N° d'ordre : 175 ID

THÈSE

présentée par

Christophe LEMAIRE

pour obtenir le titre de

DOCTEUR

DE L'UNIVERSITÉ JEAN MONNET ET
DE L'ECOLE DES MINES DE SAINT-ETIENNE

Spécialité : Informatique

TRIANGULATION DE DELAUNAY ET ARBRES MULTIDIMENSIONNELS

Date de soutenance : 19 décembre 1997.

Composition du jury :

MM.	Luc Devroye	Rapporteurs
	René Schott	
MM.	Bernard Laget	Examinateurs
	Olivier Devillers	
	Bernard Lacolle	
	Jean-Michel Moreau	
	Bernard Péroche	
	Jean-Noël Theillout	

Thèse préparée au sein du

SETRA : Service d'Etudes Techniques des Routes et Autoroutes, 92225 Bagneux, France.

Table des matières

Liste des figures

Remerciements

Je tiens tout d'abord à remercier M. Liochon, Ingénieur Général des Ponts et Chaussées, qui, il y a quelques années, m'a donné le goût de l'informatique, avec ses fameux problèmes d'IA en Prolog. Depuis, l'informatique a été le fil conducteur de mes études : Ecole Nationale des Travaux Publics de l'Etat et DEA Images à Saint-Etienne.

Je tiens à remercier Jean-Michel Moreau, Roland Jegou et Dominique Michelucci, qui, à l'occasion des cours de DEA, m'ont fait découvrir la Géométrie Algorithmique. Ces quelques années de thèse ont renforcé mon engouement pour cette matière.

Je n'oublie pas le service de documentation du SETRA, et en particulier Catherine Mallaret, Liliane Sardais et Nadine Volante pour les nombreuses recherches d'articles, de rapports, de thèses, d'ouvrages... dont certaines demandaient un réel talent de détective !

Je remercie J. N. Theillout pour avoir accepté avec enthousiasme ma candidature pour effectuer une thèse en Géométrie algorithmique au sein du SETRA, et pour m'avoir offert, surtout dans la seconde moitié de thèse, de très bonnes conditions de travail : matériel informatique, documentation, moyens de déplacement et de communication, temps libre... Merci à tous les membres du projet ARCAD pour la bonne humeur qui règne dans cette équipe.

Merci à Bernard Péroche qui a maintenu sa confiance en moi, alors que mon environnement professionnel, dans l'année qui suivait le D.E.A., était totalement incompatible avec un travail de thèse !

Je remercie aussi tout particulièrement l'Ecole des Mines de Saint-Etienne, pour son accueil chaleureux lors de mes nombreuses visites. Citons Dominique Michelucci, Marc Roelens, Annie Corbel, Bernard Péroche et surtout Jean-Michel Moreau... que ceux que j'oublie, veuillent bien m'en excuser.

Merci aussi aux doctorants en Géométrie Algorithmique, Philippe Biscondi et Hélymar Balza-Gómez, pour leur soutien.

Merci à René Schott qui m'a accueilli dans ses locaux à Nancy, pour suivre la série de cours dispensés par l'Institut Max Plank de Sarrebrück, avec des professeurs aussi prestigieux que R. Seidel... Il me fait aussi l'honneur d'accepter d'être rapporteur de cette thèse et je le remercie pour ses remarques constructives concernant ce document.

Je tiens aussi à remercier Luc Devroye pour son accueil très sympathique à l'université McGill de Montréal, et qui me fait le grand honneur d'être rapporteur de cette thèse.

Je suis reconnaissant à Ferran Hurtado de m'avoir informé, après un exposé aux

Journées Franco-Espagnoles de Géométrie Algorithmique à Barcelone, des travaux très récents de Shewchuk sur la triangulation de Delaunay.

Je remercie aussi Christine Barthe pour la relecture de ce document, Philippe Nouaille pour sa collaboration dans le codage des algorithmes de triangulation, Jacques Hervé, géomètre algorithmicien qui s'ignore, pour l'aide et l'intérêt qu'il a toujours apportés à mon travail, Tam Vo-Dinh pour son aide lors du portage de mes programmes sur le super-calculateur Convex C3, P. Brehmer et C. Simon pour avoir utilisé ce travail dans le cadre du projet OPÉRA (CAO ouvrages d'arts SETRA) et pour m'avoir en retour donné une variante algorithmique intéressante, et enfin Jean-Noël Theillout qui fut mon directeur technique.

Merci à Olivier Devillers, pour m'avoir permis dans un premier temps de mieux connaître la Géométrie Algorithmique par l'intermédiaire de son journal GéDéoN (page 189), pour sa collaboration concernant le problème cité page 87, pour le prêt de son programme de triangulation de Delaunay dynamique (qui va nous permettre d'effectuer une comparaison expérimentale avec notre programme) et enfin pour avoir accepté de faire partie du jury de cette thèse. Je remercie aussi les autres membres du jury : Bernard Lacolle, Bernard Laget, Jean-Michel Moreau, Bernard Péroche et Jean-Noël Theillout.

Je suis particulièrement reconnaissant à Olivier Devillers et Luc Devroye pour la qualité et la pertinence de leurs remarques et commentaires concernant ce travail, qui ont permis d'améliorer et de corriger certains points.

Merci aux parents et amis qui m'ont aidé et "supporté" (dans les deux sens du terme) pendant ces quelques années.

Je n'oublie pas Gregory Six pour sa participation au codage des algorithmes de localisation, et enfin, je souligne la contribution déterminante de Jean-Michel Moreau qui a encadré cette thèse. Il a toujours su être présent et disponible quand il le fallait, et a permis de débloquer des situations "critiques". J'ai particulièrement apprécié l'intérêt qu'il porte à mon travail et son souci constant d'en améliorer la qualité. Il m'a initié au monde de la recherche avec tout ce qui l'accompagne : publications, conférences, enseignement, lectures, échanges d'idées avec d'autres chercheurs... et j'ai pu découvrir à quel point ce travail est passionnant.

NOTATIONS

Les complexités

$O()$	majorant pour la complexité asymptotique
$\Omega()$	minorant pour la complexité asymptotique
$\Theta()$	valeur exacte pour la complexité asymptotique
$o()$	majorant jamais atteint pour la complexité asymptotique
$\omega()$	minorant jamais atteint pour la complexité asymptotique

Les ensembles

$\mathbb{N}$	ensemble des entiers naturels
$\mathbb{R}$	ensemble des nombres réels
$\mathbb{E}^k$	espace euclidien de dimension k
k	dimension de l'espace
S	ensemble de points (sites)
A	ensemble d'arêtes
$\|E\|$	cardinal de l'ensemble E
$<_l$	relation d'ordre lexicographique

Les fonctions mathématiques

$\alpha(n)$	inverse de la fonction d'Ackermann
$\mathcal{B}()$	fonction Beta()
$\Gamma()$	fonction eulérienne de seconde espèce Gamma()
$\log^*$	logarithme itéré (nombre d'itérations de la fonction log nécessaires pour obtenir une valeur inférieure à 1)
$\lceil\ \rceil$	partie entière supérieure appelée aussi plafond
$\lfloor\ \rfloor$	partie entière inférieure appelée aussi plancher

La géométrie

$\triangle$	triangle
$\triangle abc$	triangle de sommets a, b, c
$\Box$	carré
$\Box abcd$	carré de sommets a, b, c, d
$\bigcirc$	cercle
$\bigcirc(A, B, C)$	cercle circonscrit aux points A, B et C
$\mathcal{H}$	hyperplan
$\mathcal{H}^\perp$	droite orthogonale à l'hyperplan $\mathcal{H}$
$\mathcal{S}()$	surface d'un polyèdre
$\mathcal{V}()$	volume d'un polyèdre
$\mathcal{B}(s, r_1)$	boule centrée en s de rayon r_1
$\mathcal{C}(sp, \Theta)$	cône de sommet s, d'axe de symétrie sp et d'angle Θ
$\mathcal{C}_{B1}(sp, \Theta)$	intersection du cône $\mathcal{C}(sp, \Theta)$ et de la boule $\mathcal{B}_1$ centrée en s
$\mathcal{P}$	pyramide (c'est ici l'intersection d'un cône et d'une boule centrée au sommet de ce cône)
$\vec{a}$	vecteur $\vec{a}$
$d(A, B)$	distance euclidienne entre les points A et B
$\Vert\vec{a}\Vert$	norme euclidienne du vecteur $\vec{a}$
$\mathrm{Det}(\vec{a}, \vec{b})$	déterminant des vecteurs $\vec{a}$ et $\vec{b}$
$\widehat{ABC}$	angle(ABC) : angle formé par les vecteurs $\overrightarrow{BA}$ et $\overrightarrow{BC}$

Les probabilités

$\binom{n}{k}$	coefficient binomial C_n^k
p() ou P()	probabilité associée à un événement
E()	espérance d'une variable aléatoire
$N(m, \sigma)$	distribution normale de moyenne m et d'écart-type σ
$U(a, b)$	distribution uniforme sur le segment $[a, b]$

Les triangulations

$T(S)$	triangulation de l'ensemble de points S
$DT(S)$	triangulation de Delaunay de l'ensemble de points S
$PPV(M)$	Plus Proche Voisin du point M
$V(M)$	région de Voronoi associée au site M
$\mathcal{G}()$	grain d'une triangulation
$\mathcal{F}(T)$	finesse d'une triangulation
$<_\triangle$	relation d'ordre "inférieur à" pour comparer des triangulations
n ou N	nombre de points (sites)
n_{ec}	nombre de points de l'enveloppe convexe d'une triangulation
e	nombre d'arêtes d'une triangulation
t	nombre de triangles

EQUIVALENCES français–anglais

Beaucoup de termes techniques en algorithmique (géométrique ou non) ont une dénomination anglaise. Leur traduction est parfois difficile et inadaptée. Souvent, nous avons préféré garder la terminologie anglaise, plus universelle. Voici toutefois un essai de traduction de quelques termes anglais utilisés dans la rédaction de ce mémoire.

2d-tree	2d-arbre (équilibré)
2d-Search-and-Walk	2d-arbre-et-marche
adaptive 2d-tree	2d-arbre adapté
Binary-Search-and-Walk	dichotomie-et-marche
buckets	cellules, tiroirs
bucketing	hachage en cellules, tri par paquets
Closest-and-Walk	plus-proche-voisin-et-marche
clusters	amas (fortes concentrations de sites)
Delaunay tree	arbre de Delaunay
derandomization	dérandomisation
Divide and Conquer	Diviser pour Régner, Diviser et Fusionner Partage et Fusion, Division – Fusion...
Euclidean Minimum Spanning Tree	EMST, arbre couvrant minimal eucidien
Jump-and-Walk	sélection-et-marche
kd-tree	kd-arbre
Nearest Neighbo(u)r	plus proche voisin
off-line	qui n'est pas en ligne
on-line	en ligne (pour un algorithme)
Oversampled-Binary-Search-and-Walk	dichotomie-suréchantillonnée-et-marche
random	aléatoire
randomization	randomisation
random 2d-tree	2d-arbre randomisé
random adaptive 2d-tree	2d-arbre adapté randomisé
range searching	recherche dans un domaine
trapezoidation	carte des trapèzes
Walk-Through	marche-vers

INTRODUCTION

La Géométrie Algorithmique est une science relativement récente, née avec la thèse de Shamos autour des années 1975 ([180]). Il a poursuivi ses travaux avec la collaboration de Preparata pour constituer le livre “Computational Geometry - an introduction” ([165]), qui sert aujourd'hui encore de référence.

Les premiers résultats en géométrie constructive remontent à Euclide et les instruments utilisés étaient alors le compas et la règle. Des développements remarquables ont eu lieu au cours des deux derniers siècles. Maintenant, il s'agit d'exploiter au mieux les possibilités des ordinateurs capables de traiter des milliers de segments en une fraction de seconde : c'est l'objet de la Géométrie Algorithmique. La puissance toujours accrue des machines ne donne que plus d'intérêt à cette nouvelle discipline.

Ses principaux domaines d'application se trouvent en robotique, vision par ordinateur, conception assistée par ordinateur, “design” industriel, images de synthèse, systèmes d'information géographique et même en médecine, biologie moléculaire, cristallographie, astrophysique, mécanique des fluides...

Cette science présente de nombreux attraits : l'algorithmicien sera fasciné par sa richesse algorithmique exceptionnelle; le néophyte, s'il fait preuve d'intuition, pourra découvrir de nouveaux algorithmes (c'était surtout vrai du temps des pionniers de la Géométrie Algorithmique) ; le mathématicien pourra utiliser des techniques très sophistiquées pour l'analyse des algorithmes et, bien entendu, le géomètre pourra mettre en pratique ses connaissances.

Actuellement, cette science se situe à un carrefour : des algorithmes extrêmement sophistiqués sont apparus ([45] par exemple) et une scission entre la théorie et la pratique devient flagrante. Dans la pratique, les performances d'un algorithme se mesurent en termes de temps d'exécution et de place mémoire. Etant donné que ces paramètres dépendent de la machine utilisée, de l'habileté du programmeur ainsi que du langage de programmation choisi, la Géométrie Algorithmique s'est défini un modèle abstrait de calculateur (*calculateur réel à accès aléatoire*). En particulier, la machine est supposée travailler en précision infinie (ce n'est bien sûr jamais le cas en pratique !), avoir une capacité mémoire infinie (en réalité, la gestion de la mémoire est un problème presque quotidien pour le programmeur !) et les algorithmes sont jugés d'après leurs comportements asymptotiques en fonction de la taille des données en entrée (mais l'industriel n'a que faire d'un algorithme qui devient plus performant à partir d'une taille des données en 10^{80} par exemple, ce qui est à peu près le nombre estimé d'atomes présents dans l'univers observable!). **Le calculateur réel à accès aléatoire n'existe pas !** Actuellement, une réflexion est en cours pour essayer de recréer des liens entre théoriciens et praticiens ([46] par exemple). Souhaitons que cette tentative ne reste pas vaine! La Géométrie Algorithmique devrait être aussi utile au programmeur que l'est, par exemple, la théorie de la Résistance des Matériaux à l'ingénieur en Génie Civil!

Cette thèse a voulu se situer à la frontière commune (elle existe encore!) entre ces deux mondes: seuls des algorithmes qui fonctionnent bien en pratique, ont fait l'objet d'une analyse théorique.

Revenons aux deux pierres d'achoppement que constituent la précision supposée infinie des machines et l'évaluation des algorithmes d'après leurs comportements asymptotiques.

Le principal inconvénient de la précision finie des ordinateurs est de rendre incohérents la plupart des algorithmes géométriques. Par exemple, si un point est très proche d'un segment, la machine ne pourra décider avec certitude si le point est à gauche ou à droite du segment. Des incohérences topologiques risquent d'apparaître après plusieurs choix erronés, qui provoqueront l'arrêt de l'algorithme avant la fin du calcul. Résoudre les problèmes dus à l'imprécision numérique constitue certainement l'un des principaux défis de la Géométrie Algorithmique dans les prochaines années. Certaines solutions commencent déjà à apparaître (arithmétique paresseuse [14] [145], calcul exact du signe d'un déterminant [8], prédicats de Shewchuck [183] [184], classe de nombres réels LEDA [41] [42]...).

Juger un algorithme d'après son comportement asymptotique ne reflète pas toujours la valeur pratique d'un algorithme. Certains théoriciens essaient, par exemple, de transformer un log itéré par l'"inverse" de la fonction d'Ackermann[1] dans une complexité, ce qui a un intérêt pratique limité. La notation "O" a un effet pervers dans le sens où elle fait oublier les constantes multiplicatives dans les complexités, constantes qui sont parfois prédominantes en pratique. Dans ce mémoire, on trouve par exemple une fonction de tri (en $O(N \log N)$) qui est plus rapide qu'une fonction de triangulation (en $O(N)$ en moyenne), au moins jusqu'à des données en entrée de taille 10^7!

Revenons au contenu de ce mémoire qui traite principalement des maillages de Delaunay dont la structure duale est le diagramme de Voronoi (le diagramme de Voronoi d'un ensemble de points M représente les classes d'équivalence de la relation "avoir même(s) plus proche(s) voisin(s) dans M"). Rappelons que les diagrammes de Voronoi

[1]Le log itéré est une fonction à croissance très lente ($log^* 2^{65536} = 5$) mais l'"inverse" $\alpha(n)$ de la fonction d'Ackermann croît encore légèrement moins vite ($\alpha(10^{100}) = 4$)!

ne sont pas seulement un objet mathématique à l'état pur et qu'ils se rencontrent très fréquemment dans la nature. Ils permettent aussi bien de modéliser des structures cristallines microscopiques que de comprendre la distribution des galaxies, on peut même les apercevoir sur la carapace d'une tortue ou sur le cou d'une girafe réticulée! La triangulation de Delaunay est utilisée dans cette étude pour modéliser le terrain naturel sous forme d'une surface triangulée. En effet, cette triangulation produit des triangles proches du triangle équilatéral et fournit une bonne approximation du terrain naturel qui est utilisée aussi bien pour les calculs (interpolations, cubatures, intersection avec d'autres surfaces...) que pour la visualisation (images de synthèse...).

Après le rappel de quelques propriétés fondamentales de la triangulation de Delaunay, un rapide état de l'art sur les innombrables algorithmes de triangulation de Delaunay est présenté. Ensuite, la fonction de fusion unidirectionnelle de deux sous-triangulations linéairement séparables ([100]) est généralisée à deux directions. Cette nouvelle fonction est utilisée dans de nouveaux algorithmes de triangulation de Delaunay dans le plan, basés sur des arbres bidimensionnels (2d-tree, random 2d-tree, adaptive 2d-tree, random adaptive 2d-tree, quadtree, bucket-tree...). En ce qui concerne l'étude théorique, un résultat général est établi sur la complexité en moyenne – en termes de sites inachevés – du processus de fusion multidimensionnelle dans l'hypothèse de distribution quasi-uniforme dans un hypercube. Ce résultat général dans un espace de dimension quelconque est ensuite appliqué au cas du plan et il est montré que la construction "Divide-and-Conquer" de la triangulation de Delaunay d'un ensemble de sites du plan quasi-uniformément distribués dans un carré peut être effectuée en temps moyen linéaire, après un tri dans deux directions, qui sert à construire le 2d-tree. L'étude de la complexité dans le pire des cas est beaucoup plus simple. Une comparaison expérimentale de ces nouveaux algorithmes avec les "meilleurs" algorithmes de triangulation de Delaunay (d'après l'étude de Su et Drysdale [187] [188] réalisée en 1995) est effectuée sur divers types de distributions et sur des ensembles atteignant 7 millions de sites. Il apparaît que les algorithmes basés sur des fusions bidirectionnelles l'emportent dans tous les cas. Les tests sont réalisés sur les mêmes ensembles de points, sur la même machine (super-calculateur Convex C3) et la plupart des algorithmes sont codés par l'auteur de ce mémoire, ce qui rend les comparaisons plus fiables. Deux exemples simples de bugs liés à l'imprécision numérique sont exhibés. Ensuite, des algorithmes pratiques de localisation dans une surface triangulée sont présentés dont certains sont originaux : en particulier, on utilise la randomisation à partir d'un arbre binaire de recherche. Ce nouvel algorithme de localisation donne naissance à un algorithme incrémental dynamique de triangulation de Delaunay, probablement parmi les plus performants de ceux connus à ce jour. On peut construire la triangulation *on-line*[2] et de manière très efficace. Des tests de performances sont réalisés pour ces

[2]Un algorithme *en ligne* (on-line) est un algorithme maintenant la solution d'un problème, lors de l'introduction successive de données, sans connaître a priori l'ensemble des données à traiter.

algorithmes de localisation, avec une comparaison avec le célèbre algorithme optimal de Kirkpatrick, et on montre que le nouvel algorithme de localisation présenté dans ce mémoire est plus rapide que celui de Kirkpatrick, au moins jusqu'à douze millions de sites ! Tout au long de ce mémoire, sont aussi évoqués de futurs sujets de recherche, ainsi qu'une réflexion sur les algorithmes dynamiques.

Pendant ces quelques années de thèse au Service d'Etudes Techniques des Routes et Autoroutes (SETRA), j'ai participé à la réalisation d'un logiciel de CAO routière. Il faut pouvoir trianguler rapidement de vastes ensembles de points (plusieurs centaines de milliers). La triangulation obtenue constitue un *Modèle Numérique de Terrain*[3] (MNT). Il est utilisé pour représenter et visualiser (images de synthèse...) le terrain naturel[4], et aussi pour effectuer des calculs : interpolations ([30] page 86), cubatures, intersection avec d'autres surfaces... L'algorithme basé sur le 2d-tree, proposé dans le chapitre 5, répond à ce besoin. On peut aussi remarquer comment un besoin industriel peut provoquer la découverte d'algorithmes (chapitre 5) avec de nouveaux développements théoriques (chapitre 6) et noter que le retour vers le monde industriel est profitable (algorithmes plus performants, voir chapitre 7).

Les contraintes qui servent à représenter des lignes caractéristiques du relief (lignes de faille, falaises, ruptures de pente, crêtes, talwegs...), sont intégrées a posteriori à l'aide de l'algorithme de localisation du 2d-Search-and-Walk (voir paragraphe 9.3.2.5). Ces algorithmes, codés en langage C, ont été intégrés dans la plate-forme de développement orientée objet *Cas.Cade* de *Matra Data Vision*. Mon travail a aussi consisté à concevoir et coder des algorithmes pratiques d'exploitation de surfaces triangulées : projection et intersection d'une polyligne $3d$ avec le terrain naturel, calcul des courbes de niveau, appartenance d'un point à un polygone convexe et quelconque, surface d'un polygone, volume entre deux surfaces triangulées. Un algorithme linéaire en moyenne d'intersection de surfaces[5] triangulées a aussi été conçu, différent de celui proposé dans [53] et [108].

En conclusion, je voudrais souligner l'importance du monde industriel pour la Géométrie Algorithmique : elle a besoin du monde industriel, non seulement pour justifier son existence, mais aussi pour motiver de nouvelles recherches. La collaboration entre ces deux mondes, qui est souvent trop rare, peut se révéler très fructueuse, aussi bien pour l'industrie que pour l'évolution des connaissances théoriques.

[3]Un Modèle Numérique de Terrain [43] est une nappe surfacique déformée en fonction de l'altitude. On dit parfois que c'est un modèle $2,5D$. Ce n'est pas un véritable $3D$ dans la mesure où seule la hauteur maximale de l'objet est prise en compte. Il n'est pas possible de représenter plusieurs z pour un même (x, y), ainsi ni l'épaisseur d'un objet, ni, a fortiori, sa forme exacte ne sont représentées.

[4]"The problem in modelling geological systems therefore moves out of the realm of exact definition based on observed data found in general surveying to that of surface interpolation based on islands of knowledge surrounded by oceans of ignorance" (M. J. McCullagh [59]).

[5]L'intersection de surfaces paramétriques est davantage traitée dans la littérature [10] [122].

Résumé des contributions personnelles

- Nouveaux algorithmes de triangulation de Delaunay dans le plan basés sur des arbres bidimensionnels (2d-tree, random 2d-tree, adaptive 2d-tree, random adaptive 2d-tree, quadtree, bucket-tree...).
- Création d'une fonction générale permettant la fusion bidirectionnelle de deux sous-triangulations de Delaunay linéairement séparables.
- Analyse de la complexité dans le pire des cas et en moyenne de ces algorithmes : en particulier, on montre que la phase de triangulation est en moyenne linéaire.
- Etude de la triangulation de Delaunay en dimension quelconque, en termes de sites inachevés : probabilité qu'un site soit inachevé, espérance du nombre de sites inachevés dans un hyper-rectangle.
- Application à l'analyse d'une classe d'algorithmes "Divide-and-Conquer" en dimension quelconque.
- Comparaison expérimentale d'un grand nombre d'algorithmes de triangulation de Delaunay, dans le but de déterminer à ce jour, le ou les plus performants en fonction des types de distribution.
- Deux exemples simples de "bugs" liés à l'imprécision numérique.
- Algorithmes pratiques de localisation dans une triangulation de Delaunay, basés sur la randomisation à partir d'un arbre binaire de recherche.
- Comparaison expérimentale d'un grand nombre d'algorithmes pratiques de localisation.
- Algorithme incrémental *on-line* (dynamique) de triangulation de Delaunay, utilisant les algorithmes précédents.

Chapitre 1

TRIANGULATION DE DELAUNAY : définition, propriétés, état de l'art

But a wise ordinance of Nature has decreed that, in proportion as the working classes increase in intelligence, knowledge, and all virtue, in that same proportion their acute angle shall increase also and approximate to the comparatively harmless angle of the Equilateral Triangle (E. Abbott, Flatland *[1]).*

Le but de ce chapitre n'est pas d'être exhaustif (un livre entier n'y suffirait pas), mais de donner un aperçu de l'essentiel.

1.1 Différentes triangulations

Les triangulations sont des objets géométriques très fréquemment utilisés. En effet, manipuler uniquement des points n'est pas satisfaisant. Les triangulations, en reliant les points entre eux, créent une partition de l'espace ou d'un domaine. Les liaisons géométriques et topologiques ainsi définies permettent de localiser rapidement un objet, de décomposer l'espace de travail d'un robot, de reconstruire des objets tridimensionnels à partir de coupes, de visualiser une surface avec rendu d'image, d'effectuer des calculs de volume, d'effectuer des opérations booléennes sur les surfaces ... Elles sont aussi parfois un prérequis nécessaire pour résoudre d'autres problèmes de géométrie algorithmique.

Il existe deux grandes classes : les triangulations géométriques (qui ne dépendent que des coordonnées des points) et les triangulations dépendantes des données où on

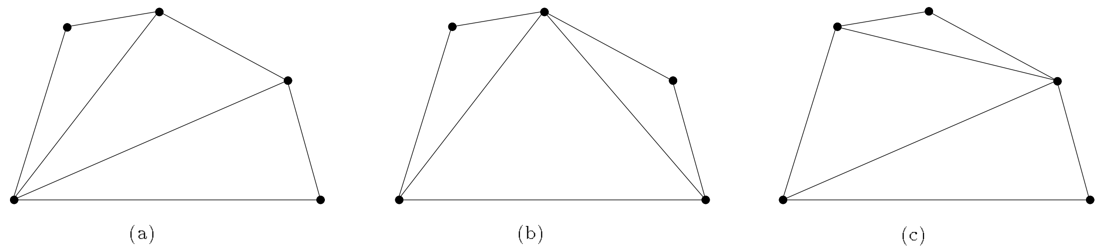


Figure 1.1: *Différentes triangulations [159] : (a) triangulation de Delaunay, (b) triangulation de poids minimal, (c) triangulation gloutonne.*

peut associer un coût aux sommets ou aux arêtes (DDT : Data Dependant Triangulations [169]). Il faut remarquer que, dans le second cas, les triangulations concernent surtout des surfaces du type $z = f(x, y)$. Dans ce mémoire, seules les triangulations géométriques vont nous intéresser.

Il existe un très grand nombre de façons pour trianguler un ensemble de points. En fonction du domaine d'application, la triangulation devra respecter certains critères, ce qui limitera le nombre des possibilités (voir par exemple [60]).

Pour des problèmes de communication, on cherchera à minimiser la longueur totale des côtés des triangles ou, plus généralement, la somme des poids associés aux trois côtés des triangles : c'est la *triangulation de poids minimal*. Elle a l'avantage d'être unique (on dit que la triangulation est *systématique*) pour des points en position générale, mais reste difficile à calculer du point de vue algorithmique. La complexité de ce problême n'est pas connue et on se demande s'il n'est pas NP-dur (il existe toutefois des heuristiques [150]). La *triangulation gloutonne* (triangulation obtenue en ajoutant une à une les arêtes et en choisissant toujours la plus courte qui ne croise aucune de celles retenues précédemment) n'est pas toujours égale à la triangulation de poids minimum (voir Figure 1.1). Elle a aussi l'avantage d'être unique pour des points en position générale, mais par contre peut se calculer en $O(n \log n)$ [194]. Pour des problèmes d'approximation de surfaces (avec interpolations) ou de calculs par éléments finis, on préfèrera avoir des triangles les plus équilatéraux possibles ce qui justifie le choix de la *triangulation de Delaunay* pour ces applications[1]. Cette dernière maximise l'angle minimal parmi tous les triangles. Elle a aussi l'énorme avantage d'être unique (pour des points en position générale) et peut se calculer efficacement (en $\Theta(n \log n)$ dans le plan). Il existe aussi une généralisation de la triangulation de Delaunay par T. Lambert [125], *the empty-shape triangulation* : la "figure" circonscrite à chaque triangle

[1] On peut trouver dans l'ouvrage de P.L. George et H. Borouchaki [36] un grand nombre d'applications de la triangulation de Delaunay (éléments finis, adaptation, $3D$, maillage de frontières. . .).

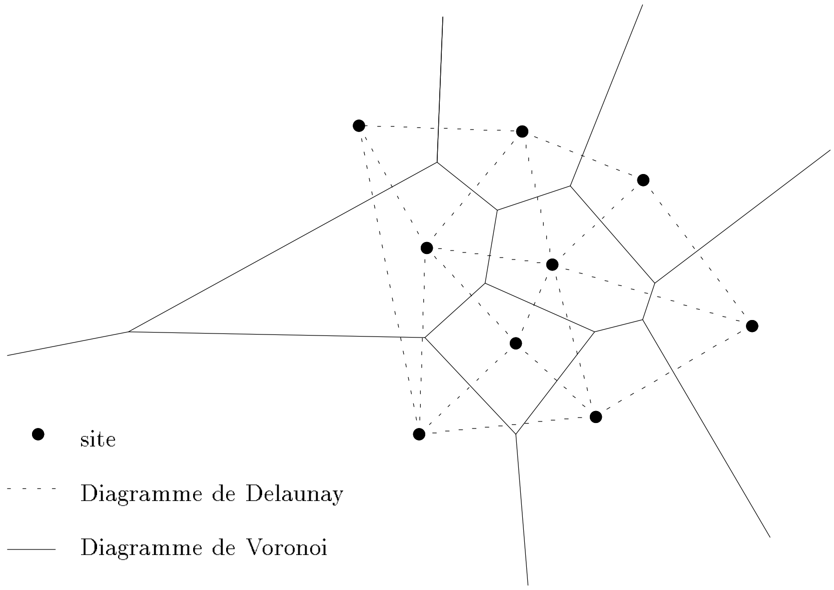


Figure 1.2: *Diagramme de Voronoi de sites du plan.*

ne doit contenir aucun site en son intérieur. Dans le cas de Delaunay, cette "figure" est un cercle. On peut aussi imaginer d'optimiser d'autres critères comme : minimiser l'aire maximum des triangles, maximiser un critère de régularité quelconque (rapport Surface/Périmètre de tous les triangles par exemple...), limiter le nombre d'arêtes autour des sommets[2] (à au plus 6 par exemple),...

1.2 Définition

1.2.1 Diagramme de Voronoi

Supposons que S soit un ensemble de n points $\{s_1, s_2, \dots s_n\}$ (appelés *sites*) de l'espace euclidien $\mathbb{E}^k$.

On associe à chaque site s_i la région $V(s_i)$ constituée des points les plus proches de s_i que de n'importe quel autre site :

$V(s_i) = \{M/d(M, s_i) \leq (d(M, s_j), \forall j \neq i\}$, d désignant la distance euclidienne.

On dit que $V(s_i)$ est la *région de Voronoi de centre* s_i.

La région $V(s_i)$ est l'intersection d'un nombre fini de demi-espaces limités par les

[2] La limitation de ce nombre d'arêtes à trois produit la triangulation ternaire [84]. Elle permet de construire trivialement une hiérarchie de triangles mais a l'inconvénient de générer beaucoup de triangles très aplatis.

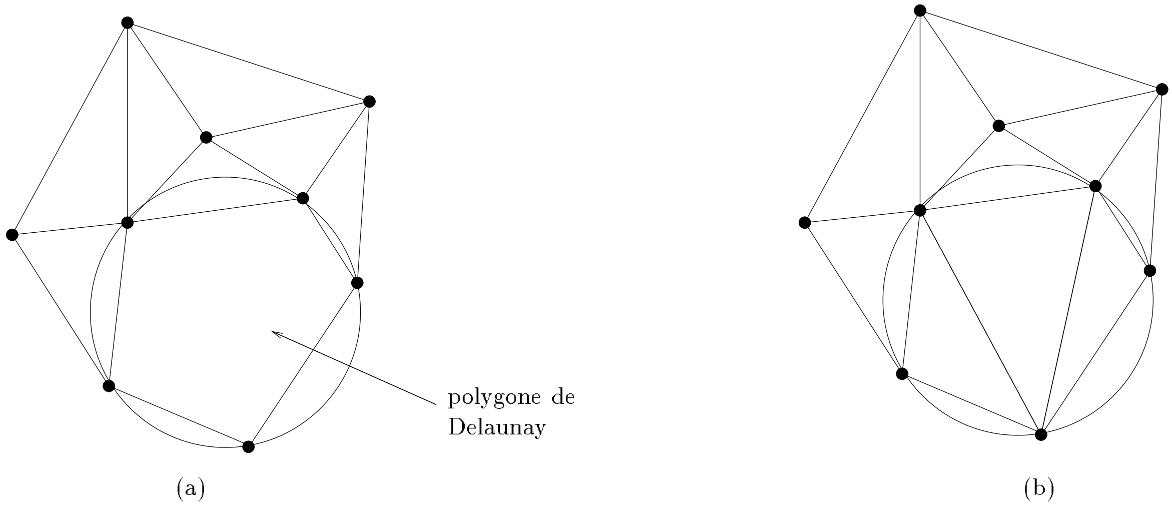


Figure 1.3: *Diagramme et triangulation de Delaunay : (a) le diagramme de Delaunay, (b) une triangulation de Delaunay.*

hyperplans médiateurs des segments $[s_i s_j]$, $j = 1, \cdots n$, $j \neq i$. C'est donc un polytope convexe, éventuellement non borné. Les $V(s_i)$ constituent le *diagramme de Voronoi* (voir Figure 1.2). Elles constituent une partition de $\mathbb{E}^k$.

1.2.2 Diagramme et triangulation de Delaunay

Le *diagramme de Delaunay* (ou *complexe de Delaunay* [34]) est le dual du diagramme de Voronoi. Dans le plan, deux sites s_i et s_j sont reliés par une arête si et seulement si les régions de Voronoi $V(s_i)$ et $V(s_j)$ qui leur sont associées, ont une arête en commun. En dimension k, les sites s_{i_0}, s_{i_1},..., s_{i_p} définissent une p-face du diagramme de Delaunay si et seulement si les régions de Voronoi $V(s_{i_0})$, $V(s_{i_1})$,..., $V(s_{i_p})$ qui leur sont associées, partagent une $(k-p)$-face dans le diagramme de Voronoi. On peut donc aisément construire l'un à partir de l'autre.

S'il n'existe aucun sous-ensemble de $l > (k+1)$ sites cosphériques tels que leur sphère circonscrite soit vide de sites (on dit que les points sont en *position L_2-générale*), alors le diagramme de Delaunay est composé de simplexes[3] qui constituent une partition de l'enveloppe convexe des sites. On l'appelle la *triangulation de Delaunay*. Dans le cas contraire, il suffit de trianguler les faces qui ne sont pas simpliciales. On obtient alors *une* triangulation de Delaunay. Elle n'est pas unique, car il y a plusieurs façons de trianguler les faces non simpliciales (voir Figure 1.3).

[3] L'enveloppe convexe de $(k+1)$ points indépendants – c'est à dire qui engendrent un espace affine de dimension k – est appelée simplexe ou k-simplexe.

Le graphe de Delaunay forme une partition de l'enveloppe convexe de S. Ce graphe est appelé *triangulation de Delaunay* si et seulement si chaque face de T est triangulaire. C'est le cas si l'on ne peut pas trouver 4 points cocycliques de S qui ne contiennent pas d'autre site dans leur cercle circonscrit. La triangulation obtenue est alors unique.

Remarque : le diagramme de Voronoi d'ordre h d'un ensemble de points S est une partition de l'espace telle que chaque région est l'ensemble des points qui sont strictement plus proches de chacun des sites d'une partie T de h sites de S que de tout autre site de $S \setminus T$. Son dual orthogonal s'appelle la triangulation de Delaunay d'ordre h (voir la thèse de D. Schmitt [173]). Par la suite, on considère que $h = 1$.

1.3 Quelques propriétés

1.3.1 Dans le plan

On peut trouver la démonstration de ces propriétés dans le livre d'Okabe, Boots et Sugihara [159], par exemple.

1.3.1.1 Propriétés fondamentales

- Le cercle circonscrit à n'importe quel triangle de Delaunay ne contient aucun site en son intérieur strict. C'est la propriété du *cercle vide* ou du *cercle circonscrit.* Réciproquement, si une triangulation vérifie ce critère, elle est de Delaunay.

- On appelle *finesse* [34] d'une triangulation $T(S)$ le vecteur $\mathcal{F}(T(S)) = (\alpha_1, \alpha_2, \ldots, \alpha_{3t})$ où les α_i sont les angles internes des t triangles, classés par ordre croissant. On définit sur ces vecteurs la relation d'ordre lexicographique $<_l$ par :

$$\mathcal{F}(T(S)) = (\alpha_1, \alpha_2, \ldots, \alpha_{3t}) \ <_l \ \mathcal{F}(T'(S)) = (\beta_1, \beta_2, \ldots, \beta_{3t})$$

$$\Longleftrightarrow \exists j \setminus \forall i < j \ \{ \alpha_i = \beta_i \text{ et } \alpha_j < \beta_j \}$$

  La triangulation qui maximise la finesse selon l'ordre lexicographique est une triangulation de Delaunay. En particulier, la triangulation de Delaunay maximise l'angle minimal parmi tous les triangles.

- On déduit de la propriété précédente la *règle de régularisation locale* ou *critère d'équiangularité locale* ou *critère de l'angle Min–Max* : si la diagonale de n'importe quel quadrilatère strictement convexe formé par l'union de deux triangles de Delaunay est remplacée par la diagonale opposée, la valeur de l'angle minimal parmi les six angles internes n'augmente pas. Réciproquement, si ce critère d'équiangularité locale est partout vérifié, alors la triangulation est de Delaunay.

- La triangulation de Delaunay d'un ensemble S de sites du plan (xOy) est la projection (suivant (Oz)) sur ce plan de l'enveloppe convexe inférieure des projections (suivant (Oz)) des sites de S sur le paraboloïde de révolution d'axe (Oz).

- Soit p un point quelconque du plan. On dit que $T_i \leq_p T_j$ s'il existe un rayon issu de p qui traverse les triangles T_i et T_j dans cet ordre. Si cette relation est une relation d'ordre, on dit que la triangulation est *monotone pour le lancer de rayons*. La triangulation de Delaunay a cette propriété (ce n'est pas le cas de toutes les triangulations). On peut en déduire un ordre partiel des triangles par rapport à un sommet [83].

- La longueur de la plus petite chaîne (composée d'arêtes de Delaunay) reliant deux sites A et B ne peut excéder la distance euclidienne entre les deux points multipliée par $\frac{2\pi}{3\cos(\pi/6)} \approx 2.42$ (on sait qu'elle peut approcher $\frac{\pi}{2}$).

1.3.1.2 Autres propriétés

- Les arêtes de l'enveloppe convexe sont des arêtes de Delaunay.

- Un point et son plus proche voisin définissent toujours une arête de Delaunay.

- Comme dans n'importe quelle triangulation de n sites, le nombre de triangles est $t = 2n - 2 - n_{ec}$, et le nombre d'arêtes $e = 3n - 3 - n_{ec}$, n_{ec} étant le nombre de points (ou d'arêtes) de l'enveloppe convexe. La relation d'Euler $t - e + n = 1$ est vérifiée.

- On a vu que la triangulation de Delaunay maximise le plus petit angle. Mais, elle optimise d'autres paramètres tels que :

  - elle minimise le plus grand cercle circonscrit.

  - elle minimise le plus grand cercle englobant (on dit aussi qu'elle a la *granularité* la plus fine). On appelle cercle englobant d'un triangle le plus petit cercle contenant ce triangle. Ce cercle est égal au cercle circonscrit si et seulement si le triangle ne contient pas d'angle obtus.

1.3.1.3 Relation avec d'autres graphes

- Le *graphe de l'enveloppe convexe* est un sous-graphe de la triangulation de Delaunay.

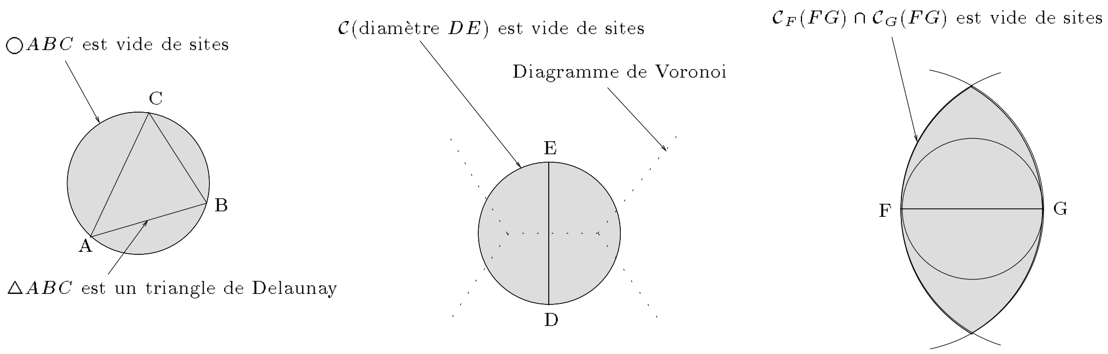


Figure 1.4: *Autres graphes : AB, AC et BC arêtes de Delaunay, DE arête de Gabriel, FG arête du graphe de voisinage relatif.*

- Le *graphe de Gabriel* est un sous-graphe de la triangulation de Delaunay. Le graphe de Gabriel d'un ensemble S de sites $s_1, s_2, \ldots, s_n$ est composé de toutes les arêtes $s_i s_j$ telles que les cercles ayant ces arêtes pour diamètre soient vides de sites. $s_i s_j$ est une arête du graphe de Gabriel de S si et seulement si $s_i s_j$ est une arête de Delaunay de S qui coupe l'arête de Voronoi correspondante.

- Le *graphe de voisinage relatif* est un sous-graphe du graphe de Gabriel, donc du graphe de Delaunay. Le graphe de voisinage relatif est composé des arêtes $s_i s_j$ qui vérifient : $d(s_i, s_j) \leq \min\limits_{\substack{s \in S \\ s \neq s_i, s_j}} \max(d(s_i, s), d(s_j, s))$

- L'*arbre couvrant minimal euclidien* est un sous-graphe du graphe de voisinage relatif. L'arbre couvrant minimal euclidien d'un ensemble S de points est un arbre dont les noeuds sont exactement les points de S et tel que la somme des longueurs euclidiennes des arêtes de l'arbre soit minimale.

1.3.2 En dimension quelconque

Certaines propriétés existantes dans le plan, se généralisent à une dimension quelconque. On peut trouver les démonstrations dans le livre de Boissonnat et Yvinec [34].

- La boule circonscrite à n'importe quel simplexe de Delaunay ne contient aucun site en son intérieur strict. C'est la propriété de la boule vide ou de la sphère circonscrite. Réciproquement, si une triangulation vérifie ce critère, elle est de Delaunay.

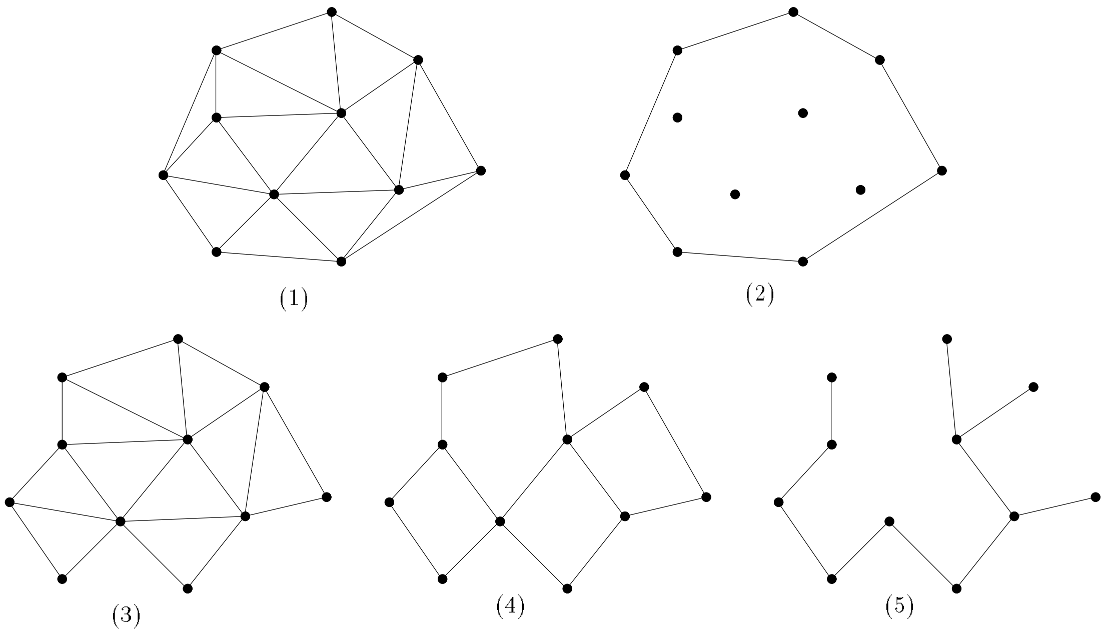


Figure 1.5: *Sous-graphes [159] de la triangulation de Delaunay plane : (1) triangulation de Delaunay, (2) enveloppe convexe, (3) graphe de Gabriel, (4) graphe de voisinage relatif, (5) arbre couvrant minimal euclidien.*

- Soit une triangulation $T(S)$ d'un ensemble S de sites de $\mathbb{E}^k$. On dit que la paire de k-simplexes adjacents (c.a.d. ayant une $(k-1)$-face commune) (X_1, X_2) est *régulière* si et seulement si le site de X_1 qui n'appartient pas à leur face commune, n'appartient pas à l'intérieur de la sphère circonscrite à X_2. Toutes les paires de k-simplexes adjacents de $T(S)$ sont régulières si et seulement si $T(S)$ est une triangulation de Delaunay de S.

- La triangulation de Delaunay d'un ensemble S de sites d'un hyperplan $\mathcal{H}$ est la projection (suivant $\mathcal{H}^\perp$) sur cet hyperplan de l'enveloppe convexe inférieure des projections (suivant $\mathcal{H}^\perp$) des sites de S sur le paraboloïde de révolution d'axe $\mathcal{H}^\perp$.

- On associe à chaque simplexe X la plus petite sphère qui le contienne. Soit r_X son rayon. Le *grain* d'une triangulation est $\mathcal{G}(T(S)) = \max_{X \in T(S)} r_X$. Les triangulations de Delaunay de S sont celles qui ont le grain le plus fin.

- Les faces de l'enveloppe convexe sont des faces de la triangulation de Delaunay.

- Un point et son plus proche voisin définissent toujours une arête de Delaunay.

- La taille (c'est à dire le nombre de faces) d'une triangulation de Delaunay en dimension k est bornée par $\theta(n^{\lceil k/2 \rceil})$.

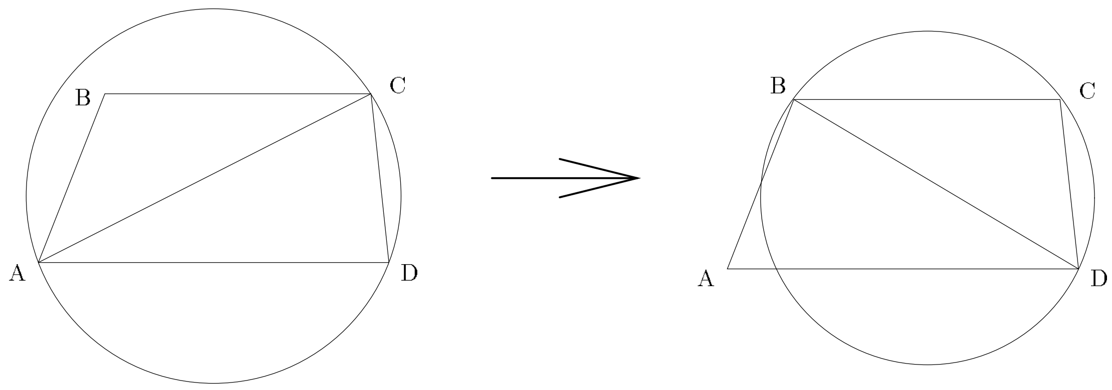


Figure 1.6: *Algorithmes à base d'échanges d'arêtes, swap (flip) de la diagonale du quadrilatère : critère d'équiangularité local ou règle de régularisation locale ou règle du Min-Max .*

1.4 Etat de l'art des algorithmes

Nous ne cherchons pas à être exhaustifs, on peut trouver dans la thèse de T. Lambert [125] 125 références bibliographiques sur les algorithmes de triangulation de Delaunay. Nous conseillons cette lecture à ceux qui veulent en savoir plus sur ces algorithmes. Nous allons simplement décrire les principaux paradigmes algorithmiques[4] utilisés avec quelques exemples. Curieusement, ce sont presque les mêmes paradigmes qui ont permis à Knuth [121] en 1973 de classer les algorithmes de tri. On peut noter qu'il existe une forte analogie entre le tri et la triangulation de Delaunay : trianguler dans un espace de dimension 1 consiste à trier les points, puis à les relier selon l'ordre fourni par le tri.

1.4.1 Algorithmes à base d'échanges d'arêtes

Son homologue parmi les algorithmes de tri est le tri à bulles. Historiquement, c'est l'un des premiers algorithmes de triangulation de Delaunay [127]. Il faut d'abord construire une triangulation quelconque (par exemple, une triangulation en étoile en reliant un point donné à tous les autres, ou une triangulation "lexicographique", ou celle du paragraphe 7.2...). Ensuite, on considère chaque arête en tant que diagonale d'un quadrilatère et, à l'aide du critère du cercle vide par exemple, on teste si l'échange

[4]On peut aussi classer les algorithmes de triangulation en algorithmes statiques, semi-dynamiques ou dynamiques [82] [40]. Nous n'avons pas utilisé cette taxinomie car la frontière entre ces classes n'est pas toujours nette (certains algorithmes statiques peuvent être utilisés dynamiquement, mais avec de moins bonnes performances). De plus, à l'intérieur de chaque catégorie, on répète souvent les mêmes paradigmes.

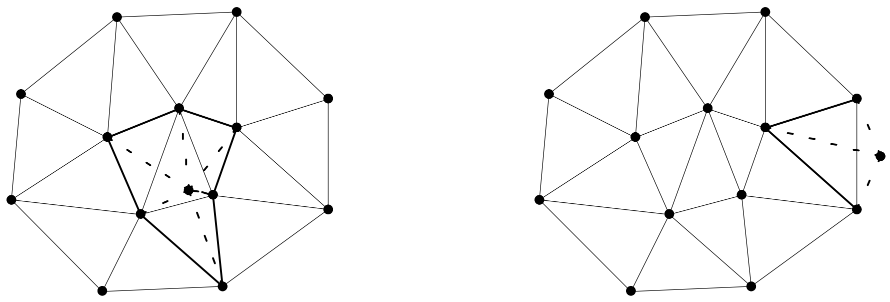


Figure 1.7: *L'algorithme incrémental de Watson.*

("swap") de la diagonale est nécessaire. Si une arête est modifiée, il faudra ensuite retester les quatre arêtes frontières du quadrilatère qui la contient. Lawson [128] a prouvé que cette méthode converge toujours vers la triangulation de Delaunay. C'est un exemple très rare où un critère local d'optimisation aboutit à l'optimalité globale. Quand une arête a été détruite, elle ne peut plus être créée. Comme il y a $\binom{n}{2}$ arêtes possibles, cet algorithme est en $O(n^2)$. Mais en pratique, il n'y a pas tant d'échanges et leur nombre est généralement linéaire comme le confirment les résultats expérimentaux du chapitre 7. C'est actuellement un problème ouvert ([46] page 20) de prouver que, pour la plupart des triangulations, le nombre d'échanges nécessaires pour obtenir une triangulation de Delaunay, est linéaire. Néanmoins, quelques résultats théoriques ([104], [109], [110] et [111]) sur les échanges d'arêtes dans une triangulation existent.

1.4.2 Algorithmes incrémentaux

Son homologue parmi les algorithmes de tri est le tri par insertion. C'est une méthode presque aussi ancienne que la précédente et qui a donné naissance à un nombre impressionnant de publications. Le principe est d'ajouter les points un par un et, à chaque fois, de mettre à jour la triangulation de Delaunay. Lawson [128] ajoute un point M en le reliant à tous les points visibles de la triangulation déjà construite (3 points si M est à l'intérieur de la triangulation existante, un nombre quelconque de points de l'enveloppe convexe si M est à l'extérieur de la triangulation existante). Une fois que M est inséré, on construit la triangulation de Delaunay en appliquant la méthode des échanges d'arêtes. Une stratégie consiste à choisir un ordre d'insertion pour que les points insérés soient toujours en dehors de la triangulation déjà construite (pour cela, trier les points d'après leur distance à l'origine ou suivant les x croissants) ; la stratégie inverse consiste à construire un triangle factice englobant tout le semis pour que les points insérés soient toujours dans la triangulation déjà construite, on détruit ensuite les arêtes inutiles. Dans le pire des cas (points répartis sur un arc de parabole par exemple), tous les triangles déjà construits peuvent être modifiés lors d'une insertion ; on aura alors $\Omega(n^2)$ échanges. La complexité dans le pire des cas est donc en $\Theta(n^2)$.

Watson [195], après l'insertion d'un point M, détruit tous les triangles en conflit avec M (ce sont les triangles dont le cercle circonscrit contient M). Il obtient ainsi un polygone étoilé avec M à l'intérieur. Il suffit ensuite de relier M à tous les sommets de ce polygone étoilé. Ainsi, la triangulation de Delaunay est mise à jour (voir Figure 1.7).

La faiblesse des algorithmes incrémentaux se situe en général dans l'étape de localisation qui doit permettre de trouver le triangle contenant le point à insérer. Dans le chapitre 9, nous proposons une nouvelle méthode pratique de localisation qui donne naissance à un nouvel algorithme incrémental de triangulation de Delaunay. Il a été testé expérimentalement dans le paragraphe 9.4.

Boissonnat et Teillaud [32] ont imaginé, pour la localisation, une structure hiérarchique qui permet la construction incrémentale de la triangulation de Delaunay. Toutefois, cette construction n'est pas dynamique car elle est basée sur le Graphe de Conflits [55], structure qui nécessite la connaissance de la totalité des données dès l'initialisation. Ils ont ensuite amélioré [33] [192], avec une approche randomisée, cette structure – l'Arbre de Delaunay – en la rendant semi-dynamique. Les insertions sont autorisées, sans connaissance préliminaire de l'ensemble des données. On garde dans une arborescence – l'arbre de Delaunay – l'historique de la construction. Les feuilles contiennent la triangulation courante. Localiser un point M consiste à le localiser dans toutes les triangulations ayant existé successivement. Il a été prouvé que le coût moyen de la localisation est en $O(\log n)$ et que l'algorithme a un coût total moyen en $O(n \log n)$. L'algorithme de Boissonnat et Teillaud a été testé expérimentalement dans le chapitre 7, ainsi que dans le paragraphe 9.4.

L'Arbre de Delaunay a ensuite donné naissance à une structure plus générale, le Graphe d'Influence, qui permet de construire de façon semi-dynamique de nombreuses structures géométriques [31]. Devillers, Meiser et Teillaud [62] ont ensuite rendu l'arbre de Delaunay complèment dynamique en autorisant les insertions en temps moyen $O(\log n)$ et aussi les suppressions en temps moyen $O(\log \log n)$ dans le plan. Guibas, Knuth et Sharir [102] emploient une méthode semblable, mais la mise à jour de la triangulation après insertion d'un point est faite avec des swaps et tous les triangles, pour chaque swap, sont gardés en mémoire. Ils ont prouvé que le nombre moyen de swaps est linéaire, quelle que soit la distribution des points, et que leur algorithme a aussi un coût total moyen en $O(n \log n)$. Toutefois, il consomme un peu plus de mémoire que l'algorithme de Boissonnat et Teillaud, car il conserve en mémoire des triangles temporaires créés par les swaps et qui ne font à aucun moment partie d'une triangulation.

Mulmuley [155] utilise une approche différente en évitant le stockage de l'histoire de la construction. Au niveau inférieur de l'arborescence, tous les sites sont présents. On détermine les sites présents à un étage de l'arborescence en tirant à pile ou face pour chacun des sites présents à l'étage juste en-dessous. Pour chaque niveau de l'arborescence, on calcule la triangulation de Delaunay des sites présents à cet étage. La mise à jour des conflits consiste à créer des liens entre les triangles de deux niveaux consécutifs si ceux-ci ont une intersection non vide. Pour insérer un site, on lance la pièce et le site est inséré à chaque étage, jusquà ce que la pièce donne une réponse négative. Pour chacun de ces étages, la triangulation doit être mise à jour ainsi que le graphe des conflits. Pour supprimer un site, on procède à l'inverse d'une insertion. Les complexités moyennes sont $O(\log n)$ pour une insertion, $O(1)$ pour une suppression et $O(\log^2 n)$ pour une localisation.

1.4.3 Algorithmes de sélection

Son homologue parmi les algorithmes de tri est le tri par sélection. La méthode naïve ([160] page 202) consiste à examiner tous les triplets de points (il y en a $\binom{n}{3}$) et à tester pour chacun, en temps linéaire, s'il vérifie le critère du cercle vide. La complexité totale est en $O(n^4)$. Bien que cet algorithme soit extrêmement concis (20 lignes de code), son temps d'exécution devient insupportable à partir d'une cinquantaine de points.

Un des premiers algorithmes de cette catégorie est celui de Mac Lain [140]. Le principe est de construire les triangles de Delaunay un par un et de façon définitive (ils ne seront pas modifiés par la suite du déroulement de l'algorithme). Chaque nouveau triangle est construit à partir d'une arête d'un triangle trouvé précédemment, en cherchant le point qui, joint aux sommets de l'arête, constituera un triangle de Delaunay (cercle circonscrit au triangle vide). On initialise ce processus en partant d'un point quelconque et de son plus proche voisin, qui sont les extrémités d'une première arête de Delaunay. Les nouvelles arêtes créées sont placées dans une liste et serviront à construire de nouveaux triangles de Delaunay. Quand cette liste sera vide, la triangulation sera terminée. Pour chaque arête, la recherche du sommet de Delaunay associé à cette arête peut prendre un temps $O(n)$ d'où la complexité quadratique (dans le pire des cas) de ce type d'algorithme.

Le point délicat, dans ce type d'algorithme, est la localisation du sommet de Delaunay associé à une arête. Suivant la méthode de localisation employée, les algorithmes de type sélection auront des performances très disparates.

On peut accélérer la recherche de ce sommet en utilisant un tri selon les abscisses pour obtenir une complexité totale en $O(n^{3/2})$ *en moyenne*, la complexité dans le pire des cas restant quadratique. J. Hervé [106] propose d'utiliser un boxsort [107] (c'est une généralisation du kd-tree) pour localiser ce sommet; le boxsort a l'avantage de s'adapter assez bien aux distributions non uniformes ; il obtient une complexité totale en $O(n \log n)$ *en moyenne*. A. Maus [142] utilise une grille régulière pour localiser ce sommet. C'est une structure qui n'est performante que pour des distributions quasi-uniformes. Dans ces conditions, il obtient une complexité totale linéaire *en moyenne*. Fang et Piegl [78] utilisent aussi cette structure. R. Dwyer [71] généralise cet algorithme à une dimension quelconque. Il utilise une grille régulière pour la localisation et démontre que, si les points sont uniformément distribués dans une boule, alors la complexité totale de l'algorithme est linéaire *en moyenne* et ceci quelle que soit la dimension de l'espace.

1.4.4 Algorithmes de transformation géométrique

Ils transforment un problème en un autre problème. Ici, on va transformer le problème de triangulation de Delaunay dans un espace de dimension n en un calcul d'enveloppe convexe dans un espace de dimension $n + 1$. Son équivalent pour le tri consiste à projeter les points à trier sur une parabole, à calculer leur enveloppe convexe. Le parcours de l'enveloppe convexe fournit la liste triée de ces points.

La méthode pour calculer la triangulation de Delaunay est très proche : on projette ("lifting" function) les points à trianguler sur un paraboloïde de révolution d'axe orthogonal au plan du semis. On calcule l'enveloppe convexe inférieure des points de projection. La projection de cette enveloppe sur le plan initial est égale au diagramme de Delaunay cherché (voir paragraphe 1.3.2). On peut consulter la thèse de F. Nielsen [157] pour obtenir des références sur les algorithmes de calcul d'enveloppe convexe. Barber, Dobkin et Huhdanpaa [9] donnent une implémentation robuste, au moins jusqu'à la dimension 6.

1.4.5 Algorithmes de balayage

C'est un paradigme très important en géométrie algorithmique, qui est utilisé pour résoudre de très nombreux problèmes (intersection de segments...). Son équivalent pour le tri est le heapsort, qui utilise une file de priorité pour donner en temps logarithmique l'élément cherché. C'est un algorithme de sélection particulier, mais il est suffisamment important pour en faire une catégorie à part.

Fortune [87] [88] a été le premier à utiliser cette méthode. Elle n'est pas triviale (voir Figure 1.8) car les zones d'influence de chaque triangle de Delaunay (cercle circonscrit vide) peuvent s'étendre dans n'importe quelle direction d'où une incompatibilité apparente avec le principe du balayage. Il y a deux ensembles d'états :

la liste des états qui contient la frontière de la partie de la triangulation déjà construite (partie de l'enveloppe convexe visible depuis la droite de balayage), et une

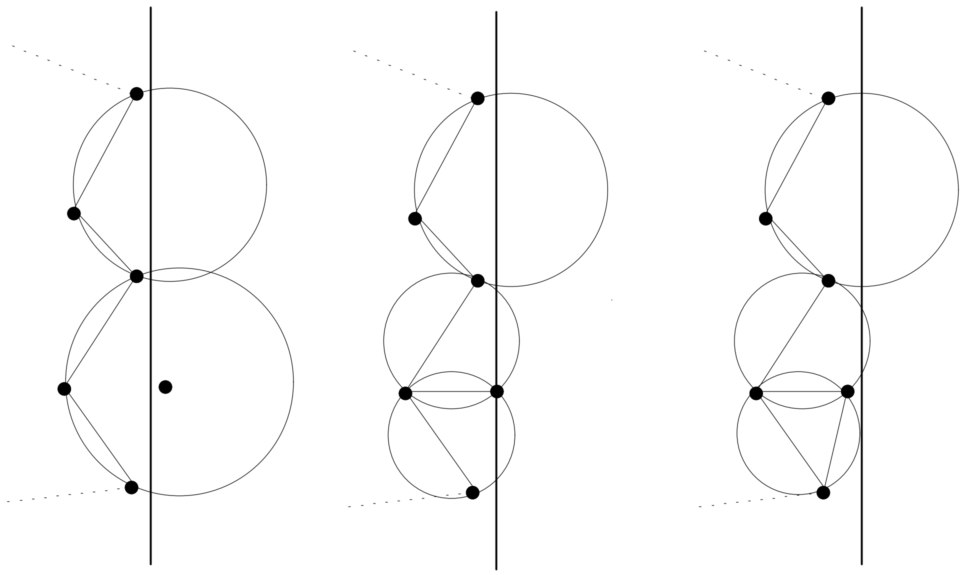


Figure 1.8: *L'algorithme de balayage de Fortune.*

liste de triplets (trois points consécutifs de la frontière) dont les cercles circonscrits sont vides de sites et coupent la droite de balayage. Ces triplets sont donc des sommets potentiels de triangles de Delaunay.

la file de priorité des événements qui contient deux types d'événements :

- l'événement *point* : la droite de balayage rencontre un point du semis. On joint ce point à son plus proche voisin et on met à jour la liste des cercles circonscrits vides.
- l'événement *cercle* : la droite de balayage atteint l'extrémité (relativement au sens du balayage) d'un cercle circonscrit (défini par trois points consécutifs de la frontière) vide de sites. Alors, on peut créer le triangle de Delaunay avec le triplet définissant ce cercle et on met à jour la liste des cercles circonscrits vides.

Cet algorithme a une complexité optimale en $O(n \log n)$, à condition d'utiliser de bonnes structures de données. L'interprétation géométrique a d'abord été suggérée par Edelsbrünner [87] en plongeant le plan contenant les points à trianguler dans un espace à trois dimensions. On construit des cônes dont les sommets sont les sites à trianguler. Les médiatrices des segments joignant deux sites deviennent, sur les surfaces des cônes, des portions d'hyperboles. La droite de balayage devient un plan oblique (voir [87] et [101] pour plus de détails). Seidel [175] propose une variante qui utilise des paraboles ayant pour foyers les sites à trianguler. Spehner et Kauffmann [116] proposent une amélioration en réduisant le nombre d'événements. Don Hatch a réalisé une implémentation très performante de l'algorithme de Seidel en accélérant la gestion des événements avec des techniques de hachage. Cette version sera comparée expérimentalement avec d'autres algorithmes de triangulation de Delaunay dans le chapitre 7.

1.4.6 Algorithmes Divide and Conquer

Son homologue parmi les algorithmes de tri est le tri par fusion (merge sort). Le principe général (voir chapitre 4) est de diviser récursivement l'ensemble des sites que l'on a préalablement triés selon l'ordre lexicographique (selon les abscisses croissantes, et, en cas d'égalité, selon les ordonnées croissantes). On obtient des bandes élémentaires contenant un nombre très faible de points (en géneral de 1 à 3) que l'on triangule trivialement. Ensuite, on fusionne par paires (voir chapitre 3). Lee et Schachter [131] ont montré comment fusionner deux triangulations séparables par une droite en temps proportionnel (dans le pire des cas) au nombre de sites contenus dans les deux sous-triangulations (Kirkpatrick [118] peut fusionner en temps linéaire deux triangulations de Delaunay quelconques mais c'est plus compliqué). Le premier algorithme est dû à Lee et Schachter [131] (voir Figure 1.9). Guibas et Stolfi l'ont ensuite étoffé [100]. G. Leach [129] en donne une implémentation plus performante, en pratique.

Cet algorithme est optimal et a une complexité dans le pire des cas en $\Theta(n \log n)$. Cependant, on peut améliorer la complexité en moyenne de cet algorithme en utilisant une grille régulière. La construction de cette grille se fait en temps linéaire. Ses cellules sont triangulées avec l'algorithme optimal de Lee et Schachter. Dwyer [70] triangule indépendamment chaque colonne de la grille en fusionnant par paires les cellules, puis fusionne par paires les colonnes jusqu'à l'obtention de la triangulation complète. Dwyer démontre que la complexité en moyenne de cet algorithme est en $O(n \log\log n)$ pour une distribution quasi-uniforme, la complexité dans le pire des cas restant optimale. Katajainen et Koppinen [117] fusionnent les cellules de la grille en utilisant l'ordre induit par un quadtree. Pour fusionner quatre cellules, on fusionne deux paires selon la direction horizontale, puis les deux paires résultantes selon la direction verticale. Ils démontrent que la complexité en moyenne de l'algorithme est linéaire (pour une distribution quasi-uniforme), la complexité dans le pire des cas restant optimale. Bien sûr, l'utilisation de techniques de bucketing rend ces méthodes peu adaptées aux distributions non uniformes. Lemaire et Moreau [133] [134] utilisent un 2d-tree pour découper le plan, puis réalisent des fusions alternativement selon les directions horizontales et verticales en utilisant l'ordre induit par le 2d-tree. Ils démontrent que la triangulation – en excluant le temps consacré à la construction du 2d-tree – est linéaire en moyenne

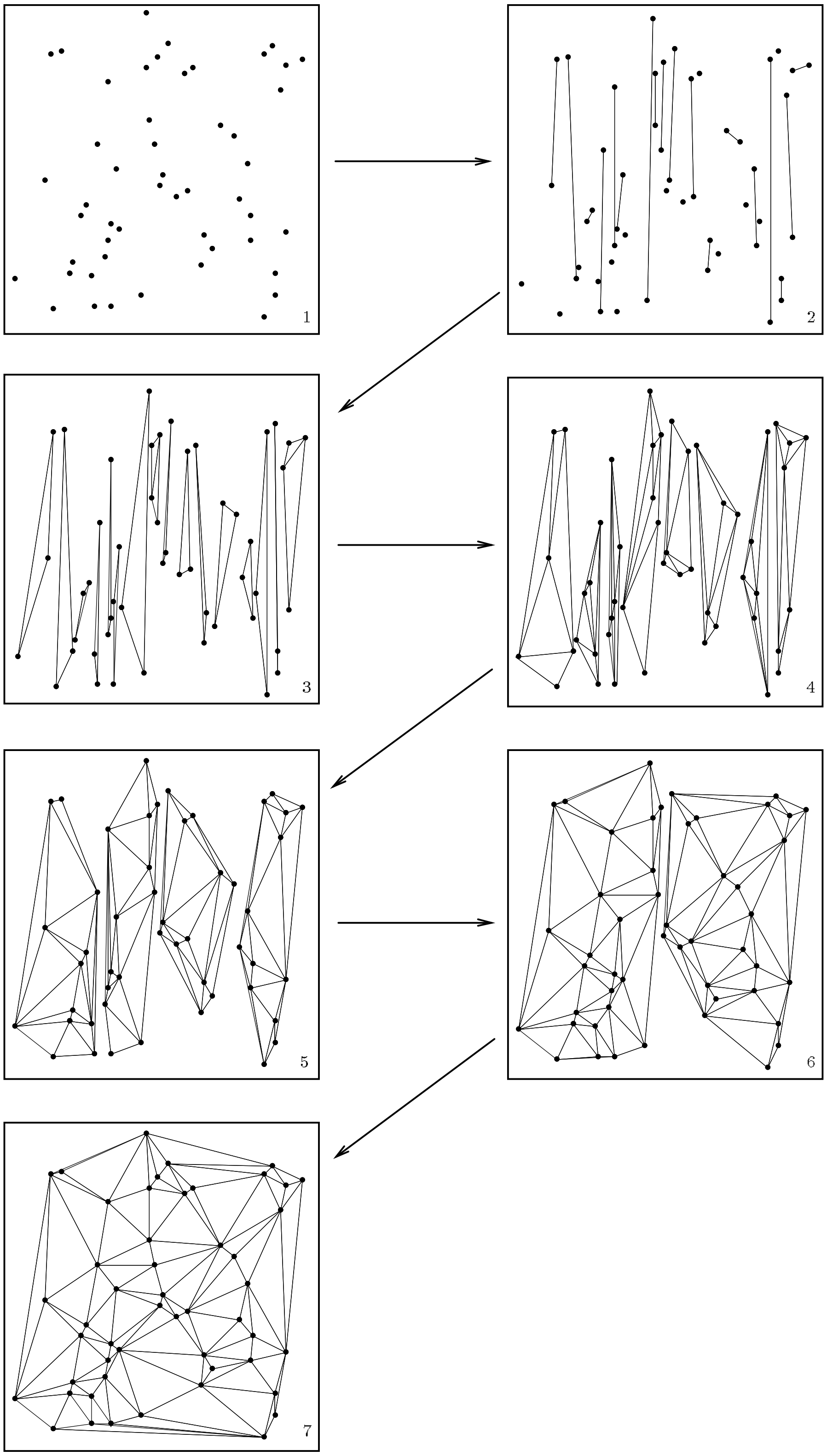


Figure 1.9: *Triangulation de Delaunay d'après Lee et Schachter.*

(pour une distribution quasi-uniforme), et optimale dans le pire des cas. Cette méthode, comme le confirment les tests du chapitre 7, s'adapte bien aux distributions non uniformes. L'algorithme global reste en $O(n \log n)$, même en moyenne, car la construction du 2d-arbre est en $\Theta(n \log n)$. Mais le temps passé à cette construction est bien moins important que le temps passé pour trianguler, tout du moins pour des ensembles jusqu'à sept millions de sites. Il faudrait atteindre plusieurs dizaines de millions de points pour que cette tendance s'inverse. Cela confirme le danger des analyses asymptotiques pour les complexités. Sur les tailles de données que l'on rencontre en pratique, elles ne rendent pas toujours compte du classement expérimental entre les algorithmes. Tous ces algorithmes – Lee et Schachter, Dwyer, Katajainen et Koppinen, Lemaire et Moreau (avec de nombreuses variantes) – ont été testés expérimentalement dans le chapitre 7.

1.5 Triangulation de Delaunay contrainte

Soit S un ensemble de sites du plan et A un ensemble d'arêtes dont les extrémités appartiennent à S. La *triangulation de Delaunay contrainte* [175] de (S, A) est une triangulation qui contient toutes les arêtes de contrainte et dont chaque arête $s_i s_j$ qui n'est pas une contrainte, vérifie la propriété suivante : il existe un cercle ayant pour corde $s_i s_j$ et tel que, si un autre site se trouve à l'intérieur de ce cercle, alors on ne peut le relier à s_i et s_j sans couper une contrainte. On peut aussi dire que ce cercle ne contient en son intérieur aucun site de S *visible*[5] de l'arête $s_i s_j$.

Le critère d'équiangularité local est toujours valable si l'on s'interdit de modifier les arêtes de contrainte. Cette triangulation maximise aussi la finesse avec la même condition. Le critère du cercle vide est légèrement modifié : on accepte qu'un site soit à l'intérieur du cercle circonscrit à un triangle à condition de ne pas être *visible* d'au moins un des sommets du triangle. Joe et Wang ont introduit une définition étendue du diagramme de Voronoi contraint [113] pour en faire le dual de la triangulation de Delaunay contrainte. Notons qu'en dimension 2, tout problème de triangulation contrainte admet une solution, ce qui n'est pas toujours le cas en dimension supérieure ([34] p 303).

On peut avoir besoin de la triangulation de Delaunay contrainte pour trianguler un polygone, trouver le chemin le plus court en présence d'obstacles, modéliser le terrain naturel (les contraintes représenteront les lignes caractéristiques du relief : lignes de crête, de talweg, failles, limites de cours d'eau...). Pour calculer la triangulation de Delaunay contrainte, on retrouve à peu près les mêmes types d'algorithmes que précédemment (voir le survol de De Floriani et Puppo [85] ainsi que la thèse de

[5]Un point est dit visible par rapport à un autre si le segment défini par ces deux points ne coupe pas une arête de contrainte.

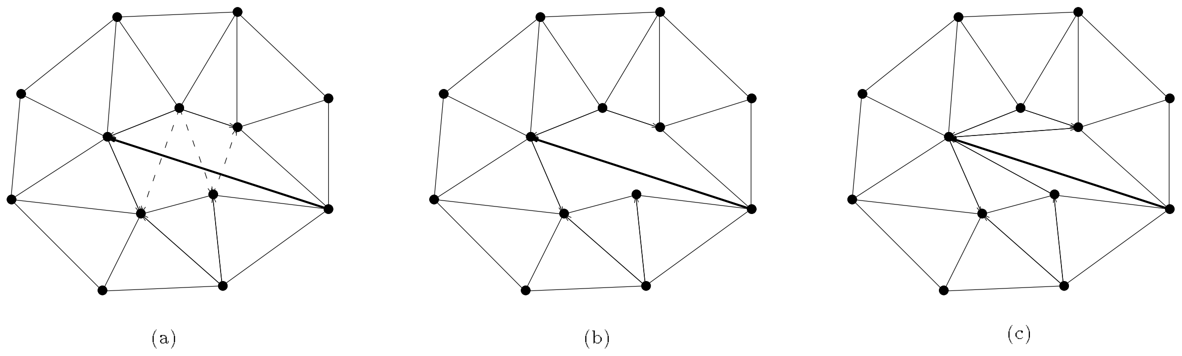


Figure 1.10: *Triangulation de Delaunay contrainte incrémentale : (a) insertion d'une contrainte, (b) destruction des arêtes coupées par la contrainte, triangulation des deux polygones de part et d'autre de la contrainte.*

T. Lambert [125]). Une méthode simple consiste à trianguler l'ensemble des sites sans tenir compte des arêtes de contrainte, puis à insérer les contraintes de façon incrémentale. Pour insérer une contrainte dans la triangulation, on retire toutes les arêtes coupées par cette contrainte, puis on retriangule le "trou" polygonal créé de part et d'autre de la contrainte (voir Figure 1.10). La complexité dans le pire des cas est en $O(n^2)$, voire même en $O(n^3)$ si l'on n'utilise pas un algorithme optimal pour retrianguler les polygones créés par les insertions de contraintes. Mais, en pratique, les arêtes de contrainte sont parfois peu nombreuses et de faible longueur et on ne se trouve pas dans le "pire des cas". Souvent, cette méthode fonctionne relativement bien avec des performances acceptables. Les algorithmes à base d'échanges d'arêtes sont aussi très simples. La seule difficulté est de construire la triangulation contrainte initiale [90]. Comme algorithme de sélection, citons par exemple celui de Fang et Piegl qui utilise une grille régulière pour la localisation [79]. Nous ne connaissons pas d'algorithmes de transformation géométrique pour le calcul de la triangulation de Delaunay contrainte. L'algorithme de balayage a été adapté à la triangulation de Delaunay contrainte par Seidel [175] en temps optimal $O(n \log n)$. L'algorithme Divide and Conquer a été adapté à la triangulation contrainte par Chew [50], aussi optimal. Moreau et Volino ont amélioré l'algorithme de Chew et l'ont rendu plus pratique [152]. Il existe même des utilisations industrielles de leur programme. J. M. Moreau a aussi conçu un algorithme optimal pour la triangulation de Delaunay contrainte et hiérarchique [153] (on choisit de trianguler ou non dans des contours polygonaux imbriqués).

Citons pour mémoire la *triangulation de Delaunay conforme* qui est une triangulation de Delaunay contrainte particulière. Les segments de contrainte sont découpés de telle sorte que leur triangulation de Delaunay contrainte soit égale à la triangulation de Delaunay non contrainte de l'ensemble total des points (sites + extrémités de contrainte + points ajoutés lors du découpage des segments). La triangulation de Delaunay conforme présente l'avantage de supprimer les triangles très aplatis que peuvent générer les arêtes de contrainte. On trouvera dans la thèse de M. Buffa [40] de nombreuses références sur les algorithmes de calcul de la triangulation de Delaunay conforme.

Chapitre 2

ARBRES MULTIDIMENSIONNELS

Ce chapitre contient quelques rappels sur les arbres multidimensionnels qui seront utilisés par la suite dans de nouveaux algorithmes de triangulation de Delaunay. On peut trouver dans l'ouvrage de Samet [171] plus de 250 références sur les quadtrees et kd-trees, et aussi beaucoup d'autres sur les techniques de bucketing.

2.1 Quadtrees

Le quadtree, structure de données inventée par Finkel et Bentley [80], est couramment utilisé en informatique graphique et aussi dans quelques systèmes d'information géographique (S.I.G.) où il permet de stocker des points, des segments, des surfaces... On trouve dans l'ouvrage de Samet [171] de nombreuses références bibliographiques sur les quadtrees et leurs variantes (point quadtree, region quadtree, pseudo quadtree, matrix (MX) quadtree, point region (PR) quadtree...).

2.1.1 Principe en dimension 2

Le *quadtree*[1] est une structure de données hiérarchique basée sur une décomposition récursive de l'espace. Il réalise une partition de l'espace par des carrés de plus en plus petits (voir figure 2.1). La racine de l'arborescence correspond au carré englobant de l'ensemble des points. Tous les carrés situés à un même niveau de l'arborescence sont de même taille. Au niveau juste en dessous, chaque carré a été divisé par quatre. Chaque noeud a quatre fils qui correspondent aux quatre quadrants de la région, que l'on repère souvent par NE, NW, SE, SW. Les points sont stockés au niveau des feuilles de cette arborescence quaternaire. C'est en quelque sorte un hachage récursif (division

[1] Il s'agit plus précisément dans ce chapitre du Point Region (PR) Quadtree : les feuilles contiennent un site unique ou alors sont vides.

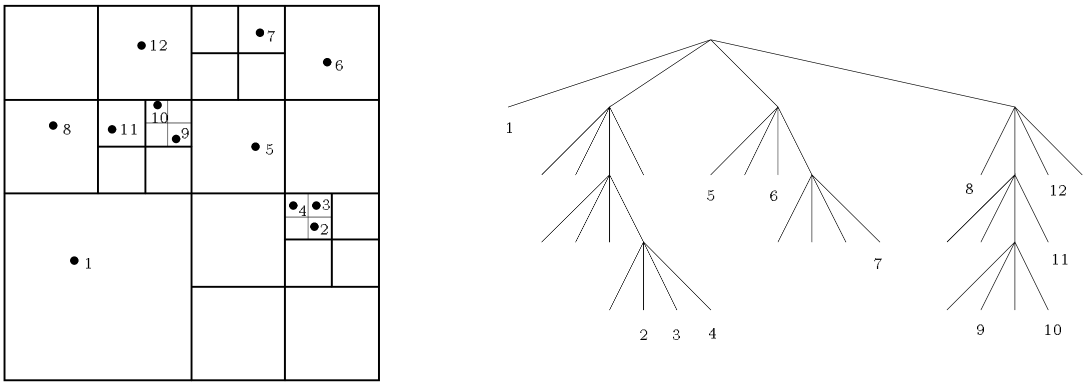


Figure 2.1: *Un quadtree.*

par quatre), qui s'arrête quand les cellules ne contiennent pas plus d'un point. Il se généralise en dimension quelconque.

2.1.2 Construction en dimension 2

On peut le construire de façon incrémentale, en insérant le point $j+1$ dans le quadtree déjà construit avec les j premiers points.

Initialisation : on crée la racine de l'arbre qui est associée au carré englobant de l'ensemble de points. On crée ses quatre fils qui sont, pour l'instant, des feuilles vides de l'arborescence et auxquels sont associés les quatre quadrants du carré initial. On place dans le bon quadrant le premier point.

Insertion d'un point M : il faut dans un premier temps trouver le quadrant élémentaire contenant M.

1. **Parcours de l'arborescence** : on descend dans l'arborescence jusqu'à atteindre la feuille dont la région associée contient le point à insérer. Pour chaque noeud, il faut connaître les coordonnées du centre du carré associé ainsi que la demi-longueur de son côté. Au niveau de la racine, ces informations sont connues. A chaque fois que l'on descend d'un niveau, on calcule la demi-longueur du côté du carré associé au noeud $dm = dm/2$ ainsi que les coordonnées de son centre $x_m = x_m \pm dm$ et $y_m = y_m \pm dm$ (les signes dépendent du quadrant vers lequel on s'est dirigé). A chaque noeud, de simples comparaisons entre les coordonnées du centre du carré associé et celles du point à insérer indiquent vers lequel des quatre quadrants il faut se diriger. On itère ce processus pour trouver la feuille dont la région associée contient le point à insérer.

2. **Ajout du point** : la procédure est différente selon que le quadrant trouvé est vide ou déjà occupé.
   (a) la feuille trouvée est vide, on lui affecte le point à insérer, la procédure est terminée.
   (b) la feuille contient déjà un point P, elle devient un noeud interne et on crée ses quatre fils. On localise le (ou les) quadrant(s) contenant P et M. Tant que P et M ne sont pas dans des quadrants distincts, on divise le quadrant en quatre et on trouve le (ou les) quadrant(s) contenant P et M. A la fin de cette itération, P et M sont rangés dans des quadrants distincts voisins.

Quand tous les points ont été ajoutés incrémentalement, le quadtree est construit. C'est une structure dynamique car on peut ajouter des points dans la structure. On peut aussi en supprimer à l'aide d'une procédure qui se déduit facilement de la procédure d'insertion. Après une mise à jour (insertion ou suppression), on n'a pas de travail de rééquilibrage puisque le quadtree n'est pas une structure équilibrée, mais par contre la hauteur de l'arborescence n'est pas logarithmique. Les inconvénients d'une telle structure vont être évoqués dans le paragraphe suivant.

2.1.3 Analyse

Les principaux défauts de cette structure sont : un espace mémoire non linéaire et un temps de construction en $\omega(n \log n)$ dans le pire des cas. Imaginons par exemple deux points extrêmement proches, situés de part et d'autre du milieu d'un quadrant. Pour pouvoir insérer ces deux points dans le quadtree, il faudra faire un nombre considérable de subdivisions ce qui ajoutera un nombre considérable de niveaux dans l'arborescence. La hauteur d'un quadtree n'est pas bornée en fonction du nombre de points ! en conséquence, le temps de construction non plus !

Plus généralement, si l'on suppose que les n points sont situés dans un carré de longueur s, et que la distance euclidienne minimale séparant deux points est d, alors la hauteur maximale du quadtree est $\lceil \log_2 \frac{s\sqrt{2}}{d} \rceil$ et le temps de construction en $O(n\lceil \log_2 \frac{s\sqrt{2}}{d} \rceil)$. Dans les applications pratiques, la dynamique des données n'est pas infinie et on peut donc appliquer le résultat précédent. Si les points sont situés sur une grille de taille $2^\alpha \times 2^\alpha$, l'espace mémoire est en $O(\alpha \times n)$ ainsi que le temps de construction (voir par exemple [72]).

Il existe des techniques pour optimiser l'espace mémoire. On observe que, dans un quadtree, il y a des chemins où les noeuds internes ont trois fils qui sont des feuilles vides. Il est possible de remplacer ce chemin par un seul noeud qui en contient une représentation codée. L'espace mémoire est alors linéaire (mais pas le temps de cons-

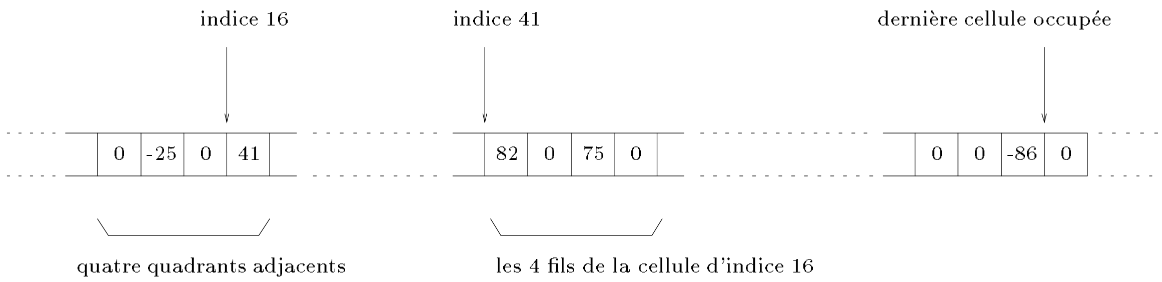


Figure 2.2: *Représentation en tableau d'un quadtree : un noeud est représenté par un nombre positif qui est l'indice du premier fils, une feuille vide est représentée par 0, une feuille pleine est représentée par un nombre négatif qui est l'opposé de l'indice du point dans le tableau de points.*

truction), c'est le *linear quadtree* [171].

En pratique, sur la plupart des distributions, on observe un espace mémoire linéaire et un temps de construction en $O(n \log n)$. Dans l'étude expérimentale du chapitre 7, le quadtree est représenté par un tableau et en occupe environ les $3n$ premières cellules (voir annexe D.1). Sa construction est en $O(n \log n)$ avec une constante multiplicative très petite (voir annexe E.1).

2.1.4 Programmation en dimension 2

Par souci d'efficacité, un tableau sera utilisé pour représenter en machine le quadtree. En effet, une représentation en arbre quaternaire oblige à allouer continuellement de petits espaces mémoire lors de la création de chaque noeud, d'où une perte de temps considérable. Pour un ensemble de n points, on alloue au début du programme un tableau de $4n$ cellules. En général, comme le confirme le chapitre 7, le tableau n'est rempli qu'aux trois quarts. Pour éviter tout débordement, on a une variable qui contient l'indice de la dernière cellule occupée du tableau. Si cet indice dépasse la taille du tableau, on réalloue un tableau plus grand. En pratique, le débordement n'est jamais survenu pour des ensembles de plus de deux mille points.

Une cellule contient un nombre entier (voir Figure 2.2). Un noeud est représenté par un nombre positif qui est l'indice du premier fils, une feuille vide est représentée par 0, une feuille pleine est représentée par un nombre négatif qui est l'opposé de l'indice du point dans le tableau de points. Quand on transforme une feuille en noeud lors de la création du quadtree, on place ses quatre fils en fin de tableau. Ainsi, ils sont toujours contenus dans quatre cellules adjacentes. L'algorithme de construction est celui décrit dans le paragraphe précédent.

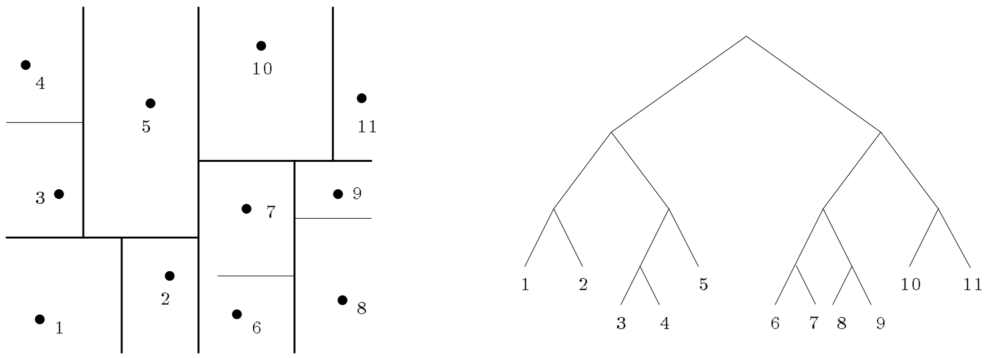


Figure 2.3: *Un kd-tree.*

2.2 Kd-trees

Le *kd-tree* est une structure de données inventée par Bentley [16], qui permet de stocker dans un arbre binaire *équilibré* un ensemble de points $\mathcal{S}$ dans un espace de dimension k. Contrairement au quadtree, les divisions de l'espace ne sont pas régulières, mais sont effectuées d'après les coordonnées des points (c'est une amélioration du Point Quadtree [171]). Le kd-tree est basé sur une division de l'espace par des rectangles irréguliers de taille décroissante. Cette structure de données permet de répondre assez facilement à des requêtes comme : est-ce qu'un point donné appartient à $\mathcal{S}$?, quel est le plus proche voisin (nearest neighbour) dans $\mathcal{S}$ d'un point donné ?, quels sont les points de $\mathcal{S}$ (range searching) inclus dans un domaine donné ?... Cette structure n'est optimale que pour le premier type de requête. Pour les deux autres types de requête, elle a de bonnes performances *en moyenne* (voir par exemple [18]) alors que les performances dans le pire des cas sont assez médiocres.

2.2.1 Définitions

Un 2d-tree est construit de la façon suivante (voir Figure 2.3) : on choisit un nombre x qui est stocké dans la racine de l'arbre. Le sous-arbre gauche contient tous les points dont la première coordonnée est inférieure à x, le sous-arbre de droite contient tous les points dont la première coordonnée est plus grande que x. Dans chaque sous-arbre, un nouveau nombre est choisi qui servira à diviser l'ensemble des points, mais par rapport à la seconde coordonnée. Les quatre sous-ensembles ainsi obtenus sont de nouveau divisés relativement à la première coordonnée. On itère ce processus jusqu'à ce que les ensembles ne contiennent plus qu'un seul point qui constitue une feuille de l'arborescence. La racine divise donc l'ensemble des points selon la première coordonnée, les deux noeuds du second niveau selon la seconde coordonnée, les quatre noeuds du troisième niveau selon la première coordonnée et ainsi de suite... Ce processus peut être généralisé dans un espace de dimension k pour construire un kd-tree.

En principe, le kd-tree est équilibré (par la suite, on emploiera abusivement le terme kd-tree au lieu de kd-tree équilibré). Pour cela, il faut que, quel que soit le noeud considéré, son sous-arbre gauche et son sous-arbre droit aient même cardinal à une unité près. Pour obtenir un kd-tree équilibré, il faut diviser l'ensemble associé à chaque noeud selon son élément médian. De cette manière, la hauteur de l'arborescence est logarithmique.

Si l'on choisit de diviser l'ensemble relatif à chaque noeud selon une valeur aléatoire, on obtiendra un *random kd-tree*. Sa construction est plus rapide (voir annexe E.1), mais on ne maîtrise plus sa hauteur [66] qui, dans le pire des cas, peut devenir linéaire.

Dans un kd-tree classique, on alterne, d'un niveau à son suivant, la dimension qui sert à faire la division. Nous allons maintenant construire un “meilleur” kd-tree où, à chaque noeud, on choisit la dimension qui servira à faire la division. Quel est le critère utilisé pour le choix ? considérons le rectangle englobant de l'ensemble de points associé à un noeud. On choisit de le diviser orthogonalement à son plus grand côté. Ainsi, les rectangles associés à chaque noeud auront une forme presque carrée. Chaque division produit des rectangles dont la largeur et la longueur sont les plus proches possibles. On appelle cette arborescence un *adaptive kd-tree* (dû à Friedman, Bentley et Finkel [89]). Il est équilibré et divise le domaine en rectangles de forme presque carrée. Sa construction est un peu plus longue (voir annexe E.1) que pour un kd-tree classique, mais ses performances (nearest neighbour et range searching) sont meilleures en moyenne.

Si l'on choisit aléatoirement la valeur qui sert à faire une division, on aura un *random adaptive kd-tree*. Sa construction est plus rapide (voir annexe E.1), mais on ne maîtrise plus sa hauteur qui, dans le pire des cas, peut devenir linéaire.

2.2.2 Divisions selon deux directions

Appelons M_x et M_y l'abscisse et l'ordonnée d'un point M. Définissons la relation $<_H$ par :

$$M <_H N \Leftrightarrow ((M_x < N_x) \textit{ ou } (M_x = N_x \textit{ et } M_y < N_y))$$

Définissons la relation $<_V$ par :

$$M <_V N \Leftrightarrow ((M_y < N_y) \textit{ ou } (M_y = N_y \textit{ et } M_x > N_x))$$

Soient A et B deux ensembles de points du plan. On dira que $A <_H B$ (resp. $A <_V B$) si et seulement si tous les points de A sont $<_H$ (resp. $<_V$) à tous les points de B.

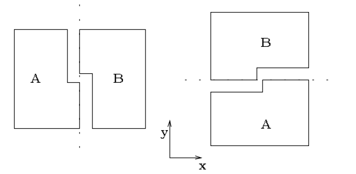


Figure 2.4: *X et Ydivisions.*

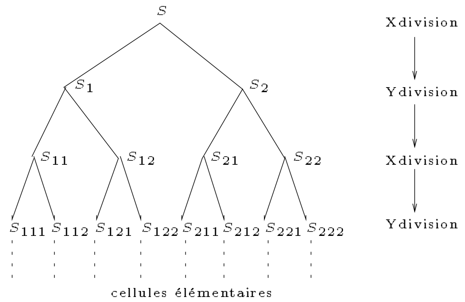


Figure 2.5: *L'alternance des divisions dans un 2d-tree.*

On dira que l'on *Xdivise* (resp. *Ydivise*) un ensemble A de points du plan en A_1 et A_2 si et seulement si $A_1 <_H A_2$ (resp. $A_1 <_V A_2$), avec $A_1 \cup A_2 = A$ et si A_1 et A_2 ont le même nombre de points (à une unité près) (remarque : A_1 ou A_2 peut être vide) (Figure 2.4).

Un $2d$-tree est obtenu en divisant alternativement le plan suivant des directions horizontales et verticales. Plus précisément, on commence par *Xdiviser* l'ensemble S en deux parties S_1 et S_2. Ensuite, on *Ydivise* S_1 (resp. S_2) en deux parties S_{11} et S_{12} (resp. S_{21} et S_{22}). Ensuite, les parties S_{11}, S_{12}, S_{21} et S_{22} sont *Xdivisées.* On itère ce processus en alternant les *Xdivisions* et les *Ydivisions* jusqu'à l'obtention de cellules élémentaires contenant un point (Figure 2.5).

2.2.3 Algorithmes de construction

Il existe au moins trois façons de construire un kd-tree :

- pour diviser chaque sous-ensemble, on peut utiliser l'algorithme linéaire déterministe de sélection de la médiane de Blum et al. [29], puis on répartit les points de part et d'autre de cette médiane, toujours en temps linéaire. La complexité totale de la construction est optimale en $\Theta(n \log n)$.
- on peut aussi réaliser deux liens de chaînage entre les points, représentant l'un l'ordre en X, l'autre l'ordre en Y. A l'aide de ces deux chaînages, il est facile de trier le tableau suivant les deux directions avec une complexité totale optimale en $\Theta(n \log n)$.
- on peut aussi trouver la médiane de chaque sous-ensemble avec une méthode dérivée de Quicksort, randomisée ([57] page 184) ou non [107]. Bien que cet algorithme ne soit pas optimal dans le pire des cas, on constate expérimentalement

qu'il est nettement plus rapide sur la plupart des distributions (en pratique, la probabilité de rencontrer un "pire des cas" – pour cet algorithme – est quasiment nulle). Dans la partie expérimentale (voir chapitre 7), le 2d-tree est construit de cette façon.

2.2.4 Programmation en dimension 2

Pour les mêmes raisons que dans le paragraphe 2.1.4, un tableau sera utilisé pour représenter le 2d-tree. Le schéma algorithmique non randomisé[2] est le suivant :

on suppose que les sites ont déjà été rangés dans un tableau $T[\ ]$ sans point double ; l et r représentent les extrémités du tableau courant dans chaque fonction, et l'on pose $\mu = \lceil (l+r)/2 \rceil$.

Fonction Twod-tree (l, r)
    Xsort (l, r)
Fonction Xsort (l, r)
    Si $(l \neq r)$ **Alors**
        $ll = l$ et $rr = r$
        Tant que $((rr - ll) \geq NCUT)$ **Faire**
            $i = ll$, $j = rr$, $cle = T[\mu]$
            Boucle
                Tant que $(T[i] <_x cle)$ **Faire** incrémenter i
                Tant que $(cle <_x T[j])$ **Faire** décrémenter j
                Si $(i < j)$ **Alors**
                    échanger $T[i]$ et $T[j]$
                    incrémenter i et décrémenter j
                Sinon
                    Fin de Boucle
            Si $(j < m)$ **Alors** $ll = i$
            Si $(m < i)$ **Alors** $rr = j$
        FinTantque
        Xtriselection (ll, rr)
        Ysort $(l, \mu - 1)$
        Ysort (μ, r)

La fonction *Xtriselection* trie selon l'ordre $<_x$ la première moitié du tableau réduit entre les indices ll et rr en utilisant un algorithme de tri par sélection. En effet, quand le

[2]la version randomisée consiste à prendre comme clé un élément aléatoire du tableau plutôt que systématiquement l'élément situé en son milieu.

nombre d'éléments du tableau devient inférieur à $NCUT$ (de l'ordre de 10), il est plus rapide de *Xdiviser* l'ensemble avec un algorithme de tri par sélection qu'avec l'autre méthode dérivée de Quicksort.

On obtient la fonction *Ysort* en remplacant $<_x$ par $<_y$ et *Xtriselection* par *Ytriselection* dans la fonction *Xsort*.

Remarques :

1. plutôt que de prendre sytématiquement comme clé l'élément situé au milieu du tableau, on peut la choisir parmi trois éléments du tableau (par exemple, un situé à gauche, l'autre au milieu et le dernier à droite). La clé sera égale à l'élément médian des trois. Cette variante porte le nom de partitionnement par la *médiane de trois* ([178] page 132). Cette technique, quand on l'utilise pour QuickSort, réduit le temps de calcul d'environ 25% en moyenne.

2. Floyd et Rivest [86] utilisent un échantillon d'environ $\sqrt{n}$ éléments pour trouver la médiane avec environ $\frac{3n}{2}$ comparaisons.

3. Bentley [18] calcule la médiane exacte d'un échantillon de $\sqrt{n}$ éléments – qui est une approximation de la médiane de l'ensemble complet – et partitionne autour de cette valeur en exactement n comparaisons. On a dans ce cas un random 2d-tree presque équilibré. Bentley utilise cette méthode pour construire un random adaptive 2d-tree – relativement bien équilibré – et il observe une réduction du temps de construction de 40%.

Il serait intéressant de programmer ces variantes pour pouvoir mesurer expérimentalement le gain qu'elles procurent.

2.2.5 *K*d-trees dynamiques

Dans un espace à une dimension, il existe de nombreux types d'arbres binaires de recherche qui s'équilibrent dynamiquement (voir par exemple [13]). Cela signifie que l'on peut insérer ou supprimer des éléments en maintenant une hauteur logarithmique pour l'arbre, le coût d'une insertion ou d'une suppression étant aussi logarithmique. L'équilibrage se fait par des rotations (simples ou doubles) au niveau de certains noeuds. Citons les arbres AVL introduits par Adelson-Velskii et Landis [5] (d'où leur nom), les arbres bicolores (appelés aussi arbres rouge-noir), dus à Guibas et Sedgewick [99] (L. Chen et R. Schott [48] optimisent l'espace mémoire.), les arbres persistants de Driscoll, Sarnak, Sleator et Tarjan [69], qui conservent en plus un historique des versions antérieures de la structure... Comme ces arbres équilibrés ont toujours une hauteur logarithmique, ils permettent de réaliser efficacement des recherches en temps logarithmique.

Malheureusement, dès que l'on se place dans un espace de dimension supérieure ou égale à 2, l'équilibrage dynamique semble impossible. Peut-être est-ce lié au fait qu'il n'existe pas d'ordre dans le plan ! En 1979, Bentley écrivait [17] : "*perhaps the most outstanding open problem is that of maintaining dynamic kd-trees*". Overmars, dont la thèse [162] est consacrée aux structures de données dynamiques, affirmait en 1983 : "*the techniques used for dynamizing simple data structures seemed not to be applicable to multidimensionnal data structures*"... "*multidimensional data structures appeared to be very hard to dynamize*".

Aucune solution comparable en *simplicité et efficacité* à celles qui existent dans le cas unidimensionnel, n'a été trouvée (voir le "survol" de Chiang et Tamassia [52] sur les algorithmes dynamiques). Overmars propose deux solutions [162] :

1. le "*partial rebuilding*" consiste à faire un certain nombre de suppressions et d'insertions sans se soucier de l'équilibrage, et, quand un certain niveau de déséquilibre est atteint, on rééquilibre les parties dégénérées. La complexité en temps *amorti* est bonne, $O(\frac{1}{k}\log^2 m)$ où k est la dimension de l'espace et m le nombre maximum de points présents au même moment dans l'arbre.

2. avec le "*global rebuilding*", on reconstruit *toute* l'arborescence quand un certain niveau de déséquilibre est atteint. Le temps *amorti* d'insertion est en $O(\log^2 n)$, le temps *amorti* de suppression est en $O(\log n)$.

Van Kreveld et Overmars ont proposé le *divided kd-tree* [124]. C'est une structure dynamique. Les mises à jour se font en temps logarithmique dans le pire des cas. Les niveaux supérieurs de l'arborescence sont obtenus par des divisions selon une seule direction, les niveaux inférieurs par des divisions selon l'autre direction. Il n'y a plus l'alternance des directions pour effectuer les divisions, caractéristique des kd-trees. Les performances pour le range searching – analysées dans le *pire des cas* – sont presque les mêmes qu'avec les kd-trees classiques. On peut regretter que l'analyse en moyenne des *divided kd-trees* ne soit pas faite (les kd-trees ont des performances très bonnes en moyenne, mais médiocres dans le pire des cas) et que les résultats expérimentaux ne soient pas donnés par les auteurs. Il est donc difficile de porter un jugement sur l'intérêt pratique d'une telle structure.

En conclusion, la dynamisation des kd-trees est assez décevante. Utiliser un kd-tree comme outil de localisation dans un algorithme incrémental de triangulation de Delaunay présente un intérêt limité.

2.3 Bucket-trees

La technique de bucketing (tri par paquets) permet de résoudre efficacement *en pratique* divers problèmes géométriques [7] [64] comme le couplage de poids minimal dans le plan, le diagramme de Voronoi, la localisation d'un point, la recherche de points dans un domaine, le plus court chemin... Cette technique est quelque peu délaissée par les théoriciens car elle n'améliore pas la complexité dans le pire des cas des algorithmes. Par contre, en faisant des hypothèses sur la distribution des sites, elle produit des algorithmes dont la complexité en moyenne est souvent meilleure que la borne inférieure de la complexité dans le pire des cas.

2.3.1 Principe

Les cellules ("buckets") sont des régions dont la réunion constitue une partition d'un domaine. Elles vérifient généralement les caractéristiques suivantes :

1. elles ont une forme simple, pour que la partition du domaine soit facile à réaliser, et aussi pour qu'il soit aisé de déterminer à quelle cellule un point appartient.

2. le nombre de sites inclus dans une cellule est borné par une constante (cette condition est facile à réaliser si la distribution des points est uniforme). On choisit en général un nombre de cellules égal au nombre de sites.

3. dans beaucoup de problèmes géométriques, l'influence entre les objets décroît en fonction de la distance. Une bonne stratégie consiste à calculer des solutions locales, puis à les combiner pour obtenir la solution globale.

Quand toutes ces propriétés sont vérifiées, les algorithmes ont souvent une complexité linéaire en moyenne.

2.3.2 Division en cellules

Considérons le rectangle englobant R de l'ensemble de points $\mathcal{S}$ tels que ses côtés soient parallèles aux axes de coordonnées. Découpons ce rectangle en $n_x \times n_y$ cellules qui sont toutes carrées et de même taille, sauf éventuellement celles qui sont adjacentes à un côté de R. Une cellule qui est la $i^{ième}$ en partant de la gauche et la $j^{ième}$ en partant du bas, est notée C_{ij}. On dit que (i, j) sont les coordonnées de C_{ij} dans le *système de coordonnées des cellules*. Soient (a_1, b_1) (resp. (a_2, b_2)) les coordonnées du coin inférieur gauche (resp. supérieur droit) de R. Soit $M(x, y)$ un point quelconque dans R. Les coordonnées de la cellule C_{ij} (c étant la longueur de son côté) qui contient ce

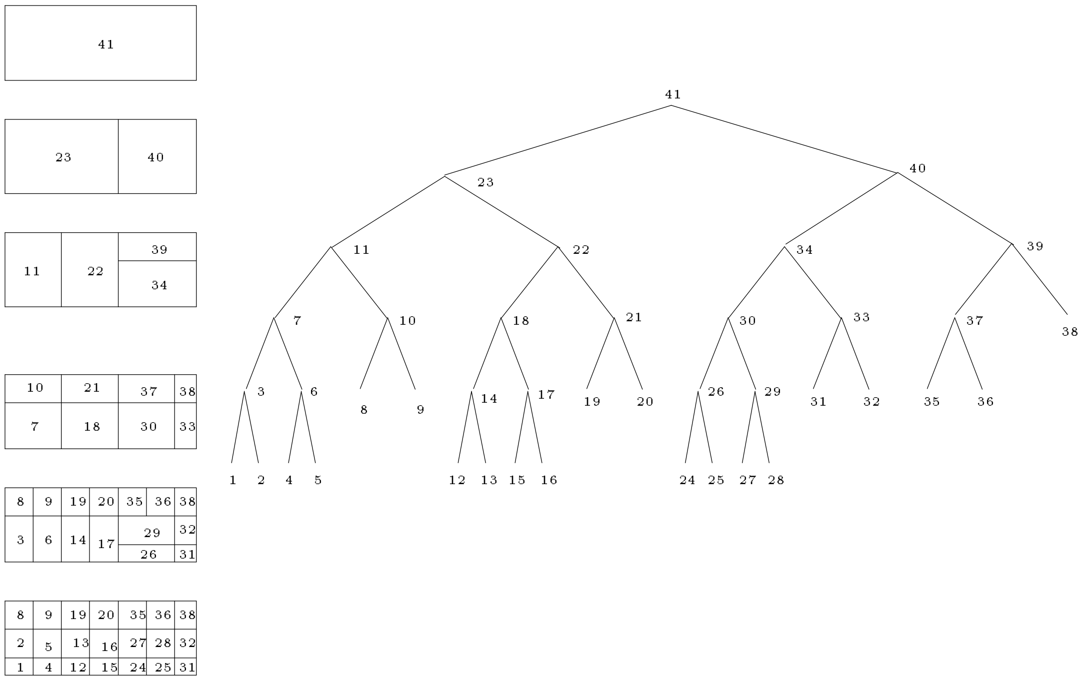


Figure 2.6: *Un 'bucket-tree'.*

point sont :

$$i = \left\lfloor \frac{x - a_1}{c} \right\rfloor \qquad j = \left\lfloor \frac{y - b_1}{c} \right\rfloor$$

Le nombre de cellules adjacentes aux côtés de R est :

$$n_x = \left\lfloor \frac{a_2 - a_1}{c} + 1 \right\rfloor \qquad n_y = \left\lfloor \frac{b_2 - b_1}{c} + 1 \right\rfloor$$

2.3.3 Construction

Il faut allouer un tableau de $n_x \times n_y$ cellules. Chaque cellule a la liste chaînée des points qu'elle contient. Comme l'affectation d'un point dans une cellule se fait en temps constant, la construction de ce tableau se fait en $O(\max(n, n_x \times n_y))$. Le nombre de cellules n'étant en principe jamais supérieur au nombre de points, on considère que la construction de ce tableau est linéaire.

2.3.4 Arborescence de cellules

Dans les algorithmes utilisant des techniques de bucketing, on calcule d'abord une solution locale pour chaque cellule. Ensuite, on combine les résultats partiels pour obtenir la solution globale. Un ordre dans l'ensemble des cellules est donc nécessaire pour gérer la fusion des résultats. Il existe des ordres en "serpentin", en spirale, quaternaires [7]... Nous proposons d'utiliser *l'ordre postfixé* [178] (voir Figure 2.6) *d'un adaptive 2d-tree* construit avec les cellules, qui semble performant pour le calcul de la triangulation de Delaunay (voir chapitre 5).

2.3.5 Programmation en dimension 2

Cet ordre s'obtient facilement avec deux fonctions récursives. Le schéma algorithmique est le suivant :

Fonction Buckettree(X_0, Y_0, dX, dY)
    Si ($dX > dY$) **alors** Xcoupe(X_0, Y_0, dX, dY)
    Sinon Ycoupe(X_0, Y_0, dX, dY)
Fonction Xcoupe(x, y, dx, dy)
    Si ($dx == 1$) **alors** TraiterCelluleElementaire(x, y)
    Sinon
        $dx_1 = \lceil dx/2 \rceil$
        $dx_2 = dx - dx_1$
        Si ($dx_1 > dy$) **alors** Xcoupe(x, y, dx_1, dy)
        Sinon Ycoupe($x, y, dx1, dy$)
        Si ($dx_2 > dy$) **alors** Xcoupe ($x + dx_1, y, dx_2, dy$)
        Sinon Ycoupe($x + dx_1, y, dx_2, dy$)
        TraiterRectangle(x, y, dx, dy)
Fonction Ycoupe(x, y, dx, dy)
    Si ($dy == 1$) **alors** TraiterCelluleElementaire(x, y)
    Sinon
        $dy_1 = \lceil dy/2 \rceil$
        $dy_2 = dy - dy_1$
        Si ($dy_1 > dx$) **alors** Ycoupe(x, y, dx, dy_1)
        Sinon Xcoupe(x, y, dx, dy_1)
        Si ($dy_2 > dx$) **alors** Ycoupe($x, y + dy_1, dx, dy_2$)
        Sinon Xcoupe($x, y + dy_1, dx, dy_2$)
        TraiterRectangle(x, y, dx, dy)

x et y sont les coordonnées (dans le système de coordonnées des cellules) du coin inférieur gauche du rectangle considéré, dx (resp. dy) la longueur (l'unité étant la longueur du côté d'une cellule élémentaire) d'un côté du rectangle parallèle à l'axe Ox (resp. Oy).

2.4 Conclusion

Le quadtree, les différents kd-trees et le bucket-tree ont été utilisés dans de nouveaux algorithmes de triangulation de Delaunay (voir chapitre 5). Mais il existe beaucoup d'autres arbres multidimensionnels qui pourraient servir de support à des algorithmes de triangulation de Delaunay. Le principe est identique. L'étude expérimentale et la comparaison avec d'autres algorithmes existants ont été réalisées dans le chapitre 7.

Chapitre 3

FUSION BIDIRECTIONNELLE DE SOUS-TRIANGULATIONS DE DELAUNAY

On se propose de résoudre le problème suivant : on dispose des triangulations de Delaunay des ensembles disjoints de sites S_1 et S_2 et on cherche à obtenir de façon optimale la triangulation de Delaunay de $S_1 \cup S_2$.

Shamos et Hoey [181] arrivent à fusionner en temps linéaire deux diagrammes de Voronoi, à condition que les ensembles de sites S_1 et S_2 soient séparables par une droite. Comme la triangulation de Delaunay est le dual du diagramme de Voronoi, on peut passer de l'un à l'autre en temps linéaire. Kirkpatrick [118] donne ensuite une solution générale, mais compliquée, pour fusionner deux diagrammes de Voronoi. Il n'a pas besoin de la condition de séparabilité précédente. Son algorithme utilise les arbres couvrants minimaux euclidiens des deux ensembles de sites.

Les deux algorithmes précédents permettaient d'obtenir de façon indirecte la triangulation de Delaunay de $S_1 \cup S_2$. La première méthode directe pour fusionner deux triangulations de Delaunay séparables par une droite, est due à Lee et Schachter [131]. Elle est optimale, linéaire et plus simple qu'en passant par les diagrammes de Voronoi associés. Ensuite, Guibas et Stolfi [100] ont repris l'algorithme de Lee et Schachter en le détaillant davantage et en rendant son implémentation plus facile. On leur attribue parfois à tort la découverte de cette méthode.

Dans ce chapitre, nous allons utiliser la méthode de fusion unidirectionnelle de Lee et Schachter (détaillée par Guibas et Stolfi) et la généraliser à des fusions bidirectionnelles. Nous donnerons ensuite la complexité dans le pire des cas du processus de fusion et l'exprimerons plus finement en fonction du nombre de sites inachevés.

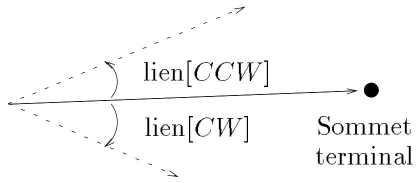


Figure 3.1: *Un brin, structure sous-tendant la "carte planaire".*

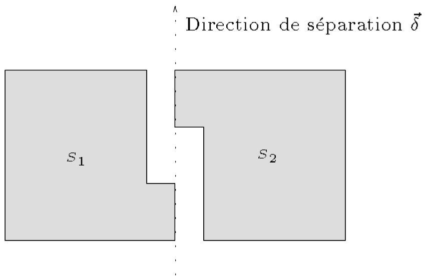


Figure 3.2: *Notion de séparabilité.*

3.1 Structure de Données

Les algorithmes, présentés dans ce mémoire, requièrent des structures de données très souples, permettant une "navigation" aisée dans le graphe de la triangulation. A partir d'un arc, on doit pouvoir accéder rapidement à ses deux extrémités, à son arc inverse ainsi qu'aux arcs voisins par rapport à l'ordre polaire autour de ses extrémités. La structure de "carte planaire" ([56]), proche de celle d'arête ailée ([11]), aussi compacte et légèrement plus souple, a été choisie pour représenter les graphes : un sommet est représenté par un couple de coordonnées ; un "brin" (de la carte planaire) comprend un pointeur sur son sommet terminal et (un tableau de) deux pointeurs sur les brins suivant et précédent (selon l'ordre polaire) autour de son propre sommet origine (implicite) (Figure 3.1). Les brins sont stockés dans un tableau de sorte qu'un brin et son inverse aient des positions symétriques par rapport au milieu du tableau, ce qui dispense de tout lien explicite de dualité.

3.2 Fusion de deux triangulations de Delaunay linéairement séparables

3.2.1 Notion de séparabilité

On dit que deux triangulations $DT(S_1)$ et $DT(S_2)$ sont *linéairement séparables* lorsqu'elles se trouvent chacune de part et d'autre d'une droite de séparation δ. On généralise cette définition en autorisant des sites de chaque triangulation à se trouver sur la droite δ. On a alors besoin d'une orientation sur la droite de séparation. Les sites de S_1 qui se trouvent sur la droite orientée $\vec{\delta}$ doivent être en dessous des sites de S_2 qui se trouvent sur la droite $\vec{\delta}$ (voir Figure 3.2).

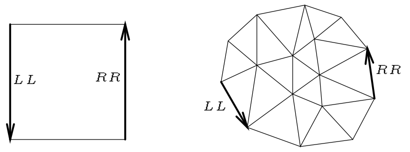


Figure 3.3: *Les deux brins extrêmes d'une triangulation.*

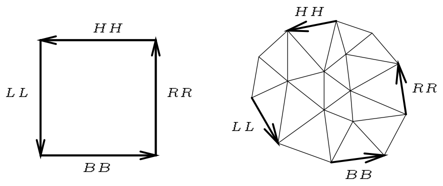


Figure 3.4: *Les quatre brins extrêmes d'une triangulation.*

3.2.2 Les arêtes extrêmes d'une triangulation

Pour fusionner deux triangulations linéairement séparables, on a besoin du site de chaque sous-triangulation le plus proche de la droite de séparation. Pour fixer les idées, on se donne un repère $(O, \overrightarrow{Ox}, \overrightarrow{Oy})$ et on suppose que la droite de séparation est parallèle et de même sens que $\overrightarrow{Oy}$. Pour accéder à ce sommet en temps constant, on garde en mémoire les deux brins extrêmes de chaque sous-triangulation, *i.e.* l'arc orienté d'enveloppe convexe issu de chaque sommet extremum pour la relation d'ordre lexicographique[1] (*cf.* Figure 3.3). On appelle ces brins extrêmes les arêtes extrêmes gauche et droite.

3.2.3 Algorithme de fusion

3.2.3.1 Recherche du "pont" inférieur

Pour fusionner deux triangulations $DT(L)$ et $DT(R)$ linéairement séparables par une droite δ, on cherche d'abord l'arête AB la plus basse de l'enveloppe convexe de $L \cup R$ qui coupe la droite δ. Pour cela, on part des sites de L et de R qui sont les plus proches de la droite δ. Ensuite, on se déplace dans le sens rétrograde le long de l'enveloppe convexe de L et dans le sens direct le long de l'enveloppe convexe de R jusqu'à ce que les extrémités de l'arête de l'enveloppe convexe la plus basse coupant δ soient trouvées. Cette arête permet d'initialiser le processus de fusion.

Remarque : Edelsbrünner a montré qu'il est plus efficace, pour trouver ce pont inférieur, de partir des sites les plus proches de la droite de séparation que des sites les plus bas de chaque triangulation (page 146, [74]).

[1]Un site $s_i(x_i, y_i)$ est inférieur lexicographiquement à un site $s_j(x_j, y_j)$ si et seulement si $x_i < x_j$ ou ($x_i = x_j$ *et* $y_i < y_j$).

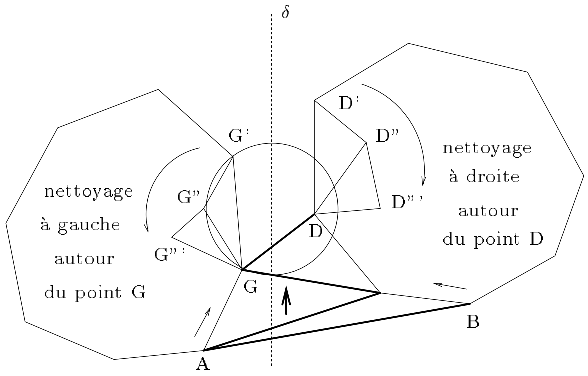


Figure 3.5: *Fusion de deux triangulations linéairement séparables.*

3.2.3.2 Fusion des deux triangulations

Une fois que le "pont" inférieur (AB sur la Figure 3.5) a été trouvé, les nouvelles arêtes sont créées dans l'ordre selon lequel elles coupent $\vec{\delta}$ jusqu'à la création de l'arête de l'enveloppe convexe la plus haute ("pont" supérieur) coupant δ. Les arêtes de $DT(L)$ et $DT(R)$ qui coupent les arêtes créées pendant la fusion, sont détruites. Plus précisément, dès qu'une arête GD est créée, on réalise les opérations suivantes (voir Figure 3.5) :

(a) *nettoyage à gauche autour du point G, dans le sens direct :* tant que G' est situé au-dessus de GD et que D est à l'intérieur de $\bigcirc(G, G', G'')$ – G'' étant le successeur de G' par rapport à l'ordre polaire direct autour de G –, on élimine l'arête GG' puis on fait $G' = G''$.

(b) *nettoyage à droite autour du point D, dans le sens rétrograde :* tant que D' est situé au-dessus de GD et que G est à l'intérieur de $\bigcirc(D, D', D'')$ – D'' étant le successeur de D' par rapport à l'ordre polaire rétrograde autour de D –, on élimine l'arête DD' puis on fait $D' = D''$.

(c) *création d'une nouvelle arête :* si G' et D' sont au-dessus de GD, on choisit dans le quadrilatère $GG'D'D$ résultant des nettoyages précédents, la diagonale satisfaisant le critère du cercle vide ("au hasard" si $GG'D'D$ est inscriptible dans un cercle), et ce choix induit la création d'une nouvelle arête (GD' ou $G'D$) qui devient la nouvelle GD dans l'itération suivante. Sinon, on a atteint le "pont" supérieur et la fusion est terminée.

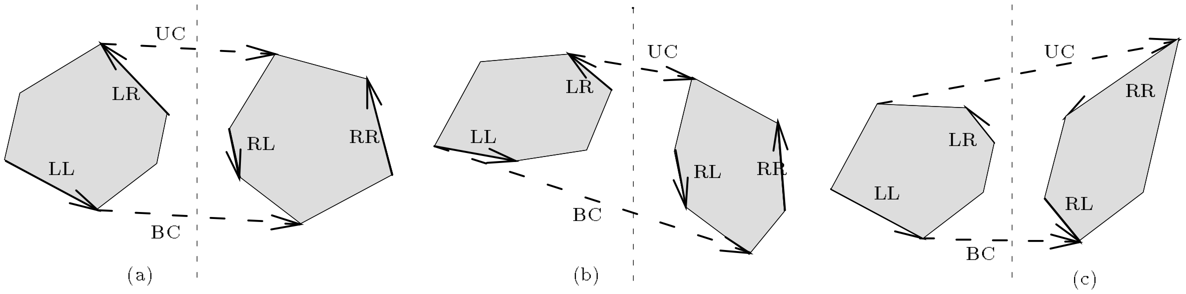


Figure 3.6: *Mise à jour des arêtes extrêmes gauche et droite : (a) cas général, LL et RR deviennent les deux arêtes extrêmes, (b) BC le "pont inférieur" devient l'arête extrême gauche, (c) UC l'inverse du "pont supérieur" devient l'arête extrême droite.*

3.2.3.3 Mise à jour des arêtes extrêmes gauche et droite

Pour pouvoir enchaîner plusieurs fusions, il faut mettre à jour, dans la triangulation résultante, les arêtes extrêmes gauche et droite. Cette opération se fait *en temps constant.* Soit LL et LR (resp. RL et RR) les arêtes extrêmes gauche et droite de la triangulation gauche (resp. droite).

Dès que l'on a créé le "pont" inférieur BC (voir Figure 3.6), on teste si l'origine de BC est égale à l'origine de LL. Si l'égalité est avérée, BC devient la nouvelle arête extrême gauche, sinon celle-ci reste égale à LL.

Dès que l'on a créé le "pont" supérieur UC, on teste si l'extrémité de UC est égale à l'origine de RR. Si l'égalité est avérée, l'inverse de UC devient la nouvelle arête extrême gauche, sinon celle-ci reste égale à RR.

3.3 Fusion bidirectionnelle

3.3.1 Principe

On a vu précédemment que la procédure de fusion de Lee et Schachter est très générale et fonctionne de la même façon quelle que soit la direction de la droite de séparation entre les deux triangulations à fusionner. Pour pouvoir effectuer des fusions bidirectionnelles, il suffit de garder en mémoire pour chaque sous-triangulation les deux arêtes extrêmes par rapport à chaque droite de séparation, soit quatre arêtes en tout. Pour fixer les idées, on supposera que les droites de séparation sont parallèles et de même sens que les axes du repère $\overrightarrow{Ox}$ et $\overrightarrow{Oy}$. On appellera arêtes extrêmes gauche et droite les arêtes extrêmes par rapport à la direction selon laquelle on est en train d'effectuer la

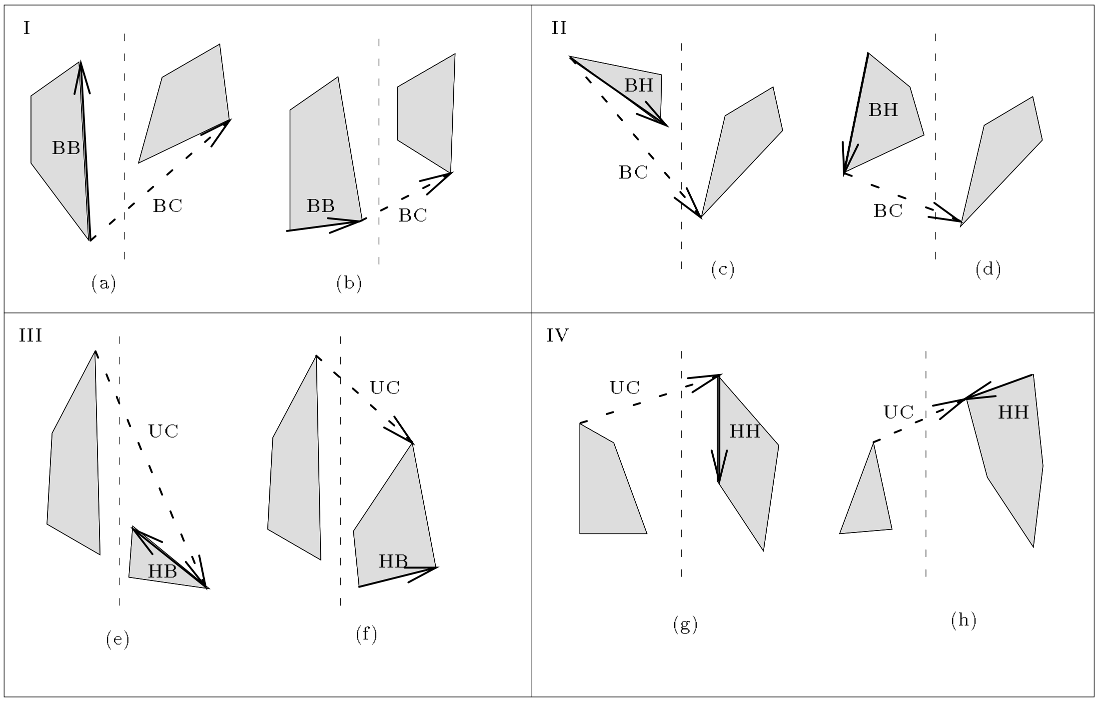


Figure 3.7: *Mise à jour des arêtes extrêmes haut et bas :*
(I) le site le plus bas est à gauche, l'arête extrême inférieure reste BB (b) ou devient égale au "pont" inférieur BC (a)
(II) le site le plus haut est à gauche, l'arête extrême supérieure reste BH (d) ou devient égale au "pont" inférieur BC (c)
(III) le site le plus bas est à droite, l'arête extrême inférieure reste HB (f) ou devient égale à l'inverse du "pont" supérieur UC (e)
(IV) le site le plus haut est à droite, l'arête extrême supérieure reste HH (h) ou devient égale à l'inverse du "pont" supérieur UC (g).

fusion, et arêtes extrêmes haut et bas les arêtes extrêmes par rapport à l'*autre* direction de séparation.

3.3.2 Mise à jour des arêtes extrêmes

Les arêtes extrêmes gauche et droite se mettent à jour comme précédemment. Soient BB et BH (resp. HB et HH) les arêtes extrêmes basses et hautes de la triangulation gauche (resp. droite) (voir Figure 3.7). On peut aussi les mettre à jour *en temps constant*, la méthode étant un peu plus complexe :

dès que l'on a créé le "pont" inférieur BC, on distingue deux cas :

1. si le site le plus bas appartient à la triangulation de gauche, on teste si BC et BB ont même origine. Si c'est le cas, l'arête extrême bas devient BC, sinon celle-ci reste égale à BB.

2. si le site le plus haut appartient à la triangulation de gauche, on teste si BC et BH ont même origine. Si c'est le cas, l'arête extrême haut devient BC, sinon celle-ci reste égale à BH.

dès que l'on a créé le "pont" supérieur UC, on distingue deux cas :

1. si le site le plus bas appartient à la triangulation de droite, on teste si l'extrémité de UC est égale à l'origine de HB. Si c'est le cas, l'arête extrême bas devient l'inverse de UC, sinon celle-ci reste égale à HB.

2. si le site le plus haut appartient à la triangulation de droite, on teste si l'extrémité de UC est égale à l'origine de HH. Si c'est le cas, l'arête extrême haut devient l'inverse de UC, sinon celle-ci reste égale à HH.

Remarque : avant l'appel de la procédure de fusion bidirectionnelle, on peut regarder dans quelle triangulation se trouve le site le plus bas et le site le plus haut et stocker ces informations dans une variable quadrivalente. Ainsi, la procédure de fusion bidirectionnelle devient indépendante du repère, comme dans le cas unidirectionnel.

3.4 Analyse de la complexité

Théorème 3.4.1 *([100]) Soient L et R deux ensembles de points situés respectivement à gauche et à droite d'une droite δ. Lors de la fusion de $DT(L)$ et $DT(R)$ pour construire $DT(L \cup R)$:*

(a) seules des arêtes reliant deux sites de L et des arêtes reliant deux sites de R sont détruites.

(b) seules des arêtes reliant un site de L à un site de R sont créées.

(c) la complexité dans le pire des cas du processus de fusion est bornée par une fonction linéaire de la somme des trois composants suivants :

1. *le nombre de sites examinés pour trouver les extrémités de l'arête la plus basse de l'enveloppe convexe de $(L \cup R)$ qui coupe δ.*
2. *le nombre d'arêtes détruites.*
3. *le nombre d'arêtes créées.*

(d) la complexité dans le pire des cas du processus de fusion est en $O(|L \cup R|)$.

Dém : voir Guibas et Stolfi [100]. $\diamond$

Lemme 3.4.1 *([70]) Toute triangulation d'un ensemble de n sites dont k (sites) appartiennent à l'enveloppe convexe, possède $e = 3n - k - 3$ arêtes et $t = 2n - k - 2$ triangles. En particulier, au plus $3n - 6$ arêtes peuvent être créées sur un ensemble de n sites.*

Dém : soient t le nombre de triangles et e le nombre d'arêtes. Toutes les arêtes appartiennent à deux triangles sauf les k arêtes de l'enveloppe convexe, d'où $2e - k = 3t$. En utilisant la formule d'Euler $n - e + (t + 1) = 2$, on en déduit t et e. En faisant $k = 3$, on déduit le dernier résultat. $\diamond$

Théorème 3.4.2 *([70]) La complexité dans le pire des cas du processus de fusion est majorée par une fonction linéaire du nombre de sites qui reçoivent de nouvelles arêtes.*

Dém : grâce au Théorème 3.4.1, il suffit de montrer que, si de nouvelles arêtes sont ancrées sur r sites, alors au plus $3r - 6$ arêtes sont créées, au plus $3r - 9$ arêtes sont détruites, et au plus r sites sont examinés pour construire l'arête la plus basse de l'enveloppe convexe de $L \cup R$ qui coupe δ.

Soient c et d le nombre d'arêtes créées et détruites. Puisque les arêtes créées ne se coupent pas, en raison du Lemme 3.4.1, on a $c \leq 3r - 6$.

Supposons que L, R et $L \cup R$ soient composés respectivement de n_1, n_2 et n sites, dont k_1, k_2 et k (sites) appartiennent à l'enveloppe convexe. Nous avons grâce au Lemme 3.4.1 :

$$(3n_1 - k_1 - 3) + (3n_2 - k_2 - 3) + (c - d) = (3n - k - 3)$$

$$\Rightarrow k - (k_1 + k_2) = 3 - c + d \text{ puisque } n = n_1 + n_2$$

Comme $k \leq k_1 + k_2$, $3 - c + d \leq 0 \Rightarrow d \leq c - 3$

Comme $c \leq 3r - 6$, on déduit $d \leq 3r - 9$

Enfin, nous observons que chaque site examiné pour trouver l'arête la plus basse de l'enveloppe convexe de $L \cup R$ qui coupe δ, reçoit une nouvelle arête pendant la fusion, donc leur nombre est inférieur ou égal à r. $\diamond$

Nous allons maintenant évaluer la complexité du processus de fusion en fonction du nombre de sites inachevés.

Définition 3.4.1 *Un site s est* achevé[2] *dans $DT(S_1)$ relativement à $DT(S_2)$ ($S_1 \subseteq S_2$) si et seulement si la liste d'adjacence de s est identique dans les deux triangulations (définition valable dans le cas où S_1 a une triangulation de Delaunay unique), sinon, il est* inachevé*.*

Dans le paragraphe 6.2 du Chapitre 6, on trouvera une définition en dimension quelconque ainsi que quelques propriétés (voir aussi Figure 6.1 du Chapitre 6).

Corollaire 3.4.1 *La complexité dans le pire des cas de la fusion de L et R est bornée par $7m$ où m est le nombre de sites inachevés dans L et dans R.*

Dém : si un site reçoit de nouvelles arêtes, il est inachevé, d'où $r \leq m$, r étant le nombre de sites qui recoivent de nouvelles arêtes et m le nombre de sites inachevés dans $DT(L)$ et dans $DT(R)$ relativement à $DT(L \cup R)$. En utilisant les majorations de la preuve du théorème 3.4.2, on déduit que le nombre total d'arêtes créées est borné par $3m$, le nombre total d'arêtes détruites est borné par $3m$, le nombre total de sites examinés pour trouver l'arête la plus basse de l'enveloppe convexe de $L \cup R$ qui coupe δ, est borné par m. Le théorème 3.4.1.c permet de conclure. $\diamond$

Pour pouvoir enchaîner plusieurs fusions, il est nécessaire d'avoir les deux arêtes extrêmes pour une fusion unidirectionnelle ou les quatre arêtes extrêmes pour une fusion bidirectionnelle. On a vu que, dans les deux cas, on peut mettre à jour en temps constant ces arêtes après une fusion. En conclusion, quel que soit le type de fusion (unidirectionnelle ou bidirectionnelle), sa complexité est **linéaire par rapport au nombre de sites inachevés**.

[2]Ce concept a été introduit dans le plan par Katajainen et Koppinen [117].

Chapitre 4

LE PARADIGME *DIVIDE AND CONQUER*

Le paradigme *Divide and Conquer* est très utilisé en Géométrie Algorithmique (enveloppe convexe, triangulation de Delaunay, diagramme de Voronoi, la paire la plus proche...) et même en algorithmique en général. Il permet de traiter efficacement des ensembles de données très vastes avec une complexité raisonnable. Cette approche consiste à diviser récursivement un problème en plusieurs sous-problèmes de plus petite taille, à résoudre les sous-problèmes et à les combiner pour obtenir la solution du problème initial. Cette méthode présente aussi d'autres avantages : elle se prête naturellement aux algorithmes parallèles, ainsi qu'à ceux qui utilisent une mémoire externe ; quand le *Divide and Conquer* est randomisé, on peut le dérandomiser[1] et obtenir un algorithme déterministe avec une complexité dans le pire des cas égale à la complexité en moyenne de son homologue randomisé.

4.1 Description

Le schéma général est le suivant :

Fonction Traiter(Ensemble)
    Si CasTrivial(Ensemble) **alors**
        Renvoyer(TraitementCasTrivial(Ensemble))
    Sinon
        (EnsembleA_1,EnsembleA_2,...,EnsembleA_k) = Partition(Ensemble)

[1] La théorie de la dérandomisation [6] fait appel à des notions mathématiques extrêmement sophistiquées (méthode des probabilités conditionnelles, ϵ-nets et ϵ-approximations, dimension de Vapnik-Chervonenkis, cuttings...). Elle a, à ce jour, un intérêt essentiellement théorique, il n'existe pas d'utilisation pratique. Cette théorie est nouvelle et constitue un important sujet de recherche (voir l'excellent "survol" de J. Matoušek [141], intégré dans l'ouvrage *Handbook on Computational Geometry* de Sack et Urrutia [170]).

$$\begin{aligned}
&\text{Resultat}A_1 = \text{Traiter}(\text{Ensemble}A_1)\\
&\text{Resultat}A_2 = \text{Traiter}(\text{Ensemble}A_2)\\
&\qquad\vdots \qquad\qquad\qquad \vdots\\
&\text{Resultat}A_k = \text{Traiter}(\text{Ensemble}A_k)\\
&\text{Renvoyer}(\text{Fusion}(\text{Resultat}A_1,\text{Resultat}A_2,\ldots,\text{Resultat}A_k))
\end{aligned}$$

CasTrivial : cas où l'ensemble se réduit à un nombre maximum donné (petit) d'éléments. Le **TraitementCasTrivial** s'en déduit aisément.

Partition : coupe l'ensemble en plusieurs sous-ensembles (en général 2) de tailles sensiblement équivalentes.

Fusion : réunit les résultats de plusieurs traitements séparés (en général 2) en un seul résultat global. C'est parfois cette opération qui supporte la plupart des opérations du traitement, et qui est la plus complexe à réaliser, par exemple dans le cas de la triangulation de Delaunay (voir chapitre 3).

On distingue trois phases : *diviser* le problème en sous-problèmes, *traiter* les sous-problèmes et *combiner* les résultats des sous-problèmes.

4.2 Exemple

Considérons le problème du tri de la liste de nombres 3, 8, 1, 7, 4, 2, 9, 6, 5, 0 avec un algorithme du type *Divide and Conquer*. Il y a au moins deux approches différentes possibles [168] :

A. diviser la liste en deux listes 3, 8, 1, 7, 4 et 2, 9, 6, 5, 0; trier récursivement chaque liste pour obtenir 1, 3, 4, 7, 8 et 0, 2, 5, 6, 9; puis fusionner ces listes pour obtenir la liste triée finale 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

B. trouver une bonne partition de la liste en deux listes à peu près de la même taille et telles que tous les éléments de la première liste soient plus petits que les éléments de la seconde, soit 3, 1, 4, 2, 0 et 8, 7, 9, 6, 5; trier récursivement chaque liste pour obtenir 0, 1, 2, 3, 4 et 5, 6, 7, 8, 9; puis fusionner ces listes pour obtenir la liste triée finale 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Les deux principales caractéristiques de ces approches sont : (1) chacune des sous-listes est plus petite que la liste originale (avec un certain facteur, 1/2 dans l'exemple) et (2) la somme des tailles des sous-listes est égale à la taille de la liste originale. Dans l'approche A, l'étape de division est facile mais l'étape de fusion est difficile. Au contraire, dans l'approche B, l'étape de division est difficile mais l'étape de fusion est triviale. On aura reconnu dans la méthode A le classique tri par fusion (*mergesort*) et dans la méthode B le tri rapide (*quicksort*). C'est la méthode B qui se généralise à beaucoup de problèmes géométriques[2], par exemple triangulation de Delaunay (voir chapitre 3), enveloppe convexe... Souvent, la condition (2) se change en une borne fonction de la taille du problème original et on peut aussi procéder de façon plus générale à des k-divisions plutôt qu'à de simples divisions par deux.

4.3 Notations asymptotiques

Pour mesurer l'efficacité d'un algorithme, on évalue l'ordre de grandeur du nombre d'opérations qu'il effectue lorsque la taille du problème qu'il résout augmente. On mesure ainsi une efficacité asymptotique (on a vu dans l'Introduction pourquoi cette méthode doit être utilisée avec discernement). On obtient ainsi une fonction que l'on appelle complexité de l'algorithme. Les notations de Landau (avec quelques abus d'écriture) sont un moyen pratique d'exprimer cet ordre de grandeur. Par commodité, on écrit $f \in O(x^2)$ si $f \in O(g)$ et $g : x \longmapsto x^2$. Par convention, on écrit $f = O(x^2)$ si $f \in O(x^2)$. Etant donné que l'on évalue des complexités d'algorithmes, les comparaisons de fonction se font au voisinage de $+\infty$.

La notation $O()$ représente un majorant (par exemple, le tri par fusion est dans $O(n^4)$ mais aussi dans $O(n \log n)$). La notation $\Omega()$ représente un minorant (par exemple, le tri par fusion est dans $\Omega(n)$ mais aussi dans $\Omega(n \log n)$). La notation $\Theta()$ représente une valeur exacte, c'est à la fois un majorant et un minorant (par exemple, le tri par fusion est dans $\Theta(n \log n)$). Il existe deux autres notations qui sont moins utilisées : la notation $o()$ représente un majorant jamais atteint (par exemple, le tri par fusion est dans $o(n^4)$, mais pas dans $o(n \log n)$) ; la notation $\omega()$ représente un minorant jamais atteint (par exemple, le tri par fusion est dans $\omega(n)$, mais pas dans $\omega(n \log n)$). Ces deux symboles servent à comparer des majorants ou minorants.

On trouve des développements plus mathématiques sur ces notations asymptotiques et leurs propriétés dans l'ouvrage de Cormen, Leiserson et Rivest [57] et aussi dans celui de Beauquier, Berstel et Chrétienne [13] (ainsi d'ailleurs que dans tout bon livre d'algorithmique).

Lors de l'analyse d'un algorithme, on détermine sa complexité dans le pire des cas, soit son comportement dans le cas le plus défavorable. Pour certains algorithmes, le cas le plus défavorable a une probabilité d'apparition presque nulle (complexité dans le pire des cas du *quicksort* dans $O(n^2)$ par exemple). Ils sont ainsi fortement pénalisés par ce type d'analyse. Souvent, on préfère calculer une complexité en moyenne qui reflète

[2]Dans les algorithmes géométriques, on divise généralement les données selon une relation d'ordre, en deux parties de même taille, mais l'étape de fusion est souvent beaucoup plus complexe que dans l'exemple du quicksort (triangulation de Delaunay par exemple) où il suffit de concaténer deux listes !

mieux le comportement réel de l'algorithme (complexité en moyenne du *quicksort* dans $O(n \log n)$). Dans certaines analyses en moyenne, on fait même en plus des hypothèses sur la distribution des données. Ainsi, il existe des algorithmes de tri linéaires en moyenne [57].

4.4 Analyse de la complexité

L'analyse de la complexité d'un algorithme du type *Divide and Conquer* se ramène en général à la résolution d'équations de récurrence de la forme suivante :

$$\begin{cases} t(n_0) &= d \\ t(n) &= at(\frac{n}{b}) + f(n) \quad n > n_0 \end{cases}$$

On suppose que l'on remplace un problème de taille n par a sous-problèmes, chacun de taille $\frac{n}{b}$. Si $t(n)$ est le coût de l'algorithme pour la taille n, il se compose donc de $at(\frac{n}{b})$ plus le temps $f(n)$ pour recombiner les solutions des problèmes partiels en une solution du problème total. d est le coût pour un problème élémentaire de petite taille n_0, pour lequel la récursion s'arrête. En général, les équations ci-dessus sont en fait des inéquations. Le coût $\tau(n)$ de l'algorithme vérifie :

$$\begin{cases} \tau(n) &\leq d & n \leq n_0 \\ \tau(n) &\leq a\tau(\frac{n}{b}) + f(n) & n > n_0 \end{cases}$$

Il en résulte que $\tau(n) \leq t(n)$ pour tout n. La fonction t constitue donc une majoration du coût de l'algorithme.

Théorème 4.4.1 *[13] Soit $t : \mathbb{N} \longrightarrow \mathbb{R}_+$ une fonction croissante au sens large à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $a > 0$, et $d > 0$ et une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ pour lesquels*

$$\begin{cases} t(n_0) &= d \\ t(n) &= at(\frac{n}{b}) + f(n) \quad n > n_0 \text{ , } \frac{n}{n_0} \text{ une puissance de } b \end{cases}$$

Supposons de plus que $f(n) = cn^k(\log_b n)^q$ pour des réels $c > 0$, $k \geq 0$ et q. Alors

$$t(n) = \begin{cases} \theta(n^k) & si\ a < b^k\ et\ q = 0 \\ \theta(n^k(\log_b n)^{1+q}) & si\ a = b^k\ et\ q > -1 \\ \theta(n^k(\log\log_b n)) & si\ a = b^k\ et\ q = -1 \\ \theta(n^{\log_b a}) & si\ a = b^k\ et\ q < -1 \\ \theta(n^{\log_b a}) & si\ a > b^k \end{cases}$$

Dém : voir [13]. ◇

Remarque : on trouve dans l'ouvrage de Beauquier, Berstel et Chrétienne [13] davantage de résultats.

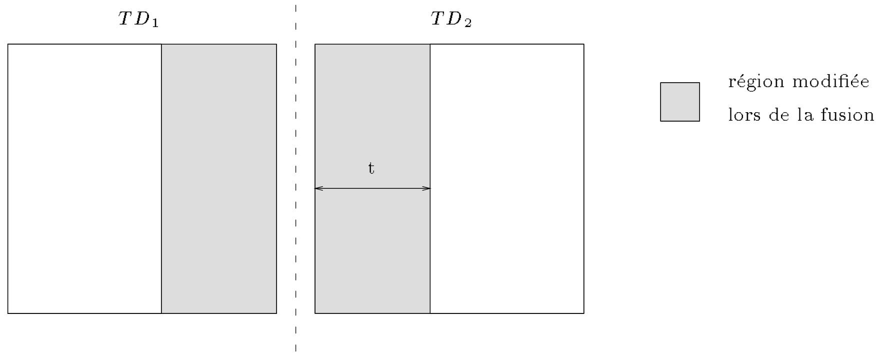


Figure 4.1: *Régions modifiées par une fusion.*

Corollaire 4.4.1 *Si l'on peut remplacer récursivement le problème de taille n par deux problèmes de taille $\frac{n}{2}$ et si la fonction f qui permet de fusionner les solutions des deux problèmes partiels en la solution du problème origine est de la forme $f(n) = cn^k(\log_2 n)^q$ avec $k < 1$, alors l'algorithme de résolution du problème est linéaire.*

Dém :

$$\left\{\begin{array}{l} k < 1 \\ a = b = 2 \end{array}\right. \Longrightarrow a > b^k \Longrightarrow t(n) = \theta(n^{\log_2 2}) = \theta(n)$$

d'après le théorème 4.4.1. ◇

4.5 Application à la triangulation de Delaunay

Ce paragraphe constitue une introduction au chapitre 5 qui présente de nouveaux algorithmes de triangulation de Delaunay. Les justifications qui suivent, n'ont pas de valeur mathématique, mais permettent de comprendre intuitivement les résultats théoriques démontrés dans les chapitres suivants.

4.5.1 Amélioration de la complexité en moyenne

Si l'on observe la fusion de deux sous-triangulations DT_1 et DT_2 séparables par une droite (voir Figure 3.5), on s'aperçoit que cette procédure ne modifie en général que les arêtes proches de l'enveloppe convexe des deux sous-triangulations. Pratiquement toutes les arêtes situées au coeur de DT_1 et de DT_2 restent intactes. Rappelons que, dans l'hypothèse d'une distribution uniforme de n sites, le nombre d'arêtes de l'enveloppe convexe ([165] page 151) d'un carré est proportionnel à $\log n$, celui d'un cercle proportionnel à $\sqrt[3]{n}$. Plus généralement, le nombre de sites à la distance maximale constante t de la frontière d'un carré, est proportionnel à $\sqrt[2]{n}$. Donc, en admettant que la complexité de la fusion des deux sous-triangulations soit proportionnelle à $\sqrt[2]{n}$, le calcul de toute la triangulation prendra un temps linéaire d'après le Corollaire 4.4.1 (faire $k = 1/2$ et $q = 0$). C'est cette constatation intuitive qui nous a fait concevoir les algorithmes présentés dans le chapitre 5, dans lequel on essaie de découper le domaine avec des cellules de la forme la plus *carrée* possible. Une analyse rigoureuse de la complexité est faite dans les chapitres 5 et 6. Les vérifications expérimentales sont faites dans le chapitre 7.

4.5.2 Analyse en moyenne de l'algorithme de Lee et Schachter

On se place dans l'hypothèse d'une distribution uniforme dans un carré unitaire. Dans l'algorithme de Lee et Schachter [131], on divise récursivement le domaine en bandes verticales. Une étape de division consiste à couper selon le sens vertical toutes les bandes existantes. le processus s'arrête quand il ne reste plus qu'un point par bande, les bandes auront alors une largeur de l'ordre de $\frac{1}{n}$. Il y a $(\log n)$ étapes de division. Au bout de $\frac{\log n}{2}$ étapes, les bandes auront une largeur d'environ $\frac{1}{\sqrt{n}}$. Donc toutes les bandes, dans la moitié inférieure de l'arborescence de division, auront une largeur inférieure ou égale à $\frac{1}{\sqrt{n}}$.

La longueur moyenne d'une arête de Delaunay, avec les hypothèses ci-dessus, est légèrement supérieure à $\frac{1}{\sqrt{n}}$. Il n'est pas choquant de supposer que, lors de la fusion de deux sous-triangulations DT_1 et DT_2, toute la partie située à une distance inférieure à $\frac{1}{\sqrt{n}}$ de la droite de séparation de DT_1 et DT_2, sera entièrement modifiée. En particulier, toutes les bandes de largeur inférieure ou égale à $\frac{1}{\sqrt{n}}$ seront *entièrement* modifiées.

Calculons le travail nécessaire pour effectuer toutes les fusions de la moitié inférieure de l'arborescence. Il y a $\frac{\log n}{2}$ niveaux et le travail pour chaque niveau est proportionnel à n car les bandes sont entièrement modifiées. Le travail total sera au moins en $n \times \frac{\log n}{2} = \Omega(n \log n)$.

Ohya, Iri et Murota [158] démontrent que l'ensemble des processus de fusion dans l'algorithme de Lee et Schachter est *en moyenne* en $\Omega(n \log n)$, même dans l'hypothèse d'une distribution uniforme. Les résultats expérimentaux du chapitre 7 le confirment.

Chapitre 5

NOUVEAUX ALGORITHMES DE TRIANGULATION DE DELAUNAY

> *"Un bon algorithme est comme un couteau tranchant – il fait exactement ce qu'on attend de lui avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire" (Cormen, Leiserson et Rivest) [57].*

Comme le faisait remarquer T. Lambert dans sa thèse [126], il existe une multitude d'algorithmes de triangulation de Delaunay qui ne présentent parfois que des différences microscopiques. Nous proposons dans ce chapitre une amélioration importante de l'algorithme de Lee et Schachter (en pratique 2 à 3 fois plus rapide) et en donnons plusieurs variantes. Dans le chapitre suivant, ces nouveaux algorithmes seront comparés avec les meilleurs algorithmes de triangulation de Delaunay, d'après l'étude comparative de Su et Drysdale [187] [188] faite en 1995.

5.1 Algorithme de Lee et Schachter

5.1.1 Description

Cet algorithme [131] a déjà été évoqué dans le paragraphe 1.4.6. Il calcule une triangulation de Delaunay avec une méthodologie du type "Divide-and-Conquer". Il faut, dans un premier temps, trier les sites selon l'ordre lexicographique (selon les x croissants, et en cas d'égalité, selon les y croissants). L'ensemble des points triés est alors

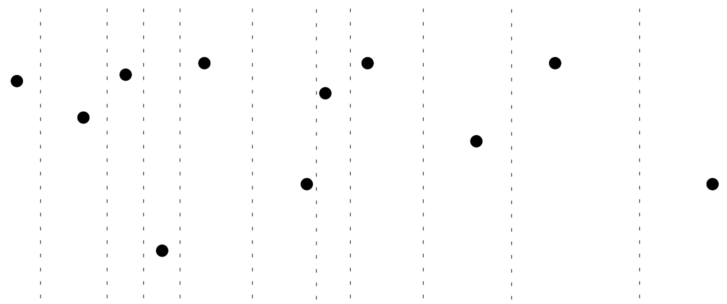


Figure 5.1: *Division en bandes élémentaires.*

divisé récursivement en deux parties séparables par une droite verticale[1], et de même taille (à une unité près) jusqu'à ce que les sous-ensembles ne contiennent qu'un seul site (Figure 5.1). Les bandes élémentaires adjacentes sont alors fusionnées par paires de telle sorte que la triangulation des points contenus dans chaque bande résultante soit de Delaunay. L'algorithme de fusion a été détaillé dans le paragraphe 3.2. Ce processus est itéré jusqu'à obtention de la triangulation de Delaunay de l'ensemble S total (Figure 1.9).

5.1.2 Schéma de l'algorithme

On suppose que les sites ont été rangés dans un tableau sans point double lors d'une **Etape 0**; l et r représentent les extrémités du tableau courant dans chaque fonction, et l'on pose $\mu = \lceil (l+r)/2 \rceil$. LL ou RL (resp. LR ou RR) est l'arête extrême gauche (resp. droite) de la triangulation courante (voir Figure 5.2 (I)).

Etape 1 : Tri lexicographique du tableau de sites

Etape 2 : Triangulation

Fonction Delaunay(l, r)
  Coupe (l, r, LL, RR)
Fonction Coupe(l, r, LL, RR)
  Si $(l = r)$ **alors**
    $LL = RR = NULL$
  Sinon
    Coupe $(l, \mu - 1, LL, LR)$
    Coupe (μ, r, RL, RR)
    Fusionne1d (l, r, LL, LR, RL, RR)

[1]En fait, cette droite peut parfois être légèrement oblique pour séparer les points ayant même abscisse.

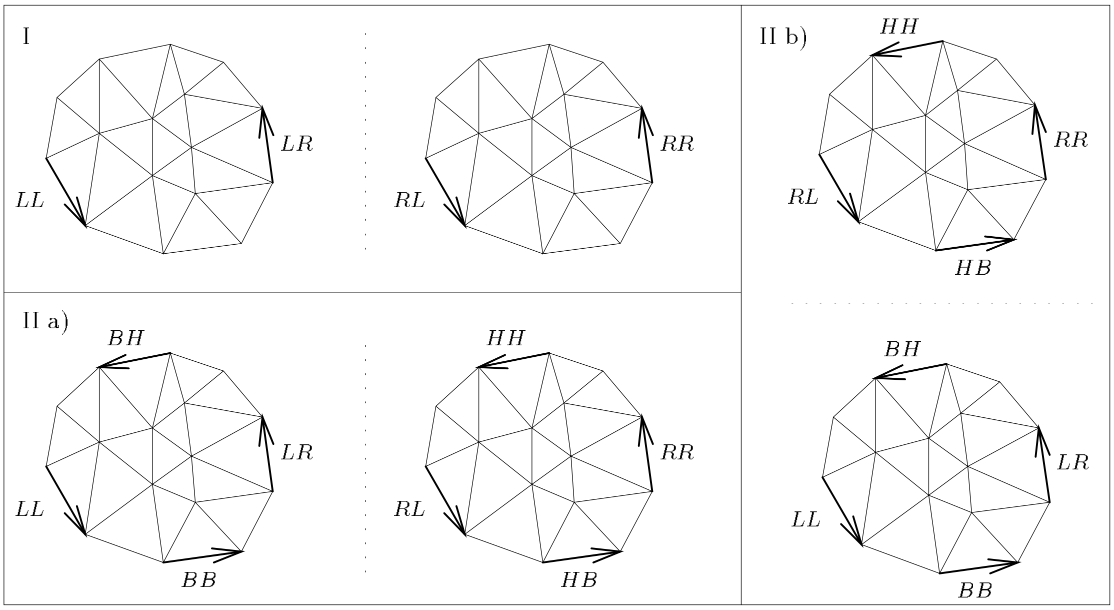


Figure 5.2: *Notations des arêtes extrêmes dans une fusion :*
(I) fusion monodirectionnelle
(II) fusion bidirectionnelle : a) horizontale, b) verticale.

5.1.3 Complexité

Elle est la somme de deux termes : le temps du tri et le temps de calcul de la triangulation.

5.1.3.1 Complexité dans le pire des cas

1. Le tri lexicographique [57] peut être effectué en $\Theta(n \log n)$.

2. La complexité d'une fusion, d'après le Théorème 3.4.1.d, est linéaire par rapport au nombre total de sites des deux triangulations à fusionner. En appliquant le Théorème 4.4.1 sur la complexité des algorithmes du type Divide-and-Conquer (avec $a = b = 2$, $k = 1$ et $q = 0$), on déduit que la complexité du processus complet de triangulation est en $O(n \log n)$. D'autre part, si l'on dispose n points sur une portion de spirale logarithmique, on peut montrer par un raisonnement quasi-identique à celui du paragraphe 5.2.3.1 que la complexité du processus complet de triangulation est en $\Omega(n \log n)$. La borne exacte est donc $\Theta(n \log n)$.

L'algorithme de Lee et Schachter est donc en $\Theta(n \log n)$.

5.1.3.2 Complexité en moyenne

On suppose que la distribution des points est uniforme.

1. Le tri lexicographique [57], puisque la distribution est uniforme, peut être effectué en temps moyen linéaire à l'aide d'un tri par paquets[2] par exemple ([57] page 177).

2. En ce qui concerne le processus de triangulation, Ohya, Iri et Murota ont montré [158] qu'il est, même en moyenne, en $\Omega(n \log n)$.

L'algorithme de Lee et Schachter est donc, *même en moyenne*, en $\Theta(n \log n)$.

5.2 Algorithme basé sur un 2d-tree

5.2.1 Description

Le découpage récursif des n sites selon une direction unique produit des bandes élémentaires très longues et très étroites. La fusion de ces bandes engendre un assez grand nombre de triangles qui n'ont pratiquement aucune chance d'appartenir à la triangulation finale, étant donné leur forme excessivement allongée (Figure 1.9). P. Volino [193] avait déjà fait cette remarque dans son mémoire de D.E.A.. Il suggérait alors d'effectuer un découpage croisé, mais sans trouver la structure de données permettant de le réaliser.

L'algorithme de Lee et Schachter perd beaucoup de temps, au fur et à mesure des fusions entre bandes adjacentes de même direction, à détruire et reconstruire des triangles. Ceci est fondamentalement dû au principe de découpage par bandes unidirectionnelles (voir la démonstration de Ohya, Iri et Murota [158] et aussi la justification intuitive du paragraphe 4.5.2).

Pour optimiser l'algorithme précédent, nous proposons de diviser le plan selon un autre processus, qui évite la création de bandes étroites : la division selon un 2d-tree (voir paragraphe 2.2). Les rectangles ainsi obtenus sont ensuite fusionnés par paires dans l'ordre inverse de leur création. Les sous-ensembles obtenus par X-division (*resp.* Y-division) sont fusionnés par X-fusion (*resp.* Y-fusion) (cf. Figure 5.4).

Un autre avantage de cet algorithme est de diminuer considérablement les risques en ce qui concerne l'imprécision numérique : en effet, on manipulera beaucoup moins souvent des triangles dont les extrémités sont quasiment alignées.

---

[2] C'est une technique de hachage.

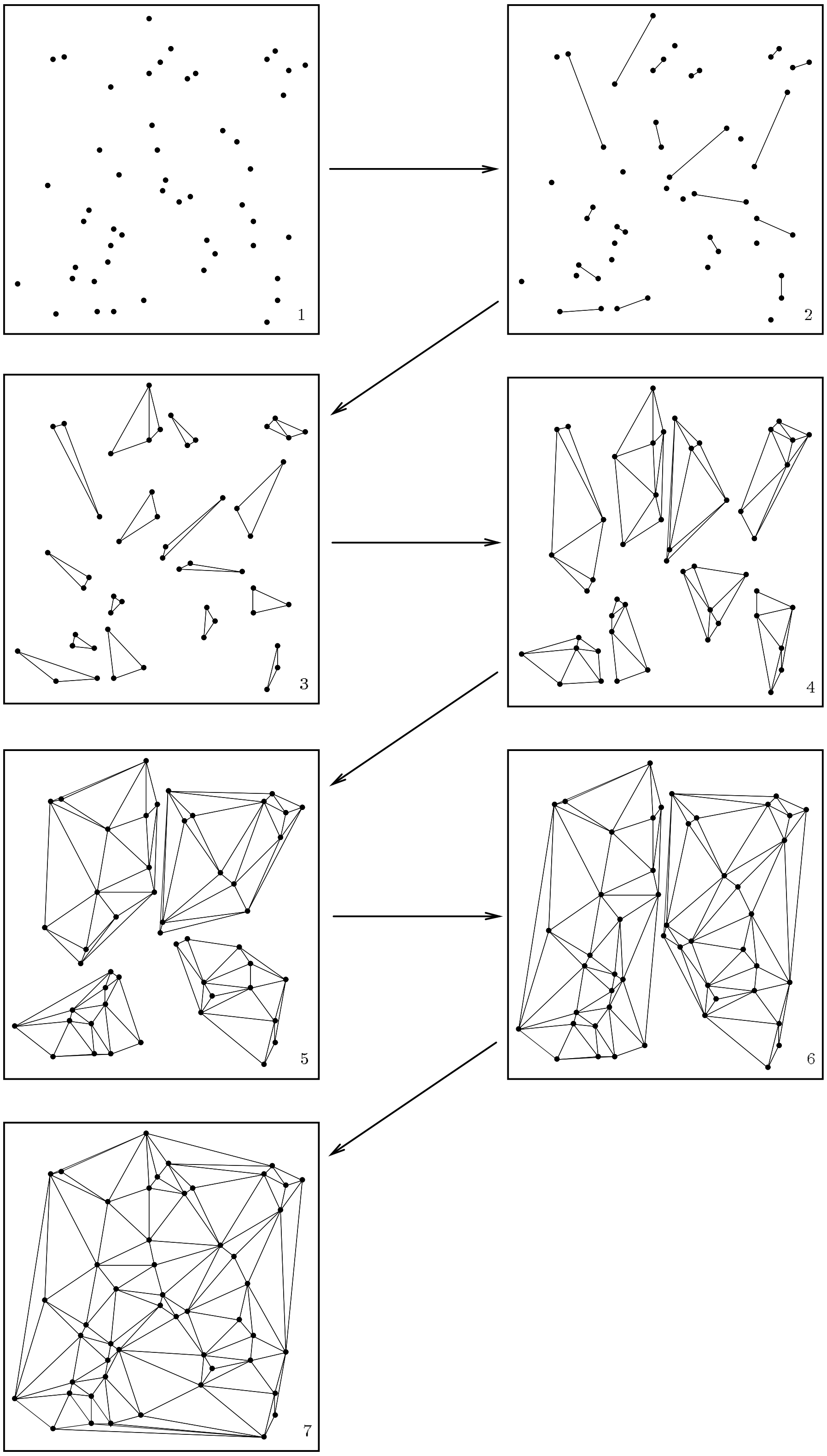


Figure 5.3: *Triangulation de Delaunay basée sur un 2d-tree.*

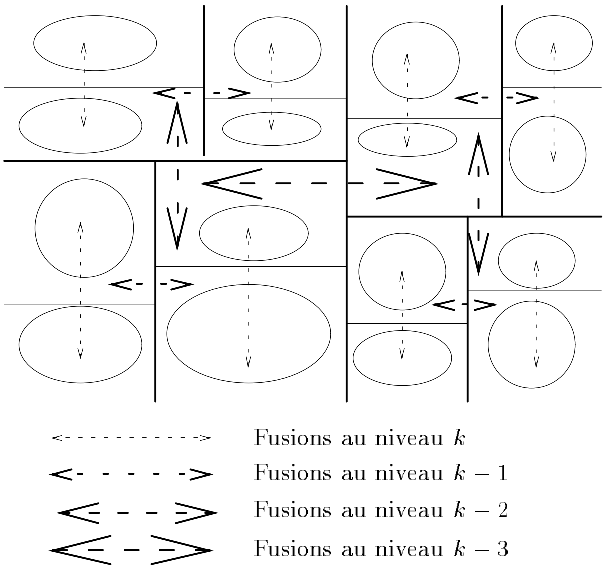


Figure 5.4: *Divisions et fusions selon un 2d-tree.*

5.2.2 Schéma de l'algorithme

On suppose que les sites ont été rangés dans un tableau sans point double lors d'une **Etape 0** ; l et r représentent les extrémités du tableau courant dans chaque fonction, et l'on pose $\mu = \lceil (l+r)/2 \rceil$. LL ou RL (resp. LR ou RR) est l'arête extrême gauche (resp. droite) de la triangulation courante. BB ou HB (resp. BH ou HH) est l'arête extrême basse (resp. haute) de la triangulation courante (voir Figure 5.2 (II)). *MergeStatus* est une variable quadrivalente qui indique la position relative des deux triangulations à fusionner (voir remarque paragraphe 3.3.2).

Etape 1 : Tri selon un 2d-tree du tableau de sites

Etape 2 : Triangulation

Fonction Delaunay(g, d)
  Xcoupe (l, r, LL, RR, BB, HH)
Fonction Xcoupe(l, r, LL, RR, BB, HH)
  Si $(l = r)$ **alors**
    $LL = RR = BB = HH = NULL$
  Sinon
    Ycoupe $(l, \mu - 1, LL, LR, BB, BH)$
    Ycoupe (μ, r, RL, RR, HB, HH)
    Fusionne2d $(MergeStatus, l, r, LL, LR, RL, RR, BB, BH, HB, HH)$
Fonction Ycoupe(l, r, LL, RR, BB, HH)
  Si $(l = r)$ **alors**

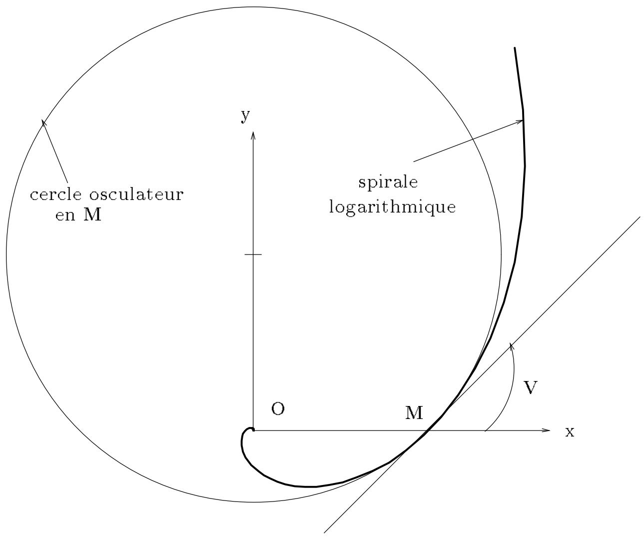


Figure 5.5: *Propriétés de la spirale logarithmique.*

$LL = RR = BB = HH = NULL$

Sinon

Ycoupe $(l, \mu - 1, LL, LR, BB, BH)$
Ycoupe (μ, r, RL, RR, HB, HH)
Fusionne2d $(MergeStatus, l, r, BB, BH, HB, HH, LR, LL, RR, RL)$
RR=LR
LL=RL

5.2.3 Complexité

Elle est la somme de deux termes : le temps de construction du 2d-tree et le temps de calcul de la triangulation.

5.2.3.1 Complexité dans le pire des cas

1. La construction du 2d-tree est en $\Theta(n \log n)$ (voir paragraphe 2.2.3).

2. Nous allons montrer que la complexité dans le pire des cas du processus total de fusion est en $\Omega(n \log n)$. Pour cela, nous allons étudier la triangulation de Delaunay de n sites répartis sur une portion de spirale logarithmique.

   La spirale logarithmique d'équation en coordonnées polaires $\rho = ae^{m\theta}$ a de nombreuses propriétés (Figure 5.5) :

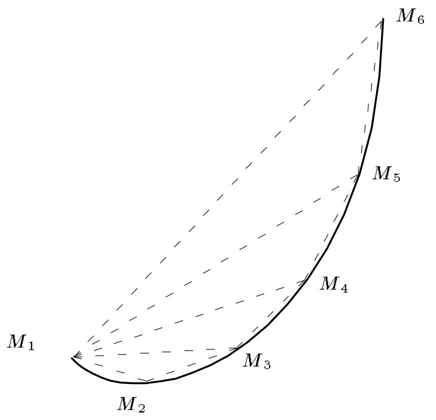


Figure 5.6: *Triangulation sur une portion de spirale.*

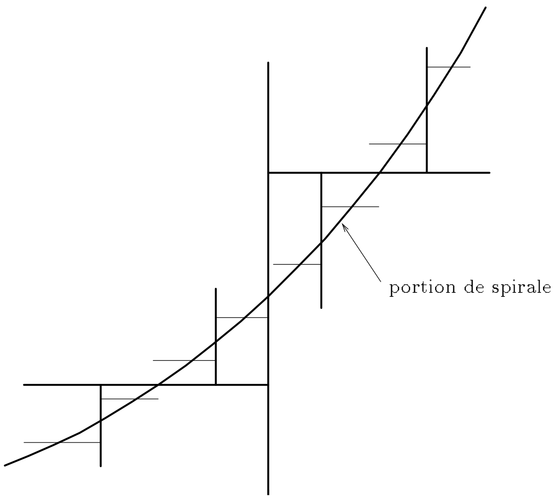


Figure 5.7: *Découpage induit par le 2d-arbre.*

(a) La tangente en M fait un angle V constant avec la droite OM ($\tan V = \frac{1}{m}$).

(b) Le cercle passant par M et centré au centre de courbure en M contient toute la section de la spirale située *avant* M (au sens de l'abscisse curviligne); toute la section de la spirale *après* M est à l'extérieur de ce cercle.

(c) Le cercle circonscrit à trois sites M_i, M_j et M_k ($i < j < k$) situés sur la spirale contient toute la partie de la spirale *avant* M_k ; toute la partie de la spirale *après* M_k est à l'extérieur de ce cercle.

Il en résulte que construire la triangulation de Delaunay d'un ensemble de sites M_i situés sur une portion de spirale consiste à relier le site M_1 à tous les autres sites M_j ($j \neq 1$), puis à relier chaque site M_i à son suivant M_{i+1}. Ainsi, si l'on ajoute un site sur la spirale avant M_1, il n'y a plus un seul triangle valide (Figure 5.6).

Considérons par exemple la spirale d'équation $\rho = e^\theta$.

$$\left\{ \begin{array}{l} x(\theta) = e^\theta \cos\theta \\ y(\theta) = e^\theta \sin\theta \end{array} \right. \Rightarrow \left\{ \begin{array}{l} x\prime(\theta) = \sqrt{2}e^\theta \cos(\theta + \frac{\pi}{4}) \\ y\prime(\theta) = \sqrt{2}e^\theta \sin(\theta + \frac{\pi}{4}) \end{array} \right.$$

Les abscisses et les ordonnées sont strictement croissantes sur l'intervalle $]-\frac{\pi}{4}, \frac{\pi}{4}[$. Supposons que l'on répartisse n sites sur cette portion de spirale. Le $2d$-arbre va découper cet ensemble en cellules qui ont la propriété suivante : si l'on considère les sites d'une cellule, ils sont soit tous *avant* les sites d'une autre cellule (au sens des relations $<_V$ et $<_H$ du paragraphe 2.2.2), soit tous *après* (Figure 5.7). Par conséquent, lors de la fusion de deux cellules, tous les triangles de l'une ou de l'autre seront détruits. On a donc l'équation de récurrence suivante (voir Paragraphe 4.4) :

$$T(n) = 2T(\frac{n}{2}) + \Omega(\frac{n}{2}) \Rightarrow T(n) \in \Omega(n \log n)$$

Le processus complet de fusion a une complexité dans le pire des cas en $\Omega(n \log n)$. ◇

Remarque : on peut aussi démontrer ce résultat en répartissant n points sur une portion de parabole, ainsi que sur certaines courbes convexes.

Comme cet algorithme est aussi en $O(n \log n)$ pour la raison évoquée dans le paragraphe 5.1.3.1, le processus de fusion est exactement en $\Theta(n \log n)$.

L'algorithme est optimal en $\Theta(n \log n)$.

5.2.3.2 Complexité en moyenne

On suppose que l'on a une distribution quasi-uniforme dans un carré unitaire (voir paragraphe 6.1).

1. Bien que la distribution soit quasi-uniforme, le temps de construction du 2d-tree reste en $\Theta(n \log n)$.

2. D'après le Corollaire 3.4.1, la complexité de la fusion de deux triangulations est proportionnelle au nombre de sites inachevés. On peut donc appliquer le théorème 6.6.1 et on déduit que **la complexité en moyenne de l'ensemble du processus de fusion est linéaire.** On peut aussi trouver une analyse de complexité spécifique au cas bidimensionnel dans l'article [134] de Lemaire et Moreau inclus dans l'annexe F.

L'algorithme reste en $\Theta(n \log n)$ même en moyenne, à cause de la construction du 2d-tree. Mais, comme les résultats expérimentaux du chapitre 7 le confirment, le calcul de la triangulation, bien que linéaire en moyenne, prend plus de temps que la construction du 2d-tree qui est en $\Theta(n \log n)$, et on l'a vérifé pour des ensembles atteignant 10 millions de sites. Globalement, la complexité en moyenne de l'algorithme reste égale à celle de l'algorithme de Lee et Schachter, mais on a considérablement amélioré la phase de triangulation qui est de loin celle qui consomme le plus de temps. Les résultats expérimentaux du chapitre 7 le montrent plus précisément.

Remarque : Shewchuk [182] a récemment implémenté cet algorithme et les résultats expérimentaux qu'il obtient sont comparables aux nôtres (voir chapitre 7). Son programme permet aussi l'utilisation d'une arithmétique exacte adaptative [183] [184], l'ajout incrémental d'arêtes de contrainte, la destruction d'arêtes extérieures ou intérieures à un contour donné, l'ajout incrémental de nouveaux sites pour rendre la triangulation conforme... Shewchuk montre aussi expérimentalement que cet algorithme pose moins souvent des problèmes vis-à-vis de l'imprécision numérique que celui de Lee et Schachter. En effet, celui de Lee et Schachter, parce qu'il triangule des bandes extrêmement étroites, doit souvent déterminer la position d'un point par rapport à un segment, alors que les extrémités du segment et ce point sont presque alignés (voir par exemple figure 1.9, cadre 3). Les risques d'erreur sont donc plus importants que dans l'algorithme basé sur le 2d-tree qui évite la plupart de ces triangles très aplatis (voir par exemple figure 5.3, cadre 3).

5.3 Algorithme basé sur un random 2d-tree

Pour gagner du temps dans la construction de l'arborescence des sites, on peut utiliser un *random 2d-tree*. Pour *Xdiviser* (ou *Ydiviser*) un ensemble de sites, on utilise le site situé au milieu du tableau pour effectuer le partage. En général, ce n'est pas le médian[3], on le baptise *faux médian*. Pour pouvoir reproduire le schéma de découpage lors de la triangulation, on stocke dans un second tableau les indices de ces "faux médians" utilisés dans chaque division. L'algorithme de construction du random 2d-tree est semblable mais plus simple que celui du paragraphe 2.2.4. Il suffit que tous les éléments inférieurs (pour les relations d'ordre $<_x$ ou $<_y$) (resp. supérieurs) au faux médian soient avant (resp. après) dans le tableau. On stocke l'indice du faux médian dans un deuxième tableau. Ces indices sont stockés suivant *l'ordre préfixé* [178] de parcours du 2d-tree. L'algorithme de triangulation est presque identique à celui du paragraphe 5.2.2. Pour obtenir les "faux-médians" μ, il suffit de prendre les indices stockés dans le second tableau, en partant du début du tableau.

Le temps de construction du 2d-tree est plus faible (voir annexe E.1), mais l'arborescence n'est plus équilibrée. Cet algorithme a été testé dans le chapitre 7.

Remarque : Luc Devroye m'a fait remarquer qu'il a démontré avec Chanzy [44] en 1994 que la somme des périmètres des cellules d'un random 2d-tree est *en moyenne* en $\Theta(n^{\frac{\sqrt{17}-3}{2}}) \approx \Theta(n^{0.56})$ et que , en utilisant le théorème 6.6.1, on déduit que la complexité *en moyenne* de l'ensemble du processus de fusion est en $O(n^{\frac{\sqrt{17}-2}{2}}) \approx O(n^{1.06})$. Ce n'est évidemment pas une borne inférieure car, comme la hauteur d'un random 2d-tree [66] est logarithmique, le processus de fusion est, même dans le pire des cas, en

[3]Seul son indice est médian parmi les indices du tableau.

$O(n \log n)$. Calculer la complexité *en moyenne* du processus de fusion de l'algorithme basé sur un random 2d-tree reste donc un problème ouvert. Le graphique 7.3 de l'étude expérimentale permet même de douter de la linéarité du processus !

5.4 Algorithme basé sur un adaptive 2d-tree

Adam, Elbaz et Spehner ont proposé cette variante et ils montrent que la complexité en moyenne de la phase de triangulation de l'algorithme basé sur un adaptive 2d-tree est aussi linéaire [3] [4].

Pour encore améliorer la phase de triangulation, on peut utiliser un *adaptive 2d-tree*. Sa construction se déduit de celle du 2d-tree. Après la division de chaque domaine, il faut calculer le rectangle englobant des deux nouveaux domaines pour trouver les nouvelles directions de division (elles se font orthogonalement au plus grand côté). Pour pouvoir retrouver le schéma de découpage lors de la triangulation (l'ordre des fusions est l'inverse de celui des divisions), on stocke dans un second tableau les directions de division, par exemple +1 (resp. -1) pour une division selon les abscisses (resp. ordonnées). Ces valeurs sont stockées suivant *l'ordre préfixé* [178] de parcours du 2d-tree. L'algorithme de triangulation est presque identique à celui du paragraphe 5.2.2. Il suffit seulement, quand on veut fusionner deux cellules, de consulter le tableau auxiliaire qui nous indique si la fusion se fait verticalement ou horizontalement.

Le temps de construction de l'adaptive 2d-tree est plus important (voir annexe E.1). Mais les cellules produites par le découpage sont de forme presque carrée. Ainsi, les fusions des triangulations de Delaunay contenues dans ces cellules détruiront peu d'arêtes. Le travail nécessaire pour construire la triangulation de Delaunay du semis sera moins important, comme le confirment les résultats expérimentaux du chapitre 7.

5.5 Algorithme basé sur un random adaptive 2d-tree

Pour accélérer la construction de l'arborescence des sites, on peut utiliser un *random adaptive 2d-tree*. L'algorithme se déduit des paragraphes 5.3 et 5.4. Lors de la construction du random adaptive 2d-tree, il faut stocker dans un tableau auxiliaire les faux médians ainsi que les directions de division. Pour optimiser la gestion de la mémoire, il suffit de stocker l'indice du site selon lequel on effectue la division, affecté d'un signe + (resp. -) si la division se fait selon le sens vertical (resp. horizontal). Quand on veut fusionner deux cellules, on consulte ce tableau auxiliaire qui indique l'indice selon lequel on doit diviser le tableau des sites et la direction selon laquelle on doit fusionner

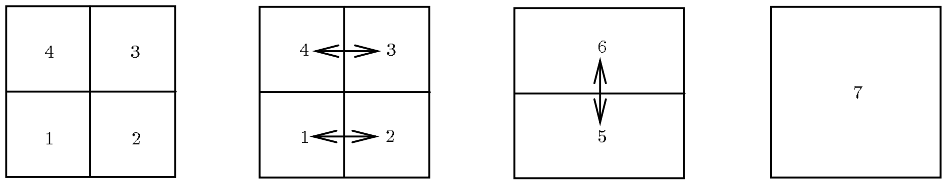


Figure 5.8: *Fusions selon un quadtree : fusions horizontales des cellules 1 et 2, puis des cellules 3 et 4, fusion verticale des cellules 5 et 6.*

les deux sous-ensembles.

Les avantages et les inconvénients se déduisent des paragraphes précédents : le temps de construction est plus faible (voir annexe E.1), car on ne calcule pas le vrai médian, mais l'arborescence peut être légèrement déséquilibrée. Les résultats expérimentaux sont donnés dans le chapitre 7.

Remarque : Calculer la complexité *en moyenne* du processus de fusion de l'algorithme basé sur un random adaptive 2d-tree est un problème ouvert. Toutefois, le graphique 7.3 de l'étude expérimentale semble montrer la linéarité du processus.

5.6 Algorithme basé sur un quadtree

Cet variante [185] a été proposée par C. Simon et P. Brehmer (Ouvrages d'Art, SETRA) après lecture du rapport technique [133] présentant le calcul de la triangulation de Delaunay à l'aide d'un 2d-tree.

Comme dans tous ces algorithmes, les fusions ont lieu dans l'ordre inverse du découpage créé par le *quadtree*[4]. Comme le découpage se fait par quatre et les fusions par deux, la figure 5.8 illustre le processus de fusion.

Dans le cas d'une distribution uniforme, comme la hauteur du quadtree est, avec cette hypothèse, logarithmique, la complexité de l'ensemble du processus de fusion est linéaire en moyenne. La démonstration se déduit trivialement de celle du paragraphe 5.2.3.2 avec le 2d-tree.

La complexité dans le pire des cas est au moins quadratique : il suffit de disposer des points sur un arc de spirale (par exemple celui de la figure 5.6) de telle sorte que la hauteur du quadtree soit linéaire. Chaque fusion (il y en a n) détruit tous les triangles déjà construits d'où la complexité quadratique du processus. Mais, si l'on utilise une précision finie (le contraire est exceptionnel !), la hauteur du quadtree sera majorée par

[4] Il s'agit du Point Region Quadtree décrit dans le paragraphe 2.1.1.

α, le nombre de bits servant à représenter les coordonnées. L'espace mémoire et le temps de construction seront en $O(\alpha n)$ (voir paragraphe 2.1.3) et, par conséquence, la complexité dans le pire des cas du processus de fusion en $O(\alpha n)$.

Le quadtree se construit plus rapidement que le 2d-tree (voir annexe E.1). Comme ses cellules sont carrées, les fusions des triangulations de Delaunay contenues dans ces cellules détruiront peu d'arêtes. Le travail nécessaire pour construire la triangulation de Delaunay du semis sera peu important, comme le confirment les résultats expérimentaux du chapitre 7. Par contre, l'espace mémoire utilisé par l'algorithme est plus important et a un caractère aléatoire. L'espace mémoire nécessaire ne peut être prévu exactement à l'avance (on ne connaît qu'une majoration large), contrairement à l'algorithme basé sur le 2d-tree où il peut être calculé de façon exacte. Ceci peut être un handicap dans certaines applications de CAO où l'on cherche à utiliser au maximum la mémoire vive.

5.7 Algorithme basé sur un bucket-tree

L'algorithme est donné dans le paragraphe 2.3.5, il suffit de remplacer *TraiterRectangle* par *fusionne2d* qui est la fonction de fusion bidirectionnelle des sous-triangulations. Les fusions se font encore dans l'ordre inverse des divisions. Cet algorithme, dans le cas d'une distribution de sites dans un carré, ressemble à celui de Katajainen et Koppinen[5] [117].

Cet algorithme est linéaire en moyenne avec l'hypothèse d'une distribution quasi-uniforme, et en $O(n \log n)$ dans le pire des cas.

Le hachage (découpage du domaine en cellules) étant extrêmement rapide, cette méthode est imbattable pour des distributions uniformes. Par contre, elle perd de son efficacité pour des distributions comportant de fortes concentrations ("clusters") de sites en certains endroits où la méthode basée sur un 2d-tree résiste mieux parce que plus adaptive à l'irrégularité de la distribution. D'autre part, l'espace mémoire nécessaire est un peu plus important qu'avec le 2d-tree.

Notons aussi l'algorithme de Dwyer [70] qui regroupe d'abord les cellules en colonnes par des fusions verticales, puis qui fusionne les colonnes horizontalement jusqu'à l'obtention de la triangulation de l'ensemble des sites. En plaçant environ $(\log n)$ points par cellule, R. Dwyer démontre que la complexité en moyenne est en $O(n \log \log n)$ et la complexité dans le pire des cas toujours en $O(n \log n)$. Cet algorithme a été testé expérimentalement. La courbe obtenue est asymptotiquement linéaire.

[5]Dans leur algorithme, chaque cellule élémentaire peut contenir entre 1 et 4 sites, parce qu'ils choisissent toujours un nombre de cellules qui est une puissance de 4, alors que dans notre algorithme, les cellules élémentaires contiennent toujours un seul site.

Mais un terme en *loglog* est très difficile à mettre en évidence expérimentalement, car il croît trop lentement !

Dans l'algorithme de Dwyer, la simple triangulation des cellules avec la méthode de Lee et Schachter entraîne une complexité en $\Theta(n \log\log n)$. En effet, il y a $\frac{n}{\log n}$ cellules qui contiennent chacune environ $(\log n)$ points, donc le coût de leur triangulation vaut $\Theta(\frac{n}{\log n} \times (\log n)\log(\log n)) = \Theta(n \log\log n)$. Aussi, on peut se demander si, en plaçant en moyenne *un point* par cellule (au lieu de $(\log n)$ points), la complexité en moyenne de la triangulation, avec le schéma de fusionnement des cellules de l'algorithme de Dwyer, n'est pas linéaire ! C'est un problème ouvert et le raisonnement (voir chapitre 6) utilisé pour analyser l'algorithme basé sur le 2d-tree ne permet pas de prouver la linéarité du processus.

5.8 Conclusion

Les algorithmes présentés dans ce chapitre améliorent considérablement l'algorithme de Lee et Schachter [131]. En effet, la fusion bidirectionnelle des sous-triangulations est beaucoup plus efficace que la fusion monodirectionnelle et conduit en général (excepté pour quelques distributions pathologiques) à une complexité *linéaire* en ce qui concerne la phase de triangulation (prétraitement exclu). La complexité dans le pire des cas reste optimale en $\Theta(n \log n)$ (sauf pour l'algorithme fondé sur le quadtree). D'autre part, le code de ces algorithmes est à peine plus long et les sites qui ne sont pas en position générale (alignés ou cocycliques) ne sont pas une gêne pour ces algorithmes (ils n'entraînent aucun cas particulier supplémentaire). L'étude expérimentale de ces algorithmes (sur un large éventail de distributions) ainsi que leur comparaison avec les "meilleurs" algorithmes de triangulation de Delaunay (d'après Su et Drysdale [188]) est faite dans le chapitre 7.

Chapitre 6

UNE CLASSE D'ALGORITHMES DIVIDE-AND-CONQUER EN DIMENSION QUELCONQUE

Laurent Schwartz, en 1959, dans son cours de "Méthodes Mathématiques de la Physique", avait avoué son regret de ne pas vivre dans un espace à 6 dimensions où la formule de la surface de la sphère[a] est si jolie [137]...

[a] *Elle est égale à $\pi^3 R^5$.*

Ce chapitre utilise la notion de *site inachevé* introduite par Katajainen et Koppinen [117] pour l'analyse d'un algorithme de triangulation de Delaunay dans le plan, basé sur un découpage du domaine à l'aide d'une grille régulière. Ce concept est généralisé à un espace de dimension quelconque. Des résultats probabilistes sont donnés dans l'hypothèse de distribution quasi-uniforme: probabilité qu'un site soit inachevé, espérance du nombre de sites inachevés dans un hyper-rectangle. Les majorations sont valables dans un espace de dimension quelconque et, si on les applique à un espace de dimension 2, elles sont meilleures que celles de Katajainen et Koppinen. Ces résultats peuvent être utilisés pour analyser des algorithmes en dimension quelconque, mais il faut toutefois admettre que ces algorithmes doivent être plus ou moins liés à des triangulations de Delaunay car le concept de "site inachevé" n'a de sens que par l'intermédiaire d'une triangulation de Delaunay. Ensuite, un algorithme du type *Divide-and-Conquer*, construit sur un kd-tree [16] (équilibré) est présenté. Il est montré que la complexité en moyenne – en termes de sites inachevés – du processus de fusion multidimensionnelle dans l'hypothèse de distribution quasi-uniforme dans un hypercube est linéaire [135] [136]. On espère que ce résultat pourra être utilisé pour calculer une triangulation de Delaunay en dimension quelconque, avec un algorithme *Divide-and-Conquer* construit sur un kd-tree. Les chapitres précédents montrent l'application à la dimension 2, ainsi que les très bons résultats obtenus. En dimension supérieure, la difficulté réside dans le processus de fusion des sous-triangulations. Dans la littérature, le processus de fusion n'existe qu'en dimension 2. Cignoni, Montani et Scopigno [54] présentent un algorithme *Divide-and-conquer* en dimension quelconque, mais le processus de construction est incrémental, et le problème de la fusion en dimension quelconque n'est pas résolu. Voici cependant quelques arguments qui permettent de garder espoir :

1. le travail de thèse de M. Elbaz [77] et de B. Adam [2] montre un algorithme de construction de la triangulation de Delaunay dans l'espace qui est vraiment du type *Divide-and-Conquer*. Le processus de fusion des sous-triangulations est correctement décrit, par contre, l'analyse de la complexité en moyenne n'est pas faite. Nous espérons que les résultats exposés dans ce chapitre pourront être utilisés pour cette analyse.

2. C. E. Buckley [39] propose un algorithme du type *Divide-and-Conquer* pour calculer une enveloppe convexe en dimension 4, algorithme qu'il généralise ensuite à une dimension quelconque. Rappelons le lien très fort entre le calcul d'une enveloppe convexe et le calcul d'une triangulation de Delaunay : le calcul d'une triangulation de Delaunay en dimension n peut être ramené au calcul d'une enveloppe convexe dans un espace de dimension $n+1$; pour cela, il suffit de projeter les points sur un paraboloïde d'axe orthogonal à l'hyperplan contenant les points à trianguler, de calculer l'enveloppe convexe de ces points, de la projeter sur l'hyperplan initial et on a ainsi la triangulation voulue.

3. personne n'a jamais montré l'impossibilité du processus de fusion en dimension supérieure à 2. Il est probable que beaucoup ont été découragés par son apparente difficulté.

Ce chapitre donne une nouvelle motivation pour chercher un algorithme de fusion de sous-triangulations en dimension quelconque, qui permettrait de calculer la triangulation de Delaunay avec une méthode du type *Divide-and-Conquer*. D'après les résultats présentés ci-dessous, cet algorithme a toutes les chances d'être très performant et de pouvoir rivaliser avec celui de Dwyer [71], qui utilise un prépartitionnement des points selon une grille régulière.

6.1 Hypothèses

On suppose que la distribution est quasi-uniforme dans un hypercube unitaire $\mathcal{U}_k$. Soit f la densité de probabilité de la distribution, dont les bornes sont des réels strictement

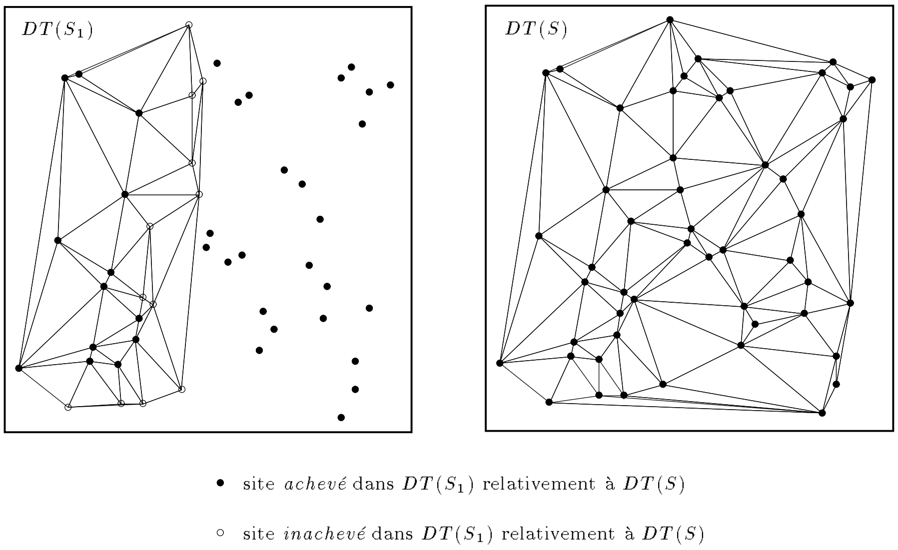


Figure 6.1: *Sites inachevés dans une sous-triangulation.*

positifs c_1 et c_2 tels que $c_1 \leq c_2$:

$$\begin{cases} \forall(x_1, x_2, \ldots, x_k) \in \mathcal{U}_k, & c_1 \leq f(x_1, x_2, \ldots, x_k) \leq c_2 \\ \forall(x_1, x_2, \ldots, x_k) \notin \mathcal{U}_k, & f(x_1, x_2, \ldots, x_k) = 0 \end{cases}$$

Par exemple, la probabilité qu'un point donné soit dans un domaine $\mathcal{D}$ est égale à $\int_{\mathcal{D}} f$.

6.2 Sites inachevés

La notion de *site inachevé* a été introduite par Katajainen et Koppinen [117] pour l'analyse d'un algorithme de triangulation de Delaunay dans le plan, basé sur un découpage du domaine à l'aide d'une grille régulière.

Considérons par exemple la triangulation de Delaunay plane $DT(S_1)$ de l'ensemble de points S_1, sous-ensemble de S (voir Figure 6.1). Lorsque l'on considère la triangulation de Delaunay $DT(S)$ de l'ensemble S, on s'aperçoit que certains points de S_1 ont la même liste d'adjacence que dans $DT(S_1)$: on dit que ces points sont *achevés* dans $DT(S_1)$ *relativement à* $DT(S)$) alors que d'autres points (cercles blancs sur Figure 6.1) reçoivent de nouvelles arêtes ou en perdent (on dit que ces points sont *inachevés* dans $DT(S_1)$ *relativement à* $DT(S)$. Intuitivement, il apparaît que les points inachevés sont généralement proches de la frontière de la sous-triangulation. Dans les paragraphes suivants, nous donnerons un majorant de la probabilité d'être inachevé pour un site d'un domaine rectangulaire sous l'hypothèse d'une distribution quasi-uniforme. Ce majorant augmente quand le site se rapproche de la frontière du rectangle qui le contient. Nous donnerons aussi une majoration de l'espérance du nombre de sites inachevés dans un domaine rectangulaire.

Une formulation mathématique de ces différentes notions, que nous avons généralisées à un espace de dimension quelconque, est présentée ci-dessous. On note $DT(S)$ la k-triangulation de Delaunay d'un ensemble S de *sites*. Le terme "triangulation" sera utilisé dans ce chapitre, au lieu de k-triangulation.

Définition 6.2.1 *Soient $T(S_1)$ et $T(S_2)$ les triangulations des ensembles S_1 et S_2. On note $T(S_1) <_\triangle T(S_2)$ si $S_1 \subseteq S_2$ et si $T(S_1)$ contient chaque arête de $T(S_2)$ dont les extrémités appartiennent à S_1.*

Définition 6.2.2 *Soient $T(S_1) <_\triangle T(S_2)$ deux triangulations et $s \in S_1$. On dit que s est achevé dans $T(S_1)$ relativement à $T(S_2)$ si l'ensemble des arêtes adjacentes à s dans $T(S_1)$ et $T(S_2)$ coïncident, sinon on dit que s est inachevé (dans $T(S_1)$ relativement à $T(S_2)$).*

La première définition crée un ordre partiel dans l'ensemble des triangulations. Remarquons que $T(S_1) <_\triangle T(S_2)$ n'implique pas que toutes les arêtes de $T(S_1)$ appartiennent à $T(S_2)$. Remarquons aussi que $DT(S_1) <_\triangle DT(S_2)$ est équivalent à $S_1 \subseteq S_2$ si S_1 a une triangulation de Delaunay unique.

Proposition 6.2.1 *Soient $T(S_1) <_\triangle T(S_2)$ deux triangulations. Un site $s \in S_1$ est achevé dans $T(S_1)$ relativement à $T(S_2)$ si et seulement si S_1 contient les extrémités de toutes les arêtes adjacentes à s dans $T(S_2)$.*

Dém : soient $(s, p_1), \ldots (s, p_m)$ les arêtes adjacentes à s dans $T(S_2)$. Si s est achevé, alors les extrémités p_i appartiennent à S_1 par définition.

Réciproquement, supposons que les extrémités p_i appartiennent à S_1 mais que s ne soit pas achevé. Alors, $(s, p_1), \ldots (s, p_m)$ appartiennent à $T(S_1)$ parce que $T(S_1) <_\triangle T(S_2)$, mais $T(S_1)$ contient au moins une autre arête (s, r). Puisque le domaine délimité par $T(S_2)$ est convexe, il contient entièrement l'arête (s, r). Par conséquent, $(s, r) \setminus \{s\}$ coupe :

- soit un des k-triangles ouverts de $T(S_2)$ ayant s pour sommet. Comme un k-triangle de $T(S_2)$ ne contient aucun site en son intérieur (et, en particulier, pas r), (s, r) coupe donc une k-face opposée à s. Mais, ceci est impossible, puisque cette k-face appartient aussi à $T(S_1)$, qui est une triangulation.
- soit une k-face contenant s, ce qui est impossible, puisque cette k-face appartient aussi à $T(S_1)$, qui est une triangulation. $\diamond$

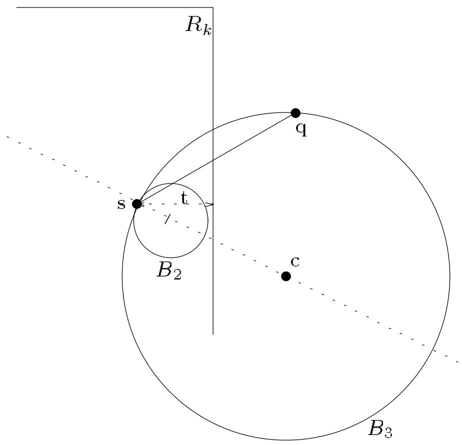


Figure 6.2: *La boule $\mathcal{B}_2$ est vide de sites.*

Corollaire 6.2.1 *Soient $T(S_1) <_\triangle T(S_2) <_\triangle T(S_3)$ trois triangulations. Un site $s \in S_1$ est achevé dans $T(S_1)$ relativement à $T(S_3)$ si et seulement si il est achevé à la fois dans $T(S_1)$ relativement à $T(S_2)$ et aussi dans $T(S_2)$ relativement à $T(S_3)$.*

Dém : la dernière condition implique la précédente à cause de la Définition 6.2.2. Réciproquement, supposons que s soit achevé dans $T(S_1)$ relativement à $T(S_3)$. Si (s, q) est une arête dans $T(S_3)$, alors $q \in S_1$; comme $q \in S_2$, et en utilisant la Proposition 6.2.1, nous obtenons que s est achevé dans $T(S_2)$ relativement à $T(S_3)$. Enfin, à cause de la Définition 6.2.2, s est aussi achevé dans $T(S_1)$ relativement à $T(S_2)$. ◇

6.3 Pavage autour d'un site inachevé

Pour pouvoir effectuer des calculs de probabilité relativement à un site, il est nécessaire de construire un pavage géométrique *fixe* autour de ce point, le volume de chaque pavé représentant une probabilité, à une constante multiplicative près.

Lemme 6.3.1 *Si s est un site inachevé du k-rectangle $\mathcal{R}_k$, situé à la distance t de la frontière de $\mathcal{R}_k$, alors il existe une k-boule $\mathcal{B}_2$ de rayon $\frac{t}{2}$, centrée à la distance $\frac{t}{2}$ de s, qui est vide de sites (voir Figure 6.2).*

Dém : si s est un site inachevé du k-rectangle $\mathcal{R}_k$, il existe un site q appartenant à $\mathcal{U}_k \setminus \mathcal{R}_k$ tel que (s,q) soit une arête de Delaunay dans $\mathcal{U}_k$. Puisque q est en dehors du k-rectangle $\mathcal{R}_k$, la longueur de (s,q) est supérieure à t. Comme (s, q) est une arête de Delaunay, il existe une k-boule $\mathcal{B}_3$ vide de sites telle que s et q soient sur sa frontière. Le diamètre de cette k-boule est donc supérieur à t. Soit c le centre de $\mathcal{B}_3$ et $\mathcal{B}_2$ la

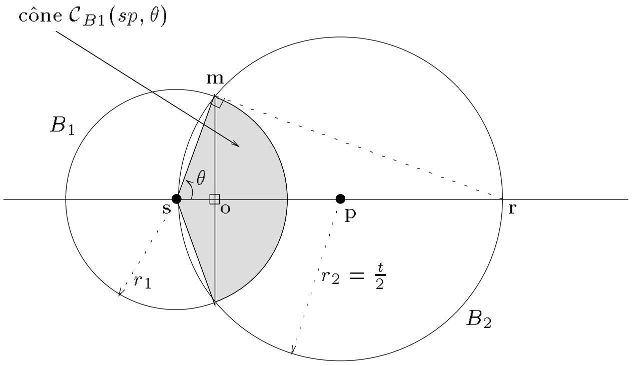


Figure 6.3: *Le cône $\mathcal{C}_{B1}(sp,\theta)$ est vide de sites.*

k-boule dont le centre est sur le segment $[sc]$ à une distance de $\frac{t}{2}$ de s. $\mathcal{B}_2$ est incluse dans $\mathcal{B}_3$, par suite $\mathcal{B}_2$ est vide de sites. $\diamond$

Corollaire 6.3.1 *On se place dans les conditions du Lemme 6.3.1. Soit la boule $\mathcal{B}_1(s,r_1)$ centrée en s et de rayon r_1 avec $r_1 \leq \frac{t}{2}$. Soit $\mathcal{C}(sp,\theta)$ le cône de sommet s, d'axe de symétrie sp et d'angle $\theta = \arccos\frac{r_1}{t}$. Alors l'intersection de ce cône avec la boule $\mathcal{B}_1$, soit $\mathcal{C}_{B1} = \mathcal{C} \cap \mathcal{B}_1$, est vide de sites (voir Figure 6.3).*

Dém : soit m un point quelconque appartenant à l'intersection des frontières des boules $\mathcal{B}_1$ et $\mathcal{B}_2$, o la projection orthogonale de m sur le segment $[sp]$, r diamétralement opposé à s sur la boule $\mathcal{B}_2$ et θ l'angle formé par les segments $[so]$ et $[sm]$ (voir Figure 6.3).

$$\cos\theta = \frac{sm}{sr} = \frac{r_1}{t} \Rightarrow \theta = \arccos\frac{r_1}{t}$$

La portion de cône $\mathcal{C}_{B1}(sp,\theta)$ est incluse dans $\mathcal{B}_2$, donc $\mathcal{C}_{B1}(sp,\theta)$ est vide de sites. $\diamond$

Lemme 6.3.2 *Soit un hypercube inscrit dans une boule $\mathcal{B}_1(s,r_1)$ centrée en s et de rayon r_1. Chaque face $\mathcal{F}$ de l'hypercube est divisée en "cellules" (ce sont des (k-1)-cubes) suivant tous les hyperplans parallèles aux* autres *faces de l'hypercube et qui passent par le centre de symétrie de la face $\mathcal{F}$. On appelle pyramide $\mathcal{P}$ l'intersection du cône de sommet s ayant pour section l'une des "cellules" ci-dessus avec la boule $\mathcal{B}_1$. On a :*

1. *la boule $\mathcal{B}_1$ est pavée par $(k2^k)$ pyramides qui ont toutes même volume.*
2. $\forall M_1, M_2 \in \mathcal{P} \setminus \{s\},\ \ \cos(\widehat{M_1 s M_2}) \geq (\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor)^{-1/2}$

Dém :

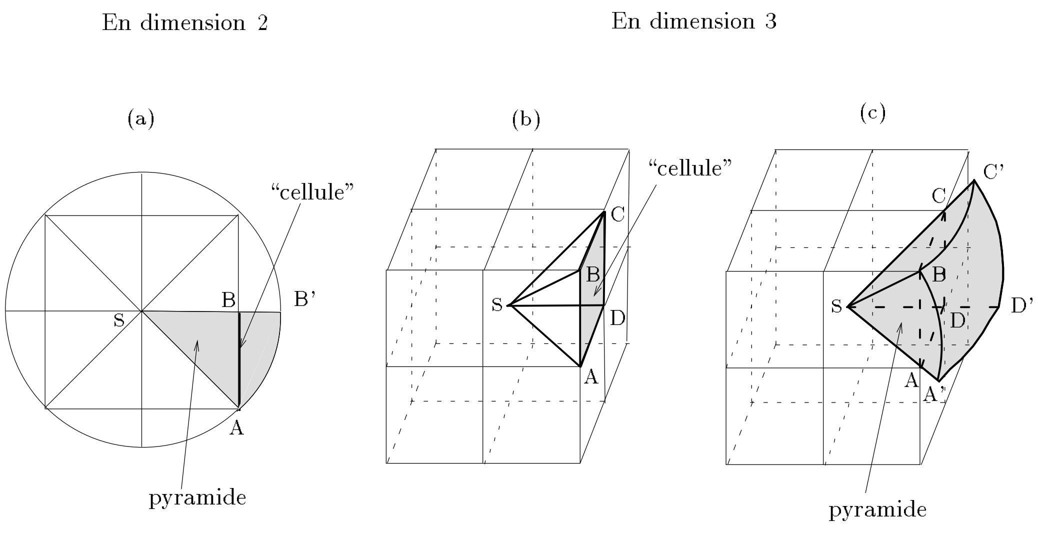


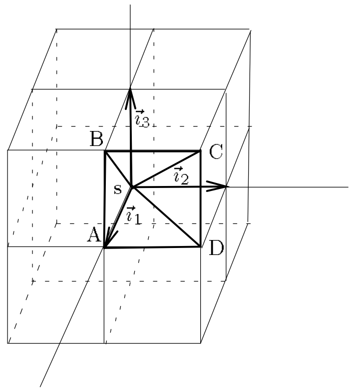
Figure 6.4: *Décomposition d'une boule en pyramides.*

1. un hypercube est un polyèdre régulier inscriptible dans une boule et ceci quelle que soit la dimension de l'espace [24]. Il a $(2k)$ faces. Chaque face $\mathcal{F}$ subit $(k-1)$ divisions et se décompose donc en 2^{k-1} "cellules", qui sont des (k-1)-cubes. Il y a donc $2k \times 2^{k-1} = k2^k$ "cellules". Sur chaque cellule, on peut construire une pyramide. La boule $\mathcal{B}_1$ est donc pavée par $(k2^k)$ pyramides qui ont toutes même volume pour des raisons de symétrie, soit $\mathcal{V}(\mathcal{P}) = \dfrac{\mathcal{V}(\mathcal{B}_1)}{k2^k} = \dfrac{\pi^{\frac{k}{2}} r_1^k}{k2^k\Gamma(\frac{k}{2}+1)}$.

   En dimension 2 (voir Figure 6.4 a), chaque côté du carré est divisé en deux segments égaux. On a donc 8 cellules qui donnent naissance à 8 pyramides qui sont ici des secteurs angulaires. En dimension 3 (voir Figure 6.4 b), chaque face du cube est divisée en 4 carrés égaux. On a donc 24 cellules qui donnent naissance à 24 pyramides (voir Figure 6.4 c). En dimension 4, chaque face est divisée en 8 cubes égaux. On a donc 64 cellules qui donnent naissance à 64 pyramides...

2. soit $(s, \vec{\imath}_1, \ldots, \vec{\imath}_k)$ un repère orthogonal tel que ses axes soient orthogonaux aux faces de l'hypercube et que la norme des vecteurs de base soit égale à la moitié de la longueur d'un côté de l'hypercube. Sans perte de généralité, on considère la "cellule" contenue dans l'hyperplan $x = 1$ et dont les coordonnées de tous ses sommets sont positives ou nulles. Soit $\mathcal{P}$ la pyramide associée à cette cellule (voir la Figure 6.5 en dimension 3).

   Soit M_1 et M_2 deux points quelconques de $\mathcal{P}$, différents de s. Nous allons rechercher le maximum de l'angle $\widehat{M_1 s M_2}$. Pour trouver ce maximum, il suffit de faire varier les points M_1 et M_2 parmi tous les sommets de la "cellule", section



pyramide (s,A,B,C,D) s(0,0,0)
A(1,0,0)
B(1,0,1)
C(1,1,1)
D(1,1,0)

$\cos(AsC) = \frac{1}{\sqrt{3}} \Rightarrow angle(AsC) \approx 54,74^0$

$\cos(BsD) = \frac{1}{2} \Rightarrow angle(BsD) = 60^0$

$angle(BsD) > angle(AsC)$

Figure 6.5: *Angle maximal au sommet d'une pyramide.*

de la pyramide. Ces sommets ont des coordonnées de la forme $(1, \alpha_2, \alpha_3, \ldots, \alpha_k)$ avec α_i valant 0 ou 1 pour i variant de 2 à k. Soit $M_1(1, \alpha_2, \alpha_3, \ldots, \alpha_k)$ et $M_2(1, \beta_2, \beta_3, \ldots, \beta_k)$ deux sommets de cette cellule.

$$\cos \widehat{M_1 s M_2} = \frac{1 + \sum_{i=2}^{k} \alpha_i \beta_i}{\sqrt{1 + \sum_{i=2}^{k} \alpha_i^2}\sqrt{1 + \sum_{i=2}^{k} \beta_i^2}} = \frac{N}{D}$$

On recherche le minimum de cette expression. Le numérateur peut prendre des valeurs entières comprises entre 1 et k. Supposons $N = u$ avec $(1 \leq u \leq k)$. Cela implique que, pour u coordonnées, on a $\alpha_i = \beta_i = 1$ et que pour les autres coordonnées, on n'a jamais $\alpha_i = \beta_i = 1$. Pour fixer les idées (sans perte de généralité), on suppose que ce sont les u premières coordonnées qui valent 1.

$$\cos \widehat{M_1 s M_2} = \frac{u}{\sqrt{u + \sum_{i=u+1}^{k} \alpha_i^2}\sqrt{u + \sum_{i=u+1}^{k} \beta_i^2}} = \frac{u}{D_u} = \frac{u}{\sqrt{A_u}\sqrt{B_u}}$$

A_u ne peut prendre que des valeurs entières comprises entre u et k. Supposons $A_u = u + l = A_u^l$ fixé $(0 \leq l \leq k - u)$. Alors, B_u^l ne peut prendre que des valeurs entières comprises entre u et $k - l$ (puisque les α_i et β_i qui restent ne peuvent être égaux à 1 en même temps). Pour A_u^l fixé, le dénominateur D_u^l est maximal pour $B_u^l = k - l$.

Recherchons maintenant la valeur de l $(0 \leq l \leq k - u)$ qui maximise D_u^l.

$$D_u^l = \sqrt{u + l}\sqrt{k - l} = \sqrt{f_u(l)}$$

$$f'_u(l) = k - u - 2l = 0 \Leftrightarrow l = \frac{k-u}{2}$$

1) (k-u) pair , le maximum est atteint pour $l = \frac{k-u}{2}$

$$\cos \widehat{M_1 s M_2} = \frac{2u}{u+k} = \frac{2}{1+\frac{k}{u}}$$

Le cosinus est minimal pour $u = 1$ (k impair) et vaut $\frac{1}{\frac{k+1}{2}} = (\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{\frac{-1}{2}}$

2) (k-u) impair , le maximum est atteint pour $l = \frac{k-u-1}{2}$

$$\cos \widehat{M_1 s M_2} = \frac{2u}{\sqrt{(u+k+1)(u+k-1)}} = \frac{2}{\sqrt{(1+\frac{k+1}{u})(1+\frac{k-1}{u})}}$$

Le cosinus est minimal pour $u = 1$ (k pair) et vaut $\frac{1}{\sqrt{\frac{k}{2}\frac{k+2}{2}}} = (\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{\frac{-1}{2}}$

On peut vérifier ces résultats dans le plan et dans l'espace à l'aide des Figures 6.4 et 6.5. ◇

Corollaire 6.3.2 *Si s est un site inachevé dans le k-rectangle $\mathcal{R}_k$ situé à la distance t de la frontière de $\mathcal{R}_k$, alors il existe un pavage fixe d'un voisinage de s par $(k2^k)$ pavés de même volume et tel qu'au moins un de ces pavés soit vide de sites, le volume d'un pavé étant* $\dfrac{\pi^{\frac{k}{2}} t^k}{k\, 2^k (\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{\frac{k}{2}} \Gamma\left(\frac{k}{2}+1\right)}$.

Dém : on est dans les conditions du Corollaire 6.3.1. On fixe le rayon de la boule $\mathcal{B}_1$ à $r_1 = t(\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{-1/2}$. On décompose la boule $\mathcal{B}_1$ en pyramides comme indiqué dans le Lemme 6.3.2. Soit q l'intersection du segment $[sp]$ et de la frontière de la boule $\mathcal{B}_1$. Il existe une pyramide $\mathcal{P}_j$ telle que $[sq] \subset \mathcal{P}_j$.

$$\begin{aligned} & \forall M \in \mathcal{P}_j, \quad \cos \widehat{qsM} \geq (\lceil \tfrac{k+1}{2} \rceil \lfloor \tfrac{k+1}{2} \rfloor)^{-1/2} = \frac{r_1}{t} \\ \Rightarrow \quad & \forall M \in \mathcal{P}_j, \quad M \in \mathcal{C}_{B1}(sp, \arccos \frac{r_1}{t}) \end{aligned}$$

$$\text{donc } \mathcal{P}_j \subset \mathcal{C}_{B1}(sp, \arccos \tfrac{r_1}{t})$$

Comme la portion de cône $\mathcal{C}_{B1}$ est vide de sites, le pavé $\mathcal{P}_j$ est vide de sites. Il a pour volume $\dfrac{\pi^{\frac{k}{2}} t^k}{k\, 2^k (\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{\frac{k}{2}} \Gamma\left(\frac{k}{2}+1\right)}$. ◇

6.4 Probabilité qu'un site soit inachevé

Le pavage réalisé autour du site s permet de majorer la probabilité que le site s soit inachevé.

Lemme 6.4.1 *Supposons que la densité de probabilité f soit quasi-uniforme avec les bornes c_1 et c_2. Considérons un rectangle $\mathcal{R}_k \subseteq \mathcal{U}_k$ et $DT(S \cap \mathcal{R}_k) <_\triangle DT(S)$. Si $s \in S \cap \mathcal{R}_k$ est un sommet situé à une distance minimale t de la frontière de $\mathcal{R}_k$, alors la probabilité $P(\mathcal{C}_1)$ que s soit inachevé dans $DT(S \cap \mathcal{R}_k)$ relativement à $DT(S)$ est au plus :*

$$P(\mathcal{C}_1) \leq k \times 2^k \times \left(1 - \frac{c_1 \times \pi^{\frac{k}{2}} \times t^k}{k \times 2^k \times (\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{\frac{k}{2}} \times \Gamma\left(\frac{k}{2}+1\right)}\right)^{n-1}$$

Dém : on considère la boule $\mathcal{B}_1$ centrée en s et de rayon $t(\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{-1/2}$, pavée par les pyramides $\mathcal{P}_j$, $j \in [1, k \times 2^k]$. D'après le Corollaire 6.3.2, la condition $\mathcal{C}_1$ – que s est inachevé dans $DT(S \cap \mathcal{R}_k)$ relativement à $DT(S)$ – entraîne la condition $\mathcal{C}_2$ – que au moins un des pavés ouverts $\mathcal{P}_j$ soit vide de sites.

$$\begin{aligned}
P(\mathcal{C}_1) &\leq P(\mathcal{C}_2) \\
&\leq \sum_{j=1}^{k \times 2^k} P(S \cap \mathcal{P}_j = \emptyset) \\
&= \sum_{j=1}^{k \times 2^k} \left(1 - \int_{\mathcal{P}_j} f\right)^{n-1} \\
&\leq k \times 2^k \times \left(1 - \frac{c_1 \times \pi^{\frac{k}{2}} \times t^k}{k \times 2^k \times (\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor)^{\frac{k}{2}} \times \Gamma\left(\frac{k}{2}+1\right)}\right)^{n-1}
\end{aligned}$$

en utilisant la propriété $f(x_1, x_2, \ldots, x_k) \geq c_1$. $\diamond$

6.5 Espérance du nombre de sites inachevés

En intégrant la probabilité précédente sur un domaine rectangulaire, on obtient un majorant de l'espérance du nombre de sites inachevés dans un hyperrectangle. Le Lemme 6.5.1 et le Corollaire 6.5.1 donnent respectivement un calcul et une majoration d'intégrale, nécessaires pour ce calcul probabiliste.

Lemme 6.5.1 *Soit n et k des entiers tels que $n \geq 0$ et $k \geq 2$. On a :*

$$\mathcal{I}_n^k = \int_0^1 \left(1 - x^k\right)^n dx = \prod_{i=1}^{n} \frac{ki}{ki+1}$$

Dém :

$$\begin{aligned}
\mathcal{I}_n^k &= \int_0^1 (1)\left(1 - x^k\right)^n dx \\
&= \left[x\left(1 - x^k\right)^n\right]_0^1 - \int_0^1 xn\left(-kx^{k-1}\right)\left(1 - x^k\right)^{n-1} dx \\
&= -nk \int_0^1 -x^k\left(1 - x^k\right)^{n-1} dx \\
&= -nk\left[\int_0^1 \left(1 - x^k\right)\left(1 - x^k\right)^{n-1} dx - \int_0^1 \left(1 - x^k\right)^{n-1} dx\right] \\
&= -nk\left[\mathcal{I}_n^k - \mathcal{I}_{n-1}^k\right]
\end{aligned}$$

$$\Rightarrow \qquad \mathcal{I}_n^k = \frac{kn}{kn+1}\mathcal{I}_{n-1}^k$$

Comme $\mathcal{I}_0^k = \int_0^1 dx = 1$, on obtient : $\quad \mathcal{I}_n^k = \prod_{i=1}^{n} \frac{ki}{ki+1}$ ◇

Remarque : en faisant le changement de variables $u = x^k$, on peut exprimer $\mathcal{I}_n^k$ par rapport à la fonction **Beta** $\mathcal{B}()$:

$$\mathcal{I}_n^k = \frac{1}{k}\int_0^1 u^{\frac{1}{k}-1}(1-u)^{(n+1)-1}du = \frac{1}{k}\mathcal{B}(\frac{1}{k}, n+1) = \frac{\Gamma\left(\frac{1}{k}\right)\Gamma(n+1)}{k\Gamma\left(\frac{1}{k}+n+1\right)}$$

$$\mathcal{I}_n^k = \frac{\Gamma(\frac{1}{k})n!}{k\left[\prod_{i=0}^{n}(\frac{1}{k}+i)\right]\Gamma(\frac{1}{k})} = \frac{n!}{\prod_{i=1}^{n}(\frac{1}{k}+i)} = \frac{\prod_{i=1}^{n} i}{\prod_{i=1}^{n}\frac{ki+1}{k}} = \prod_{i=1}^{n}\frac{ki}{ki+1}$$

Corollaire 6.5.1 *Soit n et k des entiers tels que $n \geq 1$ et $k \geq 2$. On a :*

$$\mathcal{I}_n^k = \int_0^1 \left(1 - x^k\right)^n dx < \frac{1}{\sqrt[k]{n+1}}$$

Dém : on va démontrer ce résultat par récurrence sur n, k étant fixé.

pour $n = 1$:

$$\mathcal{I}_1^k < \frac{1}{\sqrt[k]{2}} \Leftrightarrow \frac{k}{k+1} < \frac{1}{\sqrt[k]{2}} \Leftrightarrow (k+1)^k > 2 \times k^k$$

$$\Leftrightarrow k^k + kk^{k-1} + C_k^2 k^{k-2} + \cdots > k^k + k^k$$

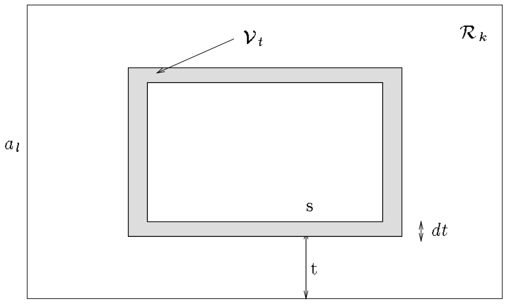


Figure 6.6: *Espérance du nombre de sites inachevés dans un k-rectangle.*

On suppose la propriété vraie pour $(n-1)$ et on montre qu'elle est vraie pour n :

$$\mathcal{I}_n^k < \frac{1}{\sqrt[k]{n+1}} \Leftrightarrow \frac{kn}{kn+1}\mathcal{I}_{n-1}^k < \frac{1}{\sqrt[k]{n+1}}$$

$$\Leftrightarrow \mathcal{I}_{n-1}^k < \frac{kn+1}{kn}\frac{1}{\sqrt[k]{n+1}}$$

Par hypothèse de récurrence, $\mathcal{I}_{n-1}^k < \frac{1}{\sqrt[k]{n}}$, d'où :

$$\frac{1}{\sqrt[k]{n}} < \frac{kn+1}{kn}\frac{1}{\sqrt[k]{n+1}} \Leftrightarrow \left(\frac{kn+1}{kn}\right)^k > \frac{n+1}{n}$$

$$\Leftrightarrow (kn+1)^k > k^k n^k + k^k n^{k-1}$$

$$\Leftrightarrow k^k n^k + k k^{k-1} n^{k-1} + C_k^2 k^{k-2} n^{k-2} + \cdots > k^k n^k + k^k n^{k-1}$$

$\diamond$

Remarque (Luc Devroye) : en utilisant la convexité de la fonction $\log \Gamma()$, on obtient $\dfrac{\Gamma(n+1)}{\Gamma(n+1+1/k)} \geq \dfrac{1}{(n+1)^{1/k}}$. Donc, $\mathcal{I}_n^k$ et le majorant proposé $(\dfrac{1}{\sqrt[k]{n+1}})$ diffèrent d'un facteur inférieur à $\dfrac{1}{\Gamma(1+1/k)}$ (la borne est atteinte pour $k=1$ et $k \to +\infty$). Quand k varie entre 1 et $+\infty$, $1 \leq \dfrac{1}{\Gamma(1+1/k)} < 1,15$.

L'erreur dans la majoration de $\mathcal{I}_n^k$ n'excède donc jamais 15%.

Théorème 6.5.1 *On se place dans les conditions du Lemme 6.4.1. Soit $\mathcal{S}_k$ la surface de $\mathcal{R}_k$ et $E(\mathcal{R}_k)$ l'espérance du nombre de sites inachevés dans $\mathcal{R}_k$, alors :*

$$E(\mathcal{R}_k) \leq \frac{n}{\sqrt[k]{n}} \frac{c_2}{\sqrt[k]{c_1}} \mathcal{S}_k \frac{k 2^{k+1} \sqrt{\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor} \sqrt[k]{k\Gamma\left(\frac{k}{2}+1\right)}}{\sqrt{\pi}}$$

Dém : soient a_1, a_2, $\ldots$, a_k les longueurs des côtés du k-rectangle $\mathcal{R}_k$. Soit a_l la longueur du plus petit côté de $\mathcal{R}_k$. Considérons le volume élémentaire $\mathcal{V}_t$ constitué des points de $\mathcal{R}_k$ situés à une distance comprise entre t et $t+dt$ de la frontière de $\mathcal{R}_k$ (avec $0 \leq t \leq \frac{a_l}{2}$) (Figure 6.6). Appelons $\mathcal{S}_t$ la surface externe de $\mathcal{V}_t$:

$$\mathcal{S}_t = 2\sum_{j=1}^{k}\left(\prod_{i\neq j}(a_i - 2t)\right) \leq \mathcal{S}_k = 2\sum_{j=1}^{k}\left(\prod_{i\neq j} a_i\right)$$

$$\Rightarrow \mathcal{V}_t \leq \mathcal{S}_t \times dt \leq \mathcal{S}_k \times dt$$

La probabilité qu'un site donné appartienne à $\mathcal{V}_t$ est :

$$\int_{\mathcal{V}_t} f \leq c_2 \times \mathcal{V}_t \leq c_2 \times \mathcal{S}_k \times dt$$

En utilisant le Lemme 6.4.1, on peut estimer :

$$\begin{aligned} E(\mathcal{R}_k) &\leq n\int_0^{\frac{a_l}{2}} k2^k\left(1 - \frac{c_1 \times \pi^{\frac{k}{2}} \times t^k}{k \times 2^k \times (\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor)^{\frac{k}{2}} \times \Gamma\left(\frac{k}{2}+1\right)}\right)^{n-1} c_2\mathcal{S}_k dt \\ &= n\frac{c_2}{\sqrt[k]{c_1}}\mathcal{S}_k\frac{k2^{k+1}\sqrt{\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor}\sqrt[k]{k\Gamma\left(\frac{k}{2}+1\right)}}{\sqrt{\pi}} \times \\ &\quad \int_0^{\frac{a_l\sqrt{\frac{\pi}{4}}\sqrt[k]{c_1}}{\sqrt{\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor}\sqrt[k]{k\Gamma(\frac{k}{2}+1)}}} \left(1 - x^k\right)^{n-1} dx \end{aligned}$$

$$\text{avec } x = t\sqrt[k]{\frac{c_1\pi^{k/2}}{k2^k\left(\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor\right)^{k/2}\Gamma\left(\frac{k}{2}+1\right)}}.$$

Puisque $a_l \leq 1$, $c_1 \leq 1$ et $k \geq 2$, on a :

$$\begin{aligned} E(\mathcal{R}_k) &\leq n\frac{c_2}{\sqrt[k]{c_1}}\mathcal{S}_k\frac{k2^{k+1}\sqrt{\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor}\sqrt[k]{k\Gamma\left(\frac{k}{2}+1\right)}}{\sqrt{\pi}} \times \int_0^1 \left(1 - x^k\right)^{n-1} dx \\ &\leq \frac{n}{\sqrt[k]{n}}\frac{c_2}{\sqrt[k]{c_1}}\mathcal{S}_k\frac{k2^{k+1}\sqrt{\lceil\frac{k+1}{2}\rceil\lfloor\frac{k+1}{2}\rfloor}\sqrt[k]{k\Gamma\left(\frac{k}{2}+1\right)}}{\sqrt{\pi}} \end{aligned}$$

en utilisant la majoration d'intégrale du Corollaire 6.5.1. Donc,

$$E(\mathcal{R}_k) = O\left(\frac{n}{\sqrt[k]{n}} \times \frac{c_2}{\sqrt[k]{c_1}} \times \mathcal{S}_k\right) = O(n^{1-1/k})$$

$\diamond$

Remarques: cette borne est meilleure que celle obtenue par Katajainen et Koppinen dans le plan [117]. Luc Devroye me signale que le théorème 6.5.1 devrait se généraliser à un ensemble convexe quelconque. Dans ce cas, $\mathcal{S}_k$ serait la surface de l'ensemble.

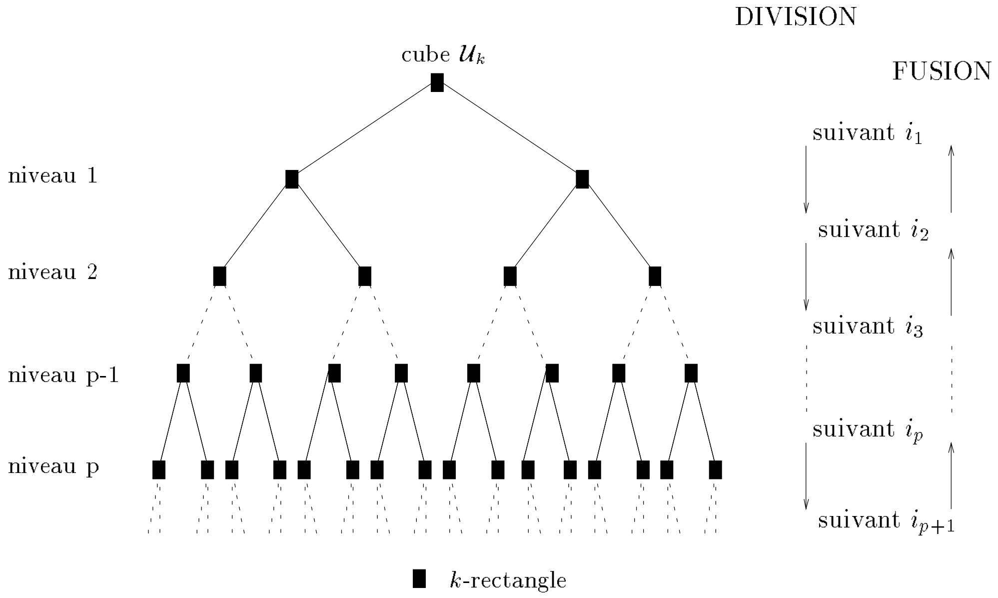


Figure 6.7: *Arborescence schématisant les divisions et les fusions selon un kd-arbre.*

6.6 Analyse d'une classe de fusions k-dimensionnelles utilisant la notion de site inachevé

Dans ce paragraphe, nous montrons comment les résultats précédents peuvent être utilisés pour analyser une certaine classe d'algorithmes *Divide-and-Conquer* en dimension quelconque (voir figures 6.7 et 6.8). On trouve déjà dans les chapitres précédents une application au calcul de la triangulation de Delaunay euclidienne en dimension 2, qui se révèle être, en pratique, très performante.

Théorème 6.6.1 *Considérons un ensemble S de n points répartis dans un hypercube unitaire $\mathcal{U}_k$ de dimension k, avec une densité de probabilité quasi-uniforme f, de bornes c_1 et c_2 ($c_1 \leq c_2$). Supposons qu'un algorithme fonctionne selon le schéma suivant :*

division: *$\mathcal{U}_k$ est divisé en cellules contenant un seul point en utilisant un kd-tree équilibré ([16]),*

fusion: *$\mathcal{U}_k$ est reconstitué par fusions successives dans l'ordre inverse du découpage.*

Si la fusion de deux sous-ensembles prend un temps proportionnel au nombre de sites inachevés (par rapport à $\mathcal{U}_k$), alors la complexité en moyenne de l'ensemble du processus de fusion sera proportionnelle à n.

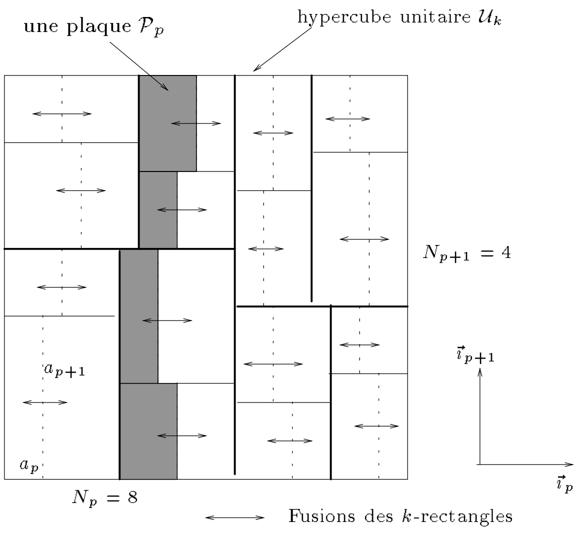


Figure 6.8: *Fusions au niveau p : la plaque $\mathcal{P}_p$ (en grisé) avec les hyperrectangles qui la constituent.*

Dém : d'après le Théorème 6.5.1, la complexité en moyenne de l'ensemble du processus de fusion dans l'algorithme est en :

$$\frac{n}{\sqrt[k]{n}} \frac{c_2}{\sqrt[k]{c_1}} \mathcal{S}_k \frac{k 2^{k+1} \sqrt{\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor} \sqrt[k]{k \Gamma\left(\frac{k}{2}+1\right)}}{\sqrt{\pi}} \sum_{k-rect.} \mathcal{S}$$

On est donc ramené à évaluer $\sigma = \sum_{k-rect.} \mathcal{S}$, cette quantité représentant la somme des surfaces de tous les k-rectangles intervenant dans les fusions. On va décomposer σ en sous-sommes σ_p, une pour chaque niveau p de fusionnement (voir Figure 6.7).

Soit $(\vec{\imath}_1, \ldots, \vec{\imath}_k)$ une base orthonormée telle que chaque axe soit orthogonal à une hyperface de $\mathcal{U}_k$. Le kd-tree crée une partition de $\mathcal{U}_k$, par n k-rectangles.

Définition 6.6.1 *On appelle plaque $\mathcal{P}_j$ relative à la direction $\vec{\imath}_j$, un sous-ensemble parmi les hyperrectangles pavant $\mathcal{U}_k$, tel que la projection orthogonale de $\mathcal{P}_j$ sur une hyperface de $\mathcal{U}_k$ orthogonale à $\vec{\imath}_j$, constitue un pavage de cette hyperface (voir Figure 6.8).*

Pendant la création du kd-arbre, la division de l'espace – orthogonalement à une direction – double le nombre de plaques relatives à cette direction. On remarque aussi que le pavage de $\mathcal{U}_k$ peut être partitionné en plaques de même direction, et ceci quelle que soit cette direction (parmi les axes du repère). On appelle N_j le nombre de plaques relativement à la direction $\vec{\imath}_j$.

On considère le niveau p de fusionnement. Excepté pour le dernier niveau – pour lequel cette valeur constitue un majorant de N_j – un simple raisonnement par récurrence donne :

$$N_j = 2^{\lceil \frac{p-j+1}{k} \rceil}$$

Calculons maintenant σ_p, en regroupant les k-rectangles en plaques, et ceci relativement à toutes les directions. Etant donné que la surface d'une hyperface de $\mathcal{U}_k$ vaut 1, la somme cherchée est égale à $2 \times \sum_{j=1}^{k} N_j$, d'où :

$$\frac{1}{2} \times \sigma_p \leq \sum_{j=1}^{k} 2^{\lceil \frac{p-j+1}{k} \rceil} \leq k \times 2^{\lceil \frac{p}{k} \rceil}$$

En additionnant sur les h niveaux de fusionnement, on a :

$$\frac{1}{2} \times \sigma = \frac{1}{2} \times \sum_{p=1}^{h} \sigma_p \leq \sum_{p=1}^{h} k \times 2^{\lceil \frac{p}{k} \rceil} = k \times \sum_{p=1}^{h} 2^{\lceil \frac{p}{k} \rceil}$$

$$\leq k \left[k \times \sum_{w=1}^{\lceil \frac{h}{k} \rceil} 2^w \right] \leq k^2 \times 2^{\lceil \frac{h}{k} \rceil + 1} \leq 4k^2 \times 2^{\lceil \frac{h}{k} \rceil - 1}$$

Puisque l'arborescence – schématisant le découpage par le kd-arbre – est équilibrée, sa hauteur h vérifie :

$$h = \lceil \log_2 n \rceil \Rightarrow 2^{h-1} < n \leq 2^h$$

$$\Rightarrow n > 2^{h-1} \Rightarrow \sqrt[k]{n} > 2^{\frac{h-1}{k}} \geq 2^{\lceil \frac{h}{k} \rceil - 1}$$

$$\text{d'où } \sigma = \sum_{k-rect.} \mathcal{S} \leq 8k^2 \sqrt[k]{n}$$

Cela implique que la complexité en moyenne de l'ensemble du processus de fusion est linéaire en fonction du nombre de sites. Elle est plus précisément proportionnelle à :

$$O \left(n \frac{c_2}{\sqrt[k]{c_1}} \frac{k^3 2^{k+4} \sqrt{\lceil \frac{k+1}{2} \rceil \lfloor \frac{k+1}{2} \rfloor} \sqrt[k]{k \Gamma \left(\frac{k}{2} + 1 \right)}}{\sqrt{\pi}} \right)$$

$\diamond$

6.7 Conclusion

Le calcul de l'espérance du nombre de sites inachevés dans un k-rectangle, avec une distribution quasi-uniforme, est un résultat théorique important en soi, qui peut être utilisé pour analyser divers algorithmes, algorithmes qui doivent toutefois être plus ou moins liés à la triangulation de Delaunay car le concept de *site inachevé* n'a de sens que par l'intermédiaire d'une triangulation de Delaunay. Ces résultats peuvent s'ajouter aux résultats probabilistes déjà obtenus par Bern, Eppstein et Yao [28] sur la triangulation de Delaunay en dimension quelconque.

On pourrait essayer de démontrer ces résultats sur d'autres types de distributions (loi normale par exemple). Mais, l'intérêt majeur est dans l'application au calcul de la triangulation de Delaunay en dimension quelconque avec un algorithme Divide-and-Conquer sur un kd-tree. Cette méthode a déjà fait ses preuves en dimension 2. L'algorithme est facile à implémenter et il est, comme le confirme le chapitre 7, parmi les plus rapides pour des distributions classiques. Un intéressant sujet de recherche consisterait à appliquer cette méthode en dimension quelconque pour obtenir, après un tri selon plusieurs directions, une triangulation dont le temps de calcul serait en moyenne linéaire par rapport au nombre de sites. La difficulté se trouve dans la fusion des sous-triangulations. Cet algorithme pourrait concurrencer celui de Dwyer ([71]), qui utilise un prépartitionnement des points selon une grille régulière, ce qui réduit considérablement les possibilités d'utilisation d'un tel algorithme.

On pourrait aussi se demander si, en dimension k, la connaissance du tri selon k directions d'un ensemble de points $\mathcal{S}$, ne permettrait pas d'abaisser la borne inférieure pour la complexité dans le pire des cas de la triangulation de Delaunay de $\mathcal{S}$. Ce problème a été évoqué il y a quelques années par O. Devillers (INRIA, Sophia Antipolis, France) en dimension 2. J'ai pu montrer que, pour un ordre fixé en x et y, il existe autant de triangulations de Delaunay topologiquement différentes de n sites que de façons de trianguler un polygone convexe, soit le nombre de Catalan [67] $C_{n-2} = \binom{2n-2}{n-2}\frac{1}{n-1}$. Si cette combinatoire est maximale (je n'ai pas réussi à le montrer), cela laisse supposer qu'il peut exister un algorithme linéaire dans le pire des cas pour calculer une triangulation de Delaunay quand on connaît les deux ordres en x et en y. Il existe bien un algorithme linéaire pour trianguler un polygone convexe ou même un polygone simple quelconque [45]. Il est aussi très probable que cet algorithme, s'il existe, sera très sophistiqué ! Cela reste actuellement un problème ouvert, quelle que soit la dimension de l'espace (excepté le cas trivial de la dimension 1 où la triangulation de Delaunay consiste à relier les points selon l'ordre dans lequel ils sont triés). Malheureusement, la méthode proposée dans ce chapitre ne permet pas d'abaisser cette borne inférieure, même dans le plan. Par contre, si l'on utilise la distance de Minkowski associée à un polygone convexe (triangle en particulier, distance $L_1, \ldots$), Chew et Fortune [51] ont montré que le calcul de la triangulation de Delaunay dans le plan peut se faire en $O(n \log \log n)$ après un tri des sites selon les abscisses et les ordonnées.

Chapitre 7

COMPARAISON EXPERIMENTALE D'ALGORITHMES DE TRIANGULATION DE DELAUNAY

Ce chapitre contient une comparaison expérimentale de nombreux algorithmes de triangulation de Delaunay. Nous affinons l'étude faite en 1995 par Su et Drysdale [187] [188]. Nous conservons les meilleurs algorithmes (d'après leur étude) et les comparons à ceux présentés dans ce mémoire. Pour cela, nous les avons tous recodés[1] et testés sur des ensembles de sites plus importants (jusqu'à 8 millions), avec de nombreux types de distributions. Nos conclusions diffèrent quelque peu de celles de Su et Drysdale.

Notons aussi la récente implémentation de l'algorithme basé sur le 2d-tree par Schewchuk [182]. Ses résultats expérimentaux sont comparables aux nôtres. La comparaison qu'il effectue entre l'algorithme de Lee et Schachter, celui basé sur le 2d-tree et l'algorithme de balayage confirme nos conclusions.

7.1 La machine utilisée

Les tests ont été réalisés sur un supercalculateur CONVEX C3, machine extrêmement fiable. Il dispose de 2 GigaOctets de mémoire vive, ce qui a permis de trianguler 2^{23} sites (plus de 8 millions) avec certains algorithmes. Il n'utilise pas de mémoire paginée, ce qui rend les mesures de temps d'exécution assez fiables (peu de variations entre

[1] Il est important que les programmes soient réalisés par la même personne. Su et Drysdale ont utilisé des codes conçus de manière différente par des personnes différentes. Ceci pose problème car les temps de calcul sont fortement liés à la manière de programmer et en particulier à la façon de gérer la mémoire.

deux exécutions d'un même programme sur le même jeu d'essai). Par contre, la vitesse de son processeur est faible. Elle est à peu près équivalente à celle d'une station de travail bas de gamme. Les temps d'exécution sont donc plus longs que ceux obtenus sur n'importe quelle station de travail (voir paragraphe 7.4.5). Ceci n'est pas un problème car l'objectif de ce chapitre est la comparaison des différents algorithmes.

7.2 Les algorithmes testés

Su et Drysdale [187] [188] ont conclu en 1995 que l'algorithme de Dwyer [70] (voir paragraphe 1.4.6) est le plus rapide et le plus résistant aux distributions non uniformes. Nous avons bien sûr choisi cet algorithme que nous appelons **Dw**. Nous l'avons codé très strictement[2].

Su et Drysdale n'ont pas testé l'algorithme de Koppinen et Katajainen [117] mais ils pensent, en se référant aux conclusions de Koppinen et Katajainen, que ses performances sont comparables à celui de Dwyer. Nous avons codé l'algorithme dirigé par un bucket-tree (voir paragraphe 5.7), appelé **bt**, qui, dans le cas d'une distribution dans un carré, est très proche de celui de Koppinen et Katajainen. Il est toujours au moins aussi bon, parce qu'il s'adapte aux distributions dans des domaines rectangulaires et que le nombre moyen de sites par cellule est toujours 1. On a vérifié expérimentalement que, plus le nombre moyen m_s de sites par cellule est grand, plus l'algorithme est lent. Katajainen et Koppinen font aussi cette remarque. Dans leur algorithme, m_s varie aléatoirement entre 1 et 4 parce qu'ils choisissent toujours un nombre de cellules qui est une puissance de 4 et donc l'ajustement est moins bon.

Nous avons codé les nouveaux algorithmes présentés dans ce mémoire, qui sont basés sur des arbres bidimensionnels. Le principe est de découper le domaine suivant un arbre puis de fusionner les cellules selon deux directions. Le programme s'appelle **2d** si l'on utilise un 2d-tree (voir paragraphe 5.2), **r2d** si l'on utilise un random 2d-tree (voir paragraphe 5.3), **a2d** si l'on utilise un adaptive 2d-tree (voir paragraphe 5.4), **ar2d** si l'on utilise un adaptive random 2d-tree (voir paragraphe 5.5).

Nous avons aussi codé **swap** qui est un nouvel algorithme à base d'échanges d'arêtes (voir paragraphe 1.4.1 et annexe C.1). Une triangulation est obtenue à l'aide d'un 2d-tree, puis on la transforme en triangulation de Delaunay en "échangeant" des arêtes.

[2] R. Dwyer n'en a codé qu'une version simplifiée où il triangule directement avec l'algorithme de Lee et Schachter les lignes de la grille, alors que cette triangulation directe ne doit s'appliquer qu'aux cellules de la grille. En revanche, nous avons codé l'algorithme exact : les cellules produites par le hachage sont triangulées selon Lee et Schachter, puis les cellules sont fusionnées verticalement par paires jusqu'à la triangulation complète des colonnes, et enfin les colonnes sont fusionnées horizontalement par paires jusqu'à la triangulation complète du semis. Mais, la différence de temps d'exécution est probablement très faible.

Le programme **LS** a été codé par J.M. Moreau d'après l'algorithme de Lee et Schachter [131] (voir paragraphe 5.1). On peut considérer que les algorithmes fondés sur des arbres bidimensionnels sont des optimisations de l'algorithme de Lee et Schachter.

Nous utilisons aussi **bal** le programme de D. Hatch, prêté à l'Ecole des Mines pour des tests. C'est une implémentation de l'algorithme de balayage de Seidel [175] (voir paragraphe 1.4.5) où la gestion des événements est accélérée par des techniques de hachage.

Nous avons aussi testé **Dt** qui utilise l'arbre de Delaunay [33] (voir paragraphe 1.4.2), implémenté par l'INRIA, en choisissant la version semi-dynamique[3] de cet algorithme incrémental (la version dynamique est plus lente et plus gourmande en espace mémoire).

Remarques : sauf pour les programmes que nous n'avons pas réalisés (**bal** et **Dt**), on a essayé de réutiliser au maximum les mêmes composants logiciels et de factoriser au maximum la programmation. En particulier, les structures de données sont identiques, les programmes utilisent la même routine *angle* (resp. *incircle*) pour connaître la position d'un point par rapport à un segment (resp. cercle) et les fonctions utilisées pour fusionner les sous-triangulations sont identiques. Ceci rend les comparaisons plus fiables.

7.3 Les types de distributions utilisées

Pour effectuer les tests, on a fait varier le nombre de sites suivant les puissances de 2, jusqu'à atteindre environ 8 millions pour certains programmes, la limite étant la mémoire vive de la machine. Les programmes ont été testés non seulement sur des distributions uniformes (dans un carré et dans un disque) qui privilégient beaucoup certains algorithmes (en particulier ceux qui utilisent des techniques de hachage), mais aussi sur des distributions non uniformes (au voisinage d'un segment avec différentes orientations, d'une croix, d'un arc de cercle, d'un anneau, des coins d'un carré, suivant une loi normale, avec de fortes concentrations aléatoires). En effet, en pratique, les distributions sont rarement uniformes : par exemple, dans la représentation d'un terrain naturel, la densité des sites est faible dans les parties plates, importantes dans les régions accidentées.

Tous les types de distributions utilisés dans l'étude de Su et Drysdale [188] ont été repris ci-dessous. On note $U(a,b)$ la distribution uniforme sur le segment $[a,b]$ et $N(m,\sigma)$ la distribution normale de moyenne m et d'écart-type σ.

[3]Un algorithme *semi-dynamique* ne permet que l'insertion incrémentale des points alors qu'un algorithme *dynamique* en autorise aussi la suppression.

unif distribution uniforme dans un carré unitaire, sites en $(U(0,1), U(0,1))$.

ball distribution uniforme dans un disque unitaire, sites en $(x = U(0,1) - 0.5, y = U(0,1) - 0.5)$ avec $\sqrt{x^2 + y^2} \leq 1$.

corn distribution au voisinage des coins d'un carré, $\frac{n}{4}$ sites en $(U(0,0.01), U(0,0.01))$, $\frac{n}{4}$ sites en $(U(0.99,1), U(0,0.01))$, $\frac{n}{4}$ sites en $(U(0,0.01), U(0.99,1))$, $\frac{n}{4}$ sites en $(U(0.99,1), U(0.99,1))$.

diam distribution au voisinage d'un segment oblique, sites en (x, y) avec $t = U(0,1)$, $x = t + U(0,0.01) - 0.005$, $y = t + U(0,0.01) - 0.005$.

rect distribution au voisinage d'un segment horizontal, sites en $(U(0,1) + U(0,0.01) - 0.005, U(0,0.01) - 0.005)$.

cross distribution au voisinage d'une croix, $\frac{n}{2}$ sites en $(U(0,1), 0.495 + U(0,0.01))$, $\frac{n}{2}$ sites en $(0.495 + U(0,0.01), U(0,1))$.

arc distribution au voisinage d'un arc de cercle, sites en (x, y) avec $t = U(0, \frac{\pi}{2})$, $x = \cos t + U(0,0.01) - 0.005$, $y = \sin t + U(0,0.01) - 0.005$.

ann distribution dans un anneau, sites en (x, y) avec $t = U(0, 2\pi)$, $x = \cos t + U(0,0.01) - 0.005$, $y = \sin t + U(0,0.01) - 0.005$.

norm distribution suivant la loi normale centrée d'écart-type 0.01, sites en $(N(0,0.01), N(0,0.01))$.

clus distribution avec "clusters", $N(0,0.01)$ en 10 points aléatoires dans le carré unitaire.

Remarques :

1. pour créer la distribution uniforme, il faut choisir un générateur de nombres aléatoires en type *double*, étant donné que les ensembles testés atteignent 8 millions de sites. Sinon, on obtient beaucoup de points confondus et la distribution n'est même plus vraiment uniforme.

2. on peut générer une distribution normale centrée réduite par :

$$\left\{ \begin{array}{lcl} X & = & \sqrt{-2\ln U_1}\cos(2\pi U_2) \\ Y & = & \sqrt{-2\ln U_1}\sin(2\pi U_2) \end{array} \right. \quad \texttt{avec} \quad \left\{ \begin{array}{lcl} U_1 & = & U(0,1) \\ U_2 & = & U(0,1) \end{array} \right.$$

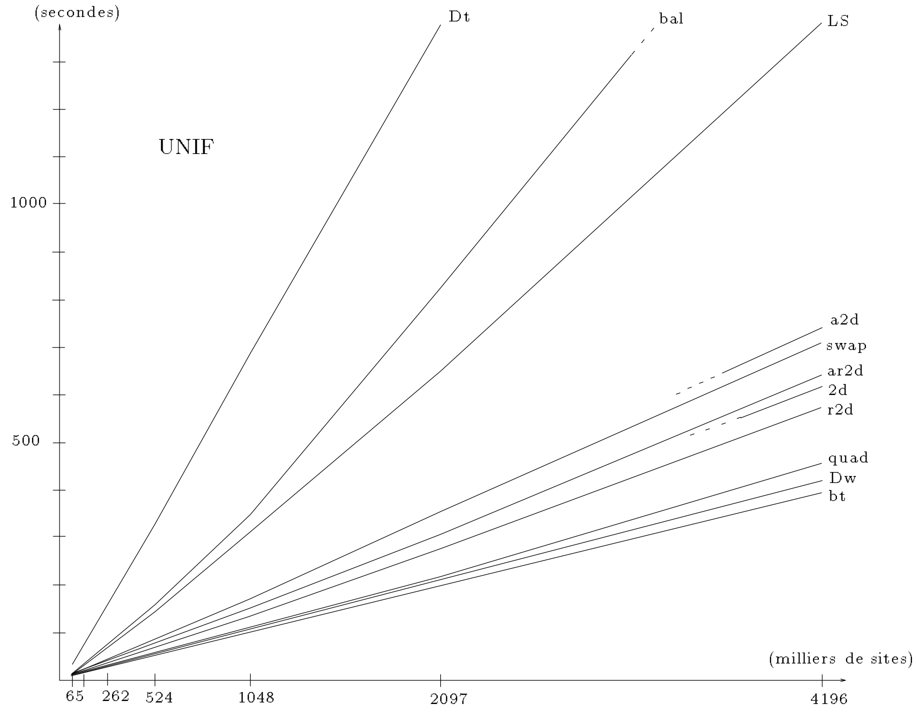


Figure 7.1: *Triangulation de Delaunay avec une distribution uniforme.*

7.4 Résultats expérimentaux et analyse

Le temps total de calcul de la triangulation est la somme du temps de prétraitement et du temps réel de triangulation (on ne tient pas compte du temps de lecture des coordonnées des sites sur disque, qui est le même quel que soit l'algorithme) :

$$T_{total} = T_{pretrait.} + T_{triangul.}$$

En général, on peut diminuer $T_{triangul.}$ à l'aide d'un prétraitement plus long. Un équilibre est à trouver : c'est aussi l'objet de ce chapitre. Dans un premier temps, on va comparer le temps total de triangulation pour différents algorithmes sur diverses distributions. Dans un second temps, on va comparer les temps de triangulation, donc étudier comment le prétraitement peut diminuer le temps de triangulation.

7.4.1 Comparaison des temps totaux de triangulation

Les figures 7.1 et 7.2 comparent les différents algorithmes sur une distribution uniforme, puis sur une distribution non uniforme avec "clusters", c'est-à-dire avec de fortes densités de sites en une dizaine de points aléatoires. L'annexe A.1 contient des comparaisons identiques sur d'autres types de distributions.

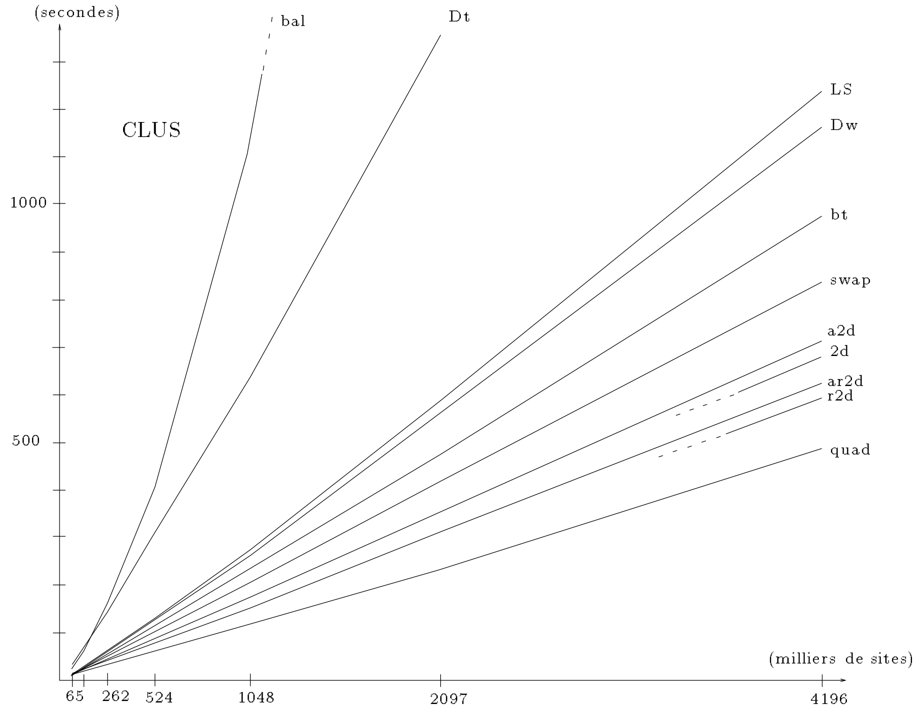


Figure 7.2: *Triangulation de Delaunay avec une distribution en clusters.*

Un des algorithmes les plus lents est l'algorithme de balayage **bal** qui est, de plus, peu stable en fonction des distributions. Une version accélérée de cet algorithme a été mise au point par Kauffmann et Spehner [116]. Ses auteurs prétendent qu'elle est un peu plus rapide que l'algorithme de Lee et Schachter, mais qu'elle reste néanmoins plus lente que la méthode basée sur le 2d-tree [3] [4]. G.Leach [129] a aussi codé une version rapide de l'algorithme de Fortune. Il prétend qu'elle est légèrement plus performante que celle de Lee et Schachter. Su et Drysdale [187] [188] concluent à l'infériorité de la méthode de balayage par rapport à celle de Dwyer.

L'autre algorithme lent est **Dt**, fondé sur l'arbre de Delaunay. Toutefois, son comportement est très stable vis-à-vis de l'hétérogénéité de la distribution. Cet algorithme est aussi le seul à être dynamique. Une comparaison expérimentale d'algorithmes *dynamiques* de triangulation de Delaunay est faite dans le chapitre 9.

Parmi les algorithmes Divide-and-Conquer, on constate que l'algorithme **bt** (proche de celui de Katajainen et Koppinen) est toujours légèrement meilleur que celui de Dwyer, lui-même nettement meilleur que celui de Lee et Schachter, et ceci quelle que soit la distribution.

L'algorithme **swap** est toujours un peu plus lent que l'algorithme dirigé par le 2d-tree. Il offre donc peu d'intérêt.

En ce qui concerne les algorithmes fondés sur des 2d-trees, on constate que les versions randomisées sont toujours plus rapides (**r2d** plus rapide que **2d**, **ar2d** plus rapide que **a2d**). La version dirigée par l'adaptive 2d-tree est légèrement meilleure que celle dirigée par le simple 2d-tree quand la distribution est très fortement irrégulière (distribution au voisinage d'un segment horizontal, d'une croix). Les algorithmes basés sur les 2d-trees offrent une bonne stabilité vis-à-vis de l'hétérogénéité de la distribution.

L'algorithme **quad**, basé sur un quadtree, a quasiment le même comportement quel que soit le type de distribution. Il est remarquablement performant, le plus rapide sur les distributions non uniformes et presque le plus rapide sur les distributions uniformes bien que sa complexité dans le pire des cas soit la moins bonne.

bt est légèrement meilleur que **quad** sur des distributions uniformes, mais ses performances s'effondrent sur une distribution en clusters. Pour ce type de distributions, les algorithmes basés sur les 2d-trees sont meilleurs que **bt**. Ce comportement était prévisible car les algorithmes utilisant des techniques de hachage ne sont très bons que sur les distributions uniformes.

En conclusion, l'algorithme basé sur le quadtree est le meilleur sur cet éventail de distributions, suivi par les algorithmes dirigés par les 2d-trees. L'algorithme fondé sur le bucket-tree n'est très bon que sur des distributions uniformes.

7.4.2 Comparaison des temps de triangulation sans le prétraitement

On examine maintenant l'influence des différents prétraitements sur les algorithmes de type Divide-and-Conquer. On mesure le temps de triangulation en comptant le nombre total d'arêtes créées. En effet, ce nombre est proportionnel au temps de triangulation et offre plus de précision et de fiabilité que les mesures de temps. Le temps de prétraitement est la différence entre le temps total du paragraphe précédent et le temps de calcul de la triangulation. Les figures 7.3 et 7.4 comparent les différents algorithmes sur une distribution uniforme, puis sur une distribution non uniforme avec “clusters”. On trouve dans l'annexe B.1 des tests identiques sur d'autres types de distributions.

L'algorithme **LS** de Lee et Schachter [131] est le moins bon. Son comportement est en $\Theta(n \log n)$, même sur des distributions uniformes. Ceci corrobore la théorie. En effet, Oya, Iri et Murota ont montré que la phase de triangulation, dans cet algorithme, est, même en moyenne, en $\Theta(n \log n)$.

En ce qui concerne les algorithmes utilisant une grille régulière (**Dw** et **bt**), leur comportement est linéaire pour une distribution uniforme. Par contre, pour une distribution en “clusters”, leur comportement semble être en $\Theta(n \log n)$. Ceci s'explique : la distribution n'étant pas uniforme, certaines cellules contiennent un très grand nombre

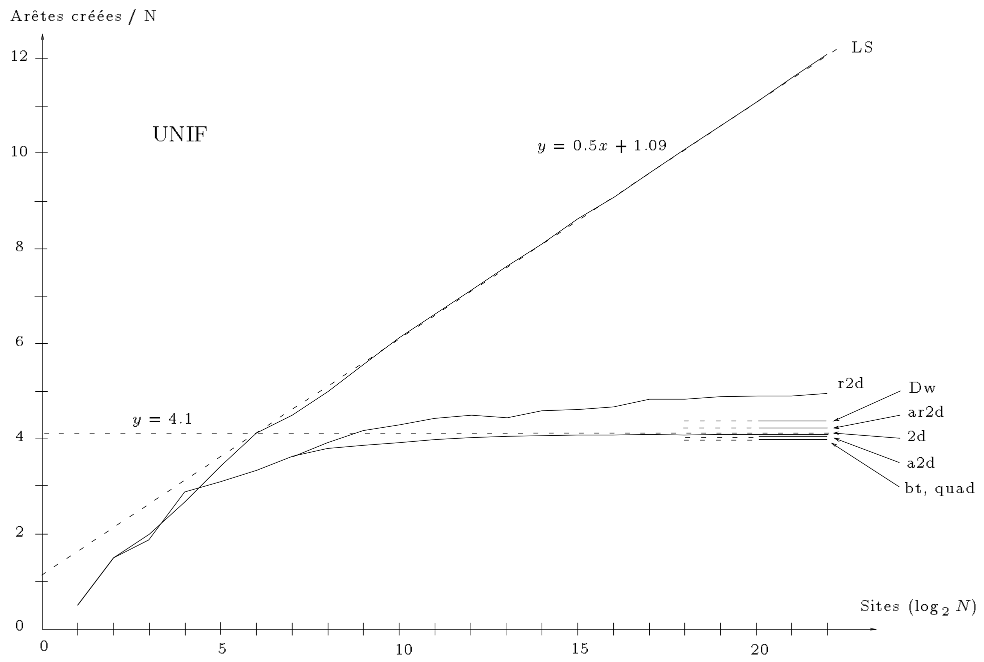


Figure 7.3: *Nombre d'arêtes créées avec une distribution uniforme.*

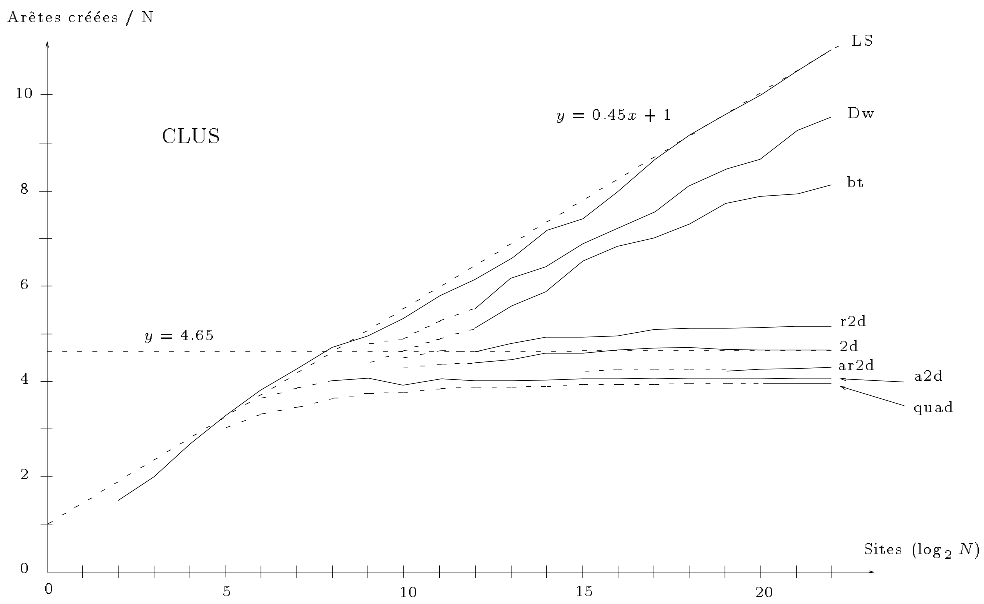


Figure 7.4: *Nombre d'arêtes créées avec une distribution en clusters.*

de sites et sont triangulées directement avec l'algorithme de Lee et Schachter. Ainsi, des composantes en $\Theta(n \log n)$ vont devenir perceptibles.

Le comportement des algorithmes dirigés par les 2d-trees semble être linéaire quelle que soit la distribution (parmi celles testées). Pour une distribution uniforme, ces résultats corroborent la théorie. Pour les autres distributions, les résultats expérimentaux montrent que ces algorithmes ont une grande capacité à s'adapter à l'hétérogénéité de la distribution. Celui basé sur l'adaptive 2d-tree (et aussi sur le random adaptive 2d-tree) a presque le même comportement quelle que soit la distribution, et le nombre total d'arêtes créées est proche de $4n$, le nombre d'arêtes détruites voisin de n. Ce bon comportement s'explique de par la forme presque carrée des cellules fusionnées. Le comportement du 2d-tree est un peu moins bon. Comme il alterne systématiquement la direction de division, cela peut produire, sur des distributions très irrégulières, des cellules fortement aplaties qui vont générer des triangles eux-mêmes fortement aplatis et qui ont peu de chances d'être des triangles de Delaunay. Mais il faut être conscient que la construction de l'adaptive 2d-tree est plus longue (voir annexe E.1) et que, globalement, l'algorithme fondé sur l'adaptive 2d-tree est plus lent, sauf pour des distributions pathologiques. Les versions randomisées sont moins efficaces car les cellules ont une forme moins carrée. Par contre, la construction de l'arbre randomisé étant plus rapide, on y gagne souvent globalement. En mesurant les temps de construction des 2d-trees, on constate que le type de distribution (parmi celles testées) a peu d'influence sur le temps de construction. La construction est la plus rapide sur certaines distributions non uniformes, et presque la plus lente sur la distribution uniforme (voir annexe E.1).

Le meilleur algorithme est celui basé sur le quadtree. Il divise récursivement l'espace en carrés. Le bon comportement de l'algorithme est dû à la forme carrée des cellules : les triangles construits auront rarement une forme aplatie (leur forme sera déjà proche de celle du triangle équilatéral) et ils seront déjà de bons candidats pour être des triangles de Delaunay du maillage final. D'autre part, le majorant obtenu (voir paragraphe 6.6) pour la complexité en moyenne de la phase de triangulation est proportionnel au périmètre des cellules, il est donc minimal pour des cellules de forme carrée. L'algorithme basé sur l'adaptive 2d-tree est un peu moins bon (en ce qui concerne la phase de triangulation). Lui aussi essaie de décomposer le domaine en cellules de la forme la plus carrée possible, mais cette décomposition ne produit pas des carrés parfaits comme avec le quadtree.

7.4.3 Comparaison de l'espace mémoire utilisé

L'algorithme basé sur le 2d-tree (ou celui de Lee et Schachter) est celui qui consomme le moins d'espace mémoire. On peut calculer la triangulation de Delaunay avec une consommation minimale de 88 octets par site en représentant les coordonnées d'un site par deux réels de type double.

L'algorithme de balayage utilise un peu plus de mémoire : en effet, il doit stocker en plus la liste des états et la file de priorité des événements. Nous n'avons pas de mesures précises, n'ayant pas réalisé ce programme.

L'algorithme basé sur l'arbre de Delaunay est le plus gourmand, sa consommation mémoire semble être six fois plus importante que celle de l'algorithme basé sur le 2d-tree.

Les autres algorithmes dérivés du 2d-tree, soit ceux fondés sur l'adaptive 2d-tree, le random 2d-tree et l'adaptive random 2d-tree, exigent un tableau auxiliaire de n entiers pour mémoriser les faux-médians et/ou la direction de division. Le surcoût par rapport à l'algorithme basé sur le 2d-tree est faible.

L'algorithme basé sur le bucket-tree, ainsi que celui de Dwyer, utilisent une grille régulière. Il faut donc construire un tableau de n cellules, chaque cellule étant la liste chaînée des sites qu'elle contient. Le surcoût en espace mémoire est modéré et reste compris entre $2n$ et $4n$ pointeurs. Mais ce prétraitement n'est efficace que sur des distributions assez uniformes.

Les algorithmes **LS**, **2d**, **a2d**, **r2d**, **ar2d**, **Dw**, **bt** présentent l'avantage suivant : on peut calculer à l'avance avec précision l'espace mémoire nécessaire, qui reste relativement modeste. En ce qui concerne l'algorithme basé sur le quadtree, cette prévision n'est plus possible. Dans le paragraphe 2.1.3, en faisant des hypothèses sur la dynamique des données, on peut donner un majorant de l'espace mémoire nécessaire mais il est élevé. Nous avons mesuré expérimentalement (voir annexe D.1) cet espace mémoire. Curieusement, pour des ensembles de sites dont la taille est comprise entre 10^4 et 10^7, le tableau stockant le quadtree a une taille proche de $3n$ cellules (et même légèrement inférieure pour plus de 10^6 sites), et ceci quelle que soit la distribution. Par contre, pour des ensembles de moins de mille sites, on peut parfois observer des pics compris entre $4n$ et $8n$ (et même plus!) pour des distributions fortement hétérogènes (**corn** par exemple). Pratiquement, on surmonte ce problème en allouant un premier tableau de taille $8n$ pour les ensembles de moins de mille points, $6n$ entre mille et dix mille points et $4n$ pour les ensembles plus importants. On teste, lors de chaque insertion, un débordement éventuel. En cas de débordement, on réalloue un tableau un peu plus grand ($2n$ cellules en plus par exemple) et ainsi de suite jusqu'à la construction complète.

7.4.4 Améliorations possibles

- On a observé que les versions randomisées sont un peu plus rapides, parce que le temps de construction de l'arbre est plus faible. Ceci montre qu'en utilisant les méthodes de construction des 2d-trees présentées dans les remarques du paragraphe 2.2.4, on peut encore accélérer les algorithmes basés sur les 2d-trees.

- Si l'on triangule directement les cellules dès qu'elles contiennent moins de quatre points, on peut gagner environ 3% dans le temps d'exécution des algorithmes du type Divide-and-Conquer.

- G.Leach [129] donne des méthodes pour accélérer les algorithmes de type Divide-and-Conquer. Il semble que la seule amélioration nouvelle par rapport à notre implémentation concerne le processus de fusion des sous-triangulations. G.Leach montre que, dans les calculs de déterminant pour tester la position d'un point par rapport à un cercle et par rapport à une droite, on évalue plusieurs fois les mêmes quantités. En stockant dans des variables ces résultats intermédiaires, on peut réaliser une économie importante dans les calculs à effectuer. Il arrive à réduire de plus de moitié le nombre des opérations dans la fonction *Incircle* (position d'un point par rapport à un cercle). Cette fonction est appelée environ $6.5n$ fois avec l'algorithme basé sur un 2d-tree pour une distribution uniforme. D'après G.Leach, le gain de temps est loin d'être négligeable. Cette optimisation pose cependant quelques problèmes vis-à-vis de l'imprécision numérique. Elle empêche l'utilisation des méthodes de calcul exact du signe d'un déterminant [8].

7.4.5 Performances et perspectives

Disposant pendant quelques jours d'une Silicon Graphics Indy (200 MHz, 60 MO), nous en avons profité pour tester l'algorithme basé sur le 2d-arbre. Après une phase de tri de 3 secondes, la triangulation d'un semis de 200 000 points s'est effectuée en 7 secondes. La vitesse de triangulation (en excluant le temps consacré au tri) est de l'ordre de 30 000 points par seconde (environ 50 000 à 60 000 triangles par seconde). Pour des semis de taille moyenne (moins de 30 000 points), on peut considérer que la triangulation s'effectue en temps réel. A l'aide des graphiques précédents, on peut extrapoler avec quasi-certitude que l'algorithme fondé sur le quadtree triangulera 200 000 points en 8 secondes environ (prétraitement compris), quelle que soit la distribution. L'optimisation de G.Leach (para. 7.4.4) devrait permettre de s'approcher des 5 secondes pour 200 000 points !

Il y a quelques années, on considérait que la triangulation était une opération longue (il était classique de la lancer le soir pour avoir le résultat le lendemain matin). Maintenant, grâce à l'évolution des machines et des algorithmes, la façon de travailler change. Pour des semis d'une centaine de milliers de points, il est inutile de stocker la triangulation sur disque. En effet, le temps nécessaire pour la relire à partir du disque est plus important que le temps nécessaire pour la recalculer. La triangulation de Delaunay en temps interactif (pour des simulateurs de vol par exemple) n'est plus un objectif irréalisable.

Il y a quelques années, l'objectif du million de triangles à la seconde avait été fixé. Nous sommes encore loin de cette performance qui était complètement utopique à l'époque. Mais, compte tenu des progrès constants et rapides dans la vitesse des processeurs, il n'est pas exclu que, dans quelques années, cet objectif soit atteint.

7.5 Conclusion

L'algorithme basé sur le quadtree est remarquablement performant. On peut cependant lui reprocher une certaine instabilité en ce qui concerne l'espace mémoire utilisé (surtout pour des ensembles de moins de mille points). Il est très peu perturbé par l'hétérogénéité de la distribution. Il est toujours le meilleur dès que la distribution n'est pas parfaitement uniforme. Il est battu par le bucket-tree, mais de façon infime, pour une distribution uniforme. Ensuite, se placent les algorithmes basés sur les 2d-trees. Ils présentent l'avantage de minimiser l'espace mémoire que l'on peut calculer avec précision avant de lancer la triangulation et d'être peu sensibles à l'hétérogénéité de la distribution. Les versions randomisées sont un peu plus rapides, parce que le temps de construction de l'arbre est plus faible. Les algorithmes de Dwyer et de Lee et Schachter sont globalement moins bons, Lee et Schachter parce que la triangulation s'effectue toujours en $\Theta(n \log n)$, Dwyer parce qu'il s'adapte moins bien aux distributions non uniformes et que, même sur ces distributions, l'algorithme dirigé par le bucket-tree le surpasse.

Chapitre 8

L'IMPRECISION NUMERIQUE

"To err is human, but to really foul things up, you need a computer."

Il s'agit là d'un des problèmes les plus cruciaux en Géométrie Algorithmique, et aussi l'un des principaux enjeux des prochaines années ([46] chap. 10). Comme les structures géométriques construites par les algorithmes dépendent de tests numériques, les problèmes de précision ont une importance capitale en Géométrie Algorithmique. Ils sont en général imputables à la troncature des nombres par la machine. Le codage des réels sur machine est réalisé sur un nombre fini de bits (unité binaires d'information) et le nombre de valeurs différentes que peut prendre une variable est donc fini (une variable codée sur b bits peut prendre 2^b valeurs différentes). Cela est bien sûr insuffisant pour pouvoir décrire exactement l'ensemble des réels ou des rationnels, même sur un intervalle borné donné. Sauf les cas discrets où la variable vaut exactement la valeur souhaitée, cette variable approche cette valeur avec une erreur dont la valeur maximale est l'erreur machine.

Les approximations qu'impliquent les calculs "qui ne tombent pas rond" au sens de la machine engendrent des erreurs numériques qui sont amplifiées au fur et à mesure de l'enchaînement des opérations. Ces erreurs restent souvent négligeables dans la plupart des applications où seule la vision "calcul numérique" est considérée, grâce à une "continuité" du comportement qui fait qu'une erreur minime en entrée provoque une erreur minime en sortie. Mais, ce n'est généralement pas le cas en géométrie algorithmique (et même en analyse numérique).

On est souvent amené à faire des alternatives sur des critères numériques représentatifs de la géométrie des données. Par exemple, entre un point et une droite, on cherche à savoir si le point est dessus, dessous ou sur la droite. Ces alternatives créent des "discontinuités" dans le comportement de l'algorithme qui, si par malheur l'erreur fait passer d'un côté à l'autre, donnera au programme une réponse totalement inadaptée. En particulier, le test d'égalité stricte risque de générer des réponses souvent hasardeuses. Un point peut aussi être déclaré intérieur ou extérieur à une région, par simple changement du contexte dans lequel les calculs sont effectués (*dynamique* des données[1], rotation, nombre de bits dans les calculs,...).

Par exemple, l'algorithme de Bentley et Ottman [19], qui permet de trouver les points d'intersection entre n segments par un balayage du plan, est très sensible à l'imprécision numérique qui conduit souvent l'algorithme à donner des résultats erronés ou même à s'arrêter avant la fin des calculs (voir deux exemples détaillés dans la thèse de Michelucci [144]). Les algorithmes de triangulation de Delaunay basés sur une méthodologie "Divide-and-Conquer" sont aussi sensibles, de façon moins significative, à l'imprécision numérique. Les exemples des paragraphes 8.2.1 et 8.2.2 l'illustrent.

Bien que beaucoup de travaux aient été publiés dans ce domaine, aucune solution, à ce jour, n'allie à la fois vitesse et fiabilité. Ce chapitre présente un rapide état de l'art sur ce sujet, qui constitue à lui seul matière à plusieurs thèses. Puis deux exemples illustrant les effets de l'imprécision numérique sur notre algorithme de triangulation de Delaunay, sont donnés et enfin un remède, propre à notre application de CAO routière, est proposé.

8.1 Etat de l'art

8.1.1 Epsilons

Une approche intuitive, utilisée dans la quasi-totalité des logiciels de CAO, consiste à introduire des marges de tolérance. Par exemple, chaque sommet appartient à un disque de rayon epsilon, chaque segment appartient à une bande de largeur 2 epsilons...Mais le choix de la valeur de epsilon est délicat et il n'est jamais possible d'en prévoir les conséquences. En fait, cette méthode ne peut pas conduire à une solution parfaitement fiable, elle n'est ni robuste, ni scientifique et ne peut fonctionner que si l'on adapte ϵ à la dynamique des données.

On peut citer les méthodes : epsilon brut (présenté ci-dessus), taille d'attributs minimum [179], epsilon-convexité [149], géométrie à epsilon près [103]...

[1] La dynamique d'un jeu de données est le rapport entre les plus grandes et les plus petites distances significatives.

8.1.2 Géométries robustes

L'idée maîtresse de ces méthodes est que la topologie propre des objets (qui est indépendante de la précision) est le seul motif de décision, lorsque la précision vient à manquer, qui permette de conserver une cohérence globale en fin d'opération, même si la cohérence locale a dû en souffrir. On peut citer : résolution finie [98], déterminisme et historique [146] [147] [148], valence et cohérence [114]...

8.1.3 Stabilité numérique et perturbations

Ces méthodes relèvent toutes du domaine de l'analyse numérique. Elles permettent d'éliminer les situations dégénérées, mais leur complexité et le coût de leur implantation les rendent difficilement exploitables. On peut citer : simulation de "simplicité" [76] [196] [197], stabilité numérique [161]...

8.1.4 Solutions structurelles

Ces méthodes ont été conçues explicitement pour répondre à des besoins spécifiques, en l'occurrence aider à la résolution d'un problème recensé, par exemple : diagramme de Voronoi [189] [190] [191], triangulation de Delaunay [115]...

8.1.5 Arithmétique entière et rationnelle

C'est une arithmétique exacte (voir D.E. Knuth [120]). L'arithmétique exacte a souvent l'inconvénient d'imposer des temps de calcul prohibitifs. Un entier peut être représenté par une liste chaînée de chiffres dans une base B (par exemple $B = 2^{16} = 65536$) bien choisie. Un rationnel est représenté par un numérateur et un dénominateur qui sont des entiers premiers entre eux.

8.1.6 Recalage sur une grille entière

Cette méthode [144] permet de réaliser un très grand nombre d'opérations de manière fiable, mais elle n'est plus applicable si la dynamique des données est trop grande. Le recalage sur une grille peut aussi provoquer des incohérences topologiques : deux segments qui se coupent en réalité ne se coupent plus après recalage.

8.1.7 Arithmétique mixte et réticente

La philosophie de cette méthode [151] est de faire d'abord les calculs en précision finie. Si l'erreur accompagnant les tests est plus grande que l'erreur maximale tolérable, on les effectue de nouveau, mais en arithmétique exacte.

8.1.8 Arithmétique paresseuse

Cette technique [14] [15] [112] [145] a été inspirée par la précédente et en constitue un prolongement automatique. Elle est fondée sur la constatation qu'en fait, les calculs conduisant aux tests litigieux sont souvent très peu nombreux par rapport à la masse des calculs, et la haute précision y est le plus souvent totalement inutile. On effectue donc les calculs avec les réels de la machine et des encadrements (intervalles) aussi fins que possible. Dès que les décisions à prendre deviennent ambiguës (intervalles non disjoints), on doit alors les prendre sur la foi d'une arithmétique rationnelle ou, en général, exacte. En utilisant une double représentation des données (approchée en flottants et symbolique/exacte), ainsi qu'une arithmétique d'intervalles, on laisse la bibliothèque paresseuse effectuer les calculs en flottant d'abord, puis en exact si ceux-ci ne permettent pas de prendre des décisions cohérentes.

Les données initiales sont comprises comme accompagnées d'une précision maximale et tout calcul effectué par un programme l'est en flottant grâce à l'arithmétique d'intervalles. Il est de même mémorisé sous forme de graphe expressionnel orienté et sans circuits. Lors de la comparaison de deux nombres "paresseux", la librairie vérifie d'abord si leurs intervalles sont disjoints. Si c'est le cas, les deux nombres peuvent être comparés sans ambiguïté à l'aide de ces intervalles. Sinon, on procède à une évaluation en profondeur de leurs graphes expressionnels. Ceci a comme effet de remplacer les feuilles par des rationnels et des intervalles plus serrés, et ainsi de suite en remontant. Si les deux intervalles encadrant les deux nombres, après évaluation, ne sont toujours pas disjoints, on compare ces derniers à l'aide de l'arithmétique rationnelle disponible dans la librairie.

La librairie est écrite en C++ et permet donc l'élaboration de programmes complètement libérés des problèmes de précision, en C++ standard. Avec cette méthode, ce n'est pas au programmeur d'améliorer ses programmes en fonction de la précision, mais à la librairie de gérer silencieusement et automatiquement les problèmes d'imprécision.

8.1.9 Calcul exact du signe d'un déterminant

On remarque que, dans la plupart des algorithmes géométriques, les tests cruciaux vis-à-vis de la topologie des données initiales, se ramènent souvent au calcul du signe d'un déterminant (par exemple déterminant 2×2 pour trouver la position d'un point par rapport à une droite, déterminant 4×4 pour trouver la position d'un point par rapport à un cercle...). L'INRIA à Sophia Antipolis a réalisé une étude théorique et expérimentale de deux méthodes différentes pour évaluer le signe d'un déterminant :

la première [38] étant une variante de celle de Clarkson, la seconde [8] conçue d'abord pour les déterminants 2×2 et 3×3, puis généralisée [38] à une dimension quelconque. Les deux méthodes calculent le signe d'un déterminant $n \times n$ dont les éléments sont des entiers codés sur b bits, en utilisant une arithmétique exacte sur seulement $b + O(n)$ bits. Ces deux méthodes sont adaptatives, statuant rapidement sur les cas non litigieux et ayant recours à des calculs plus poussés pour examiner les déterminants de valeur nulle. Le temps de calcul, avec la seconde méthode, est augmenté d'environ 15 % (resp. 60 %) pour un déterminant 2×2 (resp. 3×3) par rapport au simple calcul avec le type "double".

8.1.10 La classe de nombres réels LEDA

Cette classe C++ [41] [42] a été conçue et implémentée à l'institut Max Plank de Saarbrück. Elle permet le calcul exact avec les opérations addition, soustraction, multiplication, division et racine carrée. Une approximation dans le type "double" (avec une incertitude) est faite pour chaque nombre réel. Les tests et les calculs sont faits sur ces approximations. Les calculs de plus haute précision ne sont effectués que quand ceux réalisés sur les approximations en "double" ne permettent pas de conclure de façon certaine. L'augmentation des temps de calcul due à cette méthode reste acceptable.

8.1.11 Les prédicats de Shewchuk

C'est une arithmétique adaptative exacte [183] [184] sur des nombres réels, c'est à dire que le temps de calcul dépend du degré d'incertitude du résultat. Cette technique a été utilisée dans l'implémentation de quatre prédicats pour tester la position d'un point par rapport à un segment (resp. plan) et à un cercle (resp. sphère) dans le plan (resp. espace). Le code, comme pour celui du calcul exact du signe d'un déterminant, est du domaine public. Shewchuk [182] montre l'efficacité de cette technique pour le calcul de la triangulation de Delaunay, qui entraîne peu d'augmentation du temps de calcul, entre 10 % et 30 %. Il n'existe pas actuellement de comparaison de performances entre la méthode de l'INRIA et celle de Shewchuk, mais celle de l'INRIA ne s'applique (pour l'instant) qu'à des points dont les coordonnées sont entières.

8.2 Triangulation de Delaunay et imprécision

Dans un premier temps, nous allons mettre en évidence les conséquences désastreuses que peut entraîner l'implémentation naïve de la triangulation de Delaunay avec l'arithmétique flottante. Nous montrerons ensuite que l'introduction (presque aussi naïve) d'une marge de tolérance pour le test du signe du déterminant peut avoir des con-

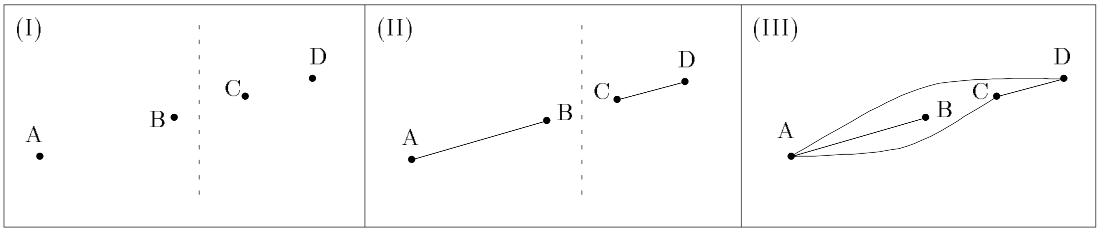


Figure 8.1: *Triangulation en arithmétique flottante :*
(I) division du semis selon l'ordre lexicographique,
(II) triangulation des deux parties,
(III) fusion des deux sous-triangulations.

séquences tout aussi désastreuses.

8.2.1 Calcul de déterminant en arithmétique flottante

L'utilisation de l'arithmétique flottante pour le calcul de la triangulation de Delaunay avec la méthode basée sur le $2d$-tree peut produire des résultats erronés et même un arrêt du programme avant la fin du calcul. L'exemple suivant montre comment ce problème peut survenir :

considérons (voir figure 8.1) les quatre points alignés suivants :

$$A\begin{pmatrix}1.74\\0.06\end{pmatrix} \qquad B\begin{pmatrix}2.7\\0.34\end{pmatrix} \qquad C\begin{pmatrix}3.18\\0.48\end{pmatrix} \qquad D\begin{pmatrix}3.66\\0.62\end{pmatrix}$$

Pendant la phase de division, ces 4 points sont séparés en deux sous-ensembles : la partie gauche comprenant les points A et B, la partie droite comprenant les points C et D. Ces deux sous-ensembles sont ensuite triangulés trivialement et on obtient les segments AB et CD.

Pour fusionner ces deux sous-triangulations (voir paragraphe 3.2.3), on considère le segment reliant le point le plus à droite de la partie gauche au point le plus à gauche de la partie droite, soit BC. Ensuite, on descend sur les deux enveloppes convexes pour trouver l'arête la plus basse reliant les deux parties.

Pour cela, on examine si C est à droite de AB en calculant le déterminant formé par les vecteurs $\overrightarrow{AB}$ et $\overrightarrow{BC}$. En réalité, les points A, B et C sont alignés mais le progamme trouve que $\mathrm{Det}(\overrightarrow{AB}, \overrightarrow{BC}) = -2.775557561562899 \times e^{-17}$ (l'ordinateur convertit les coordonnées en base 2 et ne garde que 16 chiffres alors que, dans certains cas, il faudrait une suite infinie de décimales pour représenter la valeur exacte du nombre ; les valeurs stockées par la machine sont donc parfois inexactes, elles s'écartent de la

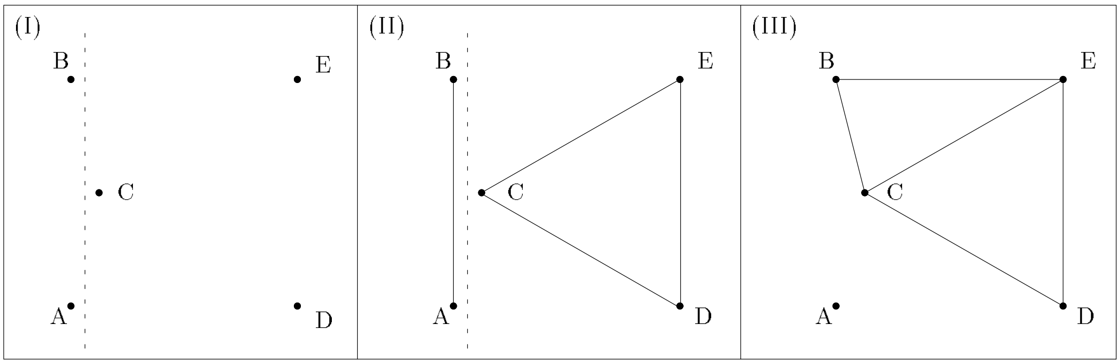


Figure 8.2: *Triangulation avec epsilons :*
(I) division du semis selon l'ordre lexicographique,
(II) triangulation des deux parties,
(III) fusion des deux sous-triangulations.

valeur exacte d'un epsilon ce qui explique la valeur du déterminant ci-dessus). On se place ensuite en A et on considère le segment AC. Comme $\mathrm{Det}(\overrightarrow{BA}, \overrightarrow{AC}) = 0.0$ (remarquez que la machine trouve $\mathrm{Det}(\overrightarrow{AB}, \overrightarrow{BC}) \neq \mathrm{Det}(\overrightarrow{BA}, \overrightarrow{AC})$!), on arrête la descente à gauche.

On cherche maintenant à redescendre du côté droit. On calcule le déterminant $\mathrm{Det}(\overrightarrow{AC}, \overrightarrow{CD}) = +5.551115123125783 \times e^{-17}$. Comme celui-ci est positif, on arrête la descente à droite.

Le programme conclut donc que AC est l'arête la plus basse reliant les deux sous-triangulations. En réalité, il aurait dû trouver BC !

La fusion commence alors à partir de l'arête AC. D'après les valeurs déjà calculées de $\mathrm{Det}(\overrightarrow{BA}, \overrightarrow{AC})$ et de $\mathrm{Det}(\overrightarrow{AC}, \overrightarrow{CD})$, le programme construit l'arête AD. Ensuite, il calcule $\mathrm{Det}(\overrightarrow{BA}, \overrightarrow{AC}) = 0.0$ et $\mathrm{Det}(\overrightarrow{BA}, \overrightarrow{AD}) = -1.110223024625157 \times e^{-16}$. La fusion est finie.

On s'aperçoit que le maillage final est formé du triangle $\triangle ACD$ avec une arête pendante AB à l'intérieur (ce n'est même pas une triangulation ! il aurait fallu trouver les trois arêtes AB, BC et CD). Si, par la suite, on doit fusionner ce maillage avec d'autres sous-triangulations, les incohérences vont devenir encore plus importantes et provoquer éventuellement l'égarement du programme avec le message bien connu *"segmentation fault : core dumped"*.

8.2.2 Calcul de déterminant avec epsilons

L'introduction d'une marge de tolérance dans le test du signe du déterminant, solution choisie par la plupart des logiciels de CAO, est loin d'être satisfaisante, comme le montre l'exemple suivant :

considérons (voir Figure 8.2) les cinq points suivants, quatre sont les sommets d'un carré et le cinquième est très proche du milieu du côté vertical gauche :

$$A\begin{pmatrix}0.0\\0.0\end{pmatrix}\qquad B\begin{pmatrix}0.0\\1.0\end{pmatrix}\qquad C\begin{pmatrix}\epsilon\\0.5\end{pmatrix}\qquad D\begin{pmatrix}1.0\\0.0\end{pmatrix}\qquad E\begin{pmatrix}1.0\\1.0\end{pmatrix}$$

Pendant la phase de division, ces 5 points sont partagés en deux sous-ensembles : la partie gauche comprenant les points A et B, la partie droite comprenant les points C, D et E. Ces deux sous-ensembles sont ensuite triangulés trivialement et on obtient le segment AB et le triangle $\triangle CDE$.

Pour fusionner ces deux sous-triangulations (voir paragraphe 3.2.3), on considère le segment reliant le point le plus à droite de la partie gauche au point le plus à gauche de la partie droite, soit BC. Ensuite, on descend sur les deux enveloppes convexes pour trouver l'arête la plus basse reliant les deux parties.

Pour cela, on examine si C est à droite de AB en calculant le déterminant formé par les vecteurs $\overrightarrow{AB}$ et $\overrightarrow{BC}$. Les points A, B et C sont presque alignés et le déterminant vaut ϵ. Si l'on choisit une marge de tolérance légèrement supérieure à ϵ, le programme considère que ce déterminant est nul.

On examine maintenant si D est à droite de BC. Ce n'est pas le cas.

Le programme conclut donc que BC est l'arête la plus basse reliant les deux triangulations à fusionner. En réalité, il aurait dû trouver AD !

La fusion commence alors à partir de l'arête BC. Les deux triangles $\triangle CBE$ et $\triangle CED$ sont créés, mais les deux triangles adjacents au point A sont oubliés. Si, par la suite, on doit fusionner ce maillage avec d'autres sous-triangulations, les incohérences vont devenir encore plus importantes et provoquer éventuellement, comme avec la méthode précédente, l'égarement du programme avec le message bien connu *"segmentation fault : core dumped"*.

8.2.3 Solution pratique retenue

Les calculs numériques dans les algorithmes de triangulation de Delaunay du type *Divide and Conquer* sont des évaluations de signe de déterminants 2×2 et 4×4.

- Le déterminant 4×4 indique si un point D est intérieur au cercle circonscrit à trois sites A, B et C. Une mauvaise évaluation du signe de ce déterminant nous conduira à créer un triangle qui n'est pas de Delaunay dans le quadrilatère **convexe** $ABCD$. L'imprécision numérique dans le calcul du déterminant 4×4 peut nous faire construire une triangulation qui n'est pas exactement de Delaunay, mais n'a pas de conséquence plus fâcheuse[2] sur le déroulement du programme. Cette triangulation est cependant suffisante pour notre application de CAO routière, où les erreurs dues aux interpolations sont bien plus importantes.

- Le déterminant 2×2 indique si un point C est à gauche ou à droite d'un segment AB. Le paragraphe 8.2.1 illustre les conséquences désastreuses d'une erreur dans l'évaluation du signe de ce déterminant. Un remède simple consiste à réduire la dynamique des données et à introduire une marge de tolérance. Supposons que les coordonnées des sites soient comprises entre 10^{-2} et 10^{+5}. Les produits apparaissant dans le calcul du déterminant seront compris entre 10^{-4} et 10^{+10}, ce qui nécessite 14 chiffres significatifs. La représentation des réels avec le type *double* (16 chiffres significatifs) garantit l'exactitude de l'évaluation du signe du déterminant, à condition de choisir une tolérance égale à 10^{-5}. Là aussi, on construit une triangulation qui n'est pas exactement de Delaunay car on considère que certains sites sont alignés (déterminant nul) alors qu'ils ne le sont pas parfaitement. Cette approximation n'est pas gênante pour notre application de CAO routière. Ce raisonnement grossier montre, conformément au paragraphe 8.1.1, que la méthode des *epsilons* peut éventuellement fonctionner à condition de réduire la dynamique des données et de faire dépendre ϵ de l'expression arithmétique du test.

8.3 Conclusion

On a résolu le problème de l'imprécision numérique par une méthode simple en réduisant la dynamique des données (semis de 100 km de côté maximum avec une précision au cm). Nous insistons sur le fait que, si l'on désire trianguler des semis de plus grande taille, ou si l'on désire une triangulation qui soit exactement de Delaunay, il est **absolument** indispensable d'utiliser les techniques plus sophistiquées des paragraphes 8.1.8, 8.1.9, 8.1.10 ou 8.1.11. Ces méthodes entraînent une augmentation du temps de calcul, mais garantissent l'exactitude du résultat et évitent même l'égarement du programme avec le message bien connu *"segmentation fault : core dumped"*.

[2] Il serait intéressant de prouver qu'une erreur dans le calcul de ce déterminant ne peut pas entraîner d'incohérences dans le déroulement de l'algorithme.

Chapitre 9

LOCALISATION DANS UNE TRIANGULATION DE DELAUNAY

Quand on dispose d'une surface triangulée, il est primordial de pouvoir localiser un point quelconque par rapport à cette surface. C'est un des plus anciens problèmes de géométrie algorithmique, connu sous le nom de *Point-Location* problème. Le *Planar Point Location* problème consiste plus généralement à déterminer dans quelle région d'une subdivision planaire se trouve un point donné. Ce problème se généralise dans un espace de dimension quelconque. Il suffit d'avoir une partition de l'espace et de rechercher la subdivision qui contient un point donné. Ce chapitre traite du cas particulier où la subdivision planaire est une triangulation.

9.1 Etat de l'art

Depuis plus d'une vingtaine d'années, le problème de la localisation motive de nombreux chercheurs. L'évolution des solutions données à ce problème reflète bien l'évolution de la géométrie algorithmique depuis sa naissance vers les années 1975. Au début, quelques solutions pratiques, mais non optimales, ont été trouvées. Ensuite, de nouvelles méthodes ont permis de diminuer la complexité dans le pire des cas jusqu'à la méthode optimale (en espace mémoire et temps de réponse) dans le plan proposée par Lipton et Tarjan (qui n'est cependant pas utilisable en pratique). D'autres solutions plus pratiques, optimales ou proches de l'optimalité, ont ensuite été trouvées : celle de Kirkpatrick avec une hiérarchie de triangulations, améliorée par Seidel avec une hiérarchie de cartes de trapèzes. Mais les solutions optimales restent encore un peu décevantes du point de vue expérimental, et des méthodes, ayant une bonne complexité en moyenne grâce à des hypothèses sur la distribution des sites, apparaissent. Plus récemment, la randomisation a donné naissance à de nouveaux algorithmes dont certains sont dynamiques. Les techniques de dérandomisation permettent parfois à partir d'un algorithme randomisé d'obtenir un algorithme déterministe ayant la même complexité dans le pire des cas.

Mais il existe toujours un fossé entre les méthodes théoriques optimales ou suboptimales et leur utilisation pratique. C'est aussi pour cette raison que le problème de la localisation est encore d'actualité. Quand une solution optimale et pratique est trouvée pour un problème, la tâche du géomètre algorithmicien est terminée !

Etant donné que ce travail de thèse se situe dans le cadre de la réalisation d'un logiciel de CAO routière, les performances pratiques des algorithmes sont très importantes. Il faut minimiser les prétraitements et l'espace mémoire tout en obtenant d'excellents temps de réponse. Les algorithmes proposés ont une bonne complexité en moyenne, mais ne présentent aucun intérêt en ce qui concerne la complexité dans le pire des cas. Mais, en pratique (tout du moins dans notre contexte), la probabilité du pire des cas est quasiment nulle.

Le but de ce paragraphe n'est pas d'être exhaustif, mais de montrer au lecteur la richesse des solutions existantes, bien qu'aucune ne soit complètement satisfaisante.

9.1.1 Méthode naïve

La méthode naïve consiste à prendre les faces une par une en testant si le point à localiser est à l'intérieur. On a besoin d'une fonction ([132] page 41) qui détermine la position d'un point par rapport à un contour (voir aussi [105]). On peut améliorer cette méthode en construisant une hiérarchie de contours ([132] page 47). La complexité dans le pire des cas est linéaire.

9.1.2 Méthode des bandes

9.1.2.1 Dobkin et Lipton

C'est le premier algorithme efficace de localisation mis au point par Dobkin et Lipton [68] en 1975. Nous donnons son principe dans le plan, mais il fonctionne en dimension quelconque. Il est simple et utilise un découpage de la subdivision en bandes verticales.

On trie les n sites par abscisses croissantes, en temps $O(n \log n)$, et on fait passer par chacun d'entre eux une droite verticale. Le plan est alors divisé en $n+1$ bandes verticales. Les côtés de la subdivision coupant une bande donnée peuvent être ordonnés de bas en haut (car deux côtés distincts ne peuvent se couper à l'intérieur d'une bande). Dans chaque bande, qui peut être coupée au pire par $O(n)$ arêtes, on ordonne les segments de bas en haut, ce qui est réalisé en $O(n \log n)$. Le tri complet des $n+1$ bandes aura une complexité en $O(n^2 \log n)$. Le prétraitement est donc en

$O(n^2 \log n)$. Chaque bande (il y en a $n+1$) peut couper $O(n)$ segments. L'espace mémoire requis est donc en $O(n^2)$.

La localisation d'un point se ramène donc à deux recherches dichotomiques successives. En temps $O(\log n)$, on détermine dans quelle bande se trouve le point à localiser, puis à l'intérieur de la bande, on détermine en temps $O(\log n)$ entre quels segments se situe le point, ce qui permet d'identifier la région. On suppose que l'on connaît le contour de chaque région ainsi que, pour chaque côté, les noms des deux régions situées de part et d'autre.

Le temps de prétraitement est en $O(n^2 \log n)$, l'espace mémoire en $O(n^2)$ et le temps de localisation en $O(\log n)$.

9.1.2.2 Amélioration : Shamos

Cette méthode, mise au point par Shamos [165] en 1975, est quasiment identique à la précédente, seul le temps de prétraitement a été réduit à $O(n^2)$ par l'utilisation d'une *droite de balayage* (sweep-line).

La barre de balayage est représentée par un arbre équilibré contenant les côtés de la subdivision qu'elle coupe, ordonnés de bas en haut. On sait que l'insertion et la suppression dans une telle structure (voir paragraphe 2.2.5) se fait en temps logarithmique. L'événement qui déclenche la mise à jour est le passage de la barre de balayage – qui se déplace de gauche à droite – par un site s (les sites ont été triés par ordre lexicographique). La mise à jour consiste à supprimer de l'arborescence les arêtes mourant en s (arêtes dont la seconde extrémité relativement à l'ordre $<_x$ est s) et à insérer les arêtes naissant en s (arêtes dont la première extrémité relativement à l'ordre $<_x$ est s). La succession de toutes les mises à jour de la barre de balayage ($O(n)$ insertions ou suppressions puisque le graphe a $O(n)$ arêtes) se fait en $O(n \log n)$. Le temps de prétraitement est donc en $O(n^2)$ à cause du stockage de toutes les bandes qui peuvent avoir chacune jusqu'à $O(n)$ éléments.

Le temps de prétraitement est en $O(n^2)$, la complexité en espace en $O(n^2)$ et le temps de localisation en $O(\log n)$.

9.1.2.3 Amélioration : Sarnak et Tarjan

Sarnak et Tarjan [172], en 1986, ont amélioré la méthode précédente en ce qui concerne le temps de prétraitement et la complexité en espace mémoire. Ils utilisent un *arbre persistant* au lieu d'avoir, pour chaque bande de balayage, un arbre binaire de recherche.

Le prétraitement consiste à balayer le plan de gauche à droite par une droite verticale. On remarque que deux bandes qui se suivent, contiennent des données presque identiques (suppression des côtés d'extrémité s et insertion des côtés d'origine s relativement à l'ordre $<_x$).

Sarnak et Tarjan ont inventé une structure appelée *arbre persistant* [13] [69] qui permet de gérer un arbre binaire subissant des modifications à des instants successifs (à chaque instant, une insertion ou une suppression est autorisée). L'arbre persistant contient l'historique de l'évolution de l'arbre au cours du temps, avec un espace mémoire linéaire $O(n)$, un temps de construction en $O(m \log n)$ et un temps de recherche en $O(\log m)$ si m est le nombre d'instants de la chronologie et n le nombre maximum d'éléments à un instant t (on a toujours $n \leq m$). Dans cet exemple, l'arbre persistant contient les côtés de la subdivision coupée par la barre de balayage.

Le prétraitement demande donc un temps en $O(n \log n)$ et un espace mémoire en $O(n)$. Le temps de localisation est logarithmique. L'algorithme est optimal (en espace mémoire et en temps de prétraitement) dans le pire des cas, mais compliqué à implanter.

9.1.3 Méthode des trapèzes

Elle a été mise au point en 1981 par Preparata [163]. C'est une amélioration de la méthode de Shamos car le temps de prétraitement et la complexité en espace sont réduits à $O(n \log n)$. Cet algorithme utilise aussi des bandes (horizontales) mais les recherches sont conduites simultanément dans les directions horizontales et verticales et un arbre unique est construit.

On construit un arbre binaire de bandes. Les sites sont classés par ordonnées croissantes. La première bande B_0 est limitée par le premier site s_1 et le dernier site s_n (en supposant que la numérotation représente l'ordre des sites). On découpe cette bande en deux, la droite médiane passant par le site $s_{\lceil (n+1)/2 \rceil}$. On continue ce découpage dichotomique jusqu'à ce que les bandes soient limitées par des sommets consécutifs dans la liste triée des sites. L'arbre binaire est réalisé de la façon suivante : on associe B_0 à la racine et pour chaque noeud, on associe à son fils gauche (resp. droit) la moitié inférieure (resp. supérieure) de la bande associée au noeud. Les arêtes sont ensuite découpées en $O(\log n)$ segments (on découpe l'arête par des bandes de largeur maximale : on appelle *arête traversière* une arête qui coupe les deux bords d'une bande, le segment égal à l'intersection d'une bande et d'une arête traversière n'est jamais redécoupé). Les arêtes traversières définissent des trapèzes (finis ou non) dans les bandes. On a des relations d'imbrication entre les différents trapèzes.

L'arbre de recherche se construit de la façon suivante : à chaque fois que l'on coupe une bande par dichotomie, on traite la liste R des arêtes qui coupent la ligne médiane. Elle donne naissance à deux listes R_1 et R_2 correspondant à chaque sous-bande. On repère les arêtes traversières qui sont toujours frontières d'un trapèze. On itère ce processus sur chaque bande, donc sur chaque trapèze. On construit un arbre de recherche qu'il faut en plus équilibrer.

La recherche se fait de la façon suivante. L'arbre contient deux types de noeuds :

1. les ∇-noeuds qui sont des sites. Le test associé est : dessous ou au-dessus de la ligne horizontale passant par le ∇-noeud.
2. les o-noeuds qui sont des arêtes traversières. Le test associé est : à gauche ou à droite de l'arête traversière contenue dans le o-noeud.

Le prétraitement et l'espace mémoire sont en $O(n \log n)$ et la localisation se fait toujours en temps logarithmique.

9.1.4 Méthode de la séparation planaire

En 1977, Lipton et Tarjan [138] découvrirent la première méthode optimale en espace mémoire et en temps de réponse. Mais ils exprimèrent l'avertissement suivant : "*we do not advocate this algorithm a practical one, but its existence suggests that there may be a practical algorithm with* $O(\log n)$ *time bound and* $O(n)$ *space bound*". Cet algorithme n'est pas utilisable en pratique : il est très complexe[1] et les constantes multiplicatives dans la complexité sont déraisonnables. Il tire avantageusement parti de la notation O. Son intérêt est uniquement théorique, mais il a incité des chercheurs à rechercher un autre algorithme optimal plus pratique.

9.1.5 Méthode des chaînes

9.1.5.1 Lee et Preparata

Cette méthode, découverte en 1977, est due à Lee et Preparata [130] et à Shamos [165]. Le principe est de *monotoniser* (ou "régulariser") le graphe, c'est-à-dire de le décomposer en chaînes monotones qui ne se croisent pas. Pendant le prétraitement, on classe ces chaînes relativement à la relation d'ordre *être à gauche de*. La localisation ou discrimination d'un point par rapport à une chaîne se fait en temps logarithmique par rapport au nombre de sommets de la chaîne (la monotonie de la chaîne a permis de classer ses sommets pendant le prétraitement). La localisation d'un point parmi l'ensemble ordonné des chaînes monotones se fait donc en $O(\log p \times \log r)$ où p est le nombre de chaînes et r le nombre maximal de sommets sur une chaîne (p et r sont bornés par n).

[1] L'algorithme est construit à partir du théorème de séparation planaire [139] :
soit G un graphe à n sommets. Les sommets de G peuvent être partitionnés en trois ensembles A, B et C tels que : aucune arête ne joint un sommet de A à un sommet de B, ni A ni B ne contiennent plus de $2n/3$ sommets et C ne contient pas plus de $2\sqrt{2n}$ sommets. Il existe un algorithme linéaire pour construire une telle partition.

Le temps de prétraitement est en $O(n \log n)$, l'espace mémoire en $O(n)$ et le temps de localisation en $O(\log^2 n)$.

9.1.5.2 Amélioration : la méthode des chaînes "en cascade"

Edelsbrünner, Guibas et Stolfi [75] ont amélioré en 1986 la méthode des chaînes. Leur algorithme est optimal en espace mémoire et en temps de réponse. Dans l'algorithme précédent, on discriminait un point par rapport à une liste de chaînes sans utiliser toute l'information produite par chaque discrimination, notamment sur la position du point par rapport à certaines droites verticales coupant les chaînes traitées. Edelsbrünner, Guibas et Stolfi ont eu l'idée de créer des "ponts" entre les différentes chaînes (*bridged chain method*) pour que les informations produites par une discrimination puissent être répercutées sur les autres chaînes. Cette structure de données sophistiquée permet une discrimination en temps logarithmique sur la première chaîne, puis en temps constant sur les autres, à condition qu'elles soient en nombre limité. Cette technique est proche du *fractional cascading* [47] qui permet la recherche en temps logarithmique d'une clé dans des dictionnaires multiples.

Le temps de prétraitement est en $O(n \log n)$, l'espace mémoire en $O(n)$ et le temps de localisation en $O(\log n)$.

9.1.6 Méthode avec une hiérarchie de triangulations

C'est probablement l'algorithme optimal [119] le plus connu, linéaire pour le prétraitement (si le graphe est connexe) et l'espace mémoire, logarithmique pour le temps de réponse. Une première étape consiste à trianguler[2] *complètement*[3] le graphe, ce qui oblige généralement à ajouter trois sommets formant un triangle englobant pour le graphe. Ensuite, on retire un certain nombre de sommets *indépendants* et on retriangule le nouveau graphe. On crée des liens de chaînage entre les triangles des deux triangulations, quand ceux-ci ont une intersection non vide. On itère ce processus jusqu'à ce que le graphe soit réduit à son triangle englobant. On a ainsi construit une arborescence dont la racine est le triangle englobant du graphe, et dont chaque niveau est une triangulation (le dernier niveau est la triangulation complète du graphe). Localiser un point consiste à descendre dans cette arborescence jusqu'au triangle cherché (voir une description plus complète dans le paragraphe 9.2).

Cette méthode présente l'inconvénient de consommer un espace mémoire impor-

[2]Il faut trianguler les faces de la subdivision. En pratique, un polygone quelconque peut facilement être triangulé en temps $O(n \log n)$ [90], bien qu'en théorie, Chazelle [45] ait montré qu'il existe un algorithme linéaire mais inutilisable pratiquement : il utilise récursivement le théorème de séparation planaire, puis une procédure de fusion sublinéaire extrêmement complexe.

[3]Une triangulation complète est une triangulation dont la frontière de la face infinie est un triangle.

tant, bien que linéaire. Il faut stocker toutes les triangulations successives. Kirkpatrick garantit le retrait d'au moins $\frac{m}{26}$ sites d'une triangulation à la suivante (m étant le nombre de sites de la triangulation courante), ce qui est peu. Edelsbrünner écrit ([74] page 266) en 1987 : "*the problem of fine-tuning the method such that it realizes the best performance in practice is still open*". Il propose une méthode basée sur le théorème des quatre couleurs[4] pour améliorer les constantes ([74] exercice 11.9 page 264). Edahiro, Kokubo et Asano [73] améliorent aussi les constantes. M. de Berg et K. Dobrindt [20] utilisent la méthode de Kirkpatrick pour représenter à des niveaux de détail différents un terrain naturel à l'aide de la triangulation de Delaunay. Seidel [177] utilise pour la localisation une hiérarchie de cartes de trapèzes qui est meilleure en pratique que la méthode de Kirkpatrick. En particulier, on peut retirer des segments sans avoir de contraintes au niveau du degré.

9.1.7 Méthode des buckets

Cette méthode [73], proposée par l'université de Tokyo, utilise une partition du rectangle englobant le graphe, en $n_x \times n_y$ rectangles identiques appelés *buckets* (on en crée $O(n)$). On gère ensuite le sous-graphe contenu dans chaque cellule en associant à chaque bucket les listes triées des sites et des fragments d'arêtes qu'il contient. On trouve en temps constant la cellule qui contient le point à localiser, puis la méthode de Shamos est utilisée pour parfaire la localisation (le prétraitement requis par cette méthode n'est effectué que dans la cellule contenant le point à localiser).

Le prétraitement et l'espace mémoire sont en $O(n\sqrt{n})$ et le temps de localisation en $O(n)$ dans le pire des cas. Par contre, le prétraitement et l'espace mémoire sont linéaires *en moyenne*, et la localisation se fait en temps constant *en moyenne*.

Remarque : cet algorithme, pour être efficace, suppose une répartition relativement uniforme des sites et des arêtes du graphe. Seule la complexité *en moyenne* est bonne. Une forte concentration de sites ou d'arêtes dans une cellule supprime tout l'intérêt de cet algorithme.

9.1.8 Méthode randomisée : algorithmes dynamiques

Seidel [176] utilise une approche randomisée pour le problème de la localisation. Il construit ainsi une hiérarchie de cartes de trapèzes. Il donne aussi un algorithme simple randomisé en $O(n \log^* n)$ en moyenne pour construire la carte des trapèzes d'un polygone quelconque, ce qui permet d'effectuer le premier prétraitement (carte des trapèzes du graphe) avec un temps moyen en $O(n \log^* n)$ pour un graphe connexe et en

[4] N'importe quel graphe planaire peut être colorié avec quatre couleurs, de sorte que deux régions ayant une frontière commune soient de couleurs différentes ([91] page 406).

$O(n \log^* n + k \log n)$ si k est le nombre de composantes connexes du graphe. L'espace mémoire est linéaire en moyenne et le temps de localisation logarithmique en moyenne. Ces résultats ne nécessitent aucune hypothèse sur la distribution des sites.

O. Devillers, M. Teillaud et M. Yvinec [63] donnent un algorithme dynamique randomisé pour maintenir dynamiquement la carte des trapèzes d'un arrangement de segments. Ils ne font aucune hypothèse sur la distribution des sites et des segments, mais supposent seulement que l'ordre d'insertion des segments est une permutation quelconque, parmi les $n!$ permutations possibles d'un ensemble de n segments, de manière équiprobable. Si un segment est supprimé, ils supposent que c'est n'importe lequel de ces segments, également de manière équiprobable. Sous ces hypothèses, l'algorithme utilise un espace mémoire moyen $O(n + a)$, un temps d'insertion moyen $O(\log n + \frac{a}{n})$, un temps de suppression moyen $O((1 + \frac{a}{n}) \log \log n)$ et un temps de localisation $O(\log n)$ pour un point quelconque du plan, où a est la taille courante de l'arrangement et n le nombre courant de segments.

K. Mulmuley [155] utilise pour la localisation une hiérarchie de triangulations de Delaunay comprenant $O(\log n)$ niveaux. Les sites présents à un niveau de la hiérarchie constituent un échantillonnage aléatoire de ceux du niveau juste en dessous. Il y a des liens entre les triangles de deux niveaux adjacents si ceux-ci ont une intersection non vide, comme dans l'algorithme de Kirkpatrick. Le temps de localisation est en $O(\log^2 n)$. Cet algorithme est compliqué et l'espace mémoire requis est important.

Il existe d'autres publications sur les algorithmes dynamiques de localisation (voir par exemple [12] [49] [94] [164] [166] [167]).

9.1.9 Amélioration des constantes multiplicatives pour les complexités

Un article récent de Goodrich, Orletsky et Ramaiyer [93] essaie d'améliorer les constantes multiplicatives (qui sont parfois extrêmement grandes) cachées par la notation $O()$.

9.2 Algorithme de Kirkpatrick

Nous étudions dans ce paragraphe l'algorithme optimal de Kirkpatrick qui sera utilisé pour faire des comparaisons de performances avec les algorithmes de localisation du paragraphe 9.3.

9.2.1 Présentation théorique

Elle a été exposée sommairement dans le paragraphe 9.1.6. On suppose que le graphe est complètement triangulé. Etudions comment construire la hiérarchie de triangulations.

Pour passer d'une triangulation à la suivante, on retire un certain nombre de *sommets indépendants* avec leurs arêtes adjacentes et on retriangule les "trous" produits par ces suppressions. On dit que des sommets sont indépendants s'il n'existe aucune arête ayant deux d'entre eux comme extrémités. On obtient facilement ces sommets par un parcours *en largeur* (ou concentrique, par niveau [178]) du graphe.

Nous allons construire un ensemble de sommets indépendants de cardinal supérieur ou égal à $\frac{m}{26}$ si m est le nombre de sommets de la triangulation courante. Comme la triangulation est complète, elle a exactement $3m-6$ arêtes. La somme des degrés de tous les sites est égale au nombre d'arcs, soit exactement $6m-12$. Le degré moyen est inférieur à 6. Par un simple raisonnement par l'absurde, on déduit qu'il y a au moins $\frac{m}{2}$ sites[5] dont le degré est inférieur ou égal à 12. Appelons D_{12} l'ensemble des sites de degré inférieur ou égal à 12. Supposons que nous placions un site s_1 de D_{12} dans notre ensemble I de sites indépendants. Aucun site adjacent à s_1 ne peut être placé dans I, ce qui en exclut au plus 12. On peut donc trouver au moins $\frac{\|D_{12}\|}{13}$ sites indépendants dans D_{12} et donc $\frac{m}{26}$ sites indépendants dans la triangulation courante.

Le nombre de sites dans chaque triangulation étant borné par une suite géométrique décroissante, le nombre de triangulations sera en $O(\log n)$ et la complexité en espace mémoire en $O(n)$.

Examinons le temps du prétraitement :

1. il faut déjà disposer d'une triangulation complète du graphe, ce qui peut être obtenu théoriquement en $O(n)$ en utilisant l'algorithme de triangulation de Chazelle [45]. Cette méthode étant inapplicable en pratique, la triangulation complète du graphe se fera généralement en $O(n \log n)$ [90]. Si le graphe est une triangulation de Delaunay, on obtient facilement une triangulation complète en $O(n)$.

2. le retrait des sommets indépendants se fait avec un parcours en largeur du graphe, soit en temps linéaire par rapport au nombre de sites de la triangulation courante.

3. quand un site indépendant est retiré avec ses arêtes adjacentes, on obtient un polygone étoilé qu'il faut retrianguler. Ceci peut être réalisé avec l'algorithme de

[5]Amélioration : comme tous les sites ont un degré au moins égal à 3 et que le degré moyen est inférieur à 6, un raisonnement par l'absurde prouve qu'au moins $\frac{m}{2}$ sites ont un degré inférieur ou égal à 9.

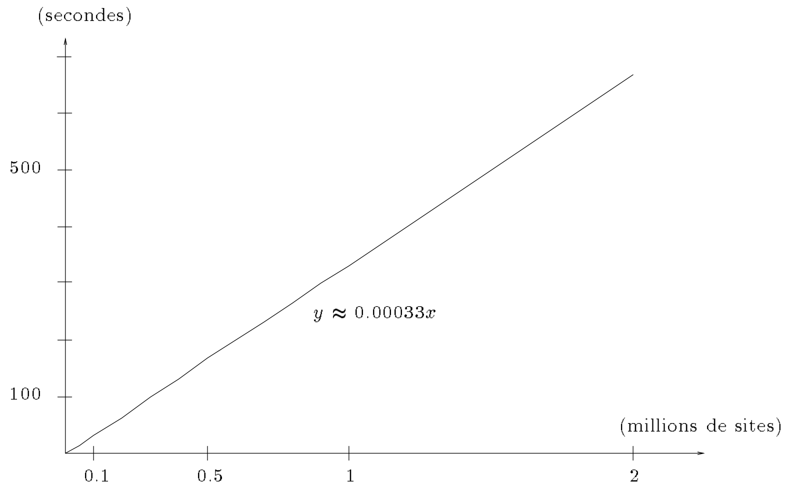


Figure 9.1: *Temps de prétraitement pour l'algorithme de Kirkpatrick.*

Schoone et van Leeuwen [174] qui est très simple, efficace et linéaire par rapport au nombre de côtés du polygone[6].

Le temps du prétraitement est en théorie linéaire pour n'importe quel graphe et en pratique linéaire pour un graphe déjà triangulé et en $O(n \log n)$ pour un graphe quelconque. Edahiro, Kokubo et Asano [73] proposent la variante suivante pour construire une surtriangulation. Soit $T(S)$ une triangulation complète de l'ensemble S des sites, D_i l'ensemble des sites de degré i, n_i le cardinal de D_i et I_i un ensemble de sommets indépendants de D_i.

Faire $I_j = \emptyset$ pour $j = 3, 4, 5, 6$
$i = 3$
Tant que $(i < 6)$ **Faire**
  Tant que $(D_i \neq \emptyset)$ **Faire**
    choisir un site s quelconque de D_i
    $I_i = I_i \cup \{s\}$
    Faire $D_j = D_j \setminus (Adj(s) \cup \{s\})$ pour $j = i, (i+1), \ldots, 6$
  Fin tant que
  $i = i + 1$
Fin tant que

On remarque que l'on cherche d'abord à retirer les sommets de faible degré. Cela

[6]Comme le nombre de côtés du polygone étoilé est borné par 12, on peut considérer que n'importe quel algorithme de triangulation est en $O(1)$.

permet de trouver à chaque étape plus de sommets indépendants, la retriangulation des polygones étoilés est plus rapide, la filiation de chaque triangle est moins nombreuse donc l'espace mémoire utilisé est plus faible.

Justification de l'algorithme : pour les mêmes raisons que précédemment, on peut trouver dans chaque D_i au moins $\frac{n_i}{i+1}$ sommets indépendants. Soit I l'ensemble des sites de degré inférieur ou égal à 6. On a, à condition de commencer à choisir les sommets de plus faible degré :

$$\begin{aligned}
|I| &= |I_3| + |I_4| + |I_5| + |I_6| \\
&\geq \frac{n_3}{4} + \frac{n_4}{5} + \frac{n_5}{6} + \frac{n_6}{7} \\
16|I| &\geq 4n_3 + \frac{16n_4}{5} + \frac{16n_5}{6} + \frac{16n_6}{7} \\
&\geq 4n_3 + 3n_4 + 2n_5 + n_6 \\
&= 6\sum_{j=0}^{+\infty} n_i - \sum_{j=0}^{+\infty}(in_i) + n_3 + n_4 + n_5 + n_6 + n_7 + 2n_8 + 3n_9 + 4n_{10} + \ldots
\end{aligned}$$

Comme $\sum_{j=0}^{+\infty}(in_i) = 6n - 12$, $\sum_{j=0}^{+\infty} n_i = n$ et que $n_0 = n_1 = n_2 = 0$, on déduit :

$$\begin{aligned}
16|I| &\geq 12 + n_3 + n_4 + n_5 + n_6 + n_7 + 2n_8 + 3n_9 + 4n_{10} + \ldots \geq 12 + n \\
\Longrightarrow |I| &> \frac{n}{16}
\end{aligned}$$

Les constantes sont légèrement améliorées.

9.2.2 Etude expérimentale

Les tests ont été réalisés sur le supercalculateur Convex C3, machine disposant d'une mémoire vive très importante (2 GO), mais dont la vitesse du processeur est équivalente à celle d'une station de travail bas de gamme. On a vérifié, avec une distribution uniforme des sites, que le temps de prétraitement (construction de la hiérarchie de triangulations) est linéaire (environ 330 microsecondes par site, voir Figure 9.1), que le temps de localisation est logarithmique (environ $31 \log N$ microsecondes par site, voir Figure 9.2) et que l'espace mémoire requis par la hiérarchie de triangulations est linéaire (environ 560 octets par site, la mémoire du calculateur sature à partir d'environ trois millions de sites).

Cet algorithme, bien qu'optimal, présente deux inconvénients majeurs : il exige un espace mémoire relativement important et il est statique (une fois que la hiérarchie de triangulations est construite, on ne peut plus ajouter ou supprimer des sites sans reconstruire complètement cette structure). Les algorithmes présentés dans la section suivante tentent de pallier ces deux inconvénients.

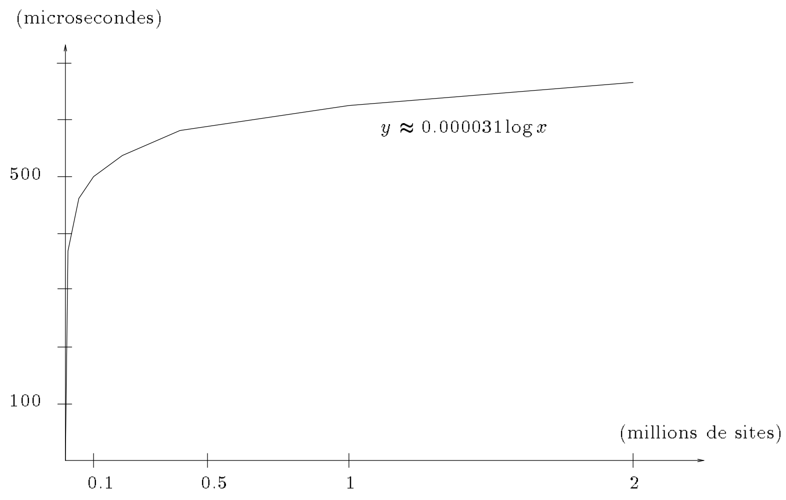


Figure 9.2: *Temps de localisation pour une requête avec l'algorithme de Kirkpatrick.*

9.3 Algorithmes pratiques

On cherche à localiser un point quelconque dans une triangulation de Delaunay. Nous avons choisi dans ce paragraphe de présenter des algorithmes de localisation ayant souvent une complexité dans le pire des cas médiocre, mais se comportant bien dans la pratique, tout au moins pour des données de taille raisonnable. Comme la probabilité d'apparition du *pire des cas* est extrêmement faible, ce choix se justifie. D'autre part, la plupart de ces algorithmes sont dynamiques et on utilise l'un d'entre eux pour construire efficacement une triangulation de Delaunay *on-line*[7].

9.3.1 Méthodes sans prétraitement

9.3.1.1 Méthode naïve

Il existe bien sûr la *méthode naïve* qui consiste à tester tous les triangles jusqu'à l'obtention du triangle contenant le point à localiser. Cette méthode est inefficace dès que le nombre de triangles devient important. Sa complexité (dans le pire des cas et en moyenne) est linéaire.

Une variante, que l'on nomme *Closest-and-Walk*, consiste à chercher le plus proche voisin (parmi les points du semis) du point à localiser, et ensuite de "marcher" par la méthode du *Walk-Through* (exposée dans le paragraphe suivant) en direction

[7]On appelle algorithme *en ligne* (on-line) un algorithme capable de maintenir la solution d'un problème lors de l'introduction successive de données, sans avoir une connaissance a priori de l'ensemble des données à traiter.

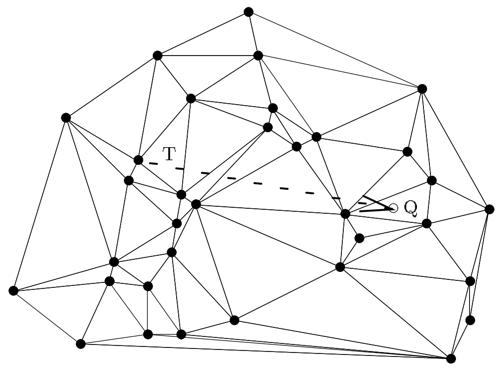


Figure 9.3: *Localisation avec le Walk-Through.*

du point à localiser. Sa complexité (dans le pire des cas et en moyenne) est linéaire puisqu'il faut tester tous les points du semis pour trouver le plus proche voisin.

9.3.1.2 Méthode du Walk-Through

Cette méthode est loin d'être nouvelle. On la trouve dans [97] et dans [37] où elle est utilisée dans un algorithme incrémental (voir paragraphe 1.4.2) de triangulation de Delaunay qui a une complexité en moyenne en $O(n^{\frac{3}{2}})$. L. de Floriani décrit aussi cette méthode dans [81]. Récemment, des chercheurs ont "réhabilité" cette méthode en l'analysant rigoureusement et en l'améliorant (voir paragraphe 9.3.1.3). Les principaux résultats sont dus à Bose et Devroye [35], qui étudient l'intersection d'objets géométriques aléatoires. Ils prouvent que, dans une triangulation de Delaunay, le nombre d'arêtes coupées par un segment de longueur $|\mathcal{L}|$ est, en moyenne, majoré par $c_3 + c_4|\mathcal{L}|\sqrt{n}$, où c_3 et c_4 sont des constantes positives indépendantes de $\mathcal{L}$ et n.

Le principe est très simple. Pour localiser un point Q quelconque (voir Figure 9.3), il suffit de prendre un triangle[8] T de la triangulation, de considérer le segment L reliant Q à un point intérieur[9] à T, de traverser tous les triangles qui coupent L en partant de T et en se dirigeant vers Q. On obtient ainsi le triangle (s'il existe) contenant Q. Il est sous-entendu que la structure de données représentant la triangulation permet de trouver en temps constant les triangles adjacents à un triangle donné, ainsi que ses trois sommets. C'est le cas de la structure de données du paragraphe 3.1, et, plus généralement, de toutes les structures de type *"carte planaire"*.

Cet algorithme peut fonctionner avec une triangulation quelconque (pas forcé-

[8] Soit l'on choisit un triangle aléatoire, soit on prend systématiquement un triangle situé approximativement au "centre" de la triangulation, soit on prend le triangle contenant le précédent point à localiser...

[9] On peut aussi initialiser cette procédure directement à partir d'un site de S_n.

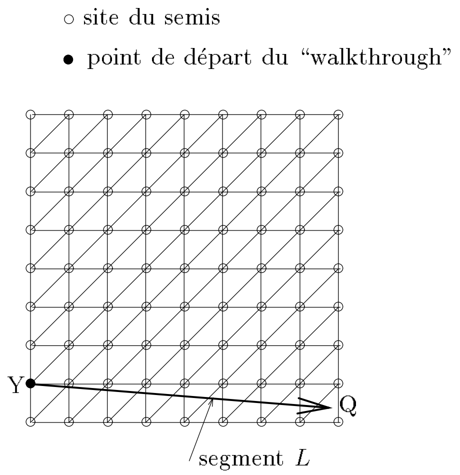


Figure 9.4: *Analyse approximative du Walk-Through : on "marche" à partir d'un site en direction de Q.*

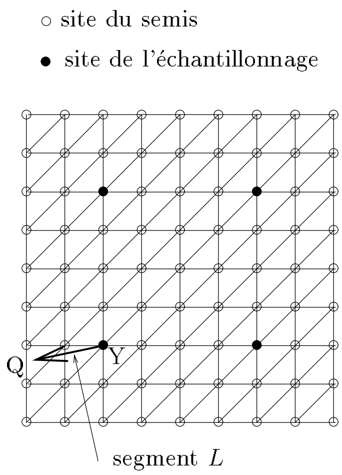


Figure 9.5: *Analyse approximative du Jump-and-Walk : échantillon E de m sites, puis Walk-Through à partir du site le plus proche de Q.*

ment de Delaunay), ou avec une triangulation contrainte à condition que son contour soit *convexe* et qu'il n'y ait pas de "trous".

La complexité dans le pire des cas est linéaire (dans le cas de la triangulation de deux segments parallèles discrétisés, le segment L peut traverser tous les triangles). La complexité en moyenne est en $O(\sqrt{n})$ [65].

Analyse intuitive : imaginons une distribution uniforme de n sites dans un carré. Pour simplifier, plaçons-les sur les noeuds d'une grille régulière de taille $\sqrt{n} \times \sqrt{n}$. Le segment L qui traverse le plus de triangles est la diagonale du carré (ou n'importe quel segment reliant un côté du carré au côté opposé) (voir Figure 9.4). $2(\sqrt{n}-1)$ triangles sont coupés, d'où la complexité en moyenne en $O(\sqrt{n})$. Expérimentalement, en choisissant un point de départ au centre de la triangulation, le segment L coupe en moyenne $0.66\ \sqrt{n}$ triangles.

9.3.1.3 Méthode du Jump-and-Walk

Devroye, Mücke et Zhu [65] ont amélioré la méthode précédente en choisissant un meilleur point de départ à l'aide d'un échantillonnage aléatoire sur l'ensemble des sites. De plus, ils prouvent que, si l'on sélectionne un échantillon aléatoire S_k de taille k parmi les sites de S_n, si l'on calcule l'élément σ de S_k qui est le plus proche du point Q à localiser, et que si l'on initialise le "Walk-Through" sur σ, alors la complexité en moyenne du "Jump-and-Walk" est dans $O(k + \sqrt{\frac{n}{k}})$. L'algorithme devient :

1. sélectionner un ensemble E de m points Y_1, Y_2,...,Y_m par un tirage aléatoire sans remise à partir de l'ensemble des sites.

2. déterminer l'index $j \in \{1, \ldots, m\}$ minimisant la distance euclidienne $d(Y_j, Q)$. Soit $Y = Y_j$.

3. trouver le triangle contenant Q en traversant tous les triangles coupés par le segment $L = [Y, Q]$.

Pour une distribution uniforme des sites, le nombre de triangles examinés par le Walk-Through est proportionnel à la longueur du segment reliant le point de départ et le point à localiser. Le Jump-and-Walk est meilleur que le Walk-Through parce qu'il permet de choisir un point de départ plus proche du point à localiser, le segment L est donc plus court. L. Devroye, E. Mücke et B. Zhu [65] démontrent que l'optimum est atteint avec un tirage aléatoire de $m = O(\sqrt[3]{n})$ sites et que la complexité en moyenne est en $O(\sqrt[3]{n})$. Ce résultat se déduit directement de la formule de complexité du Jump-and-Walk puisqu'elle est la somme d'une fonction croissante et d'une fonction décroissante. E. Mücke, I. Saias et B. Zhu [154] généralisent ce résultat (avec des complexités différentes) dans un espace à trois dimensions.

Analyse intuitive : imaginons un tirage uniforme de m sites. On peut aussi, pour simplifier, supposer que ces sites sont situés sur les noeuds d'une grille régulière de taille $\sqrt{m} \times \sqrt{m}$. Cette grille viendra se superposer à la grille plus fine de taille $\sqrt{n} \times \sqrt{n}$ contenant l'ensemble des sites. On s'aperçoit que le segment L coupera *au plus* $\dfrac{2(\sqrt{n}-1)}{2\sqrt{m}} \approx \sqrt{\dfrac{n}{m}}$ triangles (voir Figure 9.5). La complexité en moyenne de la localisation est donc en $O(\sqrt{\dfrac{n}{m}} + m)$. L'optimum est atteint pour :

$$\sqrt{\frac{n}{m}} = m \Longleftrightarrow m = \sqrt[3]{n}$$

La taille de l'échantillon est en $O(\sqrt[3]{n})$ ainsi que la complexité en moyenne de la localisation. Les constantes multiplicatives peuvent être ajustées expérimentalement.

9.3.2 Méthodes utilisant le prétraitement d'un autre algorithme

Les méthodes suivantes utilisant un AVL [5] (ou n'importe quel arbre binaire de recherche) ou un 2d-tree sont dues à Lemaire et Moreau. Remarquons que le prétraitement (AVL ou 2d-tree) peut être gratuit et avoir été calculé pour les besoins d'un autre algorithme. Par exemple, il existe des algorithmes de triangulation de Delaunay

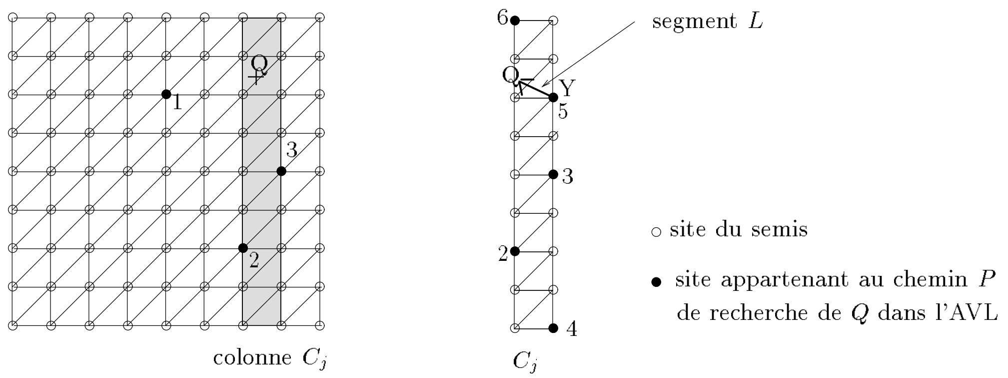


Figure 9.6: *Analyse approximative du Binary-Search-and-Walk : détermination de la colonne C_j contenant Q à l'aide de $\frac{\log_2 n}{2}$ dichotomies, puis Jump-and-Walk avec un échantillon de $\frac{\log_2 n}{2}$ sites dans la colonne C_j.*

qui utilisent pour le découpage du domaine, soit un arbre binaire de recherche comme l'algorithme "Divide-and-Conquer" de Lee et Schachter [131], soit un 2d-tree comme celui de Lemaire et Moreau [135] [136]. Tous les algorithmes de balayage utilisent aussi un tri des points qui peut être fait avec un arbre binaire. L'analyse qui en est faite n'a pas la prétention d'être une preuve mathématique, mais explique comment sont obtenus les résultats de complexité en moyenne. On essaie actuellement de donner une analyse mathématique rigoureuse de ces complexités en moyenne, comme Luc Devroye (*et al.*) l'ont fait avec le Walk-Through et le Jump-and-Walk. En ce qui concerne la complexité dans le pire des cas, on peut trouver pour tous ces algorithmes un exemple pathologique (dont la probabilité d'apparition en pratique est quasiment nulle) pour lequel la localisation est en $O(n)$.

Dans le cas où l'algorithme de triangulation est basé sur un quadtree (paragraphe 5.6) ou sur un bucket-tree (paragraphe 5.7), l'outil de localisation associé sera naturellement le quadtree ou la grille régulière.

9.3.2.1 Méthode du Binary-Search-and-Walk

Soit un AVL construit à partir des abscisses des sites (certains algorithmes de triangulation de Delaunay construisent un AVL dans une phase de prétraitement). Un AVL [5] est un arbre binaire de recherche équilibré dynamiquement. Imaginons que l'on recherche dans cette arborescence la clé égale à l'abscisse du point à localiser Q. Cette opération définit un chemin P dans l'AVL de la racine jusqu'à une feuille. Les points stockés au niveau des noeuds de ce chemin constitueront l'ensemble E du paragraphe 9.3.1.3. La suite de l'algorithme est identique.

Cette méthode présente l'avantage de placer dans l'ensemble E des points qui sont très proches en abscisse du point à localiser, mais dont la distance en ordonnée est aléatoire.

L'AVL étant dynamique, cette méthode peut être utilisée dans tout algorithme incrémental qui nécessite de la localisation, en particulier pour construire la triangulation de Delaunay de façon incrémentale.

Remarque : cet algorithme peut fonctionner avec n'importe quel arbre binaire de recherche.

Analyse intuitive : imaginons une distribution uniforme de n sites dans un carré. Pour simplifier, plaçons-les sur les noeuds d'une grille régulière de taille $\sqrt{n} \times \sqrt{n}$ (voir Figure 9.6).

- **Majorant pour la complexité en moyenne :**

  coupons le chemin P en deux parties égales P_1 et P_2 contenant environ $\frac{\log_2 n}{2}$ sites. La descente le long de P_1 revient à diviser récursivement le domaine en bandes verticales. La dernière bande, au bout de $\frac{\log_2 n}{2}$ étapes, aura une largeur égale environ à la racine carrée de la longueur du côté du carré. Cette bande est approximativement égale à une colonne C_j de la grille régulière. Tous les sites de la seconde partie P_2 du chemin sont situés à une position aléatoire en ordonnée dans cette colonne C_j qui contient le point à localiser. On peut considérer que ces $\frac{\log_2 n}{2}$ points constituent un échantillonnage aléatoire dans cette colonne. Le segment L qui traverse le plus de triangles, en coupera environ $\frac{2\sqrt{n}}{2^{\frac{\log_2 n}{2}}} = O(\frac{\sqrt{n}}{\log_2 n})$.

- **Minorant pour la complexité en moyenne :**

  au mieux, tous les sites du chemin P, soit $(\log_2 n)$, sont situés dans la même colonne C_j qui contient le point à localiser. Le segment L qui traverse le plus de triangles, en coupera environ $\frac{2\sqrt{n}}{2\log_2 n} = \Omega(\frac{\sqrt{n}}{\log_2 n})$.

- **Valeur exacte pour la complexité en moyenne :**

  elle est donc en $\Theta(\frac{\sqrt{n}}{\log_2 n} + \log_2 n) = \Theta(\frac{\sqrt{n}}{\log_2 n})$. Le premier terme correspond au nombre de triangles coupés par le segment L, le second terme est égal au nombre d'éléments dans le chemin P.

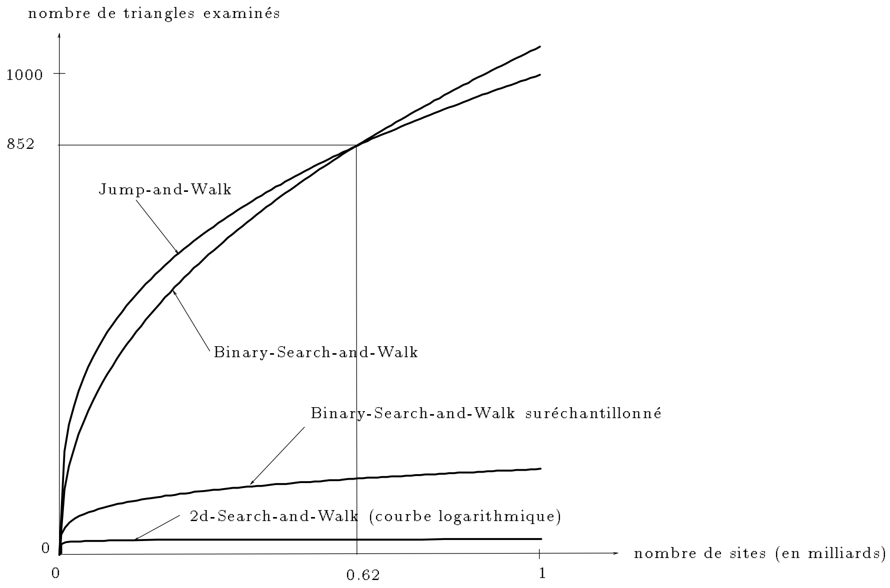


Figure 9.7: *Comportement asymptotique du Jump-and-Walk, du Binary-Search-and-Walk simple et suréchantillonné et du 2d-Search-and-Walk (courbe logarithmique).*

9.3.2.2 Comparaison entre le Jump-and-Walk et le Binary-Search-and-Walk

La fonction $\sqrt[3]{n}$ croît *asymptotiquement* moins vite que la fonction $\frac{\sqrt{n}}{\log n}$, donc la méthode du Jump-and-Walk est meilleure *asymptotiquement*. Mais, pour moins de 620 millions de sites, c'est le contraire. Pour les ensembles de points que l'on rencontre en pratique, la méthode du Binary-Search-and-Walk est plus rapide (cf. courbes de la figure 9.7). Ceci montre une nouvelle fois que les notations asymptotiques doivent être utilisées avec prudence. Dans l'étude expérimentale du paragraphe 9.3.3, on détermine le seuil exact où le classement des deux algorithmes va s'inverser. En effet, des constantes multiplicatives, ignorées dans ce raisonnement, interviennent dans les complexités.

9.3.2.3 Méthode du Binary-Search-and-Walk suréchantillonné

D'après l'analyse du paragraphe 9.3.2.1, il y a entre $\frac{\log_2 n}{2}$ et $\log_2 n$ sites qui servent à faire de l'échantillonnage aléatoire dans la colonne C_j. Ce nombre n'est pas optimal parce que trop faible, surtout quand n est grand.

Augmentons la taille k de l'échantillonnage dans la colonne C_j. La complexité de la localisation devient $(\frac{\log_2 n}{2} + \frac{2\sqrt{n}}{2k} + k)$. Le premier terme correspond au parcours

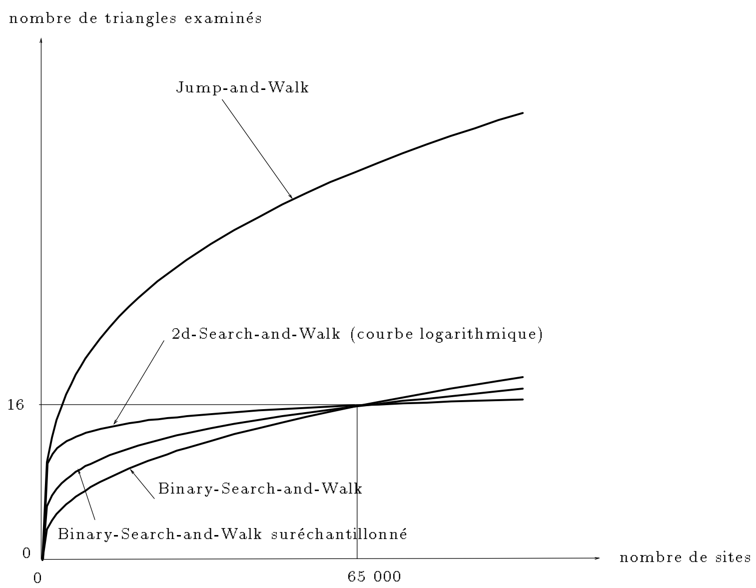


Figure 9.8: *Comparaison (N moyen) du Jump-and-Walk, du Binary-Search-and-Walk simple et suréchantillonné et du 2d-Search-and-Walk (courbe logarithmique). Curieusement, les fonctions* $\log_2 n$, $\frac{\sqrt{n}}{\log_2 n}$ *et* $\sqrt[4]{n}$ *sont égales à 16 pour* $x = 2^{16}$.

du chemin P_1, le second terme représente le nombre de triangles coupés par le segment L et le dernier terme représente les k points de l'échantillonnage qu'il faut examiner pour trouver le plus proche voisin du point à localiser. Quand k augmente, le second terme diminue, l'optimum est donc atteint pour :

$$\frac{\sqrt{n}}{k} = k \Longleftrightarrow k = \sqrt[4]{n}$$

Le terme logarithmique étant asymptotiquement plus faible, la complexité de la localisation est en $\Theta(\sqrt[4]{n})$. Remarquons que la fonction racine quatrième croît très lentement : pour environ 8 milliards de sites, sa valeur n'est que 10 fois plus importante que celle prise par la fonction logarithmique en base 2.

Pour augmenter la taille du suréchantillonnage, il suffit, à partir d'un noeud Y_s (voir Figure 9.9) sur le chemin P, de prendre l'arbre ayant Y_s pour racine et d'ajouter tous les sites stockés dans les noeuds de Y_s, dans l'ensemble d'échantillonnage E. Cette opération se fait en temps linéaire par rapport à la taille du sous-arbre et n'occasionne donc pas de coût supplémentaire. Supposons l'arbre complet : la taille du sous-arbre est environ égale à $2^{\log n - s}$. Il suffit de déterminer l'indice s tel que la taille du sous-arbre ancré en Y_s soit de taille proche de $\sqrt[4]{n}$. L'indice s est donc environ égal aux trois-quarts de la hauteur de l'arbre puisque $n^{1/4} = 2^{\frac{\log_2 n}{4}}$.

Cette méthode a l'inconvénient d'introduire des discontinuités dans les tailles des ensembles d'échantillonnage qui sont approximativement égales à des puissances de deux. D'autre part, on ne peut prévoir avec précision, excepté dans le cas d'un arbre complet, quelle va être la taille de l'arbre enraciné sur un noeud du chemin de recherche. C'est d'autant plus vrai dans le cas d'un AVL de hauteur h dont la taille peut varier entre $\Phi^{h+1} - 1$ et $2^{h+1} - 1$, $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1.61$ étant le nombre d'or (voir par exemple [13] page 153).

Il existe une autre façon de compléter l'ensemble d'échantillonnage : considérons le chemin de recherche P ordonné en sens inverse $\{Y_h, Y_{h-1}, Y_{h-2}, \ldots, Y_0\}$. On associe à chaque noeud de P celui de ses fils (quand il existe) qui n'appartient pas à P, d'où l'ensemble $\{S_h, S_{h-1}, S_{h-2}, \ldots, S_0\}$. On effectue un parcours en profondeur de l'arbre enraciné en chaque noeud S_i en commençant par S_h et en continuant avec S_{h-1}, S_{h-2}... suivant les indices décroissants. Les noeuds visités lors de ce parcours sont placés dans l'ensemble d'échantillonnage et on incrémente un compteur pour chaque noeud visité. On stoppe le parcours quand le nombre souhaité est atteint. La récursivité de cette méthode entraîne une légère perte de temps, vite compensée dès que le semis dépasse une certaine taille (voir l'étude expérimentale paragraphe 9.3.3). On a le schéma[10] algorithmique suivant :

Fonction OBSW $(Root, QueryPoint, Neighbor)$
  $K = 1.57\sqrt[4]{N}$ et dmin = RealMax
  BinSearch $(Root, QueryPoint, Neighbor, dmin, K)$

Fonction BinSearch $(Node, Q, N, dmin, K)$
  Si $(Node = NULL)$ **alors** return
  d = distance $(Node, Q)$ et $K--$
  Si $(d < dmin)$ **alors** N = Node et dmin = d
  Si $((Q)_x < (Node)_x)$ **alors** BinSearch $(LeftSon(Node), Q, N, dmin, K)$
      OtherSon = RightSon(Node)
  Sinon BinSearch $(RightSon(Node), Q, N, dmin, K)$
      OtherSon = LeftSon(Node)
  Si $(K > 0)$ **alors** OverSample $(OtherSon, Q, N, dmin, K)$

Fonction OverSample $(Node, Q, N, dmin, K)$
  Si $(Node = NULL)$ **alors** return
  d = distance $(Node, Q)$ et $K--$
  Si $(d < dmin)$ **alors** N = Node et dmin = d
  Si $(K > 0)$ **alors** OverSample $(LeftSon(Node), Q, N, dmin, K)$
  Si $(K > 0)$ **alors** OverSample $(RightSon(Node), Q, N, dmin, K)$

[10] Par souci de clarté, nous n'avons pas précisé les passages par adresse de certains paramètres. K représente la taille de l'échantillonnage dont on a déterminé expérimentalement la valeur optimale.

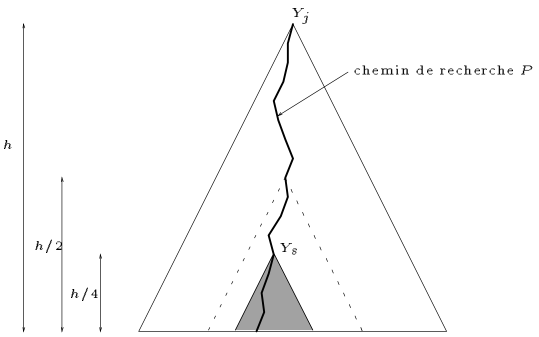


Figure 9.9: *Binary-Search-and-Walk.*

Remarque : on peut accélérer en pratique la localisation en ne calculant que la différence d'ordonnée entre le point à localiser et les points de l'ensemble d'échantillonnage. Le candidat retenu pour initialiser le "walk" sera celui qui est le plus proche en ordonnée du point à localiser. Mais dans ce cas, il ne faut pas placer dans l'ensemble d'échantillonnage les premiers sites du chemin de recherche P parce qu'ils sont généralement trop éloignés en abscisse du point à localiser.

9.3.2.4 Comparaison entre le Binary-Search-and-Walk simple et suréchantillonné

Le Binary-Search-and-Walk suréchantillonné (*Oversampled-Binary-Search-and-Walk*) est meilleur asymptotiquement (voir Figure 9.7) que le Binary-Search-and-Walk simple et même que le Jump-and-Walk. Jusqu'à 65 000 points, le terme logarithmique présent dans les deux algorithmes est prépondérant (voir Figure 9.8), le suréchantillonnage est inutile en dessous de cette valeur. L'étude expérimentale du paragraphe 9.3.3 détermine plus précisément à partir de quelle taille de semis le suréchantillonnage apporte un gain de temps. En effet, certaines constantes multiplicatives dans les complexités modifient sensiblement le seuil annoncé de 65 000.

9.3.2.5 Méthode du 2d-Search-and-Walk

Cette méthode de localisation est idéale si l'on a déjà construit la triangulation de Delaunay avec l'algorithme du chapitre 5 qui utilise le découpage du domaine par un 2d-tree. Il suffit, lors de la construction du 2d-tree, de garder en mémoire les médianes (sites) ayant servi à partager chaque région (un simple tableau de n pointeurs suffit !). Un 2d-tree peut être considéré comme une arborescence binaire de recherche : il suffit d'alterner dans les parcours de recherche, la coordonnée qui sert à faire la discrimination. Supposons que l'on recherche dans cette arborescence la clé égale aux

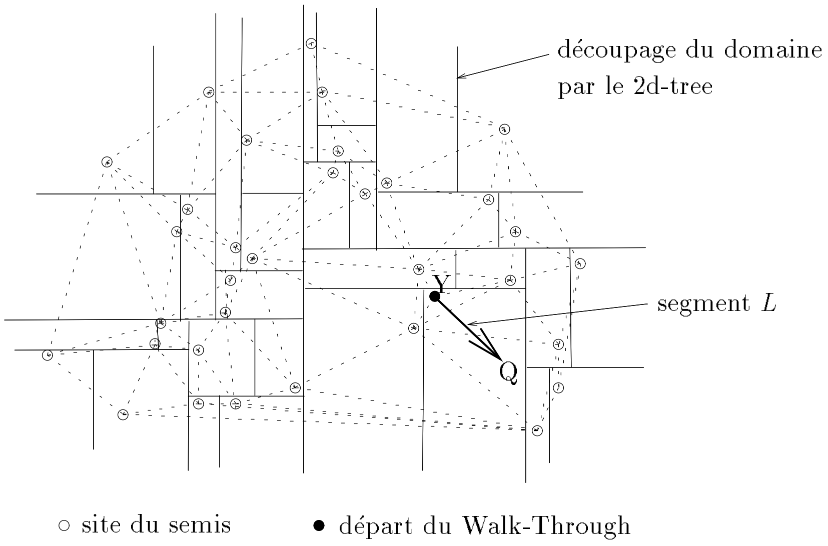


Figure 9.10: *Localisation avec un 2d-tree.*

coordonnées du point à localiser Q. Le chemin emprunté lors de cette recherche se termine sur une feuille à laquelle sont associés un site Y et un rectangle ne contenant que Y et Q. Souvent, ce site Y est le plus proche voisin de Q (voir aussi les développements de Devroye et Chanzy [44] sur la recherche du plus proche voisin et le range searching à l'aide d'un 2d-tree). On considère le segment $L = [YQ]$ et la suite de l'algorithme est identique au Walk-Through. Cette méthode est performante parce que les points Y et Q sont généralement extrêmement proches (voir Figure 9.10). *En moyenne*, L coupe un nombre constant k de triangles. La complexité de la localisation est en $O(\log_2 n + k) = O(\log_2 n)$. Cette méthode est meilleure que les précédentes, asymptotiquement et en pratique, puisque ces dernières ont au moins une composante logarithmique dans la complexité. Par contre, le 2d-tree se dynamise mal. Il sera difficilement utilisable dans un environnement dynamique.

9.3.3 Etude expérimentale

Les tests comparatifs ont été réalisés, comme dans le paragraphe 9.2.2, sur le super-calculateur Convex avec des distributions uniformes, jusqu'à saturation de la mémoire vive. On a atteint 3 millions de sites avec l'algorithme de Kirkpatrick et 13 millions de sites avec les autres algorithmes.

Les algorithmes testés sont celui de Kirkpatrick (**Kirk**), la méthode naïve (**Brute**), le Walk-Through (**Walk**), le Jump-and-Walk (**JW**), le Binary-Search-and-Walk (**BSW**), l'Oversampled-Binary-Search-and-Walk (**OBSW**) et enfin le 2d-Search-

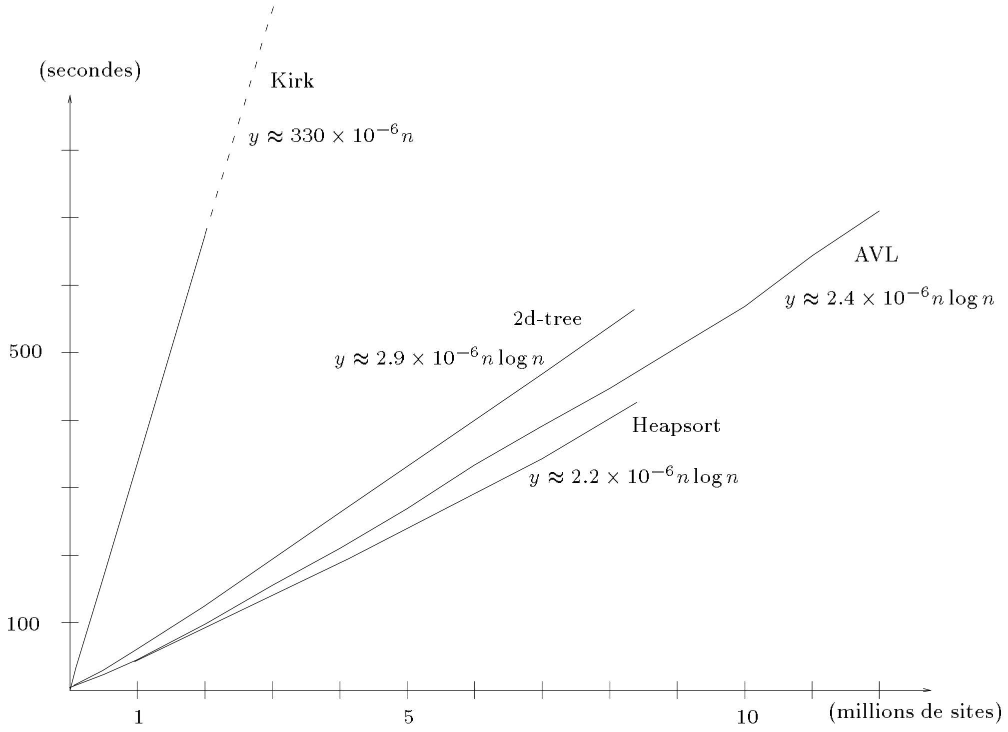


Figure 9.11: *Comparaison des temps de prétraitement des algorithmes de localisation avec une distribution uniforme.*

and-Walk (**2dSW**).

La mémoire nécessaire pour stocker la triangulation ainsi que les éventuelles structures de localisation est d'environ 560 octets par site pour **Kirk**. Elle peut être un peu plus importante dans le cas d'une distribution hétérogène : la taille de la hiérarchie de triangulations n'est pas fixe, on peut seulement la majorer. **BSW** et **OBSW** utilisent 140 octets par site : il faut stocker l'AVL dont la taille peut se calculer à l'avance et ne dépend pas du type de la distribution. Un site doit avoir un pointeur sur son fils gauche et droit ainsi qu'un entier indiquant au minimum l'équilibrage des sous-arbres. **Brute**, **Walk** et **JW** ne requièrent aucune structure particulière et 124 octets par site suffisent pour stocker la triangulation. Chaque site a un pointeur sur une arête ayant ce site pour origine, chaque arête (ou plutôt arc) a un pointeur indiquant si l'arc est sur l'enveloppe convexe. La structure de localisation pour **2dSW** est un simple tableau de n pointeurs sur les sites médians et cet algorithme nécessite 128 octets par site.

Kirk, **BSW**, **OBSW** et **2dSW** requièrent un prétraitement. Il consiste pour **Kirk** à construire une hiérarchie de triangulations, pour **BSW** et **OBSW** un AVL et pour **2dSW** un 2d-tree. On trouve dans le graphique 9.11 une comparaison expérimentale entre ces différents prétraitements (le temps de calcul du tri par *Heapsort* sert de référence). En ce qui concerne le temps de prétraitement, **Kirk** est le meilleur asymptotiquement, mais le moins bon en pratique. L'extrapolation de ces courbes montre que **Kirk** est meilleur que **2dSW** à partir d'environ 2×10^{34} sites et que **BSW** (ou **OBSW**) à partir d'environ 2.5×10^{41} sites[11] ! Remarquons aussi que le temps de construction d'un AVL est à peine supérieur au temps pris par un *Heapsort*.

Les graphiques 9.12 et 9.13 illustrent les performances de ces différents algorithmes.

Algorithmes dynamiques sans prétraitement : le moins performant est **Brute**, inutilisable au-delà de quelques centaines de points. Les performances de **Walk** restent acceptables jusqu'à environ 5000 sites, et sont, sur cet intervalle, légèrement meilleures que celles de **Kirk**. Par contre, **JW** reste utilisable sur de gros ensembles (plusieurs millions de sites), mais en étant nettement moins performant que les autres algorithmes. Il est même plus rapide que **Kirk** jusqu'à 140 000 sites.

Algorithmes dynamiques avec prétraitement : il s'agit de **BSW** et de **OBSW**. **BSW** est moins bon asymptotiquement que **JW**. Mais, si l'on extrapole les courbes des temps de localisation, **BSW** est meilleur jusqu'à environ 17.7 milliards de sites ! Jusqu'à environ 90 000 sites, **OBSW** est légèrement plus lent que **BSW**. Cette différence est due aux fonctions récursives qu'il utilise. Ensuite, **OBSW** devient nettement plus rapide. **BSW** reste plus rapide que **Kirk** jusqu'à environ 3 millions de sites. En extrapolant les courbes représentatives de **Kirk** et **OBSW**, **OBSW** reste plus rapide que **Kirk** jusqu'à environ 37 millions de sites.

Algorithmes statiques avec prétraitement : le meilleur est incontestablement **2dSW**. Sa complexité est en moyenne logarithmique, avec une très petite constante multiplicative. Mais, il a l'inconvénient de ne pas être optimal dans le pire des cas et il existe des configurations (exemples pathologiques quasiment introuvables en pratique) où son comportement pourra dégénérer vers du linéaire. **Kirk** est moins bon en pratique, mais reste optimal dans le pire des cas, ce qui signifie que, quelle que soit la distribution, son comportement restera logarithmique mais avec une constante multiplicative assez énorme.

[11]Cette quantité est à peu près la racine carrée du nombre estimé d'atomes présents dans l'univers observable !

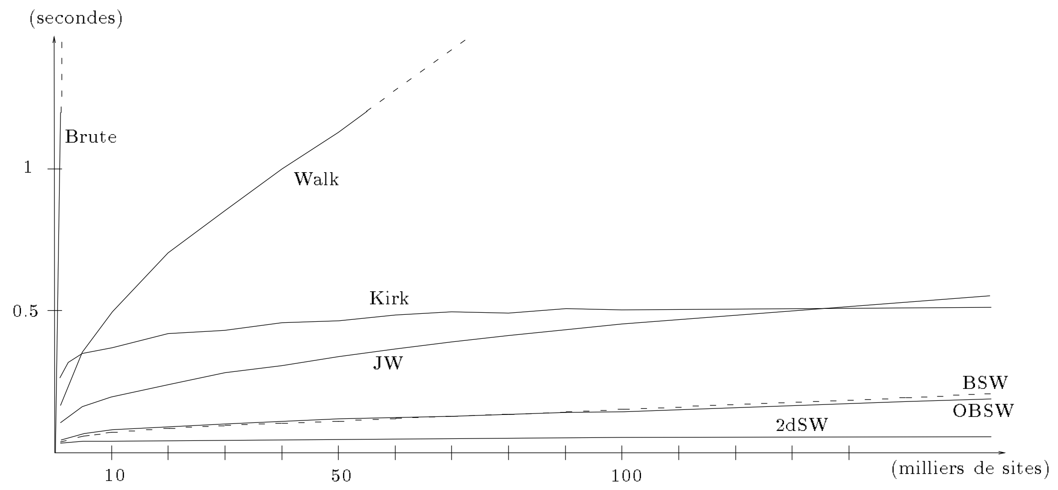


Figure 9.12: *Comparaison des algorithmes de localisation (N moyen).*

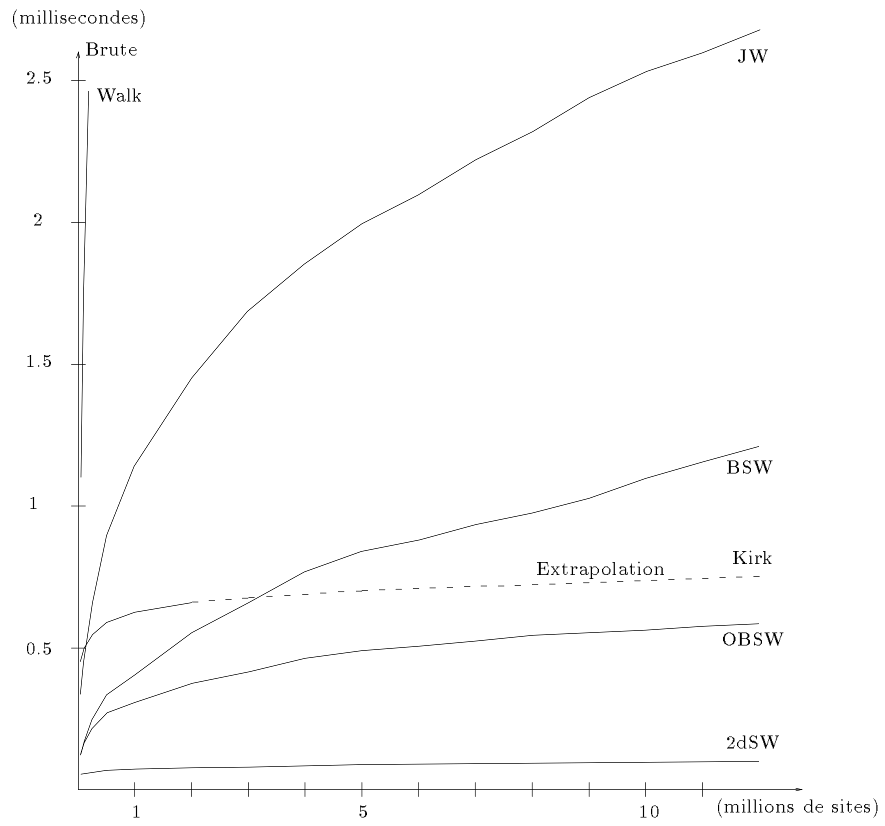


Figure 9.13: *Comparaison des algorithmes de localisation (N grand).*

Algorithme	Temps de prétraitement (μs)	Nombre d'arêtes traversées	Temps de localisation (μs)
Brute	–	1.05	$1.1N$
Walk	–	$1.12\sqrt{N}$	$7\sqrt{N}$
JW	–	$1.33\sqrt[3]{N}$	$12\sqrt[3]{N}$
BSW	$2.4N \log N$	$1.25\frac{\sqrt{N}}{\log_2 N}$	$8\frac{\sqrt{N}}{\log_2 N}$
OBSW	$2.4N \log N$	$0.73\sqrt[4]{N}$	$10\sqrt[4]{N}$
2dSW	$2.9N \log N$	1.53	$4 \log_2 N$
Kirk	$330N$	–	$31 \log_2 N$

Le tableau ci-dessus résume l'étude expérimentale, les sites variant dans l'intervalle $[10^3, 1.2 \times 10^7]$ en étant uniformément distribués dans un carré. Le nombre d'arêtes traversées reflète le travail accompli lors de la marche en direction du site à localiser. Ce nombre suit parfaitement les fonctions mentionnées dans la troisième colonne du tableau. Par contre, les temps de localisation fluctuent un peu plus par rapport aux fonctions mentionnées dans la quatrième colonne. La mesure des temps de calcul n'est pas toujours très précise et n'est pas identique d'une session à l'autre. Cette imprécision des mesures rend inexploitable les régressions que nous avons réalisées pour tenter de mettre en évidence les composantes logarithmiques qui existent dans **BSW** et dans **OBSW**. Néanmoins, ces expériences permettent de mesurer assez précisément la fonction dominante dans la complexité de ces algorithmes.

Nous n'avons pas réalisé de tests exhaustifs sur d'autres types de distributions comme nous l'avions fait pour la triangulation de Delaunay. Pour vérifier que toute cette théorie ne s'effondre pas dans le cas de distributions très hétérogènes, nous avons lancé **BSW** sur une distribution en clusters. Le temps de localisation est augmenté d'environ 25%, ce qui reste encore très acceptable. On se doute intuitivement que ces algorithmes, parce qu'ils utilisent la randomisation ou les 2d-trees, vont résister relativement bien aux distributions hétérogènes qui peuvent exister dans les applications réelles.

En conclusion, **OBSW** est le meilleur des algorithmes dynamiques de localisation pour les tailles de semis que l'on rencontre en pratique et **2dSW** est le meilleur des algorithmes statiques (ses performances doivent être comparables à celles du quadtree, mais sans en avoir les inconvénients déjà évoqués dans le chapitre 2). Ces algorithmes requièrent un espace mémoire extrêmement faible. La construction de la structure de localisation peut avoir été réalisée avec un coût presque nul pendant la construction de la triangulation de Delaunay : l'algorithme dirigé par le 2d-tree nécessite la construction du 2d-tree utilisé par **2dSW** et l'algorithme de Lee et Schachter trie les points selon les abscisses ce qui donne implicitement l'arbre binaire de recherche utilisé par **BSW** ou **OBSW**. Nous allons voir, dans le paragraphe suivant, un algorithme dynamique de triangulation de Delaunay utilisant l'*Oversampled-Binary-Search-and-Walk* comme méthode de localisation.

Remarque : nous avons réalisé les mêmes tests avec un arbre binaire parfaitement équilibré au lieu d'un AVL. On observe une amélioration des performances d'environ 5%. Le gain est nettement plus important si l'on écrit les procédures de localisation de façon non récursive (15 à 20%).

9.4 Application à la triangulation de Delaunay

Il est habituel de distinguer, parmi les algorithmes de triangulation de Delaunay, ceux qui sont :

off-line : on connaît à l'avance l'ensemble des données à traiter. Ces algorithmes sont statiques et sont inadaptés dès que l'on modifie l'ensemble des données initiales (ajout ou suppression). Dans ce cas, il faut recommencer entièrement le calcul. Les algorithmes du chapitre 5 entrent dans cette catégorie.

on-line : on ne connaît pas a priori l'ensemble des données à traiter et la triangulation est mise à jour à chaque fois que l'on ajoute ou retire un site. Ces algorithmes sont dynamiques, mais toujours plus lents que ceux qui sont statiques. Cette fonctionnalité supplémentaire n'est pas gratuite, le choix entre *off-line* et *on-line* n'est pas systématique, il dépend de l'application. Pour un simulateur de vol ou pour le déplacement d'un robot, on préfèrera bien sûr un algorithme *on-line*. En effet, la triangulation qui représente le terrain naturel, évolue sans cesse en fonction du déplacement du robot ou de l'avion. Les algorithmes du paragraphe 1.4.2 entrent dans cette catégorie (revoir leur description sommaire dans le paragraphe 1.4.2). A chaque fois que l'on ajoute (ou retire) un site, il faut localiser le triangle contenant ce site, puis mettre à jour la triangulation. C'est la phase de localisation qui est difficile. Elle est à l'origine de la très grande disparité dans les performances de ce type d'algorithme.

Nous allons comparer les meilleurs de ces algorithmes *on-line* : **Dt** utilise le Delaunay tree, **D-JW** le Jump-and-Walk, **D-BSW** le Binary-Search-and-Walk, **D-OBSW** l'Oversampled-Binary-Search-and-Walk, **D-HT** une hiérarchie de triangulations. Les graphiques 9.14 et 9.15 (excepté pour **D-HT**) illustrent les performances pratiques de ces différents algorithmes. Les temps de calcul sont toujours mesurés sur le Convex et la distribution des sites est uniforme dans un carré.

9.4.1 Algorithme avec Delaunay tree

Cet algorithme [33] a déjà été décrit dans le paragraphe 1.4.2. Il est randomisé et optimal en moyenne. Dans l'analyse, on suppose que l'ordre d'insertion des sites est une permutation quelconque parmi les $n!$ permutations possibles, de manière équiprobable. Aucune hypothèse n'est faite sur la distribution des sites. Cet algorithme est relativement lent pour des ensembles de sites de taille moyenne, mais devient excellent à partir de plusieurs millions de sites (voir Graphiques 9.14 et 9.15). Par contre, il a l'inconvénient de consommer beaucoup de mémoire : pour cette raison, nous n'avons pas pu le tester pour plus de deux millions de sites, alors que nous avons atteint 12 millions pour les algorithmes **D-JW**, **D-BSW** et **D-OBSW**.

9.4.2 Algorithme avec Jump-and-Walk

La méthode de localisation employée est le *Jump-and-Walk* (voir paragraphe 9.3.1.3). Cet algorithme [65] [154] a été proposé par Devroye, Mücke et Zhu en 1995. Pour une distribution uniforme, l'algorithme a une complexité en moyenne dans $O(n^{4/3})$, avec une petite constante multiplicative. Son comportement asymptotique n'est pas optimal, mais les performances pratiques sont bonnes jusqu'à environ 500 000 points. Au-delà, **Dt** (asymptotiquement optimal) est meilleur.

9.4.3 Algorithme avec Oversampled-Binary-Search-and-Walk

Cet algorithme a une complexité en moyenne en $O(n^{5/4})$ avec une constante multiplicative très petite. Il est un peu plus rapide que **D-BSW** à partir de 300 000 points. Ses performances restent très bonnes jusqu'à plusieurs millions de sites et il est le meilleur des algorithmes que nous avons pu tester. Il présente l'avantage d'utiliser peu de mémoire et d'être simple à coder en étant complètement dynamique. La localisation d'un site à supprimer se fait de façon optimale en $O(\log n)$ grâce à la structure d'AVL. D'autre part, beaucoup d'algorithmes géométriques utilisent un tri lexicographique des sites. Ici, le tri est maintenu dynamiquement par l'AVL.

En pratique, le schéma algorithmique est proche de celui décrit dans le paragraphe 9.3.2.3. Le travail réalisé dans la fonction *BinSearch* se fait maintenant dans la fonction d'insertion dans l'AVL et la fonction *OverSample* est déclenchée juste avant l'appel de la fonction de rééquilibrage dynamique de l'AVL.

9.4.4 Algorithme avec une hiérarchie de triangulations

O. Devillers [61] a présenté cet algorithme fin septembre 1997 aux Journées Franco-Espagnoles de Géométrie Algorithmique. La méthode employée pour la localisation est

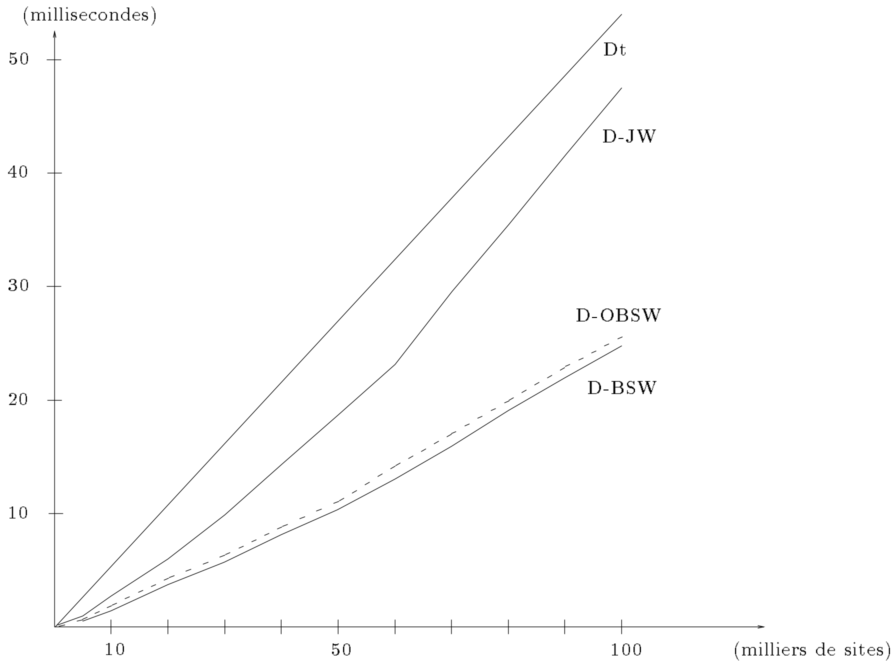


Figure 9.14: *Comparaison des algorithmes* on-line *de triangulation (N moyen).*

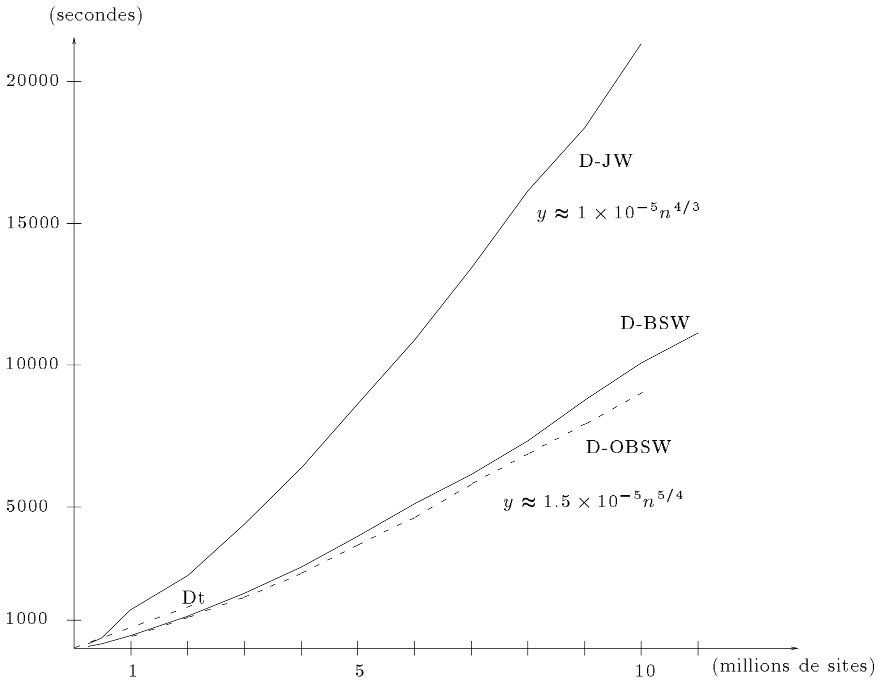


Figure 9.15: *Comparaison des algorithmes* on-line *de triangulation (N grand).*

proche de celle de Mulmuley [155] (voir paragraphe 9.1.8), mais beaucoup plus simple et moins gourmande en mémoire.

Chaque niveau de la hiérarchie est une triangulation de Delaunay. Un site a un lien sur tous les triangles dont il est le sommet. Un triangle a un lien sur les trois triangles adjacents ainsi que sur ses trois sommets. Les sites présents à un niveau de la hiérarchie constituent un échantillonnage aléatoire de ceux du niveau juste en dessous.

Pour situer un point q, on le localise dans toutes les triangulations de la hiérarchie en partant de la plus grossière vers la plus fine. Pour une triangulation de niveau k, on part d'un site de cette triangulation et on “marche” en direction de q pour trouver le triangle contenant q. Ensuite, en utilisant les liens entre triangles adjacents, on trouve le plus proche voisin v de q parmi les sites présents dans le niveau traité. Comme v a un lien sur un triangle adjacent du niveau $k-1$, on peut initialiser le “Walk-Through” dans la triangulation du niveau $k-1$. On itère ce processus jusqu'à la triangulation de Delaunay de l'ensemble des sites. Dans la triangulation la plus grossière, si elle contient plus de 20 sites, on cherche le triangle contenant q avec la méthode du Jump-and-Walk plutôt qu'avec un simple Walk-Through.

La complexité en moyenne du calcul de la triangulation (en supposant un ordre d'insertion aléatoire et sans faire d'hypothèses sur la distribution des points) est en $O(n \log n)$, l'espace mémoire (avec les mêmes hypothèses) linéaire.

O. Devillers nous a généreusement prêté son programme. Nous n'avons pas pu le tester de façon approfondie, étant à quelques jours de la fin de ce travail de thèse. D'autre part, il est codé en C++ et un portage est nécessaire pour le faire fonctionner sur le Convex. Les quelques tests réalisés sur station montrent un excellent comportement avec un léger avantage de **D-HT** sur **D-OBSW**. **D-HT** et **D-OBSW** sont un peu plus de deux fois plus rapides que **D-JW** pour 500 000 points. Objectivement, nous ne pouvons en tirer de conclusions, notre programme n'étant pas codé de façon optimale. Il est certain que ces deux algorithmes ont des performances très proches et une étude plus approfondie est nécessaire pour pouvoir les comparer.

On peut néanmoins affirmer que, comme **D-HT** est asymptotiquement optimal, il existe une taille de semis à partir de laquelle il sera le plus rapide. La question que l'on se pose est de trouver (s'il existe) le domaine pour lequel **D-OBSW** serait meilleur. L'espace mémoire utilisé par ces deux algorithmes est comparable mais **D-OBSW** a les avantages suivants : il est plus simple à coder, plus rapide pour la suppression d'un site et il maintient dynamiquement une structure d'AVL qui peut être utilisée pour résoudre beaucoup d'autres problèmes géométriques. L'espace mémoire utilisé par ces deux algorithmes est comparable.

Une étude plus approfondie s'impose qui devrait commencer par une comparaison des méthodes de localisation de **D-OBSW** et de **D-HT**.

9.5 Conclusion

Nous avons présenté des algorithmes pratiques de localisation dans une triangulation de Delaunay qui peuvent compléter naturellement les algorithmes présentés dans le chapitre 5. En particulier, l'algorithme statique *2d-Search-and-Walk* est idéal si l'on a déjà calculé la triangulation avec l'algorithme basé sur le 2d-tree.

Nous avons proposé un nouvel algorithme dynamique de localisation dans une triangulation de Delaunay, l'*Oversampled-Binary-Search-and-Walk*, dont l'étude expérimentale a mis en évidence les remarquables performances.

Cet algorithme de localisation est utilisé pour construire *on-line* une triangulation de Delaunay et cette méthode est parmi les plus performantes des méthodes *on-line* connues à ce jour.

CONCLUSION ET PERSPECTIVES

La majeure partie des travaux effectués lors de cette thèse concerne la triangulation de Delaunay. Ce sujet suscite l'enthousiasme des chercheurs depuis plus de vingt ans et donne toujours matière à de nombreuses publications.

La première partie présente de nouveaux algorithmes de triangulation euclidienne de Delaunay dans le plan, plus performants que ceux connus à ce jour. Leur principe est de diviser le domaine selon des arbres bidimensionnels (2d-tree, random 2d-tree, adaptive 2d-tree, random adaptive 2d-tree, quadtree, bucket-tree...) et ensuite de fusionner les cellules obtenues suivant deux directions.

L'étude théorique complète est réalisée. La complexité dans le pire des cas reste optimale en $O(n \log n)$. La complexité en moyenne de la phase de triangulation est linéaire pour une distribution quasi-uniforme. La complexité du prétraitement (construction de l'arbre bidimensionnel) est généralement en $O(n \log n)$, voire linéaire si l'on utilise des techniques de hachage dans l'hypothèse d'une distribution uniforme.

Une étude comparative expérimentale approfondie a mis en évidence l'efficacité de ces algorithmes dont le comportement est peu altéré par des distributions non uniformes. Pour cette raison, nous espérons qu'ils deviendront des standards au même titre que *quicksort* pour le tri. D'autre part, cette étude peut permettre au programmeur de faire le bon choix parmi la centaine d'algorithmes existants. A priori, les meilleurs algorithmes sont ceux fondés sur le 2d-tree, le quad-tree ou le bucket-tree :

- ils sont actuellement les plus rapides.
- ils sont peu sensibles à l'imprécision numérique. Seule l'évaluation du signe des déterminants peut présenter quelques risques, aisément levés à l'aide de méthodes adéquates citées dans le chapitre 8.
- ils sont peu perturbés par l'hétérogénéité de la distribution. D'après ce critère, le classement place en première position l'algorithme fondé sur le quadtree, puis celui basé sur le 2d-tree et enfin celui dirigé par le bucket-tree.
- quelques centaines de lignes suffisent pour les coder parce que nous avons pu

généraliser la fonction de fusion de Lee et Schachter [131] à des fusions bidirectionnelles avec pratiquement le même nombre d'instructions.

Dans ces triangulations, il est possible d'insérer des contraintes a posteriori. Mais, il serait fort intéressant de pouvoir calculer directement une triangulation contrainte fondée sur un arbre bidimensionnel. Dans ce dessein, nous avons commencé à généraliser l'algorithme de Volino et Moreau [152], inspiré de celui de Chew [50] dans lequel les fusions ne se font que selon une seule direction. Cette étude se poursuit actuellement.

Enfin, des calculs probabilistes en dimension quelconque sont développés (probabilité qu'un site soit inachevé, espérance du nombre de sites inachevés dans un hyperrectangle) dans l'hypothèse d'une distribution quasi-uniforme. On en déduit que la complexité en moyenne – en termes de sites inachevés – du processus de fusion multidimensionnelle est linéaire. Un prolongement à cette étude pourrait être la construction de la triangulation de Delaunay en dimension quelconque à l'aide d'un algorithme de ce type.

La seconde partie propose de nouveaux algorithmes de localisation basés sur la randomisation à partir d'un arbre binaire équilibré dynamique de type AVL. Des comparaisons expérimentales sont effectuées entre ces différents algorithmes, ainsi qu'avec l'algorithme optimal de Kirkpatrick, ce qui prouve leur grande efficacité en pratique. Jusqu'à au moins 12 millions de sites, le nouvel algorithme *Oversampled-Binary-Search-and-Walk* est plus rapide que celui de Kirkpatrick ! Une analyse sommaire de leur complexité est faite. L'analyse mathématique rigoureuse de la complexité en moyenne de ces algorithmes, qui est beaucoup plus compliquée, est en train d'être réalisée. L'algorithme de localisation basé sur la randomisation à partir d'un AVL est utilisé pour la construction "*on-line*" d'une triangulation incrémentale de Delaunay. L'étude expérimentale montre que cette méthode est parmi les plus performantes des méthodes *on-line* connues à ce jour.

ANNEXE A

A.1 Comparaison des temps de triangulation

Nous comparons ici le temps de triangulation des différents algorithmes sur diverses distributions. Il comprend les temps de prétraitement et de triangulation (le temps de lecture des sites n'est pas compris). La machine utilisée est un super-calculateur CONVEX C3, nettement plus lent qu'une station de travail ordinaire, mais qui offre une grande fiabilité et dispose d'une mémoire vive de 2 GO (il n'y a pas de swap de mémoire).

Les algorithmes (voir paragraphe 7.2)

bal	balayage d'après l'algorithme de Seidel (D. Hatch).
Dt	arbre de Delaunay (INRIA).
LS	Lee et Schachter.
Dw	Dwyer.
bt	bucket-tree à partir d'une grille régulière.
swap	échange des arêtes.
2d	2d-tree.
r2d	random 2d-tree.
a2d	adaptive 2d-tree.
ar2d	adaptive random 2d-tree.
quad	quadtree.

Les distributions (voir paragraphe 7.3)

ball	distribution uniforme dans un disque.
corn	distribution au voisinage des coins d'un carré.
diam	distribution au voisinage d'un segment oblique.
rect	distribution au voisinage d'un segment horizontal.
cross	distribution au voisinage d'une croix.
arc	distribution au voisinage d'un arc de cercle.
ann	distribution dans un anneau.
norm	distribution suivant la loi normale.

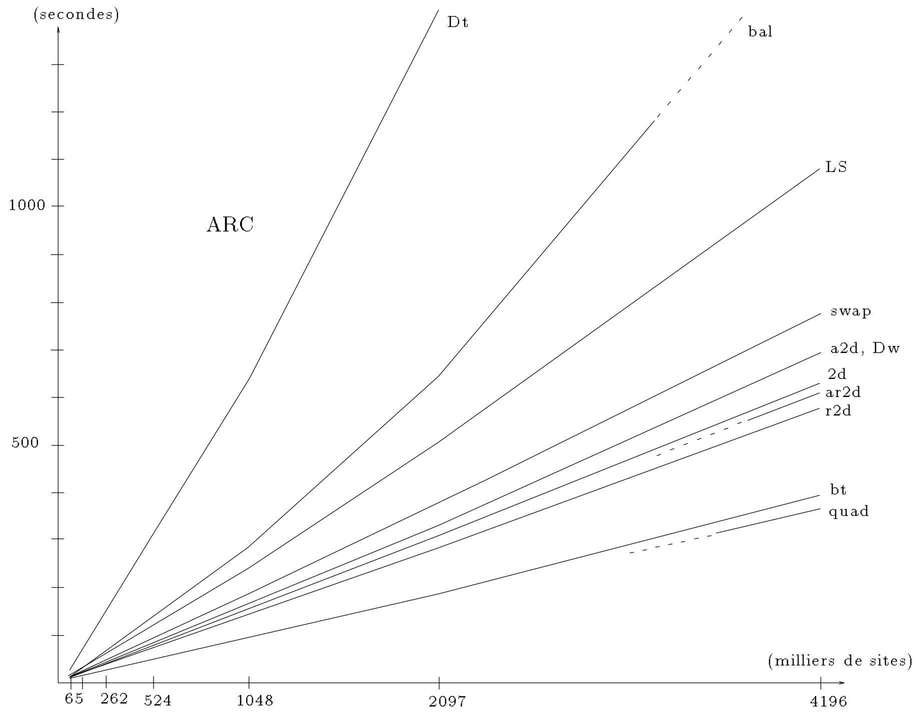


Figure A.1: *Triangulation de Delaunay : sites au voisinage d'un arc de cercle.*

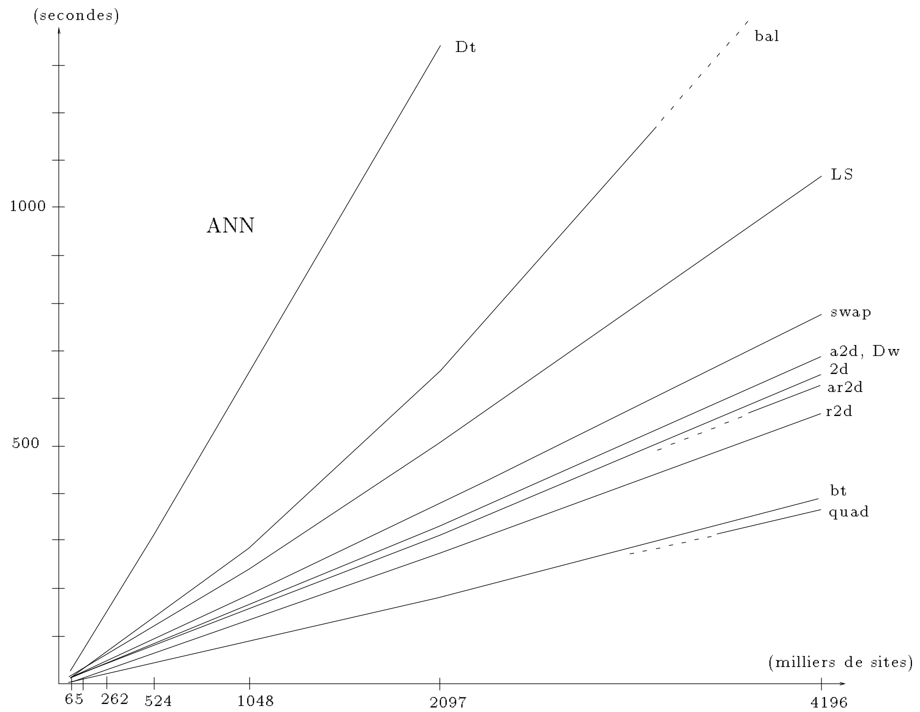


Figure A.2: *Triangulation de Delaunay : sites dans un anneau.*

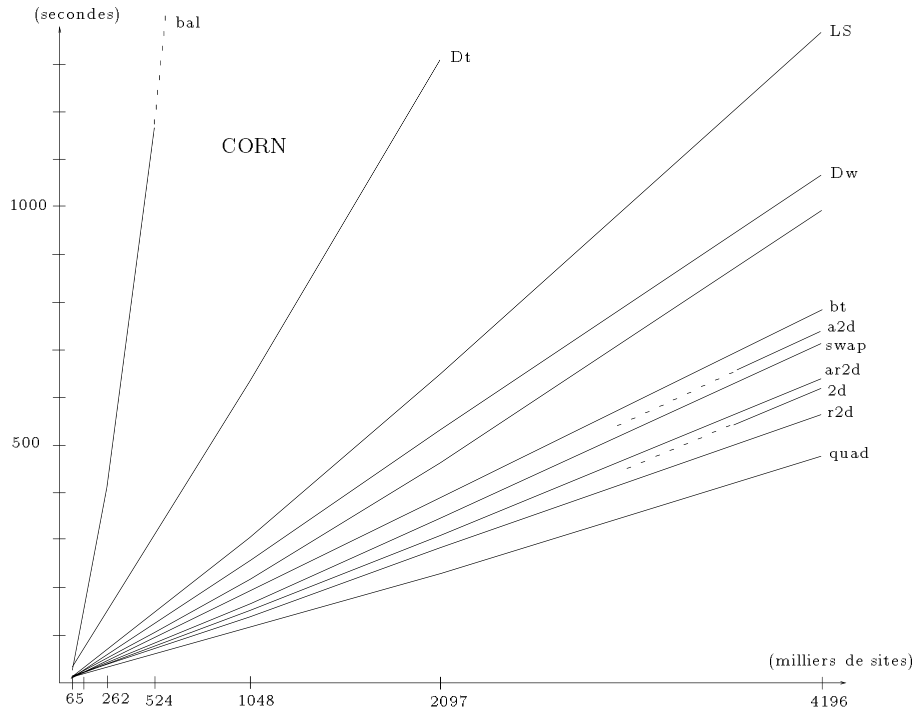


Figure A.3: *Triangulation de Delaunay : sites répartis dans les coins.*

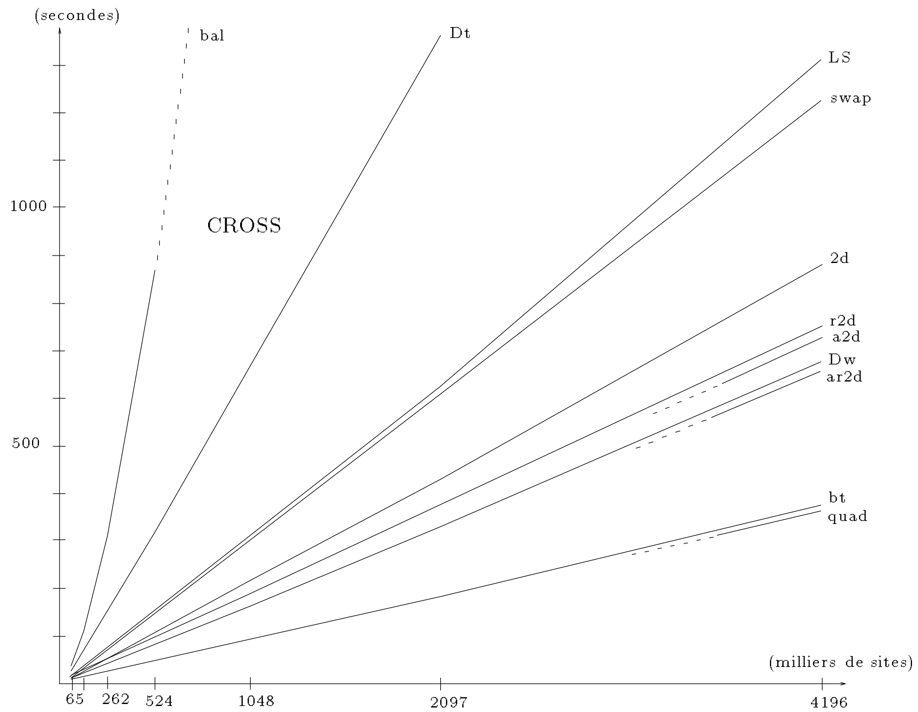


Figure A.4: *Triangulation de Delaunay : sites au voisinage d'une croix.*

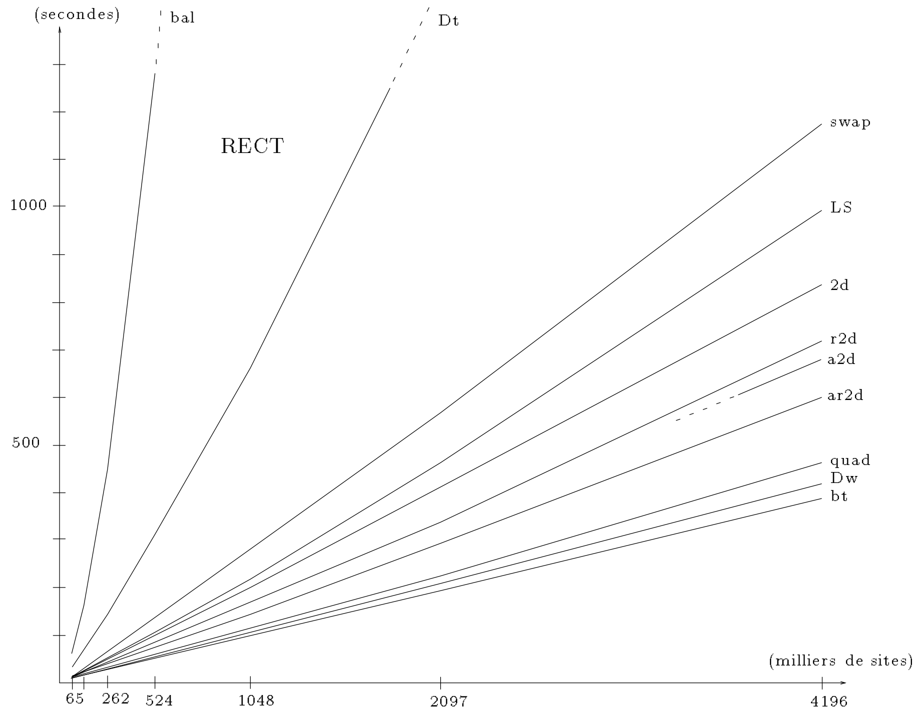


Figure A.5: *Triangulation de Delaunay : sites au voisinage d'un segment horizontal.*

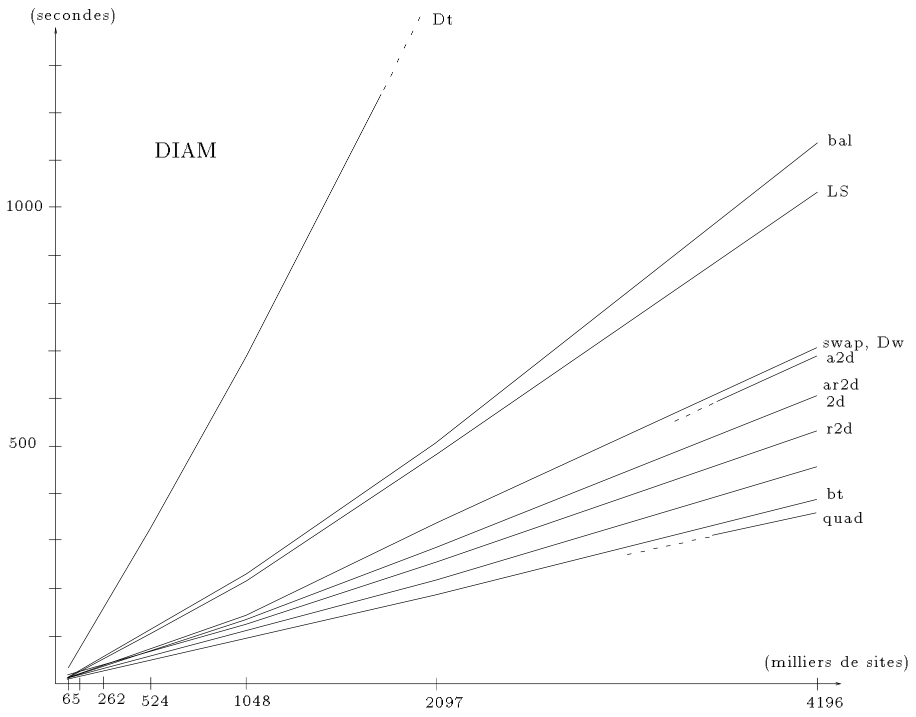


Figure A.6: *Triangulation de Delaunay : sites au voisinage d'un segment oblique.*

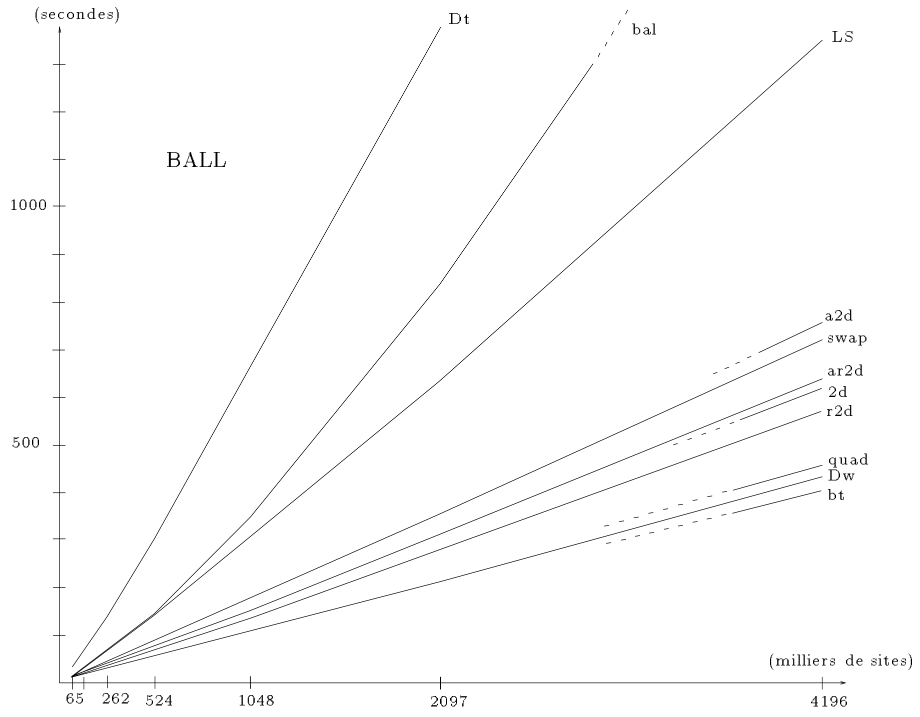


Figure A.7: *Triangulation de Delaunay : sites répartis dans un disque.*

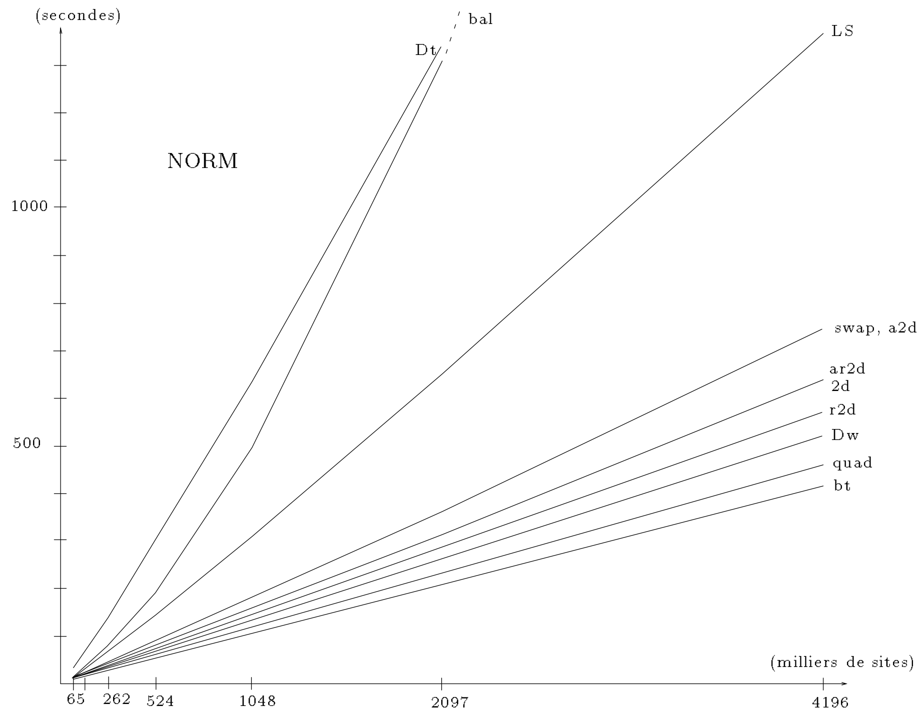


Figure A.8: *Triangulation de Delaunay avec une distribution selon la loi normale.*

ANNEXE B

B.1 Comparaison du nombre total d'arêtes créées

Nous comparons ici le nombre total $\mathcal{A}$ d'arêtes créées par les différents algorithmes sur diverses distributions. En effet, plus ce nombre est faible, plus l'algorithme est efficace. Le programme perd ainsi moins de temps à créer et détruire des arêtes. Ce nombre total d'arêtes créées est proportionnel au temps de triangulation (prétraitement exclu).

Les algorithmes (voir paragraphe 7.2)

LS	Lee et Schachter.
Dw	Dwyer.
bt	bucket-tree à partir d'une grille régulière.
2d	2d-tree.
r2d	random 2d-tree.
a2d	adaptive 2d-tree.
ar2d	adaptive random 2d-tree.
quad	quadtree.

Les distributions (voir paragraphe 7.3)

ball	distribution uniforme dans un disque.
corn	distribution au voisinage des coins d'un carré.
diam	distribution au voisinage d'un segment oblique.
rect	distribution au voisinage d'un segment horizontal.
cross	distribution au voisinage d'une croix.
arc	distribution au voisinage d'un arc de cercle.
ann	distribution dans un anneau.
norm	distribution suivant la loi normale.

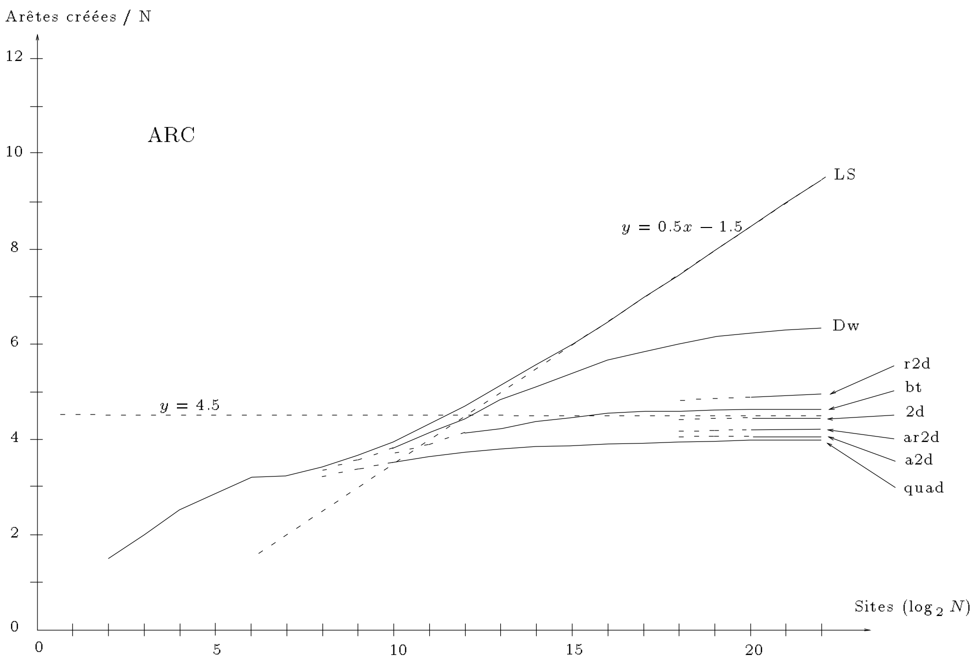


Figure B.1: *Nombre d'arêtes créées : sites au voisinage d'un arc de cercle.*

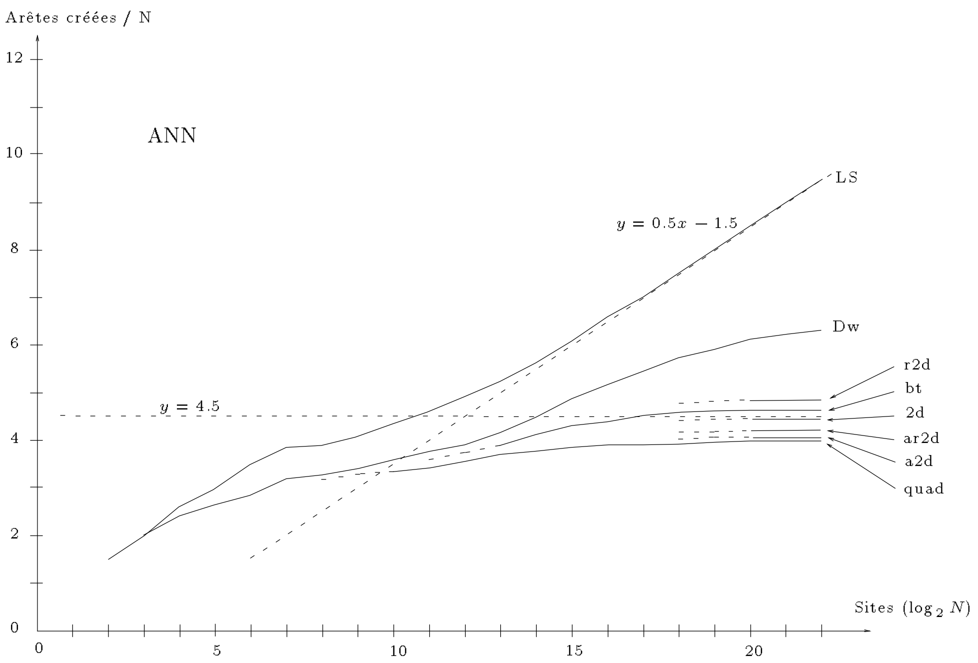


Figure B.2: *Nombre d'arêtes créées : sites au voisinage d'un anneau.*

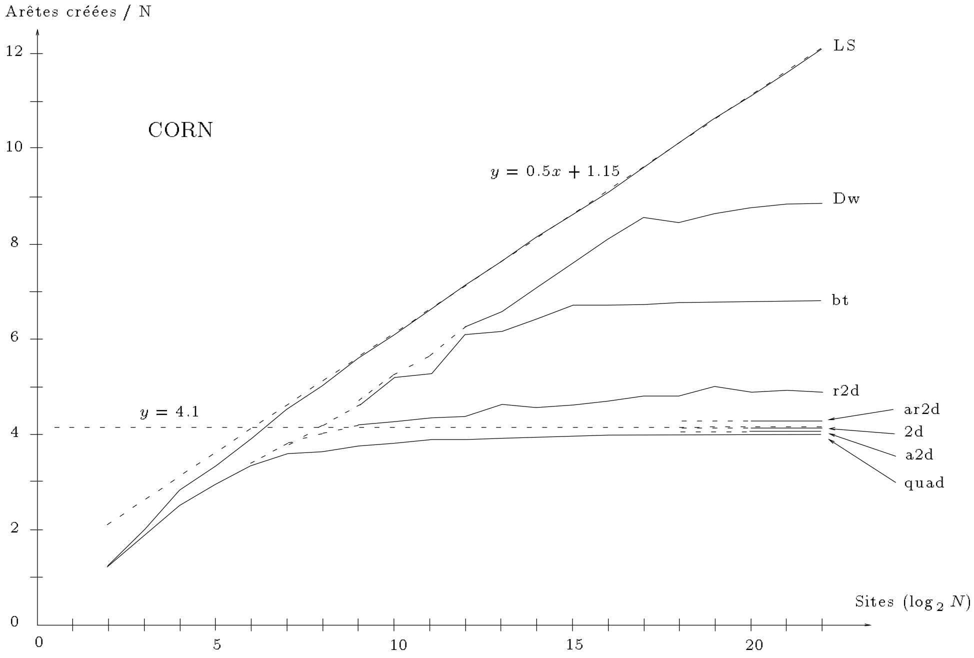


Figure B.3: *Nombre d'arêtes créées : sites répartis dans les coins.*

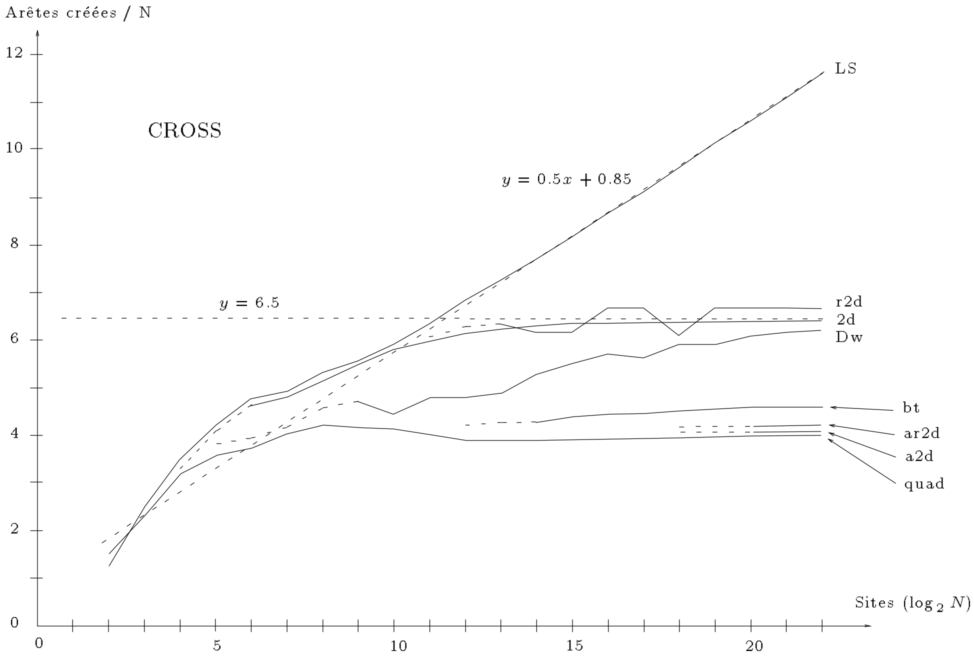


Figure B.4: *Nombre d'arêtes créées : sites au voisinage d'une croix.*

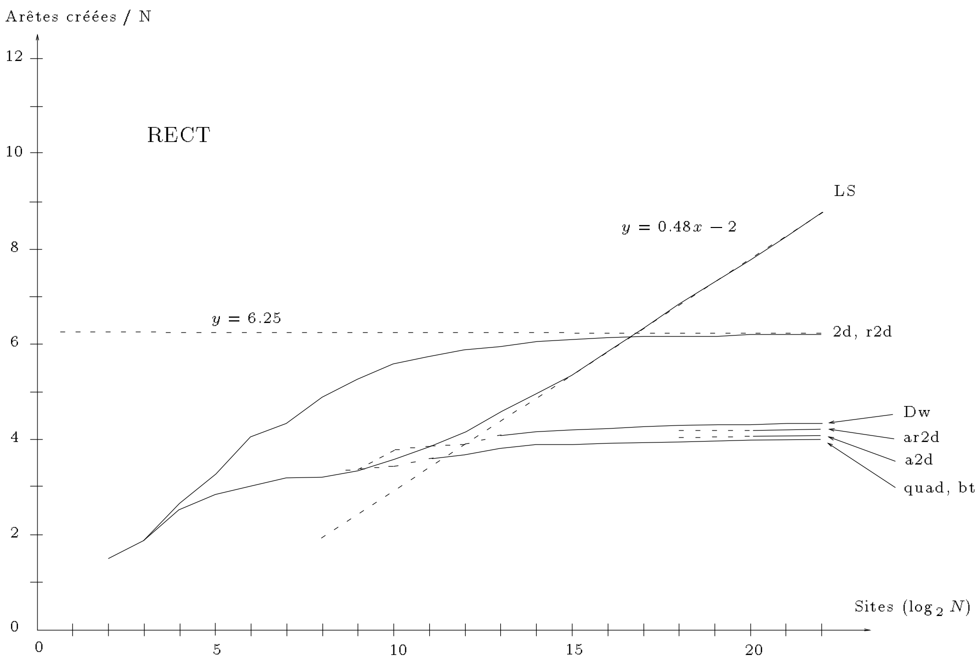


Figure B.5: *Nombre d'arêtes créées : sites au voisinage d'un segment horizontal.*

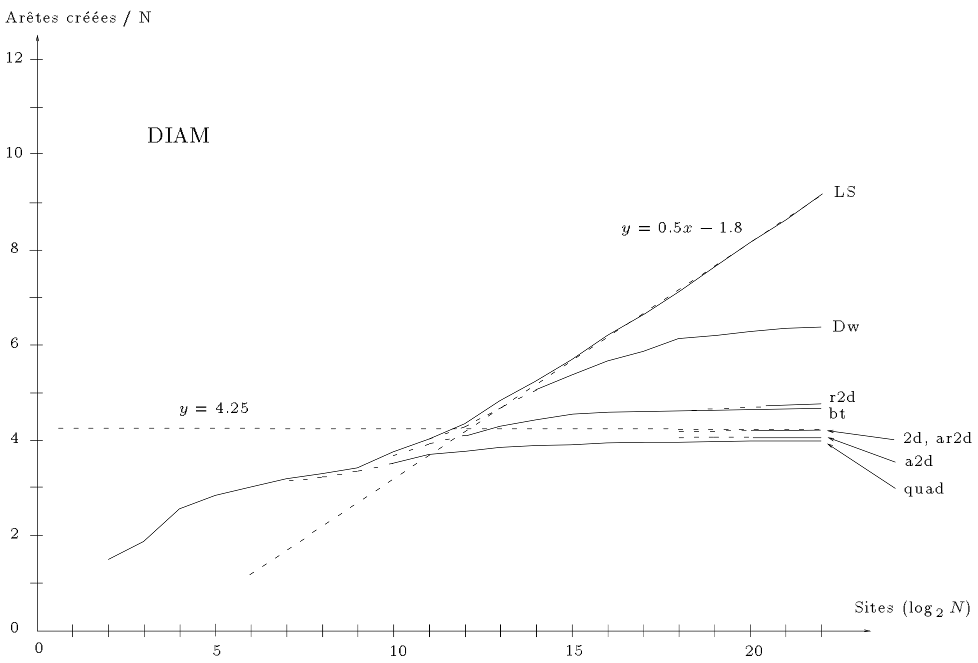


Figure B.6: *Nombre d'arêtes créées : sites au voisinage d'un segment oblique.*

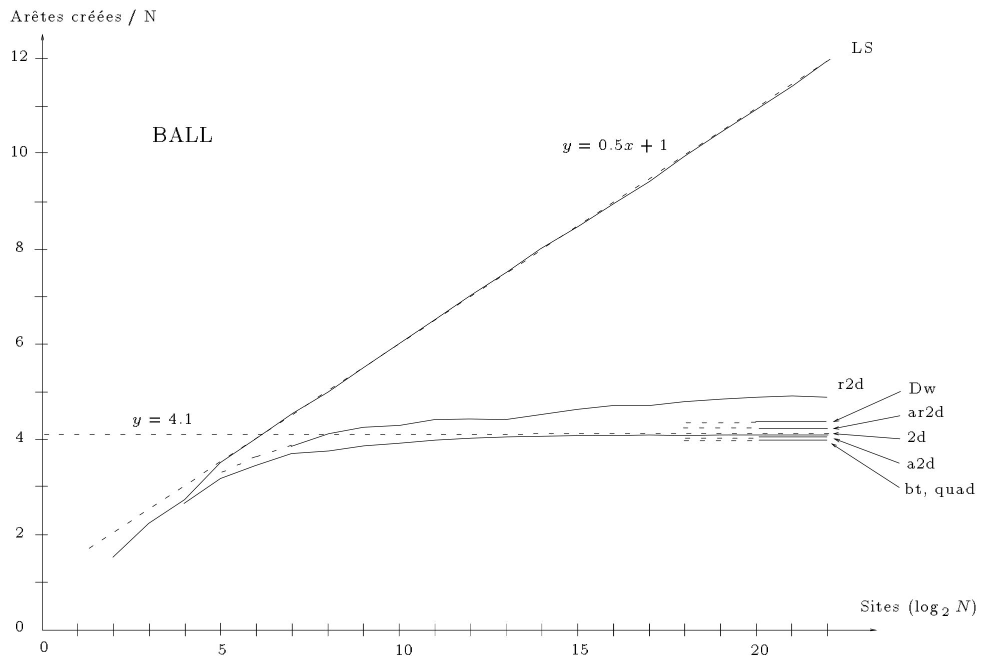


Figure B.7: *Nombre d'arêtes créées : sites répartis dans un disque.*

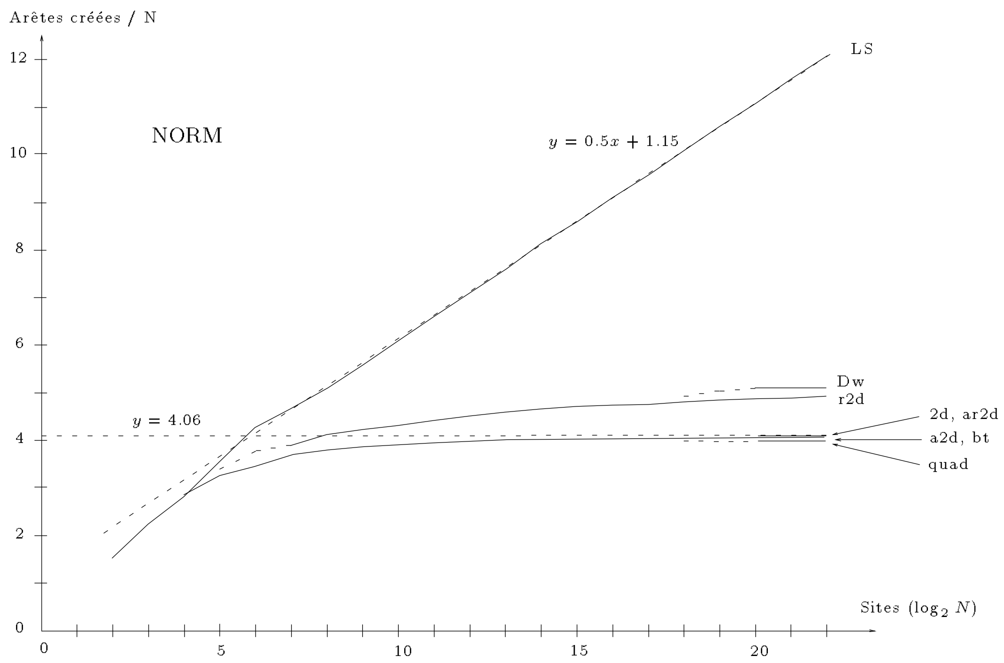


Figure B.8: *Nombre d'arêtes créées avec une distribution selon la loi normale.*

ANNEXE C

C.1 Un algorithme à base d'échange d'arêtes

Cet algorithme **swap** comprend deux phases :

A construction d'une triangulation T(S) : pour cela, on divise l'ensemble des sites à l'aide d'un 2d-tree et l'on réalise des fusions bidirectionnelles. Mais, lors de chaque fusion, on ne cherche pas à construire une triangulation de Delaunay, on construit juste une triangulation en reliant les points mutuellement visibles de telle sorte que la direction du segment qui les relie soit proche de la direction de fusion des deux sous-triangulations. Par exemple, pour la fusion horizontale de $T(S_1)$ et de $T(S_2)$, un point $M_1(x_1, y_1)$ de l'enveloppe convexe de $T(S_1)$ sera relié au point $M_2(x_2, y_2)$ de l'enveloppe convexe de $T(S_2)$ si et seulement si M_1 et M_2 sont mutuellement visibles, et M_2 a l'ordonnée la plus petite parmi les points de $T(S_2)$ au-dessus de M_1 ou alors M_2 a l'ordonnée la plus grande parmi les points de $T(S_2)$ en-dessous de M_1.

B transformation de T(S) en triangulation de Delaunay DT(S) : une fois que la triangulation $T(S)$ est obtenue, on construit la triangulation de Delaunay $DT(S)$ en *échangeant des arêtes* avec la méthode du paragraphe 1.4.1.

Les distributions (voir paragraphe 7.3)

unif	distribution uniforme dans un carré.
ball	distribution uniforme dans un disque.
corn	distribution au voisinage des coins d'un carré.
diam	distribution au voisinage d'un segment oblique.
rect	distribution au voisinage d'un segment horizontal.
cross	distribution au voisinage d'une croix.
arc	distribution au voisinage d'un arc de cercle.
ann	distribution dans un anneau.
norm	distribution suivant la loi normale.
clus	distribution avec "clusters".

Nsites	unif	clus	arc	ann	corn	cross	rect	diam	ball	norm
4	0	0	0	0	0	0.25	0	0	0	0
8	0.37	0.25	0.62	0.75	0.25	0.75	0.25	0.50	0.25	0.25
16	0.37	0.62	0.69	0.62	0.56	1.06	0.81	0.56	0.31	0.68
32	0.84	0.62	1.03	0.84	0.50	2.12	1.37	0.53	0.87	0.93
64	1.08	1	1.20	1.17	0.75	3.01	1.86	0.53	1.04	1.42
128	1.25	1.55	1.05	1.20	0.87	3.41	2.44	0.76	1.43	1.50
256	1.28	1.91	1.07	1.13	1.11	3.76	3.21	0.68	1.35	1.68
512	1.42	1.93	1.23	1.18	1.22	4.23	3.74	0.79	1.50	1.83
1 024	1.52	1.93	1.38	1.19	1.36	4.81	4.22	1.01	1.55	1.82
2 048	1.60	2.23	1.63	1.24	1.48	5.01	4.45	1.21	1.65	1.87
4 096	1.63	2.24	1.81	1.41	1.55	5.29	4.77	1.43	1.64	1.91
8 192	1.65	2.41	1.95	1.66	1.62	5.41	4.79	1.57	1.71	1.95
16 384	1.65	2.54	2.11	1.82	1.61	5.48	5.04	1.70	1.69	1.92
32 768	1.72	2.57	2.18	1.99	1.68	5.63	4.99	1.79	1.76	1.98
65 536	1.69	2.58	2.23	2.10	1.66	5.63	5.18	1.85	1.73	1.95
131 072	1.73	2.74	2.27	2.18	1.71	5.68	5.09	1.90	1.77	1.98
262 144	1.69	2.77	2.30	2.24	1.69	5.67	5.25	1.93	1.75	1.96
524 288	1.74	2.69	2.32	2.28	1.73	5.68	5.14	1.94	1.78	1.99
1 048 576	1.70	2.73	2.33	2.30	1.70	5.70	5.27	1.95	1.75	1.96
2 097 152	1.74	2.70	2.34	2.32	1.74	5.70	5.16	1.96	1.78	1.99
4 194 304	1.71	2.65	2.35	2.33	1.70	5.69	5.29	1.97	1.76	1.96

Tableau C.1: *comparaison du nombre d'arêtes échangées par site.*

Pour chaque distribution, on a calculé le nombre d'échanges d'arêtes par site, qui semble devenir à peu près constant au fur et à mesure que la taille du semis augmente. Ce nombre est plus faible pour une distribution uniforme que pour une distribution très irrégulière.

Ces résultats expérimentaux montrent la linéarité du nombre de swaps en fonction du nombre de sites, et donc la linéarité de la phase B de l'algorithme. La preuve théorique est un problème ouvert. On a constaté expérimentalement la linéarité de la phase A de l'algorithme (le travail dans la phase A est de toute facon moins important que dans la phase de triangulation de l'algorithme **2d** dirigé par un 2d-tree, qui est linéaire en moyenne sur une distribution quasi-uniforme). Une fois que le 2d-tree est construit, l'algorithme **swap** est linéaire, d'après les résultats expérimentaux.

ANNEXE D

D.1 Espace mémoire utilisé par le quadtree

Nous avons mesuré expérimentalement l'espace mémoire utilisé par le quadtree, en fonction des différentes distributions et de la taille du semis.

Les distributions (voir paragraphe 7.3)

unif	distribution uniforme dans un carré.
ball	distribution uniforme dans un disque.
corn	distribution au voisinage des coins d'un carré.
diam	distribution au voisinage d'un segment oblique.
rect	distribution au voisinage d'un segment horizontal.
cross	distribution au voisinage d'une croix.
arc	distribution au voisinage d'un arc de cercle.
ann	distribution dans un anneau.
norm	distribution suivant la loi normale.
clus	distribution avec "clusters".

Dans le tableau ci-dessous, on trouve pour chaque type de distribution le nombre de cellules par site, une cellule étant un entier de taille 4 octets. Le paragraphe 7.4.3 contient l'analyse de ces résultats.

Nsites	unif	clus	arc	ann	corn	cross	rect	diam	ball	norm
4	1.25	1.25	4.25	1.25	0.25	1.25	6.25	2.25	1.25	0.25
8	2.62	3.12	5.12	3.62	13.62	4.12	5.62	3.62	2.12	2.12
16	1.81	4.06	3.81	3.56	7.81	4.81	5.56	4.81	2.31	1.81
32	2.40	5.16	4.78	4.65	5.28	4.78	5.28	4.28	2.28	2.78
64	2.76	4.58	4.51	5.07	4.58	4.95	5.07	3.82	2.89	2.89
128	2.63	4.07	4.38	5.32	3.88	4.63	5.32	3.54	3.16	3.04
256	3.01	3.99	4.22	4.97	3.44	4.88	4.66	4.05	3.11	3.03
512	3.01	3.53	3.82	4.42	3.12	4.56	4.41	3.68	3.05	2.99
1 024	2.81	3.26	3.39	4.09	2.98	4.22	3.61	3.52	2.84	2.87
2 048	2.86	3.23	3.36	3.82	2.92	3.71	3.27	3.40	2.83	2.91
4 096	2.87	3.16	3.18	3.47	2.96	3.37	3.22	3.21	2.89	2.84
8 192	2.90	3.04	3.06	3.33	2.98	3.24	3.22	3.01	2.94	2.89
16 384	2.90	2.98	3.01	3.17	2.95	3.17	3.08	2.97	2.89	2.89
32 768	2.89	2.96	2.97	3.06	2.90	3.08	3.00	2.95	2.89	2.88
65 536	2.88	2.92	2.93	3.00	2.90	3.00	2.93	2.91	2.89	2.89
131 072	2.89	2.92	2.93	2.96	2.87	2.93	2.91	2.92	2.90	2.89
262 144	2.88	2.90	2.91	2.94	2.88	2.93	2.91	2.90	2.88	2.89
524 288	2.89	2.90	2.90	2.92	2.88	2.93	2.91	2.89	2.89	2.88
1 048 576	2.88	2.89	2.89	2.90	2.89	2.93	2.90	2.89	2.89	2.88
2 097 152	2.88	2.89	2.89	2.90	2.88	2.91	2.90	2.89	2.89	2.88
4 194 304	2.88	2.89	2.88	2.90	2.89	2.91	2.90	2.89	2.88	2.89

Tableau C.1: *espace mémoire moyen utilisé par le quadtree pour chaque site (en cellules).*

ANNEXE E

E.1 Temps de construction des 2d-trees

Le tableau ci-dessous présente la comparaison des temps (en secondes) de construction de différents arbres bidimensionnels pour un semis de taille 2^{23} (environ 4 millions de sites), en fonction du type de la distribution. La machine utilisée est le super-calculateur CONVEX C3. Il est nettement plus lent qu'une station de travail ordinaire, ce qui explique les temps de calcul relativement importants.

Les arbres bidimensionnels

2d 2d-tree.
r2d random 2d-tree.
a2d adaptive 2d-tree.
ar2d adaptive random 2d-tree.
quad quadtree.

	unif	**clus**	**arc**	**ann**	**corn**	**cross**	**rect**	**diam**	**ball**	**norm**
heapsort	192	191	191	190	191	191	190	190	194	192
2d	265	268	246	259	269	268	261	232	265	270
r2d	110	109	115	109	111	110	112	95	111	111
a2d	371	341	328	336	372	356	314	322	380	377
ar2d	246	233	223	245	244	259	215	214	243	244
quad	94	126	108	107	119	107	109	107	95	99

Tableau C.1: *temps de construction des arbres bidimensionnels (en secondes).*

La seconde ligne du tableau contient les temps de calcul d'un tri par *heapsort* qui servent de référence.

ANNEXE F

F.1 Article [134]

Amélioration de la complexité en moyenne de l'algorithme de Lee et Schachter par division et fusion bi-directionnelles

Christophe LEMAIRE[1,2]
Jean-Michel MOREAU[2]

[1] SETRA, *46 avenue Aristide Briand, BP 100*
F 92 223 Bagneux Cedex.
[2] EMSE, *158 Cours Fauriel,*
F 42023 Saint-Étienne

RÉSUMÉ. *Nous avons cherché à améliorer les performances en moyenne de l'algorithme optimal (pour le pire des cas) de Lee et Schachter [LS 80] qui calcule la triangulation de Delaunay d'un ensemble de N points du plan avec une méthode du type "Diviser et fusionner". Pour cela, nous utilisons le découpage induit par un $2d$-arbre, qui peut être construit à partir d'un tri selon deux directions. Si la distribution des sites est quasi-uniforme dans un carré unité, nous montrons alors que la phase de triangulation (après la création du $2d$-arbre) est linéaire en moyenne alors que dans l'algorithme de Lee et Schachter, elle est en $\Theta(N \log N)$, même en moyenne [OIM 84].*
En pratique, le temps de triangulation est divisé par 2 ou 3. Sur une Silicon Graphics Indy (200 MHz), on triangule 200 000 points en 7 secondes après une phase de tri de 3 secondes. Cette méthode est plus rapide que les méthodes par balayage pour le même problème.

ABSTRACT. *We have tried to improve the average running time of the (worst case) optimal algorithm by Lee and Schachter [LS 80] that computes the Delaunay triangulation of N points in the plane, using a "Divide-and-Conquer" technique. To that effect, we use the division induced by a $2d$-tree that may be constructed from a two-directional sort. If the distribution of sites is quasi-uniform in a unit square, we show that the triangulation phase (after the construction of the $2d$-tree) takes linear average time, whereas it takes $\Theta(N \log N)$ time in Lee and Schachter's case, even in the average ([OIM 84]).*
In practice, the triangulation time is divided by 2 or 3. On a Silicon Graphics Indy (200 MHz), 200,000 sites are triangulated in 7 seconds, after a sorting phase of 3 seconds. This method runs faster than sweep-line techniques for the same problem.

MOTS-CLÉS : *triangulation de Delaunay, diagramme de Voronoi, tri, $2d$-arbre, complexité dans le pire des cas, complexité en moyenne.*

KEYWORDS: *Delaunay triangulation, Voronoi diagram, sort, $2d$-tree, worst-case running time, average running time.*

1. Introduction

Soient $\mathbb{E}^2$ le plan muni de la distance euclidienne d, et $S = \{M_1, M_2, \ldots M_N\}$ un ensemble de N points de $\mathbb{E}^2$, appelés *sites*. On associe à chaque site M_i la région $V(M_i)$ de $\mathbb{E}^2$ des points du plan plus proches de M_i que de n'importe quel autre site :

$$V(M_i) = \{P \in \mathbb{E}^2 \mid d(P, M_i) \leq d(P, M_j), \forall j \in [1, n] - \{i\}\},$$

Pour tout $i \in [1, n]$, $V(M_i)$, qui est appelée la *région de Voronoi de centre* M_i, est l'intersection d'un nombre fini de demi-plans limités par les médiatrices des segments $[M_i M_j]$, $j \in [1, n] - \{i\}$. C'est donc un polygone convexe, éventuellement non borné, appelé *polygone de Voronoi de centre* M_i.

L'union des $V(M_i)$ constitue une subdivision de $\mathbb{E}^2$, appelée le *diagramme de Voronoi* de S, noté $DV(S)$. L'intersection de deux polygones de Voronoi quelconques de centre respectif M_i et M_j est soit vide (la médiatrice du segment $M_i M_j$ n'apparaît pas dans le diagramme), soit réduite à un point (M_i et M_j sont situés sur le même cercle d'intérieur vide qu'au moins deux autres sites de S), soit enfin égale à une section non dégénérée de la médiatrice de $M_i M_j$. Dans ce dernier cas, on dit que les sites M_i et M_j sont *voisins* au sens de Voronoi.

Le *dual* (faible) de $DV(S)$ est un graphe *planaire* $T = (S, A, F)$, appelé *diagramme de Delaunay*, qui forme une partition de l'enveloppe convexe de S, et dont l'ensemble des arêtes est

$$A = \{(M_i, M_j) \in S^2, i \neq j \mid M_i, M_j \text{voisins au sens de Voronoi}\},$$

et dont les faces F sont les polygones constitués par ces arêtes.

Tous les sommets définissant la frontière d'un même polygone de Delaunay sont cocycliques. Le diagramme de Delaunay devient une *triangulation* et est unique si et seulement si toutes ses faces sont des triangles, *i.e.* S ne contient aucun quadruplet de sites situés sur un même cercle d'intérieur vide. Sinon, on peut toujours obtenir "une" triangulation de Delaunay de S en rajoutant arbitrairement les cordes nécessaires dans les faces non-simpliciales de F. Par abus de langage, on parlera toujours de *la* triangulation de Delaunay de l'ensemble S, que l'on notera $TD(S)$.

Les diagrammes de Voronoi et les triangulations de Delaunay sont des structures très puissantes, souvent utilisées pour les problèmes de proximité, la planification de mouvements, le calcul par éléments finis, la représentation de terrains naturels… Pour un survol complet de ces deux structures et de leurs propriétés, se reporter, par exemple, à l'article de F. Aurenhammer [AUR 91].

Une des propriétés remarquables de la triangulation de Delaunay est le *critère du cercle vide* : le cercle circonscrit à un triangle de $TD(S)$ ne contient aucun site de S en son intérieur strict. Ceci constitue un critère de construction efficace, que Lee et Schachter ont exploité en 1980 pour obtenir le premier algorithme optimal de construction de la triangulation de Delaunay de N sites du plan en temps $O(N \log N)$.

En 1984, A. Maus ([MAU 84]) publia la première méthode linéaire en moyenne de construction de la triangulation de Delaunay d'un ensemble de sites du plan, à l'aide d'une grille pour accélérer la localisation du voisin le plus proche vérifiant le critère du cercle vide pour toute arête de Delaunay donnée.

En 1987, R. Dwyer publia une méthode de complexité $O(N \log\log N)$ en moyenne ([DWY 87]) pour le même problème dans le cas d'une distribution uniforme, à l'aide d'une grille régulière. Dans leur article de 1988, [KK 88], J. Katajainen et M. Koppinen introduisirent une variante de l'algorithme standard de Lee et Schachter, s'appuyant encore sur une grille régulière. Leur analyse de complexité en moyenne introduit et exploite la notion de *site inachevé* dans un domaine rectangulaire, qui est reprise dans cet article. Enfin, R. Dwyer publia la première méthode k-dimensionnelle de complexité moyenne linéaire en 1991 ([DWY 91]).

Par opposition, la méthode décrite dans le présent article ne découpe pas le plan à l'aide d'une grille, mais selon une arborescence équilibrée. L'article est construit de la manière suivante : la section 2 rappelle les points saillants de l'algorithme classique dû à Lee et Schachter pour la construction de la triangulation d'un ensemble de sites du plan selon le paradigme *Division-fusion*. La section 3 présente la technique de division et fusion dans deux directions et l'algorithme qui en résulte, dont la section 4 évalue la complexité asymptotique dans le pire des cas et en moyenne pour une distribution quasi-uniforme des sites dans un carré unité. La section 5 présente quelques résultats expérimentaux obtenus pour l'algorithme présenté dans cet article, qui se termine par quelques remarques de conclusion, section 6.

2. Division et fusion uni-directionnelles

Cette section présente le principe sous-tendant l'algorithme original de Lee et Schachter ([LS 80]) pour la construction de la triangulation de N sites dans le plan. Pour simplifier, on fait l'hypothèse que la triangulation de Delaunay est unique (*cf.* Introduction).

2.1. *Division de l'ensemble des sites*

On choisit une direction du plan privilégiée, par exemple verticale. L'ensemble des sites est divisé récursivement en deux parties séparables par une droite verticale [1], et de même taille (à une unité près), jusqu'à ce que les sous-ensembles ne contiennent qu'un seul site (figure 1).

Ce découpage équilibré du plan nécessite la réorganisation, à chaque niveau de récursion, de l'ensemble courant de part et d'autre de la X-médiane, ce qui peut être fait en temps optimal linéaire par l'algorithme de sélection de Blum *et al.* ([BFPRT 73]). Mais il est bien connu qu'à complexité asymptotique globale identique, il est bien plus efficace de pré-trier les sites de S selon le xy-ordre : selon les x croissants, et en cas d'égalité, selon les y croissants. Une fois ce tri opéré, l'accès à la médiane peut être effectué en temps constant.

Ce choix de la direction de division (et fusion) est arbitraire, et l'on serait tout autant fondé à choisir la direction horizontale. En fait, tout ce qui est dit dans cette section et la suivante est valable pour l'une ou l'autre direction.

2.2. *Fusion des sous-ensembles élémentaires*

Les bandes élémentaires adjacentes sont alors fusionnées par paires de telle sorte que la triangulation des points contenus dans chaque bande résultante soit de De-

1. En fait, cette droite peut parfois être légérement oblique pour séparer les points ayant même abscisse.

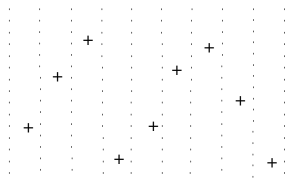


Figure 1. *Division récursive en bandes élémentaires*

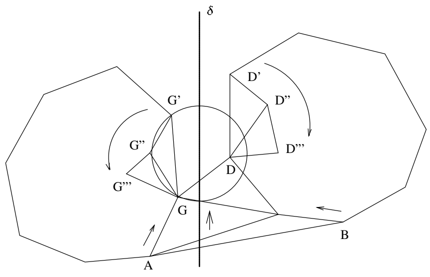


Figure 2. *Fusion de deux triangulations linéairement séparables*

launay. Ce processus est itéré jusqu'à obtention de la triangulation de Delaunay de l'ensemble S total.

Pour fusionner deux triangulations $TD(G)$ et $TD(D)$ linéairement séparables par une droite δ verticale, on part des sites de G et de D qui sont les plus proches de cette droite, et l'on progresse dans le sens horaire (*resp.* direct) sur la frontière de l'enveloppe convexe de G (*resp.* D), jusqu'à trouver les extrémités (A et B sur la figure 2) de l'arête la plus basse de l'enveloppe convexe de $G \cup D$ qui coupe δ. Ensuite, les nouvelles arêtes sont créées dans l'ordre selon lequel elles coupent δ jusqu'à la création de l'arête la plus haute de l'enveloppe convexe de $G \cup D$ coupant δ.

Les arêtes de $TD(G)$ et $TD(D)$ qui coupent les arêtes créées pendant la fusion, sont détruites. En fait, dès qu'une arête GD est créée, on réalise les opérations suivantes :

a. *Nettoyage à gauche autour du site G, dans le sens direct :* tant que G' est situé au-dessus de GD, on élimine l'arête GG' si D est inclus dans l'intérieur de $\bigcirc(G, G', G'')$, puis on passe à GG'', sinon on arrête.

b. *Nettoyage à droite autour du site D, dans le sens horaire :* tant que D' est situé au-dessus de GD, on élimine l'arête DD' si G est inclus dans l'intérieur de $\bigcirc(D, D', D'')$, puis on passe à DD'', sinon on arrête.

Dans le quadrilatère $GG'D'D$ résultant de ces nettoyages, on choisit la diagonale satisfaisant le critère du cercle vide ("au hasard" si $GG'D'D$ est inscriptible dans un cercle), et ce choix induit la création d'une nouvelle arête (GD' ou $G'D$) qui devient la nouvelle GD dans l'itération suivante, et ainsi de suite...

3. Division et fusion bi-directionnelles

Le découpage récursif des N sites selon une direction unique produit des bandes élémentaires très longues et très étroites. La fusion de ces bandes engendre un assez grand nombre de triangles qui n'ont pratiquement aucune chance d'appartenir à la triangulation finale, étant donné leur forme excessivement allongée (partie gauche de la figure 17).

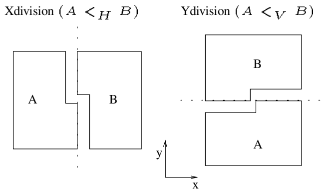


Figure 3. *X- et Y-divisions*

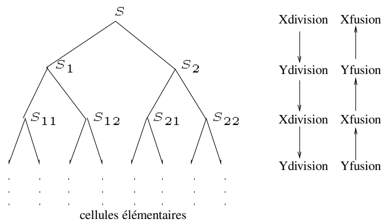


Figure 4. *Arborescence schématisant les divisions et les fusions*

L'algorithme perd beaucoup de temps, au fur et à mesure des fusions entre bandes adjacentes de même direction, à détruire et reconstruire des triangles. Ohya, Iri et Murota ([OIM 84]) ont montré que la complexité de l'étape de fusion est, *même en moyenne*, en $\Omega(N \log N)$. Ceci est fondamentalement dû au principe de découpage par bandes uni-directionnelles.

Pour optimiser l'algorithme précédent, nous proposons de diviser le plan selon un autre processus, qui évite la création de bandes étroites : la division (et la fusion) selon deux directions orthogonales (partie droite de la figure 17).

3.1. *X- et Y-divisions*

Notons M_x et M_y l'abscisse et l'ordonnée d'un point M, et définissons les relations $<_H$ et $<_V$ respectivement par :

$$M <_H N \quad \Leftrightarrow \quad ((M_x < N_x) \vee (M_x = N_x \wedge M_y < N_y)), \quad \text{et}$$
$$M <_V N \quad \Leftrightarrow \quad ((M_y < N_y) \vee (M_y = N_y \wedge M_x > N_x))$$

Soient A et B deux ensembles de points de $\mathbb{E}^2$. On dira que $A <_H B$ (*resp.* $A <_V B$) si et seulement si tous les points de A sont $<_H$ (*resp.* $<_V$) à tous les points de B.

En se référant à la figure 3, on dira que l'on X-divise (*resp.* Y-divise) un ensemble A de points du plan en A_1 et A_2 si et seulement si $A_1 <_H A_2$ (*resp.* $A_1 <_V A_2$), $A_1 \cup A_2 = A$ et si A_1 et A_2 ont le même nombre de points, à une unité près. On remarquera que A_1 ou A_2 peut être vide.

Les sites vont être divisés selon la dynamique d'un $2d$-*arbre* ([BEN 75]) *équilibré*. Un $2d$-arbre est obtenu en divisant alternativement le plan suivant les directions horizontales et verticales. Plus précisément, on commence par X-diviser l'ensemble S en S_1 et S_2. Ensuite, on Y-divise S_1 (*resp.* S_2) en S_{11} et S_{12} (*resp.* S_{21} et S_{22}). Ensuite, les parties S_{11}, S_{12}, S_{21} et S_{22} sont X-divisées. On itère ce processus en alternant les X-divisions et les Y-divisions jusqu'à l'obtention de cellules élémentaires contenant un seul site (figure 4).

L'équilibrage de l'arbre de division est garanti par le "partitionnement" du sous-ensemble relatif au nœud courant autour de la X- ou Y-médiane. Comme pour l'algorithme original, il existe plusieurs façons de procéder pour effectuer ces partitions : on

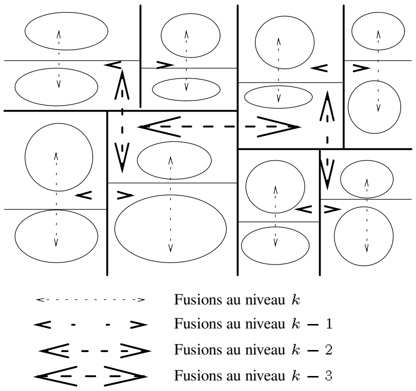


Figure 5. *Processus de fusion*

peut utiliser l'algorithme linéaire déterministe de sélection de la médiane de Blum *et al.* (*cf.* [BFPRT 73]), ou bien une méthode dérivée de *Quicksort*, randomisée ([CLR 94]) ou non ([HOU 87] [2]), ou encore tout simplement effectuer un tri de prétraitement dans les deux directions. Quelle que soit la méthode choisie, on supposera sa complexité théorique globale en $O(N \log N)$.

3.2. X*- et* Y*-fusions*

Les cellules élémentaires sont ensuite fusionnées par paires dans l'ordre inverse de leur création. Les sous-ensembles obtenus par X-division (*resp.* Y-division) sont fusionnés par X-fusion (*resp.* Y-fusion), (*cf.* figure 5).

Comme nous l'avons déjà signalé, cela est possible car la procédure de fusion de Lee et Schachter est très générale et fonctionne de la même façon, quelle que soit la direction de la droite de séparation entre les deux triangulations à fusionner, à condition de choisir des structures de données bien adaptées.

Structure de données pour la division-fusion bi-directionnelle

Les procédures de fusion (et de division) présentent un assez grand degré de symétrie, nécessitant des structures de données flexibles et symétriques. La structure de "carte planaire" ([COR 75]), proche de celle d'arête ailée ([BAU 75]), aussi compacte et légèrement plus souple, a été choisie pour représenter les graphes : un sommet est représenté par un couple de coordonnées ; un "brin" (de la carte planaire) comprend un pointeur sur son sommet terminal, et (un tableau de) deux pointeurs sur les brins suivant et précédent (selon l'ordre polaire) autour de son propre sommet origine (implicite) (figure 6). Les brins sont stockés dans un tableau, de sorte qu'un brin et son inverse aient des positions symétriques par rapport au milieu du tableau, ce qui dispense de tout lien explicite de dualité.

2. C'est la méthode que nous avons choisie pour son efficacité (en particulier dans le cas de distributions quasi-uniformes), sa simplicité, et qui correspond aux temps donnés dans la section 5.

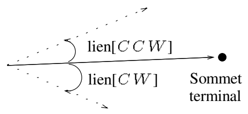


Figure 6. *Un* brin*, structure sous-tendant la "carte planaire"*

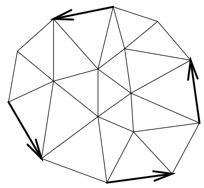


Figure 7. *Les quatre brins extrêmes d'une triangulation*

Pour fusionner deux triangulations linéairement séparables, on a besoin du site de chaque sous-triangulation le plus proche de la droite de séparation, que celle-ci soit verticale ou horizontale. Pour accéder à ces sommets en temps constant, on garde en mémoire (sous forme de matrice 2×2) les quatre brins extrêmes de chaque sous-triangulation, *i.e.* l'arc orienté d'enveloppe convexe issu de chaque sommet extremum pour les relations d'ordre $<_V$ et $<_H$ (*cf.* figure 7).

3.3. *L'algorithme*

On suppose que les sites ont été rangés dans un tableau sans point double lors d'une Étape 0 ; g et d représentent les extrémités du tableau courant dans chaque fonction, et l'on pose $\mu = \lceil (g + d)/2 \rceil$. Enfin, $\varepsilon \in \{0, 1\}$ représente la "direction" courante (0 pour verticale, 1 pour horizontale).

Etape 1 : Tri sur place du tableau de sites selon deux directions
  Fonction XY-Division (g, d, ε)
    Si $((d - g) > 0)$ **alors**
      XY-Partition(g, d, ε)
      XY-Division$(g, \mu - 1, 1 - \varepsilon)$
      XY-Division$(\mu, d, 1 - \varepsilon)$
Etape 2 : Triangulation
  Fonction Delaunay(g, d)
    XY-Coupe$(g, d, 0)$
  Fonction XY-Coupe(g, d, ε)
    Si $((d - g) > 0)$ **alors**
      XY-Coupe$(g, \mu - 1, 1 - \varepsilon)$
      XY-Coupe $(\mu, d, 1 - \varepsilon)$
      XY-Fusion(g, d, ε)

4. Analyse de complexité

On s'intéresse ici à la complexité de la phase de triangulation, une fois le tri selon deux directions des N sites effectué. Ce tri peut être réalisé en $\Theta(N \log N)$, comme nous l'avons déjà fait remarquer.

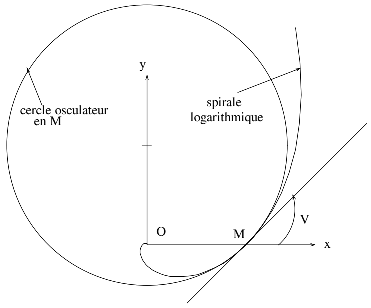


Figure 8. *Propriétés de la spirale logarithmique*

4.1. *Complexité dans le pire des cas*

Nous allons montrer que la complexité dans le pire des cas est en $\Omega(N \log N)$. Pour cela, nous allons étudier la triangulation de Delaunay de N sites répartis sur une portion de spirale logarithmique.

La spirale logarithmique d'équation en coordonnées polaires $\rho = ae^{m\theta}$ a de nombreuses propriétés (figure 8) :

1. La tangente en M fait un angle V constant avec la droite OM $(\tan V = \frac{1}{m})$.
2. Le cercle passant part M et centré au centre de courbure en M contient toute la section de la spirale situé *avant* (au sens de l'abscisse curviligne) M ; toute la section de la spirale *après* M est à l'extérieur de ce cercle.
3. Le cercle circonscrit à trois sites M_i, M_j et M_k $(i < j < k)$ situés sur la spirale contient toute la partie de la spirale *avant* M_k ; toute la partie de la spirale *après* M_k est à l'extérieur de ce cercle.
4. Il en résulte que construire la triangulation de Delaunay d'un ensemble de sites M_i situés sur une portion de spirale consiste à relier le site M_1 à tous les autres sites M_j $(j \neq 1)$, puis à relier chaque site M_i à son suivant M_{i+1} . Ainsi, si l'on ajoute un site sur la spirale avant M_1, il n'y a plus un seul triangle valide (figure 9).

Considérons par exemple la spirale d'équation $\rho = e^\theta$.

$$\left\{ \begin{array}{l} x(\theta) = e^\theta \cos\theta \\ y(\theta) = e^\theta \sin\theta \end{array} \right. \Rightarrow \left\{ \begin{array}{l} x\prime(\theta) = \sqrt{2}e^\theta \cos(\theta + \frac{\pi}{4}) \\ y\prime(\theta) = \sqrt{2}e^\theta \sin(\theta + \frac{\pi}{4}) \end{array} \right.$$

Les abscisses et les ordonnées sont strictement croissantes sur l'intervalle $]-\frac{\pi}{4}, \frac{\pi}{4}[$. Supposons que l'on répartisse N sites sur cette portion de spirale. Le $2d$-arbre va découper cet ensemble en cellules qui ont la propriété suivante : si l'on considère les sites d'une cellule, ils sont soit tous *avant* les sites d'une autre cellule (au sens des relations $<_V$ et $<_H$), soit tous *après* (figure 10). Par conséquent, lors de la fusion de deux cellules, tous les triangles de l'une ou de l'autre seront détruits.

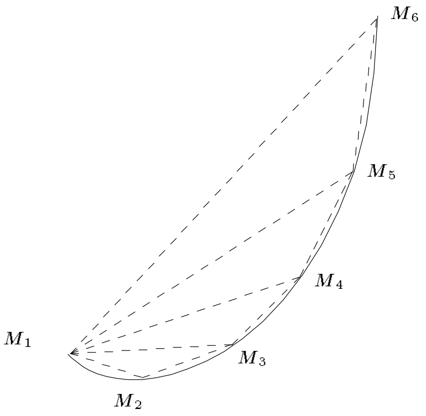


Figure 9. *Triangulation sur une portion de spirale*

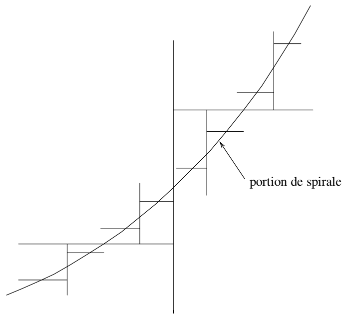


Figure 10. *Découpage induit par le $2d$-arbre*

On a donc l'équation de récurrence suivante :

$$T(N) = 2T(\frac{N}{2}) + \alpha N \Rightarrow T(N) \in \Omega(N \log N)$$

L'algorithme a une complexité dans le pire des cas dans $\Omega(N \log N)$. $\diamond$

4.2. ***Complexité en moyenne***

Pour calculer la complexité en moyenne, nous avons besoin de résultats établis par Guibas et Stolfi [GS 85], Dwyer [DWY 87], Katajainen et Koppinen [KK 88]. Ces résultats, ainsi que leur démonstration (parfois légèrement remaniée), sont rappelés dans les paragraphes 4.2.1, 4.2.2 et 4.2.3. Par contre, la partie 4.2.4 est originale et prouve que la triangulation se fait en $O(N)$ en moyenne.

On suppose que la distribution est quasi-uniforme sur un carré unité U. Soit f la densité de probabilité :

$$\begin{cases} \forall(x,y) \in U, & 0 < c_1 \leq f(x,y) \leq c_2 \\ \forall(x,y) \notin U, & f(x,y) = 0 \end{cases}$$

Par exemple, la probabilité qu'un point donné soit dans un domaine D vaut $\int_D f$.

4.2.1. *Sites inachevés*

Cette notion a été introduite par J. Katajainen et M. Koppinen dans [KK 88], et est primordiale dans la justification de la linéarité de la complexité en moyenne de l'algorithme décrit dans l'article présent. Nous en redonnons la définition :

Définition 4.1. *([KK 88]) Soient $T(S_1)$ et $T(S_2)$ des triangulations des ensembles S_1 et S_2. On note $T(S_1) <_\triangle T(S_2)$ si $S_1 \subseteq S_2$ et si $T(S_1)$ contient chaque arête de $T(S_2)$ dont les extrémités appartiennent à S_1.*

Cette définition crée un ordre partiel dans l'ensemble des triangulations. Remarquons que $T(S_1) <_\triangle T(S_2)$ n'implique pas que toutes les arêtes de $T(S_1)$ appartiennent à $T(S_2)$. Remarquons aussi que $TD(S_1) <_\triangle TD(S_2)$ est équivalent à $S_1 \subseteq S_2$ si S_1 a une triangulation de Delaunay unique.

Définition 4.2. *([KK 88]) Soient $T(S_1) <_\triangle T(S_2)$ deux triangulations et $s \in S_1$. On dit que s est achevé dans $T(S_1)$ relativement à $T(S_2)$ si l'ensemble des arêtes incidentes à s dans $T(S_1)$ et $T(S_2)$ coïncident, sinon on dit que s est inachevé.*

Proposition 4.1. *([KK 88]) Soient $T(S_1) <_\triangle T(S_2)$ deux triangulations. Un site $s \in S_1$ est achevé dans $T(S_1)$ relativement à $T(S_2)$ si et seulement si S_1 contient les extrémités de toutes les arêtes incidentes à s dans $T(S_2)$.*

Dém. Soient (s, p_1),... (s, p_m) les arêtes adjacentes à s dans $T(S_2)$. Si s est achevé, alors les extrémités p_i appartiennent à S_1 par définition.

Réciproquement, supposons que les extrémités p_i appartiennent à S_1 mais que s ne soit pas achevé. Alors, (s, p_1),... (s, p_m) appartiennent à $T(S_1)$ parce que $T(S_1) <_\triangle T(S_2)$, mais $T(S_1)$ contient au moins une autre arête (s, r). Puisque le domaine délimité par $T(S_2)$ est convexe, il contient entièrement l'arête (s, r).

Par conséquent, $(s, r) \setminus \{s\}$ coupe, soit une des arêtes (s, p_i), soit un des triangles ouverts de $T(S_2)$ ayant s pour sommet. La première alternative est évidemment impossible. Un triangle de $T(S_2)$ ne contient aucun site de S_2 en son intérieur, et en particulier pas r. Donc, (s, r) coupe le côté opposé (p_i, p_j), ce qui est tout autant impossible car (p_i, p_j) appartient aussi à $T(S_1)$, qui est une triangulation. $\diamond$

Corollaire 4.1. *([KK 88]) Soient $T(S_1) <_\triangle T(S_2) <_\triangle T(S_3)$ trois triangulations. Un site $s \in S_1$ est achevé dans $T(S_1)$ relativement à $T(S_3)$ si et seulement s'il est achevé à la fois dans $T(S_1)$ relativement à $T(S_2)$ et aussi dans $T(S_2)$ relativement à $T(S_3)$.*

Dém. La dernière condition implique la précédente à cause de la définition 4.2.

Réciproquement, supposons que s soit achevé dans $T(S_1)$ relativement à $T(S_3)$. Si (s, q) est une arête dans $T(S_3)$, alors $q \in S_1$; comme $q \in S_2$, en utilisant la proposition 4.1, on obtient que s est achevé dans $T(S_2)$ relativement à $T(S_3)$. Enfin, à cause de la définition 4.2, s est aussi achevé dans $T(S_1)$ relativement à $T(S_2)$. $\diamond$

4.2.2. *Complexité d'une fusion en fonction du nombre de sites inachevés*

Théorème 4.1. *([GS 85]) Soient G et D deux ensembles de sites situés respectivement à gauche et à droite d'une droite δ. Lors de la fusion de $TD(G)$ et $TD(D)$ pour construire $TD(G \cup D)$:*

1. *Seules des arêtes reliant deux sites de G et des arêtes reliant deux sites de D sont détruites.*
2. *Seules des arêtes reliant un site de G à un site de D sont créées.*
3. *La complexité dans le pire des cas du processus de fusion est bornée par une fonction linéaire de la somme des trois composants suivants :*

   (a) Le nombre de sites examinés pour trouver les extrémités de l'arête la plus basse de l'enveloppe convexe de $(G \cup D)$ qui coupe δ.

   (b) *Le nombre d'arêtes détruites.*

   (c) *Le nombre d'arêtes créées.*

4. *La complexité dans le pire des cas du processus de fusion est en* $O(|G \cup D|)$.

Dém. cf. [GS 85]. ◇

Lemme 4.1. *([DWY 87]) Toute triangulation d'un ensemble de n sites dont k appartiennent à l'enveloppe convexe, possède $e = 3n - k - 3$ arêtes et $t = 2n - k - 2$ triangles. En particulier, au plus $3n - 6$ arêtes peuvent être créées sur un ensemble de n sites.*

Dém. Soient t le nombre de triangles et e le nombre d'arêtes. Toutes les arêtes appartiennent à deux triangles sauf les k arêtes de l'enveloppe convexe, d'où $2e - k = 3t$. En utilisant la formule d'Euler $n - e + (t + 1) = 2$, on en déduit t et e. En faisant $k = 3$, on déduit le dernier résultat. ◇

Théorème 4.2. *([DWY 87]) La complexité dans le pire des cas du processus de fusion est bornée par une fonction linéaire du nombre de sites qui reçoivent de nouvelles arêtes.*

Dém. Grâce au théorème 4.1, il suffit de montrer que, si de nouvelles arêtes sont ancrées sur m sites, alors au plus $3m - 6$ arêtes sont créées, au plus $3m - 9$ arêtes sont détruites, et au plus m sites sont examinés pour construire l'arête la plus basse de l'enveloppe convexe de $G \cup D$ qui coupe δ.

Soient c et d le nombre d'arêtes créées et détruites. Puisque les arêtes créées ne se coupent pas, en raison du corollaire 4.1, on a $c \leq 3m - 6$.

Supposons que G, D et $G \cup D$ soient composés respectivement de n_1, n_2 et n sites, dont k_1, k_2 et k appartiennent à l'enveloppe convexe. On peut tirer l'enchaînement logique suivant :

$$\begin{array}{r}
\text{Lemme 4.1} \Rightarrow (3n_1 - k_1 - 3) + (3n_2 - k_2 - 3) + (c - d) = (3n - k - 3) \\
n = n_1 + n_2 \Rightarrow k - (k_1 + k_2) = 3 - c + d \qquad\qquad\qquad\qquad \\
\left.\begin{array}{r} k \leq k_1 + k_2 \\ 3 - c + d \leq 0 \end{array}\right\} \Rightarrow d \leq c - 3 \qquad\qquad\qquad\qquad\qquad\qquad \\
c \leq 3m - 6 \Rightarrow d \leq 3m - 9 \qquad\qquad\qquad\qquad\qquad\qquad
\end{array}$$

Enfin, nous observons que chaque site examiné pour trouver l'arête la plus basse de l'enveloppe convexe de $G \cup D$ qui coupe δ, reçoit une nouvelle arête pendant la fusion, donc leur nombre est inférieur à m. ◇

Corollaire 4.2. *([DWY 87]) La complexité dans le pire des cas du processus de fusion est majorée par une fonction linéaire du nombre d'arêtes créées.*

Dém. Évident. ◇

Corollaire 4.3. *La complexité dans le pire des cas de la fusion de G et D est majorée par $7m$ où m est le nombre de sites inachevés dans G et dans D.*

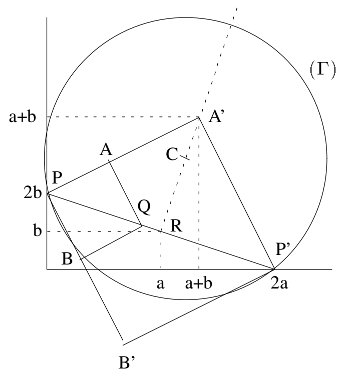


Figure 11. *Le cercle* (Γ) *contient* $\triangle PQA$ *ou* $\triangle PQB$

Dém. Le nombre total d'arêtes créées est majoré par $3m$, le nombre total d'arêtes détruites est majoré par $3m$, le nombre total de sites examinés pour trouver l'arête la plus basse de l'enveloppe convexe de $L \cup R$ qui coupe δ, est majoré par m. $\diamond$

4.2.3. *Nombre de sites inachevés dans un domaine rectangulaire*

Lemme 4.2. *([DWY 87]) Soient P, Q et P' trois points alignés dans cet ordre. Soient A et B les autres sommets du carré de diagonale PQ. Alors tout cercle (Γ) passant par P et P' contient entièrement en son intérieur soit le triangle $\triangle PQA$, soit le triangle $\triangle PQB$, soit les deux (figure 11).*

Dém. Soient A' et B' les autres sommets du carré de diagonale PP' du même côté que A et B respectivement. Puisque $\triangle PP'A'$ et $\triangle PP'B'$ contiennent respectivement $\triangle PQA$ et $\triangle PQB$, il suffit de montrer que le cercle contient soit A', soit B'. Sans perte de généralité, on peut supposer que $P(0, 2b)$ et $P'(2a, 0)$ avec $a \geq b \geq 0$ d'où $A'(a + b, a + b)$. Montrons alors que le centre C du cercle (Γ) est plus proche de A' que de P s'il est situé au-dessus de la droite PP' (l'autre cas se traite de la même manière).

Les points de la médiatrice de $[PP']$ sont à une distance de P supérieure ou égale à $\sqrt{a^2 + b^2}$.

– Si $C = R$, A' est sur le périmètre du cercle.

– Si C est situé entre R et A',

$$\left\{ \begin{array}{l} a \leq x_c \leq a + b \\ b \leq y_c \leq a + b \end{array} \right. \Longrightarrow d(C, A') = \sqrt{(a + b - x_c)^2 + (a + b - y_c)^2}$$

$$\Longrightarrow d(C, A') \leq \sqrt{a^2 + b^2} = d(R, P) \leq d(C, P)$$

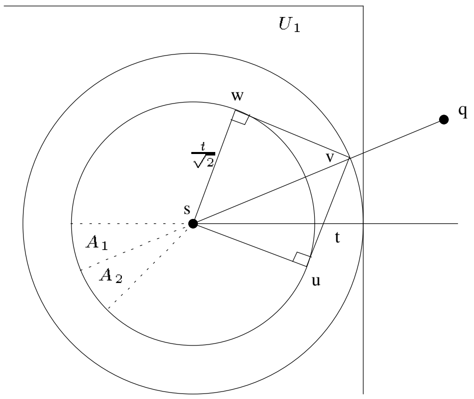


Figure 12. *L'un des secteurs A_i ne contient aucun site en son intérieur*

– Si C est situé au-dessus de A',

$$\left\{\begin{array}{l} x_c \geq a+b \\ y_c \geq a+b \end{array}\right. \Longrightarrow d(C,A') = \sqrt{(x_c-(a+b))^2+(y_c-(a+b))^2}$$

$$\Longrightarrow d(C,A') \leq \sqrt{x_c^2+(y_c-2b)^2} = d(C,P)$$

$\diamond$

Lemme 4.3. *([KK 88]) Supposons que la densité de probabilité f soit quasi-uniforme avec les bornes c_1 et c_2. Soit un rectangle $U_1 \subseteq U$ et $TD(S \cap U_1) < TD(S)$. Si $s \in S \cap U_1$ est un sommet situé à une distance t de la frontière de U_1, alors la probabilité que s soit inachevé dans $TD(S \cap U_1)$ relativement à $TD(S)$ est au plus $16\left(1-\frac{c_1\pi}{32}t^2\right)^{N-1}$.*

Dém. Soit C_1 la condition que s est inachevé dans $TD(S \cap U_1)$ relativement à $TD(S)$. Alors, d'après la proposition 4.1, $C_1 \Rightarrow C_2$, où C_2 est la condition que $TD(S)$ contient une arête (s,q) dont la longueur est plus grande que t. Considérons un cercle de centre s et de rayon $\frac{t}{\sqrt{2}}$. Divisons-le en 16 secteurs ouverts $A_1,\ldots,A_{16}$ d'angles $\frac{\pi}{8}$. Soit C_3 la condition qu'au moins un des secteurs ouverts soit vide (c'est-à-dire n'ait aucun site en son intérieur).

Supposons un instant que $C_2 \Rightarrow C_3$; grâce à la propriété $f(x,y) \geq c_1$, on obtient :

$$P(C_1) \leq P(C_3) \leq \sum_{i=1}^{16} P(S \cap A_i = \emptyset)$$

$$\Longrightarrow P(C_1) = \sum_{i=1}^{16}\left(1-\int_V f\right)^{N-1} \leq 16\left(1-\frac{c_1\pi}{32}t^2\right)^{N-1}$$

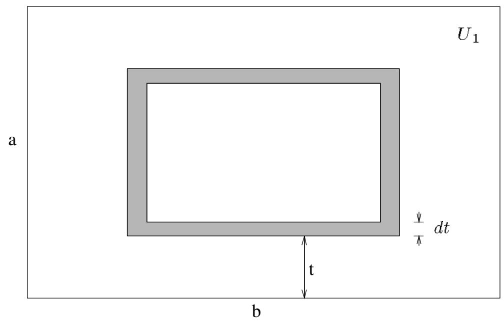


Figure 13. *Espérance du nombre de sites inachevés dans le rectangle U_1*

Il reste à montrer que $C_2 \Rightarrow C_3$. Supposons C_2, et considérons le carré $\Box suvw$ avec une diagonale sv s'appuyant sur (s,q), ayant pour extrémité s et une longueur de $\frac{t}{\sqrt{2}}$ (figure 12). Cette diagonale sv divise le carré en deux triangles $\triangle svu$ et $\triangle svw$. Puisque (s,q) est dans $TD(S)$, il existe un cercle passant par s et q, ne contenant aucun site en son intérieur. Ce cercle contient au moins l'un des deux triangles $\triangle svu$ ou $\triangle svw$ (voir lemme 4.2). Par conséquent, il contient aussi l'un des secteurs A_i. Par suite, A_i ne contient aucun site en son intérieur. $\diamond$

Corollaire 4.4. *([KK 88]) On se place dans les conditions du lemme 4.3. Soient a et b les longueurs des côtés de U_1 et $E(a,b)$ l'espérance du nombre de sites inachevés dans U_1. On a :*

$$E(a,b) < 103\left(\frac{c_2}{\sqrt{c_1}}\right)(a+b)\sqrt{N}$$

Dém. Supposons $a \leq b$. Posons $0 \leq t \leq \frac{a}{2}$. Les points de U_1 situés à une distance de la frontière du rectangle U_1 comprise entre t et $t+dt$ forment une région A_t d'aire $2(a+b-4t)dt$ (figure 13). La probabilité qu'un site donné appartienne à A_t est :

$$\int_{A_t} f \leq c_2 \times 2(a+b-4t)dt \leq 2c_2(a+b)dt$$

En utilisant le lemme 4.3, on peut estimer :

$$\begin{aligned} E(a,b) &\leq N\int_0^{\frac{a}{2}} 16\left(1-\frac{c_1\pi t^2}{32}\right)^{N-1} 2c_2(a+b)dt \\ &= 32c_2(a+b)N\sqrt{\frac{32}{\pi c_1}}\int_0^{a\sqrt{\frac{\pi c_1}{128}}}(1-x^2)^{N-1}dx \end{aligned}$$

avec $x = t\sqrt{\frac{\pi c_1}{32}}$

Appelons I l'intégrale. Puisque $a < 1$ et $c_1 \leq 1$,

$$I < \int_0^1 (1-x^2)^{N-1} dx = \int_0^{\frac{\pi}{2}} (\cos\theta)^{2N-1} d\theta = I_{2N-1},$$

en posant $x = \sin\theta$.

Cette quantité I (dite intégrale de *Wallis*) se calcule par récurrence avec une intégration par parties :

$$\begin{aligned} I_{2N-1} &= \frac{2 \times 4 \times 6 \times \cdots (2N-2)}{3 \times 5 \times 7 \times \cdots (2N-1)} \\ 2 \times I_{2N} &= \frac{3 \times 5 \times 7 \times \cdots (2N-1)}{4 \times 6 \times 8 \times \cdots (2N)} \end{aligned}$$

Puisque $\frac{n}{n+1} < \frac{n+1}{n+2}$, on obtient en comparant terme à terme : $I_{2N-1} < 2 \times I_{2N}$.

Comme $I_{2N-1} \times (2 \times I_{2N}) = \frac{1}{N}$, on déduit :

$$I_{2N-1} < \frac{1}{\sqrt{N}} \Rightarrow I < \frac{1}{\sqrt{N}}$$

d'où $E(a,b) < 103 \left(\frac{c_2}{\sqrt{c_1}}\right) (a+b)\sqrt{N}$. $\diamond$

4.2.4. *Complexité de la triangulation*

Nous allons maintenant utiliser ce résultat pour majorer l'espérance E du nombre total de sites inachevés au cours des différentes fusions. En sommant $(a+b)$ sur toutes les cellules de chaque niveau de fusionnement, on obtient :

$$E < 103 \frac{c_2}{\sqrt{c_1}} \sqrt{N} \sum_{cellules} (a+b)$$

Soit N le nombre de sites situés dans le carré unité U. Comme l'illustre la figure 14, les diverses fusions entre cellules peuvent être schématisées par une arborescence de hauteur $k = \lceil \log_2 N \rceil$. Cette quantité k, qui est, naturellement, la longueur du plus long chemin de cette arborescence, représente aussi le nombre de niveaux de fusionnement. On remarque alors :

$$k = \lceil \log_2 N \rceil \Rightarrow 2^{k-1} < N \leq 2^k \Rightarrow \sqrt{N} > 2^{\frac{k-1}{2}} \geq 2^{\lceil \frac{k}{2} \rceil - 1}$$

Au niveau p de fusionnement, il y a exactement 2^p cellules (sauf pour $p = k$ où il y en a au plus 2^k). Soit N_x le nombre de cellules adjacentes au côté horizontal du carré U et N_y le nombre de cellules adjacentes au côté vertical du carré U. Puisque le processus de division commence verticalement, on a :

$$N_x = 2^{\lceil \frac{p}{2} \rceil} \text{ et } N_y = 2^{\lfloor \frac{p}{2} \rfloor}.$$

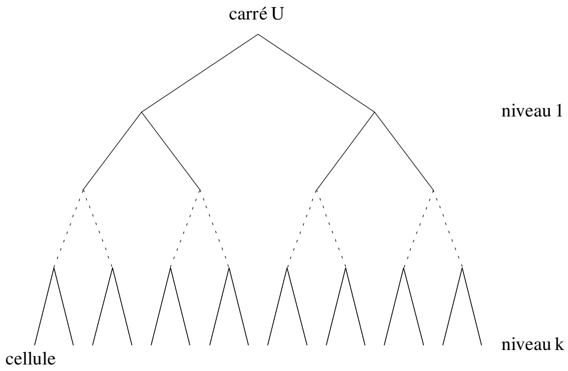


Figure 14. *Arborescence schématisant les fusions*

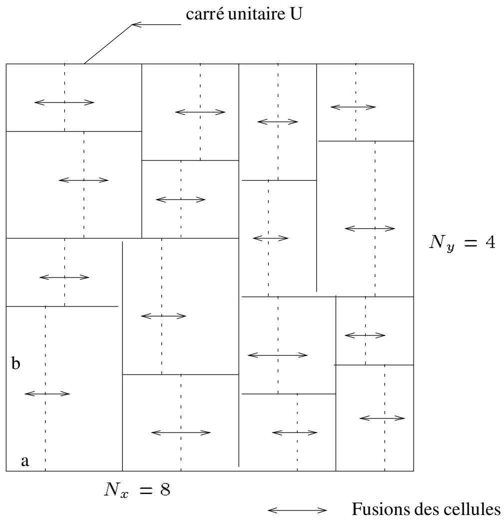


Figure 15. *Fusions au niveau p*

On remarque que la somme des côtés horizontaux (*resp.* verticaux) d'une *ligne* (*resp. colonne*) de cellules vaut 1. Comme il y a N_y lignes et N_x colonnes, on obtient en sommant sur toutes les cellules d'un niveau p de fusionnement :

$$\sum_{cellules} (a+b) = N_x + N_y = 2^{\lceil \frac{p}{2} \rceil} + 2^{\lfloor \frac{p}{2} \rfloor} < 2^{\lceil \frac{p}{2} \rceil + 1}$$

Sur l'ensemble des **k niveaux** de fusionnement :

$$\begin{aligned}
\sum_{cellules} (a+b) &< \sum_{p=1}^{k} 2^{\lceil \frac{p}{2} \rceil + 1} \\
&= 2^2 + 2^2 + 2^3 + 2^3 + \cdots + 2^{\lceil \frac{k}{2} \rceil + 1} \\
&\leq 2^3 + 2^4 + 2^5 + \cdots + 2^{\lceil \frac{k}{2} \rceil + 2} \\
&\leq 2^{\lceil \frac{k}{2} \rceil + 3}
\end{aligned}$$

Comme $\sqrt{N} \geq 2^{\lceil \frac{k}{2} \rceil - 1}$, $\sum_{cellules} (a+b) \leq 16\sqrt{N}$,

on obtient donc : $E < 1648 \frac{c_2}{\sqrt{c_1}} N = O(N)$ ce qui constitue un majorant pour l'espérance du nombre total de sites inachevés sur l'ensemble des différentes fusions.

En multipliant par 7 (*cf.* corollaire 4.3), on obtient un majorant de la complexité en moyenne du processus de triangulation (tri en $2d$ exclu bien sûr), qui est donc linéaire. $\diamond$

5. Résultats expérimentaux

Expérimentalement, nous avons constaté que le temps de triangulation est divisé d'un facteur variant entre 2 et 3 par rapport à celui de l'algorithme de Lee et Schachter. Lors du processus de fusion, le nombre total d'arêtes créées est voisin de $4N$ (N étant le nombre de sites), le nombre total d'arêtes détruites est voisin de N, et le nombre d'arêtes restant dans la triangulation définitive voisin de $3N$.

Les tests ont porté sur des ensembles atteignant sept millions de sites répartis dans un domaine carré suivant une loi uniforme. Les algorithmes ont été testés sur une station SUN, sur une Silicon Graphics Indy ainsi que sur un super-calculateur Convex C3, avec lequel les résultats présentés à la figure 16 ont été obtenus.

Ces résultats “collent” très bien à la théorie et montrent les comportements asymptotiques en $O(N \log N)$ pour l'algorithme de Lee et Schachter et en $O(N)$ pour l'algorithme optimisé (temps consacré au tri exclu). On a choisi de mesurer les performances à l'aide du nombre total d'arêtes créées par le programme puisque le temps de triangulation est proportionnel à ce nombre (voir corollaire 4.2). On s'affranchit ainsi de la vitesse du processeur utilisé, des problèmes dus au réseau et aux utilisateurs travaillant simultanément sur la même machine. L'écart entre les deux algorithmes commence à être sensible à partir de 60 sites et, pour 130 000 sites, l'algorithme optimisé est déjà deux fois plus rapide. L'écart s'amplifie ensuite au fur et à mesure que N augmente.

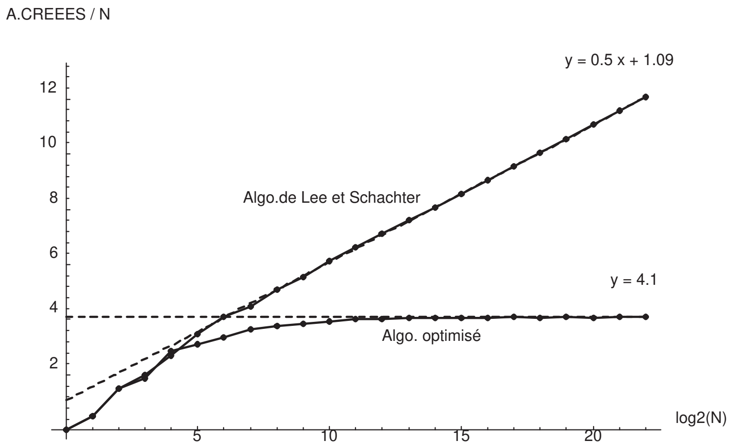


Figure 16. *Comparaison entre les deux algorithmes*

Sur une Silicon Graphics Indy (200 Mhz, 64 MO de mémoire vive), on triangule 200 000 sites en 7 secondes après une phase de tri de trois secondes. La vitesse de triangulation (en excluant le temps consacré au tri) est de l'ordre de 30 000 sites (ou 50 000 triangles) par seconde.

6. Remarques et conclusion

Nous avons présenté un algorithme qui, après un tri selon deux directions de N sites répartis de manière quasi-uniforme dans un carré unité, réalise la triangulation de Delaunay en $O(N)$ *en moyenne*.

L'efficacité de cet algorithme est due au fait que les fusions, réalisées alternativement selon le sens vertical et horizontal, mettent relativement moins souvent en cause les triangles déjà construits que dans l'algorithme classique de Lee et Schachter.

Cet algorithme semble avoir le même comportement sur beaucoup d'autres types de distribution (que quasi-uniforme), mais une étude théorique et expérimentale approfondie est nécessaire pour le prouver.

Certains algorithmes réalisent la triangulation de Delaunay en $O(N \log\log N)$ *en moyenne* ([DWY 87]), et même en $O(N)$ *en moyenne* ([KK 88]). Mais ils utilisent tous une grille régulière pour pré-partitionner les points, ce qui réduit considérablement les possibilités d'utilisation de tels algorithmes.

Nous terminons cette étude par le problème ouvert suivant, déjà mentionné par O. Devillers (*INRIA*, Sophia Antipolis) : *est-il possible, à partir de la connaissance du tri selon deux directions d'un ensemble de N sites du plan, de calculer leur triangulation de Delaunay euclidienne en $o(N \log N)$ dans le pire des cas ?*

NOTE. Signalons, pour finir, les travaux de l'équipe de J. C. Spehner (Université de Haute Alsace), qui calcule le diagramme de Delaunay en utilisant des mélanges de tris [AES 96].

7. Bibliographie

[AES 96] B. ADAM, M. ELBAZ, J.C. SPEHNER. Construction du diagramme de Delaunay dans le plan en utilisant les mélanges de tris. *Journées de l'AFIG 96*, Dijon, nov. 1996.

[AUR 91] F. AURENHAMMER. Voronoi diagrams: a survey of a fundamental geometric data structure, *ACM Comput. Surv.*, vol 23, pages 345–405, 1991.

[BAU 75] B.G. BAUMGART. A polyhedron representation for Computer Vision. *National Computer Conference*, pages 589–596, 1975.

[BEN 75] J.L. BENTLEY. Multidimensional binary search trees used for associated searching. *Communications of the ACM*, vol. 18, pages 509–517, 1975.

[BFPRT 73] M. BLUM, R.W. FLOYD, V. PRATT, R.L. RIVEST, R.E. TARJAN. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4), pages 448–461, 1973.

[CLR 94] T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST. *Introduction à l'algorithmique*. Traduction de X. Cazin, Dunod, Paris, 1994.

[COR 75] R. CORI. Un code pour les graphes planaires et ses applications. *Astérisque*, 27, 1975.

[DWY 87] R.A. DWYER. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2, pages 137–151, 1987.

[DWY 91] R.A. DWYER. Higher-Dimensional Voronoi Diagrams in Linear Expected Time. *Discrete Comput. Geom.*, 6, pages 343–367, 1991.

[GS 85] L.J. GUIBAS, J. STOLFI. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics*, 4, pages 74–123, 1985.

[HOU 87] P. HOUTHUYS. Box Sort, a multidimensional binary sorting method for rectangular boxes, used for quick range searching. *The Visual Computer*, 3, pages 236–249, 1987.

[KK 88] J. KATAJAINEN, M. KOPPINEN. Constructing Delaunay triangulations by merging buckets in quadtree order. *Annales Societatis mathematicae Polonae*, Series IV, Fundamenta Informaticæ, 11, pages 275–288, 1988.

[LS 80] D.T. LEE, B.J. SCHACHTER. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer and Information Sciences*, 9, pages 219–242, 1980.

[MAU 84] A. MAUS. Delaunay triangulation and the convex hull of n points in expected linear time. *BIT*, 24, pages 151–163, 1984.

[OIM 84] T. OHA, M. IRI, K. MUROTA. Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms. *Journal of the Operations Research Society of Japan*, 27, pages 306–337, 1984.

[SHE 96] J.R. SHEWCHUK. Triangle: Engineering a $2D$ Quality Mesh Triangulator. In *First Workshop on Applied Computational Geometry*, ACM, pages 124–133, May 1996.

Remerciements : Les auteurs tiennent à remercier, au *SETRA*, Jacques Hervé pour son aide en informatique et ses encouragements, Philippe Nouaille pour sa participation au codage de l'optimisation, Tam Vo-Dinh pour la mise à disposition du calculateur Convex C3, et enfin Jean-Noël Theillout qui a permis à C. Lemaire de mener à terme cette étude.

Nous tenons aussi à remercier le Professeur Ferran Hurtado pour nous avoir communiqué les références de l'article de J.R. Shewchuk ([SHE 96]), représentant un travail totalement indépendant du nôtre, et dans lequel ce dernier présente une technique de division-fusion bi-directionnelle similaire, mais sans en fournir l'analyse de complexité exhaustive exposée ici.

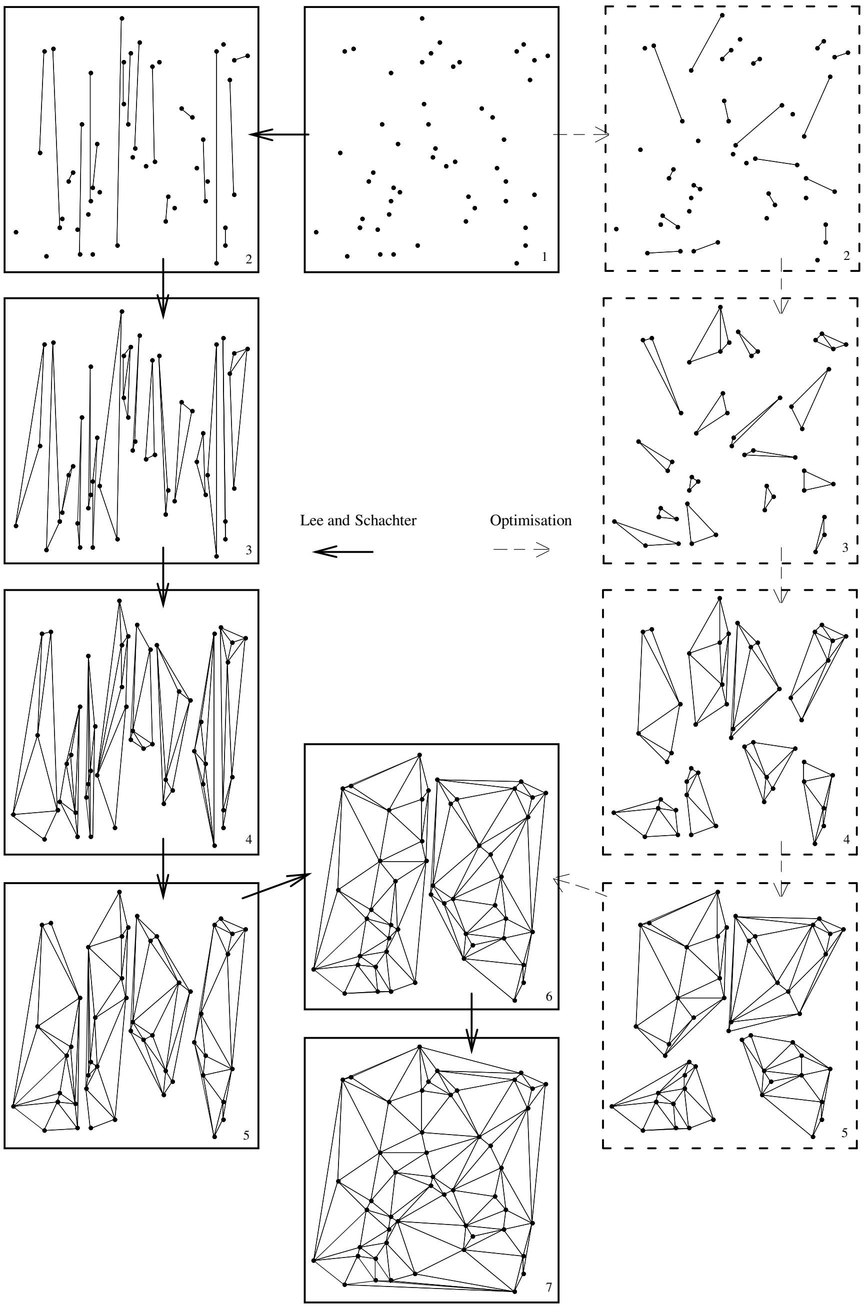


Figure 17. *Etapes successives de la triangulation : comparaison entre les deux algorithmes (ensemble de* 50 *sites)*

Le coin du Web

Voici en vrac quelques adresses internet[1] utiles au géomètre algorithmicien.

- **Généralités**
  - http://www.scs.carleton.ca/~csgs/resources/cg.html *contient des adresses vers d'autres répertoires, les centres de recherche dans le monde, bibliographie en ligne générale et par sujet, newsgroups, et logiciels freeware.*
  - http://www.cs.duke.edu/~jeffe/compgeom/compgeom.html *page maintenue par J. Erikson, beaucoup d'informations.*
  - http://forum.swarthmore.edu/advanced/geom.html *géométrie avec géométrie différentielle, fractals, résolution par contraintes, modélisation, etc...*
  - http://www.ics.uci.edu/~eppstein/junkyard/topic.html *une collection de problèmes en géométrie et informatique.*
- **Chercheurs**
  - http://www.dcc.unicamp.br/~guialbu/geompages.html *liste de chercheurs (avec e-mail ou URL) dans le domaine avec leurs labos respectifs.*
- **Événements**
  - http://www.cs.duke.edu/~jeffe/compgeom/events.html *liste des prochaines manifestations dans le domaine de la géométrie algorithmique.*
- **Bibliographie**
  - ftp://alf.usask.ca/pub/geometry/geombib *une bibliographie trés complète en géométrie algorithmique, avec un logiciel de recherche par mots clés, où chacun peut ajouter des références.*
- **Avenir de la géométrie algorithmique**

[1] Je remercie Philippe Biscondi et surtout Hélymar Balza-Gómez à qui je dois certaines de ces adresses Web.

- http://www.cs.brown.edu/people/rt/sdcr/report/report.html *groupe de travail sur les directions stratégiques en géométrie algorithmique.*

- **Cours**
  - http://www.cs.brown.edu/courses/cs252 *cours de géométrie algorithmique en ligne contenant entre autres : structures de données, balayage et intersection, localisation, enveloppe convexe, diagrammes de proximité, le tout avec notes de cours, papiers à lire et démos en ligne (Java).*
  - http://www.cs.curtin.edu.au/units/cg201/notes/cg201.html *fiches de cours de géométrie générale avec de la géométrie algorithmique.*

- **Applications**
  - http://ra.cfm.ohio-state.edu/grad/zhao/algorithms/algorithms.html *quelques algorithmes implémentés en Java.*
  - http://www.geom.umn.edu/software/cglist/ *liste de programmes ou logiciels en géométrie algorithmique, liste maintenue par N. Amenta.*
  - http://graphics.lcs.mit.edu/~seth/geomlib/geomlib.html *page où S. Teller a une collection de programmes concernant la programmation linéaire, les diagrammes de Voronoi et la manipulation des surfaces NURBS.*
  - http://www.wri.com/mathsource/ *beaucoup de programmes, documents, exemples en mathématiques (environ 100 000 pages) dont quelques-uns concernent la géométrie algorithmique.*
  - ftp://grendel.csc.smith.edu/pub/compgeom/ *page où l'on trouve le code C du livre de O'Rourke ([160]).*
  - http://www.mpi-sb.mpg.de/LEDA/leda.html *implémentations de plusieurs algorithmes de géométrie algorithmique, contenus dans le projet LEDA (Library of Efficient Data Types and Algorithms) de l'Institut Max Planck.*

- **Dictionnaires**
  - http://www.gps.caltech.edu/~eww/math/math.html *définitions de termes mathématiques avec références, très riche.*
  - http://www.postech.ac.kr/~otfried/html/francais.html *dictionnaire français-anglais de géométrie algorithmique.*

- **Maillages**
  - http://www.ce.cmu.edu/~sowen/mesh.html *bibliographie très complète sur les maillages et les méthodes d'éléments finis, par S. Owen.*

- http://www.cs.cmu.edu/~ph/mesh.html *les maillages pour les surfaces et les éléments finis par P. Heckbert.*
- http://www.ics.uci.edu/~eppstein/meshgen.txt *contient la bibliographie de l'article de synthèse de M. Bern et D. Eppstein sur les maillages et les triangulations optimales pour la méthode des éléments finis.*
- http://www-users.informatik.rwth-aachen.de/~roberts/meshgeneration.html *liens vers des adresses orientées maillages et éléments finis, ainsi qu'une liste de personnes intéressées par ce domaine, par R. Schneiders.*
- http://www.cs.wisc.edu/~deboor/bib/bib.html *bibliographie très complète sur les maillages de surfaces et en particulier sur les splines.*
- http://sass577.endo.sandia.gov:80/9225/Personnel/samitch/roundtable96 *liste des papiers de la cinquième table ronde sur le maillage.*

- **Histoire**
  - http://www-groups.dcs.st-and.ac.uk:80/~history/ *histoire très complète des mathématiques avec la biographie de plus de 1100 mathématiciens.*
  - http://www.scottlan.edu/Iriddle/women/women.html *biographie de femmes mathématiciennes.*

- **Forums**
  - http://pine.fernuni-hagen.de/GI/compgeom/compgeom.html *permet l'accès aux 'mailing' listes en géométrie algorithmique. Il y a 3 listes :* compgeom-discuss *pour les discussions et les questions-réponses,* compgeom-tribune *pour recevoir le journal CG Tribune et* compgeom-announce *pour l'annonce des principaux événements en géométrie algorithmique.*
  - http://forum.swarthmore.edu/~sarah/topics/about.newsgroups.html *permet l'accès aux "mailing" listes en géométrie en général.*

- **Journaux de géométrie algorithmique**
  - http://www.inria.fr/prisme/personnel/devillers/gedeon.html *journal gratuit édité par J. M. Moreau (précédemment par O. Devillers), où les chercheurs en géométrie algorithmique peuvent s'exprimer. On y trouve aussi le calendrier des principaux événements, des comptes rendus de conférences, des résumés de thèses et d'articles, des entretiens électroniques, une rubrique questions et réponses...*
  - http://www.inria.fr/prisme/personnel/bronnimann/cgt *le même type de journal, mais en anglais, édité par H. Brönnimann.*

– http://www.lirmm.fr/~nourine/graal.html *un journal sur les graphes et la recherche opérationnelle.*

Les ouvrages

Voici quelques ouvrages qui m'ont été utiles pendant ces années de thèse.

- **Preparata et Shamos** [165]: *les bases de la géométrie algorithmique.*
- **Edelsbrünner** [74]: *comme le précédent, il contient les bases de la géométrie algorithmique avec plus de combinatoire.*
- **Mehlhorn** [143]: *ouvrage général en géométrie algorithmique. Les deux livres précédents du même auteur traitent des graphes, des algorithmes et de leur analyse.*
- **Samet** [171]: *un livre sur les kd-trees, quadtrees et sur les méthodes de buckets.*
- **Okabe, Boots et Sugihara** [159]: *ouvrage extrêmement complet sur les triangulations de Delaunay et les diagrammes de Voronoi.*
- **Mulmuley** [156]: *très complet sur les algorithmes randomisés.*
- **Boissonnat et Yvinec** [34]: *ouvrage général traitant de l'algorithmique, de la complexité, et surtout de la randomisation en géométrie algorithmique. On y trouve les bases de la géométrie, même en dimension quelconque.*
- **de Berg, van Kreveld, Overmars, Schwarzkopk** [21]
- **Sack et Urrutia** [170]
- **Goodman et O'Rourke** (éditeurs) [92] : *très complet, chaque chapitre est écrit par des auteurs différents.*

D'autres livres, qui ne traitent pas que de géométrie algorithmique, sont néanmoins très utiles. Citons par exemple :

- **Algorithmes en langage C** [178] de R. Sedgewick : *les bases de l'algorithmique avec un chapitre spécial géométrie algorithmique.*

- **Introduction à l'algorithmique et à la programmation** [58] de P. Cousot : *toutes les bases de l'algorithmique, cours de l'Ecole Polytechnique.*
- **Eléments d'algorithmique** [13] de Beauquier, Berstel, Chrétienne : *les fondements de l'algorithmique avec des structures de données plus sophistiquées, très rigoureux, bases théoriques solides et un chapitre spécial géométrie algorithmique.*
- **Introduction à l'algorithmique** [57] de Cormen, Leiserson et Rivest : *un ouvrage extrêmement complet sur l'algorithmique en général, avec un chapitre spécial géométrie algorithmique.*
- **géométrie** [22], [23], [24], [25], [26] et [27] de M. Berger : *5 tomes de géométrie générale, souvent utilisé pour préparer l'agrégation de mathématiques.*
- **Stochastic Geometry and Its Applications** [186] de Stoyan, Kendall et Mecke : *un des rares livres traitant de géométrie stochastique.*
- **The probabilistic method** [6] de Alon et Spencer.
- **Concrete Mathematics, a foundation for computer science** [96] de Graham, Knuth et Patashnik : *bases mathématiques avancées pour l'informatique, pour l'analyse des algorithmes, notions mathématiques très sophistiquées.*

Et enfin trois ouvrages sur le traitement de texte LaTeX, qui ont été très utiles à la réalisation de ce mémoire :

- **LaTeX A Document Preparation System** [126] de Lamport : *permet de débuter avec LaTeX.*
- **The LaTeX Companion** [95] de Goossens, Mittelbach et Samarin : *ouvrage de base sur le traitement de texte LaTeX.*
- **A Guide to LaTeX2ϵ** [123] de Kopka et Daly : *ouvrage très complet sur LaTeX2ϵ.*

Bibliographie

[1] E.A. Abbott. Flatland. Ed. *Dover*, New York, 1952 (publié pour la première fois en 1884) (http://www.tiac.net/users/eldred/eaa/FL.HTM#textnote).

[2] B. Adam. Construction des diagrammes de Delaunay dans le plan et dans l'espace. *Thèse de l'Université de Haute-Alsace*, N^0 453, 208 pages, 1996.

[3] B. Adam, M. Elbaz, J.C. Spehner. Construction du diagramme de Delaunay dans le plan en utilisant les mélanges de tris. In *Quatrièmes Journées de l'AFIG*, Dijon, pages 215–223, 1996.

[4] B. Adam, M. Elbaz, J.C. Spehner. Construction du diagramme de Delaunay dans le plan en utilisant les mélanges de tris. proposé à *Revue internationale de CFAO et d'informatique graphique*, Hermès, 1997.

[5] G.M. Adelson-Velskii, E.M. Landis. An information organization algorithm. In *Doklady Akad. Nauk SSSR*, vol. 146, pages 263–266, 1962.

[6] N. Alon, J.H. Spencer. The probabilistic method. Ed *J. Wiley & Sons*, 245 pages, 1993 (http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/english/scripts.html#Alon-Spencer).

[7] T. Asano, M. Edahiro, H. Imai, M. Iri, K. Murota. Practical Use of Bucketing Techniques in Computational Geometry. In *Computational Geometry*, édité par G. T. Toussaint, Elsever Science Publishers B. V. (North Holland), pages 153–195, 1985.

[8] F. Avnaim, J.D. Boissonnat, O. Devillers, F.P. Preparata, M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Communication at the 11th Annu. ACM Sympos. Comput. Geom.*, 1995 (http://www.inria.prisme/personnel/devillers/anglais/determinant.html).

[9] C.B. Barber, D.P. Dobkin, H. Huhdanpaa. The Quickhull algorithm for convex hull. *Technical Report GCG53*, Geometry Center, Univ. of Minnesota, juillet 1993.

[10] R.E. Barnhill, S.N. Kersey. A marching method for parametric surface / surface intersection. In *Computer Aided Geometric Design*, 7, pages 257–280, 1990.

[11] B.G. Baumgart. A polyhedron representation for Computer Vision. In *National Computer Conference*, pages 589–596, 1975.

[12] H. Baumgarten, H. Jung, K. Melhorn. Dynamic point location in general subdivisions. In *J. Algorithms*, 17, pages 342–380, 1994.

[13] D. Beauquier, J. Berstel, Ph. Chrétienne. Eléments d'algorithmique. Ed. *Masson*, Paris, France, 463 pages, 1992.

[14] M. Benouamer, P. Jaillon, D. Michelucci, J.M. Moreau. A lazy solution to imprecision in Computational Geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, Waterloo, Canada, pages 73–78, 1993 (http://www.emse.fr/ECOLE/FRENCH/SIMADE/LISSE/GEOMETRI.HTML).

[15] M. Benouamer, P. Jaillon, D. Michelucci, J.M. Moreau. A lazy arithmetic library. In *Proc. of the IEEE 11th Symposium on Computer Arithmetic*, Windsor, Ontario, 1993.

[16] J.L. Bentley. Multidimensional binary search trees used for associated searching. In *Communications of the ACM*, vol. 18, pages 509-517, 1975.

[17] J.L. Bentley. Multidimensional Binary Search Trees in Database Applications. In *IEEE Transactions on Software Engineering*, vol. SE-5, 4, pages 333–340, juillet 1979.

[18] J.L. Bentley. Kd-trees for Semidynamic Point Sets. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[19] J.L. Bentley, T. Ottman. Algorithms for reporting and counting geometric intersections. In *IEEE Transactions on Computers*, C-28(9), pages 643–647, 1979.

[20] M. de Berg, K. Dobrindt. On Levels of Detail in Terrains. *Technical Report UU-CS-1995-12*, université d'Utrecht, 19 pages, avril 1995.

[21] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Computational Geometry, Algorithms and Applications. Ed. *Springer-Verlag*, à paraître en mai 1997.

[22] M. Berger. Géométrie 1. Action de groupes, espaces affines et projectifs. Ed. *Fernand Nathan*, Paris, 191 pages, 1977.

[23] M. Berger. Géométrie 2. Espaces euclidiens, triangles, cercles et sphères. Ed. *Fernand Nathan*, Paris, 214 pages, 1977.

[24] M. Berger. Géométrie 3. Convexes et polytopes, polyèdres réguliers, aires et volumes. Ed. *Fernand Nathan*, Paris, 176 pages, 1978.

[25] M. Berger. Géométrie 4. Formes quadratiques, quadriques et coniques. Ed. *Fernand Nathan*, Paris, 218 pages, 1978.

[26] M. Berger. Géométrie 5. La sphère pour elle-même, géométrie hyperbolique, l'espace des sphères. Ed. *Fernand Nathan*, Paris, 189 pages, 1977.

[27] M. Berger. Geometry. Ed. *Springer Verlag*, Berlin, 1987.

[28] M. Bern, D. Eppstein, F. Yao. The expected extremes in a Delaunay triangulation. In *International Journal of Computational Geometry and Applications*, 1(1), pages 79–91, 1991.

[29] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan. Time bounds for selection. In *Journal of Computer and System Sciences*, 7(4), pages 448–461, 1973.

[30] W. Boehm, H. Prautzsch. Geometrics Concepts for Geometric Design. Ed. *A K Peters*, Wellesley, Massachusetts, 393 pages, 1994.

[31] J.D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. In *Discrete Comput. Geom.*, 8, pages 51–71, 1992.

[32] J.D Boissonnat, M. Teillaud. A hierarchical representation of objects: the Delaunay tree. In *Second ACM Symposium of Comput. Geom. in Yorktown Heights*, pages 260–268, juin 1986.

[33] J.D Boissonnat, M. Teillaud. On the randomized construction of the Delaunay tree. In *Theoret. Comput. Sci.*, 112, pages 339–354, 1993.
(programme ftp://ftp-sop.inria.fr/prisme/del-tree)

[34] J.D. Boissonnat, M. Yvinec. Géométrie Algorithmique. Ed. *Ediscience*, Paris, 540 pages, 1995.

[35] P. Bose, L. Devroye. Intersections with random geometric objets. *Manuscrit*, School of Computer Science, McGill University, Montreal, 28 pages, 1995.

[36] H. Borouchaki, P.L. George. Triangulation de Delaunay et maillage. Ed. *Hermès*, ISBN 2-86601-625-4, 1997.

[37] A. Bowyer. Computing Dirichlet tesselations. In *The Computer Journal*, 24, pages 162–166, 1981.

[38] H. Brönnimann, M. Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, Nice, 8 pages, juin 1997.

[39] C.E. Buckley. A divide-and-conquer algorithm for computing 4-dimensional convex hulls. In *Lecture Notes in Computer Science*, 333 (International Workshop on Computational Geometry, Wurzburg, March 1988), Berlin: Springer-Verlag, pages. 113-135, 1988.

[40] M. Buffa. Navigation d'un robot mobile à l'aide de la stéréovision et de la triangulation de Delaunay. *Thèse de Doct. de l'Université de Nice*, N^0 4652, 331 pages, 1993.

[41] C. Burnikel, R. Fleischer, K. Melhorn, S. Schirra. A Strong and Easily Computable Separation Bound for Arithmetic Expressions Involving Radicals. In *SODA' 97*, 14 pages, 1997 (à paraître).

[42] C. Burnikel, K. Melhorn, S. Schirra. The LEDA Class Real Number. *Technical Report MPI-I96-1-001*, Max-Plank-Institut für Informatik, Saarbrücken, 52 pages, janvier 1996 (cf. http://data.mpi-sb.mpg.de/internet/reports.nsf).

[43] B. Cambray. Modèles Numériques de Terrain. *Rapport MASI 92.70*, Laboratoire MASI (CNRS URA 818), université Paris VI, 33 pages, 1992.

[44] P. Chanzy, L. Devroye. Range Search and Nearest Neighbor Search in Kd-Trees. *Manuscrit*, School of Computer Science, McGill University, Montreal, 31 pages, 1994.

[45] B. Chazelle. Triangulating a simple polygon in linear time. In *Discrete Comput Geom*, 6, pages 485–524, 1991.

[46] B. Chazelle. Application Challenges to Computational Geometry. *Technical Report TR-521-96, Princeton University*, 57 pages, avril 1996 (http://www.cs.princeton.edu/~chazelle/taskforce/CGreport.ps).

[47] B. Chazelle, L.J. Guibas. Fractional cascading : I. A data structuring technique. In *Algorithmica*, 1, pages 133–162, 1986.

[48] L. Chen, R. Schott. Optimal Operations on Red-Black Trees. In *International Journal of Foundations of Computer Science*, Vol. 7, N. 3, pages 227–239, 1996.

[49] S.W. Cheng, R. Janardan. New results on dynamic planar point location. In *SIAM J. Comput.*, 21, pages 972–999, 1992.

[50] L.P. Chew. Constrained Delaunay triangulations. In *Algorithmica*, 4(1), pages 97–108, 1989.

[51] L.P. Chew, S. Fortune. Sorting Helps for Voronoi Diagrams. *Technical Report TR 93-1347*, Department of Computer Science, Cornell University, Ithaca, 13 pages, mai 1993.

[52] Y. Chiang, R. Tamassia. Dynamic Algorithms in Computational Geometry. In *Proceedings of the IEEE*, vol. 80, 9, septembre 92.

[53] Y. Chipot. Génération et Modification de Surfaces Triangulées. *Thèse de L'Institut National Polytechnique de Lorraine*, N^0 063N, 168 pages, 1991.

[54] P. Cignoni, C. Montani, R. Scopigno. DeWall : a fast divide-and-conquer Delaunay trangulation algorithm in E^d. In *Tech. Rep. 95-22*, Instituto CNUCE-CNR, Pisa, Italy, 1995.

[55] K. L. Clarkson, P. W. Shor. Applications of random sampling in computational geometry. In *Discrete and Computational Geometry*, 4, pages 387–421, 1989.

[56] R. Cori. Un code pour les graphes planaires et ses applications. In *Astérisque*, 27, 1975.

[57] T. Cormen, C. Leiserson, R. Rivest. Introduction à l'algorithmique (traduit par X. Cazin). Ed. *Dunod*, Paris, 1019 pages, 1994.

[58] P. Cousot. Introduction à l'algorithmique et à la programmation. *Cours de l'Ecole Polytechnique*, France, 6 tomes, 1986-87.

[59] M.J. McCullagh. Terrain and surface modelling systems: theory and practice. *Photogrammetric Record*, 12(72), pages 747–779, octobre 1988.

[60] P. Desnogués. Triangulations et Quadriques. *Thèse de l'Université de Nice - Sophia Antipolis*, 202 pages, décembre 1996.

[61] O. Devillers. Further step for incremental randomized Delaunay triangulation. In *Proceedings des Journées Franco-Espagnoles de Géométrie Algorithmique*, Barcelone, pages 22–29, septembre 1997.
(Improved incremental randomized Delaunay triangulation. *Rapport de Recherche INRIA* N^0 *3298*, 21 pages, novembre 1997.)
(http://www.inria.fr/prisme/publis/dmt-fddtl-92.ps.gz)

[62] O. Devillers, S. Meiser, M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. In *Comput. Geom. Theory Appl.*, 2:2, pages 55–80, 1992.
(http://www.inria.fr/prisme/publis/dmt-fddtl-92.ps.gz)

[63] O. Devillers, M. Teillaud, M. Yvinec. Dynamic location in an arrangement of line segments in the plane. *Rapport de recherche* N^0 *1558*, INRIA Sophia Antipolis, 165 pages, 1991.

[64] L. Devroye. Lecture Notes on Bucket Algorithms. Ed. *Birkhäuser*, Progress in Computer Science 6, 146 pages, 1986.

[65] L. Devroye, E.P. Mücke, B. Zhu. A note on point location in Delaunay triangulations of random points. Submitted to *Information Processing Letters*, 7 pages, 1995.

[66] L. Devroye, B. Reed. On the variance of the height of random binary search trees. In *SIAM Journal on Computing*, 24, pages 1157–1162, 1995.

[67] H. Dörrie. Great problems of elementary mathematics. Ed. *Dover Publications*, New York, 1965.

[68] D. Dobkin, R.J. Lipton. Multidimensional searching problems. In *SIAM Journal of Computing*, 5, 2, pages 181–186, 1976.

[69] J. Driscoll, N. Sarnak, D. Sleator, R. Tarjan. Making data structures persistent. In *J. Comput. Syst. Sci.*, 38, pages 86–124, 1989.

[70] R.A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. In *Algorithmica*, 2, pages 137–151, 1987.

[71] R.A. Dwyer. Higher-Dimensional Voronoi Diagrams in Linear Expected Time. In *Discrete Comput Geom*, 6, pages 343–367, 1991.

[72] C.R. Dyer. The Space Efficiency of Quadtrees. In *Computer Graphics and Image Processing*, 19, pages 335–348, 1982.

[73] M. Edahiro, I. Kokubo, T. Asano. A new point location algorithm and its practical efficiency, comparison with existing algorithms. In *ACM Transactions on Graphics*, 3(2), pages 86–109, 1984.

[74] H. Edelsbrünner. Algorithms in Combinatorial Geometry. In *volume 10 of EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Heidelberg, West Germany, 1987.

[75] H. Edelsbrünner, L.J. Guibas, J. Stolfi. Optimal point location in a monotone subdivision. In *SIAM J. Comput*, 15(2), pages 317–340, 1986.

[76] H. Edelsbrünner, E.P. Mücke. Simulation of Simplicity : a Technique to Cope with Degenerate Cases in Geometric Algorithms. In *ACM Trans. Graph.*, 9, pages 66–104, 1990.

[77] M. Elbaz. Les diagrammes de Voronoi et de Delaunay dans le plan et dans l'espace. *Thèse de l'Université de Haute-Alsace*, N^0 209, 154 pages, 1992.

[78] T.P. Fang, L.A. Piegl. Delaunay Triangulation Using a Uniform Grid. In *IEEE Comput. Graph. Appl.*, 13(3), pages 36–48, Mai 1993.

[79] T.P. Fang, L.A. Piegl. Algorithm for constrained Delaunay triangulation. In *The Visual Computer*, 10, pages 255–265, 1994.

[80] R.A. Finkel, J.L. Bentley. Quad trees: a data structure for retrieval on composite keys. In *Acta Informatica*, vol. 4, 1, pages 1–9, 1974.

[81] L. De Floriani. Data structures for encoding triangulated irregular networks. In *Adv. Eng. Software*, 9(3), pages 122–128, 1987.

[82] L. De Floriani. Surface representations based on triangular grids. In *The Visual Computer*, 3, pages 27–50, 1987.

[83] L. De Floriani, B. Falcidieno, G. Nagy, C. Pienovi. On sorting triangles in a Delaunay Tessellation. In *Algorithmica*, 6, pages 522–532, 1991.

[84] L. De Floriani, B. Falcidieno, C. Pienovi. Structured graph representation of a hierarchical triangulation. In *Computer Vision, Graphics and Image Processing*, 45, pages 215–226, 1989.

[85] L. De Floriani, E. Puppo. A survey of constrained Delaunay triangulation algorithms for surface representation. In *Issues on Machine Vision*, édité par G.G.Pieroni, Springer Verlag, New York, pages 95–104, 1989.

[86] R.W. Floyd, R.L. Rivest. Expected time bounds for selection. In *Communications of the ACM*, 18(3), pages 165–172, mars 1975.

[87] S. Fortune. A Sweepline Algorithm for Voronoi Diagrams. In *Proc. of the 2nd ACM Symposium on Computational Geometry*, pages 313–322, 1986.

[88] S. Fortune. A Sweepline Algorithm for Voronoi Diagrams. In *Algorithmica*, 2, pages 153–174, 1987.

[89] J.H. Friedman, J.L. Bentley, R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. In *ACM Transactions on Mathematical Software*, 3, pages 209–226, septembre 1977.

[90] M.R. Garey, D.S. Johnson, F.P. Preparata, R.E. Tarjan. Triangulating a simple polygon. In *Information Processing Letters*, 7, pages 175–179, 1978.

[91] M. Gondran, M. Minoux. Graphes et algorithmes. Ed. *Eyrolles*, troisième édition, Paris, 588 pages, 1995.

[92] J.E. Goodman, J. O'Rourke (éditeurs). Handbook of Discrete and Computational Geometry. *CRC Press LLC*, Boca Raton, FL, 991 pages, juillet 1997.

[93] M. Goodrich, M. Orletsky, K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *Proc. 8th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, 1997.

[94] M. Goodrich, R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[95] M. Goossens, F. Mittelbach, A. Samarin. The LaTeX Companion. Ed. *Addison Wesley*, 530 pages, 1994.

[96] R.L. Graham, D.E. Knuth, O. Patashnik. Concrete Mathematics, a foundation for computer science. Ed. *Addison Wesley*, 657 pages, 1995.

[97] P.J. Green, R. Sibson. Computing Dirichlet tessellations in the plane. In *The Computer Journal*, 21, pages 168-173, 1978.

[98] D.H. Greene, F.F. Yao. Finite-resolution Computational Geometry. In *27th Annual Symposium of the Foundations of Computer Science*, pages 143-152, 1986.

[99] L.J. Guibas, R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. Foundations of Computer Science*, pages 8–21, 1978.

[100] L.J. Guibas, J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. In *ACM Transactions on Graphics*, 4, pages 74–123, 1985.

[101] L.J. Guibas, J. Stolfi. Ruler, compass, and computer : the design and analysis of geometric algorithms. *Technical Report*, 37, Digital Equipment Corporation systems Research Center, 1989.

[102] L.J. Guibas, D.E. Knuth, M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In *Algorithmica*, 7, pages 381–413, 1992.

[103] L.J. Guibas, D. Salesin, J. Stolfi. Epsilon Geometry : Building Robust Algorithms for Imprecise Computations. In *Proceedings of the First ACM Symposium on Computational Geometry*, pages 208–217, 1989.

[104] S. Hanke, T. Ottmann, S. Schuierer. The edge-flipping distance of triangulations. Rapport technique 76, Institut für Informatik, Université de Freiburg, Allemagne, février 1996 (In *European Workshop on Computational Geometry*, Münster, Allemagne, mars 1996.).

[105] E. Haines. Point in Polygon Strategies. In *Graphics Gems IV*, édité par Paul Heckbert, Academic Press, Boston, pages 24–46, 1994.

[106] J. Hervé. Triangulation avec boxsort. *Rapport interne*, SETRA, 92 Bagneux, France, 6 pages, 1993.

[107] P. Houthuys. Box Sort, a multidimensional binary sorting method for rectangular boxes, used for quick range searching. In *The Visual Computer*, 3, pages 236–249, 1987.

[108] Y.G. Huang. Modélisation et Manipulation des Surfaces Triangulées. *Thèse de L'Institut National Polytechnique de Lorraine*, N^0 390, 158 pages, 1990.

[109] F. Hurtado, M. Noy. The Graph of Triangulations of a Convex Polygon. *Rapport technique MA2-IR-94-13*, Université Polytechnique de Catalogne, Barcelone, 18 pages, juin 1995 (abstract In *Proc. of the 12th ACM Symp. on Comp. Geom.*, Philadelphie, C7-C8, 1996.).

[110] F. Hurtado, M. Noy, J. Urrutia. Flipping edges in triangulations. In *Proc. of the 12th ACM Symp. on Comp. Geom.*, Philadelphie, pages 214–223, 1996.

[111] F. Hurtado, M. Noy, J. Urrutia. Parallel edge flipping : combinatorial bounds. In *Proceedings des Journées Franco-Espagnoles de Géométrie Algorithmique*, Barcelone, pages 49–52, septembre 1997.

[112] P. Jaillon. Proposition d'une arithmétique rationnelle paresseuse et d'un outil d'aide à la saisie d'objets en synthèse d'images. *Thèse de l'Université de Saint-Etienne et de l'Ecole Nationale des Mines de Saint-Etienne*, Saint-Etienne, 184 pages, 1993.

[113] B. Joe, C.A. Wang. Duality of constrained Voronoi diagrams and Delaunay triangulations. In *Algorithmica*, 9, pages 142–155, 1993.

[114] M. Karasick. On the Representation and Manipulation of Rigid Solids. *Ph.D. Dissertation*, McGill University, 1988.

[115] M. Karasick, D. Lieber, L. Nackman. Efficient Delaunay Triangulation using Rational Arithmetic. *ACM Trans. Graph.*, 10, pages 71–91, 1991.

[116] P. Kauffmann, J.C. Spehner. Sur l'algorithme de Fortune. In *Revue Internationale de CFAO et d'informatique graphique*, 10(4), pages321–336, 1995.

[117] J. Katajainen, M. Koppainen. Constructing Delaunay triangulations by merging buckets in quadtree order. In *Annales Societatis mathematicae Polonae*, Series IV, Fundamenta Informaticae 11, pages 275–288, 1988.

[118] D.G. Kirkpatrick. Efficient computation of continuous skeletons. In Proceedings of the 20th Annual Symposium on Foundations of Computer Science, pages 18–27, 1979.

[119] D.G. Kirkpatrick. Optimal search in planar subdivisions. In SIAM Journal Comput., 12(1), pages 28–35, février 1983.

[120] D.E. Knuth. The Art of Computer Programming : Vol 2, Seminumerical Algorithms. Ed. *Addison Wesley*, Reading, Mass., 1969.

[121] D.E. Knuth. The Art of Computer Programming : Vol 3, Fundamental Algorithms. Ed. *Addison Wesley*, Reading, Mass., 1973.

[122] P. Koparkar. Surface intersection by switching from recursive subdivision to iterative refinement. In *The Visual Computer*, 8, pages 47–63, 1991.

[123] H. Kopka, P. Daly. A Guide to LaTeX2ϵ. Ed. *Addison Wesley*, 553 pages, 1995.

[124] M.J. van Kreveld, M.H. Overmars. Divided kd-trees. In *Algorithmica*, 6, pages 840–858, 1991.

[125] T. Lambert. Empty-Shape Triangulation Algorithms. *Ph. D. thesis*, Dept. Comput. Science, Univ. of Manitoba, Winnipeg, MB, août 1994.

[126] L. Lamport. LaTeX A Document Preparation System. Ed. *Addison Wesley*, 272 pages, 1994.

[127] C.L. Lawson. Generation of a Triangular Grid with Application to Contour Plotting. *Technical Report. Technical Memorandum 299*, California Institute Of Technology, 1972.

[128] C.L. Lawson. Software for C1 surface interpolation. In *Mathematical Software III*, Academic Press (ed : Rice J.R.), pages 161–164, 1977.

[129] G. Leach. Improving Worst-Case Optimal Delaunay Triangulation Algorithms. In *Proceedings of the 4th Canadian Conference on Computational Geometry*, St. John's, Newfoundland, pages 340–346, août 1992.

[130] D.T. Lee, F.P. Preparata. Location of a point in a planar subdivision and its applications. In *SIAM J. Comput.*, 6(3), pages 594–606, 1977.

[131] D.T. Lee, B.J. Schachter. Two algorithms for constructing a Delaunay triangulation. In *International Journal of Computer and Information Sciences*, 9, pages 219–242, 1980.

[132] C. Lemaire. Facétisation, hiérarchisation et localisation dans une représentation par segments d'un graphe planaire. *Mémoire de DEA*, Ecole des Mines de Saint-Etienne, septembre 1993.

[133] C. Lemaire, J.M. Moreau. Triangulation de Delaunay euclidienne dans le plan : optimisation du Divide and Conquer à l'aide d'un tri selon deux directions. *Rapport Technique 02D07669*, Bagneux, SETRA, 6 mars 1996. (présenté aux *Journées de Géométrie Algorithmique*, Le Bessat, 11-15 mars 1996).

[134] C. Lemaire, J.M. Moreau. Amélioration de la complexité en moyenne de l'algorithme de Lee et Schachter par division et fusion bi-directionnelles. In *4èmes Journées de l'Association Française d'Informatique Graphique*, Dijon, novembre 1996. (paru dans *Revue internationale de CFAO et d'informatique graphique*, 12, 4, Hermès, pages 317–336, septembre 1997.).

[135] C. Lemaire, J.M. Moreau. Analysis of a class of k-dimensional procedures, with an application to $2D$ Delaunay Triangulation in expected linear time after two-directional sorting. In *Proceedings of the 9th Canadian Conference on Computational Geometry*, Kingston, Ontario, août 1997.

[136] C. Lemaire, J.M. Moreau. Average-case analysis of a class of divide-and-conquer algorithms. In *Proceedings des Journées Franco-Espagnoles de Géométrie Algorithmique*, Barcelone, pages 70–77, septembre 1997.

[137] J.M. Lévy-Leblond. Archimède et la sphère à n dimensions ou le double saut de Π. In *Quadrature* 19, pages 7–12, 1994.

[138] R.J. Lipton, R.E. Tarjan. Applications of a planar separator theorem. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, New York, pages 162–170, 1977.

[139] R.J. Lipton, R.E. Tarjan. A separator theorem for planar graphs. In *SIAM J. Appl. Math.*, 36(2), pages 177–189, avril 1979.

[140] D.H. Mac Lain. Two-dimensional interpolation from random data. In *The Computer Journal*, 19(2), pages 178–191, 1976.

[141] J. Matoušek. Derandomization in computational geometry. In the web site *http://www.ms.mff.cuni.cz/acad/kam/matousek/*, à paraître dans [170], 1997.

[142] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. In *BIT* 24, pages 151–163, 1984.

[143] K. Mehlhorn. Data Structures and Efficient Algorithms 3 : Multi-dimensional Searching and Computational Geometry. Ed. *Springer-Verlag*, Berlin, Allemagne, 284 pages, 1984.

[144] D. Michelucci. Les représentations par les frontières : quelques constructions; difficultés rencontrées. *Thèse de l'Ecole Nationale des Mines de Saint-Etienne*, Saint-Etienne, 146 pages, 1987.

[145] D. Michelucci, J.M. Moreau. Lazy Arithmetic. In *IEEE Trans. on Computers*, vol. 46, 9, sept. 1997.

[146] V.J. Milenkovic. Verifiable Implementation of Geometric Algorithms Using Finite Precision Arithmetic. *Ph.D Dissertation*, CMU-CS-168, Carnegie-Mellon, 1988.

[147] V.J. Milenkovic. Double precision geometry : a general technique for calculating line and segment intersections using rounded arithmetic. In *30th Annual Symposium on the Foundations of Computer Science*, pages 500–505, 1989.

[148] V.J. Milenkovic. Robust Geometric Computations for Vision and Robotics. In *DARPA*, 1989.

[149] V.J. Milenkovic, Z.Li. Constructing Strongly Convex Hulls Using Exact or Rounded Arithmetic. In *ACM-SIAM Symposium on Discrete Algorithms*, 1989.

[150] J.S.B. Mitchell. Approximation Algorithms for Geometric Optimization Problems. In *Proceedings of the 9th Canadian Conference on Computational Geometry*, Kingston, Ontario, août 1997.

[151] J.M. Moreau. Hiérarchisation et facétisation de la représentation par segments d'un graphe planaire dans le cadre d'une arithmétique mixte. *Thèse de l'Ecole Nationale des Mines de Saint-Etienne*, Saint-Etienne, 243 pages, 1990.

[152] J.M. Moreau, P. Volino. Constrained Delaunay triangulation revisited. In *Proceedings of the Fifth Canadian Conference of Computational Geometry*, Waterloo, Ontario, Canada, pages 340–345, août 1993.

[153] J.M. Moreau. Hierarchical Delaunay triangulation. In *Proceedings of the Sixth Canadian Conference of Computational Geometry*, Saskatoon, Canada, pages 165–170, août 1994.

[154] E.P. Mücke, I. Saias, B. Zhu. Fast Randomized Point Location Without Preprocessing in Two- and Three-dimensional Delaunay Triangulations. In *Proceedings of the 12th ACM Symposium on Computational Geometry*, Philadelphie, 20 pages, 1996.

[155] K. Mulmuley. Randomized multidimensional search trees: Dynamic sampling. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry*, pages 121–131, 1991.

[156] K. Mulmuley. Computational Geometry : An Introduction through Randomized Algorithms. Ed. *Prentice Hall*, New York, 447 pages, 1984.

[157] F. Nielsen. Algorithmes géométriques adaptatifs. *PhD Thesis*, Ecole Normale Supérieure de Lyon, 1996.

[158] T. Ohya, M. Iri, K. Murota. Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms. In *Journal of the Operations Research Society of Japan*, 27, pages 306–337, 1984.

[159] A. Okabe, B. Boots, K. Sugihara. Spatial Tessellations : Concepts and Applications of Voronoi Diagrams. Ed. *John Wiley*, Chichester, Angleterre, 532 pages, 1992.

[160] J. O'Rourke. Computational Geometry in C. Ed. *Cambridge University Press*, 346 pages, 1994.

[161] T. Ottman, G. Thiemt, C. Ullrich. Numerical Stability of Geometric Algorithms. In *Proceedings of the Third ACM Symposium on Computational Geometry*, pages 119–125, 1987.

[162] M. Overmars. The Design of Dynamic Data Structures. *Lecture Notes in Computer Science*, Vol. 156, Springer-Verlag, 181 pages, 1983.

[163] F.P. Preparata. A new approach to planar point location. Ed. *SIAM Journal of Computing*, 10(3), pages 473–483, 1981.

[164] F.P. Preparata. Planar point location revisited. Ed. *Internat. J. Found. Comput. Sci.*, 1, pages 71–86, 1990.

[165] F.P. Preparata, M.I. Shamos. Computational Geometry - an introduction. Ed. *Springer Verlag, New-York*, 390 pages, 1985.

[166] F.P. Preparata, R. Tamassia. Dynamic planar point location with optimal query time. Ed. *Theoret. Comput. Sci.*, 74, pages 95–114, 1990.

[167] F.P. Preparata, R. Tamassia. Efficient point location in a convex spatial cell complex. Ed. *SIAM Journal of Computing*, 21, pages 267–280, 1992.

[168] E.A. Ramos. Lecture on Geometric Divide-and-Conquer Algorithms based on Geometric Sampling. *Cours dispensé au LORIA-CRIN de Nancy par le Max-Plank-Institut für Informatik*, Saarbrücken, 12 pages, 3 avril 1997.

[169] J.C. Roux. Méthodes d'approximation et de géométrie algorithmique pour la reconstruction de courbes et de surfaces. *Thèse de l'Université Joseph Fourier, Grenoble 1*, 195 pages, 1994.

[170] J.R. Sack, J. Urrutia. Handbook on Computational Geometry. *North Holland*, à paraître en 1997.

[171] H. Samet. The Design and Analysis of Spatial Data Structures. Ed. *Addison Wesley*, Reading, Massachussets, 493 pages, 1990.

[172] N. Sarnak, R.E. Tarjan. Planar point location using persistent search trees. In *Communications ACM*, 29, pages 669–679, 1986.

[173] D. Schmitt. Sur les diagrammes de Voronoi et de Delaunay d'ordre k dans le plan et dans l'espace. *Thèse de l'Université de Haute-Alsace*, 278 pages, 1995.

[174] A.A. Schoone, J. van Leeuwen. Triangulating a star-shaped polygon. In *Technical Report RUU-CS-80-3*, université d'Utrecht, Netherlands, avril 1980.

[175] R. Seidel. Constrained Hierarchical Delaunay Triangulation and Voronoi Diagrams with obstacles. In *Rep 260*, IIG-TU, Graz, Austria, pages 178–191, 1988.

[176] R. Seidel. A simple fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. In *Comput. Geom. Theory Appl.*, 1, pages 51–64, 1991.

[177] R. Seidel. Planar point location and randomization. *Cours dispensé au LORIA-CRIN de Nancy par le Max-Plank-Institut für Informatik*, Saarbrücken, 13 mars 1997.

[178] R. Sedgewick. Algorithmes en langage C (traduit par J. M. Moreau). Ed. *InterEditions*, Paris, France, 685 pages, 1991.

[179] M. Segual, C.H. Séquin. Consistent Calculations for solids modeling. In *Proceedings of the First ACM Symposium on Computational Geometry*, pages 29–38, 1985.

[180] M.I. Shamos. Computational Geometry. *Ph.D. thesis, Dept.of Comput. Sci., Yale Univ.*, 1978.

[181] M.I. Shamos, D. Hoey. Closest Point Problems. In *Proc. 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[182] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *First Workshop on Applied Computational Geometry*, ACM, pages 124–133, mai 1996.

[183] J. R. Shewchuk. Robust adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, ACM, pages 141–150, mai 1996.

[184] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Rapport Technique CMU-CS-96-140*, School of Computer Science, Carnegie Mellon University, Pittsburgh, 53 pages, mai 1996.

[185] C. Simon, P. Brehmer. Triangulation de Delaunay. *Communication Personnelle*, SETRA/CITS/ATA N^0 B518, Bagneux, 4 pages, novembre 1996.

[186] D. Stoyan, W.S. Kendall, J. Mecke. Stochastic Geometry and Its Applications. Ed. *John Wiley*, 345 pages, 1989.

[187] P. Su, R.L.S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 61-70, 1995.

[188] P. Su, R.L.S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Computational Geometry, Theory and Applications*, 7, pages 361-385, avril 1997.

[189] K. Sugihara, M. Iri. Two Design Principles of Geometric Algorithms in Finite-precision Arithmetic. In *Appl. math. Letter.*, 2:2, pages 203-206, 1989.

[190] K. Sugihara, M. Iri. Construction of the Voronoi Diagram for One Million Generators in Single-Precision Arithmetic. In *Proceedings of the First Canadian Conference on Computational Geometry*, 1989.

[191] K. Sugihara, M. Iri. A solid modelling system free from topological inconsistency. In *J. Inform. Proc.*, 12:4, pages 380–393, 1989.

[192] M. Teillaud. Vers des algorithmes randomisés dynamiques en géométrie algorithmique. *Thèse de doctorat en sciences*, Université Paris-Sud, Orsay, France, 176 pages, 1991.
(http://www.inria.fr/prisme/publis/these-teillaud.ps.gz)
– Version anglaise : Towards dynamic randomized algorithms in computational geometry. *Rapport de recherche INRIA N^0 1727*, 1992.
(http://www.inria.fr/RRRT/RR-1727.html)
– Version livre : Towards dynamic algorithms in computational geometry. In *Lecture Notes in Computer Science*, 758, Springer-Verlag, 1993.

[193] P. Volino. Triangulation de Delaunay contrainte : étude et implantation d'algorithmes. *Mémoire de DEA*, Ecole des Mines de Saint-Etienne, juillet 1992.

[194] C.A. Wang. An optimal algorithm for greedy triangulation of a set of points. In *Proceedings of the Sixth Canadian Conference of Computational Geometry*, Saskatoon, Canada, pages 332–338, août 1994.

[195] D.F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. In *The Computer Journal*, 24, pages 167–172, 1981.

[196] C.K. Yap. A Geometric Consistency Theorem for a Symbolic Perturbation Scheme. In *J. Comput. Syst. Sci.*, 40, pages 2–18, 1990.

[197] C.K. Yap. Symbolic Treatment of Geometric Degeneracies. In *J. Symbolic. Comput.*, 10, pages 349–370, 1990.

Triangulation de Delaunay et arbres multidimensionnels

Résumé : Les travaux effectués lors de cette thèse concernent principalement la triangulation de Delaunay. On montre que la complexité en moyenne – en termes de sites inachevés – du processus de fusion multidimensionnelle dans l'hypothèse de distribution quasi-uniforme dans un hypercube est linéaire en moyenne. Ce résultat général est appliqué au cas du plan et permet d'analyser de nouveaux algorithmes de triangulation de Delaunay plus performants que ceux connus à ce jour. Le principe sous-jacent est de diviser le domaine selon des arbres bidimensionnels (quadtree, 2d-tree, bucket-tree...), puis de fusionner les cellules obtenues selon deux directions. On étudie actuellement la prise en compte de contraintes directement pendant la phase de triangulation avec des algorithmes de ce type. De nouveaux algorithmes pratiques de localisation dans une triangulation sont proposés, basés sur la randomisation à partir d'un arbre binaire de recherche dynamique de type AVL, dont l'un est plus rapide que l'algorithme optimal de Kirkpatrick, au moins jusqu'à 12 millions de sites ! Nous travaillons actuellement sur l'analyse rigoureuse de leur complexité en moyenne. Ce nouvel algorithme est utilisé pour construire "en-ligne" une triangulation de Delaunay qui est parmi les plus performantes des méthodes "en-ligne" connues à ce jour.

Mots-clés : géométrie algorithmique, triangulation de Delaunay, diagramme de Voronoi, arbre multidimensionnel, Divide-and-Conquer, kd-tree, quadtree, bucket, complexité en moyenne, complexité dans le pire des cas, site inachevé, localisation, algorithme dynamique, surface triangulée.

Delaunay triangulation and multidimensional trees

Abstract : This thesis deals mainly with Delaunay triangulations. It is shown that the average complexity of k-dimensional merge processes – according to unfinished sites – is linear, assuming the sites to be quasy-uniformly distributed in an hypercube. This general result is applied to the two-dimensional case, and allows to analyze new Delaunay triangulation algorithms which perform better than those known to this day. The underlying principle is to divide the domain according to k-dimensional trees (quadtree, 2d-tree, bucket-tree...), and then to merge the obtained cells along two directions. We are currently trying to generalize these algorithms to the constrained case (graphs with constraints, and not only vertices). We propose new point location algorithms, based upon randomisation on a dynamic binary search tree AVL. One of them is faster than Kirkpatrick's optimal algorithm at least up to 12 millions of sites. Their formal average analysis is being done. We use this algorithm to build Delaunay triangulations "on-line" which is one of the most performant known "on-line" method.

Keywords : computational geometry, Delaunay triangulation, Voronoi diagram, multi-dimensional tree, Divide-and-Conquer, kd-tree, quadtree, bucket, average-case complexity, worst-case complexity, unfinished site, point location, intersection, triangulated surface.