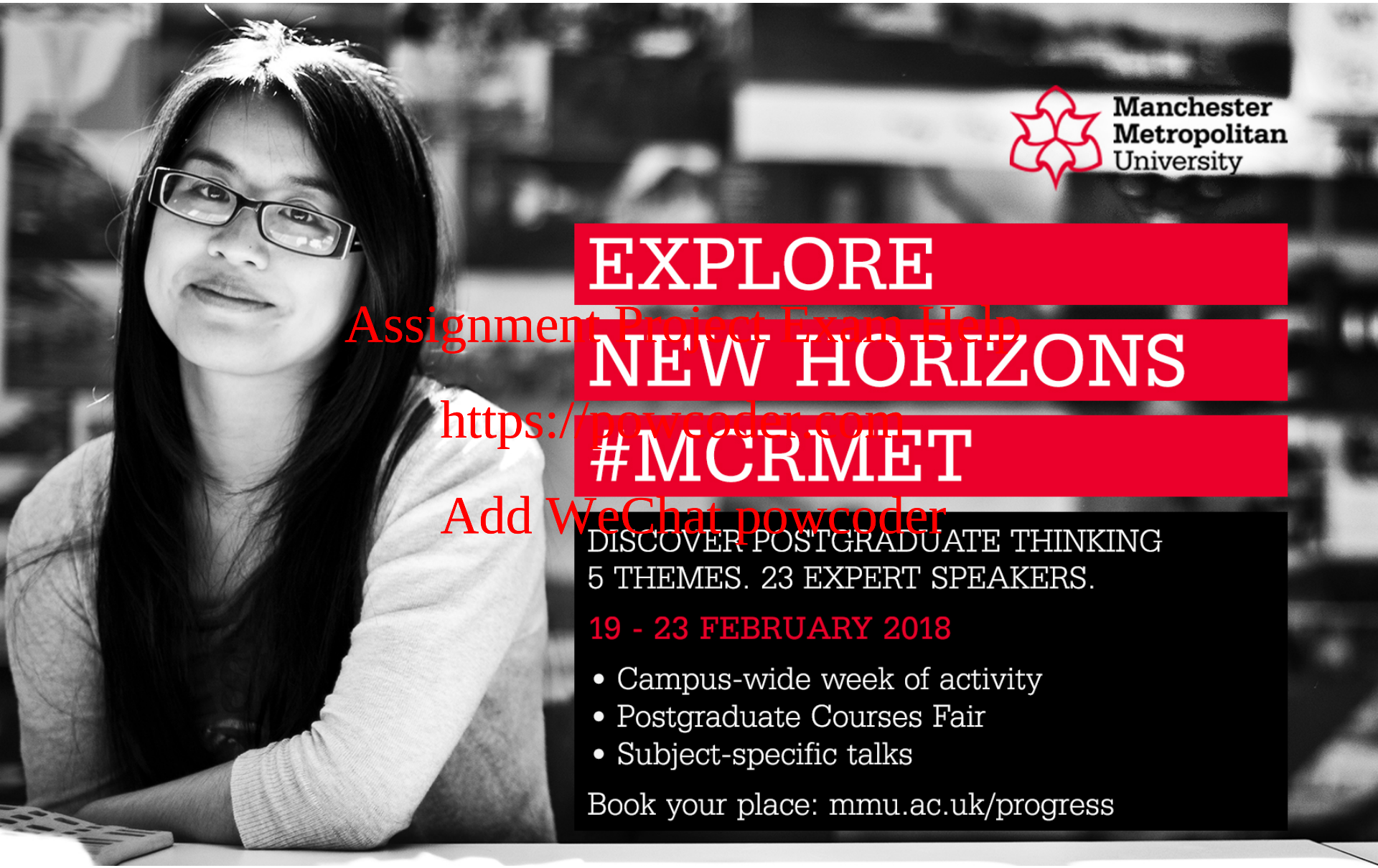


# 6G6Z1109: Software Agents and Optimisation



Term 2, Lecture 6: GA for Travelling Salesman Problem

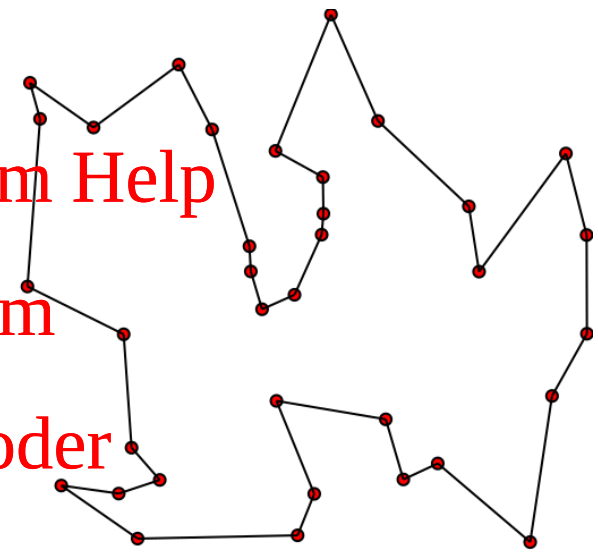
# Travelling Salesman Problem (TSP)

- We might also refer to this (in gender-neutral terms) as the "Delivery Driver Problem"
- TSP is the commonly-accepted title, so we will go with that
- Archetypal NP-complete/NP-hard problem (depending on formulation of question)
- Used as a test-bed for many different optimisation algorithms

# TSP formulation

- Given a set of *cities* distributed in space, find the *shortest* possible route that visits *all* cities *only once* and then returns to the start point
- That is, find a *circular tour* of minimum length





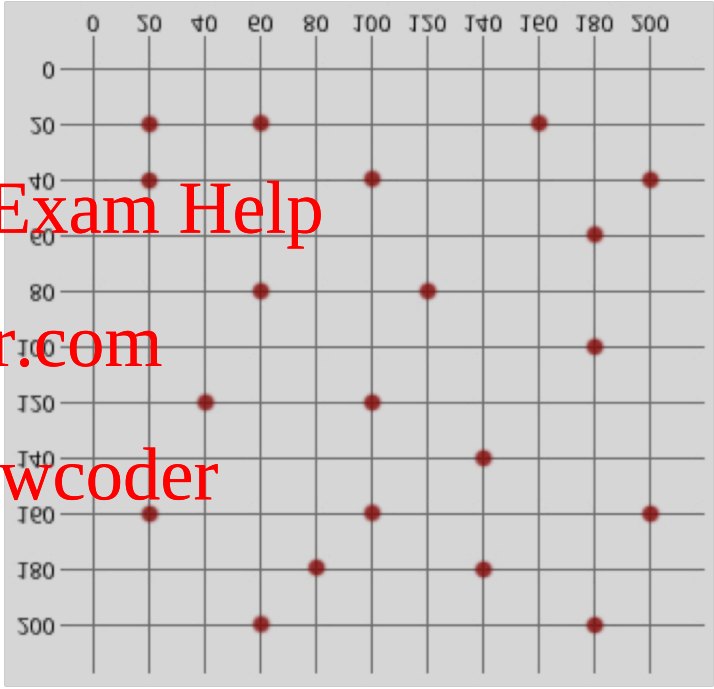
Wikipedia

# Why is the TSP interesting?

- Apart from being inherently hard (and therefore challenging), the TSP finds many applications in various domains such as
  - Transportation (best way for a school bus to pick up *n* children, best way to make deliveries to *n* locations)
  - Circuit board drilling (minimise "travel time" for drill head to make *n* holes in a board)
  - Crew scheduling, advertisement loop placement, etc. etc.

# TSP formulation

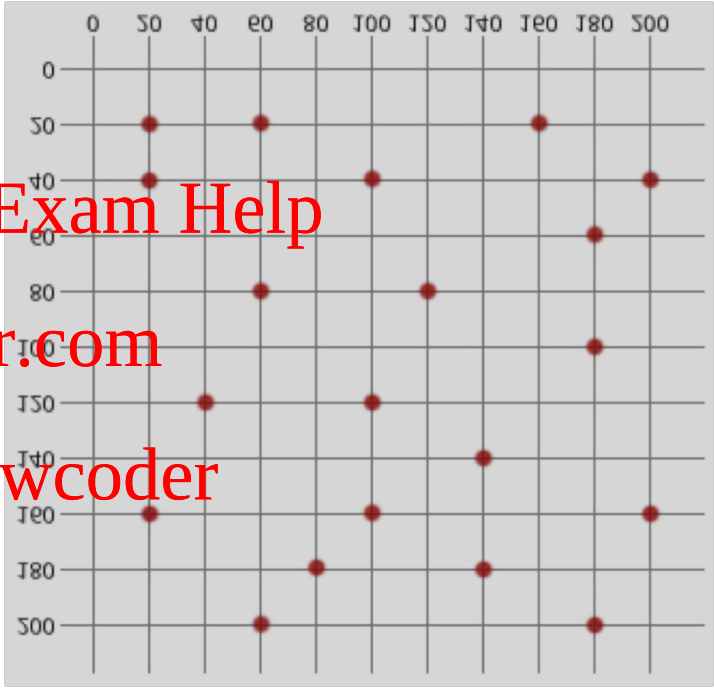
- Some variants of the TSP include the edges (corresponding to a set of roads)
- We focus on the *Euclidean* TSP, which simply distributes cities in 2D space, and assumes that there exists a direct straight road between any possible pair of cities)




Lee Jacobson

# TSP hardness

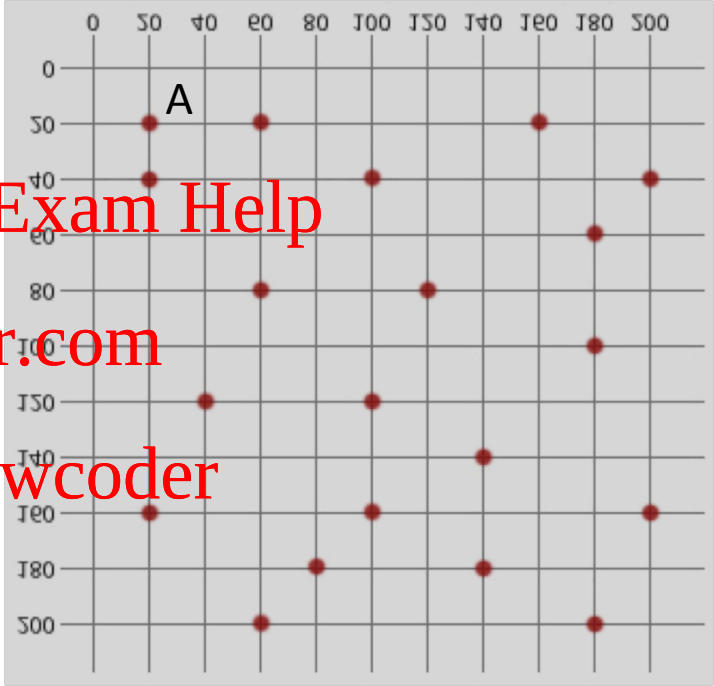
- Why is the TSP hard?
- For n cities ($n$>=3), there are (n-1)!/2 possible tours
- So, for 20 cities, there are (19*18*17*16...*1)/2 possible tours
- 60,822,550,204,416,000 for this particular instance




Lee Jacobson

# TSP hardness

- Why ($n$-1)!/2 possible tours?
- For $n$=20, if we start at arbitrary city, A, we then have 19 to choose from…
- Once we move to the next city, we then have 18 to choose from, and so on…. (which gives us the factorial)
- $n$-1, because we have one fewer choice to make (as, by default, we have to return to the start point)
- We *divide the number of tours by two*, because we don't care about the *direction* of the tour (that is, ACDBE is considered identical to EBDCA)
- A tour is a *permutation* of cities (that is, an ordering of cities)




Lee Jacobson

TSP | Processing 3.3.6

Run

TSP | Edge | Individual | Map | Node | OrderedCrossover | Population

```
import java.util.Collections;
import java.util.Arrays;

Map cities; // the graph that we are trying to 3-colour

Population MyPopulation;

int generation;
int tournamentSize;
float selectionPressure;
Individual bestSolution;
PFont f;

void setup ()
{
  size(500, 500);
  smooth(2);
  background(255);

  f = createFont("Arial",18,true);

  // set up the graph to be solved
  cities=new Map();

  cities.addCity('A', 20, 20);
  cities.addCity('B', 20, 40);
  cities.addCity('C', 20, 160);
  cities.addCity('D', 40, 120);
  cities.addCity('E', 60, 20);
  cities.addCity('F', 60, 80);
  cities.addCity('G', 60, 200);
```

# TSP encoding

- Given that a tour is simply an ordering of cities, it seems natural to use a string of city labels to represent a solution
- Each individual, therefore, starts with a shuffled (randomised) version of the city list as its genome

```
Individual (int genomeSize) // basic constructor for creating an initial individual
{
  this.genomeSize=genomeSize;
  genome = new char[genomeSize];

  // create shuffled version of city list
  char[] characters = "ABCDEFGHIJKLMNOPQRST".toCharArray();

  for (int i = 0; i < characters.length; i++)
  {
    // repeatedly pick two genes and swap them over
    int randomIndex = (int)(Math.random() * characters.length);
    char temp = characters[i];
    characters[i] = characters[randomIndex];
    characters[randomIndex] = temp;
  }
  // copy scrambled city list into genome
  for (int i=0; i<characters.length; i++)
  {
    genome[i]=characters[i];
  }
}
```

# TSP evaluation

- We then need to calculate the *length* of the tour encoded by the list of cities
- This gives us the *fitness* of the tour (where lower (ie. shorter) values are considered better)
- So, for the sequence DBCAE, we would add together distances D-B, B-C, A-E, then E back to D

```java
double calcRouteLength(String route)
{
  char s,t;
  int s_index, t_index;
  double length=0;

  // for each pair of cities, add distance between them to route length
  for (int i=0; i<route.length(); i++)
  {
    if (i<route.length()-1) // if not on last city in list...
    {
      s=route.charAt(i);    // s and t are consecutive cities in the route
      t=route.charAt(i+1);
    }
    else
    {
      s=route.charAt(i);    // otherwise, s is last city in the list
      t=route.charAt(0);    // and t is starting point (loop back)
    }

    // calculate position in city list using city's character index
    s_index=(int)s-(int)'A';
    t_index=(int)t-(int)'A';

    // calculate distance from source node to destination node
    Node source=nodelist.get(s_index);
    Node dest=nodelist.get(t_index);

    // add distance to current length
    length = length+source.calcDist(dest);
  }

  return(length);
}
```

Takes string of city labels as input



New class, `Map`, which is a version of the `Graph` class

```
double calcRouteLength(String route)
{
  char s,t;
  int s_index, t_index;
  double length=0;

  // for each pair of cities, add distance between them to route length
  for (int i=0; i<route.length(); i++)
  {
    if (i<route.length()-1) // if not on last city in list...
    {
      s=route.charAt(i);    // s and t are consecutive
      t=route.charAt(i+1);
    }
    else
    {
      s=route.charAt(i);    // otherwise, s is last city
      t=route.charAt(0);    // and t is starting point (loop back)
    }

    // calculate position in city list using city's character index
    s_index=(int)s-(int)'A';
    t_index=(int)t-(int)'A';

    // calculate distance from source node to destination node
    Node source=nodelist.get(s_index);
    Node dest=nodelist.get(t_index);

    // add distance to current length
    length = length+source.calcDist(dest);
  }

  return(length);
}
```

For each pair of city labels (including "wrap around to start")....



New class, `Map`, which is a version of the `Graph` class

```
double calcRouteLength(String route)
{
  char s,t;
  int s_index, t_index;
  double length=0;

  // for each pair of cities, add distance between them to route length
  for (int i=0; i<route.length(); i++)
  {
    if (i<route.length()-1) // if not on last city in list...
    {
      s=route.charAt(i);    // s and t are consecutive cities in the route
      t=route.charAt(i+1);
    }
    else
    {
      s=route.charAt(i);    // otherwise, s is last city in the list
      t=route.charAt(0);    // and t is starting point (loop back)
    }

    // calculate position in city list using city's
    s_index=(int)s-(int)'A';
    t_index=(int)t-(int)'A';

    // calculate distance from source node to destin
    Node source=nodelist.get(s_index);
    Node dest=nodelist.get(t_index);

    // add distance to current length
    length = length+source.calcDist(dest);
  }

  return(length);
}
```



Extract the appropriate nodes from the ordered list, calculate distance between them, and add to running total

New class, `Map`, which is a version of the `Graph` class

```java
double calcRouteLength(String route)
{
  char s,t;
  int s_index, t_index;
  double length=0;

  // for each pair of cities, add distance between them to route length
  for (int i=0; i<route.length(); i++)
  {
    if (i<route.length()-1) // if not on last city in list...
    {
      s=route.charAt(i);    // s and t are consecutive cities in the route
      t=route.charAt(i+1);
    }
    else
    {
      s=route.charAt(i);    // otherwise, s is last city in the list
      t=route.charAt(0);    // and t is starting point (loop back)
    }

    // calculate position in city list using city's character index
    s_index=(int)s-(int)'A';
    t_index=(int)t-(int)'A';

    // calculate distance from source node to destination node
    Node source=nodelist.get(s_index);
    Node dest=nodelist.get(t_index);

    // add distance to current length
    length = length+source.calcDist(dest);
  }

  return(length);
}
```



New class, `Map`, which is a version of the `Graph` class

# Distance between two points

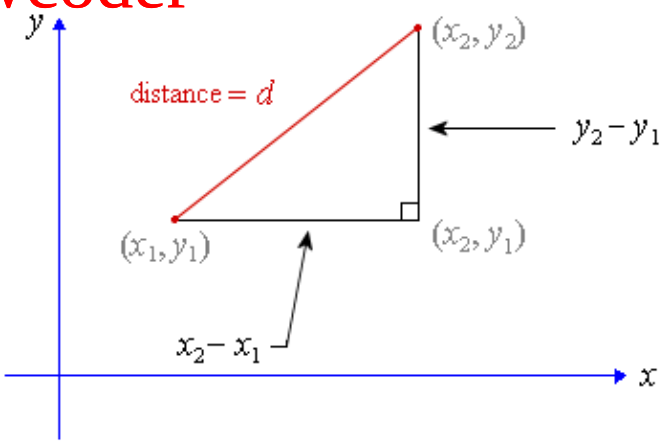
```
double calcDist(Node other)
{
   // use Pythagoras to calculate distance from this node to "other"
   int xDistance = Math.abs(this.x - other.x);
   int yDistance = Math.abs(this.y - other.y);
   double Distance = Math.sqrt((xDistance*xDistance) + (yDistance*yDistance));

   return Distance;
}
```



Added to `Node` class

Calculates *absolute* distances in X and Y (that way, it doesn't matter which heading the other node is from the current node)

Then uses Pythagoras' theorem to calculate distance




Tiger Algebra

Processing also has a `dist()` function.

# Problem (1): Crossover

Imagine we have two genomes:

ADBCE

DBCEA



Crossing them over using standard one-point operator will generate *illegal* tours…

AD | BCE

DB | CEA = ADCEA DBBCE

Visit some cities more than once, don't visit all cities

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



```
P1: IEHDGCFBJA
P2: ABCDEFGHIJ
Ch: AEHDGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



```
P1:  IEHDGCAFBJA
P2:  ABCDEFGHIJ
Ch:      DGCFB
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



```
P1:  IEHDGCAFBJA
P2:  ABCDEFGHIJ
Ch:      DGCFBI
```

Is I already in child? N, so include

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent







```
P1:  IEHDGCAFBJA
P2:  ABCDEFGHIJ
Ch:     DGCFBIJ
```

Is J already in child? No, so include

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is A already in child? N, so include

```
P1: GCAFBIJA
P2: ABCDEFGHIJ
Ch: A  DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is B already in child? Y, so skip

```
P1:  ABCDEFGHIJ
P2:  ABCDEFGHIJ
Ch:  A   DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is C already in child? Y, so skip

```
P1: I  BCJA
P2: ABCDEFGHIJ
Ch: A  DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is D already in child? Y, so skip

```
P1: IE    JA
P2: ABCDEFGHIJ
Ch: A  DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is E already in child? No, so include

```
P1: IEH      A
P2: ABCDEFGHIJ
Ch: AE DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is F already in child? Y, so skip

```
P1: IEHD
P2: ABCDEFGHIJ
Ch: AE DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is G already in child? Yes, so skip

```
P1: IEHDG
P2: ABCDEFGHIJ
Ch: AE DGCFBIJ
```

# Solution: new crossover operator

*Ordered* crossover: take a random subsequence of cities from first parent, then fill in *remaining* cities in the order in which they appear in the second parent



Is H already in child? N so include

```
P1:  IEHDGCFBJA
P2:  ABCDEFGHIJ
Ch:  AEHDGCFBIJ
```

# Problem (2): fitness values

- If we are using fitness values that can take a wide *absolute* range, some fitnesses can come to dominate, leading to early stagnation
- Also, as we saw with 3-colouring, lots of very small fitness values can lead to zero-sized mating pool
- Also, situation is complicated by the fact that we are *minimising* the function
- Solution: use a *different selection operator* that allows us to easily compare fitnesses (to minimise, and to allow for very small fitnesses) and doesn't suffer from excessive selective pressure

# Tournament selection

- Essentially, sample a random selection of *n* population members, then just pick the one with the *lowest tour length* (ie., the winner)
- *n* is called the *tournament size*, and can be tuned like other parameters
- We can also tune the *selection pressure* of the operator, by only selecting the winner a *certain proportion of the time* (otherwise, we just select a random member of the tournament)
- In practice, this prevents early stagnation

```
Individual tournamentSelection()
{
  Individual [] Tourn = new Individual[tournamentSize];
  double shortestTour=999999999;
  int shortestIndex=0; // index of individual with shortest tour

  // pick a bunch of random genomes, and add
  for (int i=0; i<tournamentSize; i++)
  {
    int randomID = (int) (Math.random() * popsize);
    Tourn[i]=genomes[randomID];
  }

  if (random(1)<selectionPressure) // use usual tournament
  {
    for (int i=0; i<tournamentSize; i++) // find winner
    {
      if (Tourn[i].length<shortestTour)
      {
        shortestTour=Tourn[i].length;
        shortestIndex=i;
      }
    }
    return (Tourn[shortestIndex]);
  }
  else
    return (Tourn[0]); // otherwise, just pick the first individual
}
```

Guarantees the first tour we check will be the shortest by default

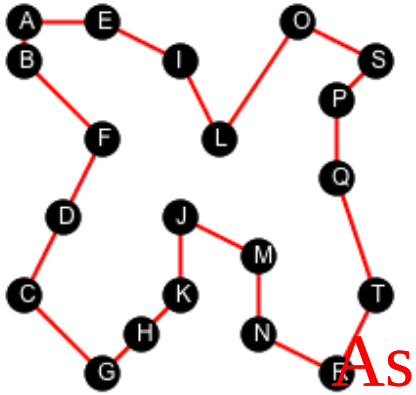
```
Individual tournamentSelection()
{
  Individual [] Tourn = new Individual[tournamentSize];
  double shortestTour=999999999;
  int shortestIndex=0; // index of individual with shortest tour

  // pick a bunch of random genomes, and add to tournament
  for (int i=0; i<tournamentSize; i++)
  {
    int randomID = (int) (Math.random() * popsize);
    Tourn[i]=genomes[randomID];
  }

  if (random(1)<selectionPressure) // use usual tournament
  {
    for (int i=0; i<tournamentSize; i++) // find winner
    {
      if (Tourn[i].length<shortestTour)
      {
        shortestTour=Tourn[i].length;
        shortestIndex=i;
      }
    }
    return (Tourn[shortestIndex]);
  }
  else
    return (Tourn[0]); // otherwise, just pick the first individual
}
```

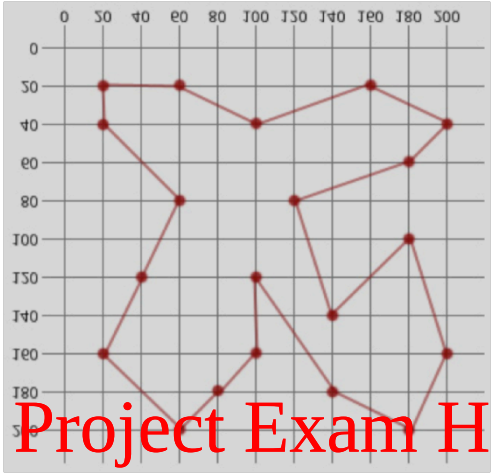
Build the tournament

```
Individual tournamentSelection()
{
  Individual [] Tourn = new Individual[tournamentSize];
  double shortestTour=999999999;
  int shortestIndex=0; // index of individual with shortest tour

  // pick a bunch of random genomes, and add to tournament
  for (int i=0; i<tournamentSize; i++)
  {
    int randomID = (int) (Math.random() * popsize);
    Tourn[i]=genomes[randomID];
  }

  if (random(1)<selectionPressure) // use usual tournament
  {
    for (int i=0; i<tournamentSize; i++) // find winner
    {
      if (Tourn[i].length<shortestTour)
      {
        shortestTour=Tourn[i].length;
        shortestIndex=i;
      }
    }
    return (Tourn[shortestIndex]);
  }
  else
    return (Tourn[0]); // otherwise, just pick the first individual
}
```



Pick winner/ return random individual





Lee Jacobson solution=940



Best route length so far: 871.1173538448469

However, our solution uses a population of 500, rather than 50. Using a population of 50 tends to generate inferior solutions.

= Population size can dramatically alter performance…

# Next lecture

- Next week: Local search
- This week's lab: Implement ordered crossover