Contents

Lin-Kernighan TSP

Background

The Travelling Salesman problem

The Travelling Salesman Problem (TSP) is, given a set of cities and the distances between them, the problem of finding a shortest path that starts and ends in the same place and visits each city exactly once. We can model an instance of the TSP as a fully-connected weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads. The weight of an edge represents the cost of crossing it (e.g. time taken in minutes), and the weight of a tour $T = (e_1, \dots, e_n)$ is the sum of the weights of the edges $e_1, \dots, e_n$. We write $cost(u, v)$ to mean the cost of crossing the edge between cities (u, v)

In this framing a valid solution T is a Hamiltonian cycle on G, and a good solution is one which minimises the cost of T.

In this post we only care about **symmetric TSP**, which are instances of the Travelling Salesman Problem where the $cost(u, v) = cost(v, u)$. In other words the distance between two cities is the same in both directions.

Another special case of the travelling salesman problem is the **metric TSP**, which are instances of the Travelling Salesman Problem where the Triangle Equality holds. To recap, the Triangle Inequality states that for points a, b and c:

$$d(a, b) + d(b, c) \geq d(a, c)$$

In other words it's always quicker travel to directly between two points rather than stopping by a third one along the way. This reflects real life, and represents the kind of graph you'd have when using the TSP to model a problem like finding the shortest path for a delivery van to take when delivering parcels.

Heuristic search algorithms

A first approach to finding a tour might be to generate a random sequence of cities.

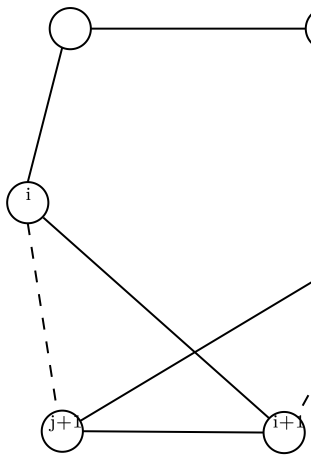
As python pseudocode where we represent cities as integers $0 \dots n-1$ and model the weighted graph as an $n \times n$ distance matrix, the random algorithm might look like:

```python
import random

def random_tour(dist_matrix: Matrix[int]) -> list[int]:
    n_cities = len(dist_matrix)
    tour = [i for i in range(n_cities)]
    random.shuffle(tour)

    return tour
```

As expected this doesn't work hugely well. One way of improving the result is to test if there are any edges in the tour we can swap to create a shorter path. Essentially, given two edges $(i, i+1), (j, j+1) \in T$ we want to test if replacing them with edges $(i, j+1), (j, i+1) \notin T$ improves the cost of T.



In the above example we can see that swapping $(i, i+1), (j, j+1)$ with $(i, j+1), (j, i+1)$ corresponds to reversing the subtour between i and j. After reversal the first node $i+1$ of the subtour becomes the last, and will be adjacent to j. Similarly, the last node $j+1$ in the subtour becomes the first, and will be adjacent to i.

If we do this for every pair of edges in T then we say the resulting tour is two-optimal, meaning that there are no pairs of two edges that we can swap to improve the length of the tour.

```python
def tour_cost(tour: list[int], dist_matrix: Matrix[int]) -> int:
    ...

def swap_subtour(tour: list[int], from_city: int, to_city: int) -> list[int]:
    ...

def two_opt(dist_matrix: Matrix[int]) -> tuple[list[int], int]:
    n_cities = len(dist_matrix)
    swapped = True

    # create an initial tour to optimise
    tour, current_cost = random_tour(dist_matrix)

    # improve `tour` by swapping edges until there are none
    # left to swap (it is two-optimal)
    while swapped:
        swapped = False

        for i in range(n_cities - 2):
            for j in range(i+1, n_cities - 1):
                # attempt to improve `tour` by swapping `(i, i+1), (j, j+1)` with
                # with `(i,j+1), (j,i+1)`. Reversing the subtour between `i` and `j`
                # does this implicitly
                new_tour = swap_subtour(tour, i, j)
                new_cost = tour_cost(new_tour, dist_matrix)

                if new_cost < current_cost:
                    tour = new_tour
                    current_cost = new_cost
                    swapped = True
                    break

    return tour, current_cost
```

A natural next step is to extend our two-edge swapping algorithm to three or more edges, and as the number of swaps we consider increases the quality of our tours improves.

Unfortunately the time complexity of our algorithm increases too. From the above code we can see that each iteration of two-opt takes **at least** $O(n^2)$, since we compare each edge in the tour with all other edges in the graph (of which there are $\frac{n(n-1)}{n} = O(n^2)$). Reversing the subtour will probably take $O(n)$ time too[1], and every edge we remove may introduce more potential edge swaps such that we need to loop over the tour many times until the solution converges to a local optimum.

At the very least, as we increase the number of edge swaps we consider in the core loop of our algorithm, $O(n^2)$ for 2Opt turns into $O(n^3)$ for 3Opt, $O(n^4)$ for 4Opt, and so on until we reach $O(n^n)$ (or $O(n!)$) for NOpt. Since the n-optimal algorithm gives us a tour where we can't rearrange any cities to improve its cost it should return the optimal solution to a TSP, but this comes at the cost of brute-forcing all possible permutations of cities, which has time complexity $O(n!)$.

While there exist exact algorithms to find an optimal tour (Dantzig-Fulkerson-Johnson & cutting planes) these algorithms are still relatively slow *read: not polynomial.*

Instead we attempt to find a solution by using **heuristic** algorithms. A heuristic algorithm finds an approximate solution by trading optimality (does it find the best solution?) and completeness (can the algorithm generate all solutions?) for speed. A heuristic algorithm uses a heuristic function f to evalutate possible solutions T. The algorithm repeatedly applies some transformation to T to generate a new solution T' such that $f(T') < f(T)$. This continues until no improving solution T' can be found. At this point we can either start again from a new candiate solution, or accept T as good enough and halt.

It is likely that f has many minima, and since they don't explore the entire solution space many heuristic algorithms use randomness to branch away from a current solution to find new, deeper local minima. Such branches are an imporant part of many popular heuristic algorithms like Ant Colony Optimisation, Simulated Annealing, and the Genetic Algorithm. One of the reasons the Lin-Kernighan algorithm is interesting to me is that it doesn't rely on randomness to escape a local minimum. Instead the heuristic function f allows a finite amount of "bad" moves that worsen $f(T)$ in the hope that the resulting moves lead to new local minima.

So, a heuristic algoritm proceeds by:

1. Starting with an arbitrary (probably random) feasible solution T
2. Attempt to find an improved solution T' by transforming T
3. If an improved solution is found (i.e. $f(T') < f(T)$), let $T = T'$ and repeat 2.
4. If no improved solution can be found then T is a locally optimal solution.

Repeat the above from step 1 until you run out of time or find a statisfactory solution. In the case of Lin-Kernighan the transformation $T \mapsto T'$ is a single k-opt move and the objective function f is the cost of the tour. Lin-Kernighan works by repeatedly applying $k \in 1, 2, ..., d$-opt moves to a candidate tour until no swap can be found that increases the cost of the tour. We start with a 2-opt move an extend it with extra edges, and evenutally apply the k-opt move that resulted in the best tour for $k \in 1..d$.

Overview

Consider a pair tours T, T' with costs $f(T), f(T')$ such that $f(T') < f(T)$. The Lin-Kernighan algorithm aims to transform T into T' by repeatedly replacing edges $X = \{x_1, x_2, \ldots, x_k\}$ in T with edges $Y = \{y_1, y_2, \ldots, y_k\}$[2]not in T.

In order to decide if a swap is good we need some measure of improvement. Let the lengths of x_i, y_i be $|x_i|, |y_i|$ respectively, and define $g_i = |x_i| - |y_i|$. The value g_i represents the gain made by swap i, and we define the improvement of one tour over the other as $G_i = \sum_i^k g_i = f(T) - f(T')$. A key part of the Lin-Kernighan algorithm is that g_i can be negative as long as the overall gain G_i is greater than 0. This allows Lin-Kernighan to avoid getting stuck in local minima by moving "uphill": we permit a bad move that might allow us to find new minima as long as the bad move doesn't ruin our tour.

The sequence of swaps $X = \{x_1, x_2, \ldots x_d\}$ with $Y = \{y_1, y_2, \ldots, y_d\}$ is sometimes called an ejection chain.

[1]good choice of tour data stucture will improve this - e.g. using a two-level tree

[2]Notice that $k \leq n/2$, where n is the number of cities

Basic algorithm

1. Generate a starting tour T.
2. Begin to search for improving tours: set $G^* = 0$, where G^* stores the best improvement made to T so far. Select any node $t_1 \in T$, and choose some adjacent node t_2 to construct the first candidate deletion edge $x_1 = (t_1, t_2)$. Let $i = 0$ be the current ejection chain depth and $d = 0$ be the most gainful chain depth found so far.
3. Choose some other city $t_3 \notin T$ from the other edge of t_2 to form an edge $y_1 = (t_2, t_3) \notin T$ such that $g_1 > 0$. If no such y_1 exists go back to step 2 and choose a different starting city t_1.
4. Let $i = i + 1$. The city t_{2i-1} is the end of the last added edge y_i. Choose a city t_{2i} to create an edge $x_i = (t_{2i-1}, t_{2i}) \in T$ and corresponding y_i[3]such that
    a) we can make a valid tour by adding the edge $y_i^* = (t_{2i}, t_1)$. Apparently x_i is uniquely determined for each choice of y_{i-1}.
    b) $x_i, y_i \notin X, Y$ (i.e. x_i and y_i haven't already been used)
    c) $G_i = \sum_1^i g_i > 0$
    d) y_i has a corresponding x_{i+1}

Before choosing y_i we first check if joining t_{2i} to t_1 results in a better tour than seen previously. As before let $y_i^* = (t_{2i}, t_1)$ be the edge completing T', and let $g_i^* = |y_i^*| - |x_i|$. If $G_{i-1}^* + g_i^* > G^*$, set $G^* = G_{i-1} + g_i^*$ and let $d = i$.

5. Stop finding new x_i, y_i when we run out of edges that satisfy the above conditions, or $G_i \leq G^*$. If $G^* > 0$, take T' to be the tour produced by the ejection chain of depth d, set $T = T'$ and $f(T') = f(T) - G^*$, and repeat the process from step 2 using T' as the initial tour.
6. If $G^* = 0$ we backtrack to progressively farther back points in the algorithm to see if making different choices of edge/city leads to better results.
    a) Repeat step 4, try different choices of y_2 in increasing length (or whatever metric you're using to select candidate edges)
    b) If 6a doesn't work, try different choices of y_1
    c) Else try different choices of x_1
    d) Else try different choices of t_1

We backtrack until we either see improvement or run out of new edges to try, though for the sake of reducing complexity only backtrack on levels 1 and 2.

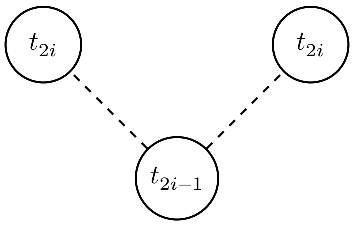
Comments

Written like this backtracking looks like a pain, but in practice the `goto`s are equivalent to a `for` loop.

Step 4 At step 4a there are two choices of x_i. Since t_{2i-1} is a city in the tour T, t_{2i} could either be the city to the "left" or "right" of t_{2i-1}:

[3]To clarify,

- x_i is adjacent to y_{i-1}
- y_i is adjacent to x_i
- x_i is an edge currently in T
- y_i is a new edge not in T



Only one of these is a valid, however. Identifying which city t_{2i} is the correct choice is an important part of the algorithm. If you choose t_{2i} on the subtour between t_1, t_{2i-1} then y^* implicitly creates a disonneced cycle $t_1, t_{2i}, \dots, t_n$.

TODO: diagram

In practice I did it by attempting to construct a tour from both choices of $x_i = (t_{2i-1}, t_{2i}), y_i^* = (t_{2i}, t_1)$ and picking the one that succeeded.

TODO: code

Paul Rubin suggested a nice algorithm for identifying the correct t_{2i} in a comment on this stackexchange post. Keld Helsgaun also chimed in to point to a section in one of his papers that describes how to test if a k-opt move is valid in $O(k \log k)$ time[4].

Psuedocode

```
# type tour = list int

function Lin-Kernighan(initial_tour: tour, n_cities: int) -> tour
    """Performs a single iteration of the lin-kernighan algorithm
    on the initial tour `initial_tour`.

    To follow the full algorithm wrap the whole thing in a while loop
    and repeatedly apply `Lin-Kernighan` to the improved tour `T`.
    """

    # Step (1)
    tour := initial_tour            # current working tour

    # Step (2)
    G_best := 0                     # best gain on `tour` seen so far
    i := 0                          # current ejection chain depth

    for city_1 in tour:
        for city_2 adjacent to city_1:
            x_1 := (city_1, city_2)         # select `x_1`

            X := {x_1}                      # initialise deleted edge tabu list
            Y := {y_1}

            k := 0                          # best seen ejection chain depth
            i := i + 1

            # Step (3)
            for city_3 adjacent to city_2:      # choose `city_3` not in tour
                if city_3 in tour or city_3 == city_1:
                    continue

                y_1 := (city_2, city_3)
                g_1 := cost(x_1) - cost(y_1)
```

[4]section 5.3 of ”General k-opt submoves for the Lin–Kernighan TSP heuristic”

```

            G := g_1                        # store total tour gain

            if g_1 <= 0:                    # apply the gain criterion
                continue

            Y := union(Y, {y_1})                     # initialise added edge tabu list
            city_prev = city_3

            # Step (4)
            while size(X) + size(Y) < n_cities:
                i := i + 1

                # select `x_i`
                for city_2i adjacent to city_prev in tour:

                    # check `city_prev` is not `city_1`, otherwise we can't
                    # relink the tour
                    if city_2i = city_1:
                        continue

                    x_i := (city_prev, city_2i)

                    if x_i in X:
                        continue

                    # check if removing `x_i` and joining back to `city_1`
                    # creates a better tour
                    y_final := (city_2i, city_1)
                    g_final := cost(x_i) - cost(y_final)

                    if G + g_final > G_best:
                        G_best := G + g_final
                        k := i
                        # store new tour?

                    # check if we can join the tour up
                    # how tf you do this bit
                    # check if the tour is valid?

                    for city_next adjacent to city_2i:                  # select `y_i`
                        y_i := (city_2i, city_next)

                        if y_i in Y:                                    # check tabu list
                            continue

                        # check y_i has correpending x_{i+1}
                        # how? lol

                        g_i := cost(x_i) - cost(y_i)
                        if G + g_i <= 0:                                # check gain criterion
                            continue

                        G := G + g_i

        # Apply the ejection chain up to `k`
        if k > 0:
            tour = apply_swaps(tour, X, Y, k)
```

```

    return tour
```