IADS Coursework 3

Samo Novák (s1865783)

March 2020

C Algorithm: Nearest Neighbour Insertion

⋆ TODO: **Name? Call it expansion?**

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 $\pi$ (represented as list) of all vertices
        representing a circuit (the path the travelling salesperson will
        take)
Initialize: $\pi \leftarrow [i, j]$, where $(i, j) = \text{argmin}_{(i,j)\in E}\, w(i, j)$ ;   /* $\Theta(n^2)$ */
Initialize: $U = \{x \in V | x \notin \pi\}$ ;   /* $\Theta(n)$ */
while $U \neq \varnothing$ ;   /* cycle will run $n - 2 = \Theta(n)$ times */
 do
    $(i, u) \leftarrow \text{argmin}_{(i,v):\, i\in\{0,\dots,|\pi|-1\},\, v\in U} \{w(\pi[i], v)\}$ ;   /* $\Theta(n)$ */
    $d_- \leftarrow w(u, \pi[(i-1) \mod |\pi|])$ ;   /* $\Theta(1)$ */
    $d_+ \leftarrow w(u, \pi[(i+1) \mod |\pi|])$ ;   /* $\Theta(1)$ */
    if $d_- < d_+$ ;   /* $\Theta(1)$ */
     then
        Insert $u$ into $\pi$ at position $i$ ;   /* $\Theta(n)$ */
    else
        Insert $u$ into $\pi$ after position $i$, so at position $(i+1) \mod |\pi|$ ;
         /* $\Theta(n)$ */
    Remove $u$ from $U$. ;   /* $O(|U|)$ (impl. dep.) */
return $\pi$;
```

On line 1, we initialize the permutation π to a list of two nodes i, j, such that their distance $w(i,j)$ in the graph is minimal. We also intialize a set U of nodes *yet unused* in π (line 2).

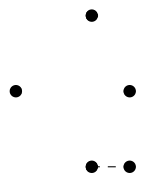
We shall repeat the following steps **while** U is not empty, meaning π does not contain all the nodes. First we need to find the nearest neighbour (an unused node) u to some node contained in π (at index i). We do this by looking for a pair where first element is the index in π and the second is an unused node, such that the nodes u and $\pi[i]$ have minimal distance (line 5).

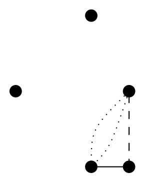
Next, we need to decide where to place the new node u. We know, that its distance to $\pi[i]$ is minimal, but there are two ways to insert it into π: before $\pi[i]$ or after it. Therefore, we need to compare the distances to the predecessor and successor of $\pi[i]$ in the permutation. We use $\pi[(i \pm 1) \mod |\pi|]$ (lines 6, 7), because the salesperson's walk is a cycle, so the first and last node in π are connected as well. We then choose the lower distance, because we want to minimize the weight of the permutation overall. (**if** statement starting at line 8)

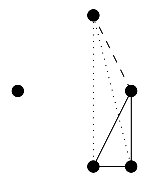
After inserting u at the appropriate position, we remove it from U – we wouldn't want to insert the same node multiple times. We also like our algorithms to terminate.

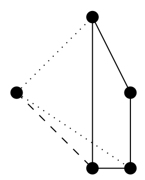
The running time of this algorithm is $\Theta(n^2)$, where $n = |V|$. This comes from the fact that the most significant contributions are done by the $\Theta(n^2)$ terms. Insertions take $\Theta(n)$ time, due to array representation of the list π. If we implement U also as a list (as is done in the program), removal from it will be $\Theta(|U|)$ (this will start of 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.

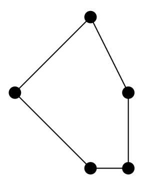
Below is a simple visualisation of the described algorithm. We have used a 2-D Euclidean graph for the visualisation, because it will be easy to see distances intuitively.


(a)


(b)


(c)


(d)


(e)

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 loop), and *dotted* lines represent the other two nodes that are being condidered (lines 6, 7).

⋆ TODO: **Elaborate? Treba vobec?**

D Experiments

In this part, we were to generate (or write a generator for) many different test cases and quantitatively measure how each heuristic (Swap, 2-Opt, Greedy, ⋆ TODO: **own**) performs.

D.1 Test generation

Firstly, let us summarize the assumptions and setting of the problem:

- We have an undirected, complete graph $G = (V, E)$ where $|V| = n$.
- We have a weight function $w : V \times V \to \mathbb{R}^+$, such that $\forall i, j \in V.\, w(i, j) = w(j, i)$, which will be represented as a symmetric matrix Graph.dists[i][j] $= w(i, j)$). We assume $\forall i \in V.\, w(i, i) = 0$.
- 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 (⋆ TODO: **how big?**), 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.1.1 Small random graphs

Generating these will be simple: we shall populate the matrix Graph.dists with random positive integers. Then we will compute 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 (in our case $d = 2$ ⋆ TODO: **more dimensions??**).

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

D.1.2 Larger graphs

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

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

The random number boundaries here are somewhat arbitrary, we just need to make sure that the cycle has smaller values.

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.

APPENDIX

i Running time of Nearest Insert

This section concerns the running time of algorithm ⋆ TODO: **name**.

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):

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

⋈

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 $\Theta(2n) = \Theta(n)$.

⋈

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 $\operatorname{argmin}_{(i,v):\ i\in\{0,\dots,|\pi|-1\},\ v\in U} \{w\,(\pi[i], v)\}$ 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**):

$$\begin{aligned}
\Theta(|\pi_0||U_0|) &= \Theta(n + kn - k^2 - k) \\
&= \Theta(n + 0 - 0 - 0) \\
\therefore \Theta(|\pi_0||U_0|) &= \Theta(n) \qquad (1)
\end{aligned}$$

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`