ΣΗΜΜΥ ΕΜΠ
ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

Χειμερινό Εξάμηνο 2012-2013

ΑΣΚΗΣΗ 1η

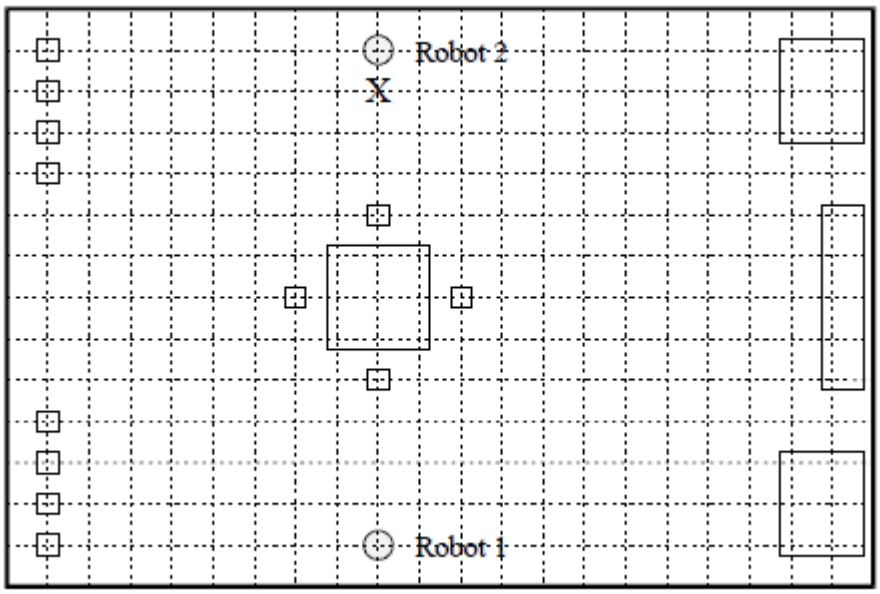
Ματίκας Γεώργιος (ΑΜ: 03109109)
Δανασής Παναγιώτης (ΑΜ: 03109004)

1. Το πρόβλημα που επιλύουμε με τη συγκεκριμένη εργασία αφορά την κίνηση 2 ρομπότ (Robot 1 και Robot 2) σε ένα χώρο που περιλαμβάνει εμπόδια, ενώ το πρώτο από αυτά προσπαθεί να βρει και να συναντηθεί με το δεύτερο. Και τα 2 ρομπότ γνωρίζουν (έχουν συνεχώς αποθηκευμένο στη μνήμη τους) τον χάρτη του χώρου μέσα στον οποίο κινούνται, ώστε να μπορούν να αποφεύγουν τα διάφορα εμπόδια. Το Robot 1 γνωρίζει, επιπροσθέτως, τη θέση που έχει το Robot 2 κάθε στιγμή, ώστε να μπορεί να επιλέγει κατάλληλα την πορεία του για να το συναντήσει όσο πιο σύντομα μπορεί. Τα 2 ρομπότ κινούνται σε γύρους. Σε κάθε γύρο κινείται πρώτο το Robot 2 και στη συνέχεια το Robot 1. Έτσι έχουμε ότι πρώτα κινείται το Robot 2, μετά το Robot 1, μετά το Robot 2 κοκ. Σε κάθε γύρο το Robot 2 κινείται κατά ένα τετράγωνο, ενώ το Robot 1 (του οποίου ουσιαστικά τον αλγόριθμο μας ενδιαφέρει να υλοποιήσουμε) κινείται κατά 3 τετράγωνα. Οι μόνες επιτρεπτές κινήσεις που μπορούν να κάνουν τα δύο ρομπότ είναι να κινηθούν πάνω, κάτω, αριστερά και δεξιά, αλλά όχι διαγώνια. Έχουμε θεωρήσει ότι το Robot 2 κινείται τυχαία στο χώρο σε κάθε κίνησή του, απλά αποφεύγοντας τα διάφορα εμπόδια που υπάρχουν (συμπεριλαμβανομένου και του Robot 1). Για το Robot 1 έχουμε υλοποιήσει τον αλγόριθμο A*, ώστε να εντοπίζει την πορεία του προς το Robot 2. Έτσι, κάθε φορά που έρχεται η σειρά του Robot 1 να κινηθεί, εκτελεί τον αλγόριθμο A* ανάμεσα στο τετράγωνο στο οποίο βρίσκεται αυτό και στο τετράγωνο στο οποίο είναι εκείνη τη στιγμή το Robot 2 και εκτελεί τις 3 πρώτες κινήσεις από το μονοπάτι που προκύπτει. Επειδή ουσιαστικά το Robot 1 κινείται με τριπλάσια ταχύτητα από το Robot 2 αναμένουμε ότι σύντομα θα το συναντήσει.

Τα δεδομένα των αρχικών θέσεων των δύο ρομπότ καθώς και ο χάρτης της αίθουσας μέσα στην οποία κινούνται δίνονται στο πρόγραμμά μας με τη μορφή ενός αρχείου κειμένου. Στο αρχείο αυτό η πρώτη γραμμή περιέχει τη διάσταση του χώρου σε αριθμό τετραγώνων, με πρώτη διάσταση την οριζόντια και δεύτερη διάσταση την κατακόρυφη. Θεωρώντας ότι η οριζόντια συντεταγμένη αυξάνεται προς τα δεξιά, η κατακόρυφη προς τα κάτω και πως οι συντεταγμένες του πάνω αριστερά τετραγώνου είναι (1,1), στις επόμενες δύο γραμμές μας δίνονται οι αρχικές θέσεις των δύο ρομπότ στην αίθουσα (πρώτα του Robot 1 και μετά του Robot 2). Ο χάρτης της αίθουσας μας δίνεται στις υπόλοιπες γραμμές του αρχείου εισόδου σε μορφή ενός πίνακα όπου όταν ένα τετράγωνο έχει 'Ο' θεωρούμε ότι είναι ελεύθερο, ενώ όταν έχει 'Χ' θεωρούμε ότι εκεί υπάρχει εμπόδιο. Δεχόμαστε ότι οι αρχικές θέσεις των δύο ρομπότ δεν είναι κατειλημμένες από εμπόδια. Έτσι το ακόλουθο αρχείο εισόδου:

```
20 13
13 9
2 9
XOOOOOOOXOOOOOOOOOXX
XOOOOOOOOOOOOOOOOOXX
XOOOOOOOOOOOOOOOOOXX
XOOOOOOOOOOOOOOOOOO
OOOOOOOOXOOOOOOOOOX
OOOOOOOXXXOOOOOOOOX
OOOOOOXXXXXOOOOOOOX
OOOOOOOXXXOOOOOOOOX
OOOOOOOOXOOOOOOOOOX
XOOOOOOOOOOOOOOOOOO
XOOOOOOOOOOOOOOOOXX
XOOOOOOOOOOOOOOOOXX
XOOOOOOOOOOOOOOOOXX
```

περιγράφει την ακόλουθη αίθουσα με εμπόδια:



2. Ως πρότυπο για την υλοποίηση του αλγορίθμου A* χρησιμοποιήσαμε τον ψευδοκώδικα που υπάρχει στο site της Wikipedia στο αντίστοιχο άρθρο εδώ:

http://en.wikipedia.org/wiki/A*_search_algorithm

Φυσικά, εφαρμόσαμε πολλές αλλαγές στον υπάρχοντα ψευδοκώδικα, ώστε να δημιουργείται παράλληλα και το δέντρο αναζήτησης με βάση τη δομή δεδομένων που δίνεται για τη μοντελοποίηση των καταστάσεων του χώρου καθώς και για να αντιμετωπίσουμε κάποιες ιδιαιτερότητες που υπάρχουν στην C++.

Για το σύνολο closedset, που αποθηκεύει τις καταστάσεις οι οποίες έχουν αναπτυχθεί πλήρως μέχρι στιγμής, χρησιμοποιήσαμε τη δομή συνόλων που ορίζεται στην βιβλιοθήκη <*set*> της C++ και δηλώνεται ως εξής: *set<int> closedset*. Για να αναπαραστήσουμε την κατάσταση στην οποία αντιστοιχεί κάθε τετράγωνο του χώρου, του αναθέσαμε έναν μοναδικό ακέραιο αριθμό. Στο πάνω αριστερά τετράγωνο αναθέσαμε την τιμή 0 και συνεχίσαμε ανά 1 για τα υπόλοιπα τετράγωνα προς τα δεξιά και προς τα κάτω. Έτσι έχουμε X*Y καταστάσεις συνολικά (όπου X και Y οι διαστάσεις του χώρου), από το 0 έως το (X*Y-1). Γι' αυτό, λοιπόν, χρησιμοποιήσαμε ένα σύνολο από ακέραιους (set<int>) για να μας δείχνει ποιες καταστάσεις (τετράγωνα) έχουμε αναπτύξει πλήρως μέχρι τώρα.

Για το μέτωπο της αναζήτησης χρησιμοποιήσαμε τη δομή της ουράς προτεραιότητας η οποία ορίζεται στην βιβλιοθήκη <*queue*> της C++. Για στοιχεία αυτής της ουράς θεωρήσαμε ένα struct που κατασκευάσαμε εμείς και περιλαμβάνει ως στοιχεία του το αναγνωριστικό κάθε κατάστασης που βρίσκεται στο μέτωπο της αναζήτησης (ένας ακέραιος όπως αναφέραμε πριν), την τιμή της συνάρτησης f(x)=g(x)+h(x) για την κατάσταση αυτή (όπου g(x) είναι η απόσταση που έχουμε διανύσει έως τώρα και h(x) η απόσταση που πιστεύουμε ότι πρέπει να διανύσουμε

ακόμα μέχρι το στόχο), καθώς και έναν pointer στο αντίστοιχο node της κατάστασης αυτής στο δέντρο αναζήτησης (ώστε να μπορούμε στο μέλλον όταν εξετάσουμε την κατάσταση αυτή να βρούμε τα παιδιά της, τα οποία δεικτοδοτούνται από το node αυτό). Το struct αυτό φαίνεται παρακάτω:

```
typedef struct {
   int s;
   int f;
   SearchGraphNode *p;
} tuple;
```

Η ουρά αυτή έχει πάντα στην κορυφή της την κατάσταση με την μικρότερη τιμή της συνάρτησης f(x). Για να το πετύχουμε αυτό χρειάστηκε να ορίσουμε τη συνάρτηση σύγκρισης για τα στοιχεία της ουράς ως εξής:

```
class mycomparison
{
public:
  bool operator() (const tuple& lhs, const tuple& rhs) const
  {
    return (lhs.f>rhs.f);
  }
};
```

Έτσι, τελικά δηλώσαμε το μέτωπο της αναζήτησης ως εξής:

priority_queue< tuple, vector<tuple>, mycomparison > frontier;

Επειδή, όμως, στη C++ και συγκεκριμένα στη δομή της ουράς προτεραιότητας δεν έχουμε διαθέσιμη κάποια μέθοδο για να ψάχνουμε αν αυτή περιλαμβάνει κάποιο συγκεκριμένο στοιχείο, ορίσαμε και ένα σύνολο με όνομα openset, το οποίο περιλαμβάνει τα αναγνωριστικά όλων των καταστάσεων οι οποίες βρίσκονται στο μέτωπο της αναζήτησης (και άρα και στην ουρά προτεραιότητας frontier). Έτσι όταν θέλουμε να δούμε αν κάποια κατάσταση βρίσκεται στο μέτωπο αναζήτησης δεν ψάχνουμε την ουρά frontier, αλλά το σύνολο openset. Το σύνολο αυτό δηλώθηκε με τον ίδιο τρόπο με το σύνολο closedset, δηλαδή ως εξής: *set<int> openset*.

Για τη μοντελοποίηση των καταστάσεων του χώρου και του δέντρου αναζήτησης χρησιμοποιήσαμε τη δομή δεδομένων που προτείνεται στην εκφώνηση με κάποιες πολύ μικρές αλλαγές. Έτσι η δομή που χρησιμοποιήσαμε είναι η εξής:

```
typedef struct Node {
   struct Node *parent;        // Πατέρας του τρέχοντος κόμβου
   int state;                  // Αναγνωριστικό της κατάστασης
   int g;                      // Κόστος μετάβασης στην κατάσταση αυτή
   int h;                      // Εκτιμώμενη υπολειπόμενη απόσταση προς στόχο
```

```
    struct Node *next;          // CLOSED/OPEN, ανάλογα αν επεκτάθηκε ή όχι
    struct Node *r_sibling;     // Δείκτης στο δεξί αδερφό
    struct Node *child1;        // Δείκτης στο πιο αριστερό παιδί
} SearchGraphNode;

SearchGraphNode *OPEN, *CLOSED;     // OPEN: προς επέκταση
                                    // CLOSED: δεν επεκτάθηκε ακόμη
```

3. Σχεδιάσαμε μία συνάρτηση για κάθε δυνατό τελεστή μετάβασης από μία κατάσταση σε μία άλλη (δηλαδή από ένα τετράγωνο σε ένα γειτονικό του). Επειδή οι μόνες δυνατές κινήσεις των ρομπότ είναι πάνω, κάτω, αριστερά και δεξιά, σχεδιάσαμε 4 συνολικά συναρτήσεις, μία για κάθε δυνατή μετάβαση. Οι συναρτήσεις αυτές επιστρέφουν στο όνομά τους την τιμή 1 αν η μετάβαση είναι δυνατή (επιτρεπτή) και 0 αν δεν είναι δυνατή (πχ υπάρχει εμπόδιο ή βγαίνουμε εκτός ορίων χάρτη). Οι νέες συντεταγμένες επιστρέφονται σε κάθε περίπτωση στις μεταβλητές nx, ny οι οποίες περνούνται στις συναρτήσεις by reference. Οι συναρτήσεις που σχεδιάσαμε είναι οι εξής:

```
int up(int x, int y, int *nx, int *ny) {

    if ((y!=0)&&(plane[y-1][x]=='O')) {
      *nx=x;
      *ny=y-1;
      return 1;
    }
    else return 0;
}

int down(int x, int y, int *nx, int *ny) {

    if ((y!=(Y-1))&&(plane[y+1][x]=='O')) {
      *nx=x;
      *ny=y+1;
      return 1;
    }
    else return 0;
}

int left(int x, int y, int *nx, int *ny) {

    if ((x!=0)&&(plane[y][x-1]=='O')) {
      *nx=x-1;
      *ny=y;
      return 1;
    }
```

```
    else return 0;
}

int right(int x, int y, int *nx, int *ny) {

    if ((x!=(X-1))&&(plane[y][x+1]=='O')) {
      *nx=x+1;
      *ny=y;
      return 1;
    }
    else return 0;
}
```

4. Για την υλοποίηση του αλγορίθμου A* χρειάζεται μια συνάρτηση εκτίμησης της υπολειπόμενης απόστασης από την τρέχουσα κατάσταση έως την τελική κατάσταση. Όταν η τιμή της συνάρτησης αυτής είναι πάντα μικρότερη ή ίση από την πραγματική υπολειπόμενη απόσταση από τον στόχο, τότε καλείται υπο-εκτιμητής και αποτελεί έναν αποδεκτό ευριστικό μηχανισμό (admissible heuristic). Σε αντίθετη περίπτωση, η συνάρτηση καλείται υπερ-εκτιμητής και αποτελεί έναν μη αποδεκτό ευριστικό μηχανισμό (non-admissible heuristic). Σε αυτή την περίπτωση η εκτιμώμενη απόσταση από τον στόχο μπορεί να προκύψει μεγαλύτερη από την πραγματική απόσταση από αυτόν.

Ο υπο-εκτιμητής που χρησιμοποιήσαμε είναι η απόσταση Manhattan ανάμεσα στα δύο τετράγωνα, η οποία ισούται με το άθροισμα των απολύτων διαφορών των συντεταγμένων των δύο τετραγώνων, δηλαδή:

$$h(x1,y1,x2,y2) = |x1-x2| + |y1-y2|$$

Περισσότερες πληροφορίες μπορούν να βρεθούν εδώ:

http://en.wikipedia.org/wiki/Manhattan_distance

Η συνάρτηση που υλοποιεί τον ευριστικό μηχανισμό αυτό είναι:

```
int heur1(int x1, int y1, int x2, int y2) {
    return (abs(x1-x2)+abs(y1-y2));
}
```

Ο υπερ-εκτιμητής που χρησιμοποιήσαμε είναι το τετράγωνο της ευκλείδιας απόστασης ανάμεσα στα δύο τετράγωνα, η οποία ισούται με το άθροισμα των τετραγώνων των διαφορών των συντεταγμένων των δύο τετραγώνων, δηλαδή:

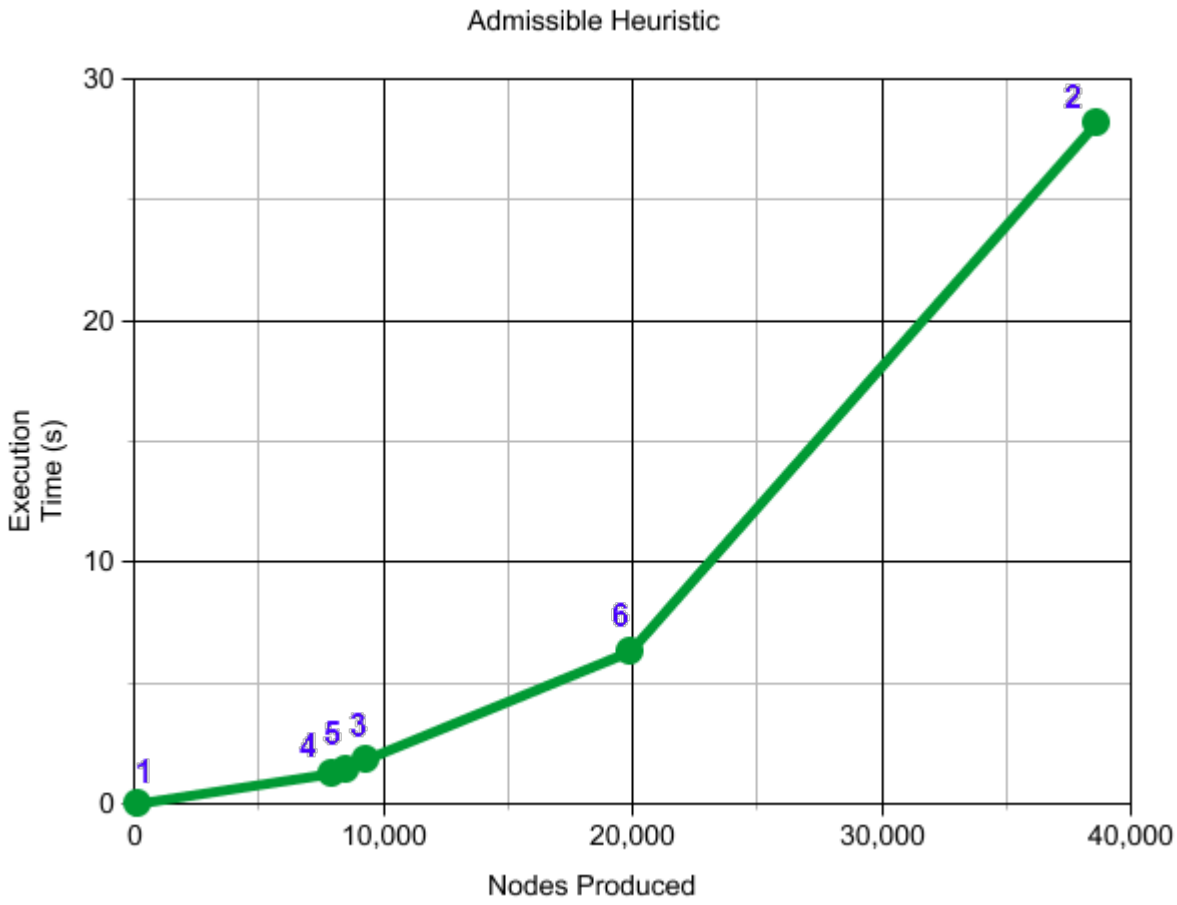
$$h(x1,y1,x2,y2) = (x1-x2)^2 + (y1-y2)^2$$

Η τιμή αυτή πολλές φορές προκύπτει μεγαλύτερη από την πραγματικά υπολειπόμενη απόσταση ανάμεσα στα δύο τετράγωνα με συντεταγμένες (x1,y1) και (x2,y2). Η συνάρτηση που υλοποιεί τον ευριστικό μηχανισμό αυτό είναι:

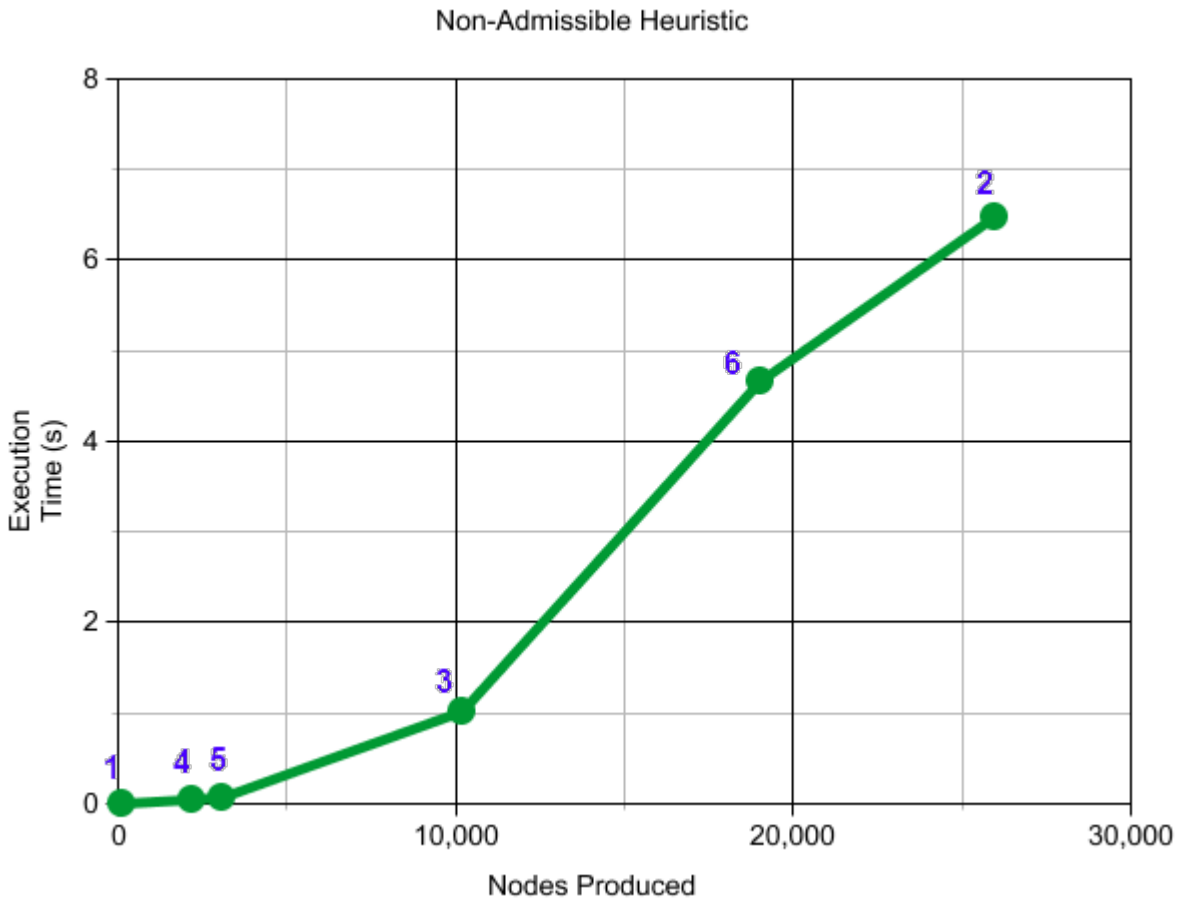
```
int heur2(int x1, int y1, int x2, int y2) {
   return (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
}
```

5. Τα αρχεία που χρησιμοποιήθηκαν βρίσκονται στον ίδιο φάκελο με την αναφορά και είναι τα *1.txt*, *2.txt*, *3.txt*, *4.txt*, *5.txt* και *6.txt*.

Υπο-εκτιμητής



Υπερ-εκτιμητής



6. Μια υπο-εκτιμήτρια ευριστική συνάρτηση αποτελεί έναν αποκετό ευριστικό μηχανισμό. Η χρήση ενός αποδεκτού ευριστικού μηχανισμού εγγυάται ότι θα έχουμε ως αποτέλεσμα τη βέλτιστη λύση, δηλαδή στο πρόβλημά μας το συντομότερο μονοπάτι ανάμεσα στον αρχικό κόμβο και στον κόμβο-στόχο. Αντιθέτως, μια υπερ-εκτιμήτρια ευριστική συνάρτηση αποτελεί έναν μη-αποδεκτό ευριστικό μηχανισμό. Η χρήση ενός μη-αποδεκτού ευριστικού μηχανισμού έχει ως αποτέλεσμα ότι η λύση που θα βρούμε ενδέχεται να μην είναι η βέλτιστη, δηλαδή στο πρόβλημά μας το μονοπάτι που θα βρούμε ενδέχεται να μην είναι το συντομότερο δυνατό. Αυτό οφείλεται στο ότι σε πολλές καταστάσεις έχουμε υπερεκτίμηση της απόστασής τους από τον στόχο και έτσι ενδέχεται οι καταστάσεις του συντομότερου μονοπατιού να έχουν τώρα μεγαλύτερη τιμή για τη συνάρτηση f από ότι άλλες. Αποτέλεσμα αυτού θα είναι ο αλγόριθμος να μην ακολουθήσει το βέλτιστο μονοπάτι αλλά κάποιο άλλο. Ωστόσο, έχει παρατηρηθεί ότι πολύ συχνά η λύση που θα δοθεί είνα αρκετά κοντά στη βέλτιστη και, μάλιστα, οι κόμβοι που εξετάζονται είναι αρκετά λιγότεροι από όταν χρησιμοποιούμε υπο-εκτιμήτρια συνάρτηση, με αντίκτυπο και στο χρόνο εκτέλεσης του αλγορίθμου (λιγότεροι εξετασθέντες κόμβοι --> λιγότερος χρόνος).

Τα παραπάνω επιβεβαιώνονται και στις δοκιμές που κάναμε στο προηγούμενο ερώτημα. Παρατηρούμε ότι με τη χρήση της υπερ-εκτιμήτριας συνάρτησης τόσο οι κόμβοι που

εισήχθησαν στο δέντρο της αναζήτησης όσο και ο χρόνος εκτέλεσης του προγράμματος έχουν αρκετά μικρότερες τιμές.

7. Ο πηγαίος κώδικας του προγράμματός μας βρίσκεται στο αρχείο *ex1.cpp* . Στην αρχή του κώδικα γίνεται ορισμός της σταθεράς *HEUR*, η οποία ορίζει ποια ευριστική συνάρτηση θα χρησιμοποιηθεί. Για να χρησιμοποιηθεί η υπο-εκτιμήτρια τη δηλώνουμε ως *heur1*, ενώ για να χρησιμοποιηθεί η υπερ-εκτιμήτρια τη δηλώνουμε ως *heur2*.