Parabola Volume 23, Issue 3 (1987)

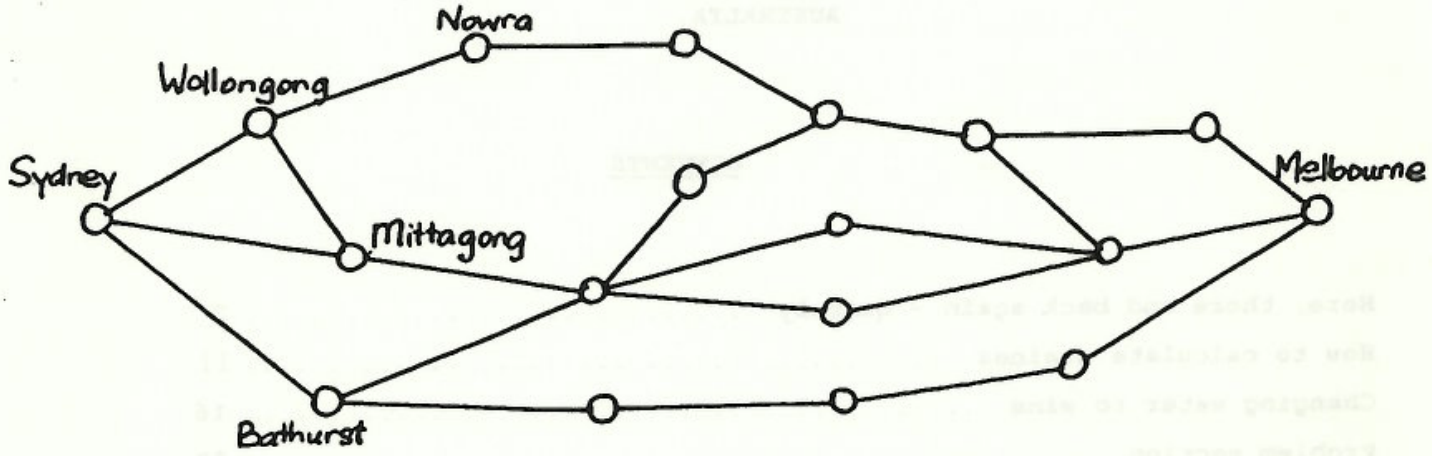
HERE, THERE AND BACK AGAIN - QUICKLY.

John Murray*

Going from one place to another can be a difficult problem. You get halfway there and you often realise that it would have been a lot quicker if you had taken another way (you also remember you've left something you wanted to bring at home and have to go back to get it, but lapses of memory are a different problem to the mathematical ones we are going to look at). How do you choose the best way to get someplace? Let's look at the easiest problem first and see how that works out before getting caught up in the fancier one.

Suppose you want to drive from Sydney to Melbourne along a sequence of roads of shortest total distance. To make it easier to describe we will name a section of road using the two major towns it joins. If there is more than one road joining the two towns, choose the shortest one and ignore the others. Then our shortest path may look something like this

The circles represent the towns and the lines represent the roads joining them. The length of the trip is obtained simply by summing the road lengths of this particular sequence. If you want to represent all the reasonable trips, then rather than drawing them as above you could put them on one picture, much like a road map.

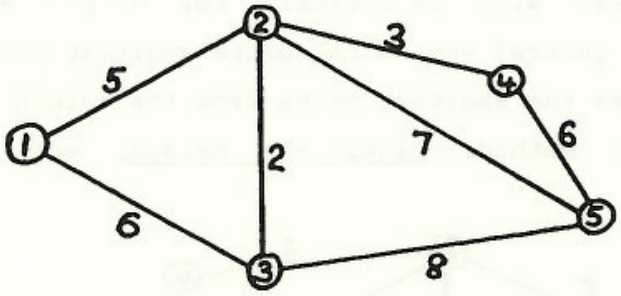


*John is an applied mathematician at the University of New South Wales.

Then any realistic trip consists of tracing out a path through the above network where no town is visited more than once. With such a network one can easily calculate the length of any chosen path by once again simply summing the lengths of the roads that make up that particular path.

These pictures and related questions appear in many different guises and in a wide array of applications so it is not surprising that an area of mathematics has been carved out to analyse and solve them. They belong to the topics of operations research or combinatorial optimization and more specifically are called graphs or networks. So let us use the conventional terminology and call the above drawing a network, the small circles representing the towns nodes and the roads joining the towns arcs. A path will consist of a sequence of connected arcs and its length will be the sum of the path's component arcs. Our problem then is to find a path of minimum length, from the node representing Sydney to the node representing Melbourne.

For a network with a very small number of nodes you can probably see straight away the optimal solution. For instance if we want to find the path of shortest length from node 1 to node 5 in the following network we could test all possible paths and determine that the optimal sequence of nodes is 1, 2, 5.



However the number of possible paths quickly gets out of hand when we increase the number of nodes and arcs, and even the fastest computer would find it impossible to search through all feasible paths to find the shortest one. To get an idea of the size of the problem you may like to try the following exercise due to Victor Klee.

Exercise

Consider a network with an origin, a destination, and 100 additional nodes, with each pair of nodes connected by an arc. Show that the number of different paths (without repeated nodes, of course) from origin to destination is

$$100! + 100\ (99!) + \binom{100}{2}\ (98!) + \ldots + \binom{100}{98}\ 2! + 100 + 1$$

where the nth term counts the paths from the origin to destination which omit n-1 of the other nodes. Show that this sum is the greatest integer in 100!e. Use Stirling's formula to represent 100! in the form $a.10^b$ where b is a positive integer, $1 \le a < 10$ and a is accurate to two significant digits. Stirling's formula asserts

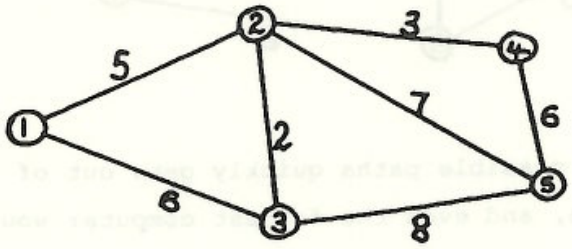
$$\sqrt{2\pi}\, n^{n+1/2}\, e^{-n} < n! < \sqrt{2\pi}\, n^{n+1/2}\, e^{-n}\left(1 + \frac{1}{4n}\right).$$

(Here e^x denotes "exponential x" or "e to the x" and is defined to be the sum of the series

$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots .$$

When we let x = 1 we obtain the value of 2.718 ... for e.)

So even though the problem could hypothetically be solved by looking through the finite number of possibilities, this number for problems of reasonable size is so large as to make this method totally impractical. Since we want to find a method that will obtain the optimal solution in a reasonable amount of time and not search through all possibilities there must be some underlying structure to the method to guarantee that the answer will be optimal. For our problem of shortest paths it turns out that the best general way to calculate shortest paths from an origin to a destination also calculates the shortest paths from the origin to all other nodes. Let's see how one such method, Dijkstra's method, works on our previous small example.



Let u_j = the length of a shortest path from node 1 to node j.

Well we obviously have $u_1 = 0$. We then search for the closest node to 1. This is node 2 and we have $u_2 = 5$. Now try to find the next closest node to node 1. It is either connected directly to node 1 or is the closest node to node 2. Both of these happen to be node 3 and its u value is the minimum of 6 and 5 + 2. So $u_3 = 6$.

The next closest node to node 1 is obtained by finding the closest nodes to 2 and to 3 which are 4 and 5 respectively. But $u_2 + 3 = 8 < u_3 + 8 = 14$ so node 4 is next in line and we have $u_4 = 8$. In the next and last step we compare the three values $u_2 + 7$ with $u_3 + 8$ and $u_4 + 6$ the smallest of which is $u_2 + 7$ and so $u_5 = 12$. We have now found the shortest paths from node 1 to all other nodes and in particular to node 5.

If you think about the way we worked from the closest nodes outwards you might be able to see why we have actually found the shortest paths without having to search through all possible paths. Anyway there is an underlying "Principle of Optimality" on which Dykstra's method is based which guarantees optimality.

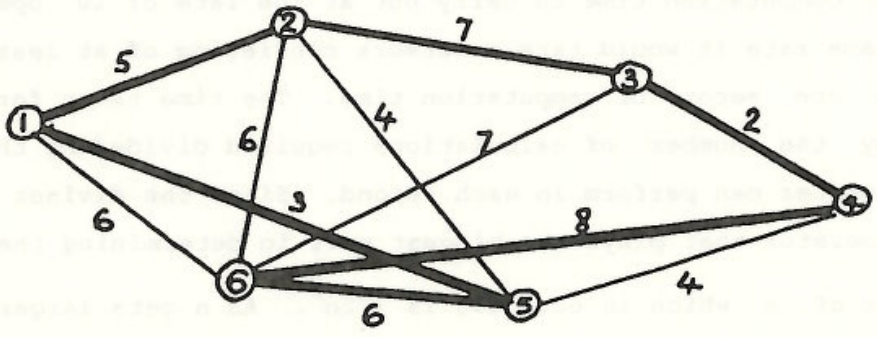
In the calculation in our example, each step involved some comparison of values and some additions. For the general method on a network with n nodes, and arcs joining each pair of nodes (a complete network) an overall total of $(n-1)(n-2)$ comparisons and $(n-1)(n-2)/2$ additions are necessary. This gives us an idea of how much work is required in the worst case (there are as many arcs as possible to find all shortest paths from the origin node. This is one way in which an algorithm can be judged. Not only should it find the correct answer but it should do so without a prohibitive amount of calculation. Our first so-called method where we checked through all possible paths found the correct answer but certainly not in a reasonable amount of time.

So Dijkstra's method seems reasonable. For a 102 node, complete network as in the exercise it requires 15,150 calculations which would take a personal computer about 0.15 seconds of computation time to carry out at the rate of 10^5 operations per second. At this same rate it would take a network consisting of at least 250 nodes before we used up one second of computation time. The time taken for Dijsktra's method is given by the number of calculations required divided by the number of calculations the computer can perform in each second. Since the divisor is so large the term in the numerator that plays the biggest part in determining the time taken is the largest power of n which in our case is $3/2n^2$. As n gets larger and larger this term will swamp out in magnitude the contributions from the other terms in the numerator (in our case these are $-\frac{9n}{2}$ and $\frac{9}{2}$). This is why Dijkstra's method is called an order n^2 (written $O(n^2)$) algorithm. In general an $O(n^2)$ algorithm takes longer than an $O(n)$ algorithm for the same value of n (for large values of n

usually). For example if we have an algorithm that performs a task using 50n-3 operations and one that takes $5n^2 + 3n-2$ operations then on our computer the first algorithm could solve problems for n as big as 2000 in one second of computation time whereas the second could only handle problems for n up to about 140 in one second. If you had a choice about which algorithm to use you would certainly use the first one; it is a lot quicker and can handle much larger problems in the same amount of time. You can also see, at least in this example, that the constant multiplying the leading power of n was not that important. It was the largest power of n that determined how "good" the algorithm was. Extrapolating the previous reasoning, we see that an $O(n^5)$ algorithm is better than an $O(n^7)$ algorithm and so on.

Now let's look at a different problem that is also related to finding the shortest way around a network. It is called the travelling salesman problem and deals with a salesman who must visit a collection of cities. This particular salesman is very efficient minded and so he wants to visit all the cities but each one only once and then return to the city from which he started, and where he presumably lives. He also wants to choose such a "tour" that minimizes the total distance; so no other possible tour would be shorter. In terms of a network this translates into starting at some particular node, visiting all other nodes of the network once and only once and then returning to the original node. Of all such tours, find the shortest one. In some ways this problem sounds similar to the problem of finding shortest paths between nodes, but with a few extra conditions.

If the salesman has to travel the following network starting at node 1, then a possible tour is given by the darker arcs.



The length of that tour is 31.

It turned out that the number of possible paths in the shortest path problem quickly got out of hand. Maybe the added restrictions on a tour cuts down the number

of possible tours enough so that for problems where n is not too large we could possibly check through all tours to find the optimal one. Unfortunately this is not true. For a complete network with n nodes there are (n-1)! possible tours so even for a 10 city problem the number of possible tours is 362,880 and since each tour would require nine additions this would mean a total of 3,265,920 computations which is more than 200 as many as required by Dijkstra's method to find shortest paths for a 102 city problem. The largest travelling salesman problem ever solved had 318 cities which gives approximately 10^{655} possible tours. Assuming that we could possibly enumerate 10^9 tours per second on a computer, it would take roughly 10^{637} years of computing to establish the optimality of the best tour by exhaustive enumeration. To put this in perspective, one estimate of the age of the universe is only 10^{11} years.

From the last example you might guess that the travelling salesman problem is not as easy as it sound - and you would be perfectly correct! The difficulty is that no one has been able to find a method that will calculate the optimal solution and do it efficiently. And what is worse is that we do not know if such a method exists. All possible methods may be "bad".

Before we go any further I suppose we should define what we mean by good and bad methods. Dijkstra's method was $O(n^2)$ and seemed to handle quite large problems fairly easily. Even if we had a method that was $O(n^7)$ say, the computation time would not grow too quickly as the size of the problem grew. Any algorithm that is $O(n^k)$ for some integer k is called a <u>polynomial time algorithm</u>. Remember an algorithm is $O(n^k)$ if it requires $c_k n^k + c_{k-1} n^{k-1} 1 -- + c_1 n + c_o$ single operations to find the optimal solution, where c_1, c_2, .., c_k are constants and n represents the size of the problem (for instance the number of cities). There are methods that require say 2^n calculations to solve the problem. This then would be $O(2^n)$ and is called an <u>exponential time algorithm</u>. If you graph a polynomial like n^7 against an exponential function like 5^n you see that before too long the exponential function is much larger than the polynomial function. That is why we call polynomial time algorithms good and exponential time algorithms bad. If you look at the following table taken from Garey and Johnson [1] you see that for the polynomial time algorithms the time taken, as the problem size increases, grows at a reasonable rate while for the exponential time algorithms it explodes.

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

It is also disturbing that not even faster computers will help much for exponential time algorithms. The following table, also from Garey and Johnson [1] shows the meagre improvement a computer 1000 times faster would make.

Size of Largest Problem Instance
Solvable in 1 Hour

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

So it seems reasonable to call polynomial time algorithms good and exponential [Bgorithms bad. Coming back to the travelling salesman problem, we have no polynomial time algorithm to solve it and do not know whether one exists. Many of the algorithms that at least come close to finding the optimal solution are exponential time algorithms and therefore cannot handle problems that are too large. Hence the 318 city problem is the largest that has been solved so far.

Now that we have commented on the difficulty of finding a solution to the travelling salesman problem let us at least look at a method that certainly does not guarantee finding an optimal solution but uses some criterion to find, hopefully, a reasonable one.

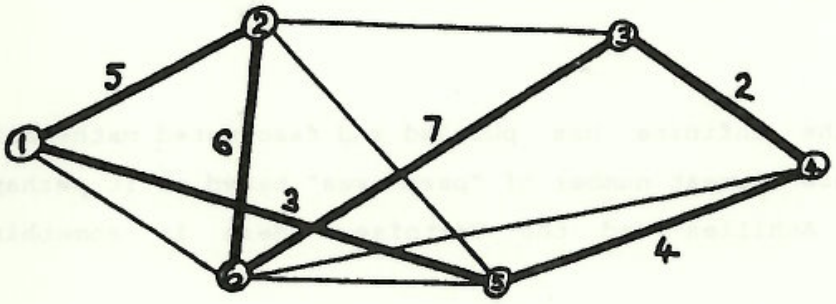
The Nearest Neighbour Algorithm generates a tour as follows.

Starting with the origin node add the nearest node as the next node visited.

Then add the closest node, not already included, to the last one.

Continue like this until all nodes have been connected, and then connect the last node added to the origin node.

Applying this to the last network we generate the following tour.



This has length 27. Notice that unless we are dealing with a complete network we can't even be sure of generating a tour with the Nearest Neighbour Algorithm. For instance if there were no arc between nodes 2 and 6 the method would not have succeeded in finding a tour for this simple network. So just finding a feasible tour can be difficult.

Some of the more sophisticated methods find an initial tour and then try to change parts of it to decrease its length, always in such a way so that we still have a tour.

Part of what's frustrating about operations research is that the problems are often extremely simple to state and in some cases, like the shortest path problem,

are easy to solve but in other cases, like the travelling salesman problem, no efficient method has been found for them and there is a good deal of debate whether there is a polynomial time method that will do so. If you ever see the cryptic notation P = NP?, that is what they are asking, where P stands for polynomial time algorithms and NP for a class of non polynomial time algorithms - it turns out that question is equivalent to "Is there a polynomial time algorithm that will solve a special, simpler travelling salesman problem?"

So if you are going on a one way trip to say Melbourne then you should have no problem in finding the quickest way there. On the other hand, if you plan a return trip with lots of other towns along the way and do not want to revisit any of them, then you had better start calculating the shortest route now. You will need all the time you can get.

M.R. Garey and D.S. Johnson "Computers and Intractability, A Guide to the theory of NP - Completeness", Freeman, 1979.

THE BAG OF THE MISSING BALLS.

The concept of the infinite has puzzled and fascinated mathematicians from ancient times. There are a great number of "paradoxes" based on it perhaps the most famous being that of Achilles and the Tortoise. Here is something a little different.

Supppose I have a bag (a very big bag for it is to hold a great number of balls). At 1 minute to 12 I place in it 10 balls labelled 1 to 10, but then I simultaneously take out ball 1. At $\frac{1}{2}$ minute to 12 I add in balls 11 to 100 but take out ball 2. At $\frac{1}{4}$ minute to 12 I add in balls 101 to 1000 but take out ball 3. And I continue so that at the nth stage, just $1/2^{n-1}$ minutes before 12 I add in balls numbered $10^{n-1}+1$ to 10^{n} but dutifully take out ball number n.

I want to know how many balls are in my bag at 12 o'clock for I'm having trouble finding just one. Is it possibly empty?