1ο Εξαμηνιαίο Project στην Τεχνητή Νοημοσύνη 7ου Εξαμήνου

Συμμετέχοντες:
- Αργυρίου Στέφανος Παναγιώτης, 03112006
- Κάτσιος Αθανάσιος, 03112151
- Κονιδάρης Φίλιππος, 03112011

Το συγκεκριμένο project αποτελείται από δύο κομμάτια, το κυρίως πρόγραμμα που υπολογίζει τα μονοπάτια των ρομπότ και ένα δεύτερο που δίνει μία γραφική προσομοίωση του μονοπατιού τους. Για να τρέξετε το πρόγραμμα θα χρειαστεί να κάνετε unzip τα περιεχόμενα του φακέλου που θα υποβληθεί στο mycourses και να ανοίξετε ένα terminal από το directory στο οποίο τα έχετε εξάγει. Για να γίνει το compilation των εκτελέσιμων είναι απαραίτητη η χρήση περιβάλλοντος linux που διαθέτει την βιβλιοθήκη sfml. Για να εγκαταστήσετε την sfml πληκτρολογήστε (για ubuntu ή debian):

```
sudo apt-get install libsfml-dev
```

Έπειτα μπορείτε να κάνετε compile τα εκτελέσιμα μέσω του Makefile πληκτρολογώντας την εντολή:

```
make
```

Για να τρέξετε τα προγράμματα (με γραφικό περιβάλλον) αρκεί να πληκτρολογήσετε στο terminal

```
./ui [όνομα αρχείου] [1|2]
```

(1, αν επιθυμείτε να τρέξετε το πρόγραμμα με υποεκτιμητή, 2, αν επιθυμείτε να το τρέξετε με υπερεκτιμητή). Αν στην παραπάνω εντολή παραλειφθεί το όνομα αρχείου, τότε πρέπει να παραλειφθεί και η επιλογή εκτιμητή, οπότε ως όνομα αρχείου λαμβάνεται το input.txt και ως εκτιμητής ένας υποεκτιμητής. Αν παραλειφθεί η επιλογή εκτιμητή, τότε, το πρόγραμμα τρέχει με υποεκτιμητή.

Για να τρέξετε μόνο το κομμάτι του ai χωρίς γραφικό περιβάλλον πληκτρολογήστε:

```
./ai [όνομα αρχείου] [1|2]
```

1 Περιγραφή του προβλήματος

Σκοπός του συγκεκριμένου project αποτελεί η υλοποίηση κατάλληλου προγράμματος που θα βασίζεται στις αρχές της τεχνητής νοημοσύνης, ικανού να υπολογίζει τη σειρά βημάτων που θα πρέπει να ακολουθήσουν δύο ρομπότ στην προσπάθεια περιήγησής τους σε ένα δισδιάστατο χώρο. Κάθε ρομπότ οφείλει να περάσει με όσο το δυνατόν πιο βέλτιστο τρόπο από μία σειρά σημείων του επιπέδου και κατόπιν να συναντήσει το άλλο σε ένα προκαθορισμένο σημείο συνάντησης. Μετά τη συνάντηση, τα δύο ρομπότ επιστρέφουν στην αρχική τους θέση, οπότε και ολοκληρώνεται η εκτέλεση του προγράμματος. Για την υλοποίηση θεωρούμε ότι τα ενδιάμεσα σημεία από τα οποία οφείλουν να περάσουν τα δύο ρομπότ δίνονται τυχαία και επομένως είναι στη δική μας κρίση η υλοποίηση κατάλληλης συνάρτησης που θα επιλέγει κάθε φορά το πιο συμφέρον μονοπάτι για τα ρομπότ.

2. Σκιαγράφηση της λύσης και βασικές συνιστώσες του προγράμματος

Στην υλοποίηση του προγράμματος κεντρική θέση κατέχει ο ευριστικός αλγόριθμος A*. Βασική ιδέα του προγράμματός μας αποτελεί η επαναλαμβανόμενη χρήση του αλγορίθμου προκειμένου να προσδιορίζουμε κάθε φορά το μικρότερο δυνατό μονοπάτι προς τον εκάστοτε στόχο. Πιο συγκεκριμένα, δοθέντων των αρχικών θέσεων των ρομπότ, των n ενδιαμέσων σημείων και του σημείου συνάντησης, για το πρώτο ρομπότ:

α. Κάνουμε n A* και κρατάμε το μονοπάτι προς το πιο κοντινό ενδιάμεσο σημείο.

β. Ξανακάνουμε A* για τα εναπομείνοντα n-1 ενδιάμεσα σημεία και κρατάμε το μονοπάτι προς το κοντινότερο.

γ. Επαναλαμβάνουμε τη διαδικασία μέχρι να έχει μείνει μόνο ένα ενδιάμεσο σημείο για το οποίο κάνουμε έναν ακόμη A*.

δ. Στο τέλος, κάνουμε έναν τελευταίο A* προς το σημείο συνάντησης και συνδέουμε όλα τα προκύπτοντα μονοπάτια για να δημιουργήσουμε το σχέδιο συνάντησης του πρώτου ρομπότ.
ε. Ξανακάνουμε την ίδια διαδικασία για το ρομπότ 2 και δημιουργούμε και το δικό του σχέδιο συνάντησης.
στ. Σε αυτό το σημείο, προτού προβούμε στον υπολογισμό των μονοπατιών επιστροφής, ελέγχουμε τα δύο σχέδια συνάντησης για συγκρούσεις (collisions) και τροποποιούμε το σχέδιο του ρομπότ 2 αναλόγως.
ζ. Με δύο ακόμα A* αυτή τη φορά προς τις αρχικές θέσεις του κάθε ρομπότ, σχεδιάζουμε τα μονοπάτια επιστροφής, τα οποία επίσης ελέγχουμε για συγκρούσεις.
η. Στο τέλος των παραπάνω βημάτων έχει προκύψει το ζητούμενο μονοπάτι για κάθε ρομπότ.

Παράλληλα με την υλοποίηση του παραπάνω αλγορίθμου που αποτελεί το κύριο μέρος της παρούσας εργασίας (AI component), υλοποιήθηκε ξεχωριστά και ένα γραφικό κομμάτι, το οποίο επιτρέπει στο χρήστη του προγράμματος να παρακολουθήσει γραφικά την κίνηση των ρομπότ σε ένα grid που αποτελεί προσομοίωση του δισδιάστατου χώρου που δίνεται ως είσοδος στο πρόγραμμα (UI component). Τόσο για την υλοποίηση του AI Component, όσο και για το UI επιλέξαμε τη γλώσσα C++ λόγω των αντικειμενοστραφών δυνατοτήτων της και της εξοικείωσης που διαθέταμε με τις αλγοριθμικές της δομές.

3. Περιγραφή AI Component:

3.1. Μορφή αρχείου εισόδου και έξοδος του προγράμματος

Το πρόγραμμά μας δέχεται ως είσοδο ένα .txt αρχείο με όνομα *“input.txt”* στο οποίο είναι σημειωμένα σε ξεχωριστές γραμμές το μέγεθος του χώρου (μήκος και πλάτος), οι αρχικές συντεταγμένες των δύο ρομπότ, ο αριθμός των ενδιαμέσων σημείων, όπως και οι συντεταγμένες τους. Επίσης, δίνεται λεπτομερώς η κάτοψη του χώρου, στην οποία σημειώνονται τα εμπόδια (τοίχοι/walls), πάνω στα οποία δεν μπορούν να πατήσουν τα δύο ρομπότ. Για τον έλεγχο της ορθότητας του προγράμματος χρησιμοποιήθηκαν randomly generated αρχεία εισόδου, τα οποία δημιουργήθηκαν μέσω του προγράμματος gen (πηγαίος κώδικας gen.cpp). Το συγκεκριμένο πρόγραμμα λαμβάνει ως παραμέτρους δύο μεταβλητές map_width, map_height, που δηλώνουν τις διαστάσεις του χώρου, μία μεταβλητή checkpoint_count που δηλώνει τον αριθμό των ενδιαμέσων σημείων που θα γίνουν generate, και μία μεταβλητή wall_ratio, που δηλώνει το λόγο των εμποδίων προς τον αριθμό των ελεύθερων τετραγώνων στο χάρτη. Αν κάποια από τις παραπάνω παραλειφθεί αποδίδονται προκαθορισμένες τιμές. Για πλήρη επεξήγηση του τρόπου χρήσης της gen, πληκτρολογήστε:

```
./gen -h
```

Δοθέντων των παραπάνω τιμών, το πρόγραμμα gen αρχικά δημιουργεί τα εμπόδια, με την χρήση τυχαίων float αριθμών μεταξύ του 0 και του 1, και στη συνέχεια χρησιμοποιεί τη συνάρτηση rand() για να δημιουργήσει τυχαίες ακέραιες τιμές, οι οποίες εν συνεχεία εντοπίζονται στις δοθείσες διαστάσεις μέσω της πράξης του ακέραιου υπολοίπου (mod). Αν κάποιο από τα τυχαία σημεία που προκύπτουν είναι ήδη κατειλημμένο από άλλο αντικείμενο ή τοίχο, τότε οι συντεταγμένες του επαναυπολογίζονται μέχρι να υπολογιστεί κάποιο άδειο σημείο του grid.
Στη διάρκεια εκτέλεσης του κυρίως προγράμματος (AI Component), εμφανίζονται στην οθόνη όλες οι διαθέσιμες επιλογές για το κάθε ρομπότ, οι τελικές επιλογές που κάνει καθώς και οι κινήσεις που αναγκάζεται να απορρίψει λόγω συγκρούσεων με εμπόδια ή με το άλλο ρομπότ. Στο τέλος, δίνεται με τη μορφή ακολουθίας συντεταγμένων η διαδρομή του κάθε ρομπότ, ενώ αυτή η ακολουθία αποθηκεύεται και σε δύο .txt αρχεία με ονόματα

“robot1.txt” και *“robot2.txt”.* Αυτά τα δύο αρχεία λαμβάνουν το ρόλο της εισόδου για το γραφικό περιβάλλον (UI Component) προκειμένου να μπορέσουν να εμφανιστούν οι κινήσεις στο προσωμειωμένο πλέγμα.

3.2. Προγραμματιστική υλοποίηση του αλγορίθμου A*

Η υλοποίηση του αλγορίθμου A* έγινε μέσω της συνάρτησης *astar* η οποία δέχεται ως είσοδο μία αρχική θέση, μία τελική θέση και την κάτοψη του χώρου και επιστρέφει μία λίστα από συντεταγμένες που αντιπροσωπεύουν το κοντινότερο δυνατό μονοπάτι από την αρχική στην τελική θέση. Πηγή για τον ίδιο τον αλγόριθμο αποτέλεσε η δεύτερη αμερικάνικη έκδοση του βιβλίου *“Τεχνητή Νοημοσύνη: Μια σύγχρονη προσέγγιση”* των Russell και Norvig.

Με βάση τη συγκεκριμένη μορφή του αλγορίθμου χρησιμοποιήσαμε:

α. Για τη δημιουργία του χώρου καταστάσεων ένα τετραδικό δέντρο – μη πλήρες στη γενική περίπτωση, το οποίο αποτελείται από κόμβους της παρακάτω μορφής:

```
struct node
{
	struct node* parent;
	int row;
	int col;
	int g;
	int count;
	struct node* north;
	struct node* east;
	struct node* south;
	struct node* west;
};
```

Κάθε κόμβος (node) του δέντρου διαθέτει ένα γονιό κόμβο (parent node), τις συντεταγμένες του σημείου του χώρου στις οποίες αντιστοιχεί (col, row), την ακέραια απόστασή του (manhattan distance) από την αρχική θέση της ρίζας του δέντρου (g), τον αριθμό της κίνησης στην οποία δημιουργήθηκε (count), καθώς και δείκτες στα τέσσερα δυνητικά του παιδιά.

β. Ένα δισδιάστατο boolean πίνακα `closed`, ο οποίος χρησιμοποιείται για την καταγραφή των σημείων του χώρου που έχουμε αναλύσει ήδη. Με 1 σημειώνονται τα σημεία που έχουμε επισκεφτεί ήδη, ενώ με 0 όσα δεν έχουμε επισκεφτεί ακόμα.

γ. Ένα δυαδικό δέντρο open, στο οποίο εισάγουμε τα στοιχεία που θα αναλυθουν στα επομενα βηματα. Τα εισαχθέντα στοιχεία ταξινομούνται σύμφωνα με το κόστος μετάβασής τους, και έτσι μπορούμε εύκολα να αντλήσουμε από το open το πρώτο στοιχείο, δηλαδή αυτό με το μικρότερο κόστος. Για την υλοποίηση της συγκεκριμένης δομής χρησιμοποιούμε την κλάση βιβλιοθήκης της c++ std::multimap.

δ. Ένα δισδιάστατο ακέραιο πίνακα `cost`, ο οποίος δίνει μία εκτίμηση για την απόσταση του κάθε σημείου του χώρου από το σημείο στόχο. Πριν από κάθε κλήση της `astar`, δημιουργούμε με βάση το σημείο-στόχο τον αντίστοιχο πίνακα `cost`. Αν το σημείο είναι εμπόδιο δίνουμε στο αντίστοιχο κελί του πίνακα μία πολύ μεγάλη τιμή, ώστε ο *astar* να το απορρίπτει εύκολα. Σε κάθε άλλη περίπτωση, συμπληρώνουμε το κελί με βάση τον εκτιμητή μας.

```
//creates cost
int** cost = new int*[gridl+1];
for(j = 0; j < gridl+1; j++)
{
```

```
    cost[j] = new int[gridc+1];
}

for(j = 0; j < gridl+1; j++)
{
    for(k = 0; k < gridc+1; k++)
    if (grid[j][k] == 'X') cost[j][k] = 100000;
    else cost[j][k] = abs(i - gx) + abs(j – gy);
}
```

Όσον αφορά την ίδια τη λειτουργία της συνάρτησης, αρχικά προσθέτουμε τη θέση εκκίνησης σε έναν κόμβο, ο οποίος θα αποτελέσει τη ρίζα του δέντρου μας και προσθέτουμε τον κόμβο αυτό στο open. Κατόπιν, δημιουργούμε τα 4 παιδιά του αρχικού κόμβου που αντιπροσωπεύουν τις 4 κατευθύνσεις στις οποίες μπορεί να κινηθεί το κάθε ρομπότ και τοποθετούμε τη ρίζα του δέντρου στο `closed` για να σηματοδοτήσουμε ότι έχουμε ήδη επισκεφτεί την αρχική θέση. Τέλος, τα τέσσερα παιδιά, εισάγονται στην open κατά τέτοιο τρόπο, ώστε αυτό που απέχει λιγότερο από το στόχο να τοποθετηθεί πρώτα. Αυτό αποτελεί και τον κόμβο που θα εξετάσουμε αμέσως μετά. Επαναληπτικά, δημιουργούμε παιδιά για κάθε κόμβο που βρίσκεται στην πρώτη θέση του open και τα τοποθετούμε στο open διατεταγμένα με βάση την εκτίμησή μας για την απόστασή τους από το στόχο. Αν κάποιο παιδί βρίσκεται πάνω σε εμπόδιο ή αν κάποιο παιδί βρίσκεται ήδη στον πίνακα `closed`, το παραλείπουμε από εισαγωγή στο open.
Είναι αποδεδειγμένο ότι ο αλγόριθμος A*, εφόσον υπάρχει διαθέσιμο μονοπάτι μεταξύ δύο σημείων θα το εντοπίσει. Πράγματι, εξερευνώντας διαδοχικά τα παιδιά των πιο “οικονομικότερων” σημείων, κάποια στιγμή θα φτάσουμε στο σημείο στόχο και η συνάρτηση θα επιστρέψει τον κόμβο του δέντρου που αποτελεί το στόχο μας. Μέσω των διαδοχικών γονέων μπορούμε εύκολα να προσδιορίσουμε το οικονομικότερο μονοπάτι σε μορφή συνδεδεμένης λίστας. Αν εξεταστούν όλα τα σημεία και δεν έχει βρεθεί κάποιο μονοπάτι (ισοδύναμα, όταν το open αδειάσει), η συνάρτηση επιστρέφει NULL γεγονός το οποίο δηλώνει την αδυναμία εύρεσης μονοπατιού.
Στη γενική περίπτωση, ο αλγόριθμος A* με τον τρόπο που τον έχουμε υλοποιήσει έχει πολυπλοκότητα 4^{n*m}, όπου n και m οι διαστάσεις του χώρου, ενώ το 4 σηματοδοτεί το branching factor του χώρου καταστάσεων. Παρόλα αυτά, η επιλογή κατάλληλων δομών μπορεί να περιορίσει σε σημαντικό βαθμό τον πραγματικό χρόνο στον οποίο τρέχει η συνάρτηση:
α. Όπως αναφέρθηκε παραπάνω, η δομή `closed` χρησιμοποιείται για τον έλεγχο επανάληψης κάποιου παιδιού (για να ελέγχουμε μήπως έχουμε ήδη επισκεφτεί κάποιο σημείο του χώρου). Σε αυτά τα πλαίσια, επιλέγεται ένας δισδιάστατος boolean πίνακας χάρη στον οποίο ο έλεγχος, αλλά και η εισαγωγή νέων στοιχείων μπορεί να γίνει σε χρόνο O(1).
β. Στη δομή `open` προσθέτουμε κάθε φορά με Insertion Sort τα νέα παιδιά εμπλουτίζοντας το χώρο καταστάσεων. Για να διευκολυνθεί η εισαγωγή νέων στοιχείων επιλέγουμε να υλοποιήσουμε την open μέσω ενός δυαδικού δέντρου. Με αυτό τον τρόπο, η εισαγωγή ενός νέου στοιχείου χρειάζεται χρόνο O(logn).

3.3. Επιλογή σειράς ενδιάμεσων σημείων – η συνάρτηση `shortest`

Ήδη από την αρχική περιγραφή του προβλήματος έχουμε αποδεχτεί ότι τα ενδιάμεσα σημεία δίνονται στο αρχείο εισόδου με τυχαίο τρόπο. Επομένως, είναι ευθύνη μας να υλοποιήσουμε μία συνάρτηση που θα μας παρέχει το κατά το δυνατόν (από άποψη υπολογιστικού χρόνου και μνήμης) συντομότερο μονοπάτι που χρειάζεται να διανύσει το κάθε ρομπότ ξεχωριστά.
Μετά από παρατήρηση του προβλήματος είναι εύκολο να διαπιστώσει κανείς ότι πρόκειται

για TSP problem (travelling salesman problem – το πρόβλημα του πλανόδιου πωλητή). Ωστόσο, με δεδομένη την εκθετική πολυπλοκότητα του αλγορίθμου A* θα ήταν πολύ δύσκολο (αν όχι αδύνατο) να υλοποιήσουμε ένα πρόγραμμα αρκετά αποδοτικό ώστε να μας παρέχει τη βέλτιστη διαδρομή στο σύνολο των περιπτώσεων αν η είσοδος είναι αρκετά μεγάλη. Υπό αυτό το πρίσμα, αναζητήσαμε εναλλακτικές μεθόδους προσέγγισης του προβλήματος και καταλήξαμε στο ότι μία καλή εναλλακτική του TSP algorithm θα ήταν κάθε φορά να κρίνουμε τις διαθέσιμες επιλογές μετακίνησης και αναλόγως να επισκεπτόμαστε την πιο συμφέρουσα. Δηλαδή:

1) Από τα n ενδιάμεσα σημεία, κάνουμε n A* με αρχή την αρχική μας θέση και τέλος τα n ενδιάμεσα σημεία και κρατάμε τη διαδρομή προς το κοντινότερο.
2) Από τα n-1 εναπομείναντα σημεία, ξανακάνουμε A* με αρχή το ενδιάμεσο σημείο στο οποίο αποφασίσαμε να πάμε στο προηγούμενο βήμα και τέλος τα n-1 εναπομείναντα σημεία και επιστρέφουμε και πάλι την πιο συμφέρουσα διαδρομή.
3) Επαναλαμβάνουμε την παραπάνω διαδικασία έως ότου μείνει ένα τελευταίο ενδιάμεσο σημείο, οπότε κάνουμε και έναν ακόμη A*.

Ο παραπάνω αλγόριθμος παρέχει λύσεις οι οποίες στην πλειονότητά τους είναι ίδιες με τον αλγόριθμο του πλανόδιου πωλητή. Ωστόσο, σε περιπτώσεις πολλών ενδιαμέσων σημείων (80+), ο χρόνος που απαιτούνταν για τον υπολογισμό του μονοπατιού ήταν εξαιρετικά μεγάλος, καθώς σε αυτό τον αλγόριθμο απαιτούνται συνολικά $\sum_{k=0}^{n}(n-k) = \frac{n^2}{2} + \frac{n}{2}$ A*.

Για να αποφύγουμε την τεράστια αυτή απαίτηση σε χρόνο, περιορίζουμε τον αριθμό των σημείων που κάνουμε A* κάθε φορά σε 5. Επιλέγουμε μάλιστα τα 5 τα οποία έχουν τη μικρότερη Manhattan Distance σε σχέση με την αρχική μας θέση. Κατά βήματα:

1) Κατατάσσουμε τα n ενδιάμεσα σημεία σύμφωνα με το Manhattan Distance τους από την αρχική θέση.
2) Κάνουμε A* για τα 5 πρώτα της κατάταξης και κρατάμε την πιο συμφέρουσα διαδρομή.
3) Κατατάσσουμε τα n-1 εναπομείνοντα ενδιάμεσα σημεία σύμφωνα με το Manhattan Distance τους από τη θέση στην οποία έχουμε βρεθεί τώρα.
4) Κάνουμε A* για τα 5 πρώτα της κατάταξης και κρατάμε την πιο συμφέρουσα διαδρομή.
5) Επαναλαμβάνουμε τη διαδικασία έως ότου μείνει 1 τελευταίο ενδιάμεσο σημείο, οπότε κάνουμε έναν ακόμη A*.

Ο παραπάνω αλγόριθμος υλοποιείται στη συνάρτηση `shortest,` η οποία συνολικά πραγματοποιεί 5n A*, όπου n ο αριθμός των ενδιαμέσων σημείων.

3.4. Εύρεση συγκρούσεων – η λειτουργία της συνάρτησης `coldet`

Το πρόγραμμα που υλοποιήθηκε στο πλαίσιο της άσκησης ξεκινάει με την εύρεση του πλάνου της διαδρομής του ρομπότ 1 και κατόπιν υπολογίζει το πλάνο της μετακίνησης του ρομπότ 2. Ωστόσο, στα δύο πλάνα μπορεί να υπάρχουν συγκρούσεις μεταξύ των δύο ρομπότ, το οποίο φυσικά θέλουμε να αποφύγουμε. Οι πιθανές συγκρούσεις μεταξύ των δύο ρομπότ κατατάσσονται σε δύο κατηγορίες:

α. Συγκρούσεις διεκδίκησης. Στις συγκρούσεις διεκδίκησης τα δύο ρομπότ θέλουν την ίδια στιγμή να βρεθούν στο ίδιο σημείο.

β. Συγκρούσεις εναλλαγής. Όταν τα ρομπότ θέλουν να ανταλλάξουν σημεία, κάτι το οποίο θα οδηγούσε στο να περάσει το ένα “μέσα” από το άλλο.

Για τις συγκρούσεις διεκδίκησης το ρομπότ 2 απλά περιμένει (κάνει stall) στο σημείο το οποίο βρίσκεται ώστε να πάει πρώτα εκεί το ρομπότ 1. Για τις συγκρούσεις εναλλαγής, το ρομπότ 2 ελέγχει από ποια κατεύθυνση έρχεται το ρομπότ 1 (βορράς-νότος ή ανατολή-δύση) και κάνει στο πλάι για να περάσει το ρομπότ 1 προς την κάθετη κατεύθυνση.

Για την αντιμετώπιση των παραπάνω συγκρούσεων υλοποιήσαμε τη συνάρτηση `coldet` η οποία δίνεται σε ψευδοκώδικα παρακάτω:

```
Όσο (ρομπότ 2 έχει ακόμα κινήσεις) επανέλαβε
        Έλεγξε κίνηση ρομπότ 1, κίνηση ρομπότ 2
        Εάν (σύγκρουση διεκδίκησης)
                ρομπότ 2 μείνε στη θέση σου
        Αλλιώς αν (σύγκρουση εναλλαγής)
                Αν (ρομπότ 1 έρχεται από τα βόρεια ή τα νότια)
                        Αν (σημείο στα ανατολικά δεν είναι εμπόδιο)
                                ρομπότ 2 πήγαινε ανατολικά και μετά πάλι δυτικά
                        Αλλιώς αν (σημείο στα δυτικά δεν είναι εμπόδιο)
                                ρομπότ 2 πήγαινε δυτικά και μετά πάλι ανατολικά
                        Αλλιώς αν (ρομπότ 1 έρχεται από τα βόρεια)
                                ρομπότ 2 πήγαινε νότια και μετά πάλι βόρεια
                        Αλλιώς
                                ρομπότ 2 πήγαινε βόρεια και μετά πάλι νότια
                Αλλιώς
                        Αν (σημεία στα βόρεια δεν είναι εμπόδιο)
                                ρομπότ 2 πήγαινε βόρεια και μετά πάλι νότια
                        Αλλιώς αν (σημεία στα νότια δεν είναι εμπόδιο)
                                ρομπότ 2 πήγαινε νότια και μετά πάλι βόρεια
                        Αλλιώς αν (ρομπότ 1 έρχεται από ανατολικά)
                                ρομπότ 2 πήγαινε δυτικά και μετά πάλι ανατολικά
                        Αλλιώς
                                ρομπότ 2 πήγαινε ανατολικά και μετά πάλι δυτικά
        Προετοίμασε το επόμενο ζευγάρι κινήσεων για έλεγχο
Τέλος Επανάληψης
```

Μετά από την εκτέλεση του παραπάνω ψευδοκώδικα, εξαλείφονται όλες οι συγκρούσεις μεταξύ των δύο ρομπότ.

4. Περιγραφή UI Component

Στη συγκεκριμένη εργασία υλοποιήσαμε και ένα γραφικό περιβάλλον (πηγαίο αρχείο ui.cpp) στο οποίο μπορούμε να παρατηρήσουμε την κίνηση των ρομπότ μέσα στο δοσμένο χάρτη. Το γραφικό περιβάλλον δημιουργήθηκε στην γλώσσα C++ με τη χρήση της βιβλιοθήκης γραφικών SFML 2.3 (http://www.sfml-dev.org/) .
Το πρόγραμμα ui, όταν εκτελεστεί, καλεί αυτόματα το πρόγραμμα υπολογισμού των συντομότερων μονοπατιών και στην συνέχεια δείχνει τις κινήσεις τους μέσα σε ένα πλέγμα όπου κάθε σημείο παριστάνεται από ένα τετράγωνο 32x32 pixels. Για ευκολότερη περιήγηση σε μεγάλους χάρτες, παρέχεται η δυνατότητα μετακίνησης του πεδίου οράσεως με τα βελάκια, η δυνατότητα μεγέθυνσης και σμίκρυνσής του με τα πλήκτρα [=] και [–] αντίστοιχα, καθώς και η δυνατότητα παρακολούθησης ενός από τους δύο χαρακτήρες με το πλήκτρο [F].

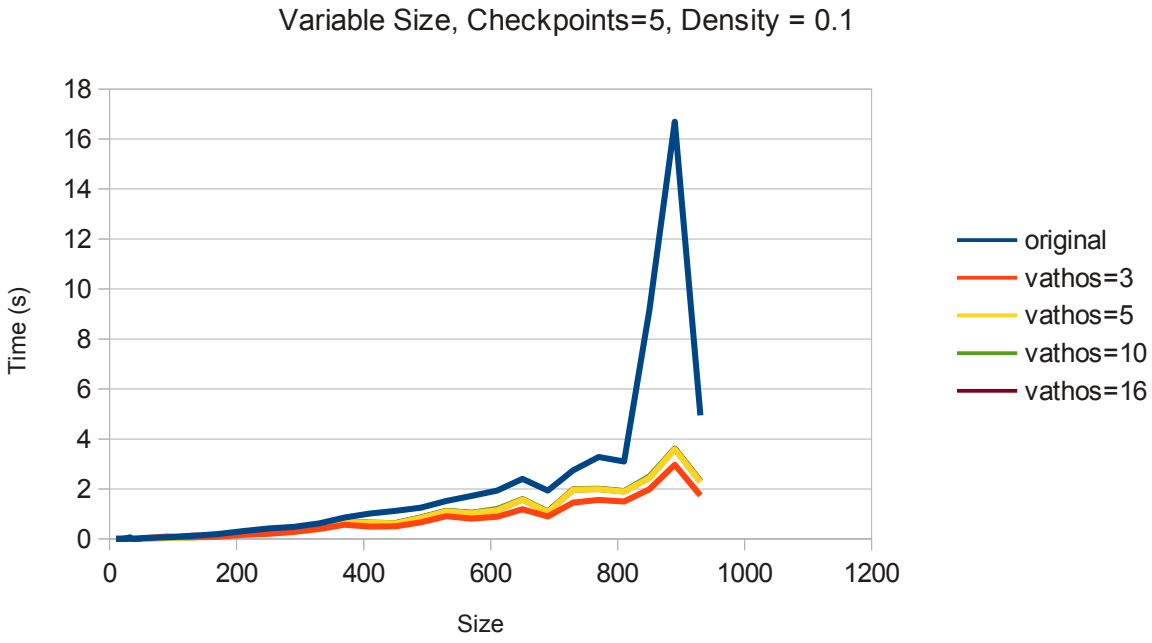
5. Καταγραφή και σύντομος σχολιασμός μετρήσεων

Η χρήση του random generator επέτρψε τη δημιουργία μίας πληθώρας χαρτών, οι οποίοι εν συνεχεία χρησιμοποιηθήκαν για την καταγραφή του χρόνου που απαιτούνταν για το τρέξιμο του προγράμματος. Μέχρι τώρα σκόπιμα δεν έχει αναφερθεί ο εκτιμητής που χρησιμοποιήθηκε στον A*. Και αυτό γιατί οι μετρήσεις για την ταχύτητα του προγράμματος πάρθηκαν αρχικά με έναν υποεκτιμητή, το Manhattan Distance και αργότερα με έναν υπερκτιμητή, το τετράγωνο του Manhattan Distance.
Επίσης, αναφέρθηκε ότι η συνάρτηση shortest, προκειμένου να αποφασίσει για το καλύτερο δυνατό μονοπάτι, έλεγχε τα 5 κοντινότερα (από άποψη Manhattan Distance) ενδιάμεσα σημεία και επέστρεφε το κοντινότερο μονοπάτι. Για τον εμπλουτισμό των

μετρήσεων, ο αριθμός σημείων που ελέγχονται (5) παραμετροποιήθηκε και αποδόθηκε στη μεταβλητή `vathos`. Όσο πιο μεγάλο είναι το `vathos`, τόσο πιο σχολαστικό “ψάξιμο” γίνεται από τη συνάρτηση `shortest`. Το original σηματοδοτεί το γεγονός ότι η `shortest` ελέγχει όλα τα εναπομείνοντα διαθέσιμα σημεία για να αποφασίσει για το κοντινότερο και όχι ένα μέρος τους.

5.1. Μετρήσεις με υποεκτιμητή

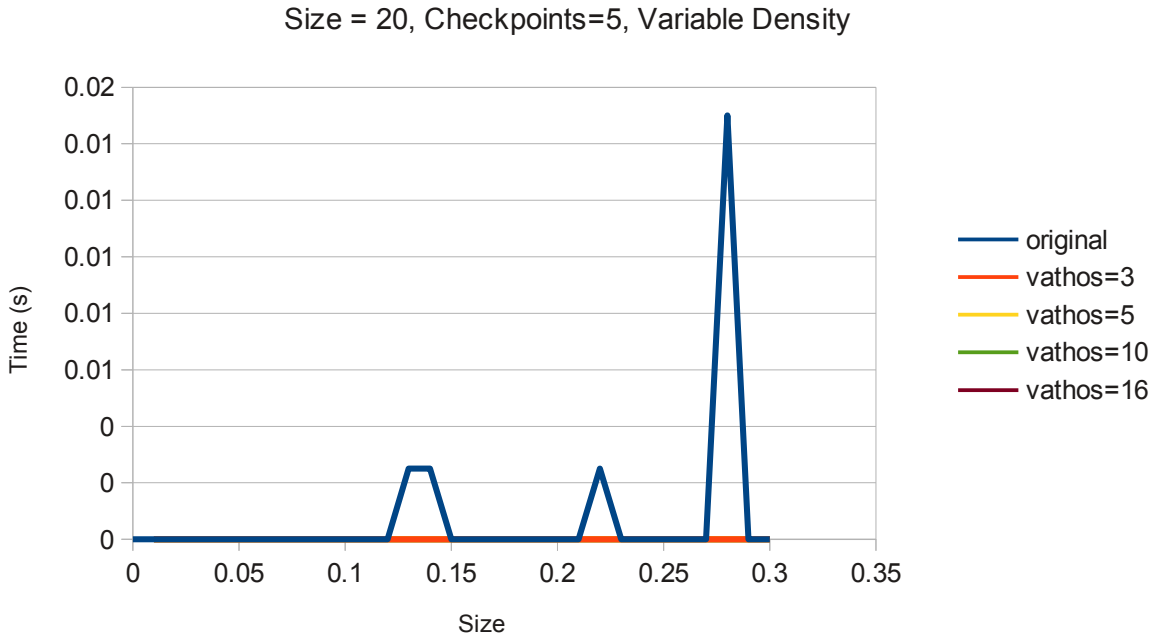
Στο παρακάτω διάγραμμα φαίνεται ο χρόνος που χρειάζεται για τον υπολογισμό των δύο μονοπατιών σε χάρτες μεταβλητού μεγέθους, με 5 ενδιάμεσα σημεία και ποσοστό εμποδίων 10%. Στον κάθετο άξονα βρίσκεται ο χρόνος σε seconds, ενώ στον οριζόντιο το μέγεθος του τετραγωνικού κάθε φορά grid.

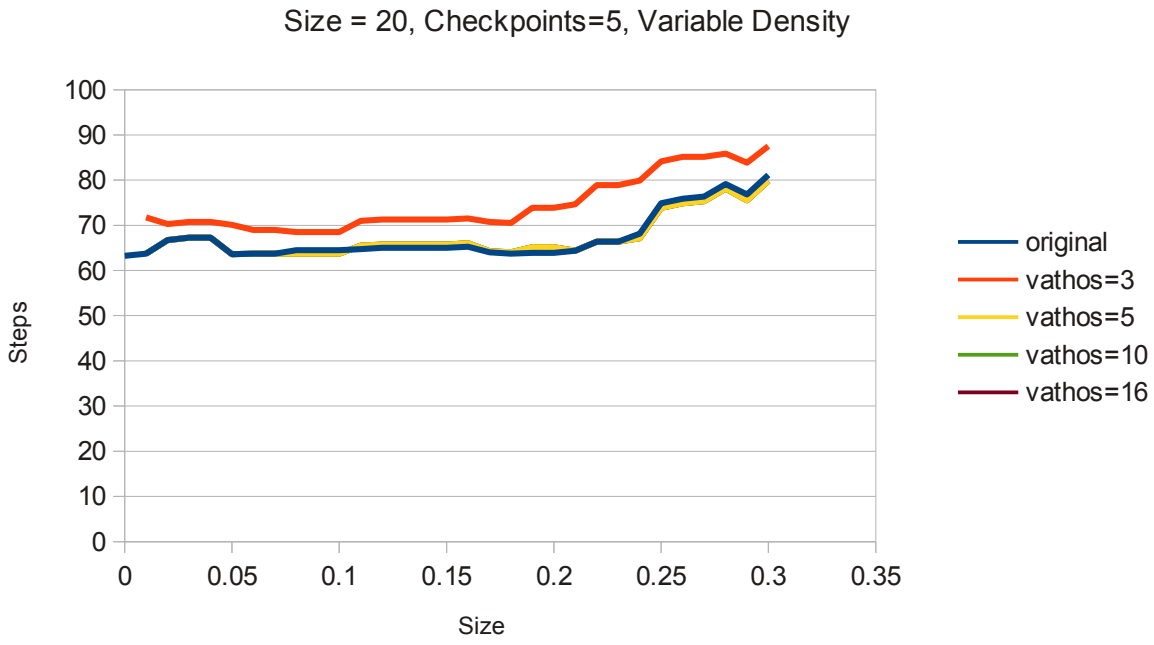


Στο επόμενο διάγραμμα έχουμε τους ίδιους πίνακες με παραπάνω, αλλά αυτή τη φορά μετράμε τον αριθμό βημάτων που επιστρέφει το πρόγραμμα για διάφορες τιμές του `vathos`.

Όπως γίνεται εύκολα φανερό, όταν το vathos είναι ίσο με 3 και λιγότερο όταν είναι ίσο με 5, σε ορισμένες περιπτώσεις δίνει περισσότερα βήματα, απ' ότι ο έλεγχος όλων των ενδιάμεσων σημείων. Παρόλα αυτά, για vathos = 10 και 16, τα αποτελέσματα ταυτίζονται στο 100% των περιπτώσεων, γεγονός το οποίο μας ωθεί να αναζητήσουμε κατάλληλο vathos μεταξύ των τιμών 5 και 10.

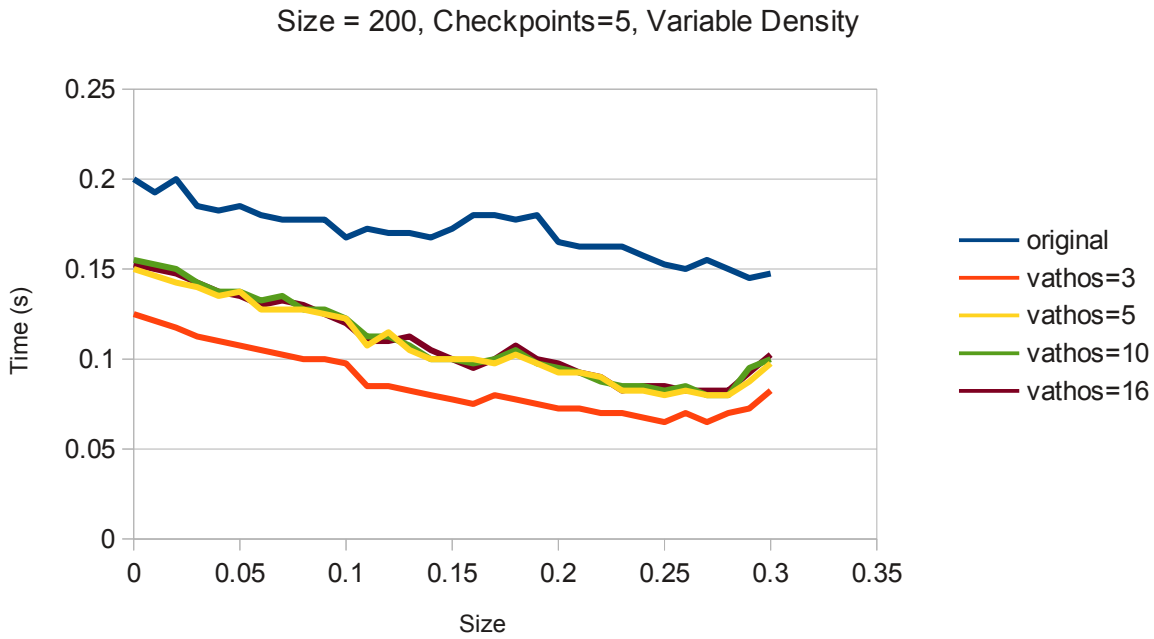
Στα επόμενα διαγράμματα φαίνονται οι μετρήσεις που πήραμε από χάρτες μικρού μεγέθους (20x20) με 5 ενδιάμεσα σημεία, αλλά με μεταβλητή πυκνότητα εμποδίων (από 0% έως 30%).

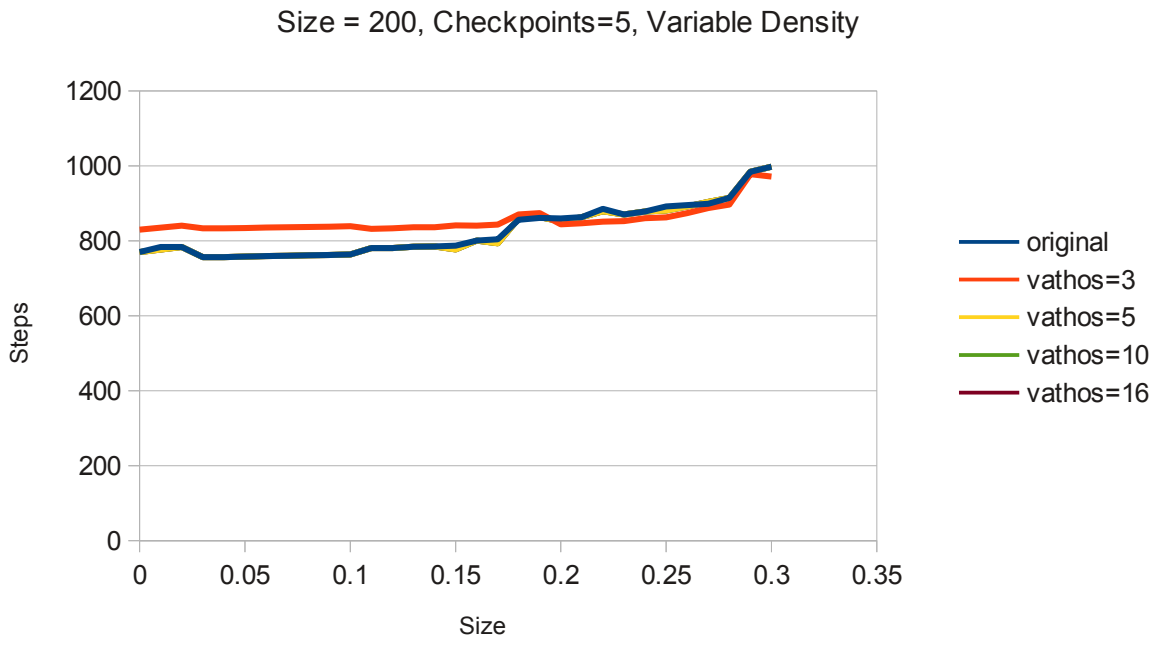




Όσον αφορά το χρόνο, για τον έλεγχο όλων των ενδιάμεσων σημείων, παρατηρούνται ορισμένα spikes, τα οποία, όμως, είναι της τάξεων του εκατοστού του δευτερολέπτου, επομένως αποδίδονται σε τυχαία γεγονότα (πχ χρήση του επεξεργαστή σε κάποια άλλη άσχετη με το πρόγραμμάς μας λειτουργία). Αυτό που έχει ιδιαίτερο ενδιαφέρον είναι το διάγραμμα των βημάτων, αφού γίνεται καταφανής η αδυναμία του vathos = 3 να δώσει ένα καλό αποτέλεσμα. Αντίθετα, τα vathos 5-16 φαίνεται να δίνουν σχεδόν σε όλες τις περιπτώσεις ίδιο αποτέλεσμα με τον πλήρη έλεγχο.

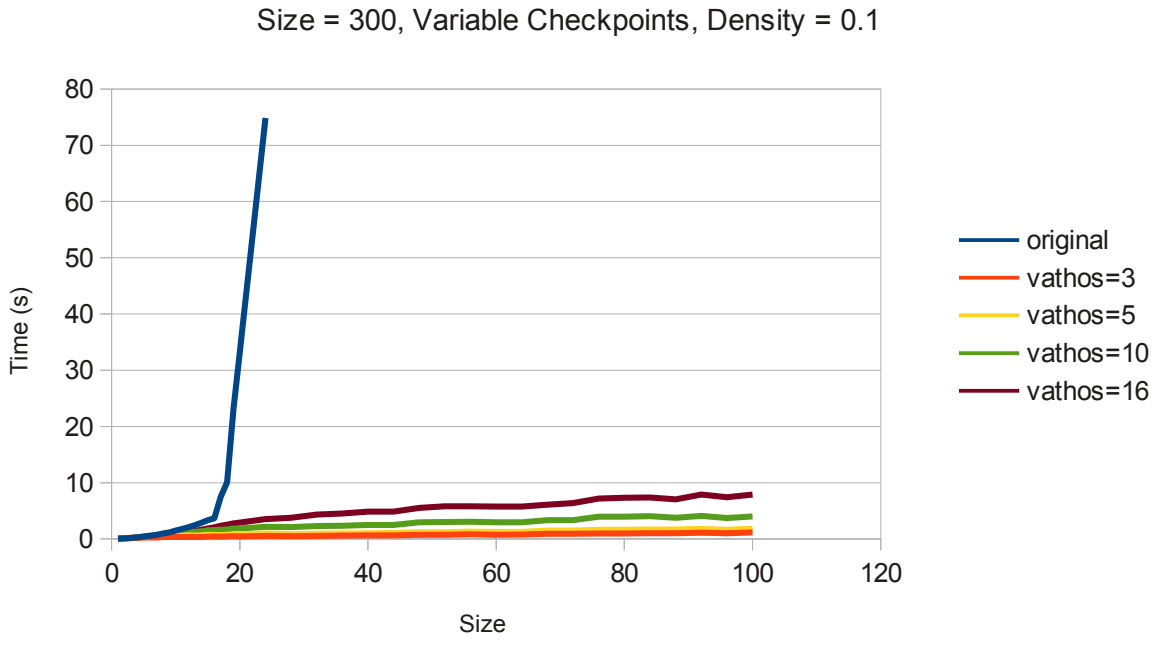
Στα παρακάτω διαγράμματα έχουμε μεγάλους χάρτες (200x200), 5 ενδιάμεσα σημεία, αλλά μεταβλητή πυκνότητα εμποδίων (0%-30%).

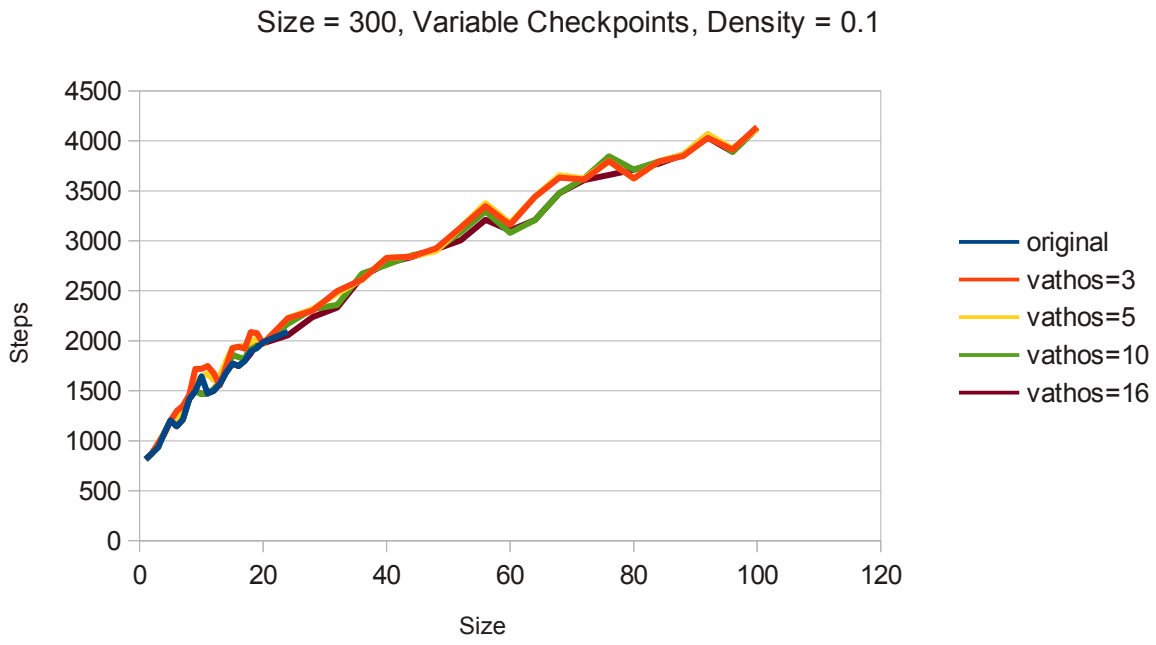




Οι παρατηρήσεις για αυτή την ομάδα μετρήσεων είναι ίδιες με τις παραπάνω, με εξαίρεση ότι σε αυτή την περίπτωση γίνεται φανερός ο χρονικός διαχωρισμός μεταξύ των διαφόρων τιμών του `vathos`.

Τα παρακάτω διαγράμματα είναι ιδιαίτερα σημαντικά, καθώς κατέδειξαν την αδυναμία του αρχικού μας σχεδίου και μας ώθησαν στην εισαγωγή της μεταβλητής `vathos`.

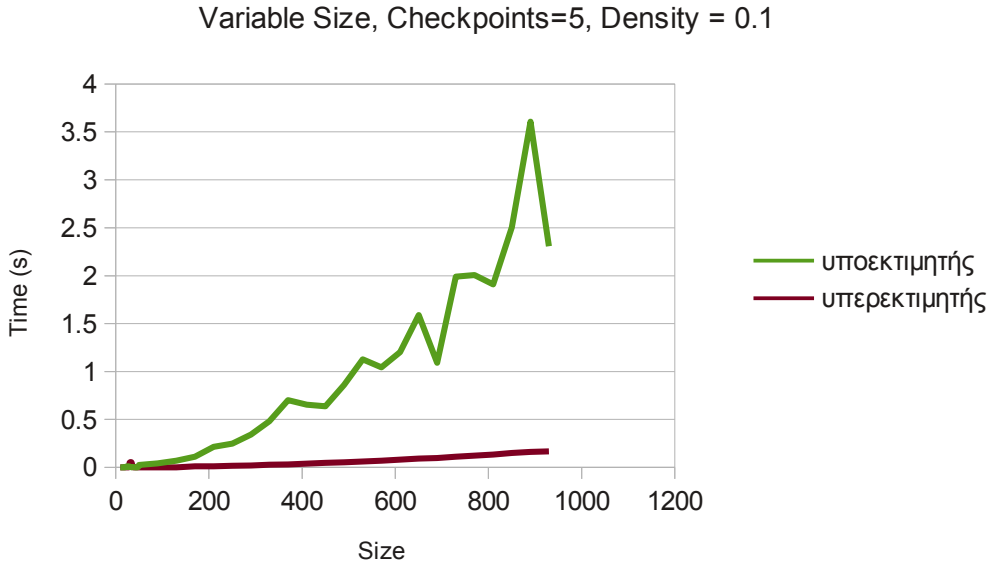


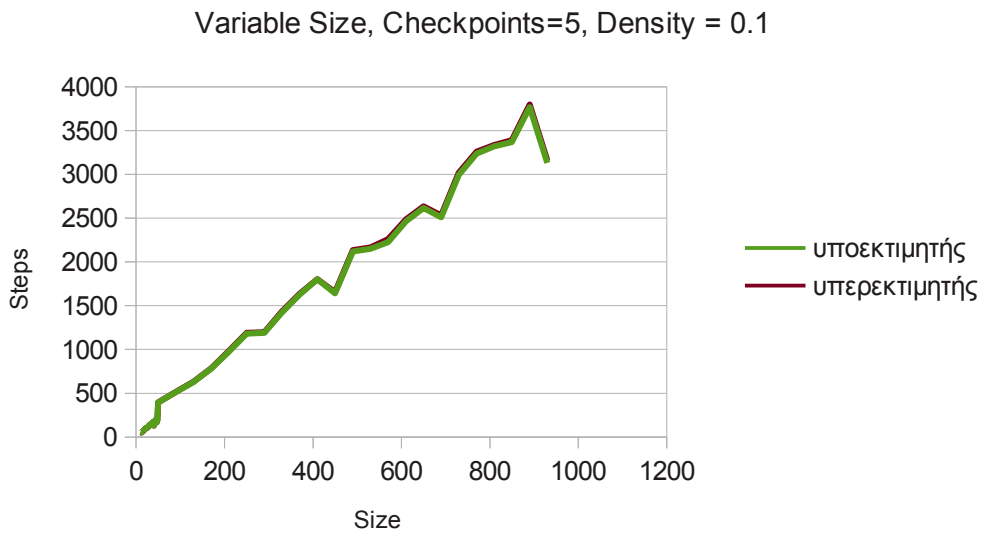


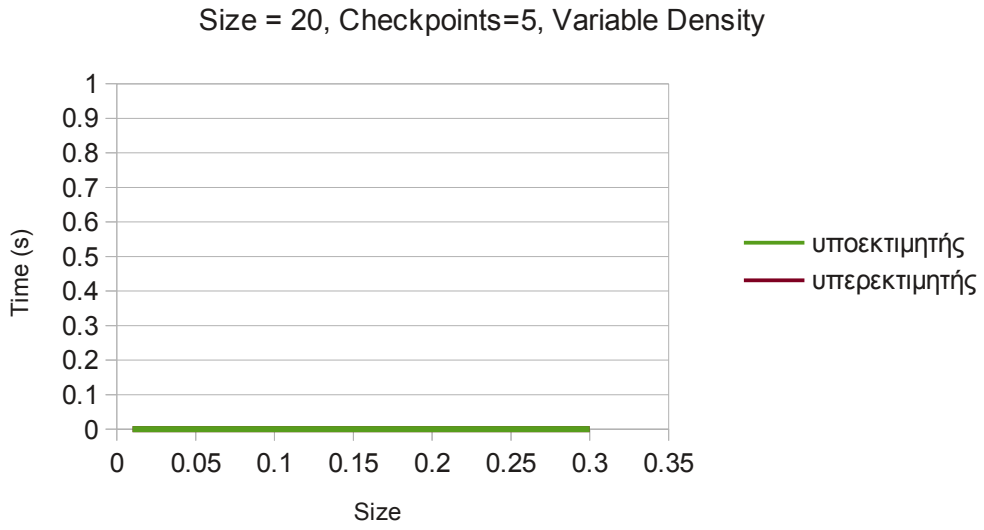
Για μεταβλητή αριθμό checkpoints (από 0 έως 100), ο χρόνος που χρειάζεται ο πλήρης έλεγχος αυξάνεται εκθετικά. Αντίθετα, μέσω της μεταβλητής vathos, ο χρόνος γίνεται σχεδόν γραμμικός και έχουμε ως έξοδο περίπου - αν όχι ακριβώς - τον ίδιο αριθμό βημάτων.

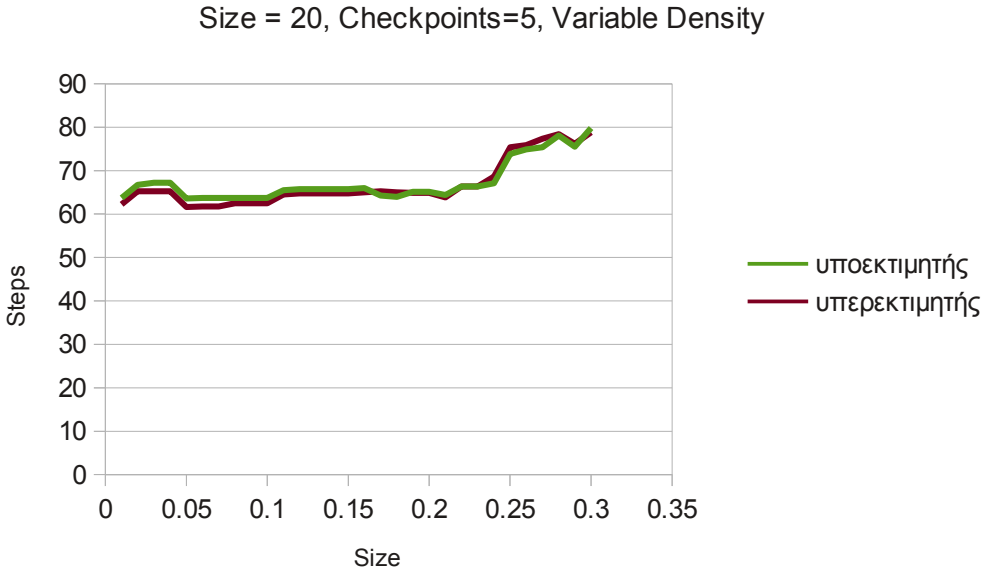
5.2. Μετρήσεις με υπερεκτιμητή – μία συγκριτική προσέγγιση

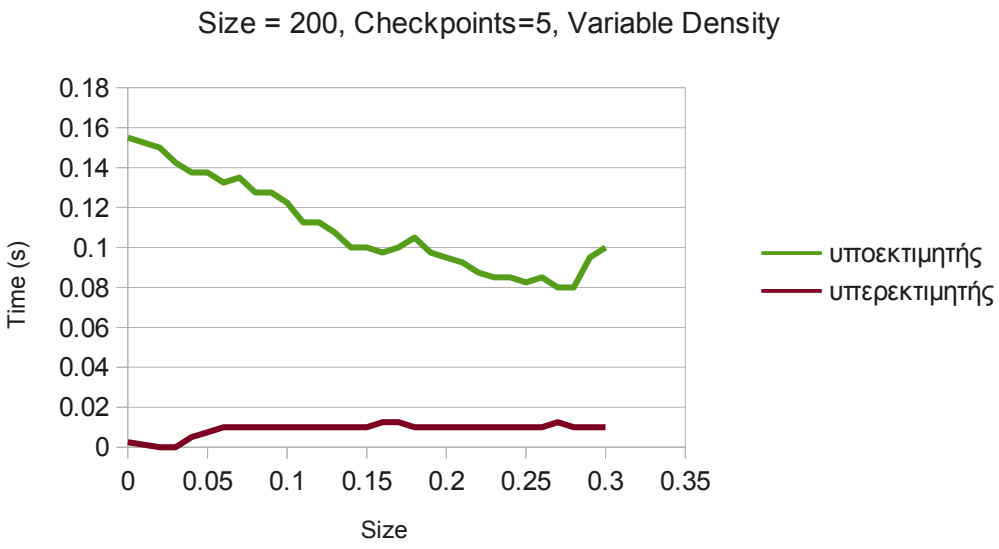
Στα παρακάτω διαγράμματα έχουμε λάβει μετρήσεις με τις ίδιες παραμέτρους για `vathos = 10` τόσο για τον υποεκτιμητή όσο και για τον υπερεκτιμητή.

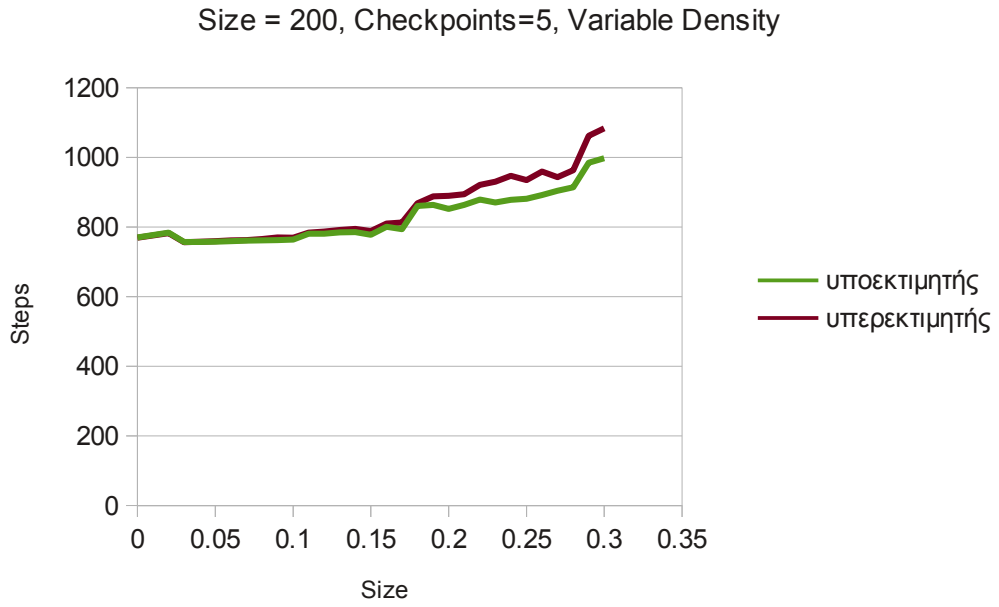


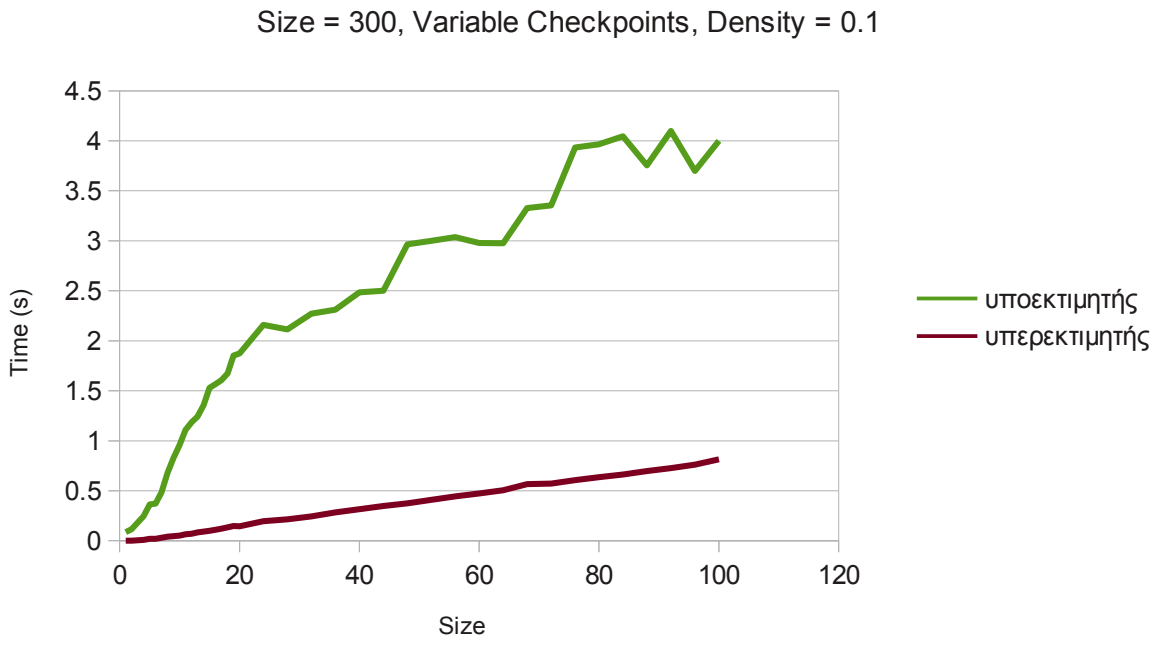

Variable Size, Checkpoints=5, Density = 0.1
Steps
Size
0
500
1000
1500
2000
2500
3000
3500
4000
0
200
400
600
800
1000
1200
υποεκτιμητής
υπερεκτιμητής

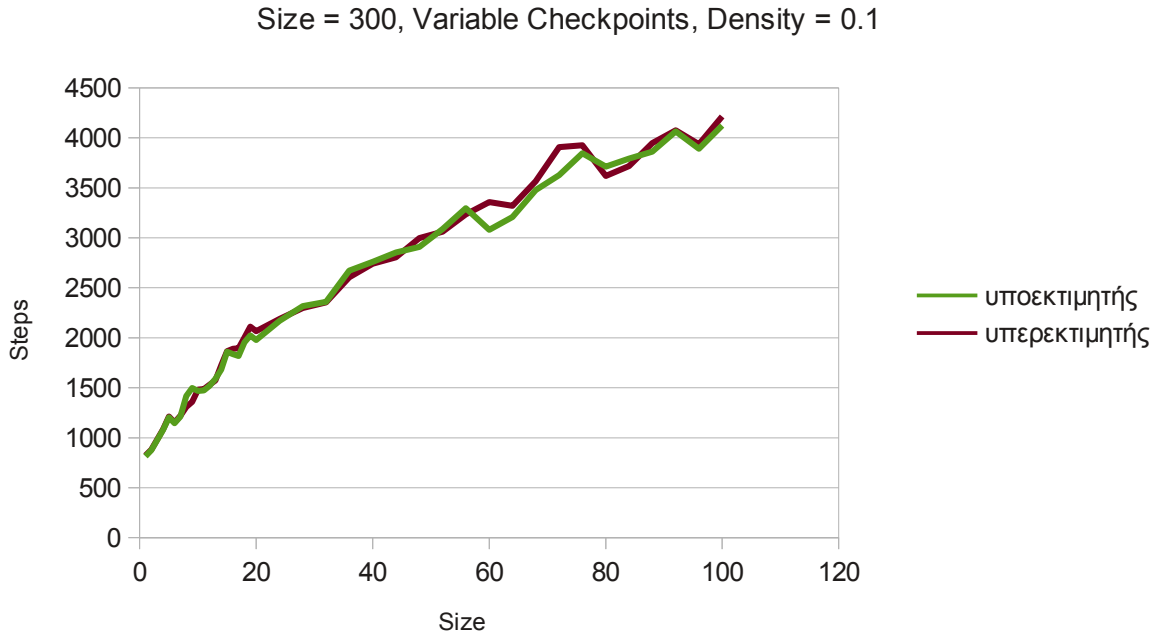

Size = 20, Checkpoints=5, Variable Density
Time (s)
Size
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
υποεκτιμητής
υπερεκτιμητής


Size = 20, Checkpoints=5, Variable Density
Steps
Size
0
10
20
30
40
50
60
70
80
90
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
υποεκτιμητής
υπερεκτιμητής


Size = 200, Checkpoints=5, Variable Density
Time (s)
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
Size
υποεκτιμητής
υπερεκτιμητής


Size = 200, Checkpoints=5, Variable Density
Steps
1200
1000
800
600
400
200
0
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
Size
υποεκτιμητής
υπερεκτιμητής





Όπως φαίνεται από τα παραπάνω διαγράμματα, ο χρόνος που χρειάζεται για να τρέξει το πρόγραμμα είναι μικρότερος στην περίπτωση του υπερεκτιμητή. Η διαφορά αυτή αποδίδεται αφενός στην ίδια την ύπαρξη του υπερεκτιμητή, αλλά και στο γεγονός ότι ο υποεκτιμητής υλοποιείται με if condition, ενώ ο υπερκτιμητής με αριθμητική πράξη.
Τα αποτελέσματα είναι πιο αναμενόμενα στην περίπτωση των βημάτων, καθώς φαίνεται ότι ο υπερκτιμητής μας δίνει καλύτερα αποτελέσματα σε περιπτώσεις χαμηλών πυκνοτήτων εμποδίων και ενδιαμέσων σημείων, αλλά ο υποεκτιμητής υπερτερεί σε πιο ακραίες περιπτώσεις.