If we have a hard optimization problem, we can try using an Evolutionary Algorithm (EA) to solve it. To start with we need:

a. An ‘encoding’, which is a way to represent any solution to the problem in a computational form (e.g. it could be a string of integers, a 2D array of real numbers, basically any datastructure ...)

b. A fitness function \( f(s) \), which works out a ‘fitness’ value for the solution encoded as ‘s’.

Once we have figured out a suitable encoding and fitness function, we can run an EA, which generally looks like this:

0. Initialise: generate a fixed-size population \( P \) of random solutions, and evaluate the fitness of each one.
1. Using a suitable selection method, select one or more from \( P \) to be the ‘parents’
2. Using suitable ‘genetic operators’, generate one or more children from the parents, and evaluate each of them
3. Using a suitable ‘population update’ or ‘reproduction’ method, replace one or more of the parents in \( P \) with some of the newly generated children.
4. If we are out of time, or have evolved a good enough solution, stop; otherwise go to step 1.

Note: one trip through steps 1,2,3 is called a generation

Some variations within this theme

Initialization: Indeed we could generate ‘random’ solutions – however, depending on the problem, there is often a simple way to generate ‘good’ solutions right from the start. Another approach, where your population size is set to 100 (say) is to generate 10,000 (say) random solutions at the start, and your first population is the best 100 of those.

Selection: lots of approaches (see mandatory additional material) – however it turns out that a rather simple selection method called ‘binary tournament selection’ tends to be as good as any other – see later slides.

Genetic Operators: Obviously, the details of these depends on the encoding. But in general there is always a choice between ‘mutation’ (generate one or more children from a single parent) and ‘recombination’ (generate one or more children from two or more parents). Often both types are used in the same EA.

Population update: the extremes are called: ‘steady state’ (in every generation, at most one new solution is added to the population, and one is removed) and ‘generational’ (in every generation, the entire population is replaced by children). Steady state is commonly used, but generational is better for parallel implementations.
Specific evolutionary algorithms

The algorithm whose pseudocode is on the next slide is a **steady state, replace-worst** EA with **tournament selection**, using mutation, but no crossover.

Parameters are \textit{popsize, tournament size, mutation-rate}.

It can be applied to any problem; the details glossed over are all problem-specific.

\textbf{A steady state, mutation-only, replace-worst EA with tournament selection}

0. Initialise: generate a population of \textit{popsize} random solutions, evaluate their fitnesses.

1. Run \textit{Select} to obtain a parent solution \(X\).
2. With probability \textit{mute_rate}, mutate a copy of \(X\) to obtain a mutant \(M\) (otherwise \(M = X\)).
3. Evaluate the fitness of \(M\).
4. Let \(W\) be the current worst in the population (BTR). If \(M\) is not less fit than \(W\), then replace \(W\) with \(M\). (otherwise do nothing)
5. If a termination condition is met (e.g. we have done 10,000 evaluations) then stop. Otherwise go to 1.

\textit{Select:} randomly choose \textit{tsize} individuals from the population. Let \(c\) be the one with best fitness (BTR); return \(X\).
A generational, elitist, crossover + mutation EA
with Rank-Based selection

0. Initialise: generate a population G of popsize random solutions, evaluate their fitnesses.

1. Run RankSelect \(2^{*}(\text{popsize} - 1)\) times to obtain a collection I of \(2^{*}(\text{popsize} - 1)\) parents.

2. Randomly pair up the parents in I (into \text{popsize} - 1 pairs) and apply \text{Vary} to produce a child from each pair. Let the set of children be C.

3. Evaluate the fitness of each child.

4. Keep the best in the population G (BTR) and delete the rest.

5. Add all the children to G.

6. If a termination condition is met (e.g. we have done 100 or more generations (runs through steps 1 -5) then stop. Otherwise go to 1,

continued....

A generational, elitist, crossover + mutation EA
with Rank-Based selection, continued ...

\textbf{RankSelect:} sort the contents of G from best to worst, assigning rank \text{popsize} to the best, \text{popsize}-1 to the next best, etc ..., and rank 1 to the worst.

The ranks sum to \(F = \text{popsize}*(\text{popsize}+1)/2\)

Associate a probability \(\text{Rank}_i/F\) with each individual \(i\).

Using these probabilities, choose one individual \(X\), and return \(X\).

\textbf{Vary:}

1. With probability \text{cross_rate}, do a crossover:
   
   i.e produce a child by applying a crossover operator to the two parents. Otherwise, let the child be a randomly chosen one of the parents.

2. Apply mutation to the child.

3. Return the mutated child.
Steady State / Generational / Elitist

We have seen these two extremes:

Steady state: population only changes slightly in each generation. (e.g. select 1 parent, produce 1 child, add that child to pop and remove the worst)

Generational: population changes completely in each generation. (select some [could still be 1] parent(s), produce popsize children, they become the next generation.

What’s commonly used are one of the following two:
Steady-state: as described above
Generational-with-Elitism.

‘elitism’ means that new generation always contains the best solution from the previous generation, the remaining \(\text{popsize-1}\) individuals being new children.

Selection and Variation

A **selection** method is a way to choose a parent from the population, in such a way that the fitter an individual is, the more likely it is to be selected.

A **variation operator** or **genetic operator** is any method that takes in a (set of) parent(s), and produces a new individual (called child). If the input is a single parent, it is called a **mutation operator**. If two parents, it is called **crossover** or **recombination**.
Population update

A population update method is a way to decide how to produce the next generation from the merged previous generation and children.

E.g. we might simply sort them in order of fitness, and take the best popsise of them. What else might we do instead?

Note: in the literature, population update is sometimes called ‘reproduction’ and sometimes called ‘replacement’.

Encodings

We want to evolve schedules, networks, routes, coffee percolators, drug designs – how do we encode or represent solutions?

How you encode dictates what your operators can be, and certain constraints that the operators must meet.

Next are four examples
A simple encoding for bin-packing

Suppose we have \( N \) items, and we have a maximum of \( B \) bins available,

A simple encoding is as follows:
- A solution is represented by a list of \( N \) integers, each of them being any number from 1 to \( B \) inclusive.
- The meaning of such a solution, like for example ‘3, 1, 2, 1, 2 …’
  is ‘the 1st item is in bin 3, the 2nd item is in bin 1, the 3rd item is in bin 2,
  the 4th item is in bin 1, the 5th item is in bin 2, [and so on …]’

So, in our simple example,

1,2,3,2,1 encodes this solution

Simple operators that work with this encoding

A **Mutation Operator**:
choose a gene at random, and change it to a random new value

So, in our simple example,

1,2,3,2,1 encodes this solution

Choose a gene at random: 1,2,3,2,1
Change it to a random new (but valid) value: 1,2,3,3,1

Our mutant is: 1, 2, 3, 3, 1
which encodes this:
Simple operators that work with this encoding

A Crossover Operator:
We can use ‘one-point crossover’. In this case, any solution has 5 ‘genes’, which means we have 6 crossover points:

```
1,2,3,2,1
```

← here is an example solution with the crossover points indicated

In one-point crossover, we select two parents (P1 and P2), we then choose a crossover point \( X \) at random. We then build a child by copying P1 up to \( X \), and then copying P2 from \( X \) onwards. E.g.

Crossover of \( 3,1,2,3,3 \) and \( 2,2,1,2,1 \) at 4th Xover point is: \( 3,1,2,2,1 \)

Crossover of \( 1,1,3,2,2 \) and \( 2,3,1,1,1 \) at 2\(^{nd} \) Xover point is: \( 1,1,1,1,1 \)

The TSP

The Travelling Salesperson Problem
You have \( N \) ‘cities’, and an associated distance matrix. Find the shortest tour through the cities. I.e., starting from one of the cities (‘A’, for example), what is the quickest way to visit all of the other cities, and end up back at ‘A’?

Many many real problems are based on this (vehicle routing probs, bus or train scheduling), or exactly this (drilling thousands of holes in a metal plate, or … an actual traveling Salesperson problem!).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
The TSP: encoding

The encoding most commonly used for the TSP is a *Permutation encoding* (also generally called an *order-based* encoding).

If we have *N* cities, 1, 2, 3, …, *N*, then any permutation of these *N* identifiers represents a solution.

*e.g.* if *N* is 8, then 3, 4, 2, 8, 7, 6, 1, 5 encodes a solution, but 3, 3, 2, 1, 7, 3, 2, 4 does not encode a solution – why?

In our previous example, we had cities A, B, C, D and E – so any permutation of these five letters represents a solution. *E.g.* ABDCE represents this:

![Diagram of cities A, B, C, D, E with connections]

The TSP: operators

The mutation operator we described for the bin packing encoding would not be valid in this case. Why?

One valid mutation operator would be *random pair swap*. *E.g.* choose two different gene positions at random, and swap their contents.

Suppose we have solution CEABD

![Diagram of cities A, B, C, D, E with connections]

Let’s randomly choose the 2\(^{nd}\) and 4\(^{th}\) genes and swap them.

We get CBAED, which is this solution:

![Diagram of cities A, B, C, D, E with connections]
The TSP: operators

The crossover operator we described for the bin packing encoding would not be valid in this case. Why?

In general, recombination operators for order-based encodings are slightly harder to design, but there are many that are commonly used.

Read the mandatory additional material ‘Operators’ lecture – this includes examples of commonly used mutation and recombination operators for order-based encodings.

Example real-number Encoding

Suppose we want to design the overall shape of a 2-phase jet-nozzle (this was the first ever EA application – mentioned in lecture 2)

We could do it by specifying a vector of 6 real numbers with the meaning indicated below:

D1, D2, D3, D4, D5, D6

Fixed at six diameters, five sections
mutation and crossover

what do you suggest?

See examples in The Operators
Mandatory additional material

Or, could use a binary encoding

Encoded solution: 01101001 00110101 01101000 00001000 100.

Mutation and crossover?
Back to Basics

With your thirst for seeing example EAs temporarily quenched, the story for the next lecture skips to simpler optimization algorithms.

1. Hillclimbing

2. Local Search

These have much in common with EAs, but with no population – they just use a single solution and keep trying to improve it.

HC and LS can be very effective algorithms when appropriately engineered. But by looking at them, we will discover certain limitations, and this leads us directly to algorithm design strategies that look like EAs. That is, we can ‘arrive’ at EAs from an algorithm design route, not just a ‘bio-inspiration’ route.

Simplest possible EA: Hillclimbing

0. Initialise: Generate a random solution \( c \); evaluate its fitness, \( f(c) \). Call \( c \) the current solution.

1. Mutate a copy of the current solution – call the mutant \( m \) Evaluate fitness of \( m \), \( f(m) \).

2. If \( f(m) \) is no worse than \( f(c) \), then replace \( c \) with \( m \), otherwise do nothing (effectively discarding \( m \)).

3. If a termination condition has been reached, stop. Otherwise, go to 1.

Note. No population (well, population of 1). This is a very simple version of an EA, although it has been around for much longer.
Why “Hillclimbing”?

Suppose that solutions are lined up along the $x$ axis, and that $mutation$ always gives you a nearby solutions. Fitness is on the $y$ axis; this is a $landscape$

1. Initial solution; 2. rejected mutant; 3. new current solution, 4. New current solution; 5. new current solution; 6. new current soln 7. Rejected mutant; 8. rejected mutant; 9. new current solution, 10. Rejected mutant, …

How will HC do on this $landscape$?
Neighbourhoods

Recall $S$, the search space, and $f(s)$, the fitness of a candidate in $S$.

Showing the neighbourhoods of two candidate solutions, assuming the mutation operator adds a random number between $-1$ and $1$.

Neighbourhoods

Recall $S$, the search space, and $f(s)$, the fitness of a candidate in $S$.

Showing the neighbourhood of a candidate solution, assuming the mutation operator adds a random integer between $-2$ and $2$. 
Neighbourhoods

Recall $S$, the search space, and $f(s)$, the fitness of a candidate in $S$

Showing the neighbourhood of a candidate solution, assuming
the mutation operator simply changes the solution to a new random
Number between 0 and 20

Neighbourhoods

Recall $S$, the search space, and $f(s)$, the fitness of a candidate in $S$

Showing the neighbourhood of a candidate solution, assuming
the mutation operator adds a Gaussian (ish) with mean zero.
Neighbourhoods

Let \( s \) be an individual in \( S \), \( f(s) \) is our fitness function, and \( M \) is our mutation operator, so that \( M(s1) \rightarrow s2 \), where \( s2 \) is a mutant of \( s1 \).

Given \( M \), we can usually work out the \textit{neighbourhood} of an individual point \( s \) – the neighbourhood of \( s \) is the set of all possible mutants of \( s \).

\begin{center}
\begin{tabular}{ | l | }
\hline
\textbf{Encoding:} & permutations of \( k \) objects (e.g. for \( k \)-city TSP) \\
\textbf{Mutation:} & swap any adjacent pair of objects. \\
\textbf{Neighbourhood:} & Each individual has \( k \) neighbours. E.g. \\
 & neighbours of EABDC are: \{AEBDC, EBADC, EADBC, EABCD, CABDE\} \\
\hline
\textbf{Encoding:} & binary strings of length \( L \) (e.g. \( L \)-item 2-bin-packing) \\
\textbf{Mutation:} & choose a bit randomly and flip it. \\
\textbf{Neighbourhood:} & Each individual has \( L \) neighbours. E.g. \\
 & neighbours of 00110 are: \{10110, 01110, 00010, 00100, 00111\} \\
\hline
\end{tabular}
\end{center}

Typical Classes of Landscapes

- Unimodal
- Plateau
- Multimodal
- Deceptive
Beyond Hillclimbing

HC clearly has problems with typical landscapes:

There are two broad ways to improve HC, from the algorithm viewpoint:

1. Allow downhill moves – a family of methods called Local Search does this in various ways.
2. Have a population – so that different regions can be explored inherently in parallel – i.e. we keep ‘poor’ solutions around and give them a chance to ‘develop’.

Local Search

<table>
<thead>
<tr>
<th>Initialise: Generate a random solution c; evaluate its fitness, ( f(s) = b ); call c the current solution, and call b the best so far.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat until termination condition reached:</td>
</tr>
<tr>
<td>1. Search the neighbourhood of c, and choose one, m</td>
</tr>
<tr>
<td>Evaluate fitness of m, call that x.</td>
</tr>
<tr>
<td>2. According to some policy, maybe replace c with x, and update c and b as appropriate.</td>
</tr>
</tbody>
</table>

E.g. Monte Carlo search: 1. same as hillclimbing; 2. If x is better, accept it as new current solution; if x is worse, accept it with some probability (e.g. 0.1).

E.g. tabu search: 1. evaluate all immediate neighbours of c
                  2. choose the best from (1) to be the next current solution, unless it is ‘tabu’ (recently visited), in which choose the next best, etc.
Population-Based Search

- Local search is fine, but tends to get stuck in local optima, less so than HC, but it still gets stuck.
- In PBS, we no longer have a single `current solution’, we now have a population of them. This leads directly to the two main algorithmic differences between PBS and LS
  - Which of the set of current solutions do we mutate? We need a selection method
  - With more than one solution available, we needn’t just mutate, we can [recombine, crossover, etc …] two or more current solutions.
- So this is an alternative route towards motivating our nature-inspired EAs – and also starts to explain why they turn out to be so good.

Lecture 5:

Why does flocking/swarming occur so much in nature?

Energy savings: Geese in V formation have around a 70% greater range than in flying individually. Individuals can fly around 25% faster (why?).

Frightening and confusing predators; avoiding being “picked off”

Helping to catch prey: e.g. tuna school in a crescent shaped flock with the concave part forward:
This is thought to help channel their prey to the “focus”, and stop them from escaping
It comes down to: **simple behaviours of individuals in a group can have useful emergent properties.**
A theme we will continue to see a lot …

**The Adaptive Culture Model**

Robert Axelrod has a well-known theory, “Axelrod’s Culture Model”, which explains how ideas spread in societies. Kennedy and Eberhart (a computer scientist and a social scientist respectively) altered this into the “Adaptive Culture Model”, which works like this:

**If you think your neighbour is good, then be more like them.**

More in the PSO lecture, but that’s basically it. Notice the important words,

- **neighbour:** you change yourself under the influence of people nearby
- **good:** in some way your neighbour is more optimal than you, otherwise why be like them?
- **more like:** this is vague, so you have freedom in how you change

This is actually a very good model for how culture and ideas spread quickly in societies. Everything from rumours to eating habits. I only hope this works with ‘green’ behaviour …

**Back to computer science …**

From the CS viewpoint, one of the interesting aspects of this is that organised, functional group behaviour emerges, without the need for a central controller (or without a single ‘brain’).

The **emergent behaviour** that we see arises purely as a result of individuals in the swarm processing information in their (fairly) immediate neighbourhood.

So, studying this in nature suggests how we can get co-ordinated behaviour from a group of individuals, without having to specify any overall controller. This is very useful, for example, for designing computer networks. If one main machine was in control of the network, and that machine crashed, …

But so far that has not been a main success area for swarm inspiration …
Two main things that come from swarm inspiration:

Optimisation algorithms.
Ants seem to find the shortest path to find food that may be quite distant from their nest. They do this via “stigmergy” – laying pheromonones on their path as they move. This has directly inspired the design of a very successful optimisation method, called Ant Colony Optimisation.

Meanwhile, the adaptive culture model has led to a different, and also very successful, new optimisation algorithm, called Particle Swarm Optimisation.

Simulations of natural flocks.
For the entertainment and gaming industries, for example.

One other thing that come from swarm inspiration:

Swarm-based construction
Not yet applied much, but soon to be: we are working on it! Swarm-based construction – how ants build their nests, bees build hives, and beavers build dams, etc – seems to be explainable by sets of simple rules that make use of stigmergy (as with other emergent behaviours). But in this case, the rules are about where the individual should put things, rather than where the individual should go.

See section 2.1.3 of the recommended reading “Swarm Intelligence Chapter” on my teaching site.
Rule 1: Separation

At each iteration, a boid makes an adjustment to its velocity according to the following rule:

Avoid getting too close to local (the ones it is aware of) flockmates.

Rule 2: Alignment

At each iteration, a boid makes an adjustment to match its velocity to the average of that of its local flockmates.

Rule 3: Cohesion

At each iteration, a boid makes an adjustment to its velocity towards the centroid of its flockmates.
Swarm Algorithms

Inspiration from swarm intelligence has led to some highly successful optimisation algorithms.

- Ant Colony (-based) Optimisation – a way to solve optimisation problems based on the way that ants indirectly communicate directions to each other.

- Particle Swarm Optimisation — a different way to solve optimisation problems, based on the swarming behaviour of several kinds of organisms.

This lecture is about Ant Colony Optimisation (PSO next week)

A key concept: Stigmergy

Stigmergy is:
indirect communication via interaction with the environment [Gassé, 59]

- A problem gets solved bit by bit ..
- Individuals communicate with each other in the above way, affecting what each other does on the task.
  
  E.g. sorting things into piles, as we did in the Introductory Swarm Intelligence lecture

- Individuals leave markers or messages – these don’t solve the problem in themselves, but they affect other individuals in a way that helps them solve the problem …

- E.g. as we will see, this is how ants find shortest paths.
Stigmergy in Ants

Ants are behaviorally unsophisticated, but collectively they can perform complex tasks.

Ants have *highly developed sophisticated sign-based stigmergy*
- They communicate using pheromones;
- They lay *trails of pheromone* that can be followed by other ants.

• If an ant has a *choice of two pheromone trails* to follow, one to the NW, one to the NE, but the NW one is *stronger* – which one will it follow?

**Pheromone Trails**

Individual ants lay pheromone trails while travelling from the nest, to the nest or possibly in both directions.
The pheromone trail gradually evaporates over time. But pheromone trail strength accumulate with multiple ants using path.
Ant Colony Optimisation
Algorithms: Basic Ideas

Ants are agents that:
Move along between nodes in a graph.

They choose where to go based on pheromone strength (and maybe other things)

An ant’s path represents a specific candidate solution.

When an ant has finished a solution, pheromone is laid on its path, according to quality of solution.

This pheromone trail affects behaviour of other ants by ‘stigmergy’…

The ACO algorithm for the TSP
[a simplified version with all essential details]

We have a TSP, with \( n \) cities.

1. We place some ants at each city. Each ant then does this:
   - It makes a complete tour of the cities, coming back to its starting city, using a transition rule to decide which links to follow. By this rule, it chooses each next-city at random, but biased partly by the pheromone levels existing at each path, and biased partly by heuristic information.

2. When all ants have completed their tours.

   **Global Pheromone Updating occurs.**
   - The current pheromone levels on all links are reduced (i.e. pheromone levels decay over time).
   - Pheromone is lain (belatedly) by each ant as follows: it places pheromone on all links of its tour, with strength depending on how good the tour was.

Then we go back to 1 and repeat the whole process many times, until we reach a termination criterion.
A very common variation, which gives the best results

We have a TSP, with $n$ cities.
1. We place some ants at each city. Each ant then does this:
   - It makes a complete tour of the cities, coming back to its starting city, using a transition rule to decide which links to follow. By this rule, it chooses each next-city at random, but biased partly by the pheromone levels existing at each path, and biased partly by heuristic information.

2. When all ants have completed their tours.

Apply some iterations of LOCAL SEARCH to the completed tour; this finds a better solution, which is now treated as the ant’s path. Then continue the next steps as normal.

Global Pheromone Updating occurs.
   - The current pheromone levels on all links are reduced (i.e. pheromone levels decay over time).
   - Pheromone is lain (belatedly) by each ant as follows: it places pheromone on all links of its tour, with strength depending on how good the tour was.

Then we go back to 1 and repeat the whole process many times, until we reach a termination criterion.

The transition rule

$T(r, s)$ is the amount of pheromone currently on the path that goes directly from city $r$ to city $s$.
$H(r, s)$ is the heuristic value of this link – in the classic TSP application, this is chosen to be $1/distance(r, s)$ -- i.e. the shorter the distance, the higher the heuristic value.

$p_k(r, s)$ is the probability that ant $k$ will choose the link that goes from $r$ to $s$.

$\beta$ is a parameter that we can call the heuristic strength.

The rule is:

$$p_k(r, s) = \frac{T(r, s) \cdot H(r, s)^\beta}{\sum_{\text{unvisited cities } c} T(r, c) \cdot H(r, c)^\beta}$$

Where our ant is at city $r$, and $s$ is a city as yet unvisited on its tour, and the summation is over all of $k$’s unvisited cities.
Global pheromone update

$A_k(r, s)$ is the amount of pheromone added to the $(r, s)$ link by ant $k$.

$m$ is the number of ants.

$\rho$ is a parameter called the pheromone decay rate.

$L_k$ is the length of the tour completed by ant $k$.

$T(r, s)$ at the next iteration becomes:

$$\rho \cdot T(r, s) + \sum_{k=1}^{m} A_k(r, s)$$

Where $A_k(r, s) = 1/L_k$

Study these – they’re not that hard.
How do you think the parameters $m, \beta, \rho$ etc … affect the search?

Not just for TSP of course

ACO is naturally applicable to any sequencing problem, or indeed any problem.
All you need is some way to represent solutions to the problem as paths in a network.
Encoding / Representation

**Maybe the main** issue in (applying) EC

Note that:

- Given an optimisation problem to solve, we need to find a way of encoding candidate solutions
- There can be many very different encodings for the same problem
- Each way affects the shape of the landscape and the choice of best strategy for climbing that landscape.

### Direct encoding

<table>
<thead>
<tr>
<th></th>
<th>mon</th>
<th>tue</th>
<th>wed</th>
<th>thur</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>E4, E5</td>
<td></td>
<td></td>
<td>E3, E7</td>
</tr>
<tr>
<td>11:00</td>
<td>E8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:00</td>
<td>E1, E6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are clashes at Mon 9am and Mon 4pm

### Indirect encoding

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tue</th>
<th>wed</th>
<th>thur</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>E4</td>
<td></td>
<td></td>
<td>E3, E7</td>
</tr>
<tr>
<td>11:00</td>
<td>E5</td>
<td></td>
<td>E2, E6</td>
<td></td>
</tr>
<tr>
<td>2:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:00</td>
<td>E1, E8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No clashes, and good slot utilisation …
The *genotype* (the data structure that encodes a solution) is basically **the same as the** *Phenotype* (the solution). There is a simple and ‘direct’ mapping between them.

* This can help with design and understanding of operators and landscapes. You basically know what type of effect a mutation will have,

**But:**
* often means very bad solutions can be easily generated. Lots of ‘dross’ to evolve past.

The *genotype* provides the parameters of and/or inputs to an algorithm that builds the *Phenotype* (the solution). There is a very ‘indirect’ mapping between them.

* This means small changes in the genotype could have huge effects on the phenotype, that are difficult to understand in advance

**But:**
* often means even the initial ‘random’ solutions are quite good, and the evolution process focuses only on a much smaller search space of good candidates.

**Hyper-heuristics: a specific type of indirect encoding ...**

**Indirect encoding:**

- Don’t encode a candidate solution directly
- ... instead encode parameters/features for a **constructive algorithm** that builds a candidate solution

**Hyperheuristic encoding:**

- Usually, this is an encoding where each ‘gene’ is a constructive algorithm (usually a simple one) which builds one part of the solution
Example using ‘First-Fit Ascending’ (FFA) constructive algorithm for bin-packing

**FFA**

FFA means we pack them one by one,
Each time taking the *smallest unpacked item*

---

Example using First-Fit Descending

**First-fit Descending**

FFA means we pack them one by one,
Each time taking the *largest unpacked item*
Other bin packing heuristics ...

FFA - pack smallest remaining into 1st available bin it will fit in
FFD - pack largest remaining into 1st available bin it will fit in
BF - 1. consider all gaps in partially full bins; find an item that exactly fills a gap. 2. If no such item currently available, do FFA
BF(2) - 1. consider all gaps in partially full bins; find an item that exactly fills a gap. 2. If no such item, find a pair of items that will together exactly fill a gap. 3. Otherwise, do FFA
FFAH(p) - if the number of Huge items remaining is < p%, use FFA, otherwise use BF
FFAS(p) - if the number of Small items remaining is < p%, use FFD, otherwise use BF(2)

... Etc ... many other possibilities

“Huge” (item is >= 2/3 of bin capacity)
“small” (item is < 1/3 of bin capacity)

hyper-heuristics

... is a research area which is based on the idea of using indirect encodings for logistics problems (mainly), where the genotype is (often) a list of heuristics
Used for: timetabling, scheduling, bin-packing, stock-cutting, vehicle routing, etc ...

Note at least two main kinds of indirect encoding that make use of ‘heuristics’.
Consider timetabling:

A. 1, 2, 4, 1, 7, 6 ... Use H1 with parameter 1, use H1 with parameter 2, use H1 with parameter 4, use H1 with parameter 1, etc ... (our indirect encoding for timetabling was like this)

B. 1, 2, 4, 1, 7, 6 ... Use H1, then use H2, then use H4, then use H1, ...
   (a ‘hyper-heuristic’ encoding – see next 2 slides)
   ... and of course there are approaches that mix up styles A and B

It gets called a ‘hyper-heuristic’ approach if there is a strong ‘type B’ element.
Example simple hyper-heuristic encoding for bin packing

weights: 2 1 3 1.5 0.5 3.5 4 4.5 bin capacity: 5
encoding

1 = FFA, 2 = FFD, 3 = BF, 4 = BF(2)
Genotype is a list of 7 heuristic choices, (why 7?),
each either 1, 2, 3 or 4
Let’s interpret: 3, 2, 4, 2, 4, 1, 0, 1 only one item left, use FFD

Notes: constructive algorithms

- Each heuristic we have seen is a simple constructive algorithm – it performs the next step in constructing a solution to the problem
- There are many constructive heuristics for bin packing, timetabling, scheduling, etc .... often called ‘dispatch heuristics’
- There are well-known constructive heuristics for many other problems, e.g. Prim’s algorithm for building the minimal spanning tree (see an earlier lecture), Djikstra’s shortest path algorithm for networks, etc...
- Suitably engineered, Prim’s algorithm can be used as part of an indirect encoding for solving hard constrained spanning-tree problems --- similar for Djikstra’s algorithm and network-path problems.
- And so on ...
Direct vs Indirect Encodings: summary & other notes

Direct:
• straightforward genotype (encoding) $\rightarrow$ phenotype (actual solution) mapping
• Easy to estimate effects of mutation
• Fast interpretation of chromosome (hence speedier fitness evaluation)

Indirect/Hybrid:
• Easier to exploit domain knowledge – (e.g. use this in the constructive heuristic)
• Hence, possible to ‘encode away’ undesirable features
• Hence, can seriously cut down the size of the search space
• But, interpretation of genotype could become computationally expensive
• Neighbourhoods tend to be highly rugged.
• $\rightarrow$ tradeoff between size of search space and ruggedness
  - sensible engineering of the indirect encoding often beats the direct
  - however, direct encodings with smarts built-in to the operators can be the best choice … that’s for further reading or an advanced module…

Lecture 8:
Cooperation example

To illustrate what “cooperation” means in PSO, here is a simplistic example.
As usually, the big fish is difficult to catch, hidden in the deepest part of the pond. At each time step, each fisherman tells to the other how deep the pond is at his place. At the very beginning, as the depths are quite similar, they both follow their own ways. Now, Fisherman 2 seems to be on a better place, so Fisherman 1 tends to go towards him quite rapidly. Now, the decision is a bit more difficult to make. On the one hand Fisherman 2 is still on a better place, but on the other hand, Fisherman 1’s position is worse than before. So Fisherman 1 comes to a compromise: he still goes towards Fisherman 2, but more slowly than before. As we can see, doing that, he escapes from the local minimum.

Of course, this example is a caricatural one, but it presents the main features of a particle in basic PSO: a position, a velocity (or, more precisely an operator which can be applied to a position in order to modify it), the ability to exchange information with its neighbours, the ability to memorize a previous position, and the ability to use information to make a decision. Remember, though, all that as to remain simple.

Let's now see more precisely these points.
Here you have another nice search space. First step: you put some particles on it. You can do it at random or on a regular way, or both. How many? In practice, for most real problems with

The basic idea II

1. The particles in the swarm *co-operate*. They exchange information about what they’ve discovered in the places they have visited.

2. The co-operation is very simple. In basic PSO it is like this:
   - A particle has a *neighbourhood* associated with it.
   - A particle knows the fitnesses of those in its neighbourhood, and uses the *position* of the one with best fitness.
   - This position is simply used to adjust the particle’s velocity.

Initialization. Positions and velocities
dimension between 2 and 100, a swarm size of 20 particles works quite good.
There are some mathematical ways to give an estimation, but a bit beyond the scope of this talk. Also, as we will see some variants use an adaptive swarm size.
Second step: you define a velocity for each particle, usually at random. You can set all initial velocities to zero but, experimentally, it is usually not the best choice.
Remember that what we call “velocity” is in fact a move, just because time is discretized.

What a particle does

- In each timestep, a particle has to move to a new position. It does this by adjusting its velocity.
  - The adjustment is essentially this:
    - The current velocity PLUS
    - A weighted random portion in the direction of its personal best PLUS
    - A weighted random portion in the direction of the neighbourhood best.
- Having worked out a new velocity, its position is simply its old position plus the new velocity.
Now, for each particle, we define what is called a *neighbourhood*. Although some variants use a “geographical” neighbourhood, that is to say compute distances and take the nearest particles, the most widely used neighbourhood is a “social” one: just a list of neighbours, regardless *where* they are.

So, you do not need to define a distance and that is a great advantage, for in some cases, particularly for discrete spaces, such a definition would be quite arbitrary.

Note that it can be proved (and it is intuitively quite obvious) that if the process converges any social neighbourhood tends to be also a geographical one.

Usually, in practice, social neighbourhoods are defined just once, at the very beginning, which is consistent with the principle "simple rules for simple agents".

Now, the size of the neighbourhood could be a problem. Fortunately, PSO is not very sensitive to this parameter and most of users just take a value of 3 or 5 with good results.

Unlike for the swarm size, there is no mathematical formula, but *like* for the swarm size, there are some adaptive variants.
Now, for each particle, we define what is called a *neighbourhood*. Although some variants use a “geographical” neighbourhood, that is to say compute distances and take the nearest particles, the most widely used neighbourhood is a “social” one: just a list of neighbours, regardless *where* they are.

So, you do not need to define a distance and that is a great advantage, for in some cases, particularly for discrete spaces, such a definition would be quite arbitrary.

Note that it can be proved (and it is intuitively quite obvious) that if the process converges any social neighbourhood tends to be also a geographical one.

Usually, in practice, social neighbourhoods are defined just once, at the very beginning, which is consistent with the principle “simple rules for simple agents”.

Now, the size of the neighbourhood could be a problem. Fortunately, PSO is not very sensitive to this parameter and most of users just take a value of 3 or 5 with good results.

Unlike for the swarm size, there is no mathematical formula, but *like* for the swarm size, there are some adaptive variants.
The most commonly used neighbourhood is the circular one. The picture is almost self explanatory. Each particle is numbered, put on a virtual circle according to its number and the neighbourhood of a given particle is built by taking its neighbours on this circle. An important point for rule simplicity is that each particle belongs to its neighbourhood. For example if a rule says “I have to check all my neighbours”, there is no need to add “and I have to check myself”. We will see that more precisely later.
This may be the most important slide of this presentation, for it summarizes the core of the method. Let’s take a bit time to comment it. You are a particle. Sorry, I don’t mean you are quite stupid, but it is just to explain how it works. (By the way, Jim Kennedy has designed a nice game in which you compete with such stupid particles. I have it here, and if we have time, you will see it is almost impossible to beat it.)

You can compute how good is your position (that is to say you can compute the objective function at the place you are). You remember the best position you ever found (and the objective function value). You can ask your neighbours for this information they also have memorized, and choose the best one.

Now, you have three tendancies,
- audacious, following your own way (just using your own velocity)
- conservative, going back more or less towards your best previous position
- sheeplike, going more or less towards your best neighbour

What PSO formalizes is how to combine these tendancies in order to be globally efficient.
Pseudocode

\[ \text{Equation (a)} \]
\[ v[] = c_0 * v[] + c_1 * \text{rand()} * (p\text{best}[] - \text{present}[]) + c_2 * \text{rand()} * (g\text{best}[] - \text{present}[]) \]
(in the original method, \(c_0=1\), but many researchers now play with this parameter)

\[ \text{Equation (b)} \]
\[ \text{present}[] = \text{present}[] + v[] \]

Particle Swarm optimisation

Pseudocode

http://www.swarmintelligence.org/tutorials.php

For each particle
   Initialize particle
END

Do
   For each particle
      Calculate fitness value
      If the fitness value is better than its personal best
         set current value as the new \textbf{pBest}
   End

Choose the particle with the best fitness value of all as \textbf{gBest}
For each particle
   Calculate particle velocity according equation (a)
   Update particle position according equation (b)
End
While maximum iterations or minimum error criteria is not attained
Particles' velocities on each dimension are clamped to a maximum velocity $V_{\text{max}}$. If the sum of accelerations would cause the velocity on that dimension to exceed $V_{\text{max}}$, which is a parameter specified by the user. Then the velocity on that dimension is limited to $V_{\text{max}}$.

**Parameters**

- ✔ Number of particles (swarmsize)
- ✔ $C_1$ (importance of personal best)
- ✔ $C_2$ (importance of neighbourhood best)
- ✔ $V_{\text{max}}$: limit on velocity
How to choose parameters?

I am sorry but in this part we have to do a bit more maths. What I call here “parameters” are in fact just the coefficients in the formula which indicates how to update the velocity of a given particle. Swarm size and neighbourhood size are also parameters but less important. I mean you can perfectly use a swarm size of 20 and a neighbourhood of 3 for a large range of problems with good results. On the contrary, PSO is more sensitive to the parameters we examine now.

Parameters

- Number of particles (10—50) are reported as usually sufficient.
- C1 (importance of personal best)
- C2 (importance of neighbourhood best)
- Usually C1+C2 = 4. No good reason other than empiricism
- Vmax – too low, too slow; too high, too unstable.
Adaptive swarm size

There has been enough improvement although I'm the worst.

I try to kill myself

I'm the best but there has been not enough improvement

I try to generate a new particle

Adaptive coefficients

$\alpha_v \quad \text{rand}(0...b)(p-x)$

The better I am, the more I follow my own way

The better is my best neighbour, the more I tend to go towards him

Additional Material Easy/Hard problems:
Search and Optimisation

Imagine we have 4 items as follows:

(item 1: 20kg; item2: 75kg; item 3: 60kg, item4: 35kg)

Suppose we want to find the subset of items with highest total weight

<table>
<thead>
<tr>
<th>0000</th>
<th>0100</th>
<th>1000</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0101</td>
<td>1001</td>
<td>1101</td>
</tr>
<tr>
<td>0010</td>
<td>0110</td>
<td>1010</td>
<td>1110</td>
</tr>
<tr>
<td>0011</td>
<td>0111</td>
<td>1011</td>
<td>1111</td>
</tr>
</tbody>
</table>

Here is a standard treatment of this as an optimisation problem.
The set $S$ of all possible solutions is indicated above.
The fitness of a solution $s$ is the function $\text{total_weight}(s)$
We can solve the problem (i.e. find the fittest $s$) by working out the
fitness of each one in turn. But … any thoughts? I mean, how hard is
this?

Search and Optimisation

In general, optimisation means that you are trying to find
the best solution you can (usually in a short time) to a
given problem.

We always have a set $S$ of all possible solutions

In realistic problems, $S$ is too large to search
one by one. So we need to find some other way
to search through $S$.

One way is random search. E.g. in a 500-iteration
random search, we might randomly choose something
in $S$ and evaluate its fitness, repeating that 500 times.
The Fitness function

Every candidate solution \( s \) in the set of all solutions \( S \) can be given a score, or a “fitness”, by a so-called fitness function. We usually write \( f(s) \) to indicate the fitness of solution \( s \). Obviously, we want to find the \( s \) in \( S \) which has the best score.

Examples

- timetabling: \( f \) could be no. of clashes.
- wing design: \( f \) could be aerodynamic drag
- delivery schedule \( f \) will be total distance travelled

An Aside about Classification

In a classification problem, we have a set of things to classify, and a number of possible classes.

To classify \( s \) we use an algorithm called a classifier. So, classifier(s) gives us a class label for \( s \).

We can assign a fitness value to a classifier – this can be simply the percentage of examples it gets right.

In finding a good classifier, we are solving the following optimisation problem: Search a space of classifiers, and find the one that gives the best accuracy.

E.g. the classifier might be a neural network, and we may use an EA to evolve the NN with the best connection weights.
Searching through $S$

When $S$ is small (e.g. 10, 100, or only 1,000,000 or so items), we can simply do so-called exhaustive search.

**Exhaustive search**: Generate every possible solution, work out its fitness, and hence discover which is best (or which set share the best fitness).

This is also called *Enumeration*

However ... 

*In all interesting/important cases, $S$ is much much too large for exhaustive search (ever).*

There are two kinds of `too-big` problem:

- **easy** (or `tractable`, or `in $P$`)
- **hard** (or `intractable`, or `not known to be in $P$`)

There are rigorous mathematical definitions of the two types — that’s what the ‘P’ is about — it stands for the class of problems that can be solved by a deterministic polynomial Turing machine. That’s outside the scope of this module.

But, what’s important for you to know is that *almost all important problems are technically hard.*
About Optimisation Problems

To **solve** a problem means to find an *optimal* solution. I.e. to deliver an element of \( S \) whose fitness is *guaranteed* to be the best in \( S \).

An *Exact algorithm* is one which can do this (i.e. solve a problem, guaranteeing to find the best).

Is 500-iteration random search an Exact algorithm?

**Problem complexity**

‘Problem complexity’, in the context of computer science, is all about characterising how hard it is to solve a given problem. Statements are made in terms of functions of \( n \), which is meant to be some indication of the size of the problem. E.g.:

- Correctly sort a set of \( n \) numbers
  - Can be done in around \( n \log n \) steps

- Find the closest pair out of \( n \) vectors
  - Can be done in \( O(n^2) \) steps

- Find best design for an \( n \)-structural-element bridge
  - Can be done in \( O(10^n) \) steps...
Polynomial and Exponential Complexity

Given some problem $Q$, with `size' $n$, imagine that $A$ is the fastest algorithm known for solving that problem exactly. The complexity of problem $Q$ is the time it takes $A$ to solve it, as a function of $n$.

There are two key kinds of complexity:

**Polynomial:** the dominant term in the expression is polynomial in $n$. E.g. $n^3$, $n \log n$, $\sin(n^{2.5})$, etc ...

**Exponential:** the dominant term is exponential in $n$. E.g. $1.1^n$, $n^n$, $2^n$, ...

Hard and Easy Problems

**Polynomial Complexity:** these are called *tractable*, and *easy* problems. Fast algorithms are known which provide the best solution. Pairwise alignment is one such problem. Sorting is another.

**Exponential Complexity:** these are called *intractable*, and *hard* problems. The fastest known algorithm which *exactly* solves it is usually not significantly faster than exhaustive search.
The problem **find the minimal cost spanning tree** (aka the `MST`) is **easy** in the technical sense.

Several fast algorithms are known which solve this in polynomial time:
Here is the classic one: Prim’s algorithm:

- **Start with empty tree (no edges)***
- **Repeat:** choose cheapest edge which feasibly extends the tree
- **Until:** \( n - 1 \) edges have been chosen.

But change the problem slightly:

<table>
<thead>
<tr>
<th>We may want the <strong>degree</strong> constrained – MST (I.e. the MST, but where no node in the tree has a degree above 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or we may want the optimal communication spanning tree – which is the MST, but constrained among those trees which satisfy certain bandwidth requirements between certain pairs of nodes</td>
</tr>
</tbody>
</table>

There are many constrained/different forms of the MST. These are essentially problems where we seek the cheapest tree structure, but where many, or even most, trees are not actually feasible solutions.

**Here’s the thing:** These constrained versions are almost always technically hard. and Real-world MST-style problems are invariably of this kind.
Approximate Algorithms

For **hard** optimisation problems (again, which turns out to be nearly all the important ones), we need *Approximate algorithms*.

These:

- deliver solutions in reasonable time
- try to find pretty good ("near optimal") solutions, and often get optimal ones.
- do not (cannot) guarantee that they have delivered the optimal solution.

Typical Performance of Approximate Methods

Evolutionary Algorithms turn out to be the most successful and generally useful approximate algorithms around. They often take a long time though – it’s worth getting used to the following curve which tends to apply across the board.

Additional “more encodings”: 
Crossover – some issues

Consider our direct encoding for timetabling: Suppose this is a perfect solution with no clashes:

Parent1:  1, 2, 5, 2, 2, 5, 2, 1, 2
         And so is this:
Parent2:  3, 4, 2, 4, 2, 4, 3, 4
Consider a two-point crossover of them, such as:
Child:   1, 2, 5, 4, 4, 2, 4, 3, 2
Would you expect this to be a perfect solution?

---------------

Probably not; let’s look at the parents in terms of the groupings into slots:

Parent:  slot1 (e1, e5); slot2 (e2, e4, e6, e7, e9); slot3 (e3, e8)
Parent:  slot1 (e1, e5); slot2 (e2, e4, e6, e7, e9); slot3 (e3, e8)

These parents are exactly the same in terms of the way exams are grouped together, and this is probably what accounts for their good fitness. I.e. it is a good idea to have e2, e4, e5, e7 and e9 in the same slot, etc.

Child:  slot1 (e1), slot2 (e2, e6, e9); slot4 (e4, e5, e7); slot5 (e3)

Our use of a standard ‘k-ary encoding’ crossover operator has disrupted these groupings.

Grouping

Falkenauer (see paper on my teaching site – this one is examinable reading for the MScs, and recommended for the UGs) was the first to come up with a highly ‘tailored’ approach to applying an EA, in this case to the bin-packing problem. He used specialised initialisation, encoding, mutation, crossover, and fitness evaluation methods. His bin-packing work is generally a good example of how to design an EA so it works as well as it can on a particular problem. Of interest here is the encoding he used combined with the crossover operator – this type of encoding/operator combination has become common in cases where the problem at hand involves finding good ‘groups’ of some kind or other.
Group Based Encoding and Crossover

Simplified from Falkenauer, a group-based encoding is simply a direct encoding of groups. E.g. for bin-packing, where we are trying to minimise the number of bins, and have 9 items, two chromosomes might be:

P1: (3, 8, 2) - (1, 4) - (6, 7, 9) - (5)
P2: (1, 6, 7, 9) - (3, 8, 2) - (4, 5)

The chromosomes are simply appropriate data structures that can hold a variable number of groups. The ordering of items within groups doesn’t matter.

Notice that the underlying encoding can just be the direct one.
The only really key point is that the crossover operator should work in the way described next.

Group Based Crossover

Take two parents:
P1: (2, 5, 1) - (3, 6, 4) - (6, 1, 9) - (2)
P2: (1, 6, 7, 9) - (3, 5, 2) - (4, 5)
Start constructing a child C, which at first is a copy of P1:
C: (1, 5, 3, 2) - (1, 4) - (8, 7, 9) - (5)
Now choose a random group from P2, and add it to the child:
C: (1, 6, 7, 9) - (3, 5, 2) - (1, 4) - (6, 7, 9) - (5)

-------------------

Original parents:
P1: (2, 5, 1) - (3, 6, 4) - (6, 1, 9) - (2)
P2: (1, 6, 7, 9) - (3, 5, 2) - (4, 5)
Child correctly: C: (1, 6, 7, 9) - (3, 5, 2) - (1, 4) - (6, 7, 9) - (5)

Now remove all previous groups from the child that contain duplicated items:
C: (1, 6, 7, 9) - (3, 5, 2) - (1, 4) - (6, 7, 9) - (5)
(note that group (1, 4) is gone, because it contained a duplicated ‘1’, and group (6, 7, 9) is gone)

We now have some missing items – which we’ll turn into, and add back to the groups using a suitable heuristic (e.g. FFD). In this case, we have only lost the 4 – suppose in this problem we find it missing the (3, 8, 2) group – we now end up with:

C: (1, 6, 7, 9) - (3, 8, 2) - (4)
Notes on group based crossover

The intuition behind crossover is that:

- The parents are presumably good (they were selected, after all)
- So the parents have good combinations of genes
- If we combine good combinations of genes from different parents, we may get even better combinations.

Fine, but we have realised it is important to have an idea, for a given problem, of what these combinations might look like. In grouping based problems, and considering the direct encoding, the combinations likely to be important are groups of genes with the same value. These are disrupted badly by ordinary crossover, but preserved with slight variation by group-based crossover.

Evolving Rules

It seems that thin married males read the Sun.

There are many many possible rules, e.g.:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Cov</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Females younger than 50 smoke and read the Mirror</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>Overweight people smoke</td>
<td>0.33</td>
<td>1.0</td>
</tr>
<tr>
<td>Graduates do not read the Mirror</td>
<td>0.5</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Each rule has a coverage and an accuracy.

**Coverage:** The proportion of the DB that it covers ... i.e. that it makes a prediction about. E.g. the second rule only covers people with BMI > 1.

**Accuracy:** The proportion of times it is correct about the records it covers.
Evolving Rules

In large datasets (perhaps many thousands of records, and 10s or 100s of attributes), finding the 'best' rules is a hard problem, and a valid area for EAs to work in.

The EA searches a space of rules, encoded somehow, with suitable operators. We'll get to that.

Fitness is a combination of coverage and accuracy.

Alternatively you might evolve a set of rules, which together cover all the data, so fitness is just the accuracy of the set.

Associated **Examinable reading** (for all) is section 3 of the Freitas paper on the web site.

---

**Evolving Graphs**

There are many reasons why we may want to evolve graphs: Usually we have a fixed set of nodes, and we wish to choose which links to use. E.g. communications networks, pipe networks. EAs, for example, are recognised as the best methods for use in the water distribution industry.
Types of Graph problem

Choosing what to use for each link:
E.g. in Water Distribution networks, the network of pipes itself is often fixed, decided in some other way. But you need to decide, for each link, what type of pipe it will use. There may be, say, 10 different diameters available for pipes, and 100 links. So you can just use a 10-ary encoding of length $L$, with standard crossover and mutation operators.

Searching the topology

More interestingly, we might need to design the topology of the network. I.e. choose which links to include and which not to. As well as design of new pipe networks, or road networks, this is used for comms networks of many kinds.

How might you encode a graph?
Graph encodings

Suppose we have \( n \) nodes to link together (buildings, routers, cities, whatever). There are \( n(n-1)/2 \) potential links.

There are two basic direct encodings:

**Binary:** \( n(n-1)/2 \) genes, each either 0 or 1.

**Edge-based:** have an ID for each of the possible \( n(n-1)/2 \) possible links, and the chromosome is simply a variable length list of links.

This graph has 7 nodes, and hence 21 possible links. Let’s order the links as follows, and identify them as 1 (AB) to 21 (FG):

AB, AC, AD, AE, AF, AG, BC, BD, BE, BF, BG, CD, CE, CF, CG, DE, DF, DG, EF, EG, FG

In the binary encoding, the above graph is:

00001010100110000101

In an edge-based encoding, this graph is:

5, 7, 9, 12, 13, 19, 21
In the binary encoding, we can of course use standard \( k \)-ary operators.

But there are some issues with the binary encoding:

- How many genes are there for a 1000-node graph?
- Very often, a graph is only a valid solution at all if it is \textit{connected} – i.e. there is a path between every two nodes. Does the binary encoding guarantee this?

Which of these issues is also an issue for the edge-based encoding?

In the edge-based encoding, we need specialised operators:

**Mutation:**

M1: Choose a random link not present, and add it in.

\[ \text{e.g. } 5, 12, 19, 20 \text{ might be mutated to } 5, 12, 14, 19, 20 \]

M2: Choose a random existing link, and delete it.

\[ \text{e.g. } 5, 12, 19, 20 \text{ might be mutated to } 5, 12, 20 \]

**Crossover:**

Many crossover operators could be designed, but an obvious one is:

C1: combine the parents, remove duplicates, and delete a small random number of nodes (why delete any?)

\[ \text{E.g. Parent1: } 1, 3, 7, 19, 20 \]
\[ \text{Parent2: } 3, 6, 7, 8, 15, 17 \]

\[ \text{combine and remove duplicates: } 1, 3, 6, 7, 8, 15, 17, 19, 20 \]
\[ \text{delete 3 at random: } 1, 3, 7, 8, 17, 19 \]
Constructive heuristics for graphs

Both of these encodings fail to guarantee connected graphs. Constructive heuristics can provide such a guarantee, along with other benefits indicated in previous lecture.

A common type of problem involves simply finding the ‘best’ spanning tree. If you have a distance matrix for the nodes, and best means minimal cost, then the problem is easy. However, almost all variants on this are hard. E.g. if there is a degree constraint (say, no more than 3 links from any node), the best degree-constrained spanning tree is hard.

So, let’s look at a constructive-heuristic based encoding for spanning trees.

Kruskal’s algorithm

We saw Prim’s algorithm in an earlier lecture. This is another one. Pretend you have to build a tree on \( n \) nodes, and you have the cost matrix, telling you what the cost is of each possible link.

 Initialise Tree (list of edges) to be empty, i.e. \{\}
 Repeat \( n-1 \) times:
   Find the lowest cost edge that does not introduce a cycle, and add it into \( T \).

Example: Let’s just take a tiny example of a 5-node graph, with cost matrix (all other costs are 100):
\[
c(\text{a,b)} = 5, c(\text{a,c)} = 15, c(\text{a,d)} = 18, c(\text{a,e)} = 22, c(\text{b,d)} = 6, c(\text{b,e)} = 7, c(\text{c,d)} = 14, c(\text{d,e)} = 4
\]

\[a\] \[b\] \[c\] \\
\[d\] \[e\]
Modifying Kruskal’s algorithm

If we need to modify it to handle degree constraints, or something similar, this is often very simple indeed. We just change the red bit:

Initialize Tree (list of edges) to be empty, I.e. []
Repeat $n-1$ times:
    Find the lowest cost edge that does not introduce a cycle, and add it into $T$.

E.g.

Initialize Tree (list of edges) to be empty, I.e. []
Repeat $n-1$ times:
    Find the lowest cost edge that does not introduce a cycle and does not violate a degree constraint, and add it into $T$.

Hybrid tree encodings

We can straightforwardly base our encoding over such a constructive method.

E.g. Consider this encoding for spanning trees.

The chromosome is simply a permutation of the nodes.
The first two nodes indicate the first edge. The interpretation of each subsequent node is: find the cheapest edge that validly links from the tree so far to this new node.

E.g. Let’s interpret: CADBE with our example

$\sigma(a,b) = 5, \sigma(a,c) = 15, \sigma(a,d) = 18, \sigma(a,e) = 22, \sigma(b,d) = 6, \sigma(b,e) = 7, \sigma(c,d) = 14, \sigma(d,e) = 40$

![Diagram of tree with nodes a, b, c, d, e and edges labeled with their costs]
Hybrid tree and graph encodings

We can therefore do EA search of spanning trees with a permutation encoding, where the tree is built from the chromosome using such a constructive heuristic.

Since all nodes are included, we are guaranteed connectedness in the graph.

So, how about a version of such an encoding, guaranteeing connectedness, but to search a general space of graphs, not just trees?

An example would be to use a variable length chromosome where each gene is a node, ensuring only that each node appears at least once. Why will this work?

Additional material “Selection”:

Today’s additional material

This lecture is mainly about the overall flow of control in some simple EA designs, and about possible methods to use for the SELECTION step. The additional material is about Operators (crossover and mutation).

Another item of additional material is a book chapter on “Selected Applications of Natural Computation” – which discusses various applications of EAs. This will appear soon in the Springer “Handbook of Natural Computation”
A steady state, mutation-only, replace-worst EA with tournament selection

0. Initialise: generate a population of popsize random solutions, evaluate their fitnesses.

1. Run Select to obtain a parent solution \( X \).
2. With probability \( \text{mute rate} \), mutate a copy of \( X \) to obtain a mutant \( M \) (otherwise \( M = X \)).
3. Evaluate the fitness of \( M \).
4. Let \( W \) be the current worst in the population (BTR). If \( M \) is not less fit than \( W \), then replace \( W \) with \( M \). (otherwise do nothing)
5. If a termination condition is met (e.g. we have done 10,000 evaluations) then stop. Otherwise go to 1.

Select: randomly choose \( \text{tsize} \) individuals from the population. Let \( c \) be the one with best fitness (BTR); return \( X \).  

Selection Issues

Very low pressure selection (e.g. random)
No evolutionary ‘progress’ at all.
Suppose the green blobs indicate the initial population.

With a modest level of pressure.
you may end up here or here:

With very high pressure
(e.g. always select \text{best}), you will end up here
Some Selection Methods

Grand old method:

*Fitness Proportionate Selection* also called *Roulette Wheel selection*

Suppose there are \( P \) individuals with fitnesses \( f_1, f_2, \ldots, f_P \); and higher values mean better fitness.

The probability of selecting individual \( i \) is simply:

\[
\frac{f_i}{\sum_{k=1}^{P} f_k}
\]

*This is equivalent to spinning a roulette wheel with sectors proportional to fitness*

Problems with Roulette Wheel Selection

Having probability of selection directly proportional to fitness has a nice ring to it. It is still used a lot, and is convenient for theoretical analyses, but:

What about when we are trying to *minimize* the *fitness* value?

What about when we may have negative fitness values?

We can modify things to sort these problems out easily, but *fitprop* remains too sensitive to fine detail of the fitness measure. Suppose we are trying to maximise something, and we have a population of 3 fitnesses:

100, 0.4, 0.3, 0.2, 0.1 — the best is 100 times more likely to be selected than all the rest put together! But a slight modification of the fitness calculation might give us:

200, 100.4, 100.3, 100.2, 100.1 — a much more reasonable situation.

Point is: *fitprop* requires us to be very careful how we design the fine detail of fitness assignment.

Other selection methods are better in this respect, and more used now.
Tournament Selection

Tournament selection: tournament size = $t$

Repeat $t$ times
choose a random individual from the pop and remember its fitness

Return the best of these $t$ individuals (BTR)

This is very tunable, avoids the problems of superfit or superpoor solutions, and is very simple to implement

---

Rank Based Selection

The fitnesses in the pop are Ranked (BTR) from Popsize (fittest) down to 1 (least fit). The selection probabilities are proportional to rank.

There are variants where the selection probabilities are a function of the rank.
Rank with low bias
Here, selective fitnesses are based on $\text{rank}^{0.5}$

Rank with high bias
Here, selective fitnesses are based on $\text{rank}^2$
Tournament Selection

Parameter: tournament size, \( t \)
To select a parent, randomly choose \( t \) individuals from the population (with replacement).
Return the fittest of these \( t \) (BTR)

What happens to selection pressure as we increase \( t \)?
What degree of selection pressure is there if \( t = 10 \) and \( popsize = 10,000 \)?

Truncation selection

Applicable only in generational algorithms, where each generation involves replacing most or all of the population.
Parameter \( pcg \) (ranging from 0 to 100%)

Take the best \( pcg\% \) of the population (BTR); produce the next generation entirely by applying variation operators to these.
How does selection pressure vary with \( pcg \)?
Spatially Structured Populations
Local Mating (Collins and Jefferson)

If doing a crossover, then do another random walk from the same cell to get another parent. If doing mutation, we just use the one we already have. Then …

1. Choose random cell
2. Random walk length \( w \) from that cell.
3. Selected: the fittest encountered on the walk
4. Child replaces individual in the starting cell, if >=

Spatially Structured Populations
The ECO Method (Davidor)

Each individual has a Neighbourhood, consisting of itself and the eight immediately surrounding it

Showing the neighbourhood of the red individual
The ECO Method (Davidor)

In ECO, run in a steady state way, each step involves:

1: Choose an individual at random.
2. Run fitness proportionate selection among only the neighbourhood of that individual, selecting a parent.
3. Select parent 2 in the same way.
4. Generate and evaluate a child from these parents (maybe via just crossover, or crossover + mutation – these details are irrelevant to the ECO scheme itself).
5. Use the replace-worst strategy within the neighbourhood to incorporate the child.

(l,m) and (l+m) schemes

An (l+m) scheme works as follows: the difference from (l, m) is highlighted in blue.

The population size is l.

In each generation, produce m mutants of the l population members. This is done by simply randomly selecting a parent from the l and mutating it – repeating that m times.

Note that m could be much bigger (or smaller) than l.

Then, the next generation becomes the best l of the combined set of the current population and the m children.

Is this an elitist strategy?
That’s it for now

The spatially structured populations techniques tend to have excellent performance. This is because of their ability to maintain diversity — i.e. they seem much better at being able to maintain lots of difference within the population, which provides fuel for the evolution to carry on, rather than too-quickly converge on a possibly non-ideal answer. Diversity maintenance in general, and more techniques for it, will be discussed in a later lecture.

Additional material “Falkenauer”:

Additional material “Freitas paper”:

Additional material “Reynolds’ paper”:

Additional material “Swarm Intelligence”:

![Figure 2: Convergence to a safer crossing over time.](image)

Additional material “Selected Applications paper”:

Additional material “CA urbanization paper, CA traffic simulation paper, CA flu paper, CA brain tumour paper, CA and P Systems for HIV”: