IIT-BOMBAY

IE-716 – Project Report

Solving Travelling Salesman Problem using Concorde Solver

Team Concorde

4/28/2023

The Travelling Salesman Problem (TSP) is a historical problem in Linear programming and optimization with significant work done by many mathematicians. Concorde is a program that has been proven to optimally solve a very large set of TSP problems. Team has attempted to understand the basics of the problem and the mechanics used by the solver to compute results. Results are recorded for further reading and research along similar lines.

Table of Contents

Introduction

The Travelling Salesman Problem

The travelling salesman problem (or **TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.

In the theory of computational complexity, the decision version of the TSP (where given a length *L*, the task is to decide whether the graph has a tour of at most *L*) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases in super-polynomial manner (but no more than exponentially) with the number of cities.

The problem has been shown to be NP-hard (more precisely, it is complete for the complexity class **FP**$^{\mathrm{NP}}$; see function problem), and the decision problem version ("given the costs and a number *x*, decide whether there is a round-trip route cheaper than *x*") is NP-complete. The bottleneck travelling salesman problem is also NP-hard. The problem remains NP-hard even for the case when the cities are in the plane with Euclidean distances, as well as in a number of other restrictive cases. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, many heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded inside an optimal control problem. In many applications, additional constraints such as limited resources or time windows may be imposed.

Concorde Solver For The Travelling Salesman Problem

Concorde is a computer code for the symmetric travelling salesman problem (TSP) and some related network optimization problems. The code is written in the ANSI C programming language and it is available for academic research use; for other uses, contact William Cook for licensing options.

Concorde's TSP solver has been used to obtain the optimal solutions to the full set of 110 TSPLIB instances, the largest having 85,900 cities.

The Concorde callable library includes over 700 functions permitting users to create specialized codes for TSP-like problems. All Concorde functions are thread-safe for programming in shared-memory parallel environments; the main TSP solver includes code for running over networks of UNIX workstations.

Concorde now supports the QSopt linear programming solver. Executable versions of Concorde with qsopt for Linux and Solaris are available

TSP Problem – Linear Programming Formulations

The TSP can be formulated as an integer linear program. Several formulations are known. Two notable formulations are the Miller–Tucker–Zemlin (MTZ) formulation and the Dantzig–Fulkerson–Johnson (DFJ) formulation. The DFJ formulation is stronger, though the MTZ formulation is still useful in certain settings.

Feature common to both these formulations is that one labels the cities with the numbers 1, 2, ... n and takes to be the cost (distance) from city i to city j. The main variables in the formulations are:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

It is because these are 0/1 variables that the formulations become integer programs; all other constraints are purely linear. In particular, the objective in the program is to:

Minimize the tour length

$$\sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}.$$

Without further constraints, the {xij}i,j will however effectively range over all subsets of the set of edges, which is very far from the sets of edges in a tour, and allows for a trivial minimum where all xij=0. Therefore, both formulations also have the constraints that there at each vertex is exactly one incoming edge and one outgoing edge, which may be expressed as the 2n linear equations -

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \quad \text{for } j = 1, \ldots, n \text{ and } \sum_{j=1, j \neq i}^{n} x_{ij} = 1 \quad \text{for } i = 1, \ldots, n.$$

These ensure that the chosen set of edges locally looks like that of a tour, but still allow for solutions violating the global requirement that there is *one* tour which visits all vertices, as the edges chosen could make up several tours each visiting only a subset of the vertices; arguably it is this global requirement that makes TSP a hard problem. The MTZ and DFJ formulations differ in how they express this final requirement as linear constraints.

Miller–Tucker–Zemlin formulation

In addition to the xij variables as above, there is for each i=2, 3...n a dummy variable ui that keeps track of the order in which the cities are visited, counting from city 1; the interpretation is that ui < uj implies city i is visited before city j. For a given tour (as encoded into values of the xij variables), one may find satisfying values for the ui variables by making them equal to the number of edges along that tour, when going from city 1 to city i. Since linear programming favours non-strict inequalities over strict (< or >), we would like to impose constraints to the effect that :

uj >= ui +1 if xij = 1.

Merely requiring uj>= ui + xij would *not* achieve that, because this also requires uj >= ui when xij = 0, which is not correct. Instead MTZ use the (n-1)(n-2) linear constraints uj + (n-2) >= ui +(n-1)xij for all distinct i, j in {2, 3, ... n} where the constant term (n-2) provides sufficient slack that xij = 0 does not impose a relation between uj and ui.

The way that the ui variables then enforce that a single tour visits all cities is that they increase by (at least) 1 for each step along a tour, with a decrease only allowed where the tour passes through city 1. That constraint would be violated by every tour which does not pass through city 1, so the only way to satisfy it is that the tour passing city 1 also passes through all other cities.

The MTZ formulation of TSP is thus the following integer linear programming problem:

$$
\begin{aligned}
&\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}: \\
&\qquad x_{ij} \in \{0,1\} && i,j = 1,\ldots,n; \\
&\qquad u_i \in \mathbf{Z} && i = 2,\ldots,n; \\
&\sum_{i=1, i\neq j}^{n} x_{ij} = 1 && j = 1,\ldots,n; \\
&\sum_{j=1, j\neq i}^{n} x_{ij} = 1 && i = 1,\ldots,n; \\
&u_i - u_j + (n-1)x_{ij} \leq n-2 && 2 \leq i \neq j \leq n; \\
&\qquad 1 \leq u_i \leq n-1 && 2 \leq i \leq n.
\end{aligned}
$$

The first set of equalities requires that each city is arrived at from exactly one other city, and the second set of equalities requires that from each city there is a departure to exactly one other city. The last constraints enforce that there is only a single tour covering all cities, and not two or more disjointed tours that only collectively cover all cities. To prove this, it is shown below (1) that every feasible solution contains only one closed sequence of cities, and (2) that for every single tour covering all cities, there are values for the dummy variables ui that satisfy the constraints.

To prove that every feasible solution contains only one closed sequence of cities, it suffices to show that every sub-tour in a feasible solution passes through city 1 (noting that the equalities

ensure there can only be one such tour). For if we sum all the inequalities corresponding to xij = 1 for any sub-tour of *k* steps not passing through city 1, we obtain:

$$(n-1)k <= (n-2)k, \text{ which is a contradiction.}$$

It now must be shown that for every single tour covering all cities, there are values for the dummy variables ui that satisfy the constraints.

Without loss of generality, define the tour as originating (and ending) at city 1.
Choose ui=t if city i is visited in step t (i, t=2, 3,...n).

$$\text{Then } ui-uj <= n-2$$

Since ui can be no greater than n and uj can be no less than 2; hence the constraints are satisfied whenever xij = 0.

For xij = 1, we have:

$$Ui - uj + (n-1)xij = t - (t+1) + n - 1 + n-2, \text{ satisfying the constraint.}$$

Dantzig–Fulkerson–Johnson formulation

Label the cities with the numbers 1, 2, …, *n* and define:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

Take cij > 0 to be the distance from city *i* to city *j*. Then TSP can be written as the following integer linear programming problem:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij} :$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n;$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n;$$

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 \qquad \forall Q \subsetneq \{1, \ldots, n\}, |Q| \geq 2$$

The last constraint of the DFJ formulation—called a *sub-tour elimination* constraint—ensures no proper subset Q can form a sub-tour, so the solution returned is a single tour and not the union of smaller tours. Because this leads to an exponential number of possible constraints, in practice it is solved with row generation.

Exact Algorithms & Heuristics

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute-force search). The running time for this approach lies within a polynomial factor of O(n!), the factorial of the number of cities, so this solution becomes impractical even for only 20 cities.

One of the earliest applications of dynamic programming is the Held–Karp algorithm that solves the problem in time O(n22n), This bound has also been reached by Exclusion-Inclusion in an attempt preceding the dynamic programming approach.

Improving these time bounds seems to be difficult. For example, it has not been determined whether a classical exact algorithm for TSP that runs in time O(1.9999n) exists. The currently best quantum exact algorithm for TSP due to Ambainis et al. runs in time O(1.728n).

Other approaches include:

- Various branch-and-bound algorithms, which can be used to process TSPs containing 40–60 cities.
- Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities.
- Implementations of branch-and-bound and problem-specific cut generation (branch-and-cut); this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities, see Applegate et al. (2006).

Solutions to TSP problems within the International Community

An exact solution for 15,112 German towns from TSPLIB was found in 2001 using the cutting-plane method proposed by George Dantzig, Ray Fulkerson, and Selmer M. Johnson in 1954, based on linear programming. The computations were performed on a network of 110 processors located at Rice University and Princeton University. The total computation time was equivalent to 22.6 years on a single 500 MHz Alpha processor. In May 2004, the travelling salesman problem of visiting all 24,978 towns in Sweden was solved: a tour of length approximately 72,500 kilometres was found and it was proven that no shorter tour exists.

In March 2005, the travelling salesman problem of visiting all 33,810 points in a circuit board was solved using Concorde TSP Solver: a tour of length 66,048,945 units was found and it was proven that no shorter tour exists. The computation took approximately 15.7 CPU-years (Cook et al. 2006). In April 2006 an instance with 85,900 points was solved using Concorde TSP Solver, taking over 136 CPU-years; see Applegate et al. (2006).

Benchmarks Set by Concorde Solver

We report below the performance of Concorde (03.12.19) on a subset of 87 of the easier TSPLIB instances. These instances are all those that could be solved in under 1,000 seconds on a Compaq XP1000 workstation using the (99.12.15) version of Concorde.

The benchmark runs reported below were carried out on a single processor of a dual-processor 2.8 GHz Intel Xeon PC with a 533 MHz front-side-bus and 2 GByte of RAM. The ILOG CPLEX (Version 6.5) linear programming solver was used in these computations.

For each instance we give the number of CPU seconds used in the computation and the the number of nodes in the branch-and-bound search tree. The benchmark runs were carried out with the default version of Concorde (for more difficult instances, the running times can often be improved by using the options "-m -C 24" or greater to invoke multiple passes of the local-cuts routine)

Name	Running Time	Search Nodes	Name	Running Time	Search Nodes
burma14	0.02	1	ulysses16	0.12	1
gr17	0.04	1	gr21	0.01	1
ulysses22	0.29	1	gr24	0.02	1
fri26	0.03	1	bayg29	0.05	1
bays29	0.04	1	dantzig42	0.09	1
swiss42	0.05	1	att48	0.22	1
gr48	0.18	1	hk48	0.08	1
eil51	0.12	1	berlin52	0.13	1
brazil58	0.24	1	st70	0.20	1
eil76	0.11	1	pr76	0.60	1
gr96	0.84	1	rat99	0.40	1
kroA100	0.31	1	kroB100	0.58	1
kroC100	0.30	1	kroD100	0.33	1
kroE100	0.75	1	rd100	0.22	1
eil101	0.24	1	lin105	0.22	1
pr107	0.37	1	gr120	0.48	1
pr124	1.04	1	bier127	0.53	1
ch130	0.65	1	pr136	1.12	1
gr137	2.18	1	pr144	1.01	1
ch150	0.91	1	kroA150	1.47	1
kroB150	2.50	3	pr152	2.98	1
u159	0.42	1	si175	5.75	3
brg180	0.26	1	rat195	6.25	3
d198	3.83	3	kroA200	2.24	1

kroB200	1.06	1	gr202	2.90	1
Name	**Running Time**	**Search Nodes**	**Name**	**Running Time**	**Search Nodes**
ts225	7.04	1	tsp225	5.18	1
pr226	1.06	1	gr229	12.40	7
gil262	3.61	3	pr264	1.05	1
a280	2.48	1	pr299	5.33	1
lin318	2.78	1	rd400	25.72	9
fl417	23.48	3	gr431	31.94	9
pr439	73.62	15	pcb442	9.44	9
d493	43.13	5	att532	32.24	5
ali535	9.50	1	si535	21.73	3
pa561	52.06	7	u574	22.99	3
rat575	70.30	17	p654	8.57	1
d657	83.04	13	gr666	31.31	7
u724	73.78	15	rat783	16.79	1
dsj1000	153.16	13	pr1002	9.92	1
si1032	107.19	1	u1060	141.12	13
vm1084	234.66	13	pcb1173	168.11	13
rl1304	103.01	1	nrw1379	108.12	7
u1432	35.90	1	d1655	69.08	3
pr2392	35.04	1			

Heuristics/ Algorithms Used in Concorde Solver

Nearest Neighbour Algorithm

The nearest neighbour (NN) algorithm (a greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path. However, there exist many specially arranged city distributions which make the NN algorithm give the worst route. This is true for both asymmetric and symmetric TSPs. Rosenkrantz et al. showed that the NN algorithm has the approximation factor CC for instances satisfying the triangle inequality. A variation of NN algorithm, called nearest fragment (NF) operator, which connects a group (fragment) of nearest unvisited cities, can find shorter routes with successive iterations. The NF operator can also be applied on an initial solution obtained by NN algorithm for further improvement in an elitist model, where only better solutions are accepted.

Pair-wise exchange

The pair-wise exchange or *2-opt* technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments created by edge removal into a new and shorter tour. Similarly, the 3-opt technique removes 3 edges and reconnects them to form a shorter tour. These are special cases of the *k*-opt method. The label *Lin–Kernighan* is an often heard misnomer for 2-opt. Lin–Kernighan is actually the more general k-opt method.

For Euclidean instances, 2-opt heuristics give on average solutions that are about 5% better than Christofides' algorithm. If we start with an initial solution made with a greedy algorithm, the average number of moves greatly decreases again and is CC. For random starts however, the average number of moves is CC. However whilst in order this is a small increase in size, the initial number of moves for small problems is 10 times as big for a random start compared to one made from a greedy heuristic. This is because such 2-opt heuristics exploit 'bad' parts of a solution such as crossings. These types of heuristics are often used within Vehicle routing problem heuristics to re-optimize route solutions.

V-opt heuristic

The variable-opt method is related to, and a generalization of the *k*-opt method. Whereas the *k*-opt methods remove a fixed number (*k*) of edges from the original tour, the variable-opt methods do not fix the size of the edge set to remove. Instead, they grow the set as the search process continues. Shen Lin and Brian Kernighan first published their method in 1972, and it was the most reliable heuristic for solving travelling salesman problems for nearly two decades. More advanced variable-opt methods were developed at Bell Labs in the late 1980s by David Johnson and his research team. These methods (sometimes called Lin–Kernighan–Johnson) build on the Lin–Kernighan method, adding ideas from tabu search and evolutionary computing. The basic Lin–Kernighan technique gives results that are guaranteed to be at least 3-opt. The Lin–Kernighan–Johnson methods compute a Lin–Kernighan tour, and then perturb the tour by what has been described as a mutation that removes at least four edges and reconnects the tour in a different way, then *V*-opting the new tour. For many years Lin–Kernighan–Johnson had identified optimal solutions for all TSPs where an optimal solution was known and had identified the best-known solutions for all other TSPs on which the method had been tried.

Boruvka's Algorithm

Borůvka's algorithm is a greedy algorithm for finding a minimum spanning tree in a graph, or a minimum spanning forest in the case of a graph that is not connected.It was first published in 1926 by Otakar Borůvka as a method of constructing an efficient electricity network for Moravia. The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki in 1951; and again by Georges Sollin in 1965. This algorithm is frequently called Sollin's algorithm, especially in the parallel computing literature.

The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of those edges to the forest. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree, and adding all of those edges to the forest. Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest.

Borůvka's algorithm can be shown to take O(log V) iterations of the outer loop until it terminates, and therefore to run in time O(E log V), where E is the number of edges, and V is the number of vertices in G (assuming $E \geq V$). In planar graphs, and more generally in families of graphs closed under graph minor operations, it can be made to run in linear time, by removing all but the cheapest edge between each pair of components after each stage of the algorithm.

k-opt heuristic, or Lin–Kernighan heuristics

The Lin–Kernighan heuristic is a special case of the *V*-opt or variable-opt technique. It involves the following steps:

- Given a tour, delete *k* mutually disjoint edges.
- Reassemble the remaining fragments into a tour, leaving no disjoint sub-tours (that is, don't connect a fragment's endpoints together). This in effect simplifies the TSP under consideration into a much simpler problem.
- Each fragment endpoint can be connected to 2k − 2 other possibilities: of 2k total fragment endpoints available, the two endpoints of the fragment under consideration are disallowed. Such a constrained 2*k*-city TSP can then be solved with brute force methods to find the least-cost recombination of the original fragments.

The most popular of the *k*-opt methods are 3-opt, as introduced by Shen Lin of Bell Labs in 1965. A special case of 3-opt is where the edges are not disjoint (two of the edges are adjacent to one another). In practice, it is often possible to achieve substantial improvement over 2-opt without the combinatorial cost of the general 3-opt by restricting the 3-changes to this special subset where two of the removed edges are adjacent. This so-called two-and-a-half-opt typically falls roughly midway between 2-opt and 3-opt, both in terms of the quality of tours achieved and the time required to achieve those tours.

Results reported by Team Concorde

A – Single Optimal Heuristic for different set of problems

Heuristic used – X-Heuristic Lin Kernighan Method

Effect to be compared – Size of data vs Solving time/ iterations

Sr No	Problem Case	Optimal Tour Length	Min. Spanning Tree Size	Remarks
1	Berlin52- 52 Nodes	7,542	6,078	LPs solved – 2, Time taken – 0.02 seconds
2	Pr76 – 76 nodes	1,08,159	87,217	LPs solved – 12, Time taken –1.22 seconds
3	Rd100 – 100 nodes	7,910	6,962	LPs solved – 4, Time taken –0.12 seconds
4	TS225 – 225 Nodes	1,26,643 units	1,12,000 units	LPs solved – 7, Time taken – 439.78 seconds
5	Lin318 – 318 Nodes	42,029	37,906	LPs solved – 10, Time taken – 52.2 seconds

B - Solving Same Case with multiple heuristics to compare optimal tour results

1. Data Set – Wi29 – 29 Nodes

Sr No	Heuristic/ Edge Norm selected	Tour Length	Remarks
1	Min. Spanning Tree	21,753	
2	Greedy Algorithm	39,691	
3	Boruvka Tour	32,225	
4	Quick Boruvka	33,239	
5	Nearest Neighbour	32,164	
6	Lin Kernighan	27,603	*Optimal Tour

2. Data Set – Pr76 – 76 Nodes

Sr No	Heuristic/ Edge Norm selected	Tour Length	Remarks
1	Min. Spanning Tree	87,217	
2	Greedy Algorithm	1,40,349	
3	Boruvka Tour	1,40,574	
4	Quick Boruvka	1,27,137	
5	Nearest Neighbour	1,30,921	
6	Lin Kernighan	1,08,159	*Optimal Tour

3. Data Set – Rd100 – 100 Nodes

Sr No	Heuristic/ Edge Norm selected	Tour Length	Remarks
1	Min. Spanning Tree	6,962	
2	Greedy Algorithm	9,224	
3	Boruvka Tour	9,518	
4	Quick Boruvka	9,072	
5	Nearest Neighbour	9,423	
6	Lin Kernighan	7,910	*Optimal Tour

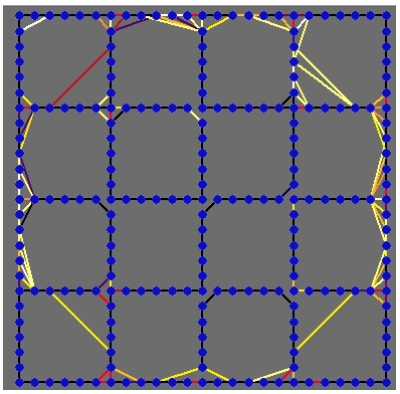
4. Data Set – Att532 – 532 nodes

Sr No	Heuristic/ Edge Norm selected	Tour Length	Remarks
1	Min. Spanning Tree	75,872	
2	Greedy Algorithm	1,05,591	
3	Boruvka Tour	1,06,673	
4	Quick Boruvka	1,06,269	
5	Nearest Neighbour	1,04,684	
6	Lin Kernighan	86,729	*Optimal Tour

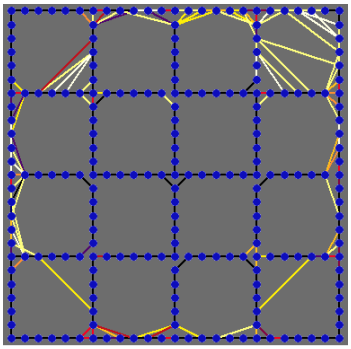
Tour Visualisation offered by Concorde Solver

Visualisation of tour as given by the graphic window of Concorde Solver-

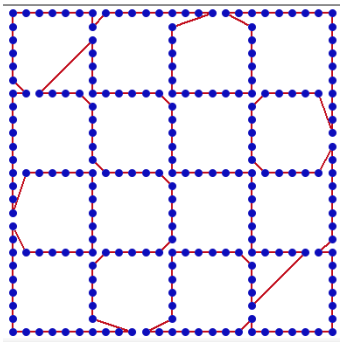
A. On Initialisation, alogorithm defines definite and probabilistic paths. Darker shades indicate higher probability of selection of the path in final optimal tour.



B. Each probabilistic path is solved and gain for each alternate path is computed. Path with maximum gain, and thus, minimum distance between nodes, is selected in trip. In next iteration, this path is made fixed with black shade.



C. Final tour is then shown console and total tour length along with solver statistics are reported in output window.



Concluding Remarks & Further research scope

References

1. https://en.wikipedia.org/wiki/Travelling_salesman_problem
2. https://www.math.uwaterloo.ca/tsp/concorde/index.html
3. Concorde Solver Documentation available on the internet