Techniques Algorithmiques et Programmation

Traveling Salesperson Problem

Recall the problem definition :

Traveling Salesperson Problem

Instance: A set $V = \{v_0, \ldots, v_{n-1}\}$ of points and a distance d on V.
Question: Find a tour of minimum length passing through all points of V, i.e., a permutation σ of the indices of the elements of V such that $\sum_{i=0}^{n-1} d(v_{\sigma(i)}, v_{\sigma(i+1 \bmod n)})$ is minimum.

In the following, we assume that $V \subset \mathbb{R}^2$ is a set of n points in the plane and that d is the Euclidean distance between two points. The objective of the lab sessions will be to program and test the performance of several algorithms on specific instances of the problem.

We will use the data structure `point` defined by `struct{ double x,y; }`.

Question 1. *Provide the* `C` *code for the function* `dist(A,B)` *returning the distance between points* `A` *and* `B`.

We will represent a permutation σ of $\{0, \ldots, n-1\}$ by an array `P` of size n such that `P[i]` $= \sigma(i)$. For example, `int P[]={3,1,2,0};` will represent a permutation for $n = 4$ that swaps the first and last elements, defining the tour v_3, v_1, v_2, v_0 with a return to v_3. In other words, `P` represents the visit order of the points of V in the tour.

Question 2. *Write the function* `value(point *V, int n, int *P)` *which returns the length of the tour of the* `n` *points of* `V` *according to the permutation* `P`.

"Brute-Force" Approach

Permutations can be ordered lexicographically from the smallest (in our example `P={0,1,2,3}`) to the largest (`P={3,2,1,0}`). We assume the function `bool NextPermutation(int *P,int n)` is given; it calculates, by updating `P`, the permutation immediately following `P` in lexicographical order. Furthermore, the function returns `false` if and only if, when the function is called, the permutation `P` corresponds to the largest permutation. You may use, though it is not necessary, the constant `DBL_MAX` (defined in `float.h`) which defines the largest value representable by a `double` type variable.

Question 3. *Write the function* `double tsp_brute_force(point *V,int n,int *Q)` *which, starting from a set* `V` *of* `n` *points, returns the permutation* `Q` *minimizing the length of the tour according to the exhaustive approach (or "Brute-Force") as well as the length of this tour.*

We will now optimize the previous procedure into `tsp_brute_force_opt()`. The first optimization consists of fixing one of the points of the permutation, either the first or the last (whichever is more convenient). This allows saving a factor of n on the number of permutations to test.

The second consists of stopping the evaluation during the calculation of `value()` as soon as the current length reaches or exceeds that of the best tour already obtained, say w. In this optimization,

do not forget the return to the starting point ; that is, a partial tour using the first $i+1$ points $v_{\sigma(0)}, \ldots, v_{\sigma(i)}$ will have $i+1$ edges and a length of $w_i = \left(\sum_{j=0}^{i-1} d(v_{\sigma(j)}, v_{\sigma(j+1)})\right) + d(v_{\sigma(i)}, v_{\sigma(0)})$. The observation is that if the length of the partial tour $w_i \geqslant w$, then one can directly skip to the largest permutation with prefix $\sigma(0), \ldots, \sigma(i)$.

Question 4. *Does the observation assume the triangle inequality ?*

Question 5. *Write a function* `double value_opt(V,n,P,w)` *which returns :*

- *the length of the tour if it is smaller than* `w` *; or*
- `-k` *if* `k` *is the number of edges of the first partial tour that exceeds* `w`*.*

(NB : Pay attention to the returned index `k` *! The function can be further improved by considering the return distance* $d(v_{\sigma(i)}, v_{\sigma(n-1)}) + d(v_{\sigma(n-1)}, v_{\sigma(0)})$ *instead of* $d(v_{\sigma(i)}, v_{\sigma(0)})$*. Why ?)*

Question 6. *Write a function* `MaxPermutation(P,n,k)` *which returns in* `P` *the largest permutation, in lexicographical order, whose prefix of size* `k` *is that of* `P`*. (You may assume that* `P` *was the smallest permutation possessing this prefix.) What is the complexity of your function ? Deduce the code for* `double tsp_brute_force_opt(point *V,int n,int *Q)`.

Lab Session

Download (*Save link as...*) the files corresponding to the lab session from the course page available below, and place everything in the same directory :

http://dept-info.labri.fr/~gavoille/UE-TAP/

You will need to edit `tsp_brute_force.c`, compile with `make tsp_main` or `make -B tsp_main`, and run the execution with `./tsp_main`. Implement and test `tsp_brute_force()` and then the optimized version `tsp_brute_force_opt()`.

Start with a small number of points (default $n = 10$) then test the various optimizations by increasing n, which you can pass in the command line to `./tsp_main`. One way to test your results is to use the point generation method `generateGrid()`. With points on regular grids such as 2×2 or 1×6, you should be able to easily confirm/compare your results.

If necessary, use the same seed in the random generator, with `srandom()` (via the variable `seed`), so that you can compare your experiments (comment/uncomment parts of `main()`). You should observe a gain of a factor of 20 to 100 for the optimized versions.

To be able to exit the window during execution, press `'q'` but also change the `while` of your `tsp_brute_force` functions to `while(NextPermutation(...) && running)`. For an animation, add the instructions `drawTour(V,n,P); SDL_Delay(500);` in the heart of your `tsp_brute_force()` functions to visualize the current tour given by `P` with a delay in milliseconds (cf. `man SDL_Delay`). But beware ! $n! \times 0.5$s can be very long ! Only use this in combination with `(... && running)` to be able to exit prematurely. Press `'h'` to discover the graphical interface commands.

Other optimizations. You can use the `hypot()` function from `math.h` to simplify the distance calculation (see online help `man hypot`), but in terms of computation time the gain is not obvious. Test it. You can also avoid the `%n` in `value()`, which saves about 15%. Next, you can rewrite `value()` (or `value_opt()`) to pre-calculate all distances in a table `D[][]` rather than calling `dist()`. This should only be done once, of course. Your new function should then look like :

```
double value(point *V, int n, int *P){
          static double **D=NULL; // initialized at compilation
          if(D==NULL){            // will be executed only once
            D=malloc(n*sizeof(double*));
            for(int i=0;i<n;i++){
              D[i]=malloc(n*sizeof(double));
              for(...) ...;
```

You should gain a factor of two. How can you free the table `D` (or what conventions should be adopted to do so) ?

You could also gain an additional factor of two by noticing that the `tsp_brute_force()` function (as well as the `tsp_brute_force_opt` variant) visits twice as many tours as necessary. Indeed, traveling in one direction or the other results in a tour of the same length, even though the two tours will be considered different by `tsp_brute_force()`. How can you change the main loop to visit at most $(n-1)!/2$ tours ?