UNIVERSITATEA ALEXANDRU IOAN CUZA IAŞI

FACULTATEA DE INFORMATICĂ

Utilizarea programării bazate pe constrângeri în încărcarea containerelor

Avram Andreea

Sesiunea: *februarie 2019*

Coordonator ştiinţific

Lector Dr. Cristian Frăsinaru

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemntatul(a) ……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …… specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul:

_____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie

răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Utilizarea programării bazate pe constrângeri în încărcarea containerelor*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, *data*

Absolvent *Prenume Nume*

_____

(semnătura în original)

Cuprins

INTRODUCERE

Călătoria pe apă este una din cele mai vechi metode descoperite de om pentru a putea traversa râuri, mări și oceane, de a călători de pe un țărm pe altul. Chiar dacă prima barcă descoperită de arheologi datează din anul 6000 i.Hr a durat mii de ani până s-a ajuns la nave capabile să transporte nu doar oameni ci și cantități semnificative de mărfuri.

Dacă inițial primele dispozitive plutitoare au fost bușteni, apoi plutele pe care oamenii le-au folosit pentru a scăpa de inundații, de alte primejdii sau din dorința de a se deplasa în noi locuri mai bogate, acestea au evoluat treptat ajungându-se la corăbii de luptă cu vâsle și vele, la dispozitive de navigație din ce în ce mai complexe ca vapoarele dotate cu cârmă și motoare. Este vizibil că acest mijloc de deplasare pe apă a avut o evoluție permanentă de-a lungul anilor astfel că în zilele noastre există nave și vapoare gigantice care navighează peste mări și oceane cu scop turistic , cum sunt vapoarele de croazieră, sau cu scopul de a transporta pasageri sau cantități însemnate de mărfuri de toate categoriile.

Datorită procesului de globalizare în care se află planeta, comerțul care presupune exportul și importul de mărfuri , este foarte dezvoltat. Comerțul a apărut în urma necesității țărilor de a compensa lipsa unor materii de care nu dispunea fiecare țară în parte și bineînțeles de a aduce contribuții la bugetul țării. Cea mai mare parte din comerțul mondial se desfășoară pe mare datorită costurilor relativ mici în comparație cu cantitatea de marfă transportată.

De la an la an crește volum mărfurilor lichide și vrac ce sunt transportate, mărfuri ce reprezintă o parte importantă a tuturor mărfurilor transportate în întreaga lume. Segmentul mărfurilor lichide și vrac este dominat de produse chimice, petrochimice, minerit, construcții și agricultură făcând acest segment important din punct de vedere strategic pentru economia mondială. Livrarea acestor bunuri este un proces complex, care necesită respectarea cerințelor tehnice și sanitare.

După cum am precizat deja transportul maritim este esențial în comerțul și economia mondială, datorită volumului mare de marfă ce poate fi transportat la un preț relativ mic.

Mărfurile sunt depozitate în containere și transportate cu ajutorul navelor.Pentru a maximiza cantitatea de marfă transportată cu o singură navă și pentru a minimiza implicit costurile, încărcarea navelor cu mărfuri a dus la apariția a diverse probleme de optimizare a spațiului folosit (exemplu bin packing).

O problemă de ambalare specifică transportului maritim, adică a alocării de mărfuri containerelor unei nave, este reprezentată de transportul substanțelor chimice. Acestea trebuie transportate în condiții specifice, fiecare substanță având propriile caracteristici ce trebuie respectate. Transportul acestora se face în rezervoare/containere ce trebuie să ofere condiții propice de transport substanței alocate lui.Pentru transportul substanțelor chimice există nave speciale care permit transportul substanțelor astfel încât riscurile să fie minore, atât pentru oameni, cât și în ceea ce privește compoziția lor. Deci transportul substanțelor

chimice necesită o deosebită atenție la alocarea în containerele chimice ale navei. De obicei alocarea încărcăturilor la containere se face manual de către planificatorul navei, iar soluția nu este una de o calitate foarte mare.

Tema abordată în această lucrare presupune rezolvarea unei probleme des întâlnită în transportul substanțelor chimice lichide, adică o ramură importantă a transportului maritim. Această a apărut că o necesitate a rezolvării problemei alocării substanțelor la containerele de pe navă cât mai eficient, atât din punct de vedere al costurilor de curățare al containerelor cât și al siguranței și cantității transportate cu o singură cursă a navei.

Problema alocării a unor substanțe la containerele unei nave se poate modela ca o problema de satisfacere a constrângerilor. O soluție valida fiind acea alocare care satisface toate constrângerile care se impun. Programarea bazată pe constrângeri își are originea în aria Inteligenței artificiale, primele ideii datând din anii '60-'70, dar abia în ultimii câțiva ani experții din mai multe domenii au început să arate un interes special pentru această paradigmă datorită posibilității de a rezolva probleme dificile din viață reală." Nu este surprinzător că a fost recent identificată de ACM (Asociația pentru Calculatoare) ca fiind una dintre direcțiile strategice în domeniul cercetării pe calculator. Cu toate acestea, în același timp, CP este încă una dintre tehnologiile cele mai puțin cunoscute și înțelese."[1]

Pentru obținerea unei soluții pentru această problema, datele au fost modelate că o problema de satisfacere a constrângerilor prin definirea variabilelor, a domeniilor în care pot lua valori variabilele și bineînțeles constrângerile care trebuie satisfăcute de fiecare variabila. O dată modelate aceste entități cu ajutorul bibliotecii java „Choco Solver" se folosește un solver care va conduce la găsirea unei soluții satisfiabile.

Această lucrare este divizată în patru capitole după cum urmează:

- În primul capitol este descris ce este aceea o problemă de satisfacere a constrângerilor, mai exact modelul matematic al unei probleme.
- În cel de-al doilea capitol sunt prezentate tehnicile care stau la baza programării bazate pe constrângeri: tehnici de consistență, propagarea constrângerilor și căutarea sistematică a soluției.
- Cel de-al treilea capitol conține descrierea unei biblioteci Java care oferă suport pentru rezolvarea unei probleme de satisfacere a cosntrangerilor. Această bibliotecă permite crearea unui model pe baza celui matematic și rezolvarea acestuia folosind un solver deja implementat care utilizează tehnicile de consistență, de propagare a constrângerilor și algoritmii de căutare sistematică.
- Ultimul capitol conține prezentarea unei aplicații demonstrative, aplicație care presupune încărcarea unor cantități de substanțe în containerele unei nave.

Capitolul I – Probleme de satisfacere a constrangerilor

Raţionamentul bazat pe constrângeri este o paradigmă foarte simplă şi puternică, cu ajutorul căreia pot fi formulate şi rezolvate multe probleme dificile.

I.1 Ce sunt problemele de satisfacere a constrangerilor?

Problemele de satisfacere a constrângerilor (CSP) sunt probleme matematice definite ca o mulţime de obiecte a căror stare trebuie să îndeplinească o serie de constrângeri sau limitări. CSP-urile reprezintă entitățile dintr-o problemă ca o colecție omogenă de constrângeri finite față de variabile, care este rezolvată prin metode de satisfacere a constrângerilor. De asemenea, CSP-urile reprezintă subiectul unor cercetări intense în domeniul Inteligenței artificiale, dar și în domeniul cercetărilor operaționale, deoarece ,în general, formularea acestora oferă o bază comună pentru a analiza și a rezolva o varietate largă de probleme. Acestea au o complexitate mare, necesitând o combinație între metodele de căutare euristice și combinatoriale pentru ale putea rezolva într-un timp rezonabil.[2]

Raţionamentul bazat pe constrângeri este o paradigmă foarte simplă şi puternică, cu ajutorul căreia pot fi formulate şi rezolvate multe probleme interesante. O constrângere este o relaţie logică între mai multe variabile, fiecare luând valori din câte un domeniu precizat. Astfel, o constrângere indică restricţii asupra valorilor posibile pe care o variabilă le poate lua, datorită condiţionărilor dintre aceasta şi alte variabile. Constrângerile au o serie de proprietăţi remarcabile:

• constrângerile pot specifica informaţie parţială ;

• constrângerile sunt non-direcţionale ;

• constrângerile sunt declarative;

• constrângerile sunt aditive;

• constrângerile sunt rareori independente.

Programarea prin constrângeri reprezintă studiul sistemelor computaţionale bazate pe constrângeri. Ideea este de a rezolva probleme prin specificarea condiţiilor care trebuie satisfăcute de soluţie. Această abordare este, astfel, înrudită cu paradigma programării declarative, unde programul reprezintă o descriere a problemei, şi nu o reţetă de rezolvare a acesteia. Problemele de satisfacere a constrângerilor (CSP) presupun:

• o mulţime finită de variabile, fiecare având câte un domeniu finit de valori;

• o mulţime finită de constrângeri, fiecare introducând restricţii asupra valorilor pe care anumite variabile le pot lua simultan.

O soluţie a unei probleme CSP reprezintă atribuirea câte unei valori fiecărei variabile, în aşa fel încât toate constrângerile existente să fie satisfăcute. În funcţie de problema de

rezolvat, trebuie identificată fie o soluţie oarecare, fie o soluţie anume, fie toate soluţiile. Spre deosebire de problema satisfacerii constrângerilor, problema rezolvării constrângerilor operează cu variabile având domenii infinite, constrângerile putând fi descrise prin ecuaţii matematice complicate. Aplicaţiile practice ale problematicii satisfacerii constrângerilor sunt nelimitate, mergând de la probleme de puzzle simple (problema damelor, criptaritmetică) şi până la controlul traficului aerian, de metrou şi căi ferate.[3]

I.2 Definiţia unei probleme CSP

O problemă de satisfacere a constrângerilor (CSP) este alcatuita din:

• o mulţime finită de variabile X=$\{x_1, x_2, \ldots, x_n\}$;

• o mulțime de valori finită, numită si domeniul variabilei, D_i pentru fiecare variabilă x_i;

• o mulţime finită de constrângeri, definite asupra valorilor pe care le poate lua fiecare variabilă;

Soluția unei probleme CSP este reprezentată de atribuirea unei valori fiecărei variabile definite, valoare care se află în domeniul de definiție al variabilei respective și care satisface toate constrângerile definite asupra acelei variabile. O problemă CSP poate avea mai multe soluții viabile, deci scopul rezolvării unei astfel de probleme poate fi găsirea unei soluții oarecare, a tuturor soluțiilor sau a unei soluții care să maximizeze/ minimizeze o anumită variabilă.

Soluţiile unei probleme CSP se pot obţine prin căutare sistematică prin toate valorile posibile. Metodele de căutare fie traversează spaţiul soluţiilor parţiale, fie explorează spaţiul atribuirilor complete de valori. [3]

Majoritatea algoritmilor de satisfacere a constrângerilor se referă la probleme în care fiecare constrângere este fie unară, fie binară. Astfel de probleme sunt denumite probleme CSP binare.[3] Orice constrângere n-ară poate fi redusă la o mulțime de constrângeri binare. Pentru a ilustra aceast lucru, se introduce o variabilă suplimentară u care încapsulează cele n variabile originale, domeniul său fiind produsul cartezian al domeniilor variabilelor originale. O constrângere n-ară asupra valorilor variabilelor originale va fi înlocuită cu o constrângere unară asupra valorilor variabilei nou introduse. Practic, pentru fiecare constrângere n-ară se introduce câte o variabilă suplimentară cu domeniul modificat prin reducerea constrângerii n-are originale. Rămâne ca variabilele suplimentare introduse să fie legate de variabilele originale. Există două modalităţi de a realiza acest lucru: cu păstrarea variabilelor originale, şi cu eliminarea variabilelor originale. Dacă se doreşte păstrarea variabilelor originale, pentru fiecare variabilă originală x_i şi pentru fiecare variabilă suplimentară u, se introduce constrângerea x_i =$arg_i(u)$ (x_i este a i-a componentă a lui u). Dacă se doreşte eliminarea variabilelor originale, pentru oricare variabile suplimentare u şi v, se introduc constrângerile $arg_i(u) = arg_i(v)$ pentru fiecare valoare a lui i. Prin urmare orice problemă de satisfacere a

constrângerilor se poate reprezenta printr-o problemă binară echivalentă. Din acest motiv, faptul că întreaga literatură se concentrează asupra problemelor binare nu reprezintă o pierdere a generalităţii. [3]

Problemele de satisfacere a constrângerilor pe domenii finite sunt de obicei rezolvate folosind o formă de căutare. Cele mai utilizate tehnici sunt variantele de backtracking, de *propagare a constrângerilor* și *de căutare locală*.[3]

1.Backtracking

Backtracking este un algoritm recursiv ce verifică constrângerile în timpul căutării nu doar la final. Backtracking este un algoritm de căutare în adâncime, acesta asignează pe rând fiecărei variabile valorile din domeniul de definiție al variabilei respective. În momentul în care se realizează asignarea unei valori ce nu satisface toate constrângerile definite, algoritmul se întoarce la variabila anterioară pentru asignarea unei alte valori satisfiabile din domeniul ei de definiție.

2.Propagarea constrângerilor

Propagarea constrângerilor presupune eliminarea din domeniul de definiție al variabilelor a valorilor care sunt inconsistente cu constrângerile definite. Pentru fiecare variabilă din fiecare constrângere se elimină valorile inconsistente din domeniul acesteia. O valoare este inconsistentă dacă nu poate fi atribuită unei variabile astfel încât toate constrângerile să fie respectate. Propagarea unei constrângeri face ca aceasta să fie consistentă local. Propagarea constrângerilor are diverse utilizări. În primul rând , transformă o problemă într-una echivalentă, dar mai simplu de rezolvat. În al doilea rând, poate dovedi satisfiabilitatea sau nesatisfiabilitatea problemelor. Dar trebuie precizat că acest lucru nu este garantat, în general. Cea mai populară metodă de propagare a constrângerilor este algoritmul AC-3, care impune consistența arcului. Deși metoda consistenței arcului este definită pentru constrângeri binare, există o metodă generalizată a acesteia ce poate fi aplicată și pe constrângeri ce nu sunt binare. Mai jos este explicat print-un mic exemplu propagarea costrangerilor . Veți putea observa cum este micșorat spațiul de căutare datorită eliminării din domeniul fiecărei variabile a valorilor inconsistente.

Exemplu 1.

Fie:

- $X = \{X_1, X_2, X_3\}$ multimea variabilelor
- D(X_1)={1,2,3} , D(X_2)={1,2}, D(X_3)={1,2} domeniile variabilelor
- C: $X_1 \neq X_2 \neq X_3$ constrângerea definită asupra valorilor variabilelor

Constrângerea de mai sus poate fi descompusă în 3 constrângeri binare. Deci aceasta poate fi scrisă astfel:

$$C1\colon\ X_1 \neq X_2$$

$$C2\colon\ X_1 \neq X_3$$

$$C3\colon\ X_2 \neq X_3$$

În urma aplicării metodei consistenței arcului din constrângerile C1 si C2 rezultă fapul că $X_1 = 1\ si\ X_1 = 2$ sunt valori inconsistente, deci domeniul variabilei X_1 va fi alcatuit doar din valoarea 3, D(X_1)={3};

Așa cum se poate observa și în exemplul de mai sus, propagarea constrângerilor ajută la diminuarea semnificativă a spațiului de căutare.

3.Cautarea locala(Local search)

Metodele de căutare locală sunt algoritmi de satisfacere incompleti. Ei pot găsi o soluție a unei probleme, dar pot eșua chiar dacă problema este satisfiabilă. Ei lucrează prin îmbunătățirea iterativa a asignarii complete asupra variabilelor. La fiecare pas, un număr mic de variabile sunt modificate, scopul fiind creșterea numărului de constrângeri îndeplinite de această alocare. Algoritmul „Min-conflicts" este un algoritm de căutare locală specific pentru CSP și bazat pe acest principiu.

Algoritmul Min-Conflicts

Algoritmul poate fi considerat și o euristică pentru rezolvarea problemelor de satisfacere a constrangerlor. Acesta este similar cu algorimul genetic Hill Climbing care urmărește îmbunătățirea soluției pe parcurs, dar care poate obține sau nu o soluție. Algorimul Min-Conflicts pornește cu o instanțiere random a variabilelor problemei CSP, iar apoi acesta alege random o variabilă care nu satisface una sau mai multe constrângeri. Apoi îi asignează acestei variabile valoarea care produce un număr cât mai mic de conflicte (constrageri care nu sunt satisfăcute). Algoritmul repetă pașii de mai sus, adică alegerea random a unei variabile ce cauzează conflicte și asignarea unei valori ce minimizează numărul de conflicte, până când se obține o soluție sau un număr maxim dat de pași este atins. Trebuie precizat faptul că, algoritmul nu conduce întotdeauna la o soluție satisfiabilă și că soluția depinde de instantirea inițială a variabilelor.

```
algorithm MIN-CONFLICTS
    input: csp, a constraint satisfaction problem
           max_steps,the number of steps allowed before giving up
           current_state, an initial assignment of values for the variables in the csp
    output: a solution set of values for the variable or failure
  for i ← 1 to max_steps do
      if current_state is a solution of csp then return current_state
      set var ← a randomly chosen variable from the set of conflicted variables CONFLICTED[csp]
      set value ← the value v for var that minimizes CONFLICTS(var,v,current_state,csp)
      set var ← value in current_state
  return failure
```

Fig.1 Min-conflicts-pseudocode[4]

O bună instanțiere poate conduce rapid la obținerea soluției pentru problema CSP.

I.3 Exemple de probleme CSP

1.Problema reginelor

Această problemă presupune așezarea a N regine pe o table de șah de dimensiune NxN astfel încât nici o regină să nu atace o altă regină. Spunem că o regină atacă o alta atunci când acestea se află pe aceeași coloană, aceeași linie sau aceeași diagonală a tablei de șah.

Pentru exemplificare vom consideră N=8. În acest caz există 92 de configurații distincte pentru această problemă. Pentru a rezolva această problemă oamenii experimentează diverse configurații, așezări ale reginelor pe table de șah. Pentru un N destul de mic, ca cel considerat mai sus spre exemplu, aceștia pot converge la o soluție consistentă, în sensul că pot găsi o așezare a celor N regine astfel încât acestea să nu se atace între ele, fără a testa toate cele 92 de configurații. Dar pentru un N=1000 oamenilor le-ar fi dificil să gasească o soluție consistentă.

Calculatoarele sunt bune în a face rapid un număr mare de lucruri simple. Deci o soluție ar fi plasarea sistematică a reginelor până când găsim o soluție. Acest lucru presupune generarea și testarea configurațiilor.

Valoarea lui N	Nr. total de configuratii
4	256
8	16,777,216
16	18,446,744,073,705,551,616

După cum se poate observa pentru N=16 numărul total de configurații este foarte mare, iar rezolvarea problemei în acest caz ar dură aproximativ 12 ani pe o mașină rapidă și modernă. În general există N^N configuratii pentru N regine.

O metodă de rezolvare a acestei probleme o constituie folosirea algoritmului Backtracking. Un lucru pe care îl putem observa este că, de exemplu, în cazul problemei celor 8 regine, de îndată ce plasăm o parte din regine știm că o întregă mulțime suplimentară de configurații este nevalidă. Bactracking este una din metodele principale pentru rezolvarea unei probleme cum ar fi N-regine. Dar nu dorim să creăm un algoritm doar pentru a rezolva N-regine. Trebuie să exprimăm problema celor N-regine ca o instanță a unei clase generale de probleme și să construim un algoritm pentru rezolvarea unei clase generale de probleme. Acest lucru este posibil modelând problema ca o instanță CSP. Deoarece există deja algoritmi dezvoltați pentru rezolvarea unor astfel de probleme, deci a unei clase largi de probleme, pentru rezolvarea acestei probleme și a multor altele tot ceea ce trebuie să facem este să modelăm datele problemelor ca instanțe CSP. În cele ce urmează vom modela problema celor 8-regine ca o instanță CSP. Acest lucru presupune definirea a trei componente:

1. O mulțime de variabile
2. O mulțime de valori pentru fiecare variabilă, domeniul
3. O mulțime de constrângeri între diferite colecții de variabile

Rezolvarea acestei instanțe presupune atribuirea unei valori din domeniul său fiecărei variabile astfel încât fiecare constrângere să fie satifacută.

O constrângere este o relație între o colecție locală de variabile, aceasta restricționează valorile pe care variabilele le pot lua simultan. De exemplu, all-diff(x_1, x_2, x_3), această constrângere presupune ca cele 3 variabile să ia valori diferite. Să spunem că mulțimea $\{1,2,3\}$ reprezintă setul de valori pentru fiecare din cele 3 variabile x_1, x_2, x_3. Atunci o solutie consistenta ar fi $x_1 = 1, x_2 = 2, x_3 = 3$. Pe cand o astfel de atribuire $x_1 = 1, x_2 = 1, x_3 = 3$ nu ar constitui o soluție validă, deoarece constrângerea nu este satisfăcută. Deși fiecare constrângere este peste o colecție locală de variabile, găsirea unei asignări globale pentru toate variabilele care să satisfacă toate constrângerile este dificil: NP-complet. Tehnicile de găsire a soluțiilor funcționează prin căutarea inteligentă în spațiul de valori posibile a fi alocate variabilelor. Dacă fiecare variabilă ar avea d valori si exista n variabile cărora trebuie să le fie asignate valori, atunci numărul total de asignări posibile ar fi d^n.

Revenind la problema celor N-regine ca instanță CSP, trebuie să determinăm locul în care să fie plasată fiecare regină pe tabla de șah. Deci avem N variabile fiecare putând lua valori în mulțimea $\{1, \dots, N^2\}$, fiecare reprezentând locul pe tabla de șah unde va fi plasată fiecare regină. Pentru această reprezentare, cu N=8, numărul total de asignarii posibile este de ordinul bilioanelor (64^8).

În acest caz știm că niciodată nu putem plasa două regine pe aceeași coloană. Deci putem configura problemă astfel încât să asignăm o regină fiecărei coloane și acum trebuie să găsim doar linia pe care să fie plasate reginele. Deci putem aveam N variabile: $Q_1, \dots, Q^N$ si setul de valori $\{1,2, \dots, N\}$ pentru fiecare din aceste variabile. Astfel numărul total de asignări pentru N=8 este $(8^8 = 16{,}777{,}216)$ de ordinal milioanelor. Numărul asignarilor posibile este încă destul de larg, dar cu o îmbunătățire considerabilă.

Constrângerile reprezintă componenta cheie în exprimarea unei probleme ca o instanță CSP. Ideea este că problema este împărțită într-un set de condiții distincte și fiecare trebuie satisfăcută pentru ca problema să fie rezolvată. În problema considerată, N-regine:

- Nici o regină nu poate ataca o altă regină.
  - Fiind date oricare două regine $Q_i \; si \; Q_j$ ele nu se pot ataca una pe alta
- Vom transforma condițiile individuale în constrângeri separate. Q_i nu poate ataca $Q_j \; (i \neq j)$
  - Q_i este regina așezată in coloana i, iar Q_j regina așezată in coloana j.
  - Valorile pentru $Q_i \; si \; Q_j$ sunt liniile pe care vor fi așezate.

Transformarea depinde de reprezentarea pe care o alegem.
O regină poate atacă o alta în trei cazuri:

1. Vertical, dacă ele se află pe aceeași coloană, ceea ce este imposibil deoarece $Q_i \; si \; Q_j$ sunt plasate pe coloane diferite.
2. Orizontal, dacă se află pe aceeași linie, deci avem nevoie de constrângerea $Q_i \neq Q_j$.
3. De-a lungul unei diagonale, deci avem nevoie de constrângerea $: |i - j| \neq |Q_i - Q_j|$

Reprezentarea constrângerilor:

1. Între fiecare pereche de variabile (Q_i, Q_j), $i \neq j$, există o constrângere C_{ij}
2. Pentru fiecare constrângere C_{ij} , asignarea valorilor variabilelor $Q_i = A \; si \; Q_j = B$ satisface aceasta constrângere doar dacă:
   a. $A \neq$B
   b. $|A - B| \neq |i - j|$

O solutie pentru problema celor N regine va fi o asignare de valori la variabilele $Q_1, \dots, Q^N$ astfel încât toate constrângerile sunt satisfăcute. Constrângerile pot fi peste orice colecție de variabile. În problema celor N regine avem nevoie doar de constrângeri binare, peste perechi de variabile.

2. The Propositional Satisfiability Problem (SAT)

- O formulă în logica propozițională conține doar variabile booleene.
  - Notație: x pentru $X = true$ si $\neg x$ pentru $X = false$
  - Literali: $x, \neg x$
- Formula se află în forma normal conjunctivă (CNF): conjuncție de clause (disjuncție de literali)
  - Ex.: $F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$

Se scrie formula F ca o instanță CSP:

- 3 variabile: $X_1, X_2, X_{3,}$
- Domenii: pentru fiecare variabilă $\{true, false\}$
- Constrângeri (clauze):
  - $(x_1 \vee x_2 \vee x_3)$
  - $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$
  - $(\neg x_1 \vee \neg x_2 \vee x_3)$

O soluție ar fi: $X_1 = true, , X_2 = false, X_{3,} = true$

Problema 3-SAT a fost prima problemă ce a fost arătată că este NP-hard. Aceasta este una dintre cele mai importante probleme din domeniul informaticii teoretice.

Importanța în practică pentru:

- Verificarea software-ului: găsirea de erori în Windows etc.
- Verificarea hardware: verificarea chip-urilor de computer (IBM, Intel)

Capitolul II – Programarea bazată pe constrângeri (Constraint Programming)

II.1 Ce este programarea bazată pe constrângeri?

Definitia 1. Programarea bazată pe constrângeri poate fi privită ca o paradigmă de programare ce constă în găsirea unei soluții la o problemă de către un calculator, atunci când acesta primește problema ca un asamblu de constrângeri.[5]

Ce este o paradigmă de programare?

Cuvântul paradigmă (din grecescul παράδειγμα, exemplu) este folosit de obicei pentru a referi o categorie de entităţi având o caracteristică comună.Termenul paradigmă în ştiinţă este introdus de Thomas Kuhn [Kuhn, 1970], care îl foloseşte pentru a denota o colecţie consistentă de metode şi tehnici acceptate de comunitatea ştiinţifică ca fiind predominante într-un anumit domeniu într-o anumită perioadă. În ştiinţa calculatoarelor, termenul de paradigmă denotă esenţa procesului de programare, dând astfel naştere la paradigme de programare.[6]

O paradigmă de programare este un stil fundamental de a programa, care impune modul de reprezentare al datelor problemei (variabile, funcții, obiecte, constrângeri etc.) și modul în care se prelucrează reprezentarea (atribuiri, evaluări, fire de execuție, continuări, fluxuri). De asemenea impune un anumit set de concepte și tehnici de programare și influenteza felul în care sunt gândiți algoritmii de rezolvare a problemelor.[7]

Cele mai importante paradigmele de programare existente sunt: programarea orientată obiect, programarea imperativă, programarea declarativă, programarea paralelă și bineînțeles programarea bazată pe constrângeri.

Definiția 2. Programarea bazată pe constrângeri este o paradigmă declarativă în care programatorul specifică un set de constrângeri exprimate asupra unui domeniu (domeniu finit, rațional, real), sistemul rezolvând automat aceste constrângeri.[6]

Programarea bazată pe constrângeri poate fi definită mai riguros astfel: programarea constrângerilor constă în optimizarea unei funcții supuse constrângerilor logice, aritmetice sau funcționale asupra variabilelor discrete sau variabilelor de interval sau găsirea unei soluții fezabile la o problemă definită de constrângerile logice, aritmetice sau funcționale asupra variabilelor discrete sau variabilelor de interval.

Cum funcționează programarea bazată pe constrângeri?

Ideea de bază în această metodă de programare constă în modelarea problemei ca una de satisfacere a constrângerilor. Adică, așa cum am precizat în capitolul destinat problemelor de satisfacere a constrângerilor, stabilirea variabilelor, a domeniului în care poate lua valori fiecare variabilă și definirea constrângerilor care trebuie respectate de aceste variabile la asignarea valorilor. Această paradigmă ajută la rezolvarea multor probleme din viața reală, deoarece multe din problemele reale sau întâlnite zi de zi se reduc la o mulțime de

condiții ce trebuie respectate. După crearea modelului CSP pentru problemă pe care dorim să o rezolvăm, se folosește un solver care va găsi o soluție ce satisface toate constrângerile impuse. O problemă dificil de rezolvat este, spre exemplu, crearea unui orar de examene la universități. Acest lucru este dificil din pricina multiplelor condiții care trebuie respectate. În stabilirea unui astfel de orar trebuie să se țină cont de: disponibilitatea și capacitatea sălilor, de timpul care trebuie să existe între două examene la care participă aceiași studenți, de disponibilitatea studenților (aceștia nu pot susține mai multe examene în același timp). [8]

Rezolvarea unei probleme CSP folosind programarea bazată pe constrângeri presupune îmbinarea tehnicilor următoare:

- Propagarea constrângerilor;
- Căutarea sistematică: Backtracking;
- Tehnici de consistență;

Există, desigur, și probleme a căror soluție poate fi descoperită doar aplicând tehnica de propagarea a constrângerilor. Aceste tehnici sunt folosite la implementarea solver-elor.

Solver-ele caută sistematic în spațiul de soluții, utilizând diverși algoritmi cum ar fi: backtracking, branch and bound sau diferite forme de căutare locală. Trebuie precizat faptul că algoritmii de căutare locală pot fi incompleti, în sensul că nu conduc întotdeauna la o soluție chiar dacă problema este satisfiabilă. De cele mai multe ori căutarea sistematică interferează cu tehnicile de propagarea a constrângerilor. Propagarea constrângerilor presupune micșorarea domeniului în care iau valori variabilele. Acest lucru se întâmplă datorită inferenței constrângerilor între ele, conducând la identificarea unor valori ce nu pot satisface niciodată toate constrângerile. Deoarece acestea nu vor putea fi atribuite niciodată, ele sunt eliminate din domeniile de valori și astfel spațiul de căutare se restrânge, în unele cazuri considerabil de mult.

Un solver poate fi implementat în orice limbaj de programare. Însă există limbaje special create pentru a reprezenta relații de constrângere și pentru a putea alege strategia de căutare.

II.2 Propagarea constrângerilor

Tehnicile de propagare a constrângerilor sunt metode utilizate pentru a modifica o problemă de satisfacere a constrângerilor. Propagarea constrângerilor are diverse utilizări. În primul rând, transformă o problemă într-una echivalentă, dar care este, de obicei, mai simplă de rezolvat. În al doilea rând, poate dovedi satisfiabilitatea sau nesatisfiabilitatea problemelor. Acest lucru nu este garantat în general; totuși, acest lucru se intampla întotdeauna pentru anumite forme de propagare a constrângerilor și / sau pentru anumite tipuri de probleme. Propagarea constrângerilor a apărut ca o consecință a faptului că de obicei variabilele sunt incluse în mai multe constrângeri. Propagarea constrângerilor poate reduce spațiile de valori ale variabilelor astfel încât să obținem soluția problemei, în acest caz este dovedită și satisfiabilitatea acesteia. Dar propagarea constrângerilor poate elimina și toate valorile din

domeniul unei variabile, caz în care singura concluzie ce poate fi trasă, este că problema nu are soluție, adică nu există o asignare validă pentru variabile astfel încât toate constrângerile să fie respectate, deci problema este nesatisfiabila. Pe lângă cele două cazuri enumerate mai sus, există și cazul în care pentru determinarea soluției trebuie aplicat un algoritm de căutare.

Propagarea constrângerilor nu este întotdeauna suficientă pentru rezolvarea unei probleme de satisfacere a constrângerilor, dar optimizează considerabil timpul de execuție al unui algoritm de căutare. Algoritmul de căutare asignează valori variabilelor, astfel încât constrângerile să fie respectate, iar după fiecare asignare se aplică propagarea constrângerilor asupra acestora utilizând valorile asignate. Dacă se detecteza o contradicție în urma propagării, algoritmul trebuie să revizuiască asignarile făcute.

II.3 Tehnici de consistență

O altă strategie de rezolvare a problemelor de satisfacere a constrângerilor presupune eliminarea valorilor incosistente din domeniul de definiție al fiecărei variabile, până când se obține soluția. Metodele care abordează această strategie se numesc tehnici de consistență. O diferență majoră între tehnicile de consistență și căutarea sistematică a soluție o reprezintă faptul că cele din urmă sunt nedeterministe, iar celelalte sunt deterministe.

Există mai multe tehnici de consistentă, dar majoritatea nu sunt complete. Prin urmare, tehnicile de consistentă sunt rareori utilizate singure pentru a rezolva complet CSP.

Tehnicile de consistență au fost introduse de cercetătorii din domeniul Inteligenței artificiale cu scopul de a îmbunătății eficiența algoritmilor de recunoaștere a imaginilor. Aceste tehnici s-au dovedit a fi eficiente pe o mare varietate de probleme de căutare dificilă.

În CSP-urile binare, s-au introdus diferite tehnici de consistentă pentru graficele de constrângeri pentru a tăia spațiul de căutare. Algoritmul de implementare a consistenței face ca orice soluție parțială a unei subrețele mici să fie extensibilă pentru o rețea din jur. Astfel, inconsistența potentială este detectată cât mai curând posibil.

Așa cum am precizat mai sus, tehnicile de consistență elimină valorile inconsistente din domeniul de definiție al variabilelor, adică valorile care dacă ar fi atribuite variabilelor nu ar satisface constrângerile impuse de modelul CSP al problemei. Aceste tehnici presupun reprezentarea problemei ca un graf, în care variabilele cărora trebuie asignate valori sunt reprezentate prin noduri în graf, iar constrângerile prin muchii între variabilele implicate în constrângere. Această reprezentare se folosește doar pentru constrângeri binare și unare, dar acest fapt nu este o probleme, deoarece orice constrângere n-ară poate fi transformată în mai multe constrângeri binare.

Tehnici de consistență:

- node consistency (NC)
- arc consistency (AC)
- path consistency (PC)
- (strong) k-consistency.

Dintre toate tipurile de consistență enumerate mai sus, consistența arcului este cea mai des folosită și cea mai utilizată tehnică de consistență.

Consistența arcului (Arc Consistency):

Consistența arcului este tehnica prin care sunt eliminate valori din domeniile de definiție al variabilelor. Reducându-se astfel spațiul de căutare al algoritmilor de căutare sistematică, implicit al timpului de execuție. Deci tehnicile de consistență pot fi considerate și o metodă de optimizare a algoritmilor de căutare în rezolvarea problemelor CSP.

Definitie[9]

- Fie $P = (X, D, C)$ o problema CSP (binara)
- Valoarea $a \in d_i$ a variabilei $x_i \in X$ este arc-consistenta cu privire la x_j dacă există $b \in d_j$ astfel încat $c_{ij}(a, b) = true$

Atunci arc-consistenta poate fi definită astfel:

- ❖ Variabila $x_i \in X$ este arc-consistenta cu privire la x_j dacă toate valorile din domeniul său sunt arc-consistente cu privire la x_j
- ❖ Constrângerea $c_{ij} \in C$ este arc-consistenta dacă x_i este arc-consistenta cu privire la x_j si invers
- ❖ O problemă CSP este arc-consistenta dacă toate constrângerile $c \in C$ sunt arc- consistente

Dacă valoarea $a \in d_i$ nu este arc-consistentă cu privire la x_j, spunem ca valoarea este nefezabilă. Prin urmare, poate fi eliminată din domeniul de valori al variabilei x_i.

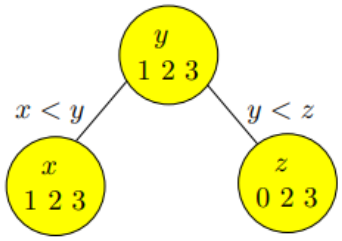
Exemplu [9]

Fie problema CSP, cu variabilele x, y, z unde:

- $D_x = \{1,2,3\},\ D_y = \{1,2,3\},\ D_z = \{0,2,3\}$
- $C_{xy}: x < y \quad si \quad C_{yz}: y < z$

Relovare:

După cum bine se știe, noțiunea de arc este folosită în teoria grafurilor. Așa că pentru exemplificarea acestui concept vom folosi o reprezentare grafică.

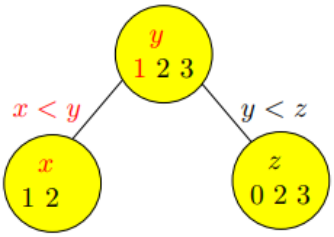


Conform definitie vom testa :

➢ $C_{xy}: x < y$

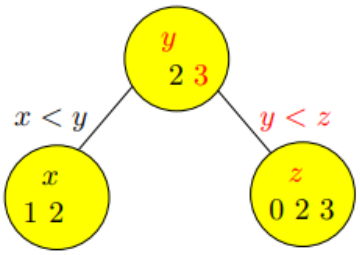
1. Pentru fiecare valoare $a \in D_x$ dacă există o valoare $b \in D_y$ astfel încat $C_{xy}(a,b) = true$
   - $a = 1, b = 1,\ a < b \Leftrightarrow 1 < 1 \Longrightarrow C_{xy}(1,1) = false$
   - $a = 1, b = 2,\ a < b \Leftrightarrow 1 < 2 \Longrightarrow C_{xy}(1,2) = true \Longrightarrow \exists\, b \in D_y$
   - $a = 2, b = 1,\ a < b \Leftrightarrow 2 < 1 \Longrightarrow C_{xy}(2,1) = false$
   - $a = 2, b = 2,\ a < b \Leftrightarrow 2 < 2 \Longrightarrow C_{xy}(2,2) = false$
   - $a = 2, b = 3,\ a < b \Leftrightarrow 2 < 3 \Longrightarrow C_{xy}(2,3) = true \Longrightarrow \exists\, b \in D_y$
   - $a = 3, b = 1,\ a < b \Leftrightarrow 3 < 1 \Longrightarrow C_{xy}(3,1) = false$
   - $a = 3, b = 2,\ a < b \Leftrightarrow 3 < 2 \Longrightarrow C_{xy}(3,2) = false$
   - $a = 3, b = 3,\ a < b \Leftrightarrow 3 < 3 \Longrightarrow C_{xy}(3,3) = false$

   Deoarece pentru valoarea $a = 3, \nexists b \in D_y$ valoarea 3 va fi eliminată din domeniul de definitie al variabilei x.



2. Pentru fiecare valoare $a \in D_y$ dacă există o valoare $b \in D_x$ astfel încat $C_{xy}(a,b) = true$
   - $a = 1, b = 1,\ b < a \Leftrightarrow 1 < 1 \Longrightarrow C_{xy}(1,1) = false$
   - $a = 1, b = 2,\ b < a \Leftrightarrow 2 < 1 \Longrightarrow C_{xy}(1,2) = false$
   - $a = 2, b = 1,\ b < a \Leftrightarrow 1 < 2 \Longrightarrow C_{xy}(2,1) = true \Longrightarrow \exists\, b \in D_x$
   - $a = 2, b = 2,\ b < a \Leftrightarrow 2 < 2 \Longrightarrow C_{xy}(2,2) = false$
   - $a = 3, b = 1,\ b < a \Leftrightarrow 1 < 3 \Longrightarrow C_{xy}(3,1) = true \Longrightarrow \exists\, b \in D_x$

Deoarece pentru valoarea $a = 1 \in D_y, \nexists b \in D_x$ valoarea 1 va fi eliminată din domeniul de definitie al variabilei y.

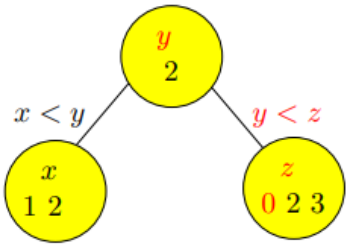


➢ $C_{yz}: y < z$

1. Pentru fiecare valoare $a \in D_y$ dacă există o valoare $b \in D_z$ astfel incat $C_{yz}(a,b) = true$
   - $a = 2, b = 0,\ a < b \Leftrightarrow 2 < 0 \Longrightarrow C_{yz}(2,0) = false$

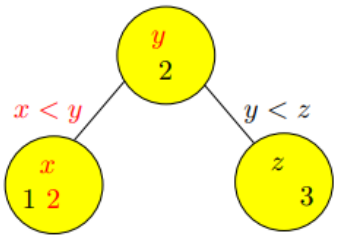
- $a = 2, b = 2,\ a < b \Leftrightarrow\ 2 < 2 \implies C_{yz}(2,2) = false$
- $a = 2, b = 3,\ a < b \Leftrightarrow\ 2 < 3 \implies C_{yz}(2,3) = true \implies \exists\ b\ \in D_y$
- $a = 3, b = 0,\ a < b \Leftrightarrow\ 3 < 0 \implies C_{yz}(3,0) = false$
- $a = 3, b = 2,\ a < b \Leftrightarrow\ 3 < 2 \implies C_{yz}(3,2) = false$
- $a = 3, b = 3,\ a < b \Leftrightarrow\ 3 < 3 \implies C_{yz}(3,3) = false$

Deoarece pentru valoarea $a = 3 \in D_y, \nexists b \in D_z$ valoarea 3 va fi eliminată din domeniul de definitie al variabilei y.



2. Pentru fiecare valoare $a \in D_z$ dacă există o valoare $b \in D_y$ astfel incat $C_{xy}(a, b) = true$
   - $a = 0, b = 2,\ b < a \Leftrightarrow\ 2 < 0 \implies C_{yz}(2,0) = false$
   - $a = 2, b = 2,\ b < a \Leftrightarrow\ 2 < 2 \implies C_{yz}(2,2) = false$
   - $a = 3, b = 2,\ b < a \Leftrightarrow\ 2 < 3 \implies C_{yz}(2,3) =\ true \implies \exists\ b\ \in D_y$

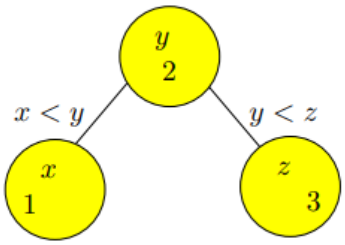
Deoarece pentru valoarile $a = 0\ si\ a = 2 \in D_y, \nexists b \in D_z$ valorile 0 si 2 vor fi eliminate din domeniul de definitie al variabilei y.



Din cauza eliminarii unei valori din domeniul de definitiei al variabilei y in urma aplicarii arc-consistentei asupra constrangerii C_{yz}, trebuie aplicata iar arc-consistenta asupra constrangerii C_{xy}.

1. Pentru fiecare valoare $a \in D_x$ daca exista o valoare $b \in D_y$ astfel încat $C_{xy}(a, b) = true$
   - $a = 1, b = 2,\ a < b \Leftrightarrow\ 1 < 2 \implies C_{xy}(1,2) = true \implies \exists\ b\ \in D_y$
   - $a = 2, b = 2,\ a < b \Leftrightarrow\ 2 < 2 \implies C_{xy}(2,2) = false$

Deoarece pentru valoarea $a = 2 \in D_y, \nexists b \in D_y$ valoarea 2 va fi eliminata din domeniul de definitie al variabilei x.



În acest moment nu mai există valori inconsistente în domeniile variabilelor. De asemenea putem observa că în urma aplicării conceptului de arc-consistență am obținut și soluția problemei. În cazul în care domeniul de definiție al unei variabile, în urma aplicării arc-consistenței, devine mulțimea vidă atunci se dovedește faptul că problema nu are soluție. De asemenea trebuie precizat că în urma aplicării arc-consistenței nu se pierde nici o soluție, se obține o problemă echivalentă, cu un spațiu de căutare mai mic,iar ordinea în care sunt eliminate valorile inconsistente din domeniile de definiție al variabilelor nu are importantă.

Așa cum am precizat deja în capitolul I când am descris pe scurt câteva metode de propagare a constrângerilor, cel mai folosit algoritm pentru asigurarea arc-consistenței este algoritmul AC-3. În cele ce urmează vom ilustra modul de funcționare al algoritmului printr-un mic exemplu având ca suport și pseudocodul algoritmului.

Algoritmul AC-3 – exemplu[9]

procedure AC-3(X, D, C)
  $Q := \{(i,j),(j,i) \mid c_{ij} \in C\}$
  while $Q \neq \emptyset$ **do**
    $(i,j) :=$ Fetch(Q)
    if Revise(i,j) **then**
      $Q := Q \cup \{(k,i) \mid c_{ki} \in C, k \neq j\}$

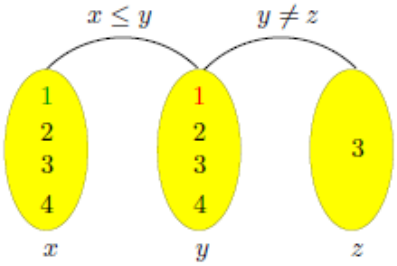
Fig.2 AC-3[10]

- Funcția $Reverse(i,j)$ elimină valori din domeniul D_i care nu au suport în domeniul D_j, în sensul că nu există o valoare $b \in D_j$ care să satisfacă constrângerea definită peste valorile variabilei i respectiv j. Funcția returnează *true* dacă valoarea a fost eliminată.

function Revise(i,j)
  change := false
  for each $a \in d_i$ **do**
    if $\forall_{b \in d_j} \neg c_{ij}(a,b)$ **ther**
      change := true
      remove a **from** d_i
  return change

Fig.3 [11]

- Consideram variabilele x, y, z cu domeniile $D_x = D_y = \{1,2,3,4\}, D_z = \{3\}$ si constrângerile $C_1: x \leq y$, $C_2: y \neq z$



Algoritmul folosește o mulțime de perechi Q alcătuită din posibile asignări pentru variabilele care au constrângeri asupra lor. Fiecare pereche este adăugată în această mulțime de două ori. Pentru exemplul considerat mulțimea Q va fi:

$$Q = \{(x, y), (y, x), (y, z), (z, y)\}$$

Pas1 : Algoritmul alege o pereche oarecare din mulțimea Q atâta timp cât mulțimea este diferită de mulțimea vidă. Perechea aleasă este eliminată din mulțimea Q.

Pas2 : Se aplică arc-consistența pentru perechea de variabile aleasă. Cum se aplică acesta metodă a fost arătat într-un exemplu mai sus. (pag 16)

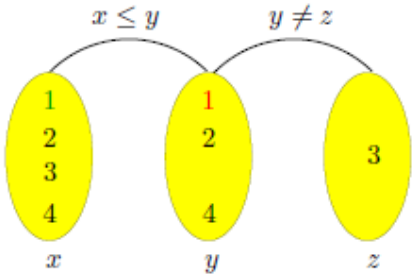
Pas3: Dacă au fost eliminate valori din domeniul unei variabile, atunci mulțimea Q trebuie actualizată. Vor fi adăugate acele perechi de variabile care la aplicarea arc-consistenței caută suport în domeniul variabilei care a fost modificat, perechea trebuie să fie diferită de cea asupra căreia tocmai a fost făcută verificarea.

Pentru exemplul considerat:

- alegem din mulțimea Q perechea (x, y). Aplicăm arc-consistența pentru aceste variabile, cautand suport pentru valorile variabilei x în domeniul variabilei y. In urma aplicarii arc-consistenței constatam că nu exista valori inconsistente în domeniul variabilei x. Mulțimea $Q = \{(y, x), (y, z), (z, y)\}$.
- Alegem o altă pereche din noua mulțime Q : (y, x). Aplicand arc-consistența domeniul variabilei y nu se modifică, deoarece nu se gasesc valori inconsistente. Mulțimea $Q = \{(y, z), (z, y)\}$.
- Alegem perechea (y, z). Aplicăm arc-inconsistenta si descoperim că valoarea 3 din domeniul variabilei y este valoare inconsistentă, deci este eliminată din domeniul acesteia. Mulțimea $Q = \{(z, y)\} \cup \{(x, y)\}$.
- Din nou alegem o pereche din Q : (z, y). Aplicam arc-consistenta si nu descoperim valori inconsistente în domeniul variabilei z. Mulțimea $Q = \{(z, y)\}$.
- Aplicăm arc-consistența si pentru ultima pereche din multimea Q si nu descoperim valori inconsistente. Mulțimea $Q = \{\emptyset\}$ deci algoritmul se opreste.

După aplicarea algoritmului AC-3 domeniile variabilelor sunt :

$$D_x = \{1,2,3\} \quad D_y = \{1,2,4\} \quad D_z = \{3\}$$



Complexitate spațiu: $O(e)$
Complexitate timp:$O(e \cdot m^3)$, unde $m = max_i\{|D_i|\}\, si\ e = |C|$

Algoritmul AC-3 a fost dezvoltat de Alan Mackworth în 1977 și este cel mai folosit algoritm pentru obținerea arc-consistenței, deoarece algoritmi anteriori au fost considerați ineficienți de cele mai multe ori, iar cei dezvoltați după acesta dificili de implementat.

II.4 Căutarea sistematică

Cei mai cunoscuți algoritmi de căutare sistematică sunt generate-and-test (GT) și Backtracking (BT).

1.Algoritmul generate-and-test (GT)

Ideea acestui algoritm provine din abordarea matematică în rezolvarea problemelor combinatoriale. Prima dată algoritmul ghicește Soluția și apoi testează dacă aceasta este consistentă, adică dacă soluția satisface constrângerile inițiale. În această paradigmă, se generează sistematic fiecare combinație posibilă de valori pentru variabilele implicate s se testează dacă asignarea de valori generată satisface toate constrângerile. Dacă rezultatul testului este pozitiv atunci acea asignare este soluția, deci prima combinație de valori ce satisface toate constrângerile este soluția problemei modelate. Numărul de combinații posibile prin această metodă este mărimea produsului cartezian al tuturor domeniilor variabilelor.

Algoritmul GT este un algoritm generic slab care se utilizează dacă totul esuează. Eficiența sa este slabă din cauza unui generator neinformat și descoperirea târzie a inconsistențelor. În mod vizibil, se poate obține o eficientă mult mai bună dacă validitatea constrângerii este testată de indată ce variabilele respective sunt instantiate. De fapt, această metodă este utilizată de abordarea backtracking.

2.Backtracking este numele unui algoritm general de descoperire a tuturor soluțiilor unei probleme de calcul, algoritm ce se bazează pe construirea incrementală de soluții-candidat, abandonând fiecare candidat parțial imediat ce devine clar că acesta nu are șanse să devină o soluție validă.[Wikipedia]

Exemplu [12]
În exemplul ce urmează vom ilustra pe scurt modul de funcționare al algoritmului. Vom considera problema CSP:

Variabile : x, y, z,t;

- Domeniile de definitie:

$$D_x = \{black, yellow\}, D_y = \{red, green\}, D_z = \{blue, green\}, D_t = \{black\}$$

- Constrangerile: $x \neq z$, $z \neq t$, $z \neq y$, $y \neq t$, $x \neq t$;

Problema este de fapt o problemă de colorare a nodurilor unui graf în care nodurile alăturate nu trebuie să aibă aceeași culoare. Graful este următorul:

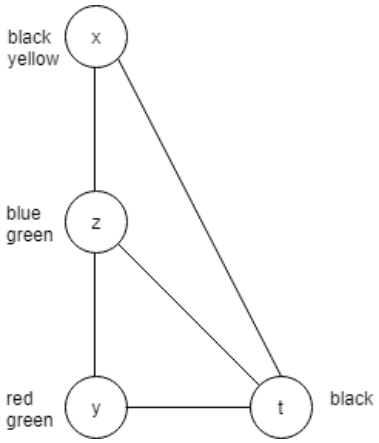


Fig.4

Arborele explorat de algoritmul Backtracking este:

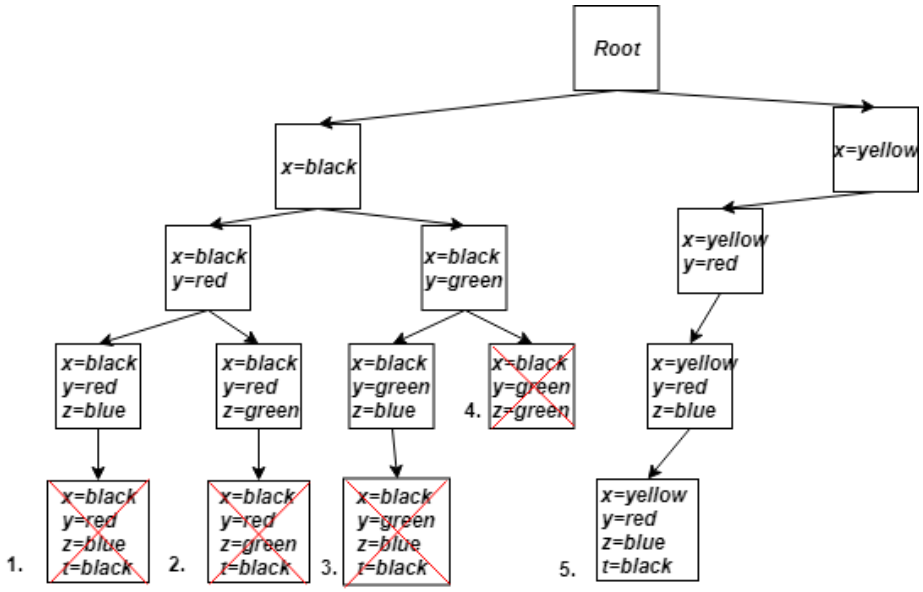


Fig.5

Observăm că problema are o singură soluție care este găsită la a cincea asignare completă a variabilelor. De fiecare dată când unei valori îi este asignată o valoare care nu poate conduce la găsirea soluției, algoritmul se întoarce un pas,adică la asignarea precedentă. Algoritmul poate să se oprească la găsirea primei soluții, așa cum se întâmplă în exemplul de mai sus, sau poate contiuna să caute toate soluțiile.

Există trei dezavantaje majore ale backtracking-ului standard:

- Thrashing, adică erori repetate din același motiv;
- Munca redundantă, adică valorile conflictuale ale variabilelor nu sunt amintite;
- Detectarea tardivă a conflictului, adică conflictul nu este detectat înainte ca acesta să apară cu adevărat;

Spațiul de căutare pentru acest algoritm este mare, de aceea este impracticabil pentru probleme de dimensiuni mari. O metodă de evitare a trashing-ului este „Backjumping", care este o variantă îmbunătățită a algoritmului Backtracking. Cu ce vine nou acest algoritm? Ei bine, acesta oferă posibilitatea de a reveni și la alte variabile, nu doar la variabila precedentă. Algoritmul poate identifica variabila care conduce la obținerea unei asignări nesatisfiabile. Cu alte cuvinte identifică „vinovatul" și face saltul înapoi la acesta.

3.Backjumping

In algoritmii de backtracking, backjumping este o tehnică care reduce spațiul de căutare, sporind astfel eficiența. În timp ce backtracking merge întotdeauna un singur nivel în arborele de căutare atunci când toate valorile pentru o variabilă au fost testate, backjumping poate merge mai multe nivele.[Wikipedia]

Pentru a ști câte nivele trebuie să se întoarcă backjumping-ul identifică vinovatul de obținerea unei asignări complete inconsistente. După identificare acestuia algoritmul revine la nivelul în care trebuie să asigneze o nouă valoare pentru variabila „vinovata".

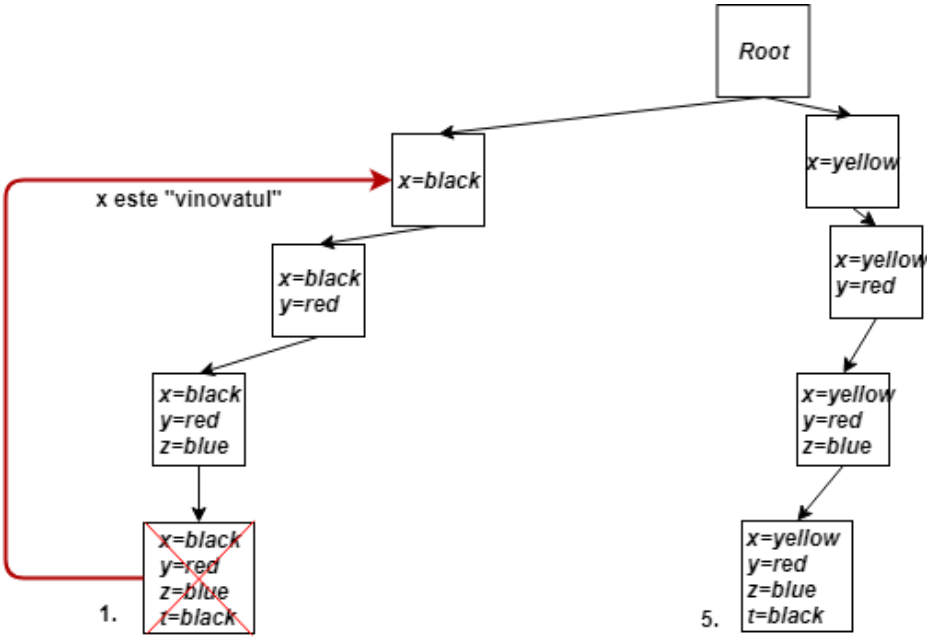


Fig.6

Pentru identificarea „vinovatului" există mai multe metode, unele dintre ele fiind: gaschnig's backjumping, graph-based backjumping, conflict-directed backjumping. Dar nu vom detalia aceste metode, intenția fiind de înțelegere a conceptului de backjumping, nu a modului de implementare.

În urma aplicării tehnicii de backjumping, așa cum putem observa comparând Fig.5 și Fig.6 Spațiul de căutare este mult mai mic, algoritmul găsind soluția a adoua asignare completă. Agoritmul backjumping nu testează niciodată asignarile complete numerotate în Fig5. cu 2 și

3, deoarece motivul inconsistenței celor două posibile soluții este același ca și în cazul asignări numerotate cu 1. Deci acest algoritm minimizează unul din dezavantajele backtracking-ului, numit trashing.

2.Backmarking

Backmarking este o variantă de backtracking optimizată din punct de vedere al verificării consistenței. Acest algoritm impune verificarea consistențelor în aceeași ordine în care sunt asignate valori variabilelor. Deoarece backtraking nu reține verificările de consistență pe care le-a verificat în trecut, acesta va efectua verificări de consistență care conduc la același rezultat din exact același motiv ca și la verificarea anterioară. Backmarking face acest lucru, adică reține verificările anterioare și astfel numărul de verificări este redus și algoritmul de bactracking optimizat.

Capitolul III – CHOCO SOLVER

III.1 Ce este Choco Solver?

Choco este un software gratuit și open-source dedicat programării constrângerilor. Este scris în Java, sub licență BSD. Scopul său este de a descrie problemele combinatoriale reale sub formă problemelor de satisfacție a constrângerilor și de a le rezolva cu ajutorul tehnicilor de programare a constrângerilor. Choco este utilizat pentru: [13]

- predare: ușor de utilizat
- cercetare: ușor de extins
- aplicații în viața reală: ușor de integrat

Utilizatorul își modelează problema într-un mod declarativ, precizând setul de constrângeri care trebuie îndeplinit pentru găsirea unei soluții. Apoi, problema este rezolvată prin alternarea algoritmilor de filtrare a constrângerilor cu un mecanism de căutare.[13]

Choco se numără printre cei mai rapizi solveri CP de pe piață. În 2013 și 2014, Choco a primit două medalii de argint și trei medalii de bronz la provocarea MiniZinc, care este competiția mondială a solver-elor CP. [13]

Choco solver este o biblioteca Java care include:

- diferite tipuri de variabile: integer, boolean, set, real;
- constrângeri: alldifferent, count, nvalues, etc.
- o căutare configurabilă: custom search, activity-based search, large neighborhood search, etc.
- explicații conflictuale: conflict-based back jumping, dynamic backtracking, path repair, etc.[13]

III.2 Modelare utilizand Choco Solver

1.Modelul[13]

Obiectul *Model* este componenta cheie. Acesta este construit după cum urmează:

```
Model model = new Model();
```

sau

```
Model model = new Model("my problem");
```

Aceasta trebuie să fie prima instrucțiune executată atunci când se crează un model în Choco pentru o problemă CSP.

2. Definirea variabilelor[13]

Tipurile de variabile existente: `BoolVar`, `IntVar`, `SetVar` si `RealVar;`

Pentru crearea unei variabile trebuie utilizat obiectul de tip Model definit ca mai sus. Variabilele create in modelul Choco sunt variabile a caror valoari nu sunt cunoscute, ele fiind determinate de solver, acesta fiind si scopul pentru care se creaza modelul. Crearea variabilelor se poate face astfel:

- variabile de tip integer: IntVar

```
// Create a constant variable equal to 42
IntVar v0 = model.intVar("v0", 42);
// Create a variable taking its value in [1, 3] (the value is 1, 2 or 3)
IntVar v1 = model.intVar("v1", 1, 3);
// Create a variable taking its value in {1, 3} (the value is 1 or 3)
IntVar v2 = model.intVar("v2", new int[]{1, 3});
```

Este posibila si crearea de vectori si matrici.

```
// Create an array of 5 variables taking their value in [-1, 1]
IntVar[] vs = model.intVarArray("vs", 5, -1, 1);
// Create a matrix of 5x6 variables taking their value in [-1, 1]
IntVar[][] vs = model.intVarMatrix("vs", 5, 6, -1, 1);
```

- variabile booleene: BoolVar

```
BoolVar b = model.boolVar("b");
```

- variabile de tip set : SetVar

O variabilă de tip set reprezintă o mulțime de numere întregi, adică valoarea unei astfel de variabile este o mulțime de numere. Domeniul său fiind definit de intervalul de setare $[LB, UB]$, unde:

  - limita inferioară, LB, este un obiect ISet care conține numere întregi care figurează în fiecare soluție;
  - limita superioară, UB, este un obiect ISet care conține numere întregi care se găsesc în cel puțin o soluție;

Valorile inițiale pentru LB și UB ar trebui să fie astfel încât LB să fie un subset al UB. Apoi, algoritmii de decizie si filtrare vor elimina numerele întregi din UB și vor adăuga altele la LB. O variabilă de tip set este instanțiată dacă și numai dacă LB = UB.

```
// Constant SetVar equal to {2,3,12}
SetVar x = model.setVar("x", new int[]{2,3,12});
```

```
// SetVar representing a subset of {1,2,3,5,12}
SetVar y = model.setVar("y", new int[]{}, new int[]{1,2,3,5,12});
// possible values: {}, {2}, {1,3,5} ...
// SetVar representing a superset of {2,3} and a subset of {1,2,3,5,12}
SetVar z = model.setVar("z", new int[]{2,3}, new int[]{1,2,3,5,12});
// possible values: {2,3}, {2,3,5}, {1,2,3,5} ...
```

- variabile de tip real : RealVar

Domeniul unei variabile de tip real este un interval de tip double. Conceptual valoarea unei variabile RealVar este o valoarea de tip double.

```
RealVar x = model.realVar("x", 0.2d, 3.4d, 0.001d);
```

3.Definirea constrângerilor[13]

Obiectul `Model` are o lista de constrângeri definite ce pot fi adaugate la modelul creat prin apelarea functiei `post()`.

```
model.allDifferent(vars).post();
```

Dacă metoda `post()`nu este apelată, constrângerea nu va fi luată în considerare în timpul procesului de soluționare: este posibil să nu fie satisfăcută în soluții.

```
// if x<0 then y>42
model.ifThen(
   model.arithm(x,"<",0),
   model.arithm(y,">",42)
);
```

4.Solver[13]

Obiectul de tip Solver rezolvă modelul creat utilizând obiectul de tip Model.

```
Solver solver = model.getSolver();
```

Solverul se ocupă de alternarea propagării constrângerilor cu căutarea și, eventual, învățarea, pentru a calcula soluțiile. Solver-ul poate calcula o singura soluție sau poate calcula toate soluțtiile unei probleme. Apelarea solver-ului se face astfel:

- pentru enumerarea tuturor solutiilor:

```
while(solver.solve()){
  // do something, e.g. print out variable values
}
```

- pentru gasirea unei singure soluții:

```
if(solver.solve()){
    // do something, e.g. print out variable values
}else if(solver.hasReachedLimit()){
    System.out.println("The solver could not find a solution
                        nor prove that none exists in the given
limits");
}else {
    System.out.println("The solver has proved the problem has no
solution");

}
```

5.Exemplu[13]

```
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;

/**
 * Trivial example showing how to use Choco Solver
 * to solve the equation system
 * x + y < 5
 * x * y = 4
 * with x in [0,5] and y in {2, 3, 8}
 *
 * @author Charles Prud'homme, Jean-Guillaume Fages
 * @since 9/02/2016
 */
public class Overview {

    public static void main(String[] args) {
        // 1. Create a Model
        Model model = new Model("my first problem");
        // 2. Create variables
        IntVar x = model.intVar("X", 0, 5);                 // x in [0,5]
        IntVar y = model.intVar("Y", new int[]{2, 3, 8});  // y in {2, 3,8}

        // 3. Post constraints
        model.arithm(x, "+", y, "<", 5).post(); // x + y < 5
        model.times(x,y,4).post();              // x * y = 4
        // 4. Solve the problem
        model.getSolver().solve();
        // 5. Print the solution
        System.out.println(x); // Prints X = 2
        System.out.println(y); // Prints Y = 2
    }
}
```

Capitolul IV – Prezentare aplicație practică

Aplicația practică are ca scop rezolvarea unei probleme de ambalare (bin packing) folosind paradigma de programare bazată pe constrângeri. Este cunoscut faptul că aceste probleme de ambalare sunt probleme NP-complete.

IV.1 Descrierea problemei

Problema presupune încărcarea diferitelor mărfuri (volume de produse chimice, care urmează să fie expediate pe navă) la containerele chimice disponibile ale navei.

De obicei planurile de încărcare a mărfurilor vrac sunt generate manual de planificatorii navei, deși este dificil să se genereze soluții de înaltă calitate. La alocarea substanțelor chimice la containerele navei trebuie să se țină cont atât de caracteristicile substanțelor, cât și de cele ale containerelor. Constrângerile care trebuie îndeplinite în general sunt constrângeri de segregare.

O primă constrângere care se impune este cea a împiedicării încărcării substanțelor chimice în anumite tipuri de containere. Este posibil ca substanța chimică ce urmează a fi alocată unui container să aibă nevoie de o temperatură controlată, iar acesta trebuie să fie echipat cu sistem de încălzire, în caz contrar substanța nu poate fi atribuită rezervorului respectiv. De asemenea rezervorul în care urmează a fi încărcată substanța chimică trebuie să fie rezistent la aceasta.

Datorită refolosirii rezervoarelor și a dorinței de minimizare a costului de curățare a acestora în urma efectuării unui transport, un rezervor poate fi contaminat de încărcăturile anterioare, incompatibile cu substanța chimică ce se dorește a se încărca în acesta.

O a doua constrângere este legată de poziționarea unor încărcături în containere adiacente, vecine. Trebuie prevenit ca unele perechi de substanțe să fie plasate unele lângă celelalte: trebuie luate în considerare nu doar interacțiunile chimice dintre diferite substanțe, ci și temperatura la care trebuie transportate . Cerințele de temperatură prea diferite pentru containere adiacente determină ca încărcătura din al doilea container să se solidifice din cauza răcirii primei încărcături sau prima încărcătură poate deveni instabilă din punct de vedere chimic datorită încălzirii celei de-a doua încărcături și reciproc.

Pentru a minimiza costurile și neplăcerile curățării rezervoarelor, un plan de încărcare ideal ar trebui să maximizeze volumul total al rezervoarelor neutilizate.

Pentru fiecare container se cunosc substanțele ce nu pot fi încărcate în acesta, precum și vecinii acestuia. De asemenea se cunosc și perechile de substanțe ce nu pot fi adiacente.

IV.2 Modelul CSP al problemei

1.Definirea variabilelor:

- Fie:
  - n : numărul de containere de pe navă
  - p : numărul de substanțe chimice
  - $neighbours_i$ mulțimea tuturor vecinilor containerului i , $i = \{1, \dots, n\}$
  - $volumeCargo_j$: volumul de substanță de tip j , $j = \{1, \dots, p\}$
  - $capa$: capacitatea containerelor
  - $imposCargo_i$: mulțimea substanțelor incompatibile cu containerul i, $i = \{1, \dots, n\}$
  - $incompatibilities$: mulțimea de perechi de substanțe ce nu pot fi adiacente

Datele de mai sus sunt cele cunoscute, iar cu ajutorul acestora vom determina constrângerile pentru a obține o asignare consistentă pentru mulțimea de variabile de mai jos.

- tankAlloc = $\{tankAlloc_1, tankAlloc_2, \dots, tankAlloc_n\}$ – mulțimea variabilelor cărora trebuie să li se atribuie valori

2.Definirea domeniului fiecărei variabile:

- $dom(tankAlloc_i) = \{0,1, \dots, \mathrm{p}\}$ – domeniul fiecărei variabile din mulțimea tankAlloc

Fiecărui container îi poate fi atribuită cel mult o substanță. Fiecare tip de substanță este definit printr-un număr din mulțimea {0,1,…,p} valoarea -1 semnificând faptul că respectivului container nu îi este alocată nicio substanță.

3.Definirea constrângerilor

- O primă constrângere poate fi definită asupra cantității de substanță de fiecare tip ce trebuie încărcată în containere. Aceasta va asigura faptul că întregul volum de substanță va fi încărcat în containere. Deci suma volumelor tuturor containerelor cărora le este atribuită substanța j trebuie să fie cel puțin egală cu volumul de substanță de tip j ce trebuie încărcat pe navă.

  C1: $\sum_{i=1}^{n}(tankAlloc_i = j) \cdot capa \geq volumeCargo_j$, pentru fiecare $j = \{1, \dots, p\}$;

- A doua constrângere ce se impune a fi definită este asupra tipurilor de substanțe ce pot fi alocate unui container. Aceata constrângere va evita încărcarea unor substanțe în containere ce sunt incompatibile cu acestea. Adică se va evita încărcarea unei substanțe chimice într-un container ce nu este rezistent la aceasta sau într-un container ce conține reziduri de la încărcări anterioare.

  C2: $tankAlloc_i \neq imposCago_{ir,}\ r = \{1, \dots, len(imposCargo_i)\}$, pentru $\forall\ i = \{1, \dots, n\}$;

- O ultimă constrângere ce se impune a fi definită este asupra substanțelor adiacente. Mai exact a asigurării faptului că substanțele cu caracteristici și condiții de transport foarte diferite nu sunt încărcate în containere adiacente, acest lucru ducând la alterarea acestora sau la pericole mai mari.

C3: $\{tankAlloc_i, tankAlloc_{neighbours_{ir}}\} \notin incompatibilities$
$, \forall\, i = \{1, .., n\}\, si\, \forall\, r = \{1, \dots, len(neighbours_i)\}$

4.Exemplu

Fie:

- $n = 17\ ,\ p = 10\ , capa = 1000$
- Substanțele:
  1. Fenol $volumeCargo_1 = 1200;$
  2. Toluen $volumeCargo_2 = 100;$
  3. Acid sulfuric $volumeCargo_3 = 900;$
  4. Ulei de palmier $volumeCargo_4 = 1700;$
  5. Acetona $volumeCargo_5 = 800;$
  6. Benzen $volumeCargo_6 = 500;$
  7. Tetraclorura de carbon $volumeCargo_7 = 1300;$
  8. Acrilonitril $volumeCargo_8 = 540;$
  9. Acid azotic $volumeCargo_9 = 630;$
  10. Furfurol $volumeCargo_{10} = 720;$
- Incompatibilitățile container – substanță:
  - $imposCargo_1 = \{9,10\}$
  - $imposCargo_5 = \{4\}$
  - $imposCargo_9 = \{10\}$
  - $imposCargo_{10} = \{7\}$
  - $imposCargo_{13} = \{1\}$
  - $imposCargo_{15} = \{8\}$
- Incompatibilitățile substanță - substanță :
  - $incompatibilities = \{(5,6)\ ,(7,3)\ ,(8,9)\ ,(10,4)\}\ ;$
- Aşezarea containerelor (vecinii):

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17			

Soluția problemei definite mai sus este:

- $tankAlloc_1 = empty$
- $tankAlloc_2 = Ulei\ de\ palmier$
- $tankAlloc_3 = Tetraclorura\ de\ carbon$
- $tankAlloc_4 = empty$
- $tankAlloc_5 = Acrilonitril$
- $tankAlloc_6 = Acetona$
- $tankAlloc_7 = Fenol$
- $tankAlloc_8 = empty$
- $tankAlloc_9 = Benzen$
- $tankAlloc_{10} = Ulei\ de\ palmier$
- $tankAlloc_{11} = Tetraclorura\ de\ carbon$
- $tankAlloc_{12} = Fenol$
- $tankAlloc_{13} = empty$
- $tankAlloc_{14} = Acid\ azotic$
- $tankAlloc_{15} = Toluen$
- $tankAlloc_{16} = Acid\ sulfuric$
- $tankAlloc_{17} = Furfurol$

IV.3 Modelul Choco al problemei

Modelarea problemei în Choco presupune crearea unui model pentru care sunt definite variabile și constrângeri.

- Variabilele Choco nu sunt instantiate, pentru ele se specifică doar intervalul în care pot lua valori. Scopul modelului Choco fiind să asigneze valori acestor variabile respectând constrângerile definite.
- Constrângerile sunt definite peste variabilele Choco, dar și peste variabilele Java.

De obicei, variabilele Java stochează informațiile oferite de cerința problemei și ajută la construirea constrângerilor asupra variabilelor Choco.

Modelarea problemei se face astfel:

1. **Crearea modelului:**

```
Model model = new Model("TankAllocation");
```

Prin această instrucțiune se crează un obiect de tipul Model, cu numele „TankAllocation”. Această instrucțiune trebuie să fie prioritară față de orice altă instrucțiune, deoarece acest obiect este cheia rezolvării unei probleme în Choco.

```
public Model(String name)
```

> Creates a Model object to formulate a decision problem by declaring variables and posting constraints. The model is named `name` and uses the default (trailing) backtracking environment.
> **Parameters:** `name` - The name of the model (for logging purpose) [14]

2. **Crearea variabilelor specifice limbajului Java(**în care sunt stocate informații)

```
static int dTanks;
static int dCargos;
static int capa;
```

Unde:

- *dTanks* = numarul de containere de pe nava
- *capa* = capacitatea unui container
- *dCargos* = numarul de substante ce trebuie incarcate in containere

```
//lista substantelor ce trebuie incarcate pe nava
static ArrayList<Cargo> aCargo= new ArrayList<Cargo>();
   public class Cargo {
         int id;
         String name;
         int volume;
         public int getId() {
               return id;
         }
         public void setId(int id) {
               this.id = id;
         }
         public String getName() {
               return name;
         }
         public void setName(String name) {
               this.name = name;
         }
         public int getVolume() {
               return volume;
         }
         public void setVolume(int volume) {
               this.volume = volume;
         }

   }
```

- *aCargo* este o listă de obiecte de tip `Cargo` în care sunt stocate informațiile despre substanțele ce trebuie încărcate în containerele navei. Așa cum se observă în definirea clasei Cargo de mai sus, pentru fiecare substanță sunt reținute:
  - un id de identificare al substanţei
  - numele acesteia
  - volumul de substanţă ce trebuie încărcat în containere

```
//lista containerelor adiacente
static ArrayList<ArrayList<Integer>> neighbours = new
ArrayList<ArrayList<Integer>>();
```

- Variabila *neighbours* este o listă de liste în care lista cu indexul i, $neighbours_i$, reprezintă mulţimea tuturor id – urilor containerelor adiacente cu container-ul al carui id este egal cu i.

```
//lista substantelor ce nu pot fi incarcate in fiecare container
   static ArrayList<ArrayList<Integer>> impossiblecargos = new
ArrayList<ArrayList<Integer>>();
```

- Variabila *impossiblecargos* este o listă de liste în care lista cu indexul i, $impossiblecargos_i$, reprezintă mulţimea id – urilor substanţelor ce nu pot fi încărcate în containerul cu id - ul egal cu i.

```
// lista perechilor de substante ce nu pot fi incarcate in containere
adiacente
static ArrayList<ArrayList<Integer>> incompatibilities = new
ArrayList<ArrayList<Integer>>();
```

- $incompatibilities_i$ reprezintă o listă ce conține 2 id – uri semnificând substanțele ce nu pot fi încărcate în containere vecine.

3. Definirea variabilelor Choco

```
IntVar tankAllocation[]=model.intVarArray("tank",dTanks, -1, dCargos-1);
```

intVarArray [15]

`default IntVar[] intVarArray(String name, int size, int lb, int ub)`
Creates an array of *size* integer variables, taking their domain in [*lb*, *ub*] Uses an enumerated domain if *ub-lb* is small, and a bounded domain otherwise

Parameters:

`name` - prefix name of the variables to create. The ith variable will be named *name*[i]

`size` - number of variables

`lb` - initial domain lower bound of each variable

`ub` - initial domain upper bound of each variable

Returns:

an array of *size* IntVar of domain [*lb*, *ub*]

Instrucțiunea de mai sus creează un array numit „tank” de dimensiunea dTanks, unde fiecare variabilă $tank_i$ ia valori în intervalul $[-1, dCargos - 1]$. Numele „tank” este folosit la afișarea valorii variabilei $tankAllocation_i$. De exemplu, atunci când vom dori să afișăm variabila $tankAllocation_3$, care presupunem că are asignată valoarea 5, ceea ce vom obține la consolă va fi $tank[3] = 5$. Pentru acest lucru este dat primul parametru al funcției `intVarArray`, pentru definirea constrângerilor fiind folosit în continuarea denumirea $tankAllocation$.

Variabila $tankAllocation$ este cea care va conține soluția problemei modelate. Scopul modelului creat fiind să se asigneze valori variabilelor $tankAllocation_i$ astfel încât să fie respectate toate constrângerile ce vor fi definite mai jos. O asignare ca $tank[3] = 5$ înseamnă că substanța cu id −ul 5 a fost repartizată containerului 3.

Valoarea -1 asignată unei variabile $tankAllocation_i$ înseamnă că nu a fost repartizată nici o substanță în containerul i. De asemenea trebuie precizat că pot exista mai multe containere

care vor fi încărcate cu aceeași substanță. Acest lucru se datorează volumului mare al substanței de un anumit tip care nu va încăpea într-un singur container.

4. Definirea constrângerilor

Mai jos vom prezenta si fiecare element folosit din API-ul Choco pentru modelarea constrângerilor , informații preluate din documentația Choco.

- **C1**: $\sum_{i=1}^{n}(tankAlloc_i = j) \cdot capa \geq volumeCargo_j$, pentru fiecare $j = \{1, \dots, p\}$;

```
for(int i=0;i<dCargos;i++){
   IntVar occ = model.intVar("occur_" + i, 0, dTanks, true);
   model.count(i,tankAllocation,occ).post();
   IntVar cap = model.intVar("cap",capa);
   model.arithm(occ,"*",cap, ">=", aCargo.get(i).volume).post();
   model.arithm(occ,"*",cap, "<=", aCargo.get(i).volume+capa).post();

   }
```

```
IntVar occ = model.intVar("occur_" + i, 0, dTanks, true);
```

intVar[16]

```
default IntVar intVar(String name, int lb, int ub,
boolean boundedDomain)
```
Create an integer variable of initial domain $[lb, ub]$

Parameters:

`name` - name of the variable

`lb` - initial domain lower bound

`ub` - initial domain upper bound

`boundedDomain` - specifies whether to use a bounded domain or an enumerated domain. When 'boundedDomain' only bounds modifications are handled (any value removals in the middle of the domain will be ignored).

Returns:

an IntVar of domain $[lb, ub]$

Variabila `occ` de tip intVar va conține numărul de containere la care a fost repartizată aceeași substanță. Aceasta putand lua valori in intervalul $[0, dTanks]$.

```
model.count(i,tankAllocation,occ).post();
```

Count[17]

```
default Constraint count(IntVar value, IntVar[] vars, IntVar limit)
```
Creates a count constraint. Let N be the number of variables of the vars collection assigned to value value; Enforce condition N = limit to hold.

Parameters:

`value` - a variable

`vars` - a vector of variables

`limit` - a variable

Instrucțiunea `model.count(i,tankAllocation,occ).post();` definește o constrângere care determină în variabila `occ` câte containere vor fi încărcate cu substanța i. Adică câte variabile din vectorul $tankAllocation$ vor avea atribuită valoarea i.

```
IntVar cap = model.intVar("cap",capa);
```

intVar[16]

```
default IntVar intVar(String name, int value)
```
Create a constant integer variable equal to *value*

Parameters:

`name` - name of the variable

`value` - value of the variable

Returns:

a constant IntVar of domain {*value*}

Funcția intVar creează o variabilă constantă de tip întreg.

```
model.arithm(occ,"*",cap, ">=", aCargo.get(i).volume).post();
model.arithm(occ,"*",cap, "<=", aCargo.get(i).volume+capa).post();
```

arithm[17]

```
default Constraint arithm(IntVar var, String op, int cste)
```
Creates an arithmetic constraint : var op cste, where op in {"=", "!=", ">","<",">=","<="}

Parameters:

`var` - a variable

`op` - an operator

`cste` - a constant

Prima instrucțiune în care este folosită metoda *arithm* creează constrângerea care impune ca totalul capacități containerelor cărora le este repartizată substanța i să fie mai mare sau egală cu volumul de substanță i c e trebuie încărcat în containere. Știm că în variabila occ va fi numărul de containere în care va fi substanța i, iar în variabila *cap* capacitatea unui container, deci `occ,"*",cap` reprezintă capacitatea totală a containerelor care vor fi încărcate cu substanța i.

În cea de-a doua instrucțiune în care este folosită metoda *arithm*, constrângerea creată impune ca totalul capacități containerelor cărora le este repartizată substanța i să fie mai mică sau egală cu suma volumului substanței respective și capacitatea unui container. Această constrângere a fost necesară pentru a nu se repartiza mai multe containere decât este nevoie pentru o substanță.

În concluzie, constrângerea C1 asigură repartizarea unei substanțe la exact atâtea containere încât întreaga cantitate de substanță să fie încărcată și să nu existe containere rezervate pentru o substanță, dar să nu avem cu ce să le încărcăm.

- **C2**: $tankAlloc_i \neq imposCago_{ir,}\ r = \{1, \dots, len(imposCargo_i)\}$, pentru $\forall\ i = \{1, \dots, n\}$;

```
    for(int i=0;i<dTanks;i++)
    {
       for(int j=0;j<impossiblecargos.get(i).size();j++)
       {
model.arithm(tankAllocation[i],"!=",impossiblecargos.get(i).get(j)).post();
       }

}
```

Metoda *arithm* este cea folosită și în definirea constrângerii C1, fiind diferit operatorul. În modelarea acestei constrângeri a fost folosit operatorul ≠. Pentru fiecare container se impune constrângerea ca substanța repartizată acestuia să fie diferită de substanțele aflate în lista de substanțe ce nu trebuie încărcate în acest container datorită condițiilor nepropice oferite de acesta.

- **C3:** $\{tankAlloc_i, tankAlloc_{neighbours_{ir}}\} \notin incompatibilities$

  $, \forall\ i = \{1, .., n\}\ si\ \forall\ r = \{1, \dots, len(neighbours_i)\}$

```
for(int i=0;i<dTanks;i++)
            for(int j=0;j<neighbours.get(i).size();j++)
                  for(int k=0;k<incompatibilities.size();k++)
                        {
                         model.ifThen(model.arithm(tankAllocation[i] ,"=",
incompatibilities.get(k).get(0)),model.arithm(tankAllocation[neighbours.get
(i).get(j)] ,"!=", incompatibilities.get(k).get(1)));
```

ifThen[18]

```
default void ifThen(Constraint ifCstr, Constraint thenCstr)
```

Posts a constraint ensuring that if *ifCstr* is satisfied, then *thenCstr* is satisfied as well BEWARE : it is automatically posted (it cannot be reified)

Parameters:

`ifCstr` - a constraint

`thenCstr` - a constraint

Constrângerea C3 asigură faptul că nici o pereche de substanțe incompatibile nu este încărcată în containere vecine. Astfel se verifică ca atunci când unui container îi este repartizată o substanță nici unul din vecinii săi nu au repartizată deja o substanță ce nu poate fi vecină cu cea care se dorește repartizată containerului în cauză.

Metoda ifThen primește ca parametrii două constrângeri aritmetice create prin apelul metodei arithm întâlnită și în definirea celorlate constrângeri. Metoda ifThen funcționează astfel: dacă prima constrângere este satisfăcută, atunci cea de-a doua constrângere este automat adăugată modelului.

5. Obţinerea soluţiei

Pentru obtinerea solutiei trebuie invocat un solver care sa rezolve modelul creat prin definirea variabilelor si postarea constrangerilor.

```
model.getSolver().solve();
            for(int i=0;i<dTanks;i++)
            System.out.println("Solution " +tankAllocation[i]);
```

getSolver[18]

```
public Solver getSolver()
```

Returns the unique and internal propagation and search object to solve this model.

Returns:

the unique and internal `Resolver` object.

Funcția getSolver returnează un obiect de tip Solver ce va fi folosit pentru rezolvarea modelului creat. Solver-ul utilizează tehnici de propagarea a constrângerilor și algoritmi de căutare pentru găsirea soluției.

Funcția solve lansează în execuție solver-ul returnat de funcția getSolver. După apelul funcției solve, în variabilele $tankAllocation_i$ vom avea soluția problemei modelate.

IV.4 Interfața aplicației

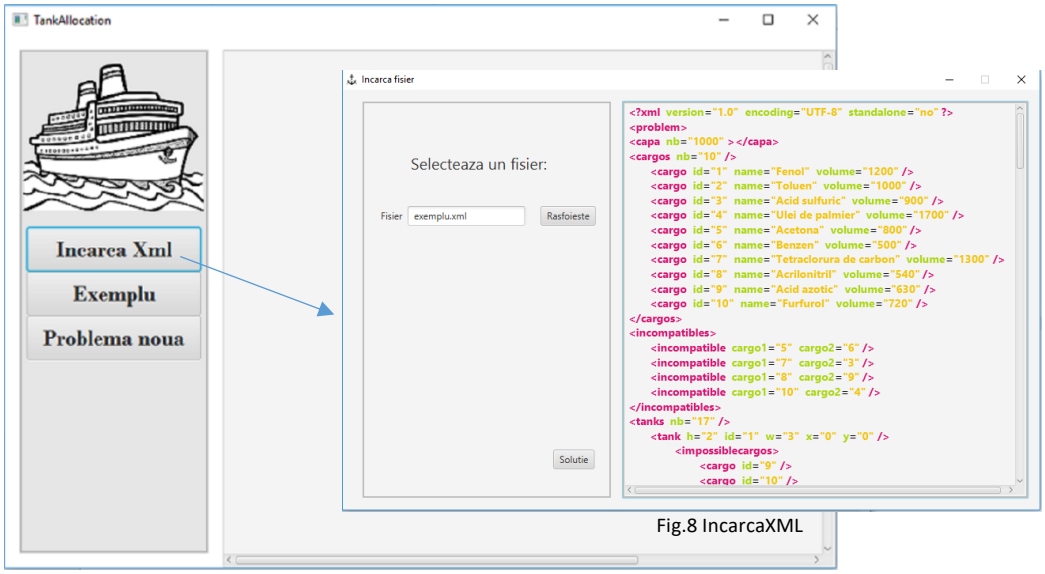


Fig.8 IncarcaXML

Fig.7 Meniul aplicatiei

Meniul aplicatiei:

- **Incarca XML** : la acționarea acestui buton, se deschide o nouă fereastră (Fig.8), care permite selectarea unui document în format XML din computer prin acționarea butonului *Rasfoieste*, document care trebuie să aibă o anumită structură specifică problemei modelate. După alegerea fișierului utilizatorul poate vizualiza soluția problemei. Soluția problemei va fi calculată la acționarea butonului *Solutie* care va închide fereastra *Incarca fisier* și va afișa soluția problemei în fereastra principală (Fig.7).

- **Exemplu** : această opțiune permite utilizatorului să vadă cum functionează aplicația de atribuire a substanțelor containerelor unei nave. Prin acționarea acestui buton este aplicat algoritmul asupra unui fișier xml stabilit. Fișierul fiind același ori de câte ori este acționat butonul Exemplu. Dacă în zona în care se afișează soluția este reprezentată soluția în urma acționarii butonului Exemplu, la o a doua acționare a acestuia se vor schimba doar culorile cu care sunt reprezentate substanțele.

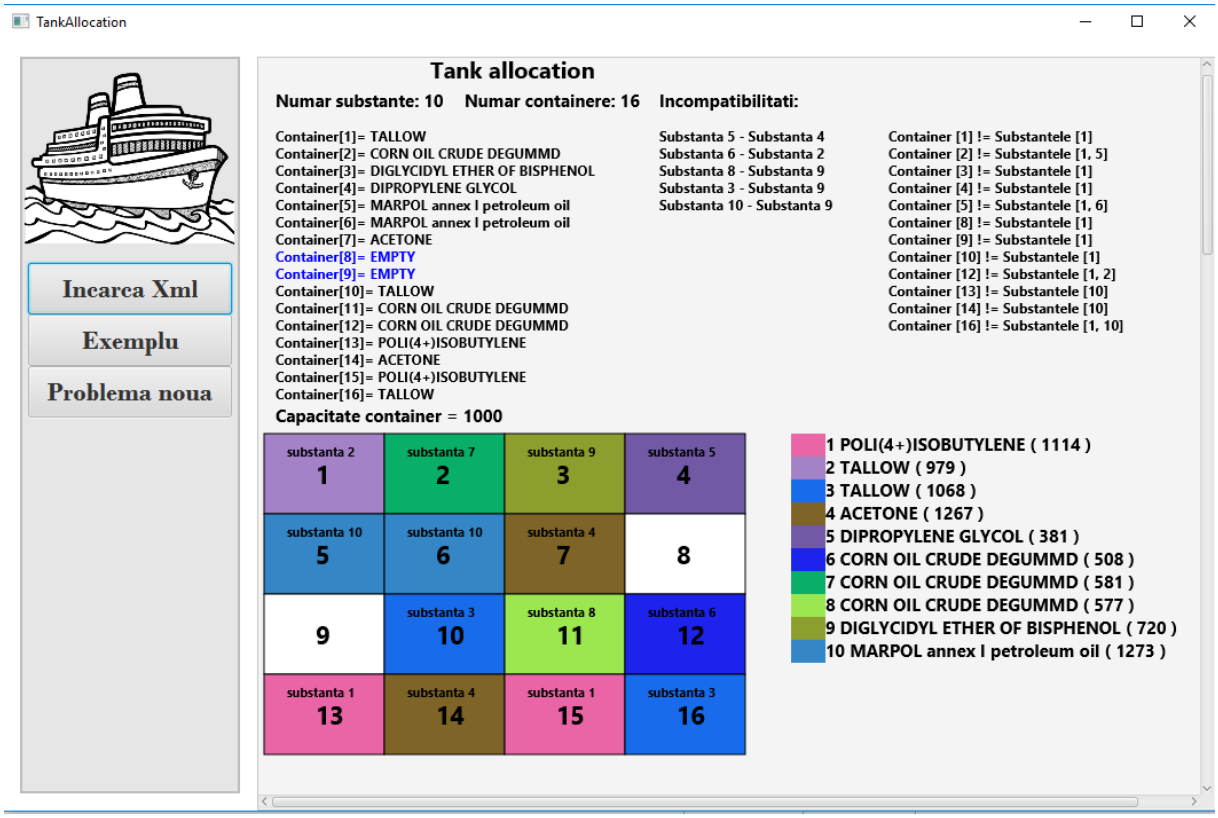


Fig.9 Exemplu

Afisarea solutiei unei probleme permite, asa cum se poate observa in Fig.9, vizualizarea:

- specificațiilor problemei rezolvate (nr de containere, substanțe, incompatibilități etc.) substanţei asignate fiecărui container
- constrângerilor pe care asignarea finală trebuie să le respecte
- unei reprezentări grafice a soluției, în care fiecare substanță are atribuită o culoare, ușurând identificarea containerelor ce au atribuită aceeași substanță
- containerele reprezentate cu alb sunt cele care nu au atribuită nici o substanță

- **Problema noua**: această opțiune permite utilizatorului să-și creeze propria problemă. Crearea unei noii probleme se face specificând următoarele informații:
  - numărul de containere
  - numărul de containere pe un rând
  - capacitatea unui container
  - numărul de substanțe

- denumirea și cantitatea de substanță
- perechile de substanțe ce nu pot fi încărcate în containere adiacente
- lista de substanțe ce nu pot fi încărcate în fiecare container

La acționarea butonului *Problema noua* se deschide o nouă fereastră în care utilizatorul poate introduce toate informațiile enumerate mai sus.

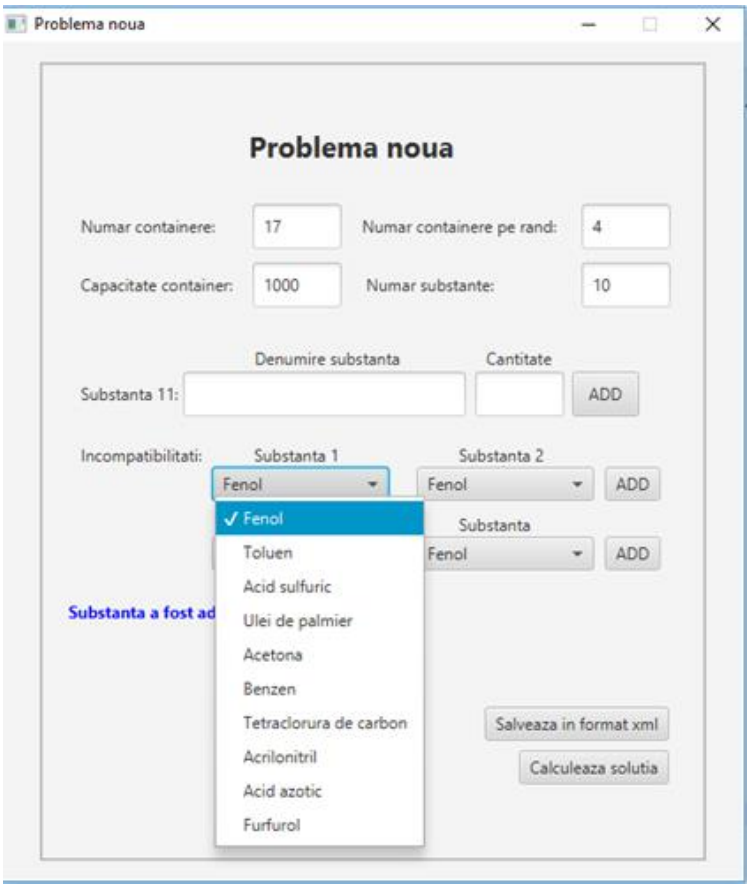


Fig.10

Crearea unei noii probleme necesită completarea mai întâi a câmpurilor : număr containere, număr containere pe rând, capacitate container, număr substanțe. Dacă utilizatorul nu comleteaza aceste 4 câmpuri, acesta nu va putea adăuga substanțele. Odată cu completarea celor 4 câmpuri, butonul *ADD* poate fi acționat, iar informația din cele 4 câmpuri este salvată, modificările ulterioare fiind nevalide.

În timp ce sunt adăugate substanțele, listele pentru adăugarea incompatiilitatilor sunt și ele actualizate.

După completarea tuturor datelor dorite, utilizatorul poate opta pentru una din opțiunile

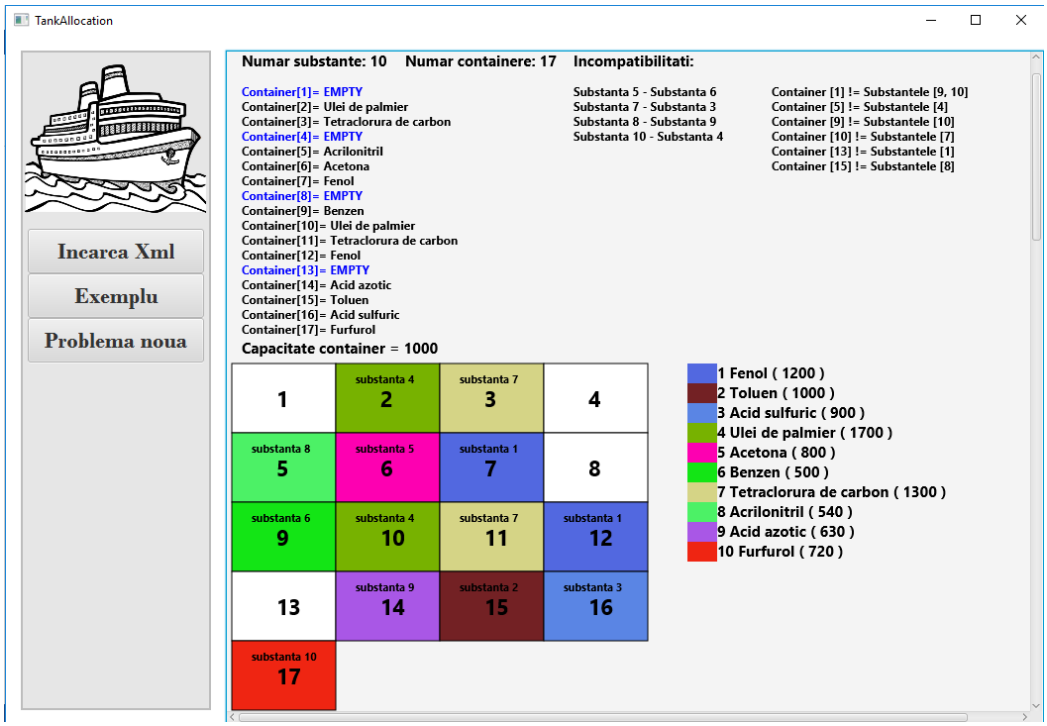
- *Salveaza in format xml*
- *Calculeaza solutie*

Functia *Salveaza in format xml* permite utilizatorului sa memoreze intr-un fisier xml datele introduse, respectandu-se un anumit format. Fișierul xml trebuie să respecte schema următoare :

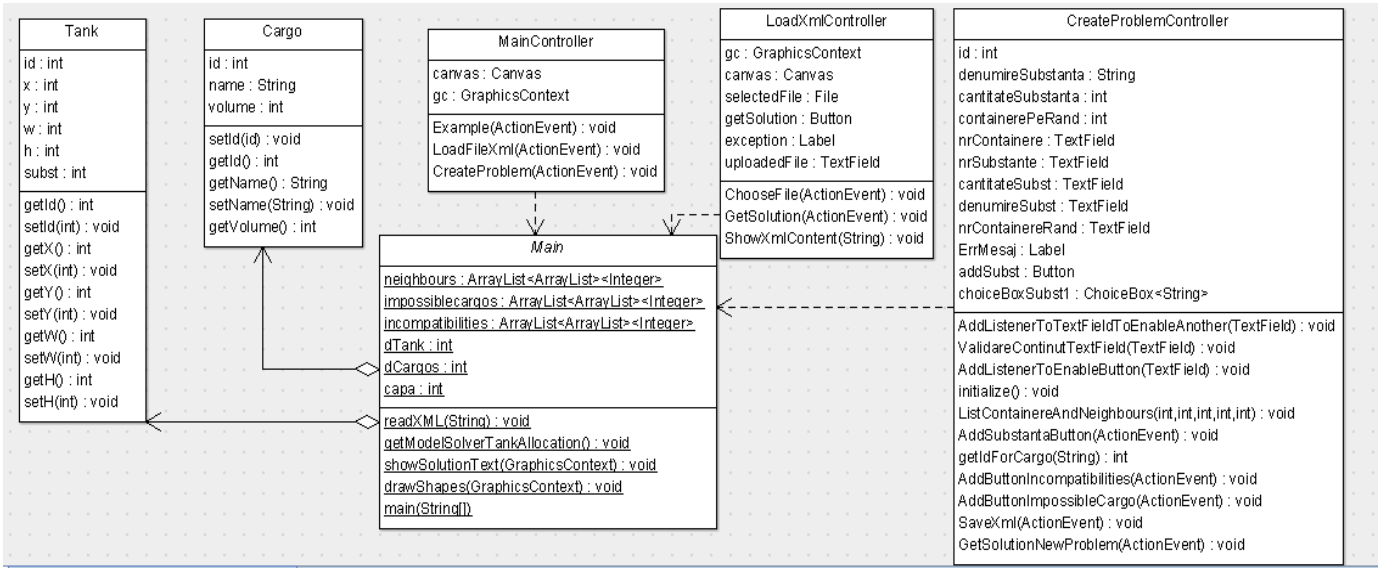
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" at
tributeFormDefault="unqualified">
<xs:element name="problem">
       <xs:element name="capa">
             <xs:attribute name="nb" type="xs:int"></xs:attribute>
      </xs:element>
      <xs:element name="cargos">
        <xs:sequence>
            <xs:element name="cargo" maxOccurs="unbounded">
                  <xs:attribute name="id" type="xs:int"></xs:attribute>
                  <xs:attribute name="name" type="xs:string"></xs:attribute>
                  <xs:attribute name="volume" type="xs:int"></xs:attribute>
            </xs:element>
        </xs:sequence>
            <xs:attribute name="nb" type="xs:int"></xs:attribute>
      </xs:element>
      <xs:element name="incompatibles">
        <xs:sequence>
            <xs:element name="incompatible" maxOccurs="unbounded">
                  <xs:attribute name="cargo1" type="xs:int"></xs:attribute>
                  <xs:attribute name="cargo2" type="xs:int"></xs:attribute>
            </xs:element>
        </xs:sequence>
     </xs:element>
     <xs:element name="tanks">
        <xs:sequence>
            <xs:element name="tank" maxOccurs="unbounded">
                  <xs:element name="impossiblecargos">
                     <xs:sequence>
                       <xs:element name="cargo">
                             <xs:attribute name="id" type="xs:int"></xs:attribute>
                       </xs:element>
                     </xs:sequence>
                  </xs:element>
                  <xs:element name="neighbours">
                     <xs:sequence>
                        <xs:element name="tank" maxOccurs="unbounded">
                              <xs:attribute name="id" type="xs:int"></xs:attribute>
                        </xs:element>
                     </xs:sequence>
                  </xs:element>
                  <xs:attribute name="id" type="xs:int"></xs:attribute>
                  <xs:attribute name="x" type="xs:int"></xs:attribute>
                  <xs:attribute name="y" type="xs:int"></xs:attribute>
                  <xs:attribute name="w" type="xs:int"></xs:attribute>
                  <xs:attribute name="h" type="xs:int"></xs:attribute>
                  </xs:element>
        <xs:sequence>
        <xs:attribute name="nb" type="xs:int"></xs:attribute>
  </xs:element>
</xs:element>
</xs:schema>
</problem>
```

După salvarea fișierului în format xml prin acționarea butonului *Salveaza in format xml* fereastra *Problema noua* rămâne în continuare deschisă, utilizatorul putând efectua și cealaltă opțiune, *Calculeaza solutia*.

Funcția *Calculeaza solutia* închide fereastra *Problema noua* și afișează soluția problemei in fereastra principală. Solutia pentru exemplul de la pagina 31 fiind:



IV.5 Diagrama de clase



IV.6 Rezultate experimentale

Nr.crt	Nr. containere	Nr. substanțe	Nr. Incomp. substanță- substanță	Nr. Incomp. substanță- container	Timp de execuție
1	28	17	7	32	0.03 min
2	28	18	7	32	0.16 min
3	28	19	7	32	1.16 min
4	28	20	7	32	9.85 min
5	28	20	8	32	0.13 min
6	28	20	8	35	0.08 min
7	28	20	7	35	3.78 min
8	30	20	7	32	10.35 min
9	30	20	8	32	0.02 min
10	30	20	7	35	7.94 min
11	34	19	7	32	5.74 min
12	34	19	8	32	0.03 min
13	34	20	7	32	24 min
14	34	20	8	32	0.42 min
15	34	20	7	41	18.38 min
16	34	20	8	41	1.33 min
17	34	20	7	45	24.45 min
18	34	20	8	45	1.06 min

Tabelul 1.

În urma analizării informațiilor din tabelul de mai sus, putem concluziona că:

- Performanța algoritmului depinde de mărimea datelor de intrare , pentru un număr relativ mic de substanțe și containere (20 respectiv 34) spațiul de căutare al soluției este foarte mare.
- Adăugarea unei singure substanțe la o problemă existentă crește semnificativ timpul de execuție (vezi liniile 3- 4, 11-13 din tabel)
- Adăugarea unei constrângeri de tipul substanță-substanță ce nu pot fi încărcate în containere adiacente scade timpul de execuție al algoritmului, în unele cazuri considerabil de mult. Acest lucru se întâmplă datorită tehnicii de propagare a constrângerilor care reduc domeniile variabilelor. (vezi liniile 8-9, 15-16, 17-18)
- Adăugarea unor constrângeri de tipul container- substanță, care restricționează încărcarea unei substanțe într-un anumit container, reduce de asemenea timpul de execuție, din același motiv ca și mai sus. (vezi liniile 4-7,8-10)

CONCLUZII

Această lucrare prezintă modalitatea de rezolvare a unei probleme din viața reală folosind paradigma de programare bazată pe constrângeri. Primul capitol ne ajută să înțelegem ce este aceea o problemă de satisfacere a constrângerilor, cum definim o astfel de problemă, ce presupune definirea unei probleme CSP (stabilirea variabilelor și domeniile acestora, definirea constrângerilor ce trebuie satisfăcute de valorile pe care variabilele le pot lua), cum să construim modelul matematic al unei astfel de probleme, dar și cum putem să identificăm o astfel de problemă în cele pe care le întâlnim zi de zi.

După ce ne-am familiarizat cu noțiune de problemă de satisfacere a cosntrangerilor, al doilea capitol ne prezintă o paradigmă de programare ce ne permite găsirea de soluții(o soluție specifică, o soluție oarecare sau toate soluțiile posibile) pentru modelul matematic construit atunci când am definit problema ca o una de satisfacere a constrângerilor. Acest capitol prezintă tehnici de rezolvare a modelului matematic cum sunt: propagarea constrângerilor, tehnicile de consistență și algorimii de căutare sistematică. Așa cum am văzut propagarea constrângerilor și consistența arcului nu conduc întotdeauna la soluția problemei, în cazul în care există, dar îmbunătățesc semnificativ performanța algoritmilor de căutare. Astfel algoritmii de căutare folosiți împreună cu tehnicile de consistență și propagarea constrângerilor conduc la rezolvarea eficientă a unor probleme al căror spațiu de căutare este considerabil de mare.

Capitolul III descrie biblioteca Choco care conține tipuri de variabile speciale(IntVar, BoolVar etc.) și metode specifice care operează cu aceste tipuri de variabile, dar și cu tipurile de bază Java (int, array etc.) și care oferă posibilitatea de transpunere a modelului matematic într-un limbaj de programare specific. Această bibliotecă permite definirea mulțimii de variabile, a domeniului acestuia și „postarea” constrângerilor. De asemenea, această bibliotecă pune la dispoziție și un obiect de tip Solver care utilizează algoritmi de căutarea,tehnicile de consistenat și propagarea cosntrangerilor pentru a rezolva modelul creat de noi.

Ultima parte a lucrării descrie aplicația de încărcare a unor substanțe în containerele unei nave, mai exact ne arată cum să folosim noțiunile prezentate mai sus pentru rezolvarea unei probleme concrete. Având textul problemei creem modelul matematic prin definirea variabilelor, domeniile acestora și constrângerile asupra valorilor variabilelor, după aceea se creează modelul Choco, cu exemplificarea fiecărui element folosit din această bibliotecă, și se obține soluția problemei. Aplicația practică oferă o interfată care permite testarea modelului creat pentru diverse date de intrare și vizualizarea grafică a soluției problemei .

Scopul principal al acestei lucrări fiind familiarizarea cu o paradigma de programare relativ nouă, dar care atrage atenția din ce în ce mai mult.

Bibliografie

- [1] Constraint Programming: In Pursuit of the Holy Grail
  - http://ktiml.mff.cuni.cz/~bartak/downloads/WDS99.pdf
- [2] Constraint satisfaction problem
  - https://en.wikipedia.org/wiki/Constraint_satisfaction_problem
- [3] Probleme de satisfacere a constrângerilor (Constraint Satisfaction Problems -CSP)
  - http://www.cs.ubbcluj.ro/~gabis/mas/Lectures/8_CSP&DSCP/CSP_DCSP%20(Romanian).pdf
- [4] Figura 1: Algoritmul Min-Conflicts pseudocode
  - https://en.wikipedia.org/wiki/Min-conflicts_algorithm
- [5] Definitia programarii bazate pe constrangeri
  - http://www.constraint.org/en/intro.html
- [6] Definitie pogramare bazata pe constrangri
  - users.utcluj.ro - Limbaje si paradigme de programare
- [7] Paradigma de programare
  - http://andrei.clubcisco.ro/cursuri/f/f-sym/2pp/cb/curs1.pdf
- [8] Constraint Programming
  - Constraint Programming, Francesca Rossi, Peter van Beek, TobyWalsh
  - Handbook of Knowledge Representation, Edited by F. van Harmelen, V. Lifschitz and B. Porter
- [9] Local Consistency, Enric Rodriguez-Carbonell
- [10] Local Consistency, Enric Rodriguez-Carbonell –slide 16
- [11] Local Consistency, Enric Rodriguez-Carbonell –slide 14
  - https://www.cs.upc.edu/~erodri/webpage/cps/theory/cp/local-consistency/slides.pdf
- [12] Exemplu backtraking
  - http://gki.informatik.unifreiburg.de/teaching/ss12/csp/backjumping_example.pdf
  - https://www.ibm.com/support/knowledgecenter/SSSA5P_12.5.1/ilog.odms.ide.help/OPL_Studio/opllanguser/topics/opl_languser_intro_whycp.html
- [13] Documentatie Choco Solver
  - https://choco-solver.readthedocs.io/en/latest/1_overview.html#about-choco-solver
- [14] API- ul Choco
  - http://www.choco-solver.org/apidocs/org/chocosolver/solver/Model.html
- [15] API- ul Choco
  - http://www.choco-solver.org/apidocs/org/chocosolver/solver/variables/IVariableFactory.html#intVarArray(java.lang.String,int,int,int)
- [16] API- ul Choco
  - http://www.choco-solver.org/apidocs/org/chocosolver/solver/variables/IVariableFactory.html
- [17] API- ul Choco

- http://www.choco-solver.org/apidocs/org/chocosolver/solver/constraints/IIntConstraintFactory.html

- [18] API- ul Choco
  - http://www.choco-solver.org/apidocs/org/chocosolver/solver/constraints/IReificationFactory.html