

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**Problema conceperii unui plan de studii universitare
echilibrat, varianta generalizată**

propusă de

Gheorghiță Dăscălița

Sesiunea: Iunie/Iulie, 2022

Coordonator științific

Lector Dr. Cristian Frăsinaru

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

**Problema conceperii unui plan de studii
universitare echilibrat, varianta generalizată**

Gheorghiță Dăscălița

Sesiunea: Iunie/Iulie, 2022

Coordonator științific

Lector Dr. Cristian Frăsinaru

Cuprins

Introducere.....	1
1 Problema conceperii unui plan de studii universitare echilibrat, varianta generalizată.....	1
1.1 Introducere	1
1.2 Descrierea formală a problemei	2
1.3 Modelul CSP al problemei.....	3
1.3.1 Variabile și domenii.....	3
1.3.2 Constrângeri.....	3
1.3.3 Funcția obiectiv.....	4
1.4 Exemplu	5
2 Programarea bazată pe constrângeri.....	8
3 Descrierea aplicației	10
3.1 Arhitectura aplicației și tehnologii utilizate.....	10
3.2 Modelul aplicației	11
3.2.1 Clasa ProblemModel.....	11
3.2.2 Clasa UserModel.....	13
3.2.3 Baza de date Oracle și clasa DatabaseConnection.....	13
3.2.4 Clasa ModelUtils și tabela Users	13
3.3 Controlul aplicației	15
3.3.1 Modelarea problemei în Choco-solver	15
3.3.2 Servlet-uri: RegisterServlet, LoginServlet, LogoutServlet, GbaccServlet.....	21
3.4 Prezentarea aplicației	25
3.5 Ghidul de utilizare a aplicației	29
3.6 Complexitate și testare.....	33
Concluzii.....	1
Bibliografie.....	2

Introducere

Subiectul acestei lucrări de licență este “Problema conceperii unui plan de studii universitare echilibrat, varianta generalizată”. Echilibrul unui plan de studii constă în similaritatea între perioadele academice a necesarului de efort academic pe care un student trebuie să-l depună pentru urmarea cu succes a cursurilor. Astfel, fiecare curs are asociat un număr de credite care să reflecte efortul necesar. Cunosând apartenența cursurilor la perioadele de activitate didactică, se poate calcula printr-o simplă sumă numărul total de credite al unei perioade. Un plan de studii echilibrat încurajează formarea unor bune obiceiuri de învățare și facilitează succesul academic al studenților.

Scopul este atribuirea unei perioade fiecărui curs astfel încât să fie respectate condițiile și să fie maximizat echilibrul planului de studii universitare, în contextul specializărilor facultății. De asemenea, profesorii au posibilitatea să-și exprime preferința de a nu preda în anumite perioade ale anului. Această formulare a problemei se potrivește definiției date de Di Gaspero și Schaerf care au furnizat și 6 instanțe ale problemei obținute din datele reale ale Universității din Udine.

Această problemă este de o mare aplicabilitate în realitate întrucât toate facultățile o întâmpină, suferind diferite adaptări în funcție de caz. De asemenea, pentru a sublinia ideea aplicabilității acestui subiect, a fost creată o instanță și pentru Facultatea de Informatică a Universității “Alexandru Ioan Cuza” din Iași.

A fost dezvoltată o aplicație web, în limbajul de programare Java, cu ajutorul bibliotecii pentru programarea bazată pe constrângeri Choco-solver care a fost utilizată de mai mult de 20 de ani de companii și facultăți și care este una dintre cele mai rapide de pe piață. Capabilitățile *server-ului* web au fost extinse prin *Servlet-uri*. Pentru crearea în mod dinamic de pagini web generate pe baza HTML s-a folosit colecția de tehnologii *JSP*. Aplicația are o interfață grafică web cu utilizatorul, prin JavaScript făcându-se modificări HTML și CSS în mod dinamic pentru actualizarea interfeței prin *API-ul DOM*.

În primul capitol, se descrie formal problema, un aspect important al acestei lucrări fiind modelarea *CSP* a problemei. În al doilea capitol, se prezintă pe scurt paradigma programării bazate pe constrângeri, problemele de satisfacere a constrângerilor și *solver-ul* Choco-solver. Iar în al treilea capitol, se oferă detalii despre aplicația web dezvoltată.

1 Problema conceperii unui plan de studii universitare echilibrat, varianta generalizată

1.1 Introducere

"Problema conceperii unui plan de studii universitare echilibrat" ("*Balanced Academic Curriculum Problem*") este o problemă de optimizare combinatorială, pe care o are de rezolvat fiecare universitate și constă în atribuirea de cursuri perioadelor de activitate didactică, respectând condițiile și echilibrând necesarul de efort de învățare depus de studenți și reflectat în numărul de credite. Un plan de studii echilibrat facilitează succesul academic al studenților.

BACP este o problemă interesantă deoarece se află la granița dintre mai multe clase de probleme: de împachetare ("*bin-packing*"), planificare și echilibrare a sarcinii. Împachetarea reprezintă o clasă de probleme în care o mulțime de obiecte de dimensiuni diferite trebuie să încapă în cel mai mic număr de coșuri de capacitate limitată. BACP este într-un fel o problemă de împachetare în care capacitatea coșurilor este minimizată în loc de numărul coșurilor. Constanțele de condiționare reprezintă motivul pentru care BACP poate fi privită și ca o problemă de planificare. Fiecare curs este o activitate, creditele reprezintă consumarea resursei unice (efortul academic depus de studenți), iar condițiile sunt constrângeri temporale. Perioadele de activitate didactică sunt unități temporale și scopul este echilibrarea utilizării resursei. În cele din urmă, BACP reprezintă un exemplu excelent de problemă de echilibrare a sarcinii (a creditelor). (1)

Această problemă a fost inițial propusă de Castro și Manzano (2) și se regăsește în biblioteca CSPLib, ca și problema numărul 30 (3).

Totuși, această problemă este mai simplă decât problema pe care universitățile o au de rezolvat. Astfel Marco Chiarandini, Luca Di Gaspero, Stefano Gualandi și Andrea Schaerf au propus o variantă generalizată ("GBACP") (4). Varianta generalizată adaugă conceptele de specializare - planurile de studii pot avea cursuri în comun - și preferințe ale profesorilor de a nu preda în anumite perioade.

Problema BACP a fost abordată prin diverse metode: programare bazată pe constrângeri, programare liniară în numere întregi, tehnici hibride bazate pe algoritmi genetici și propagarea constrângerilor, metode hibride de căutare locală.

În această lucrare, propun o metodă de soluționare a problemei utilizând paradigma programării bazate pe constrângeri. În cele trei subcapitole următoare: se descrie formal problema, se prezintă modelul de programare bazată pe constrângeri și se exemplifică structura unui fișier de intrare ca și instanță a GBACP împreună cu soluția.

1.2 Descrierea formală a problemei

Problema BACP constă în conceperea unui plan de studii universitare, atribuind fiecărui curs o perioadă de activitate didactică astfel încât perioadele să fie la fel de încărcate, adică având un număr total de credite cât mai asemănător, respectând numărul total minim și maxim de credite și numărul minim și maxim de cursuri per perioadă de activitate didactică și relațiile de preconditionare.

Caracteristicile variantei generalizate a BACP sunt:

- plan de studii: o mulțime de cursuri și o mulțime de relații de preconditionare între ele.
- n : numărul de cursuri.
- m : numărul de perioade de activitate didactică care trebuie atribuite cursurilor.
- y : numărul de ani.
- p_y : numărul de perioade ale unui an.
- w_i : numărul întreg de credite al cursului i , pentru $i \in [0, n)$, și reprezintă efortul academic necesar pentru urmarea cu succes a cursului.
- k : numărul de specializări; în formularea BACP, se consideră un singur plan de studii, dar în practică există diferite specializări dintre care studenții pot alege; varianta generalizată a BACP ia în considerare mai multe specializări, adică planuri de studii ce pot avea cursuri în comun.
- S : matrice binară de atribuire specializare-curs, de dimensiuni $k \times n$, astfel încât $\forall (i, j) \in [0, k) \times [0, n)$, S_{ij} reprezintă legătura între specializări și cursuri, mai precis $S_{ij} = 1 \Leftrightarrow$ la specializarea i se predă cursul j .
- n_{prec} : numărul de preconditionii.
- $\text{Prec} = \{(i, j) \mid i \neq j, i, j \in [0, n)\}$: o mulțime de preconditionii, mai exact de perechi de cursuri, cursul i fiind o preconditionie pentru cursul j .
- a : numărul total de credite minim per perioadă de activitate didactică necesar pentru ca studentul să fie considerat cu norma întreagă.
- b : numărul total de credite maxim per perioadă de activitate didactică permis pentru a evita suprasolicitarea.

- c : numărul minim de cursuri per perioadă de activitate didactică necesar pentru ca studentul să fie considerat cu normă întreagă.
- d : numărul maxim de cursuri per perioadă de activitate didactică permis pentru a evita suprasolicitarea.
- $npref$: numărul de preferințe.
- $Pref$: matrice binară de dimensiuni $n \times m$, astfel încât $\forall (i, j) \in [0, n) \times [0, m)$, $Pref_{ij} = 1 \Leftrightarrow$ există o preferință a profesorilor de a nu preda cursul i în perioada j ; fixarea cursului i în perioada j , nepreferată de profesori, rezultă într-o penalizare și contribuie la funcția obiectiv, ca și constrângere slabă.

1.3 Modelul CSP al problemei

1.3.1 Variabile și domenii

Fiecare curs este identificat printr-un număr întreg din intervalul $[0, n)$ utilizat ca și indice în trei tablouri. Sunt cinci mulțimi de variabile:

- $\forall i \in [0, n)$, P_i reprezintă perioada cursului i .
- Se va utiliza o matrice binară B de atribuire curs-perioadă, de dimensiuni $n \times m$, astfel încât $\forall (i, j) \in [0, m) \times [0, n)$, B_{ij} reprezintă legătura între cursuri și perioade, mai precis $B_{ij} = 1 \Leftrightarrow$ cursul j este predat în perioada i .
- $\forall i \in [0, k)$, $\forall j \in [0, m)$, L_{ij} reprezintă numărul total de credite al perioadei j din cadrul specializării i .
- $\forall i \in [0, k)$, $\forall j \in [0, m)$, $Ldist_{ij}$ reprezintă distanța dintre numărul total de credite al perioadei j din cadrul specializării i și numărul total mediu de credite per perioadă al specializării i .
- $\forall i \in [0, n)$, $PrefP_i = 1 \Leftrightarrow$ cursul i este alocat unei perioade nepreferate de profesori și prin urmare trebuie înregistrată o penalizare.

1.3.2 Constrângeri

Valorile ce sunt atribuite variabilelor de mai sus trebuie să satisfacă mai multe constrângeri:

- $\forall (i, j) \in Prec: P_i < P_j$.
- $\forall s \in [0, k)$, $\forall j \in [0, m)$: numărul total de credite $L_{sj} \in [a, b]$. Numărul total de credite pentru fiecare specializare, pentru fiecare perioadă, este calculat în felul următor:

$$\forall s \in [0, k), \forall j \in [0, m): L_{sj} = \sum_{i=0}^{n-1} B_{ji} \times S_{si} \times w_i.$$

- $\forall s \in [0, k), \forall j \in [0, m)$: numărul de cursuri $N_{sj} \in [c, d]$. Numărul de cursuri predate la o anumită specializare, într-o anumită perioadă, este calculat în felul următor:

$$\forall s \in [0, k), \forall j \in [0, m): N_{sj} = \sum_{i=0}^{n-1} B_{ji} \times S_{si}.$$

- Există următoarea relație între vectorul de perioade P și matricea binară de atribuire curs-perioadă B :

$$\forall i \in [0, n) \text{ și } j \in [0, m): (P_i = j) \Leftrightarrow (B_{ji} = 1).$$

1.3.3 Funcția obiectiv

Numărul total mediu de credite per perioadă de la specializarea $s \in [0, k)$ este:

$$Lm_s = \frac{\sum_{i=0}^{n-1} S_{si} \times w_i}{m}.$$

Dezechilibrul necesarului de efort academic depus de studenți se masoară ca suma tuturor deviațiilor (pozitive și negative) față de numărul total mediu de credite per perioadă pentru fiecare specializare, în felul următor:

$$Cs = \sum_{s=0}^{k-1} \sum_{j=0}^{m-1} Ldist_{sj},$$

unde:

$$Ldist_{sj} = |Lm_s - L_{sj}|.$$

Suma tuturor penalizărilor asociate încălcării preferințelor profesorilor este:

$$Cp = \sum_{i=0}^{n-1} 2 \times w_i \times PrefP_i,$$

unde:

$$PrefP_i = Pref_{i[P_i]}.$$

Ca reper, pentru ca funcția obiectiv să returneze aceeași valoare în cazul în care $Cs = 0$, dar o preferință este încălcată, ca în cazul în care $Cp = 0$, dar cursul i asociat preferinței este mutat în altă perioadă, s-a considerat pentru încălcarea unei preferințe o penalizare de $2 \times w_i$.

Funcția obiectiv reflectă atât încălcarea preferințelor profesorilor, cât și dezechilibrul numărului total de credite per perioadă pentru fiecare specializare:

$$C = Cp + Cs.$$

1.4 Exemplu

Pe site-ul problemei GBAC (5) sunt puse la dispoziție atât o descriere detaliată a problemei, cât și zece instanțe CSP sub forma unor fișiere de intrare a căror structură este asemănătoare cu cea a exemplului de mai jos. Pornind de la acest exemplu, se vor identifica variabilele, se vor formula constrângerile și se va calcula costul soluției cu cele mai echilibrate specializări, luând în considerare costul încălcării preferințelor profesorilor.

Pentru exemplificare se dă următoarea mulțime de date de intrare:

- $y = 2$, numărul de ani.
- $p_y = 2$, numărul de perioade ale unui an.
- $m = 4$, numărul de perioade.
- $n = 6$, numărul de cursuri.
- $k = 3$, numărul de specializări.
- $a = 0$, numărul total de credite minim.
- $b = 30$, numărul total de credite maxim.
- $c = 0$, numărul minim de cursuri.
- $d = 2$, numărul maxim de cursuri.
- $n_{prec} = 3$, numărul de precondiții.
- $n_{pref} = 2$, numărul de preferințe.
- $w = [5, 10, 5, 4, 20, 7]$, vectorul în care se memorează numărul de credite pentru fiecare curs.
- $Specializări = \{(4, \{0, 1, 2, 3\}), (4, \{1, 2, 3, 5\}), (4, \{0, 1, 4, 5\})\}$, o mulțime de perechi de elemente, primul element reprezentând numărul de cursuri ale specializării, al doilea element reprezentând mulțimea de cursuri predate la respectiva specializare.
- $Prec = \{(3, 2), (3, 4), (2, 5)\}$, o mulțime de perechi de cursuri, primul trebuie predat înaintea celui de-al doilea.
- $Preferințe = \{(1, 0), (0, 1)\}$, o mulțime de preferințe, primul număr reprezentând cursul, iar cel de-al doilea număr reprezentând perioada anului în care să nu fie predat. Perioada 0 a anului cuprinde perioadele 0 și 2, iar perioada 1 a anului cuprinde perioadele 1 și 3, dat fiind faptul că fiecare din cei doi ani are două perioade.

Lista de variabile:

- $\forall i \in [0, 6), P_i \in [0, 4)$.
- $\forall (i, j) \in [0, 4) \times [0, 6), B_{ij} \in \{0, 1\}$.
- $\forall i \in [0, 3), \forall j \in [0, 4), L_{ij} \in [0, 50]$.
- $\forall i \in [0, 3), \forall j \in [0, 4), Ldist_{ij} \in [0, 30]$.
- $\forall i \in [0, 6), PrefP_i \in \{0, 1\}$.

Constrângerile ce sunt aplicate asupra variabilelor sunt următoarele:

- $\forall (i, j) \in \text{Prec}: P_i < P_j$.
- $\forall s \in [0, 3), \forall j \in [0, 4): L_{sj} \in [0, 30]$.
- $\forall s \in [0, 3), \forall j \in [0, 4): N_{s_{sj}} \in [0, 2]$.
- $\forall i \in [0, 6) \text{ si } \forall j \in [0, 4): (P_i = j) \Leftrightarrow (B_{ji} = 1)$.

Soluția instanței de mai sus este:

$$P = [2, 3, 1, 0, 1, 2].$$

Cu alte cuvinte:

- Cursul 0 se predă în perioada 2.
- Cursul 1 se predă în perioada 3.
- Cursul 2 se predă în perioada 1.
- Cursul 3 se predă în perioada 0.
- Cursul 4 se predă în perioada 1.
- Cursul 5 se predă în perioada 2.

Fiecărui curs îi este atribuită o perioadă, astfel încât sunt satisfăcute toate condițiile și toate constrângerile cu privire la numărul total de credite per perioadă și numărul de cursuri per perioadă, pentru fiecare specializare.

Astfel situația specializărilor este următoarea:

- La specializarea 0:
 - În perioada 0, se predă cursul 3.
 - În perioada 1, se predă cursul 2.
 - În perioada 2, se predă cursul 0.
 - În perioada 3, se predă cursul 1.

- La specializarea 1:
 - În perioada 0, se predă cursul 3.
 - În perioada 1, se predă cursul 2.
 - În perioada 2, se predă cursul 5.
 - În perioada 3, se predă cursul 1.
- La specializarea 2:
 - În perioada 0, nu se predă niciun curs.
 - În perioada 1, se predă cursul 4.
 - În perioada 2, se predau cursurile 0 și 5.
 - În perioada 3, se predă cursul 1.

Numărul total mediu de credite per perioadă, pentru fiecare din cele trei specializări se calculează aplicând formula:

$$Lm_s = \frac{\sum_{i=0}^5 S_{si} \times w_i}{4}.$$

$$Lm_0 = ((5+4)+5+0+10)/4 = 6$$

$$Lm_1 = (4+5+7+10)/4 = 6.5$$

$$Lm_2 = (5+20+7+10)/4 = 10.5$$

Costul dezechilibrului necesarului de efort de învățare este calculat folosind formula:

$$Cs = \sum_{s=0}^2 \sum_{j=0}^3 |Lm_s - L_{sj}|.$$

$$Cs = (|6-5| + |6-5| + |6-4| + |6-10|) + (|6.5 - 4| + |6.5 - 5| + |6.5 - 7| + |6.5 - 10|) + (|10.5 - 12| + |10.5 - 20| + |10.5 - 0| + |10.5 - 10|) = 8 + 8 + 22 = 38$$

În cazul în care s-ar încălca o preferință cu privire la cursul i , valoarea penalizării ar fi egală cu dublul numărului de credite asociat cursului i . Dar, deși sunt constrângeri slabe, sunt respectate toate preferințele profesorilor, astfel încât:

$$Cp = 0.$$

Costul total este suma dintre costul dezechilibrului și costul preferințelor:

$$C = 38 + 0 = 38.$$

Costul total al soluției optime este 38.

2 Programarea bazată pe constrângeri

Algoritmul de rezolvare a GBACP a fost implementat prin intermediul paradigmei programării bazate pe constrângeri folosite “pentru descrierea declarativă și rezolvarea eficientă a problemelor, mai ales combinatoriale, de dimensiuni mari în special în domeniile de planificare și programare” (6).

Problemele de Satisfacere a Constrângerilor (CSPs) sunt rezolvate în general folosind variante ale tehnicilor: *backtracking*, propagarea constrângerilor și căutare locală.

O problemă de satisfacere a constrângerilor este definită printr-o mulțime de variabile, fiecare cu propriul domeniu de valori, și o mulțime de constrângeri.

Definiția clasică a unei probleme de satisfacere a constrângerilor este după cum urmează. O CSP notată P este un triplu $P = \langle X, D, C \rangle$ unde X este un n -tuplu de variabile $X = \langle x_1, x_2, \dots, x_n \rangle$, D este un n -tuplu de domenii de valori $D = \langle D_1, D_2, \dots, D_n \rangle$ astfel încât $x_i \in D_i$, C este un t -tuplu de constrângeri $C = \langle C_1, C_2, \dots, C_t \rangle$. O constrângere C_j este o pereche $\langle R_{S_j}, S_j \rangle$ unde R_{S_j} este o relație între variabilele din $S_j = \text{domeniu}(C_j)$. Cu alte cuvinte, C_j este o submulțime a produsului cartezian dintre domeniile variabilelor din S_j . O soluție a CSP P este un n -tuplu $A = \langle a_1, a_2, \dots, a_n \rangle$ unde $a_i \in D_i$ și fiecare C_j este satisfăcută, conform relației R_{S_j} atunci când variabilele din S_j iau valorile corespunzătoare din A . (7)

Potrivit documentației Choco-solver, o constrângere este o formulă logică care definește combinațiile permise de valori pentru o mulțime de variabile, altfel spus, restricțiile asupra variabilelor care trebuie respectate pentru a obține o soluție fezabilă. O constrângere este echipată cu o mulțime de algoritmi de filtrare, numiți propagatori. Un propagator elimină, din domeniile variabilelor țintă, valori care nu pot corespunde unei combinații valide de valori. O soluție a problemei este o atribuire variabilă-valoare care respectă toate constrângerile.

Spațiul de căutare indus de domeniile variabilelor este egal cu $S = |D_1| \times |D_2| \times \dots \times |D_n|$, unde D_i este domeniul variabilei x_i . Aproape niciodată, propagarea constrângerilor nu este suficientă pentru a construi o soluție, adică să elimine toate valorile în afara uneia singure din domeniul fiecărei variabile. Astfel, spațiul de căutare trebuie explorat folosind una sau mai multe strategii de explorare.

În Choco-solver 4, variabilele au, în funcție de tipul lor, strategii implicite de căutare:

- variabilele întregi și booleene: *intVarSearch(ivars)*.
- variabilele mulțime: *setVarSearch(svars)*.
- variabilele reale: *realVarSearch(rvars)*.
- variabila obiectiv: limita inferioară sau limita superioară, în funcție de direcția de optimizare. (8)

Cu ajutorul bibliotecii Java Choco-solver, s-a creat un model pentru GBACP utilizând variabile și constrângeri. Instanțele *CSP* sunt rezolvate prin satisfacerea modelului creat.

3 Descrierea aplicației

3.1 Arhitectura aplicației și tehnologii utilizate

Aplicația web dezvoltată are o structură *Model-View-Controller*. *MVC* este un șablon arhitectural în ingineria *software* și pune accentul pe separarea dintre gestionarea (stocarea, sortarea și prelucrarea) datelor aplicației și afișarea lor (interfața cu utilizatorul). Această separare permite o mai bună divizare a muncii și o mai bună mentenanță. Cele trei părți ale șablonului de proiectare *software MVC* pot fi caracterizate în felul următor:

1. Modelul: gestionează datele.
2. Prezentarea: se ocupă de aspect și afișare.
3. Controlul: direcționează comenzi către componentele model și prezentare. (9)

În subcapitolele următoare se descrie fiecare dintre cele trei componente arhitecturale cu diferite subcomponente care intră în alcătuirea ei, cum ar fi clasele proiectului.

Tehnologiile utilizate sunt:

- *Apache Tomcat 9.0* - serverul web open-source care o găzduiește aplicația.
- *Oracle Database 21c Express Edition* - o bază de date complexă, dar ușor de utilizat unde sunt stocate datele de autentificare a utilizatorilor.
- *Apache Maven* - un instrument de administrare și construire a proiectelor *software*. Conceptul central în *Maven* este acela de *POM (Project Object Model)*, un document XML care descrie un proiect sau un modul al acestuia. Mai multe dependențe ale aplicației au fost adăugate prin *Maven*.
- Java - limbaj de programare orientat pe obiecte popular și simplu.
- *Choco-solver* - bibliotecă open-source pentru programare bazată pe constrângeri.
- HTML, CSS și JavaScript – limbajele cu ajutorul cărora s-a dezvoltat cu ușurință o interfață intuitivă și flexibilă.
- *Jakarta Server Pages (JSP; în trecut, JavaServer Pages)* - o colecție de tehnologii prin care au fost create pagini web dinamic generate pe baza HTML.
- *Servlet* – o tehnologie a platformei *Java Enterprise Edition* prin care se primesc și se răspunde la cereri *HTTP* venite din partea clienților web, extinzând capacitățile server-ului web.
- *JSON-Java (org.json)* – pachet de instrumente pentru codificarea și decodificarea datelor JSON.

3.2 Modelul aplicației

3.2.1 Clasa ProblemModel

O instanță CSP a problemei GBACP este reprezentată prin intermediul clasei *ProblemModel*. Această clasă conține mai multe atribute în care se memorează datele de intrare ale problemei, fiecare atribut având câte o metodă de tip *get* și o metodă de tip *set*. De asemenea, clasa *ProblemModel* poate fi instanțiată cu ușurință printr-un constructor care are ca parametru o dată de tipul *JSONObject*.

```
// nr ani
private int y;
// nr perioade ale anului
private int py;
// nr perioade
private int m;
// nr cursuri
private int n;
// nr specializari
private int k;
// min credite per perioada
private int a;
// max credite per perioada
private int b;
// min cursuri per perioada
private int c;
// max cursuri per perioada
private int d;
```

Fig. 1 Atribute ProblemModel

```
// nr preconditii
private int nPrec;
// nr preferinte
private int nPref;

// nr credite asociat fiecarui curs
private int[] w;
// sp[s][i] = 1 <=>
// la specializarea s se preda cursul i
private int[][] sp;

// cursul prec[i][0] inaintea
// cursului prec[i][1]
private int[][] prec;
// pref[i][j] = 1 <=> se prefera
// sa nu se predea cursul i in perioada j
private int[][] pref;
```

Fig. 2 Atribute ProblemModel

Prin apelarea constructorului, se instanțiază un tablou unidimensional și trei tablouri bidimensionale. Pentru extragerea datelor necesare, parametrul *JSONObject* este parcurs prin metodele *getJSONArray*, *getJSONObject* și, într-un final, prin *getInt*, pentru a extrage numărul întreg dorit, sau *length*, pentru a afla numărul de elemente.

Astfel, se construiesc:

- *w*: ce conține numărul de credite pentru fiecare curs.

```
w = new int[n];
for(int i = 0; i < n; i++){
    w[i] = obj.getJSONArray("courses").getJSONObject(i).getInt("credits");
}
```

Fig. 3 Popularea vectorului w

- *sp*: matrice binară în care pentru fiecare curs, pentru fiecare specializare la care se predă cursul, se marchează cu valoarea 1 legătura dintre curs și specializare.

```
sp = new int[k][n];
for(int i = 0; i < n; i++){
    for(int j = 0; j < obj.getJSONArray("courses").getJSONObject(i).getJSONArray("id_curriculum").length(); j++){
        sp[obj.getJSONArray("courses").getJSONObject(i).getJSONArray("id_curriculum").getInt(j)][i] = 1;
    }
}
```

Fig. 4 Popularea matricei sp

- *prec*: în care se înregistrează pe fiecare linie câte o pereche de identificatori de cursuri aflate în relație de condiționare.

```
prec = new int[nPrec][2];
for(int i = 0; i < nPrec; i++){
    prec[i][0] = obj.getJSONArray("preconditions").getJSONObject(i).getInt("id_course1");
    prec[i][1] = obj.getJSONArray("preconditions").getJSONObject(i).getInt("id_course2");
}
```

Fig. 5 Popularea matricei prec

- *pref*: în care se înregistrează pe fiecare linie câte o pereche identificatorul cursului
- perioada în care se preferă să nu se predea cursul.

```
pref = new int[n][m];
for(int i = 0; i < obj.getJSONArray("preferences").length(); i++){
    for(int j = 0; j < y; j++){
        pref[obj.getJSONArray("preferences").getJSONObject(i).getInt("id_course")]
            [obj.getJSONArray("preferences").getJSONObject(i).getInt("year_period") + py*j] = 1;
    }
}
```

Fig. 6 Popularea matricei pref

3.2.2 Clasa UserModel

Cu ajutorul clasei *UserModel*, se memorează și gestionează datele de autentificare ale utilizatorului: identificator, nume și parolă.

3.2.3 Baza de date Oracle și clasa DatabaseConnection

Datele despre utilizatori sunt stocate permanent într-o bază de date Oracle Database 21c Express Edition, care este o ediție a liderului mondial în domeniul bazelor de date, dezvoltată, utilizată și distribuită gratuit. (10) Driver-ul Oracle *JDBC* necesar conectării la baza de date a fost adăugat ca și dependență prin Maven.

Funcția statică *getConnection()* a clasei *DatabaseConnection* returnează o instanță a clasei *java.sql.Connection* necesară la citirea și scrierea datelor.

```
public static Connection getConnection()
    throws ClassNotFoundException, SQLException {

    String hostName = "localhost";
    String sid = "XEPPDB1";
    String userName = "student";
    String password = "student";

    return getConnection(hostName, sid, userName, password);
}

public static Connection getConnection(String hostName, String sid,
    String userName, String password) throws ClassNotFoundException,
    SQLException {

    Class.forName("oracle.jdbc.driver.OracleDriver");

    // URL Connection for Oracle
    String connectionURL = "jdbc:oracle:thin:@" + hostName + ":1521/" + sid;

    Connection conn = DriverManager.getConnection(connectionURL, userName,
        password);
    return conn;
}
```

Fig. 7 Metoda getConnection

Clasa mai conține o metodă de închidere a conexiunii și o metodă de *rollback*.

3.2.4 Clasa ModelUtils și tabela Users

Tabela *users* a fost creată folosind comanda *create table* și conține trei câmpuri: identificatorul ca și cheie primară, numele de utilizator și parola.

La înregistrarea unui utilizator, clasa de control a aplicației apelează metoda *insertUser* a clasei *ModelUtils* care prin intermediul unor instrucțiuni SQL, în cazul în care

utilizatorul nu este deja înregistrat, adaugă datele de înregistrare într-o linie nouă din tabela *users*.

```
public static boolean insertUser(Connection conn, String username, String password) throws SQLException {

    String sql = "Select a.Id, a.Username, a.Password from users a " //
        + " where a.Username = ? and a.password= ?";

    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1, username);
    pstmt.setString(2, password);
    ResultSet rs = pstmt.executeQuery();

    if (!rs.isBeforeFirst() ) {
        sql = "Insert into users(id, username, password) values (seq_users.nextval, ?, ?)";
        pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, username);
        pstmt.setString(2, password);
        pstmt.executeUpdate();
        return true;
    }

    return false;
}
```

Fig. 8 Metoda insertUser

La autentificarea unui utilizator, clasa de control a aplicației apelează metoda *findUser* a clasei *ModelUtils* care prin intermediul unei comenzi SQL de citire, verifică dacă utilizatorul este înregistrat, caz în care creează un obiect *UserModel* cu datele utilizatorului și apoi îl returnează.

```
public static UserModel findUser(Connection conn, String username, String password) throws SQLException {

    String sql = "Select a.Id, a.Username, a.Password from users a " //
        + " where a.Username = ? and a.password= ?";

    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1, username);
    pstmt.setString(2, password);
    ResultSet rs = pstmt.executeQuery();

    if (rs.next()) {
        int id = rs.getInt("Id");
        UserModel user = new UserModel();
        user.setId(id);
        user.setUsername(username);
        user.setPassword(password);
        return user;
    }

    return null;
}
```

Fig. 9 Metoda findUser

3.3 Controlul aplicației

Clasele de control a aplicației sunt mai multe clase care moștenesc clasa *HttpServlet* și o clasă care conține algoritmul de rezolvare a problemei GBAC implementat cu ajutorul bibliotecii *Choco-solver*. Clasele *Servlet* gestionează apelurile de tip *GET* și *POST* către serverul aplicației, la diferite căi *Servlet*.

3.3.1 Modelarea problemei în Choco-solver

Un important rol în proiectarea algoritmului de rezolvare a problemei GBAC îl joacă biblioteca *Open-Source* java de programare bazată pe constrângeri, *Choco-solver*. (8) Prima versiune a *Choco-solver* a fost lansată în 2003. De atunci, a trecut prin mai multe reproiectări majore, ultima fiind în anul 2013, când acest *API* de modelare s-a stabilizat (spre exemplu, cu privire la declararea de variabile și constrângeri). *Choco* este unul dintre cele mai rapide solver-e de pe piață. *Choco-solver 4* cuprinde:

- diferite tipuri de variabile (întregi, booleene, mulțimi, reale);
- multe constrângeri (*alldifferent*, *count*, *nvalues*, etc.);
- o căutare configurabilă (*custom search*, *activity-based search*, *large neighborhood search*, etc.);
- explicații de conflict (*conflict-based back jumping*, *dynamic backtracking*, *path repair*, etc.). (11)

Această bibliotecă a fost adăugată ca și dependență în fișierul *pom.xml* și prin Maven la proiectul aplicației.

```
<dependency>
  <groupId>org.choco-solver</groupId>
  <artifactId>choco-solver</artifactId>
  <version>4.10.8</version>
</dependency>
```

Fig. 10 Dependența Choco-solver

Apoi, în fișierul clasei *GbacpSolver* au fost importate mai multe pachete ale *org.chocosolver*.

```
import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solution;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.variables.BoolVar;
import org.chocosolver.util.tools.ArrayUtils;
```

Fig. 11 Importarea pachetelor Choco-solver

Obiectul *Model* este componenta cheie în care se stochează variabilele declarate, constrângerile publicate și care oferă accesul la *Solver*. *Solver-ul* este responsabil de alternarea propagării constrângerilor cu căutarea, și eventual învățarea, pentru a determina soluții. Clasa *GbacpSolver* conține atributele *model* și *solver* declarate astfel:

```
private Solver solver;
private Model model;
```

Fig. 12 Declararea obiectelor solver și model

Construirea obiectului *Model* trebuie făcută înaintea oricăror altor instrucțiuni de modelare, prin apelarea metodei *createSolver*. În cadrul aceleiași metode se obține din model obiectul de tip *Solver*.

```
public void createSolver() {
    model = new Model("GBACP");
    solver = model.getSolver();
}
```

Fig. 13 Construirea obiectelor model și solver

Celelalte metode sunt: *buildModel*, *precondition* și *solve*. Metodele *buildModel* și *solve* iau ca și parametru un obiect *prob* al clasei *ProblemModel* în care sunt memorate toate datele de intrare ale problemei.

Clasa are ca atribute mai multe variabile ale modelului asupra cărora vor fi aplicate mai multe constrângeri. Variabilele sunt următoarele:

```

// prefP[i] = 1 <=> cursul i este atribuit unei perioade nepreferate
private BoolVar[] prefP;

// perioada asociata fiecarui curs
private IntVar[] p;
// x[i][j] = 1 <=> cursul j se predă in perioada i
private BoolVar[][] x;
// l[s][j] = nr total de credite al perioadei j, specializare s
private IntVar[][] l;
// distanta între l[perioada] si l mediu [specializare]
private IntVar[][] lDist;

// cost dezechilibru
private IntVar cs;
// cost preferinte
private IntVar cp;
// cost total, al functiei obiectiv
private IntVar ct;

private Solver solver;
private Model model;

```

Fig. 14 Variabile

Prin metoda *buildModel* se declară variabilele și se publică constrângerile. Fiecare variabilă este declarată prin definirea domeniului variabilei -mulțimii de valori pe care le poate lua- în model.

```

p = model.intVarArray("P", prob.getN(), 0, prob.getM() - 1);
x = model.boolVarMatrix("B", prob.getM(), prob.getN());
l = model.intVarMatrix("L", prob.getK(), prob.getM(), prob.getA(), prob.getB());
lDist = model.intVarMatrix("Ldist", prob.getK(), prob.getM(), 0, prob.getB()-prob.getA());
prefP = model.boolVarArray("PrefP", prob.getN());

cs = model.intVar("Cs", 0, 10000);
cp = model.intVar("Cp", 0, 10000);
ct = model.intVar("C", 0, 10000, true);

```

Fig. 15 Definirea domeniilor de valori

O constrângere este echipată cu unul sau mai mulți algoritmi de filtrare denumiți propagatori. Un propagator elimină din domeniile variabilelor, valori care nu pot corespunde unei combinații valide de valori. O soluție a problemei este o atribuire variabilă-valoare, pentru toate variabilele, care respectă toate constrângerile. (12)

Pentru modelarea GBACP utilizând *Choco-solver* au fost implementate următoarele constrângeri:

```

// x[i][j] = 1 <=> p[j] = i
for (int i = 0; i < prob.getM(); i++) {
    for (int j = 0; j < prob.getN(); j++) {
        model.ifThenElse(x[i][j],
            model.arithm(p[j], "=", i),
            model.arithm(p[j], "!=", i)
        );
    }
}

```

Fig. 16 Constrângere

- se definește o constrângere de conectare a celor două forme de reprezentare a atribuirilor curs-perioadă: matricea x de variabile booleene și vectorul p de variabile întregi. Mai exact, prin funcția *ifThenElse*, în cazul în care în matricea x pe linia i care semnifică perioada, coloana j care semnifică numărul cursului, avem valoarea "true", atunci în vectorul p de perioade, pe poziția j , se forțează valoarea i . În caz contrar, pe poziția j a vectorului trebuie să fie o valoare diferită de i .

```

int[] sW = new int[prob.getN()];
for (int s = 0; s < prob.getK(); s++) {
    for(int j=0; j<prob.getN(); j++)
        sW[j] = sp[s][j] * w[j];
    for (int i = 0; i < prob.getM(); i++) {
        // pt specializarea s, perioada i, nr cursuri >= c
        model.scalar(x[i], sp[s], ">=", prob.getC()).post();
        // pt specializarea s, perioada i, nr cursuri <= d
        model.scalar(x[i], sp[s], "<=", prob.getD()).post();
        // l[specializare][perioada] = nr total de credite
        model.scalar(x[i], sW, "=", l[s][i]).post();
    }
}

```

Fig. 17 Constrângeri

- pe baza matricii sp care reflectă legătura dintre specializări și cursuri, se creează vectorul sW care conține numărul de credite, extras din vectorul w , pentru fiecare curs de la specializarea s ; în cazul în care cursul nu este predat la specializarea s , se atribuie valoarea 0. Prin metoda *scalar*, se publică constrângeri asupra unor sume ponderate. Astfel, se impune ca numărul total de credite $l[s][i]$ al perioadei i de la specializarea s să fie egal cu suma ponderată dintre vectorul binar $x[i]$ de cursuri ale perioadei i și vectorul sW (se adaugă numărul de credite al fiecărui curs de la specializarea s și din perioada i). Numărul de cursuri de la specializarea s este egal cu suma ponderată dintre vectorul $x[i]$ și vectorul binar $sp[s]$ de cursuri ale specializării. Se publică două constrângeri pentru ca numărul de cursuri să fie mai mare sau egal decât numărul minim de cursuri dat c și mai mic sau egal decât numărul maxim de cursuri dat d .

```
// p[curs1] < p[curs2]
for(int i = 0; i < prob.getnPrec(); i++)
    precondition(prob.getPrec()[i][0], prob.getPrec()[i][1]);
```

Fig. 18 Apelarea metodei precondition

```
private void precondition(int course1, int course2) {
    model.arithm(p[course1], "<", p[course2]).post();
}
```

Fig. 19 Metoda precondition a constrângerii

- relațiile de condiționare dintre cursuri se modelează cu ușurință prin constrângeri de tipul "*mai mic decât*".

```
// Lm[s] = nr total mediu de credite, specializare s
int[] Lm = new int[prob.getK()];
for(int s = 0; s < prob.getK(); s++) {
    for(int i = 0; i < prob.getN(); i++)
        Lm[s] = Lm[s] + sp[s][i] * w[i];
    Lm[s] = Lm[s] / prob.getM();

    // se calculeaza distanta dintre Lm[s] si l[s][perioada]
    for(int j = 0; j < prob.getM(); j++)
        lDist[s][j] = l[s][j].dist(Lm[s]).intVar();
}
// cs = suma deviatiilor lDist
model.sum(ArrayUtils.flatten(lDist), "=", cs).post();
```

Fig. 20 Constrângerea costului dezechilibrului

- Numărul total mediu de credite de la specializarea s se obține însumând numerele de credite $w[i]$ de la specializarea s ($sp[s][i] = 1$) și împărțind rezultatul la numărul de perioade m . Apoi, în matricea $lDist$ se memorează distanța dintre numărul total de credite pentru fiecare perioadă și numărul total mediu de credite, pentru fiecare specializare. Distanța este calculată prin aplicarea funcției *dist* asupra fiecărui element al matricei l , rezultatul fiind convertit într-o variabilă întreagă a modelului utilizând funcția *intVar*. Astfel $lDist$ este de fapt o matrice de deviații, iar prin însumarea lor se obține costul dezechilibrului specializărilor, notat cs . Metoda *flatten* a fost aplicată pe matricea $lDist$ deoarece primul parametru al metodei *sum* trebuie să fie vector.

```

for(int i = 0; i < prob.getN(); i++) {
    // prefP[i] = pref[i][p[i]]
    model.element(prefP[i], pref[i], p[i]).post();
}
int[] prefW = new int[prob.getN()];
for(int i=0; i<prob.getN(); i++)
    // penalizare = 2 * nr credite
    prefW[i] = 2 * w[i];
// cp = suma penalizarilor
// prefP[curs] = 1 => penalizare
model.scalar(prefP, prefW, "=", cp).post();

model.arithm(cs, "+", cp, "=", ct).post();

```

Fig. 21 Constrângeri

- vectorul binar *prefP* de preferințe încălcate este populat pe baza matricei de preferințe *pref*; dacă se regăsește valoarea 1 pentru cursul *i* și perioada *p[i]* selectată de *solver*, atunci o preferință a fost încălcată și prin intermediul metodei *element* variabila *prefP[i]* va fi constrânsă să ia valoarea „true”. În caz contrar, variabila *prefP[i]* va fi constrânsă să ia valoarea „false”. Costul preferințelor încălcate, memorat în variabila *cp*, se constrânge, prin metoda *scalar*, a fi egal cu suma ponderată aplicată pe vectorii *prefP* și *prefW*. În vectorul *prefW* se stochează penalizarile posibile, penalizarea posibilă asociată unui curs fiind egală cu dublul numărului de credite.
- costul total este egal cu suma dintre costul dezechilibrului și costul preferințelor.

În metoda *solve* a clasei *GbacpSolver* variabila costului total denumită *ct* este fixată ca și variabilă obiectiv, costurile *cs* și *cp* fiind agregate într-o singură variabilă obiectiv. GBACP este modelată ca o problemă de optimizare. Astfel, *solver-ul* calculează o soluție, apoi se stochează valoarea variabilei obiectiv și se publică constrângerea ca următoarea soluție să fie mai bună decât cea curentă, adică, în cazul de față, variabila costului total să ia o valoare mai mică decât cea curentă. Acest lucru a fost realizat prin apelarea metodei *setObjective* a obiectului *model*, având ca parametri direcția de optimizare specificată prin *Model.MINIMIZE* și variabila *ct*.

Căutarea are setată o limită temporală de 10 secunde. Într-o buclă *while*, câte o soluție este calculată la apelarea metodei *solve* a obiectului *solver* și înregistrată prin apelarea metodei *record*. Înregistrarea soluției se face folosind un obiect al clasei *Solution* care face legătura între fiecare variabilă și valoarea sa curentă. La ieșirea din buclă, în obiectul *solution* vor fi memorate valorile corespunzătoare ultimei soluții care va fi chiar soluția optimă. O

astfel de valoare poate fi accesată prin metoda *getIntVal* a obiectului *solution* având ca parametru variabila valorii dorite. Prin *solution.exists()* se verifică dacă s-a găsit măcar o soluție. În acest caz, funcția *solve* a clasei *GbacpSolver* returnează într-un singur vector valorile vectorului de perioade *p* și, pe ultimele poziții, valorile celor trei costuri. În caz contrar, se returnează un vector gol.

```
public int[] solve(ProblemModel prob) {
    // minimizarea costului total (variabilei obiectiv)
    model.setObjective(Model.MINIMIZE, ct);

    solver.limitTime("10s");

    Solution solution = new Solution(model);
    while (solver.solve()) {
        solution.record();
    }
    if(solution.exists()){
        // se trimite p, pt solutia optima
        int[] solP = new int[prob.getN()+3];
        for(int j = 0; j < prob.getN(); j++)
            solP[j] = solution.getIntVal(p[j]);
        // se trimit costurile: dezechilibrului, preferintelor si total
        solP[prob.getN()] = solution.getIntVal(cs);
        solP[prob.getN()+1] = solution.getIntVal(cp);
        solP[prob.getN()+2] = solution.getIntVal(ct);
        return solP;
    }
    return new int[0];
}
```

Fig. 22 Metoda solve

3.3.2 Servlet-uri: RegisterServlet, LoginServlet, LogoutServlet, GbacpServlet

Aplicația dezvoltată conține patru subclase ale *HttpServlet*. Un *HttpServlet* este o componentă care se bazează pe un model de programare cerere-răspuns și care oferă metode, cum ar fi *doGet* și *doPost*, pentru a gestiona servicii specifice *HTTP*, extinzând capacitățile server-ului web care gazduiește aplicația. Execuția *HttpServlet-urilor* presupune șase pași:

- 1) Clientul trimite o cerere către server-ul web.
- 2) Server-ul web primește cererea.
- 3) Server-ul web transmite cererea *HttpServlet-ului* corespunzător.
- 4) *HttpServlet-ul* procesează cererea și generează un răspuns ca și ieșire.
- 5) *HttpServlet-ul* trimite răspunsul înapoi server-ului web.
- 6) Server-ul web trimite răspunsul înapoi clientului și navigatorul clientului îl afișează pe ecran. (13)

Utilizând adnotări `@WebServlet` se creează legături între clase și anumite șabloane URL. Astfel șablonul URL al clasei `RegisterServlet` este specificat în interiorul fișierului clasei în felul următor:

```
@WebServlet(name = "register", urlPatterns = "/register")
```

Atunci când se trimite o cerere *GET* către `RegisterServlet`, se apelează metoda `doGet` care, prin `getRequestDispatcher`, trimite cererea mai departe la fișierul *JSP* de pe server.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.getRequestDispatcher("/WEB-INF/views/registerView.jsp").forward(request, response);
}
```

Fig. 23 Metoda `doGet`

Atunci când se trimite o cerere *POST* către `RegisterServlet`, se apelează metoda `doPost` care creează o conexiune la baza de date și apoi se încearcă adăugarea datelor unui nou utilizator prin apelarea funcției statice `insertUser` a clasei `ModelUtils`, clasă responsabilă de executarea instrucțiunilor asupra tabelelor bazei de date. Parametrii cererii *GET* sunt accesați prin `request.getParameter`. În cazul în care adăugarea se încheie cu succes se trimite un răspuns de redirectare prin `response.sendRedirect`, la un URL alcătuit din contextul căii (`request.getContextPath`) și șablonul URL al `LoginServlet`. În caz contrar, se retrimite fișierul de înregistrare prin apelarea `doGet`. În cazul unei excepții se execută un *rollback*. La final, se închide conexiunea la baza de date.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    Connection conn = null;
    try {
        conn = DatabaseConnection.getConnection();
        conn.setAutoCommit(false);

        if(ModelUtils.insertUser(conn, request.getParameter("username"),
            request.getParameter("password")) == true) {
            response.sendRedirect(request.getContextPath() + "/login");
        }
        else {
            doGet(request, response);
        }

        conn.commit();
    } catch (Exception e) {
        e.printStackTrace();
        DatabaseConnection.rollback(conn);
        throw new ServletException();
    } finally {
        DatabaseConnection.closeConnection(conn);
    }
}
```

Fig. 24 Metoda `doPost`

LoginServlet are ca șablon URL *"/login"*, iar metoda *doGet* trimite cererea la template-ul *JSP* de pe server, *loginView*. Metoda *doPost* interacționează cu baza de date asemănător metodei *doPost* a *RegisterServlet*, cu o diferență: prin apelarea *ModelUtils.findUser* se verifică dacă utilizatorul este înregistrat, caz în care metoda returnează un obiect *UserModel* cu datele utilizatorului; se creează o sesiune prin *getSession* și apoi obiectul *user* este stocat în sesiune prin *setAttribute*. Pe baza acestui obiect din sesiune se va controla accesul clientului la resursele de pe server. Se trimite un răspuns de redirectare la *GbacpServlet* pe baza șablonului său URL *"/gbacpForm"*.

```
UserModel user = ModelUtils.findUser(conn, request.getParameter("username"),
request.getParameter("password"));
if(user != null) {
    HttpSession session = request.getSession();
    session.setAttribute("user", user);
    response.sendRedirect(request.getContextPath() + "/gbacpForm");
}
```

Fig. 25 Autentificarea utilizatorului

LogoutServlet are ca șablon URL *"/logout"*. Metoda *doGet* elimină obiectul *user* din sesiune prin *invalidate* și apoi trimite o comandă clientului de a executa o redirectare la *LoginServlet* și, într-un final, către pagina de autentificare.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.getSession().invalidate();
    response.sendRedirect(request.getContextPath() + "/login");
}
```

Fig. 26 Metoda *doGet* de deconectare a utilizatorului

GbacpServlet are ca șablon URL *"/gbacpForm"*. Metoda *doGet* verifică dacă obiectul *user* este prezent în sesiunea curentă. Dacă utilizatorul este autentificat se trimite cererea la template-ul *JSP* de pe server, *gbacpFormView*. Dacă nu este autentificat se trimite un răspuns de redirectare la *LoginServlet*.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    HttpSession session = request.getSession();
    if(session.getAttribute("user") != null)
        request.getRequestDispatcher("/WEB-INF/views/gbacpFormView.jsp")
        .forward(request, response);
    else
        response.sendRedirect(request.getContextPath() + "/login");
}
```

Fig. 27 Metoda *doGet*

Metoda *doPost* transformă *body-ul* cererii într-un *String* prin apelarea metodei *bodyToString*, pentru ca apoi, pe baza acestui *String* să se instanțieze clasa *JSONObject*. Pe baza acestei instanțe se creează un obiect *p* al clasei *ProblemModel* prin care se reprezintă toate datele de intrare ale problemei GBAC. Se construiește un obiect al clasei *GBACPSolver*, apoi prin apelarea *createSolver* se obține *solver-ul* bibliotecii *Choco-solver*, prin *buildModel(p)* se definesc domeniile de valori ale variabilelor și se publică constrângerile, modelându-se problema GBAC într-o problemă de satisfacere a constrângerilor. Prin *solve(p)* se obține soluția optimă sau cea mai bună soluție găsită în timpul limită. Metoda răspunde prin *getWriter().print* cu un obiect JSON creat cu ajutorul *JSONObject*, conținând un atribut cu numele "*solution*" și având ca valoare vectorul soluției găsite. S-a ajuns la această valoare prin crearea unui obiect *JSONArray* din vectorul de variabile *int* astfel: *new JSONArray(Arrays.stream(sol).boxed().toList())*.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String body = bodyToString(request);
    JSONObject obj = new JSONObject(body);
    ProblemModel p = new ProblemModel(obj);

    GbacpSolver gbacp = new GbacpSolver();
    gbacp.createSolver();
    gbacp.buildModel(p);
    int[] sol = gbacp.solve(p);

    JSONObject objResponse = new JSONObject();
    objResponse.put("solution", new JSONArray(Arrays.stream(sol).boxed().toList()));
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    response.getWriter().print(objResponse);
    response.getWriter().flush();
}
```

Fig. 28 Metoda doPost

Construirea unui *String* pornind de la *body-ul* cererii s-a realizat cu ajutorul claselor *StringBuilder*, *BufferedReader*, *InputStream* și *InputStreamReader*. În bucla *while* sunt citiți câte maxim 128 de octeți. Atunci când *bufferedReader.read* returnează valoarea -1 înseamnă că s-a ajuns la sfârșitul conținutului obiectului *bufferedReader*. Într-un final prin apelarea metodei *toString* asupra obiectului *stringBuilder* se obține un obiect al clasei *String*.

```

StringBuilder stringBuilder = new StringBuilder();
BufferedReader bufferedReader = null;

try {
    InputStream inputStream = request.getInputStream();
    if (inputStream != null) {
        bufferedReader = new BufferedReader(new InputStreamReader(inputStream));
        char[] charBuffer = new char[128];
        int bytesRead = bufferedReader.read(charBuffer);
        // bytesRead = -1 => s-a citit tot din bufferedReader (EOF)
        while (bytesRead > 0) {
            stringBuilder.append(charBuffer, 0, bytesRead);
            bytesRead = bufferedReader.read(charBuffer);
        }
    }
}

```

Fig. 29 Cod parțial al metodei bodyToString

3.4 Prezentarea aplicației

Interfața aplicației web se bazează pe trei template-uri *JSP* ca și componente ale nivelului de prezentare: *registerView*, *loginView* și *gbacpFormView*.

Jakarta Server Pages (JSP; în trecut, *JavaServer Pages*) este o colecție de tehnologii care ajută la crearea de pagini web dinamic generate pe baza HTML (în cazul acestei aplicații), XML, SOAP sau alte tipuri de documente. Lansat în 1999 de *Sun Microsystems*, *JSP* este similar cu PHP și ASP, dar folosește limbajul de programare Java. Într-un fișier *JSP*, conținutul de marcare web HTML poate fi intercalat cu codul Java. Pagina rezultată este compilată și executată pe server pentru a livra documentul. *Container-ul Web* creează mai multe obiecte *JSP* implicite cum ar fi *request*, *response*, *session*, *application*, *config*, *page*, *pageContext*, *out* și *exception*. (14)

Fiecare componentă *JSP* are asociată câte un fișier CSS cu rolul de a defini stilurile elementelor HTML.

Pagina de introducere a datelor problemei GBAC și de afișare a soluției problemei este construită pornind de la componenta *gbacpFormView.jsp*. Pentru ca interfața web cu utilizatorul să fie cât mai interactivă, această componentă are atașat un fișier JavaScript denumit *gbacpForm*. Astfel, se implementează ca la declanșarea anumitor evenimente ale anumitor elemente HTML, prin manipularea DOM, să aibă loc o actualizare în interfață a structurii de specializări și cursuri, a listelor de precondiții și preferințe.

Există două modalități posibile de a introduce datele problemei: prin formular sau prin încărcarea unui fișier.

Datele de intrare ale problemei sunt reprezentate prin intermediul unui obiect JavaScript *prob* care conține patru liste.

```
var prob = new Object();

prob.curricula = [];
prob.courses = [];
prob.preconditions = [];
prob.preferences = [];
```

Fig. 30 Obiectul prob

Prin apelări ale *document.getElementById* se obțin, pe baza identificatorilor, elementele HTML *input* din care se pot extrage valorile introduse de utilizator.

prob.curricula este o listă de titluri de specializări de tip șir de caractere.

```
prob.curricula.push(document.getElementById("curriculum_title").value);
```

Fig. 31 Adăugarea unei specializări în listă

Se creează un obiect *course* și se adaugă în listă.

```
let course = {
    "course_title": document.getElementById("course_title").value,
    "credits": document.getElementById("credits").value,
    "id_curriculum": [(document.getElementById("id_curriculum").value - 1)]
};
prob.courses.push(course);
```

Fig. 32 Adăugarea unui curs în listă

Se adaugă un curs existent la o nouă specializare.

```
prob.courses[document.getElementById("id_course_old").value - 1]
    .id_curriculum.push(document.getElementById("id_curriculum_old").value-1);
```

Fig. 33 Adăugarea unui curs existent în obiectul prob

Se creează un obiect *precondition*, incrementându-se valoarea identificatorului, și se adaugă în listă.

```

let precId = (prob.preconditions.length === 0) ? 0 :
prob.preconditions[prob.preconditions.length - 1].id + 1;
let precondition = {
    "id": precId,
    "id_course1": (document.getElementById("id_course1").value - 1),
    "id_course2": (document.getElementById("id_course2").value - 1)
};
prob.preconditions.push(precondition);

```

Fig. 34 Adăugarea unei precondiții în listă

Se creează un obiect *preference*, incrementându-se valoarea identificatorului, și se adaugă în listă.

```

let prefId = (prob.preferences.length === 0) ? 0 :
prob.preferences[prob.preferences.length - 1].id + 1;
let preference = {
    "id": prefId,
    "id_course": (document.getElementById("id_course").value - 1),
    "year_period": (document.getElementById("year_period").value - 1)
};
prob.preferences.push(preference);

```

Fig. 35 Adăugarea unei preferințe în listă

De asemenea, înainte de trimiterea formularului se creează șase variabile importante:

```

prob.years = document.querySelector("input[name=years]").value;
prob.yearPeriods = document.querySelector("input[name=year_periods]").value;
prob.minCredits = document.querySelector("input[name=min_credits]").value;
prob.maxCredits = document.querySelector("input[name=max_credits]").value;
prob.minCourses = document.querySelector("input[name=min_courses]").value;
prob.maxCourses = document.querySelector("input[name=max_courses]").value;

```

Fig. 36 Alte date ale problemei

Datele pot fi introduse prin formular sau prin încărcarea unui fișier. Fișierul este citit prin apelarea metodei *readAsBinaryString* a obiectului clasei *FileReader*. Datele conținutului fișierului sunt extrase prin funcția *parse*.

```

const fileInput = document.getElementById('gbacpFile');
fileInput.addEventListener('change', function(e) {
    if(e.target.files.length) {
        let file = e.target.files[0];
        let reader = new FileReader();
        reader.addEventListener('load', function (e2) {
            let resultData = e2.target.result;
            parse(resultData);
        });

        reader.readAsBinaryString(file);
    }
});

```

Fig. 37 Funcția asociată încărcării unui fișier

Prin *parse*, inițial datele din fișier sunt separate în funcție de caracterele "\n" și " " și stocate într-o matrice *parsed*. Apoi, cunoscând structura fișierului, datele dorite sunt extrase din *parsed* și mutate în obiectul *prob*.

```

function parse(data) {
    let parsed = [];

    let newLinebrk = data.split("\n");
    for(let i = 0; i < newLinebrk.length; i++) {
        parsed.push(newLinebrk[i].split(" "));
    }
}

```

Fig. 38 Cod parțial al funcției parse

Prin *AJAX*, prin metoda *fetch*, se trimite o cerere *POST* către server cu obiectul *prob*. Pentru serializarea obiectului JavaScript *prob* se folosește *JSON.stringify*. De la server se primește un răspuns din care se obține un obiect JSON conținând un vector *data.solution* cu atribuirile curs-perioadă care sunt afișate sub forma unui tabel și costurile dezechilibrului, preferințelor și cel total afișate separat.

```

fetch("http://localhost:8080/gbacpApp/gbacpForm", {
    method: 'post',
    body: JSON.stringify(prob),
    headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
    }
}).then((response) => {
    return response.json();
})
.then((data) => {
    if(data.solution.length !== 0) {

```

Fig. 39 Apel AJAX prin fetch

3.5 Ghidul de utilizare a aplicației

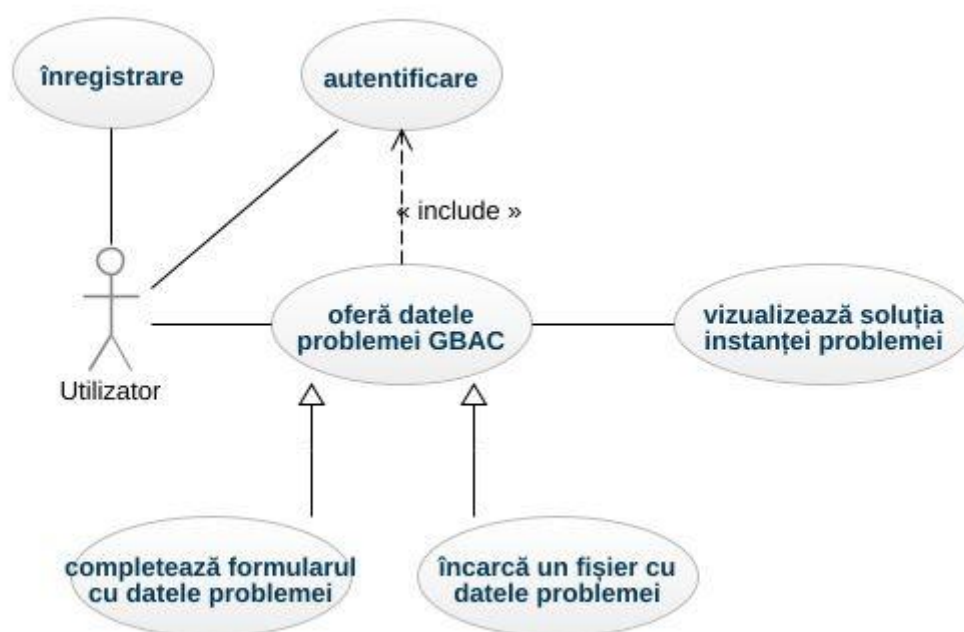


Fig. 40 Diagrama Use case

Aplicația web dezvoltată pune la dispoziția utilizatorilor o modalitate ușoară și rapidă de concepere a unor planuri de studii universitare echilibrate, introducând instanțe GBACP printr-o interfață cu utilizatorul intuitivă și flexibilă, instanțele fiind rezolvate cu ajutorul *solver-ului CSP Choco-solver*.

Înainte de introducerea datelor problemei este necesară înregistrarea și autentificarea pe baza unui nume și a unei parole.

Inregistrare

Nume de utilizator:

Parola:

[Inregistreaza-te](#)

[Conecteaza-te](#)

Fig. 41 Formular de înregistrare

Autentificare

Nume de utilizator:

Parola:

[Conecteaza-te](#)

[Inregistreaza-te](#)

Fig. 42 Formular de autentificare

Odată autentificat, există două modalități de introducere a datelor problemei:

1. prin completarea unui formular.
2. prin încărcarea unui fișier.

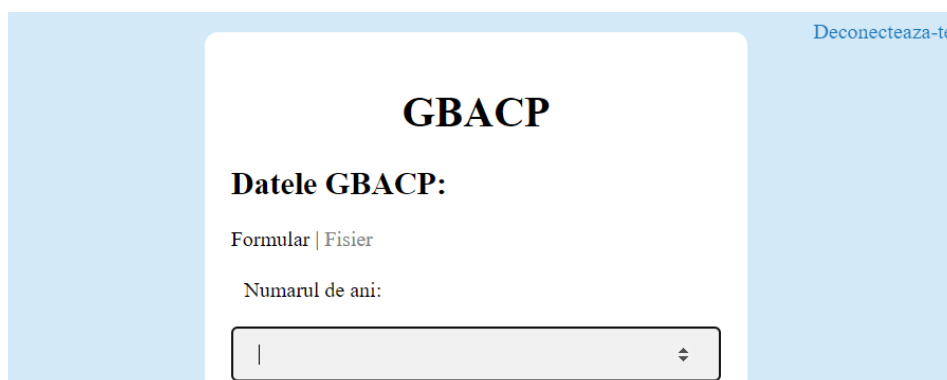


Fig. 43 Pagina GBACP

Datele introduse care apar în imagini corespund exemplului descris în 1.4.

În cazul formularului, se completează câmpurile numărului de ani și de perioade ale unui an și câmpurile limitelor de credite și cursuri asupra perioadelor.

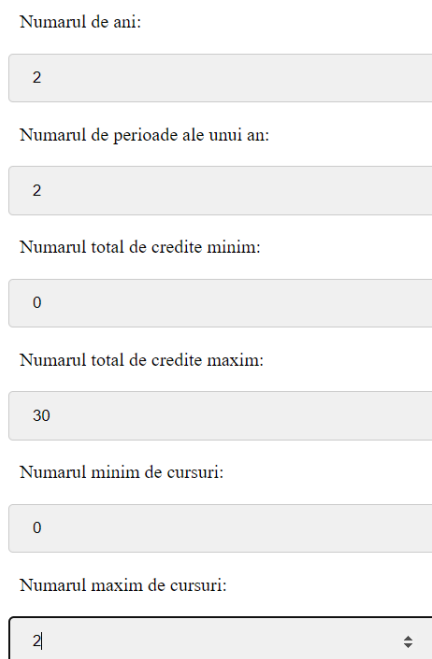


Fig. 44 Date de intrare

Se creează structura specializări-cursuri, după furnizarea informațiilor necesare, prin apăsarea butoanelor “Adaugă specializare”, “Adaugă curs”, “Adaugă curs existent”.

Specializari ✕

1. Specializare1
 - (1) Curs1
 - (2) Curs2
 - (3) Curs3
 - (4) Curs4
2. Specializare2
 - (2) Curs2
 - (3) Curs3
 - (4) Curs4
 - (6) Curs6
3. Specializare3
 - (5) Curs5
 - (1) Curs1
 - (2) Curs2
 - (6) Curs6

Titlul specializarii:

Fig. 45 Structura specializări-cursuri

Titlul specializarii:

Adauga specializare

Fig. 47 Adăugarea unei specializări

Titlul cursului:

Numarul de credite:

Numarul specializarii:

Adauga curs

Fig. 46 Adăugarea unui curs

Numarul cursului:

Numarul specializarii:

Adauga curs existent

Fig. 48 Adăugarea unui curs existent

Se creează listele de precondiții și preferințe, într-un mod asemănător, prin apăsarea butoanelor “Adaugă preferință” și “Adaugă precondiție”.

Preconditii

Cursul 4 inaintea cursului 3 ×

Cursul 4 inaintea cursului 5 ×

Cursul 3 inaintea cursului 6 ×

Numarul primului curs:

Fig. 49 Lista de precondiții

Numarul primului curs:

Numarul celui de-al doilea curs:

Adauga preconditie

Fig. 51 Adăugarea unei precondiții

Preferinte

Cursul 2 sa nu fie predat in perioada 1 a anului ×

Cursul 1 sa nu fie predat in perioada 2 a anului ×

Numarul cursului:

Fig. 50 Lista de preferințe

Numarul cursului:

Perioada anului:

Adauga preferinta

Fig. 52 Adăugarea unei preferințe

În cazul în care se dorește furnizarea datelor de intrare prin intermediul încărcării unui fișier, structura fișierului trebuie să fie asemănătoare cu cea a celor zece instanțe CSP de pe site-ul GBACP (5). Pentru acest exemplu, conținutul fișierului este:

```
DESCRIPTION: Exemplu
YEARS: 2
PERIODS_PER_YEAR: 2
NUM_COURSES: 6
NUM_CURRICULA: 3
MIN_MAX_COURSE_LOAD_PER_PERIOD: 0 2
NUM_PRECEDENCES: 3
NUM_UNDESIRED_PERIODS: 2

COURSES:
Curs1 5
Curs2 10
Curs3 5
Curs4 4
Curs5 20
Curs6 7
```

Fig. 53 Fișierul exemplului

```
CURRICULA:
Specializare1 4 Curs1 Curs2 Curs3 Curs4
Specializare2 4 Curs2 Curs3 Curs4 Curs6
Specializare3 4 Curs1 Curs2 Curs5 Curs6

PRECEDENCES:
Curs4 Curs3
Curs4 Curs5
Curs3 Curs6

UNDESIRED_PERIODS:
Curs2 0
Curs1 1

END.
```

Fig. 54 Fișierul exemplului

Se selectează opțiunea “Fișier”, se apasă pe butonul “*Choose File*” și se selectează fișierul instanței dorite, denumit “Exemplu”.

Datele GBACP:

Formular | Fisier

Incarca un fisier

Choose File Exemplu

Calculeaza

Fig. 55 Încărcarea unui fișier

Printr-un simplu click pe butonul “Calculează”, datele problemei sunt trimise printr-un apel *AJAX* către *server*, apoi de la *server* la *Servlet-ul GbacpServlet*, pentru ca apoi să fie calculată soluția prin intermediul *solver-ului Choco-solver*. Într-un final, clientul primește un *JSON* cu datele soluției pe care le afișează, prin manipularea *DOM* în limbajul de programare *JavaScript*, astfel:

Solutia GBACP:

Specializarea: Specializare1

Anul 1

Perioada 1: Curs4

Perioada 2: Curs3

Anul 2

Perioada 1: Curs1

Perioada 2: Curs2

Specializarea: Specializare2

Anul 1

Perioada 1: Curs4

Perioada 2: Curs3

Anul 2

Perioada 1: Curs6

Perioada 2: Curs2

Specializarea: Specializare3

Anul 1

Perioada 1:

Perioada 2: Curs5

Anul 2

Perioada 1: Curs1, Curs6

Perioada 2: Curs2

Costul dezechilibrului: 38

Costul preferintelor: 0

Costul total al solutiei: 38

Reseteaza

Fig. 57 Afișarea soluției

Fig. 56 Afișarea soluției

3.6 Complexitate și testare

Versiunea de satisfacere a BACP (“Există o soluție, dată fiind o valoare a echilibrului?”) este o problemă NP-completă. Acest lucru poate fi arătat prin reducerea de la versiunea de satisfacere a problemei de *bin-packing* (“Există o soluție cu cel mult un anumit număr dat de coșuri?”) care se cunoaște a fi NP-completă. Obiectele devin cursuri, creditele sunt dimensiuni și coșurile sunt perioade. Nu se consideră precondițiile. Respectiva problemă

de *bin-packing* este rezolvată căutând o soluție pentru BACP. Problema devine mai ușoară prin adăugarea de precondiții, reducându-se numărul de perioade disponibile pentru fiecare curs. (1)

Pentru instanța GBACP dată ca exemplu în 1.4, dar și pentru o instanță GBACP adaptată pentru cazul Facultății de Informatică (cu 3 ani, 2 semestre, 1 specializare, 36 de cursuri și 80 de precondiții) algoritmul de rezolvare a GBACP integrat în aplicație găsește soluția optimă în mai puțin de 0,1 secunde.

Di Gaspero și Schaerf au propus 6 instanțe, denumite UD1-UD6, obținute din datele reale ale Universității din Udine. De asemenea, ei propun o soluție bazată pe căutarea locală care oferă o soluție pentru aceste instanțe care sunt greu de rezolvat astfel încât costul optim rămâne necunoscut. GBACP a fost investigată apoi de Chiarandini et. al. care a studiat două abordări de soluționare, una bazată pe programarea liniară și una bazată pe căutarea locală, care este o versiune îmbunătățită a abordării lui Di Gaspero și Schaerf. Ulterior, au fost adăugate alte 4 instanțe, denumite UD7-UD10 (tot de la Universitatea din Udine). (15)

Pentru instanțele UD3 (cu 3 ani, 3 trimestre, 236 de cursuri, 31 de specializări, 1092 de precondiții și 66 de preferințe) și UD4 (cu 2 ani, 3 trimestre, 139 de cursuri, 16 specializări, 188 de precondiții și 40 de preferințe), algoritmul pe care l-am implementat oferă o soluție în mai puțin de 10 secunde, respectiv 5 secunde. În cazul instanței UD4, algoritmul de optimizare găsește 4 soluții (cu costurile totale de 360, 354, 352 și respectiv 350) în primele 5 secunde, apoi pentru 30 de minute nu determină nicio soluție mai bună. În cazul instanței UD3, algoritmul de optimizare a găsit, în 30 de minute, 5 soluții cu costurile totale în intervalul 1476-1460, așa cum se poate observa în Fig. 58.

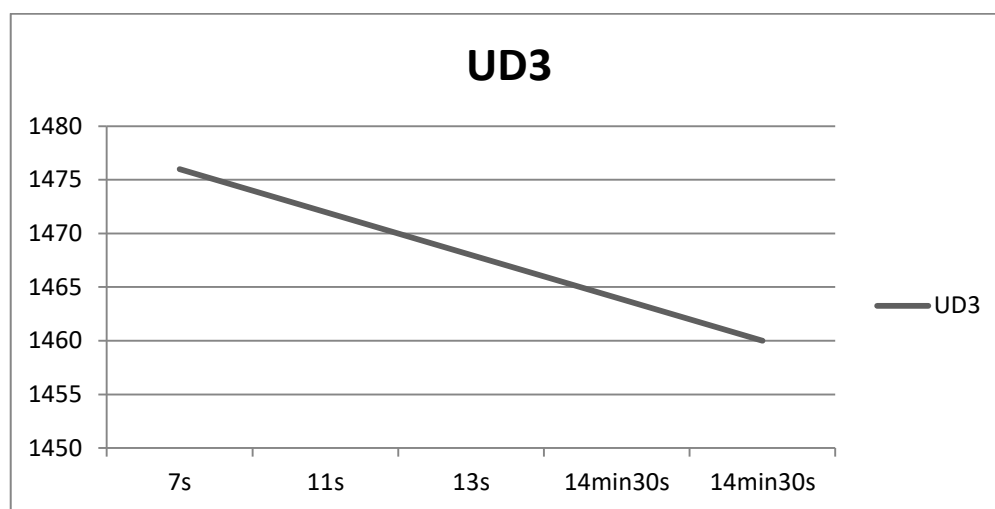


Fig. 58 Optimizarea soluției instanței UD3

Concluzii

Având în vedere că tema lucrării de licență este orientată pe rezolvarea GBACP, consider că abordarea modelării problemei prin intermediul paradigmei programării bazate pe constrângeri reprezintă un succes, punându-se la dispoziție un algoritm care oferă rapid o soluție la diferite instanțe ale unei probleme de o aplicabilitate suficient de mare pentru a fi relevantă pentru toate facultățile.

Biblioteca Choco-solver a jucat un rol important în implementarea unui algoritm cu o putere mare de calcul, simplificând crearea modelului dorit și lucrul cu variabile și constrângeri.

Algoritmul *CSP* este integrat într-o aplicație web cu o interfață grafică ușor de utilizat, dinamică și atractivă, prin care se oferă două posibilități de introducerea a datelor și o reprezentare complexă, dar în același timp ușor de citit, a soluției.

Contribuția proprie a tezei constă în dezvoltarea unei aplicații web Java care pune la dispoziție o interfață grafică cu utilizatorul pentru rezolvarea diferitelor instanțe ale variantei generalizate a problemei de concepere a unor planuri de studii universitare echilibrate, modelată ca și problemă de satisfacere a constrângerilor și optimizare cu ajutorul solver-ului de programare bazată pe constrângeri Choco-solver.

Problema tratată este de natură practică, dar lucrarea conține și o parte teoretică, explicând conceptele necesare pentru înțelegerea modelului problemei abordate și a procesului de dezvoltare a aplicației.

Ca și direcție viitoare de cercetare a temei și de dezvoltare a aplicației, se poate lua în considerare eficientizarea algoritmului prin îmbinarea mai multor abordări, spre exemplu programarea liniară și programarea bazată pe constrângeri, și stocarea persistentă a instanțelor în baza de date Oracle.

Bibliografie

1. *A CP Approach to the Balanced Academic Curriculum Problem.* **Jean-Noel Monette, Pierre Schaus, Stephane Zampelli, Yves Deville and Pierre Dupont.** 2007.
2. *Variable and value ordering when solving balanced academic curriculum problem.* In: *Proc. of the ERCIM WG on constraints.* **Castro, C. and Manzano, S.** 2001.
3. [Online] <https://www.csplib.org/Problems/prob030/>.
4. *The balanced academic curriculum problem revisited.* *Journal of Heuristics*, 18(1):119–148. **Marco Chiarandini, Luca Di Gaspero, Stefano Gualandi, and Andrea Schaerf.** 2012.
5. [Online] <https://opthub.uniud.it/problem/timetabling/gbac>.
6. *Constraint Programming: In Pursuit of the Holy Grail.* **Barták, Roman.** 1999.
7. **F. Rossi, P. van Beek and T. Walsh.** *Handbook of Constraint Programming.* 2006.
8. *Choco solver documentation.* **Prud’homme, Charles and Fages, Jean-Guillaume and Lorca, Xavier.** s.l. : TASC, INRIA Rennes, LINA CNRS UMR, 2016, Vol. 6241.
9. [Online] <https://developer.mozilla.org/en-US/docs/Glossary/MVC>.
10. [Online] <https://www.oracle.com/ro/database/technologies/appdev/xe.html>.
11. [Online] <https://choco-solver.org/docs/considerations/>.
12. [Online] <https://choco-solver.org/docs/modeling/constraints/>.
13. [Online] <https://www.geeksforgeeks.org/introduction-java-servlets/>.
14. [Online] https://en.wikipedia.org/wiki/Jakarta_Server_Pages.
15. **Andrea Schaerf, Marco Chiarandini, Luca Di Gaspero.** *Modelling and Solving the Generalised Balanced Academic Curriculum Problem with Heterogeneous Classes.* 2014.