Heuristics for the Traveling Salesman Problem

Christian Nilsson
Linköping University
chrni794@student.liu.se

Abstract

The traveling salesman problem is a well known optimization problem. Optimal solutions to small instances can be found in reasonable time by linear programming. However, since the TSP is NP-hard, it will be very time consuming to solve larger instances with guaranteed optimality.

Setting optimality aside, there's a bunch of algorithms offering comparably fast running time and still yielding near optimal solutions.

1. Introduction

The traveling salesman problem (TSP) is to find the shortest hamiltonian cycle in a graph. This problem is NP-hard and thus interesting. There are a number of algorithms used to find optimal tours, but none are feasible for large instances since they all grow exponentially.

We can get down to polynomial growth if we settle for near optimal tours. We gain speed, speed and speed at the cost of tour quality. So the interesting properties of heuristics for the TSP is mainly speed and closeness to optimal solutions.

There are mainly two ways of finding the optimal length of a TSP instance. The first is to solve it optimally and thus finding the length. The other is to calculate the Held-Karp lower bound, which produces a lower bound to the optimal solution (see section 7). This lower bound is the de facto standard when judging the performance of an approximation algorithm for the TSP.

The heuristics discussed here will mainly concern the Symmetric TSP, however some may be modified to handle the Asymmetric TSP. When I speak of TSP I will refer to the Symmetric TSP.

2. Approximation

Solving the TSP optimally takes to long, instead one normally uses approximation algorithms, or heuristics. The difference is approximation algorithms give us a guarantee as to how bad solutions we can get. Normally specified as c times the optimal value.

The best approximation algorithm stated is that of Sanjeev Arora[9]. The algorithm guarantees a $(1+1/c)$-approximation for every $c > 1$. It is based om geometric partitioning and quad trees. Although theoretically c can be very large, it will have a negative effect on its running time ($O(n(log_2 n)^{O(c)})$ for two-dimensional problem instances).

3. Tour Construction

Tour construction algorithms have one thing in commmon, they stop when a solution is found and never tries to improve it. The best tour construction algorithms usually gets within 10-15% of optimality.

3.1. Nearest Neighbor

This is perhaps the simplest and most straightforward TSP heuristic. The key to this algorithm is to always visit the nearest city.

Nearest Neighbor, $O(n^2)$

1. Select a random city.
2. Find the nearest unvisited city and go there.
3. Are there any unvisitied cities left? If yes, repeat step 2.
4. Return to the first city.

The Nearest Neighbor algorithm will often keep its tours within 25% of the Held-Karp lower bound [1].

3.2. Greedy

The Greedy heuristic gradually constructs a tour by repeatedly selecting the shortest edge and adding it to the tour as long as it doesn't create a cycle with less than N edges, or increases the degree of any node to more than 2. We must not add the same edge twice of course.

Greedy, $O(n^2 log_2(n))$

1. Sort all edges.
2. Select the shortest edge and add it to our tour if it doesn't violate any of the above constraints.
3. Do we have N edges in our tour? If no, repeat step 2.

The Greedy algorithm normally keeps within 15-20% of the Held-Karp lower bound [1].

3.3. Insertion Heuristics

Insertion heuristics are quite straighforward, and there are many variants to choose from. The basics of insertion heuristics is to start with a tour of a subset of all cities, and then inserting the rest by some heuristic. The initial subtour is often a triangle or the convex hull. One can also start with a single edge as subtour.

Nearest Insertion, $O(n^2)$

1. Select the shortest edge, and make a subtour of it.
2. Select a city not in the subtour, having the shortest distance to any one of the cities in the subtoor.
3. Find an edge in the subtour such that the cost of inserting the selected city between the edge's cities will be minimal.
4. Repeat step 2 until no more cities remain.

Convex Hull, $O(n^2 log_2(n))$

1. Find the convex hull of our set of cities, and make it our initial subtour.
2. For each city not in the subtour, find its cheapest insertion (as in step 3 of Nearest Insertion). Then chose the city with the least cost/increase ratio, and insert it.
3. Repeat step 2 until no more cities remain.

3.4. Christofides

Most heuristics can only guarante a worst-case ratio of 2 (i.e. a tour with twice the length of the optimal tour). Professor Nicos Christofides extended one of these algorithms and concluded that the worst-case ratio of that extended algorithm was 3/2. This algorithm is commonly known as Christofides heuristic.

Original Algorithm (Double Minimum Spanning Tree), worst-case ratio 2, $O(n^2 log_2(n))$

1. Build a minimal spanning tree (MST) from the set of all cities.
2. Duplicate all edges, we can now easily construct an Euler cycle.
3. Traverse the cycle, but do not visit any node more than once, taking shortcuts when a node has been visited.

Christofides Algorithm, worst-case ratio 3/2, $O(n^3)$

1. Build a minimal spanning tree from the set of all cities.
2. Create a minimum-weight matching (MWM) on the set of nodes having an odd degree. Add the MST together with the MWM.
3. Create an Euler cycle from the combined graph, and traverse it taking shortcuts to avoid visited nodes.

The main difference is the additional MWM calculation. This part is also the most time consuming one, having a time complexity of $O(n^3)$. Tests have shown that Christofides' algorithm tends to place itself around 10% above he Held-Karp lower bound.

For more information on tour construction heuristics see [2].

4. Tour Improvement

Once a tour has been generated by some tour construction heuristic, we might wish to improve that solution. There are several ways to do this, but the most common ones are the 2-opt and 3-opt local searches. Their performances are somewhat linked to the construction heuristic used.

Other ways of improving our solution is to do a tabu search using 2-opt and 3-opt moves. Simulated annealing also use these moves to find neighboring solutions. Genetic algorithms generally use the 2-opt move as a means of mutating the population.

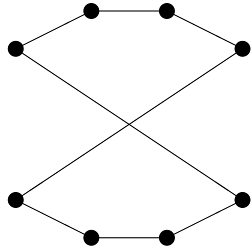

Figure 1. A 2-opt move

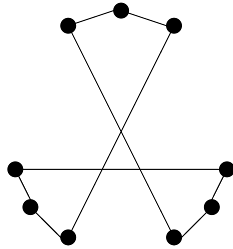

Figure 2. A 3-opt move

4.1. 2-opt and 3-opt

The 2-opt algorithm basically removes two edges from the tour, and reconnects the two paths created. This is often refered to as a 2-opt move. There is only one way to reconnect the two paths so that we still have a valid tour (figure 1). We do this only if the new tour will be shorter. Continue removing and reconnecting the tour until no 2-opt improvements can be found. The tour is now 2-optimal.

The 3-opt algorithm works in a similar fashion, but instead of removing two edges we remove three. This means that we have two ways of reconnecting the three paths into a valid tour[1](figure 2 and figure 3). A 3-opt move can actually be seen as two or three 2-opt moves.

We finish our search when no more 3-opt moves can improve the tour. If a tour is 3-optimal it is also 2-optimal [5].

If we look at the tour as a permutation of all the cities, a 2-opt move will result in reversing a segment of the permutation. A 3-opt move can be seen as two or three segment reversals.

Running the 2-opt heuristic will often result in a tour with a length less than 5% above the Held-Karp bound. The improvements of a 3-opt heuristic will usually give us a tour about 3% above the Held-Karp bound [1].

[1]not including the connections being identical to a single 2-opt move

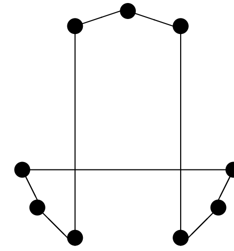

Figure 3. A 3-opt move

4.2. Speeding up 2-opt and 3-opt

When talking about the complexity of these k-opt algorithms, one tends to omit the fact that a move can take up to $O(n)$ to perform (see section 4.8).

A naive implementation of 2-opt runs in $O(n^2)$, this involves selecting an edge (c_1, c_2) and searching for another edge (c_3, c_4), completing a move only if $dist(c_1, c_2) + dist(c_3, c_4) > dist(c_2, c_3) + dist(c_1, c_4)$.

An observation made by Steiglitz and Weiner tells us that we can prune our search if $dist(c1, c2) > dist(c2, c3)$ does not hold. This means that we can cut a large piece of our search by keeping a list of each city's closest neighbors. This extra information will of course take extra time to calculate ($O(n^2 log_2)$), and also needs a substantial amount of space. Reducing the number of neighbors in out lists will alow this idea to be put in practice.

By keeping the m nearest neighbors of each city we can improve the complexity to $O(mn)$. But we still have to find the nearest neighbors for each city. Luckily this information is static for each problem instance, so we need only do this calulation once and can reuse it for any subsequent runs on that particular problem.

This speedup will remove the 2-optimality guarantee, but the loss in tour quality is small if we choose m wisely. Choosing $m = 20$ will probably reduce the quality by little or nothing. Choosing $m = 5$ will give us a very nice increase of speed at the cost of some quality[1].

4.3. k-opt

We don't necessarilly have to stop at 3-opt, we can continue with 4-opt and so on, but each of these will take more and more time and will only yield a small improvement on the 2- and 3-opt heuristics.

Mainly one 4-opt move is used, called "the crossing bridges" (Figure 4). This particular move can not be sequentially constructed using 2-opt moves. For this to be possible two of these moves would have to be illegal [5].

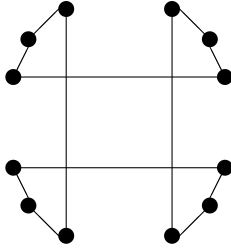

Figure 4. The double-bridge move

4.4. Lin-Kernighan

Lin and Kernighan constructed an algorithm making it possible to get within 2% of the Held-Karp lower bound. The Lin-Kernighan algorithm (LK) is a variable k-opt algorithm. It decides which k is the most suitable at each iteration step. This makes the algorithm quite complex, and few have been able to make improvements to it. For a more in-depth study of the LK algorithm and possible improvements, see [5].

The time complexity of LK is approximately $O(n^{2.2})$ [5], making it slower than a simple 2-opt implementation. However the results are much better with LK, and given improvements suggested by Helsgaun [5], it will probably not be that much slower.

4.5. Tabu-Search

A neighborhood-search algorithm searches among the neighbors of a candidate to find a better one. When running a neighborhood-search on the TSP, neighboring moves are often normal 2-opt moves.

A problem with neighborhood searches is that one can easily get stuck in a local optimum. This can be avoided by using a tabu-search.

The tabu-search will allow moves with negative gain if we can not find a positive one. By allowing negative gain me may end up running in circles, as one move may counteract the previous. To avoid this the tabu-search keeps a tabu list containing illegal moves. After moving to a neighboring solution the move will be put on the tabu-list and will thus never be applied again unless it improves our best tour or the tabu has been pruned from our list.

There are several ways of implementing the tabu list. One involves adding the two edges being removed by a 2-opt move to the list. A move will then be considered tabu if it tries to add the same pair of edges again. Another way is to add the shortest edge removed by a 2-opt move, and then making any move involving this edge tabu. Other methods keep the endpoints of each move, making a move tabu if it uses these endpoints [1].

A big problem with the tabu search is its running time. Most implementations for the TSP will take $O(n^3)$ [1], making it far slower than a 2-opt local search. Given that we use 2-opt moves, the length of our tours will be slightly better than that of a standard 2-opt search.

4.6. Simulated Annealing

Simulated Annealing (SA) has been successfully adapted to give approximate solutions for the TSP. SA is basically a randomized local search algorithm allowing moves with negative gain.

A baseline implementation of SA for the TSP is presented in [1]. They use 2-opt moves to find neighboring solutions. Not surprisingly the resulting tours are comparable to those of a normal 2-opt algorithm. Better results can be obtained by increasing the running time of the SA algorithm, showing results comparable to the LK algorithm.

Due to the 2-opt neighborhood, this particular implementation takes $O(n^2)$ with a large constant of proportionality[1]. Some speed-ups are necessary to make larger instances feasible to run, and also to make it competitive to other existing approximation algorithms.

The first thing to improve is the 2-opt neighborhood. By keeping neighborhood lists as described in section 4.2, one can cut down this part dramatically. By incorporating neighborhood lists and other improvments mentioned in [1], the algorithm can actually compete with the LK algorithm.

4.7. Genetic Algorithms

Continuing on the randomized path will take us to Genetic Algorithms (GA). GAs work in a way similar to nature. An evolutionary process takes place within a population of candidate solutions.

A basic GA starts out with a randomly generated population of candidate solutions. Some (or all) candidates are then mated to produce offspring and some go through a mutating process. Each candidate has a fitness value telling us how good they are. By selecting the most fit candidates for mating and mutation the overall fitness of the population will increase.

Applying GA to the TSP involves implementing a crossover routine, a measure of fitness, and also a mutation routine. A good measure of fitness is the actual length of the candidate solution.

Different approaches to the crossover and mutation routines are discussed in[1]. Some implementations have shown good results, even better than the best of several LK runs. But as with both the Tabu-search and the SA algorithm running time is an issue.

4.8. Tour Data Structure

The implementation of a k-opt algorithm will involve reversing segments of the tour. This reversal can take from $O(log_2(n))$ to $O(n)$, depending on your choice of data structure for the tour.

If you plan to use a vector as data structure, a single reversal will take $O(n)$, and a simple lookup like finding the previous or next city will be possible in $O(1)$. The $O(n)$ complexity for reversing segments is very bad for large instances. It will totally dominate the running time on instances with more than 10^3 cities [3].

Another data structure is needed for instances with more than 10^3 cities. The answer to our problem is a two-level tree [3]. They are more difficult to implement, but will give us the advantage of $O(\sqrt{n})$ per reversal. A lookup is still $O(1)$. These two-level trees will do the job for problem sizes of up to 10^6 cities [3].

This is where splay trees enter. Having an amortised worst-case time complexity of $O(log_2(n))$ for both moves and lookups [3], it will outperform both the previous structures for large problem instances. The implementation is a bit tricky, but will be well worth it.

A mixture of these three representations would be the best choice. Using arrays for problems with less than 10^3 cities, two-level trees for instances with up to 10^6 cities and finally splay trees for the largest instances.

5. Branch & Bound

Branch & Bound algorithms are often used to find optimal solutions for combinatorial optimiziation problems. The method can easily be applied to the TSP no matter if it is the Asymmetric TSP (ATSP) or the Symmetric TSP (STSP).

A method for solving the ATSP using a Depth-First Branch & Bound (DFBnB) algorithm is studied in [7]. The DFBnB starts with the original ATSP and solves the Assignment Problem (AP). The assignment problem is to connect each city with its nearest city such that the total cost of all connections is minimized. The AP is a relaxation of the ATSP, thus acting as a lower bound to the optimal solution of the ATSP.

We have found an optimal solution to the ATSP if the solution to the AP is a complete tour. If the solution is not a complete tour we must find a subtour within the AP solution and exclude edges from it. Each exclusion will branch our search.

The actual cutting is done by first choosing α as the length of the best solution currently known. All branches are cut if the cost of the AP solution exceeds α.

The DFBnB solves the ATSP with optimality. However, it can also be used as an approximation algorithm by adding extra constraints. One is to only remove the most costly edges of a subtour, thus decreasing the branching factor.

6. Ant Colony Optimization

Researchers are often trying to mimic nature when solving complex problems, one such example is the very successful use of Genetic Algorithms. Another interesting idea is to mimic the movements of ants.

This idea has been quite successful when applied to the TSP, giving optimal solutions to small problems quickly [8]. However, as small as an ant's brain might be, it is still far to complex to simulate completely. But we only need a small part of their behaviour to solve our problem.

Ants leave a trail of pheromones when they explore new areas. This trail is meant to guide other ants to possible food sources. The key to the success of ants is strength in numbers, and the same goes for ant colony optimization.

We start with a group of ants, typically 20 or so. They are placed in random cities, and are then asked to move to another city. They are not allowed to enter a city already visited by themselves, unless they are heading for the completion of our tour. The ant who picked the shortest tour will be leaving a trail of pheromones inversely proportional to the length of the tour.

This pheromone trail will be taken in account when an ant is choosing a city to move to, making it more prone to walk the path with the strongest pheromone trail. This process is repeated until a tour being short enough is found. Consult [8] for more detailed information on ant colony opimization for the TSP.

7. The Held-Karp Lower Bound

A common way of measuring the performance of TSP heuristics is to compare its results to the Held-Karp (HK) lower bound. This lower bound is actually the solution to the linear programming relaxation of the integer programming formulation of the TSP. The solution can be found in polynomial time by us-

ing the Simplex method and a polynomial constraint-separation algorithm[4].

A HK lower bound averages about 0.8% below the optimal tour length [4]. But its guaranteed lower lower bound is only 2/3 of the optimal tour.

It is not feasible to compute the solution exactly for very large instances using this method. Instead Held and Karp has proposed an iterative algorithm to approximate the solution. It involves computing a large amount of minimum spanning trees (each taking $O(nlog_2(n))$). The iterative version of the HK will often keep within 0.01% of the optimal HK lower bound [4].

8. Conclusion

Selecting an approximation algorithm for the TSP involves several choices. Do we need a solution with less than 1% excess over the of the Held-Karp bound, or do we settle with 4%? The difference in running time can be substantial.

The Lin-Kernighan algorithm will most likely be the best candidate in most situations, leaving 2-opt as a faster alternative.

It all comes down to one key detail, speed.

References

[1] D.S. Johnson and L.A. McGeoch, "The Traveling Salesman Problem: A Case Study in Local Optimization", November 20, 1995.

[2] D.S. Johnson and L.A. McGeoch, "Experimental Analysis of Heuristics for the STSP", *The Traveling Salesman Problem and its Variations*, Gutin and Punnen (eds), Kluwer Academic Publishers, 2002, pp. 369-443.

[3] M.L Fredman, D.S. Johnson, L.A. McGeoch, G. Ostheimer, "Data Structures For Traveling Salesmen", *J. ALGORITHMS 18*, 1995, pp. 432-479.

[4] D.S. Johnson, L.A. McGeoch, E.E. Rothberg, "Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound" *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 1996, pp. 341-350.

[5] K. Helsgaun, "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic", Department of Computer Science, Roskilde University.

[6] D. Applegate, W. Cook and A. Rohe, "Chained Lin-Kernighan for large traveling salesman problems", July 27, 2000.

[7] W. Zhang, "Depth-First Branch-and-Bound versus Local Search: A Case Study", Information Sciences Institute and Computer Science Department University of Southern California.

[8] M. Dorigo, L.M. Gambardella, "Ant Colonies for the Traveling Salesman Problem", Universit Libre de Bruxelles, Belgium, 1996.

[9] S. Arora, "Polynomial Time Approximation Schemes for Euclidian Traveling Salesman and Other Geometric Problems", *Journal of the ACM*, Vol. 45, No. 5, September 1998, pp. 753-782.