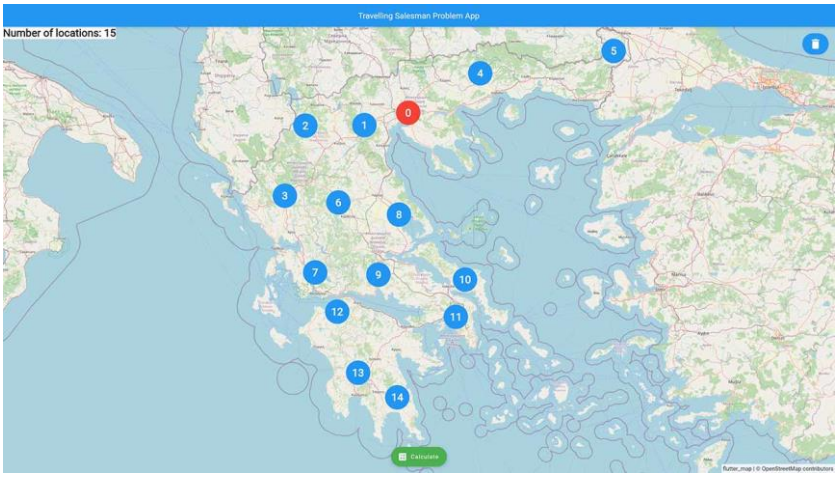
ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«Ανάπτυξη εφαρμογής για την επίλυση του προβλήματος του πλανόδιου πωλητή με πραγματικά γεωγραφικά δεδομένα»



Του φοιτητή
Κεσάνογλου Αντώνιου
Αρ. Μητρώου: 174889

Επιβλέπων
Αντωνίου Ευστάθιος
Βαθμίδα ……

Ημερομηνία 20-01-2023

Τίτλος Π.Ε. Ανάπτυξη εφαρμογής για την επίλυση του προβλήματος του πλανόδιου πωλητή με πραγματικά γεωγραφικά δεδομένα

Κωδικός Π.Ε. 21227

Ονοματεπώνυμο φοιτητή Αντώνιος Κεσάνογλου

Ονοματεπώνυμο εισηγητή Ευστάθιος Αντωνίου

Ημερομηνία ανάληψης Π.Ε. 31-03-2021

Ημερομηνία περάτωσης Π.Ε. 20-01-2023

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Πρόλογος

Το θέμα της εργασίας αυτής είναι το πρόβλημα του πλανόδιου πωλητή το οποίο αποτελείται από έναν πωλητή και ένα σύνολο πόλεων το οποίο πρέπει να επισκεφτεί. Η πρόκληση στο πρόβλημα αυτό είναι να βρούμε με ποια σειρά να επισκεφτεί καθεμία από αυτές τις πόλεις, ακριβώς μία φορά, διανύοντας την μικρότερη δυνατή απόσταση. Έχουν αναπτυχθεί διάφορες μεθοδολογίες τις τελευταίες δεκαετίες για την επίλυση του προβλήματος με την χρήση αλγορίθμων.

Επιλέξαμε να ερευνήσουμε αυτό το πρόβλημα στην παρούσα εργασία καθώς έχει εφαρμογές σε πολλά συναφή προβλήματα της καθημερινότητας όπως είναι η διακίνηση και παράδοση προϊόντων σε πολλά σημεία με το χαμηλότερο δυνατό κόστος, τα logistics καθώς και σε πιο εξειδικευμένα αντικείμενα όπως την σχεδίαση κυκλωμάτων. Στα πλαίσια της εργασίας, έγινε μια επισκόπηση της θεωρίας του προβλήματος καθώς και των μεθοδολογιών επίλυσής του. Παράλληλα, αναπτύξαμε μια εφαρμογή ιστού η οποία δίνει την δυνατότητα στον καθένα να επιλύσει το TSP με πραγματικά γεωγραφικά δεδομένα επιλέγοντας σημεία στον χάρτη και έναν αλγόριθμο για την εύρεση της λύσης.

Περίληψη

Το πρόβλημα του πλανόδιου πωλητή (Traveling Salesman Problem - TSP) αφορά τον υπολογισμό μιας κλειστής διαδρομής, κατά την οποία ο υποτιθέμενος πωλητής πρέπει να επισκεφθεί ακριβώς μια φορά μια σειρά από πόλεις και να επιστρέψει στην αφετηρία του, ελαχιστοποιώντας την συνολική απόσταση που θα διανύσει, δεδομένων των αποστάσεων μεταξύ κάθε ζεύγους πόλεων. Πρόκειται για ένα από τα πιο γνωστά NP-Hard προβλήματα συνδυαστικής βελτιστοποίησης, με τεράστια σημασία τόσο για τη θεωρητική πληροφορική, όσο και για την επιχειρησιακή έρευνα.

Ο συνηθέστερος τρόπος μοντελοποίησης του προβλήματος βασίζεται σε μη κατευθυνόμενα γραφήματα των οποίων οι κορυφές παριστάνουν τα σημεία ενδιαφέροντος (πόλεις) στην περιοδεία του πλανόδιου πωλητή, ενώ οι ακμές μεταξύ αυτών εφοδιάζονται με βάρη που εκφράζουν το κόστος της χρήσης της αντίστοιχης διαδρομής. Για την επίλυση του προβλήματος ακολουθούνται γενικά τρεις διαφορετικές προσεγγίσεις, οι οποίες είναι:

- Ακριβείς αλγόριθμοι που υπολογίζουν τη βέλτιστη διαδρομή αλλά μπορούν να εφαρμοστούν μόνο σε μικρής έκτασης προβλήματα,

- Προσεγγιστικοί αλγόριθμοι που υπολογίζουν λύσεις που δεν είναι σίγουρα βέλτιστες αλλά εφαρμόζονται σε προβλήματα μεγαλύτερης κλίμακας σε ανεκτό χρόνο,

- Εξειδικευμένοι αλγόριθμοι που αφορούν προβλήματα των οποίων το γράφημα έχει κάποια ιδιαίτερα χαρακτηριστικά.

Στα πλαίσια της εργασίας αυτής έχει γίνει μια επισκόπηση της αντίστοιχης θεωρίας πίσω από το πρόβλημα του πλανόδιου πωλητή και σε μερικές από τις προσεγγίσεις που υπάρχουν σήμερα για την επίλυση του χρησιμοποιώντας αλγορίθμους. Επίσης, αναπτύχθηκε εφαρμογή ιστού η οποία θα επιλύει το πρόβλημα του πλανόδιου πωλητή χρησιμοποιώντας πραγματικά γεωγραφικά δεδομένα που θα αντλούνται από online χάρτη, το OpenStreetMap. Η εφαρμογή επιτρέπει στον χρήστη την επιλογή N σημείων ενδιαφέροντος στο χάρτη από τα οποία δημιουργείται αρχικά ο πίνακας των $N(N-1)/2$ αποστάσεων μεταξύ αυτών και στη συνέχεια εφαρμόζεται αλγόριθμος επιλεγόμενος από τον χρήστη για την επίλυση του προβλήματος του πλανόδιου πωλητή. Με την ολοκλήρωση της εκτέλεσης του αλγορίθμου εμφανίζονται στο χρήστη τα αποτελέσματα του.

«Application Development to solve the traveling salesman problem using real geographic data»

«Antonios Kesanoglou»

Abstract

The Traveling Salesman Problem (TSP) involves the calculation of a closed route, in which the salesman must visit exactly once a number of cities and return to his starting point. This route must have the minimum travel distance, given the distances between each pair of cities. It is one of the most well-known NP-Hard combinatorial optimization problems, with enormous importance for both theoretical computer science and operations research.

The most common way of illustrating the problem is with undirected graphs whose vertices represent the points of interest (cities) in the salesman' tour, where the edges between them are provided with weights that express the cost of using the corresponding route. To solve the problem, three different approaches are generally followed, which are:

- Exact algorithms that calculate the optimal path but can only be applied to small-scale problems,

- Approximation algorithms that compute solutions that are definitely not optimal but are applied to larger scale problems in a tolerable execution time,

- Specialized algorithms concerning problems whose graph has some special features.

In the context of this thesis, an overview has been made of the corresponding theory behind the Traveling Salesman Problem and some of the approaches that exist today to solve it using algorithms. A web application was also developed that would solve the problem using real geographical data drawn from an online map, OpenStreetMap. The application allows the user to select N points of interest on the map from which the distance matrix is initially created and then an algorithm selected by the user is applied to solve the problem. After the execution of the algorithm finishes, the app displays the results.

Ευχαριστίες

Θα ήθελα να πω ένα μεγάλο ευχαριστώ στον κ. Ευστάθιο Αντωνίου, τον υπεύθυνο καθηγητή της παρούσας εργασίας, για την υπομονή και κατανόησή του καθώς και για την πολύτιμη βοήθεια και καθοδήγηση που μου παρείχε καθ' όλη την διάρκεια εκπόνησης.

Επίσης, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου για την στήριξή τους όλα αυτά τα χρόνια των σπουδών μου, πόσο μάλλον τον τελευταίο καιρό για την περάτωση της εργασίας αυτής.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Συντομογραφίες

Π.Ε.	Πτυχιακή Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
TSP	Traveling Salesman Problem
MST	Minimum Spanning Tree

Κεφάλαιο 1ο: Εισαγωγή

1.1 Η ιστορία του προβλήματος

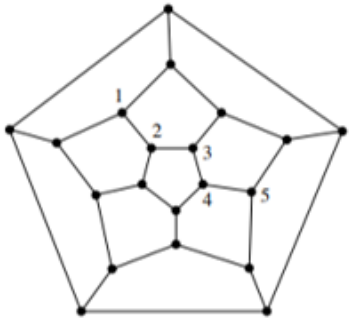
Αν και η ακριβής προέλευση του προβλήματος του πλανόδιου πωλητή είναι ασαφής, το πρώτο παράδειγμα τέτοιου προβλήματος εμφανίστηκε στο γερμανικό εγχειρίδιο Der Handlungsreisende - Von einem alten Commis - Voyageur για έναν πωλητή που ταξίδευε μέσω της Γερμανίας και της Ελβετίας το 1832, όπως εξηγείται από τον Cook [1]. Αυτό το εγχειρίδιο ήταν απλώς ένας οδηγός, αφού δεν περιείχε καμία μαθηματική γλώσσα. Οι άνθρωποι άρχισαν να συνειδητοποιούν ότι δεν πρέπει να παραβλέπεται η δυνατότητα εξοικονόμησης χρόνου και κόστους από τη δημιουργία βέλτιστων μονοπατιών και έτσι υπάρχει ένα πλεονέκτημα για τον υπολογισμό τέτοιων βέλτιστων διαδρομών.

Αυτή η ιδέα μετατράπηκε σε παιχνίδι κάποια στιγμή κατά τη διάρκεια του 1800 από τους W. R. Hamilton και T. Kirkman. Το Icosian Game του Hamilton ήταν ένα ψυχαγωγικό παζλ που βασίστηκε στην εύρεση ενός κύκλου Hamilton στο γράφημα των Δωδεκάεδρων, όπως φαίνεται παρακάτω (Σχήμα 1). Ένα άτομο θα έβαζε τα αριθμημένα σημεία, από το 1 έως το 5, σε γειτονικές κορυφές στο γράφημα των δωδεκαέδρων. Ένα δεύτερο άτομο θα χρησιμοποιούσε τους υπόλοιπους αριθμούς, 6 έως 20, σε γειτονικές κορυφές μετά την κορυφή με το 5 για να δημιουργηθεί ένας Χαμιλτονιανός Κύκλος που θα τελειώνει στο πρώτο σημείο με την ένδειξη «1». Είναι η περίπτωση που κάθε αρχική θέση έχει μια λύση. Το παιχνίδι δεν είχε μεγάλη επιτυχία, γιατί τα παιδιά παραπονέθηκαν ότι ήταν πολύ εύκολο [1].

Το πρόβλημα του ταξιδιώτη πωλητή, όπως το γνωρίζουμε (ή TSP- Traveler Salesman Problem) μελετήθηκε για πρώτη φορά τη δεκαετία του 1930 στη Βιέννη και το Harvard. Ο Richard M. Karp έδειξε το 1972 ότι το πρόβλημα του Χαμιλτονιανού κύκλου ήταν NP-complete, που υποδηλώνει τη NP-δυσκολία του TSP Αυτό παρέχει μια μαθηματική εξήγηση για τη φαινομενική υπολογιστική δυσκολία εύρεσης βέλτιστων περιηγήσεων [1].

Το τρέχον ρεκόρ για το μεγαλύτερο Πρόβλημα Βέλτιστων Περιηγήσεων Περιπλανώμενων Πωλητών, συμπεριλαμβανομένων 85.900 πόλεων, λύθηκε το 2006. Οι υπολογιστές χρησιμοποιούσαν εκδόσεις της μεθόδου διακλάδωσης και δέσμευσης καθώς και των επιπέδων κοπής (δύο φαινομενικά στοιχειώδεις μέθοδοι γραμμικού προγραμματισμού ακεραίων). Ο κώδικας που χρησιμοποιείται σε αυτές τις λύσεις ονομάζεται Concorde και είναι διαθέσιμος για δωρεάν προβολή μέσω Διαδικτύου [1].

Θα ήθελε κανείς να πιστεύει ότι το πρόβλημα του περιπλανώμενου πωλητή μπορεί πάντα να λυθεί από έναν υπολογιστή. Ο μέγιστος αριθμός Χαμιλτονιανών Κύκλων σε ένα δεδομένο γράφημα με τρεις ή περισσότερες κορυφές είναι $(n-1)!/2$. Ως εκ τούτου, θεωρητικά ένας υπολογιστής θα μπορούσε να εξετάσει καθεμία από αυτές τις πιθανές περιηγήσεις. Το πρόβλημα εδώ είναι ότι όσο το n μεγαλώνει, μεγαλώνει και το $(n-1)!/2$, το οποίο γίνεται εξαιρετικά μεγάλο, όπως μπορούμε να δούμε στον Πίνακα 1 [1].



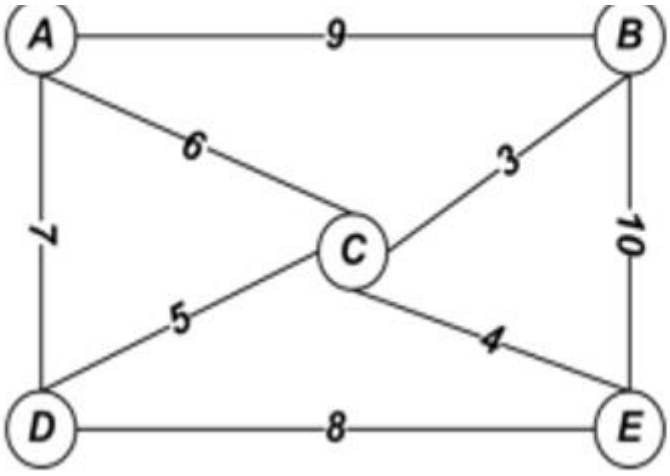
Σχήμα 1.1: Γράφημα Δωδεκάεδρου [1]

Πίνακας 1.1: Αύξηση του (n-1)!/2, καθώς αυξάνεται το n [1]

n	(n-1)!/2
3	1
4	3
5	12
6	60
7	360
8	2,520
9	20,160
10	181,440
20	6.08226E+16
30	4.42088E+30
40	1.01989E+46

Καθώς αυτοί οι αριθμοί μεγαλώνουν, γίνεται σαφές ότι ένας υπολογιστής δεν θα είναι σε θέση να λύσει τέτοια προβλήματα σε ένα εύλογο χρονικό διάστημα. Η επίλυση του προβλήματος του πλανόδιου πωλητή σημαίνει την εύρεση ενός αλγόριθμου που θα λύνει το πρόβλημα με αποδεκτό αριθμό των υπολογισμών [1].

Στη συνηθισμένη μορφή του TSP, ένας χάρτης με πόλεις δίνεται στον πωλητή ο οποίος πρέπει να επισκεφθεί όλες τις πόλεις μόνο μία φορά για να ολοκληρώσει μια περιήγηση έτσι ώστε η διάρκεια της περιήγησης να είναι η μικρότερη από όλες τις δυνατές περιηγήσεις για αυτόν τον χάρτη. Τα δεδομένα αποτελούνται από κόστη που αποδίδονται στις άκρες ενός πεπερασμένου πλήρους γραφήματος και ο στόχος είναι να βρεθεί ένας Χαμιλτονιανός κύκλος που διέρχεται από όλες τις κορυφές, του γραφήματος, έχοντας διανύσει το ελάχιστο συνολικό κόστος. Στο πλαίσιο του TSP, οι κύκλοι Hamilton ονομάζονται περιηγήσεις. Για παράδειγμα, δεδομένου του χάρτη που φαίνεται στο Σχήμα 2, η διαδρομή με το χαμηλότερο κόστος θα ήταν η (A,B,C,E,D,A), με κόστος 31 [2].



Σχήμα 1.2: Η περιήγηση με A=>B =>C =>E =>D => A είναι η βέλτιστη περιήγηση [2]

1.2 Περιγραφή του προβλήματος και ορισμοί

Στα μέσα της δεκαετίας του 1930 η επιστήμη των υπολογιστών δεν ήταν ακόμη ένας καλά καθορισμένος ακαδημαϊκός κλάδος. Στην πραγματικότητα, θεμελιώδεις έννοιες, όπως «αλγόριθμος» ή «υπολογιστικό πρόβλημα», μόλις είχαν εμφανιστεί. Σε αυτά τα χρόνια ο Αυστριακός μαθηματικός Karl Menger κάλεσε την ερευνητική κοινότητα να εξετάσουν από μαθηματική άποψη το παρακάτω πρόβλημα που μπορεί να συναντηθεί στην καθημερινή ζωή. Ένας ταξιδιώτης πωλητής πρέπει να επισκεφτεί ακριβώς μία φορά κάθε μία από μια λίστα με n πόλεις και μετά να επιστρέψει στη βάση του. Γνωρίζει το κόστος του ταξιδιού από οποιαδήποτε πόλη i σε οποιαδήποτε άλλη πόλη j. Επομένως, ποια είναι η περιήγηση με το μικρότερο δυνατό κόστος που μπορεί να κάνει ο πωλητής; [3].

Ο περιπλανώμενος πωλητής πρέπει να επισκεφτεί όλους τους πελάτες του, να επιστρέψει στο σπίτι έχοντας διανύσει τον συντομότερο δρόμο. Οι επισκέψεις αυτές σε όλες τις κορυφές του γραφήματος μπορούν να μοντελοποιηθούν ως ένα μονοπάτι Hamilton. Ένα μονοπάτι σε ένα γράφημα G είναι ένα μονοπάτι Hamilton στο γράφημα G που περιέχει όλα τις κορυφές του γραφήματος G. Αυτός είναι ακόμη και ο ορισμός του Κύκλου Χάμιλτον, γιατί και οι δύο είναι ειδικές περιπτώσεις του μονοπατιού του Χάμιλτον. Το γράφημα G είναι Γράφημα του Χάμιλτον, αν στο γράφημα G υπάρχει κύκλος του Χάμιλτον. Ο στόχος έχει βαθμολογηθεί η ακμή που ενώνει τα σημεία για να βρεθεί ο συντομότερος κύκλος του Χάμιλτον το συντομότερο μονοπάτι του Hamilton (Σχήμα 1) [2].

Αλγόριθμος: είναι μια μηχανική διαδικασία που αποτελείται από στοιχειώδη βήματα και για οποιοδήποτε είσοδο παράγει μια έξοδο.

Ο αλγόριθμος μπορεί να περιγράφει ως:

• λεκτική περιγραφή στη φυσική γλώσσα

• πρόγραμμα υπολογιστή σε οποιαδήποτε γλώσσα προγραμματισμού

• κύκλωμα υλικού κ.λπ.

Ο χρόνος υπολογισμού: Αν και οι υπολογιστές μπορούν να υπολογίσουν πολύ γρήγορα (δισεκατομμύρια λειτουργίες ανά δευτερόλεπτο), η υπολογιστική τους ικανότητα είναι πεπερασμένη. Για να εκτελεστεί οποιαδήποτε λειτουργία χρειάζεται κάποιο μη μηδενικό χρόνο. Ο υπολογισμός για την ίδια είσοδο μπορεί να κοστίσει διαφορετική χρονική κατανάλωση για διαφορετικούς αλγόριθμους.

Αυτό μπορεί να εξαρτάται από:

• την υλοποίηση αλγορίθμου

• τη γλώσσα προγραμματισμού

• τον χρησιμοποιημένο διερμηνέα

• το χρησιμοποιημένο υλικό (επεξεργαστής, μνήμη κ.λπ.).

Επίλυση Υπολογιστικού Προβλήματος: είναι η διαδικασία εύρεσης ενός αλγορίθμου τέτοιου που για κάθε είσοδο $x \in I$, να προκύπτει τέτοια έξοδος $y \in 0$, ώστε $(x, y) \in R$. Γίνεται φανερό ότι αν δεν υπάρχει y ώστε $(x, y) \in R$, ο αλγόριθμος θα πρέπει να ενημερώνει για την αποτυχία επίλυσης με επιστρεφόμενο μήνυμα. Έστω και αν για κάθε είσοδο να προκύπτει μόνο μία έξοδος (και όχι δεύτερη), η R εξακολουθεί να είναι μια συνάρτηση από το I στο O.

Τα υπολογιστικά προβλήματα διακρίνονται βασικά σε επιλύσιμα και μη επιλύσιμα, ενώ τα επιλύσιμα διακρίνονται περεταίρω σε ευεπίλυτα και δυσεπίλυτα, ανάλογα τις απαιτήσεις τους σε υπολογιστικούς πόρους. Μία ακόμα διάκριση, βάσει ζητούμενου, μπορεί να γίνει σε προβλήματα βελτιστοποίησης (αναζήτηση της βέλτιστης λύσης) και σε προβλήματα απόφασης, ανάλογα αν στο πρόβλημα αναζητείται μία «ΝΑΙ» ή «ΟΧΙ» απόφαση.

Για την αλγοριθμική επίλυση ενός υπολογιστικού προβλήματος, η πορεία μετατροπών αναμένεται να είναι :

Πρόβλημα Βελτιστοποίησης ⇔ Πρόβλημα Απόφασης → Τυπική γλώσσα με κωδικοποίηση.

Σχήμα 1.3: Αλγοριθμική επίλυση ενός υπολογιστικού προβλήματος

Συνεπώς για κάθε πρόβλημα βελτιστοποίησης, στο οποίο αναζητείται η βέλτιστη λύση, αντιστοιχεί με ισοδυναμία ένα πρόβλημα απόφασης [4].

Για να επιστρέψουμε στον ορισμό του αλγόριθμου, ο αλγόριθμος αποτελεί τη μέθοδο επίλυσης ενός προβλήματος σε μία υπολογιστική μηχανή, ή μία μηχανή Turing. Οι μηχανές Turing αναφέρονται ως αιτιοκρατικές μηχανές Turing (Deterministic Turing Machines) και μη αιτιοκρατικές μηχανές Turing (Non Deterministic Turing Machines). Μια μηχανή Turing είναι μια θεωρητική μηχανή που χειρίζεται σύμβολα σε μια λωρίδα ταινίας, με βάση έναν πίνακα κανόνων. Παρόλο που η μηχανή Turing είναι απλή, μπορεί να προσαρμοστεί ώστε να αναπαράγει τη λογική που σχετίζεται με οποιονδήποτε αλγόριθμο υπολογιστή. Η διαφορά των μη αιτιοκρατικών μηχανών Turing είναι η ύπαρξη πολλαπλών επιλογών συμπεριφοράς [4].

Αν η επίλυση ενός υπολογιστικού προβλήματος έχει ως στόχο την διερεύνηση της δυνατότητας επίλυσης ενός προβλήματος από έναν ηλεκτρονικό υπολογιστή, η θεωρία της πολυπλοκότητας εξετάζει πόσο εύκολα ή δύσκολα προς την επίλυσή τους είναι τα υπολογιστικά προβλήματα, αν ληφθούν υπόψη οι πόροι του υπολογιστή (χρόνος και χώρος). Έτσι, η θεωρία της πολυπλοκότητας ορίζει τις κλάσεις πολυπλοκότητας προβλημάτων (complexity classes), στις οποίες κατατάσσονται τα προβλήματα, ανάλογα με την υπολογιστική τους δυσκολία [4].

Κλάση P (Polynomial) είναι η κλάση πολυπλοκότητας που περιλαμβάνει το σύνολο των προβλημάτων που μπορούν να επιλυθούν εντός πολυωνυμικού χρόνου. Συνήθως πρόκειται για ευεπίλυτα προβλήματα.

Η **κλάση NP (Non - Deterministic Polynomial)** περιλαμβάνει εκείνα τα προβλήματα των οποίων η λύση είναι δυνατόν να επαληθευτεί εντός πολυωνυμικού χρόνου, ή εκείνα τα προβλήματα για τα οποία μπορεί να βρεθεί λύση εντός πολυωνυμικού χρόνου, χρησιμοποιώντας μία μη αιτιοκρατική μηχανή Turing.

Στην **κλάση NP-πλήρη (Non – Deterministic Polynomial – Complete)** ανήκουν τα προβλήματα στα οποία, εάν εντοπιστεί πολυωνυμικός αλγόριθμος για ένα από τα προβλήματα της κλάσης, αυτό σημαίνει την ύπαρξη πολυωνυμικών αλγόριθμων για το σύνολο των προβλημάτων της κλάσης NP.

Στην **κλάση NP-hard (Non-Deterministic Polynomial – hard)** εμπεριέχονται τα προβλήματα που μπορούν να θεωρηθούν τουλάχιστον όσο δύσκολα όσο και τα υπόλοιπα της κλάσης NP [5].

Η έκδοση βελτιστοποίησης του προβλήματος του πλανόδιου πωλητή ανήκει στην κλάση των NP-hard προβλημάτων. Σε γενικές γραμμές, είναι ένα πρόβλημα το οποίο η λύση του μπορεί να επαληθευτεί σε πολυωνυμικό χρόνο αλλά οι αλγόριθμοι επίλυσης για αυτό το πρόβλημα δεν είναι πολυωνυμικού χρόνου, ούτε είναι ακόμη γνωστό αν μπορεί να υπάρξει ένας αλγόριθμος πολυωνυμικής πολυπλοκότητας.

Το πρόβλημα του περιπλανώμενου πωλητή μπορεί να χωριστεί σε δύο διαφορετικά είδη προβλημάτων - συμμετρικά και μη συμμετρικά. Στη συμμετρική περίπτωση του περιοδεύοντος πωλητή το πρόβλημα είναι η απόσταση μεταξύ δύο σημείων που είναι ίδια και στις δύο κατευθύνσεις. Η βασική δομή του γραφήματος είναι ένα μη κατευθυνόμενο γράφημα. Στην μη συμμετρική περίπτωση η απόσταση μεταξύ δύο σημείων στις δύο κατευθύνσεις μπορεί να είναι διαφορετική ή μπορεί να υπάρχει απλώς ένας τρόπος προς μία κατεύθυνση από/προς την κορυφή. Αυτή η δομή ονομάζεται κατευθυνόμενο γράφημα. Στο παράδειγμα για το πρόβλημα του ασύμμετρου περιπλανώμενου πωλητή μπορεί κάποιος να αναζητά τη συντομότερη περιήγηση σε μια πόλη με μονόδρομους [6].

Ο πιο συχνά χρησιμοποιούμενος τύπος προβλήματος περιπλανώμενου πωλητή είναι το λεγόμενο Metric-TSP . Σε αυτή την περίπτωση όλες οι αποστάσεις είναι βάρη ακμών σε γράφημα που πληροί τριγωνική ανισότητα. Αυτός ο τύπος προβλήματος συμπίπτει με τα περισσότερα πραγματικά προβλήματα. Αυτή η περίπτωση επιτρέπει την κατασκευή κατά προσέγγιση αλγορίθμων NP [6].

Αυτό το πρόβλημα μπορεί να διατυπωθεί ως εξής:

Υπάρχουν m σημεία και τρόποι μετάβασης μεταξύ όλων αυτών με γνωστά μήκη (που σημαίνει είναι απλό να βρεθεί ο συντομότερος δρόμος μεταξύ οποιωνδήποτε δύο σημείων). Ο στόχος είναι να βρεθεί ο συντομότερος δρόμος, ο οποίος περνάει όλα τα σημεία μία φορά, αρχίζει και τελειώνει στο ίδιο σημείο. Αυτό σημαίνει ότι ο στόχος είναι να βρεθεί το συντομότερο δυνατό ταξίδι μετ' επιστροφής.

Το πραγματικό πρόβλημα δεν είναι να αναζητηθεί το συντομότερο ταξίδι μετ' επιστροφής. Το πρόβλημα είναι σε ένα πεπερασμένο σύνολο πόλεων, να βρεθεί η βέλτιστη δυνατή διαδρομή, η οποία θα διέρχεται από όλες τις πόλεις ακριβώς μια φορά και θα καταλήγει στην πόλη αφετηρία, ελαχιστοποιώντας το συνολικό κόστος της διαδρομής. Σε αυτή την περίπτωση είναι απαραίτητο να δοκιμαστούν όλοι οι πιθανοί συνδυασμοί μεταξύ όλων των σημείων, το οποίο είναι ακριβώς το πρόβλημα. Σε μια περίπτωση λίγων σημείων (π.χ. τέσσερα ή πέντε) το πρόβλημα δεν είναι τόσο περίπλοκο. Αν και ο καθένας μπορεί να το λύσει με ένα στυλό και ένα χαρτί, κάποιος ικανός μαθηματικός μπορεί να λύσει αυτό το πρόβλημα ακόμα και χωρίς εργαλεία και βοήθεια. Σε περίπτωση περισσότερων σημείων (π.χ. οκτώ ή εννέα) είναι δυνατό να χρησιμοποιηθεί κάποιος απλός αλγόριθμος ακόμα και αυτός με δοκιμή όλων των δυνατών λύσεων. Αλλά τι γίνεται αν υπάρχει περίπτωση όπου υπάρχει η ανάγκη να λυθεί ο περιπλανώμενος πωλητής με χιλιάδες ή δεκάδες χιλιάδες σημεία; Είναι

δυνατόν με αυτό τον αριθμό σημείων για επίλυση σε πραγματικό χρόνο, ο υπολογιστής να πρέπει να υπολογίζει για πολλά χρόνια;

Τι γίνεται όταν θα πρέπει επίσης να υπολογιστεί το κόστος του ταξιδιού; Πιο τυπικά, ένα στιγμιότυπο του TSP δίνεται από ένα πλήρες γράφημα G σε ένα σύνολο κόμβων $V = \{1,2, \dots m\}$, για κάποιο ακέραιο m, και με μια συνάρτηση κόστους που εκχωρεί ένα κόστος cij στο τόξο (i, j) , για οποιοδήποτε i, j στο V. Το TSP είναι αντιπρόσωπος μιας μεγάλης κατηγορίας προβλημάτων γνωστών ως συνδυαστικά προβλήματα βελτιστοποίησης. Μεταξύ αυτών, το TSP είναι ένα από τα πιο σημαντικά, αφού είναι πολύ εύκολο να περιγραφεί, αλλά πολύ δύσκολο να λυθεί. Στην πραγματικότητα, το TSP ανήκει στην κατηγορία NP-hard. Ως εκ τούτου, ένας αποτελεσματικός αλγόριθμος για το TSP (δηλαδή, ένας αλγόριθμος που υπολογίζει, για οποιαδήποτε περίπτωση TSP με m κόμβους, την περιήγηση στο ελάχιστο δυνατό κόστος σε πολυωνυμικό χρόνο ως προς το m) μάλλον δεν υπάρχει. Πιο συγκεκριμένα, ένας τέτοιος αλγόριθμος υπάρχει εάν και μόνο εάν οι δύο υπολογιστικές κλάσεις P και NP συμπίπτουν, δηλαδή πολύ απίθανη υπόθεση, σύμφωνα με τις εξελίξεις των τελευταίων ετών [7].

Από πρακτικής άποψης, σημαίνει ότι είναι αρκετά απίθανο να βρεθεί ακριβής αλγόριθμος για οποιοδήποτε στιγμιότυπο TSP με m κόμβους, για μεγάλο m, που έχει αρκετά καλύτερη συμπεριφορά από τον αλγόριθμο που υπολογίζει οποιαδήποτε από τις $(m - 1)!$ πιθανές ξεχωριστές περιηγήσεις και στη συνέχεια υπολογίζει την λιγότερο δαπανηρή [8].

Αν ψάχνουμε για εφαρμογές, μπορεί να χρησιμοποιηθεί διαφορετική προσέγγιση. Για παράδειγμα δόθηκε ένα TSP με m κόμβους, και οποιαδήποτε περιήγηση που περνάει μια φορά από οποιαδήποτε πόλη είναι μια εφικτή λύση, και το κόστος οδηγεί σε ένα ανώτερο φράγμα στο ελάχιστο δυνατό κόστος. Αλγόριθμοι που εκτελούνται σε πολυωνυμικό χρόνο σε σχέση με m εφικτές λύσεις και υπολογίζουν άνω όρια για τη βέλτιστη τιμή, ονομάζονται ευρετικοί. Γενικά, αυτοί οι αλγόριθμοι παράγουν λύσεις αλλά χωρίς καμία εγγύηση ποιότητας για το πόσο απέχει το υπολογιζόμενο κόστος τους από το ελάχιστο δυνατό. Αν μπορεί να αποδειχθεί ότι το κόστος της επιστρεφόμενης λύσης είναι πάντα μικρότερο από k φορές το ελάχιστο δυνατό κόστος, για κάποιο πραγματικό αριθμό $k > 1$, η ευρετική μέθοδος ονομάζεται αλγόριθμος προσέγγισης k. Δυστυχώς, ο προσεγγιστικός αλγόριθμος k για το TSP δεν είναι γνωστός, για οποιοδήποτε $k > 1$ [7].

Επιπλέον, σε μια εργασία που εμφανίστηκε το 2000, οι Papadimitriou et al. έδειξαν ότι ένας προσεγγιστικός αλγόριθμος για το TSP για οποιοδήποτε 97/96>k>1 υπάρχει εάν και μόνο εάν $P = NP$. Ως εκ τούτου, η εύρεση μιας καλής ευρετικής για το TSP φαίνεται επίσης πολύ δύσκολη [9].

Γενικά, τα προβλήματα για τα οποία υπάρχουν αποδοτικοί αλγόριθμοι είναι γνωστά ως κλάση P, για πολυωνυμικό χρόνο. Με τον όρο αποδοτικός αλγόριθμος εννοούμε ότι για μια δεδομένη είσοδο υπάρχουν το πολύ $O\ (nk)$ βήματα όπου n είναι η πολυπλοκότητα της εισόδου και k είναι ένας μη αρνητικός ακέραιος. Ένα πρόβλημα είναι στο NP εάν όποτε η απάντηση σε μια απόφαση είναι ναι, τότε υπάρχει τρόπος να πιστοποιηθεί αυτή η θετική απάντηση με τέτοιο τρόπο ώστε η λύση να μπορεί να ελεγχθεί σε πολυωνυμικό χρόνο.

Χρησιμοποιώντας αυτήν την ορολογία μπορούμε να εξηγήσουμε τη δυσκολία του Προβλήματος του Περιπλανώμενου Πωλητή. Αναφέρθηκε ότι το TSP εμπίπτει στη γενική κατηγορία του προβλήματος NP. Το TSP μπορεί να λυθεί. Ωστόσο, χρειάζεται n! δυνατές περιηγήσεις για έλεγχο. Αυτό συμβαίνει γιατί όταν κάνουμε μια περιήγηση ουσιαστικά βρίσκουμε μια σειρά από πόλεις και υπάρχουν n! τρόποι ταξινόμησης n αντικειμένων (n μεταθέσεις). Υπάρχουν πολλές μέθοδοι στις οποίες δεν ελέγχονται όλες οι πιθανότητες, εξοικονομώντας πολύ χρόνο, αλλά κάθε μέθοδος έχει διαφορετικά αποτελέσματα στην ακρίβεια της περιήγησης. Ένα σενάριο χειρότερης περίπτωσης ορίζει μαθηματικά το απόλυτο χειρότερο

αλγόριθμο που μπορεί να εκτελέσει. Αυτοί περιγράφονται ως η αναλογία της χειρότερης πιθανότητας προς τη βέλτιστη λύση. Με άλλα λόγια τα χειρότερα σενάρια μπορούν να βοηθήσουν στη μέτρηση της ακρίβειας, ενώ η σημειογραφία Big O βοηθά στη μέτρηση της ταχύτητας. Έτσι, οι υπάρχουσες μέθοδοι επίλυσης του TSP ενσωματώνουν τον ανταγωνισμό μεταξύ ακρίβειας και ταχύτητας [10].

Αρχικά, πρέπει να αποδείξουμε ότι το TSP ανήκει στα NP. Αν θέλουμε να ελέγξουμε μια περιήγηση για αξιοπιστία, ελέγχουμε ότι η περιήγηση περιέχει κάθε κορυφή μία φορά. Στη συνέχεια αθροίζουμε το συνολικό κόστος των διαδρομών και τέλος ελέγχουμε αν το κόστος είναι ελάχιστο. Αυτό μπορεί να ολοκληρωθεί σε πολυωνυμικό χρόνο, έτσι το TSP ανήκει στο NP.

Κεφάλαιο 2ο: Μέθοδοι Λύσης

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα αναφερθούμε στις προσεγγίσεις που έχουν επινοηθεί τα τελευταία χρόνια για την επίλυση του προβλήματος του πλανόδιου πωλητή χρησιμοποιώντας αλγορίθμους. Θα αναφερθούμε σε δύο κατηγορίες αλγορίθμων. Η πρώτη είναι προσεγγιστικοί αλγόριθμοι οι οποίοι υπολογίζουν λύσεις που δεν είναι οι καλύτερες αλλά εφαρμόζονται σε προβλήματα μεγαλύτερης κλίμακας σε ανεκτό χρόνο. Αυτοί στους οποίους θα αναφερθούμε είναι ο αλγόριθμος του Χριστοφίδη και ο αλγόριθμος Πλησιέστερου Γείτονα. Η δεύτερη κατηγορία είναι ακριβείς αλγόριθμοι που βρίσκουν τη βέλτιστη διαδρομή αλλά μπορούν να εφαρμοστούν μόνο σε μικρής έκτασης προβλήματα λόγω της υψηλής πολυπλοκότητας τους και των περιορισμένων υπολογιστικών πόρων ενός υπολογιστή. Θα αναλύσουμε τον αλγόριθμο Held-Karp και την προσέγγιση brute-force.

2.2 Εκτίμηση φραγμάτων

Προτού παρουσιάσουμε τους αλγορίθμους, πρέπει να αναφερθούμε πρώτα στην εκτίμηση των φραγμάτων όταν χρησιμοποιούμε προσεγγιστικούς αλγορίθμους. Ο υπολογισμός άνω και κάτω φραγμάτων στο TSP βοηθάει να προσδιορίσουμε το εύρος στο οποίο κυμαίνεται το μήκος της βέλτιστης διαδρομής του προβλήματος. Όσο μικρότερη είναι η τιμή του άνω φράγματος τόσο πιο ξεκάθαρη εικόνα θα έχουμε για την βέλτιστη λύση. Αντιστοίχως, θέλουμε να είναι και η τιμή του κάτω φράγματος να είναι όσο μεγαλύτερη γίνεται [11]. Όλη αυτή η διαδικασία πραγματοποιείται όταν χρησιμοποιούμε αλγορίθμους οι οποίοι βρίσκουν μια κατά προσέγγιση λύση. Αυτό το κάνουμε για να έχουμε ένα σημείο αναφοράς ώστε να αξιολογήσουμε τα αποτελέσματα των αλγορίθμων αυτών. Ακόμα και για τους καλύτερους αλγορίθμους που υπάρχουν, ο χρόνος εκτέλεσης τους αυξάνεται εκθετικά με το πλήθος των πόλεων του προβλήματος. Συνεπώς, ακόμα και μεσαίου μεγέθους προβλήματα δεν μπορούν να λυθούν σε ανεκτό χρόνο με τέτοιους αλγορίθμους [12].

2.2.1 Υπολογισμός άνω φράγματος

Το άνω φράγμα μπορεί να υπολογιστεί με τον αλγόριθμο του Πλησιέστερου Γείτονα. Δεδομένου ενός πίνακα αποστάσεων (πίνακας 2.1) των κορυφών του γραφήματός του TSP (σχήμα 2.1), εφαρμόζουμε τον αλγόριθμο επαναληπτικά για κάθε κορυφή ως αφετηρία. Σύμφωνα με τον αλγόριθμο, μεταβαίνουμε στην κοντινότερη κορυφή που δεν έχουμε επισκεφτεί έως ότου τις επισκεφτούμε όλες από μία φορά και τέλος επιστρέφουμε στην αφετηρία. Έτσι, βρίσκουμε τις εξής διαδρομές:

Αφετηρία κορυφή A: Διαδρομή A-B-G-F-D-E-H-C-A, μήκος 71

Αφετηρία κορυφή B: Διαδρομή B-A-G-F-D-E-H-C-A, μήκος 72

Αφετηρία κορυφή C: Διαδρομή C-E-H-D-F-G-B-A-C, μήκος 69

Αφετηρία κορυφή D: Διαδρομή D-E-H-C-G-F-B-A-D, μήκος 71

Αφετηρία κορυφή E: Διαδρομή E-H-D-C-G-F-B-A-E, μήκος 78

Αφετηρία κορυφή F: Διαδρομή F-G-B-A-D-E-H-C-F, μήκος 71

Αφετηρία κορυφή G: Διαδρομή-1 G-F-B-A-D-E-H-C-G, μήκος 71

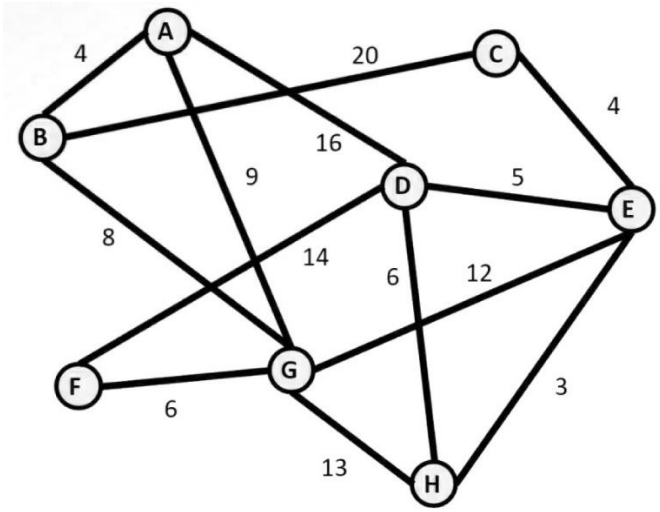
Αφετηρία κορυφή G: Διαδρομή-2 G-F-D-E-H-C-B-A-G, μήκος 68

Αφετηρία κορυφή H: Διαδρομή H-E-C-D-F-G-B-A-H, μήκος 70

Από αυτές θεωρούμε την μικρότερη σε μήκος ως άνω φράγμα του προβλήματος, δηλαδή την διαδρομή G-F-D-E-H-C-B-A-G με αφετηρία την κορυφή G και μήκος 68 [13].

Πίνακας 2.1: Πίνακας αποστάσεων

	A	B	C	D	E	F	G	H
A	-	4	24	16	20	15	9	22
B	4	-	20	20	20	14	8	21
C	24	20	-	9	4	22	16	7
D	16	20	9	-	5	14	17	6
E	20	20	4	5	-	18	12	3
F	15	14	22	14	18	-	6	19
G	9	8	16	17	12	6	-	13
H	22	21	7	6	3	19	13	-



Σχήμα 2.1: Γράφημα προβλήματος

2.2.2 Υπολογισμός κάτω φράγματος

Δεδομένου ότι ένα μονοπάτι Hamilton με μία κορυφή αφαιρεμένη θα είναι οπωσδήποτε ένα συνδετικό δέντρο των υπολειπόμενων κορυφών, είναι επακόλουθο ότι το μήκος του θα είναι ίσο ή μεγαλύτερο με αυτό του ελάχιστου συνδετικού δέντρου των κορυφών αυτών. Ο καλύτερος τρόπος σύνδεσης αυτού του δέντρου στην αφαιρεμένη κορυφή είναι με τις δύο μικρότερες ακμές. Το κάτω φράγμα μπορεί να υπολογιστεί αφαιρώντας μία κορυφή του γραφήματος και βρίσκοντας ένα ελάχιστο συνδετικό δέντρο (MST) με τον αλγόριθμο του Prim ή του Kruskal. Αθροίζουμε το μήκος του δέντρου αυτού με τα μήκη των δύο μικρότερων ακμών που συνδέονται στην αφαιρεμένη κορυφή. Επαναλαμβάνουμε την διαδικασία αυτή για κάθε κορυφή του προβλήματος και από τα αθροίσματα που βρούμε θεωρούμε το

μεγαλύτερο από αυτά το κάτω φράγμα [11]. Αν εφαρμόσουμε την διαδικασία αυτή στο γράφημα του σχήματος 4, βρίσκουμε τα παρακάτω:

Αφαίρεση κορυφής A: MST=38, άθροισμα: 38+4+9=51

Αφαίρεση κορυφής B: MST=39, άθροισμα: 39+4+8=51

Αφαίρεση κορυφής C: MST=38, άθροισμα: 38+4+20=62

Αφαίρεση κορυφής D: MST=37, άθροισμα: 37+5+6=48

Αφαίρεση κορυφής E: MST=57, άθροισμα: 57+4+3=64

Αφαίρεση κορυφής F: MST=36, άθροισμα: 36+14+6=56

Αφαίρεση κορυφής G: MST=46, άθροισμα: 46+6+8=60

Αφαίρεση κορυφής H: MST=39, άθροισμα: 39+6+3=48

Από αυτά θεωρούμε το μεγαλύτερο άθροισμα ως το κάτω φράγμα του προβλήματος [14].

2.3 Ακριβείς αλγόριθμοι

2.3.1 Αλγόριθμος Held-Karp

Ο αλγόριθμος Held-Karp, γνωστός και ως Bellman-Held-Karp, είναι ένας αλγόριθμος δυναμικού προγραμματισμού διατυπωμένος το 1962 ανεξάρτητα από τον Richard E. Bellman και από τους Michael Held και Richard M. Karp για την επίλυση του TSP. Ο αλγόριθμος δέχεται ως είσοδο το πίνακα των αποστάσεων μεταξύ ενός συνόλου πόλεων με απώτερο σκοπό την εύρεση μιας διαδρομής ελάχιστης απόστασης η οποία να διέρχεται από κάθε πόλη από μία φορά και να επιστρέφει στο σημείο εκκίνησης. Βρίσκει την ακριβή λύση στο πρόβλημα αυτό σε εκθετικό χρόνο [15]. Η λογική του δυναμικού προγραμματισμού είναι να αναγνωρίσουμε τα υποπροβλήματα του δοθέντος προβλήματος, να επιλύσουμε το καθένα από αυτά, δουλεύοντας από το μικρότερο στο μεγαλύτερο και να συνάγουμε την τελική λύση από αυτές των υποπροβλημάτων. Διαπιστώνουμε λοιπόν ότι ο συνολικός χρόνος εκτέλεσης του αλγορίθμου που βασίζεται στην λογική αυτή προκύπτει από το πλήθος των υποπροβλημάτων πολλαπλασιαζόμενο με τον χρόνο για το κάθε υποπρόβλημα και αθροίζοντας το γινόμενο με τον χρόνο επεξεργασίας των αποτελεσμάτων για την εύρεση της τελικής λύσης. Αυτό μαθηματικά εκφράζεται ως [16]:

$$f(n) \times g(n) + h(n)$$

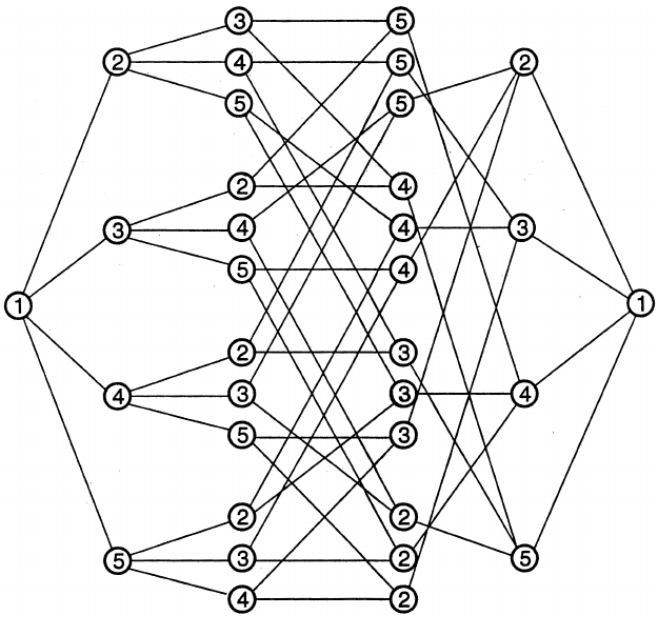
Οι Held και Karp το 1962 παρουσίασαν την προσέγγιση τους βασιζόμενη στον δυναμικό προγραμματισμό για την επίλυση του TSP, ένα πρόβλημα βελτιστοποίησης το οποίο ορίζεται ως εξής: Έχουμε ένα σύνολο πόλεων$\{1, \dots, J\}$, όπου J το πλήθος των πόλεων και για κάθε ζευγάρι πόλεων j, j' το κόστος $d_{jj'} > 0$ μετάβασης από την πόλη j στην πόλη j'. Αναζητούμε την συντομότερη διαδρομή, ξεκινώντας από την πόλη 1 και επιστρέφοντας στο τέλος σε αυτήν, η οποία να επισκέπτεται όλες τις πόλεις του συνόλου ακριβώς μία φορά. Ο αλγόριθμος με χρήση δυναμικού προγραμματισμού έχει πολυπλοκότητα ή αλλιώς λέμε ότι λύνει το πρόβλημα σε χρόνο $O(J^2 \cdot 2^J)$, όπου J είναι το πλήθος των πόλεων. Με χρήση αναδρομής υπολογίζουμε την τιμή του $D(C, j)$, το οποίο αντιπροσωπεύει τα κόστη της διαδρομής με αφετηρία την πόλη 1, κατάληξη την πόλη j και επισκέπτωντας όλες τις πόλεις στο C, όπου C το σύνολο των πόλεων.

Table 1
DP-based algorithm for solving traveling-salesman problems due to Held and Karp. The outermost loop is over the cardinality of subsets of already visited cities.

input: cities $j = 1, \ldots, J$ with distance matrix $d_{jj'}$
initialization: $D(\{k\}, k) := d_{1k}$
for each path length $c = 2, \ldots, J$ do
    for each pair $(\mathcal{C}, j)$, where $\mathcal{C} \subseteq \{2, \ldots, J\}$ and $j \in \mathcal{C}$ and $|\mathcal{C}| = c$ do
        $D(\mathcal{C}, j) = \min_{j' \in \mathcal{C} \setminus \{j\}} \{d_{jj'} + D(\mathcal{C} \setminus \{j\}, j')\}$
traceback:
- find shortest tour: $D^* = \min_{k \in \{2,\ldots,J\}} [D(\{2, \ldots, J\}, k) + d_{k1}]$
- recover optimal sequence of cities

Σχήμα 2.2: Αλγόριθμος Held-Karp δυναμικού προγραμματισμού

Ο αλγόριθμος απεικονίζεται στο σχήμα 2.2. Στην αρχικοποίηση (initialization) θέτουμε τα κόστη από την αφετηρία πόλη 1, τα οποία προκύπτουν από την σχέση: $D(\{k\}, k) = d_{1k}$, για κάθε $k \in \{2, \ldots, |C|\}$. Επεξεργαζόμαστε υποσύνολα πόλεων αυξανόμενου πληθάριθμου c. Στο τέλος του αλγορίθμου βρίσκουμε το κόστος (δηλαδή την απόσταση) της βέλτιστης διαδρομής και ποια είναι αυτή, δηλαδή την σειρά επίσκεψης, χρησιμοποιώντας έναν πίνακα στον οποίο αποθηκεύεται η άριστη λύση για κάθε (C, j). Να σημειώσουμε ότι κατά τους υπολογισμούς που γίνονται, για ένα ζευγάρι (C, j), η σειρά με την οποία έγινε επίσκεψη στις πόλεις του συνόλου C μπορεί να παραβλεφθεί (εκτός της πόλης j) καθώς μόνο τα κόστη της καλύτερης διαδρομής προς την πόλη j χρειαζόμαστε να αποθηκεύσουμε. Στο σχήμα 2.3 απεικονίζεται ένα "supergraph" αναζητήσεων που κάνει ο αλγόριθμος για ένα TSP με $J = 5$ πόλεις. Αξίζει να σημειωθεί ότι από όλους τους συνδυασμούς που συγκλίνουν προς έναν κόμβο j, κρατάμε μόνο αυτόν με το μικρότερο κόστος τον οποίο θα χρειαστούμε για επόμενους υπολογισμούς [17].



Σχήμα 2.3: Απεικόνιση αλγορίθμου σε TSP με 5 πόλεις

2.3.2 Προσέγγιση brute-force

Η brute-force προσέγγιση υπολογίζει όλες τις πιθανές διαδρομές, δηλαδή συνδυασμούς σειράς επίσκεψης πόλεων, για τον προσδιορισμό της καλύτερης, σε μήκος, διαδρομής. Αρχικά, υπολογίζουμε το πλήθος όλων των διαδρομών και βρίσκουμε ποιες είναι αυτές. Στην συνέχεια, υπολογίζουμε την απόσταση καθεμιάς και τέλος βρίσκουμε ποια από αυτές είναι η συντομότερη, δηλαδή η βέλτιστη διαδρομή στο πρόβλημα [18]. Διαπιστώνουμε ότι μια διαδρομή σε ένα TSP είναι μια μετάθεση (μαθηματική έννοια) των πόλεων του προβλήματος με την μοναδική προϋπόθεση η πρώτη πόλη να είναι η 0 (αφετηρία) καθεμιάς από αυτές τις μεταθέσεις. Ένας αλγόριθμος τέτοιας προσέγγισης βρίσκει πάντα ακριβή λύση στο πρόβλημα. Δεδομένου ότι ενδιαφερόμαστε αφετηρία να είναι πάντα το 0, ο χρόνος που απαιτεί για να βρει την λύση, προκύπτει από το πλήθος των διαφορετικών μεταθέσεων του προβλήματος το οποίο υπολογίζεται, με n ο αριθμός των πόλεων, ως εξής:

$$n-1! \;=\; 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1)$$

Δηλαδή, η πολυπλοκότητα του αλγορίθμου αυτού για την επίλυση του TSP είναι $O((n-1)!)$. Σε περίπτωση που θέλουμε να βρούμε την βέλτιστή διαδρομή, χωρίς περιορισμούς για την πόλη αφετηρίας, η πολυπλοκότητα του αλγορίθμου είναι $O(n!)$, όπου $n! \;=\; 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.

Χρησιμοποιώντας μια τέτοια προσέγγιση στην επίλυση του TSP είμαστε σίγουροι ότι θα βρούμε την βέλτιστη διαδρομή που να επισκέπτεται όλες τις πόλεις από μία φορά και μάλιστα με έναν εύκολο κατανοητό τρόπο. Το μειονέκτημα όμως του αλγορίθμου αυτού είναι ότι σε προβλήματα με μεγάλο αριθμό πόλεων θα χρειαστεί πολύ παραπάνω χρόνος για τον υπολογισμό της λύσης. Για παράδειγμα ένα πρόβλημα με 20 πόλεις θα απαιτήσει να παράγουμε 121.645.100.408.832.000 διαφορετικές μεταθέσεις. Παρόλο της μεγάλης ταχύτητας των υπολογιστών είναι αδύνατον να παράγουμε τόσες πολλές μεταθέσεις σε ανεκτό χρόνο [19].

2.4 Προσεγγιστικοί αλγόριθμοι

2.4.1 Αλγόριθμος του Χριστοφίδη

Ο αλγόριθμος του Χριστοφίδη, γνωστός και ως αλγόριθμος Christofides–Serdyukov, είναι ένας αλγόριθμος ο οποίος χρησιμοποιείται για την προσεγγιστική επίλυση του μετρικού TSP, δηλαδή οι αποστάσεις των πόλεων είναι συμμετρικές και ικανοποιούν την τριγωνική ανισότητα. Οι λύσεις που βρίσκει είναι πάντα της τάξεως 3/2 από το μήκος της βέλτιστης λύσης. Ανακαλύφθηκε το 1976 ανεξάρτητα από τους Nicos Christofides και Anatoliy I. Serdyukov. Μέχρι σήμερα θεωρείται ο καλύτερος προσεγγιστικός αλγόριθμος πολυωνυμικού χρόνου αν και υπάρχουν περιπτώσεις βελτίωσής του. Τέτοια είναι αυτή των Karlin, Klein και Gharan, οι οποίοι το 2020 παρουσιάσαν έναν πρωτότυπο προσεγγιστικό αλγόριθμο παρόμοιο με τη λογική του αλγόριθμου του Χριστοφίδη, με τον οποίο ισχυρίζονται ότι πέτυχαν καλύτερο λόγο προσέγγισης $1.5 \;-\; 10^{-36}$.

Δεδομένου ενός μετρικού TSP το οποίο αναπαριστάται ως ένα πλήρες γράφημα $G \;=\; (V, w)$ με βάρη στις ακμές τους που ικανοποιούν την τριγωνική ανισότητα, δηλαδή για κάθε τρεις κορυφές u, v και x να ισχύει $w(uv) \;+\; w(vx) \;\geq\; w(ux)$, όπου η συνάρτηση w δίνει το βάρος κάθε ακμής του G [20].

Προτού αναλύσουμε την λειτουργία του αλγορίθμου, πρέπει πρώτα να επεξηγήσουμε κάποιες έννοιες:

Ελάχιστο Συνδετικό Δέντρο (Minimum Spanning Tree): Είναι ένα συνδετικό δέντρο (spanning tree) στο οποίο το άθροισμα των βαρών των ακμών του είναι το ελάχιστο δυνατό. Ένα συνδετικό δέντρο

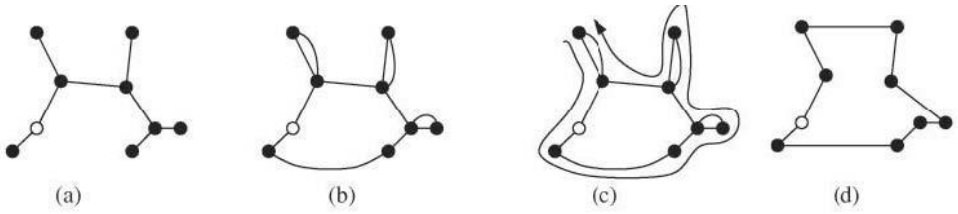
είναι ένας υπογράφος ενός μη-κατευθυνόμενου γραφήματος, το οποίο περιέχει όλες τις κορυφές του γραφήματος με τον ελάχιστο δυνατό αριθμό ακμών που να τις συνδέει [21].

Μονοπάτι Euler (Euler path): Ένα μονοπάτι Euler είναι μια διαδρομή σε ένα γράφημα η οποία διασχίζει κάθε ακμή ακριβώς μία φορά.

Κύκλος Euler (Euler circuit): Είναι ένα μονοπάτι Euler το οποίο έχει ως αφετηρία και τέλος την ίδια κορυφή [22].

Τα βήματα του αλγορίθμου είναι τα εξής:

1. Κατασκευάζουμε ένα ελάχιστο συνδετικό δέντρο M για το γράφημα G (σχήμα 2.4 a).
2. Θεωρούμε σύνολο W με τις κορυφές του γραφήματος οι οποίες είναι περιττού βαθμού στο M και H ένα υπογράφημα του G το οποίο προκύπτει από τις κορυφές του W. Δηλαδή, το H είναι γράφημα με κορυφές αυτές του W και ακμές αυτές του G που να συνδέουν τις κορυφές αυτές. Σύμφωνα με το handshaking lemma, ο αριθμός των κορυφών του W είναι άρτιος [20].
3. Υπολογίζουμε ένα ελαχίστου κόστους perfect matching P στο γράφημα H (σχήμα 2.4 b).
4. Συνδυάζουμε το ελάχιστου συνδετικού δέντρου M με το P του βήματος 3 σε έναν νέο γράφημα G', χωρίς όμως να συγχωνεύσουμε παράλληλες ακμές σε μία.
5. Κατασκευάζουμε το C, ένα κύκλο Euler στο G', το οποίο να διέρχεται από μία ακμή ακριβώς μία φορά (σχήμα 2.4 c).
6. Μετατρέπουμε το C σε μια διαδρομή T, παρακάμπτοντας κορυφές που έχουμε επισκεφτεί (σχήμα 2.4 d).



Σχήμα 2.4: Στάδια εκτέλεσης αλγορίθμου Χριστοφίδη

Ο χρόνος εκτέλεσης του αλγορίθμου εξαρτάται σχεδόν αποκλειστικά από τα βήματα 2 και 3 στα οποία γίνεται ο υπολογισμός ενός ελαχίστου κόστους perfect matching το οποίο απαιτεί χρόνο $O(n^3)$ [23].

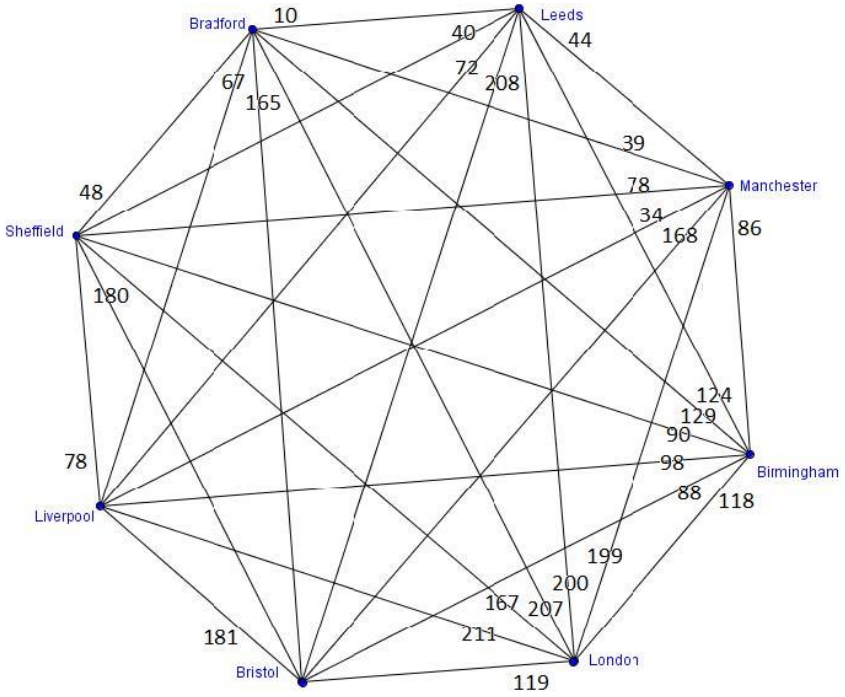
2.4.2 Αλγόριθμος Πλησιέστερου Γείτονα

Ο αλγόριθμος του πλησιέστερου γείτονα (nearest neighbour) είναι ένας προσεγγιστικός αλγόριθμος για την εύρεση μιας υποβέλτιστης λύσης στο πρόβλημα του πλανόδιου πωλητή. Ήταν ο πρώτος «άπληστος» αλγόριθμος ο οποίος έδινε λύση στο πρόβλημα. Πρωτοδιατυπώθηκε από τον J.G. Skellam και συνεχίστηκε από τους F.C. Evans και P.J. Clark. Η λύση που παράγει ο αλγόριθμος δεν είναι η βέλτιστη καθώς λόγω της «άπληστης» φύσης του δεν βρίσκει άλλες συντομότερες διαδρομές οι οποίες μπορούν να εντοπιστούν εύκολα ακόμα με την ανθρώπινη ενόραση [24]. Είναι ο πιο απλός ευρετικός αλγόριθμος και λειτουργεί ως εξής:

1. Επιλέγουμε μια τυχαία πόλη n ως αφετηρία.
2. Βρίσκουμε την κοντινότερη πόλη που δεν έχουμε επισκεφτεί και μεταβαίνουμε σε αυτήν.
3. Σημειώνουμε ότι επισκεφτήκαμε την τρέχουσα πόλη
4. Ελέγχουμε αν υπάρχουν πόλεις που δεν έχουμε επισκεφτεί. Αν υπάρχουν, πηγαίνουμε στο βήμα 2. Διαφορετικά, πηγαίνουμε στο επόμενο βήμα.
5. Επιστρέφουμε στην πόλη αφετηρίας.

Ο αλγόριθμος βρίσκει σε χρόνο $O(N^2 \log_2(N)$, όπου N είναι το πλήθος των πόλεων, λύση η οποία κυμαίνεται σε 25% απόκλιση από το κάτω φράγμα του Held-Karp [25].

Επί παραδείγματι, θα εφαρμόσουμε τον αλγόριθμο σε ένα πρόβλημα πλανόδιου πωλητή ο οποίος πρέπει να επισκεφτεί τις 8 μεγαλύτερες πόλεις στην Αγγλία (σχήμα 2.5) με αφετηρία το Λονδίνο και έχοντας ως δεδομένα τον πίνακα αποστάσεων (πίνακας 2.2).

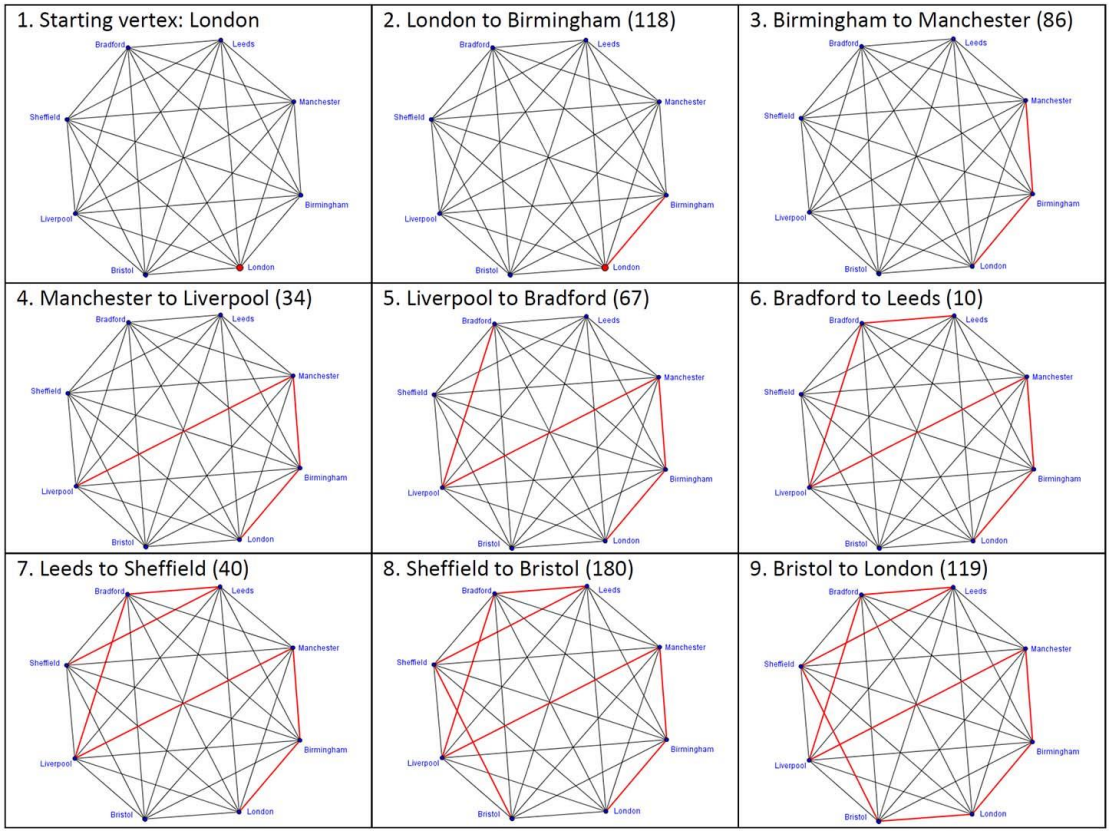


Σχήμα 2.5: Γράφημα προβλήματος

Πίνακας 2.2: Πίνακας αποστάσεων

	London	Birmingham	Leeds	Sheffield	Bradford	Liverpool	Manchester	Bristol
London		118	200	167	207	211	199	119
Birmingham	118		124	90	129	98	86	88
Leeds	200	124		40	10	72	44	208
Sheffield	167	90	40		48	78	78	180
Bradford	207	129	10	48		67	39	165
Liverpool	211	98	72	78	67		34	181
Manchester	199	86	44	78	39	34		168
Bristol	119	88	208	180	165	181	168	

Εφαρμόζοντας τον αλγόριθμο (σχήμα 2.6) όπως περιεγράφηκε προηγουμένως βρίσκουμε την διαδρομή **London – Birmingham – Manchester – Liverpool – Bradford – Leeds – Sheffield – Bristol – London** με συνολικό μήκος **654 μίλια** [11].



Σχήμα 2.6: Βήματα αλγορίθμου Nearest-Neighbour

Κεφάλαιο 3ο: Υλοποίηση της εφαρμογής

3.1 Εισαγωγή

Στα πλαίσια της εργασίας αυτής, αναπτύχθηκε web εφαρμογή η οποία επιλύει το πρόβλημα του πλανόδιου πωλητή χρησιμοποιώντας πραγματικά γεωγραφικά δεδομένα, δηλαδή σημεία στον χάρτη επιλεγόμενα από τον χρήστη. Στο κεφάλαιο αυτό θα αναφερθούμε στα εργαλεία και τις τεχνολογίες που χρησιμοποιήθηκαν για την περάτωση του έργου αυτού, την δομή και λειτουργία του και τέλος, θα παρουσιαστούν κάποια ενδεικτικά αποτελέσματα που προκύπτουν από την εκτέλεση των TSP αλγορίθμων που υλοποιήθηκαν.

3.2 Εργαλεία που χρησιμοποιήθηκαν

Για την δημιουργία αυτής της εφαρμογής χρησιμοποιήθηκαν τα ακόλουθα εργαλεία:

Visual Studio Code: Γνωστό και ως VS Code, είναι ένα πρόγραμμα σύνταξης κώδικα της Microsoft, ειδικό για την ανάπτυξη και αποσφαλμάτωση web και cloud εφαρμογών [26].

Google Chrome: Είναι πρόγραμμα περιήγησης στο Διαδίκτυο, το οποίο χρησιμοποιήθηκε για την εκτέλεση και αποσφαλμάτωση της web εφαρμογής κατά την ανάπτυξή της.

Git: Ένα κατανεμημένο σύστημα version control που χρησιμοποιείται για την διαχείριση μικρών και μεγάλων έργων προγραμματισμού [27], καταγράφοντας το ιστορικό των αλλαγών στον πηγαίο κώδικα κατά την ανάπτυξη λογισμικού [28].

Flutter και γλώσσα προγραμματισμού Dart: Το Flutter είναι μία ανοικτού κώδικα εργαλειοθήκη ανάπτυξης λογισμικού διεπαφής χρήστη (UI Software Development Kit), το οποίο δημιουργήθηκε από την Google. Ο σκοπός χρήσης του είναι η ανάπτυξη εφαρμογών για πολλαπλές πλατφόρμες (Android, iOS, Windows, Web, κλπ.) χρησιμοποιώντας τον ίδιο κώδικα για όλες αυτές [29]. Αυτό που διαφοροποιεί το Flutter από άλλες επιλογές ανάπτυξης εφαρμογών είναι ότι χρησιμοποιεί τη δική του μηχανή απεικόνισης γραφικών για την σχεδίαση των widgets και ότι η αρχιτεκτονική του βασίζεται ελάχιστα σε κώδικα C/C++. Συγκεκριμένα, τα περισσότερα τμήματα της αρχιτεκτονικής του είναι γραμμένα σε Dart, μια σύγχρονη αντικειμενοστραφή γλώσσα προγραμματισμού προσφέροντας ευελιξία και ευκολία στους προγραμματιστές [30].

OpenStreetMap και flutter_map: Το OpenStreetMap είναι ένας δωρεάν online χάρτης [31]. Για την ενσωμάτωση του στην εφαρμογή μας χρησιμοποιήθηκε το flutter_map. Το flutter_map είναι ένα package του Flutter, το οποίο είναι βασισμένο στο leaflet.js, προσφέροντας ευκολία και ευελιξία στην παραμετροποίησή του και χρησιμοποιείται για την ενσωμάτωση χαρτών σε εφαρμογές Flutter [32]. Η ευελιξία που προσφέρει επιβεβαιώνεται από την πληθώρα plugins που υποστηρίζει με σκοπό την περαιτέρω παραμετροποίηση του χάρτη. Ένα από αυτά τα plugins είναι το flutter_map_dragmarker το οποίο χρησιμοποιήσαμε στην εφαρμογή μας με σκοπό την υποστήριξη της δυνατότητας να μπορούν να μετατοπιστούν τα σημεία (markers) στον χάρτη αφού έχουν επιλεγεί από τον χρήστη [33].

Open Route Service: Η Open Route Service προσφέρει global spatial υπηρεσίες μέσω API, αντλώντας δωρεάν γεωγραφικά δεδομένα από το OpenStreetMap. Ανάμεσα σε αυτές τις υπηρεσίες που παρέχει είναι το Matrix API, το οποίο χρειαζόμαστε στην εφαρμογή μας για τον υπολογισμό του πίνακα αποστάσεων μεταξύ των σημείων ενδιαφέροντος που έχουν επιλεγεί στον χάρτη. Για την χρήση αυτού

του API στην εφαρμογή χρησιμοποιούμε το open_route_service flutter package [34]. Αυτό το πακέτο περιλαμβάνει μεθόδους και μοντέλα, απαραίτητα για την εύκολη χρήση του Open Route Service API.

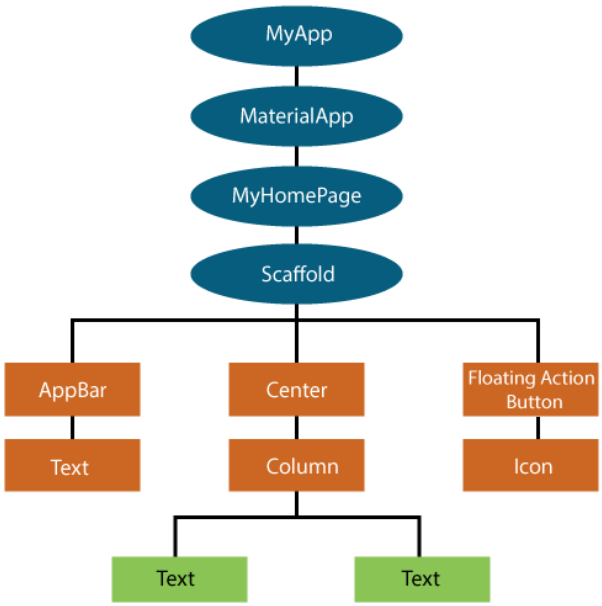
Syncfusion's Flutter Charts: Είναι μία βιβλιοθήκη για την απεικόνιση δεδομένων σε μορφή γραφημάτων [35]. Χρησιμοποιήθηκε για την δημιουργία των γραφημάτων με δεδομένα τα αποτελέσματα εκτέλεσης των TSP αλγορίθμων.

3.3 Δομή της εφαρμογής

3.3.1 Εισαγωγή

Προτού περιγράψουμε την δομή και τον τρόπο λειτουργίας της εφαρμογής μας, είναι απαραίτητο να αναφερθούμε πρώτα στα βασικά χαρακτηριστικά που διέπουν τις εφαρμογές Flutter.

Στο Flutter τα widgets αποτελούν τα δομικά στοιχεία κάθε εφαρμογής καθώς καθορίζουν την εμφάνιση του UI. Κάθε στοιχείο σε κάθε οθόνη μιας εφαρμογής Flutter είναι ένα widget και το UI της εφαρμογής εξαρτάται από ποια widgets χρησιμοποιούνται και την σειρά με την οποία τα τοποθετούμε στο widget tree [36]. Το widget tree είναι ο τρόπος με τον οποίο δημιουργούμε το UI της εφαρμογής και δουλεύει τοποθετώντας widgets εμφωλεύοντας το ένα μέσα στο άλλο κατασκευάζοντας έτσι απλές ή περίπλοκες διατάξεις στο UI [37].



Σχήμα 3.1: Απεικόνιση ενός widget tree [38]

Το Flutter διαθέτει μια ευρεία γκάμα από έτοιμα widgets όπως text, buttons, lists, layouts. Εκτός όμως από αυτά, δίνει και την δυνατότητα να δημιουργήσουμε δικά μας widgets. Υπάρχουν 2 τύποι widgets:

Stateless Widgets: Είναι αμετάβλητα widgets καθώς δεν διατηρούν κάποιου είδους κατάσταση, δηλαδή δεν έχουν μεταβλητές των οποίων το περιεχόμενο μπορεί να αλλάξει. Μερικά παραδείγματα stateless widgets είναι ένα Text widget, ένα Icon widget και ένα IconButton widget. Όλα αυτά εμφανίζουν κάποιο κείμενο, ένα εικονίδιο και ένα κουμπί αντιστοίχως, τα οποία δεν αλλάζουν κατά την διάρκεια εκτέλεσης του προγράμματος.

Stateful Widgets: Είναι δυναμικά widgets, ανανεώνουν την εμφάνιση του UI βάσει αλλαγών που γίνονται κατά την εκτέλεση του κώδικα. Ο χρήστης μπορεί και αλληλοεπιδρά με αυτά. Επί παραδείγματι, με το πάτημα ενός κουμπιού εκτελείται μια ενέργεια, κάνοντας swipe σε ένα στοιχείο μιας λίστας διαγράφεται το στοιχείο αυτό από την λίστα, ένα πεδίο εισαγωγής κειμένου, ένα κουτί το οποίο μπορείς να τσεκάρεις. Τα παραπάνω παραδείγματα αντιπροσωπεύουν stateful widgets.

Ουσιαστικά αυτό που διαπιστώνουμε είναι ότι αν ένα widget μεταβάλλεται είναι stateful, αν δεν μεταβάλλεται τότε είναι stateless [39].

3.3.2 Δομή φακέλου project

Η δομή της εφαρμογής με τα σημαντικότερα αρχεία και φακέλους είναι η ακόλουθη:

- lib/
    - algorithms/
    - christofides/
        - christofides.dart
        - fleury.dart
        - lists.dart
        - munkres.dart
        - prims.dart
    - held_karp.dart
    - nearest_neighbor.dart
    - tsp_bounds.dart
- models/
- tsp_result.dart
- screens/
- charts_screen.dart
- map_screen.dart
- calculate_screen.dart
- widgets/
- flutter_map.dart
- data.dart
- helper.dart
- main.dart
- ors.dart
- tsp.dart
- web/
- index.html
- pubspec.yaml

3.3.3 Τα κυριότερα αρχεία σε Flutter projects ιστού

Το αρχείο **main.dart** είναι το βασικότερο αρχείο κώδικα σε ένα έργο με Flutter. Σε αυτό υπάρχει η main μέθοδος, η οποία είναι το σημείο εισόδου μιας εφαρμογής Flutter και είναι υπεύθυνη για την εκτέλεσή της με την εντολή runApp(const MyApp()) [40]. Η runApp δημιουργεί το Widget που της δίνεται ως παράμετρος και το κάνει attach στην οθόνη [41].

Η κλάση **MyApp** είναι ένα stateless widget το οποίο είναι η ρίζα της εφαρμογής μας μέσω του οποίου θα κατασκευαστεί το layout της οθόνης. Μέσα σε αυτήν υπάρχει η build μέθοδος η οποία επιστρέφει το MaterialApp widget.

Η μέθοδος **build** «περιγράφει» το μέρος του UI το οποίο αναπαριστά το widget στο οποίο περιέχεται. Την συναντάμε στα Stateless widgets και στο state των stateful widgets. Στα Stateless widgets, το Flutter framework καλεί αυτήν την μέθοδο όταν το widget προστίθεται στο widget tree ή όταν οι εξαρτήσεις αυτού του widget μεταβληθούν [42]. Στο State των stateful widgets, η μέθοδος μπορεί να κληθεί από το framework σε πολλές διαφορετικές περιπτώσεις. Μερικές από αυτές είναι με την κλήση της initState, της didUpdateWidget, της setState [43]. Στην εφαρμογή μας, η setState είναι αυτή που χρησιμοποιούμε συχνότερα, την οποία καλούμε έτσι ώστε να κληθεί από το framework η build μέθοδος όταν θέλουμε να ανανεώσουμε το UI.

Το **MaterialApp** είναι μια προκαθορισμένη κλάση στο Flutter η οποία είναι ένα κύριο component του Flutter. Μέσω αυτού έχουμε πρόσβαση σε όλα τα components και widgets του Flutter SDK όπως το Scaffold widget [44]. Σε αυτό ορίζουμε την κεντρική σελίδα-οθόνη της εφαρμογής μας η οποία είναι η κλάση MyHomePage, η οποία εντοπίζεται στο αρχείο map_screen.dart. Περισσότερα για αυτό θα αναφερθούμε στην επόμενη ενότητα.

Σε αυτό το σημείο καλό είναι να επισημάνουμε ότι κάθε ξεχωριστή σελίδα-οθόνη που θέλουμε να υπάρχει στην εφαρμογή μας, βρίσκεται ξεχωριστά στο δικό της αρχείο dart για καλύτερη οργάνωση. Για την λειτουργία αυτής της οθόνης επιλέγουμε την χρήση stateful widget και όχι stateless καθώς θέλουμε να υπάρχει αλληλεπίδραση μεταξύ χρήστη και εφαρμογής. Ένα stateful widget κατασκευάζεται με 2 κλάσεις: μία η οποία είναι subclass της StatefulWidget και ορίζει το widget μας και μία η οποία είναι subclass της State και περιέχει το state και την build() μέθοδο του widget αυτού [45]. Η subclass της StatefulWidget πάντα θα περιέχει τον constructor της, final μεταβλητές που χρησιμοποιούνται από την build μέθοδο της State του και State αντικείμενο στο οποίο διατηρεί την ευμετάβλητη State του και το οποίο δημιουργείται με την createState μέθοδο [46].

Ένα άλλο σημαντικό αρχείο στο οποίο θα αναφερθούμε είναι το **pubspec.yaml**. Κάθε έργο Flutter έχει ένα τέτοιο αρχείο. Το pubspec δημιουργείται αυτόματα στην ρίζα του project folder tree όταν δημιουργείς ένα νέο Flutter έργο από την κονσόλα (εντολή flutter create). Εμπεριέχει metadata σχετικά με το έργο, τα οποία είναι απαραίτητα για την Dart και το Flutter. Είναι γραμμένο σε γλώσσα YAML, μια γλώσσα προγραμματισμού ευανάγνωστη από ανθρώπους. Το pubspec ορίζει τις εξαρτήσεις του Flutter έργου μας, δηλαδή συγκεκριμένα packages, γραμματοσειρές, αρχεία εικόνων που χρησιμοποιούνται στην εφαρμογή μας καθώς και περιορισμούς στην έκδοση του Flutter SDK [47].

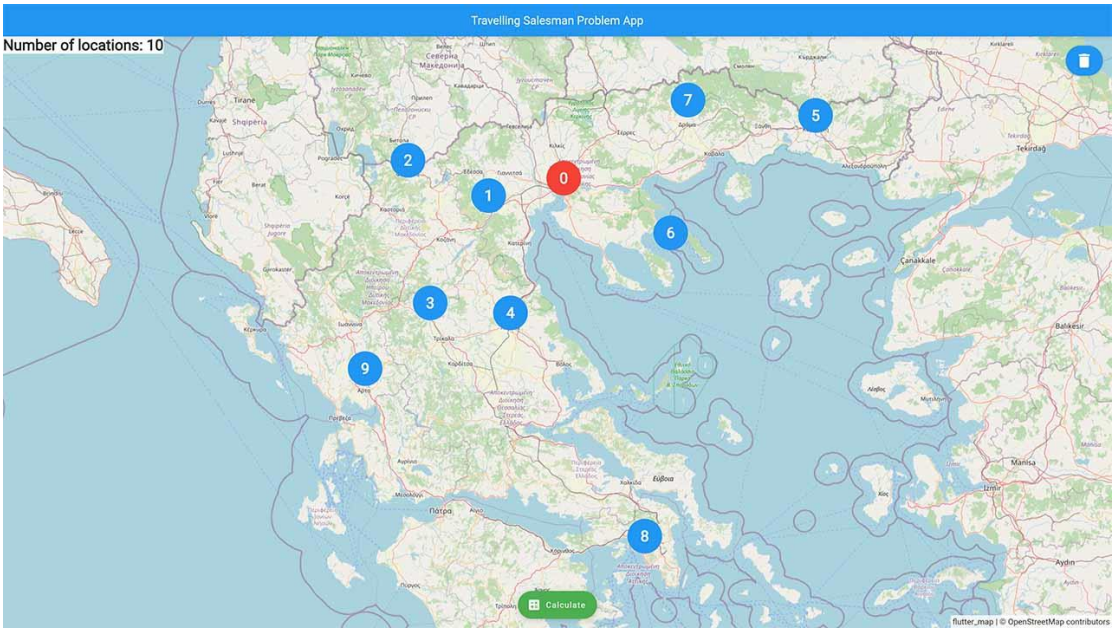
Το **index.html** είναι το σημαντικότερο αρχείο σε οποιοδήποτε έργο ιστού. Δημιουργείται και αυτό αυτόματα με την δημιουργία ενός νέου Flutter έργου (εντολή flutter create). Περιέχει ένα script tag μέσα στο οποίο εκτελείται ασύγχρονα κώδικας για την φόρτωση της Flutter web engine και εν τέλει την εκτέλεση της εφαρμογής μας με την runApp(), στην οποία αναφερθήκαμε παραπάνω [48].

3.3.4 Αρχεία σελίδων web εφαρμογής

Αρχείο /screens/map_screen.dart

Εδώ βρίσκεται η κεντρική σελίδα της εφαρμογή μας η οποία αποτελείται από τον χάρτη που εμφανίζεται στον χρήστη μέσα από τον οποίο επιλέγει τα σημεία ενδιαφέροντος για το πρόβλημα TSP. Η σελίδα αυτή κατασκευάστηκε με την κλάση MyHomePage, την οποία έχουμε ορίσει ως StatefulWidget και το State του στην κλάση _MyHomePageState. Την κλάση με το State καθώς και τις μεταβλητές και

μεθόδους είναι καθιερωμένο στην Flutter να τα ορίζουμε ως private, το οποίο γίνεται με την κάτω παύλα _ μπροστά από το όνομά τους.



Σχήμα 3.2: Κεντρική σελίδα (MyHomePage)

Η **MyHomePage** κλάση είναι ένα stateful widget. Περιέχει τον constructor της, τον τίτλο της εφαρμογής τον οποίο αποκτά από το γονικό widget, στην προκειμένη περίπτωση αυτό είναι το MaterialApp στην main.dart, και το State αντικείμενο το οποίο δημιουργεί την State στην _MyHomePageState κλάση με την μέθοδο createState.

Στην **_MyHomePageState** κλάση έχουμε τις μεθόδους _addLocation, _clearLocations και την build.

Η **build** μέθοδος καθορίζει την εμφάνιση του UI της MyHomePage, επιστρέφοντας ένα Widget. Το Widget που επιστρέφουμε είναι ένα Scaffold. Αυτό υλοποιεί ένα απλό Material Design layout και παρέχει APIs για την εμφάνιση διάφορων αντικειμένων στο UI όπως κουμπιά (FloatingActionButton), γραμμή τίτλου (AppBar), κείμενο (Text) ή οτιδήποτε άλλο με το body property [49]. Στην περίπτωσή μας, το Scaffold χρησιμοποιεί τα εξής:

appBar: η γραμμή τίτλου της εφαρμογής, ο οποίος τίτλος βρίσκεται στην κλάση MyHomePage (final String title).

body: το κυρίως περιεχόμενο του Scaffold, το οποίο εμφανίζεται κάτω από την γραμμή τίτλου. Περιέχει ένα Stack widget, το οποίο καθορίζει τον τρόπο εμφάνισης των widgets που βρίσκονται κάτω από αυτό (children). Συγκεκριμένα επιτρέπει να στοιβάξουμε widgets το ένα πάνω στο άλλο [50]. Ως children ορίζουμε έναν πίνακα <Widget>[], μέσα στον οποίο έχουμε τα ακόλουθα widgets:

flutterMap(_addLocation): μέθοδος η οποία επιστρέφει ένα FlutterMap widget, το οποίο βρίσκεται στο αρχείο ./widgets/flutter_map.dart.

Text widget: εμφανίζει τον αριθμό των τοποθεσιών που έχει επιλέξει ο χρήστης. Τον τρόπο εμφάνισης του κειμένου τον έχουμε ορίσει με τις ιδιότητες textAlign, overflow και style. Αξίζει να σημειωθεί ότι το widget αυτό το περικλείουμε από ένα PointerInterceptor widget με σκοπό να αποτραπεί η προσθήκη

τοποθεσίας στον χάρτη σε περιπτώσεις που ο χρήστης πατήσει πάνω στο κείμενο, λόγω της onTap του FlutterMap widget, το οποίο βρίσκεται ιεραρχικά κάτω από το Text widget στο Stack.

ElevatedButton widget: κουμπί με εικονίδιο έναν κάδο για την εκκαθάριση των επιλεγμένων τοποθεσιών από τον χάρτη. Σε αυτό έχουμε ορίσει μεταξύ άλλων ιδιοτήτων που καθορίζουν το σχήμα και το μέγεθός του και την ιδιότητα onPressed, με την οποία όταν πατιέται το κουμπί να καλείται η μέθοδος _clearLocations(). Περικλείεται από ένα Padding widget για το κεντράρισμα του εικονιδίου μέσα στο κουμπί και από ένα Align widget με ιδιότητα alignment για την τοποθέτηση του κουμπιού στο επάνω δεξιό μέρος της σελίδας.

floatingActionButton: ένα κουμπί «Calculate» το οποίο εμφανίζεται πάνω από το body του Scaffold. Χρησιμοποιούμε την ιδιότητα onPressed έτσι ώστε όταν ο χρήστης πατήσει το κουμπί να εκτελεστεί κώδικας ο οποίος θα κατευθύνει στην επόμενη σελίδα της εφαρμογής, την CalculateScreen. Αυτό θα συμβεί υπό την προϋπόθεση ότι έχει επιλέξει ο χρήστης τουλάχιστον 3 σημεία στον χάρτη. Αν δεν ισχύει αυτό, τότε εμφανίζεται AlertDialog ειδοποιώντας ότι απαιτούνται τουλάχιστον 3 για την εύρεση λύσης για το πρόβλημα του πλανόδιου πωλητή. Η ανακατεύθυνση στην CalculateScreen γίνεται με την εντολή Navigator.push, μια λειτουργία στο Flutter που εξηγούμε παρακάτω.

floatingActionButtonLocation: ορίζει την θέση του floatingActionButton στην οθόνη.

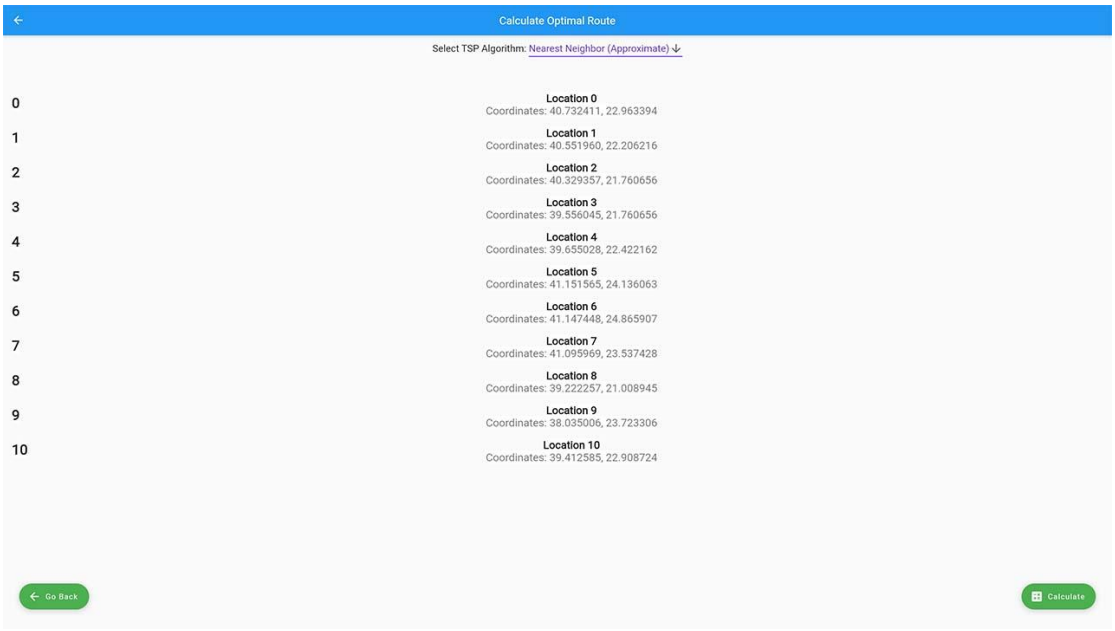
Στο Flutter οι διαφορετικές οθόνες και σελίδες που έχει μια εφαρμογή ονομάζονται routes. Το Flutter παρέχει μηχανισμούς για την περιήγηση σε διαφορετικές οθόνες ή σελίδες σε μια εφαρμογή. Ένας από αυτούς είναι ο Navigator, ένα widget το οποίο διαχειρίζεται ένα σετ από child widgets με την λογική της στοίβας. Μέσω αυτού καλούμε την μέθοδο push() η οποία προσθέτει μια Route στην στοίβα του Navigator. Η Route που χρησιμοποιούμε είναι ένα MaterialPageRoute στο οποίο δίνουμε ως παράμετρο το widget της σελίδας ή οθόνης [51] [52] [53].

Η μέθοδος **_addLocation** καλείται όταν χρήστης πατάει πάνω σε οποιοδήποτε σημείο στον χάρτη. Δέχεται ως είσοδο γεωγραφικές συντεταγμένες LatLng. Καλείται με την onTap του FlutterMap widget, στο οποίο θα αναφερθούμε αναλυτικά σε επόμενη ενότητα. Σε αυτήν δημιουργείται ένα αντικείμενο τύπου DragMarker το οποίο προστίθεται στην λίστα Data.locations, η οποία περιέχει όλα τα σημεία ενδιαφέροντος. Το DragMarker ορίζεται από τις γεωγραφικές συντεταγμένες του σημείου που πατήθηκε, τις οποίες δέχεται ως είσοδο η μέθοδος, από τις διαστάσεις του και το εικονίδιό του. Σημειωτέων ότι για να εμφανιστεί ο marker στον χάρτη καλούμε την μέθοδο setState() έτσι ώστε το Flutter framework καλέσει ακολούθως την build του State μας και να ανανεωθεί το UI. Επίσης, έχουμε ορίσει να γίνεται έλεγχος ώστε όταν ξεπερνάει η λίστα Data.locations τις 50 τοποθεσίες να εμφανίζει ένα AlertDialog στον χρήστη, ενημερώνοντας τον ότι υπάρχει όριο 50 πόλεων, λόγω περιορισμών του Distance Matrix API.

Η μέθοδος **_clearLocations** καλείται όταν πατιέται το κουμπί εκκαθάρισης των επιλεγμένων σημείων στο χάρτη (κουμπί κάδου στο πάνω δεξιό μέρος της οθόνης). Αδειάζει την λίστα τοποθεσιών Data.locations και καλεί την setState() για την ανανέωση του UI.

Αρχείο /screens/calculate_screen.dart

Σε αυτό το αρχείο έχουμε την οθόνη **CalculateScreen** (stateful widget) στην οποία γίνεται ο υπολογισμός της λύσης του προβλήματος TSP με έναν αλγόριθμο επιλεγόμενο από τον χρήστη.



Σχήμα 3.3: Σελίδα CalculateScreen

Η κλάση **CalculateScreen** είναι StatefulWidget. Σε αυτήν βρίσκεται ο constructor της, η createState() και 2 μεταβλητές που χρησιμοποιούνται για την ένδειξη σε ποσοστό της προόδου υπολογισμού του αλγορίθμου TSP.

Η κλάση **_CalculateScreenState** είναι το State της CalculateScreen. Μέσα σε αυτήν έχουμε διάφορες βοηθητικές μεταβλητές και τις μεθόδους toggleProgress(), calculate(), algorithmDropdown και build.

Η μέθοδος **build** κατασκευάζει το UI της σελίδας αυτής. Περιέχει τα εξής:

- Ένα dropdown menu καλώντας την algorithmDropdown(). Μέσω αυτού ο χρήστης επιλέγει αλγόριθμο για τον υπολογισμό της λύσης στο πρόβλημα.
- Ένα Text widget για την εμφάνιση της προόδου εκτέλεσης του αλγορίθμου σε ποσοστό και ένα LinearProgressIndicator για την εμφάνιση της προόδου ως γραφικό στοιχείο. Το καθένα τους βρίσκεται μέσα σε ένα ValueListenableBuilder, ένα widget το οποίο κάνει build ό,τι βρίσκεται ιεραρχικά κάτω από αυτό κάθε φορά που η τιμή του valueListenable αλλάζει. Το χρησιμοποιούμε καθώς προσφέρει την δυνατότητα να εμφανίζεται το ποσοστό στο UI χωρίς να χρειάζεται να καλείται κάθε φορά η setState() για να ενημερωθεί το UI, καθώς είναι περισσότερο κοστοβόρο να γίνει build ολόκληρο το widget tree.

Σχήμα 3.4: Ένδειξη προόδου αλγορίθμου (LinearProgressIndicator)

- Μια λίστα με τις επιλεγμένες τοποθεσίες στον χάρτη χρησιμοποιώντας ένα ListView widget. Το ListView μέσω του itemBuilder δημιουργεί ένα scrollable πίνακα από widgets ο οποίος γεμίζει κατ' απαίτηση καλώντας την μέθοδο getLocationsList().
- Ένα κουμπί επιστροφής στην προηγούμενη οθόνη (MyHomePage) το οποίο υλοποιείται με ένα FloatingActionButton. Για την ανακατεύθυνση στην προηγούμενη σελίδα χρησιμοποιείται κώδικας με την εντολή Navigator.pop().
- Ένα κουμπί «Calculate» για να ξεκινήσει ο αλγόριθμος που επιλέχθηκε. Υλοποιείται και αυτό με ένα FloatingActionButton. Με την παράμετρο onPressed ορίζουμε όταν πατηθεί το κουμπί να κληθεί η μέθοδος calculate().
- Διάφορα widgets όπως Scaffold, Center, Column, Row, SizedBox, Visibility, Stack, Positioned. Αυτά περικλείουν τα προαναφερθέντα widgets, δηλαδή είναι parent αυτών, βρίσκονται

ιεραρχικά πάνω από αυτά και χρησιμοποιούνται για τον καθορισμό του τρόπου εμφάνισής τους (layout) στο UI.

Η μέθοδος **toggleProgress** χρησιμοποιείται για την εμφάνιση και την απόκρυψη του LinearProgressIndicator widget έτσι ώστε να εμφανίζεται μόνο όταν βρίσκεται σε εξέλιξη κάποιος αλγόριθμος υπολογισμού λύσης TSP. Το LinearProgressIndicator είναι ένα widget το οποίο εμφανίζει γραφικά το ποσοστό προόδου του αλγορίθμου (εικόνα Χ γραφικού πιο πάνω).

Η μέθοδος **calculate** ξεκινάει τον υπολογισμό της λύσης στο πρόβλημα του πλανόδιου πωλητή με τον επιλεγμένο αλγόριθμο και τις επιλεγμένες τοποθεσίες. Οι διεργασίες που εκτελεί είναι οι εξής:

1. Εμφανίζει στο UI την πρόοδο εκτέλεσης του αλγορίθμου.
2. Υπολογίζει τον πίνακα αποστάσεων. Σε περίπτωση σφάλματος του API, εμφανίζεται μήνυμα λάθους στον χρήστη.
3. Υπολογίζει τα φράγματα του TSP.
4. Επιλύει το TSP με τον επιλεγμένο αλγόριθμο. Αν έχει επιλεγεί η εκτέλεση όλων των αλγορίθμων, τότε εκτελείται διαδοχικά ο καθένας αλγόριθμος, τα αποτελέσματα των οποίων αποθηκεύονται σε μια λίστα.
5. Εξαφανίζει το γραφικό της προόδου εκτέλεσης από το UI.
6. Εμφανίζει σε AlertDialog τα φράγματα, την λύση που βρέθηκε και δίνει την επιλογή εμφάνιση της λύσης πάνω στον χάρτη σε δεύτερο AlertDialog.
7. Κάνει επαναφορά κάποιες βοηθητικές μεταβλητές.

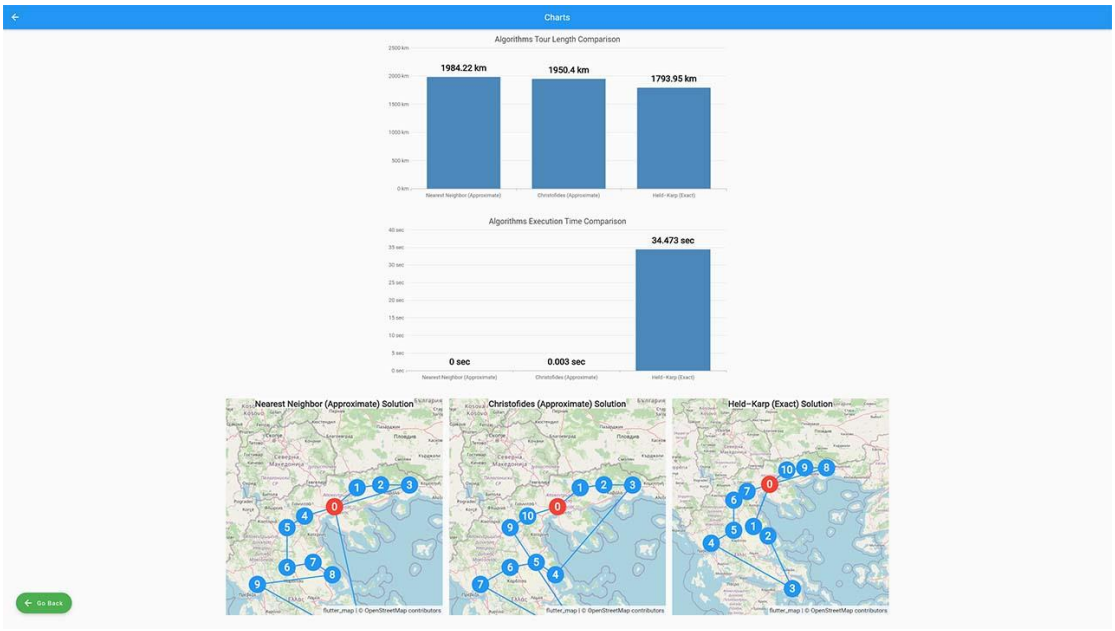
Η μέθοδος **algorithmDropdown** επιστρέφει ένα DropdownButton widget τύπου String. Το widget αυτό εμφανίζει μια αναδυομένη λίστα για να επιλέξει ο χρήστης έναν από τους διαθέσιμους αλγορίθμους. Η λίστα αυτή αποτελείται από DropdownMenuItems τα οποία έχουν ως κείμενο το String που επιστρέφει η μέθοδος getAlgoText() στο helper.dart.

Η μέθοδος **getLocationsList** δέχεται ως είσοδο BuildContext και τον δείκτη (index) του πίνακα στον itemBuilder του ListView. Επιστρέφει ένα ListTile το οποίο περιέχει κείμενο που δηλώνει τον αριθμό της τοποθεσίας και τις γεωγραφικές συντεταγμένες της.

Η μέθοδος **getListOfLocations** χρησιμοποιείται για την μετατροπή της λίστας με τα DragMarkers (Data.locations) σε διαφορετική μορφή λίστας που απαιτείται από το Distance Matrix API.

Αρχείο /screens/charts_screen.dart

Σε αυτό το αρχείο έχουμε την σελίδα με τα διαγράμματα των αποτελεσμάτων εκτέλεσης όλων των αλγορίθμων και τους χάρτες με την λύση του καθενός. Η σελίδα αυτή μπορεί να εμφανιστεί μόνο στην περίπτωση που ο χρήστης επιλέξει την εκτέλεση και των τριών αλγορίθμων από το dropdown menu στην calculate_screen.dart. Εμφανίζεται με το πάτημα του κουμπιού “Show Charts” στο αναδυόμενο παραθυράκι που εμφανίζεται με τα αποτελέσματα της εκτέλεσης. Η λειτουργικότητα των διαγραμμάτων παρέχεται από το package syncfusion_flutter_charts.



Σχήμα 3.5: Οθόνη διαγραμμάτων (ChartsScreen)

Η κλάση **ChartsScreen** είναι StatefulWidget και περιέχει τον constructor της και την createState().

Η κλάση **_ChartsScreenState** είναι το State της ChartsScreen. Περιέχει την build για την κατασκευή του UI και άλλες μεθόδους για την κατασκευή των διαγραμμάτων .

Οι μέθοδοι **_buildTourLengthColumnChart** και **_buildTimeColumnChart** επιστρέφουν ένα SfCartesianChart widget. Το widget αυτό χρησιμοποιείται για την απεικόνιση πολλών ειδών γραφημάτων σε καρτεσιανό σύστημα συντεταγμένων [54]. Στην προκειμένη περίπτωση τα γραφήματα που απεικονίζουν είναι ραβδογράμματα. Στην πρώτη μέθοδο το widget είναι διαμορφωμένο για την συγκριτική απεικόνιση του μήκους της βέλτιστης διαδρομής που βρήκε ο κάθε αλγόριθμος. Στην δεύτερη το widget που επιστρέφεται απεικονίζει τους χρόνους εκτέλεσης των αλγορίθμων.

Οι μέθοδοι **_getTourLengthColumnSeries** και **_getTimeColumnSeries** χρησιμοποιούνται από τις προαναφερθέντες μεθόδους αντιστοίχως για τον καθορισμό της πηγής των δεδομένων των γραφημάτων (Data.allAlgorithms, αρχείο data.dart) και την αντιστοίχισή τους στους άξονες X και Y. Και στις δύο μεθόδους στον άξονα X αντιστοιχίζεται το όνομα του αλγορίθμου. Στον άξονα Y αντιστοιχίζεται το μήκος της της διαδρομής για την μέθοδο _getTourLengthColumnSeries και ο χρόνος εκτέλεσης σε δευτερόλεπτα για την μέθοδο _getTimeColumnSeries.

Η **build** μέθοδος διαμορφώνει την σελίδα αυτή καλώντας τις μεθόδους που αναφέρθηκαν παραπάνω για την απεικόνιση των γραφημάτων και καλώντας εις τριπλούν την μέθοδο flutterMapWithTour του αρχείου flutter_map.dart για την εμφάνιση των χαρτών με τις διαδρομές που βρήκαν και οι τρεις αλγόριθμοι. Επίσης, χρησιμοποιούνται widgets του Flutter όπως Scaffold, Center, SingleChildScrollView, Column, SizedBox, Stack και Positioned για τον καθορισμό του τρόπου εμφάνισής τους (layout) στο UI. Τέλος, ορίζεται floatingActionButton στο Scaffold για την εμφάνιση κουμπιού “Go Back” κάτω αριστερά για την επιστροφή στην προηγούμενη σελίδα (Navigator.pop) όταν πατηθεί (onPressed).

3.3.5 Αρχεία widgets

Αρχείο /widgets/flutter_map.dart

Η μέθοδος **flutterMap** χρησιμοποιείται για την χρήση του FlutterMap widget σε οποιοδήποτε μέρος της εφαρμογής μας. Αυτό το widget παρέχεται από το package flutter_map, το οποίο μας δίνει πολλές δυνατότητες παραμετροποίησης σύμφωνα με τις απαιτήσεις τις εφαρμογής μας. Η μέθοδος δέχεται ως είσοδο την μέθοδο addLocation της _MyHomePageState στο map_screen.dart και επιστρέφει FlutterMap widget. Το widget αυτό το κατασκευάζουμε με τις παραμέτρους:

options: ορίζουμε ένα αντικείμενο MapOptions με διάφορες παραμέτρους για την συμπεριφορά του χάρτη. Συγκεκριμένα, ορίζουμε γεωγραφικές συντεταγμένες στις οποίες θα είναι αρχικά κεντραρισμένος ο χάρτης (center), προεπιλεγμένη κλίμακα-μεγέθυνση (zoom), μέγιστη μεγέθυνση (maxZoom), κλήση της μεθόδου addLocation όταν πατιέται οποιοδήποτε σημείο του χάρτη (onTap), interactiveFlags και absorbPanEventsOnScrollables.

notRotatedChildren: το χρησιμοποιούμε για να ορίσουμε layers τα οποία δεν περιστρέφονται και υπάρχει ώστε να εμφανίζεται στο κάτω δεξιό μέρος του χάρτη attribution για την πηγή του χάρτη.

children: ορίζουμε ένα πίνακα που περιέχει TileLayer για να ορίσουμε την πηγή δεδομένων χαρτών η οποία είναι το OpenStreetMap και DragMarkers που προσθέτει την λειτουργικότητα των markers στον χάρτη. Υπόψιν ότι την λειτουργικότητα του DragMarkers μας την παρέχει το package flutter_map_dragmarker. Οι markers του DragMarkers widget ορίζονται η static λίστα List<DragMarker> locations στην κλάση Data του αρχείου data.dart. Χρησιμοποιούμε αυτό το package καθώς προσθέτει στους markers του χάρτη την λειτουργία μετακίνησης τους στον χάρτη.

Στο αρχείο αυτό συναντούμε επίσης την μέθοδο **flutterMapWithTour** η οποία χρησιμοποιείται για την εμφάνιση της λύσης που βρίσκει ο κάθε αλγόριθμος TSP με το FlutterMap widget. Δέχεται ως είσοδο ένα TspResult αντικείμενο και επιστρέφει ένα Stack widget μέσα στο οποίο βρίσκεται ο χάρτης (FlutterMap widget) και κείμενο με τον αλγόριθμο της λύσης που απεικονίζεται σε αυτόν (Text widget). Ο χάρτης που επιστρέφεται είναι λίγο διαφορετικός από αυτόν που επιστρέφει η flutterMap μέθοδος που περιεγράφηκε προηγουμένως. Κέντρο του χάρτη είναι η πόλη εκκίνησης (0), η κλίμακα του χάρτη είναι μικρότερη και οι markers του χάρτη δεν είναι μετακινούμενοι καθώς χρησιμοποιείται MarkerLayer αντί για DragMarkers και χρησιμοποιείται επιπλέον το PolylineLayer. Το PolylineLayer συνδέει με γραμμές τις πόλεις του προβλήματος για την καλύτερη απεικόνιση της λύσης που βρέθηκε.

3.3.6 Άλλα αρχεία

Αρχείο ors.dart

Σε αυτό το αρχείο έχουμε δημιουργήσει την κλάση ORS με μεθόδους που σχετίζονται με το Open Route Service. Για να έχουμε πρόσβαση στα APIs του χρησιμοποιούμε το package open_route_service.

Στην κλάση **ORS** βρίσκεται η μέθοδος distanceMatrixApi η οποία μέσω ενός αντικειμένου client τύπου OpenRouteService πραγματοποιεί αίτημα matrixPost για να λάβει το πίνακα αποστάσεων των τοποθεσιών, στον οποίον οι αποστάσεις είναι σε μέτρα. Για να γίνει το αίτημα αυτό γίνεται προηγουμένως αρχικοποίηση του client με ένα apiKey,το οποίο αποκτήσαμε δωρεάν από το openrouteservice.org, και μετατροπή της λίστας τοποθεσιών σε τύπο List<ORSCoordinate> με την μέθοδο _convertToORS. Το αίτημα γίνεται μέσα σε μία try catch έτσι ώστε σε περίπτωση σφάλματος να επιστρέφεται άδειος πίνακας και για την εμφάνιση σχετικού μηνύματος στον χρήστη.

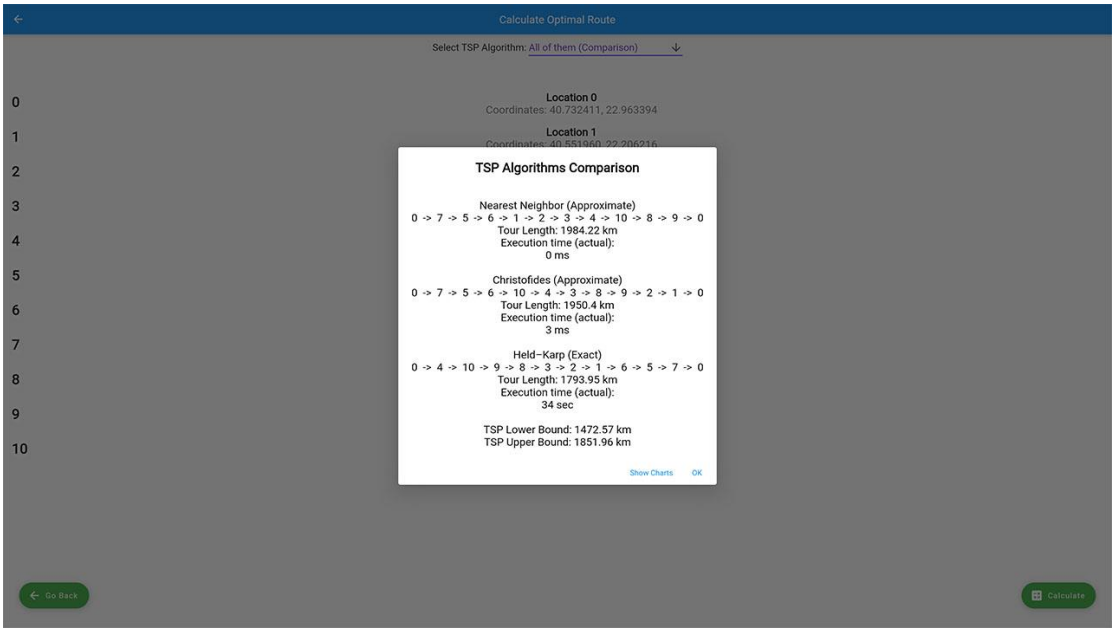
Αρχείο tsp.dart

Η μέθοδος **solveTSP** χρησιμοποιείται για την επίλυση του προβλήματος του πλανόδιου πωλητή. Είσοδος ορίζεται ο επιλεγμένος αλγόριθμος (String selectedAlgo) και ο δισδιάστατος πίνακας αποστάσεων (List<List<double>> distanceMatrix). Η μέθοδος εκτελείται ασύγχρονα και επιστρέφει ένα αντικείμενο τύπου TspResult με τα αποτελέσματα εκτέλεσης του αλγορίθμου. Βάσει της τιμής της selectedAlgo καλείται ο αντίστοιχος αλγόριθμος.

Αρχείο helper.dart

Στην κλάση **Helper** έχουμε τις μεθόδους showMyDialog και showMapDialog και τις βοηθητικες μεταβλητές isShown και isShowMap για την αποτροπή εμφάνισης πολλαπλών AlertDialogs.

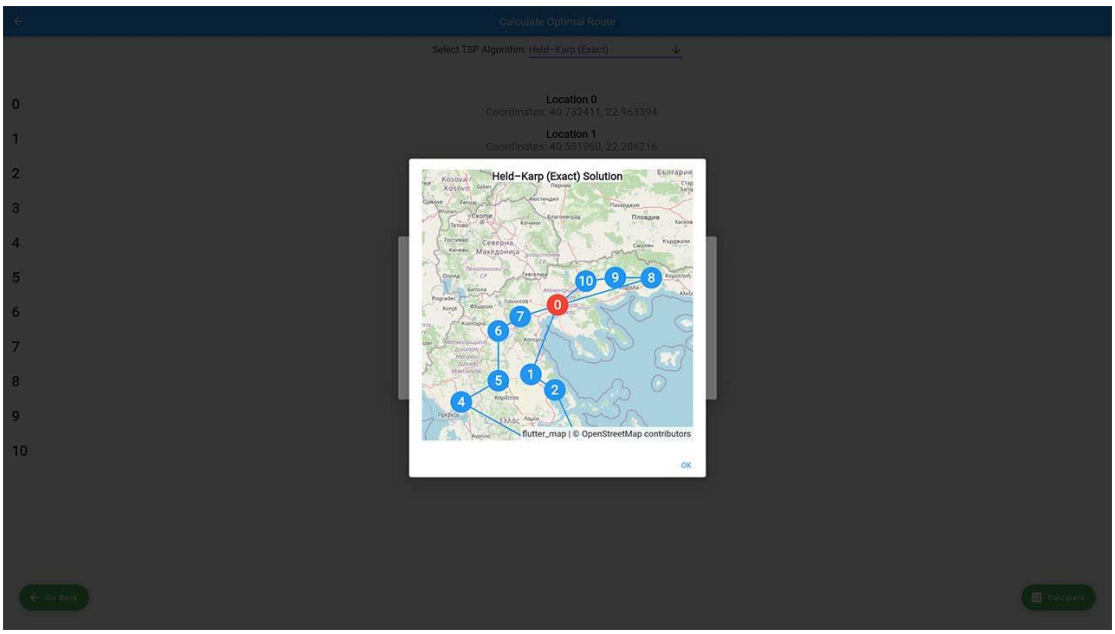
Η μέθοδος **showMyDialog** χρησιμοποιείται για την εμφάνιση AlertDialog, δηλαδή ένα αναδυόμενο παραθυράκι πάνω από το περιεχόμενο της εφαρμογής. Δέχεται ως είσοδο έναν τίτλο (dialogTitle), ένα μήνυμα (message), BuildContext για την εμφάνιση του, μια boolean μεταβλητή chartsBtn και λίστα tspResults με τα αποτελέσματα εκτέλεσης αλγορίθμου/ων εφόσον υπάρχουν. Καλείται από το map_screen.dart για την εμφάνιση μηνυμάτων προειδοποίησης στον χρήστη και από το calculate_screen.dart για την εμφάνιση των αποτελεσμάτων της εκτέλεσης του αλγορίθμου. Το AlertDialog που εμφανίζεται στον χρήστη αποτελείται από τον τίτλο, το μήνυμα και ένα κουμπί “OK” για την απόκρυψη του παραθύρου. Στην περίπτωση που εκτελέστηκαν όλοι οι αλγόριθμοι (chartsBtn = true), εμφανίζεται επίσης κουμπί για την εμφάνιση των γραφημάτων διαμέσου της μεθόδου showChartsBtn. Αντιθέτως, αν εκτελέστηκε μόνο ένας αλγόριθμος (ελέγχουμε το μέγεθος της λίστας tspResults), τότε με την μέθοδο showTourOnMap εμφανίζεται κουμπί για την εμφάνιση της διαδρομής στον χάρτη σε 2ο αναδυόμενο παραθυράκι.



Σχήμα 3.6: AlertDialog αποτελεσμάτων αλγορίθμου/ων

Η μέθοδος **showMapDialog** χρησιμοποείται για την εμφάνιση AlertDialog με περιεχόμενο έναν χάρτη με την λύση που βρήκε ο αλγόριθμος. Δέχεται ως είσοδο το αποτέλεσμα του αλγορίθμου υπό μορφή αντικειμένου TspResult και το BuildContext, απαραίτητο για την εμφάνιση του AlertDialog. Μαζί με

τον χάρτη εμφανίζεται και ένα κουμπί “OK” για την απόκρυψη του παραθύρου. Καλείται από το την onPressed του κουμπιού στην μέθοδο showTourOnMap.



Σχήμα 3.7: AlertDialog χάρτη με την βέλτιστη διαδρομή

Η μέθοδος **showTourOnMap** επιστρέφει κουμπί TextButton widget με κείμενο “Show tour on map”. Με την ιδιότητα onPressed εκτελείται κώδικας με την εντολή Helper.showMapDialog για την εμφάνιση AlertDialog με την λύση του αλγορίθμου σε χάρτη. Καλείται από την μέθοδο showMapDialog.

Η μέθοδος **showChartsBtn** επιστρέφει κουμπί TextButton widget με κείμενο “Show Charts”. Με την ιδιότητα onPressed εκτελείται κώδικας με την εντολή Navigator.push για την ανακατεύθυνση του χρήστη στην σελίδα ChartsScreen. Καλείται από την μέθοδο showMyDialog.

Η μέθοδος **tspSolutionText** καλείται από την CalculateScreen για την δημιουργία του κειμένου που θέλουμε να εμφανιστεί στο AlertDialog όταν κληθεί η Helper.showMyDialog. Δημιουργεί κείμενο String το οποίο περιέχει το μήκος της βέλτιστης διαδρομής, τον χρόνο εκτέλεσης του αλγορίθμου και τα φράγματα.

Η μέθοδος **algorithmsComparison** επιτελεί την ίδια λειτουργία με την tspSolutionText. Η διαφορά είναι ότι αυτή χρησιμοποιείται στην περίπτωση που εκτελέστηκαν όλοι οι αλγόριθμοι έτσι ώστε να δημιουργηθεί κείμενο String με τα αποτελέσματα όλων των αλγορίθμων.

Η μέθοδος **refreshUI** χρησιμοποιείται κατά την εκτέλεση του αλγορίθμου Held-Karp στο αρχείο held_karp.dart. Όταν κληθεί, σταματάει την εκτέλεση του αλγορίθμου για μερικά χιλιοστά του δευτερολέπτου με την εντωλή await Future.delayed. Είναι μια παράτυπη πρακτική που επινοήθηκε για την αντιμετώπιση ενός προβλήματος που δημιουργούσε η υψηλή πολυπλοκότητα του Held-Karp αλγορίθμου. Το πρόβλημα αυτό ήταν το «πάγωμα» του UI της εφαρμογής με αποτέλεσμα να μην ενημερώνεται η ένδειξη της προόδου εκτέλεσης του αλγορίθμου. Με την μέθοδο αυτή, απελευθερώνεται υπολογιστικοί πόροι για αρκετή ώρα ώστε να ανανεωθεί το UI. Ο λόγος που συμβαίνει αυτό είναι η απουσία υποστήριξης CPU πολυνημάτωσης (multithreading) για εφαρμογές ιστού με Flutter.

Η μέθοδος **getAlgoText** δέχεται ως είσοδο ένα String value και επιστρέφει String. Οι αλγόριθμοι σε πολλά σημεία του κώδικα ορίζονται μονολεκτικά για λόγους ευκολίας ως nearest, christofides, heldKarp και all. Η χρησιμότητα της μεθόδου είναι να γίνει η μετατροπή του μονολεκτικού αυτού String σε κάτι φιλικό προς τον χρήστη σε όποια σημεία του κώδικα καλείται για την εμφάνιση στο UI. Επί παραδείγματι το nearest επιστρέφει "Nearest Neighbor (Approximate)" και το all "All of them (Comparison)".

Η μέθοδος **resetVariables** χρησιμοποιείται για την αρχικοποίηση διάφορων βοηθητικών μεταβλητών μόλις τελειώσει η εκτέλεση της μεθόδου calculate στο calculate_screen.dart

Η μέθοδος **metersToKilometers** χρησιμοποιείται σε διάφορα αρχεία του project μας για την μετατροπή μέτρων σε χιλιόμετρα.

Η μέθοδος **createMarkerIcon** χρησιμοποιείται από την μέθοδο flutterMapWithTour για την δημιουργία του εικονίδιου του marker που τοποθείται στον χάρτη. Το εικονίδιο αυτό είναι σε σχήμα κύκλου και περιέχει τον αριθμό της πόλης. Σε περίπτωση που η πόλη αυτή είναι η αφετηρία (0), τότε το χρώμα του εικονιδίου καθορίζεται κόκκινο. Διαφορετικά είναι μπλε.

Αρχείο data.dart

Στον αρχείο αυτό έχουμε την κλάση **Data** για την αποθήκευση δεδομένων στα οποία θέλουμε να έχουμε εύκολη πρόσβαση από οποιοδήποτε σημείο του κώδικα, θέτοντάς τα ως static. Στην κλάση αυτή έχουμε ορίσει τα εξής:

- List<DragMarker> locations για τους markers στον χάρτη
- String selectedAlgo για τον επιλεγμένο αλγόριθμο στο dropdown menu
- List<List<double>> distanceMatrix για τον πίνακα αποστάσεων
- Duration timeWasted βοηθητική μεταβλητή για την μέθοδο refreshUI στο αρχείο helper.dart
- List<TspResult> allAlgorithms για την αποθήκευση των αποτελεσμάτων εκτέλεσης όλων των αλγορίθμων

3.3.7 Φάκελος αλγορίθμων

Στον φάκελο αυτό έχουμε συγκεντρώσει τους αλγορίθμους τους οποίους υλοποιήσαμε στην εφαρμογής μας. Πρέπει να σημειωθεί ότι οι πόλεις του προβλήματος στον κώδικα των αλγορίθμων αναπαριστώνται με έναν ακέραιο αριθμό (τύπου int) ο οποίος καθορίζεται από την σειρά με την οποία επέλεξε ο χρήστης τα σημεία ενδιαφέροντος στον χάρτη.

Αρχείο /algorithms/tsp_bounds.dart

Η μέθοδος **tspLowerBound** χρησιμοποιείται για τον υπολογισμό του κάτω φράγματος του TSP. Δέχεται ως είσοδο τον πίνακα αποστάσεων (distanceMatrix) και επιστρέφει τιμή double η οποία είναι το κάτω φράγμα σε μέτρα. Για τον υπολογισμό του ακολουθούνται τα παρακάτω βήματα:

1. Αφαιρούμε επαναληπτικά κάθε κορυφή από τον πίνακα αποστάσεων
2. Βρίσκουμε το Residual Minimum Spanning Tree (RMST) με τον αλγόριθμο του Prim καλώντας την μέθοδο primsFunc στο αρχείο prims.dart και παίρνοντας ως παράμετρο τον πίνακα αποστάσεων με την αφαιρεμένη κορυφή. Στην συνέχεια βρίσκουμε την συνολική απόσταση ή βάρος του RMST (mstWeight).
3. Προσθέτουμε στο mstWeight τα μήκη των δύο μικρότερων ακμών που συνδέονταν στην αφαιρεμένη κορυφή.
4. Το μεγαλύτερο από όλα τα mstWeight θεωρούμε ότι είναι το κάτω φράγμα [14].

Η μέθοδος **tspUpperBound** υπολογίζει το άνω φράγμα του TSP. Είσοδος είναι ο πίνακας αποστάσεων (distancematrix) και έξοδος είναι τιμή double η οποία είναι το άνω φράγμα σε μέτρα. Για τον υπολογισμό του άνω φράγματος χρησιμοποιούμε επαναληπτικά κάθε πόλη ως σημείο αφετηρίας στον αλγόριθμο του πλησιέστερου γείτονα (nearestNeighbor.dart). Από όλες τις περιηγήσεις που βρεθούν, θεωρούμε ως άνω φράγμα την περιοδεία με την μικρότερη διανυθείσα απόσταση [13] [11].

Η μέθοδος **shallowCopyList** δέχεται ως είσοδο μια λίστα και επιστρέφει ένα αντίγραφό της. Χρησιμοποιείται στην μέθοδο tspLowerBound ώστε να παραμείνει απείραχτη η αρχική λίστα distanceMatrix όταν αφαιρούμε κάποια κορυφή από τον πίνακα αυτό.

Αρχείο nearest_neighbor.dart

Η μέθοδος **nearestNeighborAlgorithm** είναι η υλοποίηση του Αλγορίθμου του Πλησιέστερου Γείτονα (Nearest Neighbor Algorithm). Δέχεται ως είσοδο έναν πίνακα αποστάσεων (List<List<double>> distanceMatrix) και τον αριθμό της πόλης αφετηρίας, ο οποίος από προεπιλογή είναι 0. Επιστρέφει τα αποτελέσματα του αλγορίθμου με ένα αντικείμενο τύπου TspResult.. Για τους σκοπούς του αλγορίθμου έχουμε δημιουργήσει στο αρχείο αυτό μια κλάση City με πεδία id και visited, την οποία χρησιμοποιούμε για να ελέγχουμε αν έχει επισκεφτεί μια πόλη ή όχι.

Πριν ξεκινήσουμε την εκτέλεση του αλγορίθμου χρησιμοποιούμε την κλάση Stopwatch για την καταγραφή του χρόνου εκτέλεσης του αλγορίθμου. Τα βασικά σημεία της μεθόδου είναι τα εξής:

1. Η List<int> solution αποθηκεύει την σειρά επίσκεψης των πόλεων, δηλαδή την βέλτιστη διαδρομή.
2. Η λίστα List<City> cities γεμίζει με τις πόλεις του προβλήματος καλώντας την μέθοδο createCitibiesObjList η οποία επιστρέφει αντικείμενα τύπου City.
3. Ξεκινάμε από την πόλη αφετηρίας προσθέτοντάς την στην διαδρομή (solution) και θέτοντάς την ως τρέχουσα πόλη (currentCity).
4. Θέτουμε την κοντινότερη πόλη αυτήν με την μεγαλύτερη απόσταση από την τρέχουσα (μέθοδος findMax).
5. Επαναληπτικά ελέγχουμε την απόσταση της τρέχουσας πόλης (currentCity) με καθεμία από τις υπόλοιπες πόλεις του προβλήματος (j). Παραλείπουμε τις περιπτώσεις όπου το j συμπίπτει με την πόλη αφετηρίας ή την τρέχουσα πόλη ή η j κατά σειρά πόλη στην λίστα cities έχει ήδη επισκεφτεί. Αν η απόσταση currentCity – j είναι μικρότερη από την πόλη που θέσαμε ως κοντινότερη στο βήμα 5, τότε θέτουμε ως κοντινότερη πόλη την j.
6. Προσθέτουμε στην διαδρομή (λίστα solution) την πόλη που βρήκαμε να είναι κοντινότερη στην τρέχουσα πόλη στο βήμα 6 και προσθέτουμε την απόσταση αυτή στην συνολική διανυθείσα απόσταση. Η πόλη αυτή γίνεται πλέον τρέχουσα πόλη και την θέτουμε ότι έχει επισκεφτεί.
7. Επαναλαμβάνουμε τα βήματα 5-7 έως ότου επισκεφθούμε όλες τις πόλεις του προβλήματος.
8. Επιστρέφουμε στην πόλη αφετηρίας από την τελευταία πόλη που ήμασταν. Την προσθέτουμε στην διαδρομή και στην συνολική διανυθείσα απόσταση.

Σταματάμε την καταγραφή του χρόνου εκτέλεσης και κατασκευάζουμε αντικείμενο TspResult με τα αποτελέσματα του αλγορίθμου το οποίο επιστρέφει η μέθοδος.

Η μέθοδος **createCitiesObjList** δέχεται ως είσοδο τον αριθμό των πόλεων και την πόλη αφετηρίας. Επιστρέφει λίστα με αντικείμενα τύπου City.

Η μέθοδος **findMax** δέχεται ως είσοδο την τρέχουσα πόλη στην εκτέλεση του αλγορίθμου, τον πίνακαν αποστάσεων και την πόλη αφετηρίας. Επιστρέφει την πόλη με την μεγαλύτερη απόσταση από την τρέχουσα πόλη.

Αρχείο held_karp.dart [55] [56]

Στο αρχείο αυτό έχουμε κατασκευάσει κλάσεις και μεθόδους για την υλοποίηση του αλγορίθμου Held-Karp.

Η μέθοδος **heldKarp** είναι η κύρια μέθοδος, αυτήν καλούμε από την solveTSP στο αρχείο tsp.dart. Είσοδος είναι ο πίνακας αποστάσεων List<List<double>> distance και έξοδος τα αποτελέσματα του αλγορίθμου σε ένα αντικείμενο TspResult. Τρέχει ασύγχρονα έτσι ώστε να αποτρέψουμε το πάγωμα του UI λόγω της μεγάλης πολυπλοκότητας του αλγορίθμου και περιορισμών στο web. Κάτι που χρησιμοποιείται αρκετά στην υλοποίηση αυτή είναι το HashMap. Αυτό είναι μια υλοποίηση του Map χρησιμοποιώντας έναν πίνακα κατακερματισμού, δηλαδή μια δομή δεδομένων για την αποθήκευση συνόλων στοιχείων [57]. Το Map είναι μια συλλογή από ζεύγη κλειδιού-τιμής από τα οποία γίνεται η ανάκτηση μια τιμής χρησιμοποιώντας ένα κλειδί [58]. Χρησιμοποιούμε την κλάση Stopwatch για την καταγραφή του χρόνου εκτέλεσης του αλγορίθμου. Η μέθοδος υλοποιεί τον αλγόριθμο ως εξής:

1. Δεδομένου της κορυφής 0 ως την πόλη αφετηρίας, παράγουμε όλα τα πιθανά υποσύνολα (allSets) με τις υπόλοιπες κορυφές του προβλήματος (generateCombination).
2. Επαναληπτικά για κάθε σύνολο (set) στα allSets, ελέγχουμε επαναληπτικά κάθε κορυφή του γραφήματος εκτός από την κορυφή αφετηρίας (0) και τις κορυφές που περιέχονται στο set αυτό. Βρίσκουμε τα κόστη μετάβασης στην κορυφή αυτή (currentVertex) από την τρέχουσα (prevVertex) διαμέσου όλων των υπολοίπων κορυφών στο set με την βοήθεια του πίνακα αποστάσεων και της μεθόδου getCost. Από αυτά κρατάμε το μικρότερο κόστος (minCost) και την κορυφή (prevVertex) από την οποία μεταβήκαμε στην επόμενη, αποθηκεύοντάς τα στους πίνακες κατακερματισμού minCostDP και parent.
3. Δημιουργούμε ένα σύνολο set με όλες τις κορυφές εκτός από την αφετηρία (0). Για κάθε κορυφή (k) στο σύνολο αυτό βρίσκουμε τα κόστη μετάβασης στην αφετηρία (0) διαμέσου όλων των υπόλοιπων κορυφών στο set με την βοήθεια του πίνακα αποστάσεων και της μεθόδου getCost, τα οποία έχουν ήδη υπολογιστεί στο προηγούμενο βήμα και βρίσκονται αποθηκευμένα στον minCostDP. Από τα κόστη αυτά συγκρατούμε το μικρότερο (min) και την κορυφή (prevVertex) από την οποία θα επιστρέψουμε στην αφετηρία (0), την οποία προσθέτουμε στον parent.
4. Το ελάχιστο κόστος (min) που βρήκαμε στο προηγούμενο βήμα είναι το μήκος της βέλτιστης διαδρομής που βρήκε ο αλγόριθμος για το γράφημα του προβλήματος του πλανόδιου πωλητή.
5. Βρίσκουμε ποια είναι η διαδρομή αυτή με την μέθοδο getSolution.

Σταματάμε την καταγραφή του χρόνου εκτέλεσης και επιστρέφουμε αντικείμενο TspResult με τα ευρήματα του αλγορίθμου. Μην παραλείψουμε να αναφέρουμε ότι στην παραπάνω μέθοδο χρησιμοποιούμε σε ορισμένα σημεία κατά την εκτέλεση του αλγορίθμου μια βοηθητική μέθοδο refreshUI, η οποία βρίσκεται στο αρχείο helper.dart. Περισσότερα για αυτήν θα πούμε σε επόμενη ενότητα για τον αρχείο στο οποίο βρίσκεται.

Η μέθοδος **generateCombination**, σε συνέργεια με τις μεθόδους generateCombination2 setSizeComparator και createSet , χρησιμοποιείται για την δημιουργία συνόλων με όλους τους πιθανούς συνδυασμούς κορυφών. Είσοδος είναι ένας αριθμός int και έξοδος μια λίστα με Set<int>.

Η μέθοδος **getCost** δέχεται ως είσοδο ένα Set<int> set, μια κορυφή int prevVertex και τον πίνακα κατακερματισμού Map<Index, double> minCostDP. Χρησιμοποιείται στην μέθοδο heldKarp για την εύρεση του κόστους από την κορυφή prevVertex στον minCostDP, το οποίο και επιστρέφει.

Η μέθοδος **getSolution** βρίσκει την βέλτιστη διαδρομή ψάχνοντας την κάθε κορυφή στον πίνακα κατακερματισμού parent. Δέχεται ως είσοδο τον Map<Index, int> parent και το πλήθος των κορυφών int totalVertices. Επιστρέφει σε δομή δεδομένων ουράς την βέλτιστη διαδρομή (Queue<int>).

Η κλάση **Index** χρησιμοποιείται βοηθητικά για την αποθήκευση ζεύγη κορυφών int currentVertex και συνόλων Set<int>.

Φάκελος christofides/ [59]

Στον φάκελο αυτό περιέχονται όλα τα αρχεία τα οποία σχετίζονται με την υλοποίηση του αλγορίθμου του Χριστοφίδη. Βασικό αρχείο της υλοποίησης είναι το christofides.dart.

Αρχείο christofides.dart

Στο αρχείο αυτό έχουμε τις μεθόδους christofides, perfectMatching, createBipartiteGraphs, createEulerTour και createHamiltonianTour.

Η μέθοδος **christofides** είναι η κύρια μέθοδος η οποία καλείται από την solveTSP στο αρχείο tsp.dart. Αυτή δέχεται ως είσοδο των πίνακα αποστάσεων (List<List<double>> distanceMatrix) και επιστρέφει τα αποτελέσματα του αλγορίθμου σε αντικείμενο TspResult. Ομοίως με τις προηγούμενες μεθόδους αλγορίθμων, πριν και μετά την εκτέλεση των διεργασιών του αλγορίθμου γίνεται καταγραφή του χρόνου εκτέλεσης με την κλάση Stopwatch. Ο τρόπος λειτουργίας της μεθόδου είναι ο εξής:

1. Κατασκευάζουμε ένα ελάχιστο συνδετικό δέντρο (List mst) και βρίσκουμε τις περιττές κορυφές (List<int> oddVertices) του γραφήματος καλώντας την μέθοδο primsFunc του αρχείου prims.dart.
2. Βρίσκουμε ένα minimum-weight perfect matching (List matches) καλώντας την μέθοδο perfectMatching με παραμέτρους τον πίνακα αποστάσεων distanceMatrix και τις περιττές κορυφές oddVertices.
3. Δημιουργούμε μια διαδρομή Euler (List<int> euler) καλώντας την μέθοδο createEulerTour με παραμέτρους την λίστα mst του βήματος 1 και την λίστα matches του βήματος 2.
4. Μετατρέπουμε την διαδρομή Euler που βρήκαμε στο βήμα 3 σε ένα μονοπάτι Hamilton με shortcutting καλώντας την μέθοδο createHamiltonianTour με παράμετρο την λίστα euler. Το μονοπάτι Hamilton (List<int> hamiltonian) είναι η βέλτιστη διαδρομή που βρίσκει ο αλγόριθμος.
5. Υπολογίζουμε το μήκος της διαδρομής αυτής (distance) με την βοήθεια του πίνακα αποστάσεων (distanceMatrix).

Η μέθοδος **perfectMatching** χρησιμοποιείται για την εύρεση minimum-weight perfect matching με brute force προσέγγιση. Είσοδος είναι ένας πίνακας αποστάσεων List<List<double>> adjMatrix και λίστα List<int> oddVertices με περιττές κορυφές. Επιστρέφει το minimum-weight perfect matching ως λίστα. Χρησιμοποιεί τις μεθόδους createBipartiteGraphs (christofides.dart) και την munkres (munkres.dart)

Η μέθοδος **createBipartiteGraphs** είναι μια μέθοδος-γεννήτρια η οποία δημιουργεί όλες τις πιθανές παραλλαγές διμερών γραφημάτων (bipartite graphs) από άρτιο σύνολο κορυφών. Καλείται από την μέθοδο perfectMatching. Είσοδος της μεθόδου είναι ο πίνακας αποστάσεων, δηλαδή τα μήκη των ακμών του γραφήματος (List<List<double>> edgeWeights) και η λίστα με τις περιττές κορυφές (List<int> oddVertices). Χρησιμοποιεί την μέθοδο combinations (lists.dart).

Αρχείο prims.dart

Εδώ υλοποιείται ο αλγόριθμος του Prim με την μέθοδο primsFunc. Δέχεται ως είσοδο τον πίνακα αποστάσεων List<List<double>> distanceMatrix και επιστρέφει λίστα που περιέχει ένα ελάχιστο συνδετικό δέντρο (mst) και τις περιττές κορυφές (odds).

Αρχείο lists.dart

Στο αρχείο lists.dart βρίσκεται η μέθοδος combinations. Και αυτή είναι μια μέθοδος-γεννήτρια όπως η createBipartiteGraphs. Καλείται από την createBipartiteGraphs με παραμέτρους λίστα με τις περιττές κορυφές και το πλήθος των περιττών κορυφών διαιρεμένο με το 2.

Αρχείο munkres.dart [60]

Στο αρχείο **munkres.dart** βρίσκεται η μέθοδος munkres η οποία καλείται από την perfectMatching. Η munkres δέχεται ως είσοδο έναν πίνακα αποστάσεων και επιστρέφει μια λίστα List<List<int>> με τα minimum matches. Αυτά υπολογίζονται καλώντας την μέθοδο compute της κλάσης Munkres. Στην κλάση Munkres έχουμε υλοποιήσει τον Hungarian αλγόριθμο, γνωστό και ως Munkres assignment αλγόριθμος, ο οποίος βρίσκει minimum matching.

Αρχείο fleury.dart

Η μέθοδος fleury καλείται από την createEulerTour. Δέχεται ως είσοδο ένα multigraph, δηλαδή μια λίστα List<List> αποτελούμενη από το MST και matched περιττές κορυφές. Επιστρέφει το Euler path σε λίστα List<int>. Η μέθοδος αυτή μαζί με άλλες μεθόδους στο αρχείο αυτό υλοποιούν τον αλγόριθμο του Fleury, ο οποίος χρησιμοποιείται για την εύρεση ενός μονοπατιού Euler.

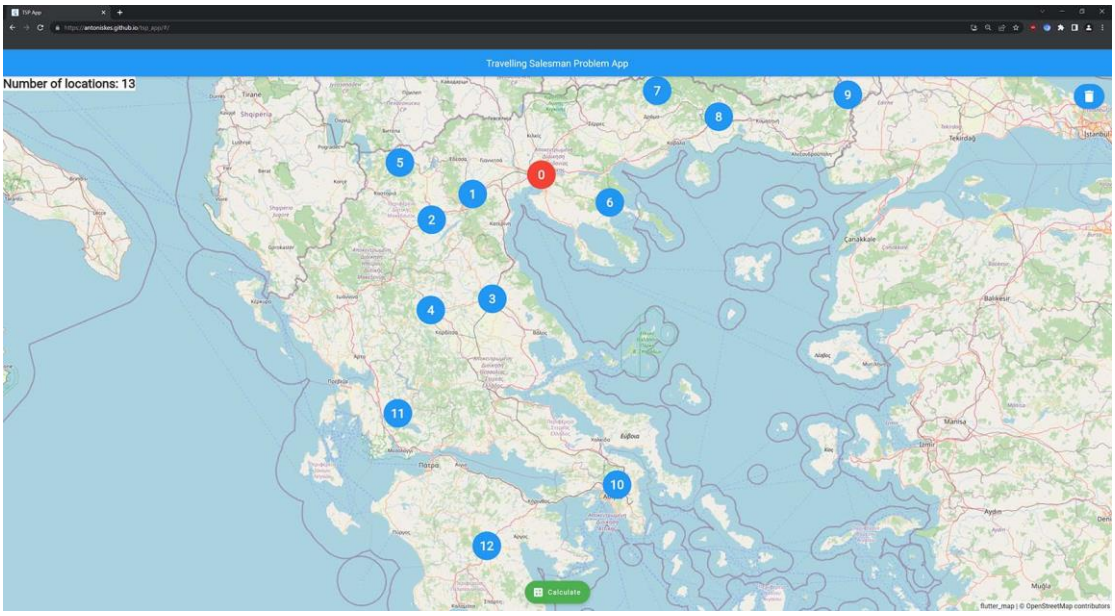
3.3.8 Αρχεία μοντέλων

Αρχείο /models/tsp_result.dart

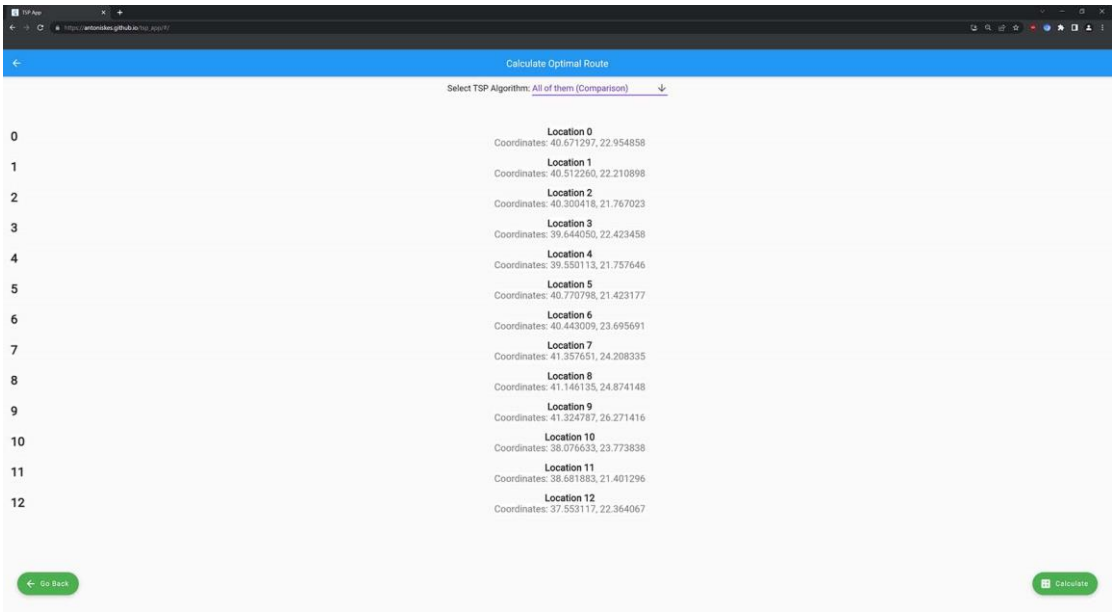
Ορίζουμε κλάση **TspResult** την οποία χρησιμοποιούμε στην εφαρμογή μας για την αποθήκευση των αποτελεσμάτων εκτέλεσης ενός αλγορίθμου σε αντικείμενα τύπου TspResult. Ένα τέτοιο αντικείμενο θα έχει πεδία που υποδεικνύουν τον αλγόριθμο (String algorithm), την λύση που βρέθηκε (List<int> tour), το μήκος της λύσης σε μέτρα (double tourLength) και τον χρόνο εκτέλεσης του αλγορίθμου (Duration executionTime). Στην κλάση αυτή ορίζουμε επίσης τον constructor και μια μέθοδο εκτύπωσης των πεδίων που αναφέραμε στην κονσόλα (printResult) για λόγους debugging.

3.4 Παρουσίαση ενδεικτικών αποτελεσμάτων εκτέλεσης

Σε αυτό το κεφάλαιο θα παρουσιάσουμε τα αποτελέσματα εκτέλεσης της εφαρμογής μας. Επιλέξαμε ενδεικτικά 13 τυχαίες πόλεις στον χάρτη για τις οποίες θα εκτελέσουμε και τους 3 αλγορίθμους που υλοποιήσαμε στην εφαρμογή μας. Αυτοί είναι ο Αλγόριθμός του Πλησιέστερου Γείτονα, ο Αλγόριθμος του Χριστοφίδη και ο Αλγόριθμος Held-Karp.

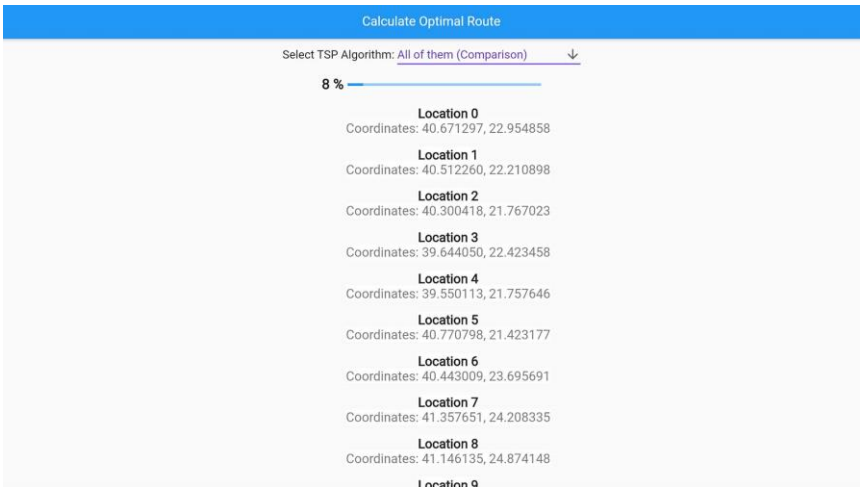


Σχήμα 3.8: Επιλεγμένα σημεία ενδιαφέροντος στον χάρτη



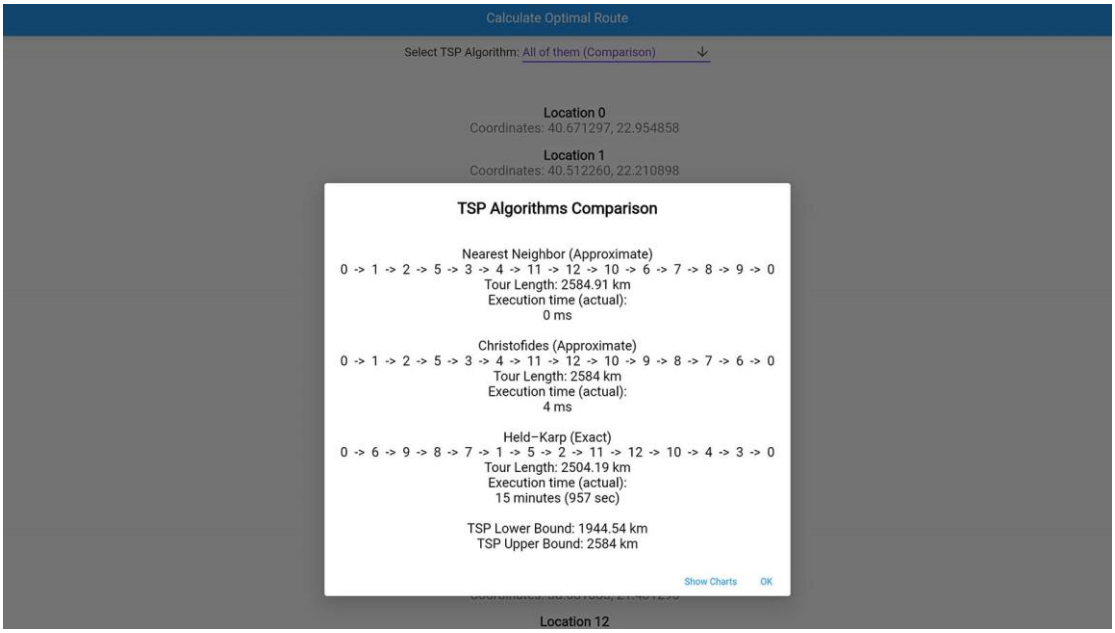
Σχήμα 3.9: Λίστα τοποθεσιών - Επιλογή εκτέλεσης όλων των αλγορίθμων

Μόλις πατήσουμε το κουμπί Calculate κάτω δεξιά, ξεκινάει η εκτέλεση και των 3 αλγορίθμων. Εν τω μεταξύ, εμφανίζεται κάτω από το dropdown menu μια μπάρα προόδου με το ποσοστό κατά το οποίο ολοκληρώθηκαν οι διεργασίες της εκτέλεσης των αλγορίθμων.



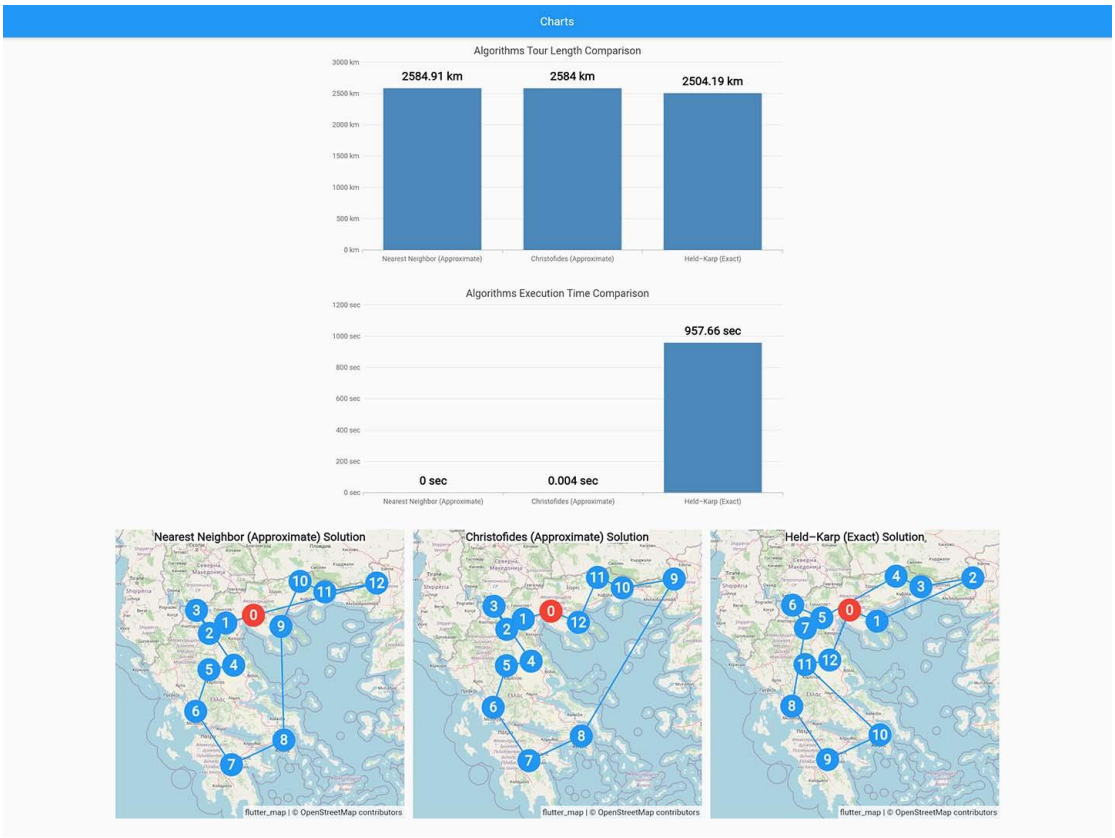
Σχήμα 3.10: Γραφική ένδειξη προόδου εκτέλεσης αλγορίθμων

Μετά από αρκετή ώρα ολοκληρώνεται η εκτέλεση των αλγορίθμων και εμφανίζεται AlertDialog με αποτελέσματα.



Σχήμα 3.11: Αποτελέσματα εκτέλεσης αλγορίθμων

Διαπιστώνουμε ότι ο χρόνος εκτέλεσης των προσεγγιστικών αλγορίθμων Πλησιέστερου Γείτονα και του Χριστοφίδη ήταν απειροελάχιστος σε σύγκριση με τον χρόνο που χρειάστηκε ο ακριβής αλγόριθμος Held-Karp για την εύρεση της βέλτιστης διαδρομής. Αυτό οφείλεται στην υψηλή περιπλοκότητα του αλγορίθμου σε σχέση με τους υπόλοιπους, πράγμα που δικαιολογείται από το γεγονός ότι βρίσκει την καλύτερη λύση. Πατώντας το κουμπί Show Charts μπορούμε να δούμε τα αποτελέσματα σε μορφή γραφημάτων καθώς και την λύση του κάθε αλγορίθμου (σειρά απίσκεψης πόλεων) πάνω στον χάρτη.



Σχήμα 3.12: Γραφήματα μήκους διαδρομών και χρόνων εκτέλεσης

Κεφάλαιο 4ο: Συμπεράσματα και προτάσεις βελτίωσης

4.1 Συμπεράσματα

Στην παρούσα εργασία έγινε μια επισκόπηση της θεωρίας του προβλήματος του πλανόδιου πωλητή και των μεθοδολογιών λύσης του. Υπάρχουν διάφοροι αλγόριθμοι για την επίλυση του προβλήματος, ακριβείς και προσεγγιστικοί. Οι ακριβείς αλγόριθμου απαιτούν μεγάλο χρονικό διάστημα για την εύρεση της άριστης λύσης, χρόνος ο οποίος γίνεται όλο και μεγαλύτερος όσο αυξάνονται οι πόλεις του προβλήματος κάτι το οποίο δεν είναι πρακτικό όταν χρειαζόμαστε να βρούμε λύση σύντομα. Σε μεγάλα μεγέθους προβλήματα αναγκαζόμαστε να επιλέγουμε προσεγγιστικούς αλγόριθμους, ο καθένας από αυτούς βρίσκοντας την δική του «βέλτιστη» διαδρομή προσεγγιστικά, σε ανεκτό για τον άνθρωπο χρόνο. Ωστόσο, πρέπει να αξιολογούμε τις προσεγγιστικές αυτές λύσεις που βρίσκουν τέτοιοι αλγόριθμοι με τον υπολογισμό των φραγμάτων έτσι ώστε να προσδιορίσουμε πόσο κοντά ή πόσο μακριά κυμαίνεται η λύση αυτή από την πραγματικά βέλτιστη διαδρομή.

Στην εφαρμογή που αναπτύξαμε υλοποιήσαμε δύο προσεγγιστικούς αλγορίθμους, του Χριστοφίδη και του Πλησιέστερου Γείτονα και έναν ακριβή αλγόριθμο, τον Held-Karp. Εκτελώντας τους αλγορίθμους αυτούς διαπιστώσαμε ότι οι προσεγγιστικοί αλγόριθμοι αν και δεν βρίσκουν την καλύτερη λύση στο TSP, βρίσκουν όμως μία κοντά σε αυτήν σχεδόν στιγμιαία σε προβλήματα μεσαίου μεγέθους ή σε πολύ μικρό χρόνο σε μεγάλου μεγέθους προβλήματα. Εν αντιθέσει, οι ακριβείς αλγόριθμοι και ειδικά στην περίπτωση μας ο Held-Karp λόγω της μεγάλης πολυπλοκότητάς τους για την εύρεση άριστης λύσης, έχουν πολύ υψηλές απαιτήσεις σε υπολογιστικούς πόρους και σε χρόνο εκτέλεσης.

4.2 Προτάσεις βελτίωσης

Ένα βήμα παραπέρα για την βελτίωση της εργασίας θα ήταν η αναφορά σε περισσότερους αλγορίθμους που υπάρχουν μέχρι σήμερα για την επίλυση του TSP, ιδιαίτερα σε μερικούς που αναπτύχθηκαν τα τελευταία χρόνια για την βελτίωση ήδη υπάρχοντών αλγορίθμων. Επίσης, θα μπορούσαμε να αναφερθούμε τις συσχετίσεις του προβλήματος του πλανόδιου πωλητή με σημερινά προβλήματα της καθημερινότητας καθώς και στην εφαρμογή των μεθοδολογιών επίλυσής του σε τέτοια προβλήματα. Όσον αφορά την εφαρμογή που αναπτύξαμε, θα μπορούσε να βελτιωθεί ως εξής:

- Υλοποίηση περισσότερων αλγορίθμων.
- Επίλυση του προβλήματος με περισσότερους περιορισμούς εκτός της απόστασης όπως το κόστος σε χρόνο ή σε χρήμα καθώς και διορίες επίσκεψης σε ορισμένες πόλεις.
- Χρήση της εφαρμογής σε περισσότερες πλατφόρμες (Windows, Android, iOS) αξιοποιώντας την λειτουργία του Flutter framework για χτίσιμο της εφαρμογής σε πολλές πλατφόρμες εκτός του web, χρησιμοποιώντας τον ίδιο κώδικα.
- Καλύτερη αξιοποίηση των υπολογιστικών πόρων του υπολογιστή κατά την εκτέλεση των αλγορίθμων με χρήση multithreading, κάτι που δεν υποστηρίζεται εγγενώς από το Flutter.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Cook, W.J., In Pursuit of the Traveling Salesman: Mathematics at the Limit of Computation, Princeton, NJ: Princeton UP, 2012.

[2] Bonyadi, M.R., Aghadi, M.R., & Shah-Hosseini, H., "Population-Based Optimization Algorithms for Solving the Travelling Salesman Problem," in *Travelling Salesman Problem*, In-Tech, 2008.

[3] Hasler, M., & Hornik, K. , "TPS- Infrastructure for the Traveling Salesperson Problem," *Journal of Statistical Software, 23(2),* 2008.

[4] Lewis, R.H., & Papadimitriou, C.H. (Μετάφραση Αντωνιάδης, Π.), Στοιχεία Θεωρίας Υπολογισμού, Κριτική, 2005.

[5] Sipser, M. (Μετάφραση Παγουρτζής, Α.Θ.), Εισαγωγή στη Θεωρία του Υπολογισμού, ΠΕΚ (Πανεπιστημιακές Εκδόσεις Κρήτης), 2012.

[6] J. Scholz, "Genetic Algorithms and the Traveling Selesman Problem. A hisorical review," *ACM Trans.Web,* 2018.

[7] F. Greco, Traveling Salesman Problem, In-teh, 2008.

[8] C. Brucato, "THE TRAVELING SALESMAN PROBLEM," University of Pittsburgh, 2013.

[9] Papadimitriou, C.H., Raghavan, P., Tamaki, H., & Vempala, S., "Latent Semantic Indexing: A Probabilistic Analysis," *Journal of Computer and System Sciences, 61(2),* pp. 217-235, 2000.

[10] D. Applegate., The Traveling Salesman Problem: A Computational Study, Princeton University Press , 2006.

[11] «The Travelling Salesman Problem England,» [Ηλεκτρονικό]. Available: https://www.thechalkface.net/resources/Travelling_Salesman_England.pdf.

[12] N. Christofides, «Bounds for the Travelling-Salesman Problem,» *Operations Research, 20 (5),* pp. 1044-1056, 1972.

[13] «Travelling Salesman Problem - Upper Bound - Nearest Neighbour method,» [Ηλεκτρονικό]. Available: https://youtu.be/ojjnd5gEMukTo.

[14] «Travelling Salesman Problem - Lower Bound - Minimum Spanning Tree Method,» [Ηλεκτρονικό]. Available: https://youtu.be/VcugXirtWOo.

[15] «Held–Karp algorithm - Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Held-Karp_algorithm.

[16] T. Roughgarden, Algorithms Illuminated (Part 4): Algorithms for NP-Hard Problems, 2020.

[17] C. Tillmann και N. Hermann, «Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation,» *Computational Linguistics, 29 (1),* pp. 97-133, 2003.

[18] «Solving The Traveling Salesman Problem For Deliveries,» [Ηλεκτρονικό]. Available: https://blog.routific.com/blog/travelling-salesman-problem.

[19] «13. Case Study: Solving the Traveling Salesman Problem - CMPT 125 Summer 2012 1 documentation,» [Ηλεκτρονικό]. Available: https://www2.cs.sfu.ca/CourseCentral/125/tjd/tsp_example.html.

[20] «Christofides algorithm - Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Christofides_algorithm.

[21] «Spanning Tree and Minimum Spanning Tree - Programiz,» [Ηλεκτρονικό]. Available: https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree.

[22] «Eulerian path and circuit for undirected graph - GeeksforGeeks,» [Ηλεκτρονικό]. Available: https://www.geeksforgeeks.org/eulerian-path-and-circuit/.

[23] M. T. Goodrich και R. Tamassia, Algorithm Design and Applications, Wiley, 2014.

[24] K. Arora, S. Agarwal και R. Tanwar, «Solving TSP using Genetic Algorithm and Nearest Neighbour Algorithm and their Comparison,» *International Journal of Scientific & Engineering Research, 7 (1),* 2016.

[25] H. A. Abdulkarim και I. F. Alshammari, «Comparison of Algorithms for Solving Traveling Salesman Problem,» *International Journal of Engineering and Advanced Technology, 4 (6),* 2015.

[26] «Visual Studio Code,» [Ηλεκτρονικό]. Available: https://code.visualstudio.com/.

[27] «Git,» [Ηλεκτρονικό]. Available: https://git-scm.com/.

[28] «Git - Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Git.

[29] «Flutter (software) - Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Flutter_(software).

[30] «FAQ - Flutter,» [Ηλεκτρονικό]. Available: https://docs.flutter.dev/resources/faq#what-is-flutter.

[31] «OpenStreetMap,» [Ηλεκτρονικό]. Available: https://www.openstreetmap.org/.

[32] «flutter_map | Flutter Package,» [Ηλεκτρονικό]. Available: https://pub.dev/packages/flutter_map.

[33] «flutter_map_dragmarker | Flutter Package,» [Ηλεκτρονικό]. Available: https://pub.dev/packages/flutter_map_dragmarker.

[34] «open_route_service | Flutter Package,» [Ηλεκτρονικό]. Available: https://pub.dev/packages/open_route_service.

[35] «syncfusion_flutter_charts | Flutter Package,» [Ηλεκτρονικό]. Available: https://pub.dev/packages/syncfusion_flutter_charts.

[36] «What is widgets in Flutter - GeeksforGeeks,» [Ηλεκτρονικό]. Available: https://www.geeksforgeeks.org/what-is-widgets-in-flutter/.

[37] «Beginning Flutter — Understanding the Widget Tree,» [Ηλεκτρονικό]. Available: https://medium.com/@JediPixels/beginning-flutter-understanding-the-widget-tree-3513c94dc356.

[38] «Flutter widgets - JavaPoint,» [Ηλεκτρονικό]. Available: https://www.javatpoint.com/flutter-widgets.

[39] «What is Widget in flutter ? Let’s clear the basics first - Medium,» [Ηλεκτρονικό]. Available: https://medium.com/jay-tillu/4-what-is-widget-in-flutter-lets-clear-the-basics-first-82f501c8d0f0.

[40] «The main function - Flutter by Example,» [Ηλεκτρονικό]. Available: https://flutterbyexample.com/lesson/the-main-function.

[41] «runApp function - widgets library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/widgets/runApp.html.

[42] «build method - Stateless Widget class - widgets library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/widgets/StatelessWidget/build.html.

[43] «build method - state c;lass - widgets library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/widgets/State/build.html.

[44] «What is the Difference Between Scaffold and MaterialApp In Flutter?,» [Ηλεκτρονικό]. Available: https://flutteragency.com/what-is-the-difference-between-scaffold-and-materialapp-in-flutter/.

[45] «Adding interactivity to your Flutter app | Flutter,» [Ηλεκτρονικό]. Available: https://docs.flutter.dev/development/ui/interactive.

[46] «StatefulWidget class - widgets library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html.

[47] «Flutter and the pubspec file | Flutter,» [Ηλεκτρονικό]. Available: https://docs.flutter.dev/development/tools/pubspec.

[48] «Customizing web app initialization | Flutter,» [Ηλεκτρονικό]. Available: https://docs.flutter.dev/development/platform-integration/web/initialization.

[49] «Scaffold class - material library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/material/Scaffold-class.html.

[50] «Flutter - Stack Widget - GeeksforGeeks,» [Ηλεκτρονικό]. Available: https://www.geeksforgeeks.org/flutter-stack-widget/.

[51] «Navigate to a new screen and back | Flutter,» [Ηλεκτρονικό]. Available: https://docs.flutter.dev/cookbook/navigation/navigation-basics.

[52] «Navigator class - widgets library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/widgets/Navigator-class.html.

[53] «Navigation and routing | Flutter,» [Ηλεκτρονικό]. Available: https://docs.flutter.dev/development/ui/navigation.

[54] «Getting started with Flutter Cartesian Charts widget | Syncfusion,» [Ηλεκτρονικό]. Available: https://help.syncfusion.com/flutter/cartesian-charts/getting-started.

[55] «Traveling Salesman Held Karp Java - Github,» [Ηλεκτρονικό]. Available: https://github.com/mission-peace/interview/blob/master/src/com/interview/graph/TravelingSalesmanHeldKarp.java.

[56] « Traveling Salesman Problem Dynamic Programming Held-Karp - Youtube,» [Ηλεκτρονικό]. Available: https://youtu.be/-JjA4BLQyqE.

[57] «Hash table - Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Hash_table.

[58] «Map class - dart:core library - Dart API,» [Ηλεκτρονικό]. Available: https://api.flutter.dev/flutter/dart-core/Map-class.html.

[59] «tsp-node - Github,» [Ηλεκτρονικό]. Available: https://github.com/ggat/tsp-node.

[60] «munkres-js - Github,» [Ηλεκτρονικό]. Available: https://github.com/addaleax/munkres-js.

[61] Applegate, D. L., Bixby, R. E., Chvatal, V., & Cook, W. J. , The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2007.

[62] «Plugins List - flutter_map Documentation,» [Ηλεκτρονικό]. Available: https://docs.fleaflet.dev/plugins/list.