Nature Inspired Optimisation for Delivery Problems
Chapter 1: The Traveling Salesperson Problem

Neil Urquhart

May 27, 2022

These slides are designed to accompany the book "Nature Inspired Optimisation for Delivery Problems : From Theory to the Real World".

https://link.springer.com/book/10.1007/978-3-030-98108-2

1 The Travelling Salesperson Problem

2 Techniques for Solving the TSP

3 Summary

The Travelling Salesperson Problem

Definition

A Salesperson must visit at a number of cities, each city must be visited once and only once, with the Salesperson ending up back at the start point. The aim being to find the shortest route that will enable the Salesperson to make their visits.

The problem of finding the optimum order of set of visits remains at the core of most delivery problems.

Solving the TSP

A valid answer to the tsp takes the form of *permutation*

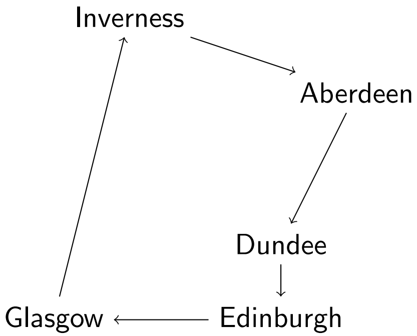
For a 5 city TSP with cities A-E, examples of valid answers would include:

- ABCDE
- BEACD

but **not**

- ABCD (No visit to E)
- BCDEAC (Two visits to C)

A TSP Example: 5 Scottish Cities



The reader should remember that there are 5! (120) possible ways of touring these cities of which this is only one...

Complexity

- The number of possible solutions to a TSP problem is the number of valid permutations of cities.
- The number of permutations that can be constructed for a set of n cities is $n!$ (n factorial) which is simply $n * n - 1 * n - 2 \ldots n - n$.
- In plain language if we have 5 cities the number of permutations is $5 * 4 * 3 * 2 * 1$ which is 120.

Scaleability

Cities	Possible Solutions
5	120
10	3628800
15	$1.307\,67 \times 10^{12}$
20	2.4329×10^{18}
25	$1.551\,12 \times 10^{25}$
30	$2.652\,53 \times 10^{32}$
35	$1.033\,31 \times 10^{40}$
40	$8.159\,15 \times 10^{47}$
45	$1.196\,22 \times 10^{56}$
50	$3.041\,41 \times 10^{64}$
60	$8.320\,99 \times 10^{81}$

Complexity

- As soon as the number of cities rises beyond 10 we're into very large numbers of solutions to search.
- This increase in the number of possible solutions poses a significant problem for us in attempting to solve TSP instances.
- The TSP is often described by researchers as being *NP-Complete* .

NP

- Problems that are considered NP-Complete are those which, although they are possible to solve, cannot be solved using an exhaustive search in a reasonable time.
- This is normally defined as the problem being solvable in *polynomial time*, that is to say there is a polynomial link between the problem size and the time to solve.

NP - an example

- It takes me 50ms to solve a 5 city TSP problem.
- How long does it take to solve a 6 city TSP problem?
    - 60ms?
    - Longer than 60ms?
- It will most likely take longer than 60ms - due to the number of possible solutions to consider.
- As the problem gets bigger so the length of time to solve increases as the number of possible solutions increases.

Considerations for TSP Solvers

- We cannot adopt an *exhaustive search* approach for anything but the smallest problem instances.
- We may never find the optimum solution
- We must encourage the end user to be satisfied with a high quality solution, rather than the optimal solution.

Considerations for TSP Solvers ... cont

- For many real-world problems the difference between the optimum solution and a high quality solution may be little in practical terms.
- For example if the optimum solution for a TSP find a route around a city which is 2 minutes shorter than an easily found high quality solution, could be argued that the saving of two minutes is insignificant in practical terms .

Techniques for Solving the TSP

Exhaustive Search

- Creates and evaluates every possible solution
- Keeps a note of the best solution found so far
- Because *every* possible solution is evaluated it is guaranteed to find the shortest route.
- But is impractical for all but the smallest problem instances.

Exhaustive Search

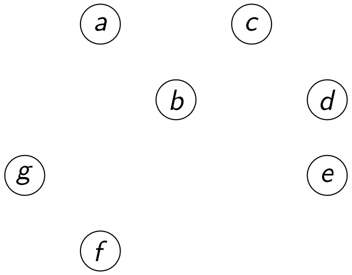
```
**begin**
bestSoFar $= \infty$
solution = getNextSolution()
**while** *solution != null* **do**
    **if** *tourLength(solution)* $<$ *bestSoFar* **then**
        bestSoFar = tourLength(solution)
    **end**
    solution = getNextSolution()
**end**
**return** bestSoFar
```

The Nearest Neighbour Heuristic

- From the starting city, the salesman travels to the nearest *unvisited* city.
- The nearest neighbour (NN) heuristic is a *approximation algorithm* - It will find a good solution, but not necessarily the optimum solution.

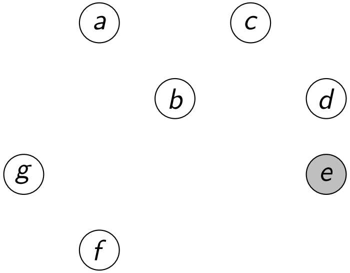
The Nearest Neighbour Heuristic

A simple 7 city TSP problem.



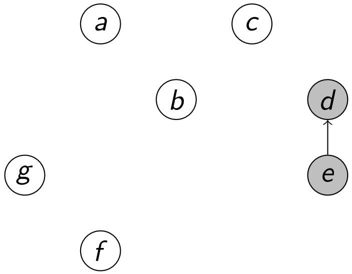
The Nearest Neighbour Heuristic

City E is selected at random as the starting point (coloured grey to show that it has been visited).



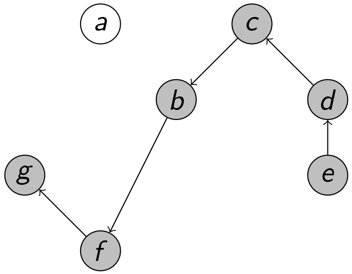
The Nearest Neighbour Heuristic

D is closest to E, so it is added next.



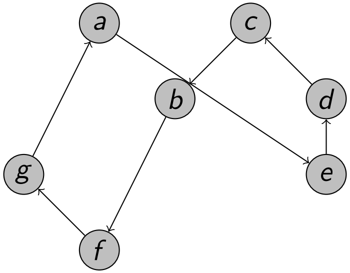
The Nearest Neighbour Heuristic

After E, the closest is C then B, F and G.



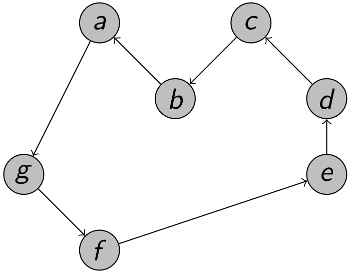
The Nearest Neighbour Heuristic

The last unvisited city is A, which is added at the end of tour



The Nearest Neighbour Heuristic

The tour created by NN is obviously sub-optimal, a simple change such as this can improve upon the NN tour.



The Nearest Neighbour Heuristic

```
**begin**
notVisited = 1 ... n
solution = $\emptyset$
current = removeRand(notVisited)
solution $\cup$ current
while notVisited != $\emptyset$ do
    best = $\infty$
    for possible : notVisited do
        if dist(current,possible) $<$ best then
            next = possible
        end
    end
    notVisited = notVisited - next
    solution $\cup$ next
    current = next
end
return solution
```

The 2-OPT heuristic

- The 2-OPT heuristic is a *local search* algorithm
- 2-OPT iteratively improves a TSP solution.
- Starting from an initial random solution, 2-OPT makes changes to the solution
    - Changes which make a tour shorter are adopted
    - Changes that make a tour longer are rejected.

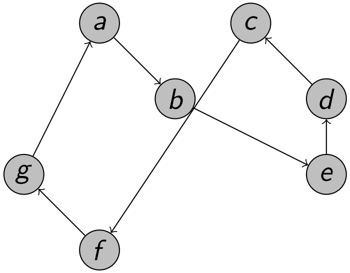
Iterative Improvement

2-OPT is an example of *Iterative Improvement*, an initial solution is created which is then improved (according to a fitness function) over a number of iterations.

```
solution = randomSolution()
while !done do
    solution' = randomChange(solution)
    if quality(solution') < quality(solution) then
        solution = solution'
    end
end
return solution
```

The 2-OPT heuristic

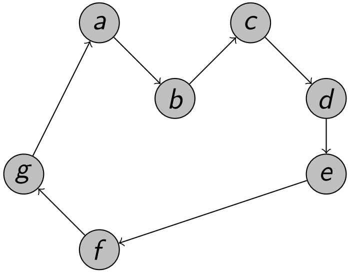
Consider the tour $[ABEDCFG]$ as follows:



We can see that this tour is clearly sub-optimal.

The 2-OPT heuristic

By reversing the order of cities *EDC* to *CDE* we see the following improved solution:



The 2-OPT heuristic

Assume that we have a 10 city tour as follows:

$$ABCDEFGHIJ$$

Now we select positions 4 and 8 for the swap:

$$ABC\mathrm{DEFGH}IJ$$

Applying the swap the tour would become:

$$ABCHGFEDIJ$$

There is no guarantee that a swap will improve a tour, the power of 2-opt is that it attempts many swaps and retains those that improve the tour and discards those that are not an improvement.

The 2-OPT heuristic

```
begin
solution = randomSolution()
improved = true
while improved do
    improved = false
    for $x = 0; x < solution.length; x++$ do
        for $y = x + 1y < solution.length; y++$ do
            solution' = swap(solution,x,y)
            if dist(solution') < dist(solution) then
                solution = solution'
                improved = true
            end
        end
    end
end
return solution
```

The 2-OPT heuristic ... continued

Where: **swap(solution,x,y)**: reverses the route between cities *x* and *y*
The two For loops select the sections to swap as follows:

i	*j*	Swap
0	1	**AB** CDE
0	2	**ABC** DE
0	3	**ABCD** E
...	..	..
1	2	A **BC** DE
1	3	A **BCD** E
...	...	..

Summary

Summary

- This chapter has introduced the TSP.
- We have examined a number of techniques for solving the TSP including exhaustive search, and iterative improvement.
- The reader is encouraged to read the TSP Case study in Chapter 1 and to download and run the example code supplied.