Samo Novák
(s1865783)

IADS Coursework 3 March 2020

C Algorithm: Nearest Neighbour Insertion

We draw inspiration for this algorithm from the outline of Nearest Insertion (and other TSP heuristics) by Nilsson[1]. We chose this particular algorithm because of its elegance – it follows a very simple principle: start with some trivial circuit of minimum weight and then find the most suitable nodes to add to it, until all nodes are used. The pseudocode for this algorithm is presented below, as is some more discussion and visual explanation. In the program submitted, this is Graph.NearestExpand().

Algorithm 1: Nearest Insert

Input: an undirected complete graph $G = (V, E)$, where $|V| = n$
Input: a weight function $w : V \times V \to \mathbb{R}^+$, note: $\forall v \in V.\, w(v, v) = 0$
Result: a permutation π (represented as list) of all vertices representing a circuit (the path the travelling salesperson will take)

1 Initialize: $\pi \leftarrow [i, j]$, where $(i, j) = \operatorname{argmin}_{(i,j)\in E} w(i, j)$; /* $\Theta(n^2)$ */
2 Initialize: $U = \{x \in V | x \notin \pi\}$; /* $\Theta(n)$ */
3 **while** $U \neq \varnothing$; /* cycle will run $n - 2 = \Theta(n)$ times */
4 **do**
5 $(i, u) \leftarrow \operatorname{argmin}_{(i,v):\, i\in\{0,\ldots,|\pi|-1\},\, v\in U} \{w(\pi[i], v)\}$; /* $\Theta(n)$ */
6 $d_- \leftarrow w(u, \pi[(i - 1) \mod |\pi|])$; /* $\Theta(1)$ */
7 $d_+ \leftarrow w(u, \pi[(i + 1) \mod |\pi|])$; /* $\Theta(1)$ */
8 **if** $d_- < d_+$; /* $\Theta(1)$ */
9 **then**
10 Insert u into π at position i ; /* $\Theta(n)$ */
11 **else**
12 Insert u into π after position i, so at position $(i + 1) \mod |\pi|$; /* $\Theta(n)$ */
13 Remove u from U. ; /* $\Theta(|U|)$ */
14 **return** π;

First, we initialize the permutation π to a list of two nodes with minimum distance (line 1). We also initialize a set U of nodes *yet unused* in π (line 2).

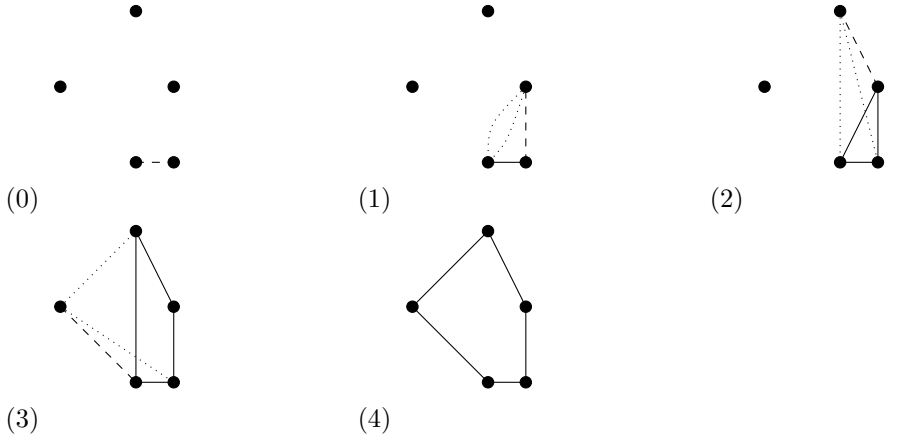
Then we will repeat the following **while** we still have unused nodes (U is nonempty). We first find an unused node (u) that is a nearest neighbour to one of the nodes in π (at index i) – so that $w(\pi[i], u)$ is minimal (line 5).

Next, we need to decide where to insert the new node u. We know that its distance to $\pi[i]$ is minimal, but we need to decide whether to insert it before $\pi[i]$ or after it. Because the permutation is a cycle, we can use $\pi[(i \pm 1) \mod |\pi|]$, and then we pick the one that has lower distance to u, because we are trying to build a cycle of minimum distance (lines 6, 7 and the **if** starting at line 8).

After inserting u at the appropriate position, we remove it from U – we wouldn't want to insert the same node multiple times.

The running time of this algorithm is $\Theta(n^2)$, where $n = |V|$. This is because the most significant contributions are done by the $\Theta(n^2)$ terms. Insertions take $\Theta(n)$ time, due to array representation of the list π. If we implement U also as a list (as is done in the program), removal from it will be $\Theta(|U|)$ (this will start off as $\Theta(n)$ and decrease). The running time of lines 1, 2, 5 will be elaborated in the Appendix i, in the interest of preserving clarity in this section.

Below is a simple visualisation of the described algorithm. We have used a 2-D Euclidean graph for the visualisation, because it will be easy to see distances intuitively.



In these diagrams, *full* lines represent the current permutation π, *dashed* lines represent the edge that will be definitely added into π next (line 1 in initialization, line 5 in the **while** loop), and *dotted* lines represent the other two nodes that are being condidered to close the cycle (lines 6, 7).

D Experiments

In this part, we created generators for different types and sizes of graphs, and then comparatively tested six heuristics: *Swap*, *2-Opt*, *Swap* followed by *2-Opt*, *Greedy* and *Nearest Insert*.

We have three cases of our problem: general TSP (no other constraints), metric TSP (w is a metric, so $\forall u, v, x \in V. w(u, v) \leq w(u, x) + w(x, v)$), and Euclidean TSP (the graph is embedded in a plane, this is also metric).

The test cases could be generated as completely random graphs. For small graphs (here $n \leq 7$), it is then feasible to compute the best path, so that we can compare it to the results from the tested heuristics. However, for larger graphs, it will be more useful to generate a graph with some pre-planned best path.

D.1 Small random graphs

Generating these will be simple: we will populate the matrix Graph.dists with random positive integers. Then we will brute-force the best TSP path.

In the Euclidean case, we will instead generate randomly positioned cities in a plane.

For a general metric case, we will generate the cities randomly embedded in a plane, same as in the Euclidean case. However, we will use the taxicab metric (also called the Manhattan metric), which measures the distance along the axes individually and sums them. In the case of discrete positions (which we will use):

$$w_M : \mathbb{N}^d \times \mathbb{N}^d \to \mathbb{N}$$
$$w_M\left(\langle u_1, ..., u_d\rangle, \langle v_1, ..., v_d\rangle\right) = \sum_{i=1}^{d} |u_i - v_i|$$

where d is the number of dimensions of the space (we may use arbitrary $d \in \mathbb{Z}^+$).

Since all these graphs will be small, we can get the optimal TSP solution by simply trying out all the permutations.

D.2 Larger graphs

A good way to intentionally generate an optimal path in a (general) graph is to create a cycle with small weights in it and then assign larger values to all the other edges. So:

$$\forall i \in \{0, ..., n-2\}.\, w(i, i+1) = \text{RANDINT}(1, 5)$$
$$\forall i, j \in 0, ..., n-1.\, i < j \wedge j \neq i+1.\, w(i, j) = \text{RANDINT}(6, 10)$$

The random number boundaries here are somewhat arbitrary, we just need to make sure that the cycle has smaller values. In the code, the threshold between these (here 5) is the argument low_cycle_threshold.

Once we have this matrix, we swap the vertices around, so that the initial permutation $[0, 1, ..., n-1]$ won't immediately be the solution. We do this by generating a graph that is isomorphic to the original one, i.e. $\exists$ *bijection* between the vertices of the old one and the new one that preserves the graph structure. We generate a permutation $p : V \to V$ to be this bijection and then create a corresponding new weight function $w^\star : V \times V \to \mathbb{R}^+$:

$$\forall i, j \in V.\, w^\star\left(p(i), p(j)\right) x = w(i, j)$$

For metric (and Euclidean) graphs, we generate a circle w.r.t. the particular metric we are using, and then reorder the nodes similarly.

D.3 Results

We have tested multiple graphs of different types and sizes. In the following table, W_0 is the optimal path weight (total distance), and W_i is the weight of the initial permutation $\forall i \in \{0, ..., n-1\}. \pi(i) = i$. Types of graphs are **G** for general, **M** for metric, and **E** for Euclidean. All measurements are averaged over 100 random graphs of the specified type and size.

Type	n	Swap vs. W_0 vs. W_i	2-Opt vs. W_0 vs. W_i	Swap, 2-Opt vs. W_0 vs. W_i	Greedy vs. W_0 vs. W_i	NearestInsert vs. W_0 vs. W_i
G	5	1.002 0.728	1.003 0.729	1.002 0.728	1.053 0.766	1.133 0.819
G	7	1.21 0.69	1.028 0.598	1.03 0.598	1.157 0.671	1.246 0.718
G	10	1.593 0.802	1.018 0.521	1.019 0.522	1.05 0.536	1.433 0.726
G	20	1.969 0.835	1.044 0.447	1.049 0.449	1.061 0.453	1.543 0.656
G	50	2.186 0.868	1.063 0.425	1.062 0.424	1.038 0.415	1.601 0.637
G	100	2.288 0.879	1.071 0.412	1.065 0.41	1.027 0.395	1.636 0.629
G	200	2.343 0.89	1.066 0.406	1.066 0.405	1.015 0.386	1.647 0.626
M	5	1.001 0.831	1.001 0.831	1.001 0.831	1.039 0.862	1.048 0.867
M	7	1.16 0.779	1.012 0.687	1.01 0.685	1.068 0.726	1.17 0.791
M	10	1.604 0.792	1.051 0.522	1.072 0.533	1.199 0.595	1.556 0.771
M	20	2.742 0.77	1.014 0.286	1.032 0.29	1.369 0.384	1.723 0.485
M	50	6.527 0.747	1.011 0.116	1.032 0.118	1.291 0.148	1.926 0.221
M	100	12.764 0.76	1.014 0.06	1.018 0.061	1.421 0.085	1.943 0.116
M	200	25.35 0.755	1.006 0.03	1.007 0.03	1.424 0.042	1.963 0.058
E	5	1.001 0.846	1.001 0.846	1.001 0.846	1.049 0.886	1.066 0.898
E	7	1.176 0.796	1.002 0.683	1.003 0.684	1.062 0.726	1.153 0.785
E	10	1.697 0.789	1.0 0.468	1.0 0.468	1.0 0.468	1.623 0.759
E	20	3.019 0.778	1.0 0.258	1.0 0.258	1.0 0.258	1.795 0.463
E	50	7.255 0.768	1.0 0.106	1.0 0.106	1.0 0.106	1.928 0.204
E	100	14.051 0.755	1.0 0.054	1.0 0.054	1.0 0.054	1.957 0.105
E	200	28.026 0.76	1.0 0.027	1.0 0.027	1.0 0.027	1.978 0.054

We see that overall, *Nearest Insert* performs rather poorly compared to the other ones. This is true for all kinds of graphs, as *N.I.* doesn't use any special properties of metric or Euclidean graphs. It still manages to decrease the total weight of the path (W) w.r.t. the original permutation.

The one that performs surprisingly good is the combination of *Swap* followed by *2-Opt*. In a lot of cases, it even finds the optimal solution. Individually though, they are not very good.

Right behind those is *Greedy*, which also delivers rather good results. In some cases, it can also get the optimal solution.

In conclusion, *Swap + 2-Opt* and *Greedy* deliver very good results usually, and most of the times better than *Nearest Insert*.

APPENDIX

i Running time of Nearest Insert

i.1 argmin on line 1

By the concrete properties of our weight functions $w : V \times V \to \mathbb{R}^+$ (symmetry and $\forall u \in V.\, w(u,u) = 0$):

$$\operatorname*{argmin}_{(i,j)\in E} w(i,j) = \operatorname*{argmin}_{\substack{(i,j)\in V\times V \\ j<i}} w(i,j)$$

So the running time, which depends on the number of lookups (where a lookup of $w(i,j)$ is $\Theta(1)$, because we represent w as a matrix):

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}\Theta(1) = \Theta(1)\sum_{i=0}^{n-1} i = \Theta(1)\cdot\frac{(n-1)(n-1+1)}{2} = \Theta(n^2)$$

$\bowtie$

i.2 Unused nodes set on line 2

Here we iterate over n nodes and check their membership in a list of 2. This means running time $\Theta(2n) = \Theta(n)$.

$\bowtie$

i.3 argmin on line 5

In each iteration of the **while** loop let k be the number of iteration, starting at $k = 0$ just after initialization. Let π_k and U_k be the permutation, and unused nodes set, respectively, after k^{th} iteration. Then $|\pi_k| = |\pi_0| + k = k + 2$, since at every iteration we add one node to π, and we start off with two nodes ($|\pi_0| = 2$). In all iterations, $\pi_k \cup U_k = V$ and $\pi_k \cap U_k = \varnothing$, so $|U_k| = n - |\pi_k|$. A lookup takes $\Theta(1)$ time. Then running time of $\operatorname{argmin}_{(i,v):\, i\in\{0,\ldots,|\pi|-1\},\, v\in U} \{w(\pi[i], v)\}$ is, in every iteration:

$$\begin{aligned}\Theta(|\pi_k||U_k|) &= \Theta\left[(k+2)(n-(k+2))\right] \\ &= \Theta\left[(k+2)(n-k-2)\right] \\ &= \Theta(kn-k^2-2k+2n-2k-4) \\ &= \Theta(2n+kn-k^2-4k-4) \\ &= \Theta(n+kn-k^2-k)\end{aligned}$$

Our k will range over $\{0,...,n-2\}$. If $k=0$ (**base case**):

$$\begin{aligned}\Theta(|\pi_0||U_0|) &= \Theta(n+kn-k^2-k) \\ &= \Theta(n+0-0-0)\end{aligned}$$

$$\therefore \Theta(|\pi_0||U_0|) = \Theta(n) \tag{1}$$

Now show that if for arbirary $k \in \{0,...,n-3\}$ it is true thaty $\Theta(|\pi_k||U_k|) = \Theta(n+kn-k^2-k) = \Theta(n)$ then for $k+1$ it holds too.

$$\begin{aligned}\Theta(|\pi_{k+1}||U_{k+1}|) &= \Theta(n+(k+1)n-(k+1)^2-(k+1)) \\ &= \Theta(n+kn+n-k^2-2k-1-k-1) \\ &= \Theta(2n+kn-k^2-3k-2) \\ &= \Theta(n+kn-k^2-k) \\ &= \Theta(|\pi_k||U_k|)\end{aligned}$$

And now by induction from (1), we get that:

$$\therefore \forall k \in \{0,...,n-2\}.\, \Theta(|\pi_k||U_k|) = \Theta(n)$$

$\bowtie$

References

[1] Christian Nilsson, *Heuristics for the Traveling Salesman Problem*, Linköping University, 2003. Accessible online [27-03-2020]: `https://pdfs.semanticscholar.org/7b80/bfc1c5dd4e10ec807c6f56d0f31f8bf86bc6.pdf`