


| | | | |
|--|---|--|---|
|  Πανεπιστήμιο Ιωαννίνων | Τμήμα Πληροφορικής & Τηλεπικοινωνιών Διδάσκων: Στύλιος Χρυσόστομος Επιμέλεια σημειώσεων: Γκόγκος Χρήστος Μάθημα: Τεχνητή Νοημοσύνη (εργαστήριο Ε' εξαμήνου) | Ακαδημαϊκό έτος 2021-2022 Χειμερινό Εξάμηνο | 2 |
|--|---|--|---|

Ευρετική αναζήτηση (πληροφορημένη αναζήτηση)

Οι ευρετικές τεχνικές (heuristic techniques) είναι τεχνικές επίλυσης προβλημάτων που χρησιμοποιούν κάποια γνώση για το προς επίλυση πρόβλημα έτσι ώστε να οδηγηθούν στη γρήγορη επίλυσή του. Πλεονεκτήματά τους είναι ότι δίνουν καλά αποτελέσματα και ότι απαιτούν λιγότερο χρόνο σε σχέση με άλλες προσεγγίσεις. Ωστόσο πολλές φορές οι ευρετικές τεχνικές δεν μπορούν να εγγυηθούν ότι η συμπεριφορά αυτή θα ισχύει για όλες τις διαφορετικές περιπτώσεις προβλημάτων. Η ευρετική αναζήτηση χρησιμοποιεί την λεγόμενη ευρετική συνάρτηση (heuristic function) που για κάθε κατάσταση S του προβλήματος επιστρέφει μια αριθμητική τιμή που αποτελεί εκτίμηση της απόστασης της κατάστασης S από την κατάσταση στόχο G . Η τιμή που επιστρέφεται από την ευρετική συνάρτηση συμμετέχει στην επιλογή ανάμεσα στις εναλλακτικές καταστάσεις που πρέπει να εξεταστούν κατά τη διάρκεια της αναζήτησης. Στη συνέχεια θα εξεταστούν 3 αλγόριθμοι ευρετικής αναζήτησης: ο αλγόριθμος αναρρίχησης λόφου, ο αλγόριθμος αναζήτησης πρώτα στο καλύτερο και ο αλγόριθμος A^* [Amit], [Norvig03], [Βλαχαβάς06].

Αναρρίχηση λόφου (HC=Hill Climbing)

Στον αλγόριθμο αναρρίχησης λόφου (HC) επιλέγεται σε κάθε βήμα από τις καταστάσεις παιδιά η κατάσταση για την οποία η ευρετική συνάρτηση επιστρέφει την καλύτερη τιμή. Όλες οι υπόλοιπες καταστάσεις απορρίπτονται. Η κατάσταση που επιλέγεται εφόσον έχει καλύτερη τιμή από την τρέχουσα κατάσταση επεκτείνεται και για τα παιδιά της ακολουθείται η ίδια διαδικασία. Αν δεν παρουσιαστεί πρόοδος σε κάποιο βήμα τότε ο αλγόριθμος αναρρίχησης λόφου καταλήγει σε αδιέξοδο. Συνεπώς, ο αλγόριθμος αναρρίχησης λόφου διατηρεί ανά πάσα στιγμή στο μέτωπο αναζήτησης μόνο μια κατάσταση.

Ο αλγόριθμος HC δεν παρέχει εγγύηση ότι θα βρει λύση. Μπορεί να καταλήξει σε αδιέξοδο ενώ υπάρχει λύση.

```

algorithm hc(InitialState, FinalState)
begin
  CurrentState ← InitialState;
  while CurrentState ≠ FinalState do
    Children ← Expand(CurrentState);
    if Children = ∅ then return failure;
    EvaluatedChildren ← Heuristic(Children);
    bestChild ← best(EvaluatedChildren);
    if hValue(CurrentState) ≥ hValue(bestChild)
      then return failure;
      else CurrentState ← bestChild;
    endif;
  endwhile;
  return success;
end.

```

Ψευδοκώδικας για τον αλγόριθμο αναρρίχησης λόφου (Βλαχαβάς06)

Αναζήτηση πρώτα στο καλύτερο (BestFS=Best First Search)

Ο αλγόριθμος αναζήτησης πρώτα στο καλύτερο (BestFS) λειτουργεί παρόμοια με την αναρρίχηση λόφου με τη διαφορά ότι διατηρεί όλες τις καταστάσεις στο μέτωπο αναζήτησης. Σε κάθε βήμα επιλέγεται η κατάσταση για την οποία η ευρετική συνάρτηση επιστρέφει την καλύτερη τιμή. Αν η κατάσταση που επιλέχθηκε είναι η κατάσταση στόχος τότε η αναζήτηση τερματίζεται επιτυχώς. Αλλιώς αφαιρείται από το μέτωπο αναζήτησης, προστίθεται στο

κλειστό σύνολο και επεκτείνεται (δηλαδή οι γειτονικές καταστάσεις της εισέρχονται στο μέτωπο αναζήτησης). Θα πρέπει να σημειωθεί ότι η αναζήτηση BestFS λαμβάνει υπόψη για την επιλογή της τρέχουσας κατάστασης από το μέτωπο αναζήτησης μόνο την εκτίμηση της ευρετικής συνάρτησης σχετικά με την απόστασή της από την κατάσταση στόχο.

Ο αλγόριθμος BestFS παρέχει εγγύηση ότι θα βρει λύση εφόσον υπάρχει αλλά δεν εγγυάται ότι η λύση που θα εντοπίσει θα είναι η βέλτιστη.

Άλφα-άστρο (A^*)

Η διαφορά του αλγορίθμου A^* σε σχέση με τον BestFS είναι ότι ενώ στην περίπτωση του BestFS εξετάζεται μόνο η τιμή h που επιστρέφει η ευρετική συνάρτηση ως εκτίμηση της απόστασης της τρέχουσας κατάστασης από την κατάσταση στόχο, στον A^* η τιμή h προστίθεται στην απόσταση g που έχει ήδη διανυθεί από την αρχική μέχρι την τρέχουσα κατάσταση. Άρα για κάθε κατάσταση S που προστίθεται στο μέτωπο αναζήτησης υπολογίζεται μια αριθμητική τιμή $f(S)$ που σχηματίζεται ως άθροισμα των τιμών $g(S)$ και $h(S)$.

- $g(S)$ = απόσταση που έχει ήδη διανυθεί για να φτάσουμε στην κατάσταση S
- $h(S)$ = εκτίμηση της απόστασης της κατάστασης S από την κατάσταση στόχο
- $f(S) = g(S) + h(S)$

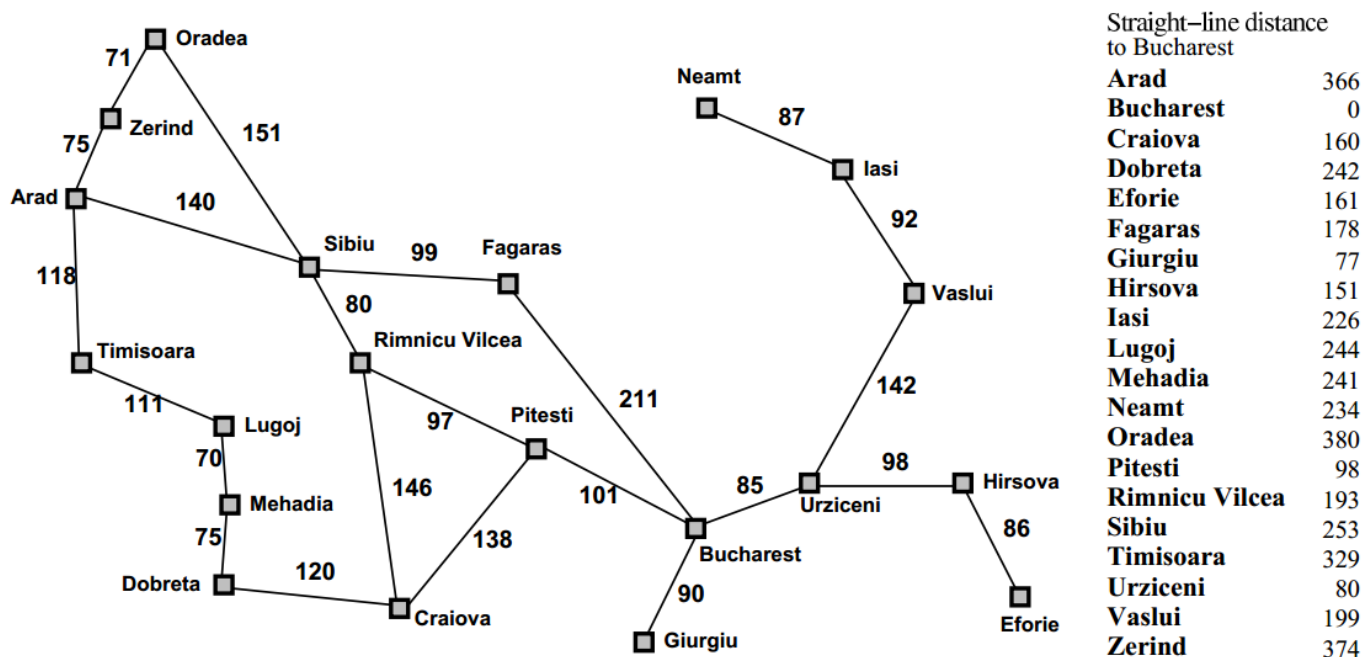
Αν για κάθε κατάσταση S η εκτίμηση της απόστασης της S από την κατάσταση στόχο όπως επιστρέφεται από την ευρετική συνάρτηση είναι μικρότερη ή ίση από την πραγματική απόσταση της S από την κατάσταση στόχο τότε αποδεικνύεται ότι ο αλγόριθμος A^* μπορεί να εντοπίσει τη βέλτιστη λύση του προβλήματος. Σε αυτή τη περίπτωση η ευρετική συνάρτηση ονομάζεται **αποδεκτή** ή **παραδεκτή** (admissible).

Ο αλγόριθμος A^* εφόσον υλοποιηθεί με τη χρήση κλειστού συνόλου μπορεί να απορρίπτει τη βέλτιστη διαδρομή αν η κατάσταση η οποία οδηγεί σε αυτή σε κάποιο κόμβο δεν είναι η πρώτη που παράγεται. Για να αντιμετωπιστεί αυτό το θέμα είτε μπορεί να απορρίπτεται η πιο δαπανηρή από δύο διαδρομές που προκύπτουν καταλήγοντας στον ίδιο κόμβο είτε να εξασφαλίζεται ότι η βέλτιστη διαδρομή προς οποιοδήποτε κόμβο θα είναι η πρώτη που επιλέγεται, κάτι το οποίο συμβαίνει αν η ευρετική συνάρτηση είναι **συνεπής** (consistent). Μια ευρετική συνάρτηση είναι συνεπής αν για κάθε κόμβο n και για κάθε διάδοχο της n' , το κόστος της ευρετικής συνάρτησης για το n είναι μικρότερο ή ίσο από κόστος της μετάβασης από το n στο n' συν το κόστος της ευρετικής συνάρτησης για το n' (πρόκειται για μια γενική μορφή της τριγωνικής ανισότητας).

Συνεπώς ο αλγόριθμος A^* επιστρέφει τη βέλτιστη λύση σε ένα πρόβλημα αναζήτησης αν η ευρετική συνάρτηση είναι αποδεκτή και συνεπής.

Πρόβλημα εύρεσης συντομότερης διαδρομής

Το πρόβλημα που θα χρησιμοποιηθεί για την επίδειξη των αλγορίθμων ευρετικής αναζήτησης είναι το ίδιο πρόβλημα με αυτό που παρουσιάστηκε στην εργαστηριακή άσκηση 1 με την προσθήκη ότι στα δεδομένα υπάρχουν πλέον και οι αποστάσεις σε ευθεία γραμμή κάθε πόλης από την πόλη Β (Bucharest). Οι αποστάσεις αυτές αναπαριστούν την τιμή που επιστρέφεται από την ευρετική συνάρτηση για κάθε μια από τις πόλεις και αποτελεί ένδειξη για το μήκος της συντομότερης διαδρομής που μπορεί να ακολουθηθεί από κάθε πόλη προς την πόλη προορισμό.



Εικόνα 1. Romania tour (για κάθε πόλη καταγράφεται η απόστασή της σε ευθεία γραμμή από την πόλη Bucharest)

Εφαρμογή του αλγορίθμου αναρρίχησης λόφου (HC) για τη μετάβαση από A στο B

| Τρέχουσα Κατάσταση | Παιδιά |
|--------------------|------------------------------|
| (A-366) | (S 253)(T 329)(Z 374) |
| (S-253) | (A 366)(F 178)(O 380)(R 193) |
| (F-178) | (B 0)(S 253) |
| (B-0) | Στόχος |

Διαδρομή και κόστος: A-S-F-B με μήκος διαδρομής 450

Εφαρμογή του αλγορίθμου αναζήτησης πρώτα στο καλύτερο (BestFS) για τη μετάβαση από A στο B

| Μέτωπο αναζήτησης | Κλειστό σύνολο | Τρέχουσα Κατάσταση | Παιδιά |
|---|----------------|--------------------|---------|
| (A 366) | \emptyset | A | S T Z |
| (A-S 253) (A-T 329) (A-Z 374) | A | S | A F O R |
| (A-S-F 178) (A-S-R 193) (A-T 329) (A-S-A 366) (A-Z 374) (A-S-O 380) | A, S | F | B S |
| (A-S-F-B 0) (A-S-R 193) (A-S-F-S 253) (A-T 329) (A-S-A 366) (A-Z 374) (A-S-O 380) | A, S, F | B | Στόχος |

Διαδρομή και κόστος: A-S-F-B με μήκος διαδρομής 450

Εφαρμογή του αλγορίθμου A* για τη μετάβαση από A στο B

| Μέτωπο αναζήτησης | Κλειστό σύνολο | Τρέχουσα Κατάσταση | Παιδιά |
|--|----------------|--------------------|---------|
| (A $0+366=366$) | \emptyset | A | S T Z |
| (A-S $140+253=393$) (A-T $118+329=447$) (A-Z $75+374=449$) | A | S | A F O R |
| (A-S-R $220+193=413$) (A-S-F $239+178=417$) (A-T $118+329=447$) (A-Z $75+374=449$) (A-S-A $280+366=646$) (A-S-O $291+380=671$) | A S | R | C P S |
| (A-S-R-P $317+98=415$) (A-S-F $239+178=417$) (A-T $118+329=447$) (A-Z $75+374=449$) (A-S-R- | A R S | P | B C R |

| | | | |
|--|-----------|---|-----|
| C 366+160=526) (A-S-R-S 300+253=553) (A-S-A 280+366=646) (A-S-O 291+380=671) | | | |
| (A-S-F 239+178=417)(A-S-R-P-B 418+0=418)(A-T 118+329=447)(A-Z 75+374=449)(A-S-R-C 366+160=526)(A-S-R-S 300+253=553)(A-S-R-P-R 414+193=607)(A-S-R-P-C 455+160=615)(A-S-A 280+366=646)(A-S-O 291+380=671) | A P R S | F | B S |
| (A-S-R-P-B 418+0=418) (A-T 118+329=447) (A-Z 75+374=449) (A-S-F-B 450+0=450) (A-S-R-C 366+160=526) (A-S-R-S 300+253=553) (A-S-F-S 338+253=591) (A-S-R-P-R 414+193=607) (A-S-R-P-C 455+160=615) (A-S-A 280+366=646) (A-S-O 291+380=671) | A F P R S | B | |

Διαδρομή και κόστος: A-S-R-P-B με μήκος διαδρομής 418

Κωδικοποίηση δεδομένων προβλήματος

Τα δεδομένα του προβλήματος είναι όμοια με την εργαστηριακή άσκηση 1 με την επιπλέον προσθήκη ότι στο τέλος του αρχείου υπάρχει ένα τμήμα δεδομένων (το τμήμα [STRAIGHT_LINE_DISTANCE_TO: ?]) που καταγράφει την απόσταση κάθε πόλης από την πόλη προορισμό.

| | | |
|--|--|--|
| # tour romania example [VERTICES:20] A,Arad B,Bucharest C,Craiova D,Drobeta E,Eforie F,Fagaras G,Giurgiu H,Hirsova I,Iasi L,Lugoj M,Mehadia N,Neamt O,Oradea P,Pitesti R,Rimnii Vilcea S,Sibiu T,Timisoara U,Urziceni V,Vaslui Z,Zerind [EDGES:46] A,S,140 A,T,118 A,Z,75 B,F,211 B,G,90 B,P,101 B,U,85 | C,D,120 C,P,138 C,R,146 D,C,120 D,M,75 E,H,86 F,B,211 F,S,99 G,B,90 H,E,86 H,U,98 I,N,87 I,V,92 L,M,70 L,T,111 M,D,75 M,L,70 N,I,87 O,S,151 O,Z,71 P,B,101 P,C,138 P,R,97 R,C,146 R,P,97 R,S,80 S,A,140 S,F,99 S,O,151 S,R,80 | T,A,118 T,L,111 U,B,85 U,H,98 U,V,142 V,I,92 V,U,142 Z,A,75 Z,O,71 [STRAIGHT_LINE_DISTANCE_TO:B] A,366 B,0 C,160 D,242 E,161 F,178 G,77 H,151 I,226 L,244 M,241 N,234 O,380 P,98 R,193 S,253 T,329 U,80 V,199 Z,374 |
|--|--|--|

Υλοποίηση κωδικοποίησης γραφήματος σε C++

Στη συνέχεια παρουσιάζεται ο κώδικας που ορίζει τη δομή του γραφήματος και την ανάγνωση των δεδομένων του γραφήματος συμπεριλαμβανομένων των αποστάσεων για όλους τους κόμβους σε ευθεία γραμμή από τον κόμβο προορισμό. Η υλοποίηση έχει οργανωθεί στα ακόλουθα αρχεία:

- lab02_graph.hpp
- lab02_graph.cpp
- lab02_02.cpp

```
#include <iostream>
#include <list>
```

```

#include <sstream>
#include <fstream>

using namespace std;

// δομή γραφήματος
struct di_graph {
    int V;           // αριθμός κορυφών γραφήματος
    int E;           // αριθμός ακμών γραφήματος
    string *vertices; // συντομογραφίες ονομάτων κορυφών
    string *vertices_full_names; // πλήρη ονόματα κορυφών
    int **adjacency_matrix; // πίνακας γειτνίασης
    string goal_vertex{}; // κορυφή στόχος
    int *distances_to_goal_vertex; // αποστάσεις σε ευθεία γραμμή από την κορυφή
                                // στόχο
};

// επιστροφή του ονόματος της κορυφής με δεδομένη τη θέση της κορυφής
string get_vertex_label(struct di_graph graph, int vertex_index);

// επιστροφή της θέσης της κορυφής με δεδομένο το όνομά της
int get_vertex_index(struct di_graph graph, string vertex);

// επιστροφή του μήκους της ακμής ανάμεσα σε 2 κορυφές
int get_weight(di_graph graph, string source_vertex, string destination_vertex);

// επιστροφή λίστας με τις γειτονικές κορυφές μιας κορυφής του γραφήματος
// ταξινομημένες σε αύξουσα ή σε φθίνουσα αλφαβητική σειρά
list<string> get_successors(struct di_graph graph, string node,
                           bool asc_order = true);

// επιστροφή της απόστασης σε ευθεία γραμμή μιας κορυφής από τον προορισμό
int get_heuristic(struct di_graph graph, string node);

// εκτύπωση της πληροφορίας του γραφήματος
void print_graph_info(struct di_graph graph);

// ανάγνωση γραφήματος από αρχείο κειμένου
struct di_graph read_data(string fn);

// απελευθέρωση της μνήμης που καταλαμβάνει το γράφημα
void free_memory(struct di_graph graph);
lab02_graph.hpp

```

```

#include "lab02_graph.hpp"

string get_vertex_label(struct di_graph graph, int vertex_index) {
    return graph.vertices[vertex_index];
}

int get_vertex_index(struct di_graph graph, string vertex) {
    int vertex_index{-1};

```

```

for (int i = 0; i < graph.V; i++)
    if (graph.vertices[i].compare(vertex) == 0) {
        vertex_index = i;
        break;
    }
return vertex_index;
}

int get_weight(di_graph graph, string source_vertex,
              string destination_vertex) {
    return graph.adjacency_matrix[get_vertex_index(
        graph, source_vertex)][get_vertex_index(graph, destination_vertex)];
}

list<string> get_successors(struct di_graph graph, string node,
                          bool asc_order) {
    list<string> successors{};
    int node_index = get_vertex_index(graph, node);
    for (int j = 0; j < graph.V; j++)
        if (graph.adjacency_matrix[node_index][j] != 0)
            successors.push_back(get_vertex_label(graph, j));
    successors.sort();
    if (!asc_order)
        successors.reverse();
    return successors;
}

int get_heuristic(struct di_graph graph, string node) {
    int node_index = get_vertex_index(graph, node);
    return graph.distances_to_goal_vertex[node_index];
}

void print_graph_info(struct di_graph graph) {
    cout << "Vertices=" << graph.V << endl;
    for (int i = 0; i < graph.V; i++)
        cout << "Vertex " << graph.vertices_full_names[i] << "("
             << graph.vertices[i] << ")" << endl;
    cout << "Edges=" << graph.E << endl;
    for (int i = 0; i < graph.V; i++)
        for (int j = 0; j < graph.V; j++)
            if (graph.adjacency_matrix[i][j] != 0)
                cout << graph.vertices_full_names[i] << "(" << graph.vertices[i]
                     << "--" << graph.adjacency_matrix[i][j] << "-->"
                     << graph.vertices_full_names[j] << "(" << graph.vertices[j] << ")"
                     << endl;
    for (int i = 0; i < graph.V; i++)
        cout << "Straight line distance from " << graph.vertices[i] << " to "
             << graph.goal_vertex << " is " << graph.distances_to_goal_vertex[i]
             << endl;
    cout << "Destination vertex = " << graph.goal_vertex << endl;
}

struct di_graph read_data(string fn) {

```

```

struct di_graph graph {};
fstream filestr{};
filestr.open(fn.c_str());
if (filestr.is_open()) {
    string vertices_header{};
    getline(filestr, vertices_header);
    // αγνόησε τις αρχικές γραμμές που ξεκινάνε με τον χαρακτήρα #
    while (vertices_header.at(0) == '#')
        getline(filestr, vertices_header);
    graph.V = stoi(vertices_header.substr(vertices_header.find(":") + 1,
                                           vertices_header.length() - 1));

    graph.vertices = new string[graph.V]{};
    graph.vertices_full_names = new string[graph.V]{};
    for (int i = 0; i < graph.V; i++) {
        string vertex{};
        getline(filestr, vertex);
        // trim κενών χαρακτήρων δεξιά του λεκτικού vertex
        vertex.erase(vertex.find_last_not_of(" \n\r\t") + 1);
        int pos = vertex.find(",");
        if (pos == -1) {
            // αν δεν υπάρχει δεύτερο όνομα τότε χρησιμοποιείται η συντομογραφία
            graph.vertices[i] = vertex.substr(0, vertex.length());
            graph.vertices_full_names[i] = vertex.substr(0, vertex.length());
        } else {
            graph.vertices[i] = vertex.substr(0, pos);
            graph.vertices_full_names[i] =
                vertex.substr(pos + 1, vertex.length() - pos - 1);
        }
    }
    string edges_header{};
    getline(filestr, edges_header);
    graph.E = stoi(edges_header.substr(edges_header.find(":") + 1,
                                       edges_header.length() - 1));

    graph.adjacency_matrix = new int *[graph.V]{};
    for (int i = 0; i < graph.V; i++)
        graph.adjacency_matrix[i] = new int[graph.V]{};
    for (int i = 0; i < graph.E; i++) {
        string edge{};
        getline(filestr, edge);
        edge.erase(edge.find_last_not_of(" \n\r\t") + 1);
        int pos1 = edge.find(",");
        int pos2 = edge.find(",", pos1 + 1);
        string source_vertex = edge.substr(0, pos1);
        string destination_vertex = edge.substr(pos1 + 1, pos2 - pos1 - 1);
        int weight = stoi(edge.substr(pos2 + 1, edge.length() - pos2));
        int source_vertex_index = get_vertex_index(graph, source_vertex);
        int destination_vertex_index =
            get_vertex_index(graph, destination_vertex);
        if ((source_vertex_index >= 0) && (source_vertex_index < graph.V))
            graph.adjacency_matrix[source_vertex_index][destination_vertex_index] =
                weight;
        else {
            cerr << "Edge data problem" << endl;
        }
    }
}

```



```

        exit(-1);
    }
}
string straight_line_distance_header{};
getline(filestr, straight_line_distance_header);
int pos1 = straight_line_distance_header.find(":");
int pos2 = straight_line_distance_header.find("]");
int goal_vertex_name_length = pos2 - pos1 - 1;
graph.goal_vertex =
    straight_line_distance_header.substr(pos1 + 1, goal_vertex_name_length);
graph.distances_to_goal_vertex = new int[graph.V]{};
for (int i = 0; i < graph.V; i++) {
    string straight_line_distance{};
    getline(filestr, straight_line_distance);
    pos2 = straight_line_distance.find(",");
    string from_vertex = straight_line_distance.substr(0, pos2);
    int from_vertex_index = get_vertex_index(graph, from_vertex);
    int distance = stoi(straight_line_distance.substr(
        pos2 + 1, straight_line_distance.length() - 1));
    graph.distances_to_goal_vertex[from_vertex_index] = distance;
}
} else {
    cout << "Error opening file: " << fn << endl;
    exit(-1);
}
return graph;
}

void free_memory(struct di_graph graph) {
    delete[] graph.vertices;
    delete[] graph.vertices_full_names;
    for (int i = 0; i < graph.V; i++)
        delete[] graph.adjacency_matrix[i];
    delete[] graph.adjacency_matrix;
    delete[] graph.distances_to_goal_vertex;
}

```

lab02_graph.cpp

```

#include "lab02_graph.hpp"

int main(int argc, char **argv) {
    struct di_graph graph { };
    if (argc != 2) {
        cout << "Wrong number of arguments" << endl;
        exit(-1);
    }
    string fn{argv[1]};
    graph = read_data(fn);
    print_graph_info(graph);
    free_memory(graph);
}

```

lab02_01.cpp

Ο κώδικας μεταγλωττίζεται με την εντολή:

```
g++ lab02_graph.cpp lab02_01.cpp -o lab02_01 -Wall -std=c++11
```

Η εκτέλεση και τα αποτελέσματα της εκτέλεσης του κώδικα παρουσιάζονται στη συνέχεια:

```
./lab02_01 data/tour_romania_h.txt
Vertices=20
Vertex Arad(A)
Vertex Bucharest(B)
...
Vertex Zerind(Z)
Edges=46
Arad(A)--140-->Sibiu(S)
Arad(A)--118-->Timisoara(T)
...
Zerind(Z)--71-->Oradea(O)
Straight line distance from A to B is 366
Straight line distance from B to B is 0
...
Straight line distance from Z to B is 374
```

Υλοποίηση των αλγορίθμων HC, BestFS και A*

Ο ακόλουθος κώδικας υλοποιεί τους αλγορίθμους αναρρίχησης λόφου, αναζήτησης πρώτα στο καλύτερο και A*. Η υλοποίηση έχει οργανωθεί στα ακόλουθα αρχεία:

- lab02_graph.hpp
- lab02_graph.cpp
- lab02_search.hpp
- lab02_search.cpp
- lab02_02.cpp

```
#include "lab02_graph.hpp"
#include <set>
#include <stack>
#include <queue>

struct search_node {
    list<string> path;
    int cost = 0;
    int heuristic = 0;
    // για την ταξινόμηση των κόμβων σε φθίνουσα σειρά κόστους
    bool operator<(search_node other) const {
        return cost + heuristic > other.cost + other.heuristic;
    }
};

// επιστροφή περιεχομένων ουράς προτεραιότητας ως λίστα
list<search_node> priority_queue_to_list(priority_queue<search_node> frontier);

// pretty print ενός search_node (bestfs)
string search_node_bestfs_as_string(search_node sn);

// pretty print ενός search_node (A*)
string search_node_astar_as_string(search_node sn);
```

```

// διαδρομή με πλήρη ονόματα κορυφών και κόστος διαδρομής
string solution_path_cost(di_graph graph, search_node sn);

// διαδρομή με πλήρη ονόματα κορυφών και υπολογισμός κόστους διαδρομής
string solution_path_compute_cost(di_graph graph, search_node sn);

// pretty print λίστας
string list_as_string(list<string> alist);

// pretty print συνόλου
string set_as_string(set<string> aset);

void print_status_astar(set<string> closed, list<search_node> frontier,
                        string current_state, list<string> successors);

void print_status_bestfs(set<string> closed, list<search_node> frontier,
                          string current_state, list<string> successors);

void print_status_hc(di_graph graph, string current_state,
                     list<string> successors);

// χρησιμοποιείται μόνο για τον κόμβο αφητηρία (start_node)
// έτσι ώστε να αρχικοποιήσει τη διαδρομή
search_node to_search_node(string node, int heuristic = 0);

// προσθήκη ενός επιπλέον κόμβου στη διαδρομή και ενημέρωση κόστους διαδρομής
search_node to_search_node(di_graph graph, search_node parent_sn, string node,
                           int heuristic = 0);

// προσθήκη ενός επιπλέον κόμβου στη διαδρομή
search_node to_bestfs_search_node(di_graph graph, search_node parent_sn,
                                  string node);

// αναρρίχηση λόφου
void hill_climbing(struct di_graph graph, string start_vertex,
                  string goal_vertex);

// αναζήτηση πρώτα στο καλύτερο
void best_first_search(struct di_graph graph, string start_vertex,
                      string goal_vertex);

// A* (Άλφα Άστρο)
void alpha_star_search(struct di_graph graph, string start_vertex,
                      string goal_vertex);

```

lab02_search.hpp

```

#include "lab02_search.hpp"

list<search_node> priority_queue_to_list(priority_queue<search_node> frontier) {
    list<search_node> alist{};
    while (!frontier.empty()) {

```

```

    alist.push_back(frontier.top());
    frontier.pop();
}
return alist;
}

string search_node_bestfs_as_string(search_node sn) {
    string s{};
    s.append("(");
    for (string v : sn.path) {
        s.append(v);
        s.append("-");
    }
    if (s.length() > 1)
        s.pop_back();
    s.append(" ");
    s.append(to_string(sn.heuristic));
    s.append(")");
    return s;
}

string search_node_astar_as_string(search_node sn) {
    string s{};
    s.append("(");
    for (string v : sn.path) {
        s.append(v);
        s.append("-");
    }
    if (s.length() > 1)
        s.pop_back();
    s.append(" ");
    s.append(to_string(sn.cost));
    s.append("+");
    s.append(to_string(sn.heuristic));
    s.append("=");
    s.append(to_string(sn.cost + sn.heuristic));
    s.append(")");
    return s;
}

string solution_path_cost(di_graph graph, search_node sn) {
    string s{};
    s.append("(");
    for (string v : sn.path) {
        s.append(graph.vertices_full_names[get_vertex_index(graph, v)]);
        s.append("-");
    }
    if (s.length() > 1)
        s.pop_back();
    s.append(" ");
    s.append(to_string(sn.cost));
    s.append(")");
    return s;
}

```

```

}

string solution_path_compute_cost(di_graph graph, search_node sn) {
    string s{};
    s.append("(");
    string pv = "";
    sn.cost = 0;
    for (string v : sn.path) {
        s.append(graph.vertices_full_names[get_vertex_index(graph, v)]);
        s.append("-");
        if (pv.compare("") != 0)
            sn.cost += get_weight(graph, pv, v);
        pv = v;
    }
    if (s.length() > 1)
        s.pop_back();
    s.append(" ");
    s.append(to_string(sn.cost));
    s.append(")");
    return s;
}

string list_as_string(list<string> alist) {
    string s{};
    s.append("[");
    for (string v : alist) {
        s.append(v);
        s.append(" ");
    }
    if (s.length() > 1)
        s.pop_back();
    s.append("]");
    return s;
}

string set_as_string(set<string> aset) {
    string s{};
    s.append("[");
    for (string v : aset) {
        s.append(v);
        s.append(" ");
    }
    if (s.length() > 1)
        s.pop_back();
    s.append("]");
    return s;
}

void print_status_astar(set<string> closed, list<search_node> frontier,
                       string current_state, list<string> successors) {
    cout << "frontier:[";
    for (search_node sn : frontier)
        cout << search_node_astar_as_string(sn);
}

```

```

cout << "]";
cout << " closed set:" << set_as_string(closed);
cout << " current node:" << current_state;
bool is_in{closed.find(current_state) != closed.end()};
if (is_in)
    cout << " successors:[loop]" << endl;
else {
    cout << " successors:" << list_as_string(successors) << endl;
}
}

void print_status_bestfs(set<string> closed, list<search_node> frontier,
                        string current_state, list<string> successors) {
    cout << "frontier:[";
    for (search_node sn : frontier)
        cout << search_node_bestfs_as_string(sn);
    cout << "]";
    cout << " closed set:" << set_as_string(closed);
    cout << " current node:" << current_state;
    bool is_in{closed.find(current_state) != closed.end()};
    if (is_in)
        cout << " successors:[loop]" << endl;
    else {
        cout << " successors:" << list_as_string(successors) << endl;
    }
}

void print_status_hc(di_graph graph, string current_state,
                    list<string> successors) {
    string s{};
    s.append("current node:");
    s.append(current_state);
    s.append(" ");
    s.append(to_string(get_heuristic(graph, current_state)));
    s.append(" successors:");
    for (string v : successors) {
        s.append("(");
        s.append(v);
        s.append(" ");
        s.append(to_string(get_heuristic(graph, v)));
        s.append(")");
    }
    cout << s << endl;
}

search_node to_search_node(string node, int heuristic) {
    search_node sn{};
    sn.path.push_back(node);
    sn.cost = 0;
    sn.heuristic = heuristic;
    return sn;
}

```

```

search_node to_search_node(di_graph graph, search_node parent_sn, string node,
                           int heuristic) {
    search_node sn{};
    for (string v : parent_sn.path) {
        sn.path.push_back(v);
    }
    sn.path.push_back(node);
    sn.cost = parent_sn.cost + get_weight(graph, parent_sn.path.back(), node);
    sn.heuristic = heuristic;
    return sn;
}

search_node to_bestfs_search_node(di_graph graph, search_node parent_sn,
                                  string node) {
    search_node sn{};
    for (string v : parent_sn.path) {
        sn.path.push_back(v);
    }
    sn.path.push_back(node);
    sn.cost = 0;
    sn.heuristic = get_heuristic(graph, node);
    return sn;
}

void hill_climbing(struct di_graph graph, string start_vertex,
                   string goal_vertex) {
    cout << "Hill Climbing" << endl;
    int best_cost = get_heuristic(graph, start_vertex);
    search_node current_state =
        to_search_node(start_vertex, get_heuristic(graph, start_vertex));
    string current_state_back = start_vertex;
    bool deadlock{false};
    while (goal_vertex.compare(current_state_back) != 0) {
        print_status_hc(graph, current_state_back,
                        get_successors(graph, current_state_back));
        deadlock = true;
        string best_v{};
        for (string v : get_successors(graph, current_state_back)) {
            if (get_heuristic(graph, v) < best_cost) {
                best_cost = get_heuristic(graph, v);
                best_v = v;
                deadlock = false;
            }
        }
        if (deadlock)
            break;
        else {
            current_state = to_search_node(graph, current_state, best_v);
            current_state_back = current_state.path.back();
        }
    }
    if (!deadlock) {
        print_status_hc(graph, current_state_back,

```

```

        get_successors(graph, current_state_back));
    cout << "Path to goal node found: "
        << solution_path_cost(graph, current_state) << endl;
} else
    cout << "Goal not found!" << endl;
}

void best_first_search(struct di_graph graph, string start_vertex,
                      string goal_vertex) {
    cout << "BestFS" << endl;
    set<string> closed{};
    priority_queue<search_node> frontier{}; // MIN HEAP
    frontier.push(
        to_search_node(start_vertex, get_heuristic(graph, start_vertex)));
    search_node current_state = frontier.top();
    string current_state_back = current_state.path.back();
    bool found{true};
    while (goal_vertex.compare(current_state_back) != 0) {
        print_status_bestfs(closed, priority_queue_to_list(frontier),
                           current_state_back,
                           get_successors(graph, current_state_back));

        frontier.pop();
        bool is_in{closed.find(current_state_back) != closed.end()};
        if (!is_in) {
            for (string v : get_successors(graph, current_state_back))
                frontier.push(to_bestfs_search_node(graph, current_state, v));
            closed.insert(current_state_back);
        }
        if (frontier.empty()) {
            found = false;
            break;
        }
        current_state = frontier.top();
        current_state_back = current_state.path.back();
    }
    if (found) {
        print_status_bestfs(closed, priority_queue_to_list(frontier),
                           current_state_back,
                           get_successors(graph, current_state_back));
        cout << "Path to goal node found: "
            << solution_path_compute_cost(graph, current_state) << endl;
    } else
        cout << "Goal not found!" << endl;
}

void alpha_star_search(struct di_graph graph, string start_vertex,
                       string goal_vertex) {
    cout << "A*" << endl;
    set<string> closed{};
    priority_queue<search_node> frontier{}; // MIN HEAP
    frontier.push(
        to_search_node(start_vertex, get_heuristic(graph, start_vertex)));
    search_node current_state = frontier.top();

```



```

string current_state_back = current_state.path.back();
bool found{true};
while (goal_vertex.compare(current_state_back) != 0) {
    print_status_astar(closed, priority_queue_to_list(frontier),
                      current_state_back,
                      get_successors(graph, current_state_back));

    frontier.pop();
    bool is_in{closed.find(current_state_back) != closed.end()};
    if (!is_in) {
        for (string v : get_successors(graph, current_state_back))
            frontier.push(
                to_search_node(graph, current_state, v, get_heuristic(graph, v)));
        closed.insert(current_state_back);
    }
    if (frontier.empty()) {
        found = false;
        break;
    }
    current_state = frontier.top();
    current_state_back = current_state.path.back();
}
if (found) {
    print_status_astar(closed, priority_queue_to_list(frontier),
                      current_state_back,
                      get_successors(graph, current_state_back));
    cout << "Path to goal node found: "
         << solution_path_cost(graph, current_state) << endl;
} else
    cout << "Goal not found!" << endl;
}

```

lab02_search.cpp

```

#include "lab02_search.hpp"
#include <stdlib.h>

int main(int argc, char **argv) {
    struct di_graph graph { };
    string fn{};
    string start_vertex{}, search_method{};
    if (argc != 4) {
        printf("Wrong number of arguments\n");
        printf("Usage   : %s <problem_instance> <source> <algorithm> \n", argv[0]);
        printf("Example : %s tour_romania_h.txt A ASTAR \n", argv[0]);
        exit(-1);
    }
    fn = argv[1];
    start_vertex = argv[2];
    search_method = argv[3];
    graph = read_data(fn);
    cout << "Origin: " << start_vertex << " destination: " << graph.goal_vertex << endl;
    if (search_method.compare("HC") == 0)
        hill_climbing(graph, start_vertex, graph.goal_vertex);
}

```

```

else if (search_method.compare("BESTFS") == 0)
    best_first_search(graph, start_vertex, graph.goal_vertex);
else if (search_method.compare("ASTAR") == 0)
    alpha_star_search(graph, start_vertex, graph.goal_vertex);
else
    cerr << "invalid option" << endl;
free_memory(graph);
}

```

lab02_02.cpp

Ο κώδικας μεταγλωττίζεται με την εντολή:

```
g++ lab02_graph.cpp lab02_search.cpp lab02_02.cpp -o lab02_02 -Wall -std=c++11
```

Η εκτέλεση και τα αποτελέσματα της εκτέλεσης του κώδικα παρουσιάζονται στη συνέχεια:

```
./lab02_02 data/tour_romania_h.txt A HC
```

```

Origin: A destination: B
Hill Climbing
current node:(A 366) successors:(S 253)(T 329)(Z 374)
current node:(S 253) successors:(A 366)(F 178)(O 380)(R 193)
current node:(F 178) successors:(B 0)(S 253)
current node:(B 0) successors:(F 178)(G 77)(P 98)(U 80)
Path to goal node found: (Arad-Sibiu-Fagaras-Bucharest 450)

```

```
./lab02_02 data/tour_romania_h.txt A BESTFS
```

```

Origin: A destination: B
BestFS
frontier:[(A 366)] closed set:[] current node:A successors:[S T Z]
frontier:[(A-S 253)(A-T 329)(A-Z 374)] closed set:[A] current node:S successors:[A F O R]
frontier:[(A-S-F 178)(A-S-R 193)(A-T 329)(A-S-A 366)(A-Z 374)(A-S-O 380)] closed set:[A S]
current node:F successors:[B S]
frontier:[(A-S-F-B 0)(A-S-R 193)(A-S-F-S 253)(A-T 329)(A-S-A 366)(A-Z 374)(A-S-O 380)] closed
set:[A F S] current node:B successors:[F G P U]
Path to goal node found: (Arad-Sibiu-Fagaras-Bucharest 450)

```

```
./lab02_02 data/tour_romania_h.txt A ASTAR
```

```

Origin: A destination: B
A*
frontier:[(A 0+366=366)] closed set:[] current node:A successors:[S T Z]
frontier:[(A-S 140+253=393)(A-T 118+329=447)(A-Z 75+374=449)] closed set:[A] current node:S
successors:[A F O R]
frontier:[(A-S-R 220+193=413)(A-S-F 239+178=417)(A-T 118+329=447)(A-Z 75+374=449)(A-S-A
280+366=646)(A-S-O 291+380=671)] closed set:[A S] current node:R successors:[C P S]
frontier:[(A-S-R-P 317+98=415)(A-S-F 239+178=417)(A-T 118+329=447)(A-Z 75+374=449)(A-S-R-C
366+160=526)(A-S-R-S 300+253=553)(A-S-A 280+366=646)(A-S-O 291+380=671)] closed set:[A R S]
current node:P successors:[B C R]
frontier:[(A-S-F 239+178=417)(A-S-R-P-B 418+0=418)(A-T 118+329=447)(A-Z 75+374=449)(A-S-R-C
366+160=526)(A-S-R-S 300+253=553)(A-S-R-P-R 414+193=607)(A-S-R-P-C 455+160=615)(A-S-A
280+366=646)(A-S-O 291+380=671)] closed set:[A P R S] current node:F successors:[B S]
frontier:[(A-S-R-P-B 418+0=418)(A-T 118+329=447)(A-Z 75+374=449)(A-S-F-B 450+0=450)(A-S-R-C
366+160=526)(A-S-R-S 300+253=553)(A-S-F-S 338+253=591)(A-S-R-P-R 414+193=607)(A-S-R-P-C
455+160=615)(A-S-A 280+366=646)(A-S-O 291+380=671)] closed set:[A F P R S] current node:B
successors:[F G P U]
Path to goal node found: (Arad-Sibiu-Rimni Vilcea-Pitesti-Bucharest 418)

```

Αναφορές

1. [Amit] Amit's A* Pages <http://theory.stanford.edu/~amitp/GameProgramming/>
2. [Norvig03] Τεχνητή Νοημοσύνη – μια σύγχρονη προσέγγιση, Β' έκδοση, Stuart Russell, Peter Norvig, Εκδόσεις Κλειδάριθμος, 2003.
3. [Βλαχαβάς06] Τεχνητή Νοημοσύνη, Γ' έκδοση, Ι. Βλαχαβάς, Π. Κεφαλάς, Ν. Βασιλειάδης, Φ. Κόκορας, Η. Σακελλαρίου, Γκιούρδας Εκδοτική, 2006.

Σύνδεσμοι οπτικής απεικόνισης αλγορίθμων

1. <http://qiao.github.io/PathFinding.js/visual/>
2. <http://aispace.org/search/>