Υλοποίηση και Οπτικοποίηση Αλγορίθμου Υπολογισμού του Τριγωνισμού Delaunay

Δημήτριος Σαμουρέλης

Διπλωματική Εργασία

Επιβλέπων: Λεωνίδας Παληός

Ιωάννινα, Φεβρουάριος 2020

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Ευχαριστίες

Με την περάτωση της παρούσας Διπλωματικής εργασίας θα ήθελα να ευχαριστήσω θερμά τον Καθηγητή του Τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής κ. Λεωνίδα Παληό για την εμπιστοσύνη που μου έδειξε στην εκπόνηση της παρούσας διπλωματικής εργασίας.

Ιδιαίτερα θερμές ευχαριστίες θέλω να δώσω στην οικογένεια μου για την αμέριστη συμπαράσταση τους, για τις πολύτιμές συμβουλές τους και για όλα όσα μου έχουν προσφέρει όλα αυτά τα χρόνια της ζωής μου αλλά και των σπουδών μου.

Φεβρουάριος 2020

Δημήτριος Σαμουρέλης

Περίληψη

Έστω P ένα σύνολο n σημείων στο επίπεδο, τα οποία θα αναφέρονται στη συνέχεια ως εστίες. Το διάγραμμα Voronoi του P είναι η υποδιαίρεση του επιπέδου σε n περιοχές, μία για κάθε εστία του P, για την οποία ισχύει ότι η περιοχή κάθε εστίας p ∈ P περιέχει όλα τα σημεία του επιπέδου που έχουν ως πλησιέστερη εστία το p. Δοθέντος ενός διαγράμματος Voronoi Vor(P) ενός συνόλου εστιών P το δυϊκό γράφημα G περιέχει έναν κόμβο για κάθε περιοχή Voronoi ισοδύναμα, για κάθε εστία και μία ακμή μεταξύ δύο κόμβων αν οι αντίστοιχες περιοχές έχουν μια κοινή ακμή. Αυτό συνεπάγεται ότι το G έχει μία ακμή για κάθε ακμή του Vor(P). Το γράφημα Delaunay του P είναι μια ευθύγραμμη εναποτύπωση του δυϊκού γραφήματος του διαγράμματος Voronoi , όπου ο κόμβος που αντιστοιχεί στην περιοχή Voronoi V(p) είναι το σημείο P, και η ακμή που συνδέει τους κόμβους των περιοχών V(p) και V(q) είναι το ευθύγραμμο τμήμα pq. Ορίζουμε λοιπόν ως τριγωνισμό Delaunay οποιονδήποτε τριγωνισμό ο οποίος προκύπτει από το γράφημα Delaunay με την προσθήκη ακμών. Ενδεικτικά, τριγωνισμός Delaunay εφαρμόζεται στην αναπαράσταση 3Δ Χαρτών (Terrain)[3], στον υπολογισμό του Ευκλείδειου ελάχιστου γενετικού δέντρου(MST). Τέλος χρησιμοποιείται σε εφαρμογές της μεθόδου πεπερασμένων στοιχείων.

Σκοπός της συγκεκριμένης διπλωματικής εργασίας είναι η μελέτη, η υλοποίηση και η οπτικοποίηση του αλγορίθμου των Guibas, Knuth και Sharir για τον υπολογισμό του τριγωνισμού ενός συνόλου σημείων στο επίπεδο. Ο αλγόριθμος είναι τυχαιοκρατικός και αυξητικός. Ο τριγωνισμός Delaunay ενός συνόλου σημείων στο επίπεδο μπορεί να υπολογιστεί σε αναμενόμενο χρόνο O(nlogn) και σε αναμενόμενο χώρο O(n). Η υλοποίηση του αλγορίθμου πραγματοποιείται στη γλώσσα python. Αναλυτικότερα, ο αλγόριθμος δέχεται ως είσοδο αρχεία με τυχαία σημεία του επιπέδου και παράγει ως έξοδο του αλγορίθμου εικόνες με τα ενδιάμεσα βήματα εκτέλεσης του αλγορίθμου. Επιπλέον, παράγει ως έξοδο εικόνες για την απεικόνιση της δομής δέντρου για την απεικόνιση των ενδιάμεσων τριγώνων.

Λέξεις Κλειδιά: Τριγωνισμός, Python,Τυχαιοκρατικός, Αυξητικός, Delaunay

Abstract

Let P be a set of n points in the plane, which will then be referred to as sites. The Voronoi diagram of P is the division of the plane into n regions, one for each point of P, for which the area of each point p ∈ P contains all points of the plane having the nearest point as p. Given a Voronoi diagram Vor(P) of a set of sites P the binary graph G contains a node for each Voronoi region equivalent, for each site and an edge between two nodes if the corresponding regions have a common edge. This implies that G has one edge for each edge of Vor(P). The Delaunay graph of P is a straightforward depiction of the binary graph of the Voronoi diagram, where the node corresponding to the Voronoi region V(p) is the point P, and the edge connecting the nodes of the regions V(p) and V(q) is the linear segment pq. So, we define as a Delaunay triangle any triangle that results from the Delaunay line with the addition of edges. Indicatively, the Delaunay triangulation is applied to the Terrain representation [3], in the calculation of the Euclidean minimum genetic tree (MST). Finally, it is used in applications of the finite element method.

The purpose of this thesis is to study, implement and visualize the Guibas, Knuth and Sharir algorithm to calculate the triangulation of a set of points on a plane. The algorithm is random and incremental. The Delaunay triangulation of a set of points in the plane can be calculated at expected time O(nlogn) and at expected space O(n). The algorithm is implemented in python language. More precisely, the algorithm accepts files with random points in the plane as input and produces images with the intermediate steps of executing the algorithm as output. In addition, it outputs images to illustrate the tree structure for illustrating the intermediate triangles.

Keywords: Triangulation, Python, Randomized, Incremental, Delaunay

Περιεχόμενα

Κεφάλαιο 1. Εισαγωγή

1.1 Βασικοί Ορισμοί

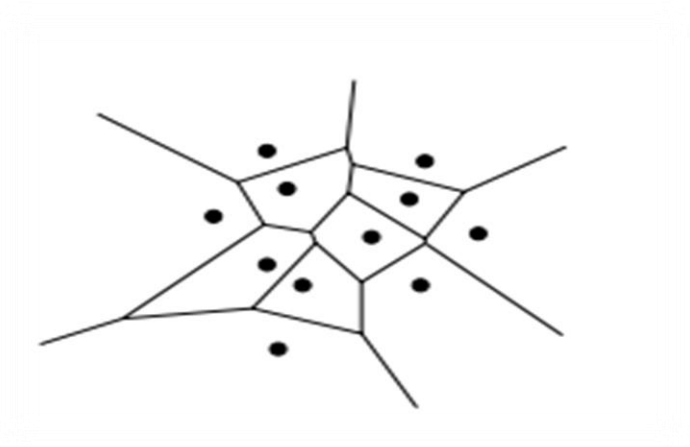
Στην ενότητα αυτή θα δώσουμε τους βασικούς ορισμούς που είναι απαραίτητοι για τον προσδιορισμό του αλγόριθμου και που θα χρειαστούμε στην συνέχεια.

1.1.1 Διαγράμματα Voronoi

Σε αυτή την υποενότητα θα δώσουμε τον ορισμό του διαγράμματος Voronoi και του Τριγωνισμού Delaunay και θα περιγράψουμε χρήσιμες γεωμετρικές ιδιότητες που προκύπτουν από αυτούς.

1.1.1.1 Ορισμός Διαγράμματος Voronoi

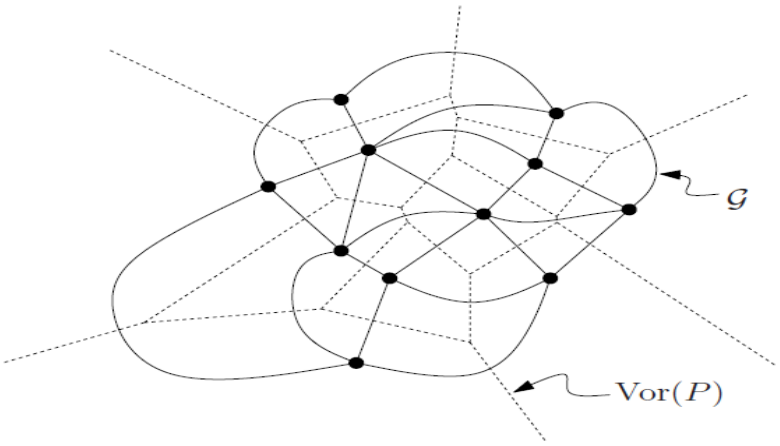
Έστω P ένα σύνολο n σημείων στο επίπεδο, τα οποία θα αναφέρονται στη συνέχεια ως εστίες. Το διάγραμμα Voronoi του P είναι η υποδιαίρεση του επιπέδου σε n περιοχές, μία για κάθε εστία του P, για την οποία ισχύει ότι η περιοχή κάθε εστίας $p \in P$ περιέχει όλα τα σημεία του επιπέδου που έχουν ως πλησιέστερη εστία το p. Το διάγραμμα Voronoi του P συμβολίζεται με Vor(P). Η περιοχή μιας εστίας p λέγεται περιοχή Voronoi του p και συμβολίζεται V(p). Το διάγραμμα Voronoi ενός συνόλου 11 εστιών φαίνεται στο Σχήμα 1.1.



Σχήμα 1.1: Διάγραμμα Voronoi n=11 σημείων [2].

1.1.1.2 Το δυϊκό γράφημα του διαγράμματος Voronoi

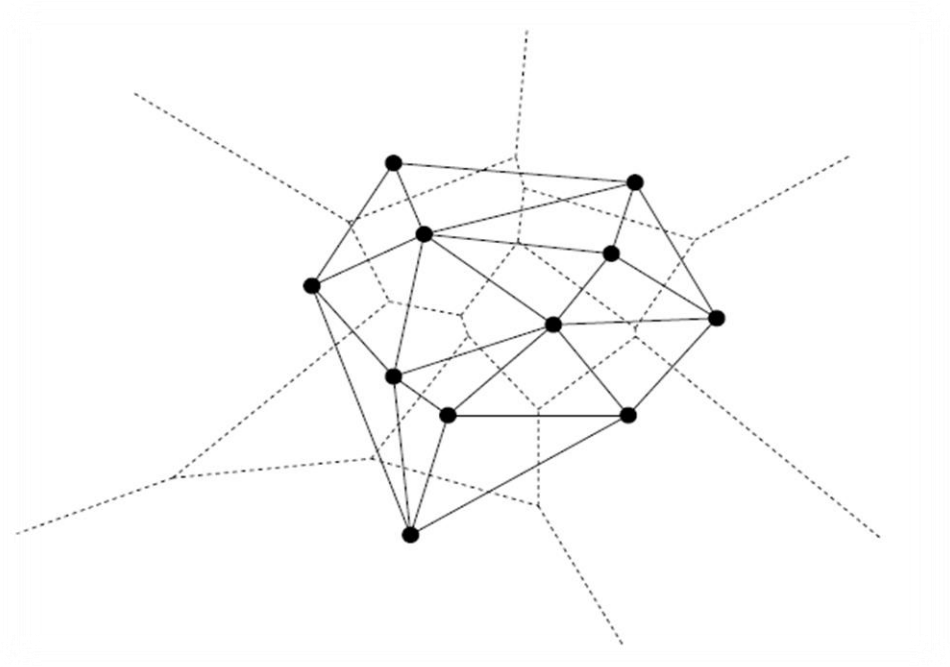
Δοθέντος ενός διαγράμματος Voronoi Vor(P) ενός συνόλου εστιών P το δυϊκό γράφημα G περιέχει έναν κόμβο για κάθε περιοχή Voronoi ισοδύναμα, για κάθε εστία και μία ακμή μεταξύ δύο κόμβων αν οι αντίστοιχες περιοχές έχουν μια κοινή ακμή. Αυτό συνεπάγεται ότι το *G* έχει μία ακμή για κάθε ακμή του Vor(P). Όπως φαίνεται στο Σχήμα 1.2, υπάρχει μια ένα προς ένα αντιστοιχία ανάμεσα στις φραγμένες έδρες του *G* και στις κορυφές του Vor(P).



Σχήμα 1.2: Το δυϊκό γράφημα του Vor(*P*) [2].

1.1.1.3 Το γράφημα Delaunay GD(P)

Το γράφημα Delaunay του P είναι μια ευθύγραμμη εναποτύπωση του δυϊκού γραφήματος του διαγράμματος Voronoi , όπου ο κόμβος που αντιστοιχεί στην περιοχή Voronoi V(p) είναι το σημείο P και η ακμή που συνδέει τους κόμβους των περιοχών V(p) και V(q) είναι το ευθύγραμμο τμήμα pq. Το γράφημα Delaunay ενός σημειοσυνόλου στο επίπεδο είναι επίπεδο γράφημα όπως εξηγείται στο Θεώρημα 2 ([2], Θεώρημα 9.5). Στο Σχήμα 1.3 αναπαρίσταται το γράφημα Delaunay GD(P).



Σχήμα 1.3: Το γράφημα Delaunay GD(P) [2].

1.1.2 Delaunay Τριγωνισμός

Όπως είδαμε παραπάνω το γράφημα Delaunay του P είναι μια εναποτύπωση του δυϊκού γραφήματος του διαγράμματος Voronoi το οποίο έχει μία έδρα για κάθε κορυφή του Vor(P). Οι ακμές στο σύνορο μιας έδρας αντιστοιχούν στις ακμές Voronoi που προσπίπτουν στην αντίστοιχη κορυφή Voronoi. Συγκεκριμένα, αν μια κορυφή v του Vor(P) αποτελεί κορυφή των περιοχών Voronoi για τις εστίες $p_1, p_2, p_3, \ldots, pk$, τότε η αντίστοιχη έδρα f του GD(P) έχει ως κορυφές της τις $p_1, p_2, p_3, \ldots, p_k$.

Τα σημεία του P είναι τυχαία κατανεμημένα και θεωρούμε ότι ένα σύνολο σημείων βρίσκεται σε γενική θέση αν δεν περιέχει καμιά τετράδα σημείων επάνω σε κύκλο. Αν το P βρίσκεται σε γενική θέση, τότε όλες οι κορυφές του διαγράμματος Voronoi έχουν βαθμό τρία, και συνεπώς όλες οι φραγμένες έδρες του GD(P) είναι τρίγωνα. Αυτός είναι ο λόγος που το GD(P) αποκαλείται συχνά και τριγωνισμός Delaunay του P.

Ορίζουμε λοιπόν ως τριγωνισμό Delaunay οποιονδήποτε τριγωνισμό ο οποίος προκύπτει από το γράφημα Delaunay με την προσθήκη ακμών. Λαμβάνοντας υπόψη ότι όλες οι έδρες του GD(P) είναι κυρτές, η κατασκευή ενός τέτοιου τριγωνισμού είναι εφικτή. Παρατηρούμε ότι ο τριγωνισμός Delaunay του P είναι μοναδικός αν και μόνο αν το GD(P) αποτελεί τριγωνισμό, το οποίο ισχύει αν το P βρίσκεται σε γενική θέση.

1.2 Αντικείμενο της Διπλωματικής Εργασίας

Αντικείμενο της συγκεκριμένης διπλωματικής εργασίας ήταν η μελέτη του αλγορίθμου των Guibas , Knuth και Sharir , η υλοποίηση και οπτικοποίηση του. Υπολογίσαμε έναν τριγωνισμό Delaunay άμεσα, χρησιμοποιώντας την τυχαιοκρατική αυξητική τεχνική λαμβάνοντας ως είσοδο σημεία στο επίπεδο τα οποία δημιουργήσαμε τυχαία. Αυτά τα τυχαία σημεία διαβάζονται από αρχείο. Στην συνέχεια τα επεξεργαζόμαστε ακολουθώντας την υλοποίηση του αλγορίθμου των Guibas , Knuth και Sharir. Τέλος οπτικοποιούμε τα αποτέλεσμα των τελικών τριγωνισμών παράγοντας αρχεία εξόδου και πιο συγκεκριμένα εικόνες με τους τελικούς τριγωνισμούς , εικόνες με τα ενδιάμεσα βήματα της εκτέλεσης του αλγορίθμου και εικόνες για την απεικόνιση του δέντρου που δείχνει την εξέλιξη της δημιουργίας τριγώνων στον αλγόριθμο.

1.3 Σχετικά Ερευνητικά Αποτελέσματα

Αλγόριθμοί τοπικών Μετασχηματισμών (Local Improvement)

Η στρατηγική τοπικού μετασχηματισμού χρησιμοποιείται κυρίως σε 2 διαστάσεων χώρους. Πρώτον, ένας γενικός τριγωνισμός δημιουργείται. Στο δεύτερο στάδιο, αυτός ο τριγωνισμός, διαδοχικά, μετατρέπεται σε τριγωνισμό Delaunay μέσω της εφαρμογής ορισμένων τοπικών μετασχηματισμών. Ο τριγωνισμός Delaunay είναι ένας τέτοιος τριγωνισμός όπου όλες οι ακμές του είναι τοπικά βέλτιστες. Η ακμή e είναι τοπικά βέλτιστη αν και μόνο αν το πολύγωνο P, που σχηματίζεται από δύο τρίγωνα που μοιράζονται αυτή την ακμή δεν είναι κυρτό, ή ισοδύναμα ο περιγεγραμμένος κύκλος ενός από αυτά τα δύο τρίγωνα δεν περιέχει το απομακρυσμένο σημείο του δεύτερου τριγώνου. Εάν η ακμή e δεν είναι βέλτιστη αφαιρείτε από τον τριγωνισμό και μια δεύτερη ακμή e’ εισάγετε σε αυτήν. Στη συνέχεια, είναι απαραίτητο να ελεγχθούν οι τέσσερις άκρες του πολυγώνου P εάν είναι βέλτιστες.

Αλγόριθμοι αυξητικής δημιουργίας (Incremental Construction)

Μια προσέγγιση του αυξητικού αλγορίθμου βασίζετε στην τακτική της σκούπας (Sweeping). Ο αλγόριθμος Sweeping από τον Fortune [4] είναι πολύ γνωστός για την επίλυση του Delaunay τριγωνισμού στις δύο διαστάσεις. Στην χειρότερη περίπτωση η πολυπλοκότητα του αλγορίθμου είναι O(NlogN). Ο αλγόριθμος επεκτείνεται και στις

τρεις διαστάσεις αλλά στην πράξη δεν χρησιμοποιείται εξαιτίας της μεγάλης προγραμματιστικής πολυπλοκότητας.

Αλγόριθμοι αυξητικής προσθήκης

Μία μέθοδος η οποία προτάθηκε από τον Watson DF το 1981 και είναι γνωστή ως Bowyer- Watson [18]. Ο αρχικός αλγόριθμος χρειαζόταν O(N(2d−1)/d) χρόνο στην χειρότερη περίπτωση, όπου d η διάσταση του χώρου σημείων που εφαρμόζεται ο τριγωνισμός. Ο αλγόριθμος εντόπιζε όλα τα τρίγωνα που είχαν στον περιγεγραμμένο κύκλο τους το σημείο που είναι για εισαγωγή και τα αφαιρούσε από τον τριγωνισμό.

Η θεωρητική ανάλυση του Lawson [5] απέδειξε ότι οποιοιδήποτε δύο τριγωνισμοί ενός επίπεδου σημειοσυνόλου μπορούν να μετασχηματιστούν ο ένας στον άλλον μέσω μεταστροφών ακμών. Έπειτα ο ίδιος πρότεινε την εύρεση ενός καλού τριγωνισμού μέσω επαναληπτικών μεταστροφών ακμών, όπου κάθε τέτοια μεταστροφή βελτιώνει κάποια συνάρτηση κόστους του τριγωνισμού [6]. Για αρκετό διάστημα ήταν γνωστό εμπειρικά ότι οι τριγωνισμοί που οδηγούν σε καλές παρεμβολές αποφεύγουν τα μακρόστενα τρίγωνα [7]. Ο Sibson [8] επισημαίνει ότι υπάρχει μόνο ένας τοπικά βέλτιστος τριγωνισμός όσον αφορά το γωνιοδιάνυσμα υπάρχει μόνο ένας τοπικά βέλτιστος τριγωνισμός όσον αφορά το γωνιοδιάνυσμα, ο τριγωνισμός Delaunay, αν εξαιρέσουμε τις εκφυλισμένες περιπτώσεις. Λαμβάνοντας υπ' όψιν μόνο το γωνιοδιάνυσμα, αγνοούμε εντελώς το ύψος των δειγματικών σημείων, οπότε ακολουθούμε τη λεγόμενη προσέγγιση της ανεξαρτησίας από τα δεδομένα.

Ο Rippa [9] απέδειξε ότι ο τριγωνισμός Delaunay είναι ο τριγωνισμός που ελαχιστοποιεί την τραχύτητα του προκύπτοντος αναγλύφου, ανεξάρτητα από τα δεδομένα του ύψους. Σε πιο πρόσφατες εργασίες άλλοι ερευνητές έχουν προσπαθήσει να βρουν βελτιωμένους τριγωνισμούς λαμβάνοντας υπ' όψιν τις πληροφορίες για τα ύψη. Dyn et al. [10] πρότειναν για πρώτη φορά την προσέγγιση της εξάρτησης από τα δεδομένα οι οποίοι προτείνουν διαφορετικά κριτήρια κόστους για τριγωνισμούς, που εξαρτώνται από τα ύψη των δειγματικών σημείων.

Αξίζει να σημειωθεί ότι για τους βελτιωμένους τριγωνισμούς τους ξεκινούν με αφετηρία τον τριγωνισμό Delaunay και εκτελούν επαναληπτικά μεταστροφές ακμών. Οι Quak και Schumaker [11] αλλά και ο Brown[12] ακολουθούν την ίδια οι οποίοι μελετούν την τμηματικά κυβική παρεμβολή. Πιο συγκεκριμένα οι Quak και Schumaker παρατηρούν ότι οι τριγωνισμοί τους αποτελούν μικρές βελτιώσεις συγκριτικά με τον τριγωνισμό Delaunay όταν προσπαθούν να προσεγγίσουν ομαλές επιφάνειες, αλλά ενδέχεται να είναι δραστικά διαφορετικοί για μη ομαλές επιφάνειες. Το σκεπτικό της επέκτασης της

ανάλυσης για την περίπτωση σημείων σε εκφυλισμένη θέση είναι πρωτότυπο. Εναλλακτικοί τυχαιοκρατικοί αλγόριθμοι έχουν αναπτυχθεί από τους Clarkson και Shor [13].

Διάφορα γεωμετρικά γραφήματα που ορίζονται για κάποιο σύνολο σημείων P αποτελούν υπογραφήματα του τριγωνισμού Delaunay του P. Το σημαντικότερο από αυτά είναι μάλλον το ευκλείδειο ελαφρύτατο συνδετικό δέντρο του συνόλου των σημείων [14]. Άλλα τέτοια γραφήματα είναι το γράφημα Gabriel [15] και το γράφημα σχετικής γειτονίας [16].

Ένας άλλος σημαντικός τριγωνισμός είναι ο ελαχιστοβαρής τριγωνισμός, δηλαδή ένας τριγωνισμός του οποίου το βάρος είναι ελαχιστιαίο, όπου ως βάρος ενός τριγωνισμού θεωρείται το άθροισμα των μηκών όλων των ακμών του. Αποδείχθηκε πρόσφατα [17], ότι ο προσδιορισμός ενός ελαχιστοβαρούς τριγωνισμού μεταξύ όλων των τριγωνισμών ενός δεδομένου σημειοσυνόλου αποτελεί NP-πλήρες πρόβλημα.

1.4 Δομή της Διπλωματικής Εργασίας

Η συγκεκριμένη διπλωματική εργασία αποτελείται από 4 κεφάλαια τα οποία αναλύονται στις παρακάτω παραγράφους.

Στο Κεφάλαιο 2 γίνεται μία αναλυτική περιγραφή του αλγορίθμου των Guibas , Knuth και Sharir. Πιο συγκεκριμένα στο πρώτο μισό του Κεφαλαίου 2 γίνεται μία θεωρητική εισαγωγή στον αλγόριθμο και παρουσιάζονται χρήσιμες παρατηρήσεις και αποτελέσματα σχετικά με την λειτουργία του αλγορίθμου. Έπειτα παρουσιάζεται η πολυπλοκότητα του αλγορίθμου, μία γενική περιγραφή της διπλά συνδεδεμένης λίστας ακμών και τέλος ένα παράδειγμα εφαρμογής του αλγορίθμου.

Στο Κεφάλαιο 3 περιγράφονται οι λεπτομέρειες υλοποίησης του λογισμικού. Περιγράφεται η μορφή των αρχείων εισόδου και εξόδου του αλγορίθμου, καθώς και οι βασικές δομές δεδομένων που είναι απαραίτητες για την λειτουργία του αλγορίθμου. Επίσης, παρουσιάζονται οι βασικότερες κλάσεις του κώδικα η οποίες συνοδεύονται με μία μικρή περιγραφή της υλοποίησης τους. Τέλος, παρουσιάζονται κάποια παραδείγματα εκτέλεσης του αλγόριθμου για διαφορετικά δεδομένα εισόδου.

Τέλος, στο Κεφάλαιο 4 παρουσιάζονται τα συμπεράσματα του αλγορίθμου και περιγράφονται οι μελλοντικές επεκτάσεις αυτού.

Κεφάλαιο 2. Ο Αλγόριθμος των Guibas, Knuth και Sharir

2.1 Θεωρητικό Υπόβαθρο

Σε αυτήν την ενότητα παρουσιάζονται χρήσιμα συμπεράσματα και παρατηρήσεις για τον αλγόριθμο.

Θεώρημα 1 ([2], 9.6) Έστω P ένα σύνολο σημείων στο επίπεδο. (i) Τρία σημεία p_i, p_j, p_k ∈ P αποτελούν κορυφές της ίδιας έδρας του γραφήματος Delaunay του P αν και μόνο αν ο κύκλος που διέρχεται από τα p_i, p_j, p_k δεν περιέχει στο εσωτερικό του κανένα σημείο του P. (ii) Δύο σημεία p_i, p_j ∈ P σχηματίζουν ακμή του γραφήματος Delaunay του P αν και μόνο αν υπάρχει κλειστός δίσκος C που περιέχει τα p_i και p_j στο σύνορό του και δεν περιέχει κανένα άλλο σημείο του P.

Θεώρημα 2 ([2], 9.5) Το γράφημα Delaunay ενός σημειοσυνόλου στο επίπεδο είναι επίπεδο γράφημα.

Θεώρημα 3 ([2], 9.7) Έστω P ένα σύνολο σημείων στο επίπεδο και T ένας τριγωνισμός του P. Ο T είναι ένας τριγωνισμός Delaunay του P αν και μόνο αν κανένα σημείο του P δεν κείται στο εσωτερικό του περιγεγραμμένου κύκλου κάποιου τριγώνου του T.

Θεώρημα 4 ([2], 9.9) Έστω P ένα σύνολο σημείων στο επίπεδο. Κάθε γωνιακά βέλτιστος τριγωνισμός του P είναι τριγωνισμός Delaunay του P. Επιπλέον, κάθε τριγωνισμός Delaunay του P μεγιστοποιεί την ελάχιστη γωνία ως προς όλους τους τριγωνισμούς του P.

2.2 Περιγραφή του Αλγορίθμου

Ο αλγόριθμος των Guibas, Knuth και Sharir είναι ένας τυχαιοκρατικός, αυξητικός αλγόριθμος στον οποίο τα σημεία εισάγονται στον τρέχοντα τριγωνισμό με τυχαία σειρά και τηρείται ένας τριγωνισμός Delaunay του τρέχοντος σημειοσυνόλου. Παρακάτω παρουσιάζεται η λειτουργία του αλγορίθμου σε μορφή ψευδοκώδικα.

1. Έστω p_0 το λεξικογραφικά υψηλότερο σημείο του P, δηλαδή το ακραίο δεξιό από τα σημεία με τη μεγαλύτερη τεταγμένη.
2. Έστω p_{-1} και p_{-2} δύο σημεία στο $\mathbb{R}^2$ σε αρκετά μεγάλη απόσταση και τέτοια ώστε το P να εμπεριέχεται στο τρίγωνο $p_0p_{-1}p_{-2}$.
3. Δημιουργούμε τον τριγωνισμό T που αποτελείται από το μοναδικό τρίγωνο $p_0p_{-1}p_{-2}$.
4. Υπολογίζουμε μια τυχαία μετάθεση $p_1, p_2,...,p_n$ του $P \setminus \{p_0\}$.
5. για $r \leftarrow 1$ έως n
    a. Εισάγουμε το p_r στον T
    b. Βρίσκουμε ένα τρίγωνο $p_ip_jp_k \in T$ που περιέχει το p_r.
    c. εάν το p_r κείται στο εσωτερικό του τριγώνου $p_ip_jp_k$
        i. τότε
            1. Προσθέτουμε ακμές από το p_r προς τις τρεις κορυφές του $p_ip_jp_k$, διασπώντας έτσι το $p_ip_jp_k$ σε τρία τρίγωνα.
            2. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_ip_j , T)
            3. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_jp_k, T)
            4. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_kp_i, T)
        ii. άλλως
            (∗ το p_r κείται επάνω σε ακμή του $p_ip_jp_k$, έστω στην ακμή p_ip_j ∗)
            1. Προσθέτουμε ακμές από το p_r προς το p_k και προς την τρίτη κορυφή p_l του άλλου τριγώνου που προσπίπτει στην p_ip_j, διασπώντας έτσι τα δύο τρίγωνα που προσπίπτουν στην p_ip_j σε τέσσερα τρίγωνα.
            2. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_ip_l, T)
            3. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_lp_j , T)
            4. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_jp_k, T)
            5. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_kp_i, T)
6. Διαγράφουμε από τον T τα σημεία p_{-1} και p_{-2}, και όλες τις προσπίπτουσες σε αυτά ακμές.
7. Επιστροφή T

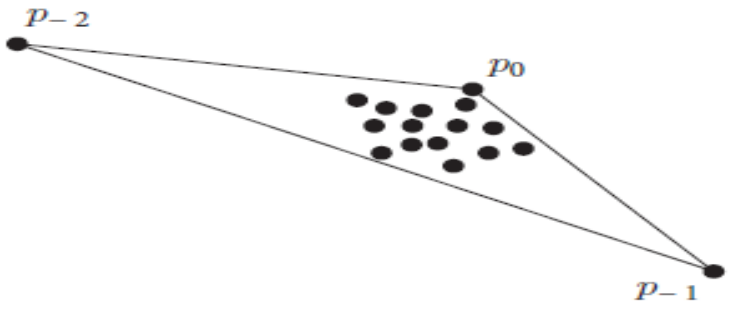
Αλγόριθμος 1: Υπολογισμός του Τριγωνισμού Delaunay

Ο Αλγόριθμος 1 με την σειρά του στις γραμμές 2, 3, 4 και 5 καλεί την αναδρομική συνάρτηση ΔΙΟΡΘΩΣΗ_ΑΣΚΜΗΣ η οποία καλείται για να μεταστρέψει πιθανόν ανεπίτρεπτες ακμές σε επιτρεπτές. Πιο συγκεκριμένα η δομή της συνάρτησης ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ παρουσιάζεται σε μορφή ψευδοκώδικα παρακάτω.

ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ(P_r, P_iP_j, T)
(∗ Το εισαγόμενο σημείο είναι το p_r, και η p_ip_j είναι η ακμή του T που πιθανόν να πρέπει να μεταστραφεί. ∗)
1. εάν η p_ip_j είναι ανεπίτρεπτη
    a. τότε
        i. Έστω $p_ip_jp_k$ το τρίγωνο που είναι παρακείμενο του $p_rp_ip_j$ με κοινή πλευρά την p_ip_j .
        (∗ Μεταστροφή της p_ip_j: ∗)
        ii. Αντικαθιστούμε την p_ip_j με την p_rp_k.
        iii. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_ip_k, T)
        iv. ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ (p_r, p_kp_j , T)

Αλγόριθμος 2: Ο αλγόριθμος ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ

Στον Αλγόριθμο 1 μας διευκολύνει να ξεκινήσουμε με ένα με ένα μεγάλο τρίγωνο που περιέχει το σύνολο P, ώστε να αποφεύγουμε τα προβλήματα που προκαλούν τα μη φραγμένα τραπέζια. Θα προσθέσουμε δύο επιπλέον σημεία, p_{-1} και p_{-2}, τα οποία, μαζί με το υψηλότερο λεξικογραφικά σημείο p_0 του P, σχηματίζουν ένα τρίγωνο που εμπεριέχει όλα τα σημεία του συνόλου P. Αυτό σημαίνει ότι θα υπολογίσουμε έναν τριγωνισμό Delaunay του συνόλου $P \in \{p_{-1}, p_{-2}\}$ αντί του P. Στο Σχήμα 2.1 απεικονίζεται το αρχικό μεγάλο τρίγωνο.

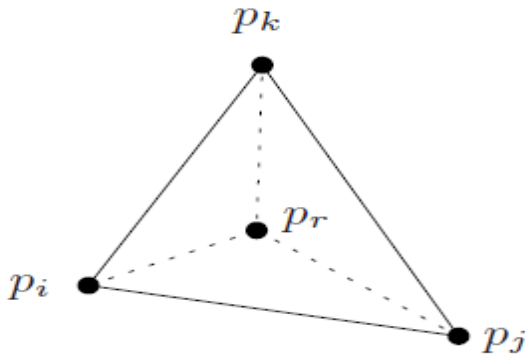


Σχήμα 2.1: Το αρχικό μεγάλο τρίγωνο [2].

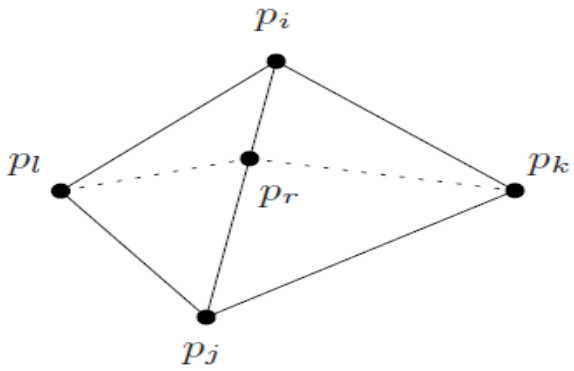
Ο τελικός στόχος μας στον Αλγόριθμο 1 είναι να πάρουμε τον τριγωνισμό Delaunay του P διαγράφοντας τα p_{-1} και p_{-2} μαζί με όλες τις προσπίπτουσες ακμές. Επομένως, θα πρέπει να επιλέξουμε τα p_{-1} και p_{-2} αρκετά απομακρυσμένα, έτσι ώστε να μην καταστρέφεται κάποιο από τα τρίγωνα του τριγωνισμού Delaunay του P. Συγκεκριμένα, θα πρέπει να φροντίσουμε να μην βρίσκονται σε κανέναν κύκλο ο οποίος διέρχεται από τρία σημεία του P.

Ο αλγόριθμος είναι τυχαιοκρατικού αυξητικού τύπου, δηλαδή τα σημεία προστίθενται με τυχαία σειρά και τηρείται ένας τριγωνισμός Delaunay του τρέχοντος συνόλου από σημεία.

Ας εξετάσουμε πως γίνεται η προσθήκη ενός σημείου p_r. Αρχικά βρίσκουμε το τρίγωνο του τρέχοντος τριγωνισμού που περιέχει το p_r με αναζήτηση στη δομή δέντρου. Αν το p_r κείται εντός του συγκεκριμένου τριγώνου, τότε προσθέτουμε ακμές από το p_r προς τις κορυφές αυτού του τριγώνου. Αν το p_r κείται πάνω σε κάποια ακμή e του τριγωνισμού, θα πρέπει να προσθέσουμε ακμές από το p_r προς τις απέναντι κορυφές στα τρίγωνα που έχουν κοινή πλευρά την e. Η πρώτη περίπτωση παρουσιάζεται στο Σχήμα 2.2 και η δεύτερη στο Σχήμα 2.3. Ως αποτέλεσμα της εισαγωγής σημείου προκύπτει και πάλι ένας τριγωνισμός, ο οποίος όμως ενδέχεται να μην είναι τριγωνισμός Delaunay. Ο λόγος είναι ότι η προσθήκη του p_r μπορεί να καταστήσει κάποιες από τις υπάρχουσες ακμές ανεπίτρεπτες.

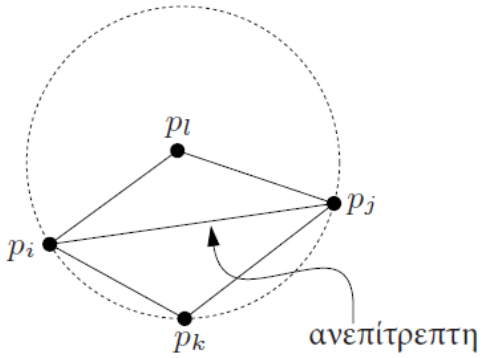


Σχήμα 2.2: Το σημείο P_r κείται εντός του τριγώνου [2].



Σχήμα 2.3: Το σημείο P_r κείται πάνω σε ακμή του τριγώνου [2].

Θεώρημα 5 ([2] , 9.4) Έστω ότι η ακμή p_ip_j προσπίπτει στα τρίγωνα pipjpk και pipjpl, και έστω C ο κύκλος που διέρχεται από τα p_i, p_j, και p_k. Η ακμή p_ip_j είναι ανεπίτρεπτη αν και μόνο αν το σημείο p_l κείται στο εσωτερικό του C. Επιπλέον, αν τα σημεία p_i, p_j, p_k, p_l σχηματίζουν κυρτό τετράπλευρο και δεν κείνται επάνω στον ίδιο κύκλο, τότε ακριβώς μία από τις ακμές p_ip_j και p_kp_l είναι ανεπίτρεπτη. Μία τέτοια περίπτωση ανεπίτρεπτης ακμής παρουσιάζεται στο Σχήμα 2.4.



Σχήμα 2.4: Η ακμή p_ip_j είναι ανεπίτρεπτη [2].

Κάθε τριγωνισμός που δεν περιέχει ανεπίτρεπτες ακμές ονομάζεται επιτρεπτός τριγωνισμός. Από την παραπάνω παρατήρηση έπεται ότι κάθε γωνιακά βέλτιστος τριγωνισμός είναι επιτρεπτός. Ο υπολογισμός ενός επιτρεπτού τριγωνισμού είναι πολύ απλός, εφόσον μας δοθεί κάποιος αρχικός τριγωνισμός. Απλώς μεταστρέφουμε

ανεπίτρεπτες ακμές έως ότου όλες οι ακμές να είναι επιτρεπτές. Στον Αλγόριθμο 3 περιγράφεται η διαδικασία μετατροπής ενός τριγωνισμού T ενός σημειοσυνόλου P σε επιτρεπτό τριγωνισμό του P.

1. Ενόσω ο T περιέχει κάποια ανεπίτρεπτη ακμή p_ip_j
    a. (*Μεταστρέφουμε την p_ip_j *)
    b. Έστω $p_ip_jp_k$ και $p_ip_jp_l$ τα δύο παρακείμενα προς την p_ip_j τρίγωνα
    c. Αφαιρούμε από τον T την p_ip_j και προσθέτουμε στην θέση της την p_kp_l
2. Επιστροφή T

Αλγόριθμος 3: Ο αλγόριθμος μετατροπής τριγωνισμού σε επιτρεπτό

Το Θεώρημα 5 ([2] , 9.4) δεν αρκεί για τον έλεγχο ανεπίτρεπτης ακμής καθώς η ύπαρξη των σημείων p_{-1}, p_{-2} δυσχεραίνει τον έλεγχο. Για να αναλύσουμε τον έλεγχο ανεπίτρεπτης ακμής αρχικά πρέπει να ορίσουμε πως επιλέγονται τα σημεία p_{-1}, p_{-2}. Αρχικά πρέπει να επιλέξουμε τα σημεία αρκετά απομακρυσμένα ώστε να μην επηρεάζεται ο τελικός μας τριγωνισμός. Ταυτοχρόνως θέλουμε να αποφύγουμε να ορίσουμε τα σημεία p_{-1}, p_{-2} με πολύ μεγάλες συντεταγμένες. Για τον λόγο αυτό θεωρούμε τα σημεία μη πραγματικά, δηλαδή δεν ορίζουμε πραγματικές συντεταγμένες και τα αντιμετωπίζουμε ως συμβολικά. Παρόλα αυτά τροποποιούμε τους ελέγχους για τον εντοπισμό σημείου αλλά και για τις ανεπίτρεπτες ακμές ώστε τα σημεία να αντιμετωπίζονται ως πραγματικά. Πιο συγκεκριμένα έστω p_ip_j η ακμή που πρέπει να ελεγχθεί και p_k και p_l οι άλλες κορυφές των τριγώνων που προσπίπτουν στην p_ip_j.

- Η p_ip_j είναι μια ακμή του τριγώνου $p_0p_{-1}p_{-2}$. Οι ακμές αυτές είναι πάντοτε επιτρεπτές.
- Οι δείκτες i, j, k, l είναι όλοι μη αρνητικοί. Αυτή είναι η κανονική περίπτωση και κανένα από τα σημεία που υπεισέρχονται στον έλεγχο δεν αντιμετωπίζεται συμβολικά. Άρα, η pipj είναι ανεπίτρεπτη εάν και μόνο εάν το pl κείται εντός του κύκλου που ορίζουν τα p_i, p_j και p_k.
- Όλες οι άλλες περιπτώσεις και πιο συγκεκριμένα η p_ip_j είναι επιτρεπτή εάν και μόνο εάν $\min(k, l) < \min(i, j)$.

Αφού η κατάσταση όπου η p_ip_j είναι η $p_{-1}p_{-2}$ αντιμετωπίζεται στην πρώτη περίπτωση, το πολύ ένας από τους δείκτες i και j είναι αρνητικός. Από την άλλη πλευρά, είτε το p_k είτε το p_l είναι το σημείο p_r που μόλις εισαγάγαμε, και συνεπώς το πολύ ένας από τους δείκτες k και l είναι αρνητικός. Εάν μόνο ένας από τους τέσσερεις δείκτες είναι αρνητικός, τότε το σημείο αυτό κείται εκτός του κύκλου που ορίζουν τα άλλα τρία σημεία, και συνεπώς η

μέθοδος είναι ορθή. Διαφορετικά τα min(i, j) και min(k, l) είναι αμφότερα αρνητικά, και το γεγονός ότι το p_{-2} κείται έξω από οποιονδήποτε κύκλο ο οποίος ορίζεται από τρία σημεία του $P \cup \{ p_{-1}\}$ συνεπάγεται ότι η μέθοδος είναι ορθή.

Αντιμετωπίζουμε το πρόβλημα της πιθανότητας ύπαρξης ανεπίτρεπτών ακμών καλώντας την διαδικασία ΔΙΟΡΘΩΣΗ_ΑΚΜΗΣ για κάθε ακμή που ενδέχεται να καταστεί ανεπίτρεπτή μετά την εισαγωγή ενός σημείου p_r. Η διαδικασία αντικαθιστά ανεπίτρεπτες ακμές με επιτρεπτές μέσω μεταστροφής ακμών όπως φαίνεται στον Αλγόριθμο 3.

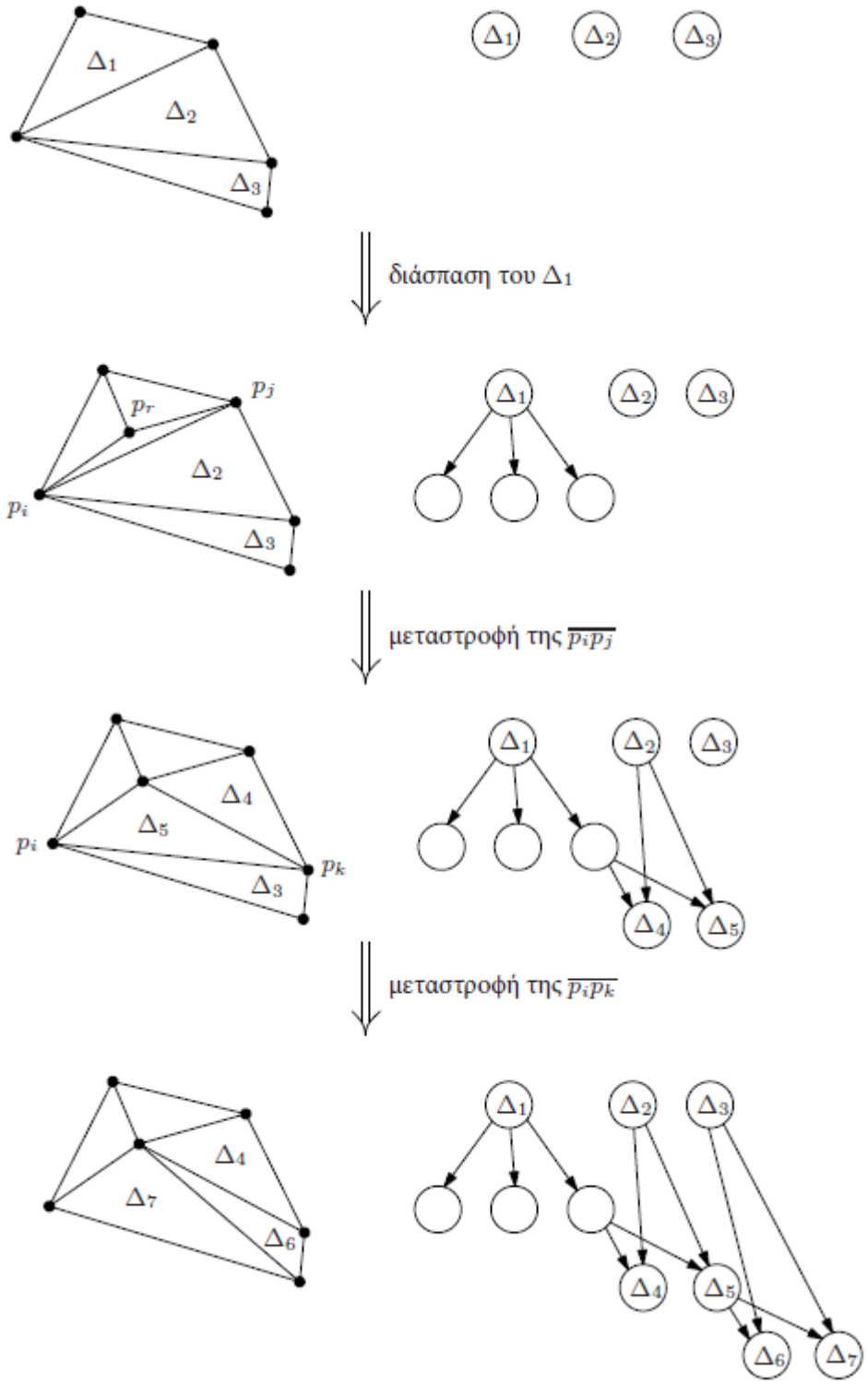
Κατά την κατασκευή του τριγωνισμού Delaunay κατασκευάζουμε και μία δομή εντοπισμού σημείου ώστε να βρούμε το τρίγωνο που περιέχει το p_r. Αρχικά κατασκευάζουμε τη δομή D ως κατευθυντικό άκυκλο γράφημα με ένα μόνο φύλλο που περιέχει το αρχικό τρίγωνο $p_0p_{-1}p_{-2}$. Τα φύλλα της D αντιστοιχούν στα τρίγωνα του τρέχοντος τριγωνισμού T .

Ας δούμε πως γίνεται η εύρεση του τριγώνου που περιέχει το σημείο p_r στην γραμμή 5b.(Αλγόριθμος 1). Όταν κατασκευάζουμε τον τριγωνισμό Delaunay κατασκευάζουμε επίσης μια δομή εντοπισμού σημείου D η οποία που αποτελεί κατευθυντικό άκυκλο γράφημα. Τα φύλλα της D αντιστοιχούν στα τρίγωνα του τρέχοντος τριγωνισμού T και επίσης κατασκευάζουμε αμφίδρομους δείκτες μεταξύ των φύλλων και του τριγωνισμού.

Οι εσωτερικοί κόμβοι της D αντιστοιχούν σε τρίγωνα που περιλαμβάνονταν στον τριγωνισμό σε κάποιο προηγούμενο στάδιο, αλλά έχουν ήδη καταστραφεί. Πιο συγκεκριμένα στη γραμμή 3 (Αλγόριθμος 1) δημιουργούμε την δομή δεδομένων D ως κατευθυντικό άκυκλο γράφημα με ένα μόνο φύλλο, που αντιστοιχεί στο τρίγωνο $p_0p_{-1}p_{-2}$. Σε κάποια χρονική στιγμή ένα τρίγωνο pipjpk του τρέχοντος τριγωνισμού σε τρία (ή δύο) νέα τρίγωνα. Η αλλαγή που παρατηρείται στην δομή δεδομένων D είναι η προσθήκη σε αυτήν τριών (ή δύο) νέων φύλλων, και η μετατροπή του φύλλου για το pipjpk σε εσωτερικό κόμβο με εξερχόμενους δείκτες προς αυτά τα τρία (ή δύο) φύλλα. Παρομοίως , όταν αντικαθιστούμε δύο τρίγωνα $p_kp_ip_j$ και $p_ip_jp_l$ με τα τρίγωνα $p_kp_ip_l$ και $p_kp_lp_j$ μέσω μιας μεταστροφής ακμής, δημιουργούμε φύλλα για τα δύο νέα τρίγωνα, και προσθέτουμε στους κόμβους των $p_kp_ip_j$ και $p_ip_jp_l$ δείκτες προς τα δύο νέα φύλλα.

Ξεκινώντας από τη ρίζα της δομής δεδομένων D (η οποία αντιστοιχεί στο αρχικό μεγάλο τρίγωνο βλέπε Σχήμα 2.1) ελέγχουμε τους τρεις θυγατρικούς κόμβους της ρίζας για να δούμε ποιο τρίγωνο περιέχει το p_r και μεταβαίνουμε στον αντίστοιχο θυγατρικό κόμβο. Στην συνέχεια ελέγχουμε τους θυγατρικούς αυτού του κόμβου, μεταβαίνουμε στον θυγατρικό που αντιστοιχεί στο τρίγωνο που περιέχει το p_r, κ.ο.κ., μέχρις ώτου να

φθάσουμε σε φύλλο της D το οποίο αντιστοιχεί σε τρίγωνο του τρέχοντος τριγωνισμού που περιέχει το p_r. Στο παρακάτω σχήμα παρουσιάζεται η μεταβολή της δομής δεδομένων δέντρου λόγω της εισαγωγής του σημείου p_r.



Σχήμα2.5: Η μεταβολής της δομής δεδομένων δέντρου[2].

2.3 Πολυπλοκότητα του Αλγορίθμου

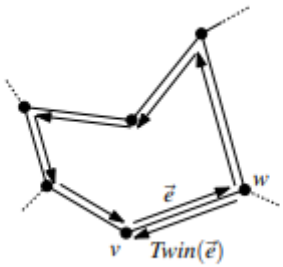
Θεώρημα 6 ([2] , 9.11) Το αναμενόμενο πλήθος τριγώνων που δημιουργούνται από τον αλγόριθμο Delaunay είναι το πολύ 9n + 1. Ο τριγωνισμός Delaunay ενός συνόλου P από n σημεία στο επίπεδο μπορεί να υπολογιστεί σε αναμενόμενο χρόνο O(n log n), και σε αναμενόμενο χώρο O(n). Το συνολικό πλήθος των δημιουργούμενων τριγώνων είναι 1, για το αρχικό τρίγωνο $p_0p_{-1}p_{-2}$, συν το πλήθος των τριγώνων που δημιουργούνται σε καθένα από τα βήματα εισαγωγής. Από τη γραμμικότητα των αναμενόμενων τιμών, συμπεραίνουμε ότι το αναμενόμενο συνολικό πλήθος δημιουργούμενων τριγώνων διέπεται από το φράγμα 1 + 9n.

Θεώρημα 7 ([2] , 9.12) Ο τριγωνισμός Delaunay ενός συνόλου P από n σημεία στο επίπεδο μπορεί να υπολογιστεί σε αναμενόμενο χρόνο O(nlogn), και σε αναμενόμενο χώρο O(n).

Παρατηρούμε ότι μόνο η δομή αναζήτησης D θα μπορούσε να απαιτεί περισσότερο από γραμμικό χώρο. Δεδομένου ότι ο βαθμός εξόδου οποιουδήποτε κόμβου είναι το πολύ τρία, η αναζήτηση απαιτεί χρόνο γραμμικό ως προς το πλήθος των κόμβων που βρίσκονται στη διαδρομή αναζήτησης, ή, με άλλα λόγια, ως προς το πλήθος των αποθηκευμένων στην D τριγώνων που περιέχουν το p_r. Κάθε κόμβος της D αντιστοιχεί σε ένα τρίγωνο που δημιουργείται από τον αλγόριθμο, και σύμφωνα με το Θεώρημα 6 ([2] , 9.11) το αναμενόμενο πλήθος αυτών των τριγώνων είναι O(n).

2.4 Η Δομή DCEL

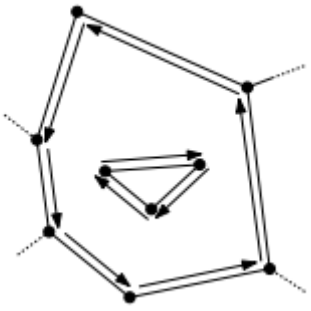
Η διπλά συνδεδεμένη λίστα ακμών βασίζεται στην παράσταση των δύο πλευρών μια ακμής ως δύο ξεχωριστές ημι-ακμές , οι οποίες ονομάζονται δίδυμες. Αυτό συνεπάγεται ότι κάθε ημι-ακμή ανήκει στο σύνορο μιας μόνο περιοχής. Οι ημι-ακμές είναι κατευθυνόμενες και συνήθως οι φορές τους είναι σύμφωνες με τη διάσχιση των περιοχών κατά την ανθωρολογιακή φορά. Έτσι μία περιοχή που είναι γειτονική σε μία ημι-ακμή βρίσκεται στα αριστερά ενός παρατηρητή που κινείται κατά μήκος της κατευθυνόμενης ημι-ακμής. Επιπλέον για μία ημι-ακμή ορίζουμε την κορυφή αφετηρίας και την κορυφή προορισμού της. Έτσι αν μία ημι-ακμή e έχει ως αφετηρία της την κορυφή v και ως απόληξη της την κορυφή w τότε η δίδυμη ακμή της twin(e) έχει ως αφετηρία της το w και ως απόληξη της το v όπως φαίνεται στο παρακάτω σχήμα 2.6.



Σχήμα 2.6: Η αφετηρία της twin(e) [2].

Μπορούμε επίσης να έχουμε πρόσβαση στο σύνορο μίας περιοχής/έδρας αποθηκεύοντας στην εγγραφή της περιοχής έναν δείκτη σε μία από τις ημι-ακμές που περικλείουν την περιοχή. Με αυτόν τον τρόπο μπορούμε να κινηθούμε κατά μήκος του συνόρου της περιοχής ξεκινώντας από αυτήν την ημι-ακμή και πηγαίνοντας στην επόμενη της.

Οι παραπάνω παρατηρήσεις δεν ισχύουν για περιοχές που έχουν οπές. Πιο συγκεκριμένα αν ένας παρατηρητής μετακινηθεί κατά μήκος του συνόρου μίας οπής κατά την ανθωρολογιακή φορά τότε η περιοχή που περικλείει την οπή θα βρίσκεται στα δεξιά του. Είναι χρησιμότερο να αποδίδουμε στις ημι-ακμές τέτοια κατεύθυνση ώστε η έδρα τους να κείται πάντα προς την ίδια πλευρά. Για να το πετύχουμε αυτό αλλάζουμε τη φορά διάνυσης του συνόρου μίας οπής σε ωρολογιακή φορά (Σχήμα 2.7). Με αυτόν τον τρόπο μια έδρα κείται πάντα στα αριστερά κάθε ημι-ακμής του συνόρου της. Ένα άλλο συμπέρασμα που προκύπτει είναι ότι οι δίδυμες ημι-ακμές έχουν πάντα αντίθετες κατευθύνσεις. Επομένως δεν ένας δείκτης από την έδρα προς μια οποιαδήποτε ημι-ακμή του συνόρου της ώστε να διατρέξουμε όλο το σύνορο. Θα χρειαστούμε επιπλέον ένα δείκτη προς μία ημι-ακμή σε κάθε συνιστώσα του συνόρου.



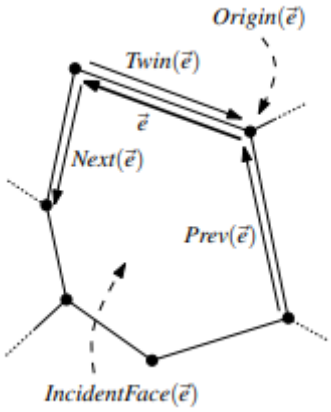
Σχήμα 2.7: Ωρολογιακή φορά διάνυσης οπής [2].

Μια διπλά συνδεδεμένη λίστα ακμών αποτελείται από τρεις συλλογές δελτίων: μία για τις κορυφές, μία για τις έδρες και μία για τις ημι-ακμές.

Το δελτίο για μία κορυφή u περιέχει τις συντεταγμένες στο πεδίο Coordinates(u). Επιπλέον περιέχει ένα δείκτη IncedentEdge(u) προς οποιαδήποτε ημι-ακμή που έχει αφετηρία την u.

Το δελτίο για μία έδρα/περιοχή περιέχει έναν δείκτη OuterComponent(f) σε μία ημι-ακμή του εξωτερικού της συνόρου (για περιοχές που δεν έχουν εξωτερικό σύνορο ο δείκτης είναι nil). Περιέχει επίσης μία λίστα InnerComponents(f) η οποία έχει έναν κόμβο για κάθε μία από τις οπές της f και κάθε τέτοιος κόμβος περιέχει έναν δείκτη σε μία ημι-ακμή του συνόρου της αντίστοιχης οπής.

Το δελτίο για μία ημι-ακμή e περιέχει έναν δείκτη Origin(e) προς την αφετηρία της , έναν δείκτη twin(e) προς τη δίδυμη της ημι-ακμή και έναν δείκτη IncidentFace(e) προς την έδρα την οποία οριοθετεί (Σχήμα 2.8). Δεν χρειάζεται να αποθηκεύσουμε την κορυφή προορισμού γιατί αυτή ταυτίζεται με την Origin(Twin(e)). Η αφετηρία επιλέγεται έτσι ώστε η έδρα IncidentFace(e) να κείται αριστερά της e όταν αυτή διανύεται από την αφετηρία προς την απόληξή της. Το δελτίο περιέχει ακόμα δύο δείκτες Prev(e) και Next(e) προς την επόμενη και την προηγούμενη ημι-ακμή του συνόρου της έδρας IncidentFace(e). Επομένως , η Next(e) είναι η μοναδική ημι-ακμή του συνόρου της έδρας IncidentFace(e) που έχει ως αφετηρία την απόληξη της e και η Prev(e) είναι η μοναδική ημι-ακμή τους συνόρου της έδρας IncidentFace(e) που έχει ως απόληξη την κορυφή Origin(e).

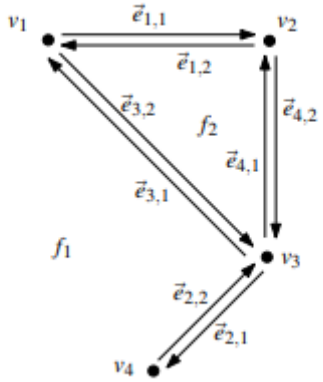


Σχήμα 2.8: Οι δείκτες των ημι-ακμών [2].

Για κάθε κορυφή και κάθε ακμή η ποσότητα των πληροφοριών που αποθηκεύονται είναι σταθερή. Ο συνολικός αποθηκευτικός χώρος είναι γραμμικός ως προς την

πολυπλοκότητα της υποδιαίρεσης, δεδομένου ότι οποιαδήποτε ημι-ακμή δείχνεται από το πολύ έναν από τους δείκτες των λιστών InnerComponents() όλων των περιοχών.

Στο Σχήμα 2.9 παρουσιάζεται ένα παράδειγμα μίας διπλά συνδεδεμένης λίστας ακμών για ένα απλό επίπεδο γράφημα.



Σχήμα 2.9: Παράδειγμα Dcel [2].

Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

Οι πληροφορίες που αποθηκεύονται στη διπλά συνδεδεμένη λίστα ακμών επαρκούν για να μας επιτρέψει να εκτελέσουμε τις βασικές λειτουργίες που επιθυμούμε. Για παράδειγμα μπορούμε να επισκεφθούμε όλες τις ακμές που προσπίπτουν σε κάποια κορυφή u.

Περιγράψαμε μία αρκετά γενική εκδοχή της διπλά συνδεδεμένης λίστας ακμών. Πιο απλές μορφές μπορεί να είναι επαρκείς για ειδικές εφαρμογές. Για παράδειγμα θα μπορούσαμε να αποθηκεύσουμε τις συντεταγμένες τους κατευθείαν στο πεδίο Origin(), δηλαδή δεν είναι απαραίτητο να έχουμε εγγραφές για τις κορυφές. Επίσης όταν σε εφαρμογές οι έδρες της υποδιαίρεσης δεν έχουν νόημα μπορούμε να αγνοήσουμε εντελώς τα δελτία των εδρών και το πεδίο IncidentFace των ημι-ακμών. Τέλος για υλοποιήσεις που απαιτούν το γράφημα των κορυφών και των ακμών να είναι συνεκτικό, μπορούμε να εισάγουμε πλασματικές ακμές. Με αυτόν τον τρόπο , θα μπορέσουμε να επισκεφτούμε όλες τις ημι-ακμές εφαρμόζοντας κάποιον αλγόριθμο διάνυσης γραφημάτων και δεύτερον η λίστα InnerComponents() για τις έδρες δεν είναι απαραίτητος.

2.5 Παραδείγματα Εφαρμογής του Αλγορίθμου

Ο τριγωνισμός Delaunay χρησιμοποιείται ως βάση σε πολλές εφαρμογές και αλγορίθμους. Οι τριγωνισμοί σημειοσυνόλων σε δύο και περισσότερες διαστάσεις έχουν τεράστια σημασία στην αριθμητική ανάλυση, π.χ. για τις μεθόδους πεπερασμένων στοιχείων, αλλά και στην υπολογιστική γραφιστική. Στο συγκεκριμένο αλγόριθμο εξετάσαμε την περίπτωση τριγωνισμών στους οποίους χρησιμοποιούνται ως κορυφές μόνο τα διδόμενα σημεία. Αν επιτρέπεται να χρησιμοποιηθούν και πρόσθετα σημεία (σημεία Steiner) το πρόβλημα είναι γνωστό και ως πλεγμάτωση.

Ένα τέτοιο παράδειγμα χρήσης είναι η αναπαράσταση 3Δ Χαρτών (Terrain)[3] είναι μια εφαρμογή η οποία έχει ευρεία χρήση ο τριγωνισμός Delaunay λόγω των ¨καλών¨ τριγώνων που παράγει. Προς το παρών η χρήση του περιορίζεται μόνο για εφαρμογές απεικόνισης γεωγραφικών Πληροφοριών (GIS). Δεν αξιοποιείται στην απεικόνιση χαρτών σε παιχνίδια λόγω του σχετικά μεγάλου χρόνου εκτέλεσης. Με την χρήση της Delaunay τριγωνοποίησης σε σχέση με την κλασική μέθοδο της αναπαράστασης πλέγματος, ο μεγαλύτερος αριθμός τριγώνων συγκεντρώνεται στα σημεία που υπάρχει περισσότερη πληροφορία.

Μια άλλη εφαρμογή του αλγορίθμου είναι ο υπολογισμός του ¨Ευκλείδειου ελάχιστου γενετικού δέντρου¨. Ένα Ευκλείδειο ελάχιστο γενετικού δένδρο βρίσκεται πάντα πάνω στις ακμές ενός Delaunay τριγωνισμού των σημείων του δένδρου, με αποτέλεσμα μετά

την εύρεση του τριγωνισμού Delaunay θέλουμε γραμμικό χρόνο για τον υπολογισμό του γενετικού δέντρου.

Κεφάλαιο 3. Η Υλοποίηση

Για τον υπολογισμό του τριγωνισμού έγινε χρήση της βασικής βιβλιοθήκης της python και της βιβλιοθήκης **numpy**. Για την απεικόνιση τόσο του αποτελέσματος όσο και των ενδιάμεσων αποτελεσμάτων χρησιμοποιήθηκαν οι βιβλιοθήκες **networkx** και **matplotlib**, ενώ για την απεικόνιση του δέντρου που δείχνει την εξέλιξη της δημιουργίας τριγώνων στον αλγόριθμο, χρησιμοποιήθηκε επιπλέον η βιβλιοθήκη **pygraphviz**.

3.1 Είσοδος – Έξοδος

Για την είσοδο του αλγορίθμου έχει κατασκευαστεί ένα python scipt με όνομα gendata.py το οποίο μετά το πέρας της εκτέλεσης του δημιουργεί ένα σύνολο δισδιάστατων σημείων σε κατάλληλο φάκελο στο οποίο θα εφαρμοστεί ο αλγόριθμος Delaunay. Αυτό το scipt παράγει τυχαία σημεία σε ένα κατάλληλο αρχείο τύπου csv. Στην συνέχεια η κλάση Dcel παίρνει ως είσοδο τα σημεία που έχουν δημιουργηθεί από το python scipt που προαναφέραμε.

Για την έξοδο του αλγορίθμου απεικονίζουμε σε εικόνες τα ενδιάμεσα βήματα του αλγορίθμου. Σημειώνεται ότι στα ενδιάμεσα βήματα δεν απεικονίζονται τα σημεία p_{-1} και p_{-2}, καθώς αυτά αντιμετωπίζονται θεωρητικά και δεν είναι πραγματικά σημεία όπως αναφέρεται στην Ενότητα 2.2. Επιπλέον για την έξοδο του αλγορίθμου παράγουμε εικόνες για την απεικόνιση του δέντρου των ενδιάμεσων τριγώνων. Τέλος για καλύτερη απεικόνιση των ενδιάμεσων βημάτων έχουμε δημιουργήσει ένα python scipt που μετατρέπει την ακολουθία εικόνων σε βίντεο.

3.2 Δομές Δεδομένων

Σε αυτή την ενότητα θα αναφέρουμε τις βασικές δομές δεδομένων που είναι απαραίτητες για την λειτουργία του αλγορίθμου.

Στην κλάση **Dcel** έχουμε την δομή πίνακα numpy με όνομα **row_ids** η οποία περιέχει τα σημεία με τις μέγιστες τεταγμένες. Επιπλέον έχουμε υλοποιήσει την δομή πίνακα numpy με όνομα **cell_row** η οποία βρίσκει από τα σημεία με τη μεγαλύτερη y συντεταγμένη το σημείο από αυτά που έχει και την μεγαλύτερη x συντεταγμένη. Επίσης έχουμε υλοποιήσει την λίστα **self.__triangles** με τα τρίγωνα που έχει αυτή τη στιγμή ο τριγωνισμός και την

λίστα **nodes** η οποία περιέχει τους κόμβους του δέντρου με τα τρίγωνα που αφαιρέσαμε. Στη λίστα **triangles_with_edge** αποθηκεύουμε από τα τρίγωνα που έχουμε στον τρέχον τριγωνισμό αυτά περιέχουν την ακμή που έχουμε δώσει σαν όρισμα. Εφόσον έχουμε βρει τα τρίγωνα που περιέχουν αυτή την ακμή βρίσκουμε και το τρίτο σημείο του κάθε τριγώνου δηλαδή πέρα από την ακμή ποιο άλλο σημείο έχει και αποθηκεύουμε αυτή τη πληροφορία στην λίστα **points**. Επιπροσθέτως, έχουμε διατηρήσει τον πίνακα **rest_points** ο οποίος περιέχει τα σημεία που δεν έχουμε εξετάσει ακόμα. Η λίστα **triangles_to remove** περιέχει τα τρίγωνα που πρέπει αν αφαιρεθούν όταν βρίσκονται πάνω σε ακμή και είναι παρακείμενα. Τέλος στη λίστα **other_points** κρατάμε τα άλλα δυο σημεία που αποτελούν τις κορυφές των δύο παρακείμενων τριγώνων που περιέχουν το αντίστοιχο σημείο.

Στην κλάση **Dtree** κάθε κόμβος του δέντρου δηλαδή κάθε στιγμιότυπο της κλάσης Dtree περιέχει το αντίστοιχο τρίγωνο, τη λίστα **self__parents** με τους γονείς του και τη λίστα **self__children** με τα παιδία του που αρχικά είναι κενή.

Στην κλάση **Point** διατηρούμε τις συντεταγμένες ενός σημείου στον πίνακα με όνομα **self.__coords**.

Στην κλάση **Triangle** διατηρούμε ένα λεξικό με όνομα **self.__points**. Επιπλέον διατηρούμε τον πίνακα **self.__nodeIds** με τα ids των τριών σημείων που σχηματίζουν το αντίστοιχο τρίγωνο. Τέλος διατηρούμε μία λίστα με όνομα **theids** με τα σημεία του τριγώνου εκτός από τα αρνητικά.

3.3 Ενδεικτικές Κλάσεις

Στην ενότητα αυτή περιγράφονται οι βασικές κλάσεις οι οποίες χρησιμοποιούνται για την υλοποίηση του αλγορίθμου. Αρχικά παρουσιάζονται οι κλάσεις που λειτουργούν ως οντότητες του αλγορίθμου, ενώ στη συνέχεια παρουσιάζονται οι κλάσεις που σχετίζονται με τη λειτουργία του.

3.3.1 Κλάσεις Οντότητες

Στην ενότητα αυτή θα περιγράψουμε τις κλάσεις που περιγράφουν ένα σημείο, το διάνυσμα που ορίζεται από 2 σημεία, το τρίγωνο που ορίζεται από τρία σημεία και τέλος την κλάση που ορίζει τον περιγεγραμμένου κύκλου ενός τριγώνου.

3.3.1.1 Η κλάση Point

Η συγκεκριμένη κλάση χρησιμοποιείται για να περιγράψει ένα σημείο του προβλήματος. Για την κατασκευή της είναι απαραίτητος ο προσδιορισμός των συντεταγμένων που το καθορίζουν, αλλά και της σειράς με την οποία εισάγεται στον τριγωνισμό. Υπάρχουν κατάλληλες μέθοδοι που επιστρέφουν το σύνολο των δύο συντεταγμένων, καθώς και την κάθε μία επιμέρους. Επιπλέον, υλοποιείται η μέθοδος τελεστή της python **__gt__** ώστε να μπορούν να συγκριθούν δύο σημεία με βάση τη λεξικογραφική διάταξη που χρησιμοποιείται στον αλγόριθμο.

```
import numpy as np

class Point:

    def __init__(self, x, y, refId):
        self.__coord = np.array([x, y])
        self.__refId = refId

    @property
    def x(self):
        return self.__coord[0]

    @property
    def y(self):
        return self.__coord[1]

    @property
    def coords(self):
        return self.__coord

    @property
    def id(self):
        return self.__refId

    def __gt__(self, otherPoint):
        if self.y > otherPoint.y:
            return True
        elif (self.y == otherPoint.y) & (self.x > otherPoint.x):
            return True
        else:
            return False

    def __str__(self):
        return "%d(%.3f, %.3f)"%(self.id, self.x, self.y)
```

3.3.1.2 Η κλάση Vector

Η συγκεκριμένη κλάση χρησιμοποιείται για να περιγράψει ένα διάνυσμα του επιπέδου. Για την κατασκευή και τον ορισμό του διανύσματος, χρειάζεται ο ορισμός του σημείου αρχής και τέλους που ορίζουν το διάνυσμα και η κλάση αφαιρεί τις συντεταγμένες τους για να προσδιορίσει τις συντεταγμένες του διανύσματος. Επιπλέον, παρέχει τη συνάρτηση **innerProduct**, η οποία μπορεί να υπολογίσει το εσωτερικό γινόμενο του

διανύσματος (self) με ένα άλλο που δίνεται ως όρισμα (aVector). Η υλοποίηση πραγματοποιείται με τη βοήθεια της βιβλιοθήκης numpy.

```
class Vector:

    def __init__(self, pointA, pointB):
        self._pointA = pointA
        self._pointB = pointB
        self._coords = pointB.coords - pointA.coords

    @property
    def pointA(self):
        return self._pointA

    @property
    def pointB(self):
        return self._pointB

    @property
    def coords(self):
        return self._coords

    def __str__(self):
        return "[%s -> %s]"%(str(self.pointA), str(self.pointB))

    def innerProduct(self, aVector):
        return np.dot(self.coords, aVector.coords)

    def formAdjacentTrianglesForCheck(self, pointA, pointB):
        first = self.pointA if self.pointA.id < self.pointB.id else self.pointB
        second = self.pointB if self.pointA.id < self.pointB.id else self.pointA
        testP1 = pointA if pointA.id < pointB.id else pointB
        testP2 = pointB if pointA.id < pointB.id else pointA

        if first.id >= 0:
            if testP1.id >= 0:
                xros1  =  np.cross(Vector(first,  second).coords,  Vector(first,
testP1).coords)
                xros2  =  np.cross(Vector(first,  second).coords,  Vector(first,
testP2).coords)
                if xros1*xros2 > 0:
                    return False
                else:
                    return True
            else:
                return False
        else:
            if testP1.id < 0:
                return False
            else:
                small = testP2 if testP1 > testP2 else testP1
                large = testP1 if testP1 > testP2 else testP2
                xros  =  np.cross(Vector(small,  second).coords,  Vector(small,
large).coords)
                return True if (((xros > 0) & (first.id == -2)) | ((xros < 0) &
(first.id == -1))) else False
```

Παράλληλα, υλοποιείται η συνάρτηση **formAdjacentTrianglesForCheck**, η οποία δέχεται ως όρισμα την ίδια την ακμή (self) και δύο σημεία (pointA, pointB) και ελέγχει εάν η ακμή μαζί με αυτά τα δύο σημεία σχηματίζουν δύο παρακείμενα τρίγωνα, στα οποία έχει νόημα να ελεγχθούν οι συνθήκες της ανεπίτρεπτης ακμής του αλγορίθμου στην διαδικασία διόρθωσης ακμής.

3.3.1.3 Η κλάση Triangle

Η συγκεκριμένη κλάση χρησιμοποιείται για να περιγράψει ένα τρίγωνο του δισδιάστατου επιπέδου. Για τον προσδιορισμό του τριγώνου είναι απαραίτητο να δοθούν τα τρία σημεία που το καθορίζουν. Η κλάση επιπλέον παρέχει τη μέθοδο **contains**, η οποία ελέγχει κατά πόσο ένα σημείο βρίσκεται εντός του ορισμένου τριγώνου. Για την περίπτωση που το τρίγωνο αποτελείται από κάποια ή και τις δύο αρνητικές κορυφές, ακολουθούμε διαφορετικό έλεγχο για να δούμε εάν ένα σημείο βρίσκεται εντός του αντίστοιχου τριγώνου (συνάρτηση **__isInMinusNodeTriangle**).

Συγκεκριμένα, ελέγχουμε τη θέση του σημείου σε σχέση με κάθε μία από τις 3 ακμές του τρίγωνου. Για διανύσματα πραγματικών σημείων χρησιμοποιείται το εξωτερικό γινόμενο και το πρόσημο του, ενώ για διανύσματα που ορίζονται μέσω των βοηθητικών αρνητικών σημείων χρησιμοποιείται η λεξικογραφική διάταξη. Επίσης έχει υλοποιηθεί η μέθοδος **hasEdge**, η οποία ελέγχει εάν μία ακμή είναι ακμή του συγκεκριμένου τριγώνου. Η συνάρτηση **getThirdNode** επιστρέφει τον τρίτο κόμβο του τριγώνου με δεδομένη την ακμή που της δίνεται ως όρισμα. Τέλος, για να μπορούμε να ελέγξουμε εάν δύο τρίγωνα είναι ίσα, ελέγχουμε εάν ορίζονται με βάση τις ίδιες κορυφές υλοποιώντας τον τελεστή **__eq__**.

```
class Triangle:
    def __init__(self, pointA, pointB, pointC):
        self.__points = {pointA.id : pointA, pointB.id : pointB, pointC.id :
pointC }

        self.__nodeIds = np.array(list(self.__points.keys()))

        self.__containsMinusOne = np.isin(-1, self.__nodeIds)
        self.__containsMinusTwo = np.isin(-2, self.__nodeIds)

    def point(self, id):
        return self.__points[id] if id in self.__points else None

    @property
    def pointIds(self):
        return self.__nodeIds

    def containsMinusOneNode(self):
        return self.__containsMinusOne

    def containsMinusTwoNode(self):
        return self.__containsMinusTwo

    def hasEdge(self, pointA, pointB):
        return np.isin([pointA.id, pointB.id], self.__nodeIds).all()

    def getThirdNode(self, vector):
        points = np.array(list(self.__points.values()))
        theid    =    self.__nodeIds[np.isin(self.__nodeIds,    [vector.pointA.id,
vector.pointB.id]) == False][0]
        return self.__points[theid]

    def __str__(self):
        return   "[%20s,   %20s,   %20s]"%tuple([str(self.point(x))   for   x   in
list(self.pointIds)])

    def __eq__(self, otherTriangle):
        return np.isin(self.__nodeIds, otherTriangle.pointIds).all()
```

```
def __isInMinusNodeTriangle(self, minusSide, aPoint):
        theids  =  list(self.__nodeIds[np.isin(self.__nodeIds,  [minusSide])  ==
False])
        points = [self.point(x) for x in theids]
        pointA = points[0] if points[0] > points[1] else points[1]
        pointB = points[1] if points[0] > points[1] else points[0]

        xros    =    np.cross(Vector(pointA,    pointB).coords,    Vector(pointA,
aPoint).coords)
        if xros > 0:
            flag1 = True if minusSide == -1 else False
        elif xros == 0:
            if (((pointA > aPoint) & (aPoint > pointB)) | ((pointB > aPoint) &
(aPoint > pointA))):
                return True, Vector(pointA, pointB)
            else:
                return False, False
        else:
            flag1 = False if minusSide == -1 else True

        xros    =    np.cross(Vector(pointB,    pointA).coords,    Vector(pointB,
aPoint).coords)
        if xros < 0:
            flag2 = True if minusSide == -1 else False
        elif xros == 0:
            if (((pointA > aPoint) & (aPoint > pointB)) | ((pointB > aPoint) &
(aPoint > pointA))):
                return True, Vector(pointA, pointB)
            else:
                return False, False
        else:
            flag2 = False if minusSide == -1 else True

        if pointA > aPoint and aPoint > pointB:
            flag3 = True
        else:
            flag3 = False

        return (flag1 & flag2 & flag3, None)
```

3.3.1.4 Η κλάση Circle

Η κλάση Circle χρησιμοποιείται για τον προσδιορισμό του περιγεγραμμένου κύκλου ενός τριγώνου. Για τον προσδιορισμό του χρειάζεται να δοθεί το τρίγωνο που είναι αναγκαίο για την περιγραφή του. Το κέντρο και η ακτίνα του κύκλου βρίσκονται μέσω της γενικής μορφής της εξίσωσης που περιγράφει τον κύκλο. Αντικαθιστώντας σε αυτήν τα τρία σημεία που περιγράφουν το τρίγωνο και λύνοντας το σύστημα των εξισώσεων που προκύπτουν, βρίσκονται οι παράμετροι της εξίσωσης του κύκλου και από αυτές το κέντρο και η ακτίνα του. Παρέχεται επιπλέον η μέθοδος contains, η οποία ελέγχει κατά πόσο ένα σημείο βρίσκεται εντός του κύκλου που κατασκευάστηκε. Επειδή ο περιγεγραμμένος κύκλος ενός τριγώνου χρησιμοποιείται στον έλεγχο ανεπίτρεπτης ακμής, μόνο στην περίπτωση που τα σημεία που ελέγχονται είναι πραγματικά και όχι τα αρνητικά, δεν είναι απαραίτητο να ληφθούν ειδικές περιπτώσεις.

```
class Circle:
    def __init__(self, triangle):

        ids = triangle.pointIds

        x1 = triangle.point(ids[0]).x
        y1 = triangle.point(ids[0]).y
        x2 = triangle.point(ids[1]).x
        y2 = triangle.point(ids[1]).y
        x3 = triangle.point(ids[2]).x
        y3 = triangle.point(ids[2]).y

        A = x1*(y2-y3) - y1*(x2-x3) + x2*y3-x3*y2
        B = (x1**2 + y1**2)*(y3-y2) + (x2**2 + y2**2)*(y1-y3) + (x3**2 +
y3**2)*(y2-y1)
        C = (x1**2 + y1**2)*(x2-x3) + (x2**2 + y2**2)*(x3-x1) + (x3**2 +
y3**2)*(x1-x2)
        D = (x1**2 + y1**2)*(x3*y2-x2*y3) + (x2**2 + y2**2)*(x1*y3-x3*y1) + (x3**2
+ y3**2)*(x2*y1-x1*y2)

        self._center = Point(-B/(2*A), -C/(2*A), np.inf)

        self._radius = np.sqrt((B**2 + C**2 - 4*A*D)/(4*A**2))

    @property
    def center(self):
        return self._center

    @property
    def radius(self):
        return self._radius

    def contains(self, point):
        v = Vector(self.center, point)
        d = np.sqrt(v.innerProduct(v))

        if d < self.radius:
            return True
        else:
            return False

    def __str__(self):
        return "Circle[Center = %s, radius = %.3f]"%(str(self.center),
self.radius)
```

```
    def contains(self, aPoint):
        if self.containsMinusOneNode() and self.containsMinusTwoNode():
            theid = self.__nodeIds[np.isin(self.__nodeIds, [-1,-2]) == False][0]
            return (True, None) if self.point(theid) > aPoint else (False, None)
        elif self.containsMinusOneNode():
            return self.__isInMinusNodeTriangle(-1, aPoint)
        elif self.containsMinusTwoNode():
            return self.__isInMinusNodeTriangle(-2, aPoint)
        else:
            points = list(self.__points.values())
            # Create essential vectors
            v0 = Vector(points[0], points[1])
            v1 = Vector(points[0], points[2])
            v2 = Vector(points[0], aPoint)

            ## Compute all the inner products
            dot00 = v0.innerProduct(v0)
            dot01 = v0.innerProduct(v1)
            dot02 = v0.innerProduct(v2)
            dot11 = v1.innerProduct(v1)
            dot12 = v1.innerProduct(v2)

            ## Compute the barycentric coordinates
            invDenom = 1 / (dot00 * dot11 - dot01 * dot01)
            u = (dot11 * dot02 - dot01 * dot12) * invDenom
            v = (dot00 * dot12 - dot01 * dot02) * invDenom

            inTriangle = (u >= 0) & (v >= 0) & (u + v <= 1)

            if inTriangle:
                if u == 0:
                    return True, Vector(points[0], points[2])
                elif v == 0:
                    return True, Vector(points[0], points[1])
                elif u +v == 1:
                    return True, Vector(points[1], points[2])
                else:
                    return True, None
            else :
                return inTriangle, False
```

3.3.2 Κλάσεις Σχετικά με την Εκτέλεση του Αλγορίθμου

Σε αυτή την ενότητα θα περιγράψουμε τις κλάσεις που είναι απαραίτητες για την λειτουργία του αλγορίθμου.

3.3.2.1 Η κλάση Dtree

Η κλάση αυτή χρησιμοποιείται για την αναπαράσταση ενός κόμβου του δέντρου που περιγράφει το ιστορικό της εξέλιξης των τριγώνων στον τριγωνισμό. Ως πληροφορία περιέχει το τρίγωνο που υπάρχει στο συγκεκριμένο στάδιο, τους γονείς του κόμβου και τα παιδιά του και σε ποιο βήμα του τριγωνισμού δημιουργήθηκε. Περιέχει την κλάση

- **checkChildrenForTriangle**: Η συνάρτηση αυτή δέχεται ως όρισμα ένα τρίγωνο και επιστρέφει τον κόμβο του δέντρου από τα παιδιά του που το περιέχει.
- **appendChild**: Η συνάρτηση αυτή δέχεται ως όρισμα έναν κόμβο και τον τοποθετεί ως παιδί στο δέντρο.
- **getNodeWithTriangle**: Η συνάρτηση δέχεται ως όρισμα ένα τρίγωνο και ελέγχει εάν ο κόμβος το περιέχει. Εάν όχι, καλεί την checkChildrenForTriangle, για να επιστρέψει το παιδί με το αντίστοιχο τρίγωνο.
- **insertChild**: Η συνάρτηση αυτή δέχεται ως όρισμα ένα τρίγωνο, κάποιον άλλον κόμβο γονιό και το βήμα που εισάγεται το τρίγωνο στον τριγωνισμό και τοποθετεί στο δέντρο έναν νέο κόμβο για αυτό το τρίγωνο με γονείς τον κόμβο αντικείμενο στον οποίο καλείται και τον άλλον που δίνεται ως όρισμα otherParent.
- **getLastContainingTriangle**: Η συνάρτηση δέχεται ως όρισμα ένα σημείο και επιστρέφει το μικρότερο τρίγωνο του δέντρου που το περιέχει και την ακμή του τρίγωνο, σε περίπτωση που το σημείο βρίσκεται πάνω σε μία ακμή.

```
class Dtree:
    def __init__(self, triangle, parents=None, step=0):
        self.__triangle = triangle
        self.__parents = parents
        self.__children = []
        self.__step = step
        if self.__parents is None:
            self.__level = 0
        else:
            self.__level = max(list(map(lambda x : x.level, self.__parents))) + 1

    @property
    def triangle(self):
        return self.__triangle

    @property
    def level(self):
        return self.__level

    @property
    def children(self):
        return self.__children

    @property
    def parents(self):
        return self.__parents

    @property
    def step(self):
        return self.__step

    def checkChildrenForTriangle(self, triangle):
        if len(self.__children) == 0:
            return None

        for child in self.__children:
            if child.triangle == triangle:
                return child

        for child in self.__children:
            res = child.checkChildrenForTriangle(triangle)
            if not (res is None):
                return res
```

```
    def appendChild(self, child):
        self.__children.append(child)

    def getNodeWithTriangle(self, triangle):
        if self.__triangle == triangle:
            return node
        else:
            return self.checkChildrenForTriangle(triangle)

    def insertChild(self, triangle, otherParent=None, step = 0):
        parents = [self]
        if not (otherParent is None):
            parents.append(otherParent)
        child = Dtree(triangle, parents, step)
        self.appendChild(child)
        if not (otherParent is None):
            otherParent.appendChild(child)

    def getLastContainingTriangle(self, point, lastEdge = None):
        if len(self.__children) == 0:
            return self, lastEdge
        else:
            for i in self.__children:
                inTriangle, edge = i.triangle.contains(point)
                if inTriangle:
                    return i.getLastContainingTriangle(point, edge)
```

3.3.2.2 Η κλάση Dcel

Αυτή είναι η βασική κλάση που υλοποιεί τον αλγόριθμο Delaunay. Αρχικοποιείται με το σύνολο σημείων στα οποία θα γίνει εύρεση του τριγωνισμού. Στη συνέχεια με χρήση της διαδικασίας **execute**, γίνεται εκτέλεση του αλγορίθμου Delaunay. Αυτή χρησιμοποιεί διάφορες βοηθητικές συναρτήσεις, όπως τις **__get_initial_point**, **__fix_edge** και **__shift_edge**, για την επιλογή του λεξικογραφικά ψηλότερου σημείου, τη διόρθωση και τη μεταστροφή ακμής αντίστοιχα.

```
class DCEL:
    def __init__(self, points):

        logging.basicConfig(level=logging.INFO)

        self.__logger = logging.getLogger(__name__)
        #self.__logger.addHandler(fh)

        self.__step = 1
        self.__triangles = []
        self.__circles = []
        self.__dtree = None
        self.__points = points

    @property
    def dtree(self):
        return self.__dtree

    def __addNewTriangle(self, newTriangle, treenode, otherParent = None):
            self.__triangles.append(newTriangle)
            treenode.insertChild(newTriangle, otherParent, step = self.__step)
            self.__infoLogger("Triangle %s inserted to
triangulization."%(str(newTriangle)))

    def __infoLogger(self, string):
        self.__logger.info(string)

    def __get_initial_point(self):
        row_ids = np.where(self.__points[:,1] == np.max(self.__points,
axis=0)[1])[0]
        sel_row = np.argmax(self.__points[row_ids, :], axis=0)[0]
        p0 = self.__points[row_ids[sel_row], :]
        self.__infoLogger("Lexicographically Distant Point : %s"%(str(p0)))
        return(p0, row_ids[sel_row])

    def __remove_triangle(self, triangle):
        index = self.__triangles.index(triangle)
        del self.__triangles[index]
        self.__infoLogger("Removed triangle %s from
triangulization."%(str(triangle)))
        return

    def __shift_edge(self, pr, pk, edge, triangles):
        self.__infoLogger("Shifting Edge " + str(edge) + " to " + str(Vector(pr,
pk)))
        for triangle in triangles:
            self.__remove_triangle(triangle)

        nodes = [self.__dtree.getNodeWithTriangle(triangle) for triangle in
triangles]

        self.__addNewTriangle(Triangle(pr, pk, edge.pointA), nodes[0], nodes[1])
        self.__addNewTriangle(Triangle(pr, pk, edge.pointB), nodes[0], nodes[1])

        self.__step = self.__step + 1
```

```
def __fix_edge(self, pr, edge):
        self.__infoLogger("Fix Edge(pr = " + str(pr) + ", edge = " + str(edge) +
")")
        if self.__dtree.triangle.hasEdge(edge.pointA, edge.pointB):
            return
        else:
            triangles_with_edge   =   [i   for   i   in   self.__triangles   if
i.hasEdge(edge.pointA, edge.pointB)]
            points    =    [triangle.getThirdNode(edge)    for    triangle    in
triangles_with_edge]
            pk = points[0] if points[0].id != pr.id else points[1]

            adjacentTrianglesExist = edge.formAdjacentTrianglesForCheck(pr, pk)

            if not adjacentTrianglesExist:
                return

            if ((pk.id < 0)  | (pr.id < 0) | (edge.pointA.id < 0) | (edge.pointB.id
< 0)):
                if min(edge.pointA.id, edge.pointB.id) > min(pk.id, pr.id):
                    return
                else:
                    self.__shift_edge(pr, pk, edge, triangles_with_edge)
                    self.__fix_edge(pr, Vector(edge.pointA, pk))
                    self.__fix_edge(pr, Vector(pk, edge.pointB))

                    return
            else:
                if Circle(Triangle(edge.pointA, edge.pointB, pk)).contains(pr):
                    self.__shift_edge(pr, pk, edge, triangles_with_edge)
                    self.__fix_edge(pr, Vector(edge.pointA, pk))
                    self.__fix_edge(pr, Vector(pk, edge.pointB))

                    return
                else:
                    return
```

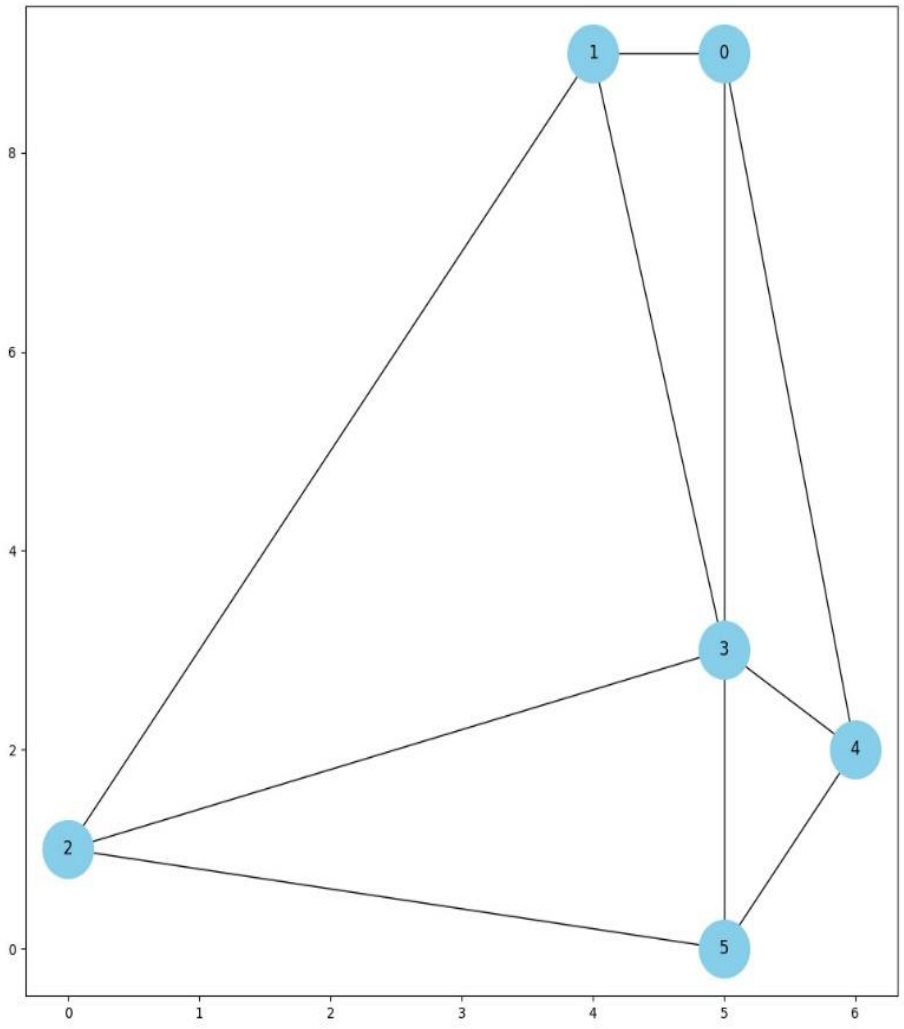
```
def execute(self):
        p0, ind = self.__get_initial_point()

        p0 = Point(p0[0], p0[1], 0)

        pm1 = Point(-1, -1, -1)

        pm2 = Point(-2, -2, -2)

        firstTriangle = Triangle(p0, pm1, pm2)
        self.__dtree = Dtree(firstTriangle, None, step = self.__step)
        self.__step = self.__step + 1

        self.__triangles.append(firstTriangle)

        self.__infoLogger("Created initial structures")

        rest_points = np.delete(self.__points, ind, axis=0)

        rest_points = np.random.permutation(rest_points)
        for i in range(np.size(rest_points, 0)):
            p = Point(rest_points[0, 0], rest_points[0,1], i+1)

            self.__infoLogger("Starting point %s insertion"%(str(p)))

            treenode, edge = self.__dtree.getLastContainingTriangle(p)

            if edge is None:
                self.__infoLogger("%s    located    inside    triangle    %s"%(str(p),
str(treenode.triangle)))

                self.__remove_triangle(treenode.triangle)
                points    =    list(map(lambda    x    :    treenode.triangle.point(x),
treenode.triangle.pointIds))
                self.__addNewTriangle(Triangle(points[0], points[1], p), treenode)
                self.__addNewTriangle(Triangle(points[0], points[2], p), treenode)
                self.__addNewTriangle(Triangle(points[1], points[2], p), treenode)

                self.__step = self.__step + 1

                self.__fix_edge(p, Vector(points[0], points[1]))
                self.__fix_edge(p, Vector(points[0], points[2]))
                self.__fix_edge(p, Vector(points[1], points[2]))
            else:
```

```
            else:
                self.__infoLogger("%s located on the edge %s of triangle
%s"%(str(p), str(edge), str(treenode.triangle)))

                triangles_to_remove = [i for i in self.__triangles if
i.hasEdge(edge.pointA, edge.pointB)]
                otherPoints = [triangle.getThirdNode(edge) for triangle in
triangles_to_remove]
                for triangle in triangles_to_remove:
                    self.__remove_triangle(triangle)

                nodes = [self.__dtree.getNodeWithTriangle(triangle) for triangle
in triangles_to_remove]
                otherParent = nodes[1] if nodes[0].triangle == treenode.triangle
else nodes[0]

                for point in otherPoints:
                    self.__addNewTriangle(Triangle(point,    edge.pointA,    p),
treenode, otherParent)
                    self.__addNewTriangle(Triangle(point,    edge.pointB,    p),
treenode, otherParent)

                self.__step = self.__step + 1

                for point in otherPoints:
                    self.__fix_edge(p, Vector(edge.pointA, point))
                    self.__fix_edge(p, Vector(edge.pointB, point))

            rest_points = np.delete(rest_points, 0, axis=0)

        self.__triangles = [x for x in self.__triangles if ((x.point(-1) is None)
and (x.point(-2) is None))]
        return self.__triangles
```

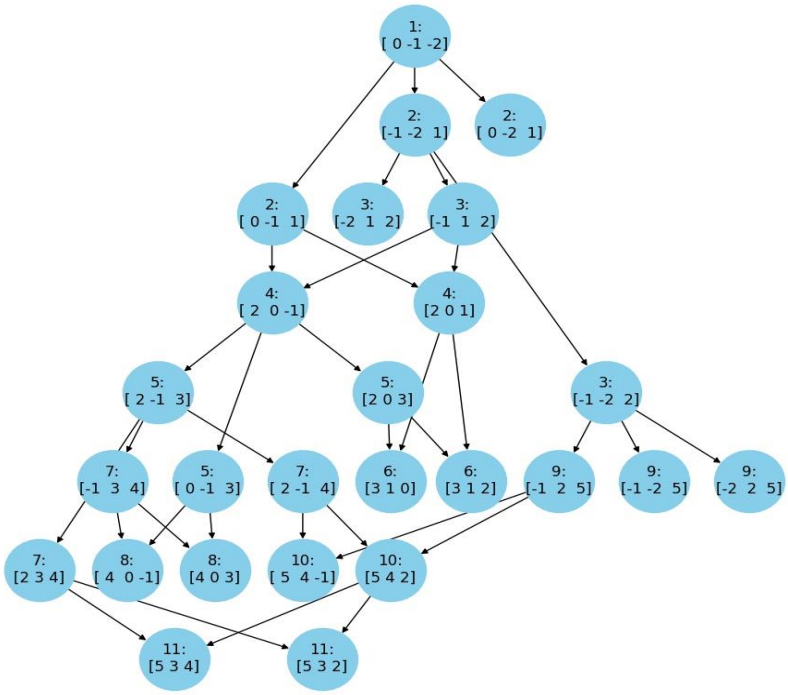
3.4 Παραδείγματα Εκτέλεσης Αλγορίθμου

Στην ενότητα αυτή παρουσιάζονται τα αποτελέσματα από την εκτέλεση διαφόρων συνόλων δισδιάστατων σημείων. Παρουσιάζεται η διαδικασία για 2 μικρού μεγέθους σύνολα σημείων και για 4 μεγάλου μεγέθους. Συγκεκριμένα, για τα μικρά σύνολα 6 , 8 εκτός από το τελικό αποτέλεσμα παρουσιάζονται το δέντρο τριγώνων για την απεικόνιση των ενδιάμεσων τριγώνων κατά την εκτέλεση του αλγορίθμου. Επιπλέον στο σύνολο 6 σημείων παρουσιάζονται και τα επιπλέον βήματα εκτέλεσης του αλγορίθμου.

3.4.1 Αποτελέσματα Αλγορίθμου για 6 τυχαία σημεία

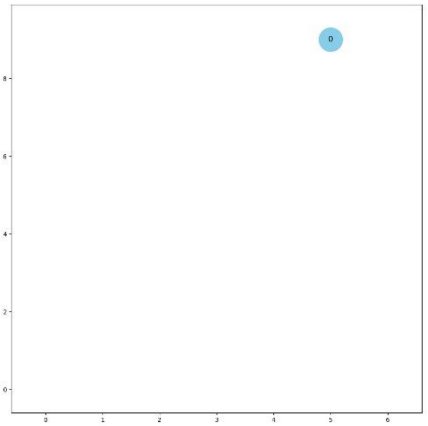


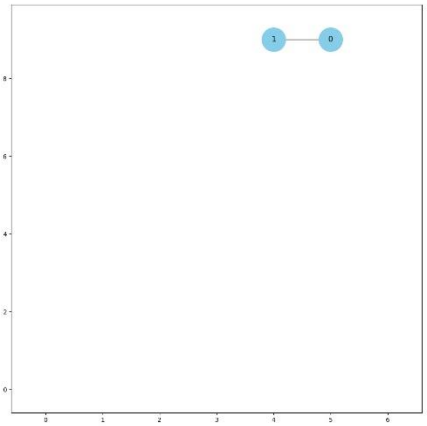
Σχήμα 3.4.1.1: Τελικός Τριγωνισμός Αλγόριθμου 6 Τυχαίων Σημείων

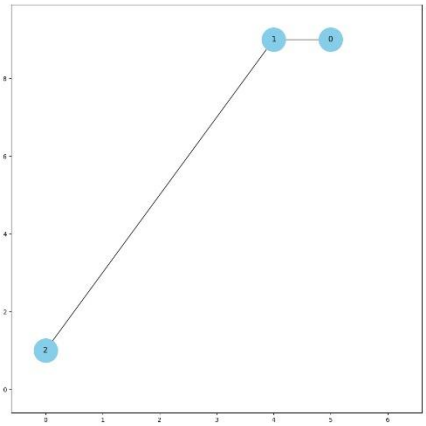


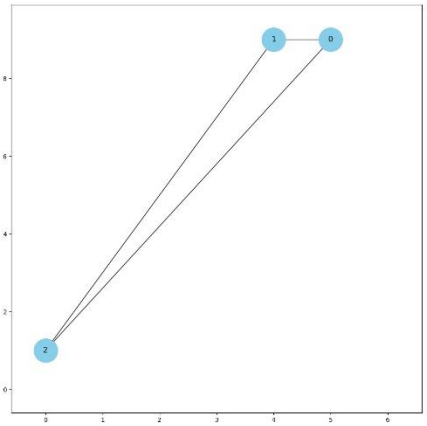
Σχήμα 3.4.1.2: Δέντρο Τριγώνων 6 Τυχαίων Σημείων

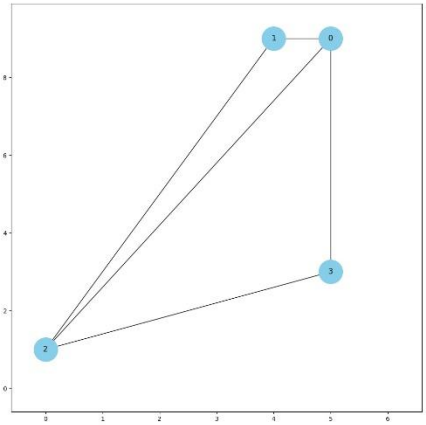
3.4.2 Ενδιάμεσα Βήματα Εκτέλεσης του Αλγορίθμου

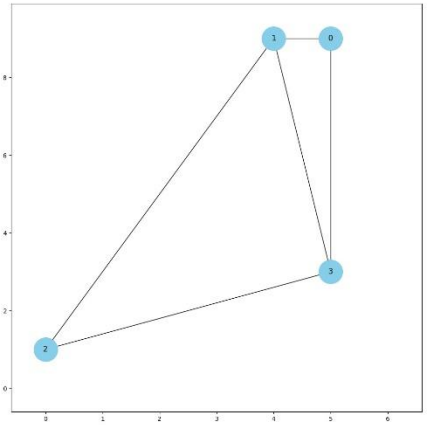


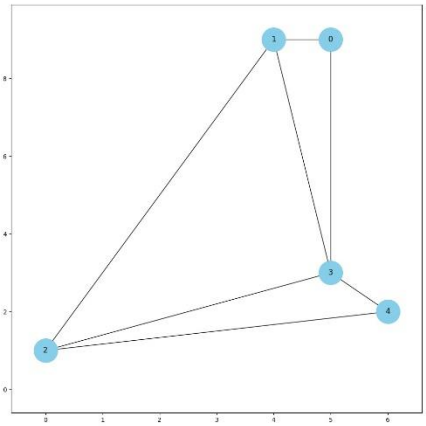


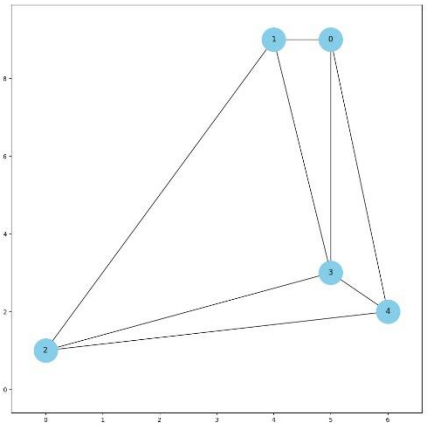

1
0
2

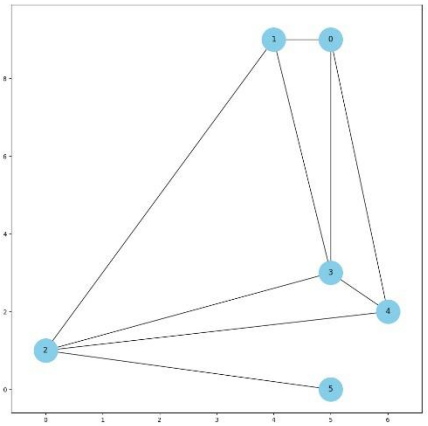

1
0
2

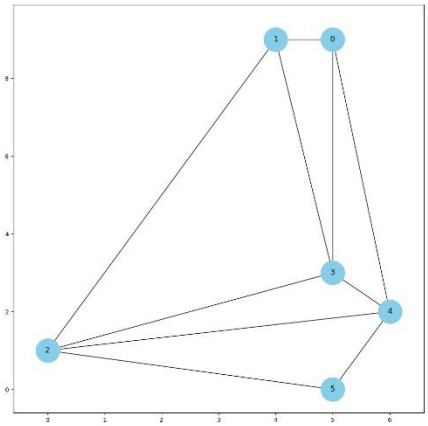

1
0
3
2

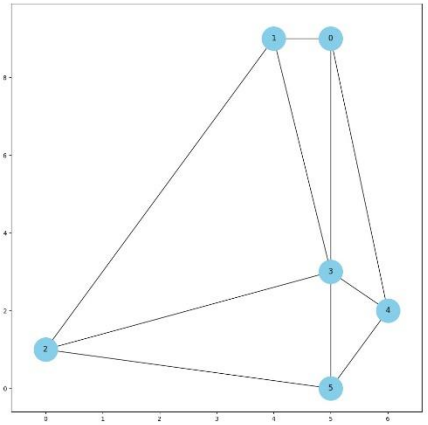

1
0
3
2


1
0
3
4
2

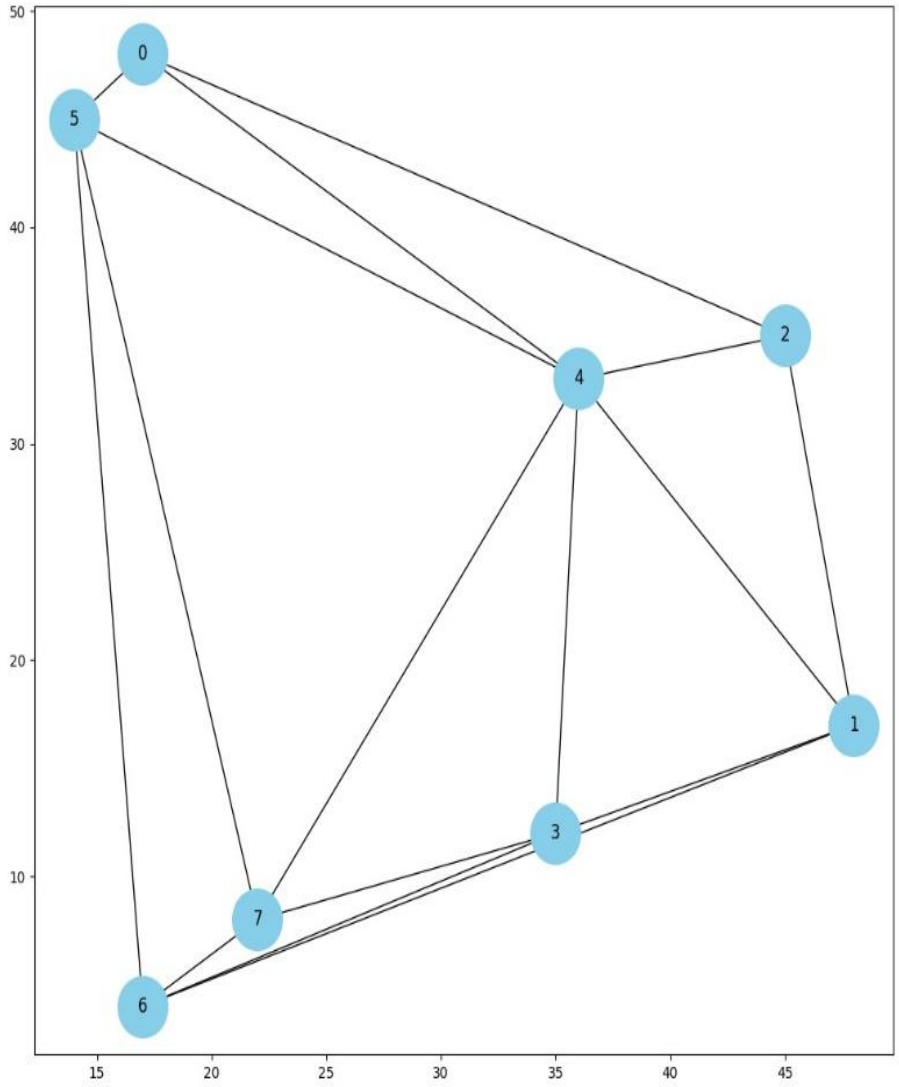

1
0
3
4
2


1
0
3
4
2
5

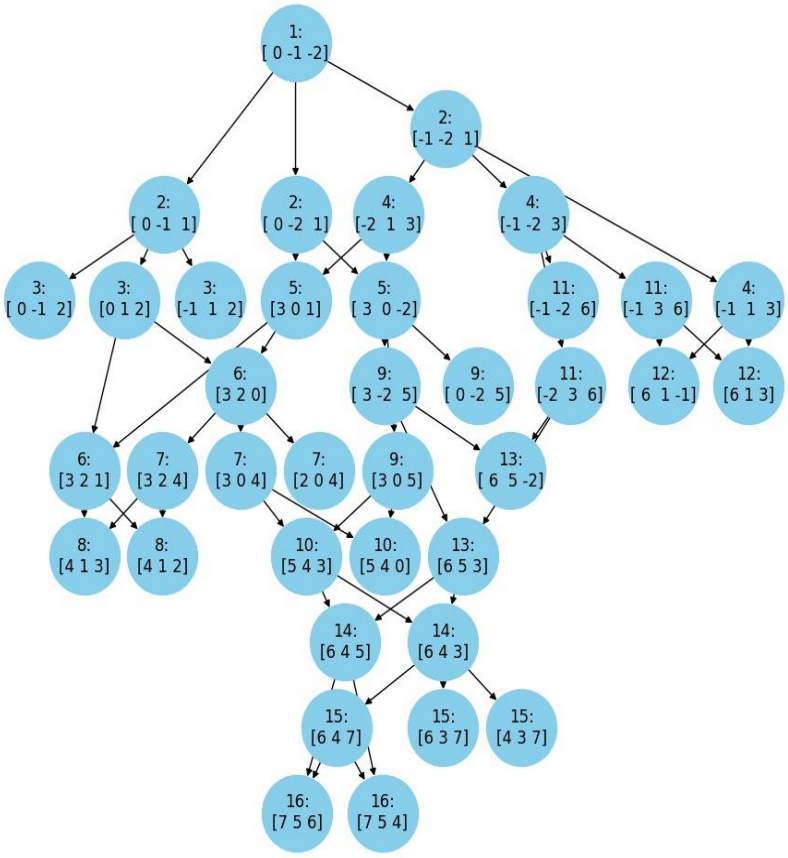

1
0
3
4
2
5


1
0
3
4
2
5

3.4.3 Αποτελέσματα Αλγορίθμου για 8 τυχαία σημεία

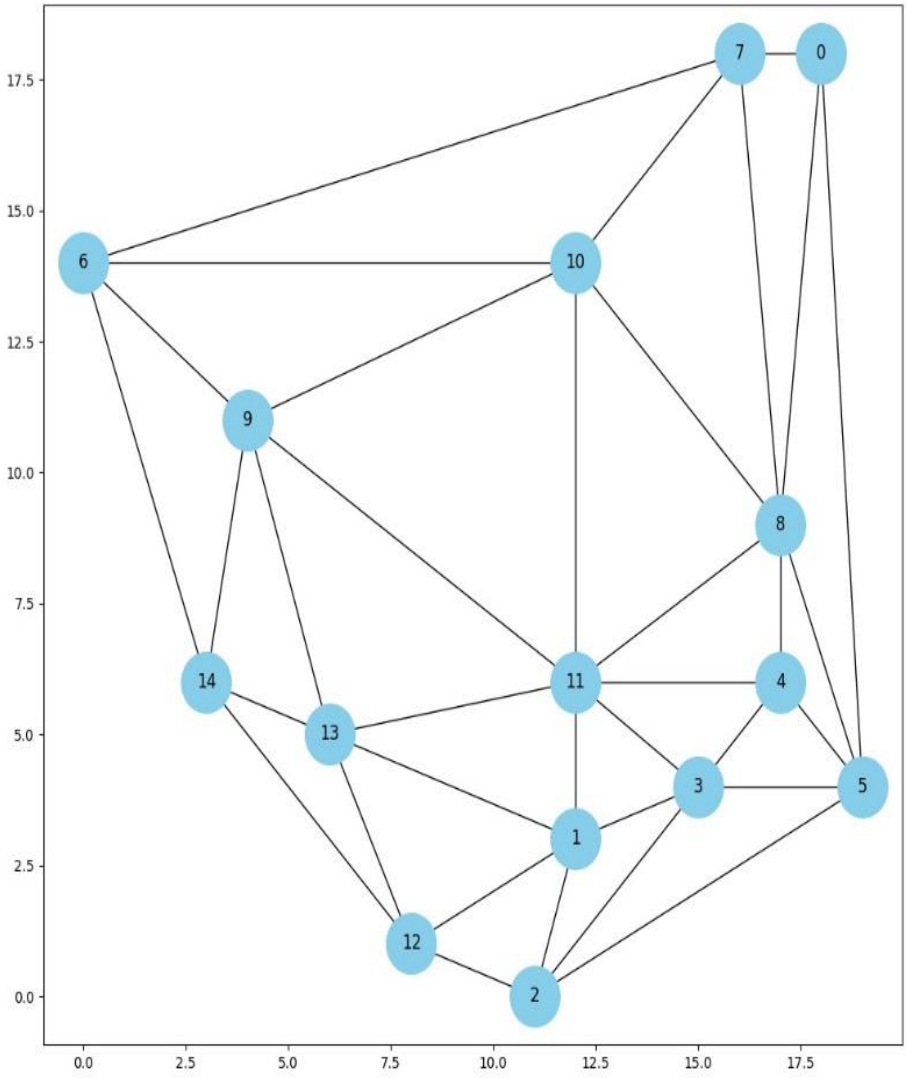


Σχήμα 3.4.2.1: Τελικός Τριγωνισμός Αλγορίθμου 8 Τυχαίων Σημείων



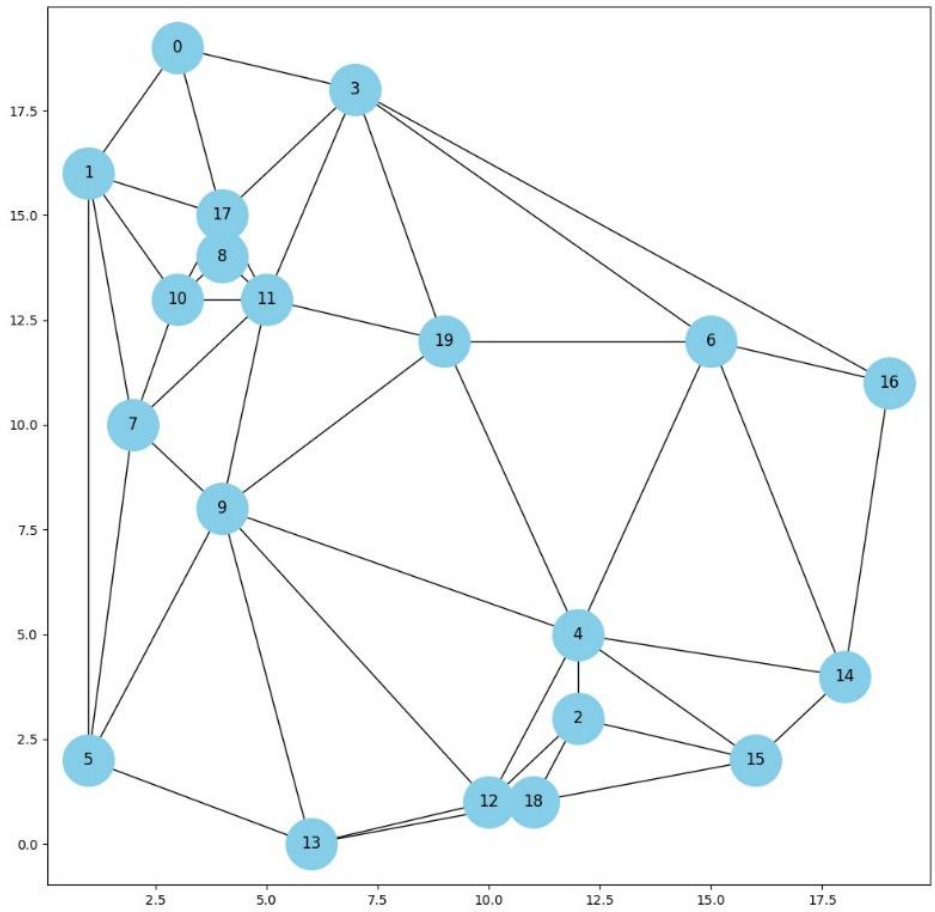
Σχήμα 3.4.2.2: Δέντρο Τριγώνων 8 Τυχαίων Σημείων

3.4.4 Αποτελέσματα Αλγορίθμου για 15 Τυχαία Σημεία



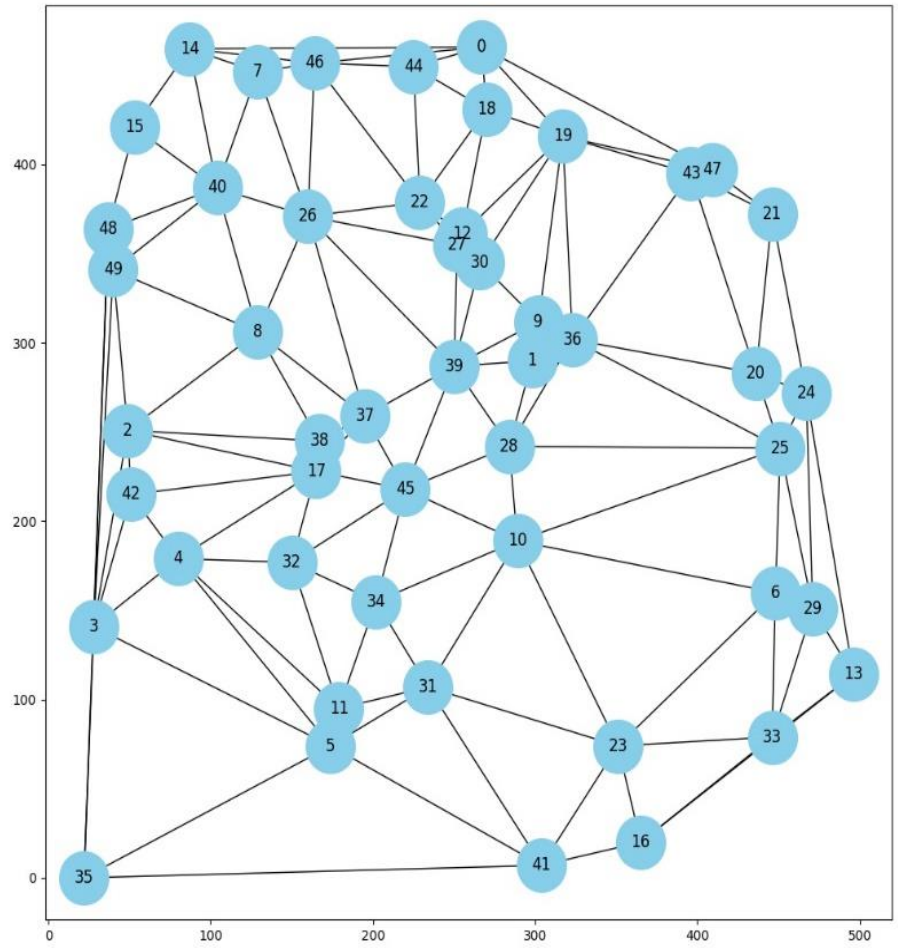
Σχήμα 3.4.3: Τελικός Τριγωνισμός Αλγόριθμου 15 Τυχαίων Σημείων

3.4.5 Αποτελέσματα Αλγορίθμου για 20 τυχαία σημεία



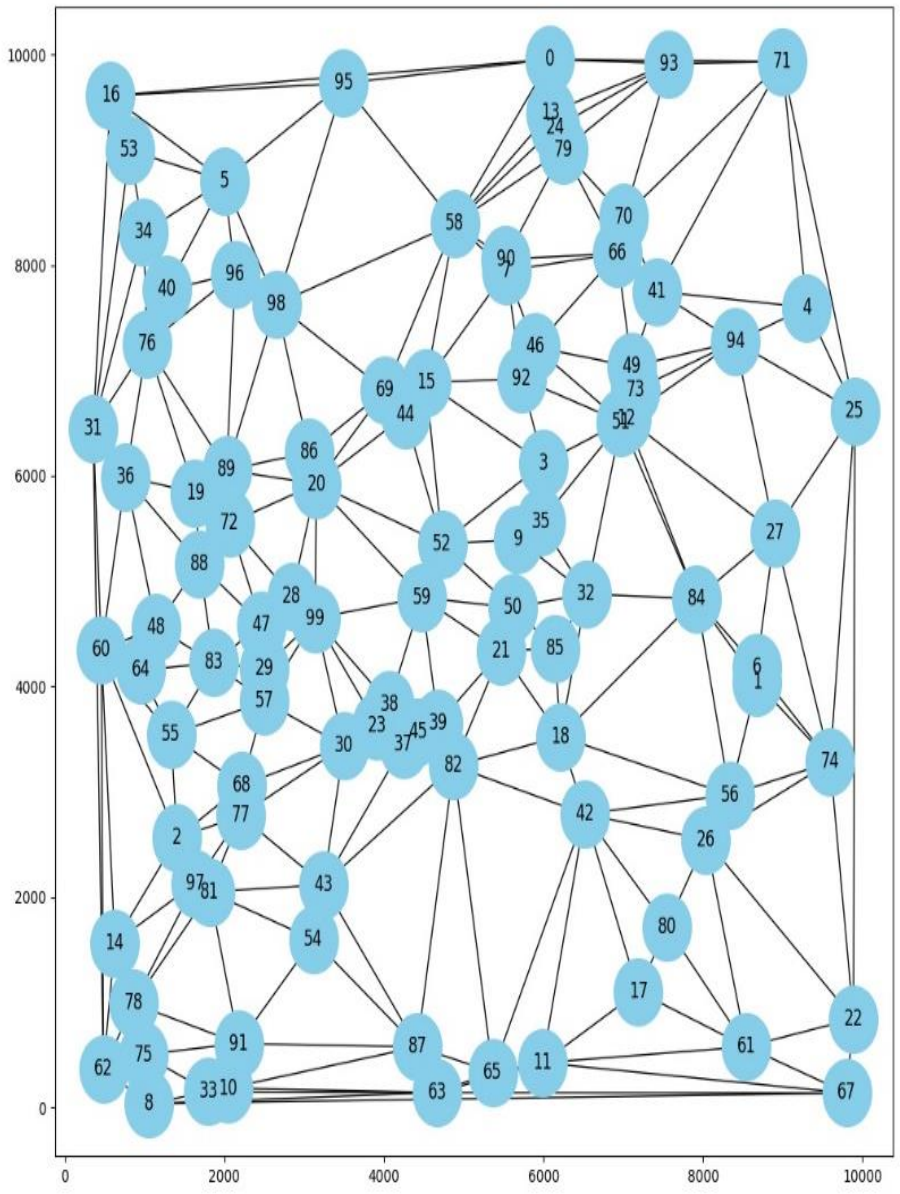
Σχήμα 3.4.4: Τελικός Τριγωνισμός Αλγόριθμου 20 Τυχαίων Σημείων

3.4.6 Αποτελέσματα Αλγορίθμου για 50 τυχαία σημεία



Σχήμα 3.4.5: Τελικός Τριγωνισμός Αλγόριθμου 50 Τυχαίων Σημείων

3.4.7 Αποτελέσματα Αλγορίθμου για 100 τυχαία σημεία



Σχήμα 3.4.6: Τελικός Τριγωνισμός Αλγόριθμου 100 Τυχαίων Σημείων

Κεφάλαιο 4. Συμπεράσματα Επεκτάσεις

4.1 Σύνοψη

Σκοπός της συγκεκριμένης διπλωματικής εργασίας ήταν η υλοποίηση του αλγορίθμου των Guibas, Knuth και Sharir , ο οποίος λαμβάνοντας ως είσοδο διάφορα τυχαία σημεία μπορεί να εξάγει ως έξοδο ακολουθίες εικόνων με τα βήματα του αλγορίθμου. Ο αλγόριθμος που υλοποιήθηκε παρέχει στο χρήστη τη δυνατότητα στο χρήστη να δημιουργήσει ένα τυχαίο σύνολο δεδομένων το όποιο θα χρησιμοποίησει για την εκτέλεση του αλγορίθμου και να εξάγει τις ακολουθίες εικόνων με τα βήματα εκτέλεσης του αλγορίθμου.

4.2 Επεκτάσεις

Υπάρχουν πολλά πράγματα που έχει ενδιαφέρον να υλοποιηθούν στο μέλλον. Τα σημεία που δίνονται ως είσοδο του τριγωνισμού στη συγκεκριμένη Διπλωματική εργασία διαβάζονται από αρχείο. Θα μπορούσαν τα σημεία που προαναφέραμε να μην διαβάζονται από αρχείο αλλά να υλοποίηθει ένα κατάλληλο γραφικό περιβάλλον στο όποιο ο κέρσορας του ποντικιού να αναγνωρίζει τη θέση του στο χώρο, να διαβάζει το συγκεκριμένο σημείο και να το προσθέτει στο τριγωνισμό.

Βιβλιογραφία

[1] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica, 7:381-413,1992.

[2] M. DE BERG, O. CHEONG, M. VAN KREVELD, M. OVERMARS, Υπολογιστική γεωμετρία : Αλγόριθμοι και Εφαρμογές , Πανεπιστημιακές Εκδόσεις Κρήτης, 2008.

[3] Koch, Thierry de, Marc van Kreveld, and Maarten Laffler: Generating realistic terrains with higher-order Delaunay triangulations. Computational Geometry, 36(1):52 – 65, 2007 http://www.sciencedirect.com/science/article/pii/S092577210600484.

[4] Fortune, Steven: A sweepline algorithm for voronoi diagrams. Algorithmica, 2:153–174, 1987, ISSN 0178-4617. http://dx.doi.org/10.1007/BF01840357

[5] C. L. Lawson. Transforming triangulations. Discrete Math., 3:365–372, 1972.

[6] C. L. Lawson. Software for C 1 surface interpolation. In J. R. Rice, editor, Math. Software III, pages 161–194. Academic Press, New York, 1977.

[7] R. E. Barnhill. Representation and approximation of surfaces. In J. R. Rice, editor, Math. Software III, pages 69–120. Academic Press, New York, 1977.

[8] R. Sibson. Locally equiangular triangulations. Comput. J., 21:243–245, 1978.

[9] S. Rippa. Minimal roughness property of the Delaunay triangulation. Comput. Aided Geom. Design, 7:489–497, 1990.

[10] N. Dyn, D. Levin, and S. Rippa. Data dependent triangulations for piecewise linear interpolation. IMA J. Numer. Anal., 10:137–154, 1990.

[11] E. Quak and L. Schumaker. Cubic spline fitting using data dependent triangulations. Comput. Aided Geom. Design, 7:293–302, 1990.

[12] J. L. Brown. Vertex based data dependent triangulations. Comput. Aided Geom. Design, 8:239–251, 1991.

[13] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. Discrete Comput. Geom., 4:387–421, 1989.

[14] M. I. Shamos. Computational Geometry. Ph.D. thesis, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1978.

[15] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. Systematic Zoology, 18:259–278, 1969.

[16] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. Pattern Recogn., 12:261–268, 1980.

[17] W. Mulzer and G. Rote. Minimum weight triangulation is NP-hard. In Proc. 22nd Annu. ACM Sympos. Comput. Geom., pages 1–10, 2006.

[18] Watson, D. F.: Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. The Computer Journal, 24(2):167–172, February, 1981.

http://comjnl.oxfordjournals.org/cgi/content/abstract/24/2/ 167