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

Project report

2 Specification and Brute-Force Algorithm

Questions 1, 2, 3

Simple réécriture de l'énoncé.

Question 4

brute_force_search_sub_array – La précondition décrit que $a[low..high-1]$ est un sous-tableau de a de longueur au moins 2, et *closest_pair_post_for* fournit exactement la postcondition voulue. Les invariants décrivent que $(!min, !f, !s)$ est toujours un triplet de réponse autorisé (indices distincts dans les bornes, $!min$ distance entre les points correspondants), et de plus $!min$ est inférieur aux distances pour les couples de points déjà parcourus dans la boucle considérée. Pour la boucle sur i, il s'agit des couples d'indices valides x et y avec au moins un d'eux (sans perdre de généralité x) strictement inférieur à i. Pour celle sur j, les couples d'indices i et y, $i+1 \leq y < j$. Si on s'en tient là (sans le dernier assert avec m_loop_i) on n'arrive pas à montrer la conservation de l'invariant de la boucle sur i exprimant que $!min$ est inférieur aux distances déjà étudiées, car pour passer de i à $i+1$ on n'arrive pas à réutiliser le résultat pour i qui porte sur $!min$ qui a pu être modifié au cours de la boucle sur j. Une solution consiste à reparler des couples avec $x < i$ dans la boucle sur j (invariant commenté, qui peut remplacer les deux derniers invariants) car pour une étape de cette boucle, le prouveur se rend compte que la nouvelle valeur de min est inférieure ou égale à la précédente. Mais j'avais envie de garder mon invariant ne portant que sur les "nouveaux" couples explorés dans la boucle sur j et de comprendre comment rajouter juste ce qu'il manquait, et cela a fonctionné en ajoutant que $!min$ au cours de la boucle sur j est toujours inférieur à la valeur de

min à l'entrée de la boucle.

brute_force_search – Le tableau doit être de longueur au moins 2, et *closest_pair_post* fournit exactement la postcondition voulue.

3 Closest Pair in 1D

Question 5

Un invariant exprimant que $(!min, !f, !f + 1)$ est une réponse valide pour le sous-tableau $a[0..i]$ a suffi.

Question 6

L'idée est que quand on considère une paire d'indices valides x et y, il y a trois cas : tous deux inférieurs à $middle - 1$, tous deux supérieurs à $middle$, et sans perdre de généralité $x < middle \leq y$. C'est le dernier cas qui pose le plus de difficulté, et l'assertion correspondante, exprimant que dans ce cas la distance est au moins d, est le plus important de la preuve. C'est une conséquence de l'ordre des indices et du fait que le tableau est trié, mais dans ce contexte les prouveurs n'arrivent pas à la montrer (en tout cas pas dans les 5s imparties). En revanche, c'est prouvé très rapidement dans le lemme *consecutive_closer_than_apart*, grâce auquel l'assertion est ensuite tout aussi rapide avec Alt-Ergo. Les deux autres cas sont immédiats suite aux appels récursifs, puis il s'agit de montrer que les distances sont toutes supérieures au minimum des trois bornes inférieures obtenues dans les trois cas : CVC3 arrive bien à effectuer cette synthèse ; quelques assertions échelonnant ce raisonnement permettent d'améliorer un peu le temps qu'il met. La dernière assertion est nécessaire, mais en comparant à celle de l'autre branche du if, je pense que c'est à cause de la limite de 5 secondes que j'impose.

Question 7

L'assertion fournit la partie la plus difficile de la postcondition, avec une étape intermédiaire essentielle qui indique comment utiliser et combiner les faits à disposition : a est une permutation de b (vice-versa historiquement, mais maintenant c'est ce sens-là qui nous intéresse) et r est une réponse de paire la plus proche pour b, conséquences des postconditions des fonctions appelées juste avant.

4 Closest Pair in 2D

Question 8

Remarque générale – *search_strip* est essentiellement une variante de la fonction $brute\_force\_search\_sub\_array$, un peu modifiée pour commencer avec une distance *delta* à renvoyer si c'est un minorant à la place de la distance minimale, et améliorée avec un tri du tableau par ordonnée pour parfois arrêter la boucle sur j prématurément

Précondition – La précondition indiquée dans l'énoncé, exigeant que tous les points sont dans une bande verticale de largeur $2 * delta$, n'est **pas nécessaire pour vérifier la fonction** *search_strip* ; je l'ai fait sans la mettre. On comprend assez facilement pourquoi : $brute\_force\_search\_sub\_array$ ne demande rien sur la disposition des points, et les modifications n'utilisent pas d'hypothèse sur les abscisses.
Il est cependant utile de laisser cette précondition, uniquement pour la question 14 dans la partie 5 sur la complexité. En effet, *search_strip_complexity* utilise vraiment la précondition, et c'est la seule partie du code étudiée dans la partie 5 avec l'idée qu'il suffit de remplacer les appels à *search_strip* par *search_strip_complexity* (à quelques détails près comme la construction de *side*). On veut donc que la précondition soit vérifiée à chaque appel à *search_strip*.

Invariants – m contient la plus petite distance rencontrée si elle est inférieure à *delta*, sinon *delta*. Dans le cas où $!m < delta$, on a vraiment presque la fonction $brute\_force\_search\_sub\_array$, c'est pourquoi on retrouve les invariants de cette fonction sous cette hypothèse (cf question 4). Si $!m = delta$, les valeurs de $!f$ et $!s$ ne sont pas importantes : on a simplement l'invariant que toutes les distances vues sont supérieures à *delta*. J'ai mis $!m \geq delta$ plutôt que $!m = delta$ à droite de $\rightarrow$; cela revient au même théoriquement car on a toujours $!m \leq delta$, mais ainsi le prouveur remarque bien qu'on est toujours dans un des deux cas. Rajouter explicitement l'invariant $!m \leq delta$ fonctionnerait sans doute aussi, mais il n'y a pas vraiment d'inconvénient à mettre $\geq$ plutôt que $=$.

Assertions – Les seules assertions nécessaires, un dans chaque branche du *if* final, sont similaires à l'assertion de la question 7 avec la même explication. L'assertion juste avant *raise Break* explique pourquoi la boucle peut être interrompue, et aide à prouver plus facilement la conservation

des invariants de la boucle sur i. Les assertions qui précèdent chacune des assertions nécessaires accélèrent ces dernières en mettant en évidence la conséquence pertinente des invariants et de la branche du *if* choisie. Enfin, bien que l'assertion sur f' et s' dans la branche *then* soit naturelle puisqu'elle consiste en deux composants de la postcondition, j'ai été étonnée par sa contribution. En effet, il y a plusieurs autres endroits dans le projet où on a une situation similaire, mais où rajouter une telle assertion ne gagne rien. De plus, si on utilise un seul prouveur parmi Alt-Ergo et CVC3, cette assertion permet de passer de 3s à 2s ou de 2s à 1s (temps total pour l'assertion et les postconditions dans le cas du *then*, comparé au temps pour les mêmes postconditions si l'assertion est absente) ce qui n'est déjà pas mal, mais en combinant les deux, CVC3 prouve presque instantanément l'assertion, grâce à laquelle Alt-Ergo prouve les postconditions presque instantanément aussi !

Question 9

left_border – En sortie de boucle, on veut que le résultat renvoyé $!l$ soit dans les bornes (premier invariant), et vérifie les propriétés sur la position des abscisses selon la position des indices par rapport à $!l$. Pour ces dernières, le tableau étant trié par abscisse, il suffit d'avoir $a[!l - 1].x \leq mu - delta$ si $!l > low$, ce qui est assuré par la négation de la condition du *while* en sortie de boucle, et $a[!l].x > mu - delta$ si $!l < middle$, ce qui est la raison d'être du deuxième invariant, dont la conservation est assurée par la condition du *while*.

right_border – Comme les inégalités stricte et large sont inversées dans les postconditions, cette fonction est construite comme le "miroir" de la précédente, notamment la boucle "monte" au lieu de "descendre". Les invariants suivent les mêmes principes.

Question 10

(a) Ces assertions ne sont pas nécessaires mais aident le prouveur. Ils contiennent la partie des postconditions des appels récursifs à *divide_and_conquer* qui sera le plus utilisée par la suite, en remplaçant déjà $r1/2.delta$ par d qui est le minimum des deux.

(b) Ces propriétés sont la raison d'être de mu et seront beaucoup utilisées par la suite.

(c) Il s'agit de la précondition pour *search_strip*, avec le *mu* définit plus haut comme témoin pour le quantificateur existentiel de la précondition.

(d) L'assertion (d3) fournit la partie la plus difficile de la postcondition à montrer. Elle s'obtient grâce à une disjonction de cas sur i et j, indices distincts entre *low* et *high*. Le cas le plus délicat et le plus récemment obtenu est celui où i et j sont entre $left$ et *right*, donné par l'assertion (d2). Comme on est dans la branche où $res.delta < d$, la postcondition de *search_strip* donne *closest_pair_post temp res* ce qui implique directement (d1), qui permet de montrer (d2) dans laquelle on indique au prouveur comment passer d'indices de a entre $left$ et *right* à des indices de *temp*. Deux autres cas pour montrer (d3) sont fournis par (a) avec toujours $res.delta < d$. Pour les cas restants, le prouveur semble capable d'utiliser (b) et les postconditions de $left\_border$ et *right_border* pour montrer que si (quitte à échanger i et j) $i < left$ et $j \geq middle$ alors *dist* $a[i]$ $a[j] \geq d$ car $a[i].x \leq mu - d$ et $a[j].x \geq mu$, de même pour $i < middle$ et $j \geq right$ avec $a[i].x \leq mu$ et $a[j].x \geq mu + d$. J'ai vraiment l'impression que le prouveur utilise à peu près cette approche car j'avais ajouté des assertions correspondant à ce raisonnement et cela ne changeait quasiment pas le temps mis pour cette preuve.

(e) Pour une fois, (d3) ne suffit pas (en tout cas pas avec une limite de 5s avec Alt-Ergo ou CVC3), il faut aussi aider le prouveur à montrer quelque chose comme (e2). L'assertion clé et un peu surprenante et (e1). Elle est destinée à être appliquée à $t = temp$ et résultat $re = res$, alors pourquoi un *forall* ? Si j'utilise directement ces valeurs, les prouveurs ont beaucoup de mal à montrer ce résultat qui semble pourtant évident puisqu'il s'agit de la définition de *closest_pair_post*, sans doute parce qu'ils disposent déjà de beaucoup d'informations sur *temp* et *res* et essaient de les utiliser. Grâce au *forall* cependant, il n'y a rien d'autre à utiliser que la définition de *closest_pair_post*, si bien que c'est presque instantané de prouver (e1). Ensuite, grâce à (e1), on arrive à montrer la postcondition, (e2) n'est même pas nécessaire mais accélère un peu la preuve et permet de mieux comprendre la présence de (e1) quand on lit le code.

(f) Similaire à (d). Pas besoin d'équivalent à (d1), sans doute parce que la postcondition de *search_strip* pertinente dans ce cas est plus facile à utiliser.

Question 11

Similaire à la question 7, avec en plus une assertion optionnelle qui met en valeur la partie de *closest_pair_post* utilisée pour l'assertion principale.

5 Complexity Analysis of the Processing of the Strip

Question 12

Le premier invariant, qui concerne nl et nr, permet d'assurer qu'en sortie de boucle le plus grand des deux est supérieur à 4. Les autres assurent que les éléments d'indices strictement inférieurs à $!nl$ ou $!nr$ selon le tableau sont du bon côté, dans $[\![k, i[\![$ et triés par ordre croissant strictement. Tout cela est directement utilisé dans la postcondition pour le tableau finalement choisi, sauf la borne supérieure i (pour la postcondition $[\![k, k+6]\!]$ suffirait) qui sert à préserver la propriété de tri.

Question 13

Il suffit de définir les bonnes translation et symétrie. Alt-Ergo prouve la fonction (après *split*) sans assistance. Les fonctions locales servent seulement à factoriser le code.

Question 14

search_strip commence par construire b, une permutation de a triée. Ici on construit aussi sb, obtenu à partir de *side* avec la permutation qui permet de passer de a à b (par construction, et exprimé par le premier invariant de la boucle). L'objectif est que les propriétés sur *side* et a des préconditions soient aussi vraies pour sb et b. Le second invariant de la boucle de construction de sb et les deux assertions qui suivent cette boucle montrent ces propriétés sur sb et b. Ces dernières seront essentielles pour montrer les préconditions correspondantes lors de l'appel à *too_many_points_in_rectangle*.

Les invariants des boucles sur i et j sont les mêmes que dans *search_strip*, avec en plus $!m \leq delta$ dans celle sur i dont on verra l'intérêt plus tard, et $j \leq i + 6$ qui est le but de la question.

L'assertion juste avant *raise Break* n'est plus facultative comme dans *search_strip* et on a même besoin de rajouter une autre assertion avant celle-ci pour rester

en-dessous de 5 secondes. Cela s'explique parce qu'il y a plus d'hypothèses à ce point-là que dans *search_strip* (propriétés sur *sb*, invariants supplémentaires...) qui "noient" les hypothèses qui sont vraiment utiles. De même, la plupart des assertions dans le *if* final non modifié deviennent sans doute nécessaires (je n'ai pas vérifié pour toutes).

Les dernières assertions rajoutées sont juste avant l'appel à *too_many_points_in_rectangle*. Elles ne sont pas nécessaires mais accélèrent beaucoup les preuves des préconditions pour cet appel. On y voit notamment l'intérêt de l'invariant $!m \leq delta$: on a $b[j].y < b[i].y + !m$ car on n'a pas levé $Break$, mais veut $b[j].y < b[i].y + delta$ qu'on obtient grâce à l'invariant.