UNIVERSITATEA „ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

**LUCRARE DE LICENȚĂ**

Coordonator științific:  
Lect.dr. Frăsinaru Cristian

Candidat:

Slusariuc Dan Dragoș Nicolae

IAȘI

2019

UNIVERSITATEA „ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

**METODE DE CREARE A RETELELOR DE SORTARE**

Coordonator științific:

Lect.dr. Frăsinaru Cristian

Candidat:

Slusariuc Dan Dragoș Nicolae

IAȘI

2019

**CUPRINS**

[INTRODUCERE 1](#_Toc12222827)

Capitolul I. [EXEMPLU 3](#_Toc12222829)

Capitolul II. [ZERO-ONE PRINCIPLE 4](#_Toc12222831)

Capitolul III. [DNF ȘI CNF 7](#_Toc12222833)

[III.1. Sorting Network Symmetries 8](#_Toc12222834)

[III.2. Minimizing Comparator Requirement 9](#_Toc12222835)

Capitolul IV. [ARHITECTURĂ 13](#_Toc12222837)

[IV.1. Algoritm 13](#_Toc12222838)

[IV.2. Grafică 14](#_Toc12222839)

[IV.3. Rețea 15](#_Toc12222840)

[IV.4 Infrastructură 16](#_Toc12222841)

Capitolul V. [ALGORITM GENETIC 17](#_Toc12222843)

[V.1. Fitness 17](#_Toc12222844)

[V.2. Initial Population 19](#_Toc12222845)

[V.3. Selecția 20](#_Toc12222846)

[V.3. Crossover 21](#_Toc12222847)

[V.4. Mutație 22](#_Toc12222848)

Capitolul VI. [INTERFAȚĂ REȚEA SORTARE 24](#_Toc12222850)

[VI.1. Reprezentarea unui individ 24](#_Toc12222851)

[VI.2. Reprezentarea corectitudinii unui individ 26](#_Toc12222852)

[CONCLUZII 29](#_Toc12222853)

[BIBLIOGRAFIE 31](#_Toc12222854)

**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:

* + Multiprocessor computers
  + Switching networks

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.

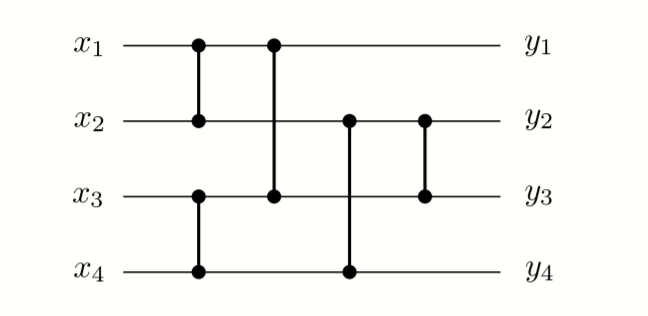
O metoda de a rezolva această problemă este aceea că descompune problema în subprobleme pentru a minimiza numărul de comparatori.

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 rezultând faptul că, având ca input { , , }, acesta să returneze { , , }, unde .

Acesta a început să fie un subiect de research din anul 1950. Prima rețea de sortare a fost realizată de O’Connor și Nelson, acesta având posibilitatea să sorteze un șir de numere de lungimea n, unde . Aceasta avea numărul minim de comparatori pentru , dar care necesită alți 2 comparatori pentru . Această rețea de sortare a fost îmbunătățire de Batchar în 1968 care a reușit să producă minimul de comparatori pentru

# *Capitolul I.*

# **EXEMPLU**



**Figura 1.** Individ

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

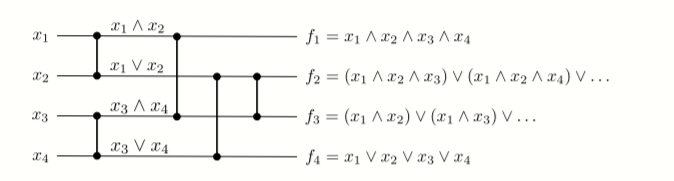
Așa cum putem observa în figura 1, având 4 input-uri , pe care se vor aplica o serie de comparatori, vom ajunge la un output , în care . Acesta este un exemplu destul de simplu, deoarece ca input se primit doar 4 valori, iar problema noastră este atunci când primim un input cu mult mai mare ca acesta.

# *Capitolul II.*

# **ZERO-ONE PRINCIPLE**

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

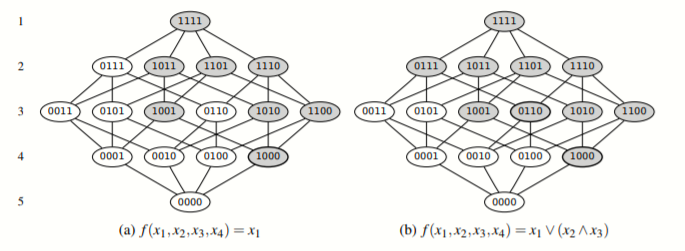
Aceasta modalitate crează mai multe funcții care sunt de fapt compoziții de conjuncții și de disjuncții care returnează o valoare booleană. De exemplu, atunci când avem mulțimea , funcției îi revine o combinație de disjuncții de conjuncții care conțin *n + 1 – i* variabile. Prin urmare, rețeaua de sortare va deveni o secvență de comparații care returnează toate funcțiile generate prin această metodă. Acest lucru ne ajută să creem 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 multțime de funcții boolene pentru 4-input sorting network. Principul zero-one este de folos în a reprezenta input-urile rețelei ca variabile boolene. Fiecare comparator produce conjuncția input-urilor pe linia superioarț și disjuncția input-urilor pe linia inferioară funcții booleene monotone. Din cauza că acestea sunt funcții booleene monotone, atunci putem vedea că ele au proprietatea ca pentru oricare n tuple unde **a =** și **b =** **,** astfel încât, **a** ≺ **b** dacă pentru . Un set format din toate tuple binare ordonate de ≺ se cheamă latice booleana. 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).



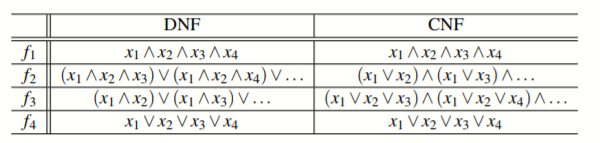
**Figura 3.** Diagrama Hasse (Valsalam și Miikkulainen, 2013)

Fiecare pereche de noduri sunt comparabile și conectate dacă pot fi ordonate de ≺. Un subset **X** de noduri sunt legate superior de nodul **y,** dacă **x** ≺ **y** pentru toate x ∈ X. La fel este și pentru legarea inferioară. Dupa cum putem vedea un nod are cale către un nod inferior , dacă pentru . Prin urmare, dacă 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 antichain-ul legăturilor nodurilor în laticea booleană. Într-o latice de dimensiunea , dimensiunea maximă a reprezentării este egală cu dimensiunea celui mai mare antichain, care este egală cu = . 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 , iar valoriile sunt grupate în funcție de nivelul la care se află în laticea booleană, astfel încât valoriile de pe orice nivel să poata 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 , care este de fapt cazul când toate output-urile funcțiilor sunt construite corect (Valsalam și Miikkulainen, 2013).

# *Capitolul III.*

# **DNF SI CNF**



**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 (DNF) și în formă normală (CNF), aceasta va fi un mod bun de a vizualiza simetria funcțiilor returnate. Dupa cum putem vedea, schimbând conjuncțiile și disjuncțiile în forma (DNF) a funcțiilor , 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 ramâne la fel ca în tabela originală. Aceasta funcționează pentru orice pereche de (). Operația care ține rezultatul funcțiilor rețelei se numește “symmetry”. Acestea pot fi folosite pentru minimiza numărul de comparatori ale rețelei (Valsalam și Miikkulainen, 2013).

## **III.1. Sorting Network Symmetries**

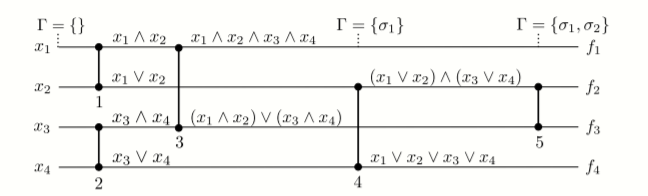
Aceasta este o operație care se aplica funcțiilor de output ale rețelei și care lasă funcțiile invariante, astfel încât rețeaua rezultată să ramână neschimbată. Comform Valsalam și Miikkulainen, (2013), rezultatul inversării rețelei de sortare poate fi reprezentată ca = , pentru output-ul funcției poate fi obtinut de la schimbând conjuncțiile și disjuncțiile și vice versa. Dacă schimbăm cele 2, iar funcțiile și sunt interschimbate, atunci rezultatul rețelei ramâne același (Valsalam și Miikkulainen, 2013).

Astfel, putem să definim operația simetriei pentru , care acționează asupra funcțiilor unei rețele schimbând funcția și dualul ei , și schimbându-le conjuncțiile și disjuncțiile. Compoziția operațiilor simetrice sunt și ele simetrice deoarece și operează independent pe diferite perechi de funcții din output. Prin urmare, acestea sunt și asociative. Prin urmare, acesta satisface toate axiomele unui grup pentru a putea fi reprezentată matematic. Din cauza că fiecare element din grup poate fi exprimat ca o compoziție de o mulțime de elemente , grupul arată că este generat de (Valsalam și Miikkulainen, 2013).

Subgrupurile lui , subseturi 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 operează peste perechi de funcții disjuncte, atunci acel subgrup poate fi scris ca , unde este un subset a lui . Inițial, înainte ca de a avea macăr un comparator, fiecare linie *i* în rețea are o funcție booleana . Ca rezultat vedem ca = {}. Adăugarea de comparatori care construiesc outputul funcției și dualul ei duce la = {}, care este o rețea parțială. Daca vom adauga mai mulți comparatori care construiesc și dualul ei , atunci vom creea altă rețea parțială cu = {}, iar această rețea parțială este mai simetrică. Continuând să adaugăm comparatori până când toate funcțiile de output care au fost construite produc o rețea completă de sortare cu = . 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 căștiguri 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   = {}, 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 si au cele mai puține combinații de variabile, așadar, ele pot fi construite adăugând comparatori. Astfel, primul câștig parțial este producerea simetriei = {}, folosind cât mai puțin comparatorii. După ce am construit și , următoare pereche de funcții de output cu cele mai puține combinații de variabile, ar fi și . Astfel, următorul câștig parțial ar fi să le construim și să producem simetria = {}. 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 adauga simetriei la , următorul câștig parțial ar fi să adăugăm la . 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. Minimizing Comparator Requirement**

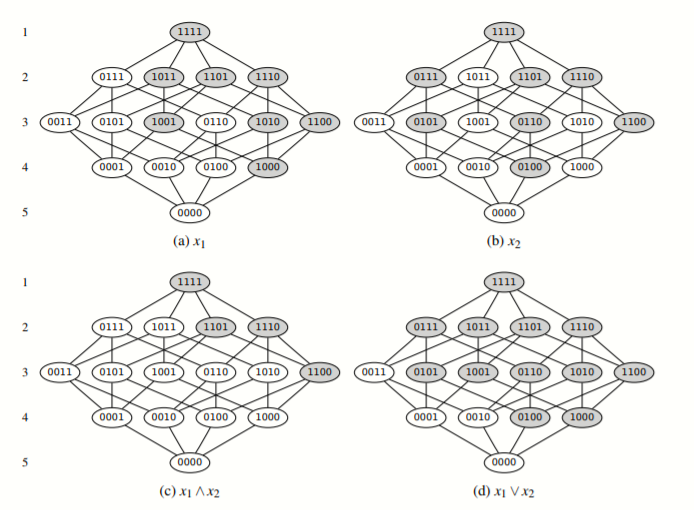


**Figura 5.** Metoda de reducere a comparatorilor (Valsalam și Miikkulainen, 2013)

Conform Valsalam și Miikkulainen, (2013), scopul final al metodei de reducere al comparatorilor este de a construi simetria = {} prin executarea celor 4 funcții prin folosirea numărului minim de comparatori. Acest lucru poate fi descompus într-o secvență de câștiguri parțiale considerându-le subgrupuri a grupului simetric . La fiecare pas al construcției, următorul câștig parțial este de fapt, subgrupul care poate fi produs adaugănd cel mai puțin un număr de comparatori. Inițial, în rețea nu avem nicio simetrie, astfel încât = {}. Dupa cum putem observa în figura 5 și 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 și pentru a ajunge la primul câștig parțial cu minimul număr de comparatori. Prin urmare, am ajuns la realizarea primei simetrii = {}. Acum, pentru a ajunge la simetria = {}, ne vom folosi de și . Adăugând comparatorul 5 ajungem la terminarea producerii câștigului parțial, având în vedere că comparatorii 3 și 4 au creat și 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 folosita sa determine daca un comparator poate fi folosit in crearea a 2 functii simultan.Sa spunem ca castigul partial este construit pe baza functilor de output si atunci functiile de output mai mici ca *i* si mai mari decat *n + 1 – i* sunt deja construite, insemnand ca fiecare functi are valoare 1 la toate nodurile in nivelele mai mici sau égale decat *j* si valoare 0 pentru toate celelalte. Putem vedea ca aceste functii 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.* Daca asta nu ar fi adevarat atunci ar fi imposibil de construit alte functii de output adaugand alti comparatori deoarece conjunctiile tin zero-uri iar disjunctiile unu-uri ale functiilor intermediare care le combina (Valsalam și Miikkulainen, 2013).

Câstigul parțial curent prin construirea funcției 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 cauza că funcțiile intermediare la liniile au deja valoare 1 la toate nodurile cu nivelul mai mic sau egal decat *i,* construind acesta va reține valoare acestor noduri automat. Atunci 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).

Impărțirea de comparatori pentru a construi funcțiile duale de output intr-o rețea de sortare cu 4 input-uri. Comform Vinod K. Valsalam 2013 in această figura putem observa reprezentarea unei latice booleene a funcțiilor care sunt construite de comparatorul 1 in Figura 5. Nivelele laticei sunt in stânga iar nodurile in care funcția ia valoare 1 sunt umbrite. Comparatorul 1 va construi conjuncția și disjuncția funcțiilor (a) și (b) pentru câstigul parțial in construirea funcției de output = . Funcția poate fi construită folosind conjuncții pentru a seta valoare la toate nodurile de pe nivelul 2 a laticei cu 0. La fel și poate fi construit folosind disjuncții pentru a seta valoare la toate nodurile a nivelului 4 cu 1. Împărțind comparatorii in aceasta maniera reduce numărul de comparatori care sunt nevoie pentru a construi reteaua de sortare (Valsalam și Miikkulainen, 2013).

Valoarea unei funcții a nodului pe nivelul *i + 1* poate fi setată la 0 adaugând un comparator care construieste 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 construieste, are mai putine noduri cu valoare 0 atunci acestea nu sunt folositoare pentru construirea lui Disjuncția va fi totuși folosită pentru a construi celelalte funcții de output implicând ca 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 care au valoarea 0 pentru orice nod de pe nivelul *i + 1* (Valsalam și Miikkulainen, 2013).

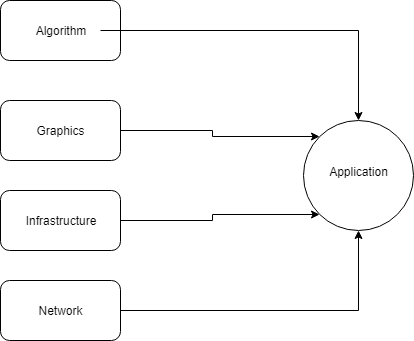
Adaugând un comparator intre perechea de funcții, acesta colectează nodurile cu valoare 0 pentru ambele funcți in conjunctiva lor. Dacă repetăm acest process recursiv, acesta va colecta nodurile cu valoriile 0 de pe nivelul *i + 1* pentru toate funcțiile până la linia *i,* acesta producând Acest lucru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

Frunzele care sunt in rezultat binar al recursivitați arborelui pentru sunt defapt funcțiile care au noduri cu valori 0 pe nivelul *i + 1* iar pe nodurile interne sunt output-urile comparatorului conjunctiv. Putem vedea ca numarul de noduri de gradul 2 in arborele binary este cu 1 mai mic decât numarul de noduri, atunci numărul de comparatori necesari depinde de numărul de funcții cu care recursivitatea incepe. Maximizând împărțirea de comparatori intre 2 arbori de recursivitate minimizează numărul de comparatori pentru castigul parțial curent. Multi comparatori pot sa aibă aceași importanță iar atunci unul dintre acestea este ales random din set și este adăugat la rețea. Dacă repetem acest proces vom produce o secventă de comparatori care optimizează împărțirea in câștigul parțial curent și castigul parțial mai indepărtat (Valsalam și Miikkulainen, 2013).

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

# *Capitolul IV.*

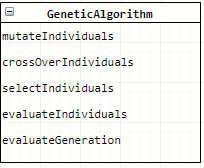
# **ARHITECTURĂ**



**Figura 7.** Structura de baza

## **IV.1. Algoritm**

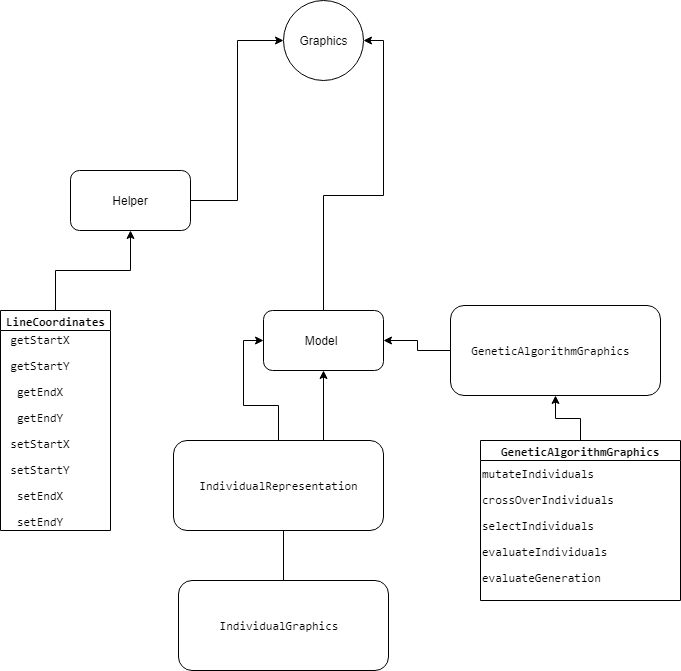
In această parte a proiectului avem partea principala a aplicației. Aceasta se ocupa cu manipularea anumitor indivizi a rețelei pentru a incerca sa găsim o rețea cât mai bună.



**Figura 8.** Metodele algoritmului genetic

In această clasa putem vedea cei 4 pași prin care trec indivizi pentru a fi manipulați astfel incât sâ aducâ sau sâ se imbunatațeascâ comparatorii acestora. Acest pachet este folosit pentru pachetul Graphics’ care se va ocupa cu afisearea acestor indivizi pentru a ințelege și mai bine care este rolul acestei aplicații.

## **IV.2. Grafică**



**Figura 9.** Modulul de Grafică

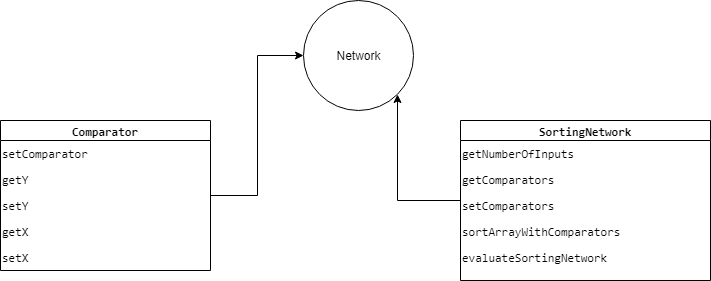
În acest pachet sunt extrași indivizi din fiecare generație, pentru a putea fie grafic afisați pe o interfată in scopul de a vizualiza ce se intamplă cu fiecare individ la fiecare generație. Pentru a ușura modalitatea de afisare a acestor indivizi au fost create 3 mari modele.

Pe primul loc se afla IndividualReprezentation. Acesta se ocupă cu afiseara unui singur individ. Acest lucru se intamplă prin generarea unei noi instante de clasă prin acordarea datelor unui anumit individ, coordonatele unde va incepe afisarea acestuia grafic și care este numarul acestuia in generatia din care face parte. Acest lucru este important pentru a putea face distincția dintre indivizii dintr-o anumită generatie. In această interfată care va fi explicată mai jos se vor vedea ca fiecare generatie este afisată rând pe rând in scopul de a putea face o evaloare vizuală asupra fiecărui individ dintr-o generatie. Cu alte cuvinte acest lucru ne ajută la o vizualizare mai buna asupra imbunătațiri 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 in 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 in 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 in 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 in funcție de comparatorii pe care îi avem in acea retea.

O altă importanță pe care o are această clasa este de a genera comparatorii pentru populația initițială. 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 Infrastructura**

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 in 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 in generațiile viitoare. Pentru a putea vedea cand un individ este mai bun decât alt individ, vom avea nevoie de fitness.

## **V.1. Fitness**

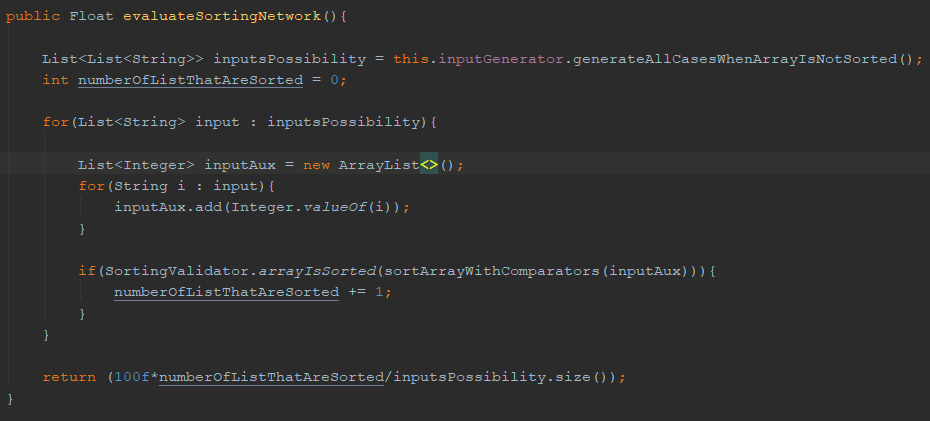
O functie fitness determină cât de potrivit un anumit individ din populatie este pentru a putea să facă parte din generațiile urmatoare. In cazul acestei licențe a fost creată o funcție fitness care returnează cât de corect sortează un anumit sir de numere. De obicei se folosește o funcție fitness deoarece se doreste a ști dacă o anumită soluție este mai bună decât alta și de aceea aceste rezultate trebuie testate și sortate in funcție de cât de bună este o solutie față de cealaltă. In cazul in 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 in plus. Având o functie fitness eficientă reducem foarte mult complexitatea algorimului genetic.

Proprietatea important pe care se bazează o funcție fitness este in a putea sa dăm un anumit grad de performantă asupra unei soluți. In 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.



**Figura 11.** Metoda de Fitness

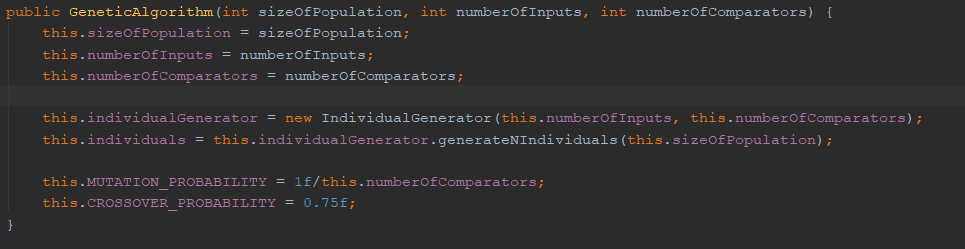
După cum putem vedea, fitness-ul este calculat in funcție de cât de multe siruri generate a reusit să sorteze comparatori unei relete din mulțimea de rețele din algoritmul nostru genetic. Aceste input-uri sunt defapt toate combinațiile in care un anumit șir de numere poate fi ordonat. Acest lucru se poate face ușor in 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. Initial Population**

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. In 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.



**Figura 12.** Metoda de Inițiere a populației

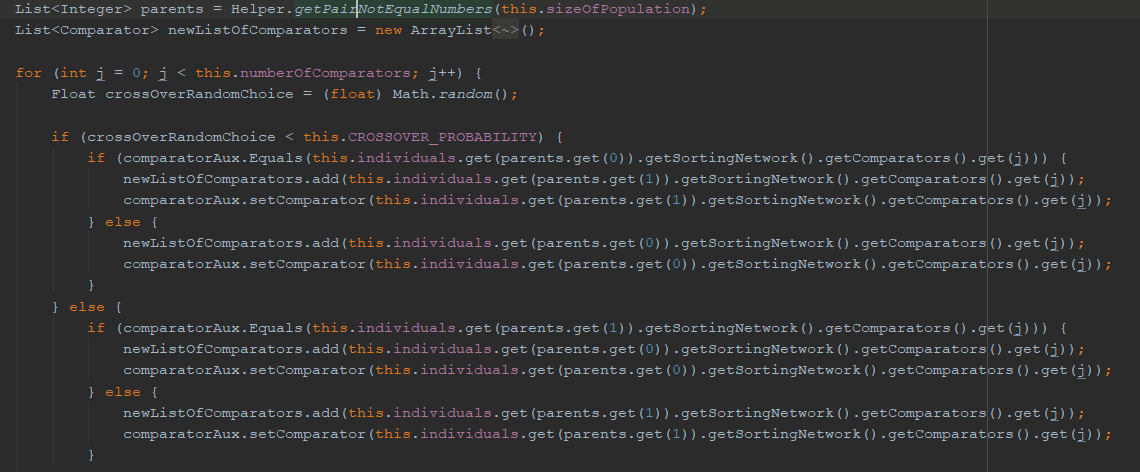
In inițializarea algoritmului genetic se va seta și care este probabilitatea de mutație și care este probabilitatea de crossover. După această inițializare vom avea un număr de n indivizi fiecare cu propria lui rețea cu proprii ei comparatori.

## **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 ințelege alegerea invizilor în funcție de fitness. Cu căt un individ are fitness-ul mai mare cu atât acesta are șansă mai mare sa treacă in urmatoarea generație.

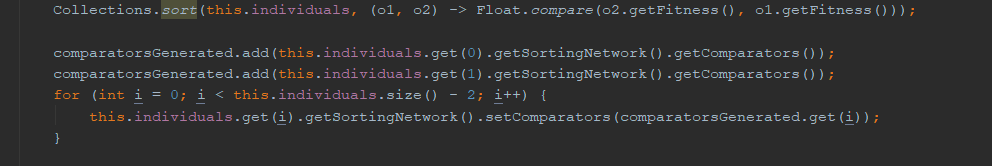
După ce sa creat aceste șanse pentru fiecare individ vom alege un numar random intre 0 și 1 iar dacă acesta se află intre cele doua atunci el va fi selectat pentru urmatoarea generație. Această metodă de selecție este una destul de bună, exceptând cazul in care indivizi au o diferentă foarte mare de 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. Crossover**

Această este o operatie care ne ajuta la a combina informații genetică a 2 cromozomi in scopul de a genera alti cromozomi.

**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 reuși să ajungem la un rezultat cat mai bun. In 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 in care verificăm dacă comparatorul nostru auxiliar care este setat prima data cu X = -1 și Y = -1 , in vederea generări unui nou comparator. Dacă comparatorul nostru auxiliar este egal cu parintele cu indexul egal cu primul numar ales random de noi atunci in noua listă de comparatori se va pune comparatorul de la părintele cu indexul Y și comparatorul j. In caz contrar atunci se va pune comparatorul j de la părintele cu indexul X. Acest lucru este invers pentru cazul in 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.

**Figura 14.** Metoda de Crossover Indivizi

După ce am facut acest lucru vom sorta toti indivizi in 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. Mutație**

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 in sperața unei găsiri mai rapide a soluției. Mutația ne mai ajută și să pătrăm o anumită diversitate intre indivizi.



**Figura 15.** Metoda de Mutație

După cum putem vedea in 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 in această fază atunci se va crea un comparator nou random, in 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 in scopul creari unei noi ci vom crea o genă nouă cu valori random și atribuită genei respective.

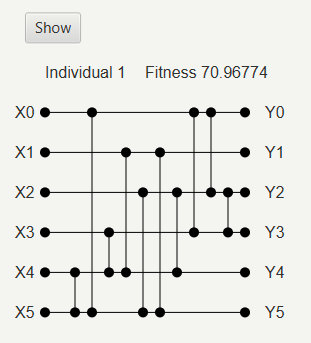
# *Capitolul VI.*

# **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 in 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

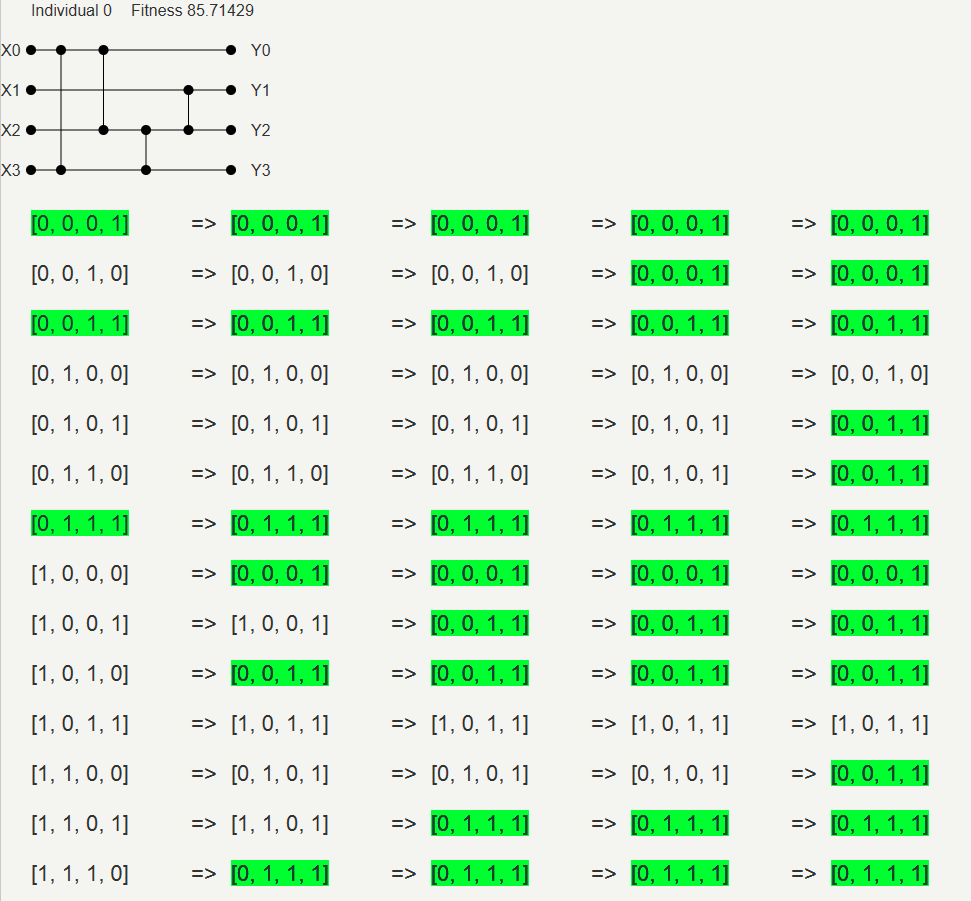
În figura 16 observam 6 fire din care putem rezulta ca există oricare 6 input-uri, și un număr comparatori. Fiecare fir are un punct la începutul lui și unul la final. Acest lucru reprezintă ca numărul va trece de la primul punct până la ultimul, parcurgând toți comparatorii care îi iese in cale. Numărul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

Un alt lucru pe care îl putem observa in imagine este fitness-ul pe care individul îl are, acesta reprezentând cat de bine se descurcă in a sorta un sir de numere de lungime 6 in cazul nostru. Reprezentarea acestor indivizi are o implementare destul de generica in 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 in afișarea unui individ este un buton cu numele ’Show’. Acesta va porni o noua interfață grafică in care vom putea observa detalii importante asupra individului nostru.

## **VI.2. Reprezentarea corectitudini unui individ**

După cum am putut observa in 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

In 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 in 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 in 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.

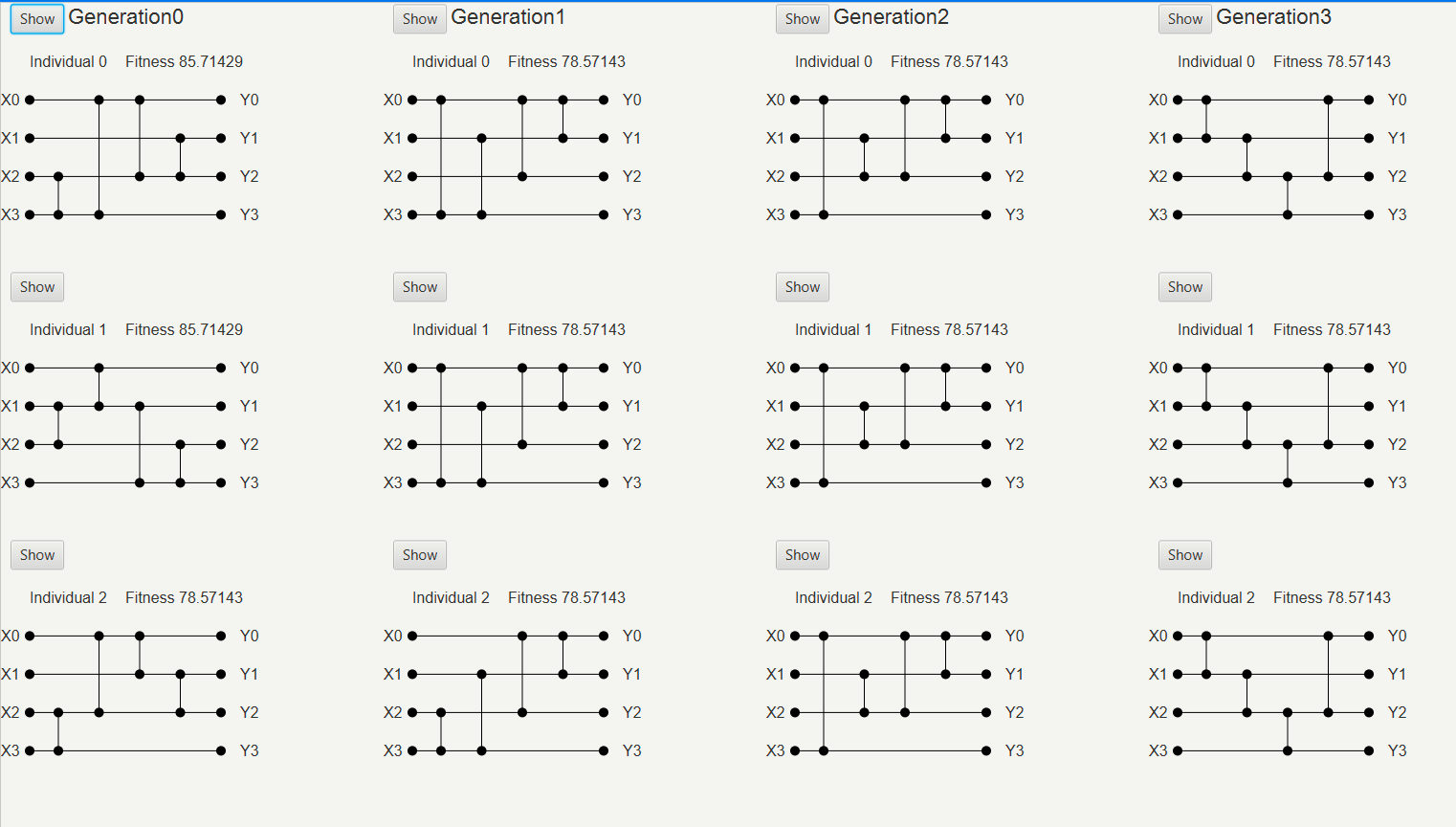
O alta remarca pe care o putem trage este ca atunci cand un sir de numere ajunge sa fie crescator atunci acesta este trasat cu o culoare verde.



**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 in care sa putem observa indivizii dintr-o anumită generație in scopul de a putea să vedem daca aceștia au ajuns să aibă o performantă mai buna in sortarea anumitor șiruri de numere. In figura 18 se vor observa avantajele aduse de decuplarea generări grafice a unui singur individ.



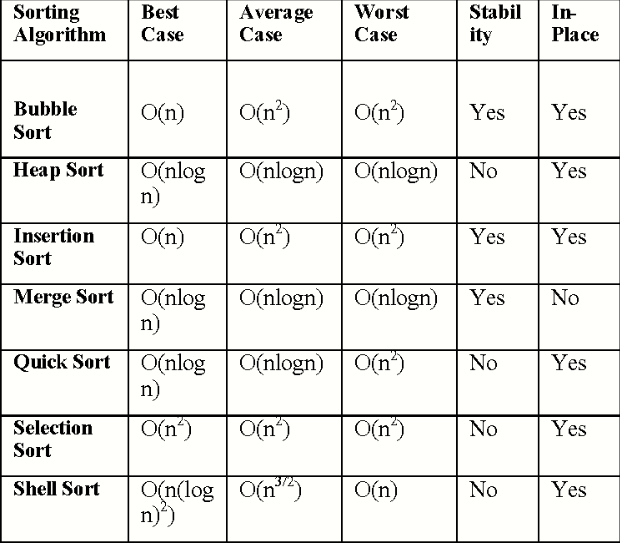
**Figura 19.** Vizualizarea Generatiilor

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 generatiei. 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.



**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 in cazul cel mai complex, asta înseamnă ca pentru a reuși sa sortăm o lista de n numere, o să fie nevoie de comparatorii, ceea ce este destul de mult. În schimb daca privim complexitățiile algoritmului de sortare QuickSort, acesta are complexitatea 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>