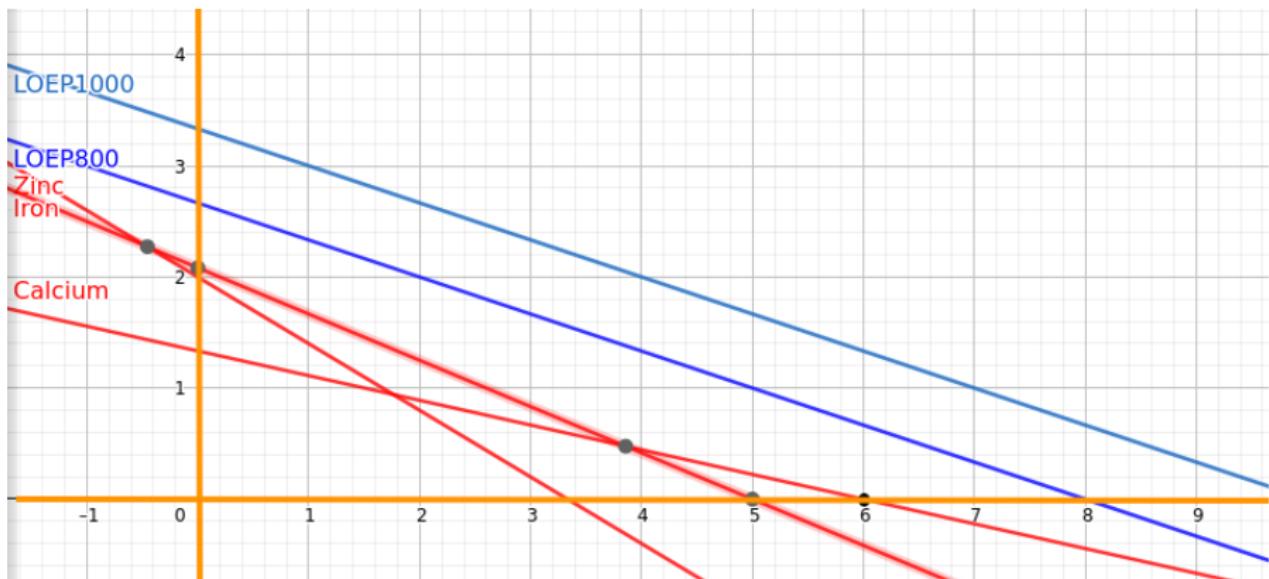
# Diet problem: solution

Since we are now minimising **weight**, the “LOEPs” are now “LOEWs”:



# Diet problem: solution

Since we are now minimising **weight**, the “LOEPs” are now “LOEWs”:



Best solution is the point approx (3.8, 0.5) (confirm by calculation)

# Geogebra graphing calculator

I've been using this to make the diagrams:

<https://www.geogebra.org/graphing/tad337zk>

# Portfolio allocation

A pension fund has a total amount of money to invest, say EUR1M, to be allocated among many possible assets. It wants to maximise the total expected return.

It is obliged by regulations to avoid certain types and combinations of risks, e.g.:

- Diversification: no more than EUR700K in any one asset.
- No more than EUR800K in assets in the same sector, to avoid risk (e.g. over-exposure to tech sector).
- No more than EUR600K in assets in the same geographical region, to avoid risk (e.g. over-exposure to European stocks).

# Portfolio allocation: problem formulation

- Suppose there are one thousand possible assets.
- Decision is represented by 1000 decision variables  $x_i \in \mathbb{R}$ .
- A solution  $x \in \mathbb{R}^{1000}$
- Maximise the total expected return: objective function  
 $f(x) = \sum_i r_i x_i$ , where  $r_i$  is the expected return for investment  $i$ .
- Constraint on total investment:  $\sum_i x_i \leq 1000000$
- Can't invest less than 0 in any asset:  $x_i \geq 0 \forall i$
- Can't invest more than 700K in any asset:  $x_i \leq 700000 \forall i$
- Sectoral diversification:  $\sum_{i \in S_j} x_i \leq 800000$  where  $S_j$  is set of indices e.g.  $S_0 = \{0, 17, 23, 190, 255\}$  for stocks in sector  $j$ .
- Regional diversification:  $\sum_{i \in R_j} x_i \leq 600000$  where  $R_j$  is a set of indices representing stocks in region  $j$ .

# Portfolio allocation: problem formulation

- Suppose there are one thousand possible assets.
- Decision is represented by 1000 decision variables  $x_i \in \mathbb{R}$ .
- A solution  $x \in \mathbb{R}^{1000}$
- Maximise the total expected return: objective function  
 $f(x) = \sum_i r_i x_i$ , where  $r_i$  is the expected return for investment  $i$ .
- Constraint on total investment:  $\sum_i x_i \leq 1000000$
- Can't invest less than 0 in any asset:  $x_i \geq 0 \forall i$
- Can't invest more than 700K in any asset:  $x_i \leq 700000 \forall i$
- Sectoral diversification:  $\sum_{i \in S_j} x_i \leq 800000$  where  $S_j$  is set of indices e.g.  $S_0 = \{0, 17, 23, 190, 255\}$  for stocks in sector  $j$ .
- Regional diversification:  $\sum_{i \in R_j} x_i \leq 600000$  where  $R_j$  is a set of indices representing stocks in region  $j$ .

Too many variables to solve by hand: postpone til we have covered suitable LP software.

# Double-subscript decision variables

Sometimes we have a **2D matrix** of decision variables, i.e.  $x_{ij}$

Two examples to follow:

- Transport problem
- Blend problem

# Canning Company (a transport problem)

A cannning company operates two cannning plants,  $P_1$  and  $P_2$ . Three growers can supply fresh fruit:

- Grower  $G_1$ : up to 200 tonnes at €11/tonne
- Grower  $G_2$ : up to 310 tonnes at €10/tonne
- Grower  $G_3$ : up to 420 tonnes at €9/tonne

Plant capacities and labour costs are:

	Plant $P_1$	Plant $P_2$
Capacity:	460 tonnes	560 tonnes
Labour cost:	€26/tonne	€21/tonne

Shipping cost €3/tonne from any supplier to any plant.

The canned fruits are sold at €50/tonne, with no upper limit on sales.

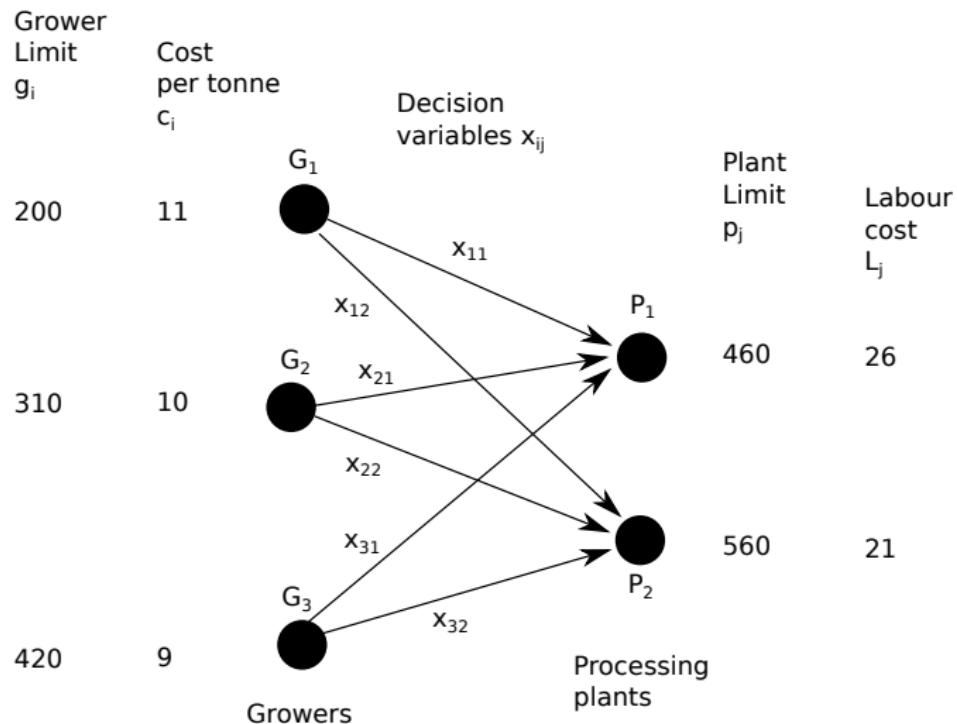
# Canning Company (a transport problem)

The objective is to maximise profits. The decision is how much each grower should supply to each plant.

**Formulate** the problem by identifying the decision variables and formulating the constraints and objective.

# Canning Company (a transport problem)

**Decision variables:** let  $x_{ij}$  be the number of tonnes supplied from grower  $i$  to plant  $j$  where  $x_{ij} \geq 0, i = 1, 2, 3; j = 1, 2$



# Canning Company (a transport problem)

The objective function is to maximise profit.

Let  $x_{ij}$  be the number of tonnes supplied from grower  $i$  to plant  $j$  where  
 $x_{ij} \geq 0, i = 1, 2, 3; j = 1, 2$

Profit for goods shipped from Grower 1 to Plant 1:

- Sale price - Labour cost - fruit cost - shipping cost =  
 $50 - 26 - 11 - 3 = 10$

# Canning Company (a transport problem)

The objective function is to maximise profit.

Let  $x_{ij}$  be the number of tonnes supplied from grower  $i$  to plant  $j$  where  $x_{ij} \geq 0, i = 1, 2, 3; j = 1, 2$

Profit for goods shipped from Grower 1 to Plant 1:

- Sale price - Labour cost - fruit cost - shipping cost =  
 $50 - 26 - 11 - 3 = 10$

So the objective function coefficient for  $x_{11}$  is  $a_{11} = 10$ .

# Canning Company (a transport problem)

The objective function is to maximise profit.

Let  $x_{ij}$  be the number of tonnes supplied from grower  $i$  to plant  $j$  where  $x_{ij} \geq 0, i = 1, 2, 3; j = 1, 2$

Profit for goods shipped from Grower 1 to Plant 1:

- Sale price - Labour cost - fruit cost - shipping cost =  
 $50 - 26 - 11 - 3 = 10$

So the objective function coefficient for  $x_{11}$  is  $a_{11} = 10$ .

Repeat to find the profit coefficient  $a_{ij}$  for every DV.

This gives the objective **maximise**  $\sum_{i,j} a_{ij}x_{ij}$ .

# Canning Company (a transport problem)

Constraint on supply for each grower:

- $x_{11} + x_{12} \leq 200$
- $x_{21} + x_{22} \leq 310$
- $x_{31} + x_{32} \leq 420$
- ie  $\forall i, \sum_j x_{ij} \leq g_i$

Constraint on processing by each canning plant:

- $x_{11} + x_{21} + x_{31} \leq 460$
- $x_{12} + x_{22} + x_{32} \leq 560$
- ie  $\forall j, \sum_i x_{ij} \leq p_j$

# More complex transport problems

Above, we had this information:

- Shipping cost €3/tonne from any supplier to any plant.

In a more complex problem, shipping costs could vary:

- Shipping cost from supply point  $i$  to demand point  $j$  is given by a constant  $s_{ij}$ .

Then we would have a table of shipping costs (e.g. shape  $3 \times 2$  for the Canning Company). This would change the calculation of the DVs' objective function coefficients  $c_{ij}$ .

# Blend problem

Suppose we work for an oil company. We have three components (ingredients), and we produce three products, each a blend of the ingredients. No processing, just blending.

We have contracts to produce at least 3,000 barrels of each grade of motor oil per day.

Determine the optimal mix of the three components in each grade of motor oil to maximize profit.

# Blend problem

Component	Maximum barrels available/day	Cost/barrel
1	4500	12
2	2700	10
3	3500	14

Grade	Component specification	Selling price/barrel
Super	At least 50% of C1	23
	No more than 30% of C2	
Premium	At least 40% of C1	20
	No more than 25% of C3	
Extra	At least 60% of C1	18
	At least 10% of C2	

# Blend problem

## Decision variables

The quantity of each of the three components used in each grade of gasoline (9 decision variables):

$x_{ij}$  = barrels of component  $i$  used in motor oil grade  $j$  per day, where  $i = 1, 2, 3$  and  $j = s$  (super),  $p$  (premium), and  $e$  (extra).

# Blend problem

## Decision variables

The quantity of each of the three components used in each grade of gasoline (9 decision variables):

$x_{ij}$  = barrels of component  $i$  used in motor oil grade  $j$  per day, where  $i = 1, 2, 3$  and  $j = s$  (super),  $p$  (premium), and  $e$  (extra).

## Objective function

Maximise:

$$11x_{1s} + 13x_{2s} + 9x_{3s} + 8x_{1p} + 10x_{2p} + 6x_{3p} + 6x_{1e} + 8x_{2e} + 4x_{3e}$$

# Blend problem constraints

What are the constraints? This is tricky! We will attempt this exercise in the lab.

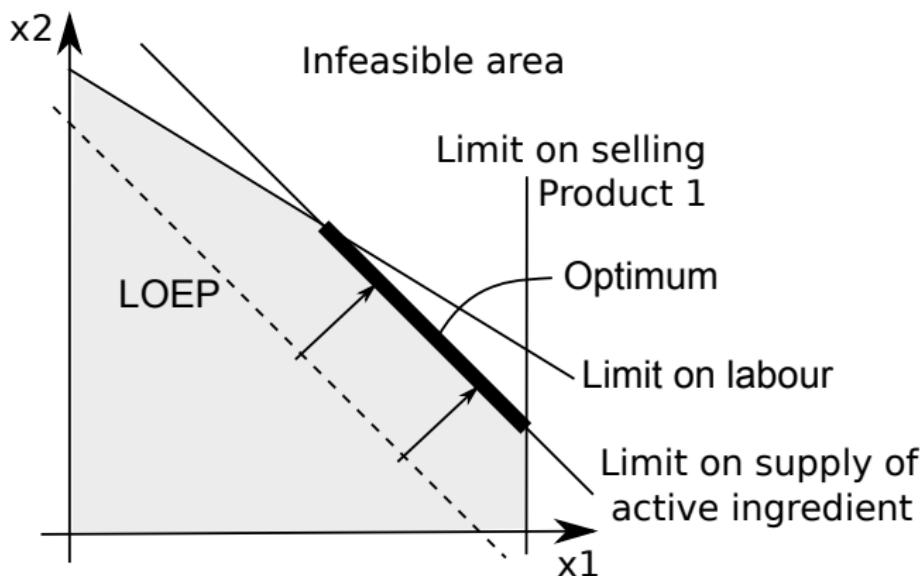
# Overview

- 1 What is linear programming? A motivating problem
- 2 Graphical solution in 2D
- 3 More applications
- 4 **A little theory**

# Possible outcomes of LP

- Normal outcome (we find the optimum)
- Multiple equal optima
- Problem is infeasible
- Problem is unbounded
- Degeneracy (one constraint is redundant)

# Multiple equal optima



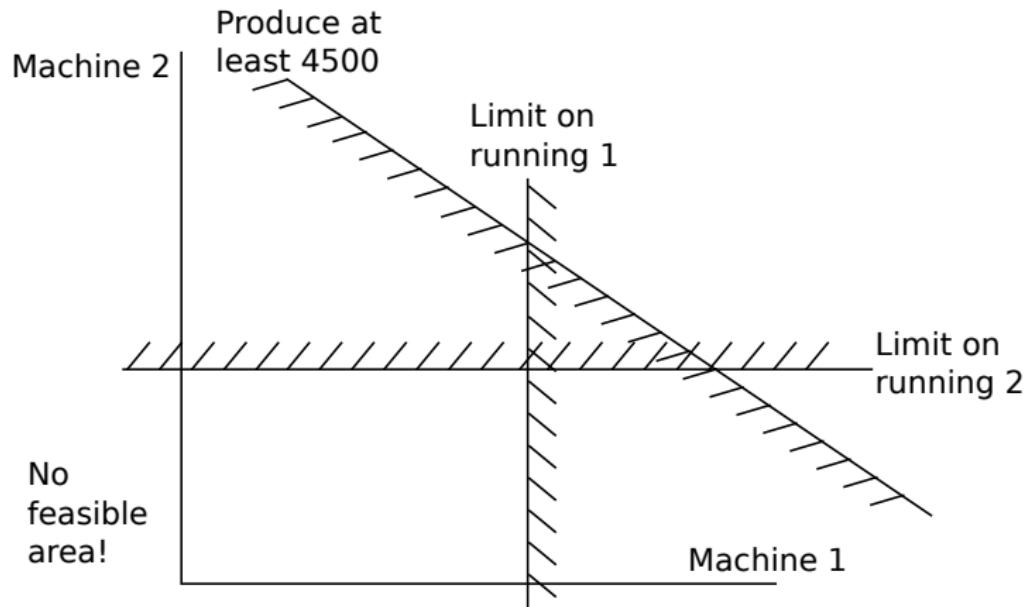
- (A variant on hand sanitizer problem)
- Slope of LOEP equals slope of “top-right” edge
- Then **all points on that edge** are equal optima.

# Infeasibility

- Definition: a problem is **infeasible** if there is no feasible area
- Every point violates some constraint
- Obvious example:  $x_1 \geq 3$ , and  $x_1 < 2$

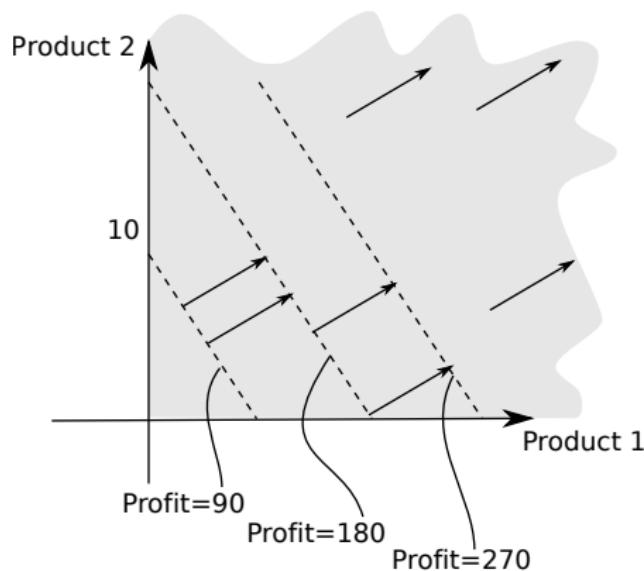
# Infeasibility

More interesting example: a manufacturer has limits on how long they can run their machines, but they also have a contract to provide a certain number of products. Can they do it?



# Unboundedness

- No limit on profits? Unbounded.
- Maybe we forgot a constraint, or tried to maximise a cost function!



# Degeneracy

E.g. we have two constraints:

$$3x_1 + 12x_2 \leq 100$$

$$x_1 + 4x_2 \leq 80$$

One is redundant and can be deleted. (Exercise: which? Draw a picture if needed.)

After deleting, we can re-run.

# Linear and non-linear functions

- Linear programming only works with **linear** objective functions and constraints
- (Otherwise the LOEPs or constraints are not straight lines!)

# LP assumptions

**Linear objective** The objective function is a linear function of the decision variables.

**Linear constraints** Each constraint of the form LHS OP RHS:

- LHS (left-hand side) is a linear function of the decision variables
- OP is an operator  $=$ ,  $\geq$ ,  $>$ ,  $\leq$ , or  $<$ .
- RHS (right-hand side) is a constant.

**Deterministic** All the parameters (objective function coefficients, LHS coefficients, RHS values) are known with certainty.

**Divisibility** Decision variables can take on fractional values.

# Rewriting for linearity

Sometimes simple algebra is needed to see that an objective function or a constraint LHS is indeed linear.

Suppose we are a bank, and we're going to make many loans of different types: normal mortgages, subprime mortgages, personal loans, small business loans, credit card debt, etc. Each loan type  $i$  carries a different risk of default,  $d_i$ . We want to limit our exposure to default, with a rule such as: "the expected amount lost to default must not exceed 15% of the total lending".

Model the amount we lend in category  $i$  as  $x_i$ , the decision variable.

$$\frac{d_1x_1 + d_2x_2 + d_3x_3}{x_1 + x_2 + x_3} \leq 0.15$$

Is this a linear constraint?

# Constraints

- Each constraint corresponds to a straight line (2D) or hyperplane (in general) in the plot, which makes **half** of the space **infeasible** (i.e. disallowed)
- If a constraint involves only one variable, it is **vertical** or **horizontal**. If it involves two or more, it is **diagonal**.

# Example

Why don't we just use LP on problems like fitting linear regression?

- In simple LR, the decision variables are  $a$  and  $b$
- There are **no constraints** on  $a$  and  $b$
- The objective is  $\sum_i(a + bx_i - y_i)^2$

# Example

Why don't we just use LP on problems like fitting linear regression?

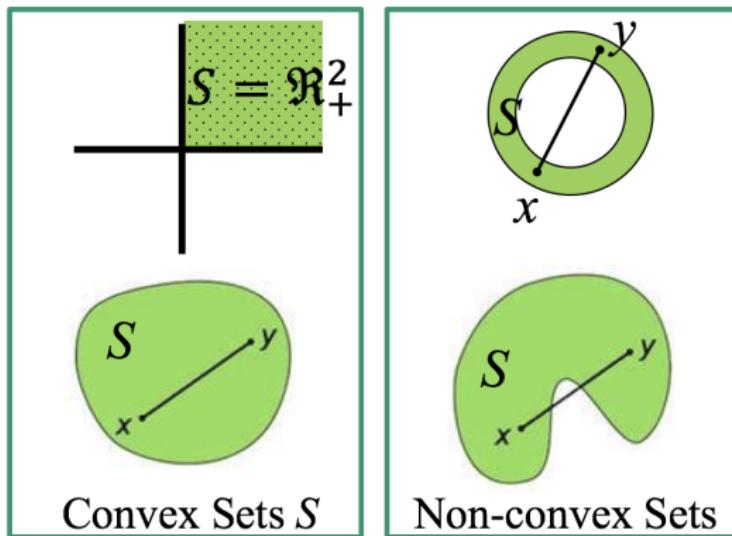
- In simple LR, the decision variables are  $a$  and  $b$
- There are **no constraints** on  $a$  and  $b$
- The objective is  $\sum_i(a + bx_i - y_i)^2$

The objective is **not linear**.

# Convexity

Intuitively, a convex set is **one** component, with **no dents**

A set  $S \subset \mathbb{R}^n$  is **convex** if the line segment between any pair of points  $x, y$  in  $S$  is itself in  $S$ . That is:  $\lambda x + (1 - \lambda)y \in S, \forall x, y \in S, \lambda \in [0, 1]$ .



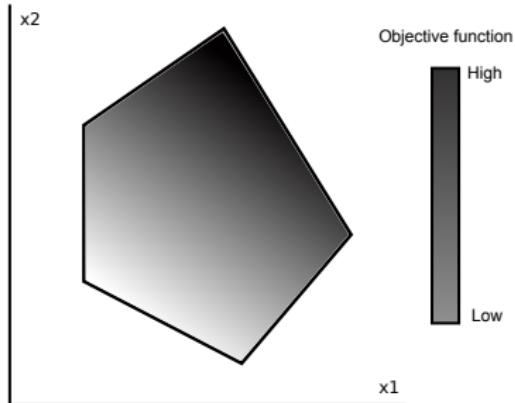
Convex sets (left); non-convex sets (right)

# Convexity

Given **linear constraints**...

- The feasible set is a **polytope**: a convex, connected set with flat, polygonal faces
- In 2D, a polytope is a **convex polygon**.

# Fundamental theorem of LP



- The extremum (min or max) of a linear function on a polytope is at one of the corner points.
- (Unless there are multiple equally-good optima: then they're at corners **and** all along the line segment or face between these corners.)
- (We won't prove this, but a picture is enough to be convincing.)

# Reflection

- What would happen with an LP with no constraints?
- Does LP (with real-valued decision variables) suffer from the curse of dimensionality?

# Next week

- **Integer programming:** the decision variables are integer-valued, not real.

## Homework

- Optional reading: “Basic OR Concepts” from Beasley’s OR-notes:  
<http://people.brunel.ac.uk/~mastjjb/jeb/or/basicor.html>
- Lab: see `lab02.pdf` and `sol02.pdf`.

# Lecture 03 – Integer Programming

## Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Overview

- 1 Integer programming
- 2 Binary integer programming
- 3 Applications and examples

# Integer programming

In **integer programming** (IP) everything is exactly the same as in linear programming, except there is a new type of constraint: **decision variables are integer**.

We drop the **divisibility** assumption of LP.

There are many important applications (and different algorithms).

# Integer programming

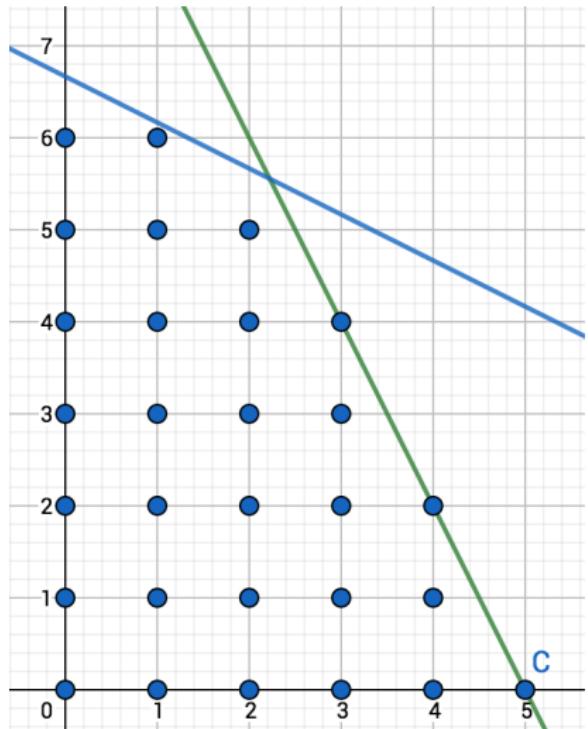
In **integer programming** (IP) everything is exactly the same as in linear programming, except there is a new type of constraint: **decision variables are integer**.

We drop the **divisibility** assumption of LP.

There are many important applications (and different algorithms).

Sometimes it is called **integer linear programming** (ILP) because the objective and constraints still have to be linear.

# IP: feasible area



In integer programming, the feasible area is a grid of points inside a polygon.

# Machine press example

- Machine shop (manufacturer) will buy new equipment of two types: presses and lathes.
- Can only buy an integer number of each.
- Marginal profitability: each press €100/day; each lathe €150/day.
- Resource constraints: budget €40,000,  $200\text{m}^2$  floor space.
- Each press requires  $15\text{m}^2$  and costs €8000
- Each lathe requires  $30\text{m}^2$  and costs €4000

# Machine press example

DVs: let  $x_1$  = number of presses,  $x_2$  = number of lathes, both integer.

Maximise:

$$100x_1 + 150x_2$$

Subject to:

$$8000x_1 + 4000x_2 \leq 40000$$

$$15x_1 + 30x_2 \leq 200m^2$$

$$x_1, x_2 \geq 0$$

$$x_1, x_2 \text{ integer}$$

# Integer constraints

- It is only the decision variables  $x_1$  and  $x_2$  which are constrained to be integer
- Don't confuse this with the **data** of the problem
- E.g. the objective function coefficients 100 and 150 happen to be integers here but could be real in another problem.

Maximise:

$$100x_1 + 150x_2$$

Subject to:

$$8000x_1 + 4000x_2 \leq 40000$$

$$15x_1 + 30x_2 \leq 200m^2$$

$$x_1, x_2 \geq 0$$

$$x_1, x_2 \text{ integer}$$

# Why don't we just...

Why don't we just ignore the integer issue, and solve the problem using LP, and then round our solution off to the nearest integer value of each decision variable?

This is called the **LP relaxation** because we **relax** the integer constraint.

- Sometimes, especially when decision variables will have large values, we **can** do this, and get a good solution

# Why don't we just...

Why don't we just ignore the integer issue, and solve the problem using LP, and then round our solution off to the nearest integer value of each decision variable?

This is called the **LP relaxation** because we **relax** the integer constraint.

- Sometimes, especially when decision variables will have large values, we **can** do this, and get a good solution
- (But we must round off carefully to stay inside constraints)

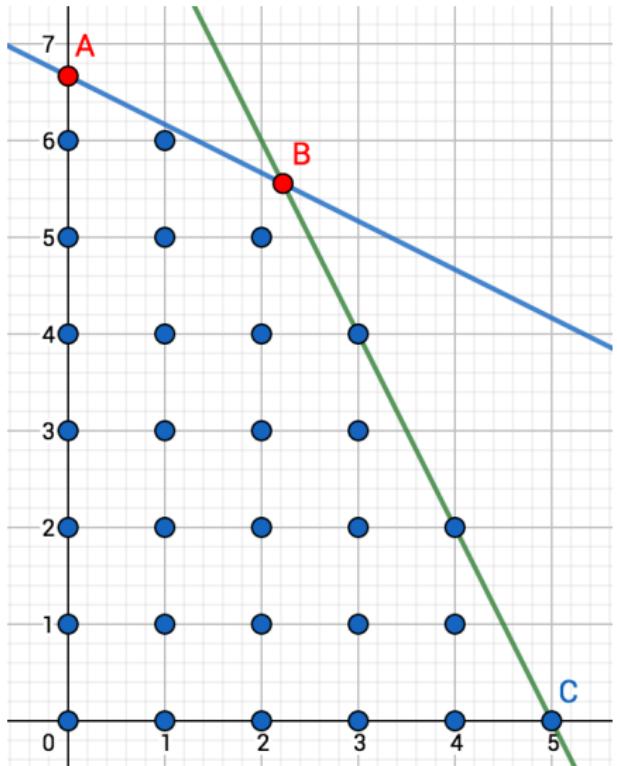
# Why don't we just...

Why don't we just ignore the integer issue, and solve the problem using LP, and then round our solution off to the nearest integer value of each decision variable?

This is called the **LP relaxation** because we **relax** the integer constraint.

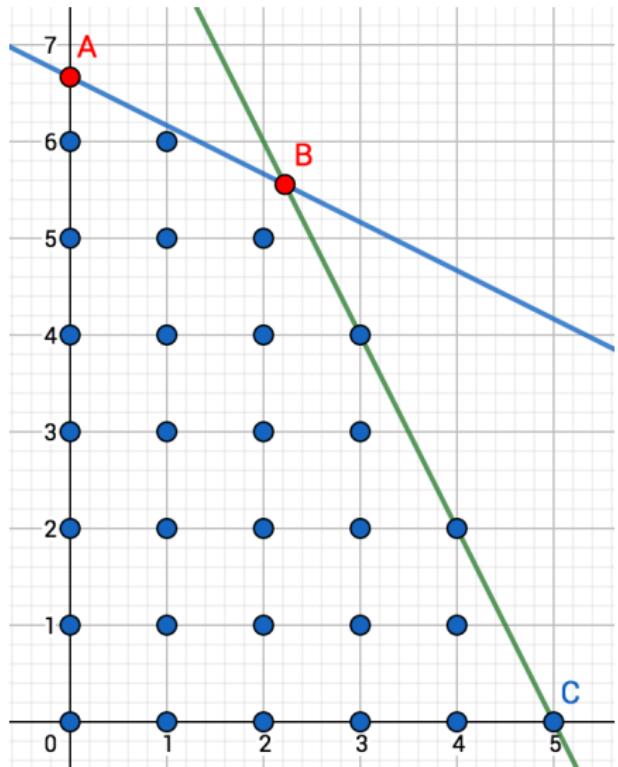
- Sometimes, especially when decision variables will have large values, we **can** do this, and get a good solution
- (But we must round off carefully to stay inside constraints)
- Sometimes, especially when decision variables will have small values, rounding off will give a **sub-optimal solution!**

# Machine press example: LP relaxation



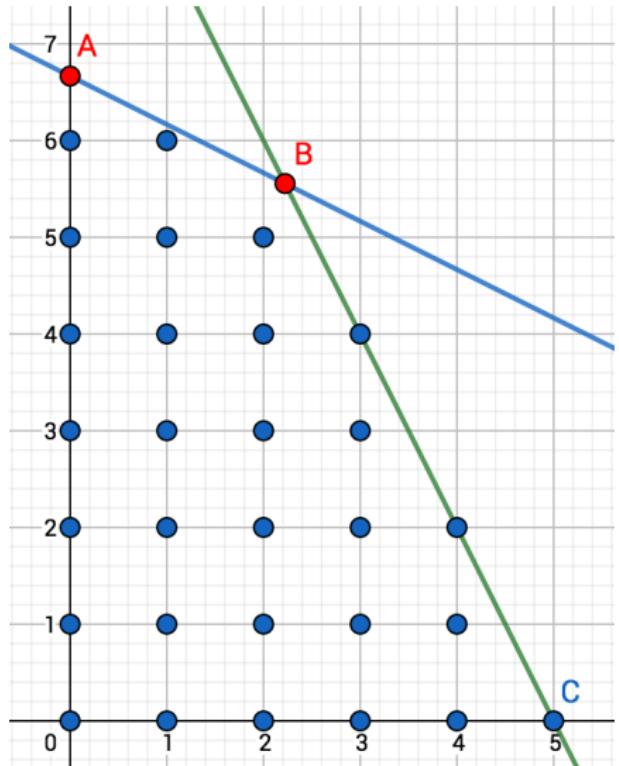
- Let's ignore the integer constraint for now

# Machine press example: LP relaxation



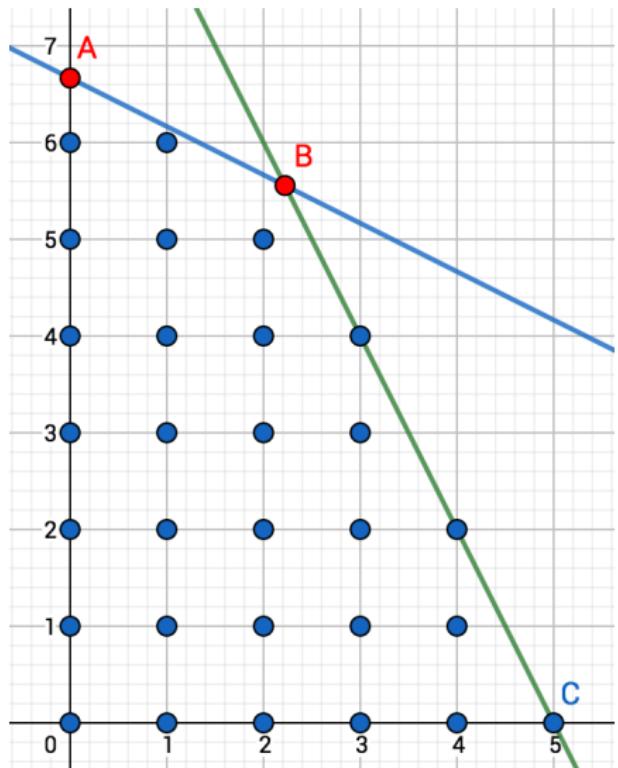
- Let's ignore the integer constraint for now
- This is called solving the LP relaxation

# Machine press example: LP relaxation



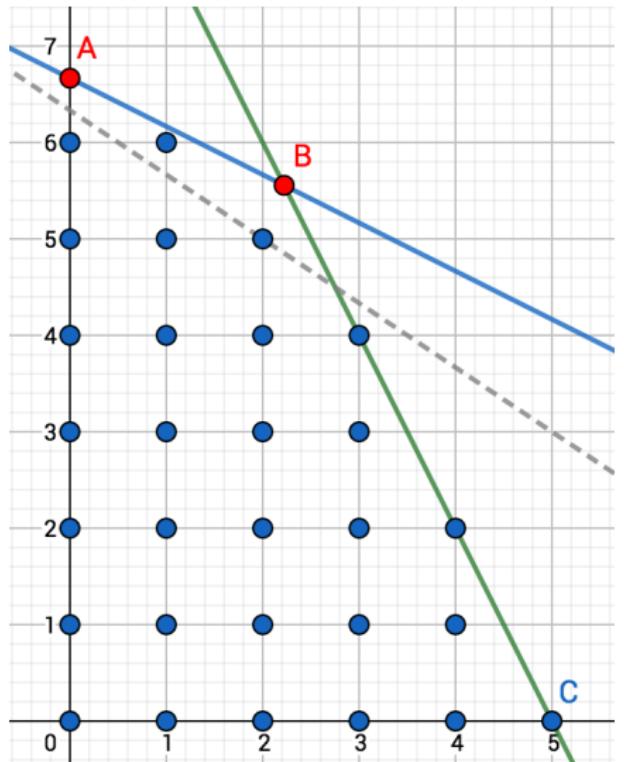
- Let's ignore the integer constraint for now
- This is called solving the LP relaxation
- This gives the optimum  $B = (2.22, 5.56)$ , with  $f(B) = 1,055.56$ .

# Machine press example: LP relaxation



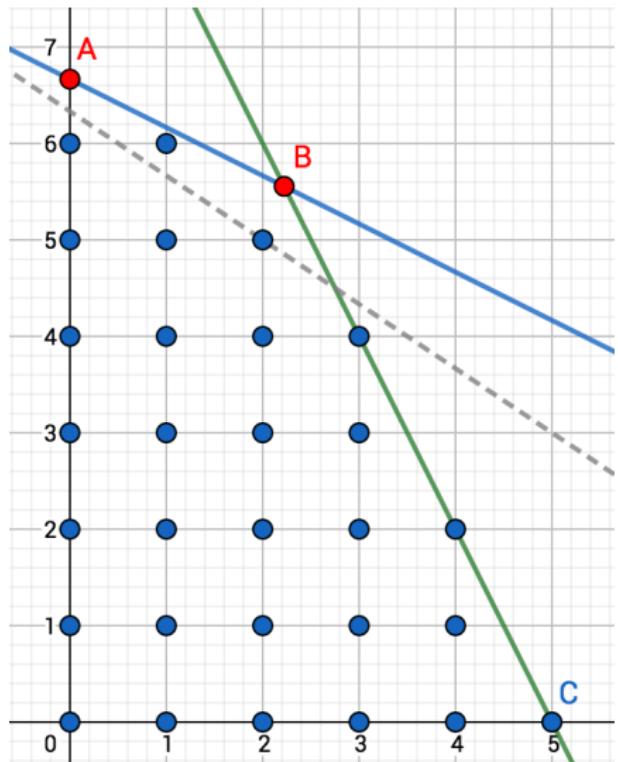
- Let's ignore the integer constraint for now
- This is called solving the LP relaxation
- This gives the optimum  $B = (2.22, 5.56)$ , with  $f(B) = 1,055.56$ .
- Rounding B off then gives  $(2, 5)$  with  $f(2, 5) = 950$  (obeying integer constraint)

# Machine press example: LP relaxation



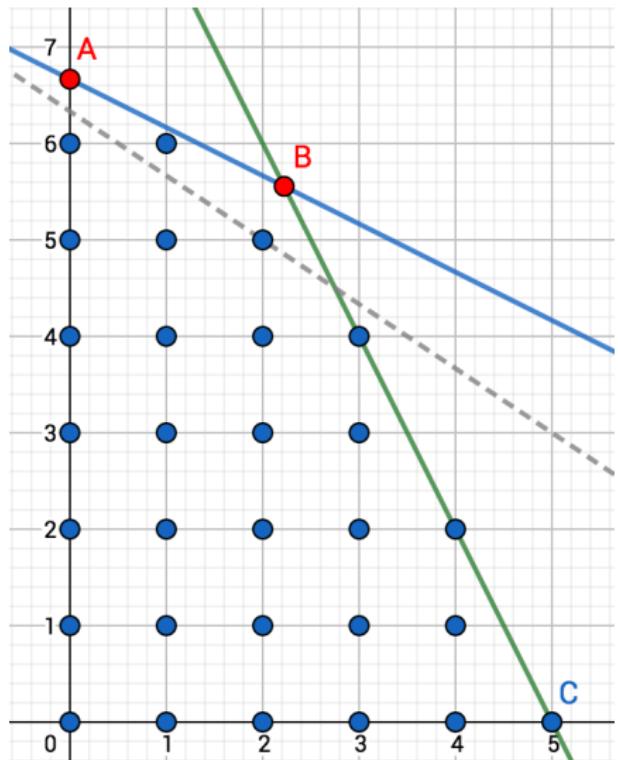
- We add in the LOEP@950 for illustration

# Machine press example: LP relaxation



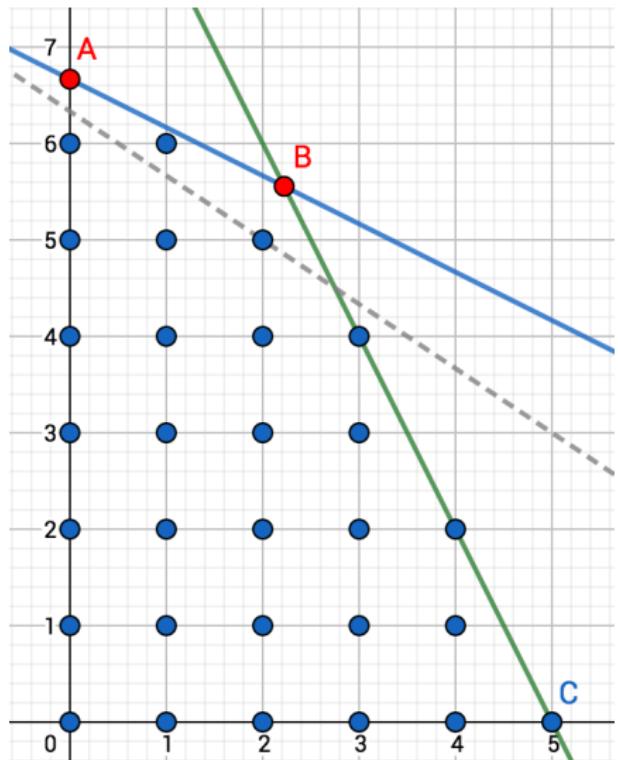
- We add in the LOEP@950 for illustration
- But  $(1, 6)$  is better!  
 $f(1, 6) = 1000$ . We could not achieve this by rounding.

# Machine press example: LP relaxation



- We add in the LOEP@950 for illustration
- But  $(1, 6)$  is better!  
 $f(1, 6) = 1000$ . We could not achieve this by rounding.
- Thus, LP relaxation gives us an **upper bound** on optimum profits, but not necessarily the true solution

# Machine press example: LP relaxation



- We add in the LOEP@950 for illustration
- But (1, 6) is better!  
 $f(1, 6) = 1000$ . We could not achieve this by rounding.
- Thus, LP relaxation gives us an **upper bound** on optimum profits, but not necessarily the true solution
- (For a **minimisation** problem, the LP relaxation gives a **lower bound**.)

# LP Relaxation

Solving LP Relaxation and rounding off doesn't find the solution to an IP problem. But it is used as a **component** of the IP **branch and bound** algorithm we will see next week.

# Overview

- 1 Integer programming
- 2 **Binary integer programming**
- 3 Applications and examples

# Binary decision variables

Binary decision variables are common in IP – sometimes called BIP.  
LP relaxation and rounding fails completely!

# “How many” versus “whether”

In LP and IP, a decision variable  $x_i$  often means **how many** of some quantity, e.g. how many of product  $i$  should we manufacture?

A binary decision variable  $x_i \in \{0, 1\}$  typically means **whether**, e.g. whether to carry out some activity  $i$ .

## “How many” versus “whether”

In LP and IP, a decision variable  $x_i$  often means **how many** of some quantity, e.g. how many of product  $i$  should we manufacture?

A binary decision variable  $x_i \in \{0, 1\}$  typically means **whether**, e.g. whether to carry out some activity  $i$ .

Notice set notation  $\{0, 1\}$ , not the real interval  $[0, 1]$ .

# Recreation centre problem

We run a recreation centre, and we have some money to invest in new facilities. Want to maximise daily **usage**.

- Resource constraints: €120,000 budget; 12 acres of land.
- Selection constraint: stakeholders say we could build **either** swimming pool **or** tennis courts, not both.

Recreation facility	Expected usage (people/day)	Cost (Euros)	Land requirement (acres)
Swimming pool	300	35000	4
Tennis centre	90	10000	2
Athletic centre	400	25000	7
Gym	150	90000	3

# Recreation centre: model

- $x_i$  means **whether** to build facility  $i$
- 1: Swimming pool, 2: Tennis centre, 3: Athletic field, 4: Gym

Maximise  $300x_1 + 90x_2 + 400x_3 + 150x_4$

Subject to:

$$35,000x_1 + 10,000x_2 + 25,000x_3 + 90,000x_4 \leq 120,000$$

$$4x_1 + 2x_2 + 7x_3 + 3x_4 \leq 12 \text{ acres}$$

$$x_1 + x_2 \leq 1 \text{ facility}$$

# Binary variables and logical constraints

In the recreation centre model we had to write a **logical** constraint: we could build the Swimming Pool OR Tennis Centre (or neither) but not both.

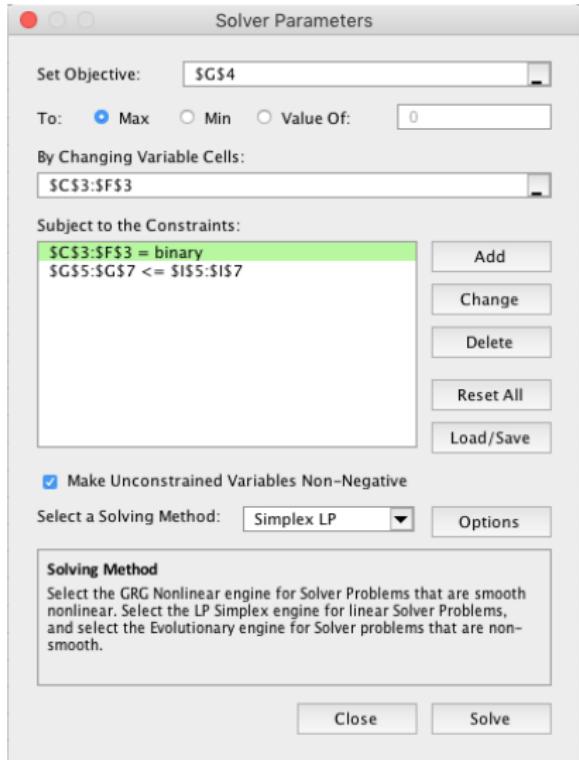
We used **binary variables** to enforce this:  $x_1 + x_2 \leq 1$ .

## Recreation centre solution

We can program the model in Excel...

	A	B	C	D	E	F	G	H	I
1			Swimming pool	Tennis centre	Athletic field				
2		Variables	x1	x2	x3	x4			
3		Values	0	0	0	0			
4	Maximise	Usage	300	90	400	150	0		
5	Subject to	Cost	35000	10000	25000	90000	0 <=	120000	
6		Land area	4	2	7	3	0 <=	12	
7		Selection	1	1	0	0	0 <=	1	

# Recreation centre solution



... and solve using a plug-in called Solver ...

# Recreation centre solution

	A	B	C	D	E	F	G	H	I
1			Swimming pool	Tennis centre	Athletic field	Gym			
2	Variables	x1	x2	x3	x4				
3	Values	1	0	1	0				
4	Maximise	Usage	300	90	400	150	700		
5	Subject to	Cost	35000	10000	25000	90000	60000	$\leq$	120000
6		Land area	4	2	7	3	11	$\leq$	12
7		Selection	1	1	0	0	1	$\leq$	1

It finds  $x_1 = x_3 = 1$  and  $x_2 = x_4 = 0$  for total usage of 700.

# Recreation centre solution

	A	B	C	D	E	F	G	H	I
1			Swimming pool	Tennis centre	Athletic field	Gym			
2	Variables	x1	x2	x3	x4				
3	Values	1	0	1	0				
4	Maximise	Usage	300	90	400	150	700		
5	Subject to	Cost	35000	10000	25000	90000	60000	$\leq$	120000
6		Land area	4	2	7	3	11	$\leq$	12
7		Selection	1	1	0	0	1	$\leq$	1

It finds  $x_1 = x_3 = 1$  and  $x_2 = x_4 = 0$  for total usage of 700.

(We'll cover Excel Solver in labs.)

# A weird language for logical constraints

When modelling problems, especially in BIP, we often need to write logical constraints. But LP/IP requires our constraints to be linear equations/inequalities.

Recall we wrote “Swimming or Tennis or neither but not both” as:  
 $x_1 + x_2 \leq 1$ .

(1: Swimming pool, 2: Tennis centre, 3: Athletic field, 4: Gym)

# A weird language for logical constraints

(1: Swimming pool, 2: Tennis centre, 3: Athletic field, 4: Gym)

How could we express each of these conditions in binary variables?

- Exactly one of Athletic and Gym
- No more than 3 in total
- If Swimming then also Tennis

# A weird language for logical constraints

(1: Swimming pool, 2: Tennis centre, 3: Athletic field, 4: Gym)

How could we express each of these conditions in binary variables?

- Exactly one of Athletic and Gym
  - No more than 3 in total
  - If Swimming then also Tennis
- 
- Exactly one of Athletic and Gym:  $x_3 + x_4 = 1$

# A weird language for logical constraints

(1: Swimming pool, 2: Tennis centre, 3: Athletic field, 4: Gym)

How could we express each of these conditions in binary variables?

- Exactly one of Athletic and Gym
  - No more than 3 in total
  - If Swimming then also Tennis
- 
- Exactly one of Athletic and Gym:  $x_3 + x_4 = 1$
  - No more than 3 in total:  $x_1 + x_2 + x_3 + x_4 \leq 3$

# A weird language for logical constraints

(1: Swimming pool, 2: Tennis centre, 3: Athletic field, 4: Gym)

How could we express each of these conditions in binary variables?

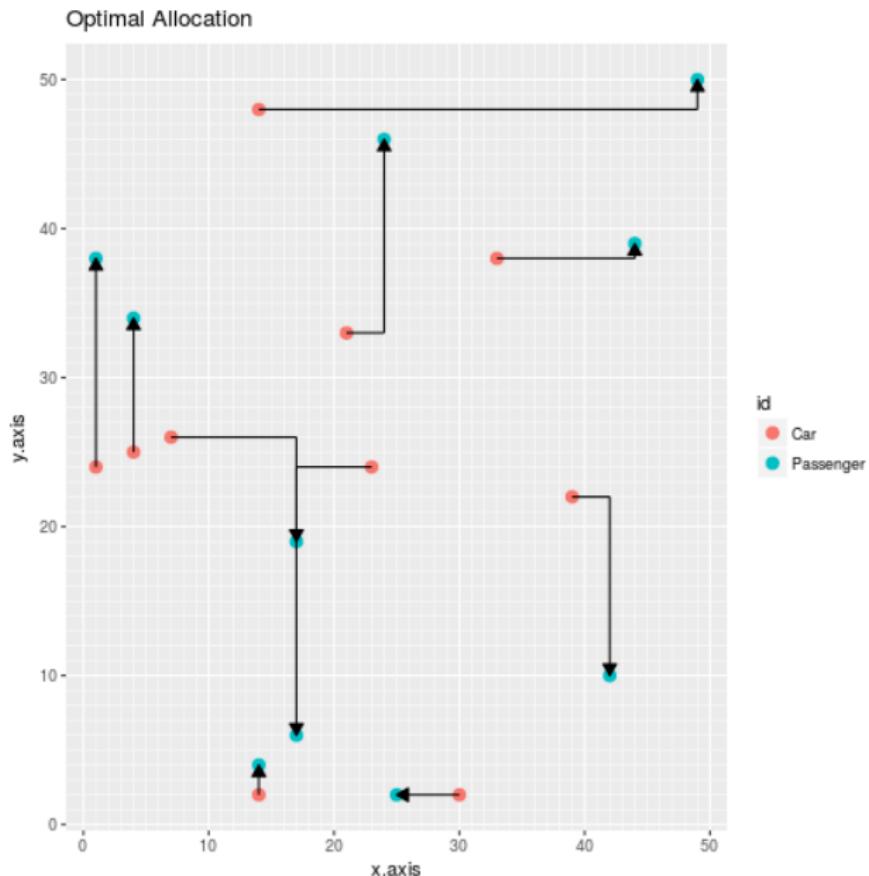
- Exactly one of Athletic and Gym
  - No more than 3 in total
  - If Swimming then also Tennis
- 
- Exactly one of Athletic and Gym:  $x_3 + x_4 = 1$
  - No more than 3 in total:  $x_1 + x_2 + x_3 + x_4 \leq 3$
  - If Swimming then also Tennis:  $x_2 \geq x_1$

# More logical constraints

In fact we can express many logical constraints:

Logical constraint	Implementation
Exactly 1 of A, B, C, D	$a + b + c + d = 1$
At most N of A, B, C, D	$a + b + c + d \leq N$
If A then B	$b \geq a$
If A then not B	$a + b \leq 1$
If not A then B	$a + b \geq 1$
If A then B, and if B then A	$a = b$
If A then B and C	$b \geq a, c \geq a$
If A then B or C	$b + c \geq a$
If B or C then A	$a \geq 0.5 * (b + c)$
If B and C then A	$a \geq b + c - 1$

# Uber: assign cars to passengers



# Uber: assign cars to passengers

We have a set of cars and a set of passengers. For simplicity we'll assume we have **enough** cars. We want to decide which car should pick up which passenger, minimising overall wait time.

# Uber: assign cars to passengers

Data needed: travel time from each car to each passenger. We have several options:

- We could say travel time is proportional to distance, e.g. Euclidean distance
- In this picture, it looks like another common definition of distance might be better: Manhattan distance, also known as taxi-driver's distance:

$$d_M(x, y) = \sum_i |x_i - y_i|$$

- We could use a geographical information system with route-finding and live traffic information, e.g. Google Maps.

This gives us a matrix of car-passenger travel times  $T$ .

# Uber: assign cars to passengers

- Decision variables (binary, double-subscripted) say **whether** car  $i$  picks up passenger  $j$ :

$$x_{ij} \in \{0, 1\}$$

- Cost function:

$$f(x) = \sum_{i,j} x_{ij} T_{ij}$$

- No car  $i$  can pick up **more than 1** passenger:

$$\sum_j x_{ij} \leq 1, \quad \forall i$$

- Every passenger  $j$  must be picked up by **exactly 1** car:

$$\sum_i x_{ij} = 1, \quad \forall j$$

# Assignment problems

The Uber problem is an **assignment problem**, meaning you have two sets of objects and you have to assign each item in one set to one or more items in the other set.

Another example is a scenario where we have several employees, each with differing skill-sets. We have several tasks, each requiring a certain number of person-hours and a certain set of skills. Which employee should work on which task?

# Overview

- 1 Integer programming**
- 2 Binary integer programming**
- 3 Applications and examples**

# Committee problem

NUI Galway are forming a review committee to consist of: at least one male, one female, one student, one academic staff and one administration staff. (One person could fulfill more than one requirement.)

Ten individuals have been nominated called 1, 2, ... 10. All participants will be paid equal expenses. Formulate the ILP to find the minimum cost committee that satisfies the requirements.

Individual	Sub-group
1, 2, 3, 4, 5	Female
6, 7, 8, 9, 10	Male
1, 2, 3, 10	Student
5, 6	Admin
4, 7, 8, 9	Academic

# Committee problem

Binary decision variables  $x_i$  “whether member  $i$  is in the committee”.

Objective: minimise number of members =

$$\sum_i x_i$$

But how to write the constraints?

# Committee problem

Rewrite the sub-groups as binary parameters  $s_{ij}$ : each person  $i$  is either in sub-group  $j$  or not.

NB  $s_{ij}$  are **data**, not **variables**. E.g. person 1 is female, so  $s_{11} = 1$ .

Individual	Sub-group	$s_{ij}$	$j$
1, 2, 3, 4, 5	Female	1111100000	1
6, 7, 8, 9, 10	Male	0000011111	2
1, 2, 3, 10	Student	1110000001	3
5, 6	Admin	0000110000	4
4, 7, 8, 9	Academic	0001001110	5

Now write one constraint for each sub-group  $j$ :

$$\sum_i x_i s_{ij} \geq 1$$

# Multi-period scheduling

A computer manufacturer has this **order schedule**:

Week	0	1	2	3	4	5
Computer Orders	105	170	230	180	150	250

- Production capacity: 160 computers/week regular time, 50 extra with overtime.
- Assembly Costs: €190/computer regular time; €260/computer overtime.
- Inventory Cost: €10/computer per week.

**Goal:** determine a **production schedule** to meet orders at minimum cost with no left over inventory at end of this production period.

# Multi-period scheduling: formulation

Define decision variables:

- $r_j$  = regular production of computers per week  $j$  (0...5)
- $o_j$  = overtime production of computers per week  $j$  (0...5)
- $i_j$  = extra computers carried forward as inventory from week  $j$  (0...5)

This makes it easy to write:

- the objective (minimize cost) and;
- constraints:
  - each week, the inventory carried in + that week's production - that week's order = inventory carried forward
  - (notice inventory carried in to week 0 is 0!)
  - at the end, inventory = 0.

# California Manufacturing Co.

The California Manufacturing Company is considering expansion by building a new factory in Los Angeles or San Francisco, or both. It is also considering building at most one new warehouse, but the choice of warehouse location is restricted to a city where a new factory is built. The total investment budget is \$10M. The net present values (NPV) for each alternative and some other information are shown below. Maximise the total NPV using an ILP. (From Hiller and Lieberman.)

Decision	Question (y/n)	Decision Variable	NPV	Capital Required
1	Build factory in LA?	$x_1$	\$9 Mil.	\$6 Mil.
2	Build factory in SF?	$x_2$	\$5 Mil.	\$3 Mil.
3	Build warehouse in LA?	$x_3$	\$6 Mil.	\$5 Mil.
4	Build warehouse in SF?	$x_4$	\$4 Mil.	\$2 Mil.

# California Manufacturing Co.: model

- Decision: Whether to build or not build a facility

$$x_i = \begin{cases} 1 & \text{facility } i \text{ is built} \\ 0 & \text{Otherwise} \end{cases}$$

- Objective function (Maximise NPV)

$$\text{Maximise } z = 9x_1 + 5x_2 + 6x_3 + 4x_4$$

- Constraints:

- 10 Million to invest:  $6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10$

- At most one warehouse:  $x_3 + x_4 \leq 1$

- Location of the new warehouse must be where a new plant is built:

- Factory in Los Angeles if warehouse:  $x_1 \geq x_3$

- Factory in San Francisco if warehouse:  $x_2 \geq x_4$

# Capital Budgeting Problem (BIP)

A company is planning its capital spending for the next  $T$  periods. There are  $N$  projects that compete for the limited capital  $B_j$ , available for investment in period  $j$ . Each project requires a certain amount of investment in each period once it is selected. Let  $a_{ij}$  be the required investment in project  $i$  for period  $j$ .  $v_i$  is the Net Present value (think: profit) of project  $i$ . The problem is to select the projects for investment that will maximise the net present value of the projects selected.

Let  $x_i = 1$  indicate we DO invest in project  $i$ . Objective: maximise  $\sum_i v_i x_i$  subject to  $\sum_i a_{ij} x_i \leq B_j, \forall j \in 1 \dots T$ . Also subject to  $x_i \in \{0, 1\}$ .

# Capital Budgeting Problem (BIP)

Project	Expenditures (in Millions of €)			NPV
	Year 1	Year 2	Year 3	
1	5	1	8	20
2	4	7	10	40
3	3	9	2	20
4	7	4	1	15
5	8	6	10	30
Funds Available	25	25	25	

Then this problem can be expressed as:

$$\text{Maximise } z = 20x_1 + 40x_2 + 20x_3 + 15x_4 + 30x_5$$

$$\text{Subject to: } 5x_1 + 4x_2 + 3x_3 + 7x_4 + 8x_5 \leq 25$$

$$1x_1 + 7x_2 + 9x_3 + 4x_4 + 6x_5 \leq 25$$

$$8x_1 + 10x_2 + 2x_3 + 1x_4 + 10x_5 \leq 25$$

# Capital Budgeting Problem (BIP)

Suppose we solve the LP relaxation. We get  $z = 125$ ,  $x = (0, 2.27, 0, 2.27, 0)$ . What does this tell us?

# Capital Budgeting Problem (BIP)

Suppose we solve the LP relaxation. We get  $z = 125$ ,  $x = (0, 2.27, 0, 2.27, 0)$ . What does this tell us?

This gives an **upper bound** on profit, but it is very unrealistic!

# Capital Budgeting Problem (BIP)

Suppose we solve the LP relaxation. We get  $z = 125$ ,  $x = (0, 2.27, 0, 2.27, 0)$ . What does this tell us?

This gives an **upper bound** on profit, but it is very unrealistic!

More realistic: instead of dropping the integer constraint  $x_i \in \{0, 1\}$ , **convert** it to  $x_i \in [0, 1]$ . This is **more** realistic. Often this is what people mean when they say LP relaxation.

Now we get a **lower** upper bound  $z = 109$ ,  $x = (0.58, 1, 1, 1, 0.74)$ .

# Capital Budgeting Problem (BIP)

Suppose we solve the LP relaxation. We get  $z = 125$ ,  $x = (0, 2.27, 0, 2.27, 0)$ . What does this tell us?

This gives an **upper bound** on profit, but it is very unrealistic!

More realistic: instead of dropping the integer constraint  $x_i \in \{0, 1\}$ , **convert** it to  $x_i \in [0, 1]$ . This is **more** realistic. Often this is what people mean when they say LP relaxation.

Now we get a **lower** upper bound  $z = 109$ ,  $x = (0.58, 1, 1, 1, 0.74)$ .

But with the true (binary) constraints we get a lower value still,  $z = 95$ ,  $x = (1, 1, 1, 1, 0)$ .

# Constraint Programming

Constraint programming is a special case of LP/IP where there is no objective function – just constraints. Any solution that obeys all constraints is good enough! Usually that's because finding valid solutions is so hard in itself. In practice, e.g. exam timetabling may be a constraint programming problem.

Specialised algorithms are used, which we won't cover, but they are mentioned in *CT5137: Knowledge Representation & Statistical Relational Learning*.

# Reading

Under “Integer Programming” from Beasley’s OR-Notes <http://people.brunel.ac.uk/~mastjjb/jeb/or/ip.html>, read:

- Branch and bound algorithm (to be covered Lecture 04)
- Facility location
- Vehicle routing

# Next week

Algorithms, software, and sensitivity.

# Lecture 04 – Sensitivity Analysis (and Algorithms and Software)

Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Overview

- 1 Algorithms
- 2 Software
- 3 Sensitivity: answering what-if questions

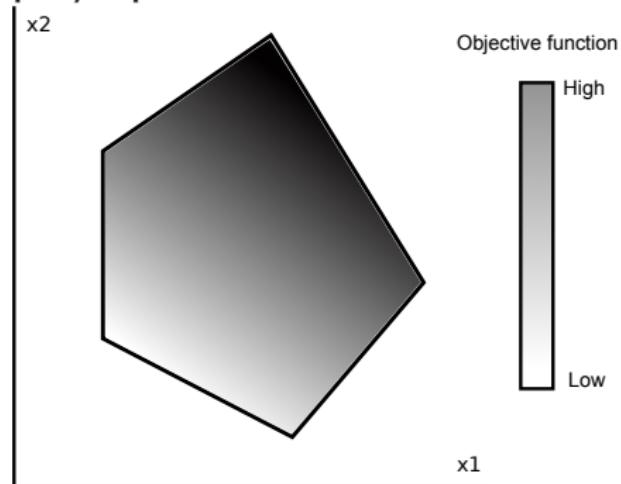
# Solving LP problems

- An LP problem in 2 decision variables (DVs) can be solved with the graphical method
- Of course, these are toy problems!
- For larger ones and practical use, we need algorithms.
- We'll mention the main algorithms and two main software implementations.

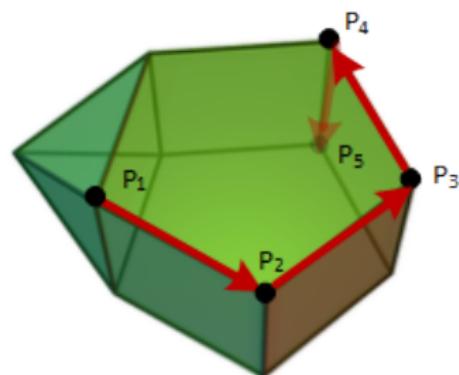
# Simplex Algorithm for LP

The main algorithm for solving **LP** problems is the **Simplex Algorithm**. It is based on the fundamental theorem and Gaussian elimination (solving equations simultaneously).

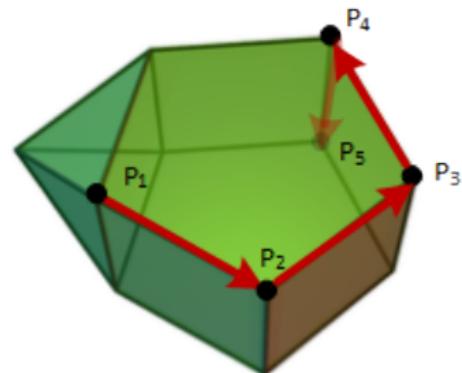
A problem with 2 DVs gives a polytope in 2D:



A problem with 3 DVs gives a polytope in 3D:



# Simplex Algorithm for LP



From [mathstools.com](http://mathstools.com)

We start at a feasible corner point and iteratively move along an edge to a better corner point. Because the feasible area is convex, an improvement is always possible (until we reach the optimum).

The edge is chosen by length and amount of improvement per unit length. This is deterministic and fast.

# Simplex Algorithm for LP

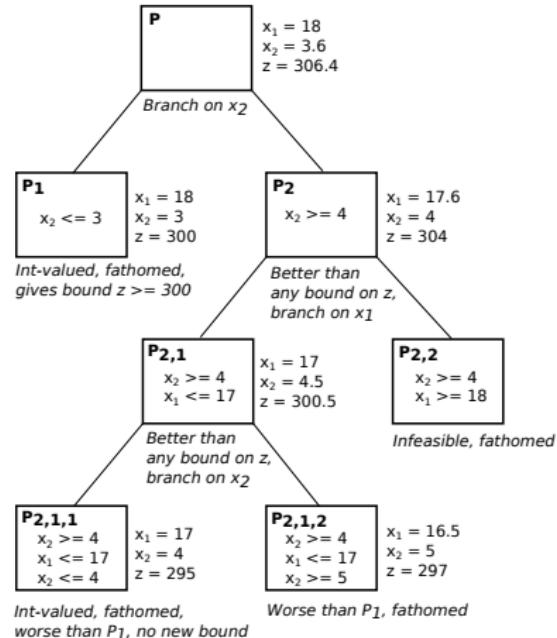
- The space that has to be explored is this feasible area
- Both number of constraints and number of DVs contribute to the number of edges of the feasible area, so both are relevant to problem difficulty.

Full description (not examinable) [here](#).

Example video (not examinable) [here](#).

# Branch and Bound for IP

The main algorithm for solving IP problems is **Branch and Bound**. It relies on the fact that the LP relaxation gives an upper bound on profit (lower bound on cost). It is recursive, leading to a tree structure.



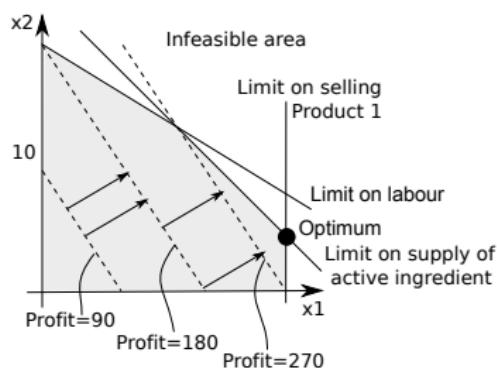
# Branch and Bound for IP

Given a problem  $P$ , this is Branch-and-bound( $P$ ):

- 1 Solve the LP relaxation of problem  $P$ .
- 2 If the LP relaxation is **infeasible** or the objective value is **worse** than some integer solution already seen, then the true solution will not come from further branching here. Mark this branch as **fathomed** (do not expand it further).
- 3 Else if the solution has any non-integer DV e.g.  $x_1 = 4.56$ , then **branch**, i.e. construct new problems:  $P_1$  adds  $x_1 \leq 4$  and  $P_2$  adds  $x_1 \geq 5$ . Call Branch-and-bound( $P_i$ ) for both  $i$ .
- 4 Else (the solution has no non-integer DV) this is an integer solution. It gives a **bound**: the true solution is not worse. Store the objective value and mark this branch as **fathomed**.
- 5 Return.

Eventually, this recursive process will **fathom** all branches and we return the best integer-valued solution we found.

# Recall: Sanitizer problem



Maximise profit:

$$15x_1 + 10x_2$$

Subject to:

$$0.1x_1 + 0.1x_2 \leq 2.2 \text{ (raw materials)}$$

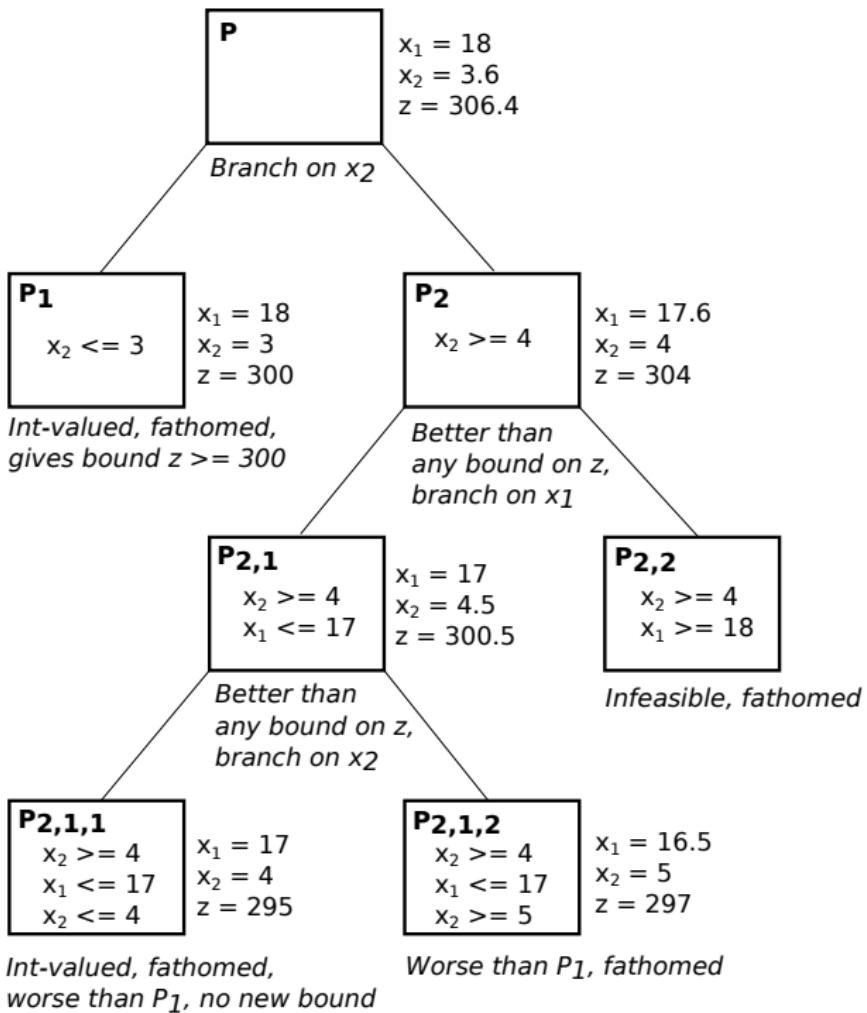
$$1.5x_1 + 2.5x_2 \leq 45 \text{ (labour)}$$

$$x_1 \leq 18 \text{ (demand for Product 1)}$$

$$x_2 \leq 30 \text{ (demand for Product 2)}$$

$$x_1, x_2 \geq 0 \text{ (non-negativity)}$$

Solution: (18, 4) giving profit  $z = 310$ .



# Why is IP slower than LP?

As we have seen, our best algorithm for IP solves **many** LP problems while solving one IP problem!

# Hungarian Algorithm

The Assignment problem has binary variables and a special structure which allows for a specialised algorithm, the **Hungarian algorithm**. This is more efficient than the branch-and-bound algorithm.

Explanation here (not examinable).

# Algorithms

We won't be programming these algorithms by hand because (in contrast to heuristic/metaheuristic optimisation) it is very rare to need to program a custom variant.

Instead we'll move straight to considering software.

# Overview

- 1 Algorithms
- 2 Software
- 3 Sensitivity: answering what-if questions

# Software

- **Excel Solver**
- **OR-Tools**
- `scipy.optimize/linprog`
- PuLP
- COIN-OR
- Xpress-MP
- ...

We've seen Excel Solver in labs. Now we'll see a little of OR-Tools and its Python interface. We'll see how to model a problem and get the solution. Later we'll see how to do some sensitivity analysis.

# OR-Tools

<https://developers.google.com/optimization/>

```
# install in bash/PowerShell  
$ pip install ortools
```

```
# check install ok  
from ortools.linear_solver import pywraplp
```

# OR-Tools modelling

(We'll use the Hand Sanitizer problem as an example)

- Instantiate a solver:

```
solver = pywraplp.Solver('Sanitizer',  
                         pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)
```

# OR-Tools modelling

- Create a continuous variable with bounds:

```
x1 = solver.NumVar(0, 18, 'x1')
```

# OR-Tools modelling

- Create a continuous variable with bounds:

```
x1 = solver.NumVar(0, 18, 'x1')
```

Notice that here we are creating variables with special library-specific types and with strings as names. The variable `x1` doesn't have a numerical value like 4.56. It is really a **placeholder**. A similar concept is used in neural network libraries like Tensorflow and symbolic maths libraries like Sympy (see CT5132/CT5148).

# OR-Tools modelling

You can give any name you like, but don't confuse yourself!

```
x1 = solver.NumVar(0, 18, 'my favourite variable')
x2 = solver.NumVar(0, 30, 'x1')
```

# OR-Tools modelling

- Create a constraint:

```
solver.Add(1.5*x1 + 2.5*x2 <= 45,  
           name="limit on labour")
```

# OR-Tools modelling

- Create a constraint:

```
solver.Add(1.5*x1 + 2.5*x2 <= 45,  
           name="limit on labour")
```

Again, notice that this expression **looks** like it will have a Boolean value, but in fact variables  $x_1$  and  $x_2$  are of a type which **over-rides**  $*$ ,  $+$  and  $\leq$ . It just becomes a constraint object, stored inside the `solver` object.

# OR-Tools modelling

- Create a constraint:

```
solver.Add(1.5*x1 + 2.5*x2 <= 45,  
           name="limit on labour")
```

Again, notice that this expression **looks** like it will have a Boolean value, but in fact variables  $x_1$  and  $x_2$  are of a type which **over-rides**  $*$ ,  $+$  and  $\leq$ . It just becomes a constraint object, stored inside the `solver` object.

By the way, the Python builtin `sum()` runs  $+$  behind the scenes, so you can use things like `sum(c*x for c, x in zip(coefs, vars))` on the LHS of a constraint.

# OR-Tools modelling

- Create objective:

```
objective = solver.Objective()  
objective.SetCoefficient(x1, 15)  
objective.SetCoefficient(x2, 10)  
objective.SetMaximization()
```

# OR-Tools modelling

- Solve:

```
result = solver.Solve()
```

- Print out solution:

```
for v in solver.variables():
    print(f"{v.name()} = {v.solution_value()}")
print(f"Obj val = {solver.Objective().Value()}")
```

# OR-Tools outcomes

`result = solver.Solve()` will give an integer. We should check it.

```
pywraplp.Solver.OPTIMAL = 0
pywraplp.Solver.FEASIBLE = 1
pywraplp.Solver.INFEASIBLE = 2
pywraplp.Solver.UNBOUNDED = 3
pywraplp.Solver.ABNORMAL = 4
pywraplp.Solver.NOT_SOLVED = 6
```

# OR-Tools outcomes

`result = solver.Solve()` will give an integer. We should check it.

```
pywraplp.Solver.OPTIMAL = 0  
pywraplp.Solver.FEASIBLE = 1  
pywraplp.Solver.INFEASIBLE = 2  
pywraplp.Solver.UNBOUNDED = 3  
pywraplp.Solver.ABNORMAL = 4  
pywraplp.Solver.NOT_SOLVED = 6
```

I have not managed to see UNBOUNDED (it wrongly gives INFEASIBLE):  
<https://github.com/google/or-tools/issues/2198>

It has no way to detect **multiple equal optima** (it just gives OPTIMAL). (Excel is the same in this.)

# OR-Tools Practicalities

Compared to Excel Solver, OR-Tools:

- Is harder to get started with
- Is more convenient for large problems:
  - Easier to type than to point-n-click
  - No need to type large matrix of constraint coefficients
- Has far more powerful solvers
- Amenable to version control.

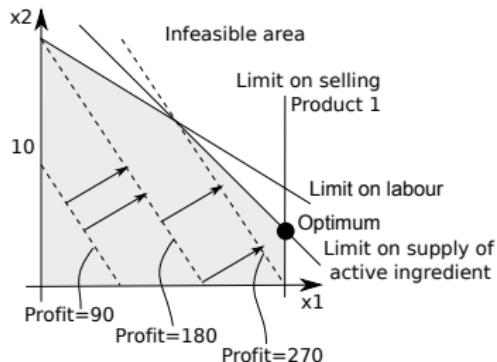
# Overview

- 1 Algorithms
- 2 Software
- 3 **Sensitivity: answering what-if questions**

# Asking what-if questions

- We should be able to **interpret** LP solutions
- Often we may be interested in follow-up “what if?” questions of high business importance
- The special assumptions of LP allow us to answer these questions
- This is called **sensitivity analysis** and sometimes **post-solution** analysis.

# Recall: Sanitizer problem



Maximise profit:

$$15x_1 + 10x_2$$

Subject to:

$$0.1x_1 + 0.1x_2 \leq 2.2 \text{ (raw materials)}$$

$$1.5x_1 + 2.5x_2 \leq 45 \text{ (labour)}$$

$$x_1 \leq 18 \text{ (demand for Product 1)}$$

$$x_2 \leq 30 \text{ (demand for Product 2)}$$

$$x_1, x_2 \geq 0 \text{ (non-negativity)}$$

# Interpreting solutions

The **solution** to this LP is:

$$(x_1, x_2) = (18, 4)$$

with objective value  $f(x_1, x_2) = 310$ .

# Interpreting solutions

The **solution** to this LP is:

$$(x_1, x_2) = (18, 4)$$

with objective value  $f(x_1, x_2) = 310$ .

We should remember to interpret our solution fully in the real-world context. The solution means we should produce 18 units of Product 1 and 4 of Product 2, to achieve profits of EUR310.

# Interpreting solutions

The **solution** to this LP is:

$$(x_1, x_2) = (18, 4)$$

with objective value  $f(x_1, x_2) = 310$ .

We should remember to interpret our solution fully in the real-world context. The solution means we should produce 18 units of Product 1 and 4 of Product 2, to achieve profits of EUR310.

Also, with this solution we use all 2.2L of active ingredient, and 37 hours of labour (leaving 8 hours of labour unused).

# Sensitivity analysis

- How much of each resource have we used?
- How much of each resource is “left over”?
- Which constraints turned out to be important?
- How much would one more unit of a resource be worth to us?  
What if we had one unit less?
- How much more of that resource would be useful, before some other constraint prevented further increases in profit?
- How large a change could happen in the profit per unit of a product, before the location of the optimum would change?

# Examples

- How many hours of labour will our plan use?
- Our value for raw materials availability is an estimate: should I go and find out the real value?
- I think we could increase the **selling price** on Product 1 with an advertising campaign – would it be worthwhile?
- I think we could increase the **demand** for Product 2 with an advertising campaign – would it be worthwhile?

# Sensitivity

Is our outcome **sensitive** to the exact values of the data?

- That is: if some parameter changed slightly, would the outcome change **slightly**, or could it even change **a lot**?
- If a slight change in a parameter wouldn't change the outcome at all, we say the outcome is **insensitive** to changes in that parameter, otherwise we say it's **sensitive** to that parameter.

# Sensitivity

Is our outcome **sensitive** to the exact values of the data?

- That is: if some parameter changed slightly, would the outcome change **slightly**, or could it even change **a lot**?
- If a slight change in a parameter wouldn't change the outcome at all, we say the outcome is **insensitive** to changes in that parameter, otherwise we say it's **sensitive** to that parameter.

(Sometimes **sensitivity analysis** is used to refer to all of these “what-if” questions.)

# Sensitivity analysis

- **Activity:** How much of each resource have we used?
- **Slack:** How much of each resource is “left over”?
- **Binding:** Which constraints turned out to be important?
- **Shadow price:** How much would one more unit of a resource be worth to us? What if we had one unit less?
- **Allowable increase of a constraint:** How large an increase in the constraint RHS resource would be useful, before some other constraint prevented further increases in profit?
- **OFC sensitivity of a DV:** How large an increase in the objective function coefficient (OFC) of the DV could happen, before the location of the optimum would change?

# Why don't we just...

Why don't we just re-run the problem, after adjusting the coefficients or RHS values, to see what happens?

# Why don't we just...

Why don't we just re-run the problem, after adjusting the coefficients or RHS values, to see what happens?

- Actually, sometimes in IP we do that!
- But there are many possible adjustments, and larger problems take a long time
- The theory of LP (but not IP) gives us all the information we want “for free”, as a **by-product** of running the simplex algorithm once.

# Interactive/animated solution

Geogebra: <https://www.geogebra.org/m/abjxez2u>

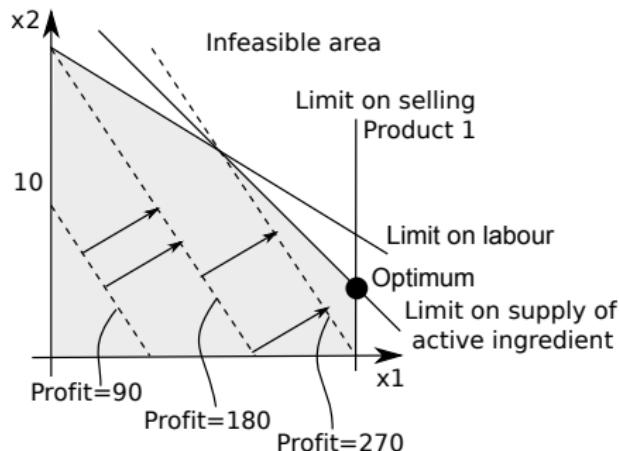
# Activity and Slack

- The **activity** of a constraint is the value of its LHS at the optimum.
- E.g. the labour constraint is:  $1.5x_1 + 2.5x_2 \leq 45$
- The optimum is (18, 4)
- The activity of the labour constraint is  
 $1.5x_1 + 2.5x_2 = 1.5 \cdot 18 + 2.5 \cdot 4 = 37$ , i.e. we are using 37 hours of labour.

# Activity and Slack

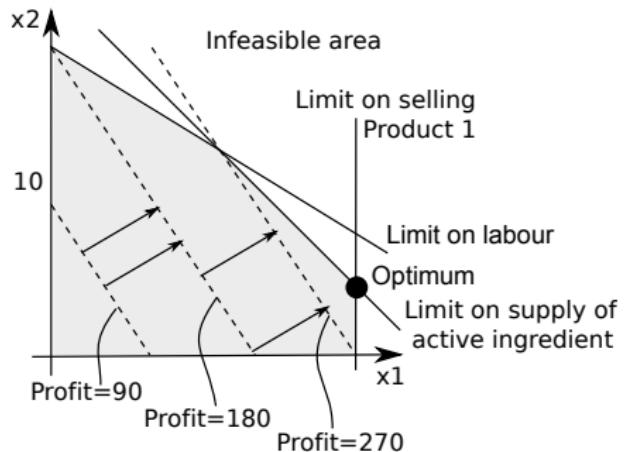
- The **activity** of a constraint is the value of its LHS at the optimum.
- E.g. the labour constraint is:  $1.5x_1 + 2.5x_2 \leq 45$
- The optimum is (18, 4)
- The activity of the labour constraint is  
 $1.5x_1 + 2.5x_2 = 1.5 \cdot 18 + 2.5 \cdot 4 = 37$ , i.e. we are using 37 hours of labour.
- The **slack** of a constraint is the amount “left over” on the RHS
- The slack of the labour constraint is 8 hours
- For any constraint: activity + slack = RHS value.

# Binding



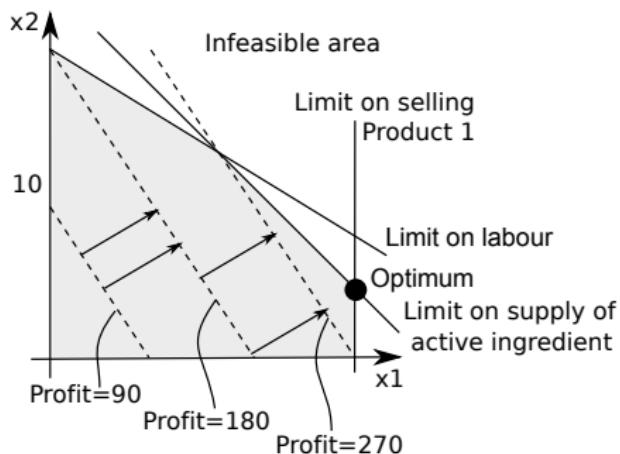
- A constraint is **binding** if its **slack is zero**: there is none of the RHS “left over”
- The constraint line **touches** the optimum
- Removing the constraint entirely would allow the optimum to be improved
- E.g., the labour constraint **is not binding**
- E.g., the active ingredient constraint **is binding**.

# Shadow price



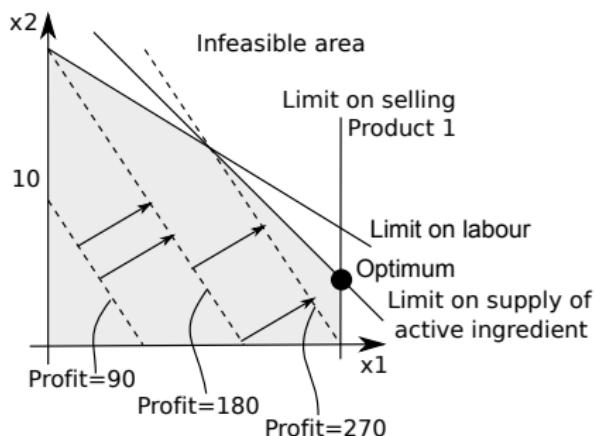
- The **shadow price** of a constraint is the improvement in the optimum we would achieve for unit increase in the RHS
- Shadow price =  $\frac{\partial z}{\partial b_i}$  where  $z$  is profit and  $b_i$  is RHS.

# Shadow price



- A **non-binding** constraint has **zero** shadow price
- E.g. the shadow price for labour is EUR0: it is non-binding, so extra available labour won't help
- E.g. the shadow price for Product 1 demand is EUR5: adding 1 extra unit of demand will allow profit EUR310 → EUR315.

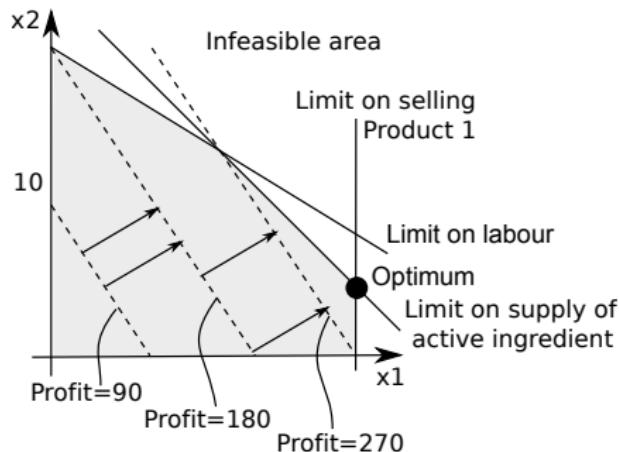
# Allowable increase in a constraint



- E.g. the shadow price for Product 1 demand is EUR5: adding 1 extra unit of demand will allow profit EUR310 → EUR315.
- The location of the optimum would change to (19, 3).

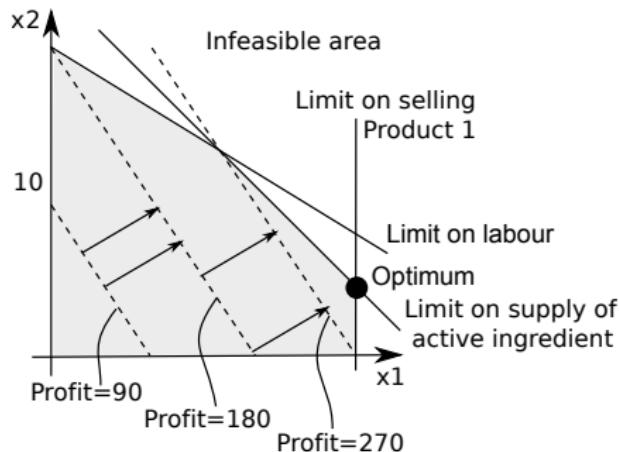
- Further increases in Product 1 demand up to 22 would be useful (giving (22, 0) and profit EUR330). But then Product 2 non-negativity becomes binding!
- The **allowable increase** in the Product 1 demand constraint was therefore 4 (from 18 up to 22).
- The shadow price of a constraint is **valid only up to the allowable increase**.

# OFC sensitivity



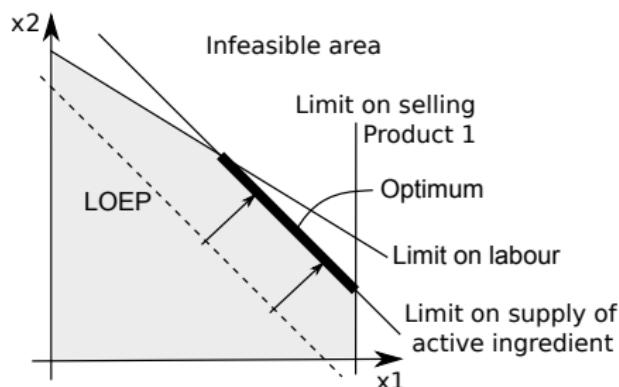
- The **sensitivity** of an objective function coefficient (OFC) is the range in which that OFC can change without changing the location of the optimum.
- Note the **profit** at the optimum will certainly change: we are concerned here with the **location**.

# OFC sensitivity



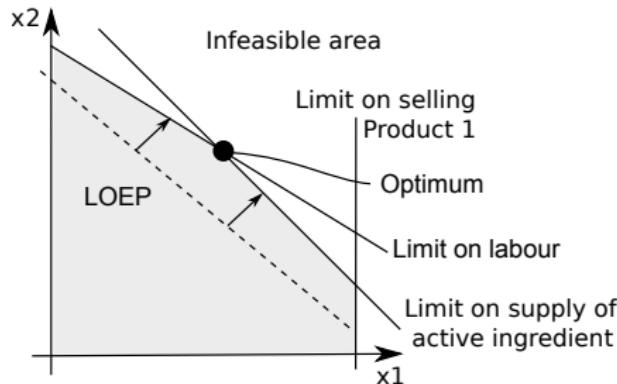
- The slope of the active ingredient constraint is -1:  
 $0.1x_1 + 0.1x_2 \leq 2.2$
- Suppose Product 1 profit changes, so that the LOEP  $15x_1 + 10x_2$  has slope -1 also
- That could happen if the 15 changes to 10:  $10x_1 + 10x_2$

# OFC sensitivity



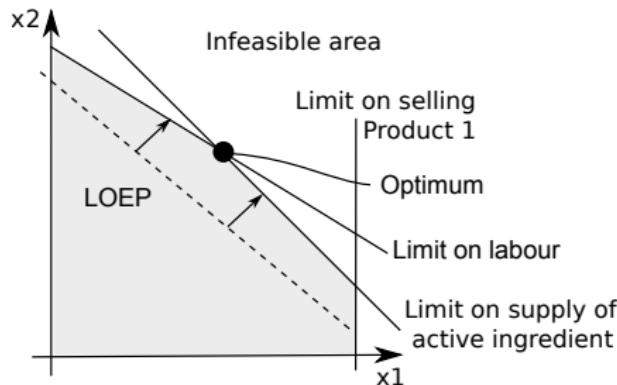
- That could happen if the 15 changes to 10:  $10x_1 + 10x_2$
- Now the LOEP is **parallel** with the active ingredient constraint
- The whole line segment becomes **multiple equal optima**

# OFC sensitivity



- If Product 1 profit decreased to  $< 10$ , the optimum would move to next corner as shown.

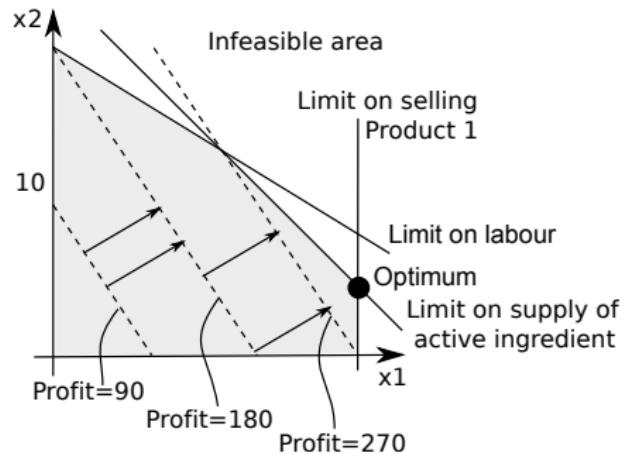
# OFC sensitivity



- If Product 1 profit decreased to  $< 10$ , the optimum would move to next corner as shown.

Thus the location of the optimum is **insensitive to decreases** of the  $x_1$  OFC from 15 down to 10.

# OFC sensitivity



Other direction: any  $x_1$  OFC **increases** will not change location.

# Sensitivity is for one change only

All of this analysis is valid when we make a single hypothetical change at a time

With 2 or more changes, some analysis is still possible, but it's more complicated!

# Sensitivity in minimisation problems

All of the above works in minimisation problems:

- Instead of a resource “left over” in a  $\leq$  constraint, our solution provides “more than was required” of some RHS limit in a  $\geq$  constraint, e.g. more than the daily minimum of vitamin C in a diet problem.
- Shadow prices must be interpreted carefully: a negative shadow price, in minimisation, means an improvement in the objective.

# Sensitivity in Excel Solver

Excel Solver does a lot of sensitivity analysis (Google Docs and Libre-Office Solvers do not provide sensitivity analysis, unfortunately).

(See also Beasley's OR-notes: [http://people.brunel.ac.uk/~mastjjb/jeb/or/lpsens\\_solver.html](http://people.brunel.ac.uk/~mastjjb/jeb/or/lpsens_solver.html))

# Sensitivity in Excel Solver

	A	B	C	D	E	F	G
1							
2							
3							
4	<b>Sanitizer</b>	x1	x2				
5	<b>values</b>	18	4				
6	<b>maximise</b>	15	10	310			
7	<b>subject to</b>	0.1	0.1	2.2	<=		2.2
8		1.5	2.5	37	<=		45
9		1	0	18	<=		18
10		0	1	4	<=		30
11							

# Sensitivity in Excel Solver

## Variable Cells

Cell	Name	Final Value	Reduced Cost	Objective Coefficient	Allowable Increase	Allowable Decrease
\$C\$5	values x1	18	0	15	1E+30	5
\$D\$5	values x2	4	0	10	5	10

## Constraints

Cell	Name	Final Value	Shadow Price	Constraint R.H. Side	Allowable Increase	Allowable Decrease
\$E\$7	subject to	2.2	100	2.2	0.32	0.4
\$E\$8		37	0	45	1E+30	8
\$E\$9		18	5	18	4	8
\$E\$10		4	0	30	1E+30	26

# Sensitivity in OR-Tools

We can access sensitivity information as follows:

- Activity: `solver.ComputeConstraintActivities()`
- Slack: `c.ub() - activity or activity - c.lb()`
- Binding: `abs(c.DualValue()) < epsilon`
- Dual Value: `c.DualValue()` (for constraint `c`)
- OFC Sensitivity and allowable increases/decreases: **not so easy**, we will not cover this.

# Summary

- Sensitivity is an important part of using LP in practice.
- Main concepts: activity, slack, binding, shadow price, allowable increase, OFC sensitivity.
- The language of sensitivity, shadow prices, etc., may not be accessible to our client or management. We should be able to provide simple explanations.

# Overview

- 1 Algorithms
- 2 Software
- 3 Sensitivity: answering what-if questions

# Lecture 05 – Black-box optimisation, hill-climbing, and search landscapes

Optimisation CT5141

James McDermott

NUI Galway

2020

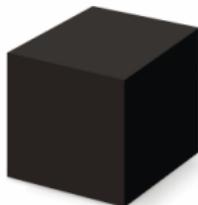


OÉ Gaillimh  
NUI Galway

# Overview

- 1 What is black-box optimisation?**
- 2 Hill climbing**
- 3 Search spaces and search landscapes**
- 4 Smarter hill-climbing**

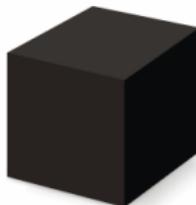
# Black-box optimisation



Source

A **black-box** is a function  $f$  which we can **run** (can call  $f(x)$ ), but we can't **read**

# Black-box optimisation



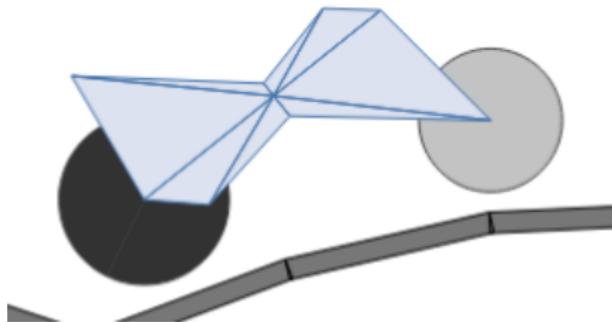
Source

A **black-box** is a function  $f$  which we can **run** (can call  $f(x)$ ), but we can't **read**

OR, we have the code, but it's so complex we can't see any easy relationship between input and output.

Either way we don't know the **gradient**.

# Example: simulation



[rednuht.org](http://rednuht.org)

Objective: distance travelled in the simulation.

Definitely not a linear objective! In fact it is hard to say anything *a priori* about what DV values are good, or about the gradient.

A car is defined by:

- Shape (8 floats, 1 per vertex)
- Wheel size (2 floats, 1 per wheel)
- Wheel position (2 ints, 1 per wheel)
- Wheel density (2 floats, 1 per wheel) darker wheels mean denser wheels
- Chassis density (1 float) darker body means denser chassis

# Metaheuristics

When the objective is a complex, black-box function of the input, we resort to **weak** methods, such as **metaheuristics**:

- Hill-climbing
- Simulated annealing
- Genetic algorithms
- Particle swarm

# Black-box optimisation = black art

Metaheuristics are **randomised** and there is **no guarantee** they will find the optimum.

# Black-box optimisation = black art

Metaheuristics are **randomised** and there is **no guarantee** they will find the optimum.

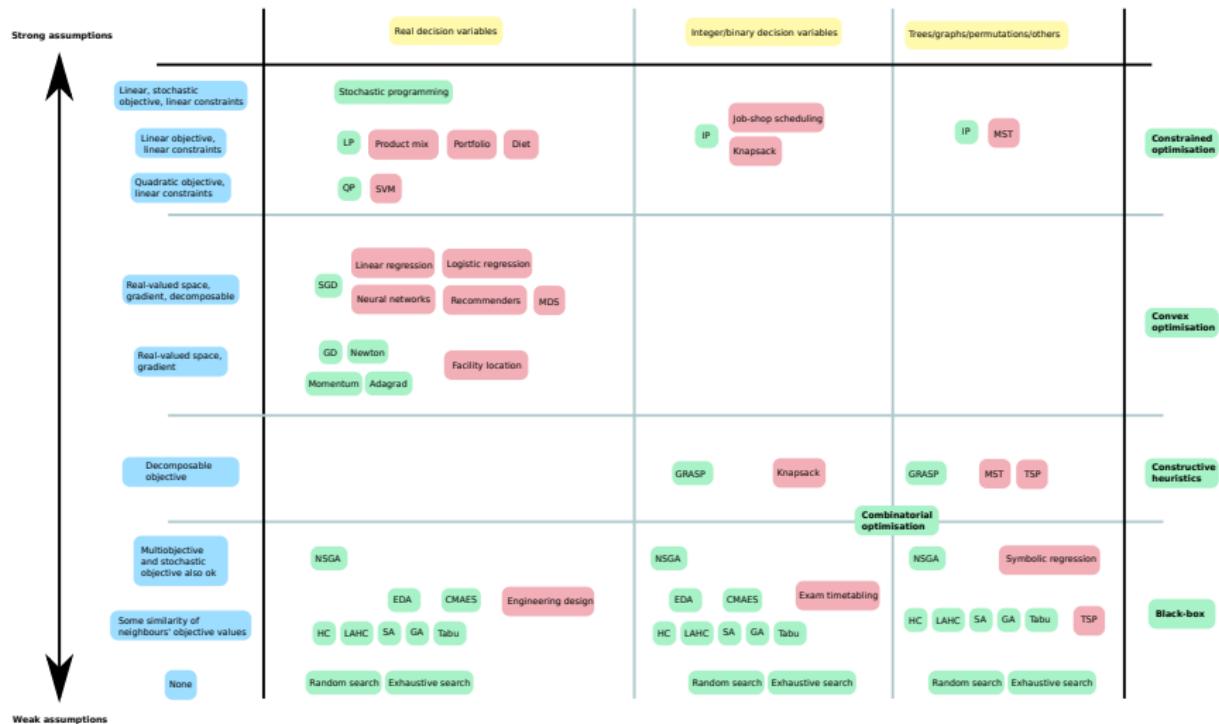
Thus, they are not true algorithms! (But we often call them algorithms for convenience.)

# Black-box optimisation = black art

Metaheuristics are **randomised** and there is **no guarantee** they will find the optimum.

Thus, they are not true algorithms! (But we often call them algorithms for convenience.)

Making them work sometimes relies on unwritten knowledge, rules of thumb, creativity...

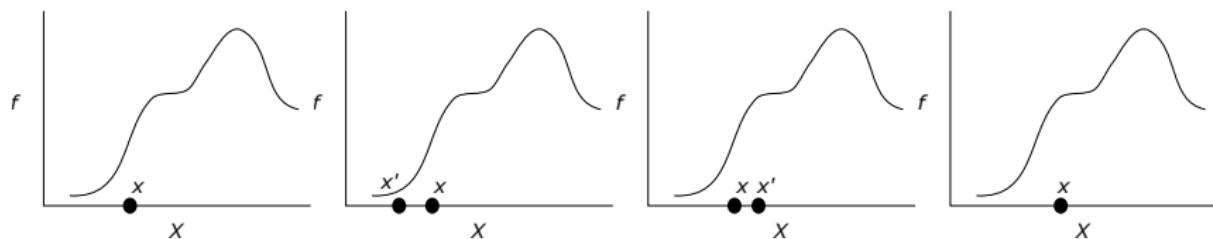


# Overview

- 1 What is black-box optimisation?
- 2 **Hill climbing**
- 3 Search spaces and search landscapes
- 4 Smarter hill-climbing

# Hill-climbing, the simplest metaheuristic

Idea: choose a random point  $x$  as our **current point**. Then try out another randomly-chosen point  $x'$  **near to  $x$** . If  $f(x') > f(x)$ , then **move** to  $x'$ , i.e. set  $x := x'$ . Repeat.



# Hill-climbing, the simplest metaheuristic

```
def hill_climb(f, init, nbr, its=10000):
    """
        f: objective function X -> R
        init: function giving random element x of X
        nbr: function X -> X, giving a neighbour of x
        its: number of iterations
        return: best ever x
    """
    x = init() # make a random point
    for i in range(its):
        xnew = nbr(x) # make new point by changing x
        if f(xnew) > f(x): # if it's better
            x = xnew # step to the new point
    return x
```

# Near?

- Our idea is to try randomly-chosen points **near** the current point.
- What does “near” mean?
- In the case of bitstrings, “near” means “create by flipping one bit”:  
E.g. 00110110 -> 01110110
- According to **Hamming distance**, the distance between these two bitstrings is 1 (because one bit differs).
- Or we might just say they are **neighbours**.

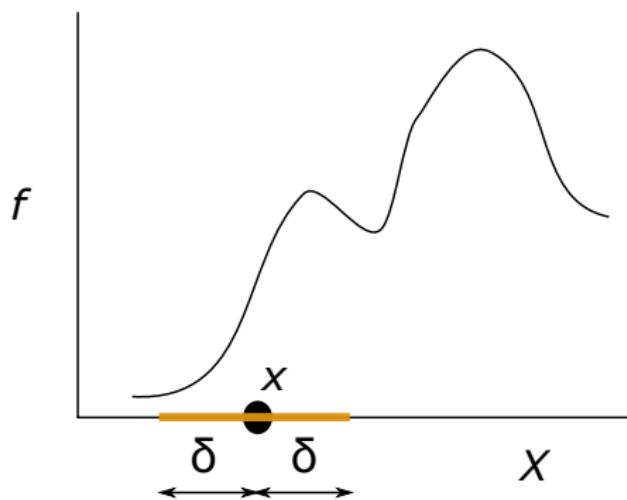
# Hill-climbing operators

```
def bitstring_init(n):
    # uniform sampling from X
    return [random.randrange(2) for i in range(n)]

def bitstring_nbr(x):
    # make a "blind", random change
    x = x.copy() # don't change x itself
    i = random.randrange(len(x))
    x[i] = 1 - x[i]
    return x
```

# Neighbours in real-valued search spaces

- In  $\mathbb{R}^1$ : a point  $x$  is just a number,
- So a neighbour  $x'$  of  $x$  could be defined as a number such that  $|x - x'| < \delta$  for some small  $\delta$ .
- Or define  $x' = x + N(0, \delta)$ , where  $N$  is a normal distribution with mean 0 and standard deviation  $\delta$



# Neighbours in real-valued search spaces

```
def real_init(n):
    return [random.random() for i in range(n)]

def real_nbr(x):
    x = x.copy()
    i = random.randrange(n)
    # add a small constant in range [-delta, delta]
    delta = 0.3
    x[i] += random.random() * 2 * delta - delta
    return x
```

## A small detail: DV bounds

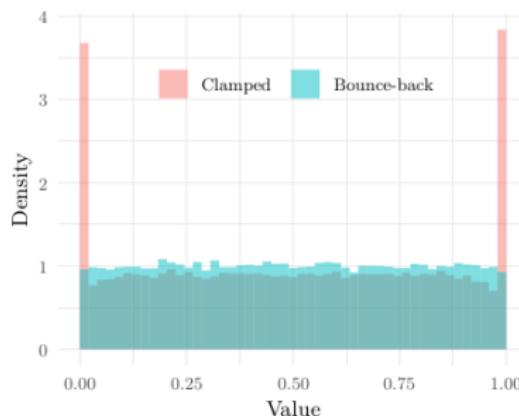
Suppose we have lower and upper bounds on DVs. What should we do if our nbr function gives a value that violates a bound? E.g. suppose UB=1.0 and nbr gives 1.1. Options:

- **Clamp** values at the bounds: output = 1.0
- **Bounce-back**, i.e. reflect the value: output = 0.9.

# A small detail: DV bounds

Suppose we have lower and upper bounds on DVs. What should we do if our nbr function gives a value that violates a bound? E.g. suppose UB=1.0 and nbr gives 1.1. Options:

- **Clamp** values at the bounds: output = 1.0
- **Bounce-back**, i.e. reflect the value: output = 0.9.



Nordmoen et al. (2020) show that bounce-back is better

# Exploration and exploitation

## Exploration:

- The search algorithm bravely tries out new areas of the search space
- E.g. **random search**

## Exploitation:

- The search algorithm concentrates on areas **near** known-good areas
- E.g. **hill-climbing**

# Exploration and exploitation

## Exploration:

- The search algorithm bravely tries out new areas of the search space
- E.g. **random search**

## Exploitation:

- The search algorithm concentrates on areas **near** known-good areas
- E.g. **hill-climbing**

More sophisticated algorithms are in the spectrum between the two.



imgflip.com

JHKE-CLARK.TUMBLR

The great dilemma of black-box optimisation

# Exploration versus exploitation in $\mathbb{R}^n$

- As we just saw, in  $\mathbb{R}^n$ , we can control the **step-size**,  $\delta$
- Small  $\delta \rightarrow$  small steps  $\rightarrow$  more exploitation
- Large  $\delta \rightarrow$  large steps  $\rightarrow$  more exploration

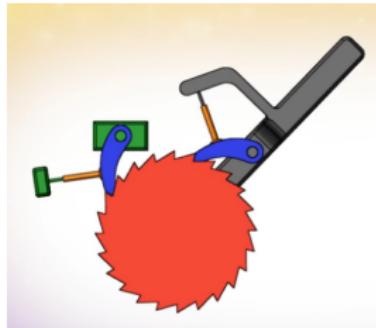
# Why does hill-climbing work?

The nbr function is **blind** or **undirected**. We do **not** try to design a neighbour operator that gives improvements.

# Why does hill-climbing work?

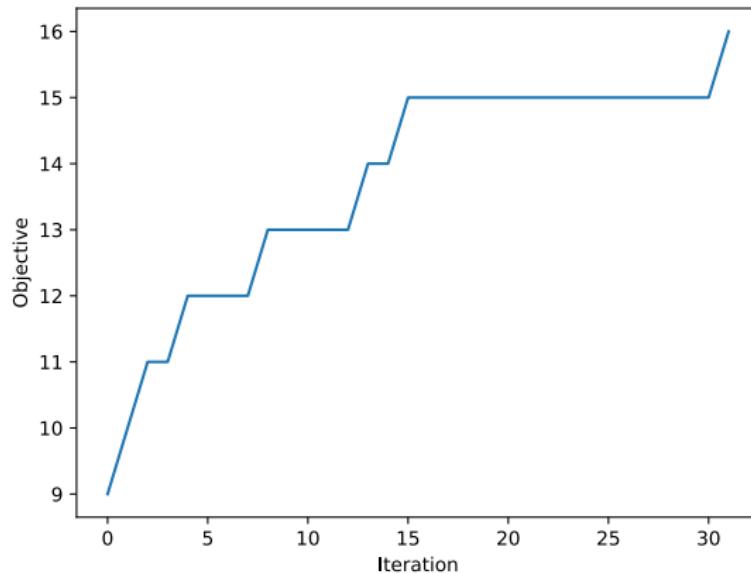
The nbr function is **blind** or **undirected**. We do **not** try to design a neighbour operator that gives improvements.

The power of hill-climbing is from the **selection** (the if-statement): we only accept moves which **improve**.



The ratchet, from <https://youtu.be/EpVPG2fZrHE?t=8>

# Why does hill-climbing work?



Thanks to the ratchet, the objective value of the best individual improves **monotonically** (never disimproves).

# When is hill-climbing better than random search?

Hill-climbing makes weak assumptions:

- Suppose whenever  $x_1$  and  $x_2$  are **similar** in some way, their objective values are also similar
- If  $|x_1 - x_2|$  is small, then  $|f(x_1) - f(x_2)|$  is small (reminiscent of defn of **continuity** in calculus)
  - Then if we have found a **good**  $x$ , it makes sense to try out some (randomly-chosen) **neighbours**, not points randomly sampled from the entire space.
  - (E.g. [Wallace and Aleti, \*The Neighbours' Similar Fitness Property for Local Search\*](#))
- Our nbr function can always find an improvement (eventually) and so make progress.

# Quiz

Hill-climbing will maximise a function  $f$ . If I want to minimise instead:

- I need a different algorithm
- I can use hill-climbing to maximise  $-f$
- I need to calculate the gradient of  $f$
- I'm out of luck because minimisation is impossible

# Quiz

Hill-climbing will maximise a function  $f$ . If I want to minimise instead:

- I need a different algorithm
- I can use hill-climbing to maximise  $-f$
- I need to calculate the gradient of  $f$
- I'm out of luck because minimisation is impossible

**Answer:** I can use hill-climbing to maximise  $-f$

# Quiz

Random search is exploitative: True or False?

# Quiz

Random search is exploitative: True or False?

**Answer:** no, in fact RS is the most explorative algorithm.

# Overview

- 1 What is black-box optimisation?
- 2 Hill climbing
- 3 Search spaces and search landscapes**
- 4 Smarter hill-climbing

# When does hill-climbing fail?

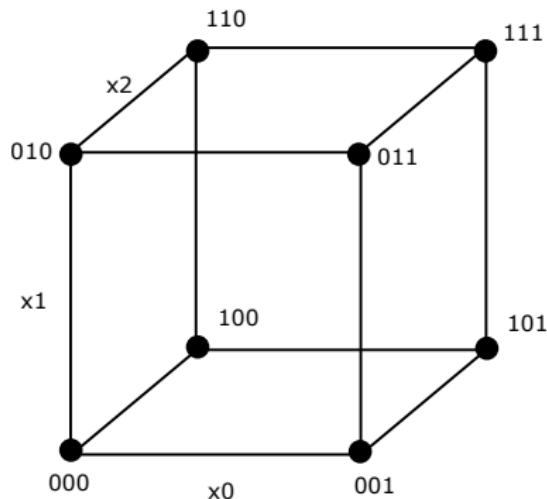
To answer this question, we need to understand the concepts of **search landscapes** and **local optima**. But before we can define **search landscape** we have to recall **search spaces**.

# Search spaces: Cartesian spaces

- This is the most common type, as in LP and IP
- We have  $n$  **decision variables**  $x_0, \dots, x_{n-1}$
- Usually all of same type, e.g. binary, integer, categorical, or real (possibly with bounds)
- Then  $X$  is the **Cartesian product** of the decision variables.
- Items in the search space are then **vectors**.

# Hamming cube

$X = \mathbb{H}^n$  (Hamming cube in  $n$  dimensions, i.e. space of bitstrings of length  $n$ )

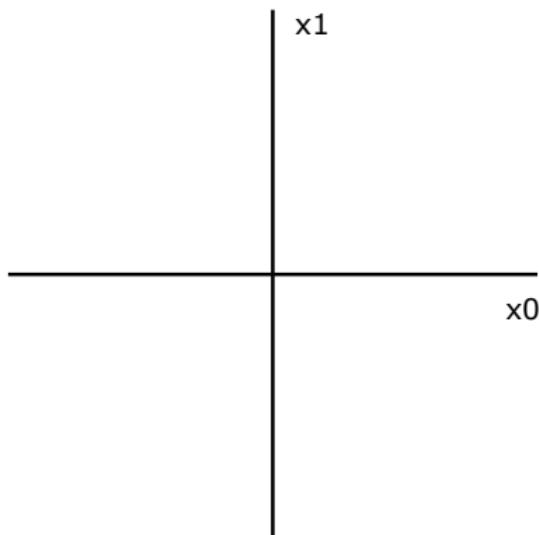


Examples:

- Binary guessing game (Week 01)
- Binary Unit Commitment: a single power plant is on or off at  $n$  intervals
- Feature Selection: given  $n$  features we choose a subset for machine learning.

# Real vector spaces

$X = \mathbb{R}^n$  (Cartesian space in  $n$  dimensions, i.e. space of real vectors of  $n$  components)

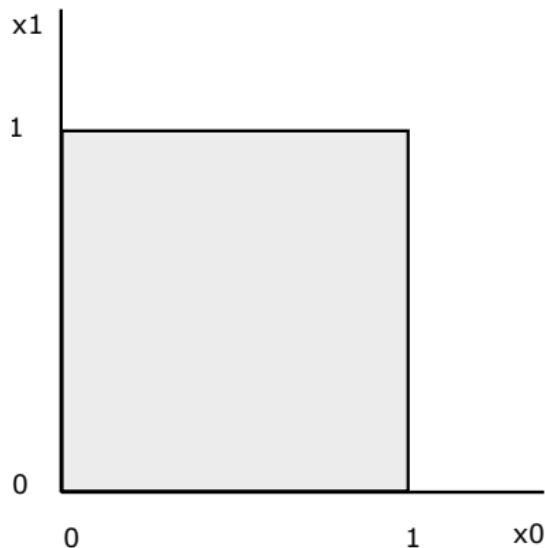


Examples:

- Logistic Regression (the parameters  $w_i$  are the optimisation DVs)
- Portfolio allocation (the amount to invest in each stock is a real vector)

# Unit hypercube

$X = [0, 1]^n$  (Unit hypercube in  $n$  dimensions, a subset of  $\mathbb{R}^n$ )



Example:

- Portfolio allocation, but all real parameters are **bounded to be in some range**, e.g.  $[0, 1]$ .

## Search spaces: other cases

Above the search space was a Cartesian product of DVs.

But the search space could be **any set**, e.g. a particular set of trees, or permutations, or graphs, or something else. We'll see these later in the module.

# What is a landscape?

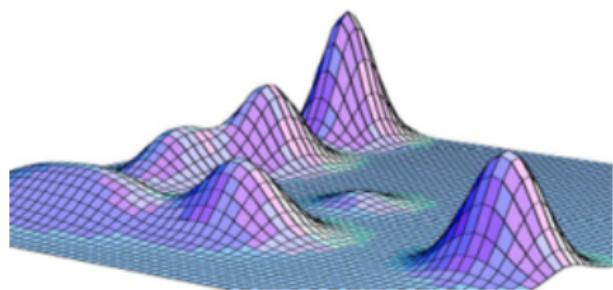
A search **landscape** is a search space  $X$ , together with a neighbourhood function  $\text{nbr}$ , and an objective function  $f$ .

Landscape is a triple:  $(X, \text{nbr}, f)$ .

# Metaphor

Imagine search as trying to walk around in the mountains, when you can't see far, and only know the height of points you have already visited.

- The search space is the north-south and east-west axes.
- The neighbour function is your footstep.
- The objective function is altitude.

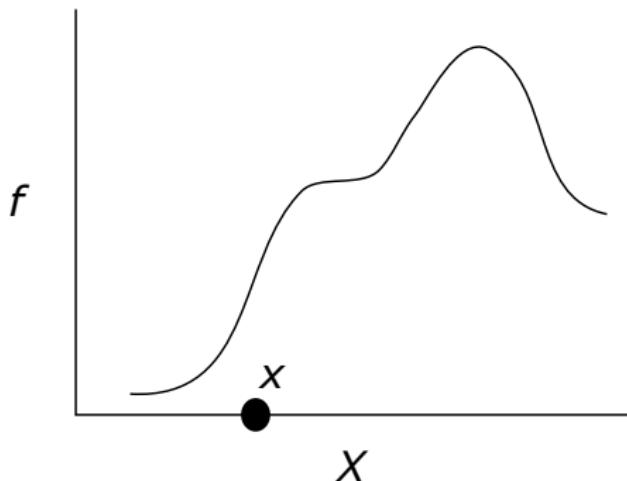


From [here](#)

The **landscape** metaphor is due to Sewall Wright.

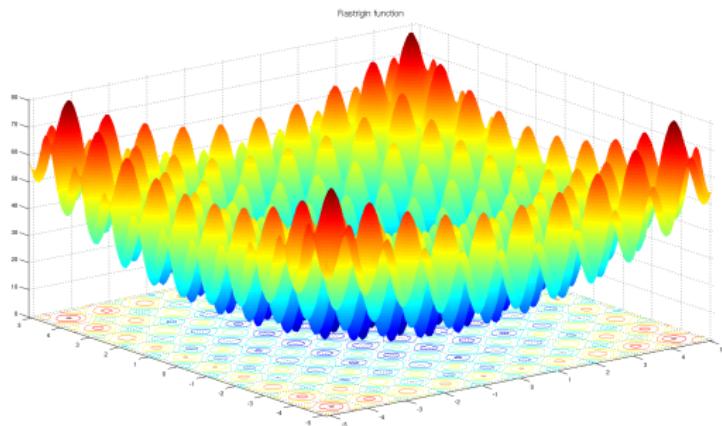
# A landscape on 1-D Euclidean space

- 1-D Euclidean space  $\mathbb{R}^1$
- Neighbour function just adds a small random number (positive or negative)
- Objective function as shown.



# A landscape on 2-D Euclidean space

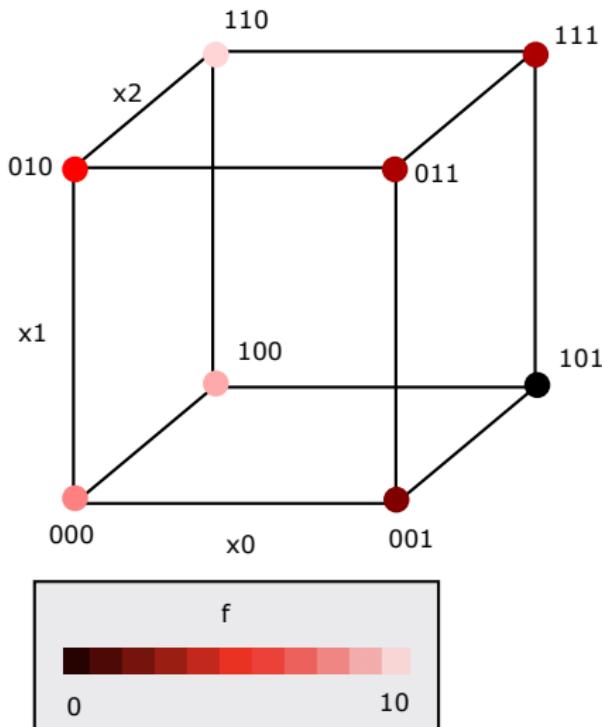
- 2-D Euclidean space  $\mathbb{R}^2$
- Neighbour function adds a random number to a randomly-chosen coordinate
- E.g. Rastrigin function, a well-known test function.



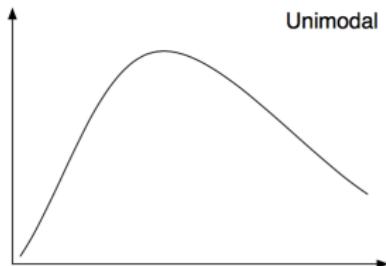
Rastrigin function, image by Diegotorquemada

# A landscape on the Hamming cube

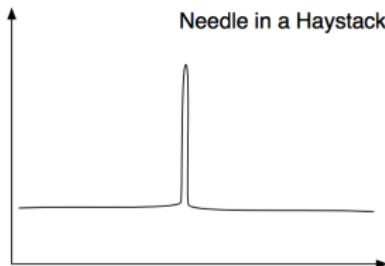
- The Hamming cube  $X = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- Neighbour function is “flip one bit”
- Objective function as shown.



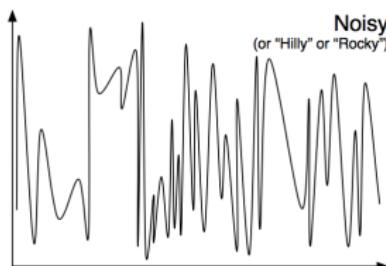
# Different types of landscapes



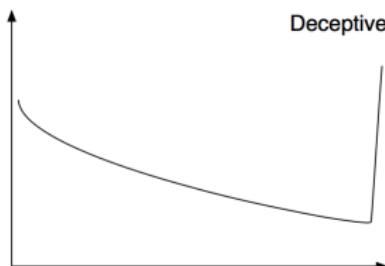
Unimodal



Needle in a Haystack



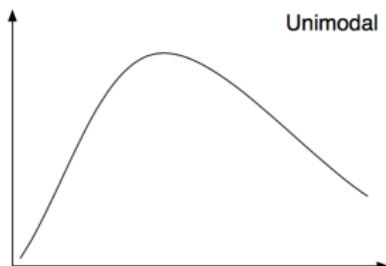
Noisy  
(or "Hilly" or "Rocky")



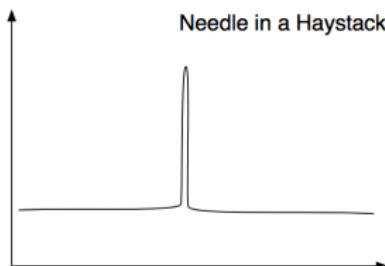
Deceptive

Luke, *Essentials*

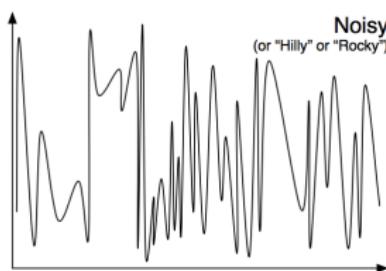
# Different types of landscapes



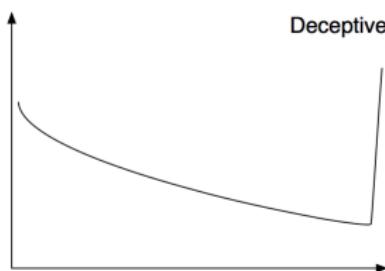
Unimodal



Needle in a Haystack



Noisy  
(or "Hilly" or "Rocky")



Deceptive

Luke, *Essentials*

Reminder: in reality we **can't see** a plot of the landscape!

# Landscape properties

- Unimodal (one peak) versus multimodal (many peaks)
- Smooth (continuous, or at least neighbours usually have similar values) versus rugged
- Informative versus deceptive (“gradient” leads towards the global optimum, or away)

# Quiz

Consider an LP **maximisation** problem with 2 real DVs. What is the “mountain landscape” like? Is the objective function unimodal? Is it rugged? Is it deceptive?

# Quiz

Consider an LP **maximisation** problem with 2 real DVs. What is the “mountain landscape” like? Is the objective function unimodal? Is it rugged? Is it deceptive?

The landscape would look like a subset of an inclined plane, e.g. think of walking up **one face** of a pyramid. It is unimodal, not rugged, not deceptive.



# When does hill-climbing fail?

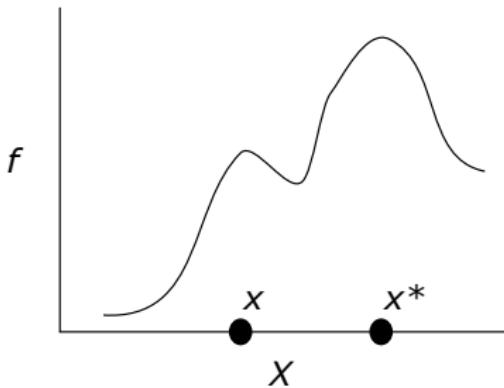
Hill-climbing fails on problems which are multimodal, rugged, and/or deceptive.

# When does hill-climbing fail?

Hill-climbing fails on problems which are multimodal, rugged, and/or deceptive.

All of these involve **local optima** in some way.

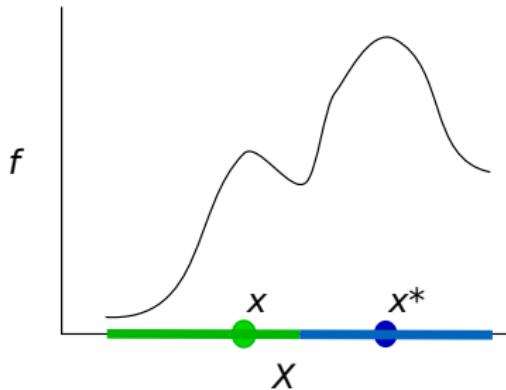
# Global and local optima



A **global optimum** is a point  $x^*$  such that  $f(x^*) \geq f(y) \forall y \in X$ .  
There may be multiple global optima (all equally good).

A **local optimum** is a point  $x$  which is **not a global optimum** such that  $f(x) \geq f(y) \forall$  neighbours  $y$  of  $x$ .

# Basins of attraction



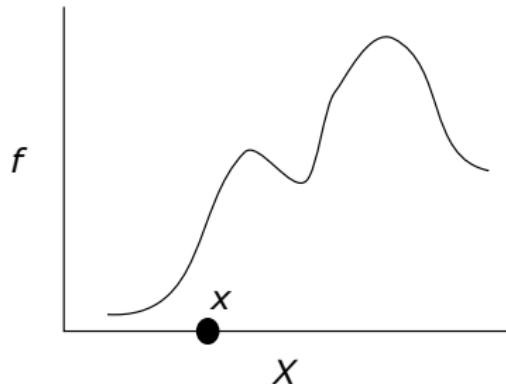
- Each optimum has a **basin of attraction**.
- A basin of attraction  $B_i$  is a subset of the search space corresponding to a particular optimum  $x_i$
- If the current point  $x \in B_i$ , then the hill-climb will end at  $x_i$ .

# Escaping local optima

Suppose we reach a local optimum. What would we need to do to **escape**...

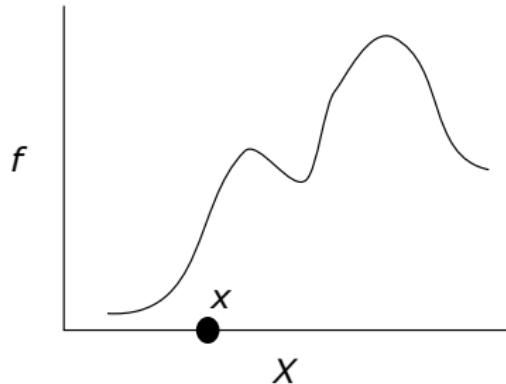
- 1 ... in a **bitstring** problem?
- 2 ... in a **real-valued** problem?

# How do local optima arise?



- If a DV has a non-linear effect, e.g. quadratic or cubic, or something weirder (e.g.: pretend the above image is a high-order polynomial in a single DV  $x$ ).

# How do local optima arise?



- If a DV has a non-linear effect, e.g. quadratic or cubic, or something weirder (e.g.: pretend the above image is a high-order polynomial in a single DV  $x$ ).
- If there is a dependency in the objective function between two or more DVs (epistasis).

# Dependency between variables

**Why don't we just** try all possible values for  $x_0$  and choose the best; then all values for  $x_1$  and choose the best; and so on?

# Example: Cookies

- “finding the most delicious chocolate chip cookie recipe from a parameterized space of recipes”
- “Parameters included baking soda, brown sugar, white sugar, butter, vanilla, egg, flour, chocolate, chip type, salt, cayenne, orange extract, baking time, and baking temperature.”
- From Golovin et al., *Google Vizier*, KDD ’17

# Dependency between variables

This example illustrates the issue of **dependency** between variables. We would not say “I’m going to try varying the amount of butter (say), and then I’ll know the right amount of butter. Then I’ll proceed to optimise the sugar in the same way. then the baking temperature, and the number of eggs, etc.”. **This doesn’t work**, because the optimum amount of butter depends on the amount of eggs, etc. We might reach a local optimum.

**When there is any dependency between variables, we have to optimise them all together.**

# Quiz

The objective function in cookie optimisation seems subjective. How could we measure it in a (somewhat) objective way?

# Quiz

The objective function in cookie optimisation seems subjective. How could we measure it in a (somewhat) objective way?

For each possible recipe we could bake a large batch and run a survey.

# Research topics

An interesting area of research: what makes one problem harder than another? We don't have a formula for it. But we do have some good clues.

- Search space size/dimensionality
- Epistasis (dependency between DVs)
- Landscape ruggedness (many local optima)
- FDC (correlation between objective values and distance from optimum)
- Locality (correlation between objective values of neighbours)

(You don't need to remember these definitions.)

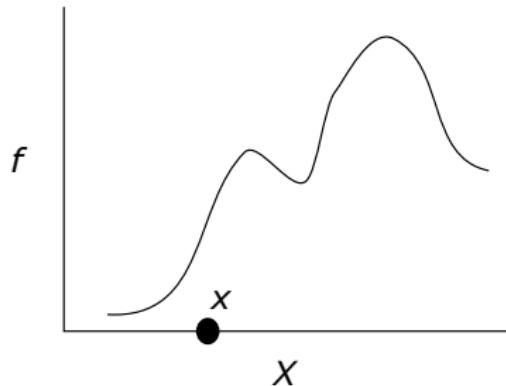
# Overview

- 1** What is black-box optimisation?
- 2** Hill climbing
- 3** Search spaces and search landscapes
- 4** Smarter hill-climbing

# Recall: Why does hill-climbing fail?

Answer: it gets stuck at local optima.

# How to escape local optima



- 1 Allow **larger jumps** in the nbr function
- 2 Restart (**iterated hill-climbing**)
- 3 Allow **disimproving moves** (**simulated annealing** and **late-acceptance hill-climbing**)
- 4 Use **multiple search points** with **information transfer** between them (**genetic algorithm**).

## Larger jumps in the nbr function

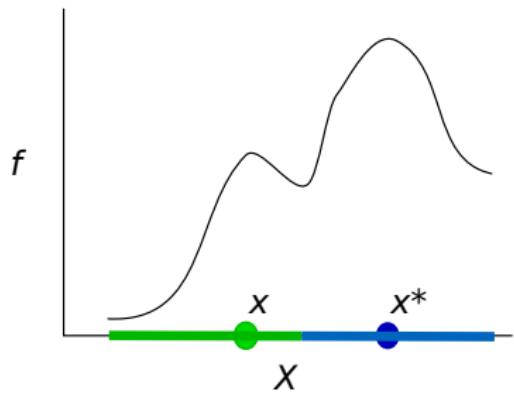
We already saw this when hill-climbing in  $\mathbb{R}^n$ : we can increase  $\delta$  to make larger jumps and sometimes escape local optima.

## Larger jumps in the nbr function

We already saw this when hill-climbing in  $\mathbb{R}^n$ : we can increase  $\delta$  to make larger jumps and sometimes escape local optima.

How could we do this in a bitstring problem?

# Hill-climbing with restarts



- Idea: start again from random  $x$  and hope  $x \in B_i$  such that  $x^* \in B_i$  also.
- Implementation: just put hill-climbing inside a loop!

# Disimproving moves

- Simulated annealing
- Late-acceptance hill-climbing

# Simulated annealing

- Simulated annealing is a variant of hill-climbing where we sometimes **accept disimproving moves**, i.e.  $x := x'$  even though  $x'$  is worse!
- The name **simulated annealing** refers to the **annealing** process used in metallurgy, where cooling a metal slowly can help it reach a better configuration at the atomic level, hence the resulting object is stronger
- Simulated annealing was introduced by Kirkpatrick et al. (1983).

## SA: accepting disimprovement

- If  $f(x')$  is better than  $f(x)$ , we always accept  $x'$

# SA: accepting disimprovement

- If  $f(x')$  is better than  $f(x)$ , we always accept  $x'$
- A disimproving move is accepted with probability

$$p = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

## SA: accepting disimprovement

- If  $f(x')$  is better than  $f(x)$ , we always accept  $x'$
- A disimproving move is accepted with probability

$$p = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

- The probability of accepting disimprovement is small when  $x'$  is much worse, but a bit larger when  $x'$  is nearly as good as  $x$

# SA: accepting disimprovement

- If  $f(x')$  is better than  $f(x)$ , we always accept  $x'$
- A disimproving move is accepted with probability

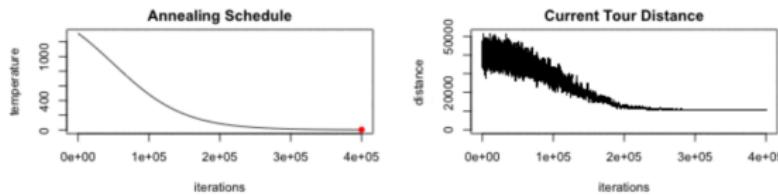
$$p = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

- The probability of accepting disimprovement is small when  $x'$  is much worse, but a bit larger when  $x'$  is nearly as good as  $x$
- $T$  is **temperature**, which decreases during the run
- The probability of accepting disimprovement is large at the start, but small later.

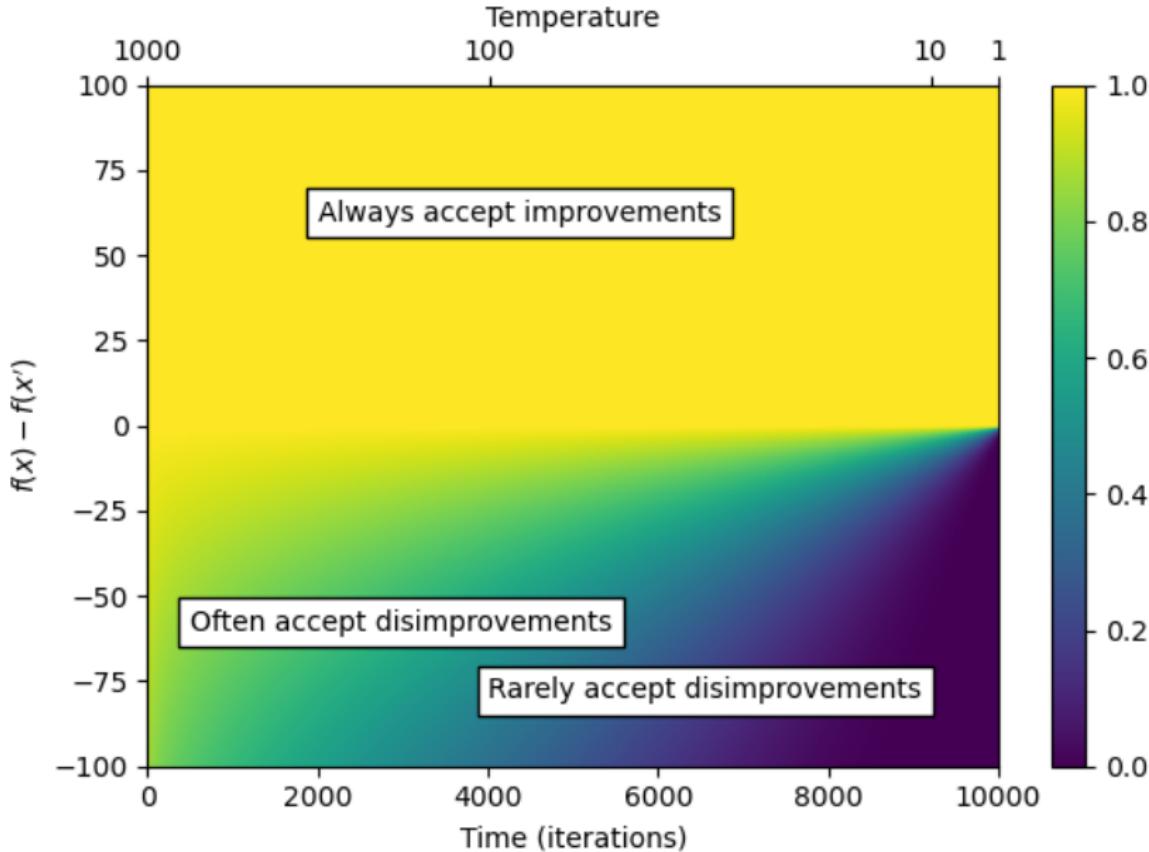
# SA: accepting disimprovement



Temperature decreases (“anneals”) over time, as shown here.



## Simulate Annealing: Probability of acceptance



```
def anneal(f, nbr, init, its):
    # assume we are minimising
    x = init() # initial random solution
    fx = f(x)
    T = 1.0 # initial temperature
    alpha = 0.99 # temperature decay per iteration
    for i in range(its):
        xnew = nbr(x) # generate a neighbour of x
        fxnew = f(xnew)
        if (fxnew < fx or
            random() < exp((fx - fxnew) / T)):
            x = xnew
            fx = fxnew
        T *= alpha
    return x, fx
```

# Simulated annealing

- Advantage: we can escape local optima. Performance is **much** better than simple hill-climbing.
- Disadvantage: hyperparameters.

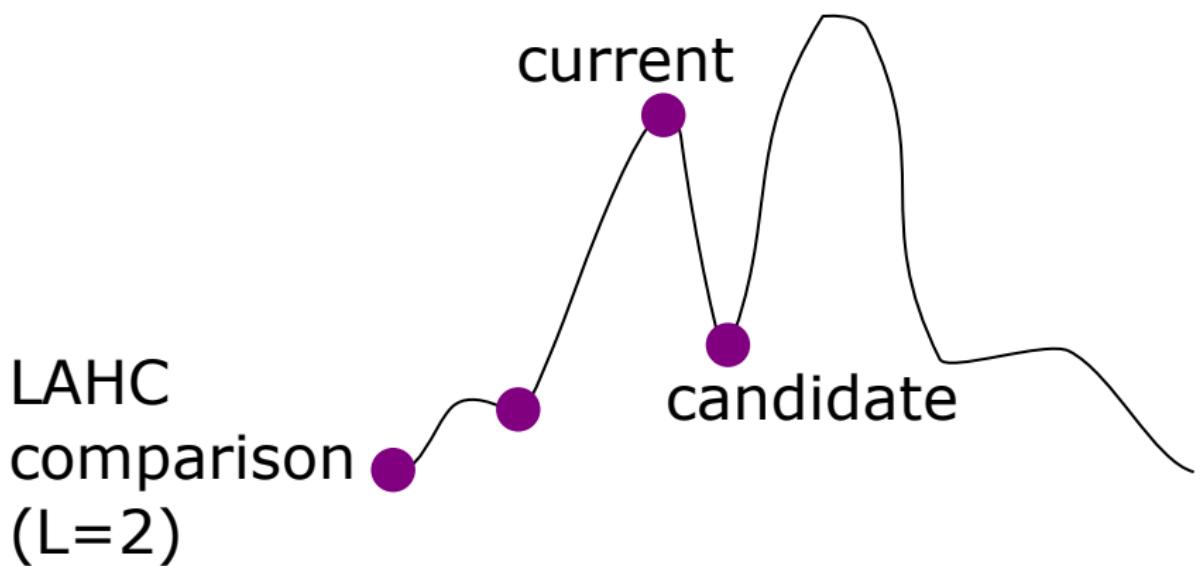
Nice explanation [here](#)

The big problem with simulated annealing is that the user has to think about the annealing schedule. What initial value should  $T$  have, and how quickly should it decrease? These are **hyperparameters**. SA might work well with one configuration of them, but badly with another. There's no configuration which works well for all problems. So, SA is harder to use than HC, according to Burke & Bykov (2008).

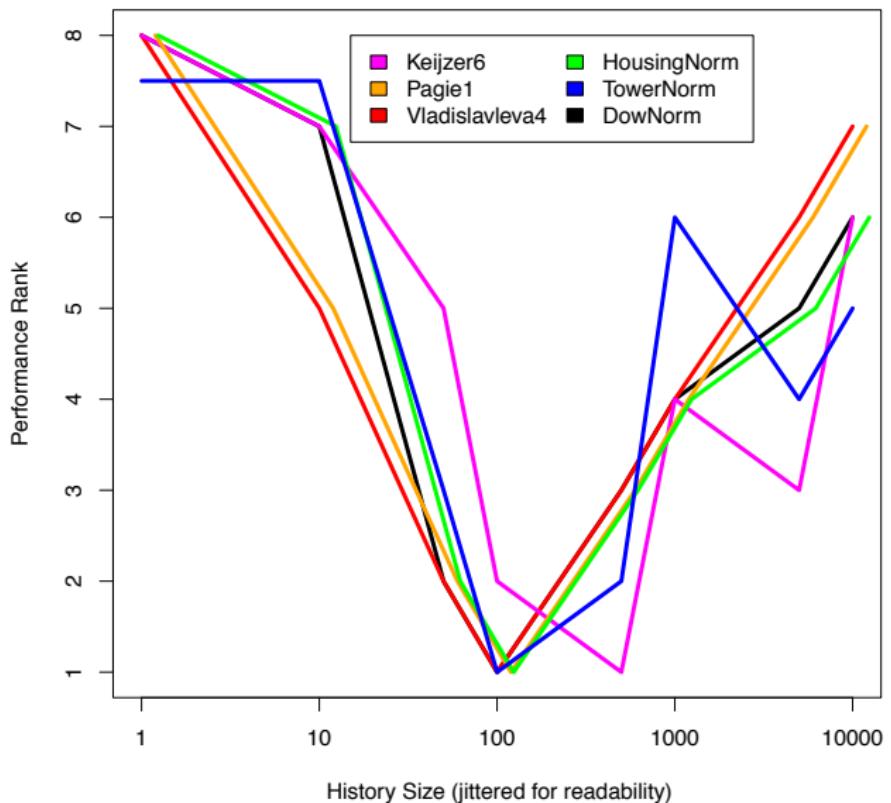
# Late-acceptance hill-climbing

- LAHC is an alternative to SA, due to Burke & Bykov
- Again, hope to escape local optima by sometimes accepting disimproving moves
- Keep a *history* of recent  $f$  values, of length  $L$
- Instead of accepting  $x'$  when it is better than  $x$  (as in hill-climbing),
- we accept  $x'$  when it is better than  $x_L$ , that is the  $x$  we had  $L$  steps previously: we make the acceptance decision **late**
- $L$  the only hyperparameter, easy to control.

# Late-acceptance hill-climbing



# $L$ is easy to control



```
def LAHC(L, n, C, init, nbr):
    s = init()                      # initial solution
    Cs = C(s)                       # cost of current solution
    best = s                         # best-ever solution
    Cbest = Cs                       # cost of best-ever
    f = [Cs] * L                    # initial history
    for I in range(n):             # number of iterations
        s_ = nbr(s)                 # candidate solution
        Cs_ = C(s_)                # cost of candidate
        if Cs_ < Cbest:            # minimising
            best = s_
            Cbest = Cs_
        v = I % L                  # v indexes f circularly
        if Cs_ <= f[v] or Cs_ <= Cs:
            s = s_
            Cs = Cs_
            f[v] = Cs
    return best, Cbest
```

# Quiz

In LAHC, if we set  $L = 1$ , what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

# Quiz

In LAHC, if we set  $L = 1$ , what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

**Answer:** it reduces to hill-climbing, because it always compares to the current best point, never to an old one.

# Quiz

In LAHC, if we set  $L = \text{its}$ , what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

# Quiz

In LAHC, if we set  $L = \text{its}$ , what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

**Answer:** it behaves like a random walk, because it never compares new points to anything.

A random **walk** is not the same as random **search**. It is like hill-climbing where we **always** accept the move.

# Quiz

In LAHC, increasing  $L$  leads to:

- More exploration
- More exploitation

# Quiz

In LAHC, increasing  $L$  leads to:

- More exploration
- More exploitation

**Answer:** Larger  $L$  allows more disimproving moves, so more exploration.

# Multiple points and information transfer

Our final option for improving on hill-climbing is to use **multiple search points** with **information transfer** between them.

This leads to the idea of a **genetic algorithm**. We'll study this next week.

# Suggested readings

Two research papers in Blackboard – not required reading:

- Kirkpatrick et al. (SA)
- Burke & Bykov (LAHC for exam timetabling)

# Lecture 06 - Genetic Algorithms

Optimisation CT5141

James McDermott

NUI Galway

2020

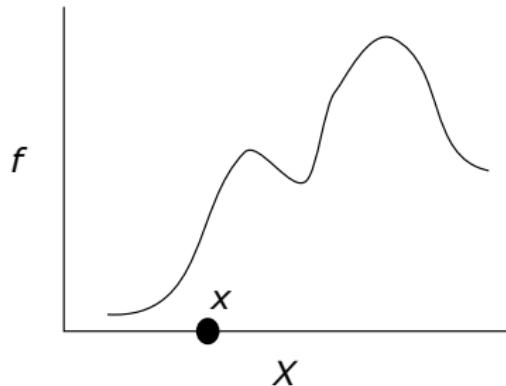


OÉ Gaillimh  
NUI Galway

# Overview

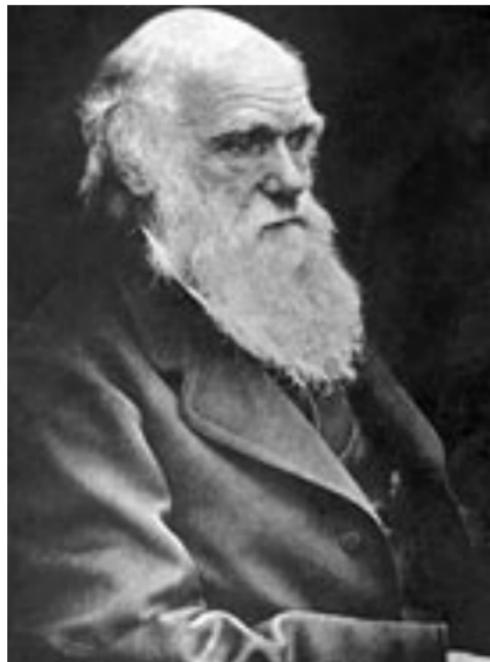
- 1 Introduction to Genetic Algorithms**
- 2 Operators**
- 3 Genotypes and phenotypes**
- 4 Experimental methodology**

# Reminder: How to escape local optima



- 1 Allow **larger jumps** in the nbr function
- 2 Restart (**iterated hill-climbing**)
- 3 Allow **disimproving moves** (**simulated annealing** and **late-acceptance hill-climbing**)
- 4 Use **multiple search points** with **information transfer** between them (**genetic algorithm** or GA).

# Darwinian evolution



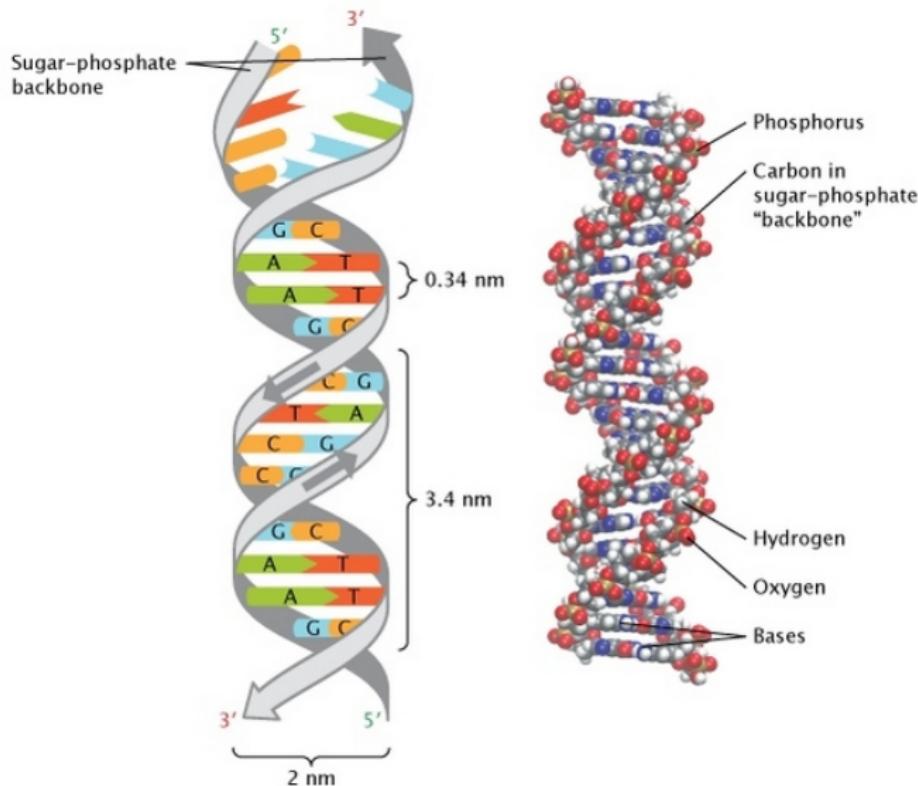
“If I were to give an award for the single best idea anyone ever had, I’d give it to Darwin.” – Dennett (1995) “Darwin’s Dangerous Idea.”



Source: CNN

- Population of organisms
- Competition within and between species
- Only the fittest survive and reproduce
- Reproduction: mating and combination of genes
- Mutation of genes
- Inheritance (but no inheritance of acquired traits)

# Genes



# Inheritance and mixing of genes



From regenerationnet

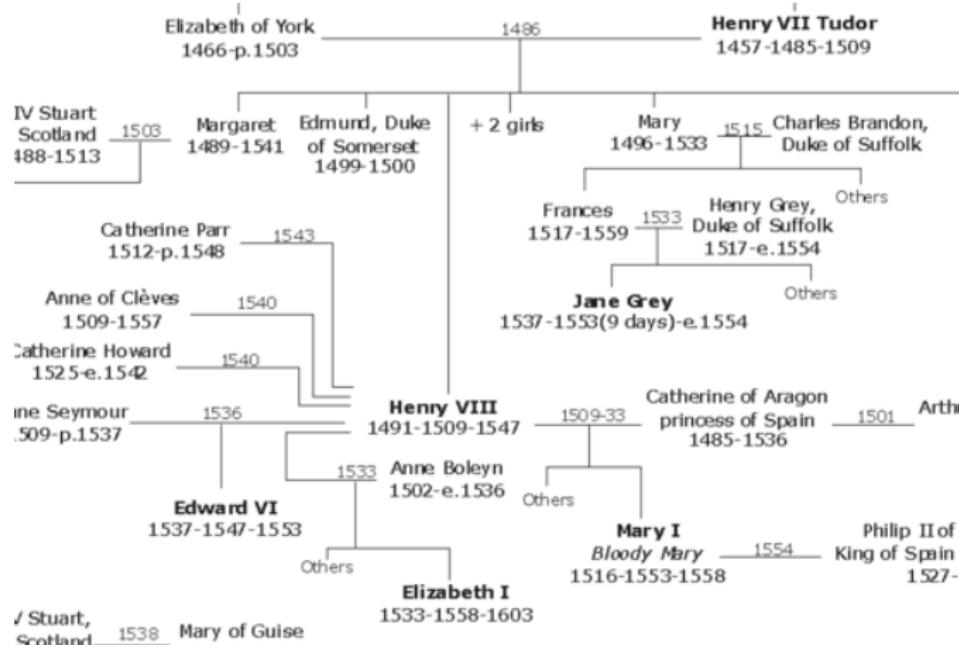
# Inheritance and mixing of genes



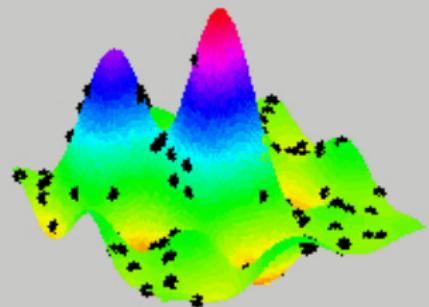
From regenerationnet

How would this tree look if we went back another 100 generations?

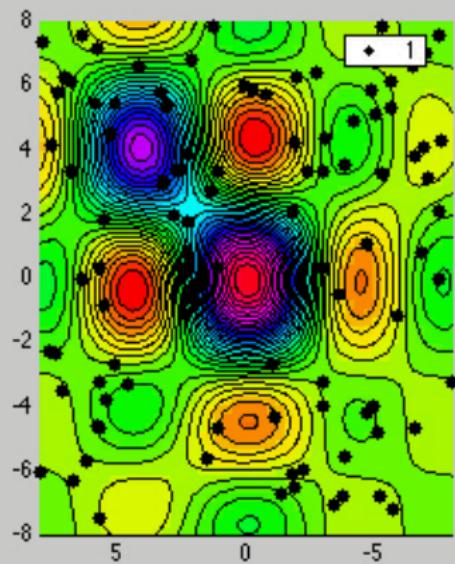
# Inheritance and mixing of genes



# Population change over time



Tom Cwik JPL/NASA



# Evolution = optimisation?



In nature, there is no explicit objective function! No-one is trying to optimise anything.

But there is an **implicit** objective function – ability to survive, compete, and reproduce.

This has the **effect** of optimisation – cheetahs gradually get faster.

# **Evolutionary algorithms**

**Evolutionary algorithms** are a **family** of black-box metaheuristic optimisation algorithms **inspired by** the main ideas of Darwinian evolution.

# Evolutionary algorithms

**Evolutionary algorithms** are a **family** of black-box metaheuristic optimisation algorithms **inspired by** the main ideas of Darwinian evolution.

- There is always a **population** – not just one current point
- There is crossover/recombination – an operator that takes in **two** solutions and outputs one or two **offspring**.
- There are **not** two sexes in the population – any individual can be recombined with any other.

# Terminology



Kirkpatrick

- **Evolutionary Algorithms** or **Evolutionary Computation** is an umbrella term
- The **Genetic Algorithm** (GA) is the central algorithm
- But really, the GA is a huge **family** of variant algorithms
- Sometimes with stupid names like **Intelligent Water-Drop Algorithm**

# Terminology

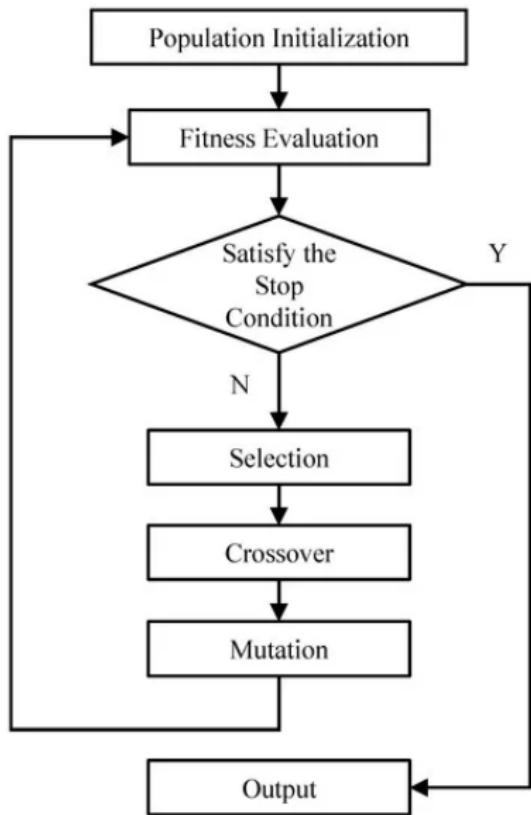
---

Other optimisation	Evolutionary
Decision variable	Gene
Point/Solution	Genotype/Individual
Multiple points	Population
Objective	Fitness
Neighbour (Combination)	Mutation
Iteration	Crossover
	Generation

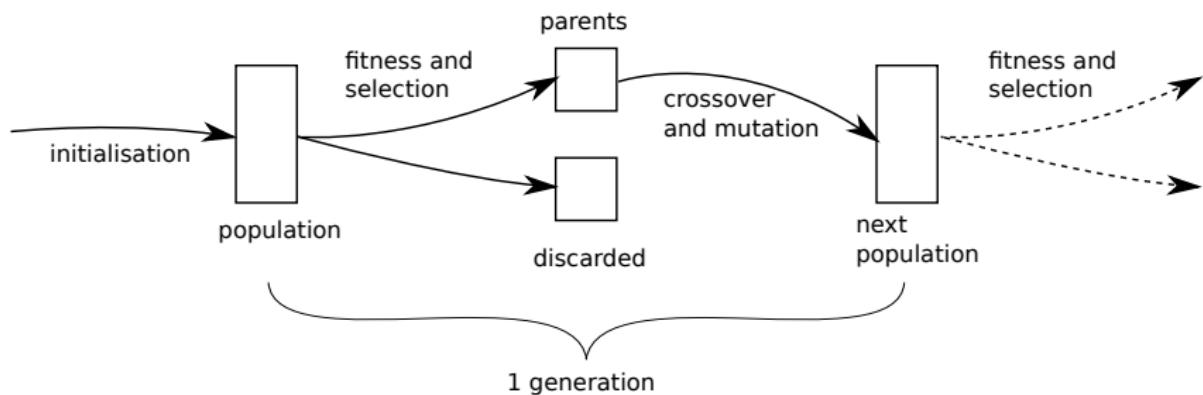
---

# GA flowchart

From Xiang et al



# GA loop



# Overview

- 1 Introduction to Genetic Algorithms
- 2 Operators
- 3 Genotypes and phenotypes
- 4 Experimental methodology

# Operators

GAs need a few main operators:

- Genetic operators
  - Initialisation
  - Mutation
  - Crossover
- Selection
- (Elitism)

# Operators

GAs need a few main operators:

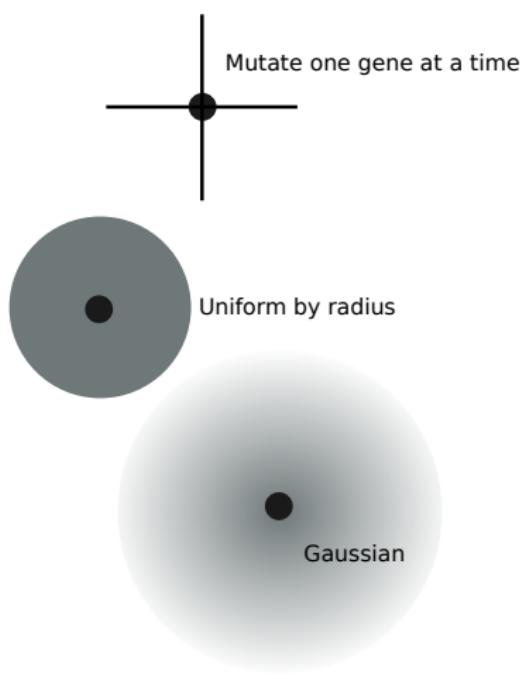
- Genetic operators
  - Initialisation
  - Mutation
  - Crossover
- Selection
- (Elitism)

For each of these, there are many possible ways to design and implement the operator.

# Initialisation

- We create a population of  $n$  individuals
- Usually **uniformly distributed** in the search space
- [init() for i in range(n)]

# Mutation



- **Mutation = neighbour**
- May be many possible mutation functions for a given search space, with different properties
  - E.g. some more explorative, some more exploitative
  - E.g. Gaussian mutation versus uniform mutation
  - E.g. mutate one gene chosen randomly, versus mutate all genes with a certain low probability, versus mutate all genes with a small step-size.

# Crossover

Crossover is the means of **information transfer** between individuals.

Crossover is the main feature that distinguishes a GA from a HC method.

Crossover is usually seen as **more important** than mutation. Mutation may be omitted or applied to a small proportion of the population.

# Crossover

Crossover is the means of **information transfer** between individuals.

Crossover is the main feature that distinguishes a GA from a HC method.

Crossover is usually seen as **more important** than mutation. Mutation may be omitted or applied to a small proportion of the population.

(The other feature is a **population** as opposed to a single current point, but crossover **implies** a population!)

# Crossover

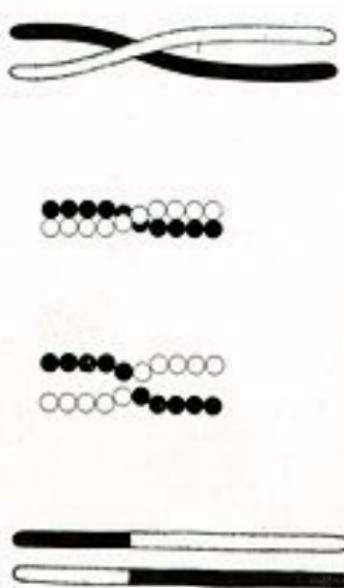


Fig. 64. Scheme to illustrate a method of crossing over of the chromosomes.

Thomas Hunt Morgan, 1916, taken from Wiki

- Each individual's **genome** is a **vector** of DVs.

# Crossover

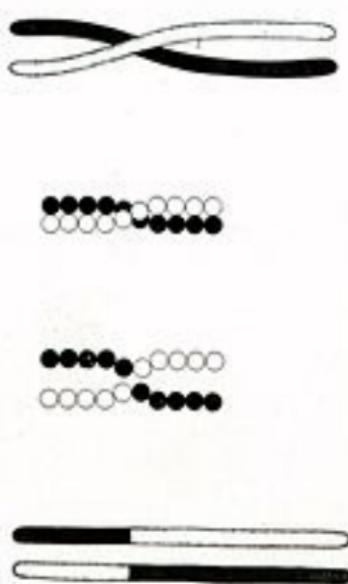
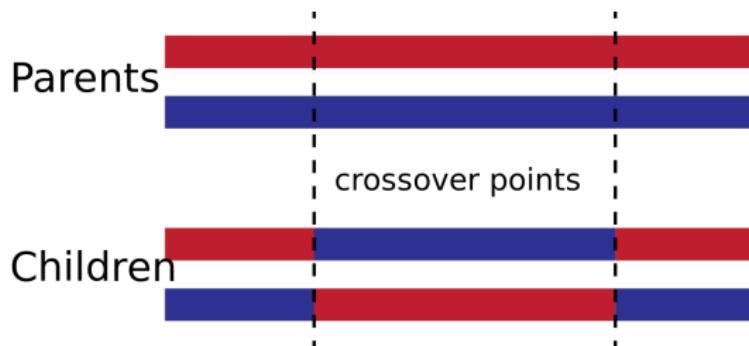


FIG. 64. Scheme to illustrate a method of crossing over of the chromosomes.

- Each individual's **genome** is a **vector** of DVs.
- The scheme shown here is called **one-point** crossover because we choose one (random) split-point.

Thomas Hunt Morgan, 1916, taken from Wiki

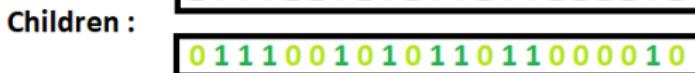
# Two-point crossover



This is **two-point** because we choose two split-points at random locations as shown.

# Uniform crossover

Every position gets a value chosen from the same position in one parent or the other



Uniform Crossover

# Uniform crossover

```
def uniform_crossover(x, y):
    c, d = [], []
    for xi, yi in zip(x, y):
        if random.random() < 0.5:
            c.append(xi); d.append(yi)
        else:
            c.append(yi); d.append(xi)
    return c, d
```

# Uniform crossover

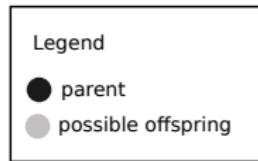
```
def uniform_crossover(x, y):
    c, d = [], []
    for xi, yi in zip(x, y):
        if random.random() < 0.5:
            c.append(xi); d.append(yi)
        else:
            c.append(yi); d.append(xi)
    return c, d
```

You can easily implement any of the crossover operators we have seen.

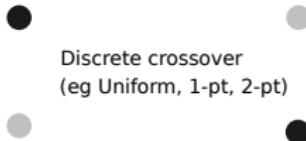
# Crossover offspring

Some operators are defined to return just one offspring; some return two.

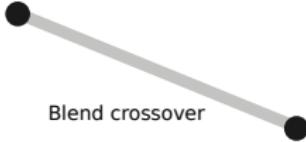
# Crossover: where do offspring go?



Discrete crossover  
(eg Uniform, 1-pt, 2-pt)



Blend crossover

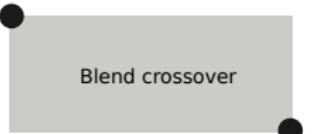
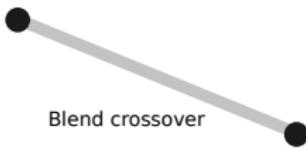
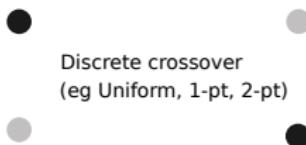
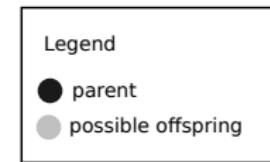


Blend crossover



- Offspring are usually somehow **intermediate** to their parents.

# Crossover: where do offspring go?



- Offspring are usually somehow **intermediate** to their parents.
- This has important implications! Crossover can only search “within” the area covered by the population (for some definition of “within”). A little mutation is needed to go outside it sometimes.

# Linkage between DVs

- Recall the California Manufacturing Co. factory/warehouse IP problem: we can only build a warehouse in LA if we also build a factory there
- That is a **dependency** between two DVs (genes)
- We implemented it as a **constraint**

# Linkage between DVs

- Recall the California Manufacturing Co. factory/warehouse IP problem: we can only build a warehouse in LA if we also build a factory there
- That is a **dependency** between two DVs (genes)
- We implemented it as a **constraint**
- Recall Week 05 lab we constructed a harder bitstring problem by **multiplying** two DVs.

# Linkage between DVs

- Recall the California Manufacturing Co. factory/warehouse IP problem: we can only build a warehouse in LA if we also build a factory there
- That is a **dependency** between two DVs (genes)
- We implemented it as a **constraint**
- Recall Week 05 lab we constructed a harder bitstring problem by **multiplying** two DVs.
- **Linkage:** dependence in the **objective** function between multiple DVs

# Linkage and building-blocks

- We hope the GA will assemble good genetic **building-blocks** – sub-sequences of genes which work well together

# Linkage and building-blocks

- We hope the GA will assemble good genetic **building-blocks** – sub-sequences of genes which work well together
- We hope that crossover will tend to preserve good building-blocks, not destroy them.

# Linkage and building-blocks

- We hope the GA will assemble good genetic **building-blocks** – sub-sequences of genes which work well together
- We hope that crossover will tend to preserve good building-blocks, not destroy them.
- Both one-point and two-point crossover tend to keep **chunks** of the genome together

# Linkage and building-blocks

- We hope the GA will assemble good genetic **building-blocks** – sub-sequences of genes which work well together
- We hope that crossover will tend to preserve good building-blocks, not destroy them.
- Both one-point and two-point crossover tend to keep **chunks** of the genome together
- Uniform crossover does not

# Linkage and building-blocks

- We hope the GA will assemble good genetic **building-blocks** – sub-sequences of genes which work well together
- We hope that crossover will tend to preserve good building-blocks, not destroy them.
- Both one-point and two-point crossover tend to keep **chunks** of the genome together
- Uniform crossover does not
- But are related genes always nearby on the genome?

# Linkage and building-blocks

Are related genes always nearby on the genome? Consider these two definitions of the warehouse problem genome (all variables binary):

[LA\_warehouse, LA\_factory, SF\_warehouse, SF\_factory]

versus

[LA\_warehouse, SF\_warehouse, LA\_factory, SF\_factory]

# Linkage and building-blocks

Are related genes always nearby on the genome? Consider these two definitions of the warehouse problem genome (all variables binary):

[LA\_warehouse, LA\_factory, SF\_warehouse, SF\_factory]

versus

[LA\_warehouse, SF\_warehouse, LA\_factory, SF\_factory]

The first one keeps linked DVs together.

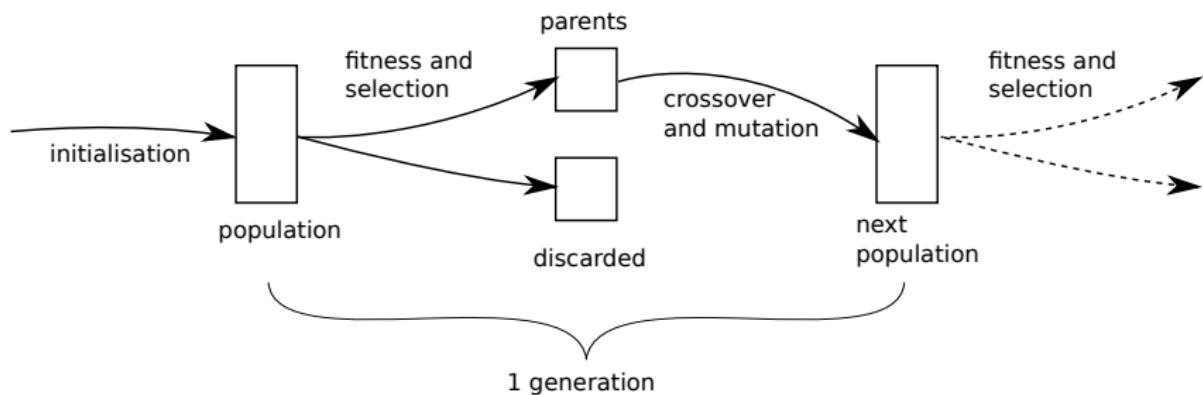
# Linkage

“Crossover methods also assume that there is some degree of linkage between genes on the chromosome: that is, settings for certain genes in groups are strongly correlated to fitness improvement. For example, genes A and B might contribute to fitness only when they’re both set to 1: if either is set to 0, then the fact that the other is set to 1 doesn’t do anything. One- and Two-point Crossover also make the even more tenuous assumption that your vector is structured such that highly linked genes are located near to one another on the vector: because such crossovers are unlikely to break apart closely-located gene groups. Unless you have carefully organized your vector, this assumption is probably a bug, not a feature. Uniform Crossover also makes some linkage assumptions but does not have this linkage-location bias. Is the general linkage assumption true for your problem? Or are your genes essentially independent of one another? For most problems of interest, it’s the former: but it’s dicey. Be careful.” – Luke, **Essentials**.

# Genetic operators' desirable properties

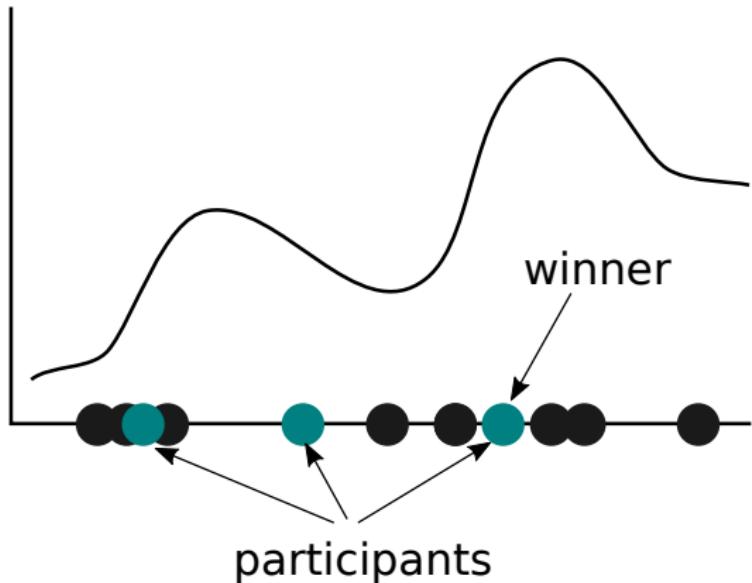
- Initialisation:
  - Uniform on the space
- Crossover:
  - Offspring are intermediate to parents
  - Undirected
  - Allow building blocks if they exist
- Mutation:
  - Mutated genotype is close to original
  - **Ergodic**: capable of reaching any point in the space (through many mutations)
  - Undirected

# GA loop



# Tournament selection

```
def tournament_select(pop, size):  
    return max(random.sample(pop, size), key=f)
```



# Tournament selection

Notice that the winner is not the very best individual in the population! (But maybe it will win another tournament later in the same generation.) We don't want the very best individual to win always – it would be too **exploitative** and would cause the algorithm to **converge** immediately.

# Tournament selection

Notice that the winner is not the very best individual in the population! (But maybe it will win another tournament later in the same generation.) We don't want the very best individual to win always – it would be too **exploitative** and would cause the algorithm to **converge** immediately.

Larger tournament sizes give a **higher selection pressure**, thus make the algorithm **more exploitative**.

# Tournament selection

Notice that the winner is not the very best individual in the population! (But maybe it will win another tournament later in the same generation.) We don't want the very best individual to win always – it would be too **exploitative** and would cause the algorithm to **converge** immediately.

Larger tournament sizes give a **higher selection pressure**, thus make the algorithm **more exploitative**.

What would happen if the tournament size equalled the population size?

# Tournament selection

Tournament selection is nice because it is robust and tunable (exploration versus exploitation). Some other methods are less robust, e.g. the **roulette wheel** can be too exploitative early on and too explorative later in the search.

# Why do GAs work?

“Consider everything. Keep the good. Avoid evil whenever you notice it.” (1 Thess. 5:21-22)

(quoted on <https://www.mat.univie.ac.at/~neum/glopt.html>)

# Why do GAs work?

“Consider everything. Keep the good. Avoid evil whenever you notice it.” (1 Thess. 5:21-22)

(quoted on <https://www.mat.univie.ac.at/~neum/glopt.html>)

Just like with hill-climbing, the **genetic operators** (mutation and crossover) are **undirected**, unaffected by fitness. It is **selection** and the **ratchet** which “drive” evolution.

# Why do GAs work?

“Consider everything. Keep the good. Avoid evil whenever you notice it.” (1 Thess. 5:21-22)

(quoted on <https://www.mat.univie.ac.at/~neum/glopt.html>)

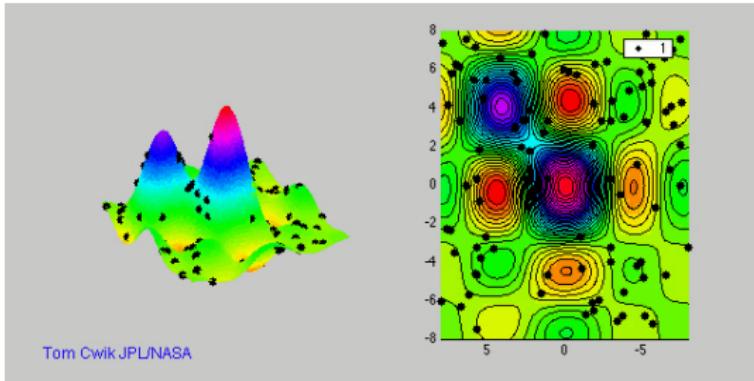
Just like with hill-climbing, the **genetic operators** (mutation and crossover) are **undirected**, unaffected by fitness. It is **selection** and the **ratchet** which “drive” evolution.

Recall the **ratchet** in hill-climbing. Can you see how to rewrite hill-climbing using **selection** in place of the **if**-statement?

# Elitism

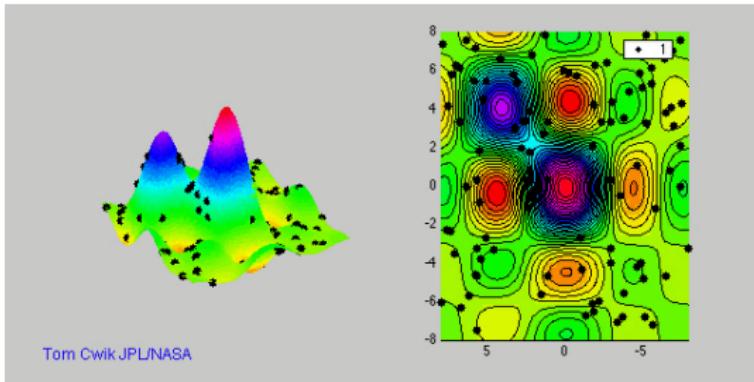
- In hill-climbing, we allow disimproving moves to help escape local optima
- In the GA, we don't have to discard our best individual to escape
- No real need to allow a disimprovement in the **best** objective value from one generation to the next
- **Elitism** means: we copy the best individual in the population directly to the next generation
- (sometimes more than one).

# Convergence



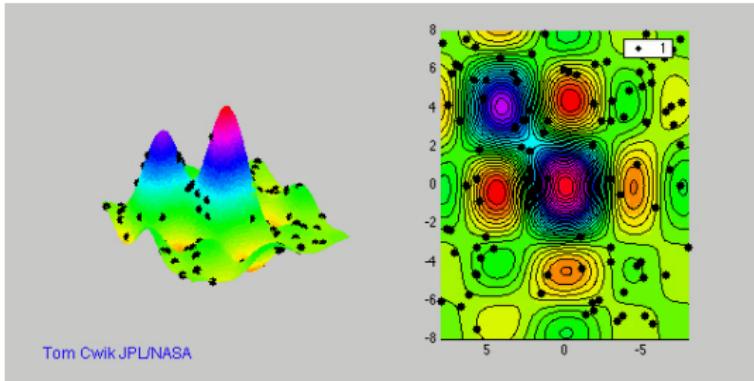
- We can measure genetic diversity in the population

# Convergence



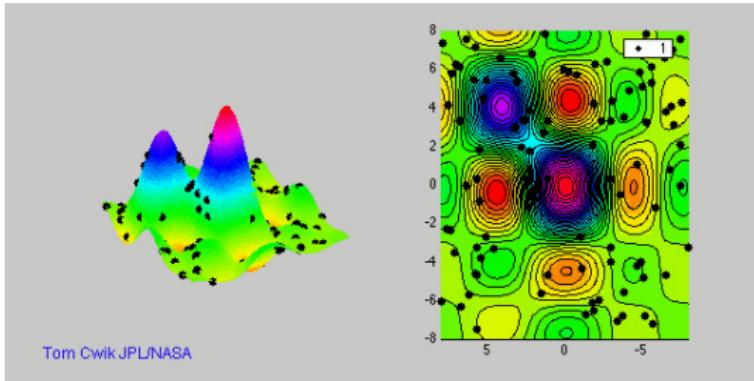
- We can measure genetic diversity in the population
- Or measure variance of objective values in the population

# Convergence



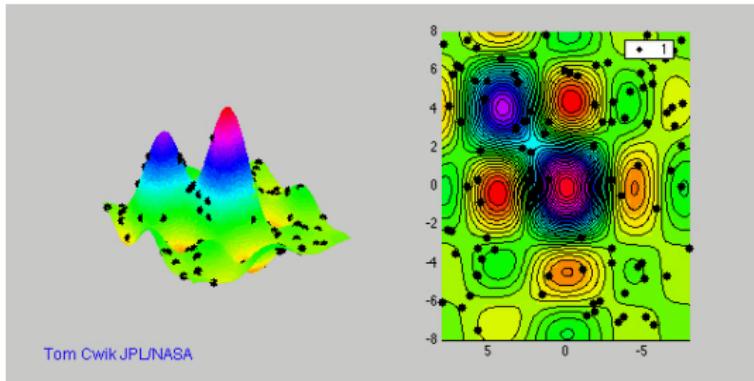
- We can measure genetic diversity in the population
- Or measure variance of objective values in the population
- Diversity **decreases over time**

# Convergence



- We can measure genetic diversity in the population
- Or measure variance of objective values in the population
- Diversity **decreases over time**
- When the population has **converged**, there will be no more improvement...

# Convergence



- We can measure genetic diversity in the population
- Or measure variance of objective values in the population
- Diversity **decreases over time**
- When the population has **converged**, there will be no more improvement...
- ... because offspring will be almost identical to parents.

# GA assumptions

GAs make the same sorts of assumptions as smart hill-climbing:

- the objective may be a bit rugged, but not totally uninformative or deceptive;

# GA assumptions

GAs make the same sorts of assumptions as smart hill-climbing:

- the objective may be a bit rugged, but not totally uninformative or deceptive;
- we don't know the gradient;

# GA assumptions

GAs make the same sorts of assumptions as smart hill-climbing:

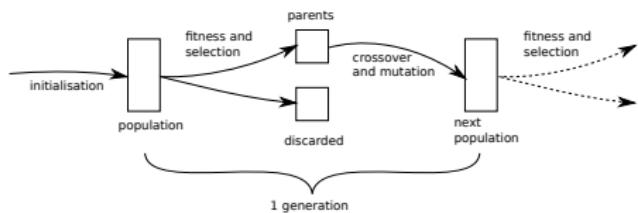
- the objective may be a bit rugged, but not totally uninformative or deceptive;
- we don't know the gradient;
- neighbours (created by mutation) usually have similar fitness;

# GA assumptions

GAs make the same sorts of assumptions as smart hill-climbing:

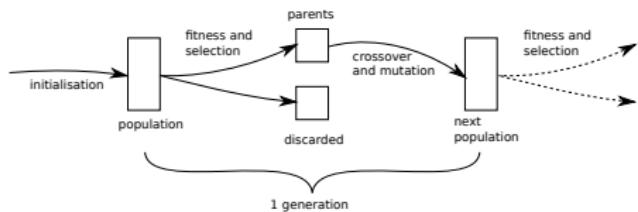
- the objective may be a bit rugged, but not totally uninformative or deceptive;
- we don't know the gradient;
- neighbours (created by mutation) usually have similar fitness;
- children (created by crossover) usually have fitness similar to parents.

# GA: alternative loops



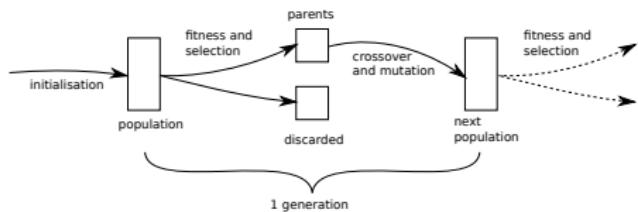
- 1 E.g. apply crossover to produce some individuals, and mutation to produce others

# GA: alternative loops



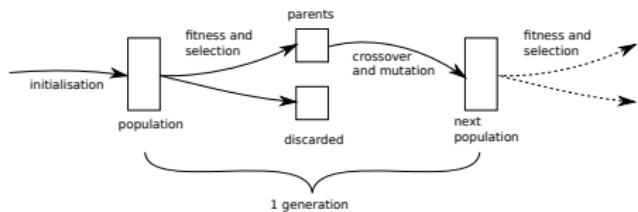
- 1 E.g. apply crossover to produce some individuals, and mutation to produce others
- 2 E.g. use entire population to produce offspring, then run fitness/selection to **discard from** (population + offspring)

# GA: alternative loops



- 1 E.g. apply crossover to produce some individuals, and mutation to produce others
- 2 E.g. use entire population to produce offspring, then run fitness/selection to **discard from** (population + offspring)
- 3 E.g. “steady-state GA” where we just produce 1 or 2 offspring at a time from any parents, and use fitness/selection to decide which individuals to **discard**. No generations.

# GA: alternative loops



- 1 E.g. apply crossover to produce some individuals, and mutation to produce others
- 2 E.g. use entire population to produce offspring, then run fitness/selection to **discard from** (population + offspring)
- 3 E.g. “steady-state GA” where we just produce 1 or 2 offspring at a time from any parents, and use fitness/selection to decide which individuals to **discard**. No generations.
- 4 E.g. “memetic algorithm”, a GA with an inner loop doing local search (e.g. mutation only).

# Memetic algorithm

In *The Selfish Gene* (1978), Dawkins proposed that ideas evolve in a way similar to genes, and nicknamed them **memes**.

A **memetic algorithm** is a GA with an inner loop doing local search (e.g. mutation only). This is **not a good name** for the algorithm.

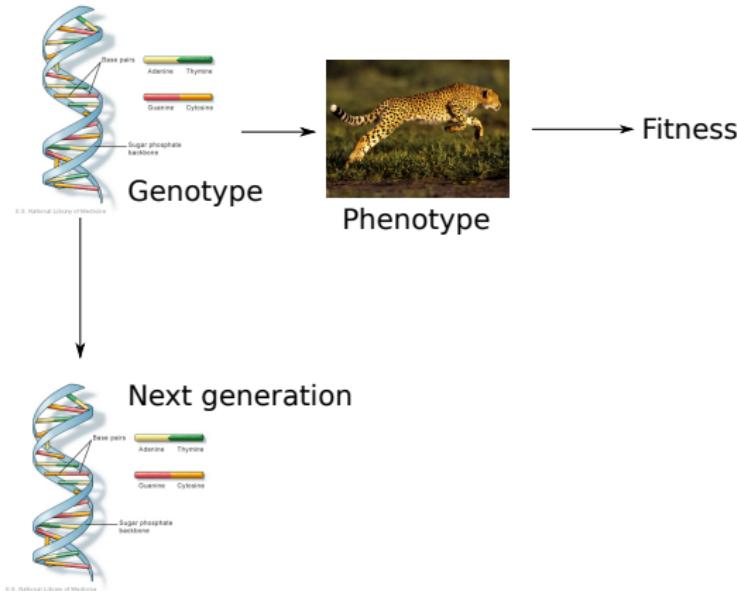
# HIPSTER DAWKINS

LIKED MEMES  
BEFORE THEY WERE COOL

# Overview

- 1 Introduction to Genetic Algorithms
- 2 Operators
- 3 **Genotypes and phenotypes**
- 4 Experimental methodology

# Genotypes and phenotypes



- Mapping from genes (**genotype**) to organism (**phenotype**)
- Only the organism is “evaluated” by nature for its “fitness”; only the genes are passed on to offspring.

# Motivation

- Q. **Why** does nature do this?

# Motivation

- Q. **Why** does nature do this?
- A. It's impossible to mutate a cheetah, or to crossover two cheetahs. Mutation and crossover work on the underlying DNA, not on the animal.

# Example

A car is defined by:

- Shape (8 floats, 1 per vertex)
- Wheel size (2 floats, 1 per wheel)
- Wheel position (2 ints, 1 per wheel)
- Wheel density (2 floats, 1 per wheel) darker means denser
- Chassis density (1 float) darker means denser

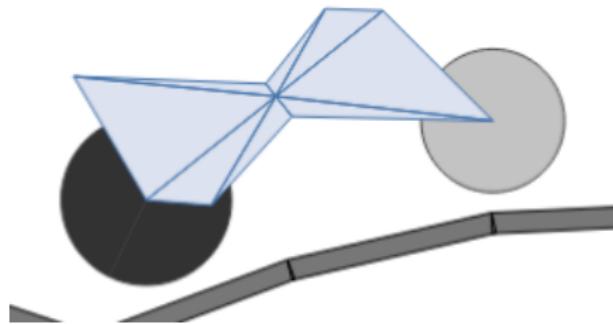


Image from [rednuht.org](http://rednuht.org)

**Phenotype:** the car.

# Example

A car is defined by:

- Shape (8 floats, 1 per vertex)
- Wheel size (2 floats, 1 per wheel)
- Wheel position (2 ints, 1 per wheel)
- Wheel density (2 floats, 1 per wheel) darker means denser
- Chassis density (1 float) darker means denser

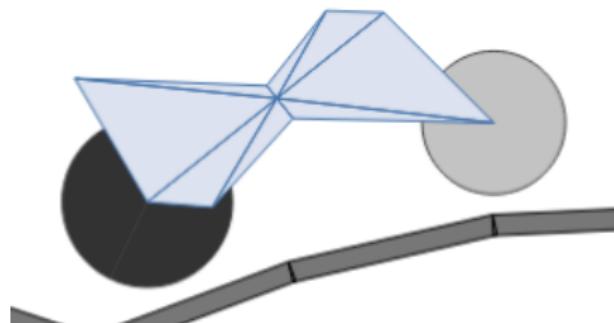


Image from [rednuht.org](http://rednuht.org)

**Phenotype:** the car.

**Genotype** (15 genes): e.g. [.3 .4 .6 .1 .3 .4 .6 .1 | .6 .7 | 3 6 | .3 .8 | .4]

(Full details of genotype-phenotype mapping [here](#).)

**Objective:** distance travelled in the simulation.

# Search space and solution space

The same idea is often used outside Evolutionary Algorithms also, but with different terminology: the **search space** (genotype space) and the **solution space** (phenotype space).

What we actually want is a **solution**, but we run our search in the **search** space, and whenever we look at a search point we map it to a solution.

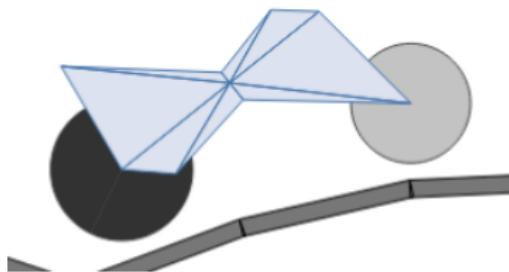
# Terminology

Other optimisation	Evolutionary
Decision variable	Gene
Point	Genotype
Search space	Genotype space
Solution space	Phenotype space

# Motivation

- Q. **Why** do we do this?
- A. Just like cheetahs, it is not possible to mutate or crossover 2D cars. But it is easy to define these operations on the underlying data structure.

# Car genotypes and phenotypes



[rednuht.org](http://rednuht.org)

The car (phenotype) has to “live” in a simulated world. If it succeeds there, its **genes** are passed on, i.e. are varied for use in the next iteration.

# Operators on car genotypes

How could we define these operators on car genotypes?

- Initialisation
- Mutation
- Crossover

Recall that a car genotype is defined as a list of 15 numbers:

- Shape (8 floats, 1 per vertex)
- Wheel size (2 floats, 1 per wheel)
- Wheel position (2 ints, 1 per wheel)
- Wheel density (2 floats, 1 per wheel) darker wheels mean denser wheels
- Chassis density (1 float) darker body means denser chassis

# Simulation

The 2D car is also a good example of **simulation**. We don't have a formula or program that directly calculates fitness. The genes give rise to a car (phenotype), and it has to "live" in a simulated world. We measure its success in that world.

# Explicit and implicit fitness

Explicit fitness (objective):

- As we saw in LP/IP
- A **formula** or function that gives a **number**

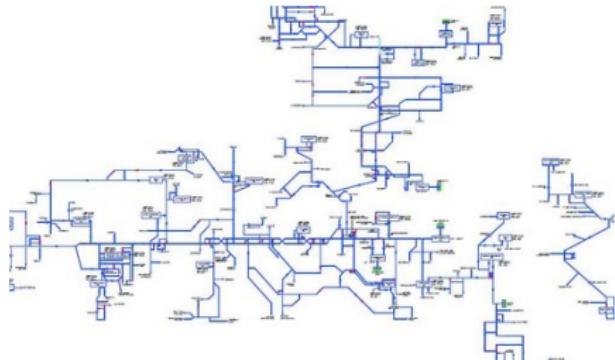
Simulation fitness:

- As in 2D Boxcar
- The solution is put in a simulation, and we somehow measure performance as a **number**

Implicit fitness:

- As in biological evolution
- There is **no number!**
- Research fields: Coevolution and Artificial Life

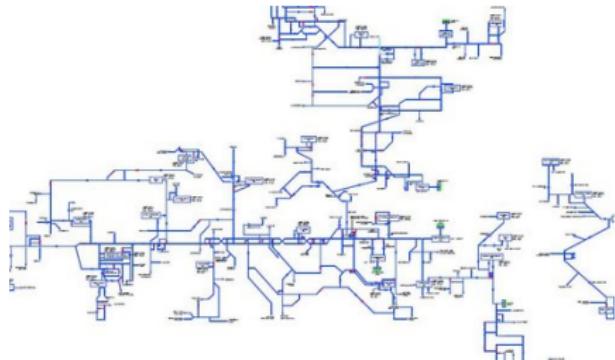
# Genotypes and phenotypes: another example



Dwr Uisce

- We want to design a water network (a graph)

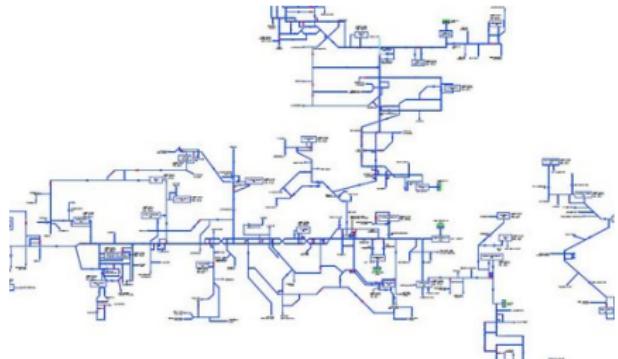
# Genotypes and phenotypes: another example



Dwr Uisce

- We want to design a water network (a graph)
- It's hard to define initialisation, mutation and crossover

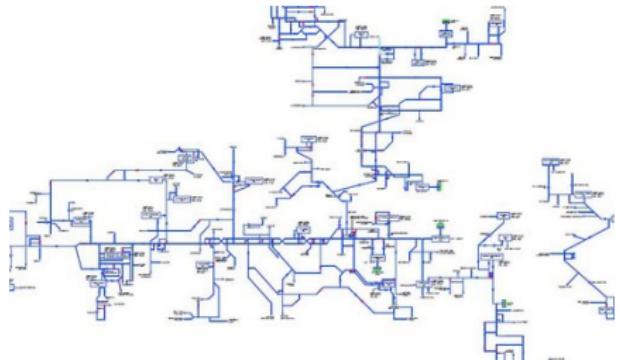
# Genotypes and phenotypes: another example



Dwr Uisce

- We want to design a water network (a graph)
- It's hard to define initialisation, mutation and crossover
- Define a mapping from bitstrings (genotypes) to graphs (phenotypes)

# Genotypes and phenotypes: another example



Dwr Uisce

- We want to design a water network (a graph)
- It's hard to define initialisation, mutation and crossover
- Define a mapping from bitstrings (genotypes) to graphs (phenotypes)
- Use standard initialisation, mutation and crossover on bitstrings.

# Developmental mappings

Above, the mapping from genotype to phenotype was straightforward. Each gene controlled one part of the phenotype directly.

In a **developmental** mapping, it is less straightforward. Multiple genes may influence each part of the phenotype, and one gene may influence multiple parts. That's how it is with real DNA!

We won't explore this further.

Recommended reading: Bentley, **Exploring Component-based Representations - The Secret of Creativity by Evolution?**, in Blackboard.

# Overview

- 1 Introduction to Genetic Algorithms**
- 2 Operators**
- 3 Genotypes and phenotypes**
- 4 Experimental methodology**

# Scientific experiments

When we're trying to solve an optimisation problem, we are free to try anything that works.

# Scientific experiments

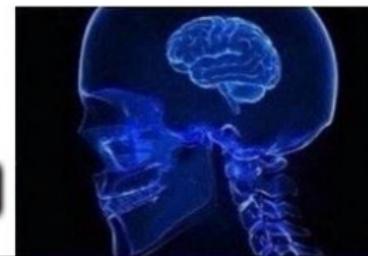
When we're trying to solve an optimisation problem, we are free to try anything that works.

If we want to make **scientific** claims:

- We have a variety of possible algorithms
  - We should compare them
- Our metaheuristics are stochastic (not the same result every time)
  - We should run many times
- Every algorithm has hyper-parameters
  - We should tune them fairly
- Larger fitness evaluation budget implies better performance
  - Use a fixed budget for fairness.

**GET A GOOD  
RESULT FROM  
OPTIMISATION**

---



**OUT-PERFORM  
RANDOM SEARCH**

---



**OUT-PERFORM  
THE SOTA**

---



**OUT-PERFORM THE  
SOTA AFTER FAIR  
HYPER-PARAMETER TUNING  
AND STATISTICAL TESTS**

---



# Experimental design

The simplest experimental design is a **factorial design**. That means we try all combinations of all factors (hyperparameters) in a **Cartesian product**.

Algo	Parameter	Parameter	Obj min	Obj mean	Obj sd
GA	popsize=10	tsize=2	?	?	?
GA	popsize=20	tsize=2	?	?	?
GA	popsize=10	tsize=4	?	?	?
GA	popsize=20	tsize=4	?	?	?

(Here we have just two hyperparameters, each with two values we want to test. Remember that `maxits = popszie * ngens`, so the value of `popszie` implies the value of `ngens`.)

# Experimental design

With multiple algorithms they may have different hyperparameters but we can put the results in the same table. In this example we have 3 algorithms but 8 configurations.

Algo	Parameter	Parameter	Obj min	Obj mean	Obj sd
RS	(none)	(none)	?	?	?
LAHC	L=1	(none)	?	?	?
LAHC	L=10	(none)	?	?	?
LAHC	L=40	(none)	?	?	?
GA	popsize=10	tsize=2	?	?	?
GA	popsize=20	tsize=2	?	?	?
GA	popsize=10	tsize=4	?	?	?
GA	popsize=20	tsize=4	?	?	?

# Procedure

- 1 We identify one or more baselines, including a random search and a **state of the art** method from recent literature.
- 2 We choose a fixed fitness evaluation budget.
- 3 We identify the key hyperparameters for each algorithm, and a range of likely values.
- 4 We do a factorial design: for each algorithm, for each combination of hyperparameter values, we run the algorithm 30 times and put results in a table.
- 5 We see which algorithms and configurations have the best **mean objective value**.
- 6 We carry out an appropriate statistical test (e.g. *t*-test, ANOVA, Kruskal-Wallis, etc.) and we report the *p*-value and whether it is statistically significant at the  $\alpha = 0.05$  level.

# Random seed

Our algorithms are **randomised**. Running twice with the same hyperparameters may give different results.

It is good practice to set a **random seed** for each run using `random.seed()`. This way our runs are **reproducible**.

```
for seed in range(5):  
    random.seed(seed)  
    run_my_algorithm()
```

# Readings

- Luke, **Essentials**, Section 11.1 (Experimental Methodology): recommended for Assignment 2.

# Lecture 07 - Real-valued optimisation

## Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Overview

- 1 Real-valued optimisation: big picture and applications
- 2 Covariance Matrix Adaptation
- 3 Particle Swarm Optimisation

# Real-valued optimisation

In real-valued optimisation we have a vector of real DVs. We already saw some black-box algorithms for this:

- In HC, LAHC, SA, GA, we can use `init`, `nbr` (and `crossover` for GA) suitable for real vectors.

# LP and gradient descent

If a stronger algorithm is available, we won't use a black-box algorithm:

- Some applications with real DVs have linear objective and constraints, so we would use LP;
- In some applications with real DVs, we can find the gradient, so we would use gradient descent.

# Black-box real-valued optimisation

But if we have to, we can use a metaheuristic for real optimisation.

# Black-box real-valued optimisation

But if we have to, we can use a metaheuristic for real optimisation.

Now, a confession:

**LAHC, SA and GA are not the best metaheuristics for this**

- Covariance Matrix Adaptation
- Particle Swarm Optimisation

# Black-box real-valued optimisation

But if we have to, we can use a metaheuristic for real optimisation.

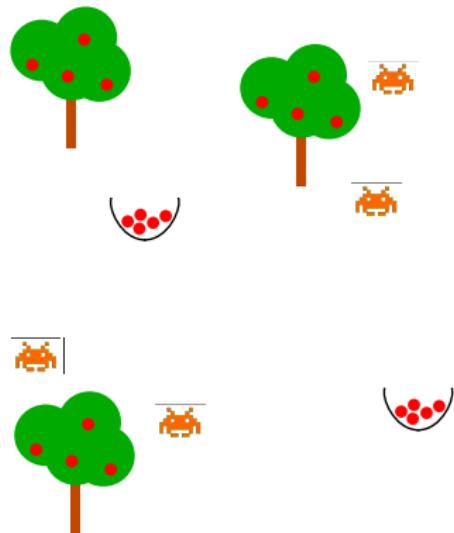
Now, a confession:

**LAHC, SA and GA are not the best metaheuristics for this**

- Covariance Matrix Adaptation
- Particle Swarm Optimisation

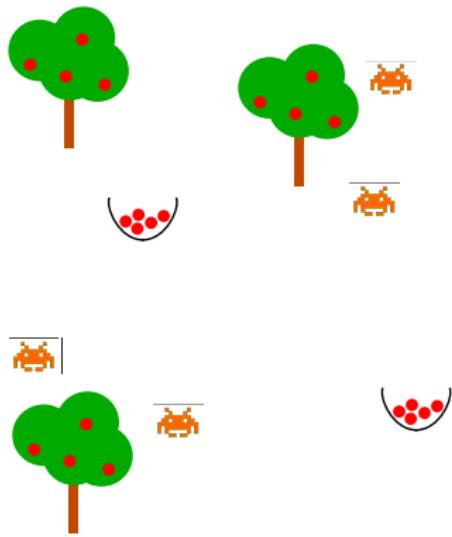
First let's consider applications.

# Facility location



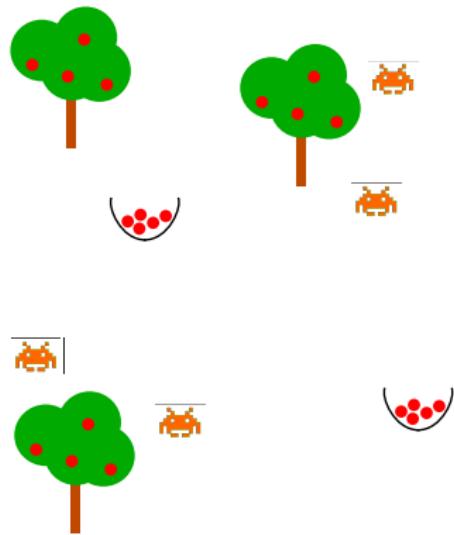
Large field with  $n$  fruit trees at locations  $t_i$ . Tree  $i$  has  $w_i$  fruit. We have a swarm of tiny flying robot fruit pickers. Each robot picks a fruit and flies to a bin. We can place  $m$  bins  $b_j$  each with unlimited capacity. Where in the field should we put them?

# Facility location



Large field with  $n$  fruit trees at locations  $t_i$ . Tree  $i$  has  $w_i$  fruit. We have a swarm of tiny flying robot fruit pickers. Each robot picks a fruit and flies to a bin. We can place  $m$  bins  $b_j$  each with unlimited capacity. Where in the field should we put them?  
Minimise  $f(x_1, x_2, \dots, x_{2m}) = \sum_i w_i \min_j(d(t_i, b_j))$   
where  $d$  is Euclidean distance.

# Facility location



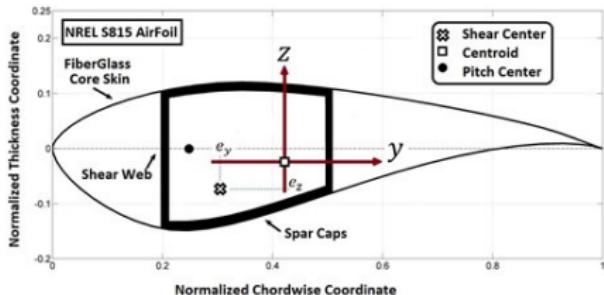
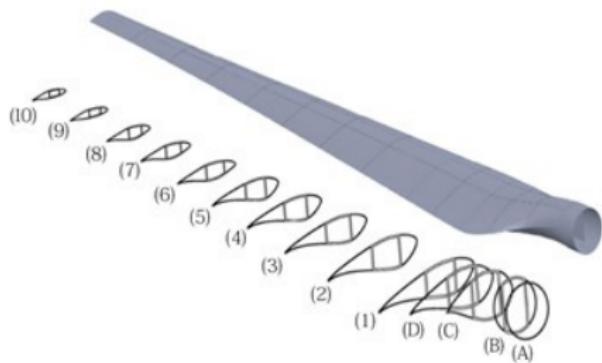
Large field with  $n$  fruit trees at locations  $t_i$ . Tree  $i$  has  $w_i$  fruit. We have a swarm of tiny flying robot fruit pickers. Each robot picks a fruit and flies to a bin. We can place  $m$  bins  $b_j$  each with unlimited capacity. Where in the field should we put them?  
Minimise  $f(x_1, x_2, \dots, x_{2m}) = \sum_i w_i \min_j(d(t_i, b_j))$   
where  $d$  is Euclidean distance.

Exactly the same problem arises when choosing where to locate a supermarket to serve several cities.

# Wind turbine engineering

In a wind turbine, what **shape** should the sails be?

Several real-valued parameters:

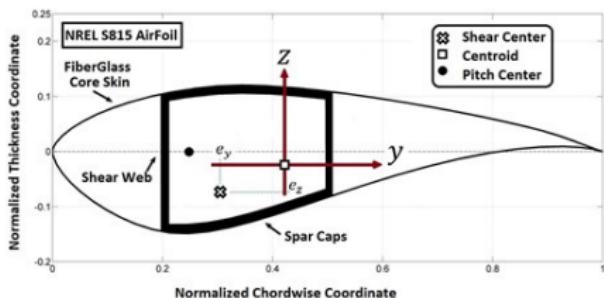
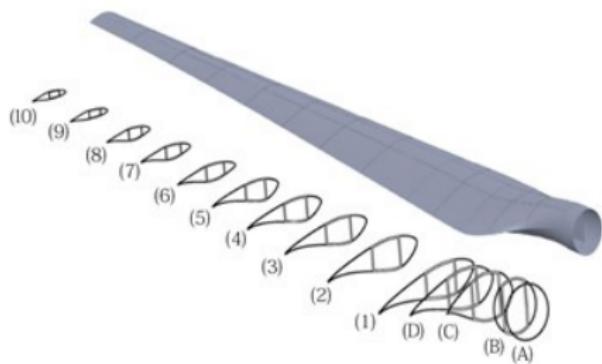


Sheibani and Akbari

# Wind turbine engineering

In a wind turbine, what **shape** should the sails be?

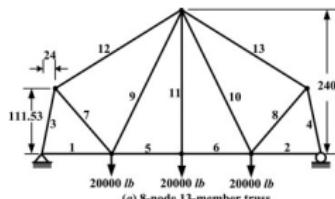
Several real-valued parameters:



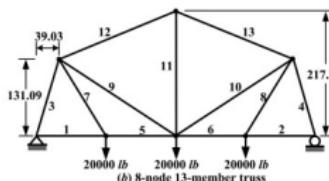
Sheibani and Akbari

We program a finite-element simulation to estimate the **annual energy production** for a given shape.

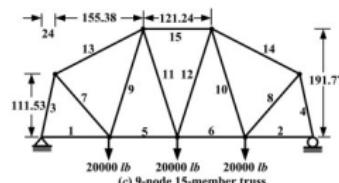
# A-frame truss design



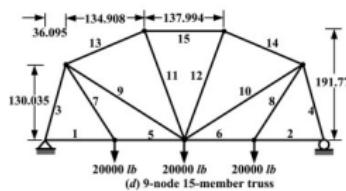
(a) 8-node 13-member truss



(b) 8-node 13-member truss



(c) 9-node 15-member truss



(d) 9-node 15-member truss

- **Truss:** a 2D or 3D engineering design of beams or rods, attached at nodes – like in the attic of a house.
- Calculate behaviour under gravity, taking account of beam width/weight etc.
- Numerical parameters: node positions and beam widths.
- Objective: minimise weight, minimise deflection, prevent collapse.
- Simple example:  
<https://pythonhosted.org/pyswarm/>
- An interactive setting:  
<https://www.polybridge2.com/>

# FM Synthesis



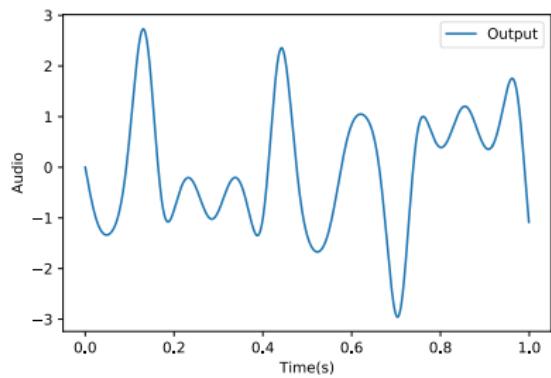
- Frequency-modulation synthesis is an approach to sound synthesis for music
- Invented by Chowning and popularised by Yamaha
- Common in 1980s synth music, e.g. Blade Runner
- Core idea: to take the sine of a sine, giving a sound spectrum with many frequencies.

# FM Synthesis control problem

The output of an FM Synth at time  $t$  is:

$$\hat{y}(t) = a_1 \sin(\omega_1 t\theta) + a_2 \sin(\omega_2 t\theta + a_3 \sin(\omega_3 t\theta))$$

where  $\theta = 2\pi/f_s$ , and  $f_s$  is sampling frequency, e.g. 44.1kHz.

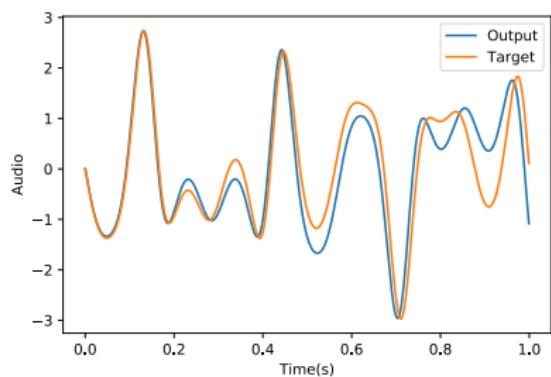


# FM Synthesis control problem

We are given some target sound  $y$  which we wish to match using FM Synthesis. We **minimise**:

$$f(a_1, \omega_1, a_2, \omega_2, a_3, \omega_3) = \text{RMSE}(y, \hat{y}).$$

(The parameters are constrained, e.g. to  $[-6.4, 6.35]$ )

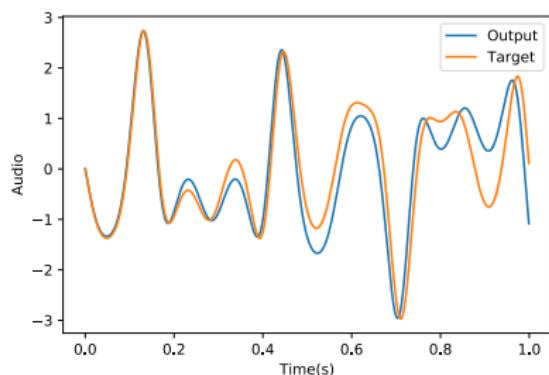


# FM Synthesis control problem

We are given some target sound  $y$  which we wish to match using FM Synthesis. We **minimise**:

$$f(a_1, \omega_1, a_2, \omega_2, a_3, \omega_3) = \text{RMSE}(y, \hat{y}).$$

(The parameters are constrained, e.g. to  $[-6.4, 6.35]$ )



(Formulation from Das and Suganthan, 2011, in Blackboard.)

# FM Synthesis control problem

(My PhD was in this area, but with a different type of synthesizer. E.g., I experimented with other objectives that work better than RMSE.)

# Some more applications

Again see Das and Suganthan (2011) in Blackboard:

- Radar pulse code optimisation
- Antenna array
- Economic dispatch (like unit commitment)
- Spacecraft trajectory optimisation

# Overview

- 1 Real-valued optimisation: big picture and applications
- 2 **Covariance Matrix Adaptation**
- 3 Particle Swarm Optimisation

# How does a GA work? Another view

- In a GA, the population implicitly represents our knowledge of  $f$
- Think of the population as samples from a distribution
- We draw new samples using crossover and mutation
- These new samples are concentrated in high-fitness regions thanks to selection.

# Genetic operators

- Mutation takes in **one** individual and returns a new one

# Genetic operators

- Mutation takes in **one** individual and returns a new one
- Crossover takes in **two** individuals and returns one or two new ones

# Genetic operators

- Mutation takes in **one** individual and returns a new one
- Crossover takes in **two** individuals and returns one or two new ones
- Could we have 3-parent operators, or higher?

# Genetic operators

- Mutation takes in **one** individual and returns a new one
- Crossover takes in **two** individuals and returns one or two new ones
- Could we have 3-parent operators, or higher?
- These do exist, but are rarely used

# Genetic operators

- Mutation takes in **one** individual and returns a new one
- Crossover takes in **two** individuals and returns one or two new ones
- Could we have 3-parent operators, or higher?
- These do exist, but are rarely used
- But taking a blend of **all parents at once** is quite common: it is called **Estimation of Distribution**.

# Estimation of Distribution Algorithms

The **Estimation of Distribution Algorithm** (EDA):

- 1 Create an initial population
- 2 Discard the worst according to  $f$
- 3 **Estimate** a statistical model of the remainder
- 4 **Draw samples** from the model to create new population
- 5 Go to 2.

# Estimation of Distribution Algorithms

The **Estimation of Distribution Algorithm** (EDA):

- 1 Create an initial population
- 2 Discard the worst according to  $f$
- 3 **Estimate** a statistical model of the remainder
- 4 **Draw samples** from the model to create new population
- 5 Go to 2.

Think of this as a **crossover** which blends **all parents at once**.

# Simple real EDA

- 1 Create population using `init`
- 2 Evaluate and discard worst to give “parents”  $P$
- 3 Create one vector representing the **mean**

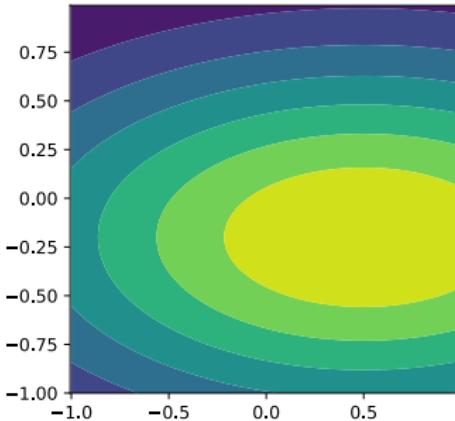
$$\mu \in \mathbb{R}^n : \mu_i = \frac{1}{|P|} \sum_{x \in P} x_i$$

and one representing the **standard deviation**

$$\sigma \in \mathbb{R}^n : \sigma_i = \sqrt{\frac{1}{|P|} \sum_{x \in P} (x_i - \mu_i)^2}$$

- 4 Draw new individuals  $x$  by sampling:  $x_i \sim N(\mu_i, \sigma_i)$
- 5 Go to 2.

# What will happen?



$$\mu = (0.5, -0.25), \sigma = (2.0, 0.5)$$

- $\mu_i$  will gradually move towards the best value for DV  $x_i$
- But this is **independent** of the values at other  $i$
- $\sigma_i$  will gradually decrease (the distribution narrows).

# Reminder: linkage

Linkage between DVs makes problems harder.

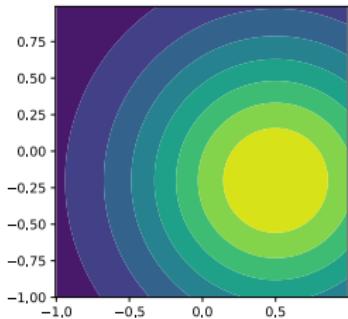
Other words for linkage:

- dependency
- epistasis
- synergy
- “more than the sum of its parts”.

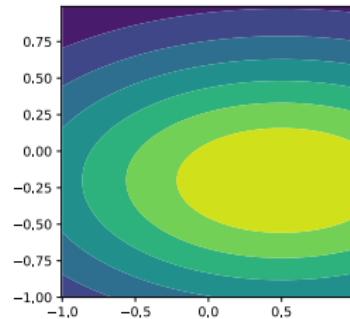
# Problem: our EDA is too simple

- All positions  $x_i$  are considered **independently**
- Cannot learn about linkage between two variables
- But the **covariance** between two variables would give that information
- (Covariance is just **correlation** in disguise)
- **Solution:** use a more complex distribution with covariance

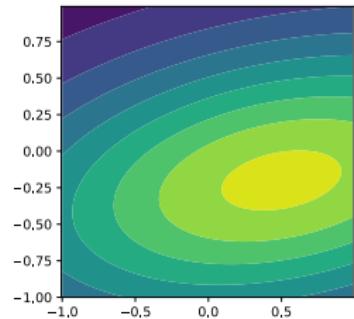
# Multivariate Normal: $N(\mu, \Sigma)$



Spherical



Diagonal



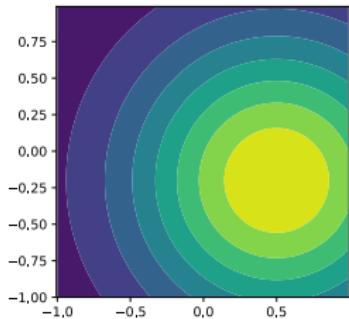
Full

$$\Sigma = \begin{bmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{bmatrix}$$

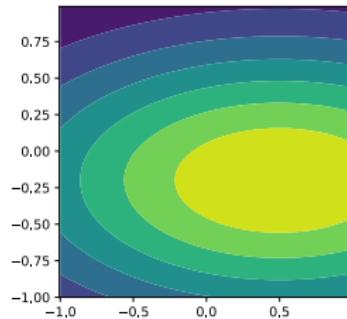
$$\Sigma = \begin{bmatrix} 2.0 & 0.0 \\ 0.0 & 0.5 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 2.0 & 0.3 \\ 0.3 & 0.5 \end{bmatrix}$$

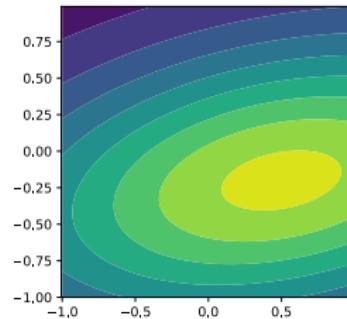
# Multivariate Normal: $N(\mu, \Sigma)$



Spherical



Diagonal



Full

$$\Sigma = \begin{bmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 2.0 & 0.0 \\ 0.0 & 0.5 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 2.0 & 0.3 \\ 0.3 & 0.5 \end{bmatrix}$$

Don't get confused: **diagonal** means  $\Sigma$  is diagonal. The distribution is then **axis-aligned**. A **full** matrix is more flexible (still must be symmetric): distribution can be diagonal.

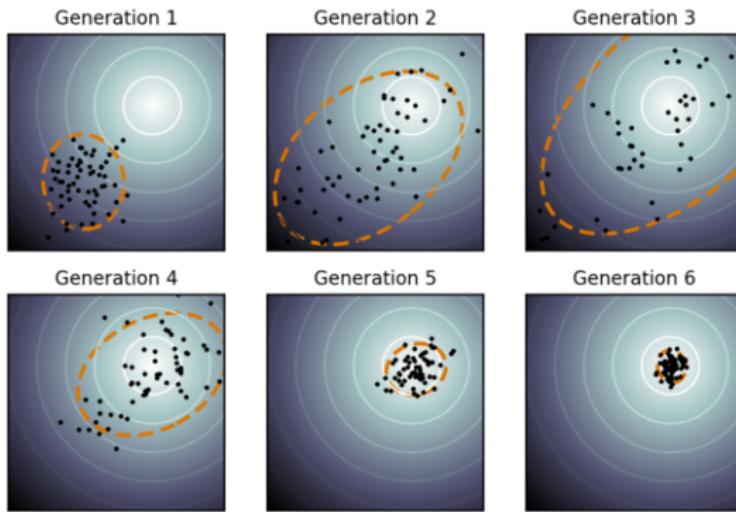
# Covariance matrix

The covariance matrix  $\Sigma$  gives the **shape** of the distribution as shown.  
In our simple EDA we had a **diagonal**  $\Sigma$ : no linkage.

# Covariance Matrix Adaptation

- In CMA, we model the population as a **multivariate normal**
- For each decision variable we have a **mean**
- We also have a **full covariance matrix**.
- (Think of the distribution as an ellipse, non-axis aligned)

# CMA: illustration of a run



Wiki

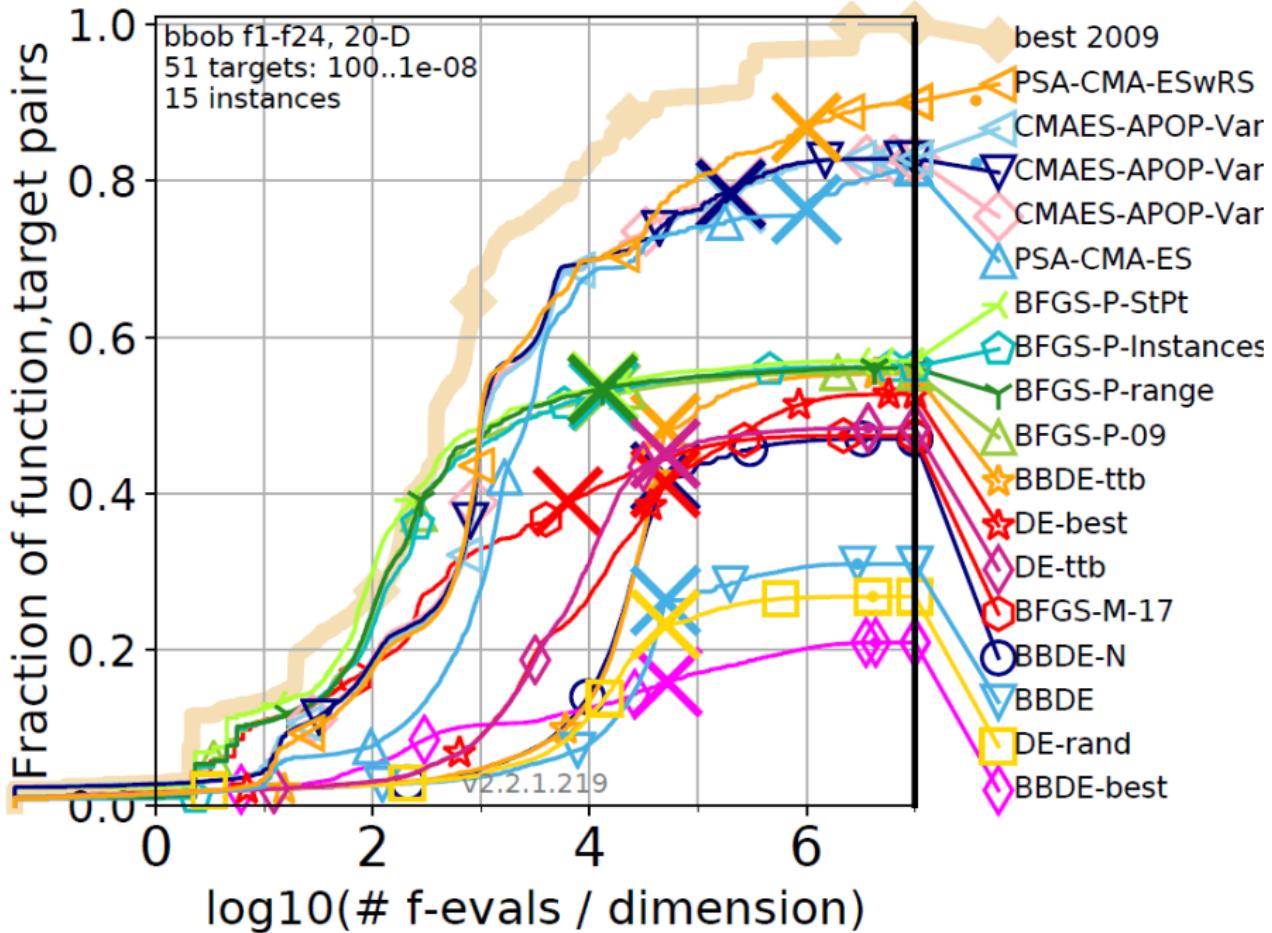
In this picture, fitness is shown as grey-scale. The current population is modelled using a multivariate normal with full covariance. This distribution changes location **and shape** to model the population.

- Uses **standard** statistical methods to draw samples from the multivariate normal, giving the new population
- Also use some **sophisticated** methods to update the distribution based on new population at each step
  - (i.e. don't just re-learn it from scratch)
- It also **self-adapts** many hyperparameters
- Further details are omitted and will not be examinable.

# CMA: performance?

CMA and variants often win the “Black-box optimisation benchmarking” competition (BBOB)

See <https://coco.gforge.inria.fr/>.



# Using CMA in practice

```
$ pip install cma  
  
import cma  
...  
# pass in initial mean and sigma - just a guess  
es = cma.CMAEvolutionStrategy(mu0, sigma0,  
                                {'bounds': (lb, ub)  
                                 'seed': seed})  
es.optimize(f, iterations=maxits / es.popsize)  
print(es.result)
```

# Using CMA in practice

```
$ pip install cma  
  
import cma  
...  
# pass in initial mean and sigma - just a guess  
es = cma.CMAEvolutionStrategy(mu0, sigma0,  
                                {'bounds': (lb, ub)  
                                 'seed': seed})  
es.optimize(f, iterations=maxits / es.popsize)  
print(es.result)
```

It chooses good hyperparameter values by itself :)

# Overview

- 1 Real-valued optimisation: big picture and applications
- 2 Covariance Matrix Adaptation
- 3 **Particle Swarm Optimisation**

# Bio-inspired optimisation

- Evolutionary algorithms: inspired by Darwinian evolution
- **Particle swarm optimisation:** inspired by bird flocking behaviour
- Ant colony optimisation: inspired by ant foraging

# Flocking

Murmuration: <https://www.youtube.com/watch?v=eakKfY5aHmY>

# Flocking seagulls



# Boids and Crowds

Reynolds' Boids website: <http://www.red3d.com/cwr/boids/>

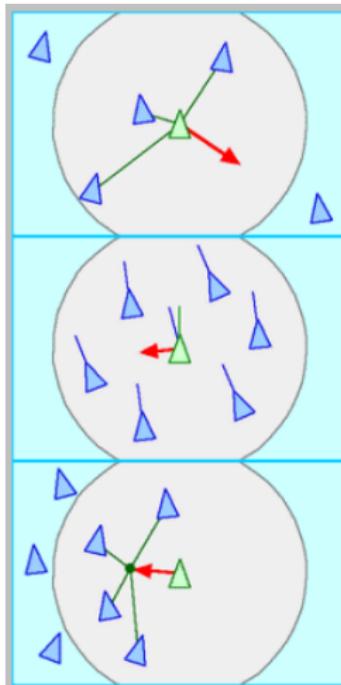
Reynolds' paper (SIGGRAPH 1987)

Boids video: <https://www.youtube.com/watch?v=QbUPfMXXQIY>

Use in movies: <https://www.youtube.com/watch?v=042YjQ9aZNk>

# Boids rules

A **boid** is a “bird” (a point moving in 2D or 3D space) obeying three rules



**Separation:** steer to avoid crowding local flockmates

**Alignment:** steer towards the average heading of local flockmates

**Cohesion:** steer to move toward the average position of local flockmates

Reynolds

# Boids properties

- **Emergence:** “complex global behavior can arise from the interaction of simple local rules” (Reynolds)
- Some **ordered** behaviour
- Also some **chaotic** behaviour: small changes in initial conditions lead to large changes later
- A system **at the edge of chaos** (Langton).

# Particle Swarm Optimisation

- PSO is “a population based stochastic optimization technique”
- “inspired by social behavior of bird flocking or fish schooling.”
- (from <http://www.swarmintelligence.org/tutorials.php>)
- Proposed by Eberhart and Kennedy (1995).

# PSO and GA

Similarities:

- Population of randomly-initialised points
- Many iterations
- Each generation formed by variation of the previous

Differences:

- Think of each point as **moving**, not being replaced by a new one
- Points have **velocity**.

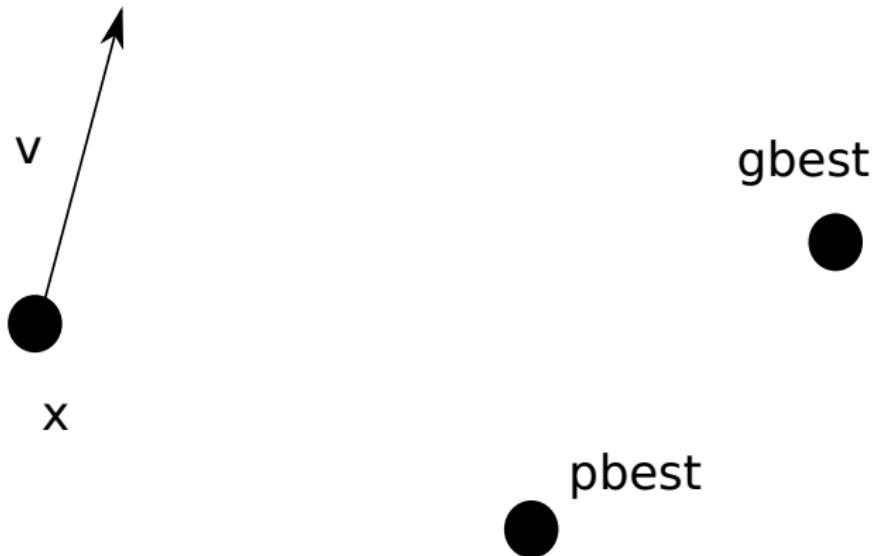
# PSO algorithm

- We store the best point ever observed, called gbest  
All particles move towards gbest. This is “social influence”.  
Think of a seagull observing **another** seagull finding food.
- Each particle stores the best point it has visited, called pbest  
Each particle moves towards pbest. This is “personal influence”.  
Think of a seagull **remembering** where it found food.

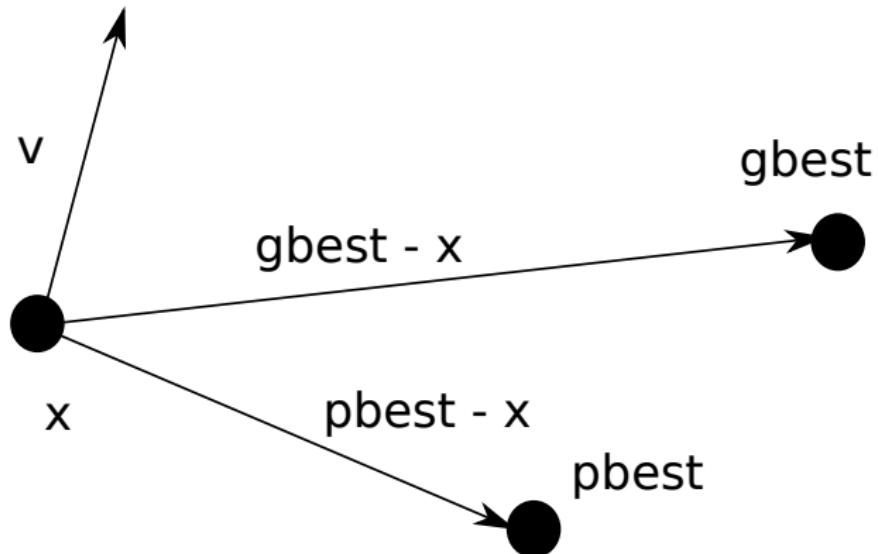
# PSO algorithm

- We store the best point ever observed, called **gbest**  
All particles move towards **gbest**. This is “social influence”.  
Think of a seagull observing **another** seagull finding food.
- Each particle stores the best point it has visited, called **pbest**  
Each particle moves towards **pbest**. This is “personal influence”.  
Think of a seagull **remembering** where it found food.
- At each “generation”, we update each particle’s velocity  $v$  and then position  $x$ .
  - 1  $v := wv + c_p \text{rand}() (pbest - x) + c_g \text{rand}() (gbest - x)$
  - 2  $X := X + v$

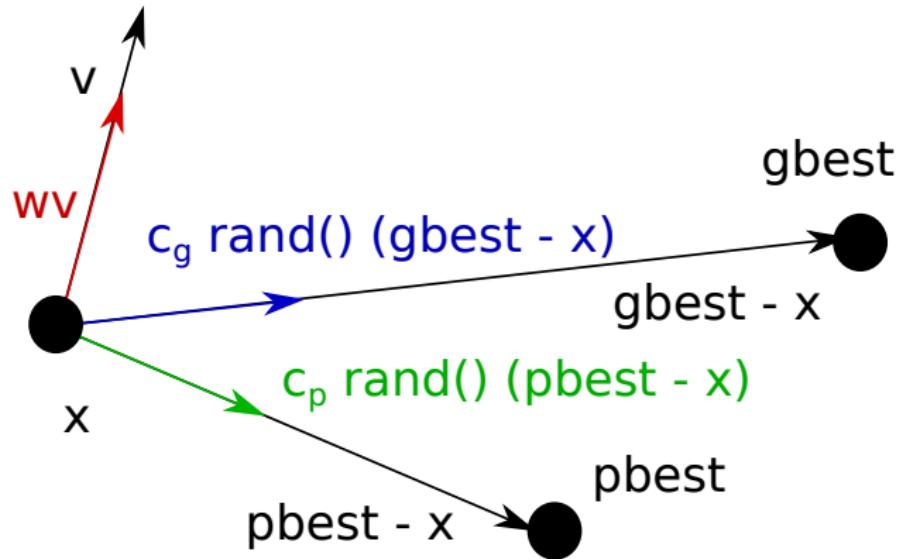
# Adding vectors



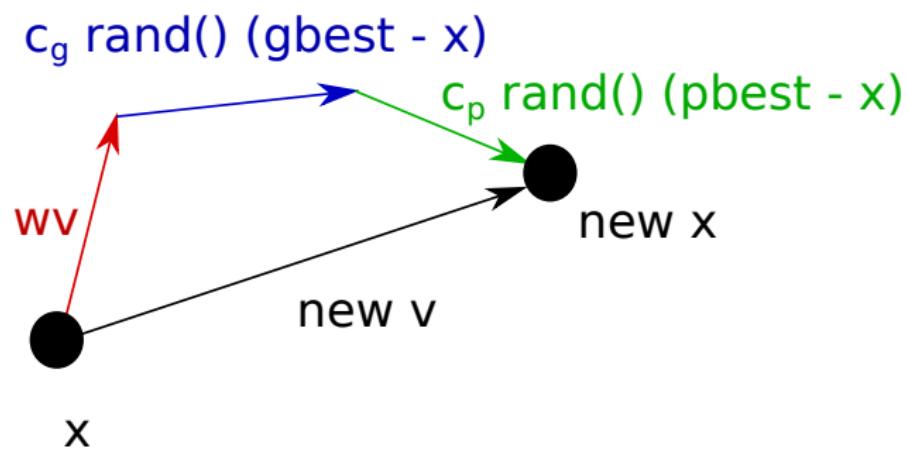
# Adding vectors



# Adding vectors



# Adding vectors



# Control parameters

$c_p$  and  $c_g$  control the strength of the two rules. Often  $c_p = c_g = 2$ .

# Inertia

- $w$  controls **inertia** (i.e. momentum)
- Large  $w$ : velocity more influential: exploration
- Small  $w$ : gbest and pbest more influential: exploitation
- Sometimes  $w$  is on an **schedule**, e.g. gradually reduce  $w$  from 0.9 to 0.4 over 1000 generations.

# Inertia

- $w$  controls **inertia** (i.e. momentum)
- Large  $w$ : velocity more influential: exploration
- Small  $w$ :  $gbest$  and  $pbest$  more influential: exploitation
- Sometimes  $w$  is on an **schedule**, e.g. gradually reduce  $w$  from 0.9 to 0.4 over 1000 generations.

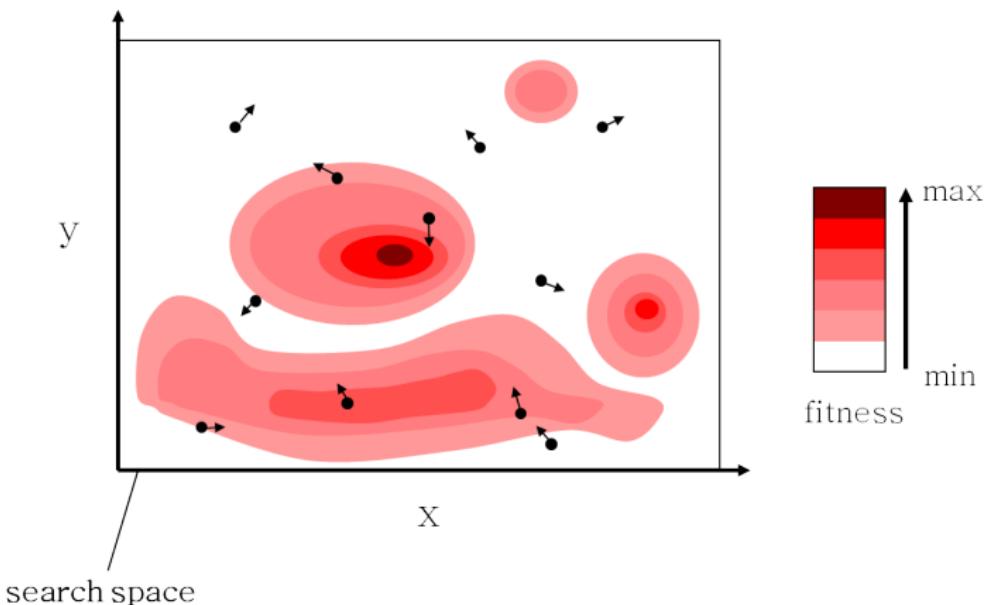
**Gradually move from exploration to exploitation over time:  
just like SA**

# PSO animation

Wiki:

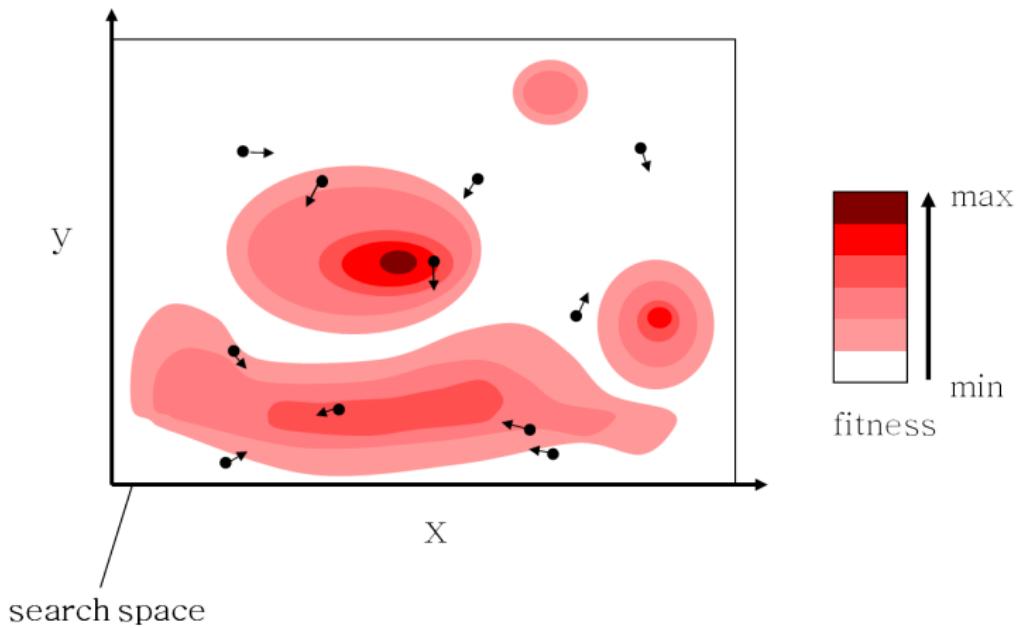
<https://commons.wikimedia.org/wiki/File:ParticleSwarmArrowsAnimation.gif>

# PSO AT WORK (MAX OPTIMIZATION PROBLEM)

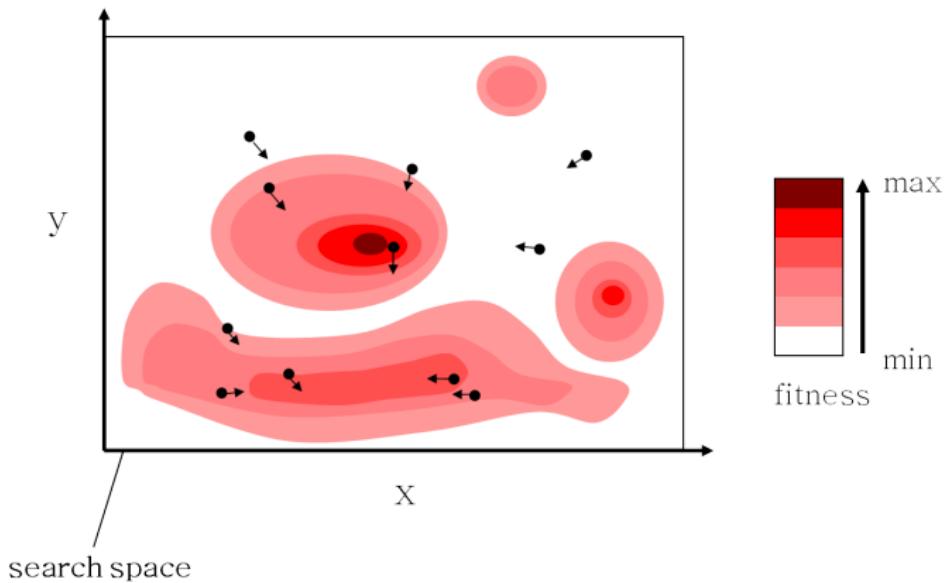


Example slides from Pinto et al.

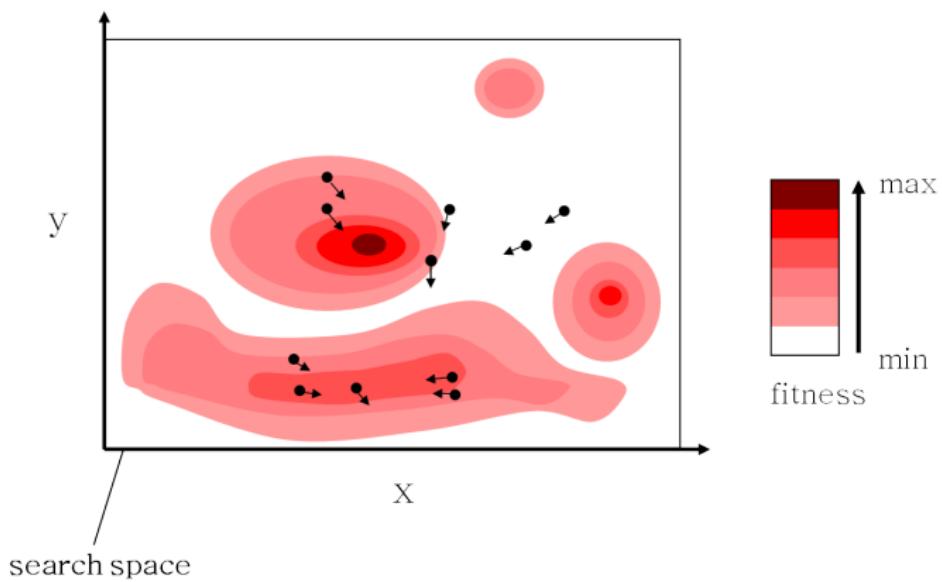
# PSO AT WORK



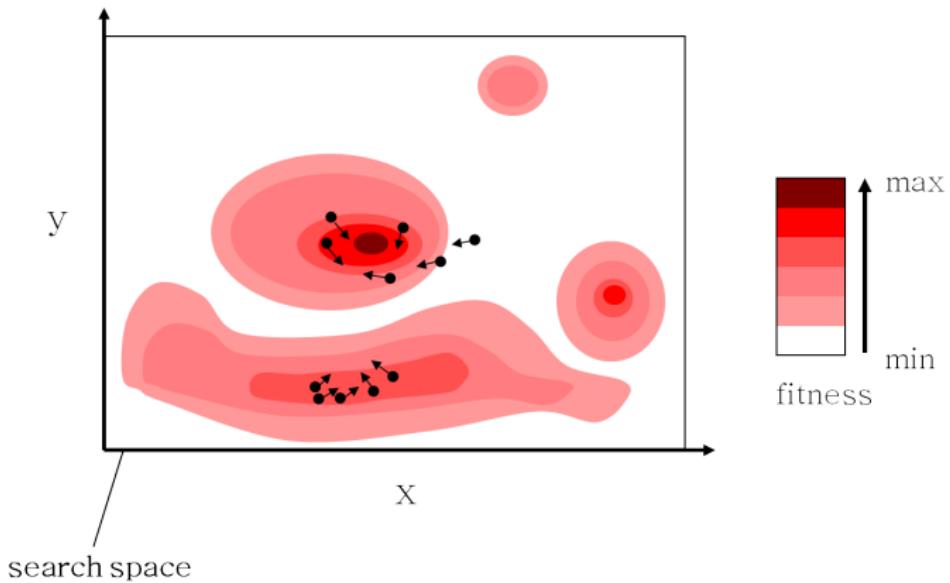
# PSO AT WORK



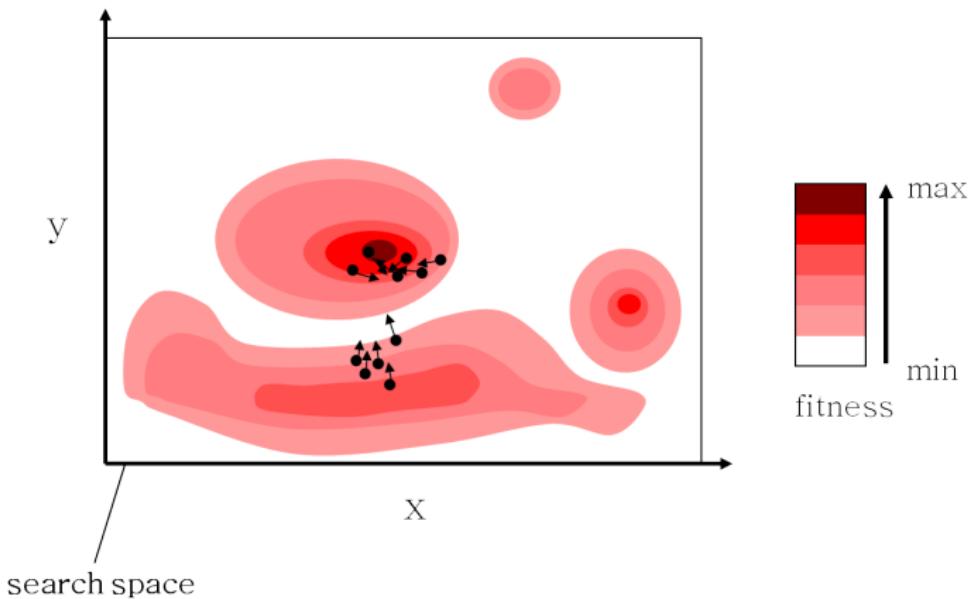
# PSO AT WORK



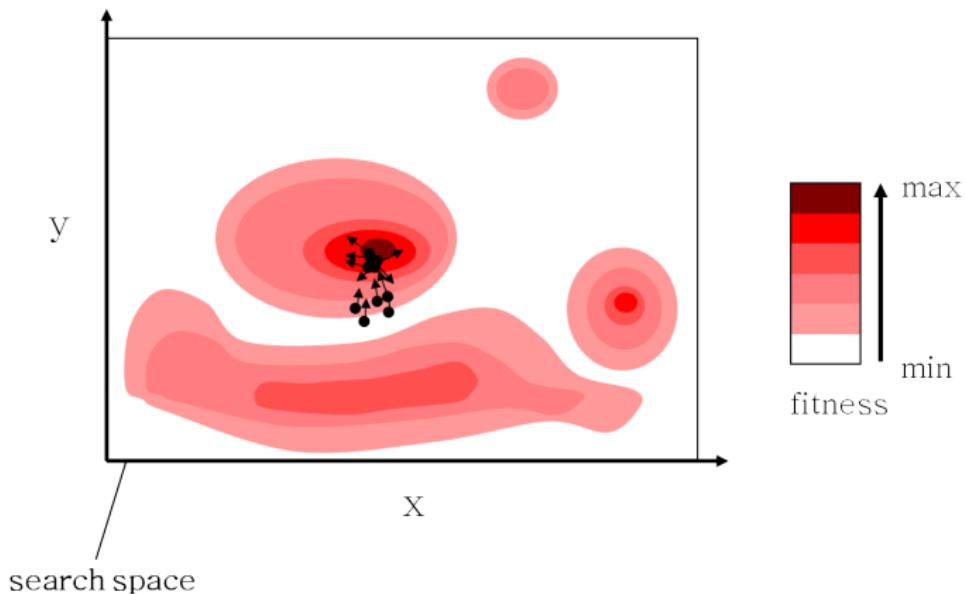
# PSO AT WORK



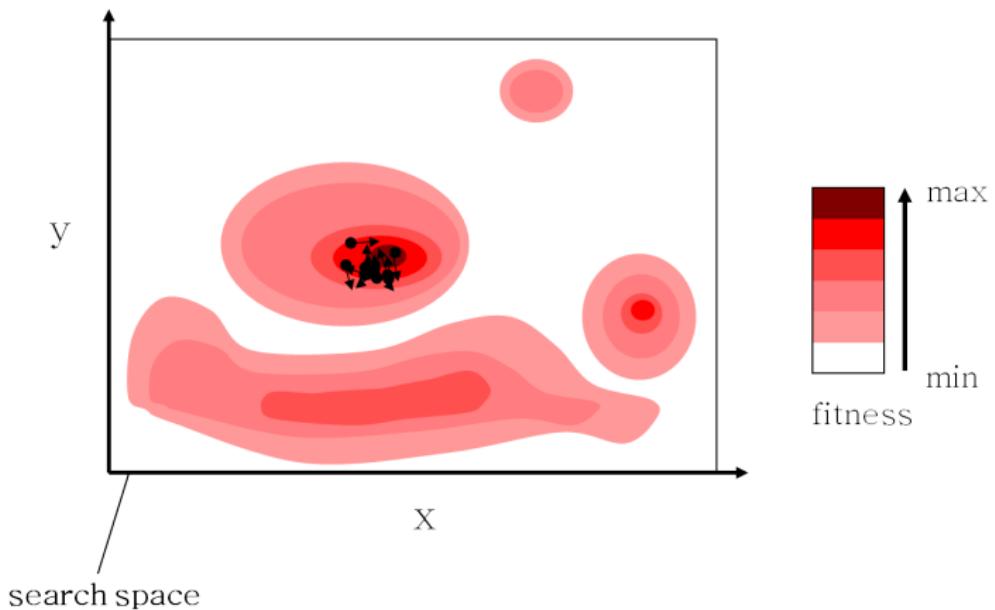
# PSO AT WORK



# PSO AT WORK



# PSO AT WORK



(These images are from Di Caro, credited there to Pinto, I can't find Pinto!)

# Why does it work?

Velocity tends to help because a particle sometimes just “flies past” local optima.

gbest means there is **information-sharing** among particles.

# Using a PSO in practice

We can program PSO without too much difficulty. But there are some nice libraries, e.g.:

- PySwarms <https://github.com/ljvmiranda921/pyswarms>
- DEAP [https://deap.readthedocs.io/en/master/examples/pso\\_basic.html](https://deap.readthedocs.io/en/master/examples/pso_basic.html)

# Real-valued optimisation: summary

- Real-valued **GA**
- **CMA** is not really bio-inspired
  - Multivariate normal with full covariance matrix
  - Smart hyperparameters
  - Wins competitions.
- **PSO** is another large branch of bio-inspired computing
  - Inspired by flocking behaviour, not evolution
  - Steer towards personal best and global best
  - But often grouped together with EC.

# Overview

- 1 Real-valued optimisation: big picture and applications
- 2 Covariance Matrix Adaptation
- 3 Particle Swarm Optimisation

# Lecture 08 – Combinatorial optimisation

## Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Combinatorial Optimisation

## Combinatorial optimisation

just means optimisation in discrete spaces, as opposed to real-valued spaces, especially:

- Binary and integer-valued decision variables
- Sets
- Trees
- Graphs
- Permutations

(We've already studied binary and integer-valued decision variables.)

# Possible approaches

- We don't need new algorithms
- Hill-climbing variants and GAs will work
- We may need new initialisation, mutation, crossover operators to deal with the new data structures
- Real-valued algorithms like PSO and CMA will not work
- (unless we define a clever genotype-phenotype mapping).

# Overview

- 1 Encodings
- 2 Sets and task allocation
- 3 Permutations and travelling salesman problems
- 4 Trees, genetic programming hyperheuristics, and keyword bidding

# Encodings

The **encoding** is the data structure used to represent solutions.

Usually it is obvious, e.g. a bitstring, or a real vector.

# Encodings

The **encoding** is the data structure used to represent solutions.

Usually it is obvious, e.g. a bitstring, or a real vector.

Sometimes a few different encodings could be suitable:

- A real vector could be encoded as itself, or with normalisation,  
...  
■ An integer vector could be encoded as itself, or a real vector  
(with rounding-off), ...  
■ A graph could be encoded as an edge list, an adjacency matrix,  
...

# Designing encodings and operators

When we design an encoding we have to design operators to go with it.

The operators should be designed to suit the search space, e.g.:

- Binary decision variable: bit-flip mutation and two-point crossover
- Real decision variable: Gaussian mutation and two-point crossover

There are many possibilities: see Luke, Chapter 4 (Representations).

# Encodings

We try to use encodings such that:

- Operators are **easy** to design
- e.g. their outputs respect all constraints (we'll see an example)
- The landscape is **smooth**: (e.g., see Luke, p. 61)
  - Outputs of `nbr` should be similar to inputs
  - Offspring genuinely inherit from parents.

# Overview

- 1 Encodings
- 2 **Sets and task allocation**
- 3 Permutations and travelling salesman problems
- 4 Trees, genetic programming hyperheuristics, and keyword bidding

# A task allocation problem

- Suppose we have a large set of tasks to be carried out
- We want to allocate them to workers
- Workers are interchangeable
- Each task has an ID and a time requirement
- Tasks can be allocated to any workers and in any order.

# Applications



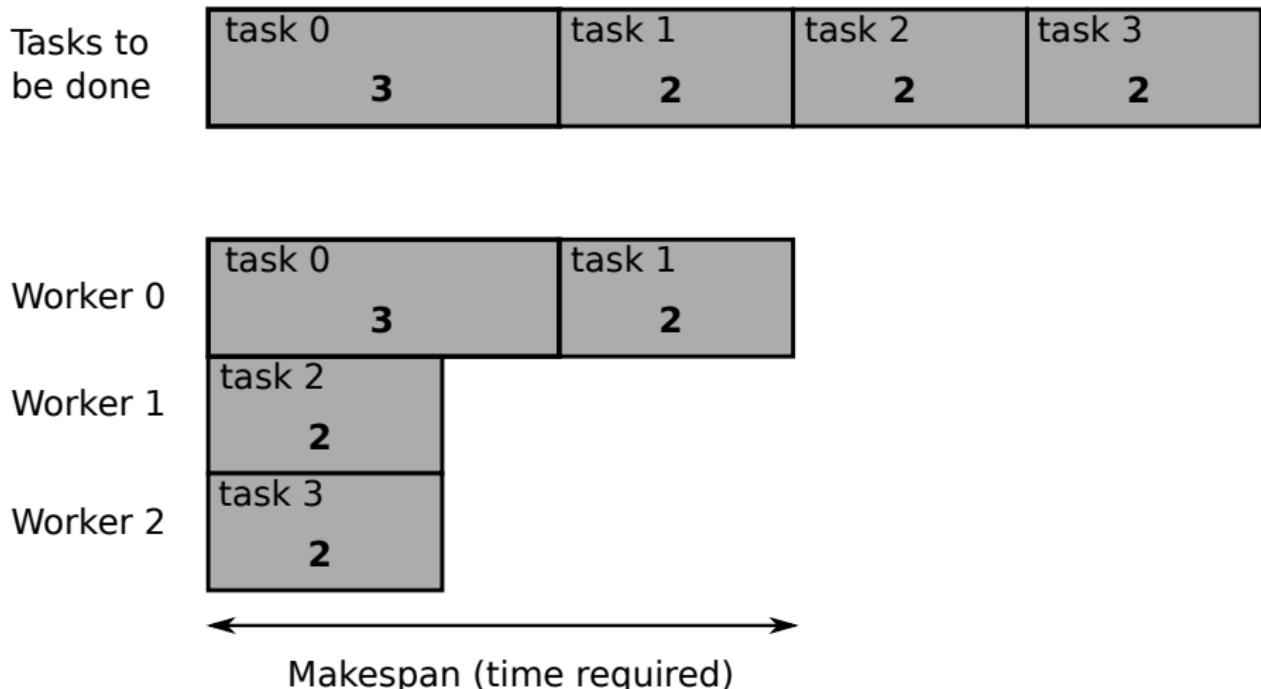
From [roboticsandautomationnews.com](http://roboticsandautomationnews.com)

This scenario arises in many applications, not just with human workers, but also e.g. in parallel computing, warehouse robot task allocation, and more.

# Clarifications

- When a worker finishes a task, they move on to their next task
- Once begun, a task cannot be transferred
- The total amount of worker time required is constant
- Workers work in parallel.

# Task allocation example



Task allocation: 4 tasks and 3 workers.

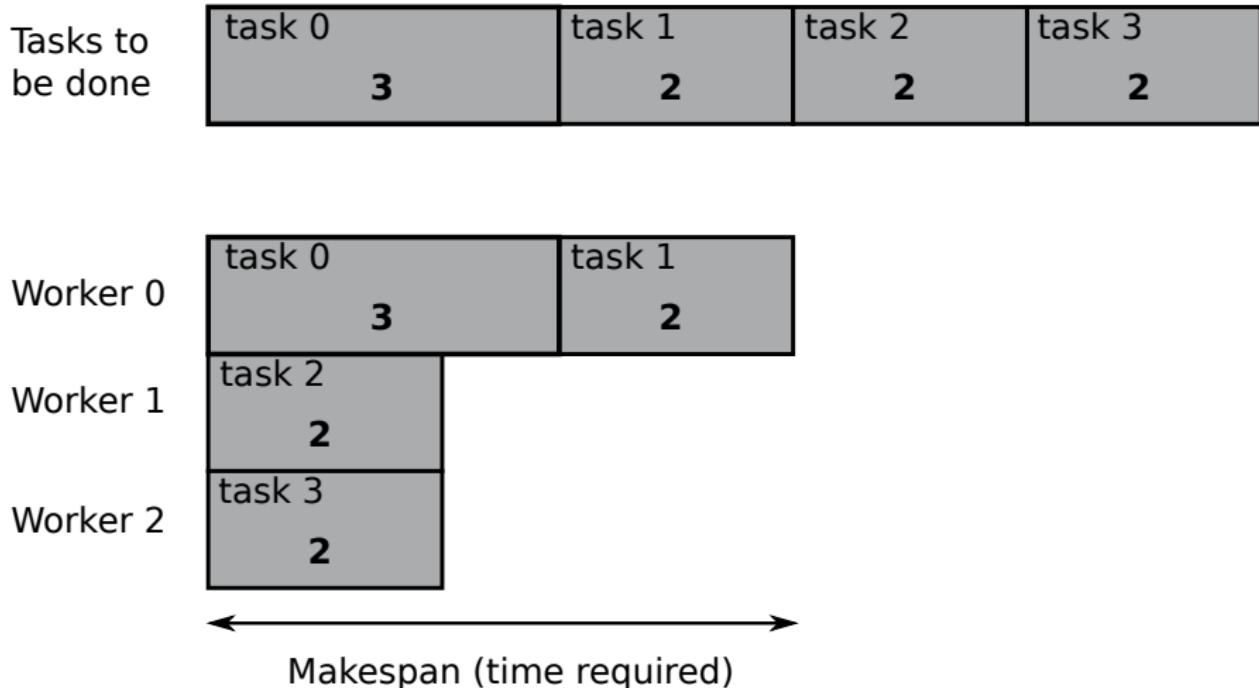
# Objective

The **makespan** means the finish time, i.e. the time at which the final task is completed. It is the maximum of the time required by any worker to complete their allocation.

We would like to allocate the tasks to workers so that the makespan is as early as possible.

So, our objective is to minimise the makespan.

# Makespan example



Task allocation: 4 tasks and 3 workers. In the proposed allocation the makespan is 5, but a better allocation is possible.

# Objective function

We minimise the makespan:

$$\text{Minimise: } \max_w \sum s_w$$

where  $s_w$  is the set of task time requirements allocated to worker  $w$ .

# Alternative objective function

Notice that a good task allocation (minimal makespan) will result in all workers finishing **at about the same time**.

Therefore an alternative objective function could be defined as the maximum minus the minimum of all workers' finish times:

$$\text{Minimise: } \max_w \sum s_w - \min_w \sum s_w$$

This will be non-negative. It will be large when the tasks are not evenly divided, and will approach zero when the tasks are almost evenly divided.

An advantage of this definition is the scale is easier to understand: good allocations will be close to zero. If we reach zero, we'll know it is the global optimum.

# Alternative objective functions

There is a possibility that changing the objective function will change the landscape in a way that is either good or bad for the algorithm, e.g. making the landscape smoother or more rugged.

An interesting exercise is to try both objectives and compare their results. Of course, we must be careful to compare them fairly – after running with either objective, we evaluate the best solution using the “true” objective, that is the makespan itself.

The makespan is the “true” objective, not the  $(\max - \min)$  variant, because in the end we don’t really care about the min.

# Another alternative objective function

$$\text{Minimise: } (\max_w \sum s_w - \min_w \sum s_w) / \max(d)$$

The motivation here is again to give us a better sense of scale: it will now be more comparable across problems.

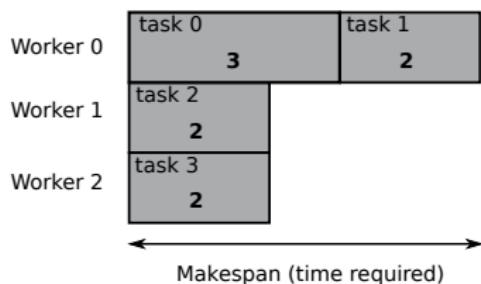
This objective is a linear scaling of the previous, so will give the same landscape.

# Encoding

We are trying to represent a **partition**, i.e. a set of sub-sets. We could represent the sub-sets literally.

Tasks to be done	task 0 3	task 1 2	task 2 2	task 3 2
------------------	-------------	-------------	-------------	-------------

We create three sets:  
 $\{\{0, 1\}, \{2\}, \{3\}\}$ .



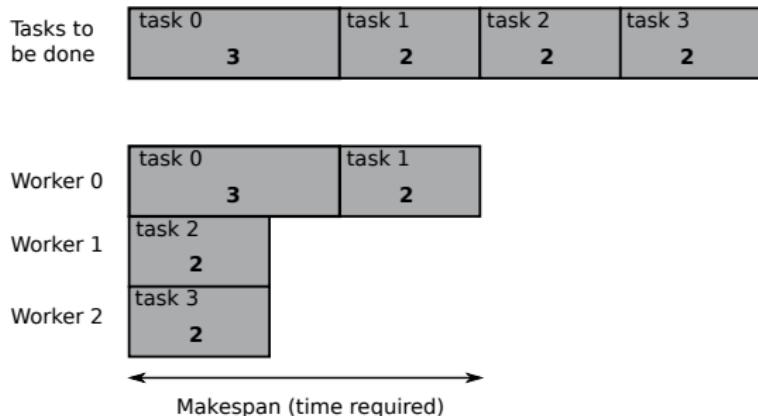
The `init` function is easy to write. What about `nbr` and `crossover`?

# Designing operators

- Our nbr function should **move** an item from one set to another.
- It should not **remove** an item from one set without adding to another.
- Crossover is hard to define in a way that offspring really inherit from their parents.

# An alternative encoding

When studying IP we saw some similar problems. We might consider a double-subscripted binary variable  $x_{ij} = 1$  if and only if worker  $i$  carries out task  $j$ .

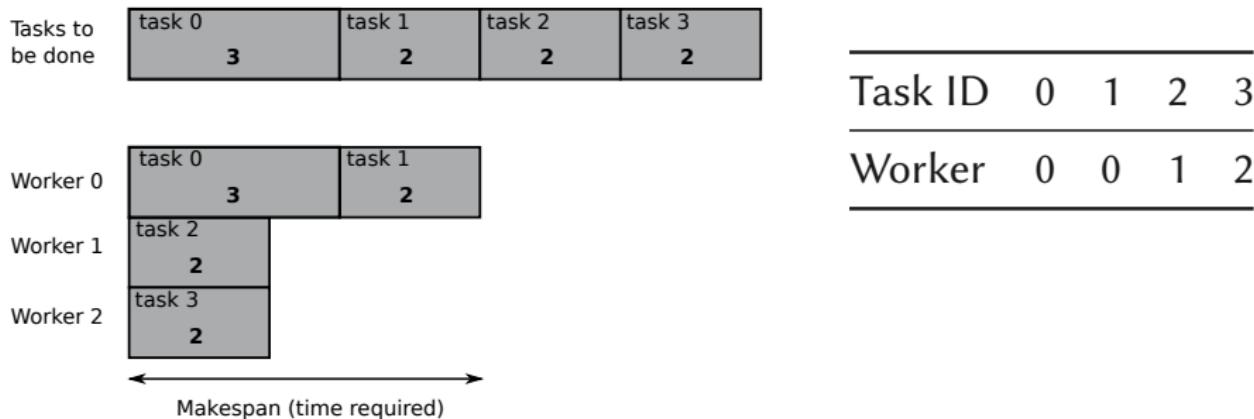


Task ID	0	1	2	3
Worker 0	1	1	0	0
Worker 1	0	0	1	0
Worker 2	0	0	0	1

The disadvantage is that our operators still have to be careful to obey constraints: every column must sum to 1.

# Another alternative encoding

Another alternative encoding might just use one **integer** decision variable per task. The integer gives the worker that it is allocated to.



An advantage of this is that it's easy to write `init` and `nbr`, and our existing crossovers (1-pt, 2-pt, uniform) will work.

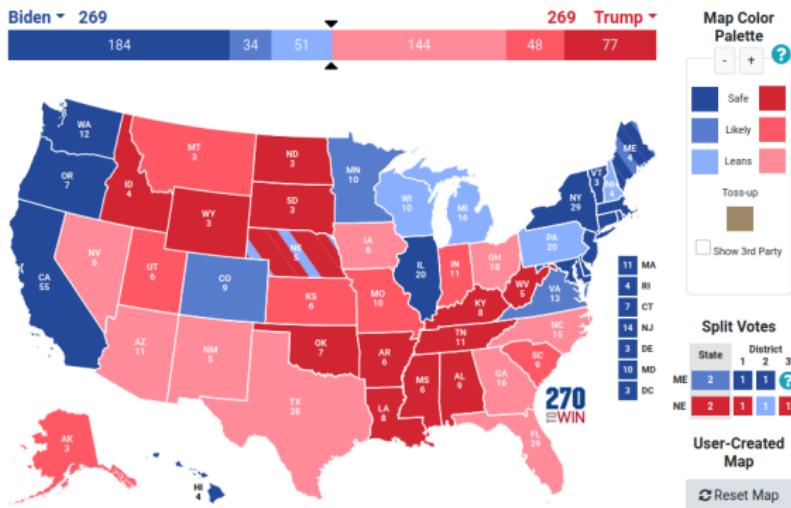
# Set partitioning

Mathematical point of view:

We have a set of numbers (task time requirements) and we want to split them up into equal subsets.

This is the **set partitioning** problem.

## Another application



From 270towin.com

Suppose we want to find scenarios where the US Electoral College is a tie. We can solve this by taking the number of electoral college votes available per state as a set to be partitioned into 2 equal subsets. We'll see this in the lab.

# Overview

- 1 Encodings
- 2 Sets and task allocation
- 3 **Permutations and travelling salesman problems**
- 4 Trees, genetic programming hyperheuristics, and keyword bidding

# Travelling salesman problem



Schneider

# Travelling salesman problem

In the TSP:

- There are  $n$  cities
- Visit every city exactly once and finally return to the start
- We wish to minimise total distance travelled
- We can represent a possible solution as a **permutation**.

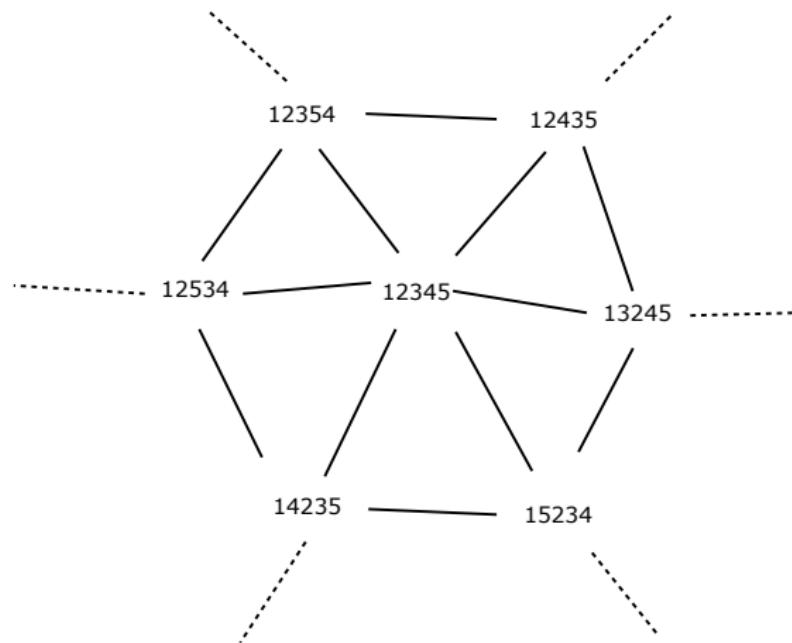
# What is a permutation?

A **permutation** is a mathematical object representing an **ordering** of objects. Usually we just use permutations on the integers between 1 and  $n$  inclusive.

In the TSP, for example, 15243 represents that we will visit city 1 first, then city 5, etc.

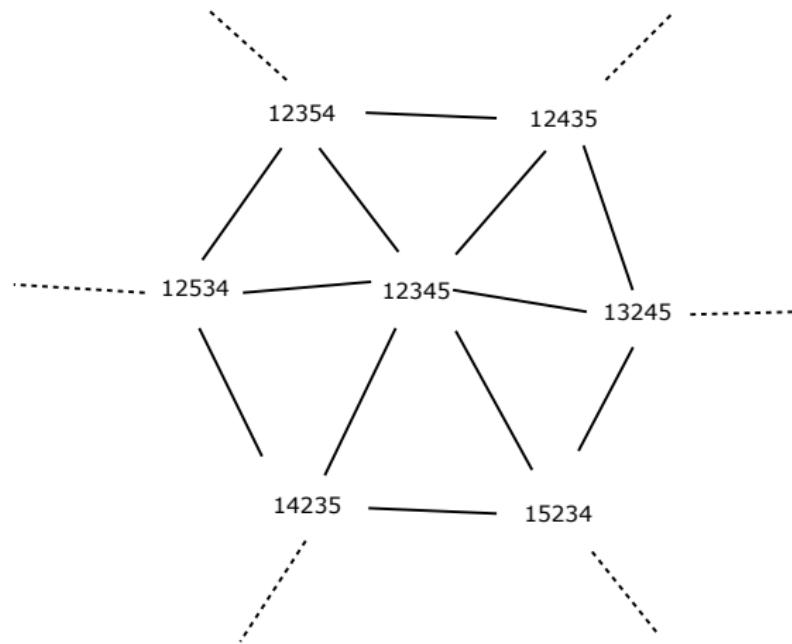
In some applications, permutations **wrap around**: the first item is considered to follow the last item, in the ordering. Because of this, 15243 and 43152 are considered to be the **same** permutation. We can define the **canonical** form as the form where 1 comes first.

# Space of permutations



- Space of permutations of  $n$  items
- $X = \mathbb{P}^n$

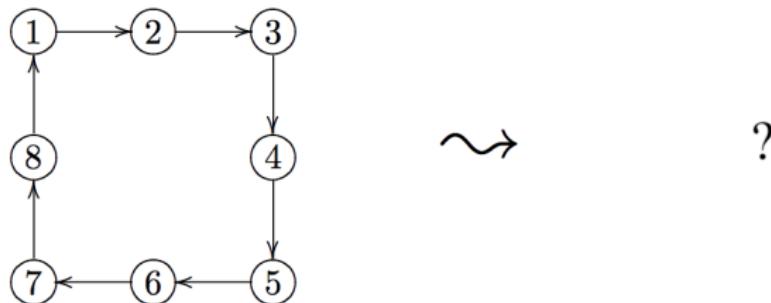
# Space of permutations



- Space of permutations of  $n$  items
- $X = \mathbb{P}^n$

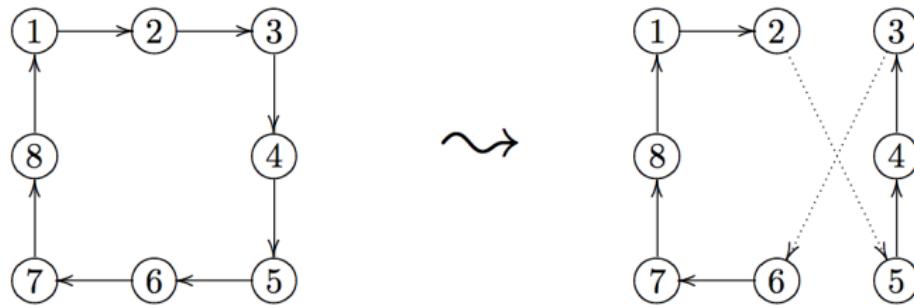
Why is a permutation different from a vector of  $n$  decision variables with integer values in  $[1, n]$ ? What would happen if we used that for a TSP?

# Neighbours in a permutation space



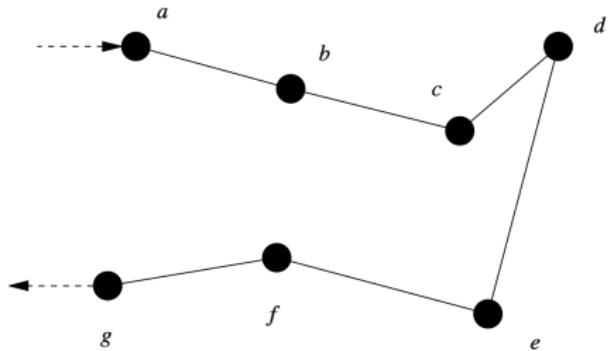
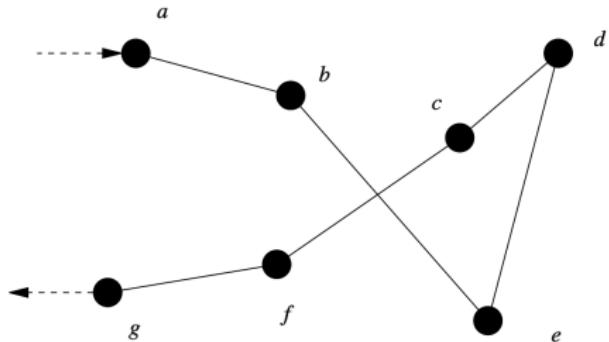
Suppose there are 8 cities to be visited. Suppose we have the solution  $x = 12345678$  as shown. How could we define a **neighbour** function to give  $\text{nbr}(x)$ ? What are the possible neighbours with that definition?

# The 2-opt neighbour function

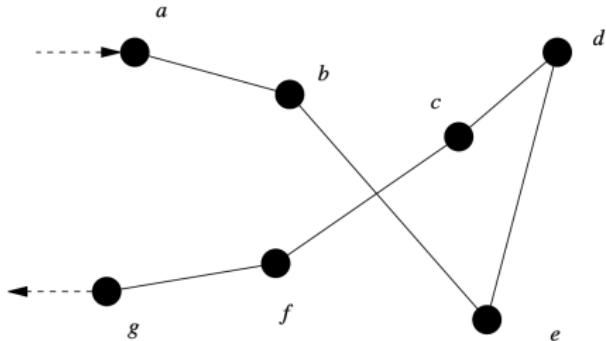


This function chooses a chunk of the permutation and **reverses it**. It's called **2-opt**.

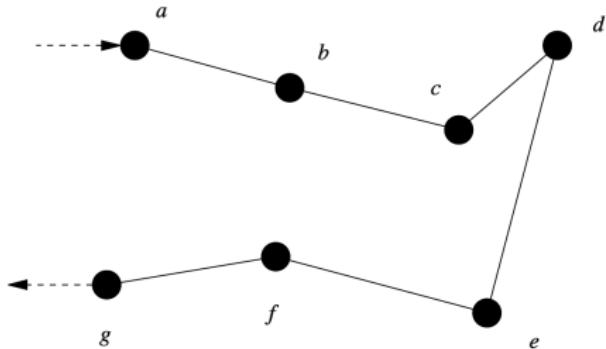
# The 2-opt neighbour function



# The 2-opt neighbour function



Hope: we will (sometimes) choose a chunk which “crosses over” and 2-opt will **uncross it**.



# Crossover in a permutation space

Some authors do define crossover operators for permutations, but most people aren't convinced that it's useful. We won't study them. So, for TSP we'll stick to hill-climbing and related methods, not GAs.

# TSP applications: manufacturing setup



Cameleon

- Paint manufacturer has a recurring order for many colours
- All colours use the same equipment
- After manufacturing one colour, the equipment must be cleaned
- Some transitions take longer than others
- E.g. it takes longer to clean the equipment when going from black to yellow than vice versa
- Goal: minimise total cleaning time.
- Minimise **time** as opposed to distance

# TSP applications: manufacturing setup



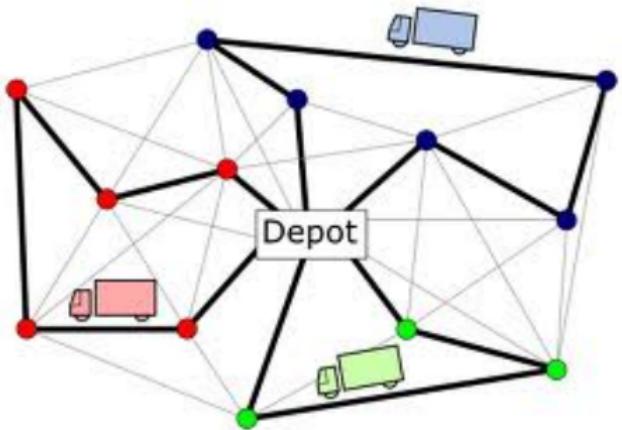
Cameleon

- Paint manufacturer has a recurring order for many colours
- All colours use the same equipment
- After manufacturing one colour, the equipment must be cleaned
- Some transitions take longer than others
  - E.g. it takes longer to clean the equipment when going from black to yellow than vice versa
- Goal: minimise total cleaning time.
- Minimise **time** as opposed to distance
- **Asymmetric** distance matrix

## TSP applications: manufacturing setup

A manufacturer has an order for  $n$  items. Each item requires a slightly different configuration of manufacturing equipment. The order recurs weekly. Changing over from configuration  $i$  to configuration  $j$  takes time  $T_{ij}$ .

# TSP applications: restaurant delivery

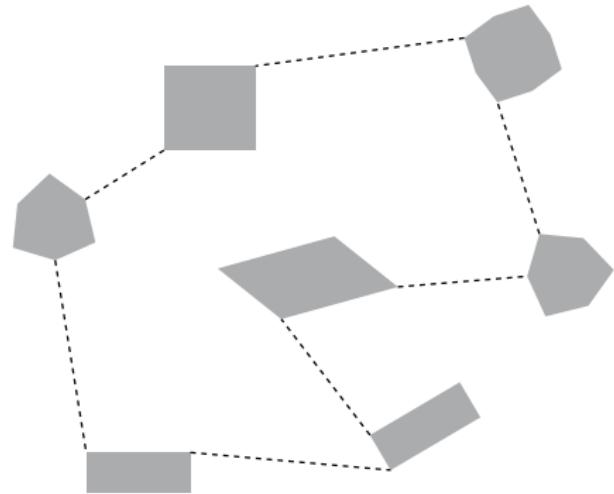


A food delivery company has  $n$  couriers and a stream of incoming orders. Each order is characterised by pick-up and drop-off points and constraints on time of delivery. The **Vehicle Routing Problem** is to assign couriers to orders and decide routes for them, obeying constraints and minimising fuel costs, pay costs, etc. Solving the VRP requires solving the TSP for each courier's route.

# TSP applications: 3D printing



UPenn



A 3D printer prints an object **layer by layer**. Within each layer, it may have to print several **areas**. The time required for printing is irreducible. However, the time required to move between areas may be reduced by ordering them correctly. This is a TSP problem, but we require a specialised distance function because the areas are not just points!

## More TSP later

We will mention the TSP again when discussing **constructive heuristics**, later in the module.

# Overview

- 1 Encodings
- 2 Sets and task allocation
- 3 Permutations and travelling salesman problems
- 4 **Trees, genetic programming hyperheuristics, and keyword bidding**

# Keyword advertising

Suppose we are a retailer and we want to advertise online.

A seller (e.g. Google) offers to show our ad when a user searches for a particular keyword, e.g. “shirt” or “laptop”. We have to decide how much to pay for various keywords.

# Online keyword bidding

A more realistic setting is the **online** version of this problem.

The seller (e.g. Google) set up a (fast, repeated) auction among the advertisers. When we decide to place a bid, we may or may not win the auction. We have to decide how much to bid on each keyword one at a time, perhaps updating our strategy in the light of previous outcomes.

Therefore, our problem is to decide a **policy** for auctions. We want to create a **function** which will return how much to bid in each auction.

# Hyperheuristics

A **hyperheuristic** is a metaheuristic where the search space consists of heuristic rules.

"The difference between conventional methods and evolving heuristics can be summarized by the proverb,

**Give a man a fish and he will eat for a day. Teach a man to fish and he will eat for a lifetime."**

– Burke et al., The Scalability of Evolved On Line Bin Packing Heuristics.

# What data is available?

- Current keyword, its click-through rate (CTR), expected revenue per click
- History of amounts we bid per keyword, and amount each keyword sold for
- ...?

# What data is available?

- Current keyword, its click-through rate (CTR), expected revenue per click
- History of amounts we bid per keyword, and amount each keyword sold for
- ...?

Let's say we have a feature vector  $x = (x_0, x_1, \dots, x_{n-1})$ .

# Some possible keyword bidding policies

- Bid EUR1
- Bid current keyword CTR multiplied by expected revenue per click multiplied by 0.5
- Bid  $\text{CTR} / 17 + \text{number of previous bids that failed}$
- ...?

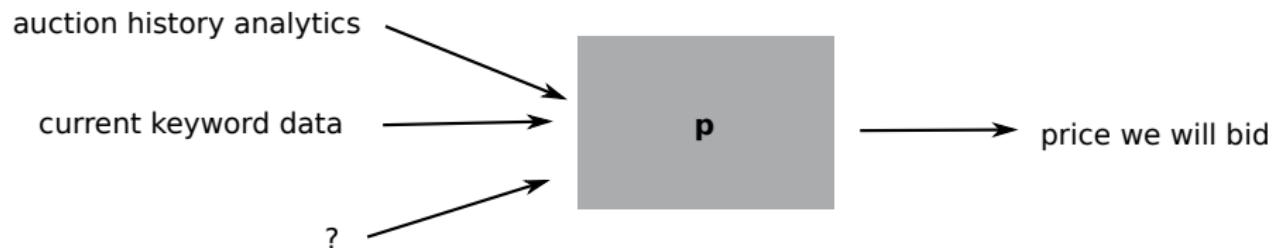
# Some possible keyword bidding policies

- Bid EUR1
- Bid current keyword CTR multiplied by expected revenue per click multiplied by 0.5
- Bid  $\text{CTR} / 17 + \text{number of previous bids that failed}$
- ...?

There are many possible policies, some easy to understand, some impossible!

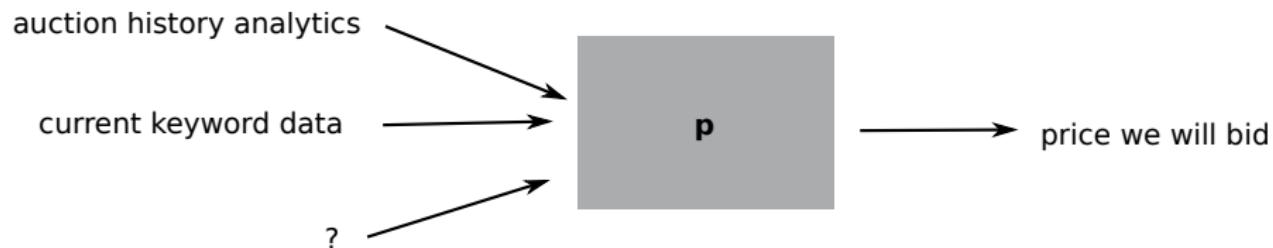
# Keyword bidding policies

In full generality a policy is a **program**  $p(x)$ . Its input is the feature vector  $x$  (including data on the current keyword) and its output is an amount we should bid for that keyword.



# Keyword bidding policies

In full generality a policy is a **program**  $p(x)$ . Its input is the feature vector  $x$  (including data on the current keyword) and its output is an amount we should bid for that keyword.

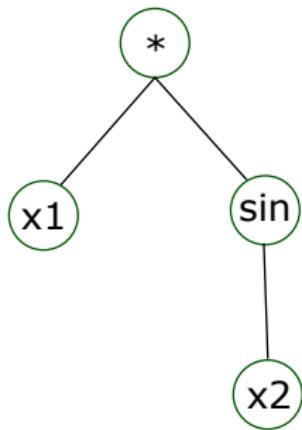


NB this program is **not** an objective function! (We will discuss the objective later.)

# Programs are trees

- We can represent a program by a **tree**
- Recall that a tree is a **graph** which is:
  - **connected** (no disconnected parts) and
  - **acyclic** (no cycles)
- A program tree is:
  - **rooted** (there is a special node called the **root**) and
  - **ordered** (the order of children is important).

# Programs are trees



$x_1 \sin(x_2)$

We should bid:

$$p(x) = x_1 \sin(x_2)$$

# Genetic Programming

GP = program synthesis (or automated programming) by evolution

Make random programs

**Initialisation**

**Fitness**

Test them

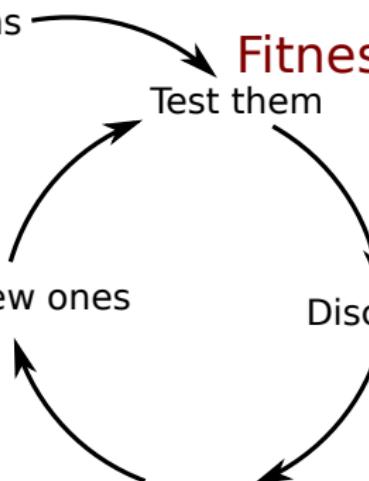
Mutate the new ones

**Mutation**

Discard bad ones

**Selection**

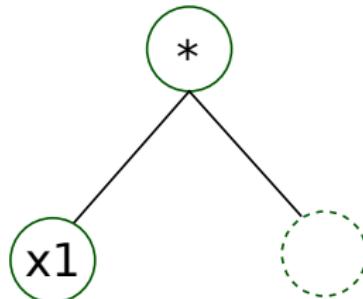
Crossover  
Mate good ones  
to make new ones



# Initialisation on trees

The **grow** method:

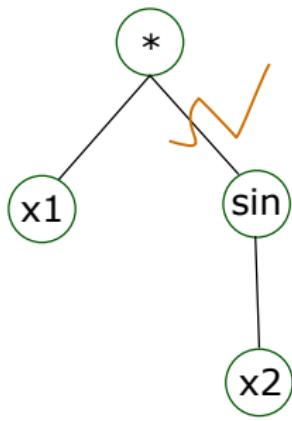
- 1 Start at the root
- 2 If we are at the maximum tree depth, choose a random **leaf** (e.g.  $x_i$  or a constant)
- 3 Else, choose a function (e.g.  $\sin$ ,  $*$ , etc.) or leaf at random
- 4 If we choose a function with  $n > 0$  children, then for each one, create a “slot” below the current node, move to it, and recurse to 2; otherwise return.



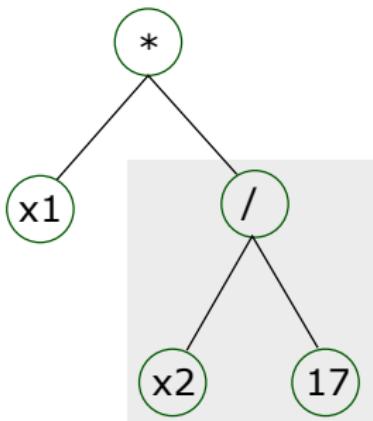
# Mutation on trees

Subtree mutation:

- 1 Choose an edge at random
- 2 Discard the subtree below that edge
- 3 Generate a new subtree using **grow**, and paste it in.



$x1 \sin(x2)$

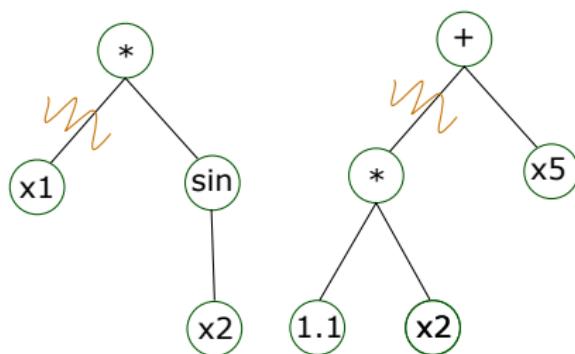


$x1(x2/17)$

# Crossover on trees

Subtree crossover: choose a cutpoint randomly in each tree, and swap the subtrees below them.

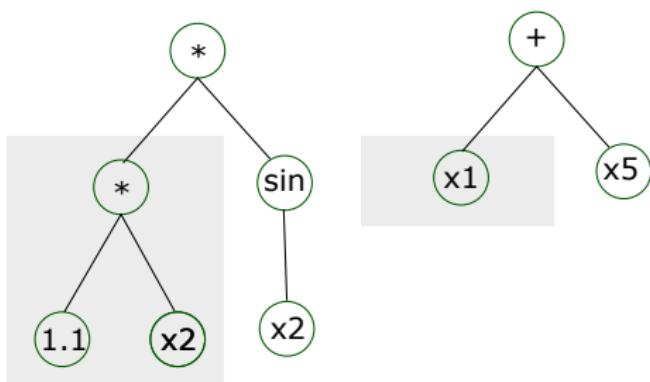
parents



$x_1 \sin(x_2)$

$1.1x_2 + x_5$

children



$1.1x_2 \sin(x_2)$

$x_1+x_5$

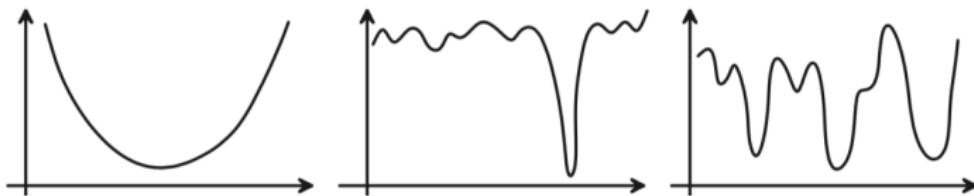
# Fitness

- To define fitness, we can use a dataset of old auctions, and simulate the auction process, and see how much profit we make.
- We'll omit lots of details here.
- (Optional, not examinable): read [Zhou et al](#) for more real-world details.

# Further reading

- Luke, **Essentials**, Section 4.3: Trees and Genetic Programming.

# Which problems are hard?



Small (easy); medium-sized (hard); large (easy)

Perhaps the biggest surprise of all about high-dimensional nonlinear search is just how well it can work. Variants of simulated annealing are now used routinely for such hard problems as routing fleets of airplanes. The explanation for this success can be called the blessing of dimensionality. [...] For a large number of [variables] it is extremely difficult to find the global minimum, but there is an enormous number of local minima that are all almost equally good [...]. Almost anywhere you look you will be near a reasonably low-energy solution.

In fact, the really hard problems are not the big ones. If there are enough planes in a fleet, there are many different routings that are comparable. Small problems by definition are not hard; for a small fleet the routing options can be exhaustively checked. What's difficult is the intermediate case, where there are too many planes for a simple search to find the global best answer, but there are too few planes for there to be many alternative acceptable routings. [...] In between is a transition between these regimes that can be very difficult to handle. This crossover has been shown to have many of the characteristics of a phase transition [Cheeseman et al., 1991; Kirkpatrick & Selman, 1994], which is usually where the most complex behavior in a system occurs.

From Gershenfeld, The Nature of Mathematical Modelling (p. 198) <http://fab.cba.mit.edu/classes/864.20/index.html>.

# Overview

- 1 Encodings
- 2 Sets and task allocation
- 3 Permutations and travelling salesman problems
- 4 Trees, genetic programming hyperheuristics, and keyword bidding

# Lecture 09 – Multiobjective Optimisation

Optimisation CT5141

James McDermott

NUI Galway

2020



OÉ Gaillimh  
NUI Galway

# Overview

- 1 Utility**
- 2 Multi-criteria decision-making**
- 3 Case study: Energy cost and emissions**
- 4 Multi-objective optimisation: problems**
- 5 Multi-objective optimisation: algorithms**

# Utility and decision-making

**Utility** is an abstract measure of how much we like something. When we have to choose among several options, we choose the one which gives us the highest utility.

# Utility and decision-making

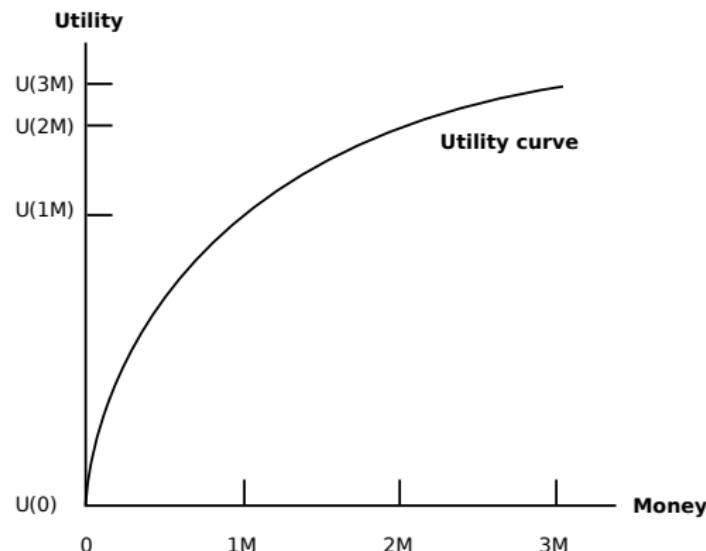
**Utility** is an abstract measure of how much we like something. When we have to choose among several options, we choose the one which gives us the highest utility.

- We choose the dessert that tastes best
- We vote for the party whose policies match our preferences
- If we can buy an item for different prices in different shops, we choose the lowest
- If we have to bet on a horse race, we make the bet with the highest **expected payoff**

# Utility is nonlinear in money

Obviously, receiving more money means higher utility. We can say utility is **monotonic** in money. But it is **nonlinear**.

EUR1m would represent a large amount of utility to me; but to Bill Gates it would barely change his utility. Another way to say it: the second million is worth less than the first!



Economists studying this make models based on observation, e.g. for some amount of money  $x$ , maybe utility  $U(x) = \log(x)$ .

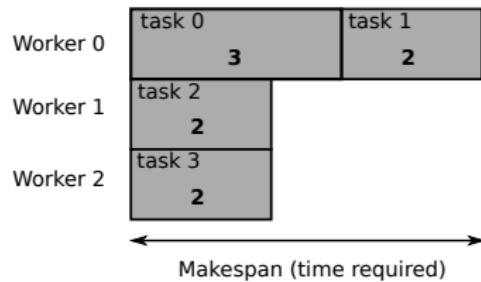
# How does this relate to optimisation?

When writing an objective function, we are really trying to express our **utility function** for the items in the search space.

# Objective function = utility function

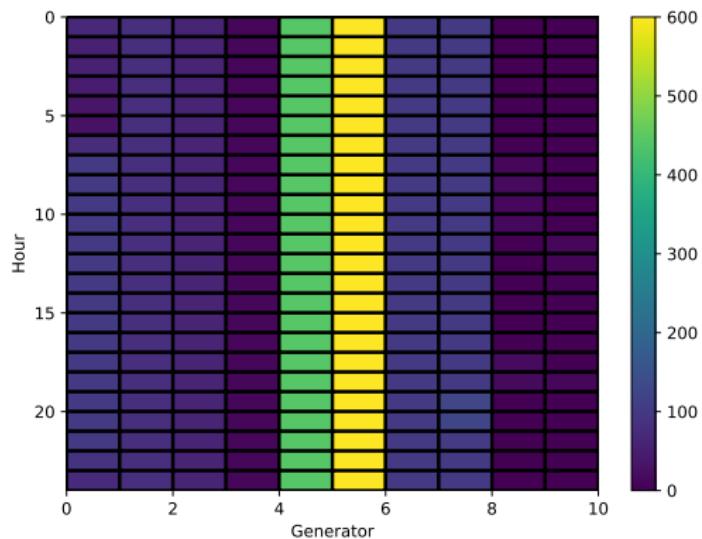
Often our utility function is obvious, e.g. in set partitioning for task allocation, the shortest makespan is the best.

Tasks to be done	task 0 3	task 1 2	task 2 2	task 3 2
------------------	-------------	-------------	-------------	-------------



# Objective function = utility function

Sometimes it might not be obvious, e.g. in Unit Commitment we might want to reduce costs and reduce emissions.



Here we have **multiple criteria** or **multiple objectives**, and they are in conflict.

# Multi-criteria decision-making and multi-objective optimisation

Multiple objectives is a problem in two situations:

- If we have just a few options, how can we decide among them?  
This is **multi-criteria decision-making**
- If we have a large search space, how can we define an  
optimisation algorithm? This is **multi-objective optimisation**.

# Off-topic: Decision Theory

(Not examinable)

- Utility is just the first topic in the field of **Decision Theory**
- If you get a chance to study more, I recommend it!
- There are fascinating concepts like the **value of information**, **decision trees**, human **heuristics and biases**, **analytical processes** for decision-making and consensus, and more.

# Overview

- 1 Utility
- 2 **Multi-criteria decision-making**
- 3 Case study: Energy cost and emissions
- 4 Multi-objective optimisation: problems
- 5 Multi-objective optimisation: algorithms

# Simple and complex decisions

- In some decisions, our utility is simple, e.g. maximising profits, or minimising the time required for a task
- Sometimes we have **multiple objectives**, but we can reduce them to one objective by applying a **weighting scheme**
- Sometimes we have multiple objectives, and we **can't** reduce them to one objective.

# Weighting schemes

A weighting scheme is a set of numerical weights  $w$ , one per objective.

- Often we insist  $\sum w = 1$ .
- If  $w_0 = 0.5$  and  $w_1 = w_2 = 0.25$ , that says objective 0 is twice as important as the other two.
- For this to make sense, we should ensure the objectives all have the same direction (maximise or minimise)...
- ... and are in the same scale.

# Example: buying cars

Suppose we're buying a second-hand car. We want a green saloon, with high fuel efficiency, low mileage, and low price.



Subjective weightings:

- $w_0 = 0.1$ : Colour = green
- $w_1 = 0.2$ : Model = saloon
- $w_2 = 0.1$ : Fuel efficiency (/100 MPG)
- $w_3 = 0.2$ : Negative mileage (/10000)
- $w_4 = 0.4$ : Negative price (/1000)

# Example: buying cars

Here are two possible cars:

- 1 Blue, saloon, 112MPG, 30,000 miles, EUR10,000
- 2 Green, saloon, 115MPG, 45,000 miles, EUR12,000

Here are my weightings:

- $w_0 = 0.1$ : Colour = green
- $w_1 = 0.2$ : Model = saloon
- $w_2 = 0.1$ : Fuel efficiency (/100 MPG)
- $w_3 = 0.2$ : Negative mileage (/10000)
- $w_4 = 0.4$ : Negative price (/1000)

# Example: buying cars

Here are two possible cars:

- 1 Blue, saloon, 112MPG, 30,000 miles, EUR10,000
- 2 Green, saloon, 115MPG, 45,000 miles, EUR12,000

Here are my weightings:

- $w_0 = 0.1$ : Colour = green
- $w_1 = 0.2$ : Model = saloon
- $w_2 = 0.1$ : Fuel efficiency (/100 MPG)
- $w_3 = 0.2$ : Negative mileage (/10000)
- $w_4 = 0.4$ : Negative price (/1000)
- **f(Car 1) =**  
$$0.1 * 0 + 0.2 * 1 + 0.1 * 1.12 + 0.2 * -3.0 + 0.4 * -10 = -4.288$$
- **f(Car 2) =**  
$$0.1 * 1 + 0.2 * 1 + 0.1 * 1.15 + 0.2 * -4.5 + 0.4 * -12 = -5.285$$

# Example: buying cars

- By assigning these weights, I'm implicitly saying e.g. "each unit of fuel efficiency maps to a certain increase in how much I'm willing to pay" – effectively a statement about **utility**.
- We end up with an unambiguous rank-ordering on all available cars.

# Multi-criteria decision-making



However, sometimes we have multiple criteria and we **can't apply weightings**.

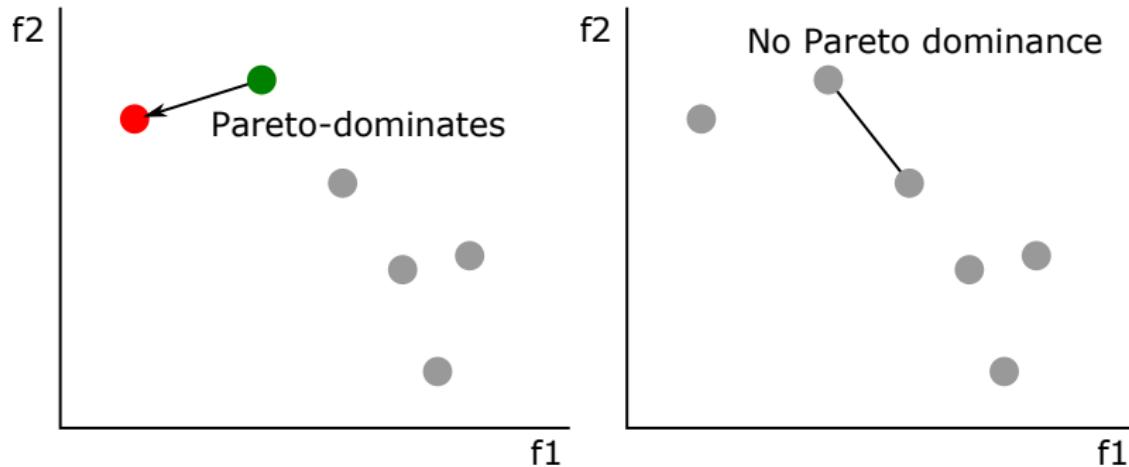
We say such criteria are **incommensurable** – no way to put them on the same scale.

# Plotting our options

For any shoe  $x$ , let's define  $f_1(x)$  = beauty;  $f_2(x)$  = comfort. (Higher is better, for both.)



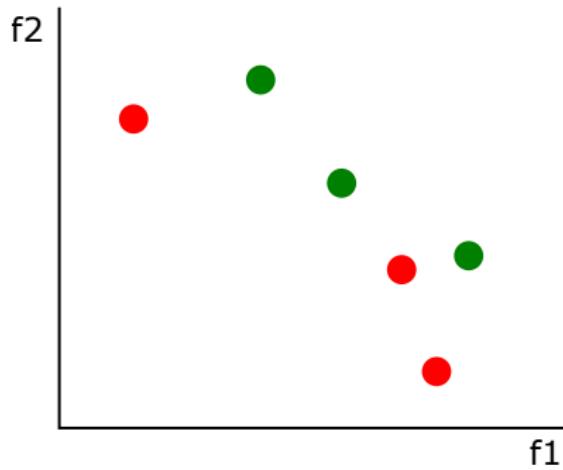
# Pareto dominance



We say that an option  $x_1$  **Pareto-dominates** an option  $x_2$  if  $x_1$  is **better than**  $x_2$  on at least one  $f_i$ , and **at least equal to**  $x_2$  on all  $f_i$ .

An option is **Pareto-dominated** if some other option Pareto-dominates it.

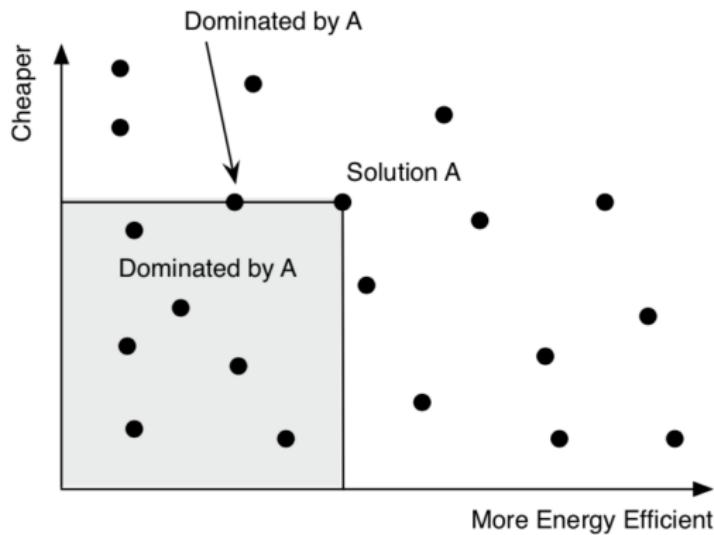
# Pareto front



This allows us to discard dominated options, those shown in red.

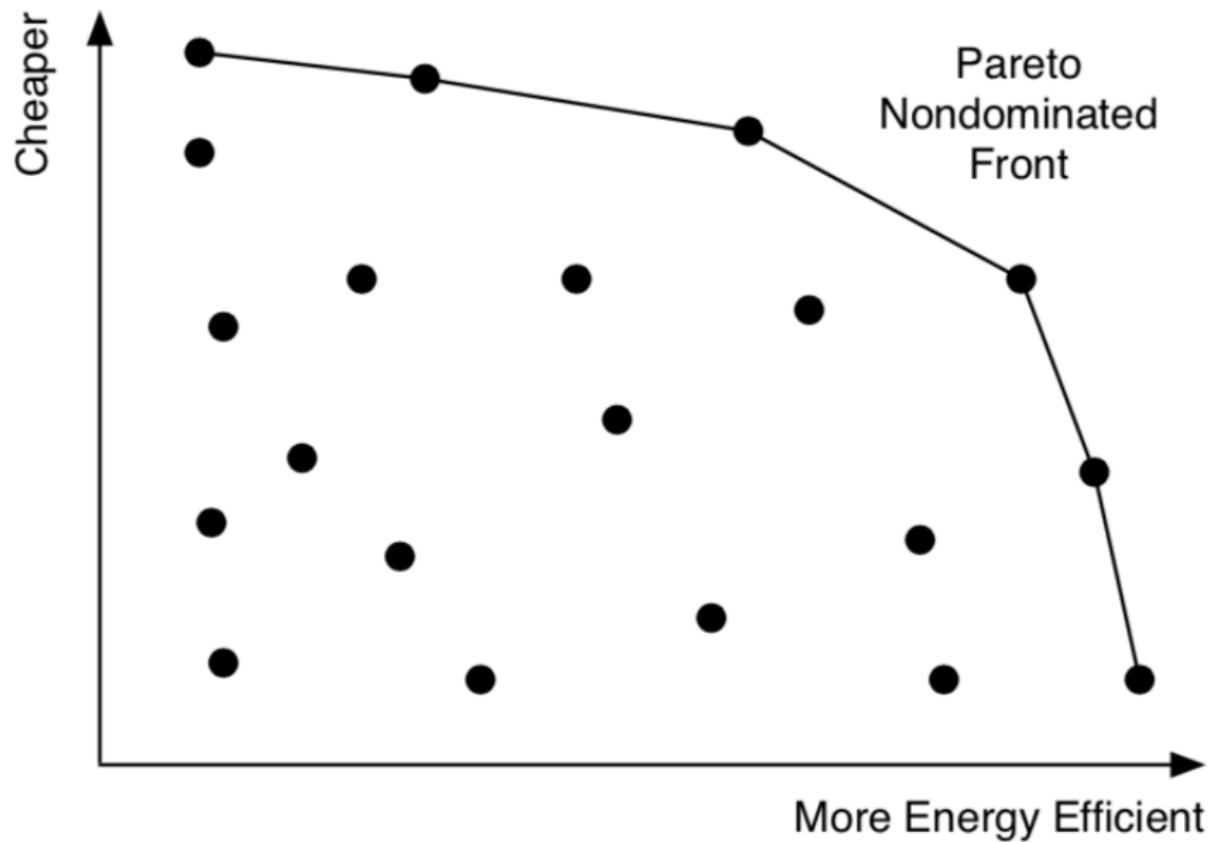
The remainder in green are called the **Pareto front**, the **efficient front**, and similar names. But we have no way to choose between them!

# Pareto Dominance



This diagram and later ones are from Luke, Essentials of Metaheuristics, Chapter 7

# Pareto Front



# Summary

Sometimes, decisions involve multiple criteria which may be in conflict:

- Buying second-hand car
- Buying shoes
- This is **multi-criteria decision-making**
- We have a small set of options, so we can look at them all
  - No need to optimise
- We can identify the Pareto front
- After identifying the Pareto front, we are on our own!

# Overview

- 1 Utility
- 2 Multi-criteria decision-making
- 3 **Case study: Energy cost and emissions**
- 4 Multi-objective optimisation: problems
- 5 Multi-objective optimisation: algorithms

# Recall Unit Commitment problem

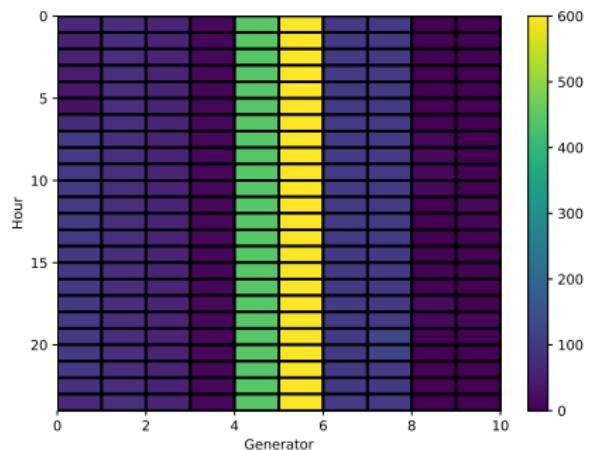
In a small country there are 10 generators of four types: Hydroelectric, Solid fuel, Gas, Solar. Each generator has a lower and upper bound on its production per hour (in MW/h).

	type	LB (MW)	UB (MW)	c (EUR/MW)	p (tons CO2/MW)
A	hydro	10	100	1.4	0.024
B	hydro	10	80	1.4	0.024
C	hydro	10	60	1.4	0.024
D	hydro	1	10	1.4	0.024
E	solid	100	900	4.4	0.82
F	solid	100	600	4.4	0.82
G	solid	10	100	4.4	0.82
H	gas	100	400	9.1	0.49
I	solar	0	70	6.6	0
J	solar	0	20	6.6	0

# Minimising cost

Minimise  $\sum_{i,j} c_j X_{i,j}$

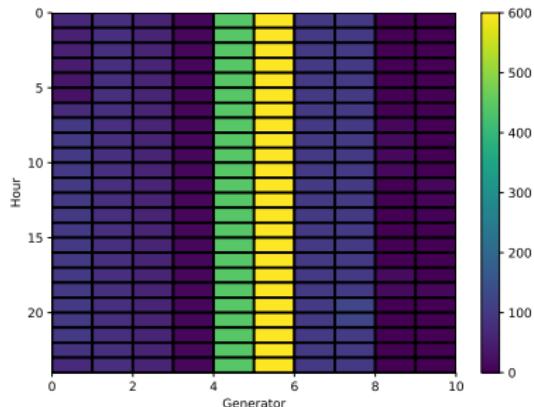
Subject to meeting demand,  $X$  within lower/upper bounds, etc.



- Cost: EUR151518
- Emissions: 23810 tons CO<sub>2</sub>

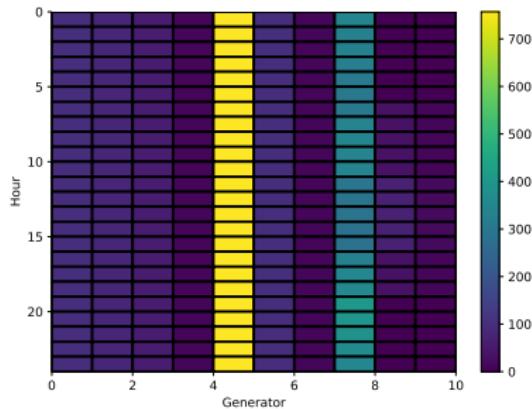
# Alternative: minimising emissions

Minimise  $\sum_{i,j} c_j X_{i,j}$



- Cost: EUR151518
- Emissions: 23810 tons CO<sub>2</sub>

Minimise  $\sum_{i,j} p_j X_{i,j}$



- Cost: EUR177846
- Emissions: 21113 tons CO<sub>2</sub>

# A spectrum of solutions

We have considered **only cost**, and then **only emissions**. We can try for a bit of both!

Define weights  $w_0$  and  $w_1$  such that  $w_0 + w_1 = 1$ . Define our objective as a weighted sum:

Minimise:

$$w_0 \sum_{i,j} c_j X_{i,j} + w_1 \sum_{i,j} p_j X_{i,j}$$

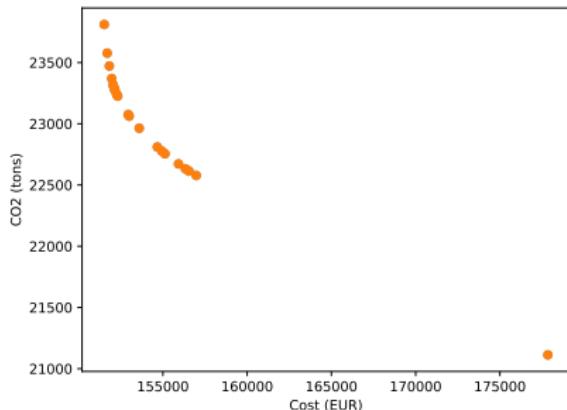
For any value of  $w_0$ ,  $w_1$ , this is now a **single objective**.

# A spectrum of solutions

## Grid search:

- 1 For many values of  $w_0$ ,
- 2 Define  $w_1 = 1 - w_0$
- 3 Define the weighted objective and solve the LP
- 4 Take the Pareto front across the resulting solutions' cost and emissions values.

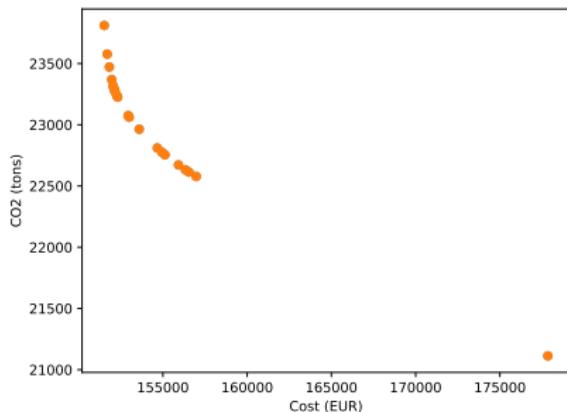
# A spectrum of solutions



Remember, lower is better for both objectives.

We obtain 1001 items, of which 21 were unique, all in the Pareto front

# A spectrum of solutions

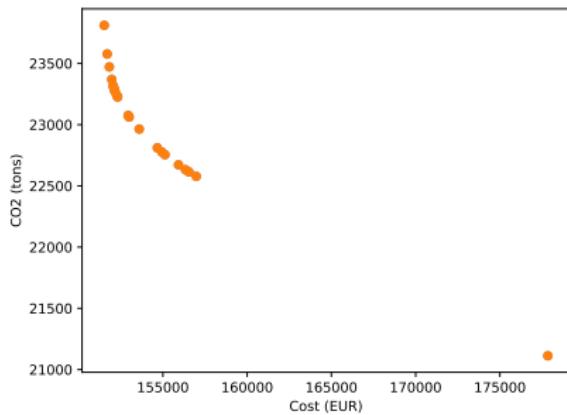


Remember, lower is better for both objectives.

We obtain 1001 items, of which 21 were unique, all in the Pareto front

There is a large gap in the front - a small change in weights leads to a large change in the solution and the objective.

# A spectrum of solutions



Remember, all of these solutions are **Pareto-optimal**.

Having obtained this Pareto front, we are finished. A decision-maker now has to **decide**. Sometimes they'll aim for a **knee** in the plot, or an extreme, or they'll introduce new criteria or other issues.

# Decision-making versus optimisation

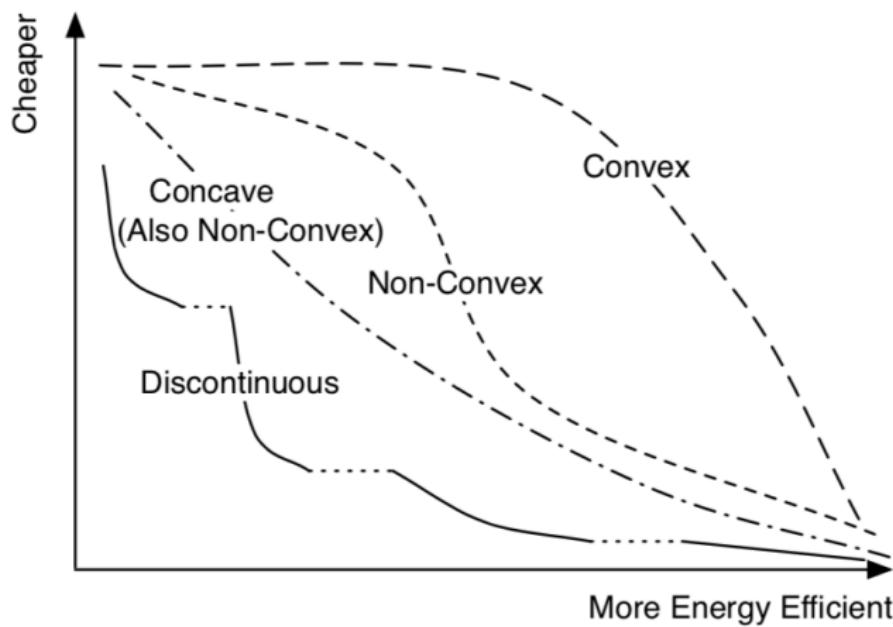
- We did not use multiobjective optimisation of UC here;
- We used grid search to define many new problems, each single-objective;
- Solved each using LP;
- Assembled all solutions to a Pareto front.

(We could say that the grid search was an optimisation **in the space of weights**, which is only 1D.)

# Code

See uc\_lp\_M00.py.

# Types of fronts



(See also NSGA2 paper)

# Overview

- 1 Utility
- 2 Multi-criteria decision-making
- 3 Case study: Energy cost and emissions
- 4 Multi-objective optimisation: problems**
- 5 Multi-objective optimisation: algorithms

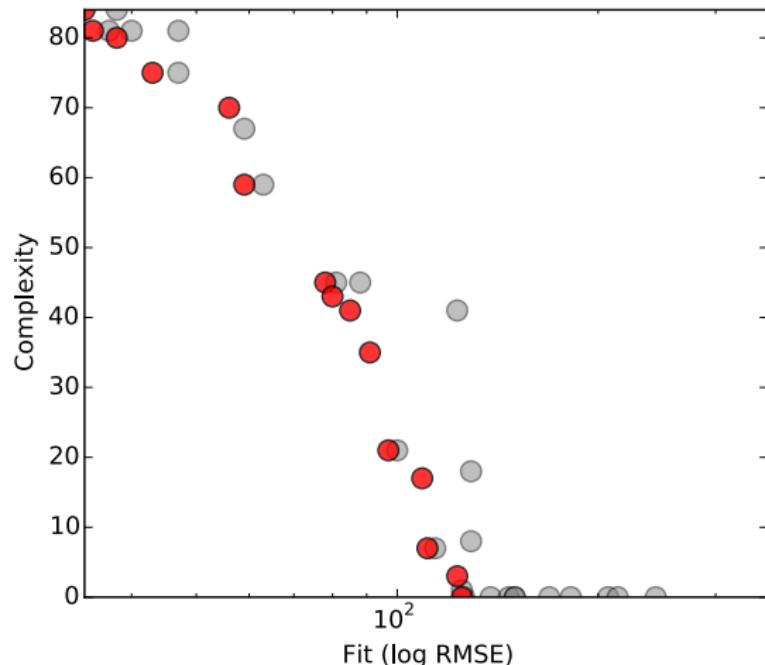
# Multi-objective optimisation

We have discussed multi-criteria decision-making (MCDM), that is where we have to choose between several options with conflicting criteria.

The same concepts are also applied in multi-objective optimisation (MOO).

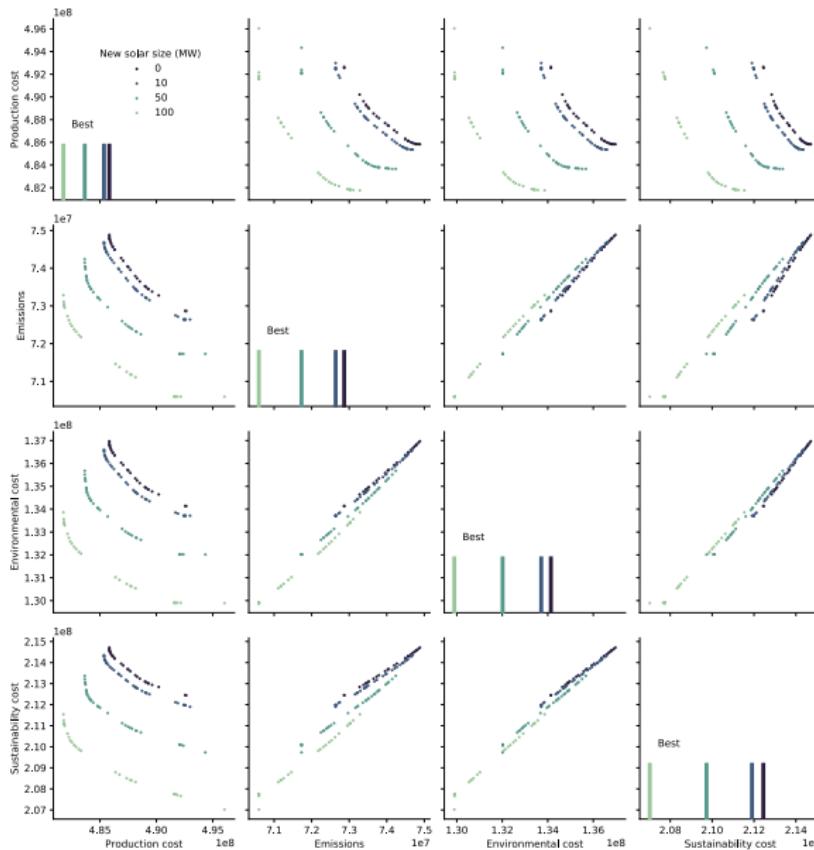
- In MCDM the “search space” is small, so we can consider all options
  - E.g. with Energy Cost versus Emissions, there were just a few unique solutions found by the grid search.
- In MOO we have to search.

# Genetic Programming symbolic regression



Minimise regression error, minimise complexity.

# Serbian Unit Commitment problem

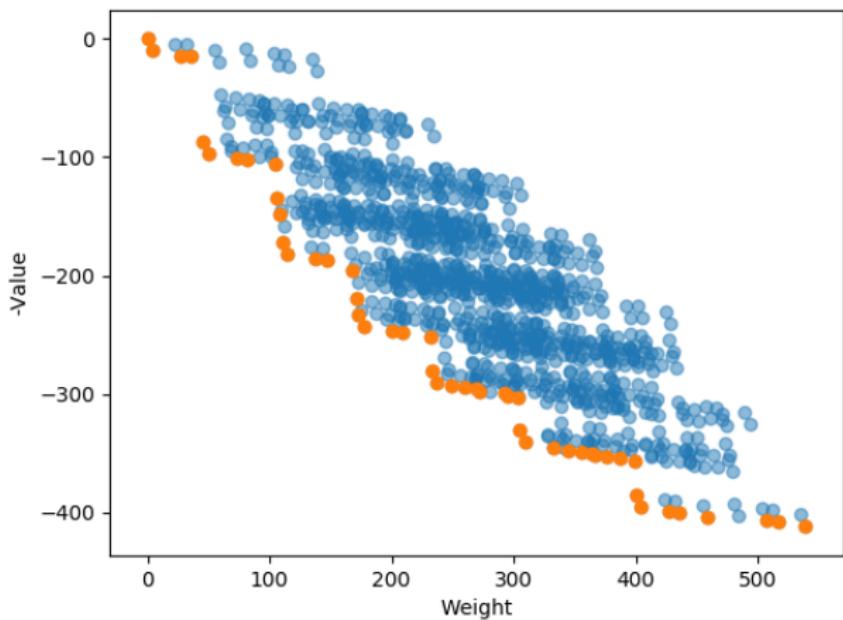


4D Pareto front:

- Production cost
- Environmental cost
- Sustainability cost
- Emissions

# Knapsack problem

There is a set of items. Every item has a **value** and a **weight**. We have to choose a subset of large value and small weight.



# Calculating the Pareto front

- A simple algorithm is slow
- A fast algorithm takes some thought
- Trade-off between fast for many points versus fast for many objectives

Code and discussion [here](#).

# How good is our solution?

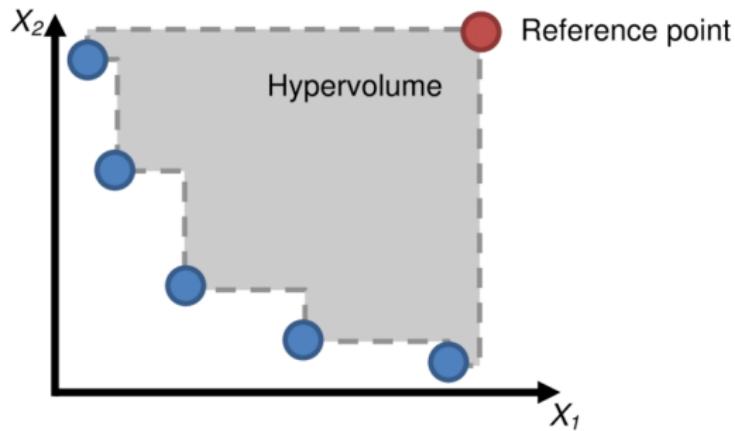
In single-objective optimisation we can just report the best value of the objective. But in MOO, we have to report a whole Pareto front. If we are comparing two algorithms, or judging progress over time, that is tricky. How do we tell whether one Pareto front is better than another?

# How good is our solution?

Principles:

- Reward Pareto Front for good extreme values
- Reward for diversity across the front (as opposed to clumping)
- If Pareto front PF1 has all the same items as PF2 plus some extra, then PF1 is better.

# Hypervolume indicator



Ribeiro

Choose some **reference point** worse than all possible solutions on all objectives and calculate this volume: the larger the better.

DEAP usage

# Overview

- 1 Utility
- 2 Multi-criteria decision-making
- 3 Case study: Energy cost and emissions
- 4 Multi-objective optimisation: problems
- 5 **Multi-objective optimisation: algorithms**

# Some Algorithms

Use a weighting scheme:

- Then apply single-objective optimisation
- Consider multiple values of weights

Simple algorithms:

- Pareto archive
- Random-objective tournament selection

Sophisticated algorithms

- **NSGA2**, NSGA3
- SPEA and successors
- Others

# Elitism in MOO

In MOO, the “elite” is just the Pareto front of all points considered so far. It seems natural to **never discard an elite individual**. This leads to a simple algorithm.

# Pareto Archive algorithm

- 1 Create a set of individuals.
- 2 Select random individual(s) from the archive.
- 3 Generate new individual(s) by mutation and/or crossover.
- 4 Add the new individual(s) to the archive. If that individual Pareto-dominates any individuals in the set, then **they** are discarded. If it is Pareto-dominated **by** any individual in the set, then **it** is discarded instead.
- 5 Go to 2.

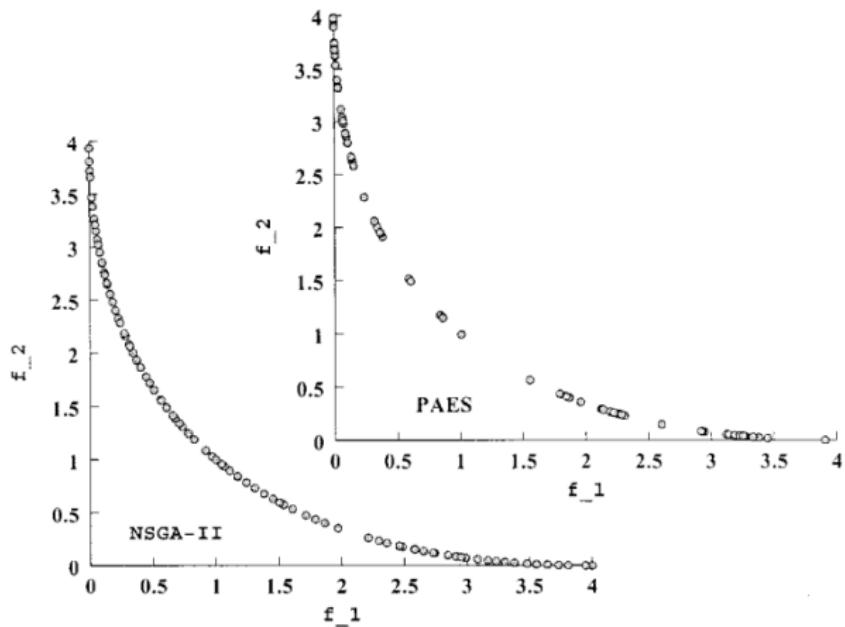
In this algorithm we don't have **generations**. We don't have a fixed population size – in fact the set is likely to grow over time.

# Random-objective tournament selection

- This uses a **normal population-based algorithm**, but:
- At each selection event, we choose **one of our objectives at random** and use it to determine the tournament winner
- Variant known as **lexicographic selection algorithm**
- See Luke, Algorithm 95.

# Drawbacks

The main drawback of these simpler algorithms (Pareto Archive algorithm and random-objective tournament selection): they fail to fully “spread” the population over the Pareto front.

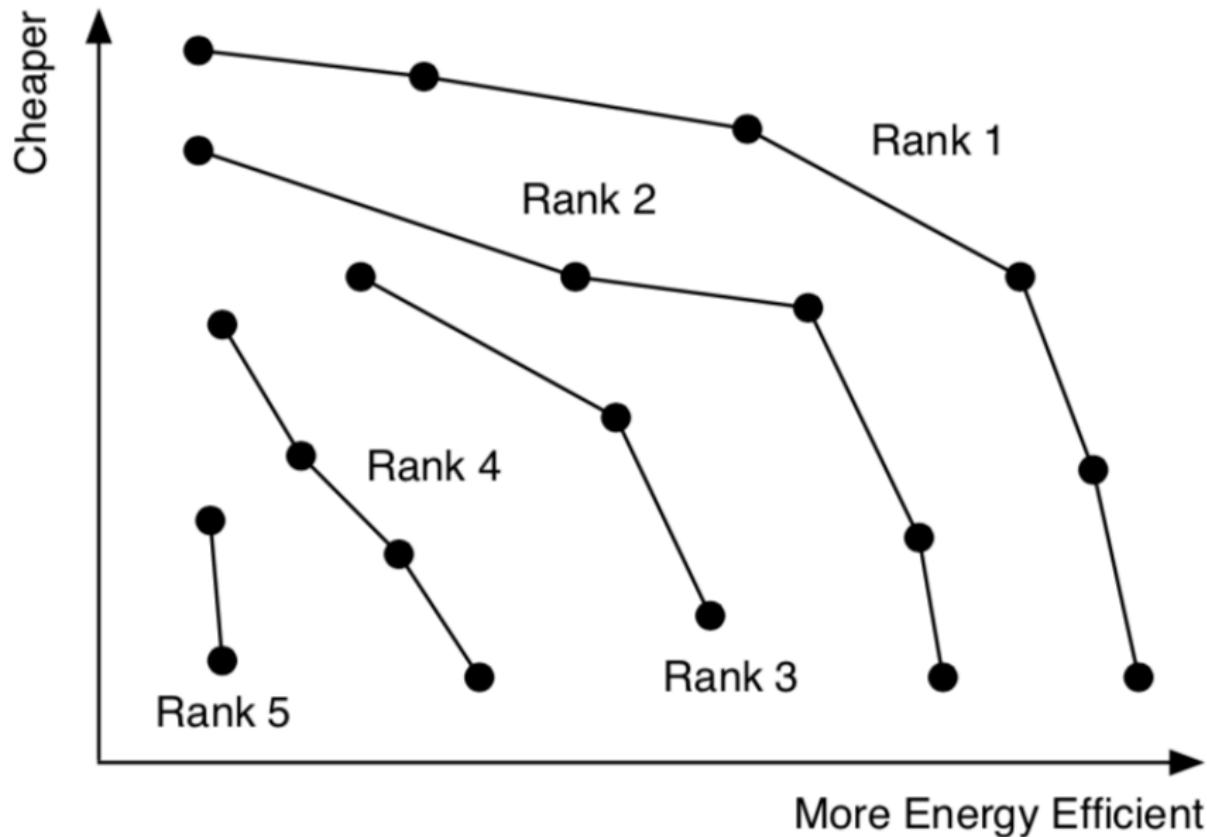


Deb et al.

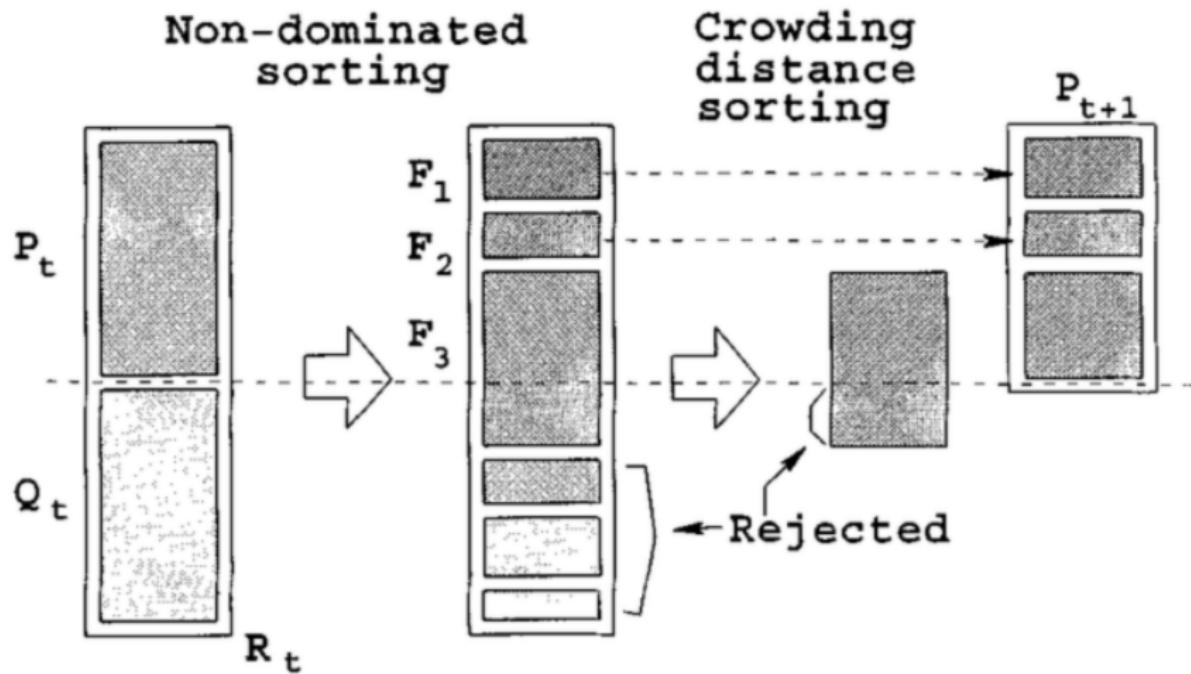
# NSGA2

- Non-dominated sorting genetic algorithm 2 (Deb et al., in Blackboard)
- Over 5000 citations.

# Sorting



# NSGA2 main loop



Deb et al.

# NSGA2

Main loop:

- Create children from current population
- Merge with population
- Select according to below procedure
- Discard the rest

# NSGA2

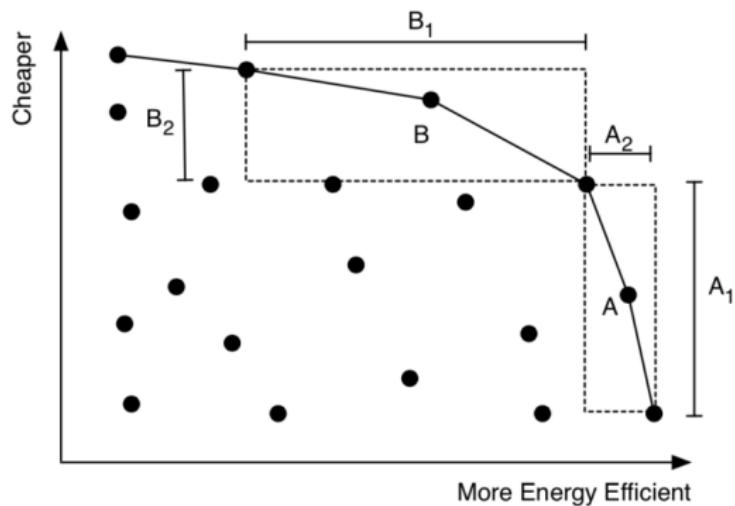
Main loop:

- Create children from current population
- Merge with population
- Select according to below procedure
- Discard the rest

Selection:

- Select all of the Pareto front
- Then remove the Pareto front and re-calculate the next Pareto front, and select all of this
- Repeat (remove, re-calculate, select) until the next Pareto front **would** make our population over-full
- From that Pareto front, just select the most **sparse**.

# Sparsity



Individual B has sparsity  $B_1+B_2$ . An extreme individual has sparsity  $\infty$ . See Luke, Algorithm 102 for details.

The second main idea in NSGA2 is: we would like to achieve a good **spread** of individuals along a wide Pareto front. We select the individuals with highest sparsity, based on how close they are (in objective space) to other individuals.

# Recap

- Objective function = Utility
  - Utility a bit more complex than we might think
  - Sub-linear in money
- Weighting schemes can sometimes be useful in MCDM and MOO, but...
- ... **Pareto dominance** is the central idea in both
- NSGA2 is the main algorithm in MOO
- Selection based on:
  - Ranking of Pareto fronts
  - Sparsity
- We'll look at more in labs.

# Reading

Luke, **Essentials**, Chapter on Multiobjective optimisation.

# Overview

- 1 Utility
- 2 Multi-criteria decision-making
- 3 Case study: Energy cost and emissions
- 4 Multi-objective optimisation: problems
- 5 Multi-objective optimisation: algorithms