UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE
AUTOMATICĂ, CALCULATOARE ŞI ELECTRONICĂ

INTELIGENTA ARTIFICIALA

TEMA LABORATOR

Problema N-Arcasi

Autor : Zane Livia, Calculatoare romana, Anul 2, Grupa CR2.3B

1 DESCRIEREA PROBLEMEI

Ca tema de casa in cadrul laboratorului la disciplina *Inteligenta Artificiala*, Calculatoare romana, anul II, semestrul II, mi-a fost alocata problema ***N-Arcasi***.

Conform cerintelor proiectului, trebuie identificat un algoritm de cautare pentru asezarea a ***n*** arcasi pe o tabla de sah de ***k * k*** patratele, fara ca arcasii sa se poata atinge cu sageti unul pe altul. Arcasii pot trage cu sageti pe linie, coloana sau diagonala, lungimea pana la care pot acestia sa traga fiind ***w patratele***, pana la marginea tablei. De asemenea, tabla este prevazuta cu un numar de ziduri, fiecare zid asezat intr-o patratica a tablei, in care se opresc sagetile (sagetile nu pot trece prin ziduri).

Trebuie definite urmatoarele aspecte:

1) formularea detaliata a acestei probleme de cautare;
2) identificarea unui algoritm de cautare pentru solutionarea problemei si explicarea alegerii respectivului algorim;
3) implementarea codului pentru rezolvarea problemei;
4) prezentarea si comentarea rezultatelor experimentale.

Totodata, exista urmatoarele cerinte generale:

1) trebuie prezentate urmatoarele livrabile:
   i) raport tehnic, cu urmatoarele capitole:
      a) pagina de stact (capacul);
      b) descrierea problemei;
      c) algoritmul in pseudocod;
      d) rezultatele de test;
      e) concluzii;
      f) bibliografia;
   ii) cod sursa (C, C++, Python, Java sau Prolog), cu explicitarea alegerii facute si respectarea urmatoarelor cerinte minime:
      a) instructiuni clare de compilare si executare;
      b) buna organizare in module si interfete;
      c) codul trebuie aliniat corespunzator, cu respectarea conventiilor de atribuire a numelor variabilelor, functiilor, claselor si pachetelor;
      d) codul trebuie bine comentat.
   iii) date experimentale, dupa cum urmeaza:
      a) cel putin 10 seturi non-triviale de test de dimensiuni variabile(mici, medii, mari si foarte mari);
      b) descrierea datelor de iesire obtinute urmare rularii setului de date de test si descrierea metodei de testare, respectiv daca aceasta este corect aplicata algoritmului;
      c) timpul de executie al algoritmului, pentru fiecare set de date de intrare.

2) temele se rezolva individual;

3) tema se transmite sub forma unui fisier arhiva de tip ZIP, in care sunt incluse toate livrabilele solicitate;

4) nu trebuie depasit termenul maxim de predare a temei (05.06.2022), altfel aceasta nu se ia in considerare.

Pentru utilizarea editorului L^AT_EX vor fi acordate puncte bonus.

Ca model de algoritm pentru cautare, a fost inclus in cerinte urmatorul pseudocod: Pornind

```
function Simple-Problem-Solving-Agent(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation
              state ← UPDATE-STATE(state, percept)
  if seq is empty then
    goal ← FORMULATE-GOAL(state)
    problem ←FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    if seq = failure then
      return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

de la acest model, va fi elaborat propriul algoritm pentru rezolvarea temei de casa repartizata, respectiv problema N-Arcasi.

2 ALGORITMI

Inainte de prezentarea algoritmilor, se impun a fi facute cateva precizari privind problema N-Arcasi, care este de fapt o extindere a problemei N-Regine[3] pe o tabla de sah ce nu se ataca reciproc (este chiar problema N-Regine, daca $w = k$ si numarul de ziduri este zero).

Suplimentar fata de problemei N-Regine, exista particularitati (cerinte suplimentare), una dintre acestea fiind aceea ca pe o linie/coloana/diagonala pot fi pozitionati mai multi arcasi, in functie de w (cate patratele poate parcurge sageata) si pozitionarea zidurilor. De asemenea, sunt celule ocupate de ziduri, iar acestea nu pot fi utilizate pentru plasarea de arcasi, insa au avantajul ca blocheaza sagetile. Asadar, daca in cazul problemei N-Regine pe o tabla de $n * n$ patrate pot fi plasate maxim $\boldsymbol{n}$ regine care sa nu se atace, in cazul problemei noastre nu poate fi precizat numarul de arcasi care pot fi plasati pe tabla.

Solutia optima pentru problema (ca si pentru problema N-Regine) ar fi algoritmul backtraking, insa cerintele problemei alocate cer implementarea unui algoritm de cautare.

La curs[1] au fost predati algoritmii de cautare neinformata prin parcurgerea arborilor/grafurilor pe nivele (BFS) si in adincime (DFS). Pentru gasirea drumurilor (de la nodul initial la nodul solutie) este mai eficient algoritmul DFS, deoarece solutia se poate afla numai pe un nod de pe ultimul nivel. In ceea ce priveste costul minim, acesta nu prezinta avantaje in cautare, deoarece costul nu este un aspect vizat de cerintele problemei, iar toti copii au acelasi cost. In ceea ce priveste cautarea euristica (informata), nu exista informatii (configuratia nodului solutie) care sa permita folosirea acesteia (distantele Hamming si Manhatan). Nici algoritmul Hill Climbing nu este eficient in cazul problemei N-Arcasi, din cauza complexitatii calculului (pentru fiecare patrat liber, sa se calculeze cu cati arcasi s-ar ataca daca s-ar pune un arcas in acel patrat, pentru a fi selectat minimul), precum si a constrangerilor (evitarea blocarii algoritmului in cazul atingerii unui minim local, prin utilizarea unor algoritmi suplimentari).

Avand in vedere cele de mai sus, consider ca varianta cea mai buna pentru solutionarea problemei, raportat la cunostintele predate la curs, ***este utilizarea alogitmului de cautare intai in adancime*** (DFS - *Depth First Search*).

Algoritmul propus pentru solutionarea problemei NArcasi este urmatorul:

function NArcasi-Problem-Solving-Agent(*n_arcasi_ce_nu_se_ataca*) **returns** *afiseaza lista_solutie sau mesaj "Fara solutie"*
  persistent: *lista_noduri, stiva, initial goala*
    tabla_curenta, initial 0, 0,..., 0
    tabla_solutia, initial necunoscuta
    n_arcasi_care_nu_se_ataca, formularea problemei
    noduri_vizitate, lista, initial goala
    lista_noduri ← *tabla_initiala*
  while *lista_noduri nu este goala* **do**
    tabla_curenta ← FIRST(*lista_noduri*)
    if *N_Arcasi_Nu_Se_Ataca*(*tabla_curenta*) = *n* **then //** sunt n arcasi care nu se ataca
      tabla_solutie ← *tabla_curenta*
      return *afiseaza tabla_solutie*
    if *tabla_curenta* ∉ *noduri_vizitate* **then**
      noduri_vizitate ← *tabla_curenta*
      tabla_curenta ← GENEREAZA(*copil*)
      lista_noduri ← *tabla_curenta*
    return *afiseaza mesaj "Fara solutie"*

Descrierea succinta a functionarii algoritmului, este urmatoare:

Anterior rularii algoritmului, vor fi citite datele de intrare: k - dimeniunea tablei, w - numarul de patratele pe care le parcurge o sageata, n - numarul de arcasi, z - numarul de ziduri si pozitionarea acestora.

Primul pas al algoritmului, este adaugarea in stiva *lista_noduri* a tablei initiale, care este goala (0 pe toate pozitiile, mai putin pentru ziduri, unde este completat cu 2).

Se continua intr-o bucla *while* cu scoaterea din stiva a primului nod. Se verifica daca acesta nu este in lista *noduri_vizitate*. Daca se regaseste, se trece la pasul urmator (se scoate un alt nod din stiva).

Daca nodul curent nu a fost vizitat, se genereaza toti copii acestuia (la *tabla_curenta* se mai

adauga un arcas), care respecta conditiile problemei (arcasii de pe tabla sa nu se atace intre ei pe linie/coloana/diagonale), iar acesti copii sunt adaugati in stiva.

La fiecare scoatere din stiva a unui nod, se verifica daca acesta indeplineste conditiile de a fi solutie a problemei, respectiv daca exista un numar de ***n arcasi***, iar acestia ***nu se ataca intre ei***, caz in care algoritmul se finalizeaza cu succes, fiind returnata *tabla_solutie* pentru a fi afisata. In cazul in care nu este gasita o tabla cu n arcasi, iar toate variantele posibile au fost explorate/vizitate, stiva se goleste si se iese din bucla cu *lista_solutie* = *lista_initala while* si se afiseaza mesajul "Fara solutie".

Lista *noduri_vizitate* este necesara pentru eliminarea ciclurilor[2].

Pentru implementarea algoritmului au fost folosite urmatoarele functii:

List < List < Integer >> ***cautare_DFS****(List < List < Integer >> tabla, int k, int n, int w)*, pentru executarea algoritmului DFS. Matricea *tabla* contine 0 in toate pozitiile si 2 in pozitiile in care au fost definite ziduri. Functia returneaza tabla, care are configuratia initiala daca nu s-a gasit solutie, sau configuratia solutiei, daca aceasta a fost gasita. Functia se afla in fisierul *Cautare_DFS.java* din folderul src/date si reprezinta implementarea algoritmului DFS;

```
List<List<Integer>> cautare_DFS(List<List<Integer>> tabla, int k, int n, int w){
    Set<String> noduri_vizitate = new HashSet<>();
    Vector<List<List<Integer>>> lista_noduri = new Vector<>();
    lista_noduri.add(tabla);

    while(lista_noduri.size() != 0) {
        List<List<Integer>> curent = copie(lista_noduri.elementAt(0));
        if (nr_arcasi(curent) == n) {
            tabla = copie(curent);                    // copiaza solutia in tabla
            return tabla;
        }
        String s=hash(curent);
        lista_noduri.remove(0);
        if (!noduri_vizitate.contains(s)) {
            noduri_vizitate.add(s);
            for (int i = 0; i < k; i++) {             // se genereaza toti copii
                for (int j = 0; j < k; j++) {         // si se adauga in stack
                    if (Verifica_plasare_arcasi.plasare_arcas(curent, i, j, k, w)) {
                        List<List<Integer>> copil = copie(curent);
                        copil.get(i).set(j, 1);
                        lista_noduri.add(0, copil);
                    }
                }
            }
        }
        }
    return tabla;                                     // tabla nemodificata
}
```

String ***hash*** *(List < List < Integer >> list)*, returneaza un sir (de tip string) cu cifrele din list, sir care este adaugat in lista *noduri_vizitate*, pentru a se verifica ulterior daca un nod a fost vizitat (stringul tablei se regaseste in *noduri_vizitate*. Este folosita in functia *cautare_DFS*. Se afla in fisierul *Cautare_DFS.java* din folderul src/date;

int ***nr_arcasi****(List < List < Integer >> list)*, realizeaza numararea arcasilor existenti la un

moment dat in tabla curenta. Este folosita in functia *cautare_DFS* pentru a verifica daca nodul scos din stiva este solutia problemei. Se afla in fisierul *Cautare_DFS.java* din folderul src/date;

List < List < Integer >> ***copie****(List < List < Integer >> list)*, realizeaza copierea unei liste. Este folosita in functia *cautare_DFS* pentru a copia nodul scos din stiva intr-un alt container de tip nod, altel aplicatia ar lucra cu acelasi nod (duplicat). Se afla in fisierul *Cautare_DFS.java* din folderul src/date;

boolean ***plasare_arcas****(List < List < Integer >> tabla, int row, int col, int k, int w)*, verifica daca un arcas poate fi pus pe tabla la pozitia row/col fara a se ataca cu ceilalti arcasi existenti pe tabla. Este folosita in functia *cautare_DFS* pentru generarea nodurilor copil. Se afla in fisierul *Verificare_plasare_arcasi.java* din folderul src/date. Mai jos **este prezentat un extras**, fisierul contine verificari la stanga/dreapta pentru linie/coloan/diagonala principala/diagonala secundara;

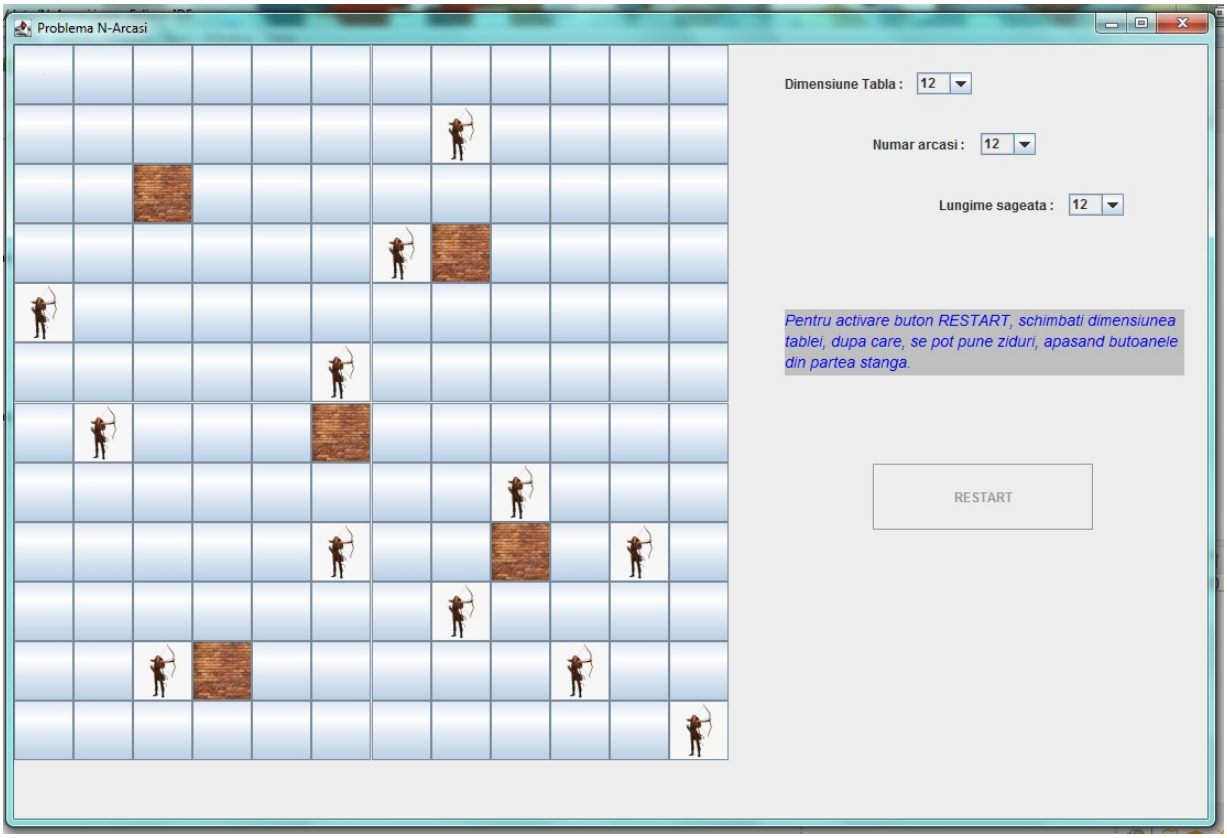
```
boolean plasare_arcas(List<List<Integer>> tabla, int row, int col, int k, int w){

    // daca exista un zid in celula row/col, returneaza false
    if(tabla.get(row).get(col).intValue() == 2)
        return false;

    int i, j;               // indecsi linie/coloana
    int ii;                 // incrementeaza doar cat este w

    // Verifica pe linie .....la dreapta
    for (i = col, ii = 0; i >= 0 && ii < w ; i--, ii++) {
        if (tabla.get(row).get(i).intValue() == 2)    // daca se intalneste un zid,
            break;    // se inceteaza cautarea, deoarece nu intereseaza arcasii de
                                // dupa zid, pentru ca nu se mai ataca intre ei
        if (tabla.get(row).get(i).intValue() == 1)
            return false;

    }

    // Verifica diagonala principala....in sus
    for (i = row, j = col, ii = 0; i >= 0 && j >= 0 && ii < w; i--, j--, ii++) {
        if (tabla.get(i).get(j).intValue() == 2)
            break;
        if (tabla.get(i).get(j).intValue() == 1)
            return false;
    }

    // Verifica diagonala secundara ...in jos
    for (i = row, j = col, ii = 0; j >= 0 && i < k && ii < w; i++, j--, ii++){
        if (tabla.get(i).get(j).intValue() == 2)
            break;
        if (tabla.get(i).get(j).intValue() == 1)
            return false;
    }

    return true;              // poate fi plasat un arcasi in patratelul row/col
}
```

void ***afisare_solutie****(List < List < Integer >> tabla, int k)*, realizeaza afisarea solutiei, daca aceasta a fost gasita. Se afla in fisierul *Afisare* din folderul src/date.

Pentru implementarea algoritmului am ales sa folosesc limbajul Java - Eclipse IDE, motivatia

fiind aceea ca in aceeasi perioada la Informatica Aplicata am ca tema realizarea unui joc Puzzle de $n * n$ patrate (n=2k+1), care este realizat, de asemenea, in Java, conform cerintei de aceasta data. Jocul are si o interfata grafica, de care m-am folosit pentru afisarea arcasilor si a zidurilor, in cazul prezentului proiect. Cu toate ca cerinta jocului Puzzle nu era sa afisam si solutia, pentru ca am invatat la cursul de Inteligenta Artificiala algoritmi de cautare, in respectivul joc am inserat si partea de cautare a solutiei, pornind de la o configuratie data a tablei Puzzle. Pentru acest lucru, am folosit algoritmul A^*, iar ca functii euristice, am folosit drumurile Hamming si Manhattan, deci doua variante de cautare a solutiei. Asadar, exista o serie de similitudini intre cele doua teme, care m-au determinat sa aleg acest limbaj de programare pentru implementarea algoritmului.

Interfata grafica a aplicatie este urmatoarea:



Pe aceeasi linie si in prezentul proiect am creat inca un algoritm pentru cautarea BFS, deoarece modificarea a fost minora, folosind un mecanism de tip coada in loc de stiva (am schimbat o singura linie de cod), parcurgerea fiind in acest caz pe nivele/latime (breadth). Rezultatele comparative intre cei doi algoritmi de cautare, pentru acelasi set de date, urmand sa fie prezentate in sectiunea urmatoare, ocazie cu care se va vedea si care dintre cei doi algoritmi este mai rapid pentru problema data.

3 DATE DE TEST

Pana la descrierea solutiei de testare a aplicatiei, se impune a fi facute cateva precizari. Pana la 12x12 patratele si 12 arcasi, fara ziduri, aplicatia merge foarte repede (2-3 secunde), explorand cateva mii de noduri. Odata cu cresterea numarului de arcasi (n) si a numarului de ziduri (z),

concomitent cu micsorarea distantei pana la care se duce o sageata (w), timpul creste foarte repede la cateva minute. Prezint mai jos cateva exemple:

1 - pentru k=8, n=10, w=8 si z=3, dupa aprox. 4 sec. si peste 170 mii noduri explorate, a gasit o solutie;

2 - pentru k=8, n=11, w=8 si z=3, dupa aprox. 15 sec. si peste 1,1 mil. noduri explorate, a afisat ca nu exista solutie;

3 - pentru k=10, n=12, w=10 si z=3, dupa aprox. 6 min. si peste 12,4 mil. noduri explorate, a gasit o solutie;

4 - pentru k=9, n=10, w=9 si z=0, dupa aprox. 4 min. si peste 5 mil. noduri explorate, a afisat ca nu exista solutie;

5 - pentru k=12, n=14, w=11 si z=3, dupa aprox. 5 min. si peste 11,5 mil. noduri explorate, a gasit o solutie;

6 - pentru k=10, n=11, w=10 si z=0, dupa aproximtiv 25 de minute si aproape 20 mil. noduri explorate, aplicatia a dat eroare - *java.lang.OutOfMemoryError: Java heap space*.

In contextul celor de mai sus, pentru a evita depasirea memoriei si blocarea aplicatiei si a testului, consider ca cele 4 seturi de cate 10 teste cu date non-triviale (generate aleatoriu), potrivit cerintelor, pot fi realizate astfel:

1 - setul mic de date, pentru tabla de 4x4-5x5 patratele (pana la 10^2 - 10^3 noduri explorate);

2 - setul mediu de date, pentru tabla de 6x6-7x7 patratele (pana la 10^4 - 10^6 noduri explorate);

3 - setul mare de date, pentru tabla de 8x8-9x9 patratele (pana la 10^6 - 10^9 noduri explorate);

4 - setul foarte mare de date, pentru tabla de 10x10-11x11 patratele (peste 10^{10} noduri explorate).

Datele de mai sus si cele care vor fi prezentate mai departe, au fost obtinute prin rularea algoritmului, realizat in Java - Eclipse IDE, pe un laptop cu sistem de operare Windows 7, procesor Intel CORE^{TM}i3 si 4 GB memorie RAM.

Pentru obtinerea unui set de date aleatorii, a fost folosita functia *Random()* din biblioteca *Math* din Java, iar pentru masurarea timpului CPU a fost folosita functia System.nanoTime(), inainte (timp_initial) si dupa rularea algoritmului (timp_final), respectiv inainte/dupa apelarea functiei Cautare_DFS si afisarea diferentei dintre cele doua ($timp\_final - timp\_initial$).

4 REZULTATE SI CONCLUZII

4.1 Rezultate

In tabelul urmator, sunt prezentate comparativ rezultatele (numarul de noduri explorate si timpii procesor) obtinte prin rularea algoritmilor DFS si BFS, pentru primul set de date non-triviale, respectiv table de $4 * 4$ sau $5 * 5$ patratele, $3 - 6$ arcasi, bataie sageata de $4 - 5$ patratele si $0 - 2$ ziduri:

Tabla(k)	Nr.arcasi(n)	Bat.sageata(w)	Ziduri	Noduri vizitate	Timp CPU [ms]
5x5 (DFS)	5	5	0	45	0.006
5x5 (BFS)	5	5	0	1,295	0.07
5x5 (DFS)	4	5	1	40	0.001
5x5 (BFS)	4	5	1	1,408	0.030
4x4 (DFS)	4	4	0	40	0.001
4x4 (BFS)	4	4	0	184	0.007
4x4 (DFS)	3	4	1	24	0.0005
4x4 (BFS)	3	4	1	151	0.002
5x5 (DFS)	5	5	0	45	0.0008
5x5 (BFS)	5	5	0	1,295	0.038
4x4 (DFS)	3	4	1	24	0.0003
4x4 (BFS)	3	4	1	164	0.0028
4x4 (DFS)	3	4	1	25	0.0007
4x4 (BFS)	3	4	1	196	0.006
4x4 (DFS)	4	4	0	40	0.002
4x4 (BFS)	4	4	0	184	0.009
5x5 (DFS)	5	5	0	45	0.0008
5x5 (BFS)	5	5	0	1,295	0.030
5x5 (DFS)	6	5	2	1,640	0.048
5x5 (BFS)	6	5	2	1,640	0.020

In continuare, sunt prezentate rezultatele obtinte prin rularea algoritmilor DFS si BFS, pentru al doi-lea set de date, respectiv table de $6 * 6$ sau $7 * 7$ patratele, $5 - 7$ arcasi, bataie sageata de $6 - 7$ patratele si $0 - 1$ ziduri:

Tabla(k)	Nr.arcasi(n)	Bat.sageata(w)	Ziduri	Noduri vizitate	Timp CPU [ms]
6x6 (DFS)	6	6	0	77	0.0004
6x6 (BFS)	6	6	0	11,965	0.19
7x7 (DFS)	6	7	1	112	0.0008
7x7 (BFS)	6	7	1	110,295	2.037

Tabla(k)	Nr.arcasi(n)	Bat.sageata(w)	Ziduri	Noduri vizitate	Timp CPU [ms]
6x6 (DFS)	6	6	0	749	0.007
6x6 (BFS)	6	6	0	10,907	0.096
6x6 (DFS)	6	6	0	749	0.007
6x6 (BFS)	6	6	0	10,907	0.096
7x7 (DFS)	6	7	1	120	0.0003
7x7 (BFS)	6	7	1	102,976	1.641
6x6 (DFS)	6	6	0	84	0.0003
6x6 (BFS)	6	6	0	11,965	0.082
7x7 (DFS)	6	7	1	107	0.0003
7x7 (BFS)	6	7	1	90,251	1.345
7x7 (DFS)	7	7	0	121	0.0004
7x7 (BFS)	7	7	0	104,096	1.697
6x6 (DFS)	5	6	1	75	0.0002
6x6 (BFS)	5	6	1	10,540	0.061
6x6 (DFS)	6	6	0	76	0.0002
6x6 (BFS)	6	6	0	10,708	0.083

In tabelul urmator sunt prezentate rezultatele obtinte prin rularea algoritmilor DFS si BFS, pentru al trei-lea set de date, respectiv table de $8*8$ sau $9*9$ patratele, $3-6$ arcasi, bataie sageata de $4-5$ patratele si $0-2$ ziduri:

Tabla(k)	Nr.arcasi(n)	Bat.sageata(w)	Ziduri	Noduri vizitate	Timp CPU [ms]
8x8 (DFS)	7	7	1	184	0.017
8x8 (BFS)	7	7	1	1,010,069	131
8x8 (DFS)	7	7	1	198	0.043
8x8 (BFS)	7	7	1	1,101,069	122
9x9 (DFS)	9	9	1	3,565	0.486
9x9 (DFS)	8	9	2	238	0.005
8x8 (DFS)	8	8	1	261	0.018
9x9 (DFS)	8	9	2	229	0.015
9x9 (DFS)	9	9	1	616	0.091
8x8 (DFS)	7	8	2	157	0.049
9x9 (DFS)	8	9	1	222	0.081
9x9 (DFS)	9	9	1	2,808	0.125

In continuare, sunt prezentate rezultatele obtinte prin rularea algoritmului DFS (faraBFS), pentru al patru-lea set de date, respectiv table de $10*10$ sau $11*11$ patratele, $5-7$ arcasi, bataie sageata de $6-7$ patratele si $0-1$ ziduri:

Tabla(k)	Nr.arcasi(n)	Bat.sageata(w)	Ziduri	Noduri vizitate	Timp CPU [ms]
11x11 (DFS)	13	11	3	470,004	13.7
10x10 (DFS)	10	10	1	1,394	6.4
11x11 (DFS)	11	11	1	9,801	0.334
11x11 (DFS)	13	11	4	2,854,578	78.4
11x11 (DFS)	11	11	1	5,626	0.149
11x11 (DFS)	13	11	4	2,003,465	58.3
10x10 (DFS)	10	10	1	2,041	0.282
11x11 (DFS)	13	11	4	8,519	4.4
10x10 (DFS)	11	10	2	544,033	12.6
11x11 (DFS)	13	11	4	19,883,802	-

Mentionez ca in cazul ultimului test, aplicatia a dat eroare de depasire memorie *Java heap* si nu am afisat timpul cat a durat rularea algoritmului (aprox. 25 min), respectiv afisarea timpului CPU la fiecare scoatere din stiva (pentru a fi afisat in consola inainte de aparitia erorii), deoarece operatiunile de afisare (I/O) sunt mari consumatoare de timp (comparativ cu celelate instructiuni) si ar fi denaturat semnificativ timpul de executie al algoritmului.

Avand in vedere numarul mare de noduri explorate in cazul algoritmului BFS, incepand cu setul de test III, testul 3, nu au mai fost testati ambii algoritmi, in continuare fiind aplicate teste numai pentru algoritmul DFS.

De retinut este faptul ca aceste operatii de testare a codului au fost executate cu datele generate aleatoriu (non-triviale), respectivele valori aleatorii (k, n, w, z), inclusiv configuratia tablelor (sub forma matriceala), fiind memorate in fisierul *RezultateTeste.txt* din folderul *src/test*. Testele ulterioare vor avea alte valori, deoarece au fost generate cu functia random().

Corectitudinea codului rezulta din faptul ca acesta a fost compilat, fara **erori** sau **atentionari**, in Java Eclipse IDE, dar si din aceea ca variantele de test generate aleatoriu au fost verificate vizual in interfata grafica, prin sondaj.

Concluzia cea mai importanta este aceea ca ***algoritmul DFS este mult mai rapid decat BFS, in cazul problemei NArcasi***, fiind explorate aproximativ de 10 ori mai putine noduri, fapt ce se repercuteaza pozitiv si asupra timpilor de executie si al memoriei ocupate. Explicatia este aceea ca solutia se regaseste pe o frunza a ultimului nivel al arborelui de cautare, iar daca algoritmul BFS trebuie sa parcurga tot arborele pana la ultimul nivel, algoritmul DFS gaseste cea mai din stanga solutie, fara sa pargurga tot arborele. Practic, daca solutia ar fi in frunza cea mai din stanga, s-ar explora n noduri (daca ar fi $n = 12$ arcasi, s-ar explora 12 noduri).

O alta concluzie desprinsa din efectuarea testelor, este aceea ca pozitionarea zidurilor, are o influenta foarte mare asupra numarului de noduri explorate, dar si asupra existentei/inexistentei unei solutii.Pentru detaliere, in cele trei figuri de mai jos, este prezentat numarul de noduri explorate si timpii procesor pentru rularea algoritmului DFS, in cazul tablei de 8x8 patratele, 9 arcasi, 4 ziduri (plasate in pozitii diferite) si o bataie de 8 patratele ale sagetilor. De precizat ca in Fig.3, cu toate ca exista acelasi numar de patratele, arcasi si ziduri ca si in cazul Fig.1 si Fig.2, pozitionarea zidurilor a condus la o tabla fara solutie. De asemenea, din Fig.1 si Fig.2, se pote vedea ca pozitionarea zidurilor generat diferente foarte mari intre numarul de noduri eplorate pentru fiecare din cele doua configuratii ale tablelor.

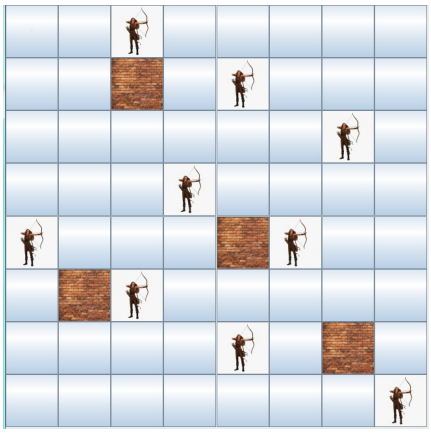


Fig.1 (7,978 noduri si 0.743 ms)

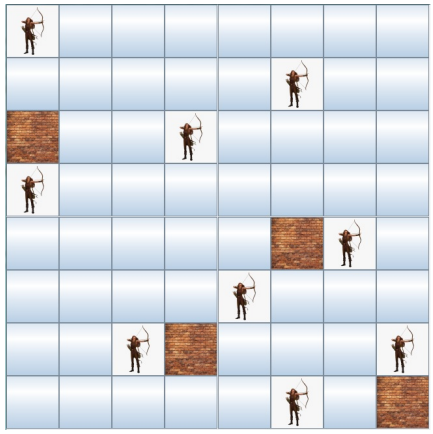


Fig.2 (38,710 noduri si 1,2 ms)

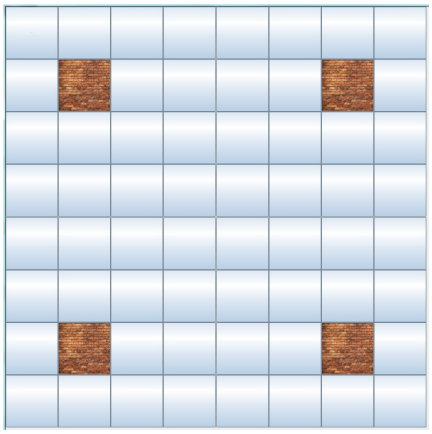


Fig.3 (825,320 noduri si 9,5 ms)

4.2 Concluzii

In prezntul proiect am implementat in Java algoritmii DFS si BFS pentru cautarea unei eventuale solutii a problemei N-Arcasi.

Am modificat codul unei interfete grafice realizat de mine anterior pentru problema N-Puzzle, in vederea usurarii modului de utilizare a aplicatiei.

Pentru testarea corectitudinii algoritmilor, conform cerintelor, am implementat si rulat 4 seturi de teste pentru date non-triviale de test mici($10^2 - 10^3$), medii($10^4 - 10^5$), mari($10^9 - 10^{10}$) si foarte mari (peste 10^{10}).Pentru aceasta, am generat cu ajutorul functiei *random()* din biblioteca *Math*, secvente aleatorii de numere intregi pentru valori ale tablei (k*k), bataia segetii (w), numarului de arcasi (n), al numarului de ziduri (z) si a pozitiilor acestora (row/col).

Am constat ca daca se coloreaza liniile unui tabel, in browser-ele PDF nu se vad foarte bine toate liniile tabelului, dar in L^AT_EX si la printare nu exista aceasta deficienta. Cauza este rasterizarea. Cea mai complicata parte a programului a fost, pentru mine, implementarea functiei care verifica daca un arcas poate fi plasat pe tabla in patratica row/col.

Sursa prezentului document, compilata fara erori/atentionari, poate fi accesata aici.

Precizari privind executia codului

Pentru executarea programului cu interfata grafica si setul de teste, se executa urmatorii pasi:

a) se copiaza folderul N-Arcasi (din folderul Code al arhivei) in worspace-ul aplicatiei Eclipse IDE (avand instalat modulul de dezvoltare Java);

b) din meniu se selecteaza File, apoi Import;

c) se selecteaza wizard-ul General, apoi Existing project into workspace;

d) se selecteaza folderul in care a fost copiat proiectul;

e) se apasa Finish.

Fisierul N_Arcasi.java se apeleaza pentru afisarea interfatei grafice a problemei, respectiv fisierul Test.java pentru lansarea seturilor de teste non-triviale, care se regaseste in folderul sr-

c/date, respectiv src/test, in care se regasete si fisierul RezultateTeste.txt, in care sunt salvate rezultatele celor 40 de teste (cate 10 mici/medii/mari/foarte mari), inclusiv tablele cu arcasi si ziduri generate aleatoriu, precum si numarul de noduri explorate si timpul de executie a algoritmului pentru fiecare test in parte.

Dupa lansarea in executie a fisierului N_Arcasi.java, interfata grafica va afisa initial o tabla de 12x12 patratele cu 12 arcasi, uterior putand fi schimbate toate variabilele. Zidurile se pun dupa selectarea unei dimensiuni a tablei. Prin lansarea fisierului Test.java se executa automat cele 40 de teste. Au existat cazuri cand executarea fisierului Test.java a dat eroare de indecsi matrici, dar la urmatoarea (eventual 2-3) rularea se duce cu testele pana la final, cu precizarea ca, cel putin pentru sistemul meu, ultimul test se termina cu eroare out of memmory Java heap.

Bibliografie

[1] Badica Costin. *Note de curs*. Google Classroom, 2021/2002.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.

[3] https://en.wikipedia.org/wiki/Eight_queens_puzzle.