Technische Universiteit
Eindhoven
University of Technology

Problem Description – Fall 2013

DBL Algorithms (2IO90)

Mark de Berg

Contents

1 Description of the problem

In the process of creating digital models of real-world objects, an important task is to reconstruct shapes from point samples. These point samples can be obtained in a variety of ways and the reconstruction can serve different purposes. For example, suppose we want to copy a 3D object. Then we could proceed as follows. First, we scan the object with a 3D scanner. This generates a large collection of points lying on the surface of the object. From this point cloud, we then have to generate a 3D model of the object. Finally, the model is sent to a 3D printer. The most difficult algorithmic problem in this process is the second step, where we have to reconstruct (the surface of) a 3D object from a collection of sample points. In this DBL project we consider several reconstruction problems. Contrary to the problem just sketched, however, we will consider reconstruction problems on 2D point sets. Moreover, we will ignore the fact that in practice there are measurement errors in the sample points, something which significantly complicates the problem. Next we discuss the specific variants of the reconstruction problem that we will consider in this DBL problem.

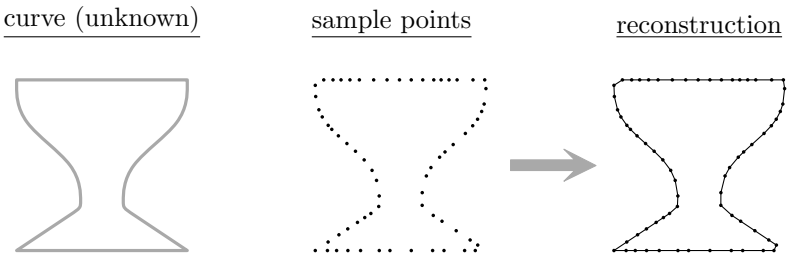


Figure 1: The curve-reconstruction problem. In this example the curve is closed.

The first problem we consider is *curve reconstruction*. The task is illustrated in Fig. 1. The input is a set of points in the plane that have been sampled from an unknown curve, and the goal is to reconstruct the curve from which the points have been sampled. Note that the points are not given in order along the curve—determining the order of the points along the curve is actually the heart of the problem. Once we have determined the order of the points, we can connect them by straight line segments to obtain a polygonal reconstruction of the underlying curve.

We will consider two variants of the curve-reconstruction problem. In the first variant we want to reconstruct a single curve. The curve can either be open (meaning that it has two endpoints) or closed (meaning that it has no endpoints, but is "circular"). In the former case, if $p_1, p_2, \ldots, p_n$ denotes the sequence of sample points ordered along the curve, then the polygonal reconstruction consists of the segments $p_i p_{i+1}$ for $1 \leqslant i < n$; in the latter case the reconstruction also contains the segment $p_n p_1$. In the second variant the sample points come from multiple curves. We then face the additional problem of identifying which points come from which curves. For both problems we have the following requirements:

- the curve segments (that is, the segment used in the reconstruction) should not intersect each other, except possibly at their endpoints;
- all input points should be used, that is, for each input point p there should be at least one curve segment having p as one of its endpoints.

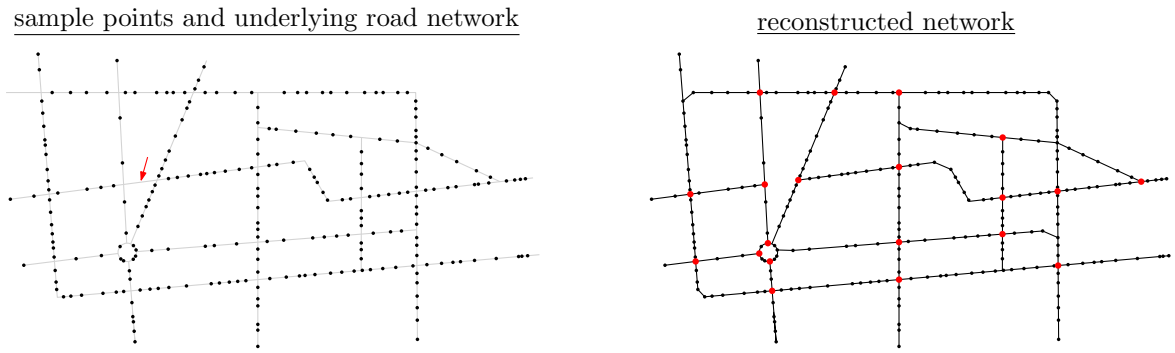


Figure 2: A road network (in light grey) with sample points, and a possible reconstruction. The sample points are indicated by small black disks, and the extra points added by the reconstruction algorithm are indicated by red, somewhat larger disks.

The second problem we consider is *network reconstruction*. A possible application is the automatic construction of road maps from a collection of car locations (GPS locations, say, obtained from smart-phone data or navigation devices). The road-network reconstruction problem is illustrated in Fig. 2. Note that a part of one of the roads in the example, indicated by an arrow has not been reconstructed. This part does not contain a sample point, so without any extra information there is probably no way of telling that this part should be present in the network. (Hence, the reconstruction could be considered better without this part.) Indeed, in reconstruction problems there is *a priori* no well-defined, unique solution. Thus the goal is to compute solutions that are reasonable (in the sense that a human user would say that the underlying curve or network might actually correspond to the true underlying curve or network) and that satisfy the requirements (such as the output segments being non-intersecting). For the network reconstruction problem we have the following requirements:

- the network must be connected;
- the road segments (that is, the segments used in the reconstruction) should not intersect each other, except possibly at their endpoints;
- all input points should be used, that is, for each input point p there should be at least one road segment having p as one of its endpoints.

Contrary to the curve reconstruction problem we allow additional points to be added to the network, for example at places where roads cross—see also Fig. 2.

2 The project

In this project you have to develop software for the variants of the curve- and network-reconstruction problem described above. The program to solve this task should be implemented in Java. It should read an input file that describes a problem instance from standard input (System.in) and write the output to standard output (System.out). You should submit your program to Peach.

The program will be evaluated by running it on a set of test instances. The goal is that your program computes a (collection of) curve(s) or a network that, to a human looking at the sample points, seems a possible solution for the given instance. In most test instances

there will only be one reasonable solution, but there can also be some instances where this is not entirely clear (and, hence, there are multiple "correct" solutions.) For both curve-reconstruction variants there will be six test instances, and for the network reconstruction problem there will be six test instances. The most important criterion for grading the software will be the quality of the output, for which the number of correctly (or: almost correctly) reconstructed inputs is the main criterion. The running time is another, less important criterion. The algorithm should be reasonably efficient: on a set of at most 10,000 sample points it should terminate within 5 minutes. (Most test cases will have far fewer sample points, by the way.)

2.1 Input format

The input file describing a problem instance has the following structure. The first line determines the variant that needs to be solved. The format of this line is:

`reconstruct` *variant*

where *variant* ∈ { `single`, `multiple`, `network` }.

The second line of the input file has the format:

n `number of sample points`

Here n is a positive integer that specifies how many sample points are provided. The rest of the file consists of n lines, each specifying one of the sample points. More precisely, the i-th line specifying a sample point has the following form:

$$id\ x\ y$$

Here id is the identifier of the sample point which for the i-th point is simply the integer i, and x and y are the x- and y-coordinate of the i-th sample point, respectively. The coordinates are floating-point numbers in the range $[0, 1]$. An example input file is given in Fig. 3. For the road-network reconstruction problem, the unit is 1 km, which means that the input lies in an area of 1 km^2.

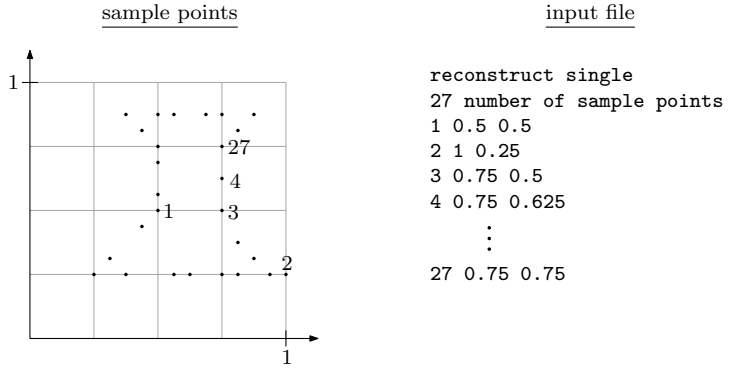


Figure 3: Example of a set of sample points and the corresponding input file for the reconstruction of a single curve.

2.2 Output format

The output file for a certain problem instance has the following structure. The first part is simply a copy of the input file. After that the reconstruction of the curve(s) or network is specified, as follows.

When the task is curve reconstruction—when *variant* ∈ { `single`, `multiple` }—then the line following the copy of the input specification has the form

$$s \texttt{ number of segments}$$

This line specifies the number of segments used in the reconstruction, which depends on the number of open curves in the output. The rest of the file consists of s lines, each specifying an output segment as follows:

$$id1 \;\; id2$$

Here *id1* and *id2* are the identifiers of the sample points that are the endpoints of the segment. Thus *id1* and *id2* are integers in the range $1, \ldots, n$. Note that the collection of reported segments uniquely determines the number of curves in the reconstruction, which points belong to which curve, and for each curve whether it is open or closed. In a correct reconstruction each identifier occurs once or twice as a segment endpoint: once when it is the start or end of an open curve, and twice otherwise.

When *variant* = `network`, then the copy of the input is followed by a specification of the extra points that are used in the reconstruction, before the segments of the reconstruction are listed. More precisely, the first line following the copy of the input has the form

$$m \texttt{ number of extra points}$$

where m is a non-negative integer. This is followed by m lines of the form

$$id \;\; x \;\; y$$

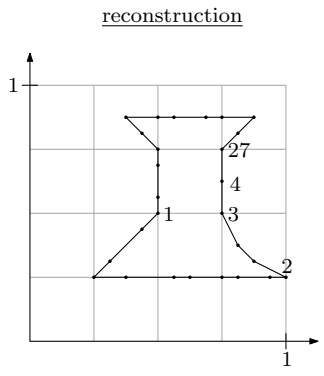
Here *id* is the identifier of the extra point which, for the i-th extra point is simply the integer $n+i$—the identifiers $1, \ldots, n$ are already taken by the sample points—and x y are the x- and y-coordinate of the i-th extra point. The coordinates are floating-point numbers in the range $[0, 1]$. The rest of the file specifies the segments of the reconstruction, in the same format as for the curve-reconstruction variants: there is a first a line of the form

$$s \texttt{ number of segments}$$

after which there all s segments are listed in the form

$$id1 \;\; id2$$

Note that now the identifiers are integers in the range $1, \ldots, n+m$. An example output file is given in Fig. 4.



output file

```
reconstruct single
27 number of sample points
1 0.5 0.5
2 1 0.25
3 0.75 0.5
4 0.75 0.625
  ⋮
27 0.75 0.75
27 number of segments
3 4
4 27
  ⋮
```

Figure 4: Example of a set of sample points and the corresponding output file for the reconstruction of a single curve.

3 Solving algorithmic problems

Modeling the problem. In the reconstruction problems discussed above, there is no well-defined ground truth. For any set of sample points there are many curves passing through the sample points, and without additional information it is impossible to say for sure whether a certain reconstruction is correct or not. This does not mean, however, that no precise statements about the problem or the algorithm can be made. In the curve-reconstruction problem, for instance, the underlying curves might be known to be non-intersecting. Then we would like to prove that the output of our algorithm is non-intersecting. Sometimes it is also possible to prove statements like the following (for the problem of reconstructing a single open curve): if the sampling density is high enough, then the algorithm is guaranteed to reconstruct the correct curve, that is, it correctly orders the sample points. To prove such a statement, we would need a precise mathematical definition of what it means that "the sampling density is high enough". (In the context of the DBL project, formulating and proving such a statement yourself is too much to ask, but you may encounter this type of guarantees in the literature.)

In general, when faced with an algorithmic problem you should always try to make the problem statement as precise as possible. You should prove that your algorithm computes a solution that adheres to the requirements (such as being non-intersecting), and you should try to formulate and prove any other interesting properties of the algorithm. It is also useful to analyze cases where the algorithm does not generate the desired output; this may help you to improve your algorithm.

Designing algorithms. Once the problem statement is clear (or: as clear as possible) you can start thinking about algorithms to solve the problem. It is allowed to use algorithms from the literature, provided you properly cite and discuss the relevant literature. However, algorithms from the literature may not solve the exact problem you want to solve (in this case you can perhaps adapt the algorithm), or they are possibly to complicated to implement. In this case you have to design your own algorithm.

Implementing algorithms. Before you start implementing your algorithms, it is good to make a design in which you establish which classes and methods need to be implemented, and what functionality they should offer. This makes it easier to divide implementation tasks among the team members. It is wise to first implement a simple, possibly less efficient, version of your algorithm. Once this version works properly, you can start replacing certain modules by ones that are more efficient or compute better solutions. It is not allowed to use other people's code (not from within the course and not from outside the course) without prior permission. This also applies to libraries that are not included in Java.

Evaluating the algorithms. Evaluating your algorithms should be done theoretically and experimentally. A theoretical analysis involves analyzing the running time, proving that the algorithm correctly solves problem and/or proving certain properties of the solutions is computes. An experimental evaluation involves investigating the quality of the computed solutions, the actual running time in practice, etcetera. For the experimental evaluation, which is one of the most important aspects of this DBL project, you should carefully generate different types of test data sets. Often the evaluation leads to ideas to improve the algorithms. For this a GUI that allows you to easily run the algorithm on different test sets and see the resulting output is quite useful. (A tool that generates a picture of the output is also useful for generating figures to be used in the final report.)