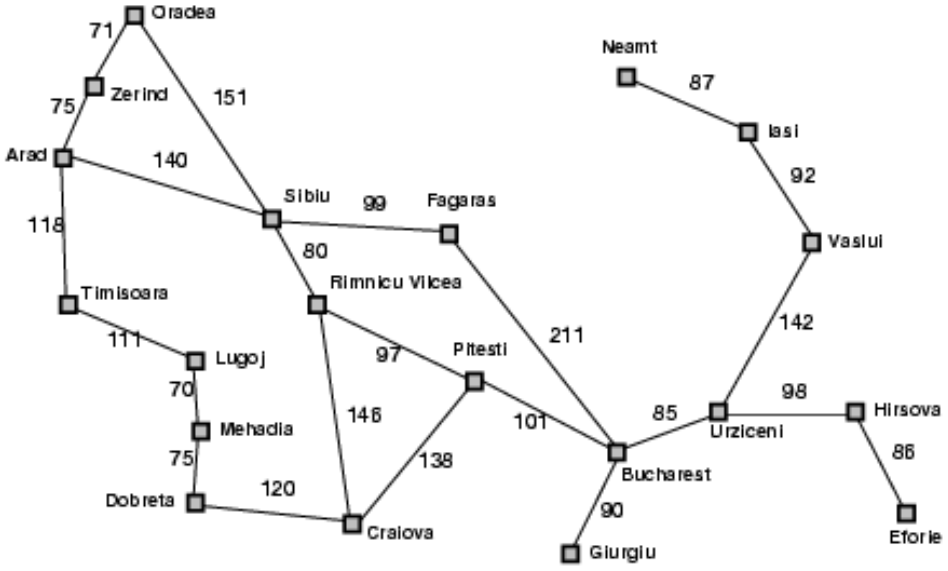
Căutare A*



Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Scrieți un program în care se aplică o căutare A* pentru a găsi ruta de la orice oraș către București.

Instrucțiuni legate de implementarea căutării A*:

1. Se folosesc aproximările (*h*) de la fiecare nod către București și costurile (*cost*).
2. În tabloul *noduri* vom ordona elementele în funcție de suma dintre aproximarea *h* și costul *cost*. Așadar, un nod *i* apare înaintea lui *j* în *noduri* dacă *cost*[*noduri*[*i*]] + *h*[*noduri*[*i*]] < *h*[*noduri*[*j*]] + *cost*[*noduri*[*j*]].

Observații legate de căutarea A* vs Greedy vs Cost uniform

- Căutarea cu cost uniform verifică la fiecare pas cât de mult de îndepărtăm de nodul rădăcină și alege mereu să extindă drumurile cele mai apropiate de rădăcină (*start*).
- Căutarea Greedy alege să extindă drumurile cele mai apropiate de țintă (*stop*) pe baza aproximărilor cunoscute.

3. Căutarea A* îmbină avantajele amândurora, evaluând astfel un drum complet de la *start* către *stop* ce trece prin nodul curent.

Soluție

Iată una din soluțiile primite la care am făcut mici modificări.

Soluție

Observații: programul conține toate căutările de până acum. La A*, se adaugă nodurile noi la final și apoi se face o ordonare a intregii liste după *h* și *cost*.

```
/**
 L09. Scrie?i un program în care se aplicã o cãutare A* pentru a gãsi
ruta de la
orice ora? cãtre Bucure?ti. Instruc?iuni legate de implementarea cãutãrii
A*:
1. Se folosesc aproximãrile (h) de la fiecare nod cãtre Bucure?ti ?i
costurile
(cost).
2. În tabloul noduri vom ordona elementele în func?ie de suma dintre
aproximarea
h ?i costul cost. A?adar, un nod i apare înaintea lui j în noduri dacã
cost[noduri[i]] + h[noduri[i]] < h[noduri[j]] + cost[noduri[j]].

Observa?ii legate de cãutarea A* vs Greedy vs Cost uniform
- Cãutarea cu cost uniform verificã la fiecare pas cât de mult de
îndepãrtãm de
nodul rãdãcinã ?i alege mereu sã extindã drumurile cele mai apropiate de
rãdãcinã (start).
- Cãutarea Greedy alege sã extindã drumurile cele mai apropiate de ?intã
(stop)
pe baza aproximãrilor cunoscute.
- Cãutarea A* îmbinã avantajele amândurora, evaluând astfel un drum
complet de
la start cãtre stop ce trece prin nodul curent.
*/
#include <iomanip>

#include <iostream>

#include <string>

using namespace std;

#
pragma region Setup
// prototipuri de functii
void uniform_cost_search(int, int, int & );
void greedy_search(int, int, int & );
void astar_search(int, int, int & );
void get_city_id(int * );
void print_problem();
string get_steps(string & , int, int[], int[], int);
void print_results(int, int, int[], int, int[]);

// variabile globale
```

```
string city_names[] = {
  "Arad",
  "Zerind",
  "Oradea",
  "Timisoara",
  "Lugoj",
  "Mehadia",
  "Drobeta",
  "Craiova",
  "Ramnicu Valcea",
  "Sibiu",
  "Fagaras",
  "Pitesti",
  "Bucuresti",
  "Giurgiu",
  "Urziceni",
  "Harsova",
  "Eforie",
  "Vaslui",
  "Iasi",
  "Neamt"
};
const int size = sizeof(city_names) / sizeof(city_names[0]);
// matricea de adiacenta dupa distanta
int distances[][size] = {0, 75, 0, 118, 0, 0, 0, 0, 0, 140, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0},
    {75, 0, 71, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 71, 0, 0, 0, 0, 0, 0, 0, 151, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {118, 0, 0, 0, 111, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 111, 0, 70, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 70, 0, 75, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 75, 0, 120, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 120, 0, 146, 0, 0, 138, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 146, 0, 80, 0, 97, 0, 0, 0, 0, 0, 0, 0, 0},
    {140, 0, 151, 0, 0, 0, 0, 0, 80, 0, 99, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 99, 0, 0, 211, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 138, 97, 0, 0, 0, 101, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 211, 101, 0, 90, 85, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 90, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 85, 0, 0, 98, 0, 142, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 98, 0, 86, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 86, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 142, 0, 0, 0, 92, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 92, 0, 87},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 87, 0}};
int estimates[size] = {366, 374, 380, 329, 244, 241, 242, 160, 193,
                       253, 176, 10, 0, 77, 80, 151, 161, 199, 226, 234};
bool display_steps = false; // setare: afiseaza pasii intermediari
string separator =
  "\n**********************************************************\n";#
pragma endregion Setup

/**
 @brief Entry point
 @param argc numar de argumente introduse
 @param *argv[] pointer la un array de caractere, fiecare element este un
 argument introdus
```

```
 @arg -d seteaza display_steps ca true
*/
int main(int argc, char * argv[]) {
  // parsarea argumentelor cu care este pornita aplicatia
  for (int i = 1; i < argc; ++i) {
    string arg = argv[i];
    if (arg == "-d") {
      display_steps = true;
    }
  }

  int start;
  int finish;
  int cost_ucs;
  int cost_greedy;
  int cost_astar;

  // input loop pentru a putea incerca diferite combinatii de orase,
  // fara a restarta aplicatia si fara a le hardcoda
  while (true) {
    print_problem();

    cout << "\n\nScrie indexul orasului de plecare: ";
    get_city_id( & start);
    cout << "Plecare: " << city_names[start];

    cout << "\n\nScrie indexul orasului de sosire: ";
    get_city_id( & finish);
    cout << "Sosire: " << city_names[finish];

    uniform_cost_search(start, finish, cost_ucs);
    greedy_search(start, finish, cost_greedy);
    astar_search(start, finish, cost_astar);

    cout << "\n\nSumar:" << separator;
    cout << "UCS    : " << setw(4) << cost_ucs << " km\n";
    cout << "Greedy : " << setw(4) << cost_greedy << " km\n";
    cout << "A*     : " << setw(4) << cost_astar << " km\n";

    cout << separator;

    // reseteaza variabilele pentru urmatorul input
    cost_ucs = 0;
    cost_greedy = 0;
    cost_astar = 0;
  }

  return 0;
}

# pragma region Algorithms
/**
 @brief Implementare a algoritmului de cautare cu cost uniform (uniform
cost
 search)
 @param start_node punct de plecare (index din city_names, definita
global)
```

```
 @param finish_node punct de sosire (idem)
 @param cost
*/
void uniform_cost_search(int start_node, int finish_node, int & cost) {
  int frontier[size] = {
    start_node
  };
  int frontier_count = 1;
  bool explored[size] = {
    false
  };
  explored[start_node] = true;
  int ascendants[size] = {
    0
  };
  bool found = false;
  int costs[size] = {
    0
  };
  int step = 1;
  string recorded_steps = "";

  while (!found && frontier_count > 0) {
    // primul element din frontiera devine nodul curent
    int current_node = frontier[0];
    int current_cost = costs[current_node];

    // inregistreza pasii
    if (display_steps) {
      get_steps(recorded_steps, step, frontier, costs, frontier_count);
    }
    step++;

    // elimina primul element din frontiera (pop)
    // deplaseaza continutul in uz al frontierei cu un element spre
stanga
    for (int i = 0; i < frontier_count - 1; i++) {
      frontier[i] = frontier[i + 1];
    }
    frontier_count--;

    // verifica daca nodul curent este tinta
    if (current_node == finish_node) {
      found = true;
    } else {
      // itereaza in linia din matricea de adiacenta,
      // corespunzatoare nodului curent pentru a gasi nodurile succesoare
      for (int i = 0; i < size; i++) {
        if ((distances[current_node][i] != 0) && !explored[i] ||
          ((distances[current_node][i] != 0) && explored[i] &&
            (distances[current_node][i] > 0) &&
            (distances[current_node][i] + current_cost < costs[i]))) {
          frontier[frontier_count] = i;
          costs[i] = current_cost + distances[current_node][i];
          frontier_count++;
          explored[i] = true;
          ascendants[i] = current_node;
```

```
        }
      }

      for (int i = 0; i < frontier_count - 1; i++) {
        for (int j = i + 1; j < frontier_count; j++) {
          if (costs[frontier[i]] > costs[frontier[j]]) {
            int temp = frontier[i];
            frontier[i] = frontier[j];
            frontier[j] = temp;
          }
        }
      }
    }
  }

  // afiseaza denumirea algoritmului
  cout << "\n\nCautare cu cost uniform (Uniform Cost Search)" <<
separator;

  // afiseaza pasii intermediari
  cout << recorded_steps;

  int solution[size] = {
    0
  }; // solutia
  int solution_count = 0; // numar de noduri din solutie
  int current_node = finish_node;

  while (current_node != start_node) {
    solution[solution_count] = current_node;
    solution_count++;
    current_node = ascendants[current_node];
  }
  solution[solution_count] = start_node;
  solution_count++;

  cost = costs[finish_node];

  print_results(start_node, finish_node, solution, solution_count,
costs);
}

/**
 @brief Implementare a algoritmului de cautare greedy
 @param start_node punct de plecare (index din city_names, definita
global)
 @param finish_node punct de sosire (idem)
 @param cost
 */
void greedy_search(int start_node, int finish_node, int & cost) {
  int frontier[size] = {
    start_node
  };
  int frontier_count = 1;
  bool explored[size] = {
    false
  };
```

```
  explored[start_node] = true;
  int ascendants[size] = {
    0
  };
  bool found = false;
  int costs[size] = {
    0
  };
  int step = 1;
  string recorded_steps = "";

  while (!found && frontier_count > 0) {
    // primul element din frontiera devine nodul curent
    int current_node = frontier[0];
    int current_cost = costs[current_node];

    // inregistreza pasii
    if (display_steps) {
      get_steps(recorded_steps, step, frontier, costs, frontier_count);
    }
    step++;

    // elimina primul element din frontiera (pop)
    // deplaseaza continutul in uz al frontierei cu un element spre
stanga
    for (int i = 0; i < frontier_count - 1; i++) {
      frontier[i] = frontier[i + 1];
    }
    frontier_count--; // actualizeaza numarul de noduri de explorat

    // verifica daca nodul curent este tinta
    if (current_node == finish_node) {
      found = true;
    } else {
      // itereaza in linia din matricea de adiacenta,
      // corespunzatoare nodului curent pentru a gasi nodurile succesoare
      for (int i = 0; i < size; i++) {
        if ((distances[current_node][i] != 0) && !explored[i]) {
          frontier[frontier_count] = i;
          costs[i] = current_cost + distances[current_node][i];
          frontier_count++;
          explored[i] = true;
          ascendants[i] = current_node;
        }
      }

      for (int i = 0; i < frontier_count - 1; i++) {
        for (int j = i + 1; j < frontier_count; j++) {
          if (estimates[frontier[i]] > estimates[frontier[j]]) {
            int temp = frontier[i];
            frontier[i] = frontier[j];
            frontier[j] = temp;
          }
        }
      }
    }
  }
```

```
  // afiseaza denumirea algoritmului
  cout << "\n\nCautare greedy" << separator;

  // afiseaza pasii intermediari
  cout << recorded_steps;

  int solution[size] = {
    0
  };
  int solution_count = 0;
  int current_node = finish_node;

  while (current_node != start_node) {
    solution[solution_count] = current_node;
    solution_count++;
    current_node = ascendants[current_node];
  }
  solution[solution_count] = start_node;
  solution_count++;

  cost = costs[finish_node];

  print_results(start_node, finish_node, solution, solution_count,
costs);
}

/**
 @brief Implementare a algoritmului de cautare A*
 @param start_node punct de plecare (index din city_names, definita
global)
 @param finish_node punct de sosire (idem)
 @param cost
 */
void astar_search(int start_node, int finish_node, int & cost) {
  int frontier[size] = {
    start_node
  };
  int frontier_count = 1;
  bool explored[size] = {
    false
  };
  explored[start_node] = true;
  int ascendants[size] = {
    0
  };
  bool found = false;
  int costs[size] = {
    0
  };
  int step = 1;
  string recorded_steps = "";

  while (!found && frontier_count > 0) {
    // primul element din frontiera devine nodul curent
    int current_node = frontier[0];
    int current_cost = costs[current_node];
```

```
    // inregistreza pasii
    if (display_steps) {
      get_steps(recorded_steps, step, frontier, costs, frontier_count);
    }
    step++;

    // elimina primul element din frontiera (pop)
    // deplaseaza continutul in uz al frontierei cu un element spre
stanga
    for (int i = 0; i < frontier_count - 1; i++) {
      frontier[i] = frontier[i + 1];
    }
    frontier_count--; // actualizeaza numarul de noduri de explorat

    // verifica daca nodul curent este tinta
    if (current_node == finish_node) {
      found = true;
    } else {
      // itereaza in linia din matricea de adiacenta,
      // corespunzatoare nodului curent pentru a gasi nodurile succesoare
      for (int i = 0; i < size; i++) {
        if ((distances[current_node][i] != 0) && !explored[i]) {
          frontier[frontier_count] = i;
          costs[i] = current_cost + distances[current_node][i];
          frontier_count++;
          explored[i] = true;
          ascendants[i] = current_node;
        }
      }

      for (int i = 0; i < frontier_count - 1; i++) {
        for (int j = i + 1; j < frontier_count; j++) {
          if ((costs[frontier[i]] + estimates[frontier[i]]) >
            (costs[frontier[j]] + estimates[frontier[j]])) {
            int temp = frontier[i];
            frontier[i] = frontier[j];
            frontier[j] = temp;
          }
        }
      }
    }
  }

  // afiseaza denumirea algoritmului
  cout << "\n\nCautare A*" << separator;

  // afiseaza pasii intermediari
  cout << recorded_steps;

  int solution[size] = {
    0
  };
  int solution_count = 0;
  int current_node = finish_node;

  while (current_node != start_node) {
```

```
    solution[solution_count] = current_node;
    solution_count++;
    current_node = ascendants[current_node];
  }
  solution[solution_count] = start_node;
  solution_count++;

  cost = costs[finish_node];

  print_results(start_node, finish_node, solution, solution_count,
costs);
}#
pragma endregion Algorithms

# pragma region Utils
/**
 @brief Obtine de la utilizator un index valabil si il aloca variabilei
 primite
 @param id pointer la variabila care va folosi valoarea introdusa
*/
void get_city_id(int * id) {
  int temp;

  while (true) {
    // FIXME: input de tip 2asd e validat ca 2
    if ((cin >> temp) && (temp >= 0) && (temp < size)) {
      cin.ignore(INT_MAX, '\n'); // INT_MAX e arbitrar ales
      break;
    }
    cout << "Indexul trebuie sa fie un intreg intre 0 si " << size - 1 <<
      "!\nMai incearca: ";
    cin.clear();
    cin.ignore(INT_MAX, '\n');
  }

  * id = temp;
}

/**
 @brief Afiseaza lista oraselor pentru ca utilizatorul sa le poata
selecta
 dupa id.
 */
void print_problem() {
  cout << "Lista oraselor:\n";
  for (int i = 0; i < size; i++) {
    cout << setw(4) << std::right << i << ". ";
    cout << setw(15) << std::left << city_names[i] << "\t";
    if ((i + 1) % 4 == 0) {
      cout << '\n';
    }
  }
}

/**
 @brief Inregistreaza pasii intermediari in functii, depinde de setarea
 display_steps
```

```
 @param recorded_steps string in care se acumuleaza inregistrarile
 @param step numarul pasului din functia apelanta
 @param frontier lista de noduri de explorat
 @param nodes lista in care se itereaza
 @param node_count numarul de elemente afisate din lista
*/
string get_steps(string & recorded_steps, int step, int frontier[], int
nodes[],
  int node_count) {
  if (display_steps) {
    char node[10];
    sprintf(node, "%3d: ", step);
    recorded_steps.append(node);

    for (int i = 0; i < node_count; i++) {
      char distance[10];
      sprintf(distance, "%d", nodes[frontier[i]]);
      string city = city_names[frontier[i]] + " [" + distance + "] ";
      recorded_steps.append(city);
    }
    recorded_steps.append("\n");
  }

  return recorded_steps;
}

/**
 @brief Returneaza un string, in format de ore, minute, (secunde),
 pornind de la timpul primit ca float
 @param time ore, ca numar real
*/
string get_formatted_time(float time) {
  // conversii implicite pentru a pastra partea intreaga
  int hours = time;
  float _minutes = (time - hours) * 60;
  int minutes = _minutes;
  // float _seconds = (_minutes - minutes) * 60;
  // int seconds = _seconds;

  char time_string[15];
  // int count = sprintf(time_string, "%d:%02d:%02d", hours, minutes,
  // seconds);
  sprintf(time_string, "%d:%02d", hours, minutes);

  return time_string;
}

/**
 @brief Afiseaza rezultatul cautarii
 @param start_node
 @param finish_node
 @param path lista de rezultate
 @param path_nodes_count numarul de elemente din lista
 @param cost lista de costuri
 @param time durata
*/
void print_results(int start_node, int finish_node, int path[size],
```

```
  int path_nodes_count, int cost[size]) {
  cout << "\nRezultat:\n" <<
    city_names[start_node] << " -> " << city_names[finish_node] << '\n';
  // deruleaza invers path[] pentru a afisa fiecare nod din solutie
  for (int i = path_nodes_count - 1; i >= 0; i--) {
    cout << setw(4) << std::right << path[i] << ". ";
    cout << setw(15) << std::left << city_names[path[i]];
    cout << std::right << setw(9) << cost[path[i]] << " km";
    cout << '\n';
  }
  cout << setw(10) << std::left << "Distanta: " << cost[path[0]] << "
km\n";
}#
pragma endregion Utils
```

Program rulat

```
Cautare A*
****************************************

Rezultat:
Arad -> Bucuresti
   0. Arad                      0 km
   9. Sibiu                   140 km
   8. Ramnicu Valcea          220 km
  11. Pitesti                 317 km
  12. Bucuresti               418 km
Distanta: 418 km
```

Soluția 2

Observație: Alternativ, pun doar metoda în care se face ordonarea elementelor din noduri după *h* si *cost* pe măsură ce sunt adăugate, adică se identifică pe ce poziție *j* trebuie să se pună elementul nou *i*.

```
void AStar(){
        int viz[N], noduri[N], parinte[N];
        int nrNoduri = 0;
        for (int i = 0; i < N; i++)
                viz[i] = 0;    //orase nevizitate
        noduri[0] = start;
        viz[start] = 1;
        nrNoduri++;
        bool gasit = false;
        viz[start] == 1;
        int k = 1;
        int cost[N];
        cost[start] = 0;


        while ((!gasit) && (nrNoduri > 0))
```

```
        {
                if (afisPasi)
                {
                        cout << "Pasul " << k++ << " ";
                        for (int i = 0; i < nrNoduri; i++)
                                cout << nume[noduri[i]] << " ";
                        cout << endl;
                }
                int curent = noduri[0];     //primul element din noduri
                //cout << nume[curent] << " ";
                for (int i = 0; i < nrNoduri - 1; i++)  //eliminam
elementul dupa prima pozitie din noduri
                        noduri[i] = noduri[i + 1];
                nrNoduri--;
                if (curent == stop)
                        gasit = true;
                else
                for (int i = 0; i < N; i++)
                if (a[curent][i] != 0)            //orasul i este conectat
de orasul curent si nu a fost vizitat anterior
                {
                        int costNou = a[curent][i] + cost[curent];
                        if ((viz[i] == 0) || (costNou < cost[i]))
                        {
                                if (viz[i] == 1)
                                { //eliminam orasul i din noduri
                                        int j = 0;
                                        while ((j < nrNoduri) && (noduri[j]
!= i))
                                                j++; //j reprezinta pozitia
pe care se gaseste i in noduri
                                        for (int q = j; q < nrNoduri - 1;
q++)
                                                noduri[q] = noduri[q + 1];
//am eliminat elementul dupa pozitia j din noduri
                                        nrNoduri--;
                                }
                                int j = 0;
                                while ((j<nrNoduri) &&
(costNou>cost[noduri[j]]+h[noduri[j]]))
                                        j++;    //j reprezinta pozitia pe
care adaugam orasul i

                                //adaugam orasul i pe pozita j
                                for (int q = nrNoduri; q > j; q--)
                                        noduri[q] = noduri[q - 1];
                                noduri[j] = i;
//noduri[j] reperzinta pozitia noastra
                                nrNoduri++;
                                cost[i] = costNou;
                                parinte[i] = curent;
                                viz[i] = 1;
                        }
                }
        }   //acolada este de la while

        cout << endl;
```

```
        int solutie[N], temp = stop, i = 0;
        while (temp != start){
                solutie[i++] = temp;
                temp = parinte[temp];
        }
        solutie[i++] = start;
        cout << "Solutia folosind AStar este urmatoarea: ";

        for (int j = i - 1; j >= 0; j--)

                cout << nume[solutie[j]] << " " << cost[solutie[j]] << "
";
        cout << endl;

        cout << endl;
}
```