Point clouds: subsampling and neighborhood

Marc Glisse*

Feel free to use any reasonably common programming language, but whatever the language, make an effort so the code is readable even by someone who does not know this particular language. Unless otherwise specified, you are welcome to use the basic datastructures and algorithms provided by your language (Java ArrayList, Python dictionaries, sorting, etc), including some external libraries (like NumPy, or some library providing a hash map if your language does not provide one by default). However, you should obviously avoid using a library that solves the whole problem or even a large part of it, contact me if you have a doubt. Since the whole point of this project is for you to practice, you should also avoid looking at existing solutions.

The exact interface of functions and classes is up to you. Coming up with interfaces that are convenient to use is an important part of programming. In particular, sharing some code between different tasks in helper functions would probably be better than copy-pasting large blocks of code. The exact datastructures are also up to you, if you are asked to store a list of points, you are free to use a sorted vector, a binary tree or a hash set instead of the informal "list", or to replace points with their index or their distance to the origin, as long as this allows you to compute the right result efficiently.

I recommend reading the whole subject before starting to program. In particular, it may be best to start by Section 3 to get some datasets on which you can test your code.

1 Subsampling

1.1 Basics

Given a set P of n points in $\mathbb{R}^d$ (for the implementation, you may limit to the case $d = 2$), the first step is often to try and reduce n, so we can use more expensive algorithms without making the running time excessive. Obviously, we still want the reduced point set S to share some properties with P so that what the next algorithm computes on S is meaningful for P. Here, we are going to consider one specific greedy algorithm that first gives a subset S_1 of size 1, then S_2 of size 2, etc, with S_i of size i, that can be interrupted after any step if we are satisfied with the current S_k, and that has the property that $S_i = S_{i-1} \cup \{p_i\}$ for some $p_i \in P \setminus S_{i-1}$, i.e. at each step we pick a point of P that hasn't been selected yet and add it to S. Except for p_1 which can be chosen arbitrarily (for instance randomly, although for testing purposes a deterministic choice would be more convenient), this new point p_i is not picked arbitrarily, we select the one that is farthest from S_{i-1}.

*marc.glisse@inria.fr

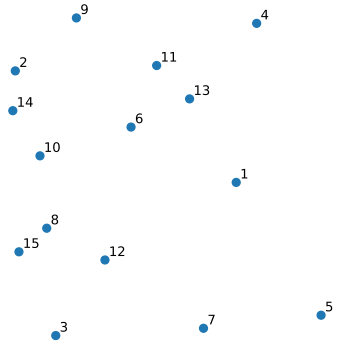

Figure 1: Sequence in order of selection by the algorithm.

More formally, for $x \in \mathbb{R}^d$, we denote $d(x, S) = \inf_{s \in S} d(x, s)$ (where the last $d(\cdot, \cdot)$ is the Euclidean distance between points) and call the s that achieves the minimum the *center* of x (at this time step). p_i is then defined as the point $p \in P \setminus S_{i-1}$ that maximizes $d(p, S_{i-1})$ (ties can be broken arbitrarily). See Fig. 1 for an example showing in which order the points are selected by the algorithm.

Task 1. *Implement a naive algorithm that takes as input P, $p_1 \in P$ and k and computes the sequence of centers $p_1, \ldots, p_k$, with complexity $O(n^3)$.*

This first implementation should be as simple as possible, so you can trust it as a reference when you compare with more advanced implementations later. If the complexity actually depends on k, you can mention a tighter bound than $O(n^3)$. A reminder: you only need to implement the case $d = 2$, although a more general implementation is fine.

1.2 More efficient

As you can guess, it is possible to compute this sequence more efficiently. The key idea is to remember, for each point in $P \setminus S$, what its distance to S is. It is then possible to find in linear time the maximum of this list. Once the new point p_i is determined, updating the list of distances to $S \cup \{p_i\}$ takes linear time as well.

Task 2. *Implement an algorithm that computes the sequence $p_1, \ldots, p_k$, with complexity $O(nk)$.*

1.3 Metric case: small k

There are 2 steps that each take linear time in the algorithm of Task 2: finding p_i, and updating the datastructures, so we need to optimize both of them to gain anything. Part of the difficulty will be that having more datastructures to speed up the first step means more costly things to maintain in the second step.

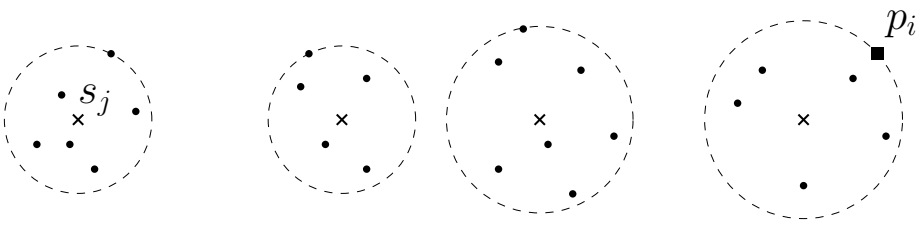

Figure 2: Benefits of the triangle inequality (S is represented with crosses).

The following is a first list of hints about possible optimizations that should help as long as k remains small compared to n.

Finding p_i. Instead of going through all the points of P (or $P \setminus S$) to find the one with maximum distance to S, we could maintain for each center $s \in S$ the list of points of $P \setminus S$ that have s as their center (we call this list the *region* of s), making sure that the one farthest from s can be accessed quickly (we call *radius* of s the maximal distance from s to a point of its region). This way, we only need to iterate on S to find the center with maximal radius.

Updating. You may have noticed that, although we used the Euclidean distance, the algorithms we have seen so far would work for an arbitrary function d on pairs of points. However, we could take advantage of the fact that d is the Euclidean distance and in particular a metric, so it satisfies the triangle inequality $d(a, c) \leq d(a, b) + d(b, c)$. Refer to Fig. 2. After selecting p_i as a new center, since the distance from p_i to s_j is very large compared to the radius of s_j, all the points that had s_j as their center will keep the same center, we do not need to check them one by one. We can test for each old center if the new center is close enough that it could steal some points for its region, and only check the points of these regions in details.

Task 3. *Implement an algorithm that computes the sequence $p_1, \ldots, p_k$ efficiently as long as k is not too large.*

1.4 Metric case (hard)

When k becomes larger, iterating on all the elements of S is too costly. Here are some further hints on how to optimize (they come on top of the previous hints, they don't replace them). In your own interest, you should make a particular effort to keep your code understandable, whether it is through its organization and comments, or with diagrams in the report.

Finding p_i. Instead of repeatedly looking for the element of S with maximum radius in linear time, it may be more efficient to store the elements of S in a datastructure that allows fast access to the one with maximum radius.

Updating. We can store, for each center s, a list of "friends", i.e. other centers $t_1, \ldots, t_{j_s}$ in S that are close enough to s that if the algorithm picks a new center p_i that had s as its center at time $i - 1$ (we call s the *parent* of p_i),

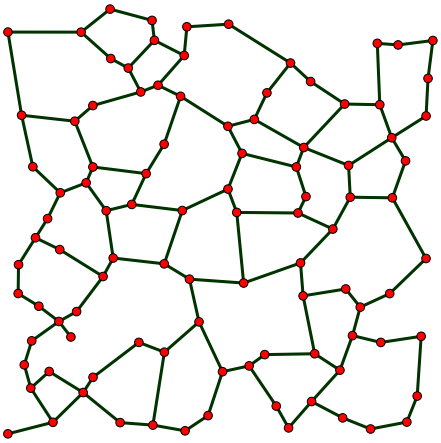

Figure 3: Example of a neighbor graph (picture by David Eppstein).

then all the points that have p_i as their new center at time i had either s or one of the t_j as their center at time $i-1$, so we only need to check if those points change region. This still works if the graph of friends is conservatively bigger than strictly necessary (in the extreme case the complete graph), although we want to keep it reasonably small for efficiency.

Task 4. *Implement an algorithm that computes the sequence $p_1, \ldots, p_k$ efficiently.*

Your report should explain how your algorithm works and why it computes the same output as the previous ones. I do not expect a formal analysis of the running time (too hard), but a quick comment about the complexity of some of the operations can be useful, and may give you hints about possible improvements. It is ok if the worst case looks like it may be worse than Task 2, as long as it helps with reasonable assumptions like a low degree for the vertices of the friends graph.

2 Neighborhood

A plain set of points, without any structure, is hard to interpret. The set starts making more sense once you connect some of the points by segments. There are many ways to decide which points to connect. Here, we will consider one particular definition of neighbor. Intuitively, a neighbor is someone you visit without visiting anyone else along the way.

More formally, given a set P of n points in $\mathbb{R}^d$ (again the implementation can restrict to $d = 2$), we define a graph G with one vertex per point of P, and where p_i and p_j are connected by an edge if there does not exist a *blocker* $p_k \in P$ such that $\max(d(p_i, p_k), d(p_j, p_k)) < d(p_i, p_j)$ (see Fig. 4). See Fig. 3 for a complete example. We will assume that all the pairwise distances in P are distinct (this is not just for convenience, it changes the complexity of the problem).

An important property of this graph (you don't need to prove it) is that the degree of the vertices is upper-bounded by a constant that depends on d

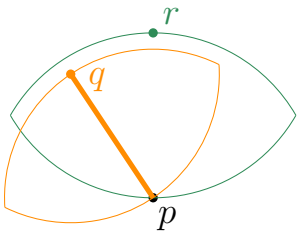

Figure 4: q is a neighbor of p, but r is not.

(intuitively, a point cannot have 2 neighbors in directions that are close, or the closer would be "on the way" when visiting the farther).

2.1 Basic

Task 5. *Implement a naive algorithm that takes as input P and computes the graph G, with cubic complexity.*

2.2 More efficient

We describe here a way to compute all the neighbors of a point p in time $O(n)$. Starting from $P \setminus \{p\}$, we first find the point q closest to p. q is clearly a neighbor of p. Now we notice that for a point $r \in P \setminus \{p, q\}$, q prevents r from being a neighbor of p if and only if r is closer to q than it is to p (why?). We thus remove all of those points r closer to q, and start again with the remaining ones: find the one closest to p, etc. We stop when there are no points left. You can admit that this algorithm outputs $O(1)$ points (the nearest points). We are not quite done though: while this list contains all the neighbors, it may contain extra points. Indeed, we removed points during the first rounds because they could not be neighbors, but they could still play a role in preventing later "closest points" from being neighbors. We thus need an extra pass to check which of those $O(1)$ candidates are actually neighbors, which we can do by brute force, checking if any other point of P prevents it (or we can merge the 2 passes, as long as the behavior is the same).

Task 6. *Implement an algorithm that takes as input P and computes the graph G, with quadratic complexity.*

Remark: this algorithm is not optimal, in particular in $\mathbb{R}^2$ it is possible to compute G in time $O(n \log n)$, but the more efficient algorithms are more complicated and would be too long for this project.

2.3 Incremental

In this section, we do not have all the points from the beginning, we get them one by one and want to maintain the graph.

Task 7. *Implement an algorithm that updates the graph G when we add a point p to P, in linear time.*

You are of course allowed to maintain other datastructures in addition to the graph, if it helps.

Figure 5: Dataset: a few clusters

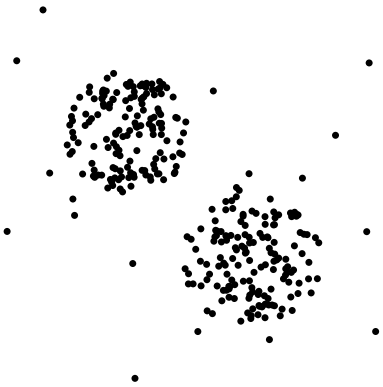

Figure 6: Dataset: a few clusters with background noise

Task 8. *Connect the code from Task 7 with one of the algorithms from Section 1.*

3 Experiments

Datasets. In order to test your algorithms for correctness and run benchmarks, you need datasets. Using some randomness makes it unlikely that distances between different pairs of points will match in a way that causes trouble. A possible first dataset is random points (independent and uniform) in a square, that should already be good enough to debug your programs. You may then want to try datasets that look like Figs. 5 to 7. A dataset where all the points are on the same line could also be interesting when benchmarking Tasks 3 and 4. Generating the datasets yourself has the advantage that you can easily vary n, which is useful for benchmarking. However, you can also download (or create by hand) a dataset where the points have a more visual distribution, for instance they could lie on the silhouette of a cartoon character. For the generation of datasets, it is ok if you want to copy code from the internet (but don't forget to cite your source or it is plagiarism).

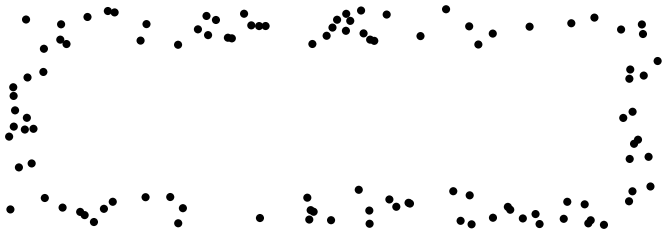

Figure 7: Dataset: an elongated shape with a hole

Plotting. A big advantage of 2d geometry is that we can make drawings. It would be good to use some plots to demonstrate the result of your algorithms. If you are so inclined, you could even make a small video showing the points change color when they enter S in Section 1 (you could also try finding a way to show the region of each point in some way), or showing the evolution of the graph in Task 8 (then you could also compare to what happens if you insert the points in random order instead of following the special subsampling procedure). Use your imagination to find nice ways to showcase your work.

Correctness. For both constructions, we started with a very basic algorithm that hopefully won't have too many bugs. You should run several algorithms on the same dataset and compare their output, as a way to detect some bugs (don't rely on it to find all the bugs though).

Benchmarking. When you implement an algorithm in real code, it is important to check how its running time compares to the theoretical complexity you expect. We expect the quadratic version to be faster than the cubic one, at least when n is not too small. You should push the experiments to values of n high enough that the running time is more than 1 second, and if some algorithms become too slow, you can continue increasing n only for the other algorithms, so they still run with interestingly large datasets.

Do not select just the benchmarks that make your algorithm look best, the surprising ones may be the most interesting. If the results are too surprising, it may indicate an issue with your code though (for instance it is very common to add or remove an element from a container without realizing that it takes more than constant time). For Task 3, it can be interesting to find for what range of values of k that algorithm is fast.

4 Going further

This section is not just optional, it is not part of the project. It is only here in case you are interested. If you want to maximize your mark, it is a better idea to improve the rest of your project and in particular make an attempt at Task 4.

- Generalize your implementation so it can work in $\mathbb{R}^d$ for $d > 2$ (maybe $d = 4$ or 5 if you want to pick just one). How is the performance of the

Figure 8: Heuristic: instead of p_2, could we select a point closer to q?

various algorithms affected? How does it depend on the dataset (some extreme cases could be points sampled uniformly in a ball vs points sampled uniformly on some segment embedded in $\mathbb{R}^d$)?

- The algorithm from Section 1 tends to pick points on the boundary first. In order to approximate a point set, it may be better to pick points that are a bit more central. Can you think of a heuristic to modify the greedy algorithm so it can pick points that are a bit more centered, while still reducing the maximal radius of the regions? See Fig. 8.
- Can you think of a way to maintain the graph of neighbors when we remove of point from P, without recomputing everything from scratch?
- If you draw both the minimum spanning tree and the neighbor graph, do you notice something?

5 Submission

You have to put everything in a single archive (ZIP, or compressed TAR), instructions will come later on where to upload this. This archive must contain your report as a PDF file, all the sources (not just the main code, but also the tests, generation of datasets, etc), and a README helping me understand in which file I can find what, and possibly instructions on how to run the code (I should be able to re-run your experiments easily). If you have a lot of code in a single file, for instance a Jupyter notebook, make sure that it is well organized and easy to navigate, with sections separating the algorithms from the tests. The report should not contain any code (it can contain pseudo-code though), and in particular no picture of code.

Separately (you don't have to include it in the archive), you should prepare slides for a presentation of no more than 10 minutes (strict). The presentation should focus on what you have done, the originality of your approach. It will be followed by questions.