Travelling Salesman
Greedy path with 2-OPT and annealing
DD2440 Advanced Algorithms

Kattis submission: 3391882 (Score:35.185162)
Refactored Kattis submission: 3392308 (Score:35.018185)
Grade level: B

Anonymous1 Anonymous2 Marcus Andersson
and8@kth.se

June 2, 2019

1 Introduction

This project has been about creating an approximate solution to the Travelling Salesman Problem. The task included a time limit for how long our program was allowed to run. In order to accommodate for the limited running time we focused on using solutions with low computational cost. We decided to use a path heuristic in combination with local optimization. As these proved relatively simple to implement, we were able to introduce additional heuristics and improvements on to the initial approach.

1.1 Project Outline

Roughly speaking, the more different TSP related materials we researched the more it became clear that more accurate methods usually require more computation time [2, 3]. Thereby, we decided to focus on the ones with least computations while still having reasonable results.

1.1.1 Initial tour

The following heuristics were considered:

- Random
- Nearest insertion
- Nearest neighbor
- Clarke-Wright
- Christofides

Since our plan was to use optimization algorithms to improve the tour, we reasoned that it was feasible to spend less time on the initial tour, thus opting for a faster heuristic. We did not test Christofides or Clarke-Wright, as both normally yield better tours than nearest neighbor, but are significantly slower to compute according to [2].

Random, nearest insertion and nearest neighbor initializations were tested. In our experiments we decided that the greedy nearest neighbour solution provided the best results while using the least amount of CPU time. It is able to provide a tour that was not too close to a local minimum but fast enough to converge during optimization. Nearest insertion, consisting of iteratively adding a city to a previously generated sub-tour, proved to be more costly than greedy, with no real tour improvements. The fully random initialization could help overcome some local minima and provide for potentially better tours in the long run. However, in our experiments it took too long to converge for large instances, as the number of possible permutations increases exponentially with the size of the input.

1.1.2 Local optimization

Initially, only 2-OPT was used in conjunction with the nearest neighbour heuristic for initialization. 2-OPT is the simplest in a series of local optimization techniques known as k-OPT, which rely on finding edges that cross over each-other and swapping the corresponding cities along the tour to provide a locally shorter tour.

Our original 2-OPT used a function to generate two random integers and check if it provides an improvement. An attempt to implement a pseudo-random function was made to increase the speed. Doing this did not show significant

changes to the results. Possibly because the random number generator caused very little delay compared to other parts of the algorithm.

1.1.3 Local minimums and simulated annealing

Only doing 2-OPT makes it likely for the solution to get stuck in a local-minimum [5, 4]. That means that there is no immediate 2-OPT operation that will provide a better result. In order to avoid local-minimums, we added a technique called simulated annealing to our solution [2, 4]. Simulated annealing uses a "temperature" variable, which is lowered in each iteration. A high temperature makes it likely to accept a worse tour, whereas a change that shortens the tour is always accepted. As the temperature gets low, bad swaps become less likely. Different values for the temperature were tested empirically. The value providing the best results was kept. In addition to annealing, we also employed random swaps, when a minimum had been detected and no more swaps could be made.

1.1.4 Re-using tours

Since the temperature was lowered quickly in the program due to time constraints, local-minima were still an issue. Therefore, we introduced a mechanism to modify the tours and pick the best one. This was done until the time ran out. The best tour was stored in its own array before any modifications were made. Then it was copied into another array and randomized a bit. This made it possible to re-use the best solution for finding a new, possibly better, tour.

1.1.5 Nearest neighbour lists

The idea, discussed in [2] and [1], is to only consider a number of closest cities when running the 2-OPT process. This yielded significant improvements for us as the optimization ran significantly faster due to the smaller amount of possible comparisons. We won't get results that are as optimal as they could be, but it allowed us to stop testing random positions in the tour for improvement and instead find the optimal pick each time. Even though this does not give a fully 2-optimal path, it means that we can run the classic 2-OPT algorithm in $O(NM)$ instead of $O(N^2)$. If M is of reasonable size, the difference with pure 2-OPT is relatively small. Additionally it allows us to

detect when we have reached a local minimum, when additional swaps that yield improvements can no longer be found.

2 Implementation

For the implementation we used C++14 and relied on the standard libraries available.

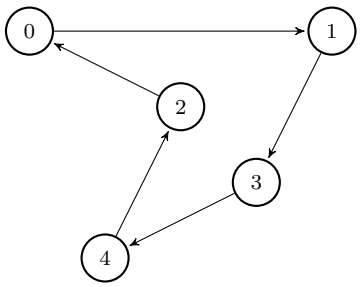
2.1 Data structures

The primary data elements of interest in our solution are the representation of the tour, the distances between the cities as well as a list of nearest neighbours for each city.

2.1.1 Tour and cities

Out of the three, the tour representation is probably the most common one. To create it, we first load each individual city from the input into a small `struct`, named `City`, which contains the following:

1. An identifier(`ID`) for each city. The `ID` corresponds to the order in which the city was read from the input, so the i'th city in the input will have `ID` i. Mainly for convenience.
2. x and y coordinates.
3. The city's position in the tour.



(a) Visual tour representation.

(b) Internal tour representation as an array.

Figure 1: Tour representation.

We store each of these `City` objects in a dynamically allocated array. They could also be stored in a static array as it is known that our problem will never have more than 1000 cities. The tour is represented as an ordered list of integers representing city `ID`s, stored as an array. The arrays are depicted in figure 1 and 2. This pattern makes it convenient to access each city's data from it's position in the tour in O(1) time.

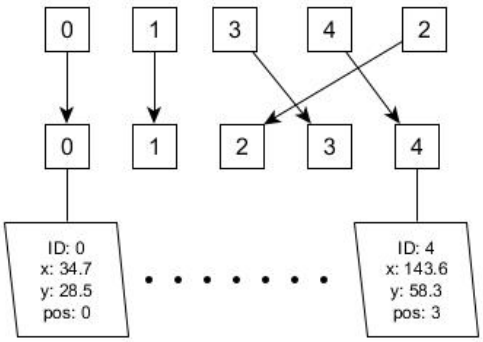


Figure 2: City array

2.1.2 Distance matrix

As euclidean norm was used to determine the distance between two cities, we decided it would be a good idea to store the computed distances between the cities when we first use them. Square root and powers can be surprisingly

expensive to compute. When iterating and doing so repeatedly, we expected this to save us time in the long run as the distances are the primary values of interest.

In order to store the distances we created an $N \times N$ matrix where each cell (i, j) represents a distance between two cities as referenced in the cities array described in the previous section. This allows us conveniently access to the pre-computed distances when considering different positions in the tour, as the tour consists of city `IDs`. The matrix itself is just an interface to a one-dimensional array that is indexed in a special way.

$$\begin{bmatrix} 0 & d_{0,1} & \dots & d_{0,n-1} \\ d_{0,1} & 0 & \dots & d_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ d_{0,n-1} & d_{1,n-1} & \dots & 0 \end{bmatrix}$$

Figure 3: Distance matrix.

2.1.3 Neighbourhood matrix

The final and probably most useful data-structure is a set of neighbourhood lists, stored again as a matrix. In order to do this quickly, we store that information in another matrix of size $N \times M$ to represent the closest cities for each city, where M is the number of other cities stored for each city. Each row i represents the list of cities nearest to city i in ascending order.

2.2 Algorithm

We will use listings of pseudo-code that roughly follow the order of things happening in our implementation. To describe our algorithm, we begin by giving a high level overview of function calls in listing 1. The relevant details of tour generation and 2-OPT will be given in listings 2 and 3 respectively. We also briefly discuss the method we use for swapping in listing 4 as it might also be of some interest.

In our final version of the code, we used an M of size 100, scaled down to $N - 1$ if $N < M$. Additionally we had an initial temperature of 15000, which decreased in time as a polynomial function to the third power of time elapsed.

```
void execution(){
    readInput()
    greedyTour()
    twoOptAnnealing()
    printOutput()
}
```

Listing 1: Algorithm pseudocode

As can be seen in listing 2, the distance and neighbour matrices are initialized alongside the calculation of the initial tour. As the sorting takes $O(nlogn)$ and we iterate over n^2 elements, we get an approximate complexity of $O(n^2logn)$ for this part. This is because we do $O(n + nlogn)$ for each n. Therefore $O(n(n + nlogn)) = O(n^2 + n^2logn) = O(n^2logn)$.

```
void greedyTour() {
    for (i = 0 to N; i++) {
        city_i = cities[i]
        for(j = 0 to N; j++){
            city_j = cities[j]
            distances.set(i,j, distance(city_i, city_j))
            distances.set(j,i, distance(city_i, city_j))
            dist[j] = distance(city_i, city_j)
            city_best = min(distance(city_i, city_j))
        }
        sorted = sort(dist)

        for (k = 0 to M; k++) {
            neighbours.set(i, k, sorted[k])
        }

        tour[i+1] = city_best
        city_best.pos = i+1
        mark city_best as used
    }
}
```

Listing 2: "Greedy" nearest neighbour tour pseudocode

The next listing - the primary body of our work - describes the operations we do during the 2-OPT optimization process. Note that the annealing part was primarily used before we implemented the nearest neighbour lists, so it is mainly an artifact of our work progress. However, in the results section it

will be shown that the annealing still yields an improvement. An additional minima breaking section was added in case it was detected that no swaps were made. In some cases doing random swaps helps break out of a local minimum and yields in a better total result.

The change calculation mentioned in listing 3 can be done in O(1) time, by considering the edges that change by doing a swap. Note that i and j in the distance equation 1 below, refer to cities at corresponding positions in the tour, instead of tour indexes themselves.

$$\begin{aligned} change =& distance(i-1,j) + distance(i,j+1) \\ & - distance(i-1,i) - distance(j,j+1) \end{aligned} \tag{1}$$

```
1  void doSwap(i, j){
2      if (j - i > 2) {
3          copy([tour[i to j], newTour[i to j])
4          for (k = j to i; k--) {
5              a = i + (j - k)
6              tour[a] = newTour[k] //Reverse path from i to j
7              cities[tour[a]].pos = a // Update position
8          }
9      } else {
10         swap: tour[i] <-> tour[j]
11         cities[tour[i]].pos = i
12         cities[tour[j]].pos = j
13     }
14 }
```

Listing 4: 2-OPT swap pseudocode

As seen in listing 4, a 2-OPT swap in our representation corresponds to a reversal of a section between two indexes of the tour. Note that our method of computing does not allow a swap to be defined using both the last and first position of the tour simultaneously. However, as all other combinations are allowed, we can still consider all possible 2-OOPT modifications to a tour. As in the worst case we will swap up to N-1 positions in a single swap it has a complexity bound of $O(n)$. This could be improved by using more efficient data structures discussed in the future works section.

In listing 4 we can also see the updating of the position variables tied to the city objects. This allows constant time look-up of position during optimization. When we do 2-OPT, the neighbour list is determined in terms of cities, not

```
void twoOptAnnealing() {
    while (dt < 1.99) {
        dt = time()
        bestChange = 0
        for (i = 0 to N-1; i++) {
            for (j in neighbours[i]) {
                swap i, j ,st i < j
                change = calculateChange(i, j)
                if (change < bestChange) {
                    //Find the the best availabe move, corresponds to
                        the most negative change
                    besti = i, bestj = j, bestChange = change
                }
            }
        }

        if (bestChange < 0) {
            //Do the best available move
            doSwap(besti, bestj)
        } else {
            //Do a bad move with some probability if no good move
                found
            random i, j ,st: i < j
            change = calculateChange(i, j)
            temperature = startTemp * (1.0f - dt / 2)^3
            if (probability(dist + change) > criteria) {
                copy(tour, bestTour) //Save best tour so far
                doSwap(i, j)
            }
        }

        if (no swaps done earlier) {
            // Local minima, do r
            copy(tour, bestTour) //Save best tour so far
            do doSwap(i,j) for a # of times:
        }
    }
}
```

Listing 3: 2-OPT pseudocode

tour positions. However, to calculate the change of a given swap, we need to know where that city is in the tour as the cities in surrounding positions of the current tour are required to calculate the change. A naive implementation might search the list in $O(n)$ or $O(logn)$ for the city id, however by updating the position during the swapping, we can find the tour position in $O(1)$, with minimal impact to the swap complexity.

3 Results

In order to test our implementation in a local environment, we used 5 different TSP instances from the VLSI data sets [6] ranging from 131 to 1083 cities; as well as the one provided by Kattis, consisting of 10 cities. Each data set has an associated optimal tour, thus allowing for comparison with our implementation's results.

Starting from a naive approach, we explored the approximation ratio provided by the nearest neighbors heuristic - greedy - compared to the optimal tour distance. These results can be seen from Table 1.

Input size	Optimum	Greedy	Approx. factor
1083	3558	4563	1.282
662	2513	3124	1.243
436	1443	1739	1.205
237	1019	1343	1.318
131	564	704	1.248
10	276	323	1.170

Table 1: Results of the greedy - nearest neighbors heuristic - for varying sized city distributions in 2D.

For improving upon the path found by the initial heuristic, we implemented a deterministic 2-OPT local search algorithm, which in $O(n^2)$ explored all possible edge swaps, effectively arriving at a local minimum. Later, we improved the complexity to $O(nm)$ via neighbor lists (kNN). The results for its approximation can be seen from Table 2.

Input size	Optimum	2-OPT	Approx. factor	2-OPT-kNN	Approx. factor
1083	3558	3929	1.104	3853	1.083
662	2513	2703	1.076	2684	1.068
436	1443	1540	1.067	1504	1.042
237	1019	1106	1.085	1082	1.062
131	564	596	1.057	597	1.059
10	276	276	1.000	276	1.000

Table 2: Results of the 2-OPT and 2-OPT-kNN implementations applied to a greedy tour, for varying sized city distributions in 2D and $k = 100$.

As a final step, we further improved the 2-OPT-kNN implementation by adding the possibility to escape local minimum via simulated annealing and random swaps. As this entails randomness, for the results to have statistical significance we took the average over 100 runs of the algorithm for each test data set. This can be seen in Table 3.

Input size	Optimum	2-OPT + Annealing	Approx. factor
1083	3558	3807.82	1.070
662	2513	2671.83	1.063
436	1443	1487.86	1.031
237	1019	1055.71	1.036
131	564	584.35	1.035
10	276	276	1.000

Table 3: Average results over 100 runs of the 2-OPT + annealing implementation applied to a greedy tour, for varying sized city distributions in 2D.

4 Future work

Our original plan was to do further code optimization when we had a working proof of concept. However, as implementing more advanced methods yielded better results, we never really got around to considering purely code related optimization. Ideas such as inlining, faster functions for random or math operations could potentially decrease the number of computations, allowing for more cases to be considered during the optimization process.

A data-structure for faster handling of tours could be implemented. At every iteration of 2-OPT a switch has to be done when modifying the path. As we discussed earlier, every switch takes O(n) time. This is because a plain array is used to store the tour.

With a two-level tree, it is possible to make a switch in $O(\sqrt{n})$ time [2]. Maybe by dividing the cities into an array of $\sqrt{n}$ nodes, each marking a range for the array. Then a Boolean value can be used for every node to determined if a range of cities in the array has been reversed or not. Instead of reversing all of the cities, the nodes representing a set of cities can be reversed instead. The first and last nodes are special cases.

References

[1] Peter Steiglitz Kenneth Weiner. "Some Improved Algorithms for Computer Solution of the Traveling Salesman Problem". In: 1968.

[2] David Johnson et al. "The Traveling Salesman Problem: A Case Study in Local Optimization". In: 1 (Jan. 1997).

[3] Olivier Martin et al. "Large-Step Markov Chains for the Traveling Salesman Problem". In: *Complex Systems* 5 (Aug. 1997).

[4] Christian Nilsson. "Heuristics for the Traveling Salesman Problem". In: 2003.

[5] Sonia Kefi et al. "Ant supervised by PSO and 2-opt algorithm, AS-PSO-2Opt, applied to traveling salesman problem". In: *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 004866–004871.

[6] Andre Rohe. *VLSI Data Sets*. `http://www.math.uwaterloo.ca/tsp/vlsi/index.html`. Accessed: 2018-11-05. Forschungsinstitut für Diskrete Mathematik.