CSE 101: Introduction to Computational and Algorithmic Thinking

Unit 12: The Limits of Computation

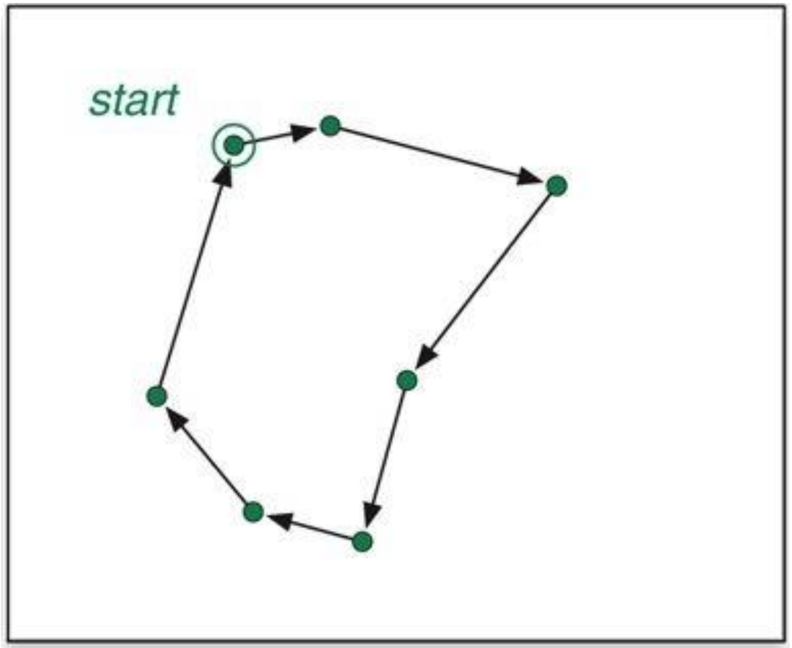
Kevin McDonnell

An Intractable Problem

- In computer science we have a broad class of problems which are generally called **intractable**, meaning that there are no known efficient solutions for these problems
- One such famous, intractable problem is called the **Traveling Salesman Problem**, or **TSP** for short
- Imagine a salesman who must travel to *n* different cities
- He doesn’t care about the order in which he visits the cities, as long as he visits each city exactly once, returning to his starting city at the end
- Between every pair of cities we have a travel time or travel cost or some other metric

An Intractable Problem

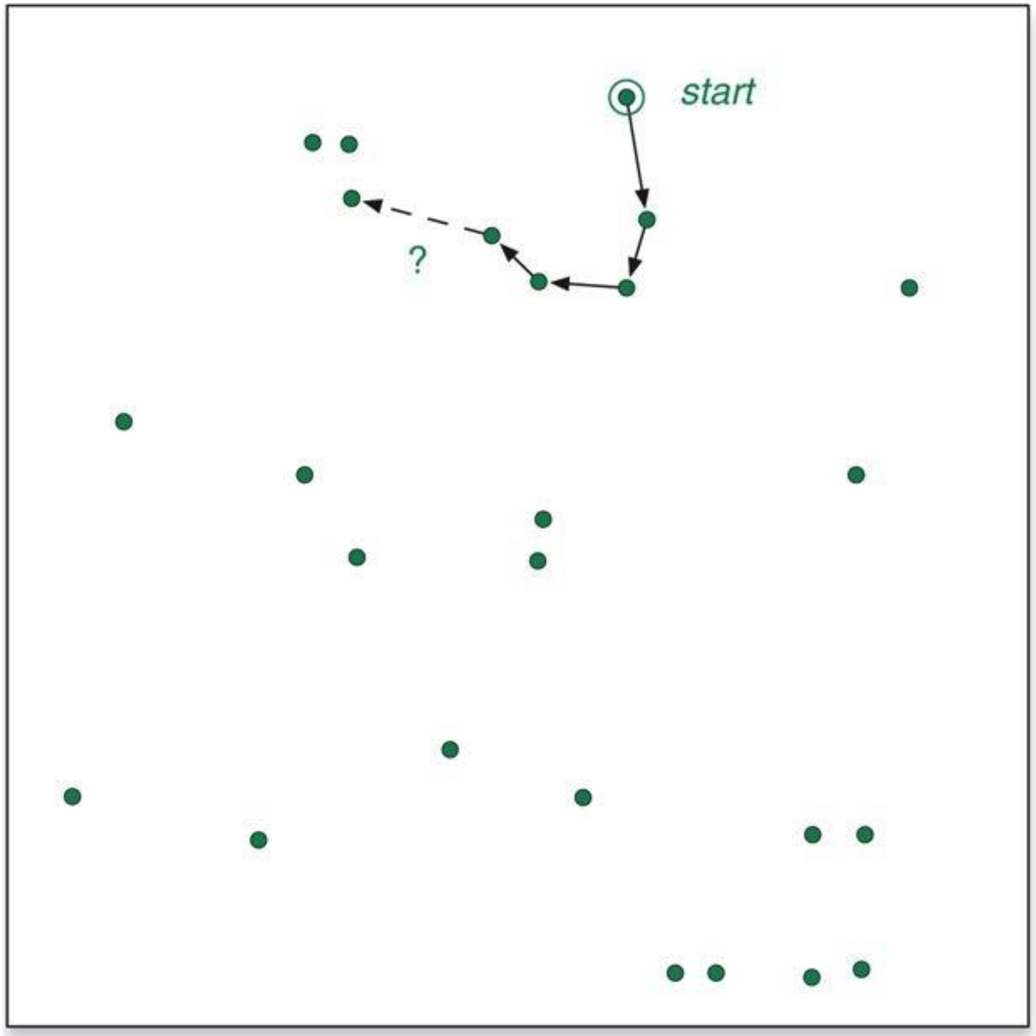
- The goal is to minimize the total cost of the trip while still visiting every city exactly once (except the starting city, which we visit twice)
- TSP arises in many real-world situations, such as routing delivery trucks or school buses and even in CPU manufacturing (to find the fastest way to make the connections between components)
- How can we solve this problem?



- It turns out that simply visiting the nearest unvisited city will not provide the optimal solution, so we need to find another strategy

An Intractable Problem

- As an example, the dashed line in the figure is actually not part of the best tour



Exhaustive Search

- One simple strategy is to list all of the possible tours through the n cities and pick the tour with the smallest cost
- We *exhaustively* list all possible tours and search through them for the one with smallest cost
- There are approximately $(n-1)!$ such tours
- For very small values of n the number of tours to check is not that large
- But the magnitude of $(n-1)!$ becomes very large very quickly as n gets bigger
- For $n = 25$ cities there are approximately 3×10^{23} tours
- That’s 300,000,000,000,000,000,000,000 possibilities!

Evolutionary Algorithms

- An algorithm based on exhaustive search is usually impractical
- In this Unit we will look at an **evolutionary algorithm**, one based on concepts from biological evolution
- The algorithm creates a set of random tours and picks a few with lowest costs (the "fittest" tours)
- The worst tours are discarded, and new tours are developed based on the "survivors" by "mutating" the survivors a little
- Once again the best tours are retained, the worst are discarded, and another round of "evolution" occurs to search for the best possible solution

Maps and Tours

- To make the problem concrete we can use some driving distances between several cities in Ireland, as organized in the **matrix** shown below:

	Belfast	Cork	Dublin	Galway	Limerick
Belfast	—				
Cork	425	—			
Dublin	167	257	—		
Galway	306	209	219	—	
Limerick	323	105	198	105	—

Maps and Tours

- **`TSP.py`** has a **`Map`** class that defines the distance between cities
- We can load a file to get the distances:

```
from TSP import *
m = Map('tsp/ireland.txt')
```

- A call to **`m.display()`** generates this output of distances between cities in Ireland:

	belfast	cork	dublin	galway	limerick
belfast	0.00				
cork	425.00	0.00			
dublin	167.00	257.00	0.00		
galway	306.00	209.00	219.00	0.00	
limerick	323.00	105.00	198.00	105.00	0.00

Exhaustive Search

- In mathematics, a **permutation** is an ordering of the items in a list or the characters in a string
- To solve TSP we could list all permutations (tours) of the *n* cities and pick the tour with lowest cost
- Three example permutations of five cities:

DCGLB DCLGB BCDGL

Exhaustive Search

- To see an example of how this might work, **`TSP.py`** contains a **generator** function that will create a list of all permutations of a string
- A generator is a function that produces a sequence of values *on demand*, instead of all at once

```
from TSP import each_permutation
for s in each_permutation('ABC'):
    print(s)
```

- Since there are 3 letters in "ABC", the function produces $3! = 6$ permutations

Exhaustive Search

- The function **xsearch** will take a **Map** object and perform an exhaustive search for the best tour

```
def xsearch(m):
    best = m.make_tour()
    for t in m.each_tour():
        if t.cost() < best.cost():
            best = t
    return best
```

- Example: for the five-city map of Ireland (**m**), **xsearch(m)** returns this **Tour** object: **['belfast', 'galway', 'limerick', 'cork', 'dublin'] 940.000** (940 is the cost of the tour)
- See **TSP_xsearch.py**

Exhaustive Search

- Earlier in the course we studied **big-Oh notation**, a mathematical technique for analyzing algorithm efficiency
- The exhaustive algorithm for finding the optimal solution to TSP has **factorial time complexity**, meaning the algorithm takes a very long time to find the solution

Random Search

- Evolutionary algorithms are based in part on randomization of data
- For TSP, imagine if we generate a few random permutations of the cities and make small adjustments to the tours until we find the optimal one
- As a first step towards implementing algorithm we need some code that will generate random permutations and pick the one with the lowest cost
- The steps to create one permutation would be this:
  1. **`a = m.cities()`** to get a list of the cities in the map
  2. **`permute(a)`** to shuffle the cities
  3. **`t = m.make_tour(a)` to compute the tour cost**

Random Search

- These three steps can be performed by **make_tour** itself if we pass it the value **'random'** as its argument
- Consider the function **rsearch**, which generates a set of *n* random tours and picks the best one:

```
def rsearch(m, n):
   best = m.make_tour('random')
   for i in range(n-1):
      t = m.make_tour('random')
      if t.cost() < best.cost():
         best = t
   return best
```

- See **TSP_rsearch.py**

Random Search

- An example of how to call this function:

```
m = Map('tsp/ireland.txt')
rsearch(m, 10) # generate 10 random tours
```

- Possible return value:

```
['belfast', 'cork', 'limerick', 'galway', `dublin']
 1021.0
```

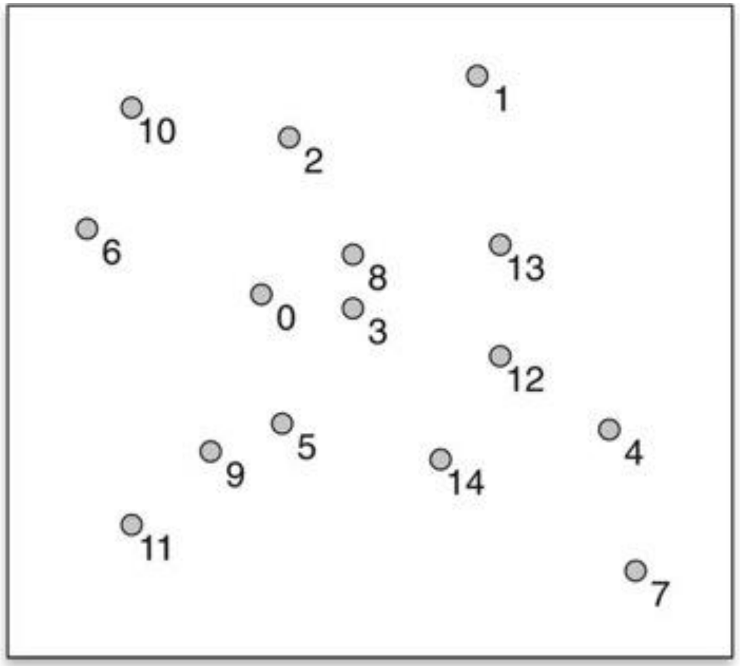
- Note that the tour generated may not be the optimal one (as in this case)
- For large numbers of cities, the exhaustive search approach is impractical
- We could try random search with a value of *n* like 1000, which will give an approximate solution pretty quickly

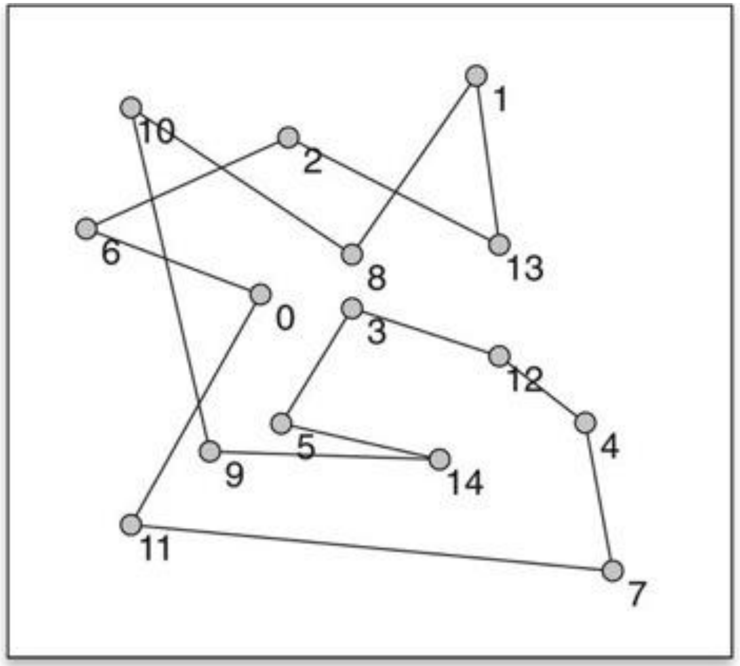
Random Search

- Another way to test the **`rsearch`** function is to have the **`Map`** class generate a map with randomly placed cities:

```
m = Map(15) # cities numbered 0 to 14
```

- **`make_tour`** would generate output like the figure on the right





Point Mutations

- The evolutionary algorithm we seek to implement is “biologically inspired” (hence the name "evolutionary")
- Specifically, it is a kind of **genetic algorithm**, wherein our list of city names serve as the “DNA” of a single tour
- Each slight change we will make to a tour is called a **point mutation**
- A point mutation for us will be to exchange a city with its neighbor in a tour
- We start with a tour:

```
m = Map('tsp/ireland.txt')
t = m.make_tour()
```

Point Mutations

- To mutate the cities we can call the **`mutate`** method and indicate at what index we want to make a mutation: **`t.mutate(2)`**
- For example, suppose we start with this tour:

`['belfast', 'cork', 'dublin', 'galway', 'limerick'] 329.0`

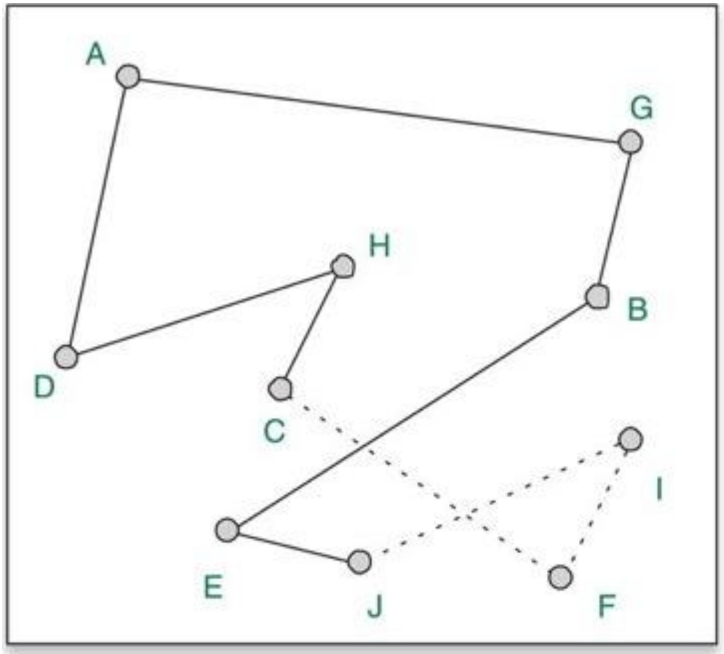
- Perhaps it will mutate into this:

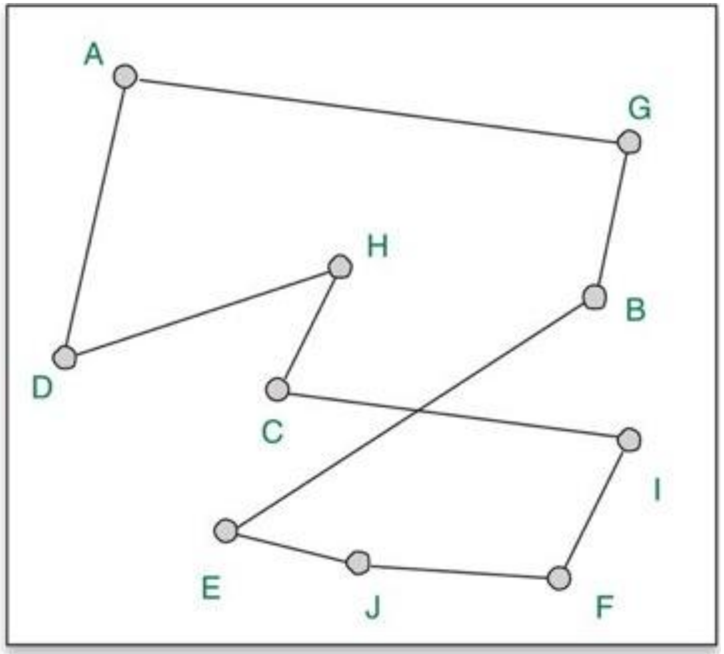
`['belfast', 'cork', 'galway', 'dublin', 'limerick'] 1374.0`

- We see that the cities at indexes 2 and 3 have been exchanged, resulting in a different tour with a higher cost

Point Mutations

- Another example, with randomized input:





- Exchanging cities I and F in the tour shortened the tour length by “untwisting” it in one place

The Genetic Algorithm

- We can now combine the ideas of random generation and mutation to derive the genetic algorithm
- To select the best tour(s) that will "evolve" into the next generation of tours, we need to have a "population" of tours to draw from
- We will write the function **`esearch`**, which takes these arguments:
  - a map, which is a list of cities in the tour
  - the number of generations to simulate
  - the number of tours in the population
- A helper function **`init_population`** will handle generating a list of tours (the "population")

The Genetic Algorithm

```
def init_population(m, popsize):
    pop = [m.make_tour('random')
                for i in range(popsize)]
    return pop
```

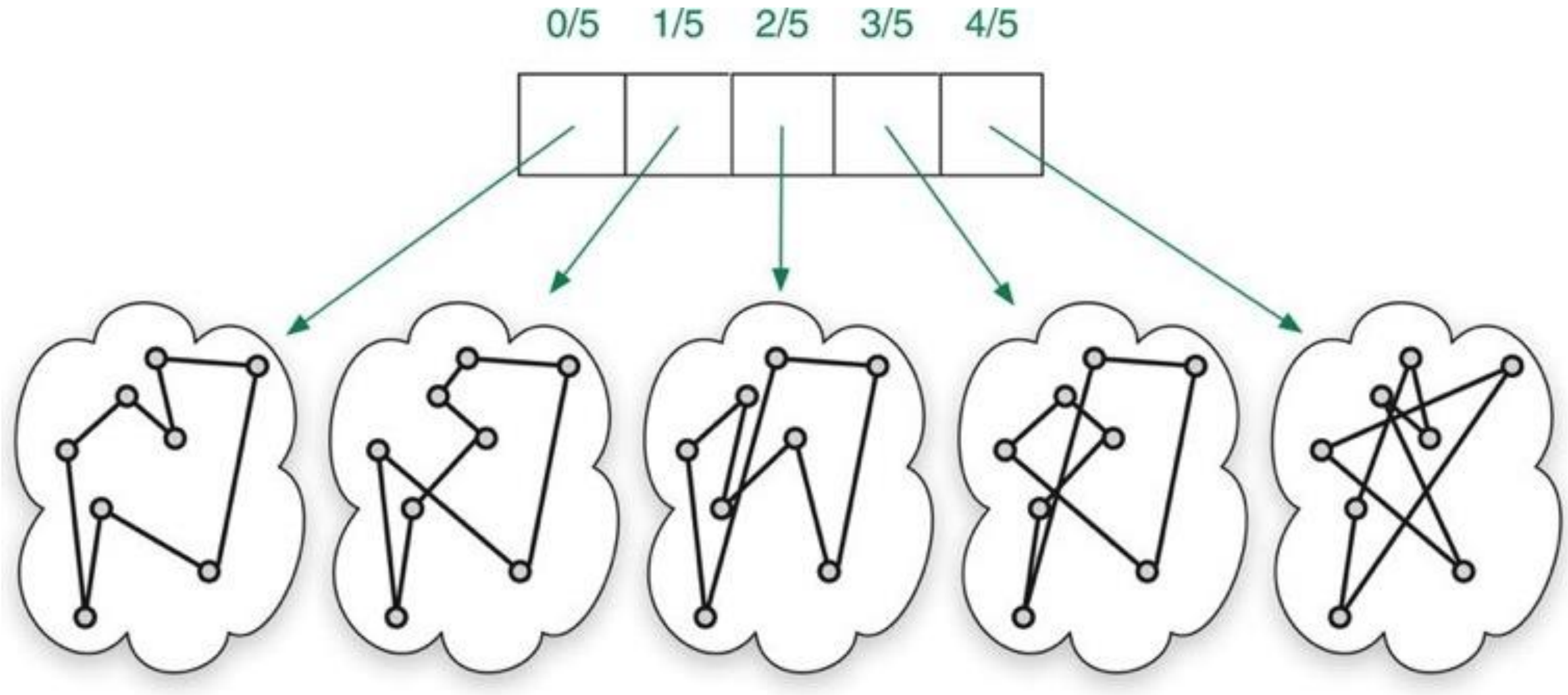
- The function generates **`popsize`** random tours from the map **`m`**
- Note that a list comprehension is being used
- The list of tours is then sorted by tour length, putting the shortest tour is at the front of the **`pop`** list
- A different helper function called **`select_survivors`** performs "random selection," similar to how the process is described for biological evolution
- See **`TSP_esearch.py`**

The Genetic Algorithm

- The **`select_survivors`** function removes tours from the list with a certain probability
  - Remember that the shortest tours are at the front of the list
  - Specifically, the tour in index i is deleted with probability i/p, where p is the size of the population
  - Note that the best tour has no chance of being deleted because its probability is $0/p$
- This gives some of the worst paths a chance to “survive” and perhaps evolve into better solutions
- On average, half of the tours will be “deleted” by being replaced with the value **`None`** in the list

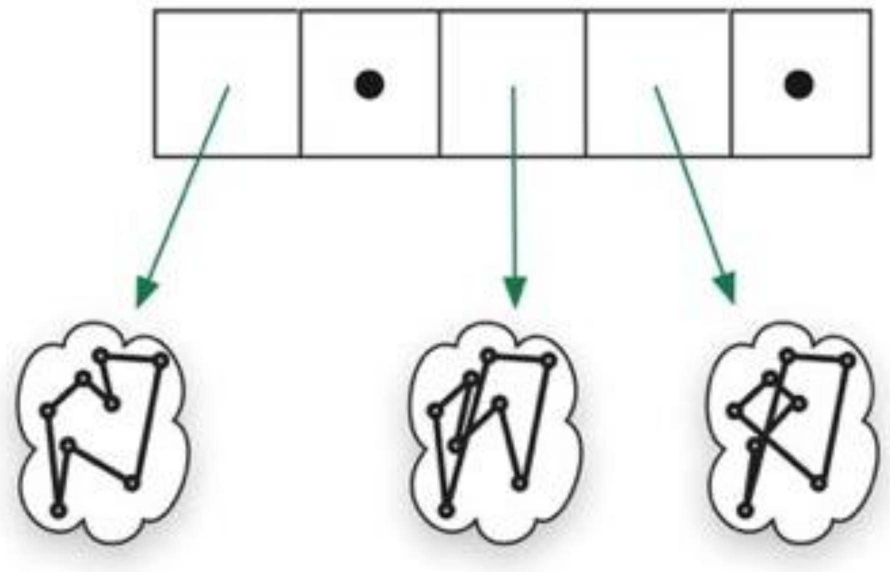
The Genetic Algorithm

- The figure below shows the sorted tours before **`select_survivors`** has been called



The Genetic Algorithm

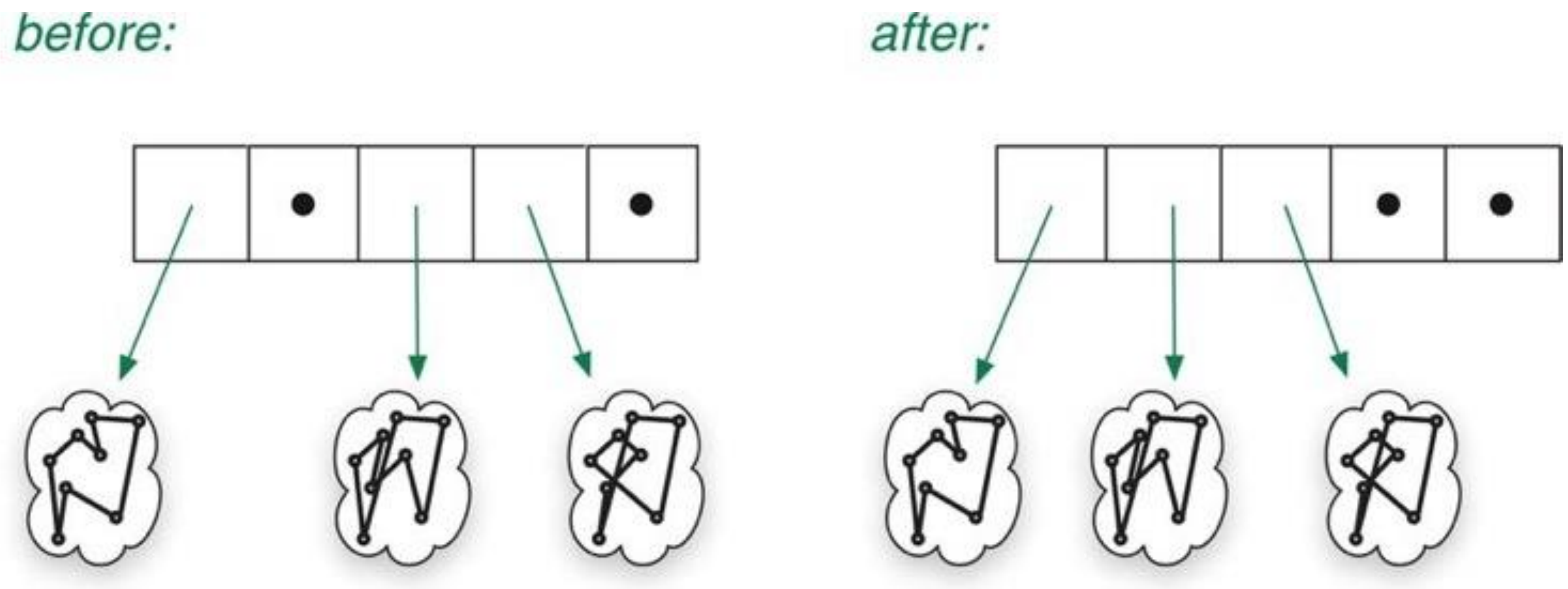
```
import random
def select_survivors(population):
    n = len(population)
    for i in range(1,n):
        if random.random() < i/n:
            population[i] = None
```



- This algorithm will introduce **`None`** objects, as shown in the example above
- See **`TSP_esearch.py`**

The Genetic Algorithm

- A helper function named **`compact_population`** moves the survivors to the front of the lists and moves None objects to the end
- The figure below shows the list of **`Tour`** objects before and after a call to **`compact_population`**:



The Genetic Algorithm

```
def compact_population(population):
    d = 0
    for i in range(1,len(population)):
        if population[i] is None:
            d += 1
        elif d > 0:
            population[i-d], population[i] =
                  population[i], population[i-d]
    return len(population) – d
```

- **d** is a running count of how many **None** objects there are
- Inside the for-loop **d** also tells us how many positions *backwards* we to move an element so that the **None** objects are shuffled to the end.
- See **TSP_esearch.py**

The Genetic Algorithm

- To see why this works, let's call the function with a list of integers and None objects

```
a = [1, 2, None, 3, 4, None, None, 5]
compact_population(a)
```

- Here is how the list is updated with each swap operation:

```
[1, 2, None, 3, 4, None, None, 5]
```

- `d = 1`, so move the 3 back 1 position:

```
[1, 2, 3, None, 4, None, None, 5]
```

- `d = 1`, so move the 4 back 1 position:

```
[1, 2, 3, 4, None, None, None, 5]
```

- `d = 3`, so move the 5 back 3 positions:

```
[1, 2, 3, 4, 5, None, None, None]
```

The Genetic Algorithm

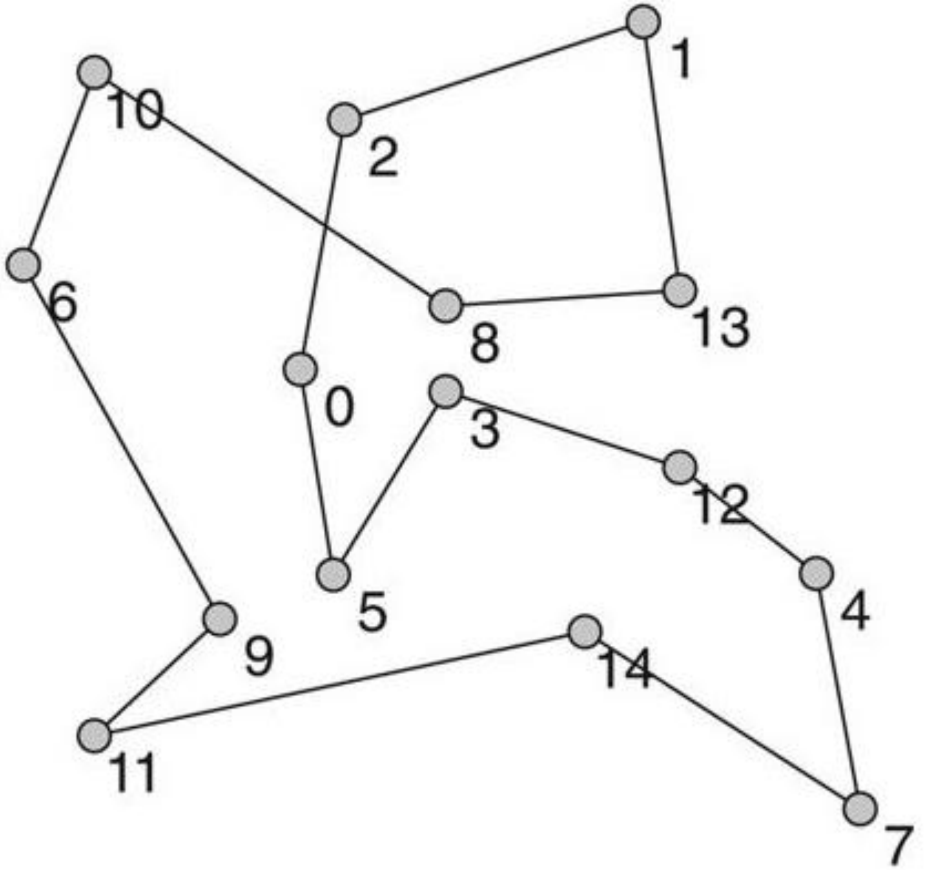
- A helper function **`rebuild_population`** replaces all the **`None`** objects by new tours
  - See **`TSP.py`** for the implementation if you are interested in the details, but it's pretty heavy stuff!

```
def esearch(m, ngen, popsize):
    pop = init_population(m, popsize)
    for i in range(ngen):
        pop.sort(key = Tour.cost)
        select_survivors(pop)
        ns = compact_population(pop)
        rebuild_population(pop, m, ns)
    return pop[0]
```

See **`TSP_esearch.py`**

Crossovers

- While **`esearch`** is running, it can generate tours with twists or kinks that are hard to remove by exchanging only two neighboring cities
- The figure on the right shows an example (2, 1, 13, 8)

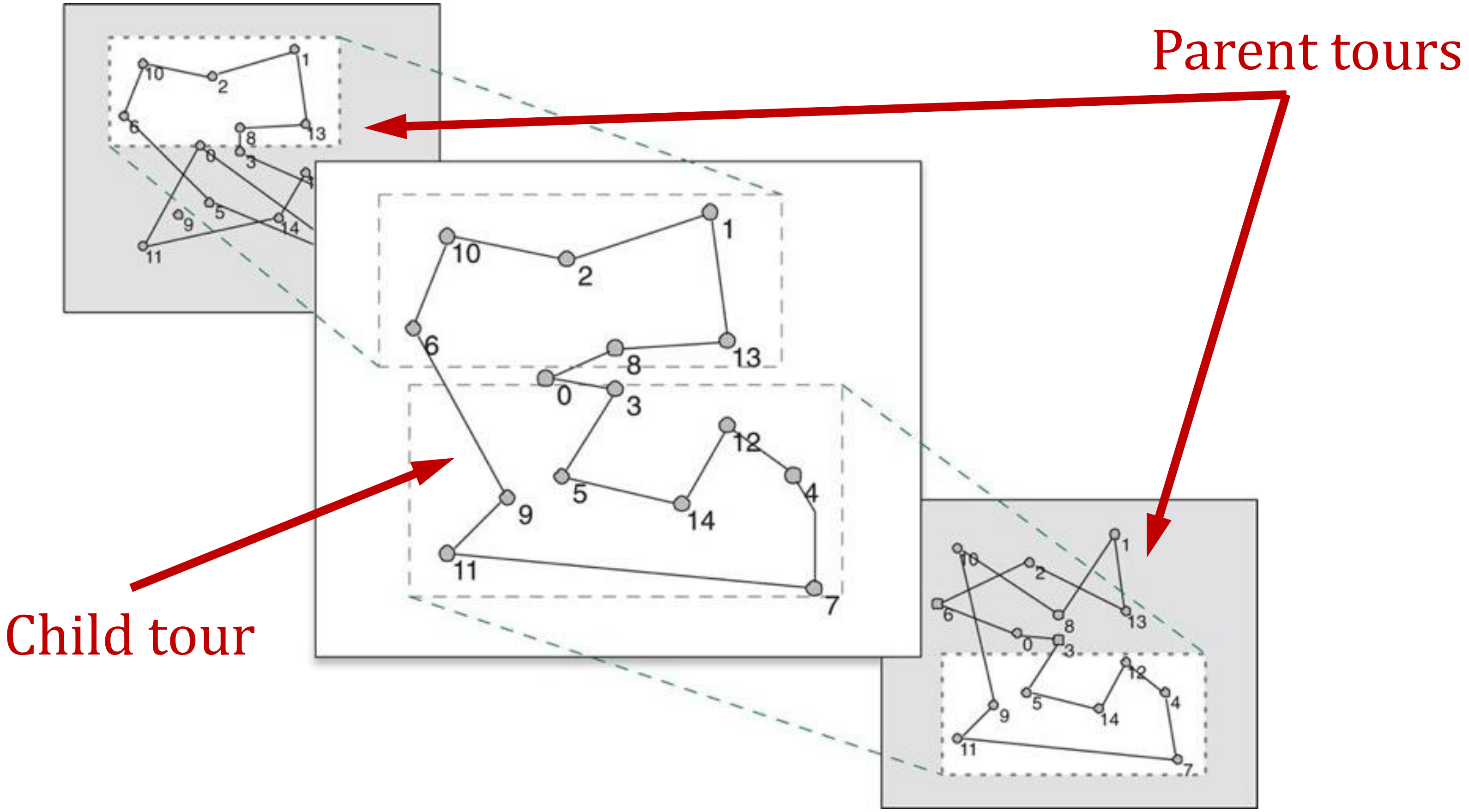


Crossovers

- In genetics, a **crossover mutation** occurs when two chromosomes (very long strands of DNA) break apart
  - And when they come back together there is some mixing, and a new combination is formed
- The idea for us in solving TSP is to take two existing tours and splice them together to form a new tour
- Ideally, the new tour will combine the best portions of its “parent” tours to generate the “child” tour
- We perform a kind of “cut-and-paste” from the parent tours
- Part of the list of city names from one tour is appended to part of the list from a second tour, resulting in a third tour that has large pieces from each of the original tours

Crossovers

- The implementation of this idea is a bit complicated
- See **`TSP.py`** if you are interested in the details



Crossovers

- Now we have two kinds of mutations: point mutations and crossover mutations
- A modified version of the **`rebuild_population`** function chooses randomly between the two types for each new tour it adds to the list of tours
- The **`esearch`** function is otherwise unchanged

The Halting Problem

- Alan Turing in 1936 discovered the **Halting Problem**, which is an **unsolvable problem**
- We have seen in programming that it is possible to create an infinite loop
- In Python it would look like this:

```
while True:
  # do something
```

- So a program that contained such a loop would never actually stop running. It would never halt (terminate).
- Infinite loops are (usually) bugs in code. Wouldn't it be nice if we could write a program to tell us if our code contains such bugs?

The Halting Problem

- Imagine we could write a program called **`HaltChecker`**
  - The **`HaltChecker`** program takes as its input the source code of another program (call it ProgramX)
  - **`HaltChecker`** examines **`ProgramX`**'s source and determines whether **`ProgramX`** will halt eventually
  - It prints either "**`ProgramX`** will halt" or "**`ProgramX`** will not halt"
- Let's say we had our infinite loop from the previous slide in a program called **`InterestingProgram`**

The Halting Problem

- Specifically, our **`InterestingProgram`** looks like this:

```
if HaltChecker says "InterestingProgram will halt" then:
    while True:
        do something
else:
    halt
```

- If **`HaltChecker`** determines that **`InterestingProgram`** will halt, then **`InterestingProgram`** starts executing an infinite loop
- Otherwise, **`HaltChecker`** determines that **`InterestingProgram`** will run forever, and so **`InterestingProgram`** terminates

The Halting Problem

```
if HaltChecker says "InterestingProgram will halt" then:
   while True:
      do something
else:
   halt
```

- **`InterestingProgram`** makes reference to itself
- **`InterestingProgram`** begins by telling **`HaltChecker`** to inspect **`InterestingProgram`** (i.e., itself)
- We know that **`HaltChecker`** will give one of two answers: “InterestingProgram will halt” or “InterestingProgram will not halt”

The Halting Problem

```
if HaltChecker says "InterestingProgram will halt" then:
   while True:
      do something
else:
   halt
```

- If **`HaltChecker`** says “InterestingProgram will halt”, then the if-statement in **`InterestingProgram`** will execute an infinite loop, and so **`InterestingProgram`** will not halt. (A contradiction!)
- Well, what if **`HaltChecker`** says “InterestingProgram will not halt”. Then the if-statement in **`InterestingProgram`** says that **`InterestingProgram`** will halt. (Also a contradiction!)

The Halting Problem

- This means that it is truly impossible to write a program that, for any possible input program, determine whether the input program will halt or not
- So **`HaltChecker`** is an "impossible algorithm" and cannot exist in the world as we understand it
- Now of course we could write a program like **`HaltChecker`** that would work some or most of the time, but not all of the time