Integer and Combinatorial Optimization: Improving Tours

John E. Mitchell

Department of Mathematical Sciences
RPI, Troy, NY 12180 USA

February 2019
The Traveling Salesman Problem

Given a graph $G = (V, E)$, a Hamiltonian tour is a cycle that contains all of the nodes.

If each edge $e$ has a length $d_e$, the traveling salesman problem is to find the tour with least total length.
Finding a good solution

Methods for finding a good solution to the TSP and other combinatorial optimization problems can often be broken into two stages:

1. construct a feasible solution
2. use local improvement methods to get a better solution.

For minimum spanning tree, an optimal solution can be constructed using a greedy algorithm, so the second stage is not necessary.

For the traveling salesman problem, greedy algorithms or the Christofides heuristic may well give non-optimal solutions.
Finding a good solution

Methods for finding a good solution to the TSP and other combinatorial optimization problems can often be broken into two stages:

1. construct a feasible solution
2. use local improvement methods to get a better solution.

For minimum spanning tree, an optimal solution can be constructed using a greedy algorithm, so the second stage is not necessary.

For the traveling salesman problem, greedy algorithms or the Christofides heuristic may well give non-optimal solutions.
Using a greedy algorithm

For example, we could construct a tour using a greedy algorithm by initializing with one vertex, choosing the cheapest edge leaving that vertex that still allows the completion of a tour, and iterating from the new endpoint.

This can be arbitrarily bad in general graphs, as in the example below.

![Graph with vertex numbers 1, 2, 3, 4 and edge lengths 1, 2, 2, 1, 100]
Using a greedy algorithm

For example, we could construct a tour using a greedy algorithm by initializing with one vertex, choosing the cheapest edge leaving that vertex that still allows the completion of a tour, and iterating from the new endpoint.

This can be arbitrarily bad in general graphs, as in the example below.

```
1
\rightarrow
2
\rightarrow
3

1
\rightarrow
4
```

construct greedy tour
Using a greedy algorithm

For example, we could construct a tour using a greedy algorithm by initializing with one vertex, choosing the cheapest edge leaving that vertex that still allows the completion of a tour, and iterating from the new endpoint.

This can be arbitrarily bad in general graphs, as in the example below.

```
construct greedy tour
```
Using a greedy algorithm

For example, we could construct a tour using a greedy algorithm by initializing with one vertex, choosing the cheapest edge leaving that vertex that still allows the completion of a tour, and iterating from the new endpoint.

This can be arbitrarily bad in general graphs, as in the example below.

construct greedy tour
Finding a good solution

Using a greedy algorithm

For example, we could construct a tour using a greedy algorithm by initializing with one vertex, choosing the cheapest edge leaving that vertex that still allows the completion of a tour, and iterating from the new endpoint.

This can be arbitrarily bad in general graphs, as in the example below.

![Graph](image)

greedy tour, cost 103
Using a greedy algorithm

For example, we could construct a tour using a greedy algorithm by initializing with one vertex, choosing the cheapest edge leaving that vertex that still allows the completion of a tour, and iterating from the new endpoint.

This can be arbitrarily bad in general graphs, as in the example below.

![Graph](image_url)

optimal tour, cost 6
Greedy with Euclidean distances

In planar graphs with Euclidean distances, greedy can be no better than the Christofides heuristic:

the path though the first $|V| - 1$ vertices is a spanning tree and we then pair up the two endpoints.
We now look at methods to improve tours.

In 2-change, we look for a pair of edges \((r, s)\) and \((u, v)\) which we can replace with edges \((r, v)\) and \((s, u)\), giving a better tour.

We need to make sure the new solution really is a tour, and not two subtours.

In planar graphs with Euclidean distances, 2-change corresponds exactly to removing edge crossings.
An example of 2-change

![Graph](https://via.placeholder.com/150)

We replace the edges \((3, 7)\) and \((4, 6)\) by the edges \((3, 4)\) and \((6, 7)\), giving a new tour which is shorter because of the triangle inequality. Note that if we’d connected the endpoints differently, as \((3, 6)\) and \((4, 7)\), we would have broken the tour into two subtours.

A solution is called **2-opt** if there is no 2-change move that will improve the solution.
An example of 2-change

We replace the edges \((3, 7)\) and \((4, 6)\) by the edges \((3, 4)\) and \((6, 7)\), giving a new tour which is shorter because of the triangle inequality. Note that if we’d connected the endpoints differently, as \((3, 6)\) and \((4, 7)\), we would have broken the tour into two subtours.

A solution is called \textbf{2-opt} if there is no 2-change move that will improve the solution.
An example of 2-change

We replace the edges \((3, 7)\) and \((4, 6)\) by the edges \((3, 4)\) and \((6, 7)\), giving a new tour which is shorter because of the triangle inequality. Note that if we’d connected the endpoints differently, as \((3, 6)\) and \((4, 7)\), we would have broken the tour into two subtours.

A solution is called **2-opt** if there is no 2-change move that will improve the solution.
An example of 2-change

We replace the edges (3, 7) and (4, 6) by the edges (3, 4) and (6, 7), giving a new tour which is shorter because of the triangle inequality. Note that if we’d connected the endpoints differently, as (3, 6) and (4, 7), we would have broken the tour into two subtours.

A solution is called **2-opt** if there is no 2-change move that will improve the solution.
3-change

We can extend 2-change out to more edges. In particular, 3-change requires finding three edges where we rearrange the connections between the corresponding six vertices. 3-change can improve on 2-change.
3-change

We can extend 2-change out to more edges.

In particular, 3-change requires finding three edges where we rearrange the connections between the corresponding six vertices. 3-change can improve on 2-change.

![Diagram of an improved tour](image)
3-opt

In practice, 3-change is powerful, often getting very close to optimality, and doing considerably better than 2-change. 3-change can be run starting with many different initial solutions. A solution is called 3-opt if there is no 3-change move that will improve the solution.
We can push this further, looking to change $k$ edges. This will lead to better solutions, but the time required to find a $k$-opt solution increases as $k$ increases.

In practice, searching for 3-opt solutions seems to work well.
Randomize greedy: GRASP

greedy randomized adaptive search procedure.

Instead of making the greediest choice at each iteration, makes one of top 5 greediest choices.