Divide-and-Conquer Algorithms Application to the Closest Pair Problem

The goal of this project is to specify and prove correct several algorithms aiming at solving the *closest pair problem*: finding the minimal distance of two points in a given set of points.

This project is to be carried out using the Why3 tool, in combination with automated provers (Alt-Ergo, CVC4 and Z3). You may use Coq for discharging particular proof obligations, although the project can be completed without it.

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `claude.marche@inria.fr` and `arthur.chargueraud@inria.fr`, no later than **Monday, February 15th, 2016** at 22:00 UTC+1. This e-mail should be entitled "Closest Pair Problem", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The source file `closest_pair.mlw`, completed with your specifications, implementations, and proofs.
- The content of the sub-directory `closest_pair` generated by Why3. In particular, this directory should contain session files `why3session.mlw` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work.

The PDF report should consists of 4 to 8 pages and should follow the sections of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you add should be explained in your report: what they mean and how they help to complete the proof. In case you are not able to fully complete a definition or a proof, carefully describe which parts are missing and explain the problems that you faced. Note that the written report counts for at least half of your grade for the project.

To get started, you need to install the **latest version** of the Why3 and automated provers, and to download the skeleton file `closest_pair.mlw`, which contains the template that you need to complete. Both the details of the installation procedure and the skeleton file may be found on the web page of the course.[1]

1 The Closest Pair Problem

Given a finite set of n points in the plane, the goal is to find the *closest pair*, that is, the distance between the two closest points. Formally, we want to find a distance δ such that there exists a pair of points (a_i, a_j) at distance δ of each other, and such that any other two points are separated by a distance greater than or equal to δ. The aim of this projet is to verify the correctness of algorithms solving the closest pair problem.

We first consider the naive algorithm, which enumerates all pairs in $O(n^2)$. We then consider the classic divide-and-conquer algorithm that solves this problem. For simplicity, we consider an implementation of complexity $O(n \log^2 n)$, even though a version of complexity $O(n \log n)$ can be obtained by applying one further optimization (essentially, integrating a merge-sort into the algorithm).

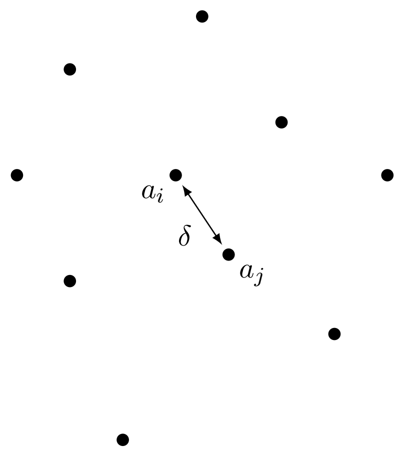

Figure 1: A set of 10 points, the closest pair is (a_i, a_j) with distance δ

[1] `https://www.lri.fr/~marche/MPRI-2-36-1/`

2 Specification and Brute-Force Algorithm

An input consists of n points, with $n \geq 2$. Note that the points are not necessarily distinct. The points are stored in an array, called a, with $a.\mathrm{length} = n$. Thereafter, we call *valid index* an integer between 0 to $n-1$. Moreover, we write $d(a,b)$ the distance between two points a and b. The output of a closest-pair algorithm is a real number δ satisfying the following two properties:

- their exist a pair (i,j) of distinct valid indices such that: $d(a[i], a[j]) = \delta$,
- for all pairs (i,j) of distinct valid indices, we have: $d(a[i], a[j]) \geq \delta$.

To avoid potential complications associated with the existential quantification on the indices of two points realizing the minimal distance, we view the output of the algorithm as a triple of the form (δ, f, s), where f and s are two ghost values, denoting the indices of two points realizing the minimal distance. We write r to denote such an answer triple.

1. *Complete the definition of the predicate* `distinct_indices_in_range`$(i, j, low, high)$*, which asserts that* i *and* j *are two distinct indices, both included in the range* $[low, high-1]$.
2. *Complete the definition of the predicate* `closest_pair_post_for`$(a, r, low, high)$*, which asserts that* r *is a correct answer to the closest pair problem associated with the points from the segment of the array* a *associate with the range* $[low, high-1]$.
3. *Define, in terms of* `closest_pair_post_for`*, the predicate* `closest_pair_post`(a, r)*, which asserts that* r *is a correct answer to the closest pair problem associated with the points contained in the array* a.
4. *The function* `brute_force_search`*, and the auxiliary function* `brute_force_search_sub_array`*, provided in the skeleton file, implements the brute-force algorithm that enumerate all pairs. Formally verify that this code, when applied to an array* a*, produces a triple* r *satisfying the post-condition* `closest_pair_post`(a, r).

3 Closest Pair in 1D

For a gentle introduction to the problem, we begin with the 1-dimensional version of the problem, where all the points lie on the x-axis. In this section, we thus assume that the array contains points that all lie on the x-axis, that is, that all have y-coordinate equal to zero. We have $d(a,b) = |a.x - b.x|$.

We consider two algorithms for solving the problem. The first algorithm sorts the points according to their x-coordinate, and then enumerates adjacent pairs of points, in time $O(n)$. The second algorithm also sorts the points, but then proceeds using a divide-and-conquer approach. More precisely, this algorithm splits the array in two halves, recursively searches for minimal distances in each halves, and then compares the two distances together and also against the distance between the two points falling on either side of the border between the two halves. This second algorithm is not more efficient than the first algorithm, but it gives good practice for the generalization to the 2D divide-and-conquer algorithm.

5. *Complete and verify the function* `linear_search`*, which assumes a sorted array and performs a linear enumeration of the pairs.*
6. *Fill and then prove the code of function* `divide_and_conquer`*, which assumes a sorted array and implements the divide-and-conquer approach.*
7. *Prove the correctness and the termination of the function* `full_divide_and_conquer`*, which takes as input an array, sorts the array into another array, and then calls the function* `divide_and_conquer`*.*

Note that the sorting function is specified using a ghost bijection function, relating indices before the sorting operation with indices after the sorting operation; details can be found in the skeleton file.

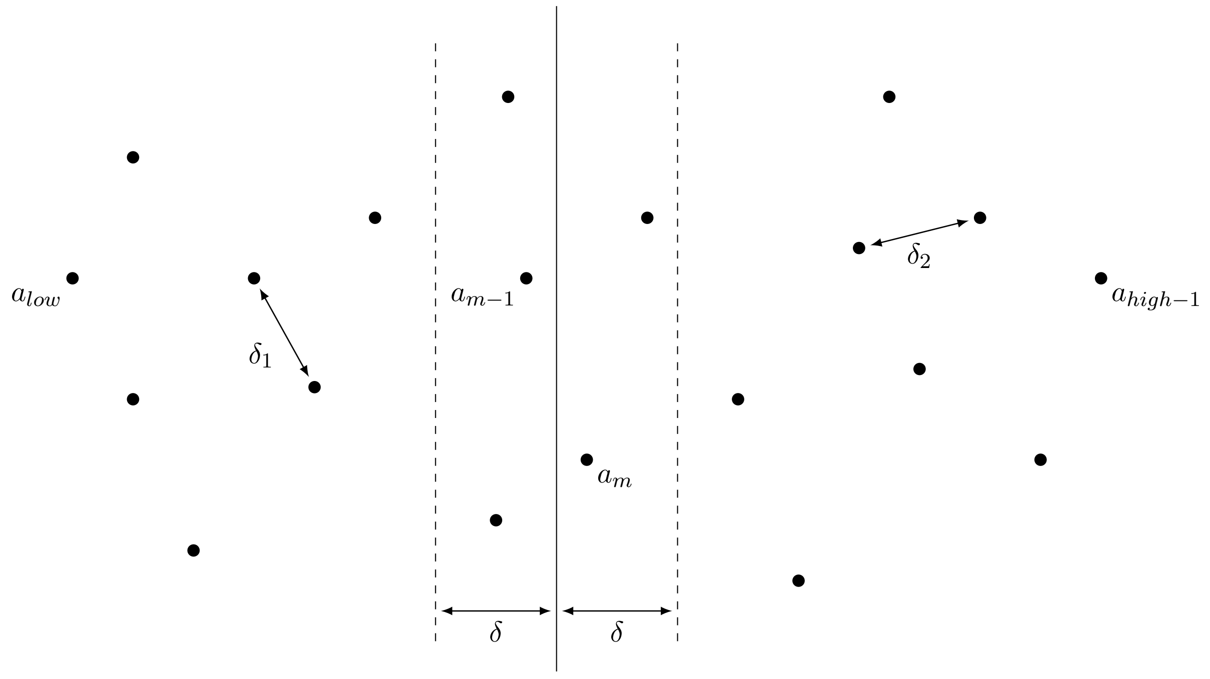

Figure 2: Divide-and-Conquer algorithm for the closest pair problem.

4 Closest Pair in 2D

The points may now lie anywhere in the plane. We could define $d(a, b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$, but in fact we leave the distance function d abstract. The algorithms that we consider only require the distance function to satisfy the following properties, which we axiomatize:

- Non-negativity: $d(a, b) \geq 0$.
- Symmetry: $d(a, b) = d(b, a)$.
- Inequality w.r.t. projections: $d(a, b) \geq |a.x - b.x|$ and $d(a, b) \geq |a.y - b.y|$.

The divide-and-conquer algorithm that we consider to solve the closest pair problem in dimension 2 is described next. First, it sorts the input array with respect to the x-coordinate. Then, it proceeds recursively. The recursive function expects two bounds low and $high$ delimiting the segment of a to process, and returns an answer triple describing the closest pair among the points in that segment. The recursive function is implemented as follows.

a. If the size of the sub-array is equal to 7 or less, invoke the brute-force algorithm and return its result.

b. Otherwise, compute the middle index $m = low + (high - low)/2$.

c. Recursively compute the minimal distance δ_1 for sub-array $a[low, m - 1]$ and δ_2 for sub-array $a[m, high - 1]$, and let $\delta = \min(\delta_1, \delta_2)$ be the current candidate.

d. As illustrated on Figure 2, if there exists a pair of points whose distance is strictly smaller than δ, then these two points must lie in the vertical *strip* between the dotted lines. This strip is centered on the x-coordinate in the middle between $a[m - 1]$ and $a[m]$, and it has total width 2δ.

e. Let δ_3 be the distance among the closest pair of points in the strip. If $\delta_3 < \delta$, then the final result is δ_3. Else, the final result is δ.

It remains to explain how to efficiently find the closest pair in the vertical strip. To that end, we rely on an auxiliary function called `search_strip`(a, δ). This function exploits the fact that we are only interested in finding pairs whose distance in less than δ. If there is no such pairs, the function returns δ. The function `search_strip` is implemented as follows. First, we sort the points in the strip with respect to the y-coordinate. We then perform a variant of the brute-force algorithm on these points, where the internal loop gets interrupted as soon as the difference along the y-axis between the point associated with the outer loop and the point associated with the inner loop exceeds δ. Indeed, any points we might find in subsequent iterations of the inner loop would necessarily be further than δ away. We refer to the code for details.

8. *Complete the code and the specification of the `search_strip` function.*
9. *Complete the specifications and code of auxiliary functions `left_border` and `right_border`.*
10. *Complete the code, specification and proof for the recursive function `divide_and_conquer` that implements the algorithm. Notice that this is the most challenging question; do not hesitate to submit a partial solution.*
11. *Complete the code and the proof of the function `main_algorithm`, which performs the initial sorting and makes the initial invocation of the recursive function.*

5 Complexity Analysis of the Processing of the Strip

Note that this section is independent from questions 9 to 11.

The function `search_strip` is implemented using two nested loops over the set of points in the vertical strip of width 2δ. At first sight, it may thus appear to have worst-case quadratic complexity. It turns out that this function is in fact of linear complexity. An essential observation is that, for any given iteration of the outer loop, the inner loop may perform at most 7 iterations. As we are going to prove, after 7 iterations of the inner loop, it must be the case that the difference along the y-axis between the point associated with the outer loop and the point associated with the inner loop exceeds δ, hence the break condition gets triggered.

We formalize the claim as follows. Let `i` denote the index associated with the outer loop and `j` denote the index associated with the inner loop. We add to the inner loop the invariant: `j ≤ i + 6`. In order to preserve this invariant when `j` increases, one need to show that, at the end of the inner loop, the assertion `j ≠ i + 6` holds. To prove this assertion, it is helpful to add a ghost statement of the form:

```
if j = i + 6 then too_many_points_in_rectangle ...;
```

where `too_many_points_in_rectangle` is a ghost function that ensures **`false`**, that is, it proves a contradiction.

The informal reasoning justifying the lemma `too_many_points_in_rectangle` is as follows. Assume that `j = i + 6` and that the break condition of inner loop has not been triggered, meaning that: `b[i+6].y < b[i].y + δ`. It means that the 7 points (from `b[i]` to `b[i+6]`) lie in a rectangle of dimensions $\delta \times 2\delta$, centered on the vertical line that that divides the set of points into two parts. Thus, at least 4 of these 7 points must fall on a same side of the vertical line. This invocation of the pigeon-hole principle is captured by a lemma called `pigeon_hole`. So, we have 4 points that lie in a square of dimensions $\delta \times \delta$ (with one of the sides being excluded from the square), and whose pairwise distance is at least δ (recall that δ is the minimum pairwise distance computed by the recursive calls). Such a configuration is impossible, as established by the geometric lemma called `too_many_points_in_square`, provided in the skeleton file.

12. *Complete missing parts from the ghost function `pigeon_hole`, which returns the indices of 4 points lying on the same side of the vertical line.*
13. *Complete missing parts from the ghost function `too_many_points_in_rectangle`, which establishes a contradiction in the case where `j = i + 6` becomes true.*
14. *Complete missing parts from the function `search_strip_complexity`, which is like `search_strip` but ensures the invariant `j ≤ i + 6` in the inner loop.*