UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

METODE DE CREARE A RETELELOR DE SORTARE

propusă de

Dan Dragoș Nicolae Slusariuc

Sesiunea: *Iulie, 2019*

Coordonator științific

Lect.dr. Frăsinaru Cristian

IAȘI

2019

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

METODE DE CREARE A RETELELOR DE SORTARE

Dan Dragoș Nicolae Slusariuc

Sesiunea: *Iulia, 2019*

Coordonator științific

Lect.dr. Frăsinaru Cristian

Avizat,

Îndrumător Lucrare de Licență

Lect.dr. Frăsinaru Cristian

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemntatul(a) SLUSARIUC DAN DRAGOS NICOLAE domiciliul în Jud. Bistrita Nasaud Sat. Sărata nr. 1B născut(ă) la data de 22-06-1997, identificat prin CNP 1970622060033, absolvent(a) al(a) Universității „Alexandru Ioan Cuza" din Iași, Facultatea de Informatică specializarea Informatică, promoția 2019, declar pe propria răspundere,cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul: METODE DE CREARE A RETELELOR DE SORTARE elaborată sub îndrumarea dl. / d-na Lect.dr. Frăsinaru Cristian, pe care urmează să o susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

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

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării fasificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură ……..

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „ *METODE DE CREARE A RETELELOR DE SORTARE*”, 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 *Slusariuc Dan Dragos Nicolae*

_____

(semnătura în original)

CUPRINS

INTRODUCERE

Rețelele de sortare sunt o modalitate de a combina un algoritm de sortare cu paralelismul, iar acesta poate fi folosită în mai multe domenii:

- Calculatoare cu procesoare multiple
- Rețele de comutare

Ele sunt construite prin executarea unor comparații în serie folosind comparatori. Problema cea mai mare a rețelelor de sortare este aceea că minimiză numărul de comparatori în funcție de numărul de input-uri pe care îl primim.

O metodă de a rezolva această problemă este aceea că descompune problema în subprobleme pentru a minimiza numărul de comparatori.

Rețeaua de sortare primește n input-uri, execută o secvență de comparații și de schimbări, astfel încât la final o să returneze n output-uri. De exemplu, având ca input o listă $\{ X_1, X_2, X_3 \}$, acesta să returneze $\{ Y_1, Y_2, Y_3 \}$, unde $Y_1 \leq Y_2 \leq Y_3$.

Acesta a început să fie un subiect de cercetare din anul 1950. Prima rețea de sortare a fost realizată de O'Connor și Nelson, aceasta având posibilitatea să sorteze un șir de numere de lungimea n, unde $4 \leq n \leq 8$. Aceasta avea numărul minim de comparatori pentru $n \in \{ 4, 5, 6, 8 \}$, dar care necesită alți 2 comparatori pentru $n \in \{ 7 \}$. Rețeaua de sortare a fost îmbunătățită de Batchar în 1968 care a reușit să producă minimul de comparatori pentru $n \leq 8$.

EXEMPLU

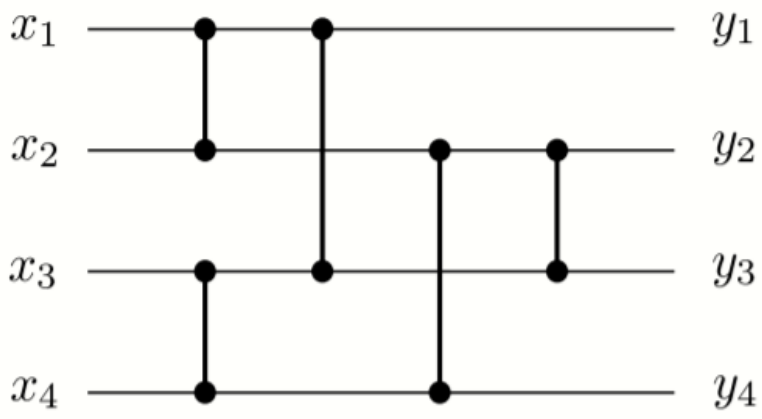

Figura 1. Individ

(http://www.technical-recipes.com/2015/a-genetic-algorithm-for-optimizing-sorting-networks-in-c/)

Așa cum putem observa în figura 1, având 4 input-uri $\{ x1, x2, x3, x4 \}$, pe care se vor aplica o serie de comparatori, vom ajunge la un output $\{ y1, y2, y3, y4 \}$, în care $\{ y1 \leq y2 \leq y3 \leq y4 \}$. Acesta este un exemplu destul de simplu, deoarece ca input se primesc doar 4 valori, iar problema noastră este atunci când primim un input cu mult mai mare ca acesta.

PRINCIPIUL ZERO-UNU

Acesta poate fi folosit prin reprezentarea input-urilor rețelei de sortare ca variabile booleane, iar output-urile ca și funcții ale acestor variabile.

Această modalitate creează mai multe funcții care sunt de fapt compoziții de conjuncții și de disjuncții care returnează o valoare booleană. De exemplu, atunci când avem mulțimea $\{ x1, x2, x3, x4 \}$, funcției f_i îi revine o combinație de disjuncții de conjuncții care conțin $n + 1 - i$ variabile. Prin urmare, rețeaua de sortare va deveni o secvență de comparații care returnează toate funcțiile generate prin această metodă. Acest lucru ne ajută să creăm o rețea minimală. Din cauza că aceste funcții au fost implementate de comparatori în rețea, problema poate fi restrânsă la găsirea unei secvențe de comparatori care să compună funcțiile finale (Valsalam și Miikkulainen, 2013).

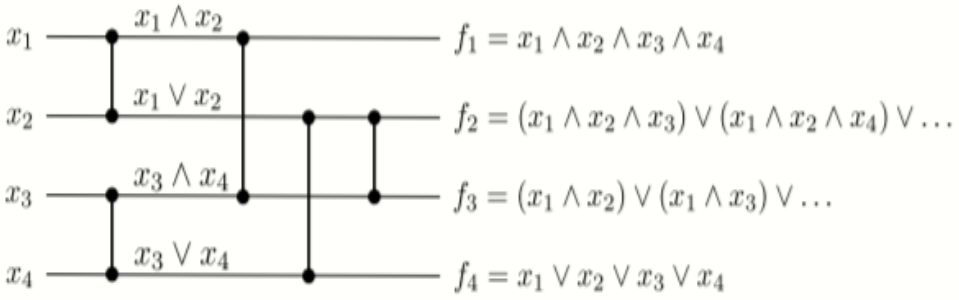

Figura 2. Convergent și Divergent

(Valsalam și Miikkulainen, 2013)

În figura 2 putem observa o mulțime de funcții boolene pentru o rețea de sortare cu patru input-uri. Principul zero-unu este de folos în a reprezenta input-urile rețelei ca variabile boolene. Fiecare comparator produce conjuncția input-urilor pe linia superioară și

disjuncția input-urilor pe linia inferioară => funcții booleene monotone. Din cauză că acestea sunt funcții booleene monotone, atunci putem vedea că ele au proprietatea ca $f(\boldsymbol{a}) \leq f(\boldsymbol{b})$ pentru oricare n tuple unde **a** = $a_1, \ldots, a_n$ și **b** = $b_1, \ldots, b_n$**,** astfel încât, **a** $\prec$ **b** dacă $a_i \leq b_i$ pentru $1 \leq i \leq n$. Un set format din toate 2^n tuple binare ordonate de $\prec$ se cheamă latice booleană. Aceasta poate fi ilustrată printr-un graf (Hasse diagram). Un exemplu de acest graf cu 4 input-uri se regăsește în figura 3 (Valsalam și Miikkulainen, 2013).

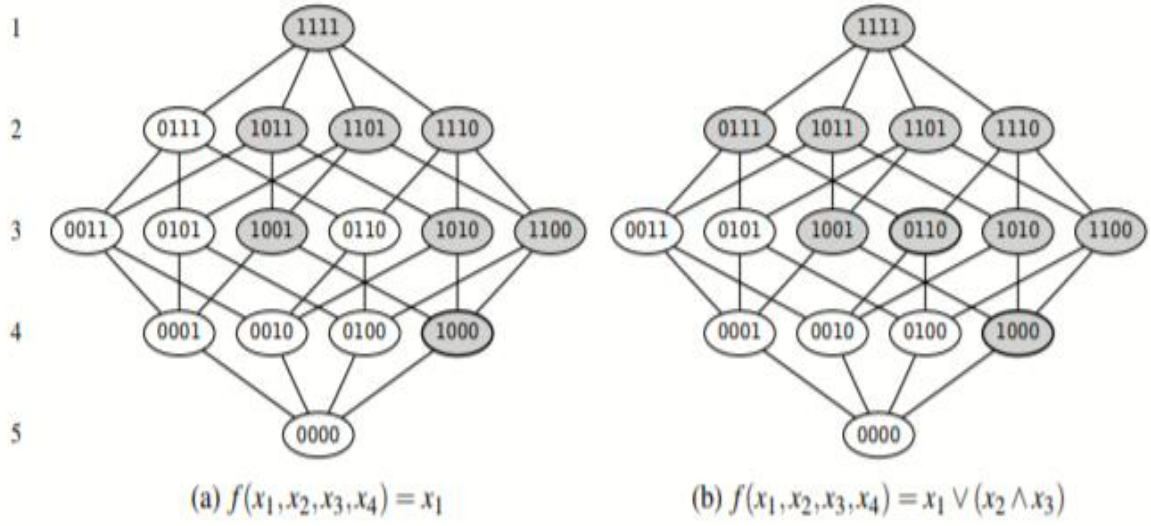

Figura 3. Diagrama Hasse (Valsalam și Miikkulainen, 2013)

Fiecare pereche de noduri sunt comparabile și conectate dacă pot fi ordonate de $\prec$. Un subset **X** de noduri sunt legate superior de nodul **y,** dacă **x** $\prec$ **y** pentru toate x ∈ X. La fel este și pentru legarea inferioară. După cum putem vedea un nod $b_1b_2b_3b_4$ are drum către un nod inferior $a_1a_2a_3a_4$, dacă $a_i \leq b_i$ pentru $1 \leq i \leq 4$. Prin urmare, dacă $a_1a_2a_3a_4$ este umbrit de o funcție monotonă f, atunci toate nodurile superioare la care poți ajunge din nodul respectiv sunt și ele umbrite, f fiind, astfel, complet definit de nodurile din

legăturile inferioare ale regiuni umbrite. Această reprezentare face posibilă crearea mai eficientă funcțiilor booleene.

Funcțiile monotone pot fi reprezentate de antilanțul legăturilor nodurilor în laticea booleană. Într-o latice de dimensiunea 2^n, dimensiunea maximă a reprezentării este egală cu dimensiunea celui mai mare antilanț, care este egală cu $\left(\frac{n}{n/2}\right) = O\left(\frac{2^n}{\sqrt{n}}\right)$. Construind conjuncții și disjuncții folosind această reprezentare, vom avea noduri redundante care trebuie eliminate. O modalitate de a rezolva această problemă este de a stoca valorile funcției într-un vector de biți de lungime 2^n, iar valoriile sunt grupate în funcție de nivelul la care se află în laticea booleană, astfel încât valoriile de pe orice nivel să poată fi extrase cu ușurință. Un algoritm eficient pentru numărarea de biți poate fi folosit pentru a determina dacă o rețea de sortare este validă, verificând dacă funcția cu output-ul i are valoarea 1 la toate nivelurile i pentru $1 \leq i \leq n$, care este de fapt cazul când toate output-urile funcțiilor f_i sunt construite corect (Valsalam și Miikkulainen, 2013).

DNF SI CNF

	DNF	CNF
f_1	$x_1 \wedge x_2 \wedge x_3 \wedge x_4$	$x_1 \wedge x_2 \wedge x_3 \wedge x_4$
f_2	$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee \ldots$	$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge \ldots$
f_3	$(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee \ldots$	$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge \ldots$
f_4	$x_1 \vee x_2 \vee x_3 \vee x_4$	$x_1 \vee x_2 \vee x_3 \vee x_4$

Figura 4. Simetria dintre DNF și CNF (Valsalam și Miikkulainen, 2013)

Dacă am scrie funcțiile booleene care ne sunt returnate de rețeaua de sortare în formă normală disjunctivă (DNF) și în formă normală conjunctivă (CNF), aceasta va fi un mod bun de a vizualiza simetria funcțiilor returnate. După cum putem vedea, schimbând conjuncțiile și disjuncțiile în forma (DNF) a funcțiilor f_2 sau f_3, aceasta va duce la forma (CNF). De asemenea, pentru operația de schimbare a disjuncțiilor și a conjuncțiilor în aceste 2 funcții și pentru schimbarea poziției liniilor, tabela rezultată a funcțiilor rămâne la fel ca în tabela originală. Aceasta funcționează pentru orice pereche de ($f_{i,}$ f_{5-i}). Operația care ține rezultatul funcțiilor rețelei se numește “simetrie”. Acestea pot fi folosite pentru minimiza numărul de comparatori ale rețelei (Valsalam și Miikkulainen, 2013).

III.1. Simetriile Rețelei de Sortare

Aceasta este o operație care se aplică funcțiilor de output ale rețelei și care lasă funcțiile invariante, astfel încât rețeaua rezultată să ramână neschimbată. Comform Valsalam și Miikkulainen, (2013), rezultatul inversării rețelei de sortare poate fi reprezentată ca $f_i(x_i, \dots, x_n, \wedge, \vee) = f_{n+1-i}(x_i, \dots, x_n, \wedge, \vee)$, pentru $1 \leq i \leq n \Rightarrow$ output-ul funcției f_{n+1-i} poate fi obținut de la f_i schimbând conjuncțiile și disjuncțiile și vice versa. Dacă schimbăm cele 2, iar funcțiile f_{n+1-i} și f_i sunt interschimbate, atunci rezultatul rețelei ramâne același (Valsalam și Miikkulainen, 2013).

Astfel, putem să definim operația simetriei σ_i pentru $1 \leq i \leq \left[\frac{n}{2}\right]$, care acționează asupra funcțiilor unei rețele schimbând funcția f_i și dualul ei f_{n+1-i}, și schimbându-le conjuncțiile și disjuncțiile. Compoziția operațiilor simetrice sunt și ele simetrice deoarece σ_i și σ_j operează independent pe diferite perechi de funcții din output. Prin urmare, acestea sunt și asociative și satisface toate axiomele unui grup pentru a putea fi reprezentată matematic. Din cauză că fiecare element din grup poate fi exprimat ca o compoziție de o mulțime de elemente $\sum = \{ \quad \sigma_1, \dots, \sigma_{\left[\frac{n}{2}\right]} \}$, grupul arată că este generat de Σ (Valsalam și Miikkulainen, 2013).

Subgrupurile lui ⟨Σ⟩, submulțimi care satisfac axiomele grupului, pot fi folosite să reprezentăm simetriile unor rețele parțiale create în procesul de construire a unei rețele complete de sortare. Deoarece fiecare element simetric în Σ operează peste perechi de funcții disjuncte, atunci acel subgrup poate fi scris ca ⟨Γ⟩, unde Γ este o submulțime a lui Σ. Inițial, înainte de a avea macăr un comparator, fiecare linie i în rețea are o funcție booleană x_i. Ca rezultat vedem ca Γ = {}. Adăugarea de comparatori care construiesc output-ul funcției f_i și dualul ei f_{n+1-i} duce la $\Gamma = \{\sigma_i\}$, care este o rețea parțială. Dacă vom adăuga mai mulți comparatori care construiesc f_i și dualul ei f_{n+1-i}, atunci vom crea altă rețea parțială cu $\Gamma = \{\sigma_i, \sigma_j\}$, iar această rețea parțială este mai simetrică. Continuând să adăugăm comparatori până când toate funcțiile de output care au fost construite produc o rețea completă de sortare cu Σ = Γ. Simetria poate fi folosită pentru a face o constrângere de căutare spațiu pentru rețele pe care le construim cu anumite proprietăți. O secvență de

subgrupuri poate reprezenta o secvență de câștiguri parțiale pentru a minimiza numărul de comparatori în rețea. Fiecare câștig parțial într-o secvență este definit ca subgrup care poate fi produs de câștigul precedent adăugând alți comparatori (Valsalam și Miikkulainen, 2013).

Dacă vom aplica acestea într-o rețea inițială cu simetria $\Gamma = \{\}$, primul câștig parțial este, de fapt, simetria care poate produce de la variabilele inputului construind perechi de funcții de output cu cât mai puțini comparatori. După cum putem observa, funcțiile $f_1 = x_1 \wedge ... \wedge x_n$ si $f_n = x_1 \vee ... \vee x_n$ au cele mai puține combinații de variabile, așadar, ele pot fi construite adăugând comparatori. Astfel, primul câștig parțial este producerea simetriei $\Gamma = \{\sigma_1\}$, folosind cât mai puțin comparatorii. După ce am construit f_1 și f_n, următoare pereche de funcții de output cu cele mai puține combinații de variabile, ar fi f_2 și f_{n-1}. Astfel, următorul câștig parțial ar fi să le construim și să producem simetria $\Gamma = \{\sigma_1, \sigma_2\}$. În acest fel, numărul de combinații de variabile în funcțiile de output vor crește de la stratul superior către cel de mijloc a rețelei. Având caștiguri parțiale care adaugă simetriei σ_k la Γ, următorul câștig parțial ar fi să adăugăm σ_{k+1} la Γ. Acest algoritm se oprește atunci când toate funcțiile de output au fost create. Prin acestea putem observa că e mai ușor să minimizăm numărul de comparatori necesari pentru fiecare caștig parțial decât pentru toată rețeaua (Valsalam și Miikkulainen, 2013).

III.2. Metoda de Reducere a Comparatorilor

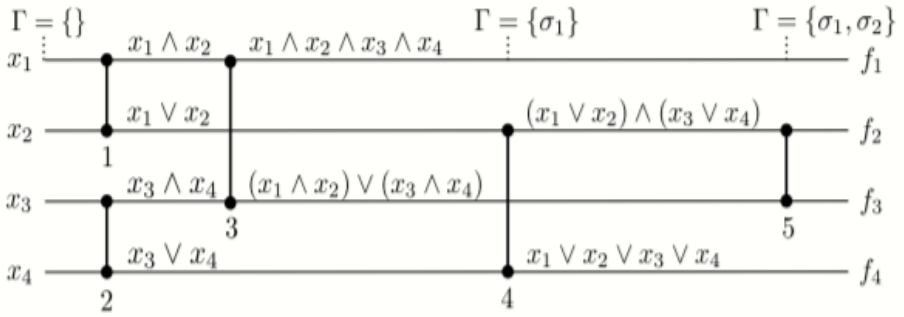

Figura 5. Metoda de reducere a comparatorilor (Valsalam și Miikkulainen, 2013)

Conform Valsalam și Miikkulainen, (2013), scopul final al metodei de reducere al comparatorilor este de a construi simetria $\Gamma = \{\sigma_i, \sigma_j\}$ prin executarea celor 4 funcții f_i prin folosirea numărului minim de comparatori. Acest lucru poate fi descompus într-o secvență de câștiguri parțiale considerându-le subgrupuri a grupului simetric $\langle\Gamma\rangle$. La fiecare pas al construcției, următorul câștig parțial este de fapt, subgrupul care poate fi produs adăugând cel mai puțin un număr de comparatori. Inițial, în rețea nu avem nicio simetrie, astfel încât $\Gamma = \{\}$. După cum putem observa în figura 5 f_1 și f_4 sunt cei mai ușor de construit, având cele mai puține combinații rezultând că numărul de comparatori va fi cel mai mic. În figura vedem că comparatorii 1 și 2 construiesc părți din f_1și f_4 pentru a ajunge la primul câștig parțial cu minimul număr de comparatori. Prin urmare, am ajuns la realizarea primei simetrii $\Gamma = \{\sigma_1\}$. Acum, pentru a ajunge la simetria $\Gamma = \{\sigma_1, \sigma_2\}$, ne vom folosi de f_2 și f_3. Adăugând comparatorul 5 ajungem la terminarea producerii câștigului parțial, având în vedere că comparatorii 3 și 4 au creat f_2și f_3 parțial. Optimizarea numărului de comparatori necesită să ajungem la fiecare câștig parțial pentru a putea să facem acest lucru scalabil, adică în a putea fi folosit pentru mai multe input-uri.

Laticea booleana poate fi folosită să determine dacă un comparator poate fi folosit în crearea a 2 funcții simultan. Să spunem că câștigul parțial este construit pe baza funcțiilor de output f_i si f_{n+1-i} atunci funcțiile de output mai mici ca *i* și mai mari decat $n + 1 - i$ sunt deja construite, însemnând că fiecare funcție f_j are valoare 1 la toate nodurile în nivelele mai mici sau egale decat *j* si valoare 0 pentru toate celelalte. Putem vedea că aceste funcții intermediare au valoare 1 la toate nodurile pe un nivel mai mic sau egal decat *i* si valoare 0 pentru toate nodurilor care sunt pe nivele mai mari decat $n + 1 - i$. Dacă asta nu ar fi adevărat atunci ar fi imposibil de construit alte funcții de output adăugând alți comparatori deoarece conjuncțiile țin zero-uri iar disjuncțiile unu-uri ale funcțiilor intermediare pe care le combină (Valsalam și Miikkulainen, 2013).

Câștigul parțial curent prin construirea funcției f_i necesită setarea valorilor a tuturor nodurilor la nivelul *i* cu 1 si la nivelul $i + 1$ cu 0, definind astfel, legăturile nodului în latice. Din cauză că funcțiile intermediare f_j' la liniile $i \leq j \leq n + 1 - i$ au deja valoare 1 la toate nodurile cu nivelul mai mic sau egal decat *i*, construind f_i acesta va reține valoare acestor

noduri automat. Atunci f_i poate fi construit prin setarea valorilor a toate nodurilor pe nivelul $i + 1$ cu 0 (Valsalam și Miikkulainen, 2013).

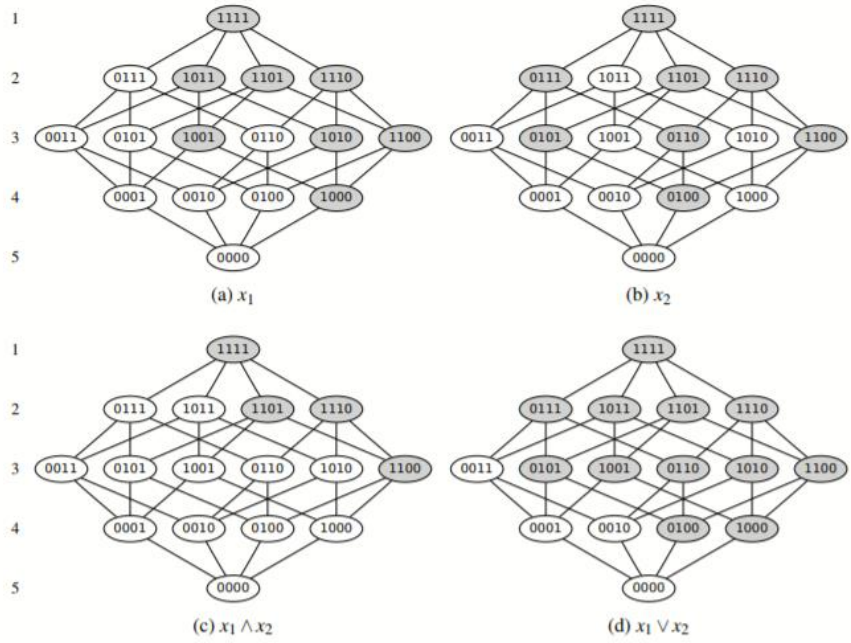

Figura 6. Hasse Graph (Valsalam și Miikkulainen, 2013).

Împărțirea de comparatori pentru a construi funcțiile duale de output intr-o rețea de sortare cu 4 input-uri. Comform Vinod K. Valsalam 2013 în această figură putem observa reprezentarea unei latice booleene a funcțiilor care sunt construite de comparatorul 1 în Figura 5. Nivelele laticei sunt în stânga iar nodurile în care funcția ia valoare 1 sunt umbrite. Comparatorul 1 va construi conjuncția și disjuncția funcțiilor (a) și (b) pentru câstigul parțial în construirea funcției de output $f_1 = x_1 \wedge x_2 \wedge x_3 \wedge x_4$. Funcția f_1 poate fi construită folosind conjuncții pentru a seta valorile la toate nodurile de pe nivelul 2 a laticei cu 0. La fel și f_4 poate fi construit folosind disjuncții pentru a seta valoare la toate nodurile a nivelului 4 cu 1. Împărțind comparatorii în această manieră reduce numărul de comparatori de care este nevoie pentru a construi rețeaua de sortare (Valsalam și Miikkulainen, 2013).

Valoarea unei funcții f_i a nodului pe nivelul $i + 1$ poate fi setată la 0 adăugând un comparator care construiește o conjuncție cu altă funcție care deja are valoare 0 la acel nod, acesta crescând numărul de noduri cu valoarea 0. Disjuncția cu care acest comparator construiește, are mai puține noduri cu valoare 0 atunci acestea nu sunt folositoare pentru

construirea lui f_i. Disjuncția va fi totuși folosită pentru a construi celelalte funcții de output implicând că nodurile de pe nivelul *i + 1* au valoare 0 și inputurile nu au noduri cu valori 0 comune pe acel nivel. Aceasta defapt sunt funcții intermediare f_j' care au valoarea 0 pentru orice nod de pe nivelul *i + 1* (Valsalam și Miikkulainen, 2013).

Adăugând un comparator între perechea de funcții, acesta colectează nodurile cu valoare 0 pentru ambele funcți în conjunctiva lor. Dacă repetăm acest proces recursiv, acesta va colecta nodurile cu valoriile 0 de pe nivelul *i + 1* pentru toate funcțiile până la linia *i*, acesta producând f_i. Acest lucru este similar și pentru funcția ei duală f_{n+1-i} care poate fi construită din funcțiile f_j' folosind disjuncții în locul conjuncțiilor pentru a seta valoare nodurilor de la nivelul *n + 1 – i* până la nivelul 1 (Valsalam și Miikkulainen, 2013).

Frunzele care sunt în rezultat binar al recursivității arborelui pentru f_i sunt de fapt funcțiile f_j' care au noduri cu valori 0 pe nivelul *i + 1* iar pe nodurile interne sunt output-urile comparatorului conjunctiv. Putem vedea că numărul de noduri de gradul 2 în arborele binar este cu 1 mai mic decât numărul de noduri, atunci numărul de comparatori necesari depinde de numărul de funcții cu care recursivitatea începe. Maximizând împărțirea de comparatori între 2 arbori de recursivitate minimizează numărul de comparatori pentru câștigul parțial curent. Mulți comparatori pot să aibă aceeași importanță iar atunci unul dintre acestea este ales aleator din listă și este adăugat la rețea. Dacă repetăm acest proces vom produce o secvență de comparatori care optimizează împărțirea în câștigul parțial curent și câștigul parțial mai îndepărtat (Valsalam și Miikkulainen, 2013).

Optimizând fiecare câstig parțial separat în această manieră constituie un algoritm greedy care produce minimul de rețele cu o probabilitate mare pentru $n \leq 8$. Pentru valori mai mare ale lui n, spațiul de cautare este prea mare pentru abordare greedy în a găsi optimul global. În aceste cazuri căutarea evolutivă poate fi folosită să exploreze vecinii soluțiilor greedy pentru o optimizare.

Capitolul IV.

ARHITECTURĂ

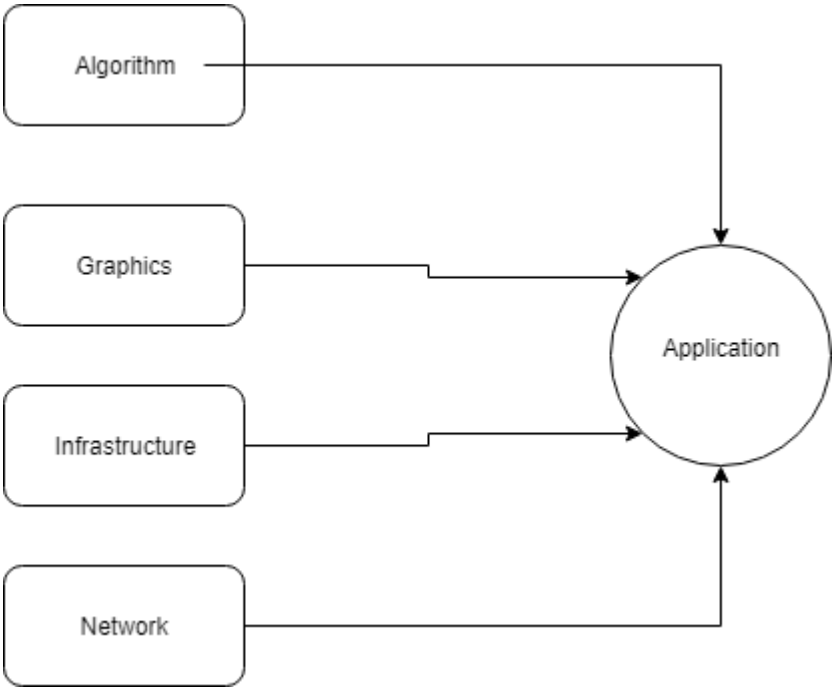

Figura 7. Structura de bază

IV.1. Algoritm

În această parte a proiectului avem partea principala a aplicației. Aceasta se ocupă cu manipularea anumitor indivizi a rețelei pentru a încerca să găsim o rețea cât mai bună.

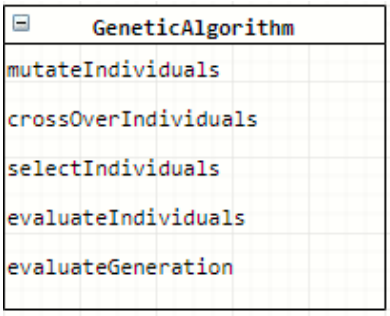

Figura 8. Metodele algoritmului genetic

În această clasă putem vedea cei 4 pași prin care trec indivizi pentru a fi manipulați astfel încât să aducă sau să se îmbunătățească comparatorii acestora. Acest pachet este

folosit pentru pachetul “Graphics’’ care se va ocupa cu afișarea acestor indivizi pentru a înțelege și mai bine care este rolul acestei aplicații.

IV.2. Grafică

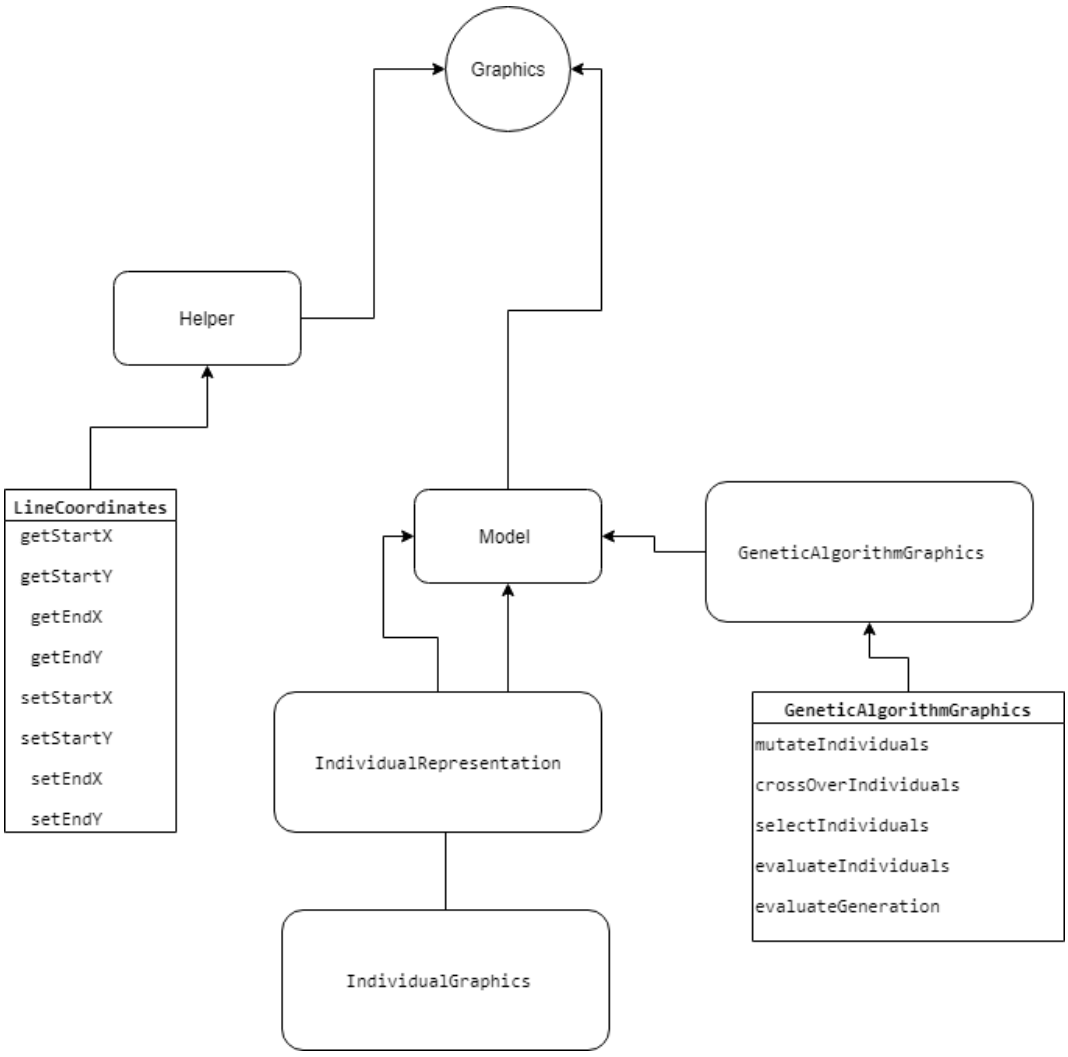

Figura 9. Modulul de Grafică

În acest pachet sunt extrași indivizi din fiecare generație, pentru a putea fi grafic afișați pe o interfață în scopul de a vizualiza ce se intamplă cu fiecare individ la fiecare generație. Pentru a ușura modalitatea de afisare a acestor indivizi au fost create 3 mari modele.

Pe primul loc se află IndividualReprezentation. Acesta se ocupă cu afișarea unui singur individ. Acest lucru se intamplă prin generarea unei noi instanțe de clasă prin acordarea datelor unui anumit individ, coordonatele unde va incepe afișarea acestuia grafic și care este numarul acestuia în generația din care face parte. Acest lucru este important pentru a putea face distincția dintre indivizii dintr-o anumită generație. În această interfață care va fi explicată mai jos se vor vedea ca fiecare generație este afisată rând pe rând în scopul de a putea face o evaloare vizuală asupra fiecărui individ dintr-o generație. Cu alte cuvinte acest lucru ne ajută la o vizualizare mai buna asupra imbunătățiri indivizilor pe parcursul generațiilor. Acest lucru poate fi observat cu ajutorul clasei GeneticAlgorithmGraphics.

O altă clasa folosită este IndividualGraphics care va afisa vizual corectitudinea unui individ și a rețelei lui de sortare. Prin acest lucru ne referim la a incerca să sortăm anumite siruri de numere în scopul de a vedea dacă acea retea reuseste să sorteze cât mai multe multimi de numere rezultând astfel un fitness mai bun.

Toate aceste lucruri sunt doar pentru o privire grafică asupra ceea ce se intamplă în spatele acestui algorim genetic. Cu toate acestea, vizualizarea indivizilor este o bună procedură pentru a putea să estimăm cat mai repede și mai eficient, cât timp ar lua o rulare a aplicației pentru a putea găsi o multime de comparatorii pentru a sorta o listă de numere de dimensiune n.

IV.3. Rețea

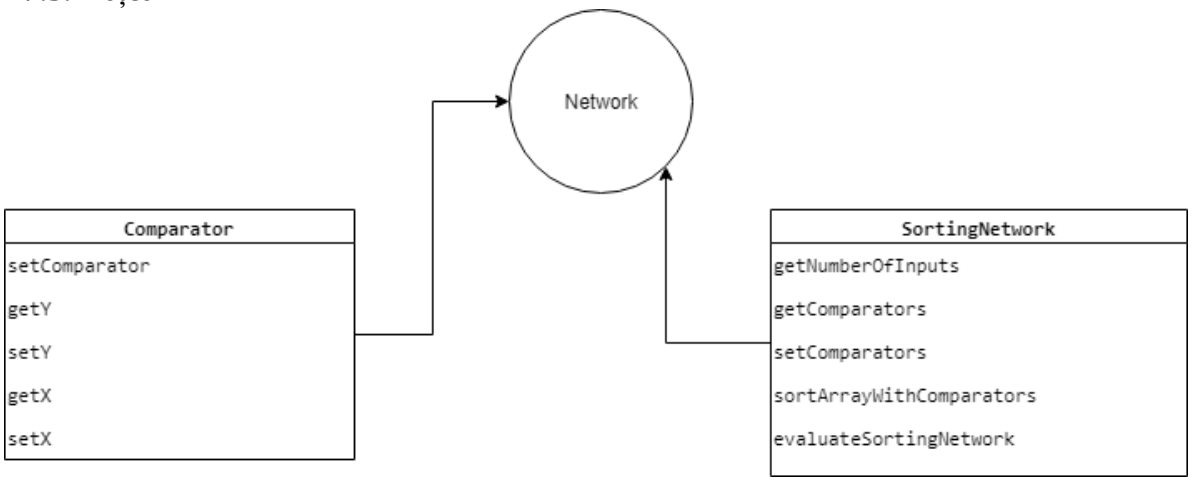

Figura 10. Modulul Rețea

În această parte a aplicației sunt realizate schimbări asupra comparatorilor, și verificarea rețelei de sortarea în scopul de a aduce anumite informați de ajutor pentru algoritmul genetic. O instantă a clasei rețea contine o listă de comparatorii și numărul de input-uri care se dau. Cu ajutorul acestora vom putea da acces algoritmului genetic de a putea extrage de la un individ toți comparatorii acestuia și să aplicăm pe acestia o sortare în funcție de comparatorii pe care îi avem în acea retea.

O altă importanță pe care o are această clasa este de a genera comparatorii pentru populația inițială. Acest lucru este foarte important deoarece aceasta se ocupa cu generarea de comparatorii astfel incât sa nu existe 2 comparatori lafel intr-o anumita rețea deoarece va rezulta printr-o redundanța unuia dintre aceștia.

IV.4 Infrastructură

In această parte a aplicatiei se vor crea posibilitati de siruri care trebuie sortate, se va verifica daca un sir este crescator și cateva proceduri de ajutor pentru celelalte parți ale aplicație.

Capitolul V.

ALGORITM GENETIC

Algoritmii genetici sunt o euristică inspirată de la Charles Darwin care se referă la evolutia naturala. Acest proces în final reflectă selecția naturală a acelor indivizi care au cel mai mare potential (care au reusit să se adapteze cel mai bine criteriilor de selecție specificați), acestia find mai predispusi să ajungă și în generațiile viitoare. Pentru a putea vedea cand un individ este mai bun decât alt individ, vom avea nevoie de fitness.

V.1. Fitness

O functie fitness determină cât de potrivit un anumit individ din populatie este pentru a putea să facă parte din generațiile urmatoare. În cazul acestei licențe a fost creată o funcție fitness care returnează cât de corect sortează un anumit sir de numere. De obicei se folosește o funcție fitness deoarece se doreste a ști dacă o anumită soluție este mai bună decât alta și de aceea aceste rezultate trebuie testate și sortate în funcție de cât de bună este o solutie față de cealaltă. În cazul în care putem ști cand o anumită soluție este o soluție completă, atunci vom putea să îi calculăm fitness-ul fără a trebui sa știm dacă această este o soluție mai bună decât alta.

In primul rând o funcție fitness trebuie să fie foarte bine definită deoarece având o functie fitness care nu știe exact daca o soluție este mai bună decât cealaltă, ar putea rezulta trecerea unor indivizi dintr-o generație în alta chiar dacă aceștia nu sunt neapărat cei mai bun indivizi. Explicarea cum se calculeaza fitness-ul se poate gasi mai jos de figura 11.

O altă proprietate pe care trebuie sa o aibe o funcție fitness este să fie cât mai eficientă. Acest lucru este important deoarece algoritmii genetici sunt algoritmi care au un timp de rulare destul de indelungat deci nu permite adăugarea de timp de executare în plus. Având o functie fitness eficientă reducem foarte mult complexitatea algorimului genetic.

Proprietatea important pe care se bazează o funcție fitness este în a putea sa dăm un anumit grad de performantă asupra unei soluți. În cazul nostru fitness-ul va fi un număr de la 1 la 100, acesta find de fapt raportul dintre numarul de liste de numere care au reușit să fie sortate la final și numărul de liste de numere care au incercat în a fi sortate.

Ultima proprietate a funcției fitness este ca acestea trebuie să aibă rezultate intuitive, asta se traduce prin concluzia ca orice individ care nu se descurcă bine va avea un fitness mai scăzut decât un individ care are o performanță mai bună.

```
public Float evaluateSortingNetwork(){

    List<List<String>> inputsPossibility = this.inputGenerator.generateAllCasesWhenArrayIsNotSorted();
    int numberOfListThatAreSorted = 0;

    for(List<String> input : inputsPossibility){

        List<Integer> inputAux = new ArrayList<>();
        for(String i : input){
            inputAux.add(Integer.valueOf(i));
        }

        if(SortingValidator.arrayIsSorted(sortArrayWithComparators(inputAux))){
            numberOfListThatAreSorted += 1;
        }
    }

    return (100f*numberOfListThatAreSorted/inputsPossibility.size());
}
```

Figura 11. Metoda de Fitness

După cum putem vedea, fitness-ul este calculat în funcție de cât de multe siruri generate a reusit să sorteze comparatori unei relete din mulțimea de rețele din algoritmul nostru genetic. Aceste input-uri sunt defapt toate combinațiile în care un anumit șir de numere poate fi ordonat. Acest lucru se poate face ușor în funcție de numărul de input-uri pe care îl facem. De exemplu dacă am avea un număr de 4 input-uri atunci vom aveam primul șir de numere [0, 0, 0, 1], [0, 0, 0, 1] … [1, 1, 1, 0]. Acest lucru ne asigură că odată ce o multime de comparatori a reusit să sorteze toate aceste siruri de numere, înseamnă că acesta poate ordona orice sir de caractere de lungime 4.

V.2. Populația inițială

La inițierea populației este important să avem grijă dacă aceea populație este corectă deoarece aceasta va influența tot algoritmul genetic. Se stie că dacă populația cu care incepe algoritmul genetic sa lucreze nu este bună atunci algorimul nostru s-ar putea să nu gasească cele mai bune soluții sau ar putea să gasesească aceste soluții dar intr-un timp mult mai îndelungat. O populație de start bună este o populație care are indivizii cât mai diversificată pentru că astfel se pot găsi rezolvări la problemă mult mai repede.

La inițializarea populatiei se va alege care este dimensiunea populației, numărul de input-uri care se dau și numarul de comparatori pe care dorim sa ii folosim pentru a ne ordona un anumit sir de numere de dimensiunea aleasă. În funcție de acestea se vor face urmatoarele.

Prima dată, cu ajutorul numărului de input-uri și cu ajutorul numărului de comparatori se va crea cate un individ cu o nouă rețea. Această rețea se va genera automat cu anumiți comparatori aleși parțial random. Acești comparatori sunt aleși astfel incât să nu existe același comparator în aceași rețea.

```
public GeneticAlgorithm(int sizeOfPopulation, int numberOfInputs, int numberOfComparators) {
    this.sizeOfPopulation = sizeOfPopulation;
    this.numberOfInputs = numberOfInputs;
    this.numberOfComparators = numberOfComparators;

    this.individualGenerator = new IndividualGenerator(this.numberOfInputs, this.numberOfComparators);
    this.individuals = this.individualGenerator.generateNIndividuals(this.sizeOfPopulation);

    this.MUTATION_PROBABILITY = 1f/this.numberOfComparators;
    this.CROSSOVER_PROBABILITY = 0.75f;
}
```

Figura 12. Metoda de Inițiere a populației

In inițializarea algoritmului genetic se va seta și care este probabilitatea de mutație și care este probabilitatea de crossover. După această inițializare vom avea un număr de n indivizi fiecare cu propria lui rețea cu proprii ei comparatori.

V.3. Selecția

In prima parte a selecție se vor strânge intr-o listă toate fitnes-urile pe care le au toti indivizi nostri. Selecția pe care o folsim este Roulette. Prin această metodă de selecție se înțelege alegerea invizilor în funcție de fitness. Cu căt un individ are fitness-ul mai mare cu atât acesta are șansă mai mare sa treacă în urmatoarea generație.

După ce sa creat aceste șanse pentru fiecare individ vom alege un numar random intre 0 și 1 iar dacă acesta se află intre cele doua atunci el va fi selectat pentru urmatoarea generație. Această metodă de selecție este una destul de bună, exceptând cazul în care indivizi au o diferentă foarte mare de fitness. Din această cauza dacă există 2 indivizi care au un fitness foarte mare, atunci aceștia ii vor domina pe ceilalți indivizi nepermițând ca indivizi cu mai putine șanse sa evolueze și să ajungă la un rezultat poate mai bun decat indivizi care sunt dominanți. Dupa ce au fost strânși numărul de indivizi din generația trecută – 2, ceilalți vor fi defapt acei indivizi din generatia trecută care au reusit sa aibe cel mai mare fitness dintre toți deoarece aceștia au ce mai bună probabilitate de a crea alți cromozomi mai buni.

V.3. Încrucișarea

Această este o operație care ne ajută la a combina informații genetică a 2 cromozomi în scopul de a genera alti cromozomi.

```
List<Integer> parents = Helper.getPairNotEqualNumbers(this.sizeOfPopulation);
List<Comparator> newListOfComparators = new ArrayList<>();

for (int j = 0; j < this.numberOfComparators; j++) {
    Float crossOverRandomChoice = (float) Math.random();

    if (crossOverRandomChoice < this.CROSSOVER_PROBABILITY) {
        if (comparatorAux.Equals(this.individuals.get(parents.get(0)).getSortingNetwork().getComparators().get(j))) {
            newListOfComparators.add(this.individuals.get(parents.get(1)).getSortingNetwork().getComparators().get(j));
            comparatorAux.setComparator(this.individuals.get(parents.get(1)).getSortingNetwork().getComparators().get(j));
        } else {
            newListOfComparators.add(this.individuals.get(parents.get(0)).getSortingNetwork().getComparators().get(j));
            comparatorAux.setComparator(this.individuals.get(parents.get(0)).getSortingNetwork().getComparators().get(j));
        }
    } else {
        if (comparatorAux.Equals(this.individuals.get(parents.get(1)).getSortingNetwork().getComparators().get(j))) {
            newListOfComparators.add(this.individuals.get(parents.get(0)).getSortingNetwork().getComparators().get(j));
            comparatorAux.setComparator(this.individuals.get(parents.get(0)).getSortingNetwork().getComparators().get(j));
        } else {
            newListOfComparators.add(this.individuals.get(parents.get(1)).getSortingNetwork().getComparators().get(j));
            comparatorAux.setComparator(this.individuals.get(parents.get(1)).getSortingNetwork().getComparators().get(j));
        }
```

Figura 13. Metoda de Crossover

În această imagine putem vedea cum am reușit să cream crossover-ul pentru algoritmul nostru. Im primul rând pentru a putea face crossover eficient, este nevoie sa avem o anumită probabilitate pentru care sa facem acest crossover deoarece daca am face pentru toti indivizi, populația ar putea sa piardă cromozomi care sunt foarte buni iar atunci nu am mai reușit să ajungem la un rezultat cat mai bun. În primul rand verificăm dacă un numar ales random este mai mic decat probabilitatea pe care am setato noi pentru crossover. Noi avand pentru fiecare individ un parent care este defapt o multime de 2 numere , X și Y care sunt diferite intre ele și au dimensiunea maximă egală cu dimensiunea populației. După care avem un switch în care verificăm dacă comparatorul nostru auxiliar care este setat prima data cu X = -1 și Y = -1 , în vederea generări unui nou comparator. Dacă comparatorul nostru auxiliar este egal cu parintele cu indexul egal cu primul numar ales random de noi atunci în noua listă de comparatori se va pune comparatorul de la părintele cu indexul Y și comparatorul j. În caz contrar atunci se va pune comparatorul j de la părintele cu indexul X. Acest lucru este invers pentru cazul în care random-ul ales este mai mic decat probabilitatea de crossover. Prin acest lucru putem vedea că parcurgând comparatori unui individ, putem extrage anumiși comparatori din acesta pentru a crea o noua listă de comparatori pe care o putem atribui unui individ nou.

```
Collections.sort(this.individuals, (o1, o2) -> Float.compare(o2.getFitness(), o1.getFitness()));

comparatorsGenerated.add(this.individuals.get(0).getSortingNetwork().getComparators());
comparatorsGenerated.add(this.individuals.get(1).getSortingNetwork().getComparators());
for (int i = 0; i < this.individuals.size() - 2; i++) {
    this.individuals.get(i).getSortingNetwork().setComparators(comparatorsGenerated.get(i));
}
```

Figura 14. Metoda de Crossover Indivizi

După ce am facut acest lucru vom sorta toti indivizi în functie de fitness-ul pe care il au și dupa care vom selecta pentru generația urmatoare comparatori primilor doi indivizi care au fitness-ul cel mai mare, dupa care vom parcurge toti ceilalți indivizi pentru a le atribui un nou set de comparatori care au fost creați dupa crossover.

V.4. Mutația

Mutațiile sunt o operație geneticș folositș pentru a diversifica cromozomii dintr-o populație. Această se referă la schimbarea anumitor gene dintr-un cromozom (individ) pentru ca acesta sa ajungă să aibe un fitness mai bun în sperața unei găsiri mai rapide a soluției. Mutația ne mai ajută și să pătrăm o anumită diversitate intre indivizi.

```
private void mutateIndividuals() {

    for (int i = 0; i < this.sizeOfPopulation; i++) {
        for (int j = 0; j < this.numberOfComparators; j++) {
            Float mutationRandomChoice = (float) Math.random();

            if (mutationRandomChoice < this.MUTATION_PROBABILITY) {
                do {
                    Comparator comparator = new Comparator();
                    comparator.setX(new Random().nextInt( bound: this.numberOfInputs - 1));
                    int aux = new Random().nextInt( bound: this.numberOfInputs - 1) + (comparator.getX() + 1);
                    while (aux >= this.numberOfInputs) {
                        aux = new Random().nextInt( bound: this.numberOfInputs - 1) + (comparator.getX() + 1);
                    }
                    comparator.setY(aux);
                    this.individuals.get(i).getSortingNetwork().getComparators().get(j).setComparator(comparator);
                } while (!Helper.checkAllComparatorsAreDifferent(this.individuals.get(i).getSortingNetwork().getComparators(), j));
            }
        }
    }

}
```

Figura 15. Metoda de Mutație

După cum putem vedea în această imaginea, pentru a face mutație este nevoie sa parcurgem toți indivizi din populatie. Pentru fiecare individ va trebui să îi parcurgem toți cromozomi acestuia ca sa vedem daca cu o anumită probabilitate pe care o putem seta la inceputul inceperi aplicatiei, gena respectivă se va schimba. Daca una dintre aceastea a ajuns în această fază atunci se va crea un comparator nou random, în funcție de numărul de fire pe care aceasta retea le are. Dupa ce s-a creat un nou comparator astfel incât să satisfaca aceste cerinte, atunci acest nou comparator va fi inlocuitorul comparatorului care a ajuns să fie schimbat. Comparatorul acesta este defapt o gena a individului. Cea mai importantă verificare care trebuie facută dupa ce am facut aceasta mutatie este să vedem daca toti comparatorii sunt diferiti pentru ca altfel vom avea inconsistentă intre comparatori acelui individ. Daca din pacate nu toți comparatori sunt diferiți, se va relua tot procesul de schimbarea genei pana cand toți comparatorii acelui individ sunt diferiți. Acest tip de mutatie se cheama 'Random Resetting' deoarece nu ne vom folosi de gena pe care dorim să o schimbăm în scopul creari unei noi ci vom crea o genă nouă cu valori random și atribuită genei respective.

Capitolul VI.

INTERFAȚĂ REȚEA SORTARE

În acest capitol vom vorbi despre vizualizarea algoritmului și a indivizilor din reteaua de sortare. Acest capitol este de a transpune implementarea de viziualizarea specifica a anumitor principi care au fost aplicati asupra indivizilor.

VI.1. Reprezentarea unui individ

După cum am putut observa în imaginile anterioare, observăm ca un individ are în primul rând un număr de fire prin care numerele din sir trec astfel încât la finalul parcurgeri acelor fire, șirul de numere să fie sortat. Un alt lucru foarte important este lista de comparatori care sunt legați de un capăt al unui fir și de un alt capăt al altui fir, asta însemnând că atunci când doua numere se întâlnesc, se va face o comparare între cele doua numere pentru ca astfel dacă primul număr este mai mic decât celălalt, se va face schimbul de fire între cele doua (dacă primul număr este mai mare decât celălalt, atunci primul număr va trece pe firul celui de al doilea număr iar al doilea număr va trece pe firul primului).

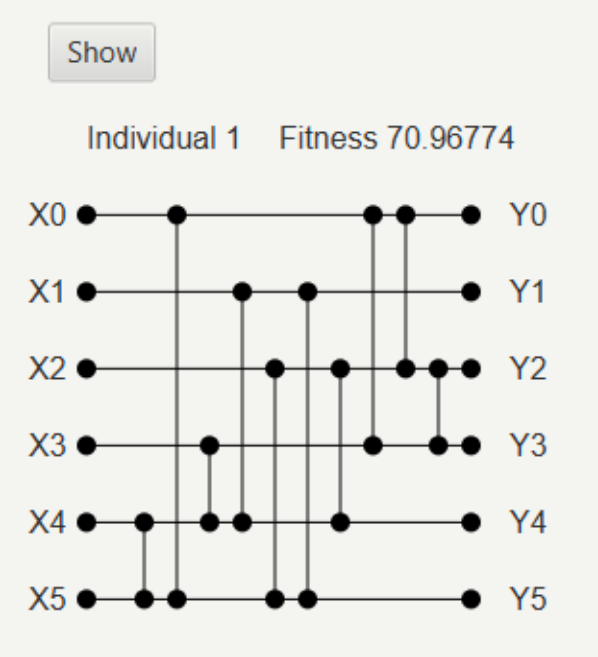

Figura 16. Reprezentarea individului

În figura 16 observam 6 fire din care putem rezulta ca există oricare 6 input-uri, și un număr comparatori. Fiecare fir are un punct la începutul lui și unul la final. Acest lucru reprezintă ca numărul va trece de la primul punct până la ultimul, parcurgând toți comparatorii care îi iese în cale. Numărul de input-uri și numărul de comparatorii este setat la începutul rulări programului. în aceasta figură putem observa că de fapt aceasta este rețeaua respectivului individ. Toate aceste numere vor trece prin acești comparatorii în speranța că aceste numere ajungând la finalul firului vor fi poziționate în funcție de valoarea pe care acestea o au. În imagine mai putem vedea care este numărul individului. Acesta reprezintă pe ce poziție se afla în populație iar acest lucru mai reprezintă și cat de valoros este el pentru populație deoarece toți indivizi sunt ordonați în funcție de fitness.

Un alt lucru pe care îl putem observa în imagine este fitness-ul pe care individul îl are, acesta reprezentând cat de bine se descurcă în a sorta un sir de numere de lungime 6 în cazul nostru. Reprezentarea acestor indivizi are o implementare destul de generica în speranța ca acesta poate sa reprezinte orice individ indiferent de numărul lui de comparatori sau numărul lui de fire.

Un ultimul lucru pe care îl putem observa în afișarea unui individ este un buton cu numele ’Show’. Acesta va porni o noua interfață grafică în care vom putea observa detalii importante asupra individului nostru.

VI.2. Reprezentarea corectitudini unui individ

După cum am putut observa în figura 16, fiecare individ are un buton cu care putem deschide o nouă interfată grafică, pentru a putea sa vedem care este corectitudinea retelei de sortare a acelui individ.

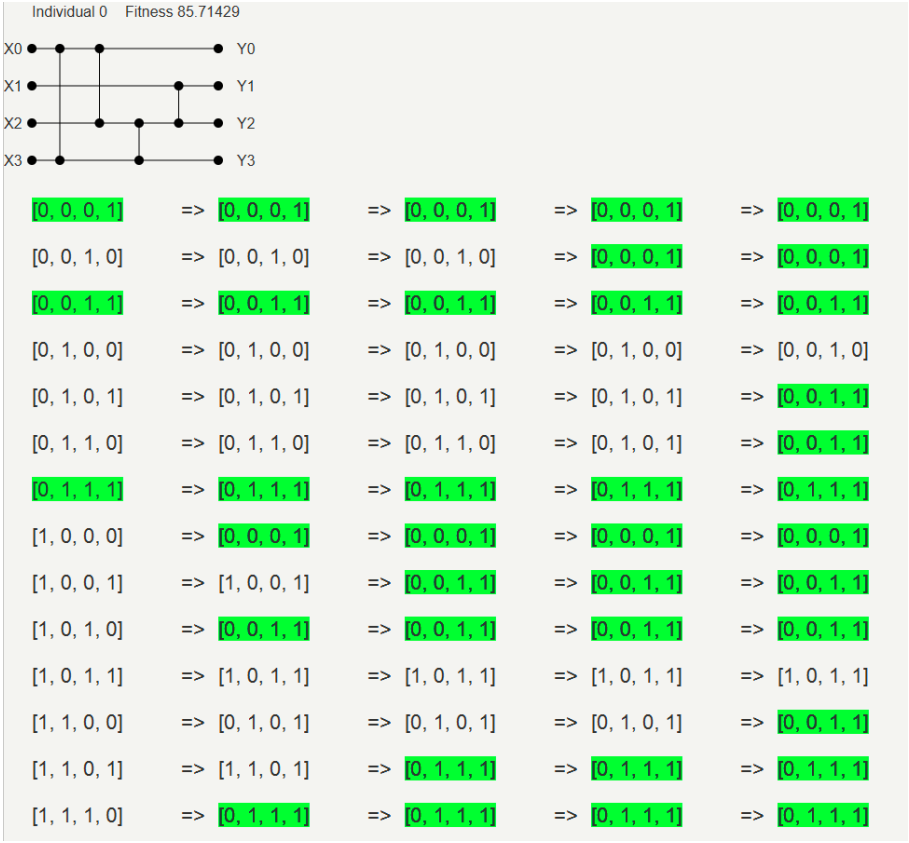

Figura 17. Corectitudinea individului

În această interfață grafică putem observa (individul căruia dorim sa îi vedem corectitudinea. Acele informații sunt cele pe care le vom vedea și în interfata cu generatiile. După cum putem observa, în această interfață sunt generate șiruri de numere de dimesiune n, care este de fapt numărul de elemente care a fost ales la începutul rulări aplicatiei. Acestea sunt generate astfel incât ele sa fie diferite și sa cumprindă toate combinariile prin care un șir de n numere poate să fie aranjat. Avem nevoie de acest lucru deoarece ne dorim

de la individul respectiv sa poate sa sorteze un șir de n numere indiferent care sunt pozitiile elementelor în șir.

Pe prima linie se află primele siruri de numere care incep sa pornească pe firele retelei. Următoarele lini sunt aceleasi siruri de numere dar după ce s-a aplicat comparatorul respectiv. Din asta putem rezulta că pe linia 0 se află sirurile de numere care au fost generate dar incă nu au trecut prin niciun comparator, pe linia 1 se vor afla aceleași siruri dar cu schimbariile pe care le-a adus primul comparator în sirul respectiv. Acest lucru este valabil și pentru urmatoarele lini. Acest lucru ne arată ca pe linia n se va afla sirul de numere dupa ce acesta a trecut de n – 1 comparatorii.

O alta remarca pe care o putem trage este ca atunci cand un sir de numere ajunge sa fie crescator atunci acesta este trasat cu o culoare verde.

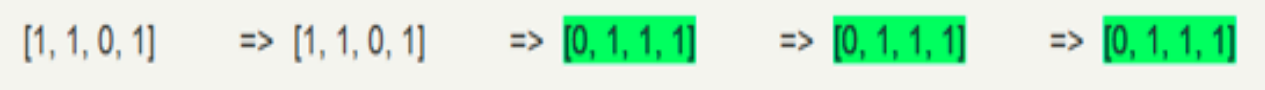

Figura 18. Ciclul unui sir de numere

Acest lucru este folositor pentru a ne atrage atentia asupra acelui șir că acesta a reusit să fie sortat după ce a trecut de un anumit numar de comparatorii.

Toate aceste reprezentări sunt de folos pentru a putea sa afisăm o interfată grafică în care sa putem observa indivizii dintr-o anumită generație în scopul de a putea să vedem daca aceștia au ajuns să aibă o performantă mai buna în sortarea anumitor șiruri de numere. În figura 18 se vor observa avantajele aduse de decuplarea generări grafice a unui singur individ.

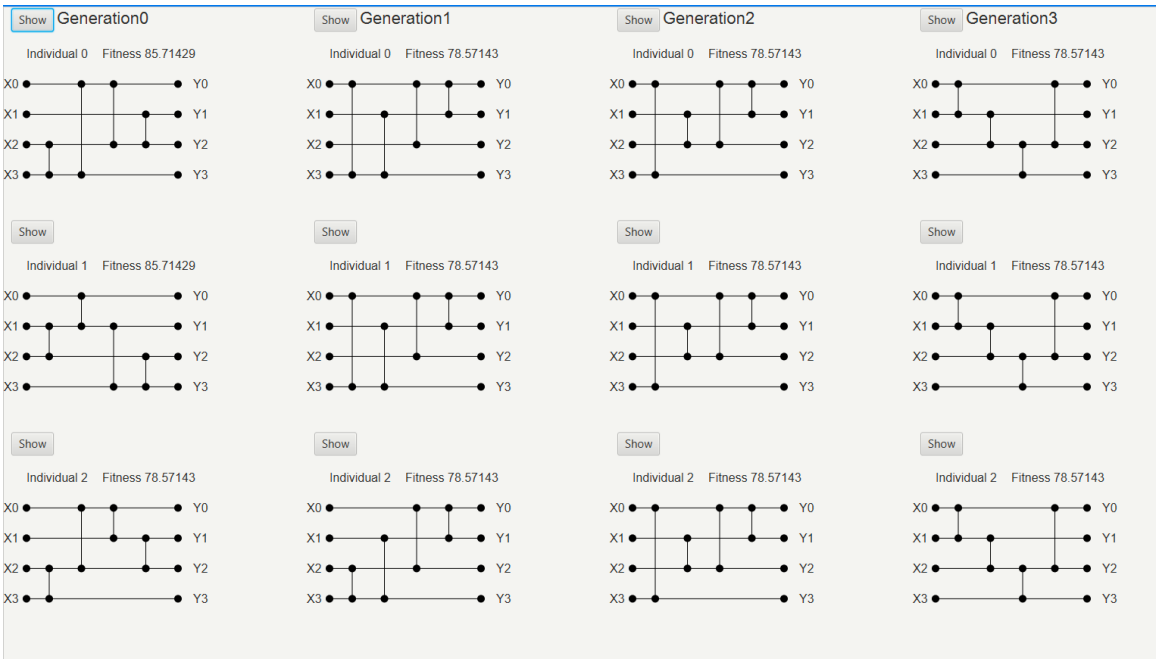

Figura 19. Vizualizarea Generatiilor

In figura 19 observăm că pentru fiecare generație avem afișati 3 indivizi. Am ales acest lucru deoarece aplicatia noastră este folosită pentru găsirea comparatorilor pentru rezolvarea sortări acestora și nu pentru vizualizarea acestora. Spunem aceste lucruri deoarece generarea reprezentări unui individ este destul de costisitoare ca și timp și astfel nu dorim sa prelungim timpul de executie pe care acest program îl face.

Pentru fiecare generație incepând cu generatia 0, vo mavea 3 indivizi iar acești indivizi sunt aceia care au cel mai mare fitness din toata populatia generației. Odata ce a fost afisate primele 3 rezultate cele mai bune ai unei generați, va incepe calculul și reprezentarea urmatoarei generati.

CONCLUZII

Pâna acum am discutat despre rețeaua de sortare, algoritmi genetici și cromozomii acestuia dar nu am tras nicio concluzie clara despre folosința lor. Ca să putem sa discutam despre acest subiect este nevoie sa facem anumite comparări cu aceasta metoda de a sorta o listă de numere.

Sorting Algorithm	Best Case	Average Case	Worst Case	Stability	In-Place
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	Yes	Yes
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	No	Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	Yes	No
Quick Sort	O(nlogn)	O(nlogn)	$O(n^2)$	No	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Shell Sort	$O(n(\log n)^2)$	$O(n^{3/2})$	O(n)	No	Yes

Figura 20. Complexitatea algoritmilor de sortare

(https://www.semanticscholar.org/paper/Fuse-sort-algorithm-a-proposal-of-divide-%26-conquer-Patel-Singh/4c6d6bea88635220083d3f6aa5b06c287ec832f4)

Dacă privim figura 20 putem observa care sunt complexitațiile algoritmilor de sortare pe care majoritatea îi cunoastem deja. Dacă luăm de exemplu primul algoritm de sortare, BubbleSort este o sortare care are ca și complexitate $O(n^2)$ în cazul cel mai complex, asta înseamnă ca pentru a reuși sa sortăm o lista de n numere, o să fie nevoie de n^2 comparatorii, ceea ce este destul de mult. În schimb daca privim complexitățiile algoritmului de sortare QuickSort, acesta are complexitatea $O(\mathrm{n}\log_2 n)$ ceea ce este cu mult mai bine fata de BubbleSort deoarce când vorbim de retele de sortare ne gândim ca aceasta o să poată sa sorteze liste de numere de dimensiuni mari.

Si acum ajungând la retelele de sortare, acestea înainte de a ști care este numărul de input-uri pe care dorești sa le sortezi, pentru a reuși asta este nevoie de un timp mult mai îndelungat ca sa poți gasi care sunt comparatorii pe care dorim să îi aplicam pe lista de numere, în schimb odată ce am reușit să gasim acei comparatorii, o să putem sa aplicăm aceași comparatorii pe toate listele de numere care au aceași lungime. Acest lucru ne permite să avem o performantă a timpului mult mai mare fată de alți algoritmi de sortare.

Un alt lucru foarte important despre aceste retele de sortare este că ele pot fi folosite cu ajutorul paralelismului, iar acest lucru ne poate oferi o complexitate mult mai mică față de alt algoritmii de sortare care au fost prezentați mai sus.

BIBLIOGRAFIE

Valsalam, V.K., Miikkulainen, R., (2013), *Using Symmetry and Evolutionary Search to Minimize Sorting Networks*, *Journal of Machine Learning Research*, Ed. Vinod K. Valsalam and Risto Miikkulainen, Vol.14, Pag 303-331.

https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4

https://www.researchgate.net/publication/220862320_Initial_Population_for_Genetic_Algorithms_A_Metric_Approach

http://www.technical-recipes.com/2015/a-genetic-algorithm-for-optimizing-sorting-networks-in-c/

https://pdfs.semanticscholar.org/bd3b/6863e6587b7e6aab8efb44662bb5cd793cf7.pdf

http://www.cs.bham.ac.uk/~wbl/biblio/gecco2002/GA249.pdf

https://en.wikipedia.org/wiki/Sorting_network

https://github.com/NickVenturini/sorting-network?fbclid=IwAR2wBocHHxeCvNTIEYzywZQzOsQ5nWGGj8Wyfz-Boo64n7K556fH9s9wc7k

https://adventuresinevolutionblog.wordpress.com/2016/09/17/minimal-sorting-networks/?fbclid=IwAR2HzkUKjhVj4b6eM7Ld5S0vuj2Cd4HGSLbcYJJw2ddcfnTeYpchs7DLZQ0

https://www.technical-recipes.com/2015/a-genetic-algorithm-for-optimizing-sorting-networks-in-c/?fbclid=IwAR3W-6TRvsfURPzDR-5BKQebD-lRDhd6Dh2I7odcVuZXo0n99qTId75pVVc

https://linuxnasm.be/media/pdf/donald-knuth/taocp/volume-3/taocp-vol3-sorting-searching.pdf

http://www.human-competitive.org/sites/default/files/genp05.pdf

https://books.google.ro/books?id=0eznlz0TF-IC&pg=PA22&lpg=PA22&dq=sorting+network+genetic+algorithm&source=bl&ots=shoB4X-4Ji&sig=ACfU3U0afHBwZTt2yohUMoPJkYBekCkZ-w&hl=ro&sa=X&ved=2ahUKEwiRmPudmIDjAhVnQkEAHfjJAZY4FBDoATABegQICRAB#v=onepage&q=sorting%20network%20genetic%20algorithm&f=false

http://www.cse.unt.edu/~tarau/teaching/PP/Sorting%20network.pdf