Contents

Lin-Kernighan TSP **1**
Background 1
The Travelling Salesman problem 1
Heruistic search algorithms 1
Overview 2
Ejection chains 2
Basic algorithm 2
Comments 3
Psuedocode 3
Enhancements 5
Candidate list 5
"Don't look" bits 5
Memoisation 5
Alpha-nearness 5
Bit arrays 5

old **5**

Lin-Kernighan TSP

Background

The Travelling Salesman problem

The travelling salesman problem is: given a collection of n cities and this distances between them, find the shortest path (tour) that visits every city and ends at the starting point. Usually we model an instance of the problem as a fully-connect weighted graph $G = (V, E)$, where the vertices V represent cities, and the edges E represent roads. In this framing the solution is the lowest-cost Hamiltonian cycle on G.

Problem: given a collection of cities and the distances between them, find a minimum-length tour that visits each city exactly once. At a glance we can see that brute forcing all possible permutations of cities (tours) has time complexity `O(n!)`, which quickly becomes impractical as n grows. While there exist algorithms to find an optimal tour in under `O(n!)` time *todo cite example*, 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 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.

Heruistic search algorithms

The spirit of heuric-based algorithms for combinatorial optimisation problems is:

1. Start 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 k-opt move and the objective function f is the cost of the tour.

Lin-Kernighan works by repeatedly applying $k \in [1..d]$-opt moves to a candidate tour until no swap can be found that doesn't increase the cost of the tour. The tour T' that we produce is the tour obtained by applying the k-opt move for the best value of $k \in [1..d]$ found.

Overview

todo talk about valid tour at every step

Consider a pair tours T, T' with lengths $f(T), f(T')$ such that $f(T') < f(T)$. The Lin-Kernighan algorithm aims to transform T into T' by repeatedly replacing edges $X = \{x_1, x_2, \dots, x_k\}$ in T with edges $Y = \{y_1, y_2, \dots, y_k\}$[1]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; 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. Being able to escape local minima is an important part of most TSP algorithms; other algorithms like Ant-Colony or Simulated-Annealing use randomness to do so.

todo compare X, Y to tabu lists (they do the same but more)

Ejection chains

todo

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[2]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 $G_i \le G^*$. 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.

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

[2]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

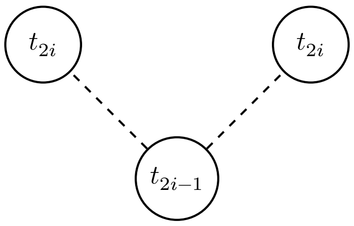
```
a) Repeat step 4, try different choices of $y_2$
in increasing length (or whatever metric you're using to select candidate edges)

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.

Written like this backtracking looks like a hassle, but in practice using a `for` loop lets you try multiple choices of nodes whith ease, while also being the most natural way to write parts of the algorithm.

Comments

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}:



Only one of these is a valid choice, however.

Psuedocode

The original paper describes the algorithm in terms of `while` loops and `goto` statements, which isn't the easiest to understand.

```
# type tour = list int

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

    To follow the full algorithm (i.e. repeatedly iterate on improved
    tours `T'`, wrap the whole thing in a while loop or call this function
    again with it's output from last time.
    """

    # 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)

                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 correponding 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
```

Enhancements

Candidate list

todo compute candidate lists beforehand and loop on/choose from them

"Don't look" bits

Memoisation

Alpha-nearness

Bit arrays

This isn't a TSP/LK-specific optimisation. How to best implement it in python:

- an array of bools?
- an integer What about an array of 8-bit integers? TODO read the code for BitVector

old

Problem: given a collection of cities and the distances between them, find a minimum-length tour that visits each city exactly once. At a glance we can see that brute forcing all possible permutations of cities (tours) has time complexity `O(n!)`, which quickly becomes impractical as n grows. While there exist algorithms to find an optimal tour in under `O(n!)` time *todo cite example*, 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 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.

In the case of the travelling salesman problem the heuristic function `f` that we are trying to minimise is the length of a tour. Our heuristic algorithm should proceed along the lines of:

1. Generate an initial tour `T`
2. Attempt to create an improved tour `T'` from `T`
3. If `T'` improves `T` (i.e `f(T') < (T)`), let `T = T'` and go to 2.
4. Otherwise we can't improve `T`. We can either go back to step 1 or declare `T` as good enough and halt.

At step 4 we have reached a local minimum. *todo visualise with a plane* To improve our solution we need to find a new tour to restart the algorithm from in the hopes that it leads to a better minimum. It follows that an important part of any heuristic algorithm is how it escapes local minima. Many algorithms do this by incorporating an element of randomness into the decision process *todo elaborate*. The Lin-Kernihan algorithm is interesting in that it doesn't do this, and instead allows some "bad" moves (with regards to `f`) in the hope that they lead us uphill into a state where we can find a new, deeper minimum.