

Γραμμική και συνδυαστική βελτιστοποίηση – Τελική εργασία

Τίτλος εργασίας: Μοντελοποίηση με γραμμικό / ακέραιο προγραμματισμό του προβλήματος εύρεσης της συντομότερης διαδρομής (shortest path problem) μεταξύ δύο σημείων στο οδικό δίκτυο μιας πόλης με σκοπό την έγκαιρη μετάβαση ασθενή στο νοσοκομείο.

Όνοματεπώνυμο: Πρίντζιος Λάμπρος

Αριθμός μητρώου: 1072817

Έτος: 4^ο

Υπεύθυνη καθηγήτρια: Σοφία Δασκαλάκη

Περιεχόμενα:

Εισαγωγή – Γενική περιγραφή του προβλήματος shortest path:	2
Δημιουργία γραφικής διεπαφής με τη βιβλιοθήκη tkinter της python:.....	3
Μοντελοποίηση του shortest path με γραμμικό / ακέραιο προγραμματισμό:	11
Ενδεικτικά παραδείγματα επίλυσης του προβλήματος στη γραφική διεπαφή:	18
Εφαρμογή ανάλυσης ευαισθησίας και χαρακτηριστικό παράδειγμα:	24
Επίλυση του shortest path με τον αλγόριθμο αναζήτησης A* (A star):.....	32
Παραπομπές – Βιβλιογραφία:	34
Κώδικας και οδηγίες εκτέλεσής του:	34

Εισαγωγή – Γενική περιγραφή του προβλήματος shortest path:

Όπως φανερώνει ο τίτλος της εργασίας, στόχος μου είναι η μελέτη του προβλήματος εύρεσης της βραχύτερης/οικονομικότερης διαδρομής ή, όπως αναφέρεται στη διεθνή βιβλιογραφία, του shortest path problem [1]. Πρόκειται για ένα πολύ γνωστό πρόβλημα βελτιστοποίησης της θεωρίας γράφων, το οποίο ασχολείται με την αναζήτηση και εύρεση σε έναν γράφο εκείνης της διαδρομής μεταξύ δύο προκαθορισμένων διακριτών κορυφών, η οποία συνίσταται από ακμές με το ελάχιστο δυνατό άθροισμα βαρών (με άλλα λόγια, της διαδρομής με το ελάχιστο συνολικό κόστος). Ο όρος “διαδρομή” υπονοεί ότι επιβάλλεται να γίνεται το πολύ μία διέλευση από κάθε κορυφή του γράφου, καθώς αυτή ακριβώς η διατύπωση αποτελεί τον ορισμό της διαδρομής – path σε γράφο.

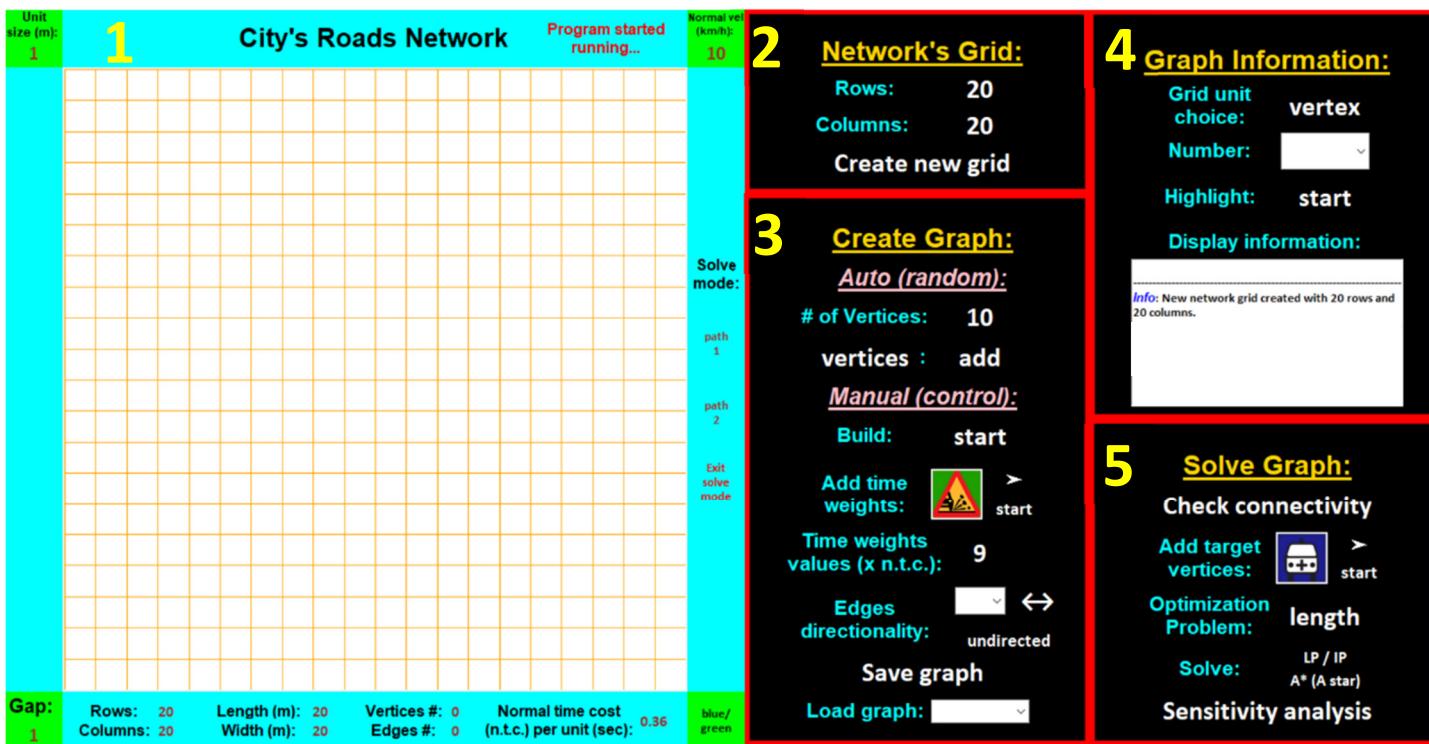
Σύμφωνα με τον Alexander Schrijver, που είναι Ολλανδός καθηγητής μαθηματικών και επιστήμης των υπολογιστών του πανεπιστημίου του Άμστερνταμ με ερευνητικό έργο στα διακριτά μαθηματικά και στη θεωρία βελτιστοποίησης, το πρόβλημα του shortest path γνώρισε ιδιαίτερα μεγάλη ερευνητική μελέτη και πρόοδο τη δεκαετία του 1950 [2]. Αρχικά, ο Shimbel επιχείρησε το 1953 να αντιμετωπίσει το πρόβλημα στην περίπτωση των αβαρών μη κατευθυνόμενων γράφων, χρησιμοποιώντας πολλαπλασιασμούς πινάκων. Ο ίδιος, το 1955, έδωσε μία περιγραφή που θα αποδεικνύταν μαθηματικώς ισοδύναμη με τη μέθοδο των Bellman – Ford. Την ίδια χρονιά, ο Orden ήταν ο πρώτος που διαπίστωσε ότι το πρόβλημα εύρεσης της συντομότερης διαδρομής αποτελεί μια ειδική περίπτωση προβλήματος ροής ελαχίστου κόστους σε δίκτυο, συνεπώς μπορεί να μοντελοποιηθεί και να επιλυθεί με γραμμικό προγραμματισμό. Το 1956 πραγματοποιήθηκε από τον Ford ένα αποφασιστικό βήμα προς την εφαρμογή αλγορίθμικών διαδικασιών για την επίλυση του προβλήματος, με την κατασκευή ενός γενικού μαθηματικού οικοδομήματος στο οποίο βασίστηκε η ανάπτυξη των αλγορίθμων Dijkstra (το 1959 ο Dijkstra παρουσίασε μία συνοπτική και σαφή περιγραφή του ομώνυμου αλγορίθμου για κατευθυνόμενους γράφους με μη αρνητικά βάρη) και Bellman – Ford (λιγότερο αποδοτικός από τον Dijkstra, αλλά χωρίς τον περιορισμό των μη αρνητικών βαρών). Επίσης το 1956, ο Rosenfeld έδωσε μια ευρετική προσέγγιση στην προσπάθεια εύρεσης της βέλτιστης διαδρομής που δύναται να ακολουθήσουν οχήματα που βρίσκονται σε δρόμους υπό κυκλοφοριακή συμφόρηση (ανοίγοντας έτσι τον δρόμο για την ανάπτυξη ευρετικών αλγορίθμων, όπως είναι ο A* που θα μελετηθεί συνοπτικά στην ενότητα “Επίλυση του shortest path με τον αλγόριθμο αναζήτησης A* (A star)” της παρούσας εργασίας). Πολλοί ακόμα, όπως ο Moore, ο Dantzig και ο Fulkerson υπήρξαν με τη συμβολή τους σημαντικές προσωπικότητες στην ιστορία του shortest path προβλήματος.

Η εύρεση της συντομότερης διαδρομής είναι αναγκαία σε πολυάριθμες πραγματικές και τεχνολογικές συνήθως εφαρμογές, που εμφανίζουν άμεση σχέση με την καθημερινότητά μας. Για παράδειγμα, χρειάζεται στην εύρεση των συντομότερων χωρικά και χρονικά κατευθύνσεων σε οδικούς χάρτες, όπως είναι οι Google Maps. Εκτός από τα οδικά δίκτυα συναντάται και στα δίκτυα υπολογιστών (π.χ. στη δρομολόγηση IP πακέτων), στα τηλεφωνικά δίκτυα, στα δίκτυα ηλεκτρικής ενέργειας, καθώς και στα δίκτυα μεταφορών (χερσαίων, θαλάσσιων κ.τ.λ.). Δεν είναι επίσης λίγες οι εφαρμογές του στη ρομποτική, όπως στο πεδίο των αυτόνομων ρομποτικών οχημάτων που χρειάζεται να εξερευνούν με έξυπνο και βέλτιστο τρόπο το περιβάλλον τους (για

να το χαρτογραφήσουν ή να εκτελέσουν κάποια δράση σ' αυτό), καθώς και στην ανάπτυξη γραφικών σε ψηφιακούς εικονικούς κόσμους. Όλα τα αναφερόμενα προβλήματα, παρότι εντάσσονται σε ποικίλα και διαφορετικά πεδία δραστηριοτήτων, είναι δυνατόν να μοντελοποιηθούν και να αντιμετωπιστούν με παρόμοιο μαθηματικό τρόπο με τη χρήση γράφων, πολλές φορές εξαιρετικά μεγάλου μεγέθους με χιλιάδες ή και εκατομμύρια κορυφές και ακμές. Στην πλειοψηφία βέβαια των περιπτώσεων, τα μεγάλα αυτά προβλήματα αντιμετωπίζονται με την εφαρμογή αλγορίθμων διαδικασιών, όπως οι αλγόριθμοι A*, Dijkstra και διάφορες παραλλαγές τους.

Στις επόμενες σελίδες ακολουθεί μια εκτενής ανάλυση του προβλήματος εύρεσης της συντομότερης διαδρομής μεταξύ δύο σημείων στο οδικό δίκτυο μιας πόλης, στα πλαίσια του υποθετικού σεναρίου της έκτακτης ανάγκης μετακίνησης με ασθενοφόρο ενός ασθενή στο νοσοκομείο. Αρχικά, θα εξηγηθούν οι δυνατότητες της γραφικής διεπαφής που σχεδιάστηκε για τη δημιουργία του γράφου του οδικού δικτύου και στη συνέχεια θα μελετηθεί η μοντελοποίηση και η επίλυση του shortest path με γραμμικό προγραμματισμό (το κεντρικό μέρος της εργασίας), θα πραγματοποιηθεί μία ανάλυση ευαισθησίας στο μοντέλο και τέλος θα επιστρατεύσω τον αλγόριθμο A* για την αλγορίθμική αντιμετώπιση του προβλήματος. Θεωρήθηκε σκόπιμο να παρουσιαστεί πρώτα η γραφική διεπαφή, πριν τη μαθηματική ανάλυση, διότι η υλοποίησή της θέτει ορισμένους περιορισμούς που είναι απαραίτητο να ληφθούν υπόψη κατά τη διάρκεια κατασκευής του μαθηματικού μοντέλου.

Δημιουργία γραφικής διεπαφής με τη βιβλιοθήκη tkinter της python:

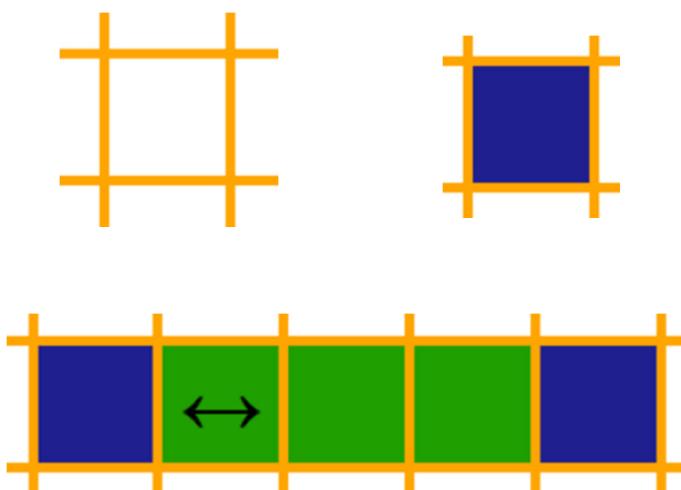


Εικόνα 1

Στην παραπάνω εικόνα 1 φαίνεται η γραφική διεπαφή που δημιουργήθηκε με τη βοήθεια της tkinter, η οποία αποτελεί την ενδεδειγμένη βιβλιοθήκη για την ανάπτυξη GUIs (Graphical User Interfaces) με τη γλώσσα προγραμματισμού python. Για την αποδοτικότερη μελέτη του γραφικού περιβάλλοντος έχουν αριθμηθεί οι πέντε σημαντικότερες αυτόνομες περιοχές της διεπαφής, η λειτουργικότητα και τα χαρακτηριστικά των οποίων εξετάζονται διεξοδικά αμέσως μετά. Το γραφικό περιβάλλον χωρίζεται σε δύο βασικά μέρη: στην περιοχή 1, όπου κατασκευάζονται οι γράφοι και προβάλλονται τα αποτελέσματα της επίλυσης του shortest path προβλήματος και στις περιοχές 2, 3, 4 και 5, οι οποίες ουσιαστικά αποτελούν τα διάφορα μενού του προγράμματος και του προσδίδουν λειτουργικότητα.

- Για την περιοχή 1 (City's Roads Network – Το οδικό δίκτυο της πόλης):

Πρόκειται για την κύρια περιοχή της γραφικής διεπαφής, την περιοχή σχεδίασης. Εδώ δημιουργείται το οδικό δίκτυο της πόλης, είτε τυχαία είτε με παρεμβάσεις και τροποποιήσεις από τον χρήστη. Στο κέντρο φαίνεται το μεταβλητό πλέγμα του οδικού δικτύου, πάνω στο οποίο μπορούν να σχεδιαστούν οι γράφοι. Το πλέγμα συνίσταται από ξεχωριστές μονάδες, οι οποίες μπορούν να μετατραπούν σε κορυφές ή μέρη ακμών του γράφου, όπως φαίνεται στην εικόνα 2. Στην κάτω γαλάζια ακριανή περιοχή του πλέγματος καταγράφονται χρήσιμες πληροφορίες για το εκάστοτε πλέγμα, όπως ο αριθμός των γραμμών και των στηλών του, καθώς και το μήκος και το πλάτος του σε μέτρα. Ακόμα, σημειώνεται ο αριθμός των κορυφών και των ακμών του γράφου, καθώς και το default χρονικό κόστος της κάθε μονάδας, εξαρτώνται από το μήκος της μονάδας του γράφου (unit size) που ορίζεται στην πάνω αριστερά πράσινη κορυφή και από την ταχύτητα διάνυσης της μονάδας (normal velocity) που ορίζεται στην πάνω δεξιά κορυφή. Επίσης, στις κάτω πράσινες κορυφές προσδιορίζονται κάποια σχεδιαστικά χαρακτηριστικά του γράφου, τα οποία είναι το χάσμα (gap) μεταξύ των μονάδων και τα χρώματα του γράφου (μπλε/πράσινο ή κόκκινο/κίτρινο για κορυφές/ακμές). Τέλος, η δεξιά γαλάζια ακριανή περιοχή σχετίζεται με την επίλυση του εκάστοτε γράφου και η λειτουργία της θα εξηγηθεί περισσότερο στην ανάλυση της περιοχής 5.

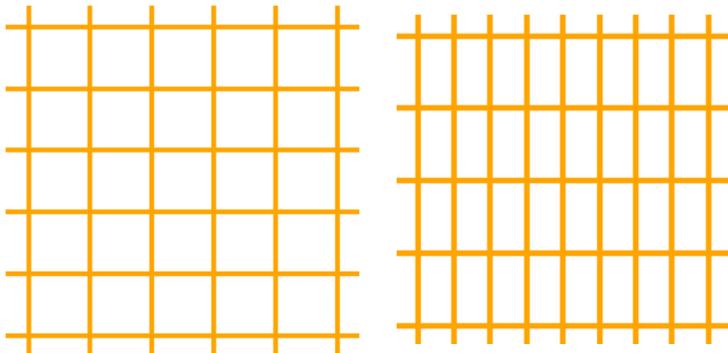


Αριστερά φαίνονται με τη σειρά μια στοιχειώδης μονάδα του πλέγματος (με άσπρο χρώμα), η κορυφή ενός γράφου (με μπλε ή κόκκινο χρώμα) και η ακμή ενός γράφου (με πράσινο ή κίτρινο χρώμα) που ενώνει δύο κορυφές.

Εικόνα 2

- Για την περιοχή 2 (Network's grid – Το πλέγμα του δικτύου):

Εδώ δίνεται απλά η δυνατότητα κατασκευής ενός νέου πλέγματος με ορισμένο αριθμό γραμμών και στηλών. Για ίσες γραμμές και στήλες οι στοιχειώδεις μονάδες του πλέγματος είναι τετράγωνες, ενώ σε αντίθετη περίπτωση είναι ορθογώνιες, όπως φαίνεται αντίστοιχα στην εικόνα 3.

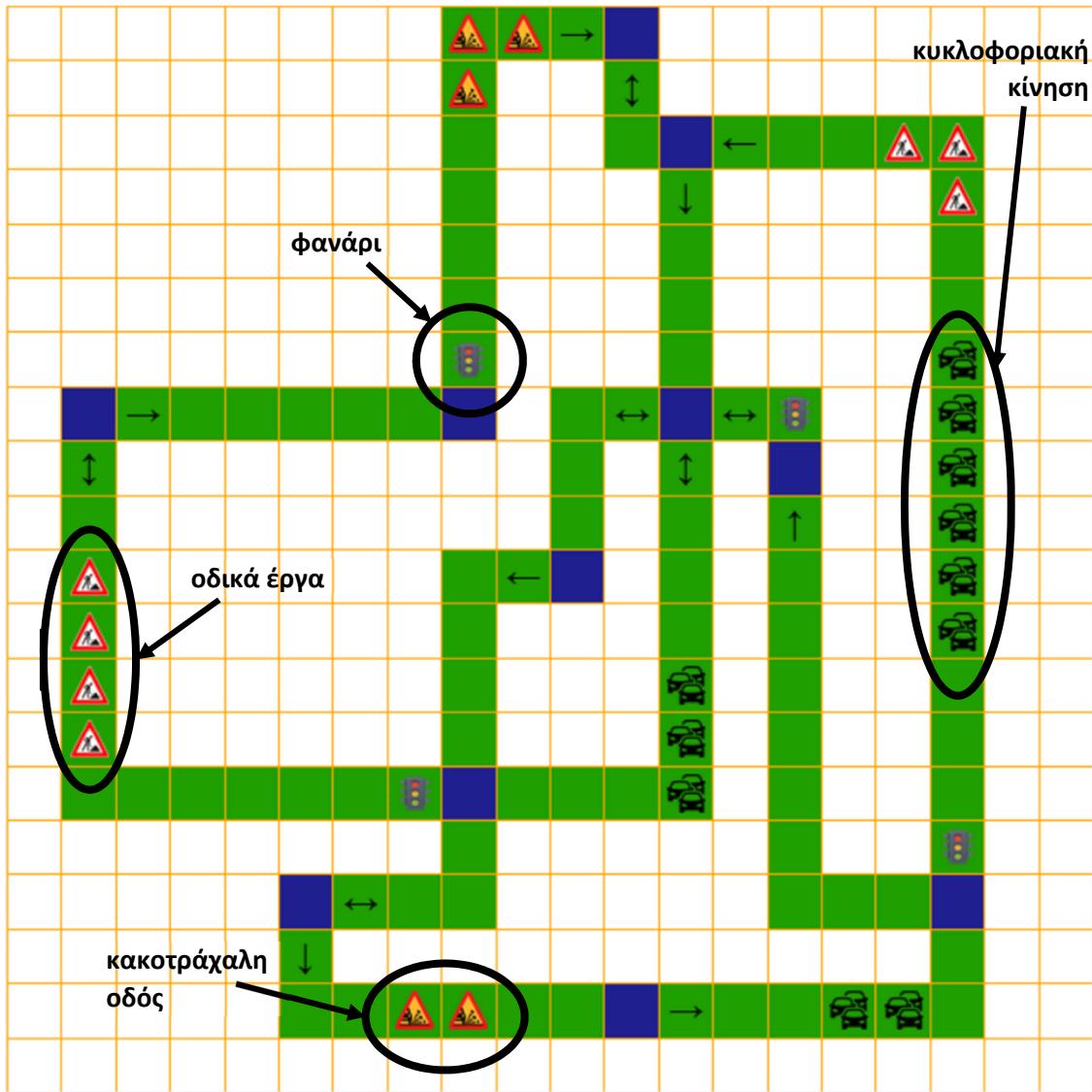


Στο αριστερό πλέγμα υπάρχει ίσος αριθμός γραμμών και στηλών, ενώ στο δεξί πλέγμα οι στήλες είναι περισσότερες από τις γραμμές.

Εικόνα 3

- Για την περιοχή 3 (Create Graph – Δημιουργία γράφου):

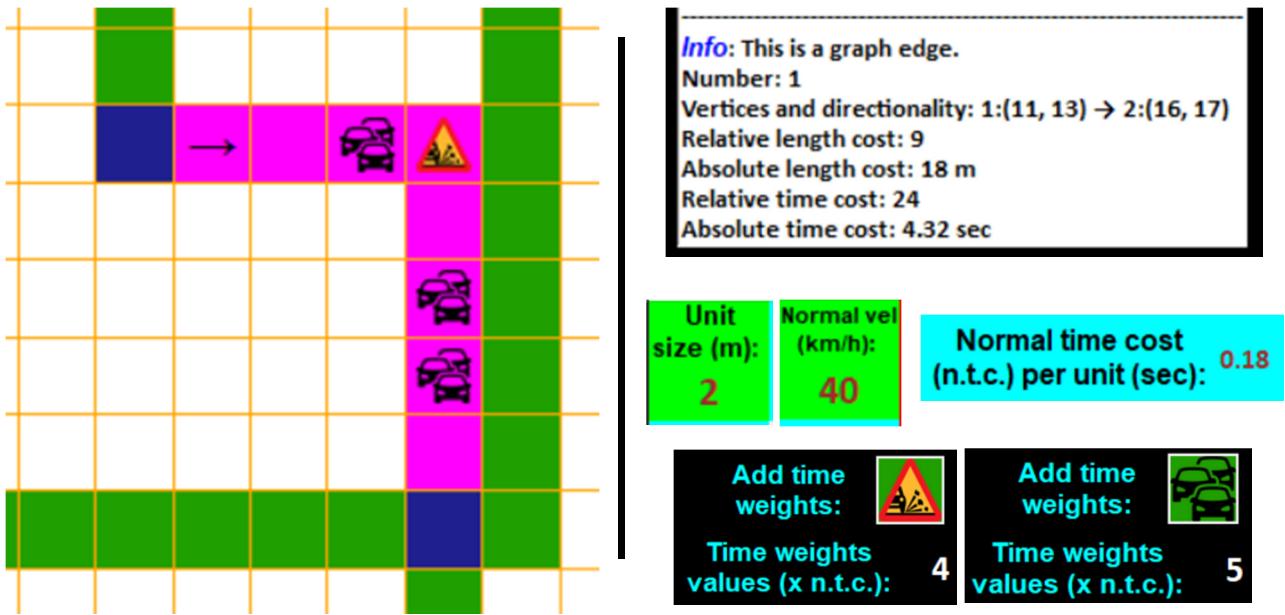
Το συγκεκριμένο μενού περιλαμβάνει όλες τις απαραίτητες λειτουργίες για τη σχεδίαση ενός ολοκληρωμένου οδικού δικτύου – γράφου. Χωρίζεται σε δύο υπο-μενού: το “auto”, που επιτρέπει την αυτόματη παραγωγή γράφων και το “manual”, που παρέχει ελευθερία εντολών στον χρήστη. Στο υπο-μενού “auto”, ο χρήστης εισάγει μόνο τον αριθμό των τυχαία παραγόμενων (ως προς τη θέση τους) κορυφών του γράφου, οι οποίες συνδέονται μεταξύ τους μέσω τυχαία παραγόμενων ακμών. Η κατευθυντικότητα των ακμών αυτών είναι επίσης τυχαία, ενώ δεν παράγονται επιπλέον χρονικά βάρη στις ακμές πέραν αυτών που υφίστανται εν γένει λόγω του φυσικού μήκους τους. Στο δεύτερο υπο-μενού “manual” παρέχονται αρκετά εργαλεία στον χρήστη για να πραγματοποιήσει ο ίδιος τον γράφο που φαντάζεται από την αρχή. Αρχικά, μπορεί να τοποθετήσει όσες κορυφές (που μπορεί να είναι κόμβοι, διασταυρώσεις ή σημεία σταθμοί του οδικού δικτύου) επιθυμεί οπουδήποτε στο πλέγμα και να τις συνδέσει μέσω ακμών (οι οδοί της πόλης) για την κατασκευή του οδικού δικτύου. Ακόμα, μπορεί να προσθέσει επιπλέον χρονικά βάρη στον γράφο, για τα οποία υπάρχουν τέσσερις επιλογές: “κακοτράχαλη οδός”, “φανάρια”, “οδικά έργα”, “κυκλοφοριακή κίνηση”. Τις τιμές των χρονικών βαρών τις ρυθμίζει επίσης ο χρήστης. Επιπλέον, υπάρχει και η δυνατότητα αλλαγής της κατευθυντικότητας κάθε ακμής, με τις δυο επιλογές να είναι: ακμή – δρόμος μονής κατεύθυνσης (κατευθυνόμενη ακμή) ή ακμή – δρόμος διπλής κατεύθυνσης (μη κατευθυνόμενη ακμή). Υπάρχουν προφανώς ορισμένοι περιορισμοί στον σχεδιασμό, που θα αναφερθούν αναλυτικότερα αργότερα. Ένα πολύ σημαντικό χαρακτηριστικό είναι η δυνατότητα αποθήκευσης του γράφου (που γίνεται σε αρχεία txt), καθώς και η δυνατότητα φόρτωσης στο πλέγμα (του γραφικού περιβάλλοντος) των σωσμένων γράφων. Στην εικόνα 4 παρουσιάζεται ένα ενδεικτικό συνεκτικό οδικό δίκτυο, με 11 κορυφές και 14 κατευθυνόμενες και μη ακμές, που περιλαμβάνει όλα τα διαφορετικά είδη χρονικών βαρών που προσφέρει η εφαρμογή.



Εικόνα 4

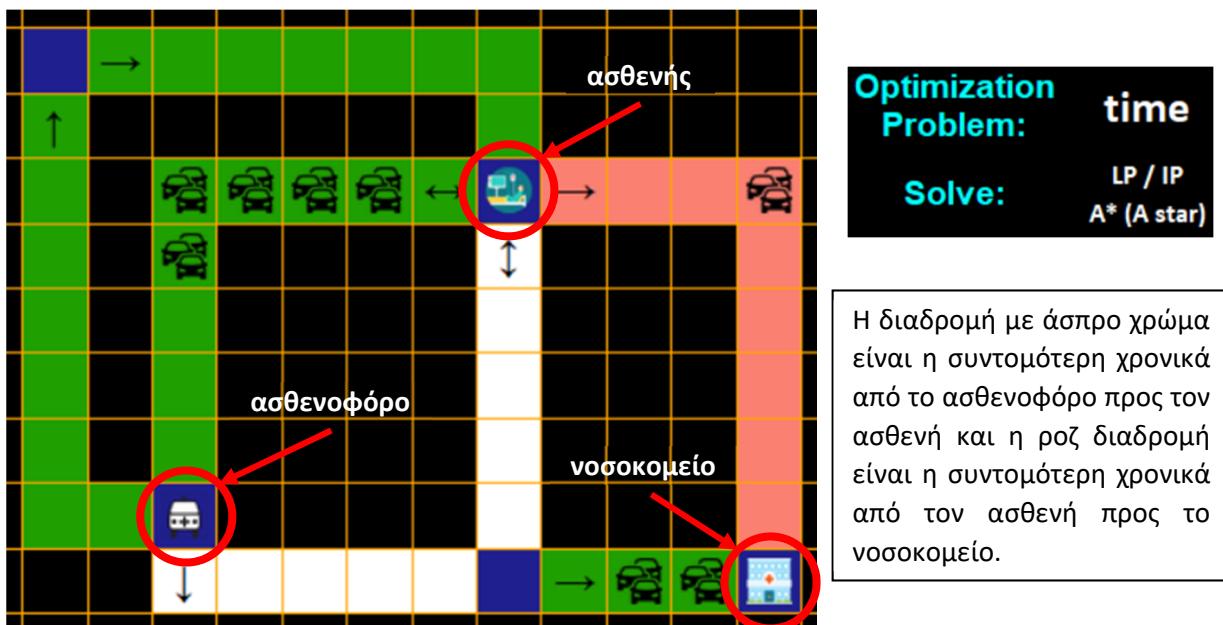
- Για την περιοχή 4 (Graph Information – Παροχή πληροφοριών σχετικών με τον γράφο):

Αυτή η περιοχή έχει σκοπό την παροχή χρήσιμων πληροφοριών για την κατάσταση του εκάστοτε οδικού δικτύου – γράφου και προσφέρει υψηλή διαδραστικότητα με τον χρήστη. Δίνει τη δυνατότητα τονισμού και καταγραφής των σημαντικότερων χαρακτηριστικών των επιλεγμένων αριθμημένων κορυφών και ακμών του γράφου, για τον εύκολο εντοπισμό τους και ταυτοποίησή τους. Τόσο οι πληροφορίες του γράφου όσο και διάφορα καθοδηγητικά και προειδοποιητικά μηνύματα καταγράφονται σε ένα πλαίσιο κειμένου, που απευθύνεται προς τον χρήστη και βοηθά στην αποτελεσματικότερη πλοιόγηση του ίδιου στην εφαρμογή. Στην εικόνα 5 παρατίθεται ένα χαρακτηριστικό παράδειγμα, όπου ο χρήστης έχει επιλέξει μια ακμή του γράφου και το πρόγραμμα τού δίνει χρήσιμες πληροφορίες για την κατεύθυντικότητά της και το συνολικό χωρικό και χρονικό βάρος της. Οι ακριβείς τρόποι υπολογισμού των βαρών θα εξηγηθούν προς το τέλος της ενότητας αυτής και έχουν σχέση με όλες τις παραμέτρους που παρατίθενται στο κάτω δεξιά μέρος της εικόνας 5.



Εικόνα 5

- Για την περιοχή 5 (Solve Graph – Επίλυση του γράφου):

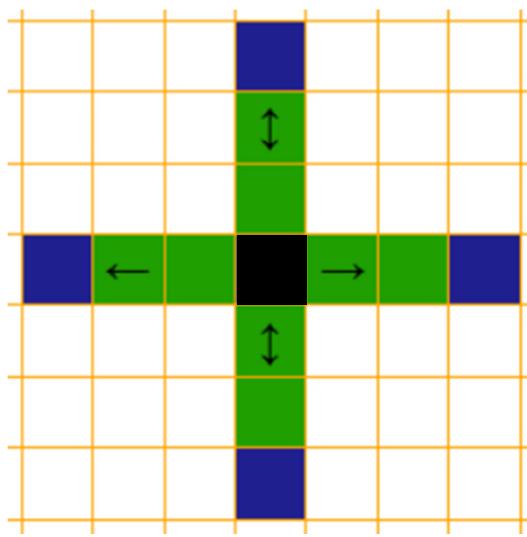


Εικόνα 6

Εδώ πραγματοποιείται η επίλυση του προβλήματος εύρεσης της συντομότερης διαδρομής, η οποία αφορά στον γράφο που σχεδιάζεται από τον χρήστη στην περιοχή 1. Ο γράφος απαιτείται να περιλαμβάνει και τις τρεις κορυφές στόχους (target vertices) που δίνονται προς τοποθέτηση, και είναι οι “ασθενοφόρο”, “ασθενής” και “νοσοκομείο”. Το ασθενοφόρο πρέπει να πάει πρώτα στον ασθενή και μετά να τον οδηγήσει στο νοσοκομείο με τον συντομότερο

δυνατό τρόπο. Το κριτήριο εύρεσης της οικονομικότερης διαδρομής μπορεί να προσαρμοστεί να είναι η διανυόμενη απόσταση (length) ή ο χρόνος διάνυσης αυτής (time). Το τελευταίο κριτήριο μάς ενδιαφέρει πρωτίστως στην εφαρμογή με την οποία ασχολούμαστε. Σε αυτήν την περιοχή ο χρήστης έχει επίσης τη δυνατότητα να διαλέξει αν θέλει η επίλυση του προβλήματος να γίνει με γραμμικό προγραμματισμό ή με τον αλγόριθμο A*. Για την περίπτωση του γραμμικού προγραμματισμού, περισσότερες και διεξοδικότερες λεπτομέρειες για τη μοντελοποίηση του προβλήματος ελαχιστοποίησης και την επίλυσή του, τόσο στη γενική του μορφή όσο και στο ειδικό σενάριο που καταστρώσαμε, βρίσκονται στις ενότητες “Μοντελοποίηση του shortest path με γραμμικό / ακέραιο προγραμματισμό” και “Ενδεικτικά παραδείγματα επίλυσης του προβλήματος στη γραφική διεπαφή”. Ακόμα, στην πέμπτη περιοχή παρέχονται λειτουργίες ελέγχου της συνδεσιμότητας του γράφου και ανάλυσης ευαισθησίας στο μοντέλο γραμμικού προγραμματισμού. Ας σημειωθεί, τέλος, ότι τα κουμπιά “path 1” και “path 2” της περιοχής 1 γίνονται ενεργά όταν έχει επιλυθεί ο γράφος με τη βοήθεια της περιοχής 5 και φανερώνουν τις λύσεις που προέκυψαν (όπως επισημαίνονται χαρακτηριστικά στο παράδειγμα της εικόνας 6).

Η ανωτέρω γραφική διεπαφή προσδίδει στους δημιουργούμενους γράφους δύο βασικά χαρακτηριστικά στοιχεία που είναι αναγκαίο να καταγραφούν και συνδέονται άμεσα με την επιλογή της ρυμοτομικής πολεοδομικής σχεδίασης. Αρχικά, παρατηρούμε ότι για έναν γράφο, έστω με N στο πλήθος κορυφές, ο μέγιστος δυνατός αριθμός ακμών είναι $\frac{N(N-1)}{2}$, διότι, προσθέτοντας μία μία τις κορυφές, η δεύτερη μπορεί να συνδεθεί με την πρώτη, η τρίτη με τις δύο πρώτες, ..., η N με τις $N - 1$ πρώτες, παράγοντας έτσι $1 + 2 + \dots + (N - 1) = \frac{N(N-1)}{2}$ ακμές το πολύ. Όπως έχουμε δομήσει όμως τη γραφική διεπαφή, από κάθε κορυφή μπορούν να οδεύουν ακμές μονάχα οριζοντίως ή καθέτως (εξαιτίας του ρυμοτομικού σχεδίου της πόλης), γεγονός που περιορίζει τον αριθμό τους σε 4 το μέγιστο όπως επιβεβαιώνεται στην εικόνα 7, επομένως οι πιθανοί γράφοι είναι λιγότεροι.



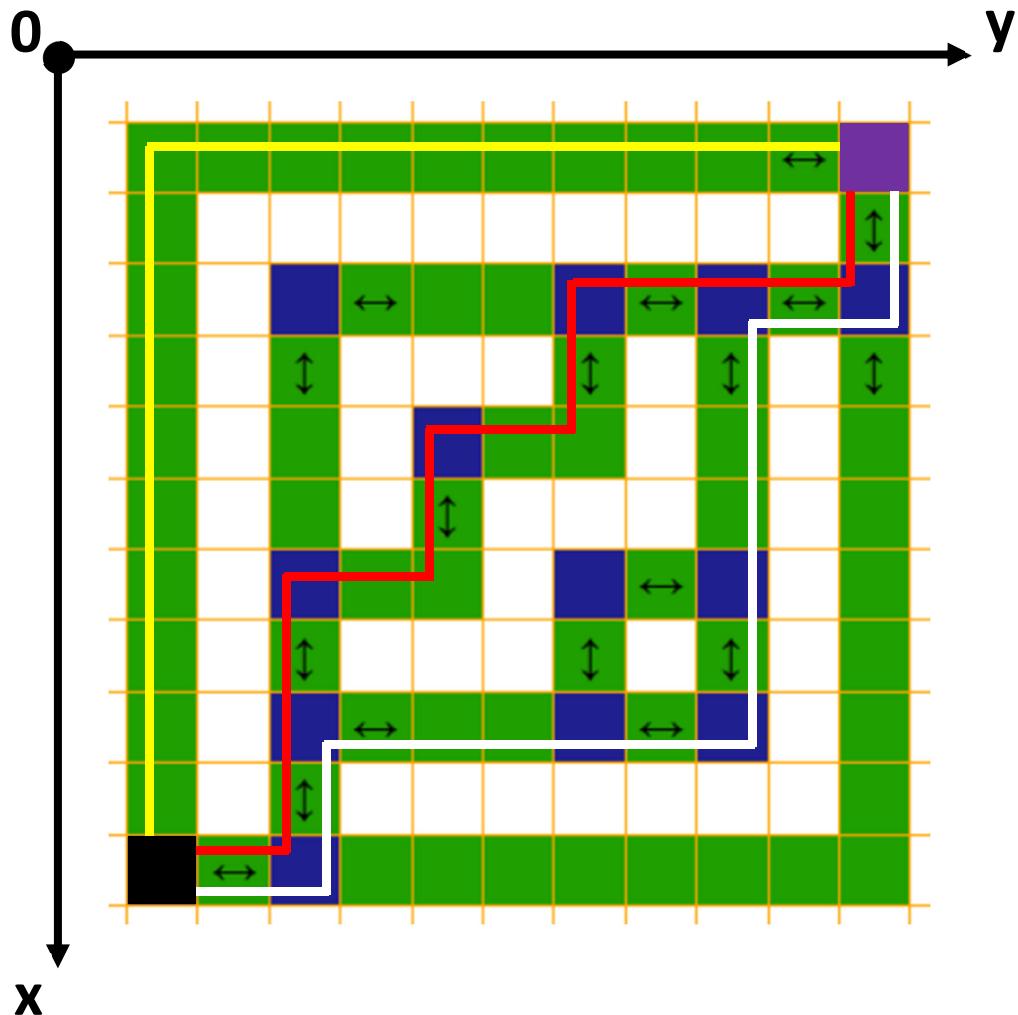
Εικόνα 7

Το δεύτερο στοιχείο έχει σχέση με τον τρόπο μέτρησης των μηκών των ακμών/δρόμων, που αποτελούν τα εν γένει χωρικά βάρη του γράφου. Λόγω της ρυμοτομικής σχεδίασης της πόλης, μία απολύτως λογική και διαισθητική επιλογή για τον υπολογισμό των μήκους των ακμών,

είναι η χρήση της απόστασης Manhattan (μετρική που αξιοποιείται στη γεωμετρία taxicab). Το οδικό μας δίκτυο τοποθετείται στο διδιάστατο επίπεδο, δηλαδή στον χώρο \mathbb{R}^2 . Έστω δύο σημεία $p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} \in \mathbb{R}^2$ και $q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \in \mathbb{R}^2$. Τότε, η απόσταση Manhattan μεταξύ των σημείων αυτών ορίζεται ως το άθροισμα των απολύτων τιμών των διαφορών των αντίστοιχων συντεταγμένων, δηλαδή:

$$d_{Manh}(p, q) = \sum_{i=1}^2 |p_i - q_i|$$

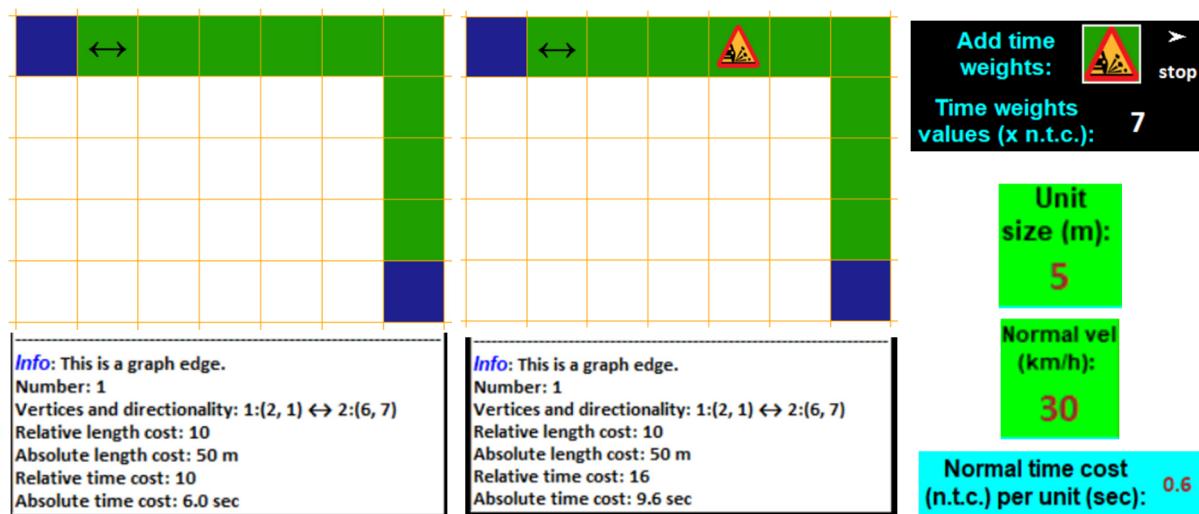
Είναι ενδιαφέρον το γεγονός ότι, λόγω της φύσης της απόστασης Manhattan, στο παρακάτω παράδειγμα (εικόνα 8) τα μήκη των σημειωμένων διαδρομών από την κάτω αριστερά μαύρη κορυφή μέχρι την πάνω δεξιά μωβ είναι ακριβώς τα ίδια (αποδεικνύεται εύκολα φέρνοντας τις προβολές των διαφόρων ευθυγράμμων τμημάτων των ακμών στους κάθετους άξονες). Τα πρόσθετα βάρη που μπορούμε να εισάγουμε («φανάρια», «κυκλοφοριακή κίνηση», «οδικά έργα», «κακοτράχαλη οδός») είναι χρονικές καθυστερήσεις και δεν μεταβάλλουν το μήκος των ακμών/δρόμων.



Εικόνα 8

Είναι χρήσιμο να καταγραφούν ακόμα οι μαθηματικές σχέσεις που συνδέουν τα χωρικά d_{rel} και χρονικά t_{rel} βάρη των ακμών (αποκαλούνται σχετικά/relative κόστη) με τις αποστάσεις d_{abs} και τις χρονικές καθυστερήσεις t_{abs} στον πραγματικό κόσμο (αποκαλούνται απόλυτα/absolute κόστη). Αρχικά, θεωρώ ότι κάθε στοιχειώδης μονάδα του πλέγματος έχει σχετικά βάρη $d'_{rel} = t'_{rel} = 1$ και απόλυτα βάρη $d'_{abs} = unit_size [m]$, $t'_{abs} = n_t_c [sec] = 3.6 \frac{unit_size [m]}{normal_vel [\frac{km}{h}]}$ (τα *unit size, normal vel* και *n. t. c.* ρυθμίζονται μέσω της γραφικής διεπαφής

και αποτελούν μεγέθη του πραγματικού κόσμου). Τότε, ορίζω για μία άδεια ακμή (χωρίς πρόσθετα χρονικά κόστη) να διαθέτει σχετικά βάρη $d_{rel} = t_{rel} = d_{Manh} \cdot d'_{rel} = d_{Manh}$ και απόλυτα βάρη $d_{abs} = unit_size \cdot d_{rel} = unit_size \cdot d_{Manh}$, $t_{abs} = n_t_c \cdot t_{rel} = n_t_c \cdot d_{Manh}$. Αν μία στοιχειώδης μονάδα επιβαρύνεται με χρονική καθυστέρηση σχετικού βάρους t_{add} , τότε διατηρεί τα ίδια χωρικά κόστη και αποκτά νέα $t'_{rel} = t_{add}$ και $t'_{abs} = n_t_c \cdot t_{add}$. Άρα λοιπόν, μια ακμή με επιπλέον χρονικές καθυστερήσεις διατηρεί τα ίδια d_{rel} και d_{abs} , αλλά διαφέρει στα $t_{rel} = \sum_{k=1}^{d_{Manh}} t_k$, $t_k = t_{k,add}$, k μονάδα επιπλέον χρονικό κόστος και $t_{abs} = n_t_c \cdot t_{rel}$. Όλα τα παραπάνω επιβεβαιώνονται και από το παράδειγμα της εικόνας 9.



Εικόνα 9

Γενικά, για τους γράφους που δημιουργούνται ισχύουν οι επόμενοι κανόνες. Αρχικά, δεν πρέπει να περιέχουν αυτο-βρόχους (self-loops), δηλαδή ακμές που συνδέουν μια κορυφή με τον εαυτό της. Επίσης, δεν επιτρέπονται οι πολλαπλές ακμές, δηλαδή οι ακμές με κοινά άκρα/κορυφές (συνεπώς αποκλείουμε την ύπαρξη πολυγράφων-multigraphs). Ακόμα, οι γράφοι διαθέτουν μόνο θετικά βάρη (positive weights), αφού ασχολούμαστε με φυσικές αποστάσεις και χρόνους. Επιπλέον, θεωρούμε ότι οι γράφοι είναι κατευθυνόμενοι (directed), αφού οι δρόμοι του οδικού δικτύου της πόλης δύνανται να είναι μονίμις (κατευθυνόμενες ακμές) ή διπλής (μη κατευθυνόμενες ακμές) κυκλοφορίας. Τέλος, είναι επιθυμητό να υφίσταται μία τουλάχιστον διαδρομή που να συνδέει δύο οποιεσδήποτε κορυφές του γράφου, με άλλα λόγια οι γράφοι να είναι ισχυρώς συνδεδεμένοι (strongly/fully connected) ή απλώς συνεκτικοί. Υπάρχει βέβαια περιθώριο να χαλαρώσουμε τον τελευταίο αυτόν περιορισμό, όπως θα φανεί αργότερα.

Μοντελοποίηση του shortest path με γραμμικό / ακέραιο προγραμματισμό:

Στόχος μας είναι η έγκαιρη μετάβαση του ασθενή από το σπίτι του στο νοσοκομείο με τη χρήση του ασθενοφόρου ως μεταφορικό μέσο. Επομένως, πρέπει να υπολογιστεί η οικονομικότερη διαδρομή, από άποψη χρόνου, με αφετηρία το σημείο που βρίσκεται αρχικά το ασθενοφόρο, προορισμό το νοσοκομείο και υπό τον περιορισμό ότι το ασθενοφόρο απαιτείται να περάσει από τον χώρο διαμονής του ασθενούς για να τον παραλάβει. Διαπιστώνουμε ότι η συγκεκριμένη διαδρομή αποτελείται από δύο επιμέρους διαδρομές: τη χρονικά συντομότερη από το ασθενοφόρο προς το σπίτι του ασθενούς και τη χρονικά συντομότερη από το σπίτι προς το νοσοκομείο. Άρα η επίλυση του σεναρίου που ορίσαμε απαιτεί την επίλυση δύο προβλημάτων ελαχιστοποίησης. Αξίζει να σημειωθεί ότι για οδικά δίκτυα χωρίς καθόλου χρονικές καθυστερήσεις η συντομότερη χρονικά διαδρομή είναι η ίδια ακριβώς με τη μικρότερη σε μήκος, κάτι που δεν ισχύει γενικά όταν προστίθενται χρονικά βάρη στον γράφο.

Η πιο εύλογη μοντελοποίηση του προβλήματος εύρεσης της συντομότερης διαδρομής είναι με ακέραιο προγραμματισμό και μάλιστα με δυαδικό ακέραιο προγραμματισμό [3 και 4]. Έστω κατευθυνόμενος γράφος (ή αλλιώς δίκτυο) $G(V, E)$, με θετικά βάρη $w_{ij} > 0 \quad \forall (i, j) \in E$. Με V συμβολίζεται το πεπερασμένο σύνολο των κορυφών – vertices (ή κόμβων) του γράφου και με E συμβολίζεται το πεπερασμένο σύνολο των ακμών – edges (ή συνδέσεων) του γράφου. Έστω ακόμα $p, q \in V$ δύο διακριτές κορυφές του γράφου και επιθυμούμε να βρούμε τη συντομότερη διαδρομή από την κορυφή αφετηρίας p ως την κορυφή προορισμού q . Πρόκειται για μια ειδική περίπτωση του προβλήματος ροής ελαχίστου κόστους σε δίκτυο, γι' αυτό μπορούμε να το μοντελοποιήσουμε ως τέτοιο. Θεωρούμε ότι η κορυφή αφετηρίας p λειτουργεί ως πηγή (source) που παράγει και στέλνει στο δίκτυο μια ροή ποσότητας φ μονάδων (για λόγους απλότητας μπορούμε να θεωρήσουμε $\varphi = 1$, αν και δεν είναι απαραίτητο αφού η ροή απλοποιείται στους τύπους όπως θα φανεί στην ανάλυση). Η ροή πρέπει να ταξιδέψει στο δίκτυο με τέτοιον τρόπο, έτσι ώστε να φτάσει αναλλοίωτη μέχρι και την κορυφή προορισμού q , που λειτουργεί ως σιφόνι (sink) και εξαφανίζει ολόκληρη τη ροή φ από το δίκτυο. Παράλληλα, απαιτείται να διατρέξει τις ακμές εκείνες με το ελάχιστο συνολικό βάρος. Είναι δυνατόν να δημιουργήσουμε το εξής πρόβλημα ακέραιου δυαδικού προγραμματισμού:

$$\min_{x_{ij}} \sum_{(i,j) \in E} w_{ij} x_{ij} \quad \text{αντικειμενική συνάρτηση}$$

$$\sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = \begin{cases} 1 & \text{αν } i = p \\ 0 & \text{αν } i \neq p, q \\ -1 & \text{αν } i = q \end{cases} \quad \forall i \in V \quad \text{ισοτικοί περιορισμοί}$$

$$\sum_{j:(i,j) \in E} x_{ij} \leq 1 \quad \forall i \in V \quad \text{ανισοτικοί περιορισμοί}$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in E \quad \text{μεταβλητές απόφασης}$$

Ας ερμηνεύσουμε τα δομικά στοιχεία της παραπάνω μοντελοποίησης. Αρχικά, οι μεταβλητές απόφασης αναφέρονται στις ακμές του γράφου. Η μεταβλητή x_{ij} αντιστοιχεί στην ακμή $(i, j) \in E$ του γράφου και δύναται να πάρει δυο τιμές, 0 ή 1. Η τιμή 0 δηλώνει την απουσία της ακμής στη συντομότερη διαδρομή, ενώ η τιμή 1 δηλώνει την παρουσία της. Είναι εξαιρετικά σημαντικό να τονιστεί ότι όταν κάνουμε λόγο εδώ (και στη συνέχεια) για ακμή (i, j) , εννοούμε μια κατευθυνόμενη ακμή με κατεύθυνση από την κορυφή $i \in V$ προς την κορυφή $j \in V$. Οι μη κατευθυνόμενες ακμές απαιτούν ειδική μεταχείριση, πιο συγκεκριμένα θεωρούμε πως κάθε μία από αυτές μπορεί να διασπαστεί σε δύο κατευθυνόμενες ακμές. Αυτό σημαίνει ότι μια μη κατευθυνόμενη ακμή που συνδέει τις κορυφές $i, j \in V$ αποτελείται από τις κατευθυνόμενες ακμές $(i, j) \in E$ και $(j, i) \in E$ ($w_{ij} = w_{ji}$). Κατά συνέπεια, το πλήθος των μεταβλητών απόφασης του προβλήματος δίνεται από τον τύπο $\#x = \frac{\text{κατευθυνόμενες}}{\text{ακμές}} + 2 \cdot \frac{\text{μη κατευθυνόμενες}}{\text{ακμές}}$.

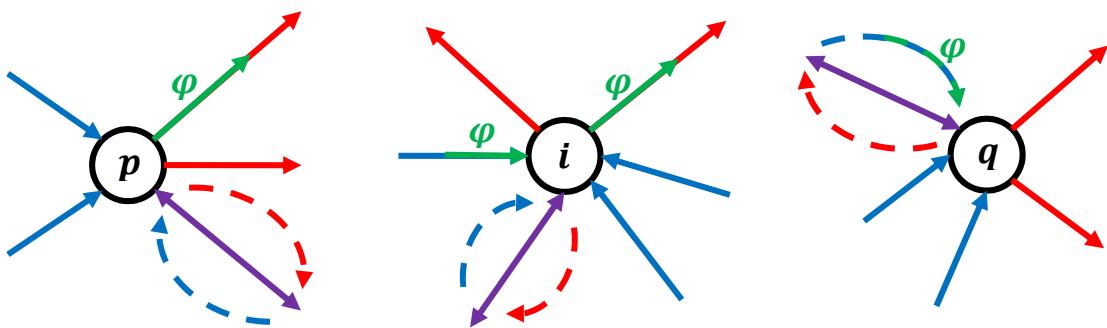
Στη συνέχεια, έχουμε την αντικειμενική συνάρτηση, που είναι το άθροισμα των όρων $w_{ij}x_{ij}$ για όλες τις ακμές $(i, j) \in E$ του γράφου ($\sum_{(i,j) \in E} w_{ij}x_{ij}$). Αν $x_{ij} = 0$, τότε $w_{ij}x_{ij} = 0$, αφού η ακμή (i, j) δεν είναι μέσα στη συντομότερη διαδρομή, με αποτέλεσμα να μη συνεισφέρει στο συνολικό κόστος της διαδρομής. Αντιθέτως, αν $x_{ij} = 1$, τότε $w_{ij}x_{ij} = w_{ij}$, αφού η ακμή (i, j) είναι εντός της συντομότερης διαδρομής και το βάρος της χρειάζεται να ληφθεί υπόψη. Ελαχιστοποιώντας την αντικειμενική συνάρτηση ως προς όλες τις μεταβλητές απόφασης x_{ij} , υπολογίζεται η συντομότερη διαδρομή από την p στην q , η οποία αποτελείται από τις ακμές με αντίστοιχες μεταβλητές απόφασης ίσες με τη μονάδα. Προφανώς, αν δεν υπήρχε κανένας περιορισμός και λόγω της μη αρνητικότητας των βαρών, η ελάχιστη τιμή της αντικειμενικής συνάρτησης θα ήταν 0, κάτι που δεν βγάζει νόημα αφού θα σήμαινε ότι η συντομότερη διαδρομή δεν περιέχει καθόλου ακμές. Για αυτό, οι ισοτικοί περιορισμοί που θα παρουσιαστούν αμέσως μετά είναι πολύ σημαντικοί, καθώς δίνουν πρακτικά τις σχέσεις μεταξύ των ακμών του γράφου και τους περιορισμούς στους οποίους αυτοί υπόκεινται.

Οι ισοτικοί περιορισμοί αναφέρονται στις κορυφές του γράφου, με αποτέλεσμα το πλήθος τους να είναι ίσο με αυτό των κορυφών. Δομούνται έτσι ώστε να ισχύει σε όλους τους κόμβους (κορυφές) η αρχή διατήρησης της ροής, που σημαίνει ότι η ροή φ που δόθηκε στο δίκτυο στην κορυφή αφετηρίας p δεν είναι δυνατόν να χάνεται ή να γεννιέται από το τίποτα στο εσωτερικό του δικτύου, με λίγα λόγια δεν επιτρέπεται να εμφανίζει ασυνέχειες. Οι εξισώσεις που προκύπτουν παρουσιάζουν ηλεκτρικό ανάλογο στα ηλεκτρικά κυκλώματα, με τον νόμο ρευμάτων του Kirchoff εκεί να επιβάλλει τη διατήρηση του ηλεκτρικού φορτίου. Η σύγκριση βέβαια είναι λίγο αδύναμη, διότι στα ηλεκτρικά κυκλώματα το ρεύμα μοιράζεται σε όλους τους κλάδους αλλού σε μικρότερο και αλλού σε μεγαλύτερο ποσοστό, ενώ στο δίκτυο μας η ροή φ ρέει πλήρως σε ορισμένες ακμές και είναι μηδενική σε άλλες (γεγονός που προκαλείται λόγω της δυαδικότητας των μεταβλητών απόφασης). Ακόμα, στα ηλεκτρικά κυκλώματα, οι αντιστάσεις των κλάδων επηρεάζουν τις εξισώσεις συνέχειας του φορτίου, όμως στον γράφο μας τα αντίστοιχα βάρη των ακμών δεν εισάγονται στους ισοτικούς περιορισμούς διατήρησης της ροής (έχουν ήδη ληφθεί υπόψη στην αντικειμενική συνάρτηση). Οι ισοτικοί περιορισμοί εξαρτώνται από τη θέση της κάθε κορυφής στον γράφο. Έτσι, εξάγονται ένας ισοτικός περιορισμός για την κορυφή αφετηρίας p , ένας ισοτικός για την κορυφή προορισμού q και $N - 2$ για τις υπόλοιπες κορυφές, όπου N το πλήθος τους. Η εικόνα 10 περιλαμβάνει και τις τρεις διαφορετικές περιπτώσεις, όπου με μπλε και με κόκκινο χρώμα χρωματίζονται οι εισερχόμενες και οι

εξερχόμενες ακμές αντίστοιχα για κάθε κορυφή του γράφου, ενώ η κατεύθυνση της ροής επισημαίνεται με τα πράσινα βέλη. Για την περίπτωση της κορυφής αφετηρίας p υφίσταται μονάχα εξερχόμενη ροή (οι εισερχόμενες ροές είναι μηδενικές, δηλαδή $\sum_{j:(j,p) \in E} x_{jp} = 0$), η οποία οδηγείται μονάχα προς μία ακμή την οποία δεν γνωρίζουμε εκ των προτέρων, οπότε μπορούμε να γράψουμε: $\varphi \sum_{j:(p,j) \in E} x_{pj} = \varphi \Rightarrow \sum_{j:(p,j) \in E} x_{pj} = 1$, όπου $x_{pj} = 1$ για ένα μόνο j . Για τις εσωτερικές κορυφές υπάρχει εισερχόμενη και εξερχόμενη ροή ίσου μεγέθους από δύο άγνωστες ακμές, άρα είναι $\varphi \sum_{j:(i,j) \in E} x_{ij} = \varphi \sum_{j:(j,i) \in E} x_{ji} \Rightarrow \sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = 0$, όπου $x_{lj} = 1$ για ένα μόνο l και $x_{jr} = 1$ για ένα μόνο r , με τις x_{ij} και x_{ji} να αναφέρονται στις εξερχόμενες και εισερχόμενες ακμές αντίστοιχα. Τέλος, για την περίπτωση της κορυφής προορισμού q υφίσταται μονάχα εισερχόμενη ροή (οι εξερχόμενες ροές είναι μηδενικές, δηλαδή $\sum_{j:(q,j) \in E} x_{jq} = 0$), η οποία έρχεται μονάχα προς μία ακμή την οποία δεν γνωρίζουμε εκ των προτέρων, οπότε μπορούμε να γράψουμε: $\varphi \sum_{j:(j,q) \in E} x_{jq} = \varphi \Rightarrow \sum_{j:(j,q) \in E} x_{jq} = 1$, με $x_{jq} = 1$ για ένα μόνο j .

Συγκεντρώνοντας όλες τις παραπάνω σχέσεις σε μία λιγότερο αυστηρή συνθήκη οδηγούμαστε στην τρίκλαδη σχέση $\sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = \begin{cases} 1 & \text{αν } i = p \\ 0 & \text{αν } i \neq p, q \\ -1 & \text{αν } i = q \end{cases} \quad \forall i \in V$.

Είναι λιγότερο αυστηρή, καθώς λέει ότι οι εισερχόμενες ροές σε σχέση με τις εξερχόμενες ροές είναι μία λιγότερες στην κορυφή αφετηρίας, μία περισσότερες στην κορυφή προορισμού και ακριβώς ίσες σε οποιαδήποτε εσωτερική κορυφή. Το ότι οι εισερχόμενες μαζί με τις εξερχόμενες ακμές που φέρουν ροές είναι το πολύ δύο για κάθε κορυφή του γράφου, όπως φαίνεται και στην εικόνα 10, υπαγορεύεται από τους ανισοτικούς περιορισμούς που παρουσιάζονται αμέσως μετά.



Εικόνα 10

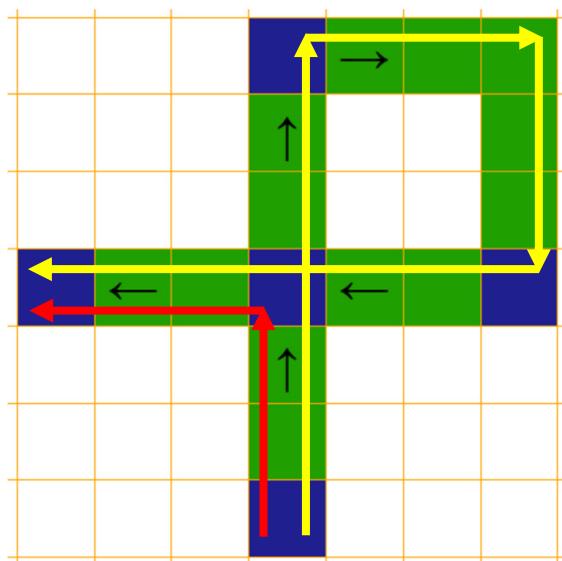
Οι ανισοτικοί περιορισμοί αναφέρονται επίσης στις κορυφές του γράφου, άρα το πλήθος τους είναι ίσο με αυτό των κορυφών. Η λειτουργία που επιτελούν είναι να απαγορεύουν τη διέλευση από μία κορυφή του γράφου παραπάνω από μία φορές. Αυτό μεταφράζεται στο ότι οι εξερχόμενες ακμές από κάθε κορυφή του γράφου που φέρουν ροή επιτρέπεται να είναι το πολύ μία. Δηλαδή, για κάθε κορυφή, ή υπάρχει εξερχόμενη ακμή που φέρει τη ροή φ ή δεν υπάρχει, με αποτέλεσμα να προκύπτουν οι εξής ανισοτικές συνθήκες: $\varphi \sum_{j:(i,j) \in E} x_{ij} \leq \varphi \Rightarrow \sum_{j:(i,j) \in E} x_{ij} \leq 1 \quad \forall i \in V$.

Στο προηγούμενο μοντέλο ακέραιου προγραμματισμού μπορούμε να πραγματοποιήσουμε ορισμένες απλοποιήσεις, αφαιρώντας περιορισμούς που τελικά δεν είναι απαραίτητοι για την εύρεση του ορθού αποτελέσματος. Πρώτον, γίνεται να αποδειχτεί ότι ο ισοτικός περιορισμός $\sum_{j:(q,j) \in E} x_{qj} - \sum_{j:(j,q) \in E} x_{jq} = -1$ είναι περιττός, διότι εξάγεται από τους υπόλοιπους ισοτικούς περιορισμούς $\sum_{j:(p,j) \in E} x_{pj} - \sum_{j:(j,p) \in E} x_{jp} = 1$ και $\sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = 0, \forall i \neq p, q$ (η απόδειξη μπορεί να βρεθεί στο [3]). Παρ' όλα αυτά τον έχω λάβει υπόψη στον κώδικα, καθώς δείχνει ξεκάθαρα ποιος είναι ο περιορισμός που αναφέρεται στην κορυφή προορισμού q . Δεύτερον, είναι δυνατόν να αγνοηθούν όλοι οι ανισοτικοί περιορισμοί, διότι η ίδια η ελαχιστοποίηση τους εξασφαλίζει (οι ανισοτικοί περιορισμοί αναγράφονται στο [4], αλλά όχι στο [3] π.χ.). Γενικά, λόγω της ελαχιστοποίησης, αποφεύγεται η ύπαρξη κύκλων στις διαδρομές, που δημιουργούνται όταν διαπερνάται μία κορυφή περισσότερες από μία φορές. Για την εξήγηση αυτού, χαρακτηριστικό είναι το παράδειγμα της εικόνας 11. Έτσι λοιπόν, λαμβάνοντας υπόψη όλα τα προλεχθέντα, το μοντέλο ακέραιου προγραμματισμού που θα αξιοποιηθεί τελικά είναι το ακόλουθο:

$$\min_{x_{ij}} \sum_{(i,j) \in E} w_{ij} x_{ij} \quad \text{αντικειμενική συνάρτηση}$$

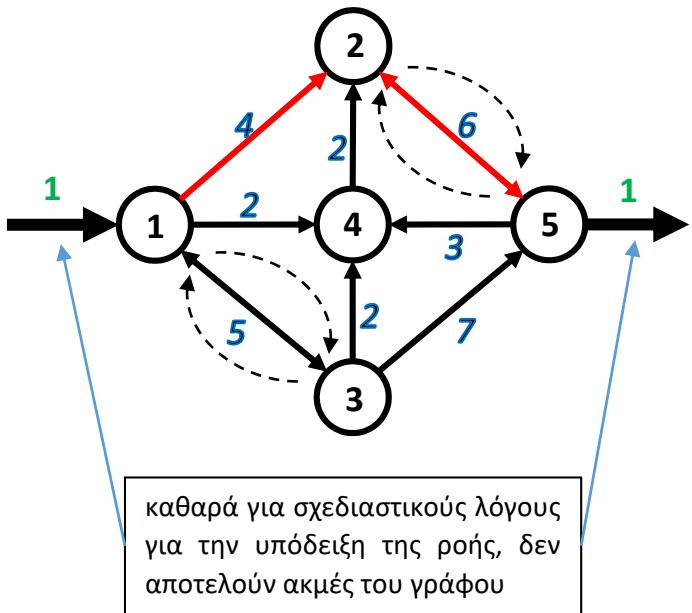
$$\sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = \begin{cases} 1 & \text{αν } i = p \\ 0 & \text{αν } i \neq p, q \\ -1 & \text{αν } i = q \end{cases} \quad \forall i \in V \quad \text{ισοτικοί περιορισμοί}$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in E \quad \text{μεταβλητές απόφασης}$$



Εικόνα 11

Η διαδρομή με το κίτρινο χρώμα είναι αδύνατη, καθώς διέρχεται από την κεντρική κορυφή δύο φορές (οι εξερχόμενες ακμές είναι 2). Απορρίπτεται λοιπόν λόγω των ανισοτικών περιορισμών. Παρατηρούμε όμως ότι απορρίπτεται και λόγω της ελαχιστοποίησης, βάσει της οποίας επιλέγεται η κόκκινη διαδρομή (η οποία διαθέτει τρεις λιγότερες ακμές από την κίτρινη και προφανώς το χωρικό και χρονικό κόστος της είναι μικρότερα). Συνεπώς, οι ανισοτικοί περιορισμοί είναι περιττοί.



Ο γράφος αποτελείται από $N = 5$ κορυφές και περιλαμβάνει $M = 10$ στο πλήθος ακμές (αφότου έχει γίνει η διάσπαση σε μονοκατευθυντικές). Η κορυφή αφετηρίας είναι η $p = 1$, η κορυφή προορισμού είναι η $q = 5$ και η ροή που διατρέχει τον γράφο είναι ίση με 1 μονάδα. Οι ακμές διαθέτουν θετικά βάρη που σημειώνονται με μπλε χρώμα. Με κόκκινο χρώμα τονίζεται η διαδρομή ελαχίστου κόστους που διαθέτει συνολικό βάρος ίσο με 10.

Εικόνα 12

Στην εικόνα 12 παρουσιάζεται ένα ενδεικτικό παράδειγμα μη κατευθυνόμενου γράφου με θετικά βάρη. Ακολούθως παρατίθεται η μοντελοποίησή του με ακέραιο προγραμματισμό, με σκοπό την εύρεση της συντομότερης διαδρομής από την κορυφή αφετηρίας προς την κορυφή προορισμού:

- Μεταβλητές απόφασης:

$$x_{1,2}, x_{1,3}, x_{1,4}, x_{2,5}, x_{3,1}, x_{3,4}, x_{3,5}, x_{4,2}, x_{5,2}, x_{5,4}$$

- Αντικειμενική συνάρτηση:

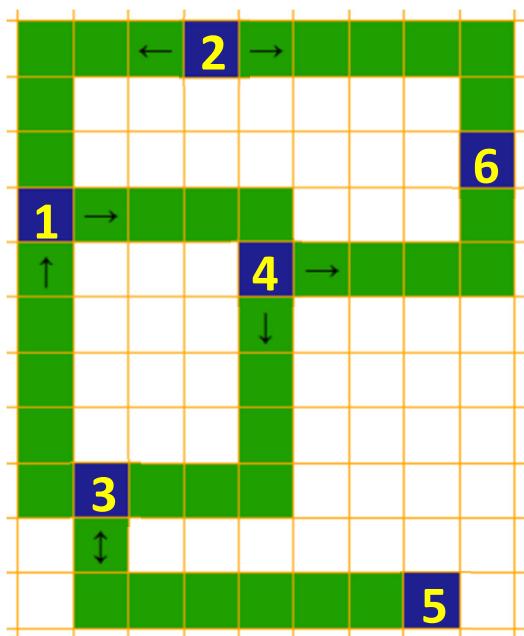
$$\min(4x_{1,2} + 5x_{1,3} + 2x_{1,4} + 6x_{2,5} + 5x_{3,1} + 2x_{3,4} + 7x_{3,5} + 2x_{4,2} + 6x_{5,2} + 3x_{5,4})$$

- Ισοτικοί περιορισμοί:

$$\begin{aligned}
 &\text{για την κορυφή 1: } x_{1,2} + x_{1,3} + x_{1,4} - x_{3,1} = 1 \\
 &\text{για την κορυφή 2: } x_{2,5} - x_{1,2} - x_{4,2} - x_{5,2} = 0 \\
 &\text{για την κορυφή 3: } x_{3,1} + x_{3,4} + x_{3,5} - x_{1,3} = 0 \\
 &\text{για την κορυφή 4: } x_{4,2} - x_{1,4} - x_{3,4} - x_{5,4} = 0 \\
 &\text{για την κορυφή 5: } x_{5,2} + x_{5,4} - x_{2,5} - x_{3,5} = -1
 \end{aligned}$$

Η επίλυση για τον συγκεκριμένο γράφο δίνει τη λύση $x_{1,2} = x_{2,5} = 1$, με όλες τις υπόλοιπες μεταβλητές απόφασης να είναι μηδενικές. Άρα η συντομότερη διαδρομή είναι η $1 \rightarrow 2 \rightarrow 5$. Ας σημειωθεί ότι και η διαδρομή $1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ έχει κόστος 10, το ίδιο ακριβώς με την $1 \rightarrow 2 \rightarrow 5$, επομένως ο γράφος διαθέτει δύο ισάξιες συντομότερες διαδρομές.

Προφανώς, έχει νόημα να αναζητήσουμε λύση στο πρόβλημα αν υπάρχει τουλάχιστον μία διαδρομή που να συνδέει το ασθενοφόρο με το σπίτι του ασθενούς και παρομοίως, τουλάχιστον μία διαδρομή που να συνδέει το σπίτι του ασθενούς με το νοσοκομείο (βέβαια, αν δεν υφίσταται μονοπάτι από τον κόμβο αφετηρίας προς τον κόμβο προορισμού, τότε η επίλυση του προβλήματος ακέραιου προγραμματισμού θα μας δώσει ως αποτέλεσμα ότι δεν υπάρχει εφικτή λύση). Αν ο γράφος μας είναι συνεκτικός, είμαστε σίγουροι ότι η παραπάνω προϋπόθεση ισχύει. Σε κάθε περίπτωση, μπορούμε να μελετήσουμε τη συνδεσιμότητα του γράφου μεταφράζοντάς τον σε κάποιο κατάλληλο μαθηματικό αντικείμενο, όπως είναι η λίστα γειτνίασης [6], με την οποία έχουμε άμεση εποπτεία των γειτονικών κορυφών κάθε κορυφής του γράφου, ή ο πίνακας γειτνίασης [5], ο οποίος μας παρέχει εύκολα την πληροφορία για το αν δύο κορυφές συνδέονται μέσω κάποιας ακμής του γράφου. Η εικόνα 13 δίνει το παράδειγμα ενός γράφου και τις αναπαραστάσεις του με λίστα γειτνίασης (adjacency list) και με πίνακα γειτνίασης (adjacency matrix):



The adjacency list of the graph is:

```
Vertex 1: [4]
Vertex 2: [1, 6]
Vertex 3: [1, 5]
Vertex 4: [3, 6]
Vertex 5: [3]
Vertex 6: []
```

The adjacency matrix of the graph is:

```
[[0 0 0 1 0 0]
 [1 0 0 0 0 1]
 [1 0 0 0 1 0]
 [0 0 1 0 0 1]
 [0 0 1 0 0 0]
 [0 0 0 0 0 0]]
```

Εικόνα 13

Ο πίνακας γειτνίασης παράγεται βάσει του ακόλουθου κανόνα (συμβολίζεται με A και προφανώς δεν έχει καμία σχέση με τον ομώνυμο πίνακα στη μοντελοποίηση με ακέραιο προγραμματισμό της επόμενης ενότητας “Ενδεικτικά παραδείγματα επίλυσης του προβλήματος στη γραφική διεπαφή”):

$$\text{adjacency matrix } A: a_{ij} = \begin{cases} 0 & (i,j) \notin E \\ 1 & (i,j) \in E \end{cases}$$

Η λίστα γειτνίασης γίνεται να κατασκευαστεί με τον παρακάτω τρόπο:

$$\text{adjacency list} = i: [j \text{ τέτοια ώστε } (i,j) \in E] \quad \forall i \in V$$

Και οι δύο μαθηματικές δομές παράγονται με τη βοήθεια των δύο συναρτήσεων που φαίνονται στην εικόνα 14 (βρίσκονται στο αρχείο “networks_algorithms.py”) και λαμβάνουν ως ορίσματα τις κορυφές, τις ακμές και την κατευθυντικότητα των ακμών του εκάστοτε γράφου.

```

13 # create_adjacency_matrix function is used to create an adjacency matrix from a graph represented by a list of vertices,
14 # a list of edges and a list of edges directionalities
15 def create_adjacency_matrix(graph_vertices, graph_edges, graph_edges_directionalities):
16     adjacent_matrix = np.zeros((len(graph_vertices), len(graph_vertices)), dtype = int)
17     for edge_number in range(len(graph_edges)): # go through all the edges of the graph
18         left_vertex = graph_vertices.index(graph_edges[edge_number][0]) # one of the vertices of the edge
19         right_vertex = graph_vertices.index(graph_edges[edge_number][1]) # the other vertex of the edge
20         edge_connection = graph_edges_directionalities[edge_number] # the directionality of the edge
21         if edge_connection == "<": # there is an edge from right_vertex to left_vertex
22             adjacent_matrix[right_vertex][left_vertex] = 1
23         if edge_connection == ">": # there is an edge from left_vertex to right_vertex
24             adjacent_matrix[left_vertex][right_vertex] = 1
25         if edge_connection == "<>": # there is an edge from left_vertex to right_vertex and from right_vertex to left_vertex
26             adjacent_matrix[left_vertex][right_vertex] = 1
27             adjacent_matrix[right_vertex][left_vertex] = 1
28     return(adjacent_matrix)
29
30 # create_adjacency_list function is used to create an adjacency list from a graph represented by a list of vertices,
31 # a list of edges and a list of edges directionalities
32 def create_adjacency_list(graph_vertices, graph_edges, graph_edges_directionalities):
33     adjacency_list = {key: [] for key in range(1, len(graph_vertices) + 1)}
34     for edge_number in range(len(graph_edges)): # go through all the edges of the graph
35         left_vertex = graph_vertices.index(graph_edges[edge_number][0]) + 1 # one of the vertices of the edge
36         right_vertex = graph_vertices.index(graph_edges[edge_number][1]) + 1 # the other vertex of the edge
37         edge_connection = graph_edges_directionalities[edge_number] # the directionality of the edge
38         if edge_connection == "<": # there is an edge from right_vertex to left_vertex
39             adjacency_list[right_vertex].append(left_vertex)
40         elif edge_connection == ">": # there is an edge from left_vertex to right_vertex
41             adjacency_list[left_vertex].append(right_vertex)
42         elif edge_connection == "<>": # there is an edge from left_vertex to right_vertex and from right_vertex to left_vertex
43             adjacency_list[right_vertex].append(left_vertex)
44             adjacency_list[left_vertex].append(right_vertex)
45     return(adjacency_list)

```

Εικόνα 14

Σχετικά με τη χρησιμότητα τους, πρώτον, η λίστα γειτνίασης αξιοποιείται στην πλήρη ανάλυση της συνδεσιμότητας του γράφου (σε σχέση με τον πίνακα γειτνίασης είναι οικονομικότερη από άποψη χώρου/μνήμης, κυρίως για μεγάλου μεγέθους αραιούς γράφους). Συγκεκριμένα, χρησιμοποιείται για την υλοποίηση του αλγορίθμου dfs (depth first search), ο κώδικας του οποίου βρίσκεται στην εικόνα 15 (εντοπίζεται στο αρχείο “networks_algorithms.py”). Για κάθε κορυφή του γράφου, ο αλγόριθμος dfs [9] επισκέπτεται τις γειτονικές κορυφές, με αναδρομική κλήση στον εαυτό του επισκέπτεται τις γειτονικές κορυφές των γειτόνων και ούτω καθεξής. Έτσι, για κάθε κορυφή του γράφου, βρίσκει όλες τις προσβάσιμες από αυτήν κορυφές. Για παράδειγμα, στον γράφο της εικόνας 13, η κορυφή 2 δεν είναι προσβάσιμη από τις υπόλοιπες κορυφές και η κορυφή 6 δεν έχει πρόσβαση στις άλλες κορυφές, άρα ο γράφος δεν είναι ισχυρώς συνδεδεμένος.

```

62 # depth_first_search function is used to perform a depth first search on a graph represented by an adjacency list
63 def dfs_for_connectivity(graph, vertex, visited_vertices):
64     visited_vertices.add(vertex) # visited vertices are added to the set of visited_vertices
65     for neighbor in graph[vertex]: # go through all the neighbors of the vertex
66         if neighbor not in visited_vertices: # if the neighbor has not been visited yet
67             dfs_for_connectivity(graph, neighbor, visited_vertices) # perform a depth first search on the neighbor

```

Εικόνα 15

Δεύτερον, ο πίνακας γειτνίασης διαθέτει μία πολύ ενδιαφέρουσα ιδιότητα που περιγράφεται στο [5] και επιτρέπει την εύρεση του αριθμού των περιπάτων (walks) συγκεκριμένου πλήθους ακμών που συνδέουν μία κορυφή του γράφου με μία άλλη. Ειδικότερα, το (i, j) στοιχείο της n -οστής δύναμης του πίνακα γειτνίασης, έστω A^n , δίνει τον αριθμό των περιπάτων πλήθους ακμών n από την κορυφή i προς την κορυφή j .

Ενδεικτικά παραδείγματα επίλυσης του προβλήματος στη γραφική διεπαφή:

Το σενάριό μας ουσιαστικά επιβάλλει τη βελτιστοποίηση ως προς τον χρόνο. Εντούτοις, η γνώση των χρονικών βαρών απαιτεί μια συνολική εποπτεία της ακριβούς κατάστασης του οδικού δικτύου της πόλης ανά τακτά και μικρά χρονικά διαστήματα, λόγω των διαρκώς και τυχαία μεταβαλλόμενων χρονικών βαρών (π.χ. η κυκλοφοριακή συμφόρηση μπορεί να αλλάξει σε έναν δρόμο ή μπορεί να ολοκληρωθούν τα οδικά έργα), κάτι που εκ των πραγμάτων δεν είναι δυνατόν να συμβεί στην πραγματικότητα. Επιπλέον, προϋποθέτει τη μη αλλαγή των χρονικών βαρών κατά τη διάρκεια κίνησης του ασθενοφόρου, ή τουλάχιστον επιτρέπει μονάχα τη μεταβολή τους στα πλαίσια που ορίζει η ανάλυση ευαισθησίας, που θα εξηγηθεί περισσότερο στην αμέσως επόμενη ενότητα “Εφαρμογή ανάλυσης ευαισθησίας και χαρακτηριστικό παράδειγμα”. Από την άλλη πλευρά, η βελτιστοποίηση ως προς τον χώρο, παρόλο που δεν λαμβάνει άμεσα υπόψη τον χρόνο, μας παρέχει μια καλή εκτίμηση της χρονικά συντομότερης διαδρομής (καθώς αρκετές φορές η χωρικά και χρονικά συντομότερη διαδρομή συμπίπτουν) και διαθέτει το σημαντικό πλεονέκτημα της σταθερότητας των χωρικών βαρών (αφού τα μήκη των δρόμων παραμένουν σταθερά μακροπρόθεσμα). Για τους λόγους αυτούς θα ακολουθήσουν παραδείγματα και με τα δύο κριτήρια βελτιστοποίησης.

Για τη μοντελοποίηση και επίλυση του προβλήματος με ακέραιο προγραμματισμό σε κώδικα `python` χρησιμοποιήθηκε το πακέτο `rymp` (PyMathProg – Python Mathematics Programming), του οποίου η εγκατάσταση μπορεί να γίνει στο terminal με την εντολή “`pip install rymp`”, (οι εγκαταστάσεις όλων των απαιτούμενων βιβλιοθηκών του προγράμματος μπορούν να γίνουν με τη βοήθεια του διαχειριστή πακέτων `pip`). Όπως αναφέρεται και στο documentation του πακέτου [7], το PyMathProg είναι ένα εύκολο στη χρήση και ευέλικτο μαθηματικό περιβάλλον προγραμματισμού για την `python`. Είναι πολύ χρήσιμο για τα προβλήματα γραμμικού προγραμματισμού, καθώς προσφέρει μεγάλη ευελιξία στη μοντελοποίηση, ανάλυση, τροποποίηση και επίλυσή τους. Η βελτιστοποίηση πραγματοποιείται με τη χρήση πακέτων βελτιστοποίησης ανοιχτού κώδικα όπως το GNU Linear Programming Kit (GLPK), στο οποίο αποκτά πρόσβαση το PyMathProg μέσω του wrapper `swiglpk`. Ο συνηθέστερος αλγόριθμος που υλοποιείται είναι η μέθοδος Simplex. Επειδή το πρόβλημα είναι ακέραιο χρησιμοποιείται από το πρόγραμμα η μέθοδος Long-step Dual Simplex.

Θέλουμε να μεταφράσουμε το τελικό μοντέλο ακέραιου προγραμματισμού που προέκυψε στην προηγούμενη ενότητα “Μοντελοποίηση του shortest path με γραμμικό / ακέραιο προγραμματισμό” σε `python` κώδικα. Προγραμματιστικά είναι πιο χρήσιμες και αποδοτικές οι πράξεις με διανύσματα και πίνακες, συνεπώς ο στόχος μας είναι η μετατροπή του μοντέλου στην πρότυπη μορφή ακέραιου προγραμματισμού με πίνακες. Θεωρούμε αρχικά ότι το πλήθος των κορυφών του γράφου είναι n και το πλήθος των ακμών του γράφου (μετά τη διάσπαση των μη κατευθυνόμενων σε κατευθυνόμενες) είναι m . Πρώτον, οι μεταβλητές απόφασης αναφέρονται στις ακμές και είναι το διάνυσμα στήλης x , μήκους m . Δεύτερον, η αντικειμενική συνάρτηση είναι $c^T \cdot x$, όπου c είναι το διάνυσμα στήλης των βαρών των ακμών του γράφου, μήκους m . Τρίτον, οι ισοτικοί περιορισμοί γράφονται $A \cdot x = b$, όπου A είναι ένας πίνακας διαστάσεων $n \times m$ και b είναι ένα διάνυσμα μήκους n . Κάθε γραμμή του A και το αντίστοιχο στοιχείο του b

αναφέρονται σε κάποιον από τους n ισοτικούς περιορισμούς. Τα στοιχεία του A και του b υπολογίζονται ως εξής:

$$a_{i,j} = \begin{cases} 1 & \text{αν } \eta_j \text{ είναι εξερχόμενη ακμή της κορυφής } i & i = 1, \dots, n \\ 0 & \text{αν } \eta_j \text{ δεν συνδέεται με την κορυφή } i & j = 1, \dots, m \\ -1 & \text{αν } \eta_j \text{ είναι εισερχόμενη ακμή στην κορυφή } i \end{cases}$$

$$b_k = \begin{cases} 1 & \text{αν } \eta_k \text{ είναι η κορυφή αφετηρίας} \\ 0 & \text{αν } \eta_k \text{ είναι εσωτερική} \\ -1 & \text{αν } \eta_k \text{ είναι η κορυφή προορισμού} \end{cases} \quad k = 1, \dots, n$$

Τελικά, η ολοκληρωμένη πρότυπη μορφή ακέραιου προγραμματισμού του μοντέλου είναι η ακόλουθη, όπου ο πίνακας A και το διάνυσμα c παραμένουν σταθερά και για τις δύο διαδρομές (ασθενοφόρο -> ασθενής, ασθενής -> νοσοκομείο), καθώς εξαρτώνται αποκλειστικά από τη δομή του γράφου – οδικού δικτύου, ενώ το διάνυσμα b διαφέρει, διότι εξαρτάται από τις κορυφές αφετηρίας και προορισμού. Στην εικόνα 16 φαίνονται οι τρόποι υπολογισμού των x , A , b και c , που τελούνται στη συνάρτηση “find_shortest_path_with_IP” του αρχείου “networks.py”.

$$\min_{\text{function}} \left(\frac{\text{Cost}}{c^T \cdot x} \right)$$

$$A \cdot x = b$$

$$x_k \in \{0, 1\}, k = 1, \dots, m$$

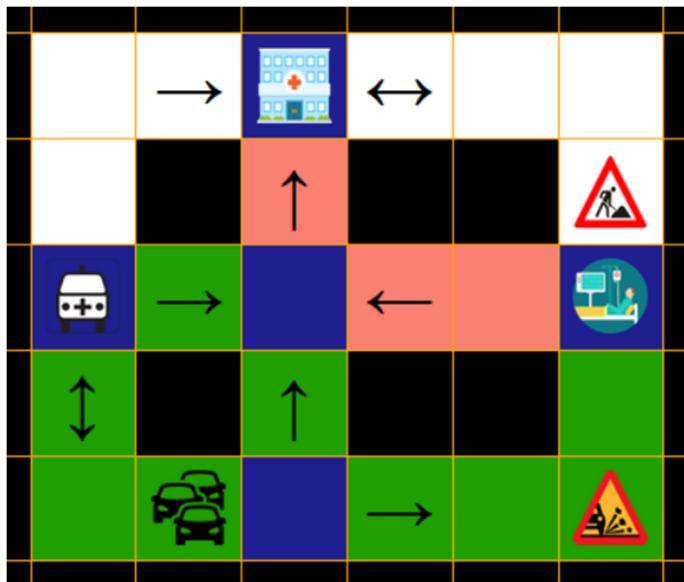
```

639 # create the decision variables that will be used in the optimization problem, based on the edges and their directionality
640 for i in range(len(adjacency_list)):
641     for j in list(adjacency_list.values())[i]:
642         self.decision_variables.append((list(adjacency_list.keys())[i]-1, j-1))
643 self.decision_variables_names_list = []
644 for decision_variable in self.decision_variables:
645     self.decision_variables_names_list.append(f"x_{decision_variable[0]+1}_{decision_variable[1]+1}")
652 # the matrix A of the coefficients of the constraints (the same for both paths)
653 self.A = np.zeros((len(self.graph_vertices), len(self.decision_variables)), dtype = int)
654 # the vector b at the right side of the constraints for the path from the ambulance to the patient
655 self.b_ap = np.zeros((self.A.shape[0], 1), dtype = int)
656 # the vector b at the right side of the constraints for the path from the patient to the hospital
657 self.b_ph = np.zeros((self.A.shape[0], 1), dtype = int)
658 # create the matrix A
659 for i in range(self.A.shape[0]):
660     for j in range(self.A.shape[1]):
661         if (i, j) in self.decision_variables:
662             self.A[i][self.decision_variables.index((i, j))] = 1
663         if (j, i) in self.decision_variables:
664             self.A[i][self.decision_variables.index((j, i))] = -1
665 # create the vectors b_ap and b_ph
666 self.b_ap[ambulance_vertex][0] = 1
667 self.b_ap[patient_vertex][0] = -1
668 self.b_ph[patient_vertex][0] = 1
669 self.b_ph[hospital_vertex][0] = -1
670
671 # calculate weights of all the edges based on the current optimization criterion
672 self.costs = []
673 for edge in self.decision_edges_list: # go through all the graph edges
674     if self.optimization_type == "time": # compute time weights of edges
675         self.costs.append(sum([self.time_weights_values[self.network_units_list[x][y].time_weight_name] for (x, y) in self.graph_edges[edge]])) + 1
676     elif self.optimization_type == "length": # compute length weights of edges
677         self.costs.append(netalg.get_manhattan_distance(edge[0], edge[1]))
678
679 # create the vector c
680 self.c = np.array(self.costs) # the vector c of the coefficients of the objective function

```

Εικόνα 16

- Πρώτο παράδειγμα:



Η άσπρη διαδρομή είναι η χρονικά συντομότερη από το ασθενοφόρο προς τον ασθενή και η ροζ διαδρομή είναι η χρονικά συντομότερη από τον ασθενή προς το νοσοκομείο. Η άσπρη διαδρομή είναι η $1 \rightarrow 2 \rightarrow 5$, γι' αυτό η αντίστοιχη λύση είναι η $x_{1,2} = x_{2,5} = 1$. Η ροζ διαδρομή είναι η $5 \rightarrow 4 \rightarrow 2$, οπότε η αντίστοιχη λύση είναι η $x_{5,4} = x_{4,2} = 1$. Οι κορυφές 1, 5 και 2 είναι οι θέσεις του ασθενοφόρου, του ασθενούς και του νοσοκομείου αντίστοιχα.

Εικόνα 17

```
A (common for the two problems) =
[[ 1  1  1  0 -1  0  0  0  0  0]
 [-1  0  0  1  0  0  0 -1 -1  0]
 [ 0 -1  0  0  1  1  1  0  0  0]
 [ 0  0 -1  0  0 -1  0  1  0 -1]
 [ 0  0  0 -1  0  0 -1  0  1  1]]
```

```
b for ambulance → patient:
[[ 1]
 [ 0]
 [ 0]
 [ 0]
 [-1]]
```

```
b for patient → hospital:
[[ 0]
 [-1]
 [ 0]
 [ 0]
 [ 1]]
```

```
c (the costs of movement):
[4 5 2 6 5 2 7 2 6 3]
```

Path 1 model and solution:

```
\* Problem: Find shortest path *\n\nMinimize\nCost_function: + 4 x_1_2 + 5 x_1_3 + 2 x_1_4 + 6 x_2_5 +\n5 x_3_1\n+ 2 x_3_4 + 7 x_3_5 + 2 x_4_2 + 6 x_5_2 + 3 x_5_4
```

Subject To

```
R1: - x_3_1 + x_1_4 + x_1_3 + x_1_2 = 1
R2: - x_5_2 - x_4_2 + x_2_5 - x_1_2 = 0
R3: + x_3_5 + x_3_4 + x_3_1 - x_1_3 = 0
R4: - x_5_4 + x_4_2 - x_3_4 - x_1_4 = 0
R5: + x_5_4 + x_5_2 - x_3_5 - x_2_5 = -1
```

Path 2 model and solution:

```
\* Problem: Find shortest path *\n\nMinimize\nCost_function: + 4 x_1_2 + 5 x_1_3 + 2 x_1_4 + 6 x_2_5 +\n5 x_3_1\n+ 2 x_3_4 + 7 x_3_5 + 2 x_4_2 + 6 x_5_2 + 3 x_5_4
```

Subject To

```
R1: - x_3_1 + x_1_4 + x_1_3 + x_1_2 = 0
R2: - x_5_2 - x_4_2 + x_2_5 - x_1_2 = -1
R3: + x_3_5 + x_3_4 + x_3_1 - x_1_3 = 0
R4: - x_5_4 + x_4_2 - x_3_4 - x_1_4 = 0
R5: + x_5_4 + x_5_2 - x_3_5 - x_2_5 = 1
```

Εικόνα 18

Το πρώτο αυτό παράδειγμα αποτελεί ακριβή μεταφορά του γράφου της εικόνας 12 (στην ενότητα “Μοντελοποίηση του shortest path με γραμμικό / ακέραιο προγραμματισμό”) στο περιβάλλον της γραφικής διεπαφής. Στην εικόνα 17 παρατίθεται η σχεδίαση του αντίστοιχου οδικού δικτύου του γράφου και χρωματίζονται οι συντομότερες χρονικά διαδρομές. Στην εικόνα 18 γράφονται τα A , b και c που παρήχθησαν από το πρόγραμμα (παρατίθενται και στη συνέχεια), καθώς και τα μοντέλα ακέραιου προγραμματισμού που δημιουργήθηκαν μέσω αυτών. Στην εικόνα 19 προβάλλεται ο κώδικας σε python (αρχείο “graph_solver.py”), που γράφτηκε με σκοπό την επίλυση του προβλήματος ακέραιου προγραμματισμού για αυτό το παράδειγμα, καθώς και για οποιοδήποτε άλλο σχεδιαζόμενο οδικό δίκτυο στη γραφική διεπαφή. Η συνάρτηση “solve_IP_problem” του κώδικα λαμβάνει ορίσματα τα x , A , b και c .

$$x = [x_{1,2} \quad x_{1,3} \quad x_{1,4} \quad x_{2,5} \quad x_{3,1} \quad x_{3,4} \quad x_{3,5} \quad x_{4,2} \quad x_{5,2} \quad x_{5,4}]^T$$

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 1 \end{bmatrix}$$

$$b_{A \rightarrow P} = [1 \quad 0 \quad 0 \quad 0 \quad -1]^T$$

$$b_{P \rightarrow H} = [0 \quad -1 \quad 0 \quad 0 \quad 1]^T$$

$$c = [4 \quad 5 \quad 2 \quad 6 \quad 5 \quad 2 \quad 7 \quad 2 \quad 6 \quad 3]^T$$

```

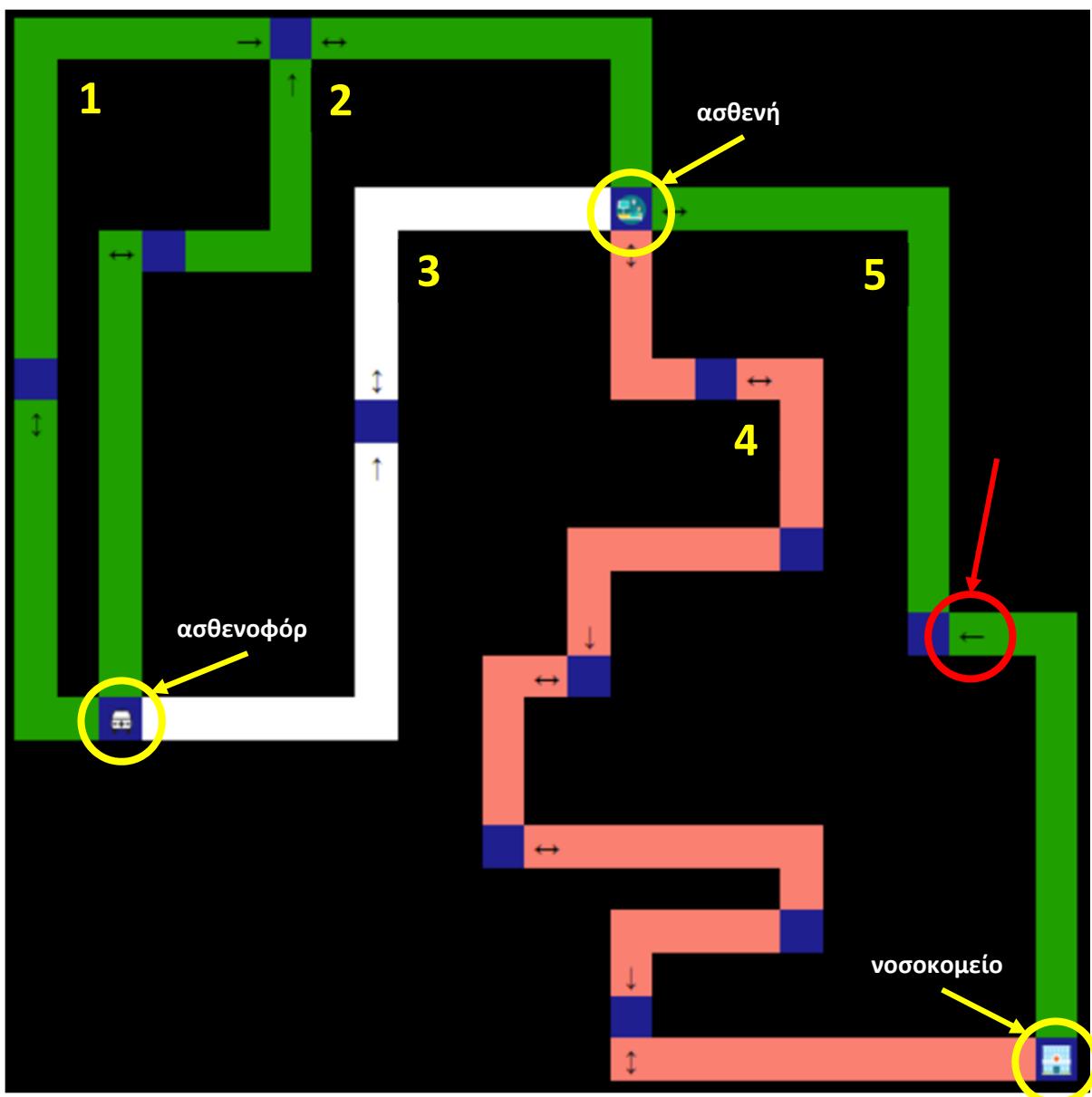
1 import pymprog
2
3 def solve_IP_problem(A, b, c, decision_variables_names, path_name = None):
4     model = pymprog.begin('Find shortest path') # begin modelling the problem
5     model.verbose(False) # be analytic when analysing the model
6     n, m = len(A), len(A[0]) # dimensions of matrix A
7     # m is the number of decision variables and n is the number of constraints
8
9     # create a list of n integer and non negative variables
10    x_vars = []
11    for i in range(m):
12        x_vars.append(model.var(decision_variables_names[i], kind = int))
13    for i in range(m):
14        x_vars[i] <= 1 # the decision variables are bounded by 1 (they are binary)
15
16    # create the constraints, y is the list of dual variables
17    y_vars = [None] * n
18    for i in range(n):
19        y_vars[i] = sum(A[i][j] * x_vars[j] for j in range(m)) == b[i]
20
21    # create the objective function and maximize it
22    model.minimize(sum(c[i] * x_vars[i] for i in range(m)), "Cost function")
23
24    # solve the model
25    model.solve()
26    if path_name != None:
27        pymprog.save(mip = f"problem_solution_{path_name}.txt") # save the problem's solution
28        pymprog.save(clp = f"problem_model_{path_name}.txt") # save the problem's model
29
30    model.end() # do away with the model
31
32    return([x_vars[i].primal for i in range(m)]) # return the optimal values of the decision variables

```

Εικόνα 19

- Δεύτερο παράδειγμα:

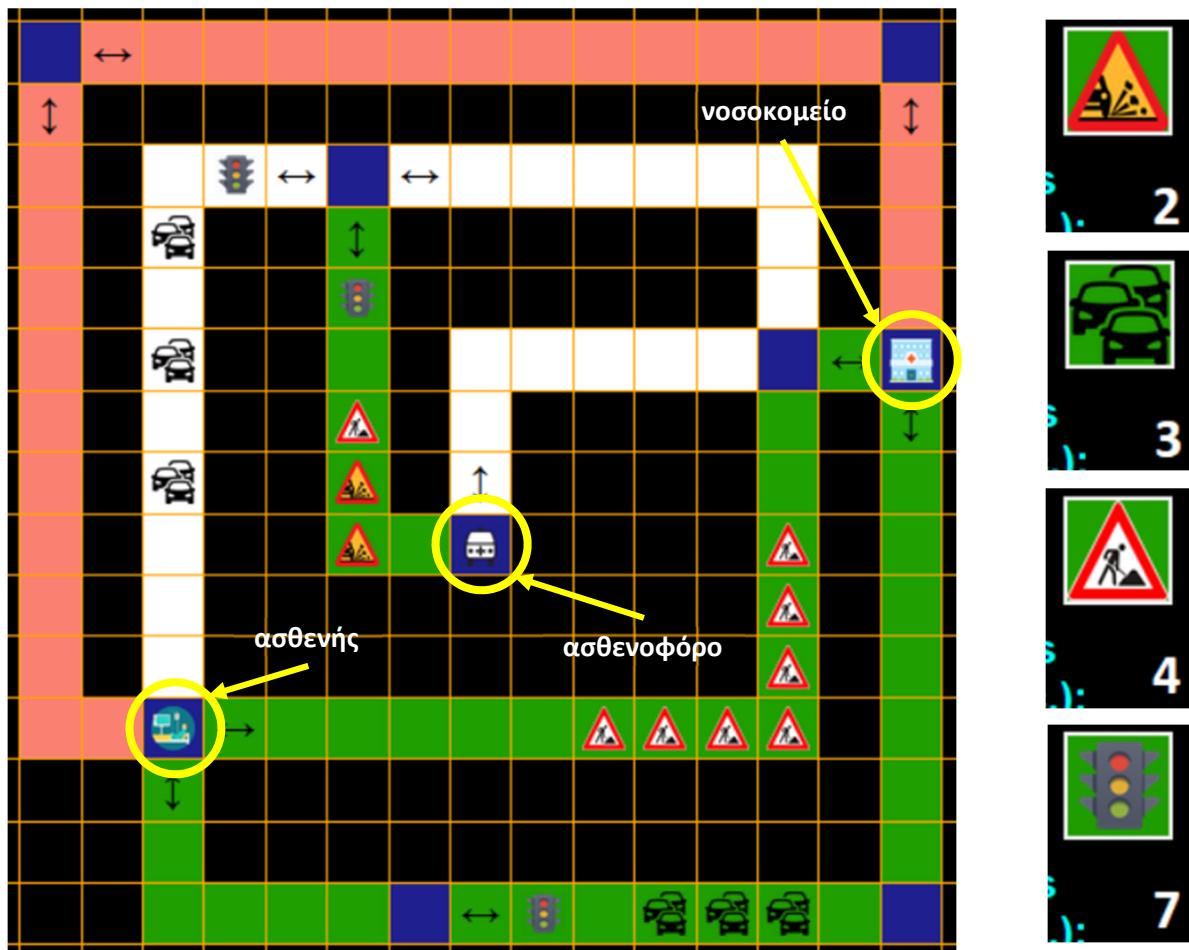
Στο συγκεκριμένο παράδειγμα πραγματοποιείται βελτιστοποίηση ως προς το μήκος της διαδρομής, στον μεγάλο γράφο της εικόνας 20. Υπάρχουν τρεις δυνατές διαδρομές από το ασθενοφόρο προς τον ασθενή και δύο από τον ασθενή προς το νοσοκομείο. Είναι αρκετά φανερό και με απλή οπτική εξέταση, ότι η συντομότερη διαδρομή από το ασθενοφόρο προς τον ασθενή είναι αυτή που έχει σημειωθεί με άσπρο χρώμα, δηλαδή η διαδρομή 3 βάσει της αρίθμησης που έχει γίνει. Ειδικότερα, το μήκος της συγκεκριμένης διαδρομής είναι 24, ενώ το μήκος της 1 είναι 36 και το μήκος της 2 είναι 32. Για τη διαδρομή όμως από τον ασθενή προς το νοσοκομείο φαίνεται με μια πρόχειρη ματιά ότι η διαδρομή 5 έχει αρκετά μικρότερο μήκος από την 4. Πράγματι, το μήκος της 4 είναι 52 και το μήκος της 5 είναι 30, εντούτοις επιλέγεται η διαδρομή 4 και τονίζεται με ροζ χρώμα. Αυτό συμβαίνει γιατί η διαδρομή 5 είναι αδύνατη εξαιτίας της κατευθυντικότητας της ακμής που έχει σημειωθεί με τον κόκκινο κύκλο.



Εικόνα 20

- Τρίτο παράδειγμα:

Σ' αυτό το παράδειγμα πραγματοποιείται βελτιστοποίηση ως προς τη χρονική διάρκεια της διαδρομής στον μη κατευθυνόμενο και συνεκτικό γράφο της εικόνας 21, όπου έχουν σημειωθεί και τα κόστη των χρονικών καθυστερήσεων. Στην εικόνα 22 φαίνονται οι βέλτιστες διαδρομές, οι ακμές από τις οποίες αποτελούνται και τα χρονικά τους κόστη (σχετικά και απόλυτα). Ακόμα, δίνονται και οι μεταβλητές απόφασης που λαμβάνουν την τιμή 1 στις δύο διαδρομές.



Εικόνα 21

Display information:		
Info: The graph is strongly connected.		
Solved: Shortest paths found with time as an optimization criterion! <ul style="list-style-type: none"> 1) Shortest path: ambulance → patient <ul style="list-style-type: none"> - Path edges: [3, 4, 6] - Total time cost: 42 (2.52 sec) 2) Shortest path: patient → hospital <ul style="list-style-type: none"> - Path edges: [12, 10, 11] - Total time cost: 32 (1.92 sec) 	ασθενοφόρο -> ασθενής	ασθενής -> νοσοκομείο
	7 x_3_1 * 0 8 x_3_4 * 1 9 x_3_5 * 0 10 x_3_7 * 0 11 x_4_3 * 0 12 x_4_1 * 1 13 x_4_5 * 0 14 x_5_4 * 0 15 x_5_3 * 1 * 0	3 x_1_4 * 0 4 x_1_9 * 1 19 x_7_6 * 0 20 x_7_8 * 0 21 x_8_7 * 1 22 x_8_9 * 0 23 x_9_8 * 1 24 x_9_1 * 0

Εικόνα 22

Εφαρμογή ανάλυσης ευαισθησίας και χαρακτηριστικό παράδειγμα:

Η εφαρμογή μιας ανάλυσης ευαισθησίας στο σενάριο μας είναι πολύ λογική αν κριτήριο βελτιστοποίησης αποτελεί ο χρόνος, αφού τα χρονικά βάρη του προβλήματος είναι εύκολο να μεταβληθούν άμεσα και να παρουσιάσουν μεγάλες και γρήγορες αλλαγές (όπως γίνεται στην περίπτωση της “κυκλοφοριακής κίνησης” και των “φαναριών”). Αντιθέτως, θα ήταν μη πρακτικό να ασχοληθούμε με την ανάλυση χωρικών μεταβολών, αφού τα μήκη των δρόμων πρακτικά παραμένουν σταθερά και αμετάβλητα για πολύ μεγάλο χρονικό διάστημα (οι μόνες αλλαγές που μπορούν να γίνουν αφορούν στην προσθήκη νέων δρόμων, δηλαδή στη δημιουργία νέου οδικού δικτύου και κατ’ επέκταση νέου γράφου). Το παράδειγμα λοιπόν που θα παρουσιαστεί έχει ως κριτήριο βελτιστοποίησης αποκλειστικά τον χρόνο. Εξετάζει ποιες είναι οι αλλαγές των αντικειμενικών συντελεστών (σχετικά χρονικά κόστη των ακμών) που μπορούν να παράγουν καινούριες συντομότερες διαδρομές.

Ένα από τα μειονεκτήματα της ανάλυσης ευαισθησίας είναι ότι αναφέρεται στην αλλαγή ενός συντελεστή κάθε φορά, υποθέτοντας ότι όλοι οι υπόλοιποι παραμένουν σταθεροί, κάτι που ίσως δεν είναι τόσο ρεαλιστικό. Ενδεικτικό παράδειγμα αυτού αποτελεί η παρακάτω εικόνα 23, όπου είναι φανερό ότι αν το χρονικό βάρος του “φαναριού” μειωθεί, π.χ. το φανάρι γίνει από κόκκινο πράσινο, τότε θα αυξηθεί η κίνηση στον δρόμο 2 λόγω της εισροής οχημάτων από τον δρόμο 1, με αποτέλεσμα να μειωθεί η κυκλοφοριακή συμφόρηση στον δρόμο 1. Οπότε αλλάζουν τα χρονικά κόστη της “κυκλοφοριακής κίνησης” και στις δύο ακμές ταυτόχρονα.



Εικόνα 23

Ας εξετάσουμε τώρα πώς πραγματοποιείται η ανάλυση ευαισθησίας. Έστω ότι έχουμε βρει τη συντομότερη διαδρομή στον γράφο μας (οδικό δίκτυο της πόλης) από τον κόμβο αφετηρίας προς τον κόμβο προορισμού (από το ασθενοφόρο προς τον ασθενή ή από τον ασθενή προς το νοσοκομείο). Όπως αναφέρθηκε, περιοριζόμαστε στη μεταβολή μόνο ενός αντικειμενικού συντελεστή, δηλαδή στη μεταβολή του σχετικού χρονικού κόστους μόνο μιας ακμής του γράφου

κάθε φορά. Με τον τρόπο που έχει δομηθεί το σενάριο επιτρέπονται μονάχα ακέραιες μεταβολές. Χρειάζεται να διακρίνουμε δύο ξεχωριστές περιπτώσεις: το χρονικό κόστος που μεταβάλλουμε ανήκει σε ακμή που συμπεριλαμβάνεται στη συντομότερη διαδρομή ή ανήκει σε ακμή που δεν μετέχει στη συντομότερη διαδρομή. Αναλόγως την περίπτωση υπάρχουν κάποιες σημαντικές διαφορές στην ανάλυση, που επισημαίνονται διεξοδικά στη συνέχεια. Υποθέτουμε ότι η αρχική συντομότερη διαδρομή p_{old} από την κορυφή αφετηρίας s προς την κορυφή προορισμού d , έχει συνολικό σχετικό χρονικό κόστος $Z \in \mathbb{N}$ και η ακμή e_{test} που χειριζόμαστε, διαθέτει αρχικό σχετικό χρονικό κόστος $c \in \mathbb{N}$.

- 1^η περίπτωση – η ακμή e_{test} βρίσκεται εντός της p_{old} :

Από τη στιγμή που η e_{test} είναι εντός της συντομότερης διαδρομής, τότε μας ενδιαφέρει πόσο μεγάλο μπορεί να γίνει το χρονικό κόστος της πριν αλλάξει η συντομότερη διαδρομή, δηλαδή πριν αλλάξουν οι ακμές που την αποτελούν. Προφανώς, αν μειωθεί το κόστος της συντομότερης διαδρομής, αυτή δεν πρόκειται να αλλάξει, αφού το κόστος της ήταν ήδη το ελάχιστο. Έστω ότι η μέγιστη αύξηση που αναζητούμε, αμέσως πριν αλλάξει η p_{old} , είναι η Δc_{max} και γι' αυτήν την τιμή το νέο κόστος της e_{test} γίνεται $c' = c + \Delta c_{max} \in \mathbb{N}$. Το c' κινείται στα όρια $c \leq c' \leq t_{rel,graph}$, όπου $t_{rel,graph}$ είναι το συνολικό σχετικό χρονικό κόστος ολόκληρου του γράφου. Προφανώς, αν δεν έχει αλλάξει η συντομότερη διαδρομή μέχρι το σημείο $c' = t_{rel,graph}$ δεν πρόκειται να αλλάξει ποτέ, δηλαδή $\Delta c_{max} = +\infty$. Αν υπάρχει πεπερασμένο Δc_{max} , τότε για $c' = c + \Delta c_{max}$ η p_{old} εμφανίζει το ίδιο ακριβώς σχετικό χρονικό κόστος (το οποίο είναι μάλιστα $Z' = Z + \Delta c_{max}$) με άλλη μία τουλάχιστον διαδρομή στον γράφο που πηγαίνει από την s προς τη d (χρησιμοποιώντας την ορολογία του ακέραιου προγραμματισμού, οι βασικές εφικτές λύσεις γίνονται περισσότερες από μία). Άρα, για $c' = c + \Delta c_{max} + 1$ είναι $Z' = Z + \Delta c_{max} + 1$ και προκύπτει μία νέα συντομότερη διαδρομή p_{new} , η οποία έχει συνολικό σχετικό χρονικό κόστος $Z_{new} = Z + \Delta c_{max}$. Για $c' > c + \Delta c_{max} + 1$ η p_{new} παραμένει η συντομότερη διαδρομή με κόστος $Z + \Delta c_{max}$ σταθερό, αφού η e_{test} δεν ανήκει στην p_{new} .

Το σχήμα της εικόνας 24 αποσαφηνίζει περισσότερο όσα ειπώθηκαν προηγουμένως. Δείχνει την περίπτωση όπου υφίσταται πεπερασμένο $\Delta c_{max} \in \mathbb{N}$, με την αλλαγή της συντομότερης διαδρομής να συμβαίνει αμέσως μετά την αύξηση του κόστους της e_{test} κατά Δc_{max} (μετάβαση από τα κίτρινα κελιά της p_{old} στα γαλάζια κελιά της p_{new}).

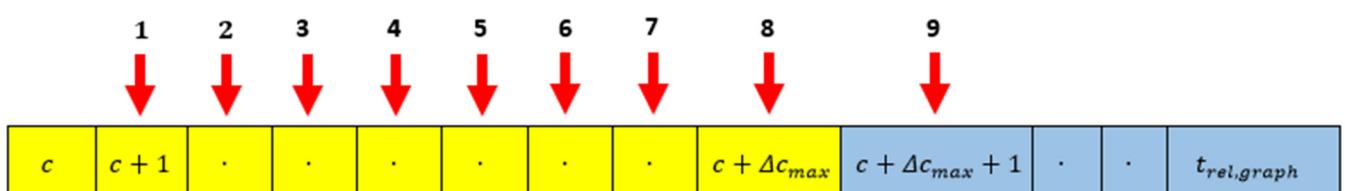
Αύξηση του κόστους της e_{test} κατά +1 από το κάθε κελί στο επόμενο	→						
Κόστος της e_{test}	c	$c + 1$...	$c + \Delta c_{max}$	$c + \Delta c_{max} + 1$	$c + \Delta c_{max} + 2$...
Κόστος της συντομότερης διαδρομής	Z	$Z + 1$...	$Z + \Delta c_{max}$	$Z + \Delta c_{max}$	$Z + \Delta c_{max}$...
Συντομότερη διαδρομή	p_{old}				p_{new}		

Εικόνα 24

Παρατηρούμε ότι η αύξηση του κόστους της e_{test} εκτελείται σταδιακά κατά $\Delta c = +1$ σε κάθε βήμα. Σε κάθε βήμα πρέπει να ελέγχουμε με επίλυση του νέου προβλήματος ακέραιου προγραμματισμού (που προκύπτει εξαιτίας της αλλαγής του κόστους της e_{test}), αν έχει αλλάξει η συντομότερη διαδρομή, έτσι ώστε να υπολογίσουμε τελικά το Δc_{max} που μας ενδιαφέρει. Η διαδικασία όμως αυτή είναι αρκετά χρονοβόρα προγραμματιστικά, διότι καταλήγουμε να λύνουμε δεκάδες, ίσως και εκατοντάδες προβλήματα γραμμικού προγραμματισμού. Πρέπει λοιπόν να σκεφτούμε έναν πιο έξυπνο τρόπο για τον εντοπισμό του μεταχιμίου μεταξύ p_{old} και p_{new} . Μια εξαιρετικά αποδοτική προσέγγιση που μειώνει κατά πολύ τον χρόνο εκτέλεσης της ανάλυσης ευαισθησίας είναι η δυαδική αναζήτηση (η υλοποίησή της πραγματοποιείται στον κώδικα της εικόνας 25, που εντοπίζεται στη συνάρτηση "apply_sensitivity_analysis" στο αρχείο "networks.py"). Αν π.χ. χρειαζόταν να επιλύσουμε το πολύ N στο πλήθος προβλήματα με τη γραμμική αύξηση του κόστους της e_{test} , με τη δυαδική αναζήτηση χρειάζεται να επιλύσουμε μονάχα το πολύ $\lceil \log_2 N \rceil$ προβλήματα. Η διαφορά χρόνου γίνεται ιδιαίτερα εμφανής στην εφαρμογή ανάλυσης ευαισθησίας σε μεγάλους σε μέγεθος γράφους. Έστω ότι η ακμή e_{test} είναι αυτή της εικόνας 26, όπου για γραμμική αύξηση χρειάζεται να λυθούν το πολύ 12 προβλήματα ακέραιου προγραμματισμού και τελικώς επιλύονται 9. Στην εικόνα 27 παρουσιάζεται η ανάλυση ευαισθησίας με δυαδική αναζήτηση για την ίδια ακμή, όπου ο αριθμός των προβλημάτων που απαιτείται να επιλυθούν μειώνεται σε 4 (με $\lceil \log_2 12 \rceil = 4$ το μέγιστο).

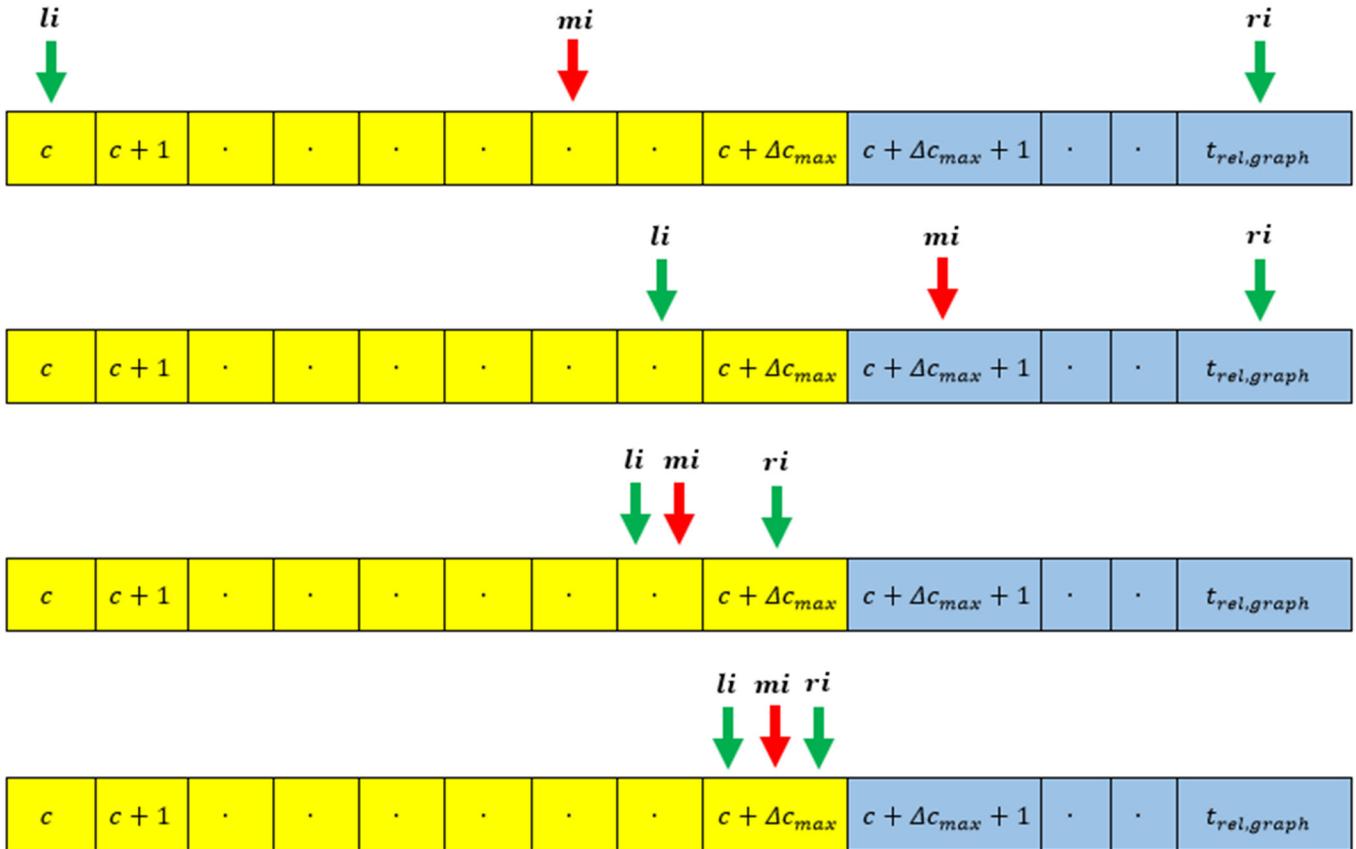
```
# apply sensitivity analysis on both shortest paths (path_1 and path_2)
for path_order in range(2): # path_order = 0 for path_1 and path_order = 1 for path_2
    for cost_order in range(len(self.c)):
        if self.old_shortest_paths[path_order][cost_order] == 1: # the edge is part of the shortest path
            # initialize the left index to be the cost of the edge (the minimum time cost) and the right index to be the total cost of the graph (the maximum time cost)
            left_index = self.c[cost_order] # left index of the binary search
            right_index = self.total_graph_cost # right index of the binary search
            costs_check = np.copy(self.c) # the costs_check are set to be the same as the original
            # we want to check if the path's cost is changed at the extreme cost case, if not then the edge cost does not affect the shortest path
            costs_check = self.change_cost(costs_check, right_index, cost_order)
            if np.dot(costs_check, self.old_shortest_paths[path_order]) != 1:
                np.dot(costs_check, graso.solver_IP_problem(self.A, self.b_vectors[path_order], costs_check, self.decision_variables_names_list))
                costs_sens = np.copy(self.c) # the costs_sens of the edges are initialized to be the same as the original
                while left_index <= right_index: # implementing binary search
                    middle_index = (left_index + right_index) // 2 # middle index of the binary search
                    new_costs = self.change_cost(costs_sens, middle_index, cost_order) # attempt to change the cost of the current edge
                    new_shortest_path = graso.solver_IP_problem(self.A, self.b_vectors[path_order], new_costs, self.decision_variables_names_list) # find the new shortest path
                    # when a new shortest path is found (with lower total cost than the previous shortest path) while changing the cost of the current edge do the following
                    if np.dot(new_costs, self.old_shortest_paths[path_order]) == np.dot(new_costs, new_shortest_path):
                        costs_sens = self.change_cost(costs_sens, middle_index, cost_order) # confirm the change of the cost of the current edge
                        left_index = middle_index + 1
                    else:
                        right_index = middle_index - 1
                if costs_sens[cost_order] != self.total_graph_cost:
                    # the edge cost affects the shortest path
                    cost_difference = costs_sens[cost_order] - self.c[cost_order]
                    self.ranges_till_paths[path_order] += f'{list(self.graph.edges.keys())[cost_order].index(self.decision_edges_list[cost_order]) + 1}"+\n" + {costs_sens[cost_order]} ({cost_difference}), "
```

Εικόνα 25



Η γραμμική αναζήτηση τερματίζεται στην 9^η επανάληψη, με το Δc_{max} να έχει βρεθεί στην 8^η.

Εικόνα 26



Όταν ο μεσαίος δείκτης mi βρίσκεται εντός της κίτρινης περιοχής, τότε στην επόμενη επανάληψη ο αριστερός δείκτης li γίνεται $mi + 1$ και ο δεξιός δείκτης ri διατηρείται σταθερός, ενώ όταν ο mi βρίσκεται εντός της γαλάζιας περιοχής, τότε στην επόμενη επανάληψη ο li παραμένει σταθερός και ο ri γίνεται $mi - 1$. Η δυαδική αναζήτηση τερματίζεται στην 5^η επανάληψη, επειδή είναι $li > ri$. Το Δc_{max} έχει βρεθεί στην 4^η επανάληψη και εντοπίζεται εκεί που δείχνει ο mi .

Εικόνα 27

Είναι αξιοσημείωτο ότι, αν μπορούμε να οδηγήσουμε στο μέγιστο όριό τους όλους τους συντελεστές της αντικειμενικής συνάρτησης που αντιστοιχούν στις ακμές εντός της p_{old} , χωρίς να αλλάξει η συντομότερη διαδρομή, αυτό σημαίνει ότι η p_{old} είναι η μοναδική διαδρομή από τον κόμβο αφετηρίας s προς τον κόμβο προορισμού d .

- 2^η περίπτωση – η ακμή e_{test} βρίσκεται εκτός της p_{old} :

Από τη στιγμή που η e_{test} είναι εκτός της συντομότερης διαδρομής, τότε μας ενδιαφέρει πόσο μπορεί να μειωθεί το χρονικό κόστος της πριν αλλάξει η συντομότερη διαδρομή. Προφανώς, αν αυξηθεί το κόστος της e_{test} , η συντομότερη διαδρομή δεν πρόκειται να αλλάξει, αφού η e_{test} δεν πρόκειται να βρίσκεται σε μια νέα συντομότερη διαδρομή με μεγαλύτερο κόστος από αυτό με το οποίο ξεκίνησε. Έστω ότι η μέγιστη μείωση που αναζητούμε, αμέσως πριν αλλάξει η p_{old} , είναι η Δc_{max} (κατ' απόλυτη τιμή), για την οποία το νέο κόστος της e_{test} γίνεται $c' = c - \Delta c_{max} \in \mathbb{N}$. Το c' κινείται στα όρια $c_{Manh} \leq c' \leq c$, όπου c_{Manh} είναι το (Manhattan)

μήκος της e_{test} , που αποτελεί και το ελάχιστο σχετικό χρονικό κόστος που δύναται να λάβει η ακμή. Αν η συντομότερη διαδρομή δεν έχει αλλάξει μέχρι το σημείο $c' = c_{Manh}$ σταματάμε την ανάλυση για την e_{test} . Αν όμως υπάρχει $0 < \Delta c_{max} \leq c - c_{Manh}$, τότε για $c' = c - \Delta c_{max}$ η p_{old} εμφανίζει το ίδιο ακριβώς σχετικό χρονικό κόστος (το οποίο διατηρείται μέχρι τότε διαρκώς στο Z) με άλλη μία τουλάχιστον διαδρομή στον γράφο που πηγαίνει από την s προς τη d . Συνεπώς ισχύει ότι, για $c' = c - \Delta c_{max} - 1$, προκύπτει μία νέα συντομότερη διαδρομή p_{new} , η οποία έχει συνολικό σχετικό χρονικό κόστος $Z_{new} = Z - 1$. Για $c' < c - \Delta c_{max} - 1$ η p_{new} παραμένει η συντομότερη διαδρομή και το κόστος της μειώνεται, αφού η e_{test} ανήκει στην p_{new} .

Το σχήμα της εικόνας 28 αποσαφηνίζει περισσότερο όσα ειπώθηκαν προηγουμένως. Δείχνει την περίπτωση όπου υφίσταται $\Delta c_{max} \in \mathbb{N}$, με $0 < \Delta c_{max} \leq c - c_{Manh}$. Η αλλαγή της συντομότερης διαδρομής συμβαίνει αμέσως μετά τη μείωση του κόστους της e_{test} κατά Δc_{max} (μετάβαση από τα κίτρινα κελιά της p_{old} στα γαλάζια κελιά της p_{new}).

Μείωση του κόστους της e_{test} κατά -1 από το κάθε κελί στο επόμενο	→	
Κόστος της e_{test}	c $c - 1$... $c - \Delta c_{max}$ $c - \Delta c_{max} - 1$ $c - \Delta c_{max} - 2$...	
Κόστος της συντομότερης διαδρομής	Z Z ... Z $Z - 1$ $Z - 2$...	
Συντομότερη διαδρομή	p_{old}	p_{new}

Εικόνα 28

Και σ' αυτήν την περίπτωση η δυαδική αναζήτηση βοηθάει σε μεγάλο βαθμό στην ταχεία εύρεση της ποσότητας Δc_{max} που μας ενδιαφέρει. Το αντίστοιχο κομμάτι κώδικα φαίνεται στην εικόνα 29 και εντοπίζεται στη συνάρτηση “apply_sensitivity_analysis” στο αρχείο “networks.py”.

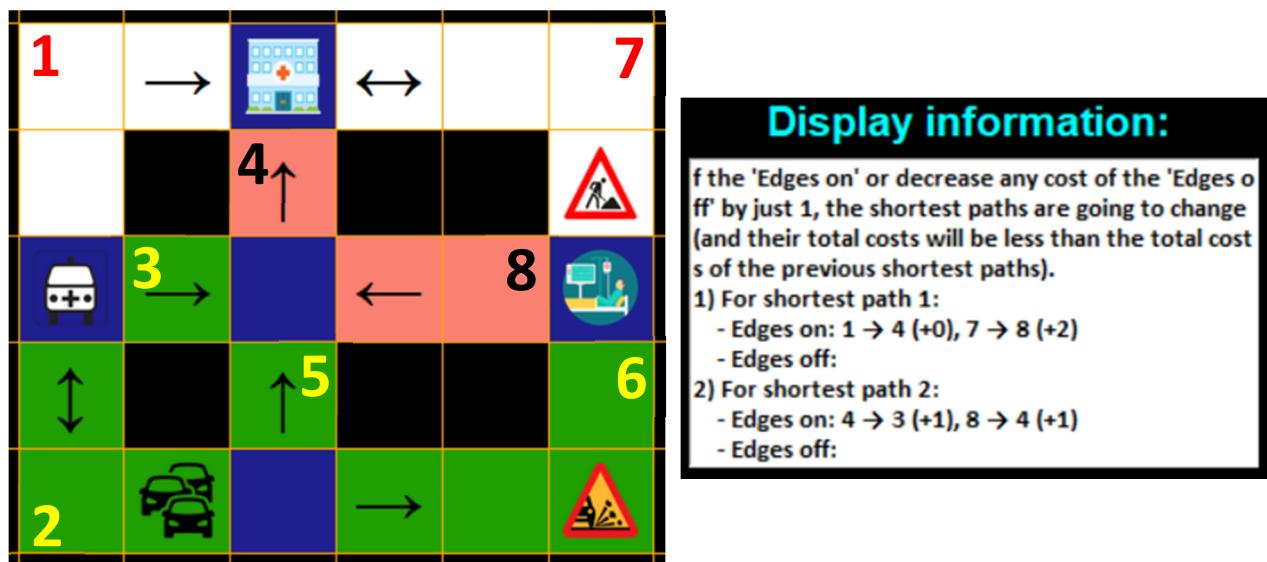
```

else: # the edge is not part of the shortest path
    # initialize the left index to be the length cost of the edge (the minimum time cost) and the right index to be the time cost of the edge (the maximum time cost)
    left_index = metalg.get_manhattan_distance(self.decision_edges_list[cost_order][0], self.decision_edges_list[cost_order][1]) # left index of the binary search
    right_index = self.c[cost_order] # right index of the binary search
    costs_check = np.copy(self.c) # the costs_check are set to be the same as the original
    # we want to check if the path's cost is changed at the extreme cost case, if not then the edge cost does not affect the shortest path
    costs_check = self.change_cost(costs_check, left_index, cost_order)
    if np.dot(costs_check, self.old_shortest_paths[path_order]) != \
        np.dot(costs_check, grasol.solve_IP_problem(self.A, self.b_vectors[path_order], costs_check, self.decision_variables_names_list)):
        costs_sens = np.copy(self.c) # the costs_sens of the edges are initialized to be the same as the original
        while left_index <= right_index: # implementing binary search
            middle_index = (left_index + right_index) // 2 # middle index of the binary search
            new_costs = self.change_cost(costs_sens, middle_index, cost_order) # attempt to change the cost of the current edge
            new_shortest_path = grasol.solve_IP_problem(self.A, self.b_vectors[path_order], new_costs, self.decision_variables_names_list) # find the new shortest path
            # when a new shortest path is found (with lower total cost than the previous shortest path) while changing the cost of the current edge do the following
            if np.dot(new_costs, self.old_shortest_paths[path_order]) == np.dot(new_costs, new_shortest_path):
                costs_sens = self.change_cost(costs_sens, middle_index, cost_order) # confirm the change of the cost of the current edge
                right_index = middle_index - 1
            else:
                left_index = middle_index + 1
        if costs_sens[cost_order] != metalg.get_manhattan_distance(self.decision_edges_list[cost_order][0], self.decision_edges_list[cost_order][1]):
            # the edge cost affects the shortest path
            cost_difference = self.c[cost_order] - costs_sens[cost_order]
            self.ranges_from_paths[path_order] += f"\n{list(self.graph.edges.keys()).index(self.decision_edges_list[cost_order]) + 1}" + \
                f" → {costs_sens[cost_order]} (-{cost_difference}), "

```

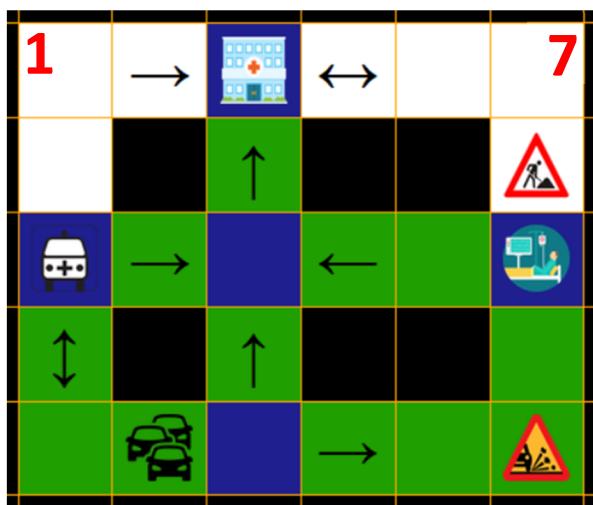
Εικόνα 29

Ας δείξουμε, τέλος, πώς εφαρμόζεται η ανάλυση ευαισθησίας σε ένα οδικό δίκτυο της γραφικής διεπαφής. Θα αξιοποιήσουμε πάλι το γνωστό παράδειγμα του γράφου της εικόνας 17. Οι χρονικές καθυστερήσεις “οδικά έργα”, “κυκλοφοριακή κίνηση” και “κακοτράχαλη οδός” φέρουν σχετικά βάρη 2, 2 και 3 αντίστοιχα. Θα εξετάσουμε τα αποτελέσματα της ανάλυσης ευαισθησίας και για τα δύο δρομολόγια. Οι αρχικές συντομότερες διαδρομές είναι οι {1, 7} (από το ασθενοφόρο προς τον ασθενή) και {8, 4} (από τον ασθενή προς το νοσοκομείο). Είναι εύκολο να διαπιστώσουμε ότι υπάρχουν 4 ξεχωριστές διαδρομές από το ασθενοφόρο προς τον ασθενή, οι {1, 7}, {3, 4, 7}, {2, 5, 4, 7} και {2, 6} (εδώ σημειώνονται οι ακμές που αποτελούν την κάθε διαδρομή). Ακόμα, υπάρχουν 2 διαφορετικές διαδρομές από τον ασθενή προς το νοσοκομείο, οι {7} και {8, 4}. Το αρχικό οδικό δίκτυο με αρίθμηση των ακμών φαίνεται στην εικόνα 30.

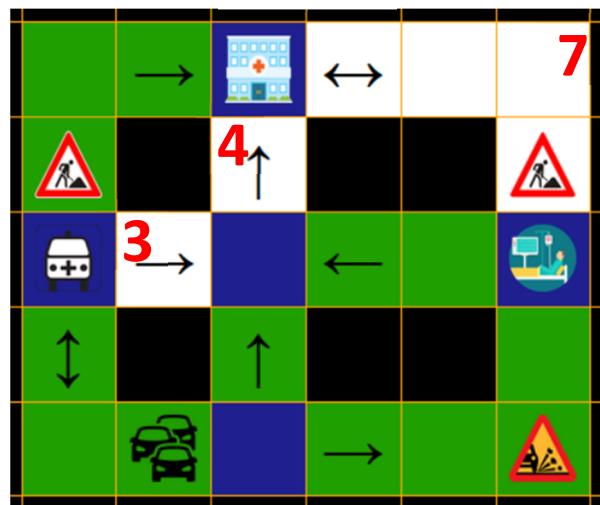


Εικόνα 30

$$\Delta c \mathbf{1}_{max} = 0$$

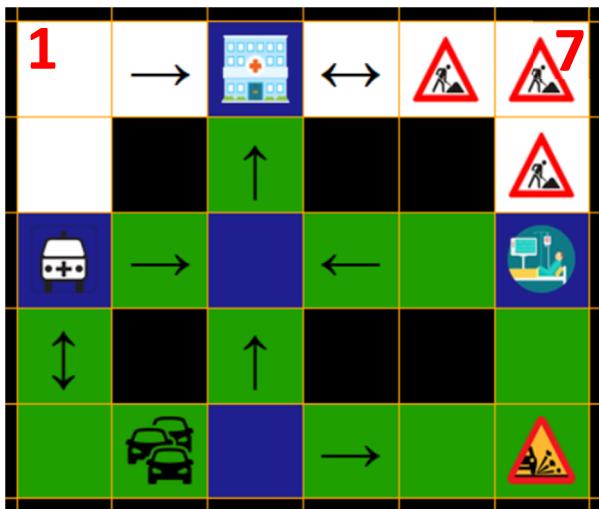


$$\Delta c \mathbf{1} = 1$$



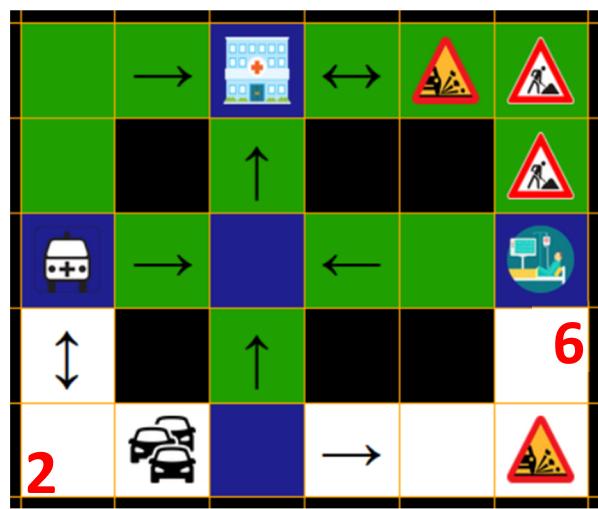
Εικόνα 31

$$\Delta c7_{max} = 2$$

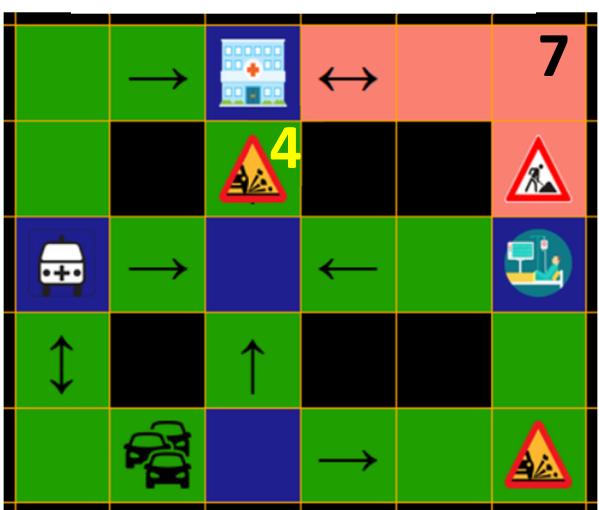


Εικόνα 32

$$\Delta c7 = 3$$

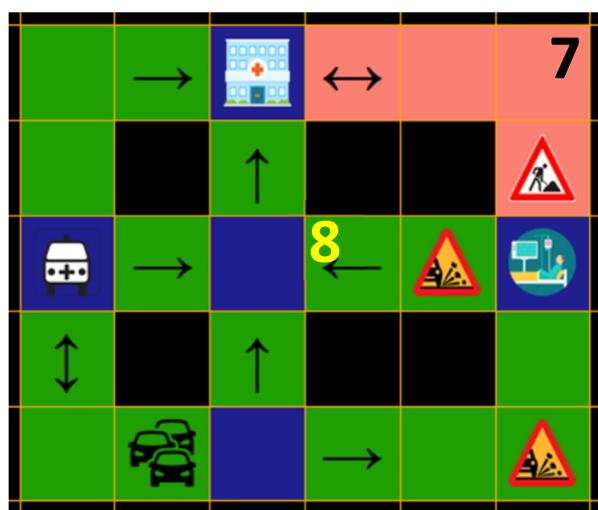


$$\Delta c4 = \Delta c4_{max} + 1 = 2$$



Εικόνα 33

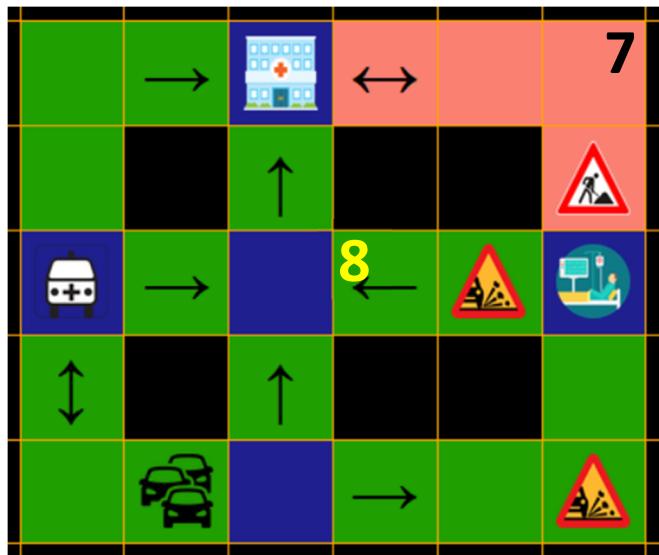
$$\Delta c8 = \Delta c8_{max} + 1 = 2$$



Εφαρμόζοντας ανάλυση ευαισθησίας, όπως παρουσιάζεται στον χώρο πληροφοριών της εικόνας 30, συμπεραίνουμε ότι οι μέγιστες αυξήσεις που μπορούν να ανεχτούν οι ακμές της άσπρης διαδρομής, χωρίς η τελευταία να αλλάξει, είναι $\Delta c1_{max} = 0$ για την ακμή 1 και $\Delta c7_{max} = 2$ για την ακμή 7. Πράγματι, για $\Delta c1 \geq 1$ η άσπρη διαδρομή μετατρέπεται στην {3, 4, 7} (εικόνα 31) και για $\Delta c7 \geq 3$ η άσπρη διαδρομή μετατρέπεται στην {2, 6} (εικόνα 32). Παρομοίως, οι μέγιστες αυξήσεις που μπορούν να ανεχτούν οι ακμές της ροζ διαδρομής, χωρίς αυτή να αλλάξει, είναι $\Delta c4_{max} = 1$ για την ακμή 4 και $\Delta c8_{max} = 1$ για την ακμή 8. Πράγματι, για $\Delta c4 \geq 2$, καθώς και για $\Delta c8 \geq 2$, η ροζ διαδρομή μετατρέπεται στην {7} (εικόνα 33).

Ας επιχειρήσουμε επιπλέον να εφαρμόσουμε ανάλυση ευαισθησίας στον γράφο της εικόνας 34, για να δούμε την περίπτωση μείωσης του κόστους ακμής που βρίσκεται εκτός της συντομότερης διαδρομής. Πρόκειται για τον ίδιο γράφο της εικόνας 30, όπου έχει τοποθετηθεί

η χρονική καθυστέρηση “κακοτράχαλη οδός” στην ακμή 8, με αποτέλεσμα (όπως είδαμε στην εικόνα 33) η ροζ διαδρομή να έχει γίνει η {7}. Βάσει των συμπερασμάτων της ανάλυσης ευαισθησίας που προβάλλονται στην εικόνα 34, μπορούμε να μειώσουμε το σχετικό χρονικό βάρος της ακμής 8 το πολύ κατά $\Delta c_{8max} = 1$ πριν αλλάξει η ροζ διαδρομή και επανέλθει στην {8, 4}. Πράγματι, αυτό επιβεβαιώνεται στην εικόνα 35 με τη μείωση $\Delta c_8 = 2$ (εδώ πρόκειται και για τη μέγιστη δυνατή μείωση).



Εικόνα 34

Display information:

If the 'Edges on' or decrease any cost of the 'Edges off' by just 1, the shortest paths are going to change (and their total costs will be less than the total costs of the previous shortest paths).

1) For shortest path 1:

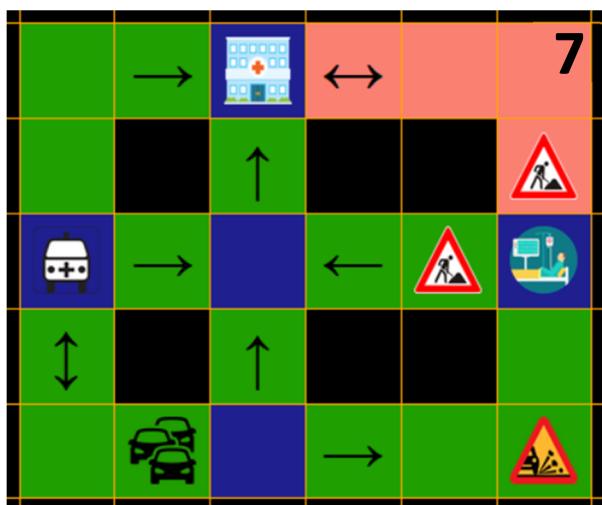
- Edges on: 1 → 4 (+0), 7 → 8 (+2)
- Edges off:

2) For shortest path 2:

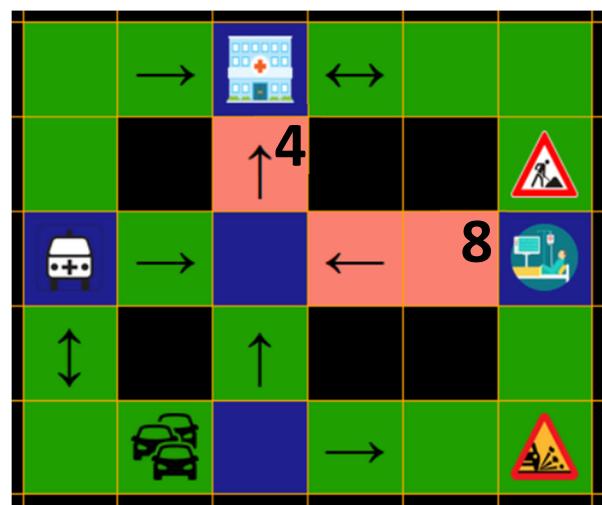
- Edges on: 7 → 7 (+1)
- Edges off: 8 → 4 (-1)

$$\Delta c_{8max} = 1 \text{ (μείωση)}$$

$$\Delta c_8 = 2 \text{ (μείωση)}$$



Εικόνα 35



Επίλυση του shortest path με τον αλγόριθμο αναζήτησης A* (A star):

Όπως προαναφέρθηκε, σε μεγάλες εφαρμογές, η επίλυση των προβλημάτων εύρεσης της συντομότερης διαδρομής γίνεται με αλγορίθμικές διαδικασίες. Αυτό επιβάλλεται διότι η μοντελοποίηση με γραμμικό/ακέραιο προγραμματισμό αποδεικνύεται εξαιρετικά απαιτητική σε θέματα μνήμης και χρόνου. Στη συγκεκριμένη ενότητα θα αναφερθούν πολύ συνοπτικά τα κύρια χαρακτηριστικά του αλγορίθμου αναζήτησης A* (A star) και θα γίνει επιπλέον μικρή αναφορά σε μια ειδική περίπτωση αυτού, τον αλγόριθμο του Dijkstra. Δείχνω επίσης πώς υλοποιώ στο πρόγραμμά μου τον αλγόριθμο A*. Επιλέγεται αυτός έναντι του Dijkstra, καθώς ταιριάζει καλύτερα στο σενάριό μου, όπου αναζητώ τη συντομότερη διαδρομή από έναν κόμβο αφετηρίας προς μονάχα έναν άλλον κόμβο προορισμού (single – pair shortest path), ενώ ο Dijkstra βρίσκει τις διαδρομές ελαχίστου κόστους από τον αρχικό κόμβο προς όλους τους υπόλοιπους (single – source shortest path), κάτι που δεν με ενδιαφέρει εδώ. Σε κάθε περίπτωση, οι δύο αυτοί αλγόριθμοι είναι πλήρεις (complete), δηλαδή εγγυώνται την εύρεση λύσης, αν αυτή υπάρχει.

Ο αλγόριθμος A* έχει ευρύ πεδίο εφαρμογών, από τις μηχανές αναζήτησης και τους οδικούς χάρτες, μέχρι τα δίκτυα υπολογιστών και τα βιντεοπαιχνίδια, και στοχεύει στην ανακάλυψη της συντομότερης διαδρομής με τη μέγιστη δυνατή αποδοτικότητα. Είναι ένας ευρετικός, επαναληπτικός αλγόριθμος αναζήτησης. Σε κάθε επανάληψή του υπολογίζει το κόστος μετακίνησης $f(n)$ από την κορυφή αφετηρίας προς οποιαδήποτε άλλη γειτονική κορυφή n σύμφωνα με τον παρακάτω τύπο, και επεκτείνει τη διαδρομή με το μικρότερο κόστος $f(n)$.

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

Όπου η $g(n)$ δίνει το πραγματικό κόστος μετακίνησης από την κορυφή αφετηρίας p προς την κορυφή n και η $h(n)$ είναι η ευρετική συνάρτηση που εκτιμά το κόστος μετακίνησης από την κορυφή n προς την κορυφή προορισμού q . Για να είναι αποδεκτή αυτή η ευρετική συνάρτηση, θα πρέπει να μην υπερεκτιμά ποτέ το πραγματικό κόστος που προσπαθεί να προσεγγίσει. Συνήθως, γίνεται η επιλογή της ευκλείδειας απόστασης ως ευρετική συνάρτηση κόστους $h(n)$, αλλά στην περίπτωσή μας καταλληλότερη επιλογή είναι η μετρική Manhattan, δηλαδή:

$$h(n) = d_{Manh}(n, q) = \sum_{i=1}^2 |n_i - q_i|$$

Η απόφαση αυτή έγκειται στο ότι η μικρότερη χωρική (και χρονική) απόσταση μεταξύ δύο κορυφών στον γράφο μας είναι πράγματι η απόσταση Manhattan εξαιτίας του ρυμοτομικού σχεδίου της πόλης. Έτσι, εξασφαλίζεται πάντα η εξής απαραίτητη συνθήκη, όπου το πραγματικό κόστος μπορεί να είναι είτε χωρικό είτε χρονικό (άρα με την $h(n)$ που ορίστηκε παραπάνω επιλύεται το πρόβλημα και με τα δύο κριτήρια ελαχιστοποίησης):

$$\frac{\text{πραγματικό}}{\text{κόστος}}(n, q) \geq h(n)$$

Η υλοποίηση του αλγορίθμου A star στο πρόγραμμα γίνεται με τον παρακάτω κώδικα (εικόνα 36), που βρίσκεται στο αρχείο “networks_algorithms.py” και προέκυψε με μεταφορά του ψευδοκώδικα της Wikipedia [10] σε γλώσσα python.

```

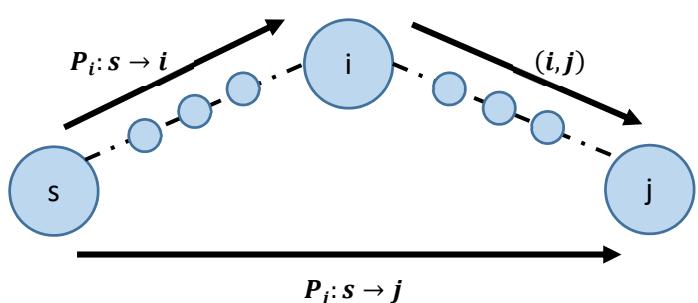
93 # the implementation of the A* algorithm
94 def A_star(adjacency_list, costs_list, start_node, goal_node):
95     open_list = [start_node] # start node first in the open list
96     parents = {} # it contains the parent nodes for each node
97     parents[start_node] = start_node # the start node has itself as parent
98     g_scores = {node: float('inf') for node in adjacency_list} # g-score (cost from start node to current node)
99     g_scores[start_node] = 0
100    f_scores = {node: float('inf') for node in adjacency_list} # f-score (g-score + heuristic)
101    f_scores[start_node] = heuristic_function(start_node, goal_node) # set the f-score of the start node to the heuristic function value
102    while len(open_list) != 0: # while the open list is not empty
103        for node in open_list: # go through all the nodes in the open list
104            f_scores[node] = g_scores[node] + heuristic_function(node, goal_node) # calculate the f-score for each node in the open list
105            current_node = min(open_list, key = lambda x: f_scores[x]) # get the node with the minimum f-score
106            if current_node == goal_node: # if goal node is reached, return the path
107                path = [] # list of nodes in the path
108                while parents[current_node] != current_node: # go through the parents of the nodes in the path
109                    path.append(current_node) # add the current node to the path
110                    current_node = parents[current_node] # go to the parent of the current node
111                path.append(start_node) # add the start node to the path
112                path.reverse() # reverse the path
113                return path # return the path
114            open_list.remove(current_node) # remove the current node from the open list
115            for neighbor_node in adjacency_list[current_node]: # go through the neighbors of the current node
116                nodes_pair = tuple(sort_vertices_by_manhattan_distance([current_node, neighbor_node], (0, 0)))
117                tentative_gScore = g_scores[current_node] + costs_list[nodes_pair] # calculate the tentative g-score
118                # if tentative_gScore is more than the g-score of the neighbor node
119                if tentative_gScore < g_scores[neighbor_node]:
120                    parents[neighbor_node] = current_node # set the parent of the neighbor node to the current node
121                    g_scores[neighbor_node] = tentative_gScore # set the g-score of the neighbor node to the calculated tentative_gScore
122                    # set the f-score of the neighbor node to the calculated f-score
123                    f_scores[neighbor_node] = g_scores[neighbor_node] + heuristic_function(neighbor_node, goal_node)
124                    if neighbor_node not in open_list: # if the neighbor node is not in the open list
125                        open_list.append(neighbor_node) # add the neighbor node to the open list
126    return None # if the goal node is not reached, return None

```

Εικόνα 36

Ο αλγόριθμος Dijkstra βρίσκει χώρο σε πολλές εφαρμογές, με τη διασημότερη ίσως συνεισφορά του να εντοπίζεται στην αποδοτική, βέλτιστη δρομολόγηση IP πακέτων (εντοπίζει τη συντομότερη διαδρομή μεταξύ δύο συσκευών σε δίκτυα υπολογιστών). Πρόκειται για έναν άπληστο (greedy) αλγόριθμο αναζήτησης σε γράφο. Σε αντίθεση με τον A*, ο αλγόριθμος Dijkstra διαθέτει μονάχα μία συνάρτηση κόστους, τη $g(n)$, που αποτιμά το πραγματικό κόστος από την κορυφή αφετηρίας p προς οποιαδήποτε κορυφή n , δηλαδή: $f(n) = g(n)$. Η κατασκευή και των δύο αλγορίθμων (A star και Dijkstra) βασίζεται στη μαθηματική αρχή του Bellman και στις εξισώσεις που προκύπτουν από αυτήν. Η αρχή του Bellman διατυπώνεται με τον εξής τρόπο [8] και μια σχεδιαστική ερμηνεία αυτής παρατίθεται στην εικόνα 37.

Αν $P_j: s \rightarrow j$ είναι η βραχύτερη διαδρομή από την κορυφή s έως την κορυφή j του γράφου $G(V, E)$ και (i, j) είναι η τελευταία ακμή της P_j , τότε η διαδρομή $P_i: s \rightarrow i$ η οποία προκύπτει αν αφαιρέσουμε την ακμή (i, j) από τη διαδρομή P_j , θα είναι η βραχύτερη διαδρομή από την κορυφή s έως την κορυφή i του γράφου.



Εικόνα 37

Παραπομπές – Βιβλιογραφία:

1. https://en.wikipedia.org/wiki/Shortest_path_problem#Linear_programming_formulation
(το κύριο άρθρο της Wikipedia για το πρόβλημα εύρεσης της συντομότερης διαδρομής)
2. https://www.math.uni-bielefeld.de/documenta/vol-ismp/32_schrijver-alexander-sp.pdf
(μελέτη της ιστορίας και της εξέλιξης του προβλήματος shortest path από τον Alexander Schrijver)
3. “Linear Programming and Algorithms for Communication Networks” του Eiji Oki, κυρίως οι σελίδες 31 – 43 (περιέχουν τη μοντελοποίηση με γραμμικό προγραμματισμό του προβλήματος εύρεσης της βραχύτερης διαδρομής σε κατευθυνόμενο γράφο με βάρη στις σελίδες 31 – 40 και την περιγραφή του αλγορίθμου Dijkstra στις σελίδες 40 – 43)
4. <https://optimization-online.org/wp-content/uploads/2014/09/4560.pdf>
(paper του Leonardo Taccari, όπου στη σελίδα 3 βρίσκεται η βασικότερη μοντελοποίηση του προβλήματος shortest path)
5. https://en.wikipedia.org/wiki/Adjacency_matrix
(το κύριο άρθρο της Wikipedia για τον πίνακα γειτνίασης)
6. <https://www.programiz.com/dsa/graph-adjacency-list>
(περιέχει πληροφορίες σχετικές με τη λίστα γειτνίασης και τους τρόπους υλοποίησής της με κώδικα)
7. <https://pymprog.sourceforge.net/index.html>
(το documentation του πακέτου pymprog)
8. “Ανώτερα μαθηματικά για μηχανικούς” του Erwin Kreyszig, κεφάλαιο 23 (Γράφοι – Συνδυαστική Βελτιστοποίηση), κυρίως οι σελίδες 740 – 746 (γενική εξέταση των προβλημάτων βραχύτερης διαδρομής στις σελίδες 740 – 744 και του αλγορίθμου Dijkstra στις σελίδες 744 – 746)
9. https://en.wikipedia.org/wiki/Depth-first_search
(το κύριο άρθρο της Wikipedia για τον αλγόριθμο depth first search, που παρέχει και ενδεικτικό ψευδοκώδικα)
10. https://en.wikipedia.org/wiki/A*_search_algorithm
(το κύριο άρθρο της Wikipedia για τον αλγόριθμο A*, που παρέχει και ενδεικτικό ψευδοκώδικα)

Κώδικας και οδηγίες εκτέλεσής του:

Λόγω της μεγάλης έκτασης του κώδικα δεν είναι εύκολο να παρατεθεί ολόκληρος εδώ. Όλος ο python κώδικας είναι αποθηκευμένος στον φάκελο “code”. Για να τρέξει το πρόγραμμα απαιτείται η εκτέλεση μονάχα του python αρχείου “networks.py”. Χρειάζονται οι βιβλιοθήκες “tkinter”, “pillow”, “random”, “time”, “os”, “numpy” και “pymprog”.