Lecture 07 – Combinatorial optimisation

Optimisation CT5141

James McDermott

University of 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.)

Often with:

- Logical constraints
- Dependencies between parts of the genome.

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

Overview

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

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

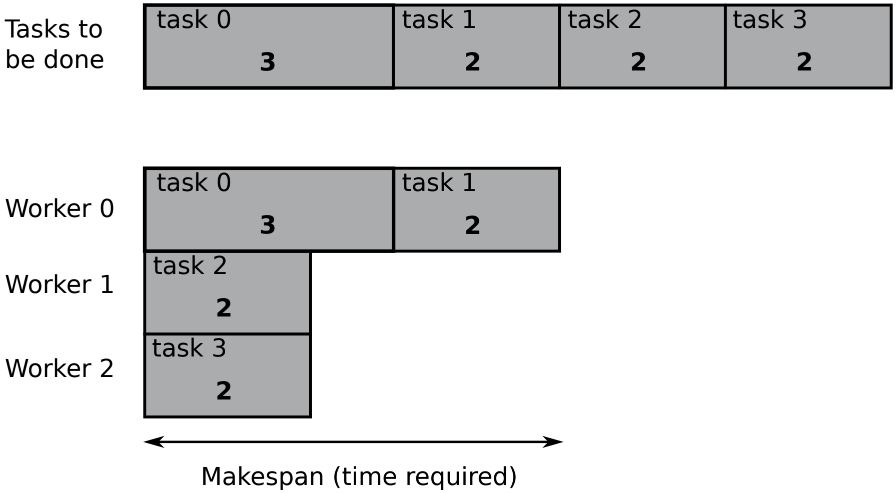
From roboticsandautomationnews.com

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.

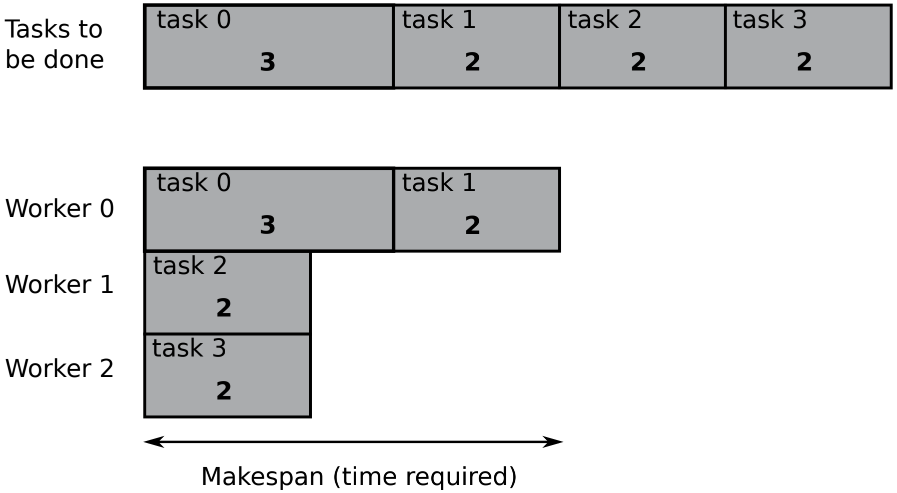
Objective

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

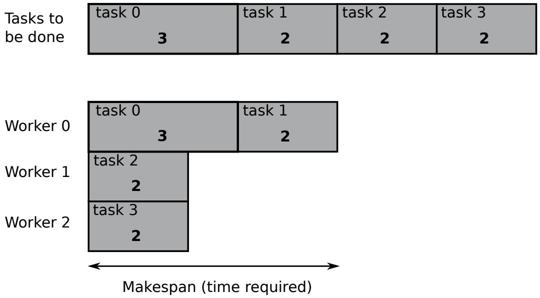
$$\text{Minimise: } (\max_w \sum s_w - \min_w \sum s_w) / \max(d)$$

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.



We create three sets: `{{0, 1}, {2}, {3}}`.

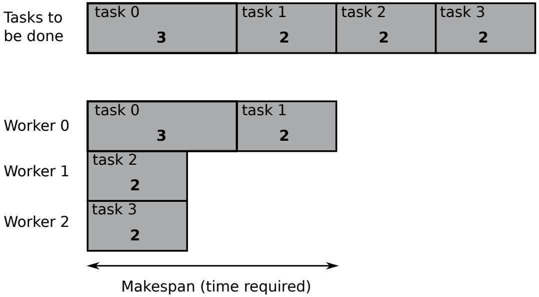
The `init` function is easy to write. What about `nbr` and `crossover`?

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.

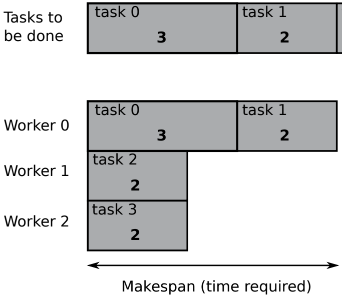


Task ID	0	1	2	3
Worker 0	1	1	0	0
Worker 1	0	0	1	0
Worker 2	0	0	0	1

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.



Task ID	0	1	2	3
Worker	0	0	1	2

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.

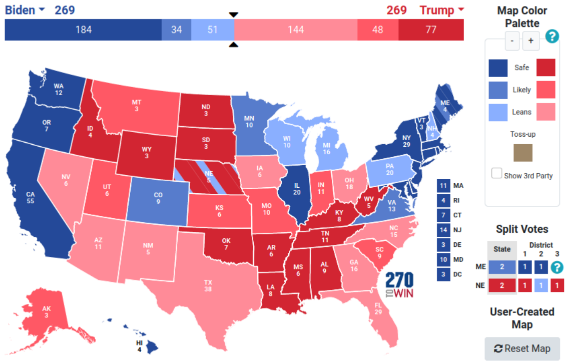
Set partitioning

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



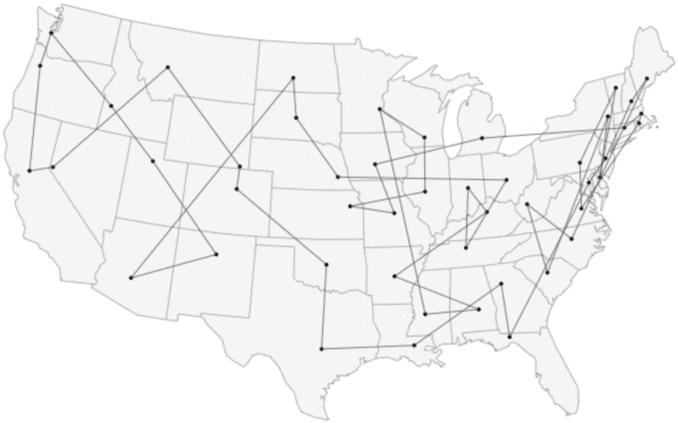
From 270towin.com

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**.

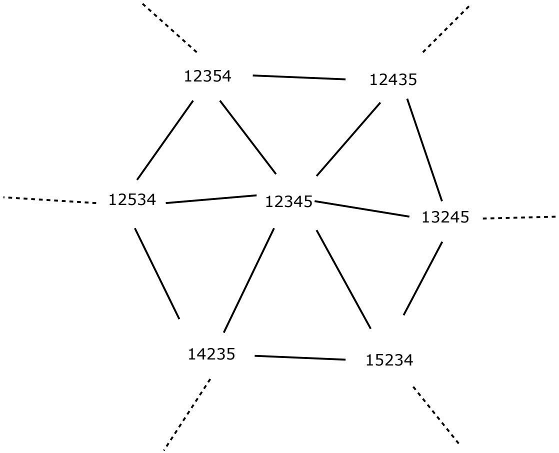
What is a permutation?

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.

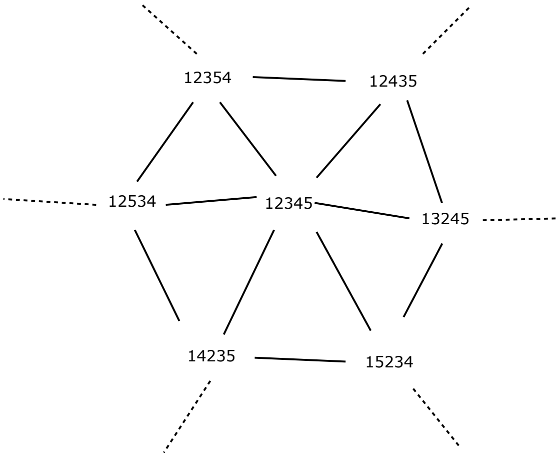
In some applications, permutations **wrap around**: the first item is considered to follow the last item, in the ordering. Because of this, 15243 and 43152 are considered to be the **same** permutation. We can define the **canonical** form as the form where 1 comes first.

Space of permutations



- Space of permutations of n items
- $X = \mathbb{P}^n$

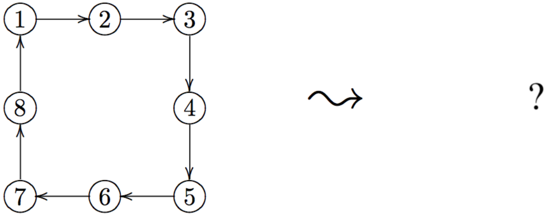
Space of permutations



- Space of permutations of n items
- $X = \mathbb{P}^n$

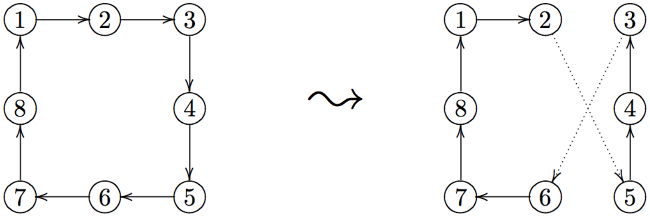
Why is a permutation different from a vector of n decision variables with integer values in $[1, n]$? What would happen if we used that for a TSP?

Neighbours in a permutation space



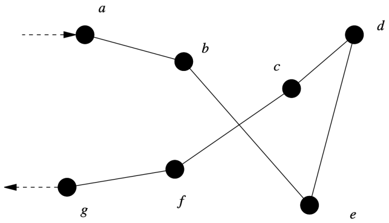
Suppose there are 8 cities to be visited. Suppose we have the solution $x = 12345678$ as shown. How could we define a **neighbour** function to give `nbr(x)`? What are the possible neighbours with that definition?

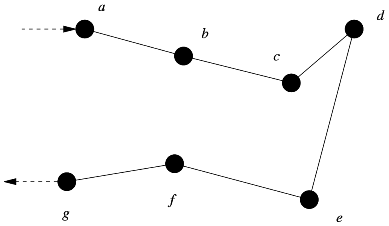
The 2-opt neighbour function



This function chooses a chunk of the permutation and **reverses it**. It’s called **2-opt**.

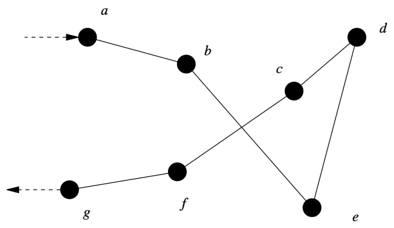
The 2-opt neighbour function

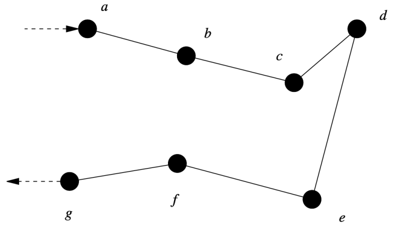




Wiki PierreSelim

The 2-opt neighbour function





Wiki PierreSelim

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

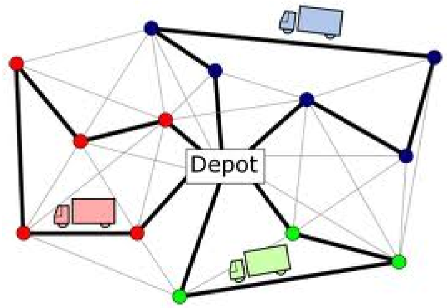
- Paint manufacturer has a recurring order for many colours
- All colours use the same equipment
- After manufacturing one colour, the equipment must be cleaned
- Some transitions take longer than others
- E.g. it takes longer to clean the equipment when going from black to yellow than vice versa
- Goal: minimise total cleaning time.

- Minimise **time** as opposed to distance

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}.

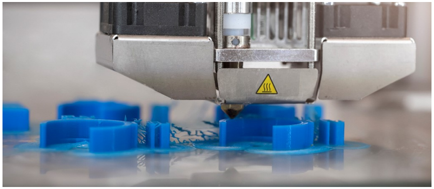
TSP applications: restaurant delivery



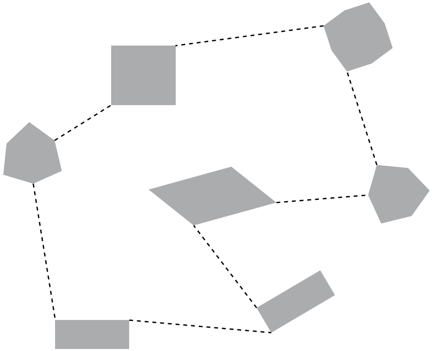
A food delivery company has *n* couriers and a stream of incoming orders. Each order is characterised by pick-up and drop-off points and constraints on time of delivery.

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 paint several **areas**. The time required for painting 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

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 $x = (x_0, x_1, \ldots, x_{n-1})$.

Some possible keyword bidding policies

- Bid EUR1
- Bid current keyword CTR multiplied by expected revenue per click multiplied by 0.5
- Bid CTR / 17 + number of previous bids that failed
- …?

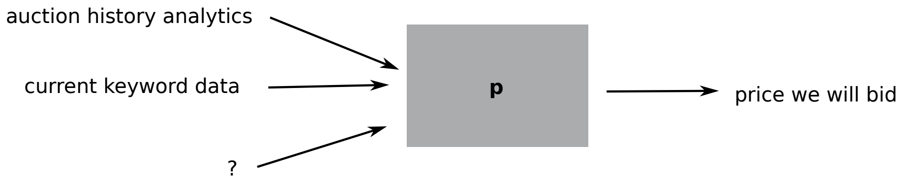
Some possible keyword bidding policies

- Bid EUR1
- Bid current keyword CTR multiplied by expected revenue per click multiplied by 0.5
- Bid CTR / 17 + number of previous bids that failed
- …?

There are many possible policies, some easy to understand, some impossible!

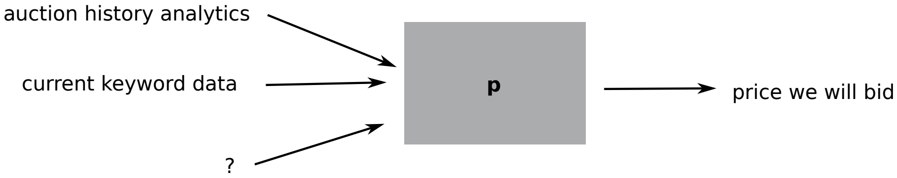
Keyword bidding policies

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.

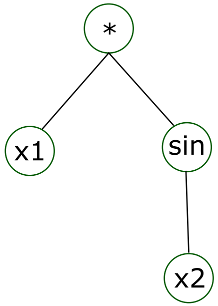


NB this program is **not** an objective function! (We will discuss the objective later.)

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



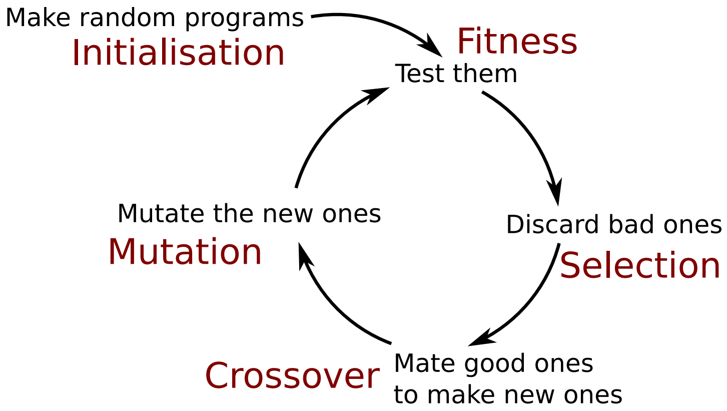
x1 sin(x2)

We should bid:

$$p(x) = x_1 \sin(x_2)$$

Genetic Programming

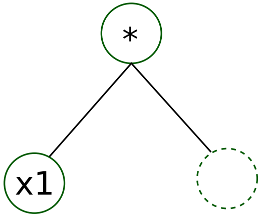
GP = program synthesis (or automated programming) by evolution



Initialisation on trees

The **grow** method:

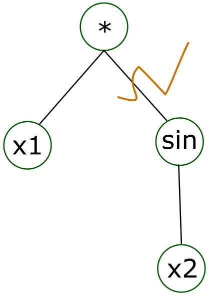
1. Start at the root
2. If we are at the maximum tree depth, choose a random **leaf** (e.g. x_i or a constant)
3. Else, choose a function (e.g. sin, $*$, etc.) or leaf at random
4. If we choose a function with $n > 0$ children, then for each one, create a “slot” below the current node, move to it, and recurse to 2; otherwise return.



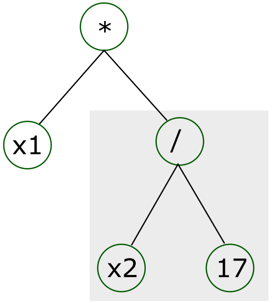
Mutation on trees

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.



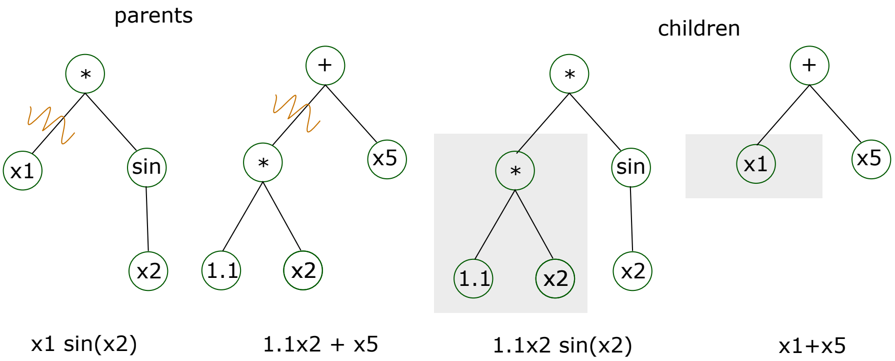
x1 sin(x2)



x1(x2/17)

Crossover on trees

Subtree crossover: choose a cutpoint randomly in each tree, and swap the subtrees below them.



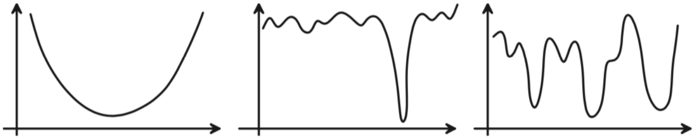
Fitness

- 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