ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΑΓΡΟΝΟΜΩΝ ΤΟΠΟΓΡΑΦΩΝ ΜΗΧΑΝΙΚΩΝ
ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΓΕΩΠΛΗΡΟΦΟΡΙΚΗ

**ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

# ΑΝΑΠΤΥΞΗ ΕΦΑΡΜΟΓΗΣ ΔΙΑΔΙΚΤΥΟΥ ΓΙΑ ΤΟΝ ΣΧΕΔΙΑΣΜΟ ΒΕΛΤΙΣΤΟΥ ΙΣΤΙΟΠΛΟΪΚΟΥ ΠΛΟΥ ΒΑΣΕΙ ΚΑΙΡΙΚΩΝ ΣΥΝΘΗΚΩΝ

WEB APPLICATION DEVELOPMENT OF
OPTIMUM SAILING ROUTE PLANNING
UNDER WEATHER CONDITIONS

**Στυλιανός Μ. Κονταρίνης**
Διατμηματικό Μεταπτυχιακό Πρόγραμμα "Γεωπληροφορική"
Ακαδ. Έτος 2014-15

Επιβλέπων Καθηγητής: **καθ. Λύσανδρος Τσούλος**

Θέλω να ευχαριστήσω θερμά τον επιβλέποντα της μεταπτυχιακής μου εργασίας, Καθηγητή κ. Λύσανδρο Τσούλο, για την υπομονή, την καθοδήγηση και τις πολύτιμες συμβουλές του καθόλη τη διάρκεια της εκπόνησής της. Επίσης, θέλω να ευχαριστήσω τον Δρ. Βύρωνα Αντωνίου, επιστημονικό συνεργάτη του Εργαστηρίου Χαρτογραφίας για τις πρακτικές συμβουλές που μου έδωσε και με βοήθησαν να απλοποιήσω θέματα που αντιμετώπιζα και να προχωρήσω σε επόμενα βήματα. Τέλος θέλω να ευχαριστήσω τον δάσκαλο μου στη ιστιοπλοΐα «καπετάνιο» Σάκη Κουτσουβέλη, ο οποίος μου μετέδωσε πριν από 15 περίπου χρόνια την απέραντη αγάπη του για τη ναυτική τέχνη.

Στην Οικογένεια μου

# ΠΕΡΙΕΧΟΜΕΝΑ

## ΠΙΝΑΚΑΣ ΕΙΚΟΝΩΝ

## ΠΡΟΟΙΜΙΟ

Με την παρούσα μεταπτυχιακή εργασία υλοποιήθηκε σύστημα που πραγματοποιεί παραμετρικό σχεδιασμό ιστιοπλοϊκού πλου, με χρήση ανοικτού λογισμικού, εφαρμόζοντας σύγχρονες μεθόδους γεωπληροφορικής σε μια πεπερασμένη περιοχή, όπως είναι η Ελλάδα με έντονα νησιωτικά συμπλέγματα. Η υλοποίηση αφορά τον αυτόματο σχεδιασμό του βέλτιστου πλου με αποτύπωση διαδρομών με την ελάχιστη συνολική απόσταση καθώς και με τον ελάχιστο χρόνο πλεύσης βάσει καιρικών συνθηκών.

Στο πλαίσιο της εργασίας μελετήθηκαν διεξοδικά αφενός οι αλγόριθμοι εύρεσης συντομότερων διαδρομών, αφετέρου η επίπτωση των καιρικών συνθηκών σε ιστιοπλοϊκό πλου και έγινε η σχετική μοντελοποίηση τους, με αποτέλεσμα την ανάπτυξη εφαρμογής διαδικτύου για το σχεδιασμό βέλτιστου ιστιοπλοϊκού πλου για θαλάσσιες περιοχές με πολυνησία όπως της Ελλάδας, της Καραϊβικής, της Κροατίας, κ.α.

## Κεφάλαιο 1 – Εισαγωγή

# Περιγραφή Ανάγκης

Ο σχεδιασμός και η αποτύπωση του πλου είναι μια πρακτική ανάγκη που απαντιέται από τα αρχαία κιόλας χρόνια. Είναι σε όλους μας γνωστή η «Αργοναυτική Εκστρατεία» και είναι αρκετές οι προσπάθειες αποτύπωσής της σε χάρτη. Προτού λύσει τους κάβους ο ναυτικός έχει προηγουμένως σχεδιάσει είτε πάνω στο χάρτη, είτε και στο μυαλό του ακόμη, από ποια σημεία θα χρειαστεί να περάσει προκειμένου να φτάσει στον προορισμό του.

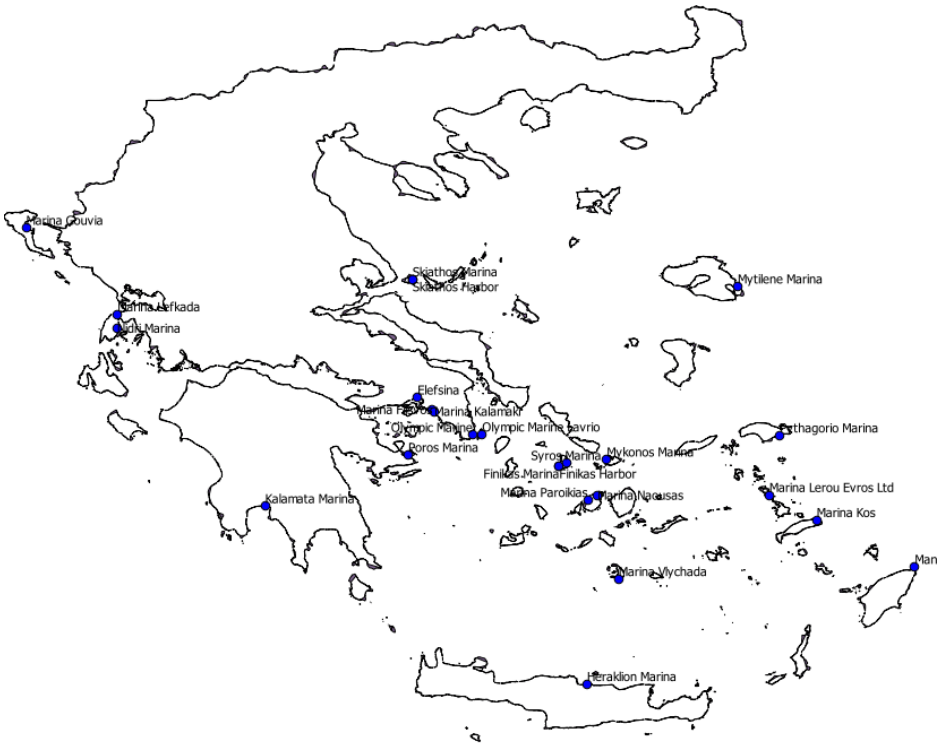

**Εικόνα 1 - Διαδρομή "Αργοναυτικής Εκστρατείας"[1]**

Στην Ελλάδα υπάρχουν πολυάριθμα σημεία θαλάσσιου τουριστικού ενδιαφέροντος όπου σκάφη αναψυχής δύναται να αγκυροβολήσουν, όπως λιμάνια, μαρίνες, αλιευτικά καταφύγια καθώς και άλλα για να κολυμπήσουν, όπως παραλίες, όρμοι κ.α. που οι θαλάσσιοι επισκέπτες αναζητούν να επισκευθούν κατά τη διάρκεια των διακοπών τους. Συνήθως η επιλογή των σημείων γίνεται με βάση τα ενδιαφέροντα του κάθε επισκέπτη και με γνώμονα την μεγιστοποίηση του οφέλους σε σχέση με αυτά. Τα ενδιαφέροντα αυτά μπορούν να κατηγοριοποιηθούν σε πεδία ενδιαφερόντων όπως ο πολιτισμός, ο αθλητισμός, η φυσιολατρία, η γαστρονομία, η διασκέδαση ακόμη και η χαλάρωση καθώς και συνδυασμός αυτών.

Στην Ελλάδα έχουν καταγραφεί[2]:

[1]https://peripluscd.wordpress.com/2013/10/
[2]http:// sail-la-vie.com/discover/sailing-in-Greece

- 71 Μαρίνες (από τις οποίες 26 είναι οργανωμένες βάσεις ενοικίασης σκαφών αναψυχής)
- 615 Αγκυροβόλια
- 2444 Παραλίες




**Εικόνα 2 - Οργανωμένες Μαρίνες στην Ελλάδα**

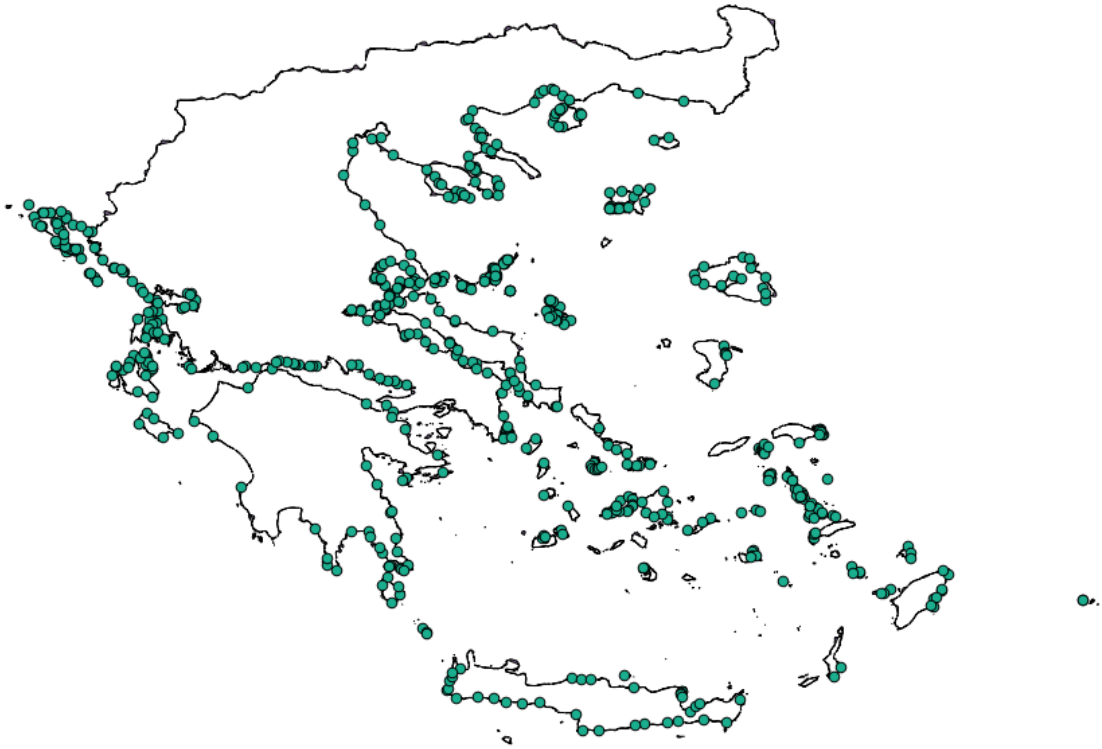
# Σχεδιασμός Ιστιοπλοϊκού Πλου

Τίθεται συνεπώς το ερώτημα για την ανάγκη του θαλάσσιου επισκέπτη που επιθυμεί να οργανώσει το ταξίδι του, αν μπορεί να επιλέγει «Σημεία Ενδιαφέροντος» (Points of Interest - POIs) και να υπάρχει ένα σύστημα το οποίο να τον βοηθά να σχεδιάζει τη βέλτιστη διαδρομή που θα περνά από τα σημεία αυτά, δίνοντάς του πληροφορίες για την εξαγόμενη διαδρομή όπως:

- Συνολική Απόσταση
- Συνολικό Χρόνο Πλεύσης
- Καιρικές Συνθήκες που επικρατούν στην περίοδο που επιθυμεί να ταξιδέψει
- και άλλες παραμέτρους.

Τη διαδρομή αυτή (Itinerary ή Route) θα μπορεί να την χρησιμοποιεί ως οδηγό κατά τη διάρκεια του πλου για την πλοήγησή του με την εισαγωγή των σημείων αλλαγής πορείας (Waypoints) σε μια συσκευή GPS.

**Εικόνα 3 - Μαρίνες στην Ελλάδα**



**Εικόνα 4 - Αγκυροβόλια στην Ελλάδα**

Εικόνα 5 - Παραλίες στην Ελλάδα

# Σκοπός της Εργασίας

Σκοπός της παρούσας εργασίας είναι η ανάπτυξη εφαρμογής λογισμικού με τίτλο «**Σύστημα Σχεδιασμού Ιστιοπλοϊκής Διαδρομής**», με τη βοήθεια του οποίου ο χρήστης θα μπορεί να επιλέγει σημεία ενδιαφέροντος πάνω στο χάρτη της Ελλάδας, και η εφαρμογή να σχεδιάζει τη βέλτιστη διαδρομή που θα περνά από τα σημεία αυτά.

Κατά τη διάρκεια εκπόνησης της εργασίας αναλύθηκαν οι παράμετροι της εφαρμογής όπως:

- η απόσταση μεταξύ των ενδιάμεσων σημείων αγκυροβολίας καθώς και
- ο χρόνος που θα χρειαστεί να πλεύσει το σκάφος με βέλτιστη ταχύτητα πλεύσης, δεδομένων των καιρικών συνθηκών που επικρατούν στην περιοχή (είτε βάσει ιστορικών στοιχείων, είτε βάσει πρόγνωσης καιρού).

Στο πλαίσιο της εργασίας έγινε επισκόπηση των πιο διαδεδομένων αλγορίθμων εύρεσης βέλτιστων διαδρομών και επιλογή αυτού που θα εφαρμόζεται στο σύστημα. Οι πιο διαδεδομένοι αλγόριθμοι για την εύρεση βέλτιστης διαδρομής που μελετήθηκαν είναι:

- Dijkstra
- A*
- Breadth First Search
- Best First Search
- IDA*

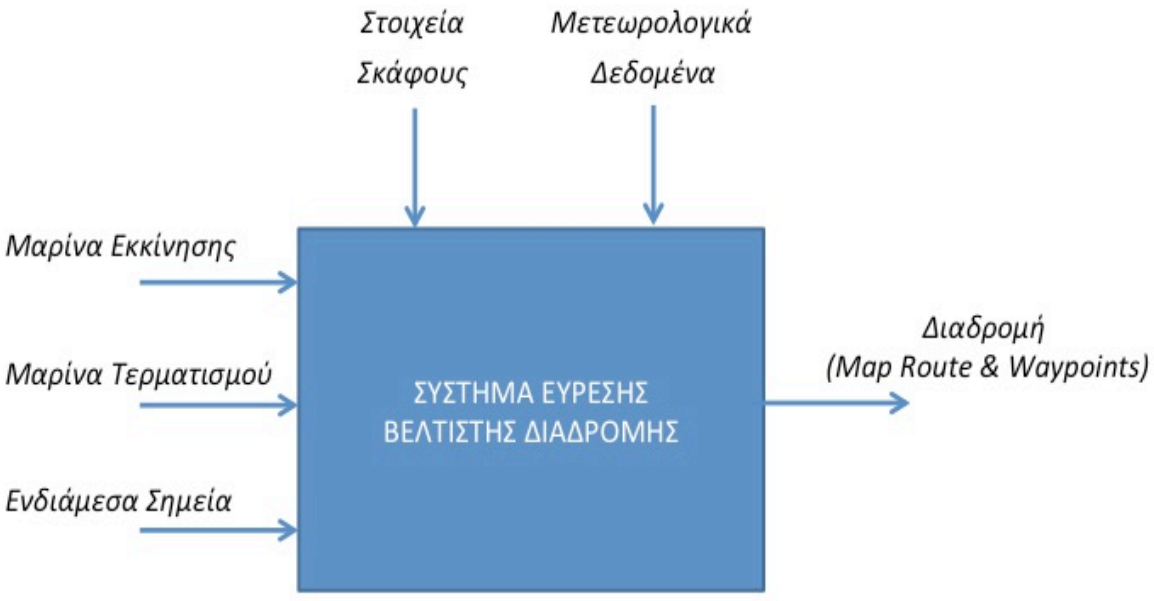
και η περιγραφή τους γίνεται στο 3ο Κεφάλαιο.

# Προσέγγιση Υλοποίησης

Η προσέγγιση της υλοποίησης βασίστηκε στη δημιουργία πλέγματος (grid) για τη θαλάσσια περιοχή της Ελλάδας που έγινε με τη βοήθεια του περιβάλλοντος ανοικτού λογισμικού QGIS. Με τη χρήση του πλέγματος αναπτύχθηκε κώδικας όπου εφαρμόστηκε ο αλγόριθμος βέλτιστης διαδρομής για τα σημεία που ενδιαφέρεται ο χρήστης.

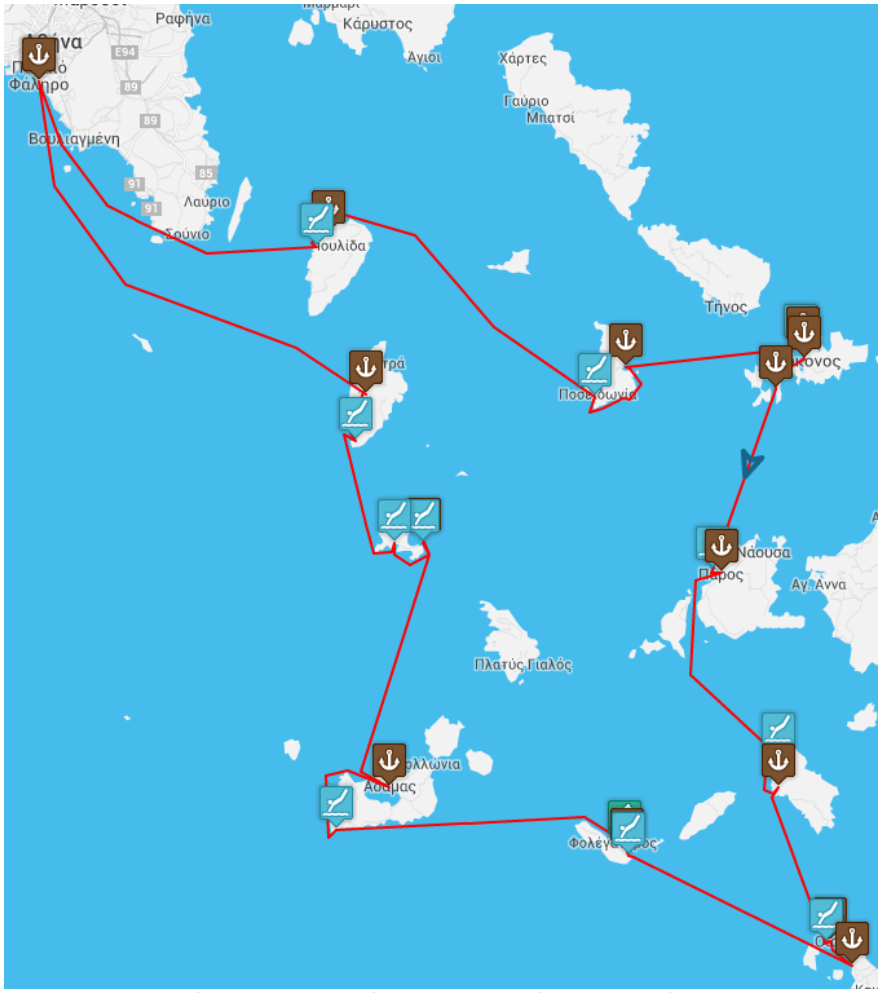
Εικόνα 6 - Θαλάσσια Περιοχή της Ελλάδας

Το αποτέλεσμα είναι να σχεδιάζεται αυτόματα μια διαδρομή (route) την οποία μπορεί ο χρήστης να κατεβάσει στο κινητό του και να την χρησιμοποιήσει με μια εφαρμογή GPS ως οδηγό για το ταξίδι του.




Εικόνα 7 - Είσοδοι, παράμετροι και έξοδος Συστήματος

Το αποτέλεσμα του συστήματος εμφανίζεται σε webmap όπως στην παρακάτω εικόνα:




**Εικόνα 8 – Προτεινόμενη Διαδρομή στις Κυκλάδες[3]**

# Περιεχόμενα & Μεθοδολογία της Εργασίας

Στο **Κεφάλαιο 2** εισάγονται βασικές έννοιες από τη θεωρία των Γράφων καθώς και κάποιων σύνθετων δομών δεδομένων, που βοηθούν στην κατανόηση των αλγορίθμων εύρεσης βέλτιστων διαδρομών. Στο **Κεφάλαιο 3** περιγράφονται οι πιο διαδεδομένοι αλγόριθμοι εύρεσης βέλτιστης διαδρομής και γίνεται μια σύντομη αναφορά στην πολυπλοκότητά τους ενώ στο **Κεφάλαιο 4** παρουσιάζονται τα βασικά εργαλεία Ανοιχτού Κώδικα που χρησιμοποιήθηκαν για την ανάπτυξη του συστήματος.

Στο **Κεφάλαιο 5** περιγράφονται η ανάπτυξη της πρωτότυπης διάταξης, της πειραματικής διάταξης καθώς και τα χαρακτηριστικά, η μεθοδολογία και τα προβλήματα που προέκυψαν για τη δημιουργία του Πλέγματος. Για την πειραματική διάταξη δημιουργήθηκαν πλέγματα για τις Κυκλάδες με διαφορετική ανάλυση (0,6ν.μ., 0,3ν.μ., 0,15ν.μ. & 0,06ν.μ.) πάνω στα οποία έγινε και η συγκριτική ανάλυση της εφαρμογής των αλγορίθμων (**Κεφάλαιο 6**). Για το τελικό σύστημα δημιουργήθηκε πλέγμα με ανάλυση 0,6ν.μ. για όλη τη θαλάσσια περιοχή της Ελλάδας.

[3]http://sail-la-vie.com/plan/routes/140/Cyclades%20Unexplored

Στο **Κεφάλαιο 7** περιγράφεται η προσέγγιση για την επίλυση του προβλήματος η διαδρομή του πλου να περνά από όλα τα επιθυμητά σημεία, το οποίο επετεύχθη με την εφαρμογή αλγόριθμου επίλυσης του Προβλήματος Πλανόδιου Πωλητή (TSP).

Στο **Κεφάλαιο 8** αναλύεται η λογική πλεύσης των ιστιοπλοϊκών σκαφών με τη δύναμη του ανέμου. Περιγράφονται τα Πολικά Διαγράμματα των Σκαφών που δίνουν οι κατασκευαστές και ο τρόπος με τον οποίο έγινε η αλγεβρική τους μοντελοποίηση προκειμένου να ενταχθεί στον αλγόριθμο του Συστήματος. Επίσης περιγράφεται ο τρόπος με τον οποίο έγινε η ενημέρωση του Grid με στοιχεία Καιρού.

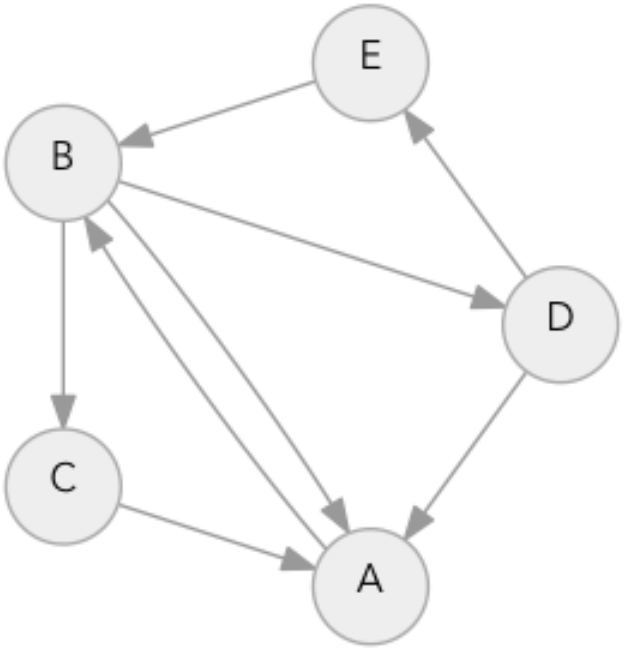
Στο **τελευταίο Κεφάλαιο** περιγράφονται δυνατότητες βελτίωσης του συστήματος μέσα από βελτιστοποιήσεις των αλγορίθμων, δυνατότητες εφαρμογής του συστήματος και σε άλλες προσεγγίσεις και γίνεται μια τελική σύνοψη από τα συμπεράσματα της εργασίας.

## Κεφάλαιο 2 – Βασικές Έννοιες Γράφων & Δομές Δεδομένων Αλγορίθμων

Προκειμένου να μελετηθούν οι Αλγόριθμοι Εύρεσης Βέλτιστων Διαδρομών θα εισαχθούν κάποιες βασικές έννοιες, από τη θεωρία των Γράφων και κάποιων σύνθετων δομών δεδομένων που θα βοηθήσουν στο επόμενο κεφάλαιο ως προς τη θεωρητική κατανόηση των αλγορίθμων καθώς των μεθόδων υλοποίησής τους.

# Γράφος (Graph)

Ένας Γράφος αποτελείται από Ακμές ή Πλευρές (στην αγγλική βιβλιογραφία αναφέρονται ως Edges, Branches, Links, Connections, Arrows ή και Arcs) και Κόμβους ή Κορυφές (στα αγγλικά απαντώνται ως Nodes, Vertices ή και Objects), όπως φαίνεται στην παρακάτω εικόνα.



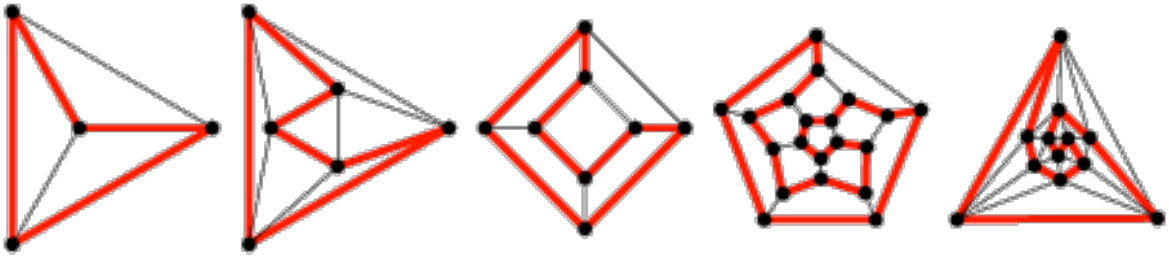

**Εικόνα 9 - Ένας Τυπικός Γράφος με 5 Κόμβους και 8 Ακμές**

Ο Γράφος μαθηματικά συμβολίζεται με G = (V, E) και ορίζεται ως ένα σύνολο από κόμβους όπου V={1,…,n} και ένα σύνολο από ακμές όπου $E \subseteq V \times V$. Οι ακμές είναι ένα διατεταγμένο ζεύγος με στοιχεία κόμβους, όπου εν γένει το (vi, vj) δεν είναι το ίδιο με το (vj, vi). Γράφεται (vi, vj) και εννοείται ότι ο κόμβος vi 'επικοινωνεί' με τον κόμβο vj. Οι κόμβοι vi και vj λέγονται άκρα της ακμής (vi, vj). Οι κόμβοι vi και vj λέγονται γείτονες αν υπάρχει η ακμή (vi, vj). Αν ο κόμβος vi έχει d γείτονες τότε λέγεται πως έχει βαθμό d.

Κατευθυνόμενος (**directed graph**) ονομάζεται ο γράφος όταν υπάρχει η ακμή (vs, vd) ενώ δεν υπάρχει η ακμή (vd, vs). Στην κατευθυνόμενη ακμή (vs, vd) η κορυφή vs λέγεται πηγή (source) και η vd προορισμός (destination). Η ακμή vd είναι γείτονας (neighbor) της vs αν υπάρχει η κατευθυνόμενη πλευρά (vs, vd).

# Μονοπάτι (Path)

Ένα Μονοπάτι μήκους k είναι μια ακολουθία από κόμβους P=(v0, v1,…,vk), έτσι ώστε να υπάρχει ακμή (vi, vi+1) για κάθε 0<=i<=k-1 και όλες οι ακμές να είναι μεταξύ τους διαφορετικές.



Εικόνα 10–Παραδείγματα Μονοπατιών σε Γράφο[4]

Ο Γράφος με Βάρη(**weighted graph**) συσχετίζει ένα υπολογιζόμενο βάρος ή κόστος με κάθε ακμή του γράφου. Τα βάρη είναι συνήθως πραγματικοί αριθμοί, αν και μπορούν να περιοριστούν σε ακέραιους ή θετικούς αριθμούς. Το βάρος μπορεί να μοντελοποιεί απόσταση, μήκος, χρόνο, κόστος, χωρητικότητα κ.ά.

Το βάρος ενός μονοπατιού σε ένα weighted graph είναι το άθροισμα των επιμέρους βαρών των ακμών του μονοπατιού. Ένα ζευγάρι κόμβων που δεν διασυνδέονται μπορεί να αναπαρασταθεί με βάρος άπειρο. Στη θεωρία Γράφων, ο γράφος με βάρη ονομάζεται και δίκτυο (network). Κλασσικά προβλήματα δικτύων είναι[5]:

- **minimum spanning tree** (εύρεσης του ελάχιστου διασυνδετικού ή γεννητικού δέντρου). Το πρόβλημα συνίσταται στην εύρεση ενός δέντρου που να περιλαμβάνει όλους τους κόμβους και ένα υποσύνολο των ακμών του γράφου, τέτοιο ώστε το άθροισμα των βαρών τους να είναι το ελάχιστο δυνατό.
- **shortest path** (συντομότερου μονοπατιού). Είναι το πρόβλημα εύρεσης μονοπατιού μεταξύ δύο κόμβων σε ένα γράφο, έτσι ώστε το άθροισμα των βαρών των ακμών που το αποτελούν να είναι το ελάχιστο δυνατό.
- **maximal flow** (εύρεσης μέγιστης ροής). Είναι το πρόβλημα εύρεσης, με δεδομένο ένα δίκτυο ροής, μιας ροής με τη μέγιστη δυνατή τιμή.

Οι αλγόριθμοι εύρεσης συντομότερου μονοπατιού όπως αυτός του Dijkstra και του A* εφαρμόζονται σε **weighted directed graphs**.

# Πλέγμα (Grid)

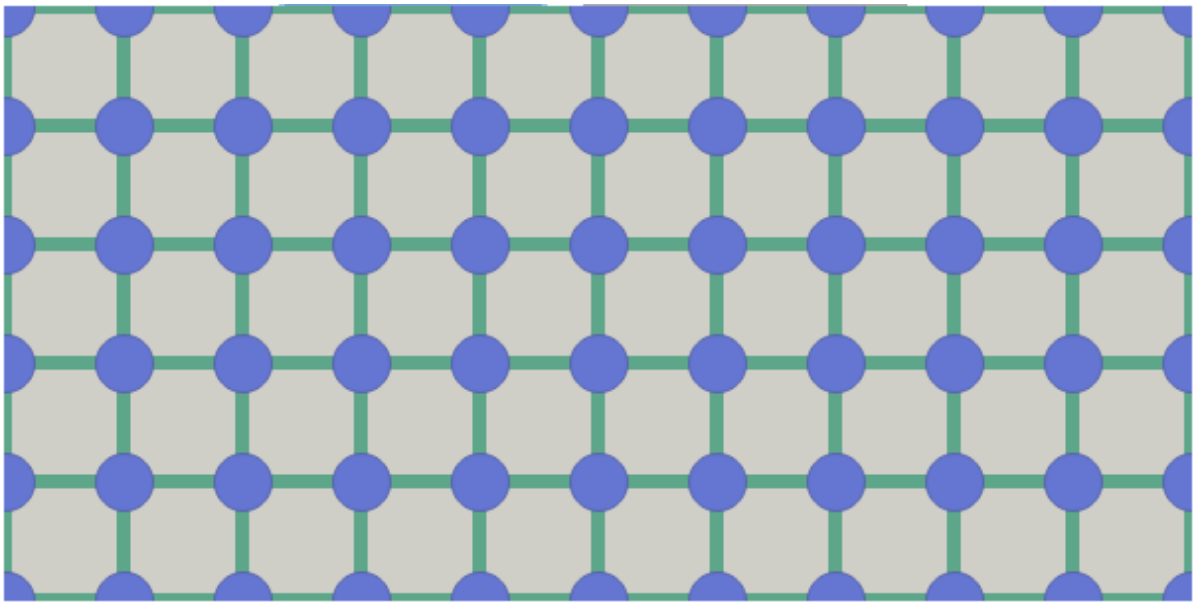
Το Πλέγμα είναι ένα Γράφος, ο οποίος έχει τη μορφή πλέγματος $n \times m$. Ένα γράφημα πλέγματος είναι ένα γράφος του οποίου το σύνολο των κόμβων είναι το σύνολο όλων των διατεταγμένων ζευγών των φυσικών αριθμών $(i, j)$, όπου $1 \le i \le n$ και $1 \le j \le m$. Οι κόμβοι $(i, j)$ και $(k, l)$ συνδέονται μεταξύ τους αν και μόνο αν $|i - k| + |j - l| = 1$. Αυτό σημαίνει ότι, ο κάθε κόμβος συνδέεται με τους 4 γειτονικούς του κόμβους (πάνω-κάτω

[4]http://mathworld.wolfram.com/HamiltonianGraph.html
[5]https://en.wikipedia.org/wiki/Glossary_of_graph_theory

και δεξιά-αριστερά), όπως φαίνεται και στο παρακάτω σχήμα, πλην αυτών των κόμβων που είναι στο σύνορο.

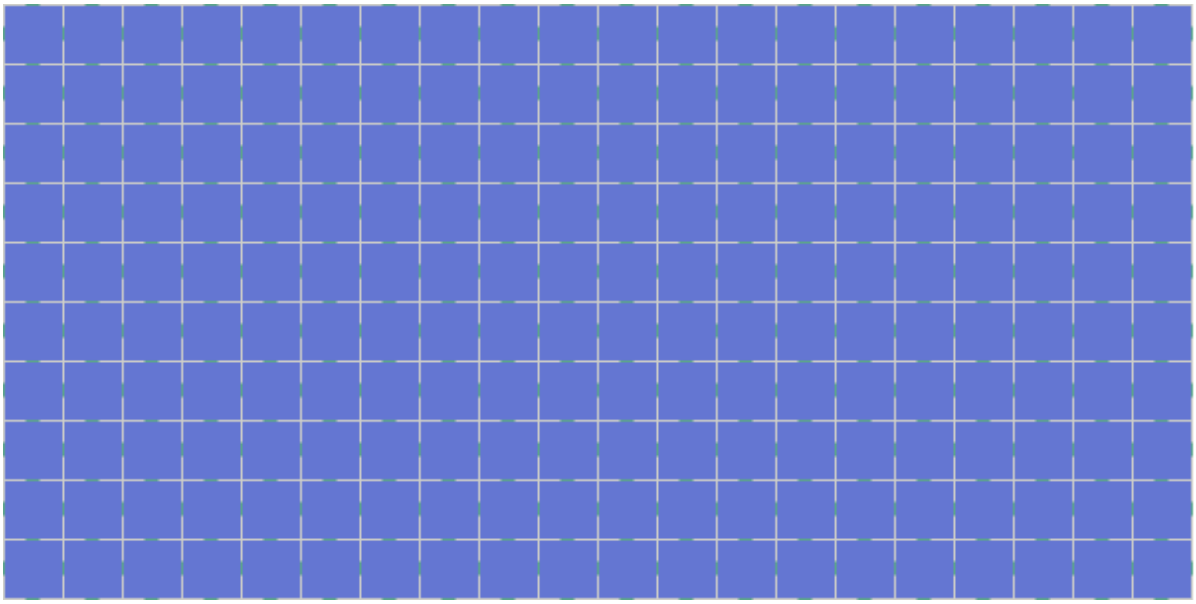
### Συσχέτιση Γράφων και Πλεγμάτων

Από τον προηγούμενο ορισμό, συνάγεται πως τα Πλέγματα μπορούν να θεωρηθούν ως ειδική περίπτωση γραφημάτων, τα οποία όμως περιέχουν πρόσθετη δομή που δεν υπάρχει γενικά στους Γράφους.



**Εικόνα 11 - Γράφος πλέγματος n x m**

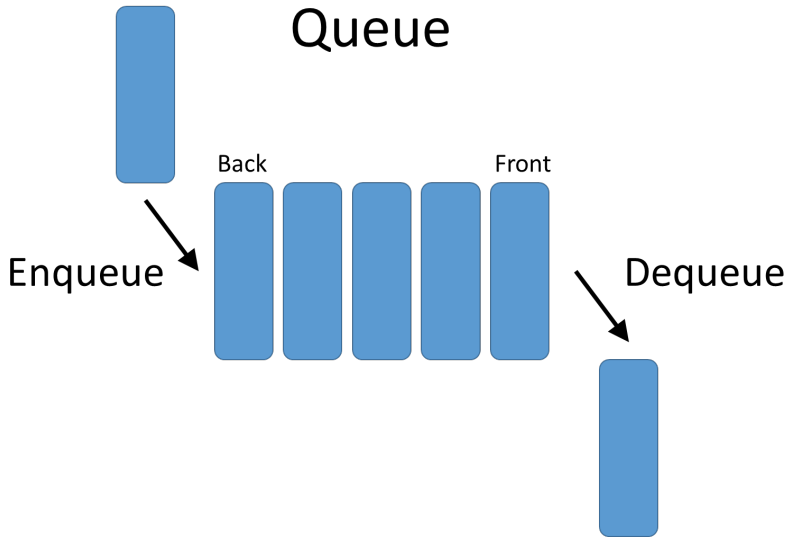
Το πλέγμα μπορεί να αναπαρασταθεί και από φατνία ή πλακίδια (tiles), διαταγμένα το ένα πλάι στο άλλο. Παρατηρώντας προσεκτικά τις 2 εικόνες (την πάνω και την κάτω) γίνεται κατανοητή η συσχέτιση αυτή. Το κέντρο των φατνίων αντιστοιχεί στη θέση των κόμβων και οι πλευρές των φατνίων στις ακμές του γράφου.



**Εικόνα 12 – Πλέγμα πλακιδίων n x m**

# Ουρά (Queue)

Στην επιστήμη της πληροφορικής, η Ουρά[6] (στα αγγλικά *Queue*) είναι ένας ειδικός τύπος δομής δεδομένων ή συλλογής στην οποία τα στοιχεία της συλλογής φυλάσσονται σε σειρά. Βασικές λειτουργίες της ουράς είναι η προσθήκη στοιχείων στην τελευταία θέση, και η απομάκρυνση των στοιχείων από την μπροστινή τελική θέση. Αυτό κάνει την ουρά μια δομή δεδομένων First-In-First-Out (FIFO) σε αντίθεση με την στοίβα (stack) δομή που είναι Last-In-First-Out (LIFO).




**Εικόνα 13 - Δομή Ουράς**

Σε μια δομή δεδομένων FIFO, το πρώτο στοιχείο που προστίθεται στην ουρά θα είναι το πρώτο που πρέπει να αφαιρεθεί. Αυτό είναι ισοδύναμο με την απαίτηση ότι μόλις προστεθεί ένα νέο στοιχείο, όλα τα στοιχεία που προστέθηκαν πριν πρέπει να αφαιρεθούν πριν μπορεί να αφαιρεθεί το νέο στοιχείο. Μια προς τα εμπρός λειτουργία της ουράς είναι η επιστροφή της αξίας του μπροστινού στοιχείου χωρίς να αφαιρείται από την ουρά. Μια ουρά είναι ένα παράδειγμα μιας γραμμικής δομής δεδομένων, ή πιο αφηρημένα σειριακή συλλογή.

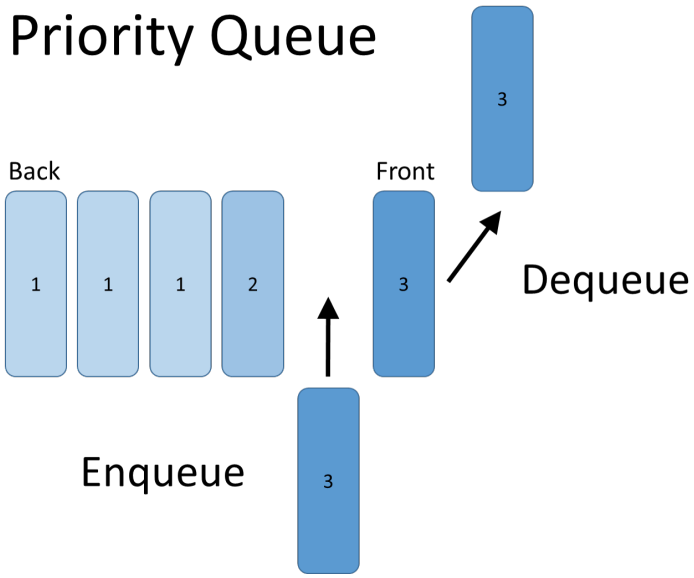
### Ουρά Προτεραιότητας (Priority Queue)

Η ουρά προτεραιότητας[7](στα αγγλικά *priority queue*) είναι ένας ειδικός τύπος της δομής δεδομένων ουρά ή στοίβα, όπου επιπλέον κάθε στοιχείο έχει μια "προτεραιότητα" που συνδέεται με αυτό. Σε μια ουρά προτεραιότητας, ένα στοιχείο με υψηλή προτεραιότητα εξυπηρετείται πριν από ένα στοιχείο με χαμηλή προτεραιότητα. Εάν τα δύο στοιχεία έχουν την ίδια προτεραιότητα, εξυπηρετούνται σύμφωνα με την σειρά τους στην ουρά.

Ενώ οι ουρές προτεραιότητας συχνά εφαρμόζονται με σωρούς, είναι εννοιολογικά διαφορετικές από σωρούς. Μια ουρά προτεραιότητας είναι μια αφηρημένη έννοια, όπως "μια λίστα" ή "ένας χάρτης", ακριβώς όπως μια λίστα μπορεί να υλοποιηθεί με

[6]https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
[7]https://en.wikipedia.org/wiki/Priority_queue

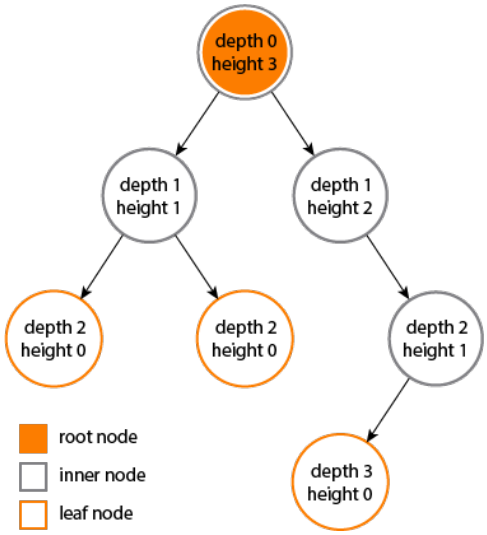
έναν πίνακα, μια ουρά προτεραιότητας μπορεί να υλοποιηθεί με ένα σωρό ή μια ποικιλία άλλων μεθόδων όπως ένα μη διατεταγμένο πίνακα.




Εικόνα 14 - Δομή Ουράς Προτεραιότητας

# Δέντρο (Tree)

Το Δέντρο είναι ένας ειδικός τύπος γραφήματος[8]. Τα δέντρα έχουν κατεύθυνση (σχέση γονέα / παιδιού) και δεν περιέχουν κύκλους. Ανήκουν στην κατηγορία των κατευθυνόμενων μη κυκλικών γράφων με τον περιορισμό ότι ένα παιδί μπορεί να έχει μόνον ένα γονέα.




Εικόνα 15 - Βάθος και Ύψος ενός Δένδρου

Το βάθος ενός κόμβου είναι ο αριθμός των ακμών από τον κόμβο στον κόμβο-ρίζα του δέντρου. Ο κόμβος-ρίζα έχει βάθος 0. Το ύψος ενός κόμβου είναι ο αριθμός των ακμών επί της μακρύτερης διαδρομής από τον κόμβο σε ένα φύλλο. Ένας κόμβος φύλλο έχει ύψος 0.

[8]http://stackoverflow.com/questions/7423401/whats-the-difference-between-the-data-structure-tree-and-graph

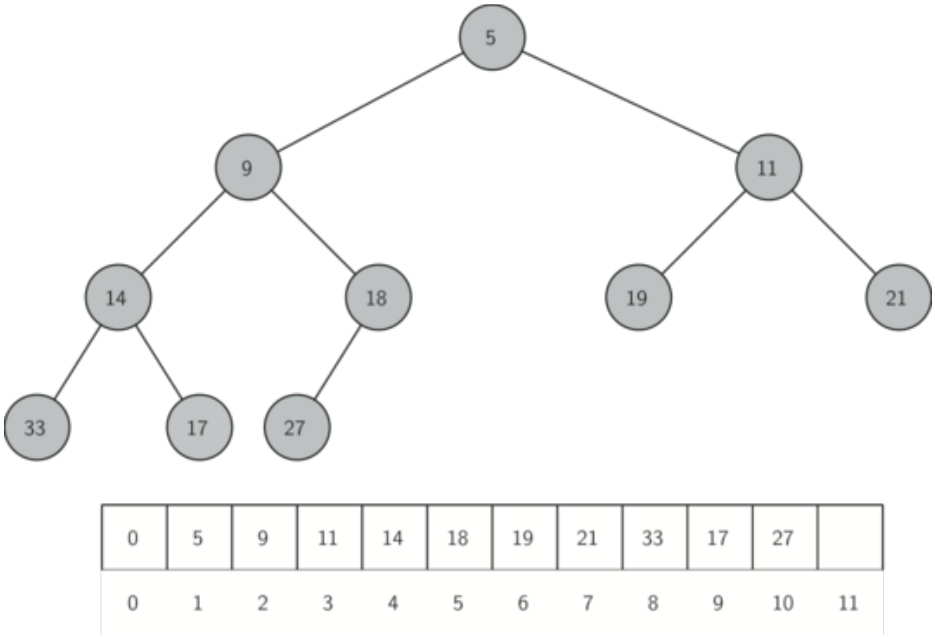
# Σωρός (Heap)

Ο σωρός (στα αγγλικά heap) είναι μια ειδική δομή δεδομένων[9] που μοιάζει με δέντρο και ικανοποιεί την ιδιότητα ότι, αν Α είναι ένας γονέας του κόμβου Β τότε το κλειδί του κόμβου Α διατάσσεται σε σχέση με το κλειδί του κόμβου Β, με την ίδια σειρά να εφαρμόζεται σε όλο το σωρό.

Ένας σωρός μπορεί να ταξινομηθεί περαιτέρω είτε ως "max heap" ή "min heap". Σε ένα max heap, τα κλειδιά των μητρικών κόμβων είναι πάντα μεγαλύτερα από ή ίσα με εκείνα των παιδιών και το υψηλότερο κλειδί είναι στον κόμβο ρίζας. Σε ένα min heap, τα κλειδιά των μητρικών κόμβων είναι μικρότερα ή ίσα με εκείνα των παιδιών και το μικρότερο κλειδί είναι στον κόμβο ρίζας. Οι σωροί είναι σημαντικοί σε πολλούς αλγόριθμους γραφημάτων, όπως ο αλγόριθμος του Dijkstra.

### Δυαδικός Σωρός

Μια κοινή υλοποίηση σωρού είναι ο δυαδικός σωρός (**binary heap**), στον οποίο το δέντρο είναι ένα πλήρες δυαδικό δένδρο όπως στο παρακάτω σχήμα[10].




Εικόνα 16 - Δομή Δυαδικού Σωρού

### Διωνυμικός Σωρός

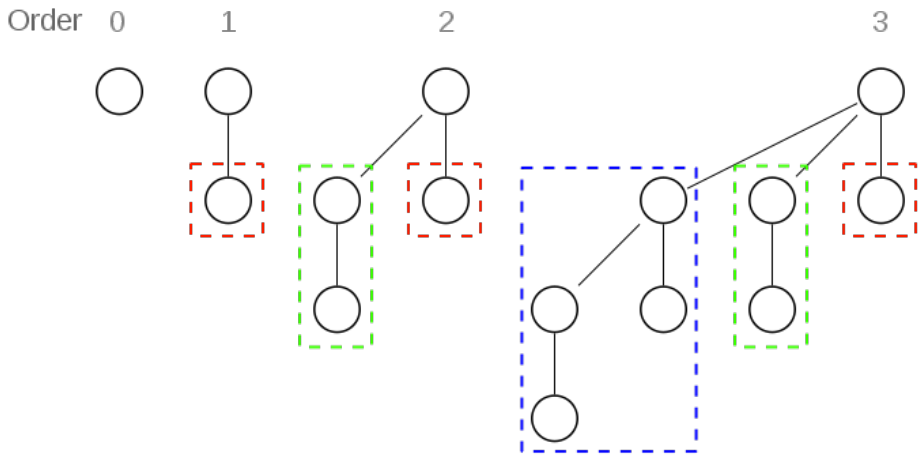
Ο διωνυμικός σωρός (**binomial heap**) είναι σωρός παρόμοιος με τον δυαδικό, ο οποίος υποστηρίζει επιπρόσθετα τη δυνατότητα συγχώνευσης δύο σωρών[11]. Αυτό επιτυγχάνεται με τη χρήση μιας ειδικής δομής δέντρου.

Ένας διωνυμικός σωρός υλοποιείται ως μια συλλογή από διωνυμικά δένδρα, η οποία ορίζεται επαναληπτικά ως εξής:

- Ένα διωνυμικό δέντρο της τάξης 0 είναι ένα μοναδιαίος κόμβος

[9]https://en.wikipedia.org/wiki/Heap_(data_structure)
[10]https://en.wikipedia.org/wiki/Binary_heap
[11]https://en.wikipedia.org/wiki/Binomial_heap

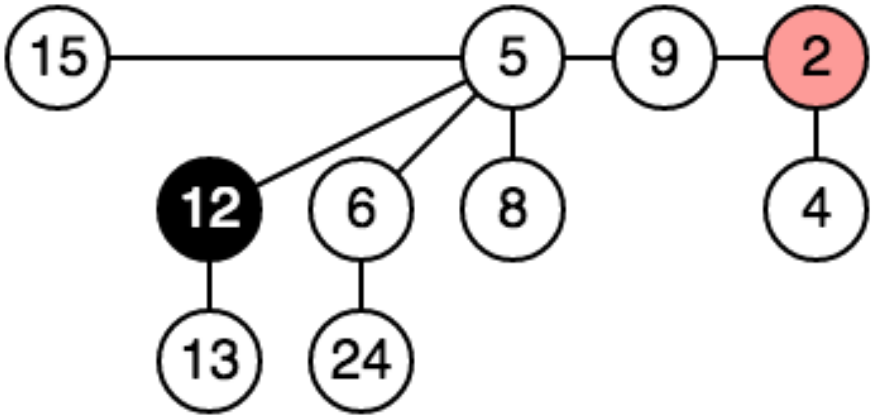
- Ένα διωνυμικό δέντρο της τάξης k έχει ένα κόμβο ρίζα των οποίων τα παιδιά είναι οι ρίζες των διωνυμικών δέντρων τάξης k-1, k-2, ..., 2, 1, 0 (με αυτή τη σειρά).




Εικόνα 17 - Δομή Διωνυμικού Σωρού

### Σωρός Fibonacci

Ονσωρός Fibonacci είναι μια δομή δεδομένων για ουρές προτεραιότητας και αποτελείται από συλλογή δέντρων διαταγμένα σε σωρούς. Έχει καλύτερο χρόνο εκτέλεσης από ότι πολλές άλλες δομές ουρών προτεραιότητας, συμπεριλαμβανομένης της δυαδικής σωρού και της διωνυμικής σωρού.




Εικόνα 18 - Σωρός Fibonacci

Ο σωρός Fibonacci αναπτύχθηκε από τους Michael L. Fredman και Robert E. Tarján το 1984 και δημοσιεύθηκε σε επιστημονικό περιοδικό το 1987[12]. Ονομάστηκε από τους αριθμούς Fibonacci, οι οποίοι χρησιμοποιούνται στην ανάλυση του χρόνου εκτέλεσης τους. Σημειώνεται πως οι αριθμοί Fibonacci είναι οι αριθμοί της ακέραιης ακολουθίας:0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 κλπ. όπου εξ ορισμού, οι πρώτοι δύο αριθμοί είναι το 0 και το 1, και κάθε επόμενος αριθμός είναι το άθροισμα των δύο προηγούμενων[13].

[12]https://en.wikipedia.org/wiki/Fibonacci_heap
[13]https://en.wikipedia.org/wiki/Fibonacci_number

## Κεφάλαιο 3 – Μελέτη Αλγορίθμων Εύρεσης Βέλτιστων Διαδρομών

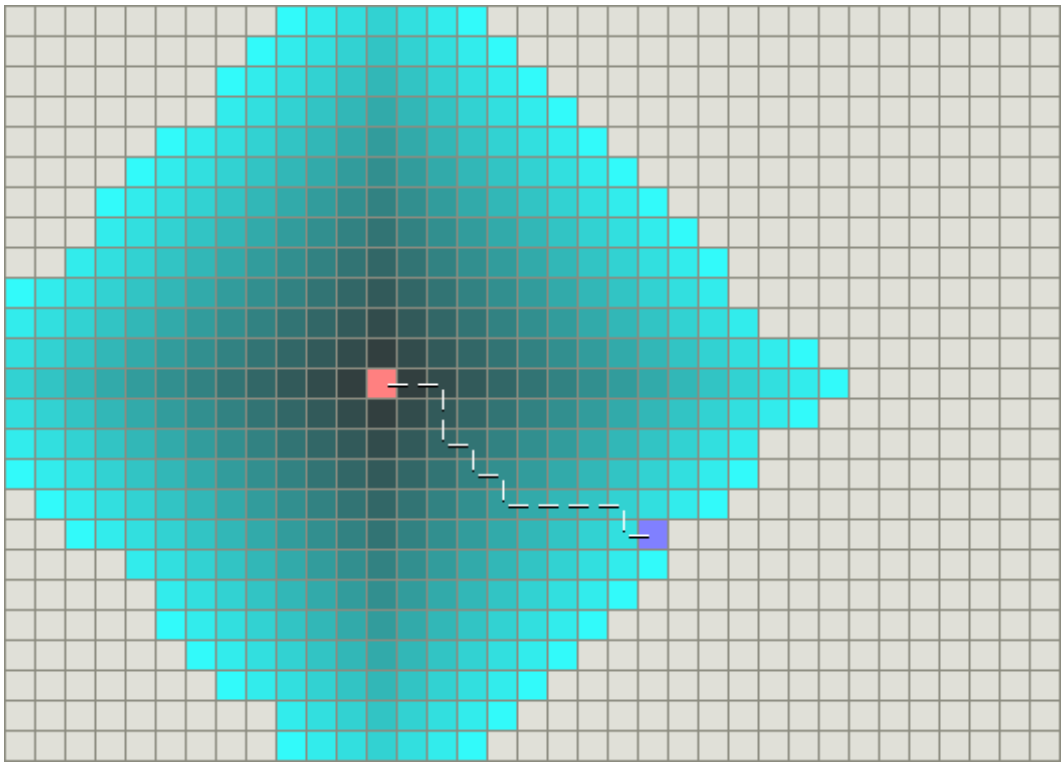
# Αλγόριθμος Dijkstra

### Περιγραφή Αλγορίθμου

Ο αλγόριθμος του Dijkstra (Ντάικστρα) είναι ένας από τα πιο γνωστούς αλγόριθμους στην επιστήμη των υπολογιστών. Το 1956, ο Edsger Dijkstra εισηγήθηκε τη μέθοδο εύρεσης συντομότερης διαδρομής μέσα σε ένα γράφημα του οποίου τα βάρη των οι ακμών είχαν μη αρνητικές τιμές. Σήμερα, σχεδόν 50 χρόνια αργότερα, ο αλγόριθμός του, εξακολουθεί να χρησιμοποιείται ευρέως.

Ο αλγόριθμος του Dijkstra λειτουργεί με το να επισκέπτεται διαδοχικά κόμβους σε ένα γράφο ξεκινώντας από τον κόμβο εκκίνησης. Εξετάζει επαναληπτικά το κόστος μετάβασης στους γειτονικούς κόμβους, προσθέτοντας τους γείτονες κόμβους στη σωρό προς εξέταση. Επεκτείνεται από τον κόμβο εκκίνησης μέχρι να φτάσει στον κόμβο προορισμού. Ο αλγόριθμος τεκμηριώνεται[14] ότι βρίσκει τη συντομότερη διαδρομή, εφόσον ισχύει ότι οι ακμές δεν έχουν αρνητικό κόστος.

Στην παρακάτω εικόνα, το κόκκινο φατνίο είναι ο κόμβος εκκίνησης, το μπλε σκούρο ο κόμβος προορισμού και τα γαλάζια είναι τα φατνία που ο αλγόριθμος εξέτασε. Τα πιο ανοιχτά σε τόνο φατνία είναι αυτά που είναι μακρύτερα από την εκκίνηση και που αποτελούν την ‘πρώτη γραμμή’ της εξερεύνησης.



Εικόνα 19–Εφαρμογή Αλγόριθμου Dijkstra

[14]http://web.cs.ucdavis.edu/~amenta/w10/dijkstra.pdf

### Ψευδοκώδικας Αλγόριθμου Dijkstra

Στον παρακάτω κώδικα[15] ορίζεται μια ουρά προτεραιότητας, η frontier, η οποία διατηρεί με προτεραιότητα τους κόμβους που εξετάζονται. Επίσης ορίζεται πίνακας came_from όπου διατηρείται η πληροφορία του προηγούμενου κόμβου από τον οποίο έγινε η μετάβαση καθώς και ένας πίνακας με πληροφορία το κόστος μετάβασης μέχρι τον εκάστοτε κόμβο.

Ο αλγόριθμος εξετάζει επαναληπτικά, για όλους τους γείτονες ενός κόμβου, το νέο κόστος μετάβασης σε αυτούς, προσθέτοντας στο κόστος μετάβασης στον υφιστάμενο κόμβο, το κόστος μετάβασης μέχρι τον γείτονα κόμβο. Αν ο γείτονας κόμβος εξετάζεται πρώτη φορά ή το κόστος μετάβασης σε αυτόν είναι μικρότερο από την προηγούμενη φορά που εξετάστηκε, ο γείτονας κόμβος προστίθεται στην ουρά με προτεραιότητα το χαμηλότερο κόστος μετάβασης σε αυτόν.

Όταν ο γείτονας κόμβος είναι ο κόμβος προορισμού ο αλγόριθμος σταματά μιας και έχει φτάσει στον στόχο.

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```

[15]http://www.redblobgames.com/pathfinding/a-star/introduction.html

Για να δημιουργηθεί η διαδρομή, δημιουργούμε λίστα αρχικά με τον τελευταίο κόμβο και προσθέτουμε στη λίστα επαναληπτικά τους κόμβους από τους οποίους μεταβήκαμε σε αυτόν με το χαμηλότερο κόστος. Τέλος με αντίστροφη επανάληψη λαμβάνεται η συντομότερη διαδρομή από την εκκίνηση προς τον προορισμό:

```
// code to reconstruct path
current = goal
path = [current]
while current != start:
   current = came_from[current]
   path.append(current)
path.reverse()
```

### Javascript κώδικας Dijkstra

Ακολούθως παρατίθεται ο JavaScript κώδικας της βιβλιοθήκης pathfinding.js που χρησιμοποιήθηκε στην εργασία. Ο κώδικας καλεί τον κώδικα για τον A* αλγόριθμο που παρατίθεται στη συνέχεια με τη μόνη διαφορά πως η ευρετική συνάρτηση είναι 0, όπως θα εξηγηθεί και στη συνέχεια.

```
/**
 * Dijkstra path-finder.
 * @constructor
 * @extends AStarFinder
 * @param {object} opt
 * @param {boolean} opt.allowDiagonal Whether diagonal movement is
allowed. Deprecated, use diagonalMovement instead.
 * @param {boolean} opt.dontCrossCorners Disallow diagonal movement
touching block corners. Deprecated, use diagonalMovement instead.
 * @param {DiagonalMovement} opt.diagonalMovement Allowed diagonal
movement.
 */
function DijkstraFinder(opt) {
    AStarFinder.call(this, opt);
    this.heuristic = function(dx, dy) {
        return 0;
    };
}

DijkstraFinder.prototype = new AStarFinder();
DijkstraFinder.prototype.constructor = DijkstraFinder;

module.exports = DijkstraFinder;
```

# Αλγόριθμος Breadth-First

### Περιγραφή Αλγορίθμου

Ο αλγόριθμος Breadth-first search (BFS) διασχίζει ή "ψάχνει" ένα γράφο ή δένδρο. Ξεκινά από τη ρίζα του δέντρου ή οποιοδήποτε κόμβο και εξετάζει τους γειτονικούς κόμβους πρώτα, προτού προχωρήσει στους γείτονες του επόμενου επιπέδου. Ο BFS ανακαλύφθηκε στα τέλη του 1950 από τον E. F. Moore και ανεξάρτητα από τον C. Y. Lee το 1961.[16]

### Ψευδοκώδικας Αλγόριθμου Breadth-First

Ο κώδικας του Breadth-First είναι αντίστοιχος του Dijkstra μόνο που η ουρά frontier δεν είναι προτεραιότητας, καθώς δεν υπολογίζεται το κόστος μετάβασης παρά μόνο η σχέση προέλευσης.

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```

### Javascript κώδικας Breadth-First

Ακολούθως παρατίθεται ο javascript κώδικας της βιβλιοθήκης pathfinding.js που χρησιμοποιήθηκε στην εργασία.

```
BreadthFirstFinder.prototype.findPath = function(startX, startY, endX,
endY, grid) {
    var openList = [],
        diagonalMovement = this.diagonalMovement,
```

[16]https://en.wikipedia.org/wiki/Breadth-first_search

```
        startNode = grid.getNodeAt(startX, startY),
        endNode = grid.getNodeAt(endX, endY),
        neighbors, neighbor, node, i, l;

    // push the start pos into the queue
    openList.push(startNode);
    startNode.opened = true;

    // while the queue is not empty
    while (openList.length) {
        // take the front node from the queue
        node = openList.shift();
        node.closed = true;

        // reached the end position
        if (node === endNode) {
            return Util.backtrace(endNode);
        }

        neighbors = grid.getNeighbors(node, diagonalMovement);
        for (i = 0, l = neighbors.length; i < l; ++i) {
            neighbor = neighbors[i];

            // skip this neighbor if it has been inspected before
            if (neighbor.closed || neighbor.opened) {
                continue;
            }

            openList.push(neighbor);
            neighbor.opened = true;
            neighbor.parent = node;
        }
    }

    // fail to find the path
    return [];
};
```

### Javascript κώδικας Bidirectional Breadth First

Στη συνέχεια παρατίθεται ο javascript κώδικας της βιβλιοθήκης pathfinding.js που χρησιμοποιήθηκε στην εργασία για bidirectional επίλυση.

```
BiBreadthFirstFinder.prototype.findPath = function(startX, startY, endX,
endY, grid) {
    var startNode = grid.getNodeAt(startX, startY),
        endNode = grid.getNodeAt(endX, endY),
        startOpenList = [], endOpenList = [],
        neighbors, neighbor, node,
        diagonalMovement = this.diagonalMovement,
        BY_START = 0, BY_END = 1,
        i, l;

    // push the start and end nodes into the queues
```

```
    startOpenList.push(startNode);
    startNode.opened = true;
    startNode.by = BY_START;

    endOpenList.push(endNode);
    endNode.opened = true;
    endNode.by = BY_END;

    // while both the queues are not empty
    while (startOpenList.length && endOpenList.length) {

        // expand start open list

        node = startOpenList.shift();
        node.closed = true;

        neighbors = grid.getNeighbors(node, diagonalMovement);
        for (i = 0, l = neighbors.length; i < l; ++i) {
            neighbor = neighbors[i];

            if (neighbor.closed) {
                continue;
            }
            if (neighbor.opened) {
                // if this node has been inspected by the reversed search,
                // then a path is found.
                if (neighbor.by === BY_END) {
                    return Util.biBacktrace(node, neighbor);
                }
                continue;
            }
            startOpenList.push(neighbor);
            neighbor.parent = node;
            neighbor.opened = true;
            neighbor.by = BY_START;
        }

        // expand end open list

        node = endOpenList.shift();
        node.closed = true;

        neighbors = grid.getNeighbors(node, diagonalMovement);
        for (i = 0, l = neighbors.length; i < l; ++i) {
            neighbor = neighbors[i];

            if (neighbor.closed) {
                continue;
            }
            if (neighbor.opened) {
                if (neighbor.by === BY_START) {
                    return Util.biBacktrace(neighbor, node);
                }
                continue;
            }
            endOpenList.push(neighbor);
            neighbor.parent = node;
            neighbor.opened = true;
            neighbor.by = BY_END;
        }
    }
```

```
    // fail to find the path
    return [];
};
```

# Αλγόριθμος Best-First Search

## Περιγραφή Αλγορίθμου

Ο Best-First Search είναι ένας αλγόριθμος αναζήτησης ο οποίος διασχίζει έναν γράφο εξετάζοντας πρώτα τον πιο «υποσχόμενο» κόμβο με βάση κάποιο κανόνα.

Ο Judea Pearl περιέγραψε τον Best-First Search εκτιμώντας την 'υποσχεσιμότητα' ενός κόμβου n με μια ευρετική συνάρτηση h(n), η οποία μπορεί να εξαρτάται από την περιγραφή του n, την περιγραφή του στόχου, την πληροφορία που συλλέγεται κατά την αναζήτηση μέχρι το σημείο εκείνο, και πιο σημαντικό, για κάθε επιπρόσθετη γνώση σε σχέση με το πεδίο.

Ο Best-First Search αναφέρεται σε αναζήτηση με μια ευρετική που επιχειρεί να προβλέψει πόσο κοντά είναι το τέλος μιας διαδρομής ως λύση, έτσι ώστε τα μονοπάτια που κρίνονται για να είναι πιο κοντά στη λύση να εξετάζονται πρώτα. Το συγκεκριμένο είδος αναζήτησης ονομάζεται και Greedy Best-First.

Για αποδοτική επιλογή του τρέχοντος καλύτερου υποψήφιου για επέκταση, συνήθως ο αλγόριθμος υλοποιείται χρησιμοποιώντας μια ουρά προτεραιότητας.

## Ψευδοκώδικας Best-First Search

Ο αλγόριθμος Best-first μπορεί να αποτυπωθεί ως εξής[17]. Η λογική είναι παρόμοια με αυτήν του Dijkstra με μόνη διαφορά πως η προτεραιότητα υπολογίζεται από το κόστος μετάβασης του υπό εξέταση κόμβου προς τον κόμβο προορισμού μέσω της ευρετικής συνάρτησης.

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
```

[17]https://en.wikipedia.org/wiki/Best-first_search

```
   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         priority = heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

### Javascript κώδικας Best-First-Search

Ακολούθως παρατίθεται ο javascript κώδικας της βιβλιοθήκης pathfinding.js που χρησιμοποιήθηκε στην εργασία.

```
/**
 * Best-First-Search path-finder.
 * @constructor
 * @extends AStarFinder
 * @param {object} opt
 * @param {boolean} opt.allowDiagonal Whether diagonal movement is
allowed. Deprecated, use diagonalMovement instead.
 * @param {boolean} opt.dontCrossCorners Disallow diagonal movement
touching block corners. Deprecated, use diagonalMovement instead.
 * @param {DiagonalMovement} opt.diagonalMovement Allowed diagonal
movement.
 * @param {function} opt.heuristic Heuristic function to estimate the
distance
 *     (defaults to manhattan).
 */
function BestFirstFinder(opt) {
    AStarFinder.call(this, opt);

    var orig = this.heuristic;
    this.heuristic = function(dx, dy) {
        return orig(dx, dy) * 1000000;
    };
};

BestFirstFinder.prototype = new AStarFinder();
BestFirstFinder.prototype.constructor = BestFirstFinder;
```

```
module.exports = BestFirstFinder;
```

# Αλγόριθμος A*

## Περιγραφή Αλγορίθμου

Ο A* (ή A star) είναι ένας αλγόριθμος που χρησιμοποιείται ευρέως σε εφαρμογές εύρεσης διαδρομής και διάσχισης γράφων, για να παρασταθεί γραφικά η διαδρομή μεταξύ κόμβων. Συνδυάζει τα χαρακτηριστικά της αναζήτησης ομοιόμορφου κόστους και ευρετικής αναζήτησης για να υπολογίζει αποτελεσματικά βέλτιστες λύσεις. Λόγω της απόδοσης και της ακρίβειάς του, απολαμβάνει πολύ ευρεία χρήση.

Οι Peter Hart, Nils Nilsson και Raphael Bertram περιέγραψαν τον αλγόριθμο για πρώτη φορά το 1968. Πρόκειται για επέκταση του αλγόριθμου Dijkstra από το 1959. Ο A * επιτυγχάνει καλύτερη απόδοση (σε σχέση με το χρόνο) χρησιμοποιώντας ευρετική συνάρτηση.

Τα βασικά χαρακτηριστικά του αλγόριθμου A* είναι η δημιουργία μιας «κλειστής λίστας» που καταγράφει τους κόμβους που εξετάστηκαν, μιας «λίστας γειτόνων» για την καταγραφή κόμβων που συνορεύουν με αυτές που έχουν ήδη αξιολογηθεί, και ο υπολογισμός των αποστάσεων από τον κόμβο εκκίνησης συν την αναμενόμενη απόσταση προς τον κόμβο προορισμού.

Η λίστα γειτόνων, που συχνά αποκαλείται η «ανοικτή λίστα», είναι μια λίστα με όλες τις τοποθεσίες που γειτνιάζει άμεσα με τους κόμβους που έχουν ήδη εξεταστεί και υπολογιστεί (κλειστή λίστα). Η κλειστή λίστα καταγράφει όλους τους κόμβους που έχουν διερευνηθεί και υπολογιστεί από τον αλγόριθμο.

Αναφορικά με την ιστορία[18] του A*, το 1964 ο Nils Nilsson ανακάλυψε μια ευρετική μέθοδο για να αυξήσει την ταχύτητα του αλγορίθμου του Dijkstra. Ο αλγόριθμός αυτός ονομάστηκε A1. Το 1967 ο Bertram Raphael έκανε σημαντικές βελτιώσεις πάνω στον αλγόριθμο, αλλά δεν κατάφερε να αποδείξει ποσοτικά τις επιπτώσεις. Αποκάλεσε τον αλγόριθμο A2. Στη συνέχεια το 1968 ο Peter E.Hart εισηγήθηκε μια θεωρία που απέδειξε πως ο A2 έδινε το βέλτιστο αποτέλεσμα χρησιμοποιώντας μια σταθερή ευρετική συνάρτηση με μικρές μόνο αλλαγές. Η απόδειξη του αλγορίθμου περιείχε μια ενότητα που έδειχνε πως ο νέος A2 αλγόριθμος ήταν ο βέλτιστος αλγόριθμος δεδομένου των συνθηκών. Μετονόμασε τον αλγόριθμό, με βάση το συντακτικό Αστέρι Κλέινι (Kleene star), ή κλειστότητα Κλέινι (Kleene closure), ώστε να είναι αυτός που περιλαμβάνει όλες τις πιθανές εκδόσεις ή αλλιώς A*.

[18]http://everything.explained.today/A*_search_algorithm/

Καθώς ο A* διασχίζει τον γράφο, ακολουθεί το μονοπάτι με το χαμηλότερο γνωστό κόστος, διατηρώντας μια ταξινομημένη ουρά προτεραιότητας εναλλακτικών τμημάτων του μονοπατιού κατά μήκος της διαδρομής. Εάν, σε κάθε σημείο, το τμήμα του μονοπατιού που διασχίζεται έχει μεγαλύτερο κόστος από ένα άλλο τμήμα που έχει υπολογιστεί, τότε εγκαταλείπει το τμήμα με το μεγαλύτερο κόστος και διασχίζει το τμήμα με το μικρότερο κόστος. Αυτή η διαδικασία συνεχίζεται έως ότου ο κόμβος προορισμού προσεγγισθεί.

### Ψευδοκώδικας A*

Ο ακόλουθος ψευδοκώδικας περιγράφει τον αλγόριθμο. Ουσιαστικά είναι συνδυασμός του Dijkstra και του Best-First Search, καθώς ως προτεραιότητα υπολογίζεται και το κόστος μετάβασης στον κόμβο που εξετάζεται αλλά και το κόστος μετάβασης στον προορισμό από αυτόν με τη βοήθεια της ευρετικής συνάρτησης.

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

### Javascript κώδικας A*

Ακολούθως παρατίθεται ο javascript κώδικας της βιβλιοθήκης pathfinding.js που χρησιμοποιήθηκε στην εργασία.

```
/**
 * Find and return the the path.
```

```
 * @return {Array.<[number, number]>} The path, including both start and
 *     end positions.
 */
AStarFinder.prototype.findPath = function(startX, startY, endX, endY,
grid) {
    var openList = new Heap(function(nodeA, nodeB) {
            return nodeA.f - nodeB.f;
        }),
        startNode = grid.getNodeAt(startX, startY),
        endNode = grid.getNodeAt(endX, endY),
        heuristic = this.heuristic,
        diagonalMovement = this.diagonalMovement,
        weight = this.weight,
        abs = Math.abs, SQRT2 = Math.SQRT2,
        node, neighbors, neighbor, i, l, x, y, ng;
// set the `g` and `f` value of the start node to be 0
    startNode.g = 0;
    startNode.f = 0;

    // push the start node into the open list
    openList.push(startNode);
    startNode.opened = true;

    // while the open list is not empty
    while (!openList.empty()) {
        // pop the position of node which has the minimum `f` value.
        node = openList.pop();
        node.closed = true;

        // if reached the end position, construct the path and return it
        if (node === endNode) {
            return Util.backtrace(endNode);
        }

        // get neigbours of the current node
        neighbors = grid.getNeighbors(node, diagonalMovement);
        //console.log('neighbors==='+neighbors.length);
        for (i = 0, l = neighbors.length; i < l; ++i) {
            neighbor = neighbors[i];

            if (neighbor.closed) {
                continue;
            }
```

```
            x = neighbor.x;
            y = neighbor.y;

            // get the distance between current node and the neighbor
            // and calculate the next g score
            ng = node.g + ((x - node.x === 0 || y - node.y === 0) ? 1 :
SQRT2);

            // check if the neighbor has not been inspected yet, or
            // can be reached with smaller cost from the current node
            if (!neighbor.opened || ng < neighbor.g) {
                neighbor.g = ng;
                neighbor.h = neighbor.h || weight * heuristic(abs(x -
endX), abs(y - endY));
                neighbor.f = neighbor.g + neighbor.h;
                neighbor.parent = node;

                if (!neighbor.opened) {
                    openList.push(neighbor);
                    neighbor.opened = true;
                } else {
                    // the neighbor can be reached with smaller cost.
                    // Since its f value has been updated, we have to
                    // update its position in the open list
                    openList.updateItem(neighbor);
                }
            }
        } // end for each neighbor
    } // end while not open list empty

    // fail to find the path
    return [];
};
```

### Javascript κώδικας Bidirectional A*

Ακολούθως παρατίθεται ο javascript κώδικας της βιβλιοθήκης pathfinding.js για Bidirectional A*:

```
BiAStarFinder.prototype.findPath = function(startX, startY, endX, endY,
grid) {
    var cmp = function(nodeA, nodeB) {
```

```
            return nodeA.f - nodeB.f;
        },
        startOpenList = new Heap(cmp),
        endOpenList = new Heap(cmp),
        startNode = grid.getNodeAt(startX, startY),
        endNode = grid.getNodeAt(endX, endY),
        heuristic = this.heuristic,
        diagonalMovement = this.diagonalMovement,
        weight = this.weight,
        abs = Math.abs, SQRT2 = Math.SQRT2,
        node, neighbors, neighbor, i, l, x, y, ng,
        BY_START = 1, BY_END = 2;

    // set the `g` and `f` value of the start node to be 0
    // and push it into the start open list
    startNode.g = 0;
    startNode.f = 0;
    startOpenList.push(startNode);
    startNode.opened = BY_START;

    // set the `g` and `f` value of the end node to be 0
    // and push it into the open list
    endNode.g = 0;
    endNode.f = 0;
    endOpenList.push(endNode);
    endNode.opened = BY_END;

    // while both the open lists are not empty
    while (!startOpenList.empty() && !endOpenList.empty()) {

        // pop the position of start node which has the minimum `f` value.
        node = startOpenList.pop();
        node.closed = true;

        // get neigbours of the current node
        neighbors = grid.getNeighbors(node, diagonalMovement);
        for (i = 0, l = neighbors.length; i < l; ++i) {
            neighbor = neighbors[i];

            if (neighbor.closed) {
                continue;
            }
            if (neighbor.opened === BY_END) {
```

```
                return Util.biBacktrace(node, neighbor);
            }

            x = neighbor.x;
            y = neighbor.y;

            // get the distance between current node and the neighbor
            // and calculate the next g score
            ng = node.g + ((x - node.x === 0 || y - node.y === 0) ? 1 :
SQRT2);

            // check if the neighbor has not been inspected yet, or
            // can be reached with smaller cost from the current node
            if (!neighbor.opened || ng < neighbor.g) {
                neighbor.g = ng;
                neighbor.h = neighbor.h || weight * heuristic(abs(x -
endX), abs(y - endY));
                neighbor.f = neighbor.g + neighbor.h;
                neighbor.parent = node;

                if (!neighbor.opened) {
                    startOpenList.push(neighbor);
                    neighbor.opened = BY_START;
                } else {
                    // the neighbor can be reached with smaller cost.
                    // Since its f value has been updated, we have to
                    // update its position in the open list
                    startOpenList.updateItem(neighbor);
                }
            }
        } // end for each neighbor


        // pop the position of end node which has the minimum `f` value.
        node = endOpenList.pop();
        node.closed = true;

        // get neigbours of the current node
        neighbors = grid.getNeighbors(node, diagonalMovement);
        for (i = 0, l = neighbors.length; i < l; ++i) {
            neighbor = neighbors[i];

            if (neighbor.closed) {
```

```
                continue;
            }
            if (neighbor.opened === BY_START) {
                return Util.biBacktrace(neighbor, node);
            }

            x = neighbor.x;
            y = neighbor.y;

            // get the distance between current node and the neighbor
            // and calculate the next g score
            ng = node.g + ((x - node.x === 0 || y - node.y === 0) ? 1 :
SQRT2);

            // check if the neighbor has not been inspected yet, or
            // can be reached with smaller cost from the current node
            if (!neighbor.opened || ng < neighbor.g) {
                neighbor.g = ng;
                neighbor.h = neighbor.h || weight * heuristic(abs(x -
startX), abs(y - startY));
                neighbor.f = neighbor.g + neighbor.h;
                neighbor.parent = node;

                if (!neighbor.opened) {
                    endOpenList.push(neighbor);
                    neighbor.opened = BY_END;
                } else {
                    // the neighbor can be reached with smaller cost.
                    // Since its f value has been updated, we have to
                    // update its position in the open list
                    endOpenList.updateItem(neighbor);
                }
            }
        } // end for each neighbor
    } // end while not open list empty

    // fail to find the path
    return [];
};
```

# Αλγόριθμος IDAStarFinder

### Περιγραφή Αλγορίθμου

Ο Iterative Deepening A* (IDA*) είναι ένας αλγόριθμος αναζήτησης διαδρομής που μπορεί να βρει τη συντομότερη διαδρομή ανάμεσα σε ένα κόμβο εκκίνησης και κάθε μέλος ενός συνόλου από κόμβους προορισμού σε ένα γράφο με βάρη. Χρησιμοποιεί ευρετική συνάρτηση για να εξετάσει το υπολειπόμενο κόστος για να φτάσει στον προορισμό.

Ο αλγόριθμος εξετάζει τους πιο υποσχόμενους κόμβους και δεν πάει στο ίδιο βάθος παντού στο δένδρο αναζήτησης. Σε αντίθεση με τον A*, ο IDA δεν χρησιμοποιεί δυναμικό προγραμματισμό και για αυτό καταλήγει πολλές φορές να εξετάσει τους ίδιους κόμβους πολλές φορές. Ο αλγόριθμος περιγράφτηκε για πρώτη φορά το 1985 από τον Richard Korf[19].

### Ψευδοκώδικας IDA*

```
node                current node
 g                   the cost to reach current node
 f                   estimated cost of the cheapest path
(root..node..goal)
 h(node)             estimated cost of the cheapest path (node..goal)
 cost(node, succ)    step cost function
 is_goal(node)       goal test
 successors(node)    node expanding function

 procedure ida_star(root)
   bound := h(root)
   loop
     t := search(root, 0, bound)
     if t = FOUND then return FOUND
     if t = ∞ then return NOT_FOUND
     bound := t
   end loop
 end procedure

 function search(node, g, bound)
   f := g + h(node)
   if f > bound then return f
   if is_goal(node) then return FOUND
   min := ∞
   for succ in successors(node) do
     t := search(succ, g + cost(node, succ), bound)
```

[19]https://cse.sc.edu/~mgv/csce580f09/gradPres/korf_IDAStar_1985.pdf

```
      if t = FOUND then return FOUND
      if t < min then min := t
```

```
   end for
   return min
 end function
```

### Javascript κώδικας IDA*

Ακολούθως παρατίθεται ο javascript κώδικας της βιβλιοθήκης pathfinding.js που χρησιμοποιήθηκε στην εργασία.

```
IDAStarFinder.prototype.findPath = function(startX, startY, endX,
endY, grid) {
    // Used for statistics:
    var nodesVisited = 0;

    // Execution time limitation:
    var startTime = new Date().getTime();

    // Heuristic helper:
    var h = function(a, b) {
        return this.heuristic(Math.abs(b.x - a.x), Math.abs(b.y -
a.y));
    }.bind(this);

    // Step cost from a to b:
    var cost = function(a, b) {
        return (a.x === b.x || a.y === b.y) ? 1 : Math.SQRT2;
    };

    /**
     * IDA* search implementation.
     *
     * @param {Node} The node currently expanding from.
     * @param {number} Cost to reach the given node.
     * @param {number} Maximum search depth (cut-off value).
     * @param {{Array.<[number, number]>}} The found route.
     * @param {number} Recursion depth.
     *
     * @return {Object} either a number with the new optimal cut-off
depth,
     * or a valid node instance, in which case a path was found.
```

```
     */
    var search = function(node, g, cutoff, route, depth) {
        nodesVisited++;

        // Enforce timelimit:
        if(this.timeLimit > 0 && new Date().getTime() - startTime >
this.timeLimit * 1000) {
            // Enforced as "path-not-found".
            return Infinity;
        }

        var f = g + h(node, end) * this.weight;

        // We've searched too deep for this iteration.
        if(f > cutoff) {
            return f;
        }

        if(node == end) {
            route[depth] = [node.x, node.y];
            return node;
        }

        var min, t, k, neighbour;

        var neighbours = grid.getNeighbors(node,
this.diagonalMovement);

        // Sort the neighbours, gives nicer paths. But, this deviates
        // from the original algorithm - so I left it out.
        //neighbours.sort(function(a, b){
        //    return h(a, end) - h(b, end);
        //});

        for(k = 0, min = Infinity; neighbour = neighbours[k]; ++k) {

            if(this.trackRecursion) {
                // Retain a copy for visualisation. Due to recursion,
this
                // node may be part of other paths too.
                neighbour.retainCount = neighbour.retainCount + 1 ||
1;
```

```
                if(neighbour.tested !== true) {
                    neighbour.tested = true;
                }
            }

            t = search(neighbour, g + cost(node, neighbour), cutoff,
route, depth + 1);

            if(t instanceof Node) {
                route[depth] = [node.x, node.y];

                // For a typical A* linked list, this would work:
                // neighbour.parent = node;
                return t;
            }

            // Decrement count, then determine whether it's actually
closed.
            if(this.trackRecursion && (--neighbour.retainCount) === 0)
{
                neighbour.tested = false;
            }

            if(t < min) {
                min = t;
            }
        }

        return min;

    }.bind(this);

    // Node instance lookups:
    var start = grid.getNodeAt(startX, startY);
    var end   = grid.getNodeAt(endX, endY);

    // Initial search depth, given the typical heuristic contraints,
    // there should be no cheaper route possible.
    var cutOff = h(start, end);

    var j, route, t;

    // With an overflow protection.
```

```
    for(j = 0; true; ++j) {
        //console.log("Iteration: " + j + ", search cut-off value: " +
cutOff + ", nodes visited thus far: " + nodesVisited + ".");

        route = [];

        // Search till cut-off depth:
        t = search(start, 0, cutOff, route, 0);

        // Route not possible, or not found in time limit.
        if(t === Infinity) {
            return [];
        }

        // If t is a node, it's also the end node. Route is now
        // populated with a valid path to the end node.
        if(t instanceof Node) {
            //console.log("Finished at iteration: " + j + ", search
cut-off value: " + cutOff + ", nodes visited: " + nodesVisited + ".");
            return route;
        }

        // Try again, this time with a deeper cut-off. The t score
        // is the closest we got to the end node.
        cutOff = t;
    }

    // This _should_ never to be reached.
    return [];
};
```

# Ο Ρόλος της Ευρετικής Συνάρτησης

Η ευρετική συνάρτηση h(n) στους αλγορίθμους αναζήτησης σε γράφους με βάρη δίνει την εκτίμηση του κόστους για τη μετάβαση από τον κόμβο n που εξετάζεται κάθε φορά προς τον κόμβο προορισμού. Προστίθεται στο γνωστό κόστος μετάβασης από τον κόμβο εκκίνησης προς τον κόμβο n, που συμβολίζεται g(n). Το συνολικό κόστος δίνεται από τη συνάρτηση f(n) και ισχύσει για κάθε κόμβο n:

$$f(n) = g(n) + h(n)$$

Είναι σημαντικό να επιλεγεί η καλύτερη δυνατή συνάρτηση, ώστε ο αλγόριθμος να δώσει την βέλτιστη λύση.

Η ευρετική συνάρτηση μπορεί να χρησιμοποιηθεί για να ελέγξει τη συμπεριφορά του αλγόριθμου A*.

- Στην περίπτωση που το h(n) είναι 0, τότε μόνο η g(n) συμμετέχει στους υπολογισμούς , και ο A* ταυτίζεται με τον αλγόριθμο Dijkstra, ο οποίος εγγυάται πως βρίσκει τη συντομότερη διαδρομή.
- Εάν το h(n) είναι πάντοτε χαμηλότερο ή ίσο με το κόστος μετάβασης στον στόχο, τότε ο A* εγγυάται πως βρίσκει τη συντομότερη διαδρομή.
- Όσο πιο χαμηλότερο είναι το h(n), τόσο περισσότερους κόμβους πρέπει να εξετάσει ο A*, το οποίο τον καθιστά πιο αργό.
- Εάν το κόστος h(n) είναι ακριβώς ίσο με το πραγματικό κόστος της μετάβασης από τον τρέχον κόμβο στον προορισμό, τότε ο A* ακολουθεί την συντομότερη διαδρομή και τον καθιστά πολύ γρήγορο. Αν και αυτό δεν μπορεί να συμβεί σε όλες τις περιπτώσεις, μπορεί να συμβεί και είναι σημαντικό πως με σωστή ευρετική συνάρτηση ο A* συμπεριφέρεται σωστά.
- Εάν το κόστος h(n) είναι μεγαλύτερο από το πραγματικό, τότε ο A* δεν εγγυάται πως θα βρει την συντομότερη διαδρομή, αλλά μπορεί να φέρει αποτέλεσμα πολύ γρηγορότερα.
- Στην περίπτωση που το h(n) είναι πολύ υψηλό σε σχέση με το g(n), τότε μόνο το h(n) παίζει ρόλο, και ο A* συμπεριφέρεται όπως ο Greedy Best-First-Search.

### Ταχύτητα vs Ακρίβεια

Η δυνατότητα του A* να μεταβάλει τη συμπεριφορά του με βάσει την επιλογή της ευρετικής συνάρτησης δίνει περαιτέρω επιλογές εφαρμογής. Πολλές φορές δεν χρειάζεται να προσδιοριστεί η πλέον σύντομη διαδρομή αλλά κάποια που να είναι πολύ κοντά σε αυτήν.

### Κλίμακα & Μονάδες

Ο A* υπολογίζει το συνολικό κόστος f(n) = g(n) + h(n) για κάθε κόμβο. Για να προστεθούν οι δύο τιμές θα πρέπει να έχουν την ίδια κλίμακα και μονάδα μέτρησης. Εάν η g(n) υπολογίζεται σε ώρες και η h(n) υπολογίζεται σε μέτρα, τότε ο A* πρόκειται να θεωρήσει το g ή το h πολύ λίγο ή πάρα πολύ, και συνεπώς δεν θα υπολογιστούν βέλτιστες δυνατές διαδρομές ή θα τρέξει πιο αργά ο αλγόριθμος από ότι είναι δυνατό να τρέξει.

### Τέλεια Ευρετική

Αν η ευρετική υπολογίζει ακριβώς την απόσταση που ακολουθεί η βέλτιστη (συντομότερη) διαδρομή, παρατηρούμε πως ο A* δεν εξετάζει περαιτέρω κόμβους πέραν των ακριβώς γειτονικών. Αυτό γιατί ο A* υπολογίζει το f(n) = g(n) + h(n) σε κάθε κόμβο. Όταν η h(n) είναι ακριβώς ίδια με την g(n) τότε η τιμή του f(n) δεν αλλάζει κατά τη εκτέλεση του αλγορίθμου. Όλοι οι κόμβοι που δεν είναι στη βέλτιστη διαδρομή θα έχουν μεγαλύτερη τιμή f από τους κόμβους που κείτονται πάνω στην βέλτιστη διαδρομή. Καθώς ο A* δεν εξετάζει κόμβους με υψηλότερο f, εφόσον έχει

υπολογίσει κόμβους με χαμηλότερο f, για αυτό δεν ξεφεύγει από την συντομότερη διαδρομή.

### Γραμμική Ευρετική

Στην περίπτωση πλέγματος χωρίς εμπόδια και χωρίς μεταβολή στο κόστος από κόμβο σε κόμβο, τότε η συντομότερη διαδρομή από τον κόμβο εκκίνησης στον κόμβο προορισμού θα πρέπει να είναι μια ευθεία γραμμή. Αν χρησιμοποιηθεί μια απλή ευρετική, κάποια που να μην γνωρίζει για τα εμπόδια πάνω στο πλέγμα, αυτή θα πρέπει να ταιριάζει με την τέλεια ευρετική.

# Ευρετικές για Πλέγματα

Σε ένα πλέγμα υπάρχουν κάποιες συνήθεις ευρετικές που μπορούν να χρησιμοποιηθούν. Είναι αυτές που υπολογίζουν απόσταση και ταιριάζουν με την κίνηση που επιτρέπεται να γίνει. Οι προτεινόμενες είναι:

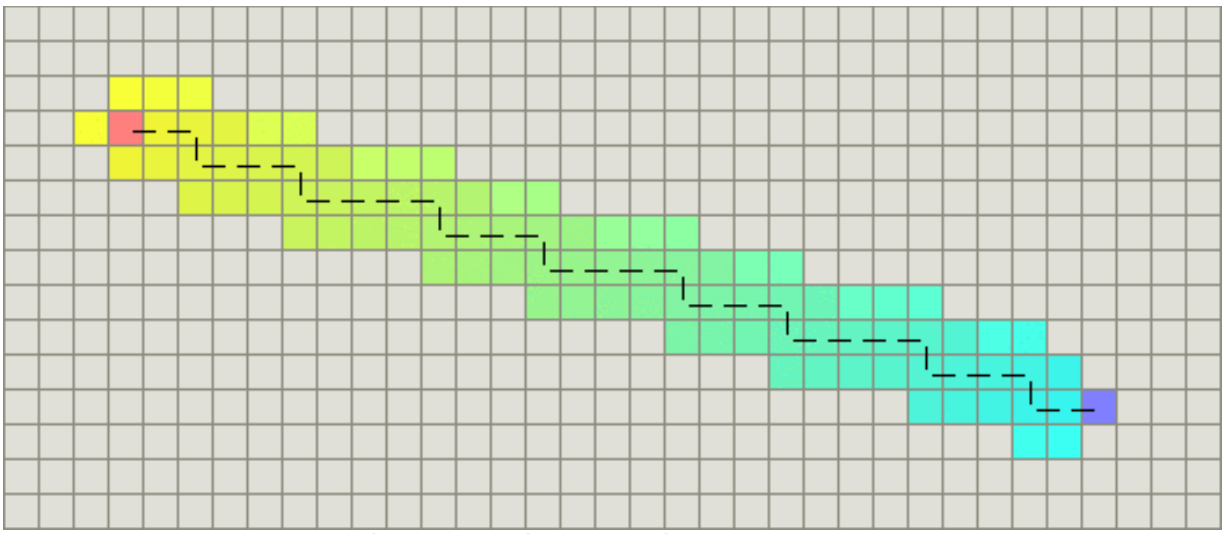
- Σε πλέγματα που επιτρέπονται 4 κατευθύνσεις για κίνηση, να χρησιμοποιηθεί η απόσταση Manhattan.
- Σε πλέγματα που επιτρέπονται 8 κατευθύνσεις για κίνηση, να χρησιμοποιηθεί κάποια διαγώνια απόσταση Chebyshev ή Octile.
- Σε πλέγματα που επιτρέπεται οποιαδήποτε κατεύθυνση, μπορεί να χρησιμοποιηθεί η ΕυκλείδιαΕυκλείδεια απόσταση.

### Απόσταση Manhattan

Η πιο γνωστή ευρετική συνάρτηση για αναζήτηση σε πλέγμα είναι η Απόσταση Manhattan. Η συνάρτηση υπολογισμού του κόστους υπολογίζει το ελάχιστο κόστος M για τη μετακίνηση από μία θέση (κόμβο) στη διπλανή της. Στην απλούστερη περίπτωση τίθεται το M να είναι ίσο με 1.

```
function heuristic(node)
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return M * (dx + dy)
```

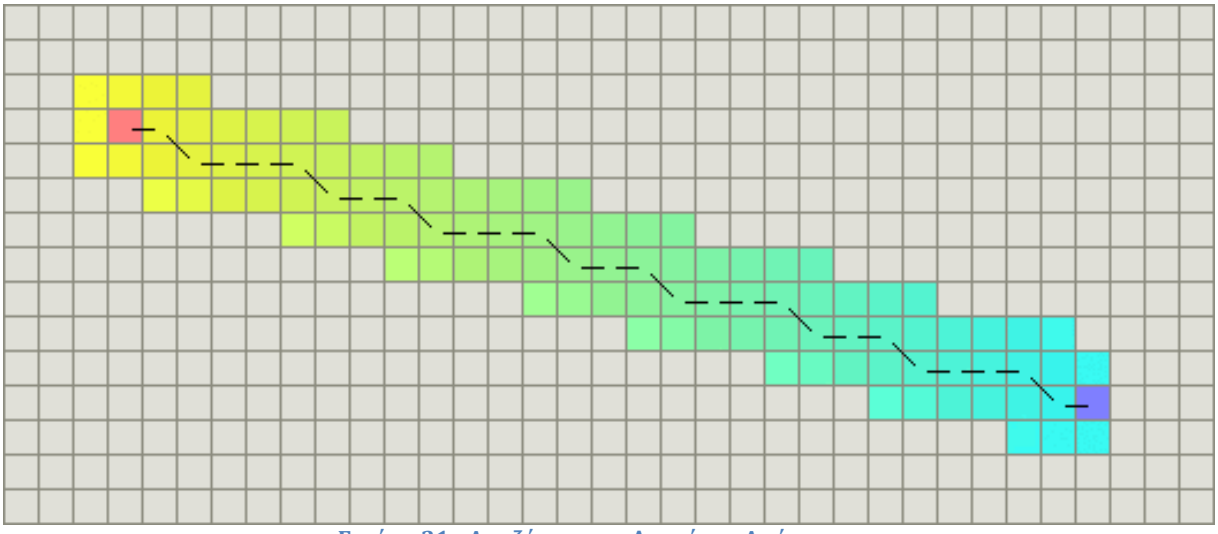
Η ευρετική συνάρτηση όπου η μετακίνηση είναι στις 4 κατευθύνσεις του πλέγματος είναι M φορές η απόσταση Manhattan.



Εικόνα 20 - Αναζήτηση με απόσταση Manhattan

### Διαγώνια απόσταση (Chebyshev &Octile)

Εάν η αναζήτηση επιτρέπει διαγώνια κίνηση χρειάζεται άλλη ευρετική. Η απόσταση Manhattan για κίνηση για παράδειγμα 4 φατνία (κόμβοι) ανατολικά και 4 βόρεια είναι 8*M. Διαφορετικά η κίνηση μπορεί να γίνει 4 φατνία βόρειοανατολικά και το κόστος να είναι 4*D, όπου D είναι το κόστος για τη διαγώνια κίνηση.



Εικόνα 21 - Αναζήτηση με Διαγώνια Απόσταση

Η Διαγώνια Απόσταση δίνεται από τον κάτωθι κώδικα:

```
function heuristic(node)
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return M * (dx + dy) + (D - 2 * M) * min(dx, dy)
```

Η συνάρτηση υπολογίζει τον αριθμό των βημάτων που γίνονται όταν δεν γίνεται διαγώνια κίνηση, και αφαιρεί την απόσταση που κερδίζεται από τα μικρότερα βήματα που γίνονται λόγω της διαγώνιας κίνησης. Υπάρχουν min(dx, dy) διαγώνια βήματα που το καθένα έχει κόστος D, αλλά εξοικονομείται 2*M μη-διαγώνια βήματα.

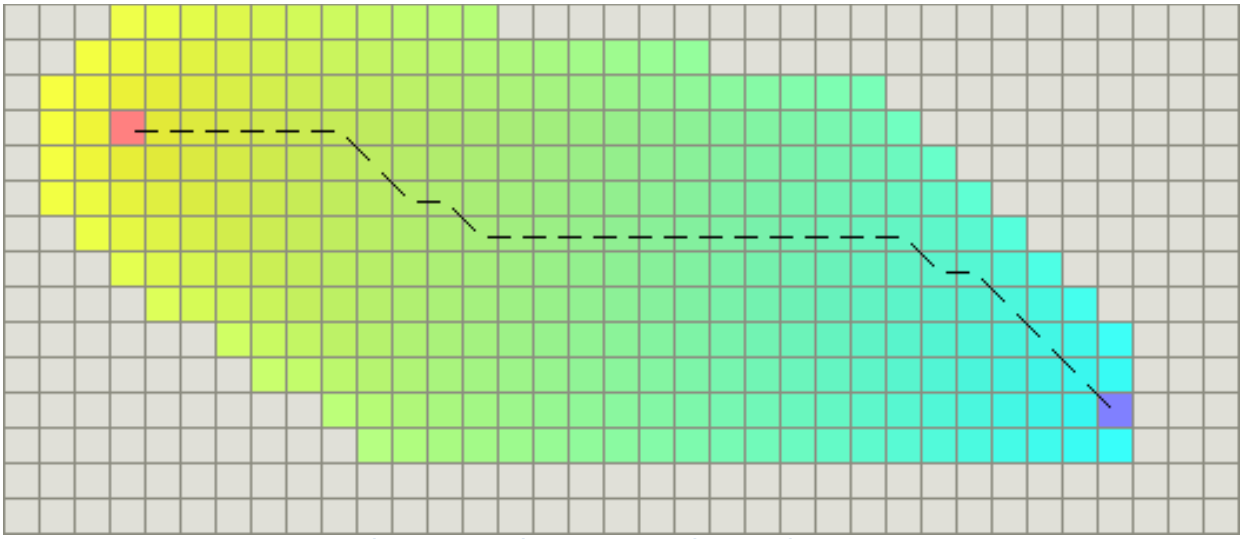
Όταν M=1 και D=1, η απόσταση ονομάζεται Chebyshev και όταν M=1 και D=sqr(2), ονομάζεται απόσταση Octile.

### Ευκλείδια Απόσταση

Σε περίπτωση που η κίνηση μπορεί να γίνει σε οποιαδήποτε γωνία (πέραν των κινήσεων του πλέγματος), τότε η προτιμητέα απόσταση είναι η ευθεία γραμμή.

```
function heuristic(node)
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx * dx + dy * dy)
```

Στην περίπτωση αυτή όμως, για τον αλγόριθμό A*, η συνάρτηση κόστους g δεν θα είναι ίδια με την συνάρτηση h. Καθώς η Ευκλείδια απόσταση είναι μικρότερη από τη Manhattan ή τη Διαγώνια απόσταση, θα υπολογιστεί συντομότερη διαδρομή, αλλά ο A* θα τρέξει περισσότερο χρόνο. Ενδεικτικά φαίνεται στην παρακάτω εικόνα:



Εικόνα 22 - Αναζήτηση με Ευκλείδεια Απόσταση

# Πολυπλοκότητα Αλγορίθμων

## Συμβολισμός Big-O

Η αλγοριθμική πολυπλοκότητα είναι ένας τρόπος να μετρηθεί πόσο γρήγορα «τρέχει» ένα πρόγραμμα ή ένας αλγόριθμος. Ο συμβολισμός BigO[20] μπορεί να χαρακτηρίσει τους αλγόριθμους και είναι ένας τρόπος περιγραφής της πολυπλοκότητάς τους. Προσδιορίζει τον ρυθμό με τον οποίο αυξάνονται οι υπολογισμοί που απαιτούνται από τον αλγόριθμο, ή αντίστοιχα, ο χώρος που απαιτείται στη μνήμη, όσο αυξάνονται τα δεδομένα που εισάγονται.

Έστω f(n) η συνάρτηση μέτρησης των εντολών ενός αλγορίθμου και n τα δεδομένα εισόδου, καλείται *ασυμπτωτική συμπεριφορά* του αλγορίθμου η εφαρμογή φίλτρου απόρριψης όλων των συντελεστών της συνάρτησης f και η διατήρηση μόνον του όρου που μεγαλώνει πιο γρήγορα. Για παράδειγμα, η ασυμπτωτική συμπεριφορά ενός αλγόριθμου με f(n) = a*n +b, όπου a και b ακέραιοι αριθμοί, περιγράφεται από τη συνάρτηση f(n) = n. Αυτό που ουσιαστικά ενδιαφέρει είναι το όριο της συνάρτησης f καθώς το n τείνει στο άπειρο.

Με αυτή τη λογική, οι αλγόριθμοι που δεν περιέχουν βρόγχους, δηλαδή for-loops, έχουν f(n) = 1, καθώς ο αριθμός των εντολών που χρειάζεται για εκτέλεση είναι σταθερός, ανεξάρτητος των δεδομένων εισόδου. Οι αλγόριθμοι που έχουν έναν μόνο βρόγχο που πάει από το 1 έως το n έχουν f(n) = n, καθώς τρέχουν ένα σταθερό πλήθος εντολών πριν τον βρόγχο, ένα σταθερό πλήθος εντολών μετά τον βρόγχο, και ένα σταθερό πλήθος εντολών μέσα στον βρόγχο κάθε μία από τις οποίες τρέχει n φορές.

Η συνάρτηση f καλείται χρονική πολυπλοκότητα ή απλώς πολυπλοκότητα του αλγορίθμου και συμβολίζεται με Θ. Ένας αλγόριθμος με Θ(n) έχει πολυπλοκότητα n. Υπάρχει ειδική ονοματολογία για τις πολυπλοκότητες Θ(1), Θ(n), Θ($n^2$) και Θ(log(n)) επειδή εμφανίζονται συχνά. Έτσι, ένας Θ(1) αλγόριθμος είναι ένας αλγόριθμος σταθερού χρόνου, ένας Θ(n) είναι γραμμικός, ένας Θ($n^2$) είναι τετραγωνικός και ένας Θ(log(n)) είναι λογαριθμικός.

Τώρα, ενώ το Θ αποδίδει την πραγματική πολυπλοκότητα του αλγορίθμου, και λέγεται ότι το Θ δίνει ένα ακριβές φράγμα, ο συμβολισμός O-πολυπλοκότητα ενός αλγορίθμου δίνει ένα άνω φράγμα για την πραγματική πολυπλοκότητα του αλγορίθμου. Έτσι, αν ένας αλγόριθμος είναι Θ(n), και η ακριβής του πολυπλοκότητα είναι n, τότε ο αλγόριθμος αυτός είναι τόσο O(n) όσο και O($n^2$). Το O(n) περιγράφει έναν αλγόριθμο του οποίου η απόδοση αυξάνεται γραμμικά και κατ' αναλογία του μεγέθους των δεδομένων που εισάγουμε.

Αν ένα φράγμα πολυπλοκότητας είναι γνωστό πως δεν είναι ακριβές, χρησιμοποιείται το πεζό γράμμα ο για να δηλωθεί. Επειδή ο αλγόριθμος είναι Θ(n), το φράγμα O(n) είναι ένα ακριβές φράγμα. Όμως το φράγμα O($n^2$) δεν είναι ακριβές, κι έτσι γράφεται ότι ο αλγόριθμος είναι ο($n^2$), το οποίο διαβάζεται «μικρό ο του εν τετράγωνο» και σημαίνει ότι το φράγμα δεν είναι ακριβές. Είναι καλύτερα όταν υπολογίζονται τα

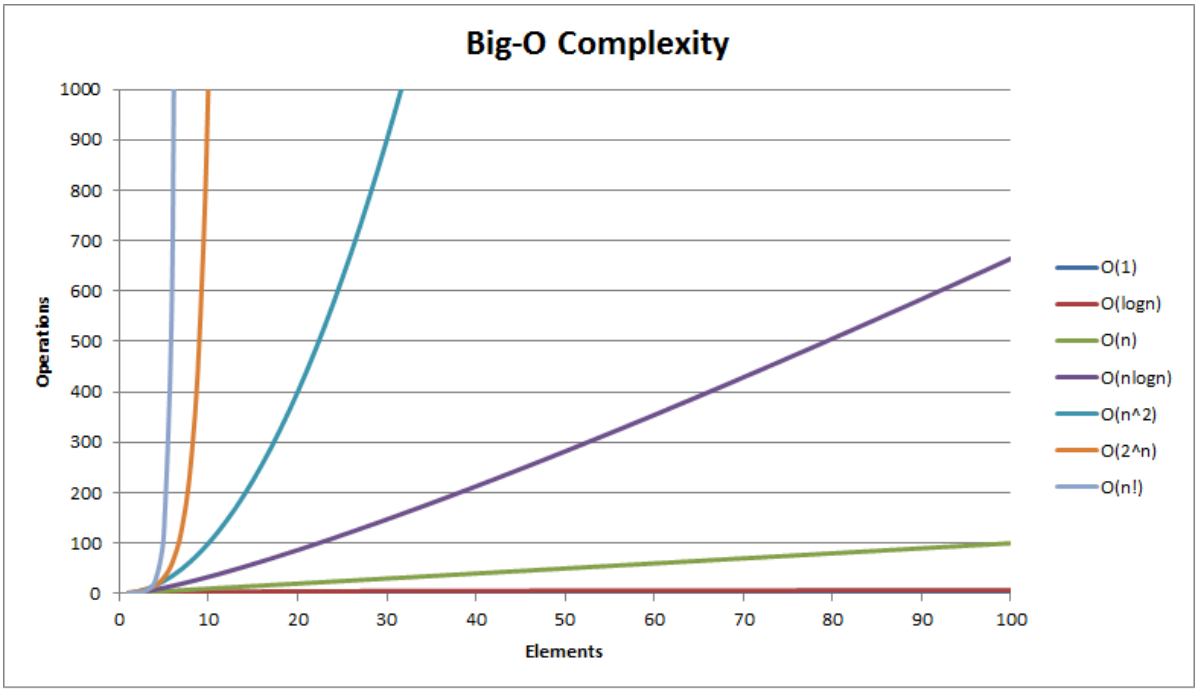
[20]http://discrete.gr/complexity/?el

ακριβή φράγματα για τους αλγορίθμους, αφού δίνουν περισσότερες πληροφορίες για τη συμπεριφορά τους, όμως κάτι τέτοιο δεν είναι πάντα εύκολο.

Ένα κλασσικό παράδειγμα λογαριθμικού αλγορίθμου είναι ο Binary Search (Δυαδική Αναζήτηση) που είναι μια μέθοδος αναζήτησης ταξινομημένων (sorted) δεδομένων. Λειτουργεί επιλέγοντας το μεσαίο στοιχείο από το σύνολο των δεδομένων που έχουν δοθεί και το συγκρίνει με την τιμή που αναζητείται. Αν είναι αυτό που αναζητείται, τελειώνει η αναζήτηση. Αν η τιμή που αναζητείται είναι μεγαλύτερη από την μεσαία τιμή, ο αλγόριθμος θα πάρει το άνω μισό των δεδομένων και θα ξεκινήσει από την αρχή. Αν η τιμή είναι μικρότερη από τη μεσαία τιμή, τότε θα πάρει το κάτω μισό των δεδομένων. Θα συνεχίσει να κόβει στη μέση τα δεδομένα και να επιλέγει το μεσαίο στοιχείο μέχρι να βρει την τιμή που αναζητείται ή μέχρι να μην μπορεί να «κόψει» άλλο τα δεδομένα.

Το γεγονός ότι ο αλγόριθμος κόβει στη μέση τα δεδομένα σε κάθε επανάληψη, έχει ως αποτέλεσμα να παραχθεί μια καμπύλη ανάπτυξης, η οποία κορυφώνεται νωρίς και στη συνέχεια γίνεται όλο και πιο ομαλή.

Αυτό σημαίνει ότι αν, για παράδειγμα, απαιτείται 1 δευτερόλεπτο για να αναζητηθεί κάτι μέσα σε ένα array 10 στοιχείων, θα χρειαστούν μόλις 3 δευτερόλεπτα για μια αναζήτηση σε ένα array 100 στοιχείων. Όσο αυξάνονται τα δεδομένα που εισάγονται, τόσο μικραίνει ο ρυθμός αύξησης του χρόνου εκτέλεσης. Στον επόμενο πίνακα φαίνονται οι ρυθμοί αύξησης για διαφορετικές πολυπλοκότητες[21]:




Εικόνα 23 - Big-O Complexity Chart

[21]http://bigocheatsheet.com/

### Πολυπλοκότητα Σωρών (Heaps)

Επίσης στον ακόλουθο πίνακα αποτυπώνονται οι χρονικές πολυπλοκότητες των βασικών λειτουργιών διαφόρων τύπων σωρών[22]:

**Heap Operations**

| Type | Time Complexity | | | | | | |
|---|---|---|---|---|---|---|---|
| | Heapify | Find Max | Extract Max | Increase Key | Insert | Delete | Merge |
| Linked List (sorted) | - | O(1) | O(1) | O(n) | O(n) | O(1) | O(m+n) |
| Linked List (unsorted) | - | O(n) | O(n) | O(1) | O(1) | O(1) | O(1) |
| Binary Heap | O(n) | O(1) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(m+n) |
| Binomial Heap | - | O(1) | O(log(n)) | O(log(n)) | O(1) | O(log(n)) | O(log(n)) |
| Fibonacci Heap | - | O(1) | O(log(n)) | O(1) | O(1) | O(log(n)) | O(1) |

Εικόνα 24 - Heap Operations Time Complexity

### Πολυπλοκότητα Αλγόριθμου Dijkstra

Η χρονική πολυπλοκότητα του αλγόριθμου Dijkstra ποικίλλει ανάλογα με τον γράφο και τις δομές δεδομένων που χρησιμοποιούνται για την υλοποίησή του. Γενικά η πολυπλοκότητα του αλγόριθμου είναι[23]:

O(|E| |decrease-key(Q)| + |V| |extract-min(Q)|)

όπου Q είναι min-priority ουρά προτεραιότητας με ταξινομημένους κόμβους με βάση την εκτίμηση της απόστασης τους.

Τόσο για ένα σωρό Fibonacci και ένα δυαδικό σωρό, η πολυπλοκότητα της λειτουργίας extract-min σε αυτήν την ουρά είναι O(log|V|). Αυτό εξηγεί το 2ο μέρος της πολυπλοκότητας |V|log|V|. Για μια ουρά που υλοποιείται με αταξινόμητη λίστα, η λειτουργία extract-min έχει πολυπλοκότητα O(|V|) (ολόκληρη η ουρά πρέπει να διασχίζεται) και αυτό το 2ο μέρος γίνεται O (|V|^2) .

Για το πρώτο τμήμα του αθροίσματος (με τον παράγοντα των ακμών |E|), το O(1) έναντι στο O(log|V|), η διαφορά προέρχεται από την χρήση ενός σωρού Fibonacci, σε σχέση με ένα δυαδικό σωρό. Η λειτουργία decrease-key που μπορεί να συμβεί σε κάθε άκρη έχει ακριβώς αυτήν την πολυπλοκότητα. Έτσι, το υπόλοιπο μέρος του αθροίσματος έχει τελικά πολυπλοκότητα O(|E|) για ένα σωρό Fibonacci και O(|E| log|V|) για ένα δυαδικό σωρό. Για μια ουρά που υλοποιείται με αταξινόμητη λίστα, η λειτουργία decrease-key θα έχει μια σταθερή-πολυπλοκότητα χρόνου (η ουρά αποθηκεύει άμεσα τα κλειδιά στο ευρετήριο από τις κορυφές) και αυτό το μέρος του ποσού ως εκ τούτου θα είναι O (|E|), με αποτέλεσμα η συνολική πολυπλοκότητα να είναι επίσης O (|V|^2).

### Πολυπλοκότητα Αλγόριθμου Breadth First Search

Η χρονική πολυπλοκότητα του Breadth First Search μπορεί να εκφραστεί ως O(|V| + |E|), δεδομένου ότι κάθε κόμβος και κάθε ακμή θα πρέπει να διερευνηθούν, στη

[22]http://bigocheatsheet.com/
[23]http://stackoverflow.com/questions/21065855/the-big-o-on-the-dijkstra-fibonacci-heap-solution

χειρότερη περίπτωση. |V| είναι ο αριθμός των κόμβων και |E| είναι ο αριθμός των ακμών στο γράφημα. Σημειώνεται ότι το O(|E|) μπορεί να κυμαίνεται από O (1) και ως O(|V|^2), ανάλογα με το πόσο αραιό είναι το γράφημα.

Όταν ο αριθμός των κόμβων του γράφου είναι γνωστός εκ των προτέρων, και οι πρόσθετες δομές δεδομένων που χρησιμοποιούνται για να καθοριστεί ποιοι κόμβοι έχουν ήδη προστεθεί στην ουρά, η πολυπλοκότητα χώρου μπορεί να εκφραστεί ως O(|V|), όπου |V| είναι ο πληθάριθμος του συνόλου των κόμβων. Αν ο γράφος αντιπροσωπεύεται από μια λίστα γειτνίασης καταλαμβάνει Θ(|V| + |E|) χώρο στη μνήμη, ενώ η αναπαράσταση του πίνακα γειτνίασης καταλαμβάνει Θ(|V|^2).

### Πολυπλοκότητα Αλγόριθμου A*

Η χρονική πολυπλοκότητα του A* εξαρτάται από την ευρετική[24]. Για να είναι αποτελεσματικός, η αναζήτηση του A* πρέπει να αποθηκεύσει σε μια ουρά προτεραιότητας τους κόμβους που πρέπει να διερευνηθούν και να είναι σε θέση να καλέσει decrease-key με βάση τις προτεραιότητες της ουράς. Ο χρόνος εκτέλεσης για αυτό είναι, στη χειρότερη περίπτωση, O(nlogn+m), εφόσον υλοποιηθεί χρησιμοποιώντας μια καλή ουρά προτεραιότητας.

Ως εκ τούτου, στη χειρότερη περίπτωση, ο A * μπορεί να υποβαθμιστεί σε αλγόριθμο του Dijkstra. Με δεδομένη μια καλή ευρετική, ο A* δεν θα επεκταθεί σε όλους τους κόμβους και ακμές, όπως ο αλγόριθμος Dijkstra, ο οποίος είναι και ο κύριος λόγος για τον οποίο A* είναι ταχύτερος.

Για τη χρονική πολυπλοκότητα του A* πρέπει να ληφθεί υπόψη το κόστος υπολογισμού της ευρετικής. Κάποιες ευρετικές μπορεί να μην υπολογίζονται σε χρόνο O(1) ως αναμένεται, και σε αυτή την περίπτωση ο χρόνος εκτέλεσης για τον A* θα μπορούσε να είναι μεγαλύτερος από αυτόν του αλγόριθμου Dijkstra.

[24]http://stackoverflow.com/questions/19869236/why-does-a-run-faster-than-dijkstras-algorithm

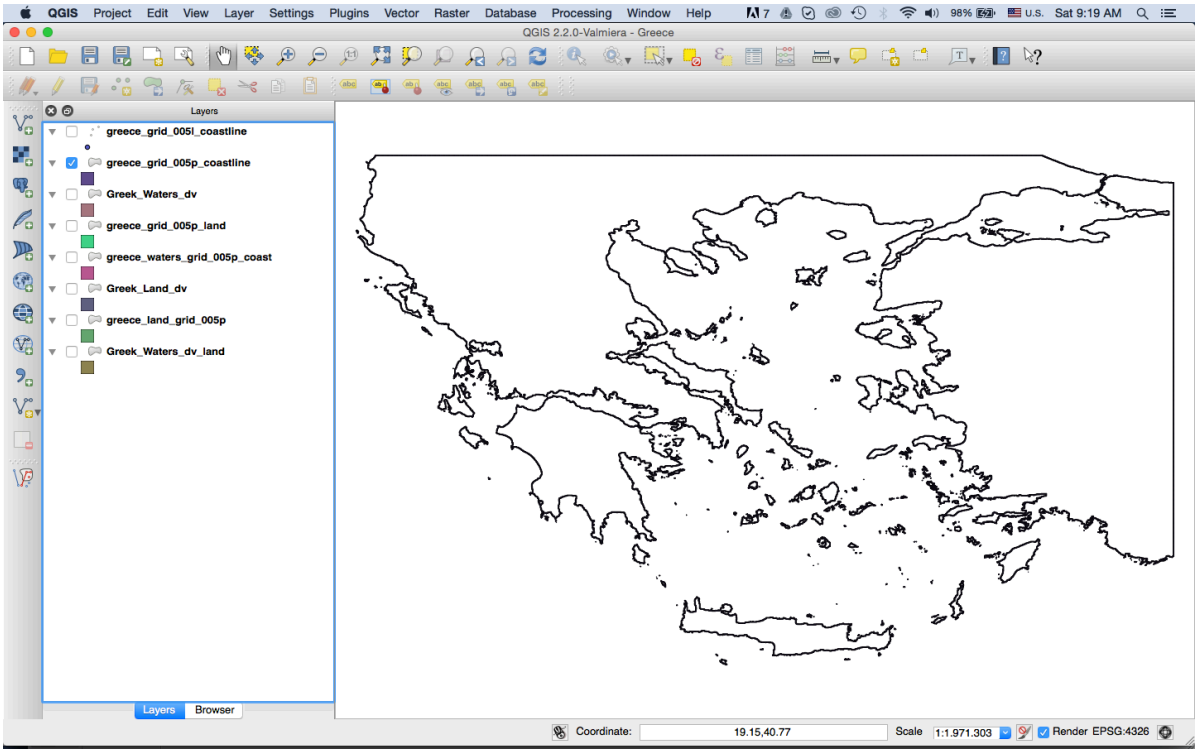
## Κεφάλαιο 4 - Εργαλεία Ανοικτού Κώδικα για την Ανάπτυξη του Συστήματος

Για την ανάπτυξη του συστήματος χρησιμοποιήθηκαν εργαλεία και τεχνολογίες ανοικτού κώδικα. Τα πιο σημαντικά από αυτά είναι:

- **QGIS** – Σύστημα για την ανάλυση γεωχωρικών δεδομένων
- **Apache HTTP Server** – Εξυπηρετητής για την φιλοξενία του κώδικα
- **HTML/ CSS / Javascript** – Γλώσσες Προγραμματισμού Web Εφαρμογών
- **Google Maps API** –Javascript βιβλιοθήκη για το σχεδιασμό χαρτών στο web
- **Pathfinding.js** – Javascript βιβλιοθήκη αλγόριθμων εύρεσης συντομότερων διαδρομών
- **JSON notation** – Δομή δεδομένων για την εισαγωγή πληροφοριών
- **Web SQL Database** - Τοπική βάση δεδομένων στον browser για την αποθήκευση των γεωγραφικών συντεταγμένων και των πληροφοριών καιρού των κόμβων του πλέγματος

# QGIS

Το QGIS (παλαιότερα γνωστό και ως Quantum GIS) είναι ένα δωρεάν, cross-platform και ανοικτού κώδικα (open-source) σύστημα γεωγραφικών πληροφοριών (GIS) που παρέχει τη δυνατότητα για προβολή, επεξεργασία και ανάλυση γεωχωρικών δεδομένων.




Εικόνα 25 - Σύστημα QGIS

Παρόμοια με άλλα συστήματα λογισμικού GIS, το QGIS επιτρέπει στους χρήστες να δημιουργήσουν χάρτες με πολλά επίπεδα (layers), χρησιμοποιώντας διαφορετικές

προβολές χάρτη (map projections). Οι χάρτες μπορούν να συντεθούν σε διαφορετικές μορφές και για διαφορετικές χρήσεις. Το QGIS επιτρέπει να δημιουργηθούν χάρτες που αποτελούνται από raster ή διανυσματικά (vector) επιθέματα. Τα διανυσματικά δεδομένα αποθηκεύονται είτε ως σημεία, γραμμές ή πολύγωνα. Υποστηρίζει διάφορα είδη από raster εικόνες, και μπορεί να κάνει γεωαναφορά εικόνων.

Το QGIS διασυνδέεται με άλλα πακέτα GIS ανοικτού κώδικα, συμπεριλαμβανομένων των PostGIS, GRASS, και MapServer και παρέχει στους χρήστες εκτεταμένες λειτουργίες. Τα plugins επεκτείνουν τις δυνατότητες του QGIS και είναι γραμμένα σε Python ή C ++. Για παράδειγμα, plugins μπορούν να κάνουν γεωκωδικοποίηση χρησιμοποιώντας το API του Google Geocoding, να εκτελέσουν γεωεπεξεργασίες χρησιμοποιώντας fTools, τα οποία είναι παρόμοια με τα τυποποιημένα εργαλεία που υπάρχουν στο ArcGIS, και να κάνουν διασύνδεση με PostgreSQL/PostGIS, SpatiaLite και MySQL βάσεις δεδομένων.

Ο Gary Sherman[25]ξεκίνησε την ανάπτυξη του Quantum GIS στις αρχές του 2002. Ακολούθως έγινε ένα βασικό έργο του οργανισμού Open Source Geospatial το 2007. Η 1η έκδοση κυκλοφόρησε τον Ιανουάριο του 2009.

Το QGIS είναι γραμμένο σε C ++ και κάνει εκτεταμένη χρήση της βιβλιοθήκης Qt. Επιτρέπει την ενσωμάτωση plugins που αναπτύσσονται χρησιμοποιώντας είτε C ++ ή Python. Εκτός από την Qt, το QGIS βασίζεται στο GEOS και SQLite. Οι βιβλιοθήκες GDAL, GRASS GIS, PostGIS, και PostgreSQL είναι σημαντικές, δεδομένου ότι παρέχουν πρόσβαση σε πρόσθετες μορφές δεδομένων.

Το QGIS τρέχει σε πολλά λειτουργικά συστήματα, συμπεριλαμβανομένων Mac OS X, Linux, UNIX και Microsoft Windows. Για τους χρήστες Mac, το πλεονέκτημα του QGIS σε σχέση με το GRASS GIS είναι ότι δεν απαιτεί το σύστημα παραθύρων X11, προκειμένου να τρέξει, και το interface είναι πολύ καθαρότερο και πιο γρήγορο. Το QGIS μπορεί επίσης να χρησιμοποιηθεί και ως μια γραφική διεπαφή χρήστη για το σύστημα GRASS. Το QGIS έχει μικρό μέγεθος αρχείου σε σύγκριση με τις εμπορικές λύσεις GIS και απαιτεί λιγότερη μνήμη RAM και ισχύ επεξεργασίας, ως εκ τούτου μπορεί να χρησιμοποιηθεί σε παλαιότερης τεχνολογίας hardware ή να τρέχει ταυτόχρονα με άλλες εφαρμογές, όπου η CPU μπορεί να είναι περιορισμένη.

Το QGIS συντηρείται από εθελοντές προγραμματιστές που εκδίδουν τακτικά ενημερώσεις και διορθώσεις σφαλμάτων. Από το 2012, προγραμματιστές έχουν μεταφράσει το QGIS σε 48 γλώσσες και η εφαρμογή χρησιμοποιείται διεθνώς σε ακαδημαϊκά αλλά και επαγγελματικά περιβάλλοντα.

# Apache HTTP Server

Ο Apache είναι ένας εξυπηρετητής του παγκόσμιου ιστού (web). Όποτε ένας χρήστης επισκέπτεται ένα ιστότοπο το πρόγραμμα πλοήγησης (browser) επικοινωνεί με έναν διακομιστή (server) μέσω του πρωτοκόλλου HTTP, ο οποίος παράγει τις ιστοσελίδες και τις αποστέλλει στο πρόγραμμα πλοήγησης.

[25]https://en.wikipedia.org/wiki/QGIS

Ο Apache είναι ένας από τους δημοφιλέστερους εξυπηρετητές ιστού, εν μέρει γιατί λειτουργεί σε διάφορες πλατφόρμες όπως τα Windows, το Linux, το Unix και το Mac OS X. Κυκλοφόρησε υπό την άδεια λογισμικού Apache και είναι λογισμικό ανοικτού κώδικα. Συντηρείται από μια κοινότητα ανοικτού κώδικα με επιτήρηση από το Ίδρυμα Λογισμικού Apache (Apache Software Foundation).

Ο Apache χρησιμοποιείται και σε τοπικά δίκτυα σαν εξυπηρετητής συνεργαζόμενος με συστήματα διαχείρισης Βάσης Δεδομένων π.χ. Oracle, MySQL.

Η πρώτη του έκδοση, γνωστή ως NCSA HTTPd, δημιουργήθηκε από τον Robert McCool και κυκλοφόρησε το 1993. Θεωρείται ότι έπαιξε σημαντικό ρόλο στην αρχική επέκταση του παγκόσμιου ιστού. Ήταν η πρώτη βιώσιμη εναλλακτική επιλογή που παρουσιάστηκε απέναντι στον εξυπηρετητή http της εταιρείας Netscape και από τότε έχει εξελιχθεί σε σημείο να ανταγωνίζεται άλλους εξυπηρετητές βασισμένους στο Unix σε λειτουργικότητα και απόδοση.

Από το 1996 είναι ο πιο δημοφιλής εξυπηρετητής, όμως από το 2006 έχει κύριο ανταγωνιστή τον Microsoft Internet Information Services και την πλατφόρμα .NET. Το 2015 το μερίδιο του, από όλους τους ιστοτόπους, ήταν γύρω στο 37%.[26]

# Δομικά Στοιχεία Web Εφαρμογών

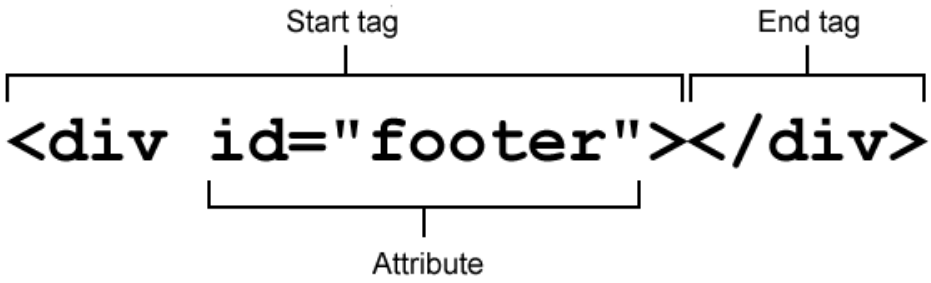
Το Web Standards Curriculum, αναφέρει ως τα βασικά δομικά στοιχεία (building blocks) των Web εφαρμογών[27] τις γλώσσες HTML, CSS and JavaScript, οι οποίες αλληλεπιδρούν με αρμονικό τρόπο και δημιουργούν μια web εφαρμογή.

### HTML

Η HTML είναι μια markup γλώσσα που αποτελείται από στοιχεία (elements), τα οποία περιέχουν χαρακτηριστικά (attributes), εκ των οποίων κάποια είναι προαιρετικά και κάποια υποχρεωτικά. Τα στοιχεία χρησιμοποιούνται για τη σήμανση των διαφόρων τύπων περιεχομένου σε web σελίδες, προσδιορίζοντας το κάθε κομμάτι του περιεχομένου πως πρέπει να αποδοθεί στον web broswer (π.χ. επικεφαλίδες, παραγράφους, πίνακες, λίστες με κουκκίδες, κλπ.).

Τα στοιχεία καθορίζουν τον βασικό τύπο του περιεχομένου, ενώ τα χαρακτηριστικά καθορίζουν επιπλέον πληροφορίες σχετικά με τα στοιχεία αυτά, όπως ένα αναγνωριστικό για την αναγνώριση αυτού του στοιχείου, ή μια θέση για μια διασύνδεση. Θα πρέπει να ληφθεί υπόψη ότι η σήμανση (markup) είναι η όσο το δυνατόν σημασιολογική, δηλαδή περιγράφει με σαφήνεια τη λειτουργία του περιεχομένου. Το παρακάτω σχήμα δείχνει την ανατομία ενός τυπικού στοιχείου.

[26] http://news.netcraft.com/archives/category/web-server-survey/
[27] https://www.w3.org/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript




Εικόνα 26 - Ανατομία ως HTML στοιχείου

### CSS

Τα Cascading Style Sheets προσδιορίζουν τη μορφοποίηση και διάταξη ενός web εγγράφου. Το CSS λειτουργεί με βάση ένα σύστημα κανόνων, το οποίο επιλέγει τα στοιχεία που χρειάζονται μορφοποίηση και ορίζονται τιμές για τις διαφορετικές ιδιότητες στυλ των στοιχείων. Μπορούν να αλλαχθούν ή να προστεθούν χρώματα, υπόβαθρα, μεγέθη γραμματοσειράς και στυλ, ακόμη και η διάταξη των αντικειμένων στην ιστοσελίδα σε διαφορετικές θέσεις. Εδώ είναι ένα παράδειγμα κανόνα CSS:

```
p {
        line-height: 2;
        color: green;
}
```

Με βάση τον παραπάνω κανόνα, οποιοδήποτε περιεχόμενο που περικλείεται μέσα σε <p></ p> tags θα έχει διπλό το ύψος της γραμμής και θα είναι χρωματισμένο πράσινο.

### Javascript

Η JavaScript είναι scripting γλώσσα που χρησιμοποιείται για να προστεθεί διαδραστικότητα στις web σελίδες. Για παράδειγμα, μπορεί να χρησιμοποιηθεί για να επικυρώσει δεδομένα, να εξετάσει αν είναι στη σωστή μορφή ή όχι, να παρέχει drag and drop λειτουργικότητα, να κάνει αλλαγή στυλ μετά την φόρτωση της σελίδας, να μετακινήσει στοιχεία, όπως μενού, να χειριστεί τη λειτουργικότητα των κουμπιών, και να κάνει πολλά άλλα πράγματα. Η σύγχρονη Javascript λειτουργεί κυρίως με την εύρεση ενός στοιχείου HTML στόχου, και στη συνέχεια κάνει κάτι σε αυτό, όπως και η CSS, αλλά ο τρόπος που λειτουργεί, η σύνταξη κλπ. είναι διαφορετική.

Η JavaScript, δεν πρέπει να συγχέεται με τη Java, δημιουργήθηκε σε 10 ημέρες το Μάιο του 1995 από τον Brendan Eich[28], τότε εργάζονταν στη Netscape και τώρα στο Mozilla. Η JavaScript ήταν αρχικά γνωστή ως Mocha, ένα όνομα που είχε επιλεγεί από τον Marc Andreessen, ιδρυτή της Netscape. Τον Σεπτέμβριο του 1995 το όνομα άλλαξε σε LiveScript, στη συνέχεια, τον Δεκέμβριο του ίδιου έτους, μετά τη λήψη της άδειας του εμπορικού σήματος από τη Sun, εκδόθηκε το όνομα JavaScript. Αυτό ήταν μια κίνηση μάρκετινγκ την περίοδο εκείνη, καθώς η Java ήταν πολύ δημοφιλής.

[28]https://en.wikipedia.org/wiki/Brendan_Eich

Την περίοδο 1996 - 1997 η JavaScript δόθηκε στον οργανισμό ECMA International [29] προκειμένου να προδιαγράψει ένα πρότυπο, το οποίο άλλοι κατασκευαστές προγραμμάτων περιήγησης θα μπορούσαν να εφαρμόσουν με βάση τη δουλειά που είχε γίνει στη Netscape. Το έργο που επιτελέστηκε κατά τη διάρκεια αυτής της περιόδου οδήγησε τελικά στην επίσημη κυκλοφορία του ECMA-262 Ed.1. ECMAScript είναι το επίσημο όνομα του προτύπου, με την Javascript να είναι πιο γνωστή από τις εφαρμογές του. ActionScript 3 είναι μια άλλη πολύ γνωστή εφαρμογή της ECMAScript, με προεκτάσεις.

Η διαδικασία προτυποποίησης συνεχίστηκε σε κύκλους, με κυκλοφορίες των ECMAScript 2 το 1998 και ECMAScript 3 το 1999, η οποία είναι η βάση για τη σύγχρονη JavaScript.

Από το 2005, κοινότητες προγραμματιστών ανοικτού κώδικα εργάστηκαν συστηματικά για να φέρουν την επανάσταση που ακολούθησε με την JavaScript. Ο Jesse James Garrett[30] δημοσίευσε τότε ένα white paper, στο οποίο επινόησε τον όρο Ajax, και περιέγραψε μια σειρά από τεχνολογίες, εκ των οποίων η JavaScript ήταν η ραχοκοκαλιά, που χρησιμοποιούνται για τη δημιουργία εφαρμογών web, όπου τα δεδομένα μπορούν να φορτωθούν στο παρασκήνιο, αποφεύγοντας την ανάγκη για πλήρη επαναφόρτωση της web σελίδας με αποτέλεσμα να αναπτύσσονται πιο δυναμικές εφαρμογές. Αυτό οδήγησε σε μια περίοδο αναγέννησης της χρήσης Javascript με αιχμή του δόρατος βιβλιοθήκες ανοικτού κώδικα και τις κοινότητες που σχηματίζονται γύρω τους, με βιβλιοθήκες όπως η Prototype, jQuery, Dojo και Mootools και άλλων που κυκλοφόρησαν στη συνέχεια.

Τον Ιούλιο του 2008, οι κοινότητες και η Ecma συναντήθηκαν από κοινού στο Όσλο. Αυτό οδήγησε σε συμφωνία στις αρχές του 2009 για να μετονομαστεί η ECMAScript 3.1 σε ECMAScript 5 και να οδηγήσει την περαιτέρω ανάπτυξη της γλώσσας με ένα πρόγραμμα που είναι γνωστό ως Harmony[31].

Σήμερα, η JavaScript εισέρχεται σε ένα εντελώς νέο και συναρπαστικό κύκλο εξέλιξης, καινοτομίας και τυποποίησης, με νέες αναπτύξεις, όπως η πλατφόρμα Nodejs, που επιτρέπει να χρησιμοποιηθεί η JavaScript server-side, τα HTML5 APIs που επιτρέπουν τον έλεγχο media χρηστών, να ανοιχτούνweb sockets για διαρκή επικοινωνία, να ληφθούν δεδομένα σχετικά με τη γεωγραφική θέση και χαρακτηριστικά συσκευών όπως το επιταχυνσιόμετρο, και πολλά άλλα.

# Google Maps API

Η Google λάνσαρε το Google Maps API τον Ιούνιο του 2005 για να επιτρέπει στους προγραμματιστές να ενσωματώσουν τους χάρτες της σε ιστοσελίδες τους. Είναι μια δωρεάν υπηρεσία, και προς το παρόν δεν περιέχει διαφημίσεις, αλλά η Google διατηρεί το δικαίωμα να προβάλλει διαφημίσεις στο μέλλον.

[29]http://www.ecma-international.org/
[30]https://en.wikipedia.org/wiki/Jesse_James_Garrett
[31]https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html

Με τη χρήση του Google Maps API, είναι δυνατόν να ενσωματωθεί ένας χάρτης με υπόβαθρο παραγόμενο από τα Google Maps σε ένα εξωτερικό website, επί του οποίου site μπορεί να επικαλύπτονται δεδομένα της εκάστοτε εφαρμογής. Αν και αρχικά υπήρχε μόνο το JavaScript API, στη συνέχεια επεκτάθηκε για να συμπεριλάβει API για εφαρμογές Adobe Flash, το οποίο καταργήθηκε στη συνέχεια, μια υπηρεσία για την ανάκτηση στατικών εικόνων χάρτη, και διαδικτυακών υπηρεσιών για την εκτέλεση γεωκωδικοποίησης, ανάκτηση οδηγιών πλοήγησης, και υψομετρικού προφίλ. Πάνω από 1.000.000 ιστοσελίδες χρησιμοποιούν το API Χαρτών Google[32], καθιστώντας το πιο διαδεδομένο web API για ανάπτυξη εφαρμογών.

Το Google Maps API διατίθεται δωρεάν για χρήση, υπό τον όρο ότι ο ιστότοπος στο οποίο χρησιμοποιείται είναι προσβάσιμος από το κοινό, δεν χρεώνει για την πρόσβαση και δεν καλεί το API περισσότερες από 25.000 φορές την ημέρα. Οι τοποθεσίες που δεν πληρούν αυτές τις απαιτήσεις μπορούν να αγοράσουν το Google Maps API για Επιχειρήσεις.

Η επιτυχία του Google Maps API έχει γεννήσει μια σειρά ανταγωνιστικών εναλλακτικών λύσεων, συμπεριλαμβανομένων των Bing Maps, Leaflet και OpenLayers.

Το Google Maps API επιτρέπει την προσαρμογή της παρουσίασης των τυποποιημένων χαρτών της Google (στυλ), αλλάζοντας την οπτικοποίηση των στοιχείων όπως δρόμοι, πάρκα, και κατοικημένες περιοχές.

Υπάρχουν δύο τρόποι για την εφαρμογή στυλ σε ένα χάρτη:

- Με τη ρύθμιση της .styles ιδιότητας του αντικειμένου MapOptions του χάρτη. Αυτή η προσέγγιση αλλάζει το στυλ των τυποποιημένων τύπων χάρτη.
- Με τη δημιουργία και τον καθορισμό ενός StyledMapType και την εφαρμογή του στο χάρτη. Αυτό δημιουργεί έναν νέο τύπο χάρτη, ο οποίος μπορεί να επιλεγεί από την επιλογή τύπου χάρτη.

Και οι δύο προσεγγίσεις λαμβάνουν μια σειρά από MapTypeStyles, καθένα από τα οποία αποτελείται από *selectors* και *stylers*. Οι *selectors* καθορίζουν ποια στοιχεία του χάρτη θα πρέπει να επιλεγούν για styling, ενώ οι *stylers* καθορίζουν την οπτική τροποποίηση των στοιχείων αυτών.

Για την αλλαγή του στυλ χρησιμοποιούνται δύο έννοιες για την εφαρμογή χρωμάτων και αλλαγών σε ένα χάρτη:

- *Map features* είναι τα γεωγραφικά στοιχεία, τα οποία μπορούν να υπάρχουν στο χάρτη. Αυτά περιλαμβάνουν δρόμους, πάρκα, υδατικά συστήματα, και πολλά άλλα, καθώς και τις ετικέτες τους.
- *Stylers* είναι ιδιότητες για το χρώμα και την χαρτογραφική απόδοση που μπορούν να εφαρμοστούν για να αποδώσουν τα Map features.

Τα **map features** και **stylers** συνδυάζονται σε μια συστοιχία από στυλ, η οποία αποδίδεται στο προεπιλεγμένο MapOptions αντικείμενο του χάρτη, ή στον constructor StyledMapType. Η συστοιχία παίρνει για παράδειγμα την ακόλουθη μορφή:

[32]http://googlegeodevelopers.blogspot.gr/2013/05/a-fresh-new-look-for-maps-api-for-all.html

```
var styleArray = [
  {
    featureType: "all",
    stylers: [
      { saturation: -80 }
    ]
  },{
    featureType: "road.arterial",
    elementType: "geometry",
    stylers: [
      { hue: "#00ffee" },
      { saturation: 50 }
    ]
  },{
    featureType: "poi.business",
    elementType: "labels",
    stylers: [
      { visibility: "off" }
    ]
  }
];
```

# Pathfinding.js

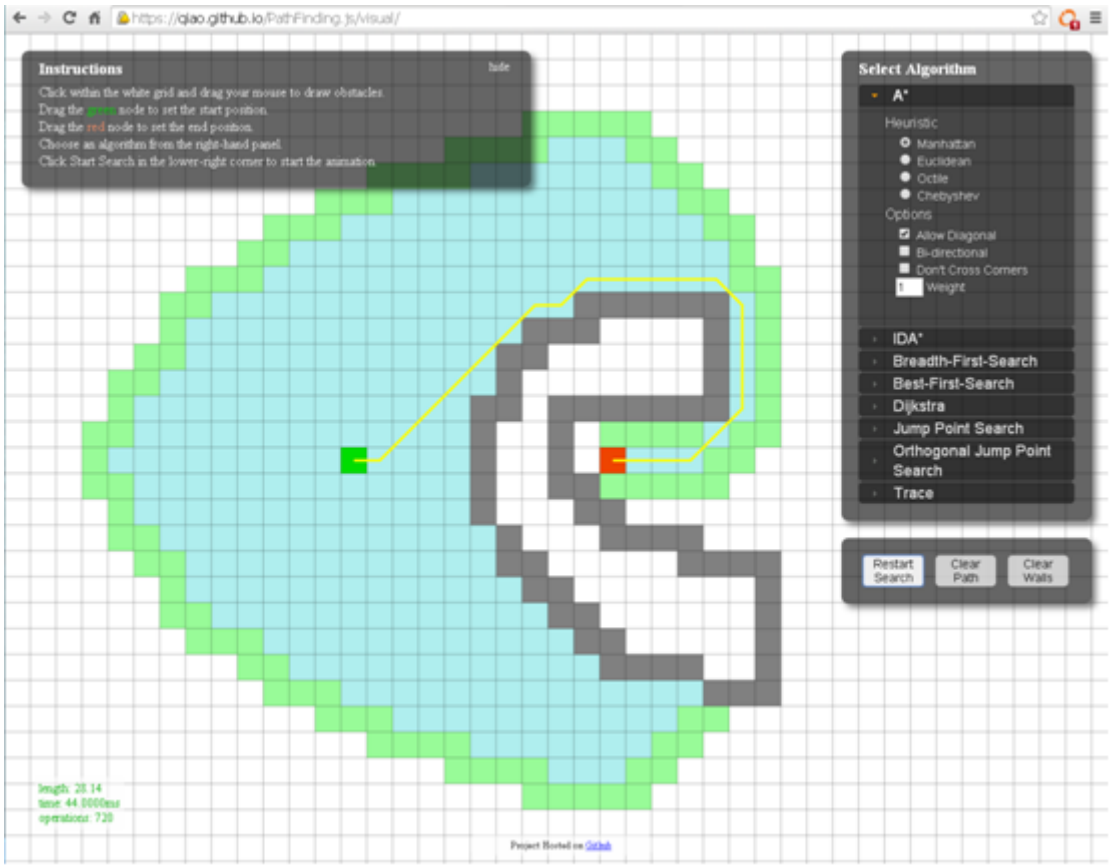
Στο πλαίσιο της φάσης μελέτης των αλγορίθμων εύρεσης βέλτιστων διαδρομών επιλέχθηκε η βιβλιοθήκηPathfinding.js που διατίθεται ως λογισμικό ανοικτού κώδικα στο Διαδίκτυο [33]. Στη βιβλιοθήκη PathFinding.js έχουν αναπτυχθεί σε γλώσσα javascript οι πιο διαδεδομένοι αλγόριθμοι για εύρεση βέλτιστης διαδρομής:

- A*
- IDA*
- Breadth First Search
- Best First Search
- Dijkstra

Η μελέτη και κατανόηση των αλγορίθμων της βιβλιοθήκης γίνεται με τη βοήθεια demo πλατφόρμας που διατίθεται στο Διαδίκτυο. Ο σχετικός σύνδεσμος είναι:

https://qiao.github.io/PathFinding.js/visual/

[33]https://github.com/qiao/PathFinding.js




**Εικόνα 27–Demo εργαλείο παρουσίασης βιβλιοθήκης Pathfinding.js**

# JSON notation

Η δομή – γλώσσα JSON (JavaScript Object Notation) είναι μια ελαφριά δομή δεδομένων που χρησιμοποιείται ευρέως για την ανταλλαγή δεδομένων. Είναι εύκολη για τους ανθρώπους να την διαβάσουν αλλά και να την γράψουν. Επίσης είναι εύκολη για τις μηχανές να αναλύσουν τα στοιχεία της, αλλά και να τη δημιουργήσουν. Βασίζεται σε ένα υποσύνολο της γλώσσας προγραμματισμού JavaScript, πρότυπο ECMA-262 (3ηέκδοση, Δεκέμβριος 1999).

Η δομή JSON είναι μια μορφή κειμένου που είναι εντελώς ανεξάρτητη από γλώσσα προγραμματισμού, αλλά χρησιμοποιεί συμβάσεις που οι προγραμματιστές της οικογένειας γλωσσών C είναι εξοικειωμένοι, συμπεριλαμβανομένων των C++, C#, Java, JavaScript, Perl, Python, και πολλών άλλων.[34]Αυτές οι ιδιότητες καθιστούν τη δομή JSON μια ιδανική γλώσσα για ανταλλαγή δεδομένων.

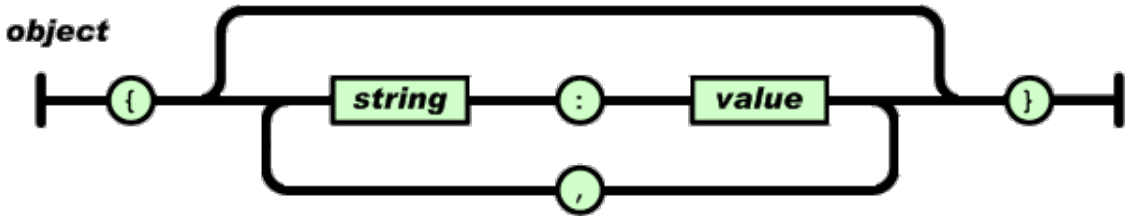
Η γλώσσα JSON βασίζεται σε δύο δομές:

- μια συλλογή από ζεύγη ονόματος/τιμής. Σε διάφορες γλώσσες, αυτό μπορεί να σημαίνει ότι αφορά αντικείμενο, εγγραφή, δομή, λεξικό, πίνακα hash, λίστα ή συσχετιζόμενο πίνακα.
- μια ταξινομημένη λίστα τιμών. Στις περισσότερες γλώσσες, η αντίστοιχη δομή μπορεί να αφορά πίνακα, διάνυσμα, λίστα, ή ακολουθία.

[34]http://www.json.org/

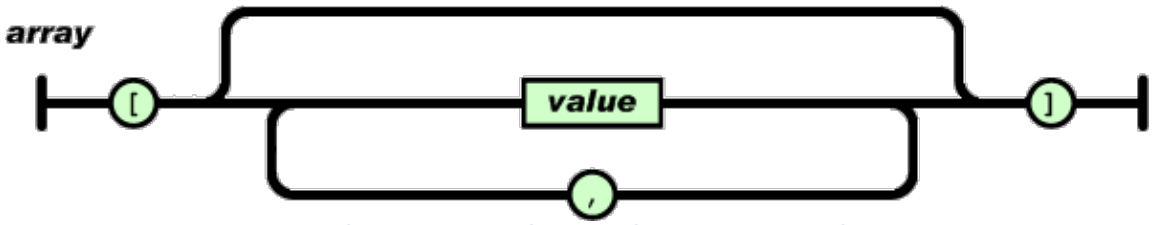
Αυτές είναι καθολικές δομές δεδομένων. Σχεδόν όλες οι σύγχρονες γλώσσες προγραμματισμού τις υποστηρίζουν σε μία ή την άλλη μορφή. Είναι λογικό ότι μια μορφή δεδομένων που είναι ανταλλάξιμη με γλώσσες προγραμματισμού θα πρέπει επίσης να βασίζεται σε αυτές τις δομές.

Στη δομή JSON, ένα αντικείμενο είναι μια μη διατεταγμένη σειρά από ζεύγη ονόματος/τιμής. Ένα αντικείμενο ξεκινά με { (αριστερό άγκιστρο) και τελειώνει με } (δεξιό άγκιστρο). Κάθε όνομα ακολουθείται από: (άνω κάτω τελεία) και τα ζεύγη ονόματα/τιμές χωρίζονται από, (κόμμα):



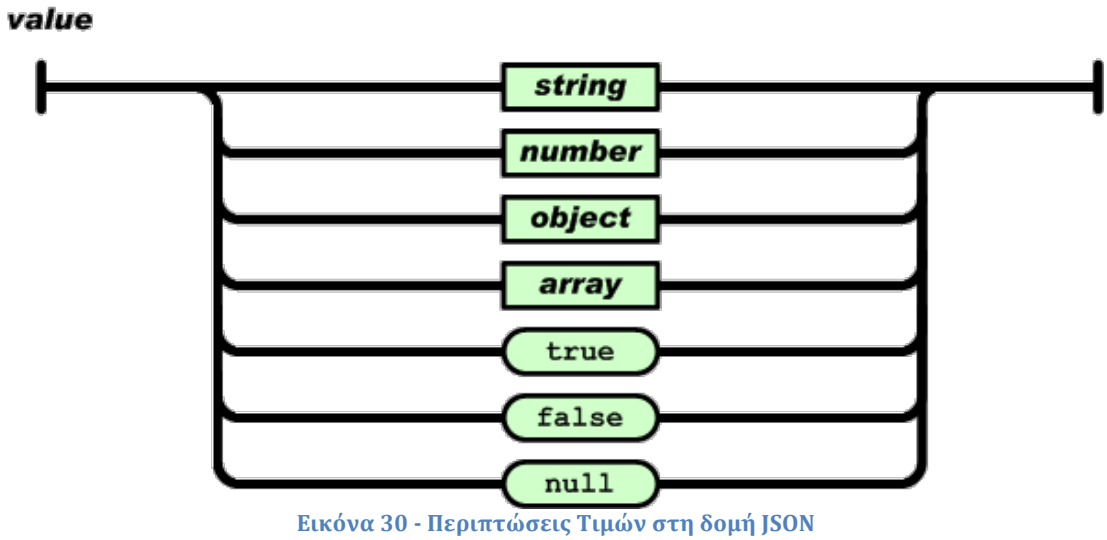

Εικόνα 28 - Αναπαράσταση Αντικειμένου σε JSON δομή

Από την άλλη, ένας πίνακας είναι μια διατεταγμένη συλλογή τιμών. Ο κάθε πίνακας αρχίζει με [(αριστερό βραχίονα) και τελειώνει με] (δεξιά αγκύλη). Οι τιμές χωρίζονται με, (κόμμα):




Εικόνα 29 - Αναπαράσταση Πίνακα σε JSON δομή

Μια τιμή μπορεί να είναι τύπου string σε διπλά εισαγωγικά, ή αριθμός, ή αληθής ή ψευδής ή μηδενική, ή αντικείμενο ή πίνακας. Οι δομές αυτές μπορεί να είναι ένθετες:




Εικόνα 30 - Περιπτώσεις Τιμών στη δομή JSON

# WebSQL Database

Web SQL Database είναι ένα API (Application Programming Interface) για ιστοσελίδες για την αποθήκευση δεδομένων σε βάσεις δεδομένων που μπορούν να ερωτηθούν χρησιμοποιώντας μια παραλλαγή της SQL. Το API υποστηρίζεται από τα προγράμματα περιήγησης Google Chrome, Opera, Safari και Android Browser.

Η Ομάδα Εργασίας του W3C για Web Εφαρμογές έπαψε να εργάζεται για την προδιαγραφή τον Νοέμβριο του 2010, επικαλούμενη την έλλειψη ανεξάρτητων εφαρμογών (δηλαδή, τη χρήση ενός συστήματος βάσης δεδομένων, εκτός από την SQLite ως backend) ως λόγο πως η προδιαγραφή δεν θα μπορούσε να κινηθεί προς τα εμπρός για να γίνει W3C σύσταση.

Το API περιλαμβάνει 3 βασικές μεθόδους[35]:

- **openDatabase** – δημιουργεί το αντικείμενο της βάσης, είτε χρησιμοποιώντας προϋπάρχουσα βάση ή δημιουργώντας νέα.
- **transaction** – δίνει την δυνατότητα να ελέγχει μια συναλλαγή και να κάνει είτε commit ή roll-back ανάλογα την κατάσταση.
- **executeSql** –χρησμοποιείται για να εκτελέσει τα επιμέρους SQL ερωτήματα.

[35]http://www.tutorialspoint.com/html5/html5_web_sql.htm

## Κεφάλαιο 5 – Ανάπτυξη του Συστήματος

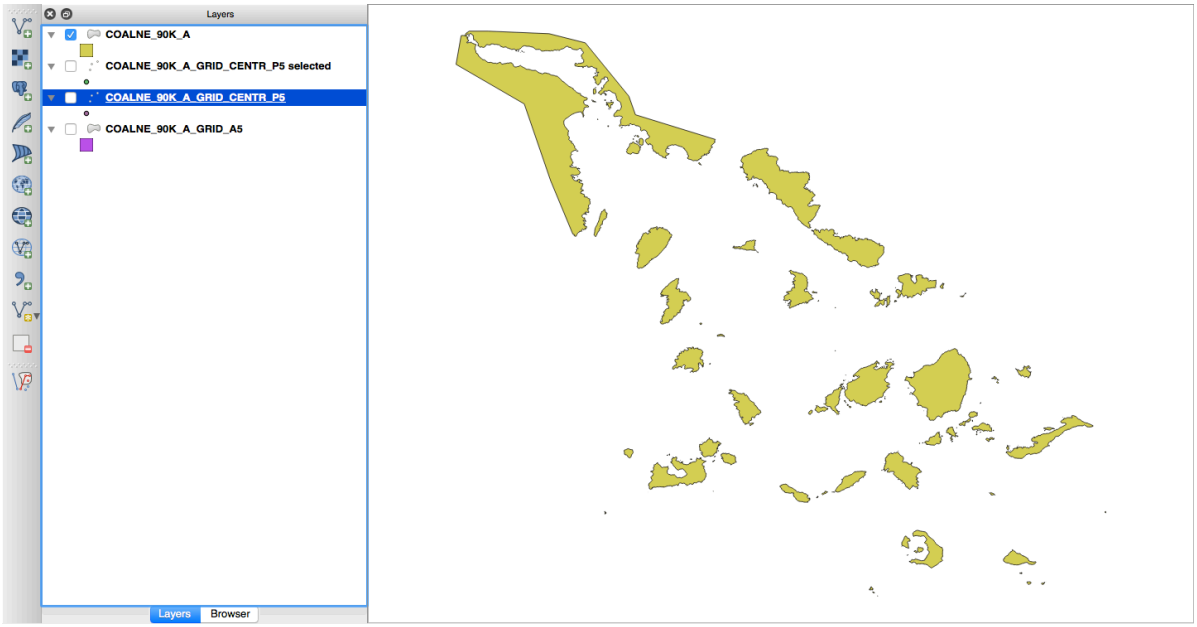
# Ανάπτυξη Πρωτότυπης Διάταξης

Όπως αναφέρθηκε στο προηγούμενο κεφάλαιο, κατά τη φάση μελέτης των αλγορίθμων εύρεσης βέλτιστων διαδρομών επιλέχθηκε η βιβλιοθήκη Pathfinding.js που διατίθεται ως λογισμικό ανοικτού κώδικα στο Διαδίκτυο. Στη βιβλιοθήκη PathFinding.js έχουν αναπτυχθεί σε γλώσσα javascript οι αλγόριθμοι που εξετάστηκαν στο Κεφάλαιο 3 για την εύρεση βέλτιστης διαδρομής.

Με βάση το εργαλείο που συνοδεύει τη βιβλιοθήκη Pathfinding.js[36] για την οπτική απεικόνιση των αποτελεσμάτων της, αναπτύχθηκε πρωτότυπη web εφαρμογή (prototype) για τη δοκιμαστική χρήση των αλγορίθμων στην περιοχή των Κυκλάδων.

### Δημιουργία Πλέγματος για τις Κυκλάδες

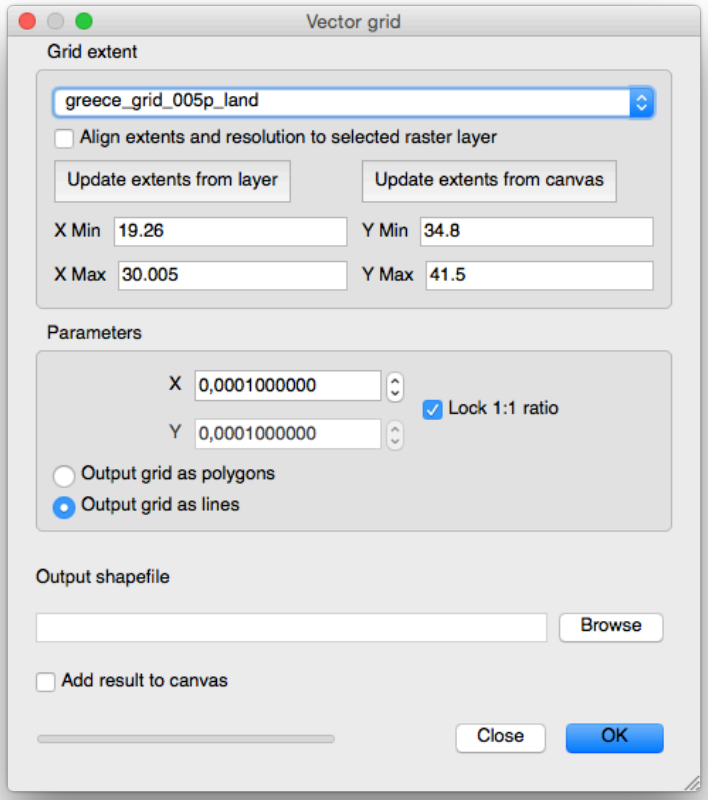
Η περιοχή των Κυκλάδων διατέθηκε ως shapefile από την Υδρογραφική Υπηρεσία του Πολεμικού Ναυτικού και μετατράπηκε σε πλέγμα (grid).




Εικόνα 31 - Shape file Κυκλάδων

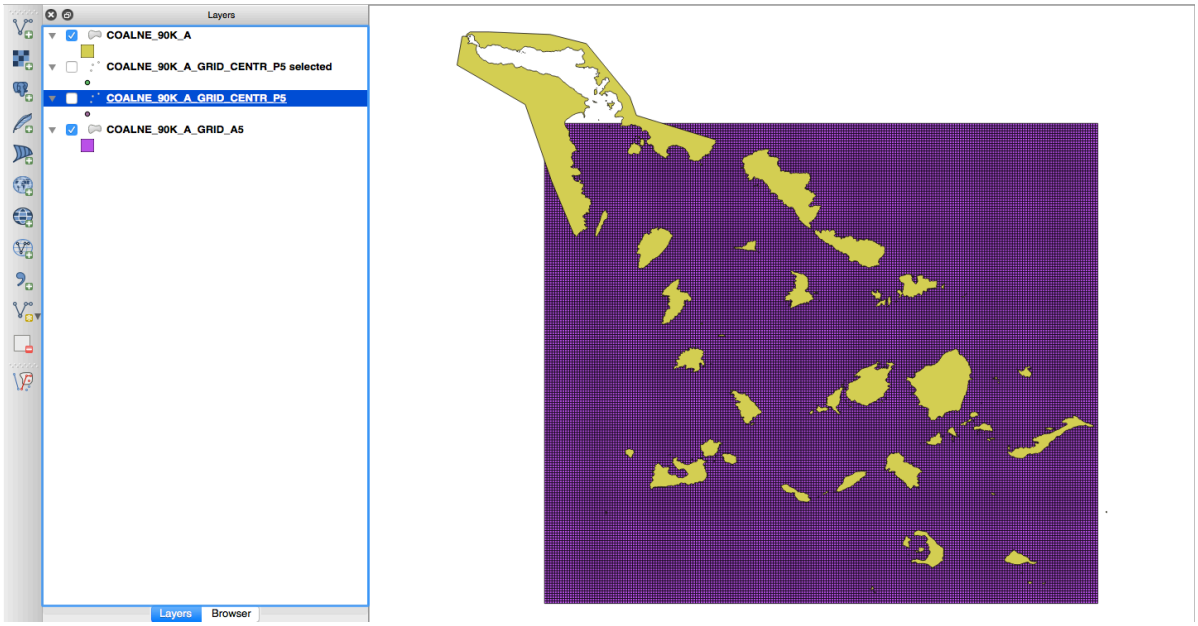
Η δημιουργία του πλέγματος έγινε με τη βοήθεια του εργαλείου QGIS. Η δυνατότητα δίνεται στο menu Vector ->.Research Tools ->Vector Grid. Το πλέγμα δημιουργήθηκε μεταξύ των ακραίων σημείων με Γεωγραφικό Μήκος από 23,9 μοίρες έως 26,1 μοίρες και Γεωγραφικό Πλάτος από 36,2 μοίρες έως 38,1 μοίρες, στο προβολικό σύστημα EPSG:4326 (WGS84)[37].

[36] http://qiao.github.io/PathFinding.js/visual/
[37] https://nsidc.org/data/atlas/epsg_4326.html



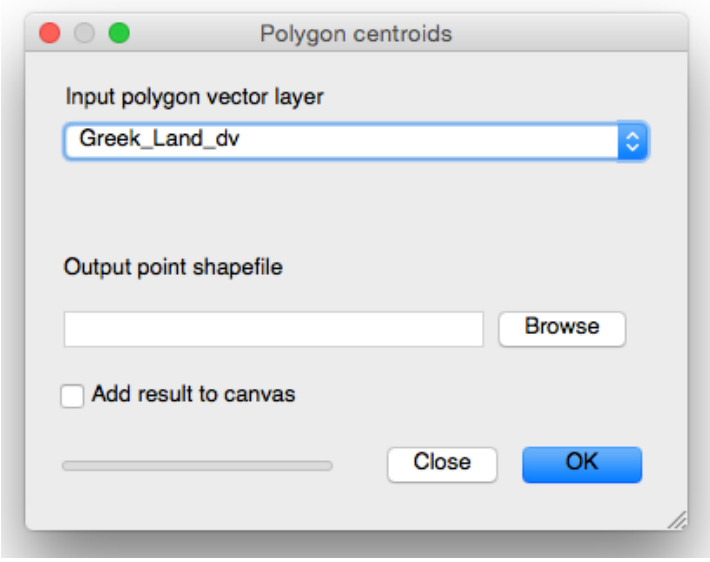

Εικόνα 32 - Δημιουργία Grid στο QGIS

Το πλάτος και το ύψος του μοναδιαίου φατνίου του πλέγματος επιλέχθηκε να είναι το ένα εκατοστό της μοίρας (0,01) του προβολικού συστήματος EPSG:4326 (WGS84) και το πλέγμα που δημιουργήθηκε αντιστοιχεί σε πίνακα διαστάσεων 220x190 κόμβων.



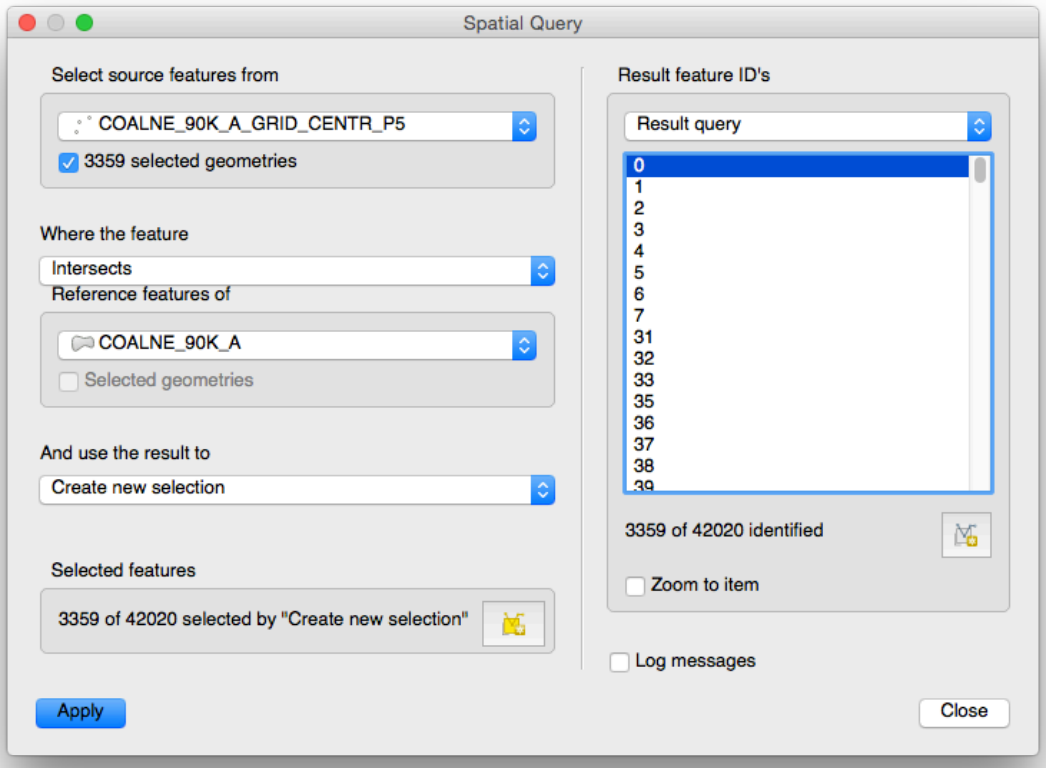

Εικόνα 33 - Εμφάνιση του Grid σε layer του QGIS

Στη συνέχεια τα πολύγωνα μετατράπηκαν σε σημεία μέσω της επιλογής Geometry tools -> Polygon centroids.




Εικόνα 34 - Δημιουργία Σημείων από Πολύγωνα

Ακολούθως έγινε χωρική ερώτηση από όπου επιλέχτηκαν τα σημεία του grid που τέμνονται με τη στεριά.




Εικόνα 35 - Χωρική ερώτηση για επιλογή σημείων στεριάς

Τελικά δημιουργήθηκε επίπεδο (layer) σημείων (points) που αφορά υποσύνολο του grid που περιλαμβάνει τα σημεία που ανήκουν στη στεριά.


Εικόνα 36 - Εμφάνιση των σημείων στεριάς στο QGIS

Αυτά τα σημεία έγιναν export σε αρχείο csv και import στην εφαρμογή ως σημεία που ο εκάστοτε αλγόριθμος θεωρεί ως NonWalkable.

Εικόνα 37 - Εμφάνιση των σημείων στεριάς στο QGIS (zoom)

### Αποτελέσματα Πρωτότυπης Διάταξης

Η πρωτότυπη web εφαρμογή παραμετροποιήθηκε ώστε να αποδώσει οπτικά με δύο τρόπους την εφαρμογή του κάθε αλγόριθμου:

(i) με χρήση του γράφου και οπτική απεικόνιση των pixels στα οποία εφαρμόστηκε ο αλγόριθμος,
(ii) απευθείας σε webmap υπόβαθρο όπως τα Google Maps.

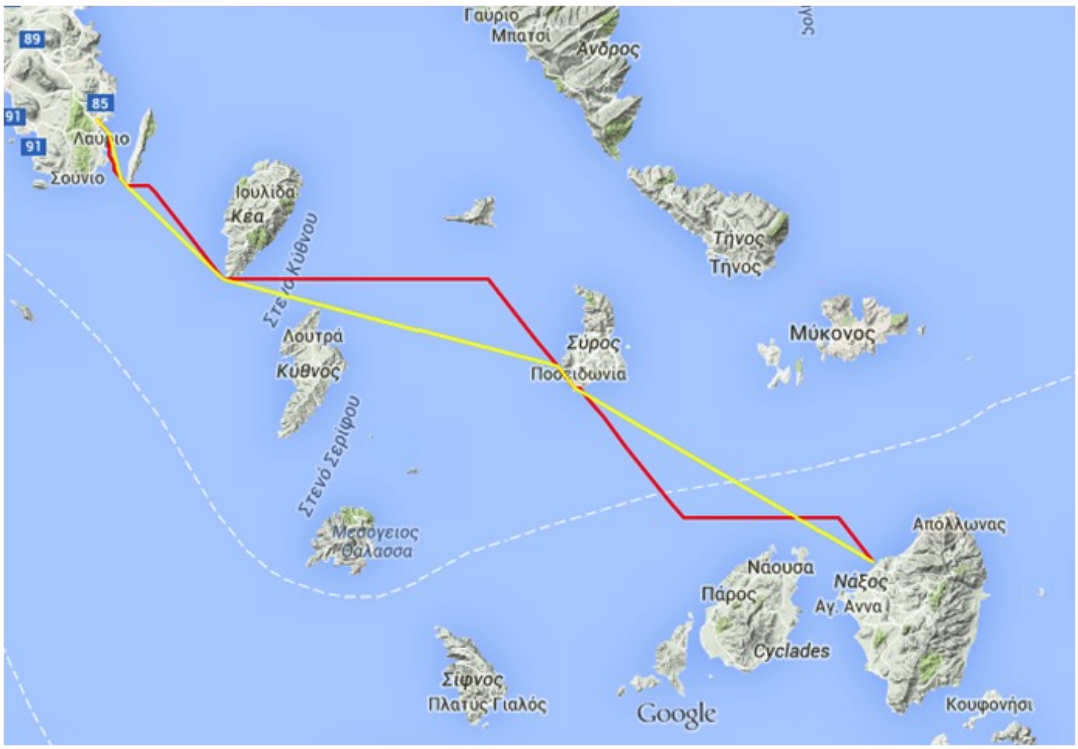
Στη συνέχεια αποτυπώνεται η χρήση της εφαρμογής για τον υπολογισμό της βέλτιστης διαδρομής μεταξύ Λιμένος Λαυρίου και Νάξου.

## 1i) Απεικόνιση της εφαρμογής του αλγόριθμου Dijkstra σε πλέγμα

Στην παρακάτω εικόνα γίνεται εφαρμογή του αλγόριθμου Dijkstra σε πλέγμα. Με γαλάζιο χρώμα αποδίδονται τα φατνία τα οποία ο αλγόριθμος υπολόγισε μέχρι να βρει την βέλτιστη διαδρομή, με ανοικτό πράσινο τα φατνία που υπολόγισε τελευταία και με κίτρινο χρώμα αποδίδεται η βέλτιστη διαδρομή που υπολογίστηκε.


Εικόνα 38 - Εφαρμογή του αλγόριθμου Dijkstra σε πλέγμα

## 1ii) Απεικόνιση της εφαρμογής του αλγόριθμου Dijkstra σε webmap




**Εικόνα 39 - Εφαρμογή του αλγόριθμου Dijkstra σε web map**

Στην παραπάνω εικόνα, όπου γίνεται εφαρμογή του αλγόριθμου Dijkstra σε web map, η κόκκινη γραμμή αφορά την αρχική απεικόνιση της εφαρμογής του αλγόριθμου (είναι ακριβώς ίδια με την υπολογισμένη διαδρομή πάνω στο πλέγμα) και η κίτρινη γραμμή αφορά την εφαρμογή αλγόριθμου εξομάλυνσης (smouth) της διαδρομής που διαθέτει η βιβλιοθήκηPathfinding.js.

```
/**
 * Smoothen the given path.
*/
function smoothenPath(grid, path) {
    var len = path.length,
        x0 = path[0][0],        // path start x
        y0 = path[0][1],        // path start y
        x1 = path[len - 1][0],  // path end x
        y1 = path[len - 1][1],  // path end y
        sx, sy,                 // current start coordinate
        ex, ey,                 // current end coordinate
        newPath,
        i, j, coord, line, testCoord, blocked;

    sx = x0;
    sy = y0;
    newPath = [[sx, sy]];

    for (i = 2; i < len; ++i) {
        coord = path[i];
        ex = coord[0];
        ey = coord[1];
        line = interpolate(sx, sy, ex, ey);

        blocked = false;
```

```
        for (j = 1; j < line.length; ++j) {
            testCoord = line[j];

            if (!grid.isWalkableAt(testCoord[0], testCoord[1])) {
                blocked = true;
                break;
            }
        }
        if (blocked) {
            lastValidCoord = path[i - 1];
            newPath.push(lastValidCoord);
            sx = lastValidCoord[0];
            sy = lastValidCoord[1];
        }
    }
    newPath.push([x1, y1]);

    return newPath;
}

/**
 * Given the start and end coordinates, return all the coordinates lying
 * on the line formed by these coordinates, based on Bresenham's
algorithm.[38]
 * @param {number} x0 Start x coordinate
 * @param {number} y0 Start y coordinate
 * @param {number} x1 End x coordinate
 * @param {number} y1 End y coordinate
 */
function interpolate(x0, y0, x1, y1) {
    var abs = Math.abs,
        line = [],
        sx, sy, dx, dy, err, e2;

    dx = abs(x1 - x0);
    dy = abs(y1 - y0);

    sx = (x0 < x1) ? 1 : -1;
    sy = (y0 < y1) ? 1 : -1;

    err = dx - dy;

    while (true) {
        line.push([x0, y0]);

        if (x0 === x1 && y0 === y1) {
            break;
        }

        e2 = 2 * err;
        if (e2 > -dy) {
            err = err - dy;
            x0 = x0 + sx;
        }
        if (e2 < dx) {
            err = err + dx;
            y0 = y0 + sy;
        }
    }
```

[38]http://en.wikipedia.org/wiki/Bresenham's_line_algorithm#Simplification

```
    return line;
}
```

## 2i) Απεικόνιση της εφαρμογής του αλγόριθμου A*σε πλέγμα


Εικόνα 40 - Εφαρμογή του αλγόριθμου A*σε πλέγμα

## 2ii) Απεικόνιση της εφαρμογή του αλγόριθμου A*σε webmap


Εικόνα 41 - Εφαρμογή του αλγόριθμου A*σε web map

## Μετρήσεις Πρωτότυπης Διάταξης

Στον παρακάτω πίνακα αποτυπώνονται οι πρώτες μετρήσεις που έγιναν με την εφαρμογή των αλγορίθμων. Από τις μετρήσεις καταδεικνύεται πως ο αλγόριθμος Dijkstra υπολογίζει τη συντομότερη διαδρομή αλλά με πολλαπλάσιο αριθμό υπολογισμών (operations), ενώ ο αλγόριθμός A* υπολογίζει μια διαδρομή πολύ κοντά στη βέλτιστη και με πολύ μικρό αριθμό υπολογιστικών κύκλων.

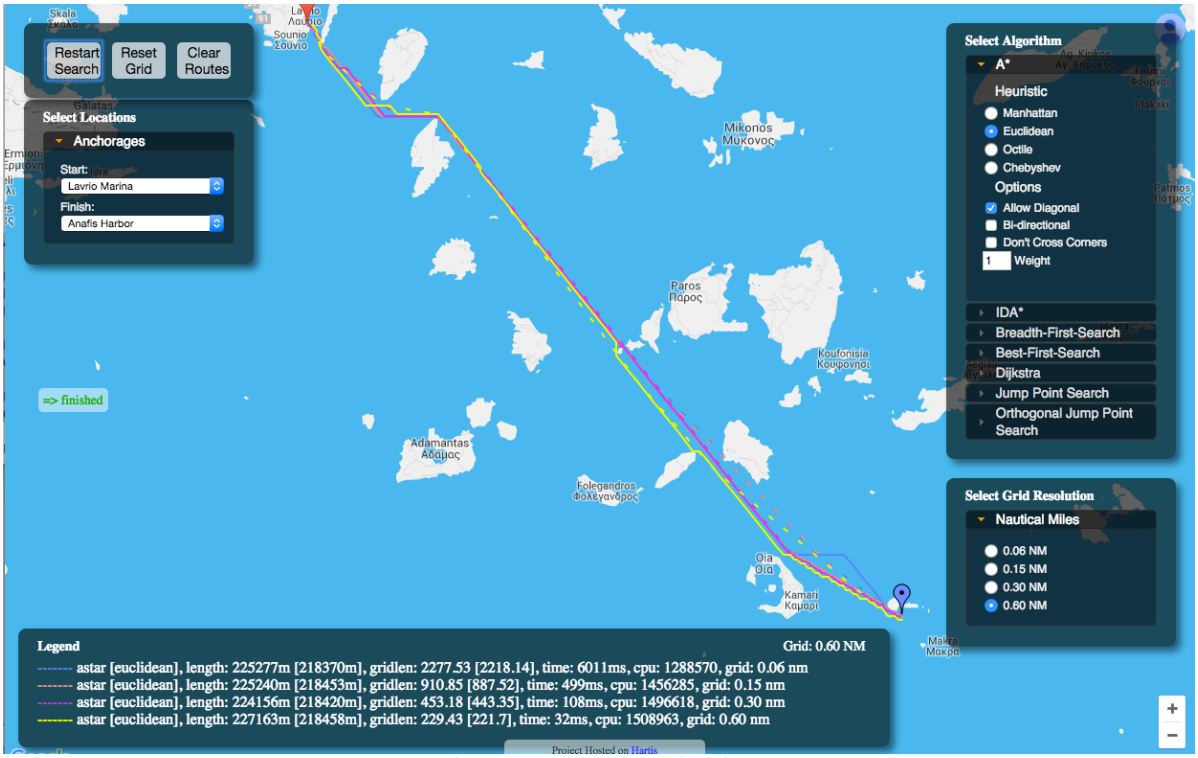
| From | To | Algorithm | Heuristic | Options | Length | Time | Operations |
|---|---|---|---|---|---|---|---|
| Lavrio | Naxos | A* | Manhattan | Allow Diagonal | 167,3 | 43 | 1323 |
| Lavrio | Naxos | A* | Manhattan | Allow Diagonal, Bi-directional | 168,95 | 43 | 2642 |
| Lavrio | Naxos | A* | Manhattan | Allow Diagonal, Bi-directional, Don't Cross Corners | 170,12 | 20 | 2633 |
| Lavrio | Naxos | Dijkstra | | Allow Diagonal | 162,61 | 101 | 48527 |

**Εικόνα 42 - Πίνακας Μετρήσεων Απόστασης, Χρόνου και Υπολογιστικών Κύκλων**

# Ανάπτυξη Πειραματικής Διάταξης

Στη συνέχεια και με βάση την πρωτότυπη διάταξη αναπτύχθηκε πειραματική διάταξη για ανάλυση παραμέτρων των αλγορίθμων. Στην πειραματική διάταξη αναπτύχθηκε:

- δυνατότητα επιλογής Σημείου Εκκίνησης και Σημείου Προορισμού
- δυνατότητα αλλαγής της ανάλυσης του πλέγματος και
- απεικόνιση των αποτελεσμάτων των υπολογισμών σε legend με διατήρηση της ιστορικότητας.




Εικόνα 43 - Πειραματική Διάταξη

Για τα σημεία εκκίνησης και προορισμού επιλέχτηκαν αγκυροβόλια της θαλάσσιας περιοχής των Κυκλάδων και δημιουργήθηκε αρχείο για τη φόρτωση των συντεταγμένων των σημείων στο σύστημα WGS 84.

Για την επιλογή της ανάλυσης του πλέγματος δημιουργήθηκαν 4 διαφορετικά πλέγματα σε ξεχωριστά αρχεία csv. Τα πλέγματα είχαν τα εξής χαρακτηριστικά:

| Ετικέτα | Απόσταση Κόμβων (WGS 84) | Απόσταση (ναυτικά μίλια) | Ελάχιστο Γεωγραφικό Μήκος (WGS 84) | Μέγιστο Γεωγραφικό Μήκος (WGS 84) | Ελάχιστο Γεωγραφικό Πλάτος (WGS 84) | Μέγιστο Γεωγραφικό Πλάτος (WGS 84) |
|---|---|---|---|---|---|---|
| 0,06 NM | 0,001 | 0,06 | 23,9 | 26,1 | 36,2 | 38,1 |
| 0,15 NM | 0,025 | 0,15 | 23,9 | 26,1 | 36,2 | 38,1 |
| 0,30 NM | 0,05 | 0,30 | 23,9 | 26,1 | 36,2 | 38,1 |
| 0,60 NM | 0,01 | 0,60 | 23,9 | 26,1 | 36,2 | 38,1 |

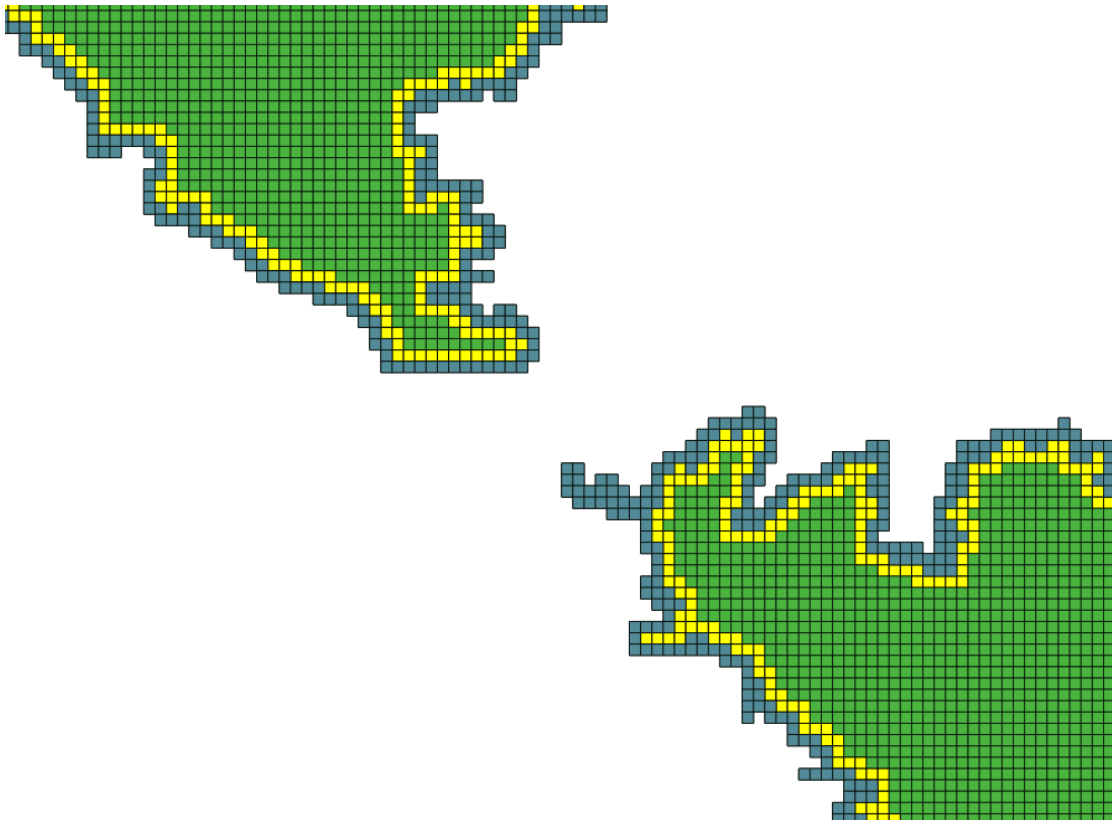
# Ελαχιστοποίηση Σημείων του Πλέγματος

Το βασικό πρόβλημα που προέκυψε κατά τη δημιουργία των νέων αρχείων πλεγμάτων, ήταν το μέγεθος των αρχείων που σχετίζεται με τα σημεία που είναι στη στεριά, ιδιαίτερα για την ανάλυση 0,06 NM. Προκειμένου να μειωθεί ο αριθμός των σημείων, έγινε επιμέρους επιλογή μόνο αυτών που τέμνονται από την ακτογραμμή με εφαρμογή χωρικής ερώτησης:

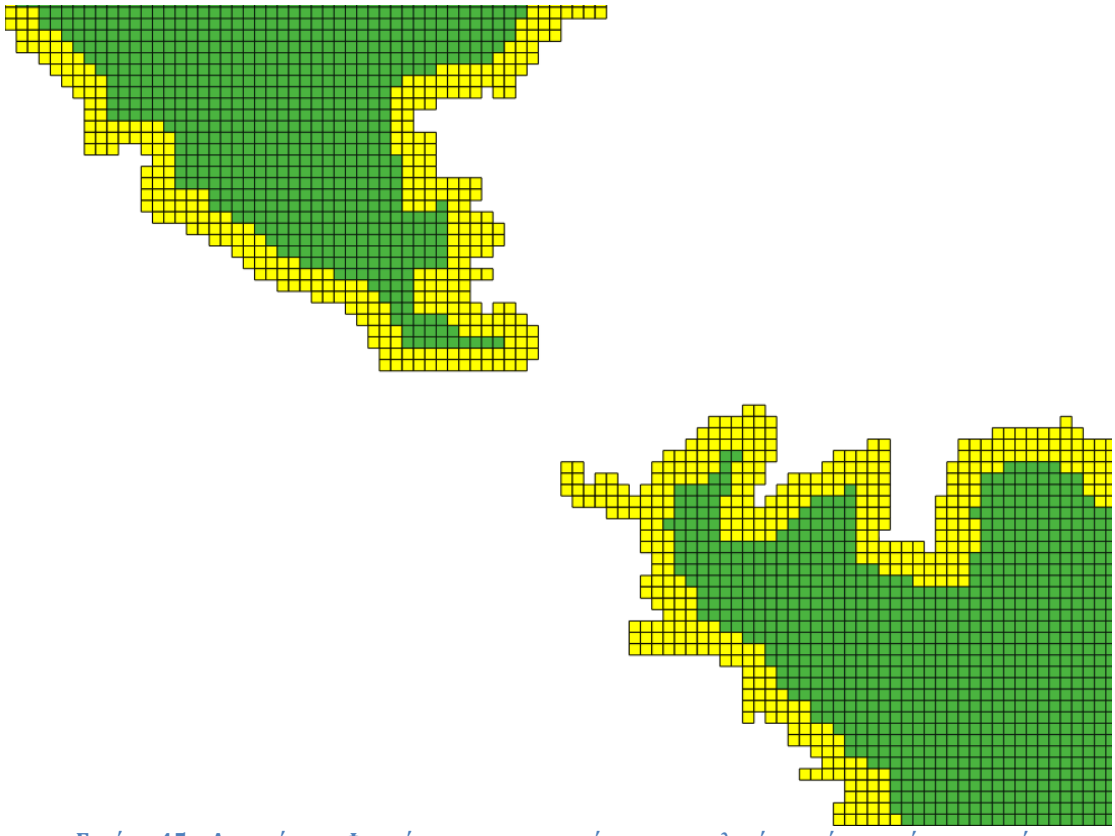
QGIS -> Spatial Query -> Select Grid polygons that intersects with Coastline

Ακολούθως όμως, με την εφαρμογή του νέου αρχείου για την ανάλυση των 0,06 NM, δημιουργήθηκε πρόβλημα με τα Σημεία Εκκίνησης και Προορισμού, που βρίσκονται στο όριο της ακτογραμμής. Ο αλγόριθμος σε πολλές περιπτώσεις τα αντιλαμβανόταν ως στεριά (non walkable), αντί για θάλασσα και ο αλγόριθμος δεν εκτελούνταν.

Για να αντιμετωπιστεί το πρόβλημα αυτό, στα σημεία της ακτογραμμής προστέθηκε και μια εσωτερική (δεύτερη) γραμμή από σημεία. Στην παρακάτω εικόνα απεικονίζεται ο τρόπος αντιμετώπισης μέσω του QGIS.

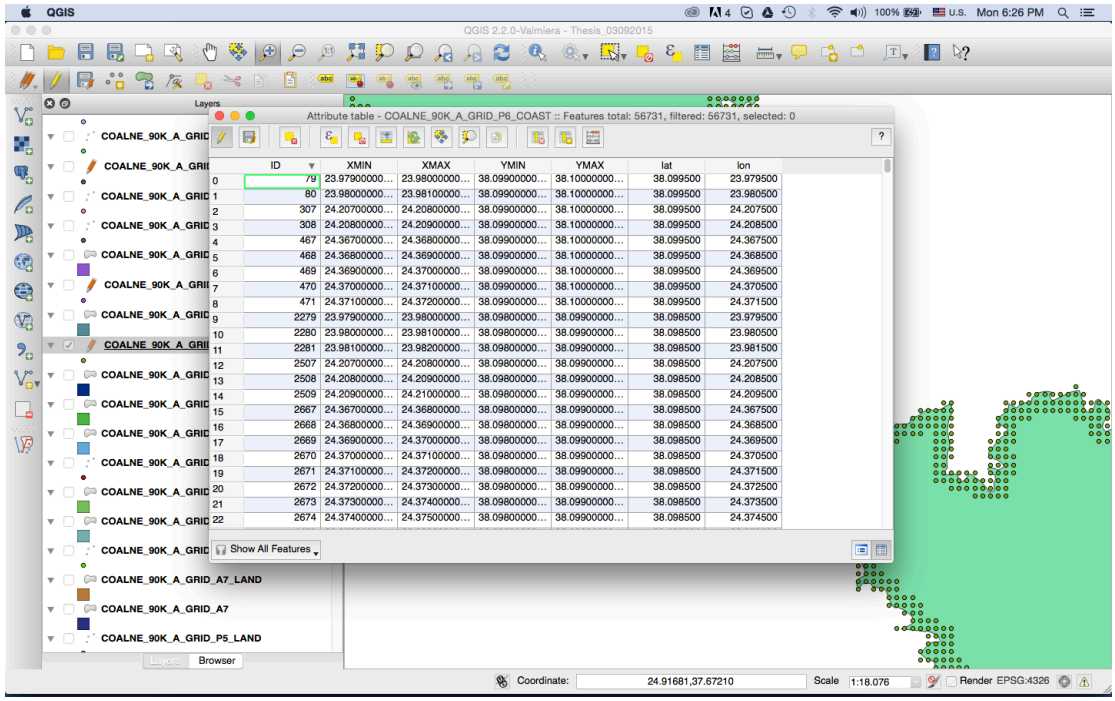


**Εικόνα 44 – Αποτύπωση πολυγώνων Ακτογραμμής και Buffer ενός φατνίου**



Εικόνα 45 – Απεικόνιση Φατνίων που συμμετείχαν στο τελικό αρχείο σημείων στεριάς

Στην παραπάνω εικόνα απεικονίζονται τα σημεία με κίτρινο χρώμα που αποτελούσαν το τελικό αρχείο ανάλυσης, σε σχέση με το πλήθος των πράσινων που αποτελούσαν το αρχικό εξαγόμενο αρχείο.




Εικόνα 46 – Εξαγωγή συντεταγμένων σημείων από επίπεδο του QGIS

# Ανάπτυξη Τελικής Διάταξης

## Δημιουργία Grid για τον Ελληνικό Θαλάσσιο Χώρο

Στη συνέχεια αναπτύχθηκε η τελική διάταξη της εφαρμογής. Το πλέγμα δημιουργήθηκε μεταξύ των ακραίων σημείων με Γεωγραφικό Μήκος 19,0 μοίρες έως 30,5 μοίρες και Γεωγραφικό Πλάτος 34,5 μοίρες έως 42,0 μοίρες, στο προβολικό σύστημα EPSG:4326 (WGS84). Το πλάτος και το ύψος του μοναδιαίου φατνίου του πλέγματος επιλέχθηκε να είναι 0,025 μοίρες, που αντιστοιχεί σε 0,15 του ναυτικού μιλίου και το πλέγμα που δημιουργήθηκε είναι διαστάσεων 2301x1501 φατνίων με 122.756 σημεία μη προσβάσιμα (non walkable) κατά μήκος της ακτογραμμής.

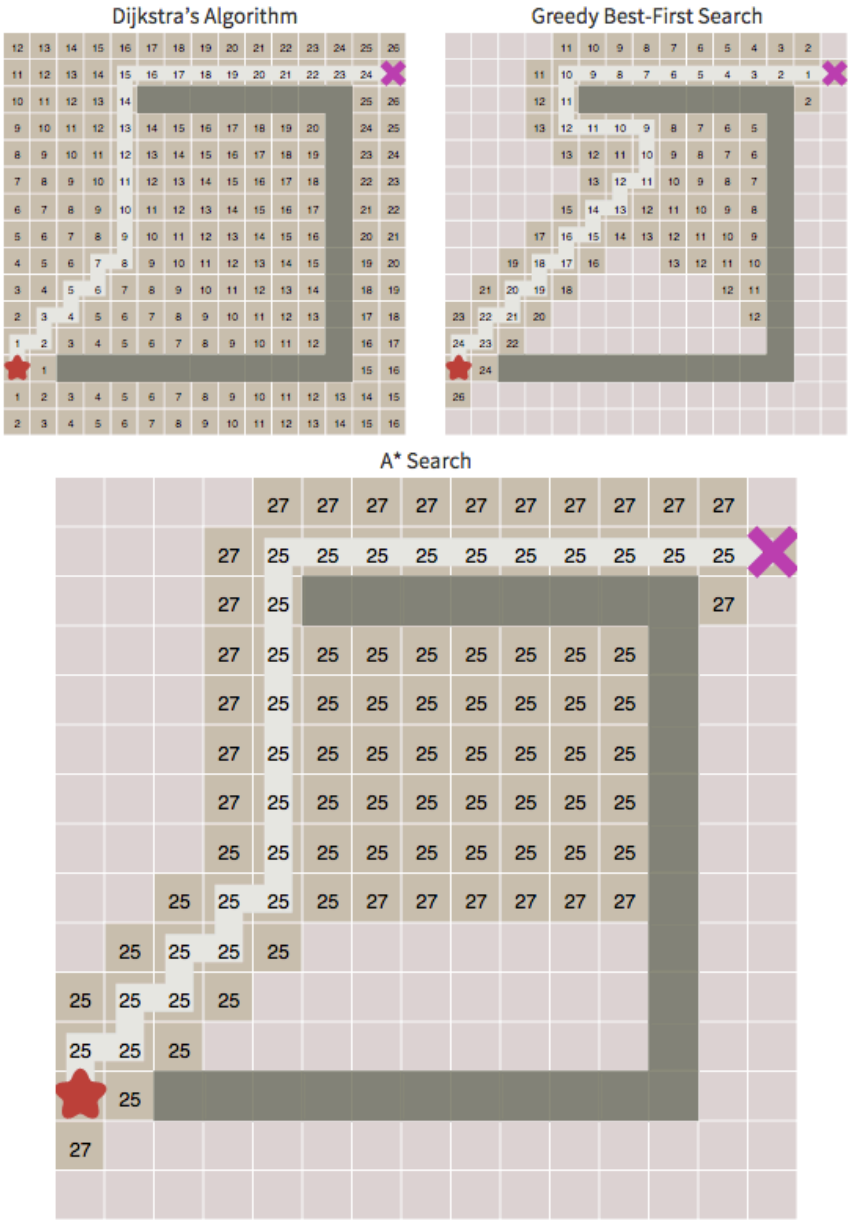
**Εικόνα 47 - Πλήθος σημείων του Πλέγματος της Ελλάδας**

Στην τελική διάταξη προστέθηκε λειτουργικότητα για επιλογή ενδιάμεσων σημείων ενδιαφέροντος. Το πρόβλημα που δημιουργήθηκε στη φάση αυτή ήταν η κατάλληλη επιλογή της σειράς των σημείων, ώστε η συνολική διαδρομή να διέρχεται από όλα τα σημεία και να είναι η ελάχιστη δυνατή. Η λύση του προβλήματος περιγράφεται στο Κεφάλαιο 7 για το σχεδιασμό πλου για πολλαπλά σημεία ενδιαφέροντος.

## Κεφάλαιο 6 – Συγκριτική Ανάλυση Επίλυσης Αλγορίθμων

# Σύγκριση Αλγορίθμων

Στην παρακάτω εικόνα[39] γίνεται μια οπτική σύγκριση της εφαρμογής τριών βασικών αλγορίθμων, του Dijkstra, του Greedy Best First Search και του A*. Ο αλγόριθμος Dijkstra για κάθε κόμβο υπολογίζει το κόστος μετάβασης στον κόμβο από τον αρχικό κόμβο. Ο BFS υπολογίζει για κάθε κόμβο μέσω της ευρετικής συνάρτησης το κόστος μετάβασης από τον κόμβο στον κόμβο προορισμού. Ενώ ο A* υπολογίζει το άθροισμα και των δύο παραμέτρων για τον εκάστοτε κόμβο.

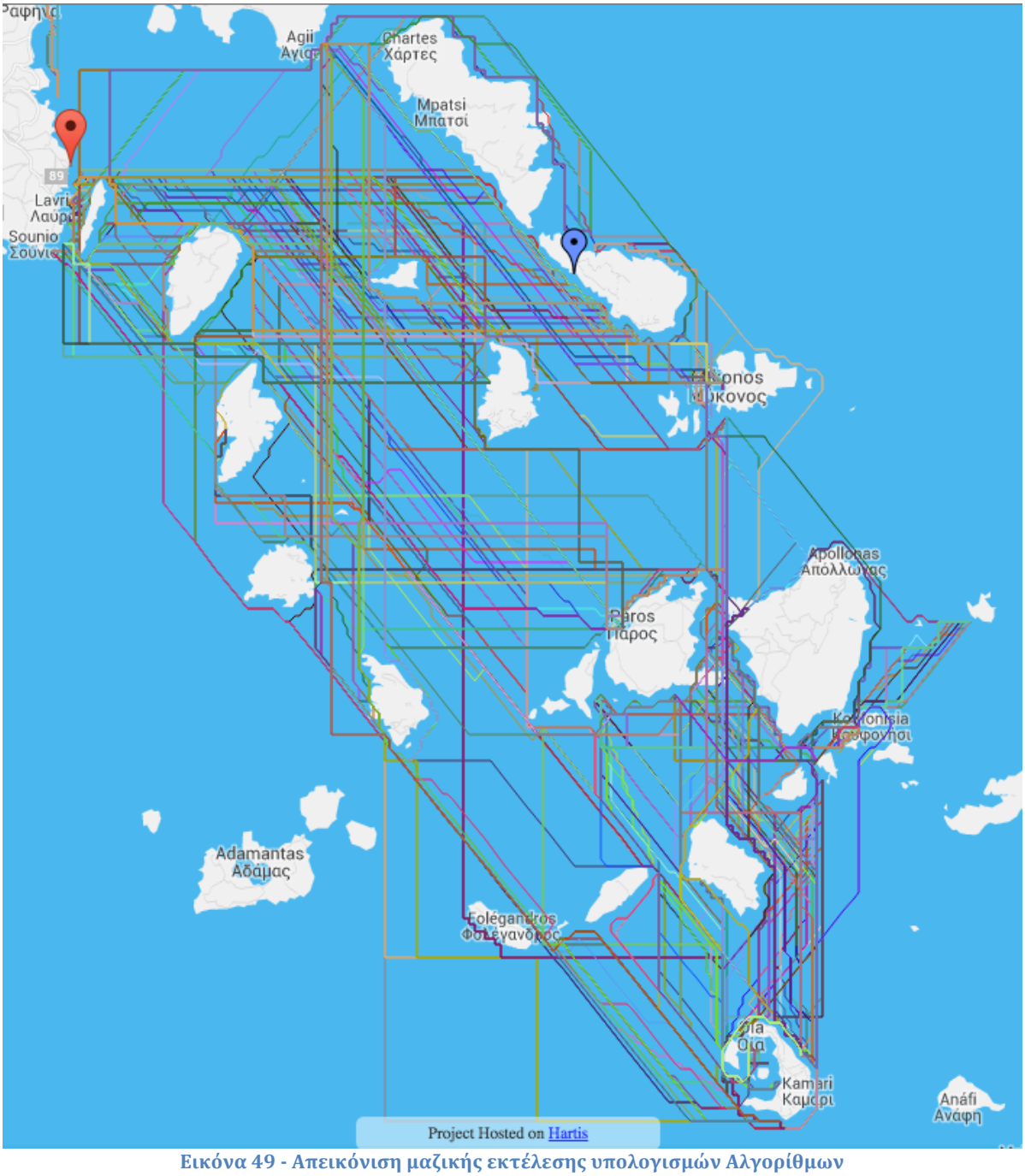



**Εικόνα 48 - Συγκριτική παρουσίαση των αλγορίθμων εύρεσης βέλτιστης διαδρομής**

[39]http://theory.stanford.edu/~amitp/GameProgramming/index.html

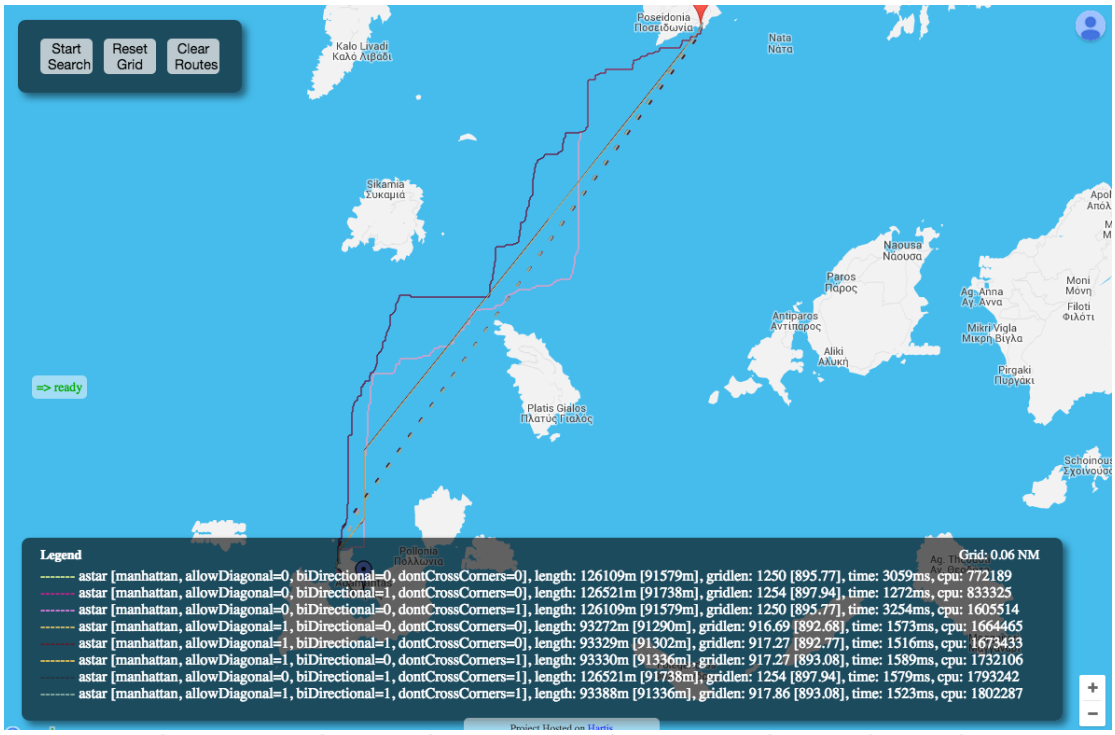
# Ανάπτυξη Διάταξη Συγκριτικής Ανάλυσης

Σύμφωνα με τα προηγούμενα, ο αλγόριθμος Dijkstra υπολογίζει πάντα τη συντομότερη διαδρομή, ο BFS όχι πάντα και ο A* σχεδόν πάντα. Στο πλαίσιο της εργασίας δημιουργήθηκε πειραματική συγκριτική διάταξη για τη μαζική εκτέλεση των αλγορίθμων προκειμένου να επιβεβαιωθεί και πειραματικά το προηγούμενο συμπέρασμα και να μελετηθούν τυχόν διαφορές.




**Εικόνα 49 - Απεικόνιση μαζικής εκτέλεσης υπολογισμών Αλγορίθμων**

Στην παραπάνω εικόνα φαίνεται η μαζική εκτέλεση υπολογισμών εύρεσης της συντομότερης διαδρομής μεταξύ πολλαπλών ζευγών σημείων εκκίνησης και προορισμού. Ενώ στην παρακάτω εικόνα φαίνεται η εκτέλεση πολλαπλών υπολογισμών μεταξύ ενός ζεύγους σημείων.




Εικόνα 50 - Απεικόνιση εκτέλεσης πολλαπλών υπολογισμών μεταξύ 2 σημείων

# Μετρήσεις – Πειραματικά Αποτελέσματα

## Πλήθος Μετρήσεων

Στον επόμενο πίνακα παρατίθεται το πλήθος των μετρήσεων – εκτελέσεις των αλγορίθμων - που έγιναν μαζικά με τη συγκριτική διάταξη. Οριζόντια αποτυπώνονται οι αλγόριθμοι και οι ευρετικές μέθοδοι για κάθε αλγόριθμο και κάθετα οι μετρήσεις ανά ανάλυση του πλέγματος.

| Count of measurements | Resolution | | | | |
|---|---|---|---|---|---|
| Algorithm/Heuristic | 0.01 | 0.005 | 0.0025 | 0.001 | Grand Total |
| astar | 616 | 668 | 656 | 638 | 2578 |
| chebyshev | 154 | 167 | 164 | 63 | 548 |
| euclidean | 154 | 167 | 164 | 132 | 617 |
| manhattan | 154 | 167 | 164 | 368 | 853 |
| octile | 154 | 167 | 164 | 75 | 560 |
| bestfirst | 608 | 652 | 601 | 150 | 2011 |
| chebyshev | 152 | 162 | 147 | 36 | 497 |
| euclidean | 152 | 163 | 152 | 34 | 501 |
| manhattan | 152 | 164 | 152 | 48 | 516 |
| octile | 152 | 163 | 150 | 32 | 497 |
| breadthfirst | 614 | 660 | 652 | 91 | 2017 |
| dijkstra | 608 | 644 | 573 | 115 | 1940 |
| jump_point | 608 | 620 | 521 | 220 | 1969 |
| chebyshev | 152 | 152 | 128 | 52 | 484 |
| euclidean | 152 | 156 | 131 | 55 | 494 |

| | | | | | |
|---|---|---|---|---|---|
| manhattan | 152 | 158 | 132 | 59 | 501 |
| octile | 152 | 154 | 130 | 54 | 490 |
| orth_jump_point | 608 | 603 | 496 | 209 | 1916 |
| chebyshev | 152 | 150 | 124 | 52 | 478 |
| euclidean | 152 | 151 | 124 | 52 | 479 |
| manhattan | 152 | 152 | 124 | 53 | 481 |
| octile | 152 | 150 | 124 | 52 | 478 |
| **Grand Total** | **3662** | **3847** | **3499** | **1423** | **12431** |

Εικόνα 51 - Πίνακας Πλήθους Μετρήσεων Συγκριτικής Διάταξης

## Πειραματικά Αποτελέσματα

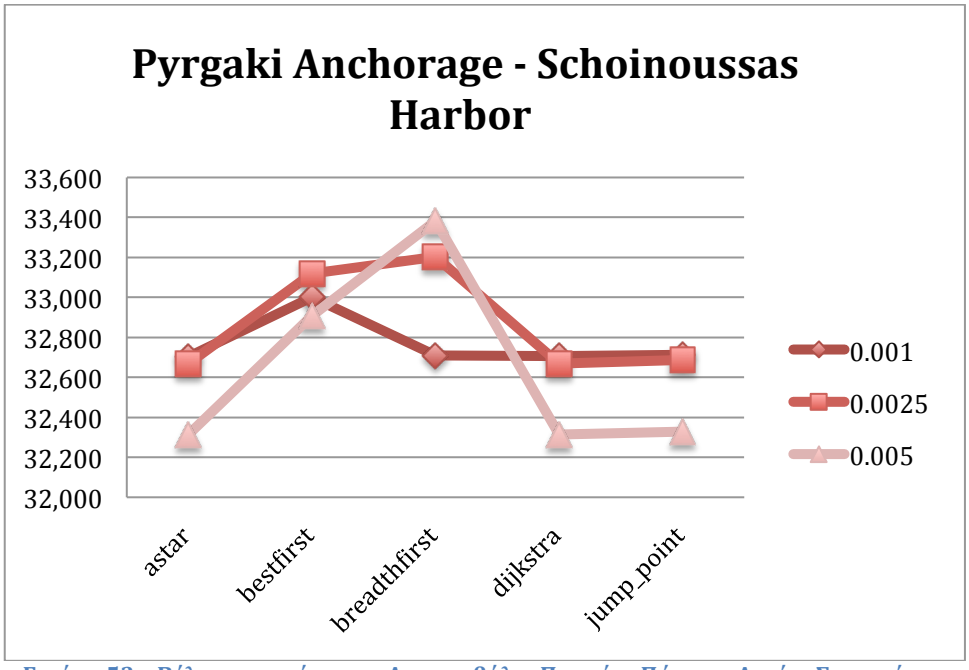
Στον επόμενο πίνακα παρατίθεται μέρος των μετρήσεων – εκτελέσεις των αλγορίθμων - που έγιναν με τη συγκριτική διάταξη ανάμεσα σε 3 ζεύγη σημείων ενδιαφέροντος.

| **Min of length (m)** | Resolution | | | |
|---|---|---|---|---|
| Start/End Marina | 0.001 | 0.0025 | 0.005 | 0.01 |
| Pyrgaki Anchorage | 32.702 | 32.669 | 32.314 | 29.374 |
| Schoinoussas Harbor | 32.702 | 32.669 | 32.314 | 29.374 |
| astar | 32.702 | 32.669 | 32.314 | 29.376 |
| bestfirst | 33.002 | 33.120 | 32.908 | 29.389 |
| breadthfirst | 32.710 | 33.202 | 33.382 | 29.374 |
| dijkstra | 32.706 | 32.669 | 32.314 | 29.375 |
| jump_point | 32.713 | 32.687 | 32.330 | 29.389 |
| orth_jump_point | 40.098 | 40.349 | 39.848 | 35.166 |
| Schoinoussas Harbor | 57.563 | 56.974 | 56.576 | 65.668 |
| Skala Thira Harbor | 57.563 | 56.974 | 56.576 | 65.668 |
| astar | 57.571 | 56.977 | 56.584 | 65.676 |
| bestfirst | 57.697 | 57.139 | 56.584 | 66.302 |
| breadthfirst | 57.583 | 56.994 | 56.583 | 65.674 |
| dijkstra | 57.568 | 56.984 | 56.594 | 65.673 |
| jump_point | 57.563 | 56.974 | 56.576 | 65.668 |
| orth_jump_point | 68.353 | 67.442 | 67.332 | 79.616 |
| Skala Thira Harbor | 74.159 | 73.813 | 73.513 | 81.835 |
| Soros Anchorage | 74.159 | 73.813 | 73.513 | 81.835 |
| astar | 74.162 | 73.830 | 73.517 | 81.835 |
| bestfirst | 79.103 | 78.088 | 76.966 | 83.570 |
| breadthfirst | 74.159 | 73.813 | 73.513 | 81.835 |
| dijkstra | 74.162 | 73.825 | 73.518 | 81.835 |
| jump_point | 74.190 | 73.844 | 73.542 | 81.868 |
| orth_jump_point | 94.392 | 93.321 | 92.880 | 98.024 |
| **Grand Total** | **32.702** | **32.669** | **32.314** | **29.374** |

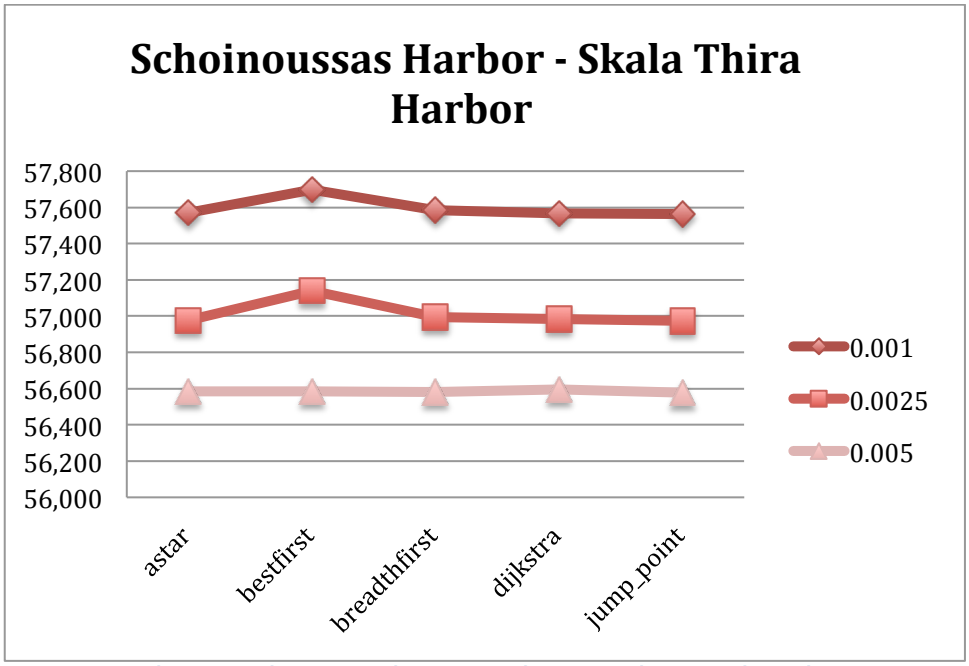
Εικόνα 52 - Πίνακας Μετρήσεων Ελάχιστης Απόστασης

## Διαγράμματα

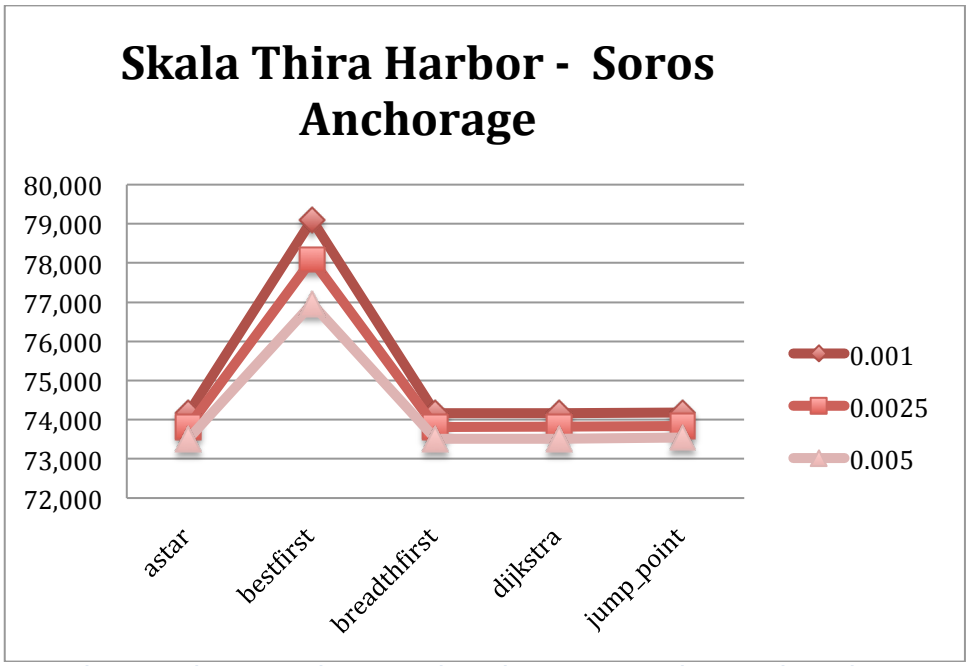
Με βάση τις παραπάνω μετρήσεις στα επόμενα διαγράμματα καταδεικνύεται πως ο αλγόριθμός A* υπολογίζει σχεδόν πάντα τη συντομότερη διαδρομή, όπως ο Dijkstra και πολλές φορές ο Breadth-First. Σε πολλές περιπτώσεις και ο Jump Point έχει επίσης βέλτιστη συμπεριφορά.




Εικόνα 53 – Βέλτιστη απόσταση Αγκυροβόλιο Πυργάκι Πάρου – Λιμάνι Σχοινούσας




Εικόνα 54 – Βέλτιστη απόσταση Λιμάνι Σχοινούσας - Σκάλα Θήρας




**Εικόνα 55 - Βέλτιστη Απόσταση Σκάλα Θήρας – Αγκυροβόλιο Σωρός Αντίπαρου**

## Disclaimer

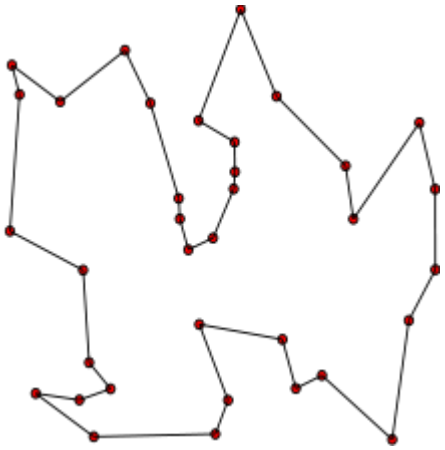
Είναι σημαντικό να σημειωθεί ότι τα πειραματικά αποτελέσματα, όπως τα runtimes των αλγορίθμων αναζήτησης, εξαρτώνται από μια ποικιλία παραγόντων, συμπεριλαμβανομένων των στοιχείων εφαρμογής (όπως οι δομές δεδομένων, tie-breaking στρατηγικές, και coding tricks που χρησιμοποιούνται) καθώς και η διαμόρφωση των πειραματικών διατάξεων, όπως το αν στο πλέγμα επιτρέπεται η διαγώνια κίνηση (που σημαίνει αν θα είναι τέσσερις γείτονες σε κάθε κόμβο ή οκτώ). Η καλύτερη μέθοδος για την αξιολόγηση των αλγορίθμων αναζήτησης, είναι η όσο το δυνατόν καλύτερη εφαρμογή τους και η δημοσίευση των χρόνων εκτέλεσής τους.

## Κεφάλαιο 7 – Σχεδιασμός Πλου για πολλαπλά Σημεία Ενδιαφέροντος

# Περιγραφή Ανάγκης

Η προαναφερθείσα εφαρμογή των αλγορίθμων συντομότερης διαδρομής αφορά τον υπολογισμό της βέλτιστης διαδρομής ανάμεσα σε δύο σημεία ενδιαφέροντος (αγκυροβόλια και παραλίες στην προκειμένη περίπτωση). Για τον σχεδιασμό όμως ενός ολοκληρωμένου πλου, η σχεδιαζόμενη διαδρομή θα χρειαστεί να περάσει από όλα τα επιθυμητά σημεία ενδιαφέροντος που τις περισσότερες των περιπτώσεων είναι πάνω από δύο και μπορεί να είναι και δέκα και δεκαπέντε, ανάλογα με τον χρόνο που μπορεί να αφιερωθεί για τον συνολικό πλου σε συνδυασμό με την ικανοποίηση της απαίτησης για επίσκεψη όσο το δυνατό περισσότερων μερών ενδιαφέροντος.

Τίθεται τώρα η ανάγκη υπολογισμού της συντομότερης διαδρομής η οποία να περνά από όλα τα σημεία ενδιαφέροντος. Στην αντιμετώπιση αυτής της ανάγκης βοήθησε το Πρόβλημα του Περιοδεύοντος Πωλητή.



**Εικόνα 56 - Λύση Προβλήματος Περιοδεύοντος Πωλητή[40]**

# Πρόβλημα Περιοδεύοντος Πωλητή

Το πρόβλημα του Περιοδεύοντος Πωλητή (στην βιβλιογραφία αναφέρεται ως TSP - Travelling Salesman Problem) είναι ένα κλασικό αλγοριθμικό πρόβλημα στην επιστήμη των υπολογιστών. Εστιάζεται στη βελτιστοποίηση και στο πλαίσιο αυτό καλύτερη λύση συνήθως σημαίνει μια λύση που είναι φθηνότερη. Είναι ένα μαθηματικό πρόβλημα και πιο συχνά εκφράζεται με ένα γράφημα που περιγράφει τις θέσεις ενός συνόλου κόμβων.

Το πρόβλημα του περιοδεύοντος (ή πλανόδιου, ή περιπλανώμενου στα Ελληνικά) πωλητή ορίστηκε το 19ο αιώνα από τον Ιρλανδό μαθηματικό W. R. Hamilton και από τον Βρετανό μαθηματικό Thomas Kirkman. Η γενική μορφή του TSP φαίνεται να έχει μελετηθεί από τους μαθηματικούς κατά τη διάρκεια της δεκαετίας του 1930 στη Βιέννη και στο Harvard, κυρίως από τον Karl Menger. Ο Menger καθορίζει το

[40] https://en.wikipedia.org/wiki/Travelling_salesman_problem

πρόβλημα, θεωρεί τον προφανή αλγόριθμο brute-force και παρατηρεί τη μη-βέλτιστη από την ευρετική εφαρμογή με τον αλγόριθμο του πλησιέστερου γείτονα.

Το πρόβλημα του περιοδεύοντος πωλητή περιγράφει την ανάγκη ενός πωλητή ο οποίος πρέπει να ταξιδέψει μεταξύ Ν πόλεων. Η σειρά με την οποία θα το κάνει δεν τον ενδιαφέρει εφ 'όσον επισκέπτεται την κάθε μία από μία φορά κατά τη διάρκεια του ταξιδιού του και επιστρέφει τελικά εκεί από όπου ξεκίνησε. Κάθε πόλη είναι συνδεδεμένη με άλλες κοντινές πόλεις, είτε με αεροπορική σύνδεση, είτε οδικώς είτε σιδηροδρομικώς. Κάθε μία από τις συνδέσεις μεταξύ των πόλεων έχει ένα ή περισσότερα βάρη (ή κόστος) το οποίο είναι γνωστό. Το κόστος περιγράφει πόσο «δύσκολο» είναι να διασχίσει την αντίστοιχη ακμή στο γράφημα, και μπορεί να δοθεί, για παράδειγμα, με το κόστος του εισιτηρίου του αεροπλάνου ή του τρένου, ή από το μήκος της ακμής, ή τον χρόνο που απαιτείται για την ολοκλήρωση του ταξιδιού. Ο πωλητής θέλει να κρατήσει τόσο τα έξοδα ταξιδίου, καθώς και την απόσταση που θα ταξιδέψει όσο το δυνατόν χαμηλότερα.

Το πρόβλημα του περιοδεύοντος πωλητή είναι χαρακτηριστικό μιας μεγάλης κατηγορίας προβλημάτων βελτιστοποίησης που έχουν κινήσει την περιέργεια μαθηματικών και επιστημόνων πληροφορικής για πολλά χρόνια. Το πιο σημαντικό είναι πως έχει εφαρμογές στον τομέα της επιστήμης και της μηχανικής. Για παράδειγμα, στην παρασκευή ενός κυκλώματος, είναι σημαντικό να καθοριστεί η καλύτερη σειρά με την οποία ένα λέιζερ θα τρυπά χιλιάδες τρύπες, καθώς μια αποτελεσματική λύση στο πρόβλημα αυτό μειώνει το κόστος παραγωγής για τον κατασκευαστή.

# 2-Opt & 3-Opt Αλγόριθμοι Επίλυσης του TSP

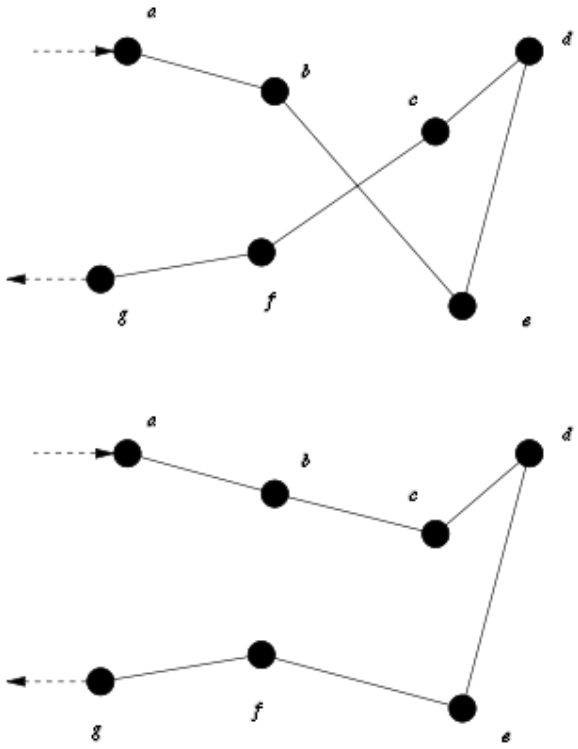
Ο 2-opt είναι ένας απλός αλγόριθμος τοπικής αναζήτησης που προτάθηκε για πρώτη φορά από τον Crows το 1958 για την επίλυση του προβλήματος του πλανόδιου πωλητή. Η κύρια ιδέα πίσω από αυτό είναι να λάβει μια διαδρομή που διασχίζει πάνω από τον εαυτό της και να την αναδιατάξει έτσι ώστε να μην το κάνει.

Ο αλγόριθμος 2-opt καταργεί βασικά δύο ακμές από την διαδρομή, και επανασυνδέει τις δύο διαδρομές που δημιουργούνται. Αυτό αναφέρεται συχνά ως μια κίνηση 2-opt[41]. Υπάρχει μόνον ένας τρόπος για να επανασυνδεθούν οι δύο διαδρομές έτσι ώστε να δημιουργηθεί ακόμα μια έγκυρη διαδρομή και αυτό γίνεται μόνον αν η νέα διαδρομή θα είναι μικρότερη. Συνεχίζεται η αφαίρεση και επανασύνδεση της διαδρομής έως ότου να βρεθούν βελτιώσεις στο συνολικό κόστος. Αν δεν υπάρχουν περαιτέρω βελτιώσεις η διαδρομή θεωρείται η βέλτιστη.

Ο αλγόριθμος 3-opt λειτουργεί με παρόμοιο τρόπο, αλλά αντί για την άρση των δύο ακμών αφαιρούνται τρεις. Αυτό σημαίνει ότι υπάρχουν δύο τρόποι επανασύνδεσης των τριών διαδρομών σε μια έγκυρη διαδρομή. Μια κίνηση 3-opt μπορεί να θεωρηθεί

[41]https://en.wikipedia.org/wiki/2-opt

ως δύο ή τρεις 2-opt κινήσεις. Τελειώνει η αναζήτηση της τοποθεσίας όταν υπάρχουν περαιτέρω 3-opt κινήσεις οι οποίες να μπορούν να βελτιώσουν τη διαδρομή. Αν μια περιήγηση είναι 3-opt βέλτιστη είναι επίσης 2-opt βέλτιστη.




**Εικόνα57–Κίνηση Αλγόριθμου 2-Opt**

Μία πλήρης 2-opt τοπική αναζήτηση θα συγκρίνει κάθε δυνατό έγκυρο συνδυασμό του μηχανισμού εναλλαγής. Αυτή η τεχνική μπορεί να εφαρμοστεί στο πρόβλημα του περιοδεύοντος πωλητή καθώς και πολλά συναφή προβλήματα.

Αυτός είναι ο μηχανισμός με τον οποίο η 2-opt εναλλαγή χειρίζεται μια δεδομένη διαδρομή:

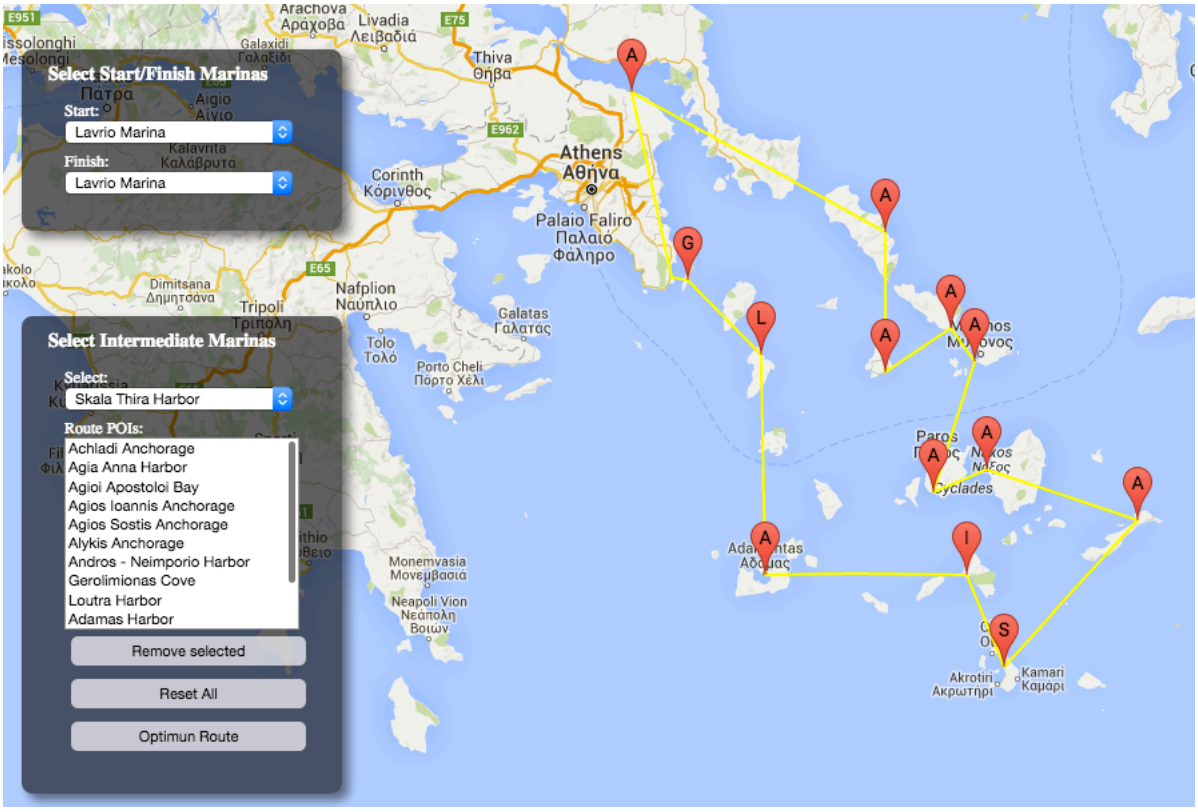
```
2optSwap(route, i, k) {
       1. take route[1] to route[i-1] and add them in order to
new_route
       2. take route[i] to route[k] and add them in reverse order to
new_route
       3. take route[k+1] to end and add them in order to new_route
       return new_route;
   }
```

Στη συνέχεια δίνεται ο πλήρης 2-opt αλγόριθμος εναλλαγής ενδιάμεσων σημείων διαδρομής του παραπάνω μηχανισμού:

```
repeat until no improvement is made {
       start_again:
       best_distance = calculateTotalDistance(existing_route)
       for (i = 0; i < number of nodes eligible to be swapped - 1;
i++) {
            for (k = i + 1; k < number of nodes eligible to be swapped;
k++) {
                new_route = 2optSwap(existing_route, i, k)
                new_distance = calculateTotalDistance(new_route)
                if (new_distance < best_distance) {
                    existing_route = new_route
                    goto start_again
                }
            }
       }
   }
```

# Δοκιμαστική Διάταξη Εφαρμογής 2-Opt Αλγόριθμου

Προκειμένου να ενταχθεί ο αλγόριθμος στο υπό ανάπτυξη σύστημα, δημιουργήθηκε δοκιμαστική διάταξη στην οποία εκτελείται ο αλγόριθμος 2-Opt χωρίς να υπολογίζονται οι θαλάσσιες βέλτιστες αποστάσεις.




Εικόνα 58 - Απλή εφαρμογή 2-opt αλγορίθμου

# Εφαρμογή στο Σύστημα Βέλτιστης Διαδρομής

Στη συνέχεια εντάχθηκε ο 2-opt αλγόριθμος στο υπό ανάπτυξη σύστημα όπου υπολογίζονται οι θαλάσσιες διαδρομές μέσω του αλγόριθμου A*. Ο κώδικας που αναπτύχθηκε έχει ως εξής:

```
setRoute: function(x0, y0, x1, y1, resolution) {
  var grid1;
var finder = Panel.getFinder();
  var baseMarinas = Panel.getMarinas();
var routeMarinas = Panel.getRouteMarinas();
var startMarina = '{"title": "' + baseMarinas.start.title + '", "lat": ' +
baseMarinas.start.lat + ', "lon":' + baseMarinas.start.lon + '}'; //Lavrio
  var finishMarina = '{"title": "' + baseMarinas.finish.title + '", "lat": ' +
baseMarinas.finish.lat + ', "lon": ' + baseMarinas.finish.lon + '}'; //Naxos
var routeArray = [];
  for (i = 0; i < routeMarinas.route.titles.length; i++) {
    coordinates = routeMarinas.route.coordinates[i].split(",");
    lat = coordinates[1];
    lon = coordinates[0];
    title = routeMarinas.route.titles[i];
    routeArray.push('{"title": "' + title + '", "lat": ' + lat + ', "lon": ' +
lon + '}');
  }
  routeArray.unshift(startMarina);
  routeArray.push(finishMarina);
  routeJSON = JSON.parse('[' + routeArray + ']');
  var routePoints = routeArray.length;
  for (point = 0; point < routePoints; point++) { //loop all start points
    var marinaXYJson = this.calculateXY(routeJSON[point].lat,
routeJSON[point].lon, x0, y0, x1, y1, resolution)
routeJSON[point].X = marinaXYJson.X;
    routeJSON[point].Y = marinaXYJson.Y;
  }
  //calculate all distances
  var distances = [];
  var distanceJSON = [];
  var paths = [];
  for (j = 0; j < routePoints; j++) { //loop all start points
    distances[j] = [];
    paths[j] = [];
    for (k = 0; k < routePoints; k++) { //loop all end points
```

```
      if (j !== k) {
        grid1 = grid.clone();
        this.path = finder.findPath(
          routeJSON[j].X, routeJSON[j].Y, routeJSON[k].X, routeJSON[k].Y,
grid1
        );
        this.smoothenPath = PF.Util.smoothenPath(grid1, this.path);
        var distance = '{"e":' + k + ',"d" : ' + this.path.length + '}';
        var tempPathString = JSON.stringify(this.smoothenPath);
        var tempPath = JSON.parse(tempPathString);
        paths[j][k] = tempPath;
      }
      else {
        var distance = '{"e":' + k + ',"d" : ' + 0 + '}';
        paths[j][k] = {};
      }
      distances[j][k] = distance;
    }
    distanceJSON[j] = '{"point":' + j + ',"edges":[' + distances[j] + ']}';
  }
  points = JSON.parse('[' + distanceJSON + ']');
  //sort distances for pointA
  var pointAString = JSON.stringify(points[0]);
  var pointA = JSON.parse(pointAString);
  pointA.edges.sort(function(a, b) {
    return parseFloat(a.d) - parseFloat(b.d);
  });
  var totalDistance = 0;
  for (k = 0; k < routePoints; k++) {
    totalDistance = totalDistance + pointA.edges[k].d;
  }
  initialRoute = [];
  for (k = 0; k < routePoints; k++) {
    initialRoute[k] = pointA.edges[k].e;
  }

  //check if start-end is the same
  endIsStart = false;
  if (pointA.edges[1].d == 0 || pointA.edges[1].d == 1 || pointA.edges[1].d ==
2) {
    endIsStart = true;
    initialRoute.move(1, routePoints - 1);
  }
```

```
  initialDistance = this.calculateTotalDistance(initialRoute, routePoints);
  var bestRouteDistance = initialDistance;
var existingRouteString = JSON.stringify(initialRoute);
  var existingRoute = JSON.parse(existingRouteString);
  var bestRouteString = JSON.stringify(initialRoute);
  var bestRoute = JSON.parse(bestRouteString);

  //perform 2-opt optimisation algorithm
  var count = 0;
  do {
    var bestRouteDistanceLast = bestRouteDistance;
    count++;
    if (count > 1) {
      var existing = JSON.stringify(bestRoute);
      var existingRoute = JSON.parse(existing);
    }
    for (i = 1; i < routePoints - 2; i++) {
      for (k = i + 1; k < routePoints - 1; k++) {

        distance1 = points[existingRoute[i]].edges[existingRoute[i - 1]].d +
points[existingRoute[k + 1]].edges[existingRoute[k]].d;
        distance2 = points[existingRoute[i]].edges[existingRoute[k + 1]].d +
points[existingRoute[i - 1]].edges[existingRoute[k]].d;
        if (distance1 > distance2) {
          newRoute = this.twoOptSwap(existingRoute, i, k)
          newRouteDistance = this.calculateTotalDistance(newRoute,
routePoints)
          if (newRouteDistance < bestRouteDistance) {
            var bestRouteString = JSON.stringify(newRoute);
            var bestRoute = JSON.parse(bestRouteString);
            bestRouteDistance = newRouteDistance;
          }
        }
      }
    }
  }
  while (bestRouteDistanceLast > bestRouteDistance)
  //consolidate individual paths
  var bestPath = [];
  var currentPath = [];
  for (i = 0; i < routePoints - 1; i++) {
var currentPath = bestPath;
    var newPath = paths[bestRoute[i]][bestRoute[i + 1]];
```

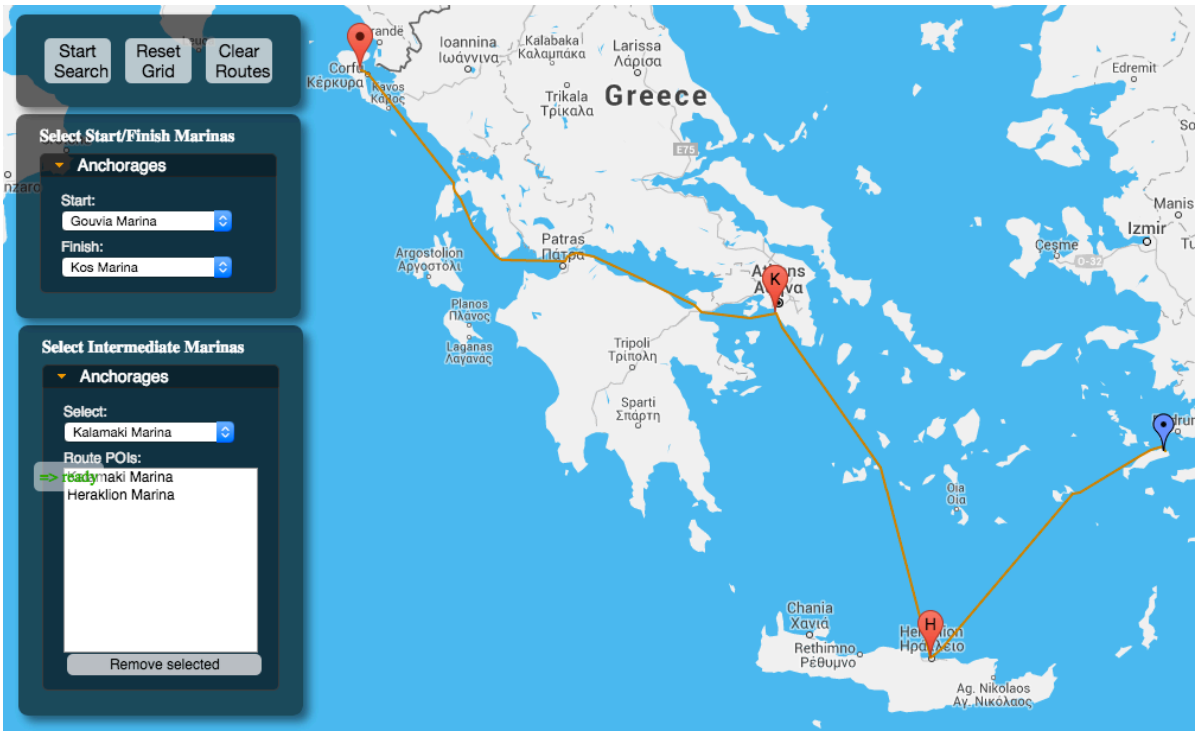
```
    bestPath = currentPath.concat(newPath);
  }
View.drawRoute(bestPath, x0, y0, x1, y1, resolution);
}

calculateTotalDistance: function(existingRoute, totalPoints) {
  var existingRouteDistance = 0;
  for (x = 0; x < totalPoints; x++) {
    currentPoint = existingRoute[x];
    if (x == totalPoints - 1) {
      nextPoint = 0;
    }
    else {
      nextPoint = existingRoute[x + 1];
    }
existingRouteDistance = existingRouteDistance +
points[currentPoint].edges[nextPoint].d;
  }
return existingRouteDistance;
}

twoOptSwap: function(route, i, k) {
  //1. take route[1] to route[i-1] and add them in order to new_route
  //2. take route[i] to route[k] and add them in reverse order to new_route
  //3. take route[k+1] to end and add them in order to new_route
var b = route[k];
  route[k] = route[i];
  route[i] = b;
return route;
}
```

# Τελική Διάταξη Συστήματος Σχεδιασμού Πλου

Στη ακόλουθη εικόνα φαίνεται η τελική διάταξη σχεδιασμού πλου με πλέγμα για όλο τον ελλαδικό θαλάσσιο χώρο και ανάλυση 0.3νμ.

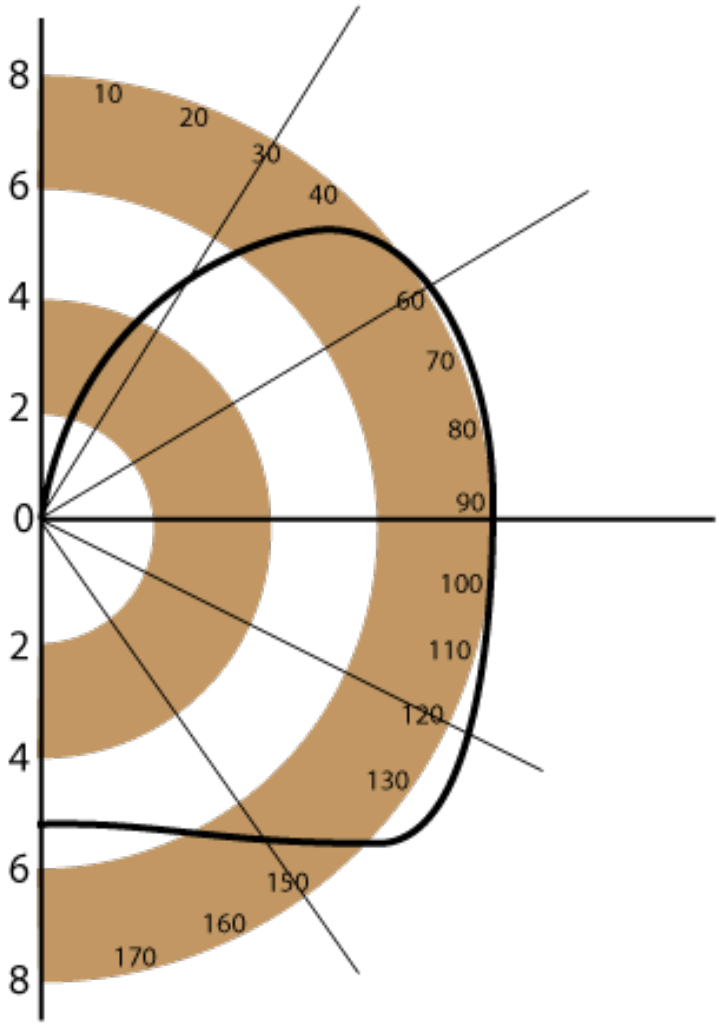



**Εικόνα 59 - Τελική Διάταξη Συστήματος Σχεδιασμού Πλου για την Ελλάδα**

## Κεφάλαιο 8 - Επίπτωση Καιρικών Συνθηκών στον Ιστιοπλοϊκό Πλου

# Πολικά Διαγράμματα

Για τη μοντελοποίηση του καιρού και τον τρόπο που θα επηρεάζει τον σχεδιασμό του πλου, έγινε χρήση των πολικών διαγραμμάτων που έχει κάθε ιστιοπλοϊκό σκάφος, όπου αποτυπώνεται η επίδραση στη ταχύτητα που μπορεί να αναπτύξει το σκάφος σε σχέση με την κατεύθυνση και το μέγεθος της ταχύτητας του ανέμου.




Εικόνα 60 - Πολικό Διάγραμμα Ιστιοπλοϊκού Σκάφους

Τα Πολικά Διαγράμματα είναι ένας βολικός και συνάμα αποτελεσματικός τρόπος να παρουσιαστούν δεδομένα πρόβλεψης ταχύτητας και είναι εξαιρετικά χρήσιμα για την κατανόηση της σχέσης μεταξύ των τριών πιο σημαντικών παραγόντων στην απόδοση ενός σκάφους: της ταχύτητας του ανέμου, της γωνίας του ανέμου και της ταχύτητας του σκάφους. Τα Πολικά Διαγράμματα παράγονται από ένα ειδικό πρόγραμμα που ονομάζεται VPP (Velocity Prediction Program).[42]

[42]http://www.offshore.org.gr/docs/ORC_Speed_Guide_Explanation_EL.pdf

Το VPP είναι πρόγραμμα πρόβλεψης ταχύτητας, το οποίο υπολογίζει την απόδοση ενός ιστιοπλοϊκού σκάφους σε διάφορες συνθήκες ανέμου εξισορροπώντας τις δυνάμεις της γάστρας (hull) και των πανιών (sails)[43]. Τα VPPs χρησιμοποιούνται από σχεδιαστές γιοτ, κατασκευαστές σκαφών, δοκιμαστές μοντέλων, ναυτικούς, ιστιοράπτες, ιστιοπλοϊκές ομάδες για να προβλέψουν την απόδοση που θα έχει ένα ιστιοπλοϊκό πριν από αυτό κατασκευαστεί ή πριν από σημαντικές τροποποιήσεις.

Οι προβλέψεις ταχύτητας κάθε ιστιοπλοϊκού σκάφους εξάγονται σε δύο στάδια. Αρχικά, καταμετράται η γάστρα και τα προσαρτήματά της με ηλεκτρονική συσκευή, ώστε το σχήμα της να εισαχθεί στη βάση δεδομένων της εφαρμογής. Επίσης μετρούνται οι διαστάσεις της εξάρτησης (rig) και των πανιών, δεδομένα πλευστότητας και ευστάθειας, τα οποία προστίθενται και αυτά στη βάση δεδομένων.

Στη συνέχεια, εκτελούνται μια σειρά πολύπλοκων υπολογισμών ώστε να βρεθούν οι ταχύτητες του σκάφους, στις οποίες το άθροισμα των παραγόντων της οπισθέλκουσας (αντίσταση του νερού και του ανέμου) ισοδυναμεί με την πρόωση που παρέχουν τα πανιά.

Το πολικό διάγραμμα είναι σημαντικό εργαλείο για τον πλοηγό και δημοσιεύεται από τους κατασκευαστές ιστιοπλοϊκών σκαφών για κάθε μοντέλο και για κάθε ζεύγος πανιών. Αποτυπώνει τη θεωρητική εφικτή ταχύτητα για ένα συγκεκριμένο ιστιοφόρο σε διάφορες ταχύτητες του ανέμου.

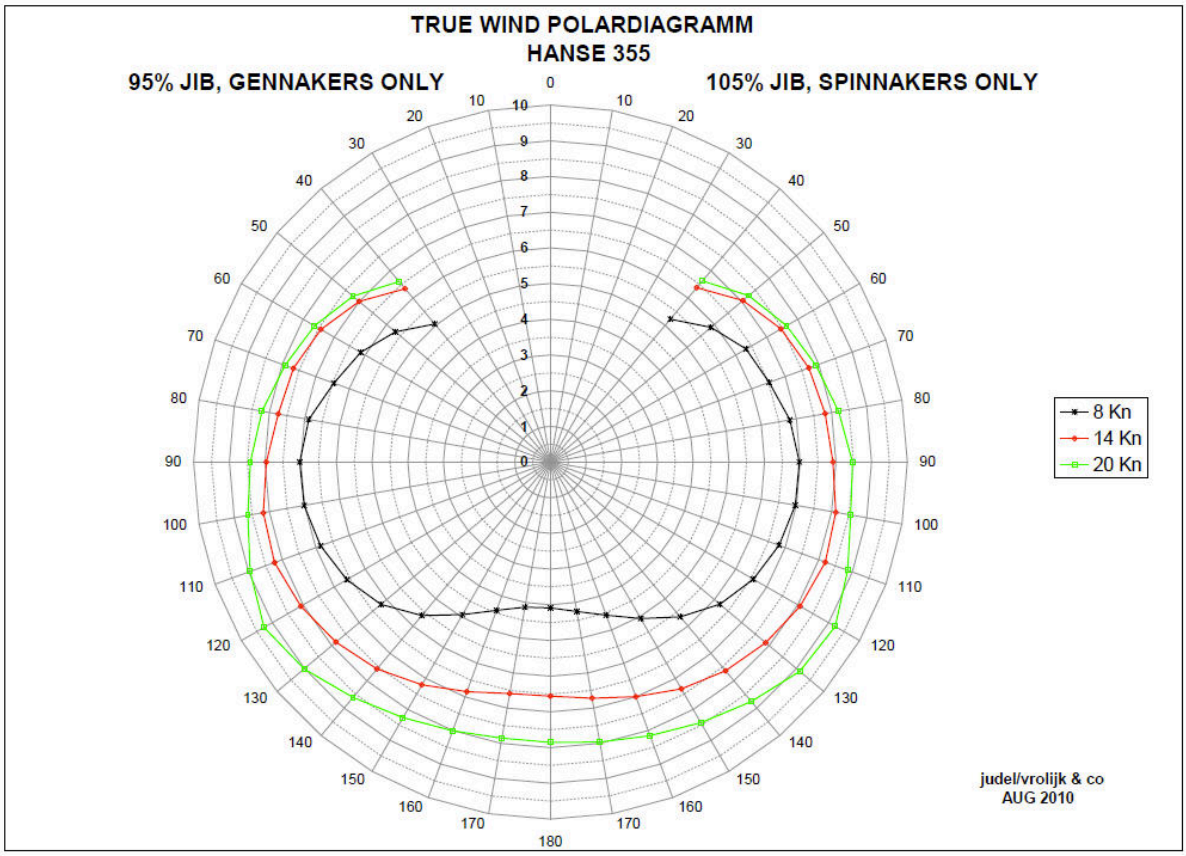
Είναι εύκολο να διαβαστεί τοποθετώντας και μετακινώντας τον διαβήτη ακτινικά κατά μήκος οποιασδήποτε πραγματικής γωνίας του ανέμου και εντοπίζεται η γραμμή επίδοσης τους σκάφους σε σχέση με την πραγματική ταχύτητα του ανέμου.

Στη συνέχεια, διαβάζεται η ταχύτητα του σκάφους στον κατακόρυφο άξονα που αντιστοιχεί για την εν λόγω δεδομένη πραγματική γωνία του ανέμου και τη δεδομένη πραγματική ταχύτητα του ανέμου. Αυτή είναι η ταχύτητα με την οποία το σκάφος πρέπει ιδανικά να πλέει και το πλήρωμα πρέπει να επιτύχει με το τριμάρισμα των πανιών.

Πολλές φορές, ο χρωματικός κώδικας χρησιμοποιείται για να υποδηλώσει την επίδοση του σκάφους σε συνδυασμούς πανιών (τζένοες/φλόκους, μπαλόνια και Code Zero[44]), δείχνοντας την επίδραση αυτών των πανιών στην επίδοση. Οι βέλτιστες γωνίες και ταχύτητες σε όρτσα και πρύμα επίσης σημειώνονται για αγώνες σε στίβους όρτσα-πρύμα.

Στα πολικά διαγράμματα, οι διευθύνσεις του Πραγματικού και του Φαινόμενου Ανέμου υποδεικνύονται από τα εκτυπωμένα βέλη και αυξάνονται ακτινωτά από τις 0° στην κορυφή έως τις 180° στο κάτω μέρος του διαγράμματος, όπως στο παρακάτω διάγραμμα ενός ιστιοπλοϊκού σκάφους HANSE 355[45].

[43]https://en.wikipedia.org/wiki/Forces_on_sails
[44]https://www.quantumsails.com/get_file.aspx?file=8741ec61-eda4-4bec-940f-92f5f8c1cd9c
[45]http://www.myhanse.com/hanse-355-polar-diagram_topic4660.html




**Εικόνα 61 - Πολικό Διάγραμμα HANSE 355**

Κάθε ακτίνα που εκτείνεται από το κέντρο αντιπροσωπεύει και μία γωνία πλεύσης σχετική με την υποδηλούμενη Γωνία του Πραγματικού Ανέμου (TWA, True Wind Angle) ή τη Γωνία του Φαινόμενου Ανέμου (AWA, Apparent Wind Angle). Η Γωνία Φαινόμενου Ανέμου αναφέρεται στον άνεμο όπως γίνεται αισθητός ή πατρατηρείται στο σκάφος ή που αντιλαμβάνεται το ανεμόμετρο στην κορυφή του καταρτιού.

Κάθε ακτίνα διαβαθμίζεται σε υποδιαιρέσεις του ενός κόμβου και σε μικρότερες υποδιαιρέσεις στα δέκατα του κόμβου. Αυτές είναι κλίμακες της προβλεπόμενης ταχύτητας του σκάφους. Όσο μεγαλύτερη η απόσταση από το κέντρο, τόσο υψηλότερη η ταχύτητα του σκάφους. Η αντίστοιχη κλίμακα ταχυτήτων σε κόμβους φαίνεται πάνω στην ευθεία των 90°.

Οι εκτυπωμένες καμπύλες αντιπροσωπεύουν την ταχύτητα του σκάφους σε επτά διαφορετικές Ταχύτητες Πραγματικού Ανέμου (TWS, True Wind Speed): 6, 8, 10, 12, 14, 16 και 20 κόμβων. Η εσωτερική καμπύλη πιο κοντά στο κέντρο δίνει τις ταχύτητες του σκάφους στην TWS των 6 κόμβων ενώ η καμπύλη μακρύτερα από το κέντρο παρουσιάζει τις ταχύτητες του σκάφους σε TWS 20 κόμβων.

Το σχήμα αυτών των καμπυλών δίνει τόσο μια ποιοτική, όσο και μια ποσοτική αντίληψη της επίδοσης: παρατηρείται για παράδειγμα ότι σε κλειστές πλεύσεις, κοντά στις 45° TWA, οι ταχύτητες του σκάφους δεν αυξάνονται ιδιαίτερα σε μεγαλύτερη Ταχύτητα Πραγματικού Ανέμου, ενώ στις ανοικτές πλεύσεις αυξάνονται σημαντικά με την ένταση του ανέμου.

# Απόδοση Πολικών Διαγραμμάτων σε δομή JSON

Στο πλαίσιο της εργασίας λήφθηκε ως αντιπροσωπευτικό πολικό διάγραμμα αυτό ενός πολύ γνωστού ιστιοπλοϊκού σκάφους του Beneteau First 40.7[46]. Τα στοιχεία του πολικού διαγράμματος δίνονται σε μορφή πίνακα όπως παρακάτω:

| **Wind Speed (kt) / Wind Angle** | **4** | **6** | **8** | **10** | **12** | **14** | **16** | **20** | **25** | **30** |
|---|---|---|---|---|---|---|---|---|---|---|
| **33.0** | 2.37 | 3.57 | 4.50 | 5.17 | 5.62 | 5.90 | 6.05 | 6.08 | 5.74 | 4.67 |
| **36.0** | 2.69 | 3.99 | 4.98 | 5.68 | 6.10 | 6.35 | 6.50 | 6.55 | 6.34 | 5.78 |
| **39.0** | 2.97 | 4.36 | 5.40 | 6.08 | 6.48 | 6.71 | 6.84 | 6.90 | 6.79 | 6.41 |
| **42.0** | 3.23 | 4.68 | 5.75 | 6.42 | 6.78 | 6.98 | 7.09 | 7.16 | 7.11 | 6.88 |
| **45.0** | 3.46 | 4.97 | 6.05 | 6.69 | 7.01 | 7.18 | 7.28 | 7.36 | 7.35 | 7.21 |
| **50.0** | 3.79 | 5.37 | 6.44 | 7.03 | 7.29 | 7.44 | 7.53 | 7.64 | 7.67 | 7.60 |
| **60.0** | 4.29 | 5.92 | 6.92 | 7.43 | 7.69 | 7.84 | 7.95 | 8.10 | 8.19 | 8.20 |
| **70.0** | 4.59 | 6.23 | 7.15 | 7.65 | 7.98 | 8.16 | 8.29 | 8.47 | 8.62 | 8.68 |
| **80.0** | 4.72 | 6.36 | 7.24 | 7.75 | 8.14 | 8.42 | 8.58 | 8.81 | 9.04 | 9.18 |
| **90.0** | 4.70 | 6.34 | 7.23 | 7.78 | 8.18 | 8.53 | 8.81 | 9.17 | 9.48 | 9.70 |
| **100.0** | 4.51 | 6.17 | 7.27 | 7.86 | 8.24 | 8.48 | 8.83 | 9.45 | 9.89 | 10.23 |
| **110.0** | 4.36 | 6.13 | 7.21 | 7.82 | 8.29 | 8.63 | 8.87 | 9.42 | 10.28 | 10.87 |
| **120.0** | 4.15 | 5.88 | 7.02 | 7.67 | 8.19 | 8.63 | 9.03 | 9.63 | 10.26 | 11.42 |
| **130.0** | 3.76 | 5.43 | 6.69 | 7.44 | 7.99 | 8.46 | 8.91 | 9.84 | 10.97 | 11.98 |
| **135.0** | 3.53 | 5.16 | 6.46 | 7.29 | 7.85 | 8.33 | 8.77 | 9.74 | 11.21 | 12.59 |
| **140.0** | 3.30 | 4.86 | 6.18 | 7.10 | 7.69 | 8.18 | 8.61 | 9.54 | 11.10 | 13.01 |
| **150.0** | 2.83 | 4.25 | 5.52 | 6.56 | 7.28 | 7.79 | 8.23 | 9.06 | 10.35 | 12.32 |
| **160.0** | 2.40 | 3.64 | 4.80 | 5.85 | 6.74 | 7.36 | 7.85 | 8.67 | 9.78 | 11.39 |
| **170.0** | 2.17 | 3.28 | 4.36 | 5.38 | 6.29 | 7.02 | 7.55 | 8.39 | 9.40 | 10.72 |
| **180.0** | 2.02 | 3.06 | 4.08 | 5.05 | 5.95 | 6.72 | 7.30 | 8.16 | 9.09 | 10.19 |

Στη συνέχεια μετατράπηκε η δομή πίνακα σε δομή JSON και το τελικό αρχείο εισόδου στο σύστημα είχε ως εξής:

```
{
        "wind_degrees": {
                "33": {
                        "speed": [{
                                "wind": 4,
                                "boat": 2.37
                        }, {
                                "wind": 6,
                                "boat": 3.57
                        }, {
                                "wind": 8,
                                "boat": 4.50
                        }, {
                        . . . . .
```

[46]http://www.blur.se/polar/first407_performance_prediction.pdf

```
                        }, {
                                "wind": 30,
                                "boat": 4.67
                        }]
                },
                "36": {
                        "speed": [{
                                "wind": 4,
                                "boat": 2.69
                        }, {
                        . . . . .
                        }, {
                                "wind": 30,
                                "boat": 10.19
                        }]
                }
        }
}
```

# Λήψη Μετεωρολογικών Δεδομένων

### OpenWeatherMapAPI

Για τη λήψη μετεωρολογικών δεδομένων χρησιμοποιήθηκε η διαδικτυακή υπηρεσία OpenWeatherMap[47]. Η υπηρεσία παρέχει δωρεάν API[48] με πληροφορίες καιρού, όπως προβλέψεις καιρού και ιστορικά μετεωρολογικά δεδομένα. Ως πηγές δεδομένων, χρησιμοποιεί μετεωρολογικές υπηρεσίες μετάδοσης δεδομένων, καθώς και πρωτογενή δεδομένα από μετεωρολογικούς σταθμούς αεροδρομίων, από σταθμούς ραντάρ και από άλλους επίσημους μετεωρολογικούς σταθμούς.

Όλα τα δεδομένα υποβάλλονται σε επεξεργασία από την OpenWeatherMap, τέτοια ώστε να παρέχει ακριβείς και σε πραγματικό χρόνο δεδομένα πρόγνωσης καιρού αλλά και χάρτες καιρού, όπως αυτοί για τα σύννεφα και την υγρασία.

Από εκεί και πέρα, η υπηρεσία εστιάζεται στην κοινωνική διάσταση, με συμμετοχή ιδιοκτητών των μετεωρολογικών σταθμών για τη διασύνδεσή τους με την υπηρεσία για τη βελτίωση της ακρίβειας των μετεωρολογικών δεδομένων. Η ιδέα είναι εμπνευσμένη από το OpenStreetMap και Wikipedia που διαθέτουν την πληροφορία δωρεάν προς όλους. Χρησιμοποιεί βασικά το περιβάλλον OpenStreetMap για την εμφάνιση των χαρτών καιρού, αλλά έχουν αναπτυχθεί διάφορα modules για την

[47]http://openweathermap.org/
[48]http://openweathermap.org/api

εμφάνιση των χαρτών και σε άλλους τύπους διαδικτυακών χαρτών καθώς και mobile εφαρμογών[49].


```
{
        "list": {
                "9_237": {
                        "id": "9_237",
                        "lat": "36.2374642487",
                        "lon": "24.0573083897",
                        "city": {
                                "id": 8133681,
                                "name": "Dimos Milos",
                                "coord": {
                                        "lon": 24.424749,
                                        "lat": 36.708641
                                },
                                "country": "GR",
                                "population": 0,
                        },
                        "forecast": {
                                "dt": 1453226400,
                                "main": {
                                        "temp": 283.44,
                                        "temp_min": 283.44,
                                        "temp_max": 283.472,
                                        "pressure": 1032.98,
                                        "sea_level": 1033.77,
                                        "grnd_level": 1032.98,
                                        "humidity": 100,
                                        "temp_kf": -0.04
                                },
                                "weather": [{
                                        "id": 801,
                                        "main": "Clouds",
                                        "description": "few clouds",
                                        "icon": "02n"
                                }],
                                "clouds": {
                                        "all": 12
                                },
                                "wind": {
```

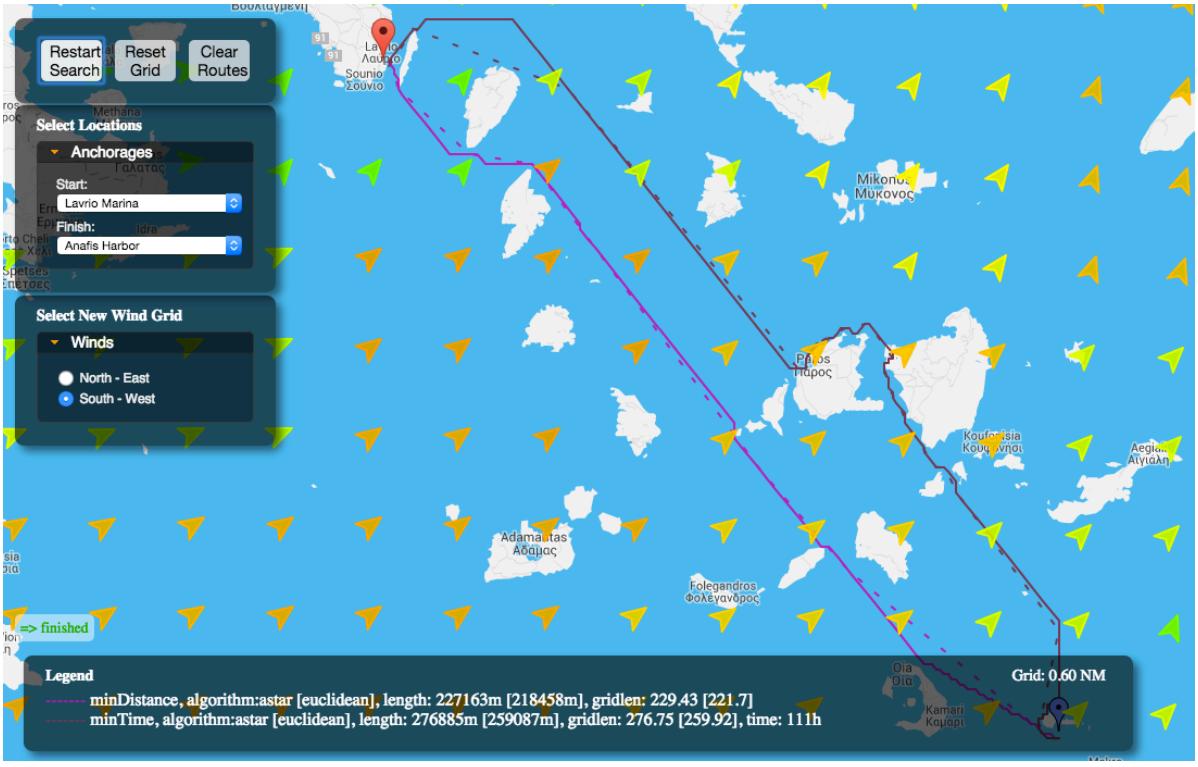

[49]http://openweathermap.org/examples

```
                                        "speed": 7.86,
                                        "deg": 30.0005
```

```
                                    },
                                    "dt_txt": "2016-01-19 18:00:00"
                            }
                    },
                    "9_238": {
                            "id": "9_238",
                            "lat": "36.2371506292",
                            "lon": "24.2798828744",
                            . . . . .
                    }
            }
}
```


# Υπολογισμός Βέλτιστου Χρόνου Πλεύσης

Για την εφαρμογή των παραπάνω, τροποποιήθηκε κατάλληλα η δομή του πλέγματος ώστε να περιλαμβάνει στοιχεία καιρού σε κάθε φατνίο, δηλαδή ταχύτητα ανέμου και κατεύθυνση. Με βάση τα στοιχεία καιρού και σε συνδυασμό με την πληροφορία του πολικού διαγράμματος ήταν δυνατό πλέον να υπολογιστεί για κάθε κόμβο και για κάθε πιθανή κατεύθυνση πλεύσης, η ταχύτητα πλεύσης βάσει των καιρικών συνθηκών.



**Εικόνα 62 - Πειραματική Διάταξη Υπολογισμού Βέλτιστου Χρόνου**

Στη συνέχεια τροποποιήθηκε κατάλληλα ο αλγόριθμος A*, ώστε ως κόστος μετάβασης να υπολογίζεται πλέον ο χρόνος μετάβασης (sailing time) και όχι η απόσταση (sailing distance) που υπολόγιζε ως τώρα.

```
sailing_time = sailing_distance / boat_speed;
```

Ο χρόνος μετάβασης υπολογίζεται διαιρώντας την απόσταση, με την υπολογιζόμενη ως ανωτέρω ταχύτητα πλεύσης, για τη μετάβαση στα υπό εξέταση γειτονικά πλακίδια. Το αποτέλεσμα είναι ο αλγόριθμος να υπολογίζει για ένα πλου και την πιο σύντομη χρονικά διαδρομή βάσει των καιρικών συνθηκών εισόδου στο πλέγμα.

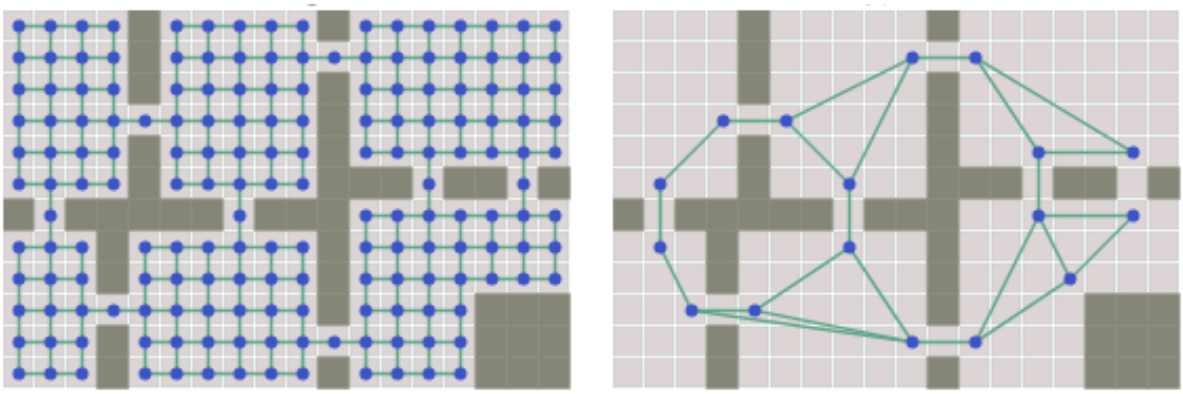
## Κεφάλαιο 9 – Βελτίωση Συστήματος &Επόμενα βήματα

# Βελτιστοποιήσεις Αλγορίθμων Pathfinding

Οι αλγόριθμοι εύρεσης βέλτιστης διαδρομής (pathfinding) όπως ο A * και ο αλγόριθμος του Dijkstra αναφέρονται σε γραφήματα. Για να χρησιμοποιηθούν σε πλέγμα, αναπαριστώνται τα πλέγματα με γραφήματα. Για τους βασικούς χάρτες πλεγμάτων η αντιστοίχιση λειτουργεί ομαλά. Ωστόσο, για εφαρμογές που μπορεί να χρειάζεται μεγαλύτερη απόδοση, υπάρχουν μια σειρά από προτεινόμενες βελτιστοποιήσεις[50].

### Απλούστερη Αναπαράσταση του Γράφου

Είναι σημαντικό να εξεταστεί αν μπορεί να χρησιμοποιηθεί ένα απλούστερο γράφημα για εύρεση της διαδρομής (pathfinding). Στην παρακάτω εικόνα απεικονίζεται το ίδιο πλέγμα με δύο διαφορετικά γραφήματα για εύρεση διαδρομής:



Εικόνα 63 - Συσχέτιση πλέγματος με Waypoints

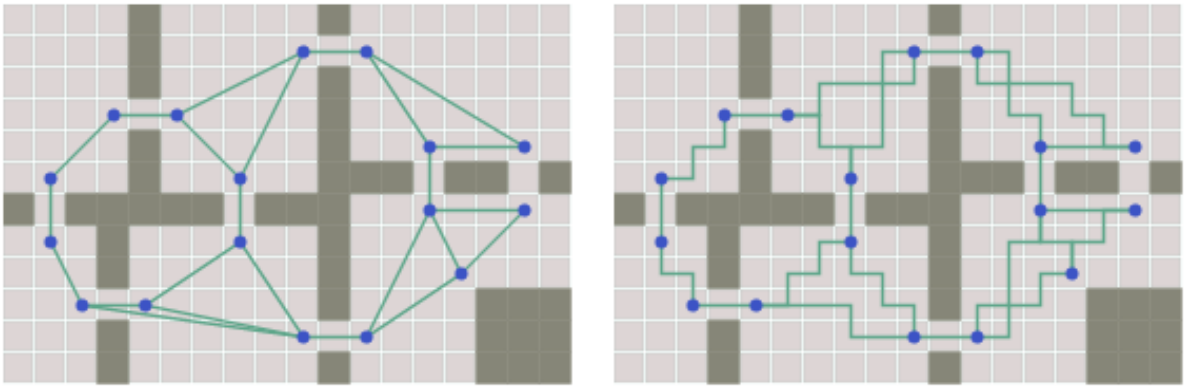
Γενικά ισχύει ότι, όσο λιγότεροι είναι οι κόμβοι που ο αλγόριθμος πρέπει να επεξεργαστεί τόσο πιο γρήγορα θα τρέξει. Κάποιοι τρόποι για να μειωθεί ο αριθμός των κόμβων του γραφήματος είναι:

- Σημεία Διαδρομής (*Waypoints*), είναι τα βασικά σημεία λήψης αποφάσεων, όπου ίσως χρειαστεί να γίνει αλλαγή κατεύθυνσης. Στο παραπάνω διάγραμμα, αποτυπώνονται τα σημεία όπου θα πρέπει το μονοπάτι να πάει γύρω από μια γωνία ή τοίχο. Στην περίπτωση του υπό ανάπτυξη συστήματος τα waypoints αφορούν ακρωτήρια, βράχους ή βραχονησίδες.
- Πλέγματα πλοήγησης (*Navigation Meshes*) είναι οι προσπελάσιμες (walkable) περιοχές του χάρτη. Οι περιοχές μπορεί να είναι οποιουδήποτε μεγέθους και σχήματος.
- Ιεραρχικές προσεγγίσεις (*Hierarchical Approaches*) έχουν γενικές και αναλυτικές αναπαραστάσεις στο χάρτη. Μια κατά προσέγγιση διαδρομή μπορεί να υπολογιστεί με το γενικό επίπεδο, και στη συνέχεια να βελτιωθεί με το αναλυτικό επίπεδο.

[50]http://www.redblobgames.com/pathfinding/grids/algorithms.html

- Τετραδικά Δέντρα (*Quad Trees*) χρησιμοποιούν τετράγωνες περιοχές με διαφορετική ανάλυση για να αντιπροσωπεύουν τον χάρτη. Μεγάλες ανοικτές περιοχές μπορούν να αναπαρασταθούν από μερικά τετράγωνα. Ακανόνιστα άκρα μπορεί να αναπαρασταθούν από πολλές μικρές περιοχές. Μειώνοντας δηλαδή το μέγεθος του χώρου αναζήτησης μέσω αφαίρεσης (abstraction).

Σημειώνεται πως μπορεί η κίνηση να είναι στο πλέγμα, ακόμη και αν η αναπαράσταση στο χάρτη δεν είναι. Αυτά τα δύο έχουν το ίδιο σύνολο από Waypoints:



Εικόνα 64 - Γραμμική κίνηση vs κίνηση σε Πλέγμα

### Αξιοποίηση της δομής του Πλέγματος

Τα Πλέγματα περιέχουν πρόσθετη δομή που οι αλγόριθμοι αναζήτησης σε γράφο δεν εκμεταλλεύονται. Για παράδειγμα, αν κινούμαστε ανατολικά ένα βήμα στο πλέγμα, είναι πιθανό ότι η κίνηση ανατολικά και πάλι πρόκειται να είναι μια καλή κίνηση. Οι αλγόριθμοι γράφων, ωστόσο, δεν γνωρίζουν τι σημαίνει «ανατολικά» ή ότι οι δύο πλευρές μπορούν να είναι προς την ίδια κατεύθυνση. Ένα άλλο παράδειγμα, μια κίνηση με κατεύθυνση τον Βορρά και στη συνέχεια ανατολικά είναι συνήθως το ίδιο όπως πηγαίνοντας ανατολικά και στη συνέχεια βόρεια. Οι αλγόριθμοι γράφων δεν το γνωρίζουν αυτό. Αυτά τα δύο παραδείγματα δείχνουν ότι δύναται να υπάρχει πρόσθετη απόδοση που θα προκύψει από τη μη χρησιμοποίηση των αλγορίθμων απευθείας στο πλέγμα.

Εάν η κίνηση είναι κατά μήκος των κόμβων του πλέγματος και υπάρχουν μεγάλες περιοχές από ομοιόμορφα καταναμεμημένους κόμβους με βάρη, ο αλγόριθμος Jump Point Search[51] κάνει άλματα μπροστά στο πλέγμα αντί να επεξεργαστεί τους κόμβους ένα προς ένα.

Επιπρόσθετα, εάν η κίνηση δεν είναι κατά μήκος των κόμβων του πλέγματος και γίνεται αναζήτηση σε πλέγμα, θα είναι επιθυμητό να ισιώσουν οι ζιγκ-ζακ διαδρομές που παράγει ο αλγόριθμος σε πλέγμα. Ωστόσο, τα μονοπάτια που ισιώνουν δεν είναι εγγυημένο πως είναι τα συντομότερα. Ο αλγόριθμος που αναλύει τα ισιωμένα μονοπάτια είναι ο Theta *[52].

[51]http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/
[52]http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/

### Γρηγορότερη διάσχιση κόμβων

Μια άλλη ιδιότητα στα γραφήματα είναι τα βάρη (ποικίλα κόστη μετακίνησης). Ο αλγόριθμος του Dijkstra και ο A* αναλύουν τα βάρη, κατά την εύρεση των μονοπατιών. Πολλές περιπτώσεις δεν απαιτούν βάρη, και τίθενται με την τιμή ένα (1) παντού. Αυτό σημαίνει ότι μπορεί να υπάρχει πρόσθετη βελτίωση της αποτελεσματικότητας της αναζήτησης, αν υπάρχουν ομοιόμορφα κόστη κίνησης στον χάρτη. Για παράδειγμα, ο Breadth First Search είναι μια καλύτερη επιλογή από ό, τι ο αλγόριθμος του Dijkstra, αν δεν χρειαζόμαστε βάρη.

Εάν τα κόστη μετακίνησης είναι ομοιόμορφα, ή σε ένα στενό εύρος τιμών, μπορούν να γίνουν τα εξής:

- να χρησιμοποιηθεί ουρά προτεραιότητας με κάδους για να επωφεληθούν από το περιορισμένο εύρος των εξόδων μετακίνησης.
- να χρησιμοποιηθεί μια ξεχωριστή ουρά προτεραιότητας για του μη πιθανούς κόμβους.
- να γίνει προ-επεξεργασία των κόμβων στην ουρά προτεραιότητας.

### Βελτίωση της Ευρετικής

Ο σκοπός της ευρετικής συνάρτησης είναι να δώσει μία εκτίμηση του μήκους της μικρότερης διαδρομής. Όσο πιο κοντά είναι η ευρετική στην πραγματική συντομότερη διαδρομή, τόσο ταχύτερα εκτελείται ο A*. Όταν η ευρετική είναι 0, ο A* γίνεται ο αλγόριθμος του Dijkstra. Όταν η ευρετική είναι ίση με το μικρότερο μήκος διαδρομής, ο A* βρίσκει αμέσως τη συντομότερη διαδρομή και δεν χρειάζεται να διερευνήσει και άλλες περιοχές. Όταν η ευρετική είναι μεγαλύτερη από το συντομότερο μήκος της διαδρομής, ο A* δεν εγγυάται πλέον συντομότερα μονοπάτια.

Στα πλέγματα, συνήθως χρησιμοποιείται η απόσταση ως ευρετική. Η απόσταση είναι μεγαλύτερη από το 0, αλλά λιγότερη από το μικρότερο μήκος της διαδρομής, έτσι ώστε μας δίνει τα συντομότερα μονοπάτια, αλλά όχι την καλύτερη ταχύτητα.

Σε περιπτώσεις χαρτών, είναι δυνατό να αναλυθεί ο χάρτης ώστε να παράγει καλύτερες ευρετικές. Μπορεί να γίνει επιλογή μια σειρά από waypoints και στη συνέχεια να βρεθούν τα συντομότερα μονοπάτια μεταξύ τους. Το μήκος της διαδρομής μεταξύ των waypoints μπορεί στη συνέχεια να χρησιμοποιηθεί στην ευρετική για να υπολογίσει μια καλύτερη εκτίμηση του μικρότερου μήκους διαδρομής, με πιθανές σημαντικές επιταχύνσεις.

# Περιπτώσεις Χρήσης του Συστήματος

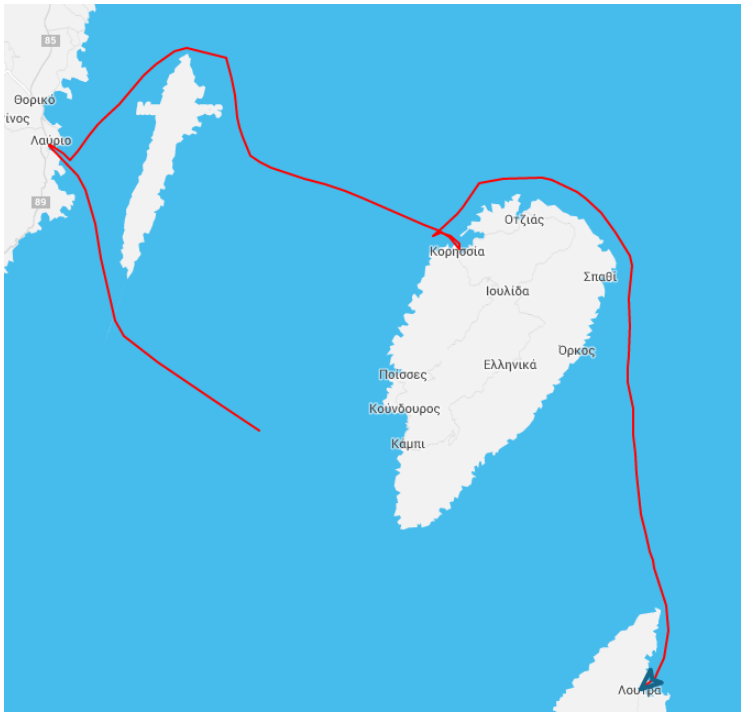
### Επικαιροποίηση διαδρομής σε πραγματικό χρόνο

Στο σύστημα που αναπτύχθηκε έγινε χρήση στατικών μετεωρολογικών δεδομένων πρόγνωσης καιρού. Αν το σύστημα χρησιμοποιηθεί πέρα από σχεδιασμό αλλά και για παρακολούθηση της πορείας, θα μπορούσε να αναπροσαρμόζει τη βέλτιστη χρονικά διαδρομή σε τακτά χρονικά διαστήματα λαμβάνοντας υπόψη επικαιροποιήσεις των

στοιχείων πρόγνωσης. Επιπρόσθετα μπορεί να γίνει διερεύνηση παραμέτρων και κανόνων με σκοπό την παραμετροποίηση του συστήματος για ασφαλή πλου.

### Σύγκριση με πραγματικές διαδρομές

Σε επόμενη φάση θα μπορεί να γίνει μελέτη και σύγκριση των υπολογιζόμενων διαδρομών με τις πραγματικές διαδρομές που ακολουθούν οι χρήστες. Οι πραγματικές διαδρομές μπορούν να καταγραφούν μέσω της υπηρεσίας AIS[53] που καταγράφει τη θέση των σκαφών σε πραγματικό χρόνο.




**Εικόνα 65 - Καταγραφή κίνησης Σκάφους μέσω AIS Συστήματος**

### Επέκταση της εφαρμογής σε Ποντοπόρο Πλου

Το σύστημα που αναπτύχθηκε είχε ως σκοπό να εξυπηρετήσει τον αυτόματο σχεδιασμό του βέλτιστου πλου - εύρεση διαδρομής με την ελάχιστη συνολική απόσταση και ελάχιστο χρόνο πλεύσης βάσει καιρικών συνθηκών – σε πεπερασμένης έκτασης περιοχή. Σε μια επόμενη εργασία μπορούν να μελετηθούν οι ανάγκες σχεδιασμού ποντοπόρου πλου και να γίνουν οι σχετικές τροποποιήσεις προκειμένου το σύστημα να παρέχει λειτουργικότητα για υπολογισμό βέλτιστου ορθοδρομικού πλου, αποφυγή έντονων καιρικών συνθηκών, εξοικονόμηση κόστους καυσίμων κ.α.

[53]http://sail-la-vie.com/enjoy/sailing-in-Greece

# Συμπεράσματα

Η εργασία είχε ως σκοπό την ανάπτυξη συστήματος για τον παραμετρικό σχεδιασμό ιστιοπλοϊκού πλου, με χρήση ανοικτού λογισμικού, εφαρμόζοντας σύγχρονες μεθόδους γεωπληροφορικής σε μια πεπερασμένη περιοχή, όπως είναι η Ελλάδα, με έντονα νησιωτικά συμπλέγματα. Η σκοπιά της προσέγγισης αφορούσε τον αυτόματο σχεδιασμό του βέλτιστου πλου με εύρεση διαδρομής με την ελάχιστη συνολική απόσταση και του ελάχιστου χρόνου πλεύσης βάσει καιρικών συνθηκών. Η εργασία δύναται να χρησιμοποιηθεί ως βάση για τη μελέτη και την ανάπτυξη συστημάτων που παρακολουθούν την πορεία του σκάφους και παράγουν αποτελέσματα σε πραγματικό χρόνο, συγκρίνουν τη διαδρομή βάσει πραγματικών διαδρομών που έχουν καταγραφεί καθώς και στην εξυπηρέτηση αναγκών σχεδιασμού και παρακολούθησης πλόων της ποντοπόρου ναυτιλίας.

Στην βιβλιογραφία παρατίθενται μελέτες για την βελτιστοποίηση των αλγορίθμων εύρεσης βέλτιστης διαδρομής καθώς και βελτίωσης της απόδοσης των εφαρμογών τους. Επειδή τα πλέγματα μπορούν να θεωρηθούν ως ειδική περίπτωση γραφημάτων και περιέχουν πρόσθετη δομή που δεν υπάρχει στους γράφους, οι βασικοί αλγόριθμοι αναζήτησης σε γράφο δεν επωφελούνται αυτής της δομής. Επίσης συχνά στα πλέγματα χαρτών παρατηρείται ομοιόμορφο κόστος μετακίνησης, οδηγώντας σε γραφήματα με ακμές χωρίς βάρη, ενώ πολλοί βασικοί αλγόριθμοι αναζήτησης σε γράφο αφιερώνουν χρόνο αναλύοντας τα βάρη. Ο υπολογισμός της απόστασης ως ευρετική του A* είναι εύκολη για κατανόηση και υπολογισμούς, αλλά συνήθως δεν είναι η καλύτερη ευρετική. Οι περισσότεροι χάρτες δύναται να προεπεξεργαστούν για να δημιουργηθεί μια καλύτερη ευρετική. Τέλος σε περιπτώσεις χαρτών η κίνηση του αντικείμενου δεν περιορίζεται στο πλέγμα και οι βασικοί αλγόριθμοι αναζήτησης σε γράφο δεν λαμβάνουν υπόψη το γεγονός αυτό.

## ΑΝΑΦΟΡΕΣ

[1] https://peripluscd.wordpress.com/2013/10/

[2] http:// sail-la-vie.com/discover/sailing-in-Greece

[3] http://sail-la-vie.com/plan/routes/140/Cyclades%20Unexplored

[4] http://mathworld.wolfram.com/HamiltonianGraph.html

[5] https://en.wikipedia.org/wiki/Glossary_of_graph_theory

[6] https://en.wikipedia.org/wiki/Queue_(abstract_data_type)

[7] https://en.wikipedia.org/wiki/Priority_queue

[8] http://stackoverflow.com/questions/7423401/whats-the-difference-between-the-data-structure-tree-and-graph

[9] https://en.wikipedia.org/wiki/Heap_(data_structure)

[10] https://en.wikipedia.org/wiki/Binary_heap

[11] https://en.wikipedia.org/wiki/Binomial_heap

[12] https://en.wikipedia.org/wiki/Fibonacci_heap

[13] https://en.wikipedia.org/wiki/Fibonacci_number

[14] http://web.cs.ucdavis.edu/~amenta/w10/dijkstra.pdf

[15] http://www.redblobgames.com/pathfinding/a-star/introduction.html

[16] https://en.wikipedia.org/wiki/Breadth-first_search

[17] https://en.wikipedia.org/wiki/Best-first_search

[18] http://everything.explained.today/A*_search_algorithm/

[19] https://cse.sc.edu/~mgv/csce580f09/gradPres/korf_IDAStar_1985.pdf

[20] http://discrete.gr/complexity/?el

[21] http://bigocheatsheet.com/

[22] http://bigocheatsheet.com/

[23] http://stackoverflow.com/questions/21065855/the-big-o-on-the-dijkstra-fibonacci-heap-solution

[24] http://stackoverflow.com/questions/19869236/why-does-a-run-faster-than-dijkstras-algorithm

[25] https://en.wikipedia.org/wiki/QGIS

[26] http://news.netcraft.com/archives/category/web-server-survey/

[27] https://www.w3.org/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript

[28] https://en.wikipedia.org/wiki/Brendan_Eich

[29] http://www.ecma-international.org/

[30] https://en.wikipedia.org/wiki/Jesse_James_Garrett

[31] https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html

[32] http://googlegeodevelopers.blogspot.gr/2013/05/a-fresh-new-look-for-maps-api-for-all.html

[33] https://github.com/qiao/PathFinding.js

[34] http://www.json.org/

[35] http://www.tutorialspoint.com/html5/html5_web_sql.htm

[36] http://qiao.github.io/PathFinding.js/visual/

[37] https://nsidc.org/data/atlas/epsg_4326.html

[38] http://en.wikipedia.org/wiki/Bresenham's_line_algorithm#Simplification

[39] http://theory.stanford.edu/~amitp/GameProgramming/index.html

[40] https://en.wikipedia.org/wiki/Travelling_salesman_problem

[41] https://en.wikipedia.org/wiki/2-opt

[42] http://www.offshore.org.gr/docs/ORC_Speed_Guide_Explanation_EL.pdf

[43] https://en.wikipedia.org/wiki/Forces_on_sails

[44] https://www.quantumsails.com/get_file.aspx?file=8741ec61-eda4-4bec-940f-92f5f8c1cd9c

[45] http://www.myhanse.com/hanse-355-polar-diagram_topic4660.html

[46] http://www.blur.se/polar/first407_performance_prediction.pdf

[47] http://openweathermap.org/

[48] http://openweathermap.org/api

[49] http://openweathermap.org/examples

[50] http://www.redblobgames.com/pathfinding/grids/algorithms.html

[51] http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/

[52] http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/

[53] http://sail-la-vie.com/enjoy/sailing-in-Greece

## ΒΙΒΛΙΟΓΡΑΦΙΑ


1. *Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James; Tarjan, Robert E. (April 1990), "Faster algorithms for the shortest path problem", Journal of the ACM (ACM) 37 (2): 213–223.*
2. *Bellman, Richard (1958), "On a routing problem", Quarterly of Applied Mathematics 16: 87–90.*
3. *Cherkassky, Boris V.; Goldberg, Andrew V.; Radzik, Tomasz (1996), "Shortest paths algorithms: theory and experimental evaluation", Mathematical Programming. Ser. A 73 (2): 129–174.*
4. *Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990], "Single-Source Shortest Paths and All-Pairs Shortest Paths", Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 580–642.*
5. *Dantzig, G. B. (January 1960), "On the Shortest Route through a Network", Management Science 6 (2): 187–190.*
6. *Dijkstra, E. W. (1959), "A note on two problems in connexion with graphs" (PDF), Numerische Mathematik 1: 269–271.*
7. *Ford, L. R. (1956), "Network Flow Theory", Rand Corporation. P-923.*
8. *Fredman, Michael Lawrence; Tarjan, Robert E. (1984), Fibonacci heaps and their uses in improved network optimization algorithms, 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346.*
9. *Fredman, Michael Lawrence; Tarjan, Robert E. (1987), "Fibonacci heaps and their uses in improved network optimization algorithms", Journal of the Association for Computing Machinery 34 (3): 596–615.*
10. *Gabow, H. N. (1983). "Scaling algorithms for network problems". Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS 1983). pp. 248–258.*
11. *Gabow, Harold N. (1985). "Scaling algorithms for network problems". Journal of Computer and System Sciences 31 (2): 148–168.*
12. *Hagerup, Torben (2000). Montanari, Ugo; Rolim, José D. P.; Welzl, Emo, eds. Improved Shortest Paths on the Word RAM. Proceedings of the 27th International Colloquium on Automata, Languages and Programming. pp. 61–72.*
13. *Johnson, Donald B. (December 1981). "A priority queue in which initialization and queue operations take O(log log D) time". Mathematical Systems Theory 15 (1): 295–309.*
14. *Karlsson, Rolf G.; Poblete, Patricio V. (1983). "An O(m log log D) algorithm for shortest paths". Discrete Applied Mathematics 6 (1): 91–93.*
15. *Kavouras M., Stefanakis, Em. JANUARY 1995, DETERMINATION OF THE OPTIMUM PATH ON THE EARTH'S SURFACE - ARTICLE - National Technical University of Athens - University of New Brunswick, Proceedings of the 17th International Cartographic Association Conference, Barcelona, Spain, September 1995*

16. Korf, Richard (1985), "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence* 27: 97–109.

17. *Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, S. R., Jr.; Petry, R. M.; Seitz, R. N. (1957). Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model Techniques for Communication Systems. Cleveland, Ohio: Case Institute of Technology.*

18. *Moore, E. F. (1959). "The shortest path through a maze". Proceedings of an International Symposium on the Theory of Switching (Cambridge, Massachusetts, 2–5 April 1957). Cambridge: Harvard University Press. pp. 285–292.*

19. Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984. p. 48.

20. *Pettie, Seth; Ramachandran, Vijaya (2002). Computing shortest paths with comparisons and additions. Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 267–276.*

21. *Pettie, Seth (26 January 2004). "A new approach to all-pairs shortest paths on real-weighted graphs". Theoretical Computer Science 312 (1): 47–74.*

22. *Pollack, M.; Wiebenson, W. (March–April 1960). "Solution of the Shortest-Route Problem—A Review". Op. Res. 8 (2): 224–230.*

23. *Schrijver, Alexander (2004). Combinatorial Optimization — Polyhedra and Efficiency. Algorithms and Combinatorics 24. Springer. ISBN 3-540-20456-3.* Here: vol.A, sect.7.5b, p.103

24. *Shimbel, Alfonso (1953). "Structural parameters of communication networks". Bulletin of Mathematical Biophysics 15 (4): 501–507. doi:10.1007/BF02476438.*

25. *Thorup, Mikkel (1999). "Undirected single-source shortest paths with positive integer weights in linear time". Journal of the ACM (JACM) 46 (3): 362–394. Retrieved 28 November 2014.*

26. *Whiting, P. D.; Hillier, J. A. (March–June 1960). "A Method for Finding the Shortest Route through a Road Network". Operational Research Quarterly 11 (1/2): 37–40.*

27. *Williams, Ryan (2014). "Faster all-pairs shortest paths via circuit complexity". Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC '14). New York: ACM. pp. 664–673.*