 Πανεπιστήμιο Ιωαννίνων	Τμήμα Πληροφορικής & Τηλεπικοινωνιών Διδάσκων: Στύλιος Χρυσόστομος Επιμέλεια σημειώσεων: Γκόγκος Χρήστος Μάθημα: Τεχνητή Νοημοσύνη (εργαστήριο Ε' εξαμήνου)	Ακαδημαϊκό έτος 2021-2022 Χειμερινό Εξάμηνο	1

Αναζήτηση χωρίς πληροφόρηση

Εισαγωγή

Οι αλγόριθμοι αναζήτησης σε συνδυασμό με την αναπαράσταση γνώσης αποτελούν τον πυρήνα των εφαρμογών Τεχνητής Νοημοσύνης (Norvig03, Βλαχαβάς06, Ertel11). Θεωρώντας κατάλληλες περιγραφές του προβλήματος, αναζητούν τη λύση του εξετάζοντας με κάποια σειρά τις διαφορετικές καταστάσεις στις οποίες μπορεί να περιέλθει ο κόσμος του προβλήματος. Οι αλγόριθμοι αναζήτησης μπορεί να διαφέρουν σε χαρακτηριστικά όπως η χρονική πολυπλοκότητα, η χωρική πολυπλοκότητα, η πληρότητα (δηλαδή η εγγύηση της εύρεσης λύσης εφόσον υπάρχει) καθώς και σε άλλα χαρακτηριστικά. Υπάρχει πληθώρα αλγορίθμων αναζήτησης και δεν υπάρχει κάποιος αλγόριθμος αναζήτησης που να υπερτερεί έναντι όλων των άλλων σε όλες τις περιπτώσεις.

Σε γενικές γραμμές οι αλγόριθμοι αναζήτησης λειτουργούν διατηρώντας μια λίστα από καταστάσεις του προβλήματος οι οποίες πρόκειται να εξεταστούν και η οποία ονομάζεται **μέτωπο αναζήτησης** (frontier ή fringe). Οι καταστάσεις του μετώπου αναζήτησης ελέγχονται με κάποια σειρά σχετικά με το εάν αποτελούν λύση του προβλήματος και πραγματοποιούνται **επεκτάσεις καταστάσεων** (expansions) κατά τις οποίες προστίθενται στο μέτωπο αναζήτησης καταστάσεις (successor states) που προκύπτουν από την τρέχουσα κατάσταση μέσω κάποιων τελεστών μετάβασης (transition operators). Για να αποφευχθεί η δημιουργία βρόχων αναζήτησης (loops) διατηρείται το **κλειστό σύνολο** (closed set) που αποτελείται από τις καταστάσεις οι οποίες έχουν ήδη επεκταθεί. Εφόσον χρησιμοποιείται κλειστό σύνολο μια κατάσταση πριν επεκταθεί εξετάζεται εάν ανήκει στο κλειστό σύνολο. Αν αυτό συμβαίνει τότε δεν επεκτείνεται ξανά.

Στα πλαίσια του παρόντος εργαστηρίου θα παρουσιαστούν οι αλγόριθμοι:

1. Αναζήτηση πρώτα κατά πλάτος (Breadth First Search)
2. Αναζήτηση πρώτα κατά βάθος (Depth First Search)
3. Αναζήτηση ομοιόμορφου κόστους (Uniform Cost Search)

Οι ανωτέρω αλγόριθμοι ανήκουν στην κατηγορία των **αλγορίθμων αναζήτησης χωρίς πληροφόρηση** (ή αλγορίθμων τυφλής αναζήτησης) δηλαδή δεδομένης μιας κατάστασης που δεν είναι κατάσταση στόχου δεν υπάρχει πληροφόρηση σχετικά με το εάν η κατάσταση βρίσκεται κοντά ή μακριά από το στόχο.

Αναζήτηση πρώτα κατά πλάτος (BFS=Breadth First Search)

Στην αναζήτηση πρώτα κατά πλάτος οι καταστάσεις που επεκτείνονται πρώτες είναι εκείνες που έχουν προστεθεί νωρίτερα στο μέτωπο αναζήτησης. Συνεπώς, μπορεί να χρησιμοποιεί η δομή δεδομένων **ουρά** (queue) για τη διαχείριση του μετώπου αναζήτησης. Αν η τρέχουσα κατάσταση δεν ανήκει στο κλειστό σύνολο η εισαγωγή των γειτονικών της καταστάσεων γίνεται στο πίσω άκρο του μετώπου αναζήτησης. Στη συνέχεια ακολουθεί ο ψευδοκώδικας του αλγορίθμου και ο κώδικας σε C++ για την αναζήτηση πρώτα κατά πλάτος.

```
algorithm bfs(InitialState, FinalStates)
begin
Closed←∅;
Frontier←<InitialState>;
CurrentState←First(Frontier);
while CurrentState ∉ FinalStates do
    Frontier←delete(CurrentState,Frontier);
    if CurrentState ∉ ClosedSet
    begin
      ChildrenStates ←Expand(CurrentState);
      Frontier← Frontier ^ ChildrenStates;
      Closed←Closed∪{CurrentState};
    end;
  if Frontier= ∅ then exit;
  CurrentState←First(Frontier);
endwhile;
return success;
end.
```

Ψευδοκώδικας για τον αλγόριθμο αναζήτησης πρώτα κατά πλάτος (Βλαχαβάς06)

```
bool bfs(struct di_graph graph, string start_vertex,
string        goal_vertex) {
 set<string> closed { };
 queue<string> frontier { }; // FIFO
 frontier.push(start_vertex);
 string current_state = frontier.front();
 while (goal_vertex.compare(current_state) != 0) {
  frontier.pop();
  if (closed.find(current_state) == closed.end()) {
   for (string v : get_successors(graph,
current_state))
    frontier.push(v);
   closed.insert(current_state);
  }
  if (frontier.empty())
   return false;
  current_state = frontier.front();
 }
 return true;
}
```

Κώδικας σε C++ για την αναζήτηση πρώτα κατά πλάτος (χωρίς δομές για την αποθήκευση της διαδρομής και του μήκους της)

Αναζήτηση πρώτα κατά βάθος (DFS=Depth First Search)

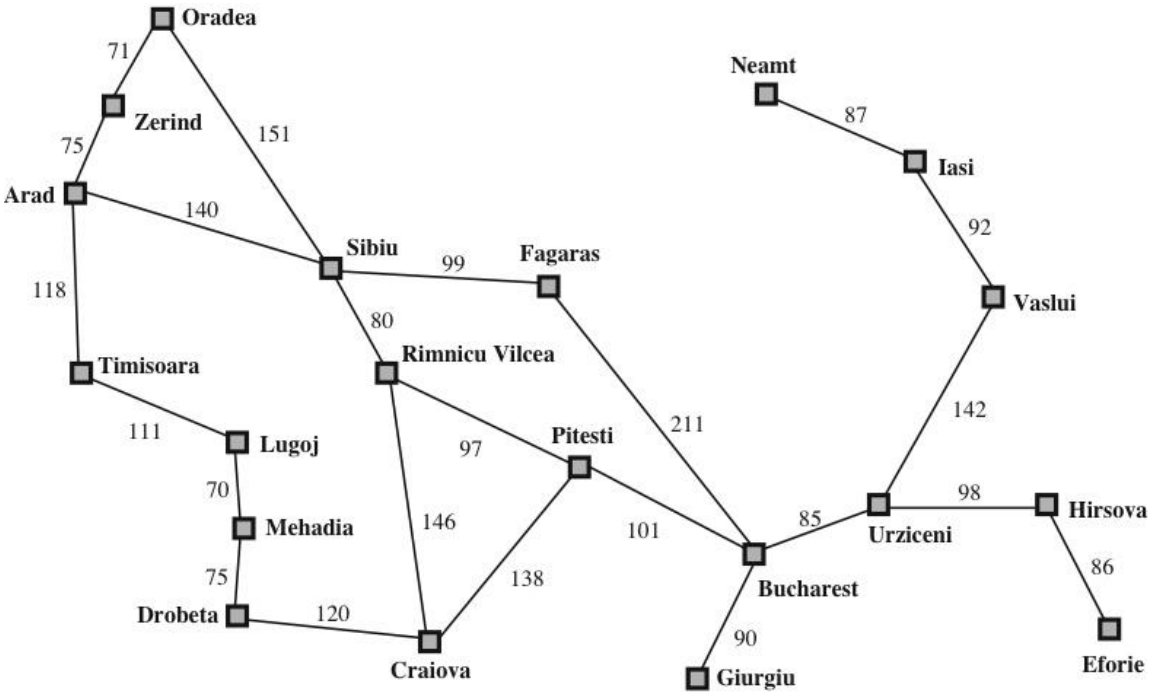
Στην αναζήτηση πρώτα κατά βάθος οι καταστάσεις που επεκτείνονται πρώτες είναι οι τελευταίες που προστέθηκαν στο μέτωπο αναζήτησης. Συνεπώς, είναι φυσικό να χρησιμοποιείται η δομή της **στοίβας** (stack) για τη διαχείριση του μετώπου αναζήτησης.

Αναζήτηση ομοιόμορφου κόστους (UCS=Uniform Cost Search)

Στην αναζήτηση ομοιόμορφου κόστους οι καταστάσεις που επεκτείνονται πρώτες είναι εκείνες που έχουν το χαμηλότερο κόστος για τη μετάβαση από την αρχική κατάσταση μέχρι αυτές. Για τη διαχείριση του μετώπου αναζήτησης μπορεί να χρησιμοποιηθεί η δομή δεδομένων **ουρά προτεραιότητας** (priority_queue).

Πρόβλημα εύρεσης διαδρομής σε ένα γράφημα πόλεων

Το πρόβλημα που θα χρησιμοποιηθεί για την επίδειξη των αλγορίθμων αναζήτησης είναι το πρόβλημα εύρεσης της διαδρομής από μια πόλη σε μια άλλη στον ακόλουθο χάρτη.



Εικόνα 1. Romania tour

Περιγράφοντας το πρόβλημα με χρήση του χώρου καταστάσεων θα πρέπει να οριστεί η ακόλουθη τετράδα (I,G,T,S):

- I: Αρχική κατάσταση
- G: Σύνολο τελικών καταστάσεων
- T: Τελεστές μετάβασης
- S: Χώρος καταστάσεων

Συνεπώς για το πρόβλημα εύρεσης της διαδρομής από μια πόλη σε μια άλλη πόλη ισχύει ότι:

- Αρχική κατάσταση I είναι η κατάσταση κατά την οποία ο ταξιδιώτης βρίσκεται στην πόλη αφετηρία και δεν έχει ξεκινήσει ακόμα τη διαδρομή του προς την πόλη προορισμό.
- Το σύνολο G αποτελείται από όλες τις καταστάσεις κατά τις οποίες ο ταξιδιώτης ξεκινώντας από την πόλη αφετηρία (π.χ. Arad) και ακολουθώντας μια διαδρομή που αποτελείται από διαδοχικές μεταβάσεις μεταξύ γειτονικών πόλεων καταλήγει να βρίσκεται στην πόλη προορισμό (π.χ. Bucharest).
- Τ είναι ο τελεστής μετάβασης του ταξιδιώτη από μια πόλη σε κάποια άλλη γειτονική πόλη σύμφωνα με τον χάρτη.
- Ο χώρος καταστάσεων S του προβλήματος αποτελείται από όλες τις πιθανές διαδρομές που μπορεί να προκύψουν κατά την αναζήτηση της διαδρομής από μια οποιαδήποτε πόλη προς κάποια άλλη πόλη. Ο δε χώρος αναζήτησης (search space) αποτελείται από όλες τις πιθανές διαδρομές που μπορεί να προκύψουν κατά την αναζήτηση της διαδρομής από την πόλη αφετηρία προς την πόλη προορισμό.

Κωδικοποίηση δεδομένων προβλήματος

Τα δεδομένα του προβλήματος βρίσκονται κωδικοποιημένα στο αρχείο tour_romania.txt το οποίο αποτελείται από 2 τμήματα. Το τμήμα καταγραφής των κορυφών (vertices) στο οποίο αποδίδεται ένα όνομα σε κάθε κορυφή του γραφήματος (το πρώτο γράμμα της αντίστοιχης πόλης) και το τμήμα καταγραφής των ακμών του γραφήματος που αποτελείται από κατευθυνόμενες ακμές της μορφής: από πόλη, προς πόλη, απόσταση (π.χ. A,S,140 σημαίνει ότι η απόσταση της διαδρομής από Arad προς Sibiu είναι 140 χιλιόμετρα)

```
# 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,Rimniu 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
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
```

C,P,138 C,R,146 D,C,120 D,M,75	V,U,142 Z,A,75 Z,O,71

Περιεχόμενα αρχείου tour_romania.txt

Για την περίπτωση του αρχείου tour_romania.txt ο πίνακας γειτνίασης του γραφήματος μετά την ανάγνωση από το αρχείο θα έχει τα ακόλουθα δεδομένα:

vertices	A	B	C	D	E	F	G	H	I	L	M	N	O	P	R	S	T	U	V	Z
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	140	118	0	0	75
B	0	0	0	0	0	211	90	0	0	0	0	0	0	101	0	0	0	85	0	0
C	0	0	0	120	0	0	0	0	0	0	0	0	0	138	146	0	0	0	0	0
D	0	0	120	0	0	0	0	0	0	0	75	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	86	0	0	0	0	0	0	0	0	0	0	0	0
F	0	211	0	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0
G	0	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	86	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0
I	0	0	0	0	0	0	0	0	0	0	0	87	0	0	0	0	0	0	92	0
L	0	0	0	0	0	0	0	0	0	0	70	0	0	0	0	0	111	0	0	0
M	0	0	0	75	0	0	0	0	0	70	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	87	0	0	0	0	0	0	0	0	0	0	0
O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	151	0	0	0	71
P	0	101	138	0	0	0	0	0	0	0	0	0	0	0	97	0	0	0	0	0
R	0	0	146	0	0	0	0	0	0	0	0	0	0	97	0	80	0	0	0	0
S	140	0	0	0	0	99	0	0	0	0	0	0	151	0	80	0	0	0	0	0
T	118	0	0	0	0	0	0	0	0	111	0	0	0	0	0	0	0	0	0	0
U	0	85	0	0	0	0	0	98	0	0	0	0	0	0	0	0	0	0	142	0
V	0	0	0	0	0	0	0	0	92	0	0	0	0	0	0	0	0	142	0	0
Z	75	0	0	0	0	0	0	0	0	0	0	71	0	0	0	0	0	0	0	0

adjacency_matrix

Υλοποίηση κωδικοποίησης γραφήματος σε C++

Στη συνέχεια παρουσιάζεται ο κώδικας που ορίζει τη δομή του γραφήματος και την ανάγνωση των δεδομένων ενός γραφήματος από ένα αρχείο κειμένου με τη μορφή που έχει περιγραφεί παραπάνω. Η υλοποίηση έχει οργανωθεί στα ακόλουθα αρχεία:

- lab01_graph.hpp
- lab01_graph.cpp
- lab01_01.cpp

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

```hpp
#include <cstring>

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
};

//  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 όνομα της κορυφής
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);

//  επισστροφή 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);

// 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);
```

lab01_graph.hpp

```hpp
#include "lab01_graph.hpp"
#include <sstream>
#include <fstream>

using namespace std;

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;
}

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;
}

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);
      }
    }
  } 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;
}
```

lab01_graph.cpp

```
#include "lab01_graph.hpp"

using namespace std;

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);
}
```

lab01_01.cpp

Ο κώδικας μεταγλωττίζεται με την εντολή:

```
g++ lab01_graph.cpp lab01_01.cpp -o lab01_01 -Wall -std=c++11
```

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

```
./lab01_01 data/tour_romania.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)
```

Υλοποίηση βασικών μορφών των αλγορίθμων BFS και DFS

Ο ακόλουθος κώδικας υλοποιεί την αναζήτηση πρώτα κατά πλάτος και την αναζήτηση πρώτα κατά βάθος χωρίς να διατηρεί τη διαδρομή που εντοπίζεται. Η υλοποίηση έχει οργανωθεί στα ακόλουθα αρχεία:

- lab01_graph.hpp
- lab01_graph.cpp
- lab01_search_simple.hpp
- lab01_search_simple.cpp
- lab01_02.cpp

```hpp
#include "lab01_graph.hpp"

// Ο αλγόριθμος αναζήτησης κατά πλάτος χωρίς αποθήκευση της διαδρομής
void breadth_first_search_base(struct di_graph graph, string start_vertex,
                               string goal_vertex);

// Ο αλγόριθμος αναζήτησης κατά βάθος χωρίς αποθήκευση της διαδρομής
void depth_first_search_base(struct di_graph graph, string start_vertex,
                             string goal_vertex);
```

lab01_search_simple.hpp

```cpp
#include "lab01_search_simple.hpp"
#include <set>
#include <queue>
#include <stack>
#include <unordered_map>

using namespace std;

void breadth_first_search_base(struct di_graph graph, string start_vertex,
                               string goal_vertex) {
  cout << "BREADTH FIRST SEARCH" << endl;
  unordered_map<string, int> hm;
  set<string> closed{};
  queue<string> frontier{}; // FIFO
  frontier.push(start_vertex);
  string current_state = frontier.front();
  hm[current_state] = 0;
  bool found{true};
  cout << "Starting from node " << start_vertex << endl;
  while (goal_vertex.compare(current_state) != 0) {
    frontier.pop();
    bool is_in{closed.find(current_state) != closed.end()};
    if (!is_in) {
      for (string v : get_successors(graph, current_state)) {
        frontier.push(v);
        hm[v] = hm[current_state] + get_weight(graph, current_state, v);
      }
      closed.insert(current_state);
    }
    if (frontier.empty()) {
      found = false;
      break;
    }
    current_state = frontier.front();
    cout << "current state: " << current_state << endl;
  }
  if (found)
    cout << "Path from " << start_vertex << " to " << goal_vertex
```

```
        << " found having length " << hm[goal_vertex] << endl;
  else
    cout << "Path from " << start_vertex << " to " << goal_vertex
         << " was not found!" << endl;
}

void depth_first_search_base(struct di_graph graph, string start_vertex,
                             string goal_vertex) {
  cout << "DEPTH FIRST SEARCH" << endl;
  unordered_map<string, int> hm;
  set<string> closed{};
  stack<string> frontier{}; // LIFO
  frontier.push(start_vertex);
  string current_state = frontier.top();
  hm[current_state] = 0;
  bool found{true};
  cout << "Starting from node " << start_vertex << endl;
  while (goal_vertex.compare(current_state) != 0) {
    frontier.pop();
    bool is_in{closed.find(current_state) != closed.end()};
    if (!is_in) {
      for (string v : get_successors(graph, current_state, false)) {
        frontier.push(v);
        hm[v] = hm[current_state] + get_weight(graph, current_state, v);
      }
      closed.insert(current_state);
    }
    if (frontier.empty()) {
      found = false;
      break;
    }
    current_state = frontier.top();
    cout << "current state: " << current_state << endl;
  }
  if (found)
    cout << "Path from " << start_vertex << " to " << goal_vertex
         << " found having length " << hm[goal_vertex] << endl;
  else
    cout << "Path from " << start_vertex << " to " << goal_vertex
         << " was not found!" << endl;
}
```

lab01_search_simple.cpp

```
#include "lab01_search_simple.hpp"

int main(int argc, char **argv) { struct di_graph graph { };
  string fn{};
  string start_vertex{}, goal_vertex{}, search_method{};
  if (argc == 5) {
    fn = argv[1];
    start_vertex = argv[2];
    goal_vertex = argv[3];
    search_method = argv[4];
    graph = read_data(fn);
    if (search_method.compare("bfs") == 0)
      breadth_first_search_base(graph, start_vertex, goal_vertex);
    else if (search_method.compare("dfs") == 0)
      depth_first_search_base(graph, start_vertex, goal_vertex);
    else
      cerr << "Invalid choice" << endl;
  }
  free_memory(graph);
}
```

lab01_02.cpp

Ο κώδικας μεταγλωττίζεται με την εντολή:

```
g++ lab01_graph.cpp lab01_search_simple.cpp lab01_02.cpp -o lab01_02 -std=c++11
```

Η εκτέλεση και τα αποτελέσματα της εκτέλεσης του κώδικα για τη μετάβαση από Craiova προς Bucharest στο πρόβλημα tour_romania.txt χρησιμοποιώντας τον αλγόριθμο αναζήτησης πρώτα κατά πλάτος παρουσιάζονται στη συνέχεια:

```
./lab01_02 data/tour_romania.txt C B bfs
BREADTH FIRST SEARCH
Starting from node C
current state: D
current state: P
current state: R
current state: C
current state: M
current state: B
Path to goal node B found with length 239
```

Υλοποίηση αλγορίθμων BFS, DFS και UCS με ταυτόχρονη αποθήκευση της διαδρομής από την αφετηρία προς τον προορισμό

Ο ακόλουθος κώδικας υλοποιεί την αναζήτηση πρώτα κατά πλάτος, την αναζήτηση πρώτα κατά βάθος και την ομοιόμορφη αναζήτηση. Η υλοποίηση έχει οργανωθεί στα ακόλουθα αρχεία:

- lab01_graph.hpp
- lab01_graph.cpp
- lab01_search.hpp
- lab01_search.cpp
- lab01_03.cpp

Σημαντικό ρόλο στην αποθήκευση των διαδρομών έχει η δομή search_node στην οποία εμπεριέχεται τόσο η διαδρομή (path) όσο και το κόστος της (cost).

```
#include "lab01_graph.hpp"
#include <set>
#include <stack>
#include <queue>

using namespace std;

struct search_node {
  list<string> path;
  int cost = 0;
  bool is_goal = false;
  bool operator<(search_node other) const { return cost > other.cost; }
};

// αναζήτηση πρώτα κατά βάθος
void depth_first_search(struct di_graph graph, string start_vertex,
                        string goal_vertex);

// αναζήτηση πρώτα κατά πλάτος
void breadth_first_search(struct di_graph graph, string start_vertex,
                          string goal_vertex);

// αναζήτηση ομοιόμορφου κόστους
void uniform_cost_search(struct di_graph graph, string start_vertex,
                         string goal_vertex);

// convenience function για μεταφορά περιεχομένων στοίβας σε λίστα
list<search_node> stack_to_list(stack<search_node> frontier);

// convenience function για μεταφορά περιεχομένων ουράς σε λίστα
```

```
list<search_node> queue_to_list(queue<search_node> frontier);

// convenience function για μεταφορά περιεχομένων ουράς προτεραιότητας σε λίστα
list<search_node> priority_queue_to_list(priority_queue<search_node> frontier);

// convenience function για pretty print ενός search_node
string search_node_as_string(search_node sn, bool show_path_cost = true);

//  επιστρέφει τη λύση μαζί με το κόστος της χρησιμοποιώντας τα πλήρη ονόματα
//  των κόμβων του γράφου
string solution_path_cost(di_graph graph, search_node sn);

// convenience function για pretty print λίστας με λεκτικά
string list_as_string(list<string> alist);

// convenience function για pretty print συνόλου με λεκτικά
string set_as_string(set<string> aset);

/**
 * χρησιμοποιείται για να εμφανίζει κατά τη διάρκεια της αναζήτησης
 * το μέτωπο αναζήτησης, το κλειστό σύνολο, τον τρέχοντα κόμβο και τους
 * γειτονικούς κόμβους του τρέχοντα κόμβου
 */
void print_status(set<string> closed, list<search_node> frontier,
                  string current_state, list<string> successors,
                  bool show_path_cost = false);

//  χρησιμοποιείται μόνο για τον κόμβο αφετηρία (start_node) έτσι ώστε να
//  αρχικοποιήσει τη διαδρομή
search_node to_search_node(string node);

// προσθήκη ενός επιπλέον κόμβου στη διαδρομή που έχει ήδη δημιουργηθεί,
// ενημέρωση κόστους
search_node to_search_node(di_graph graph, search_node parent_sn, string node);
```

lab01_search.hpp

```
#include "lab01_search.hpp"

using namespace std;

void depth_first_search(struct di_graph graph, string start_vertex,
                        string goal_vertex) {
  cout << "DEPTH FIRST SEARCH" << endl;
  set<string> closed{};
  stack<search_node> frontier{}; // LIFO
  frontier.push(to_search_node(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(closed, stack_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) {
      // Οι γειτονικοί κόμβοι λαμβάνονται σε φθίνουσα αλφαβητική σειρά έτσι ώστε
      // όταν τοποθετηθούν στη στοίβα η τιμή που βρίσκεται στη κορυφή να είναι η
      // μικρότερη αλφαβητικά από αυτές που εισήχθησαν τελευταίες
      for (string v : get_successors(graph, current_state_back, false))
        frontier.push(to_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(closed, stack_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;
}

void breadth_first_search(struct di_graph graph, string start_vertex,
                          string goal_vertex) {
  cout << "BREADTH FIRST SEARCH" << endl;
  set<string> closed{};
  queue<search_node> frontier{}; // FIFO
  frontier.push(to_search_node(start_vertex));
  search_node current_state = frontier.front();
  string current_state_back = current_state.path.back();
  bool found{true};
  while (goal_vertex.compare(current_state_back) != 0) {
    print_status(closed, 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));
      closed.insert(current_state_back);
    }
    if (frontier.empty()) {
      found = false;
      break;
    }
    current_state = frontier.front();
    current_state_back = current_state.path.back();
  }
  if (found) {
    print_status(closed, 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;
}

void uniform_cost_search(struct di_graph graph, string start_vertex,
                         string goal_vertex) {
  cout << "UNIFORM COST SEARCH" << endl;
  set<string> closed{};
  priority_queue<search_node> frontier{}; // MIN HEAP
  frontier.push(to_search_node(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(closed, priority_queue_to_list(frontier), current_state_back,
                 get_successors(graph, current_state_back), true);
    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));
      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(closed, priority_queue_to_list(frontier), current_state_back,
                 get_successors(graph, current_state_back), true);
    cout << "Path to goal node found: "
         << solution_path_cost(graph, current_state) << endl;
  } else
    cout << "Goal not found!" << endl;
}

list<search_node> stack_to_list(stack<search_node> frontier) {
  list<search_node> alist{};
  while (!frontier.empty()) {
    alist.push_back(frontier.top());
    frontier.pop();
  }
  return alist;
}

list<search_node> queue_to_list(queue<search_node> frontier) {
  list<search_node> alist{};
  while (!frontier.empty()) {
    alist.push_back(frontier.front());
    frontier.pop();
  }
  return alist;
}

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_as_string(search_node sn, bool show_path_cost) {
  string s{};
  s.append("(");
  for (string v : sn.path) {
    s.append(v);
    s.append("-");
  }
  if (s.length() > 1)
    s.pop_back();
  if (show_path_cost) {
    s.append(" ");
    s.append(to_string(sn.cost));
  }
  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 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(set<string> closed, list<search_node> frontier,
                  string current_state, list<string> successors,
                  bool show_path_cost) {
  cout << "frontier:[";
  for (search_node sn : frontier)
    cout << search_node_as_string(sn, show_path_cost);
  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;
  }
}

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

search_node to_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 = parent_sn.cost + get_weight(graph, parent_sn.path.back(), node);
```

```
  return sn;
}
```
lab01_search.cpp

```
#include "lab01_search.hpp"

int main(int argc, char **argv) { struct di_graph graph { };
  string fn{};
  string start_vertex{}, goal_vertex{}, search_method{};
  if (argc == 5) {
    fn = argv[1];
    start_vertex = argv[2];
    goal_vertex = argv[3];
    search_method = argv[4];
    graph = read_data(fn);
    if (search_method.compare("bfs") == 0)
      breadth_first_search(graph, start_vertex, goal_vertex);
    else if (search_method.compare("dfs") == 0)
      depth_first_search(graph, start_vertex, goal_vertex);
    else if (search_method.compare("ucs") == 0)
      uniform_cost_search(graph, start_vertex, goal_vertex);
    else
      cerr << "invalid option" << endl;
  }
  free_memory(graph);
}
```
lab01_03.cpp

Ο κώδικας μεταγλωττίζεται με την εντολή:

```
g++ lab01_graph.cpp lab01_search.cpp lab01_03.cpp -o lab01_03 -Wall -std=c++11
```

Εύρεση διαδρομής από Craiova προς Bucharest με BFS

```
./lab01_03 data/tour_romania.txt C B bfs
BREADTH FIRST SEARCH
frontier:[(C)] closed set:[] current node:C successors:[D P R]
frontier:[(C-D)(C-P)(C-R)] closed set:[C] current node:D successors:[C M]
frontier:[(C-P)(C-R)(C-D-C)(C-D-M)] closed set:[C D] current node:P successors:[B C R]
frontier:[(C-R)(C-D-C)(C-D-M)(C-P-B)(C-P-C)(C-P-R)] closed set:[C D P] current node:R successors:[C P S]
frontier:[(C-D-C)(C-D-M)(C-P-B)(C-P-C)(C-P-R)(C-R-C)(C-R-P)(C-R-S)] closed set:[C D P R] current node:C
successors:[loop]
frontier:[(C-D-M)(C-P-B)(C-P-C)(C-P-R)(C-R-C)(C-R-P)(C-R-S)] closed set:[C D P R] current node:M successors:[D L]
frontier:[(C-P-B)(C-P-C)(C-P-R)(C-R-C)(C-R-P)(C-R-S)(C-D-M-D)(C-D-M-L)] closed set:[C D M P R] current node:B
successors:[F G P U]
Path to goal node found: (Craiova-Pitesti-Bucharest 239)
```

Εύρεση διαδρομής από Craiova προς Bucharest με DFS

```
lab01_03 data/tour_romania.txt C B dfs
DEPTH FIRST SEARCH
frontier:[(C)] closed set:[] current node:C successors:[D P R]
frontier:[(C-D)(C-P)(C-R)] closed set:[C] current node:D successors:[C M]
frontier:[(C-D-C)(C-D-M)(C-P)(C-R)] closed set:[C D] current node:C successors:[loop]
frontier:[(C-D-M)(C-P)(C-R)] closed set:[C D] current node:M successors:[D L]
frontier:[(C-D-M-D)(C-D-M-L)(C-P)(C-R)] closed set:[C D M] current node:D successors:[loop]
frontier:[(C-D-M-L)(C-P)(C-R)] closed set:[C D M] current node:L successors:[M T]
frontier:[(C-D-M-L-M)(C-D-M-L-T)(C-P)(C-R)] closed set:[C D L M] current node:M successors:[loop]
frontier:[(C-D-M-L-T)(C-P)(C-R)] closed set:[C D L M] current node:T successors:[A L]
frontier:[(C-D-M-L-T-A)(C-D-M-L-T-L)(C-P)(C-R)] closed set:[C D L M T] current node:A successors:[S T Z]
frontier:[(C-D-M-L-T-A-S)(C-D-M-L-T-A-T)(C-D-M-L-T-A-Z)(C-D-M-L-T-L)(C-P)(C-R)] closed set:[A C D L M T] current
node:S successors:[A F O R]
frontier:[(C-D-M-L-T-A-S-A)(C-D-M-L-T-A-S-F)(C-D-M-L-T-A-S-O)(C-D-M-L-T-A-S-R)(C-D-M-L-T-A-T)(C-D-M-L-T-A-Z)(C-D-M-
L-T-L)(C-P)(C-R)] closed set:[A C D L M S T] current node:A successors:[loop]
frontier:[(C-D-M-L-T-A-S-F)(C-D-M-L-T-A-S-O)(C-D-M-L-T-A-S-R)(C-D-M-L-T-A-T)(C-D-M-L-T-A-Z)(C-D-M-L-T-L)(C-P)(C-R)]
closed set:[A C D L M S T] current node:F successors:[B S]
frontier:[(C-D-M-L-T-A-S-F-B)(C-D-M-L-T-A-S-F-S)(C-D-M-L-T-A-S-O)(C-D-M-L-T-A-S-R)(C-D-M-L-T-A-T)(C-D-M-L-T-A-Z)(C-
D-M-L-T-L)(C-P)(C-R)] closed set:[A C D F L M S T] current node:B successors:[F G P U]
Path to goal node found: (Craiova-Drobeta-Mehadia-Lugoj-Timisoara-Arad-Sibiu-Fagaras-Bucharest 944)
```

Εύρεση διαδρομής από Craiova προς Bucharest με UCS

```
lab01_03 data/tour_romania.txt C B ucs
UNIFORM COST SEARCH
frontier:[(C 0)] closed set:[] current node:C successors:[D P R]
frontier:[(C-D 120)(C-P 138)(C-R 146)] closed set:[C] current node:D successors:[C M]
frontier:[(C-P 138)(C-R 146)(C-D-M 195)(C-D-C 240)] closed set:[C D] current node:P successors:[B C R]
frontier:[(C-R 146)(C-D-M 195)(C-P-R 235)(C-P-B 239)(C-D-C 240)(C-P-C 276)] closed set:[C D P] current node:R
successors:[C P S]
frontier:[(C-D-M 195)(C-R-S 226)(C-P-R 235)(C-P-B 239)(C-D-C 240)(C-R-P 243)(C-P-C 276)(C-R-C 292)] closed set:[C D
P R] current node:M successors:[D L]
frontier:[(C-R-S 226)(C-P-R 235)(C-P-B 239)(C-D-C 240)(C-R-P 243)(C-D-M-L 265)(C-D-M-D 270)(C-P-C 276)(C-R-C 292)]
closed set:[C D M P R] current node:S successors:[A F O R]
frontier:[(C-P-R 235)(C-P-B 239)(C-D-C 240)(C-R-P 243)(C-D-M-L 265)(C-D-M-D 270)(C-P-C 276)(C-R-C 292)(C-R-S-R
306)(C-R-S-F 325)(C-R-S-A 366)(C-R-S-O 377)] closed set:[C D M P R S] current node:R successors:[loop]
frontier:[(C-P-B 239)(C-D-C 240)(C-R-P 243)(C-D-M-L 265)(C-D-M-D 270)(C-P-C 276)(C-R-C 292)(C-R-S-R 306)(C-R-S-F
325)(C-R-S-A 366)(C-R-S-O 377)] closed set:[C D M P R S] current node:B successors:[F G P U]
Path to goal node found: (Craiova-Pitesti-Bucharest 239)
```

Αναφορές

1. [Ertel11] Introduction to Artificial Intelligence, Wolfgang Ertel, Springer, 2011.
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 (visualization) αλγορίθμων BFS και DFS

1. https://visualgo.net/dfsbfs
2. https://www.cs.usfca.edu/~galles/visualization/DFS.html
3. https://www.cs.usfca.edu/~galles/visualization/BFS.html
4. http://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html