2-OPT Optimization Applied to TSP

Kai Weight, Ammon Warnick, Jacob Walker

Dr. Tony Martinez
CS 312: Algorithm Design and Analysis
April 9th, 2021

Abstract:

2-opt is a simple local search algorithm for solving the traveling salesman problem (Wikipedia). In this paper we analyze our own version of 2-opt optimization developed to solve the Traveling Salesperson Problem (TSP). We compare its performance to greedy and branch and bound algorithms on the same dataset.

Introduction:

The traveling salesperson problem is a well-known analytical problem that deals with the optimization of distance between certain points, also contextually called “cities”. The idea behind the algorithm is that you, a travelling salesperson, must visit a certain collection of cities and return home. Not wanting to waste time or resources on unnecessary transport, you’d like to find the shortest complete circuit between all of the cities.

There are several ways to attempt optimization on this problem, including a greedy approach, which would involve always travelling to the nearest city. However, this won’t always result in the optimal solution. More thoughtful approaches, such as the branch and bound algorithm, take into account multiple potential circuits in order to find the shortest path.

Another approach is to optimize the greedy approach with a 2-OPT algorithm. It is this approach that we explored and analyzed.

Greedy Algorithm Explanation & Complexity:

The greedy algorithm has O(n^2) time and O(n) space. The greedy algorithm works by visiting a city, moving to the next nearest city, and then the next nearest, and so on. Because of this, you will only have to visit and record each city once, giving you O(n). However, you’ll be repeating that step for as long as you have paths to iterate through, meaning you’ll repeat those n steps n times, giving you O(n^2).

Greedy Algorithm Code:

```
def greedy(self, time_allowance=60.0, startNode=0):
   results = {}
   cities = self._scenario.getCities().copy()
   count = 0
   start_time = time.time()
   route = [cities[startNode]]
   cities.pop(startNode)
   while time.time() - start_time < time_allowance:
       if len(cities) == 0:
```

```
                break
            best = 0
            m = math.inf
            for i in range(len(cities)):
                new_route = [route[-1], cities[i]]
                short = TSPSolution(new_route)
                if short.cost < m:
                    m = short.cost
                    best = i
            route.append(cities[best])
            cities.pop(best)
        bssf = TSPSolution(route)
        end_time = time.time()
        results['cost'] = bssf.cost
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None
        return results
```

Explanation of 2-OPT Optimization:

We became interested in the 2-OPT method after finding an explanation of it on this page. The idea for this method is that tours that have crossed edges aren't optimal. The algorithm will consider every possible 2-edge swap, and if the resulting tour is an improvement, it will maintain the change. As explained there, each k-OPT iteration takes O(n^k) time. This means that for our 2-OPT algorithm, it is O(n^2) time.

Advantages of 2-OPT:

The 2-OPT is faster than the Branch and Bound algorithm and it can calculate a shorter path than what the Greedy will give us. This algorithm is also lightweight because the space complexity is O(n^3).

Disadvantages of 2-OPT:

We know that the 2-OPT can't give us the most optimal path available.

2-OPT Code:

```
def fancy(self, time_allowance=60.0):
    results = {}
    bssf = self.greedy(time_allowance=time_allowance)['soln']
```

```
best = bssf.getListOfCities()
route = best
start_time = time.time()
count = 0

improve = True
while time.time() - start_time < time_allowance and improve:
    improve = False
    for i in range(1, len(route) - 2):
        for j in range(i + 1, len(route)):
            if j - i == 1: continue
            new_route = route[:]
            new_route[i:j] = route[j - 1:i - 1:-1]
            newSol = TSPSolution(new_route)
            if newSol._costOfRoute() < bssf._costOfRoute():
                bssf = newSol
                best = new_route
                count += 1
                improve = True
    route = best
end_time = time.time()
results['cost'] = bssf.cost
results['time'] = end_time - start_time
results['count'] = count
results['soln'] = bssf
results['max'] = None
results['total'] = None
results['pruned'] = None
return results
```

Results (Table):

	Random		Greedy			Branch & Bound			Our Algorithm		
# Cities	Time (Sec)	Path Length	Time (sec)	Path Length	% of Random	Time (Sec)	Path Length	% of Greedy	Time (sec)	Path Length	% Greedy
15	.000433	23343	.000733	16223	.69	5.7	9090	.38	.050274	12055	.74
30	.133483	40918	.002148	20948	.51	600	23307	.57	.358088	16266	.77
60	23.866238	78780	0.007743	29750	.38	600	31360	0	3.166986	25249	0
100	60	inf	0.018255	inf	inf	600	38154	0	2.1234588	34606	0
200	60	inf	0.069264	inf	inf	600	168650	0	60	56923	0

Analysis of Results:

When comparing our 2-OPT algorithm to Greedy, we can see that our algorithm is able to give a more optimal path in a similar time-frame. When compared to the Branch and Bound, we realized that there were different pro's and con's for each. We recognized that in order for our Branch and Bound algorithm to be at its full potential, we would need it to run longer than the normal 600 seconds. Although the B&B gives a more optimal path length, it takes a significantly longer time than our own 2-OPT. We decided that our algorithm is most effective when searching at about 30-100 cities. It allows for a quick calculation that gives a reasonable path length. After that, however, we decided the B&B is the best option. These results are what we were expecting according to our theoretical analysis.

Future Work:

During our work we experimented with a version of the algorithm that would use the greedy algorithm to generate more routes that start at different nodes, and seeing if any mutations of those routes would be shorter. This would run much longer than our base algorithm, but with the potential to find shorter routes. It would require some tinkering to see if it could run faster, and this is an area that could be explored in the future.

Another area of exploration could be other k-opt modifications, i.e. 3-opt, 4-opt, etc., to see if they run faster or produce more optimal tours.