**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**

****

LUCRARE DE LICENȚĂ

**Rețele de sortare de adâncime minimă**

**propusă de**

***Vlad Corjuc***

**Sesiunea: Iulie, 2021**

**Coordonator științific**

Lector Doctor Cristian Frăsinaru

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**

Rețele de sortare de adâncime minimă

*Vlad Corjuc*

**Sesiunea:** *Iulie, 2021*

**Coordonator științific**

*Lector Doctor Cristian Frăsinaru*

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Data \_\_\_\_\_\_\_\_\_\_\_\_ Semnătura \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DECLARAȚIE privind originalitatea conținutului lucrării de licență**

Subsemnatul(a) ………………………………………………………………………………………

domiciliul în …………………………………………………………………………………………………..

născut(ă) la data de ………………..…., identificat prin CNP ………….……………..………………..., absolvent(a) al(a) Universității „Alexandru Ioan Cuza” din Iași, Facultatea de ………………………. specializarea …………………………………………………………, promoția …………………………., 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: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_elaborată sub îndrumarea dl. / d-na \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, 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 in vederea facilitării falsificării 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 student …………………………

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Titlul complet al lucrării*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de testetc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

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

Iași, *data*

Absolvent *Prenume Nume*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

**Cuprins**

[**Introducere** 1](#_Toc75215967)

[**Capitolul 1 – Rețele de sortare** 2](#_Toc75215968)

[**1.1 Concepte de bază privind rețelele de sortare** 2](#_Toc75215969)

[**1.2 Aplicații ale rețelelor de sortare** 3](#_Toc75215970)

[**Capitolul 2 – Rețele de sortare de adâncime minimă** 4](#_Toc75215971)

[**2.1 Prezentarea problemei** 4](#_Toc75215972)

[**2.3 Complexitatea problemei** 5](#_Toc75215973)

[**2.4 Lucrări Conexe** 6](#_Toc75215974)

[**Capitolul 3 – Rețele de sortare, aspecte suplimentare** 8](#_Toc75215975)

[**3.1 Notații utilizate** 8](#_Toc75215976)

[**3.2 Elemente necesare** 8](#_Toc75215977)

[**Capitolul 4 – Algoritmii Genetici** 9](#_Toc75215978)

[**4.1 Prezentare generală** 9](#_Toc75215979)

[**4.2 Terminologie** 10](#_Toc75215980)

[**Capitolul 5 – Algoritm genetic hibrid cu optimizare locală** 11](#_Toc75215981)

[**5.1 Prezentare Generală** 11](#_Toc75215982)

[**5.2 Reprezentarea problemei** 11](#_Toc75215983)

[**5.3 Perechi ordonate și neordonate** 13](#_Toc75215984)

[**5.4 Framework-ul algoritmului genetic** 14](#_Toc75215985)

[**5.4 Codificare** 14](#_Toc75215986)

[**5.5 Inițializarea populației** 15](#_Toc75215987)

[**5.6 Selecție** 16](#_Toc75215988)

[**5.6.1 Funcția fitness** 16](#_Toc75215989)

[**5.7 Crossover** 17](#_Toc75215990)

[**5.8 Mutație** 18](#_Toc75215991)

[**5.9 Înlocuirea populației** 19](#_Toc75215992)

[**5.10 Optimizarea locală** 19](#_Toc75215993)

[**5.10.1 Edit** 19](#_Toc75215994)

[**5.10.1 Repair** 20](#_Toc75215995)

[**5.11 Helper** 23](#_Toc75215996)

[**Capitolul 6 - Optimizare multi-obiectiv** 24](#_Toc75215997)

[**6.1 Noțiuni Fundamentale** 24](#_Toc75215998)

[**6.2 NSGA-II** 25](#_Toc75215999)

[**6.2.1 Sortarea de tip non-dominated** 25](#_Toc75216000)

[**6.2.2 Sortarea de tip crowding distance** 27](#_Toc75216001)

[**Capitolul 7 – Algoritm genetic multi-obiectiv cu selecție bazată pe *novelty*** 28](#_Toc75216002)

[**7.1 Prezentare Generală** 28](#_Toc75216003)

[**7.2 Codificare** 29](#_Toc75216004)

[**7.3 Inițializarea populației** 29](#_Toc75216005)

[**7.4 Crossover** 30](#_Toc75216006)

[**7.5 Mutație** 30](#_Toc75216007)

[**7.6 Funcțiile Obiectiv** 31](#_Toc75216008)

[**7.7 Selecția cu “novelty”** 31](#_Toc75216009)

[**Capitolul 8 – Rezultate** 33](#_Toc75216010)

[**8.1 Rezultate algoritm genetic hibrid cu optimizare locală** 33](#_Toc75216011)

[**8.2 Rezultate algoritm genetic multi-obiectiv cu novelty** 35](#_Toc75216012)

[**8.3 Comparație algoritmi** 36](#_Toc75216013)

[**Capitolul 9 - Concluzie** 37](#_Toc75216014)

[**Bibliografie** 37](#_Toc75216015)

# **Introducere**

Unul din cele mai studiate subiecte încă de la nașterea primelor calculatoare a fost *sortarea*, definită de Knuth D. E.ca o operație vitală nu doar în informatică, ci și în viața cotidiană [1, p. 1]. Sortarea unei grupări de elemente pe baza unui criteriu stabilit a avut o importanță deosebită datorită aplicabilității acesteia în majoritatea domeniilor.

În decursul anilor au fost prezentați numeroși algoritmi de sortare precum și structuri de date specifice pentru această operație. Astfel, una din primele structuri de date prezentate a fost *rețeaua de sortare*. Rețelele de sortare sunt structuri de date simple, bazate pe o serie de operații elementare (compararea și interschimbarea a două elemente), dar care au devenit centrul atenției multor cercetători interesați de studierea metodelor de sortare.

În această lucrare am abordat una din principalele probleme prezentată în studiul acestor structuri : *căutarea unei rețele de sortare de adâncime optimă*.

Alegerea temei de față a fost rezultatul curiozității mele cu privire la această metodă de sortare atât de simplă, dar cu un suport teoretic atât de vast, precum și rezultatul provocării pe care o lansează problema propusă, o problemă care este încă nerezolvată.

În soluționarea acestei probleme am utilizat două metode distincte bazate pe o clasă de algoritmi de optimizare inspirați din procesul evolutiv. Prima metodă este formată dintr-un algoritm genetic hibrid care utilizează diverse operații de optimizare locală. Cea de a doua metodă este reprezentată de un algoritm genetic multi-obiectiv în care o importanță majoră este oferită factorului de noutate.

Prin această lucrare am dorit sa ofer o soluție atât teoretică cât și practică asupra problemei căutării unei rețele de sortare de adâncime optimă folosind algoritmi dintr-o clasă de algoritmi de optimizare inspirați din procesul evolutiv.

# **Capitolul 1 – Rețele de sortare**

## **1.1 Concepte de bază privind rețelele de sortare**

Rețelele de sortare fac parte dintr-o clasă specială a algoritmilor de sortare prin comparare, independent de tipul datelor, ce sortează un număr fix de elemente. Rețelele de sortare adaugă această restricție a numărului fix de elemente, pe când algoritmii de sortare prin comparare, de obicei, sortează orice număr de elemente [1, p. 219].

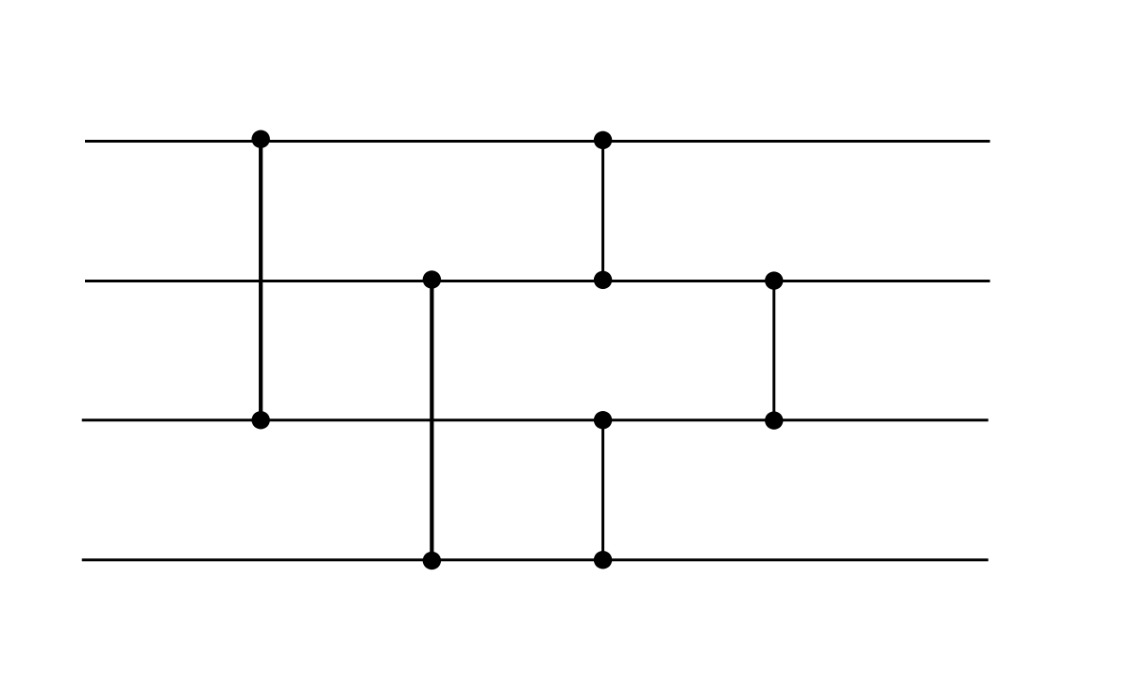
O rețea de sortare este formată din două componente : *comparatori* și *fire de date.*

Numărul *firelor* esteegal cu cel al elementelor de sortat, ele conținând, la fiecare pas, câte un element din șirul inițial (de sortat). Pentru o înțelegere mai clară putem considera *firele* asemenea unor canale de date ce parcurg rețeaua de sortare de la stânga la dreapta, precum se poate observa în Figura 1.1.

Fiecare *comparator* va conecta două dintre *firele* rețelei. Aceștia vor compara și inversa elementele conținute de *firele* care le poartă astfel încât elementul cu valoarea minimă să fie pe *firul* superior iar cel cu valoarea maximă pe *firul* inferior*.*

În practică vom reprezenta *firele* utilizând linii orizontale iar comparatorii prin linii verticale între două fire. Pentru a evidenția conexiunea pe care o face un comparator între două *fire* vom utiliza mici cerculețe negre. Un exemplu de rețea de sortare este cel prezentat în [Figura 1.1](#Figura11), pe când un exemplu de funcționare al rețelei se poate observa în [Figura 1.2](#Figura12).

*comparator*

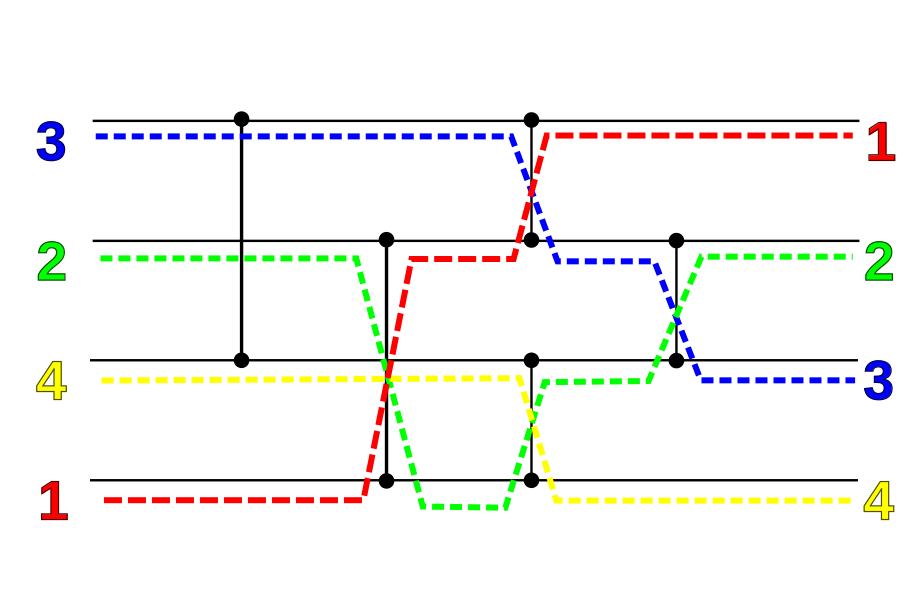


**Figura 1.1** : Rețea de sortare cu 4 fire

**Figura 1.2** : Funționarea unei rețele de sortare

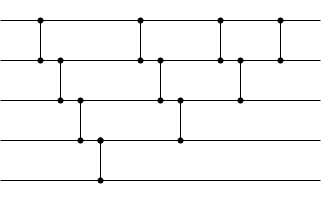
*comparator*

*fir*



Rețelele de sortare pot reprezenta orice algoritm de sortare bazat pe comparare și care este independent de tipul de date sortate.

În [Figura 1.3](#Figura13) este prezentat algoritmului *Bubble Sort* (în cazul sortării unui șir de cinci elemente) sub forma unei rețele de sortare cu cinci fire și un număr de comparatori egal cu numărul comparărilor efectuate de algoritmul în cauză. O diferență conceptuală între cele două sortări se poate observa la fiecare iterație: algoritmul Bubble Sort va aduce elementul cel mai mare „la suprafață” pe când rețeaua de sortare prezentată va aduce elementul cel mai mare cât mai jos posibil, pe firul cel mai inferior.



**Figura 1.3**: Rețea de sortare reprezentând algoritmul Bubble Sort

## **1.2 Aplicații ale rețelelor de sortare**

Rețelele de sortare pot fi implementate ușor fie în hardware, fie în software.

Rețelele de sortare sunt utilizate pentru a codifica constrângeri de tip *at-most-k* în SAT deoarece pot fi simulate în logica propozițională [2]. Astfel de constrângeri asigură faptul că cel mult k dintr-un set de variabile se află într-un set satisfăcător.

Batcher, în 1968, a sugerat utilizarea acestora pentru a construi rețele de comutare pentru hardware-ul computerului [3].

Din anii 2000, rețelele de sortare sunt utilizate de comunitatea GPGPU pentru construirea algoritmilor de sortare care să ruleze pe GPU.

Un alt exemplu al utilizării acestora este construirea unui filtru median pentru procesarea de imagini, utilizând câte o rețea de sortare pentru fiecare pixel care sortează pixeli vecini în funcție de intensitatea acestora [4, p. 8].

# **Capitolul 2 – Rețele de sortare de adâncime minimă**

## **2.1 Prezentarea problemei**

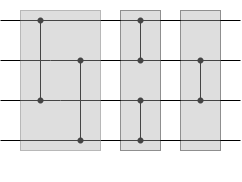
Încă de la introducerea rețelelor de sortare, cercetătorii au căutat rețele cu un număr minim de comparatoare astfel încât un prin criteriu pentru optimalitatea unei rețele a fost *dimensiunea* acestei: numărul de comparatori.

Ulterior s-a realizat posibilitatea aplicării în paralel a unui grup de comparatori (comparatori ce funcționează pe fire diferite). Un grup de comparatori consecutivi ce pot să fie aplicați în paralel se numește *strat*. Astfel a fost introdus un alt criteriu prin care putem analiza optimalitatea unei rețele de sortare, *adâncimea* (numărul de straturi).

În [Figura 2.1](#Figura21) sunt exemplificate aceste concepte noi asupra rețelei de sortare cu patru fire prezentate anterior.

**Figura 2.1**: Rețea de sortare de dimensiune 5 și adâncime 3

*strat*



În acest cadru au apărut două probleme de optimizare considerând dimensiunea și adâncimea unei rețele de sortare:

1. Care este dimensiunea minimă a unei rețele de sortare cu *n* fire? Sau echivalent, care este numărul minim de comparări necesare pentru a sorta un șir cu *n* elemente?

2. Care este adâncimea minimă a unei rețele de sortare cu *n* fire? Sau echivalent, care este numărul minim de operații paralele necesare pentru a sorta un șir cu *n* elemente unde fiecare operație este formată dintr-un număr de comparări ?

Aceste probleme pot să fie reduse în timp polinomial (demonstrat ulterior) la două probleme de decizie:

1. Există o rețea de sortare cu *n* fire și de dimensiune *k?*
2. Există o rețea de sortare cu *n* fire și adâncime *d?*

Reducerea problemelor de optimizare la cele prezentate se poate realiza cu ușurință. Putem observa că odată cu creșterea numărului de fire, dimensiunea și adâncimea minimă nu pot să scadă : Având o rețea optimală (din punct de vedere al dimensiunii sau al adâncimii) cu *n+1* fire, putem construi o rețea de sortare cu *n* fire eliminând ultimul fir, se poate observa că aceasta va avea dimensiunea și adâncimea cel mult egale cu cele ale rețelei cu *n+1* fire.

Utilizând afirmația anterioară putem transforma problemele de optimizare în probleme de decizie utilizând un algoritm de căutare precum *căutarea binară.*

În cadrul acestei lucrări am analizat problema determinării unei rețele de sortare cu *n* fire și adâncime minimă.

## **2.3 Complexitatea problemei**

Pentru a determina complexitatea problemei putem utiliza o metodă non-deterministă: găsim non-determinist o rețea cu n fire și de adâncime d și verificăm dacă aceasta sortează orice șir de *n* elemente. În urma considerației putem eticheta problema inclusă în NP. Însă această considerație a fost respinsă de rezultatele prezentate de Ian Parberry. Acesta a demonstrat că problema verificării faptului că o anumită rețea este o rețea de sortare este co-NP complete chiar și pentru rețelele de sortare cu adâncimea doar de mai mare decât adâncimea optimă [5].

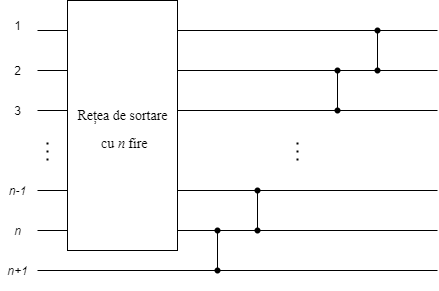
Utilizând un oracol co-NP putem clasifica problema de optimizare a rețelelor de sortare din punct de vedere al adâncimii în clasa de complexitate . Deoarece putem transforma problema de optimizare în problema de existență prezentată anterior în timp polinomial, putem încadra problema existenței unei rețele de sortare cu n fire și de adâncime d în aceiași clasă de complexitate.

## **2.4 Lucrări Conexe**

Rețele de sortare de adâncime optimă cu un număr de fire *n* ≤16 au fost determinate de Knuth în *" The Art of Computer Programming "* [1] în 1998; optimalitatea primelor opt a fost demonstrată încă din anul 1960 de Knuth și Floyd [1, p. 226]. În schimb, pentru rețele cu 9 și 10 fire a fost găsită o demonstrație în 1991 [7]. Alte rezultate nu au fost obținute până în anul 2014, când Závodný [8, pp. 236-249] a reușit să dovedească corectitudinea acestei proprietăți pentru cazurile restante analizate de Knuth.

În studiul rețelelor de sortare de dimensiune minimă a fost demonstrată optimalitatea primelor 10 rețele de sortare prezentate de Knuth utilizând un super computer. Determinarea corectitudinii rețelei de sortare de dimensiune minimă pentru 11 fire a făcut posibilă stabilirea dimensiunii pentru rețeaua de 12 fire deoarece limita superioară și cea inferioară a acestei proprietăți au devenit egale.

Utilizând schema de construcție prezentată în [Figura 2.2](#Figura22) , putem crea o rețea de sortare cu n+1 fire pornind de la una cu n fire adăugând un număr mic de comparatori. Utilizând schema prezentată sau echivalentul ei matematic: , denumită Lema Van Voorhis, a fost posibilă determinarea limitelor superioare pentru dimensiunea și adâncimea unei rețele de sortare.



**Figura 2.2**: Schema de construire a unei rețele de sortare ce are un fir în plus față de rețeaua de sortare de la care se pornește

Pe lângă această inegalitate, în decursul anilor au fost utilizate metode asistate de computer precum algoritmii evolutivi care au obținut rezultate remarcabile în determinarea de rețele de sortare de adâncime sau dimensiune minimă, astfel au fost obținute alte limite superioare pentru dimensiune și adâncime. Cu toate acestea, algoritmii evolutivi sunt incapabili să găsească rețele de sortare mai mici(de dimensiune sau adâncime mai mică) deoarece acest lucru necesită o căutare exhaustivă.

Una din soluțiile care merită menționată este transformarea problemei determinării unei rețele de sortare de adâncime d într-o propoziție din logica prepozițională, apoi utilizarea unui SAT-solver pentru a determina satisfiabilitatea acestei propoziții.

În [Tabelul 1.1](#Tabel11) sunt prezentate cele mai bune rezultate obținute în decursul anilor utilizând diverse metode exhaustive și non.

**Tabel 1.1**: Tabel cu dimensiunea minimă (*s*n) și adâncimea minimă (*d*n) a rețelelor de sortare cu n fire. Pentru valori necunoscute, celula conține limita inferioară și superioară. Pentru n ≤ 9 putem determina rețele de sortare care sunt atât de dimensiune optimă cât și de adâncime optimă, dar acest lucru nu este cazul pentru n=10 și n= 12 [4]



În studiul lucrărilor care utilizează algoritmi genetici cu rezultate remarcabilă se pot număra și "*A Hybrid Genetic Search for the Sorting Network Problem with Evolving Parallel Layers*" de Sung-Soon Choi și Byung-Ro Moon, și "*Enhanced Optimization with Composite Objectives and Novelty Selection*" de Hormoz Shahrzad, Daniel Fink și Risto Miikkulainen, lucrări de la care am pornit în realizarea soluțiilor propuse în această lucrare.

# **Capitolul 3 – Rețele de sortare, aspecte suplimentare**

## **3.1 Notații utilizate**

Pe lângă aspectele fundamentale necesare pentru a înțelege soluțiile propuse în această lucrare este necesară stabilirea notației pentru diversele elemente.

Valorile care se află la un moment dat pe cele n fire ale rețelei sunt reprezentate ca un vector . Aplicarea unui comparator peste vectorul va modifica valoarea în și valoarea în . Cu vom nota output-ul rezultat în urma aplicarea tuturor comparatorilor unei rețele C cu n fire asupra unui input ; putem observa că  este mereu o permutare a vectorului inițial .

## **3.2 Elemente necesare**

*Principiul zero-one*, prezentat de Knuth [1], afirmă că se poate determina validitatea unei rețele de sortare testând doar primele secvențe binare, acest lucru aduce o mare diferență față de cele secvențe necesare care făceau problema intratabilă pentru valori mai mari. Prin urmare, fără pierderea generalității, de acum înainte luăm în considerare doar rețelele de comparare cu secvențe de intrare binare cu .

Mulțimea tuturor secvențelor de ieșire produse de o rețea este notată ca . Este ușor de observat că o rețea este o rețea de sortare .

Putem extinde o rețea adăugând un alt comparator (sau un set de comparatoare) cu , noua rețea este notată cu . Fiecare rețea are mai multe rețele de prefix și sufix. Numim o rețea un prefix al și o rețea un sufix al lui dacă există o descompunere a lui în și adică . În special, permitem unei rețele să nu aibă comparatoare astfel încât fiecare rețea este un prefix și un sufix propriu.

Fie x o secvență binară de intrare de lungime n. Vom utiliza următoarele notații: și .

# **Capitolul 4 – Algoritmii Genetici**

## **4.1 Prezentare generală**

Algoritmii genetici au fost propuși de John Holland in 1973 după mulți ani de studiere a ideii de simulare a evoluției. Acești algoritmi modelează moștenirea genetica si lupta Darwiniana pentru supraviețuire. Lucrează cu o populație de soluții candidat, care evoluează și se adaptează unui mediu (în cazul de față, mediul este funcția de optimizat). Alături de alte doua direcții: strategiile evolutive si programarea evolutiva, formează clasa algoritmilor evolutivi.

Genotipul suferă modificări:

* în urma mutației (modificarea unei gene din codificarea unui individ)
* în urma recombinării codului genetic a doi indivizi)

După principiul evoluționist „survival of the fittest”, de la o generație la alta se va favoriza supraviețuirea celor mai buni (bine adaptați) indivizi [9].

Pseudocodul unui astfel de algoritm este prezentat în [Figura 4.1](#Figura41).

|  |
| --- |
| t := 0  generează populația inițială P(t)  evaluează P(t)  while not Condiție De Oprire  t := t + 1  selectează P(t) pornind de la P(t-1)  mutație pe P(t)  cross-over pe P(t)  evaluează P(t) |
|  |

**Figura 4.1**: Pseudocodul unui algoritm genetic

## **4.2 Terminologie**

Algoritmii evolutivi utilizează un vocabular împrumutat din genetica [10]:

* evoluția este simulata printr-o succesiune de *generații* ale unei *populații* de soluții candidat;
* o soluție candidat poarta numele de *cromozom* si este reprezentata ca un sir de *gene*;
* *gena* este informația atomica dintr-un cromozom;
* toate valorile posibile pentru o gena formează setul de *alele* ale genei;
* populația evoluează prin aplicarea operatorilor genetici: *mutația* si *încrucișarea* (crossover);
* cromozomul asupra căruia se aplica un operator genetic se numește *părinte* (parent) iar cromozomul rezultat se numește *descendent* (offspring);
* *selecția* este procedura prin care sunt aleși cromozomii ce vor supraviețui in generația următoare; indivizilor mai bine adaptați li se vor da șanse mai mari;
* gradul de adaptare la mediu este măsurat de funcția *fitness*;
* soluția returnata de un algoritm genetic este cel mai bun individ din ultima generație.

# **Capitolul 5 – Algoritm genetic hibrid cu optimizare locală**

## **5.1 Prezentare Generală**

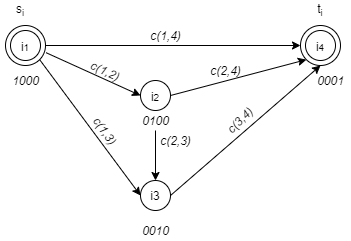
Prima dintre cele două soluții propuse în rezolvarea problemei este un algoritm hibrid cu optimizare locală. Utilizând reprezentarea sub formă de graf a problemei am definit o nouă euristică de optimizare locală care a adus la rezultate comparabile cu cei mai buni algoritmi pentru rezolvarea acestei probleme.

## **5.2 Reprezentarea problemei**

Un lucru esențial în elaborarea euristicii de căutare a fost transformarea problemei în graf pentru a putea determina diverse metode de optimizare locală pentru diversele stări ale rețelei.

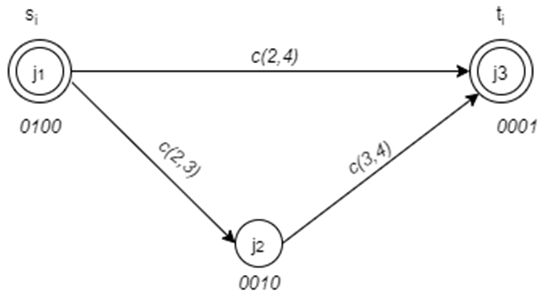
Considerăm o secvență de intrare *si* și rezultatul sortării acestei secvențe *ti*, de exemplu, pentru o rețea de sortare cu 4 fire în care *si* = 1000vom obține *ti* = 0001. Pornind de la acest exemplu vom construi graful direcționat Gi care să conțină nodul *si* ca punct de plecare (source) și nodul *ti* ca nod terminal (target). Restul nodurilor intermediare o să conțină starea secvenței în urma aplicării comparatorului indicat de muchia ce intră în nod. Un exemplu al acestui graf pentru exemplul prezentat anterior se poate observa în [Figura 5.1](#Figura51). În acest exemplu cea mai bună rețea de sortare este rețeaua formată din comparatorul (1,4) deoarece reprezintă cel mai scurt drum de la nodul de start la cel final, deci, cele va conține cele mai puține layere. Nodul din graf obținut prin tranziția de la nodul curent utilizând comparatorul (i,j) îl vom nota cu .

**Figura 5.1**: Graful problemei pentru *si* = 1000



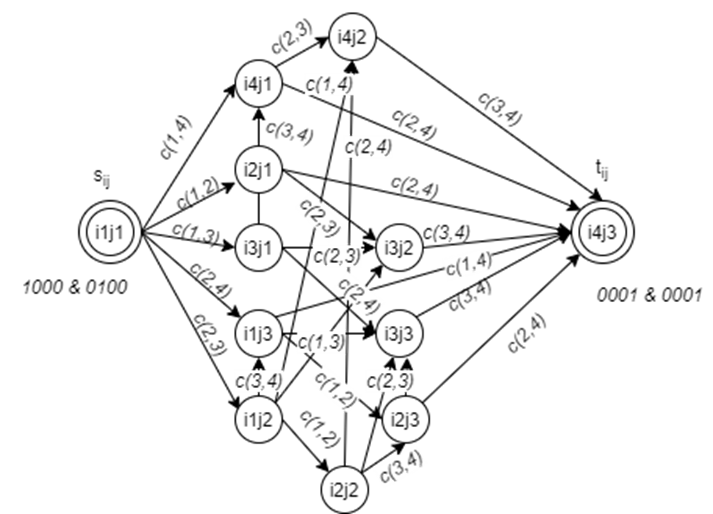
Am putea deduce că rezolvarea problemei se poate reduce la căutarea unui drum minim în graful problemei, dar se observă rapid că deși aceasta este soluția pentru o singură secvență, e imposibil de găsit o soluție analizând doar grafurile pentru fiecare secvență de intrare în parte. Din această cauză introducem operatorul ⊗ pentru a combina două grafuri Gi și Gj.

Pentru a înțelege funcționarea operatorului ⊗ vom utiliza exemplu anterior al grafului Gi . Introducem un nou graf Gj care are ca nod de start *sj* = 0100 și nodul țintă *tj* = *ti* ([Figura 5.2](#Figura52)). Graful Gi ⊗ Gj prezentat în [Figura 5.3](#Figura53) este graful rezultat în urma utilizării operatorului ⊗ definit anterior. Din graful rezultat, putem determina cu ușurință rețelele de sortare de adâncime minimă care să sorteze cele două secvențe, în acest caz avem există mai multe variante dacă parcurgem drumurile de lungime minimă. Putem deci să construim graful G utilizând [Formula 5.1](#Figura52).



**Figura 5.2**: Graful problemei pentru *sj* = 0100

**Formula 5.1**: Formula de construire al grafului G



**Figura 5.3**: Graful problemei pentru Gi ⊗ Gj

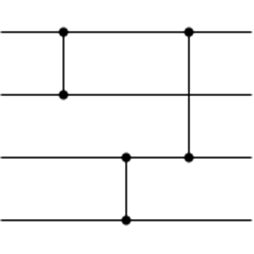
## **5.3 Perechi ordonate și neordonate**

Numim *pereche ordonată* a unei rețele de sortare , orice pereche de fire . Această pereche asigură ordonarea oricărui șir de intrare cu privire la cele două fire, vom nota o astfel de pereche .

Pe de altă parte, orice pereche de fire o vom numi *pereche neordonată.* Existența unei astfel de perechi face imposibilă sortarea oricărei intrări pentru rețea, vom nota o astfel de pereche .

Figura 5.4 prezintă o rețea cu 4 fire; indiferent de ce secvență de intrare oferim rețelei observăm că perechea (1,2) o să fie mereu ordonată în secvența de ieșire a rețelei. Pe lângă această pereche ordonată avem și perechile și . Pe de altă parte pentru perechea de fire (2,3) există secvențe de intrare pentru care firele 2 și 3 nu sunt ordonate (precum ). Deci este o pereche neordonată. Alte perechi neordonate ale acestei rețele sunt și .

Vom nota prin mulțimea perechilor ordonate a rețelei iar cu mulțimea perechilor neordonate. Este clar că (deorece numărul total de perechi pentru o rețea cu n fire este ).



**Figura 5.4**: O rețea de sortare invalidă

## **5.4 Framework-ul algoritmului genetic**

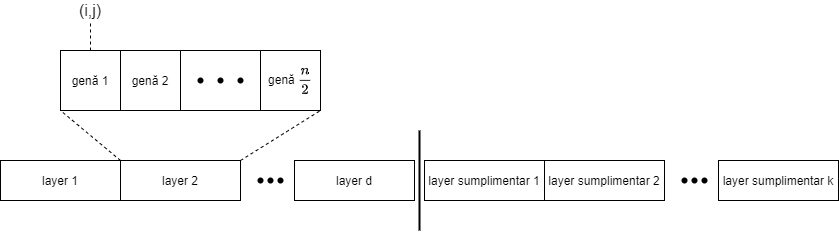
În [Figura 5.5](#Figura55) este prezentată schița algoritmului genetic utilizat pentru prima soluție. Pentru a evolua corect straturile paralele, am utilizat o nouă schemă de codificare și diverși operatori genetici specifici precum și o funcție de *fitness* nouă. În cele ce urmează, descriem detaliile algoritmului nostru genetic.

|  |
| --- |
| {       } |

**Figura 5.5**: Schița algoritmului genetic

## **5.4 Codificare**

Fiecare cromozom al populație va reprezenta o rețea de sortare(validă sau invalidă). În [Figura 5.6](#Figura56) se poate observa structura unui cromozom. Un cromozom este alcătuit dintr-un număr fix de *layere* (≥ adâncimea dorită), fiecare layer este alcătuit dintr-un număr delimitat de gene. În practică am observat că introducând două *layere* suplementare obținem rezultatele optime. O genă reprezintă un comparator și este formată dintr-o pereche de fire. Ordinea straturilor din cromozom este în mod trivial același cu cel al straturilor rețelei de sortare; și ordinea genelor din din cromozom este aceeași cu cea a comparatorilor în stratul corespunzător al rețele de sortare. Numărul de gene din fiecare strat este cel mult jumătate din numărul de fire.(Într-o rețea cu 16 fire, fiecare strat paralel are cel mult 8 gene.)



**Figura 5.6**: Codificarea unei rețele de sortare

## **5.5 Inițializarea populației**

Am stabilit dimensiunea populației la 100 de indivizi, acest număr este semnificativ mai mic față de lucrările anterioare pentru această problemă, Lee Graham, Hassan Masum și Franz Oppacher în " *Statistical Analysis of Heuristics for Evolving Sorting Networks* " [11] determină dimensiunea optimă pentru algoritmii *state-of-the-art* ai problemei ca 500 de indivizi.

Din cauza complexității problemei am utilizat ca prefix pentru fiecare individ o rețea de comparare predefinită inspirată din *filtrul Green.* M. W. Green, în 1969 a găsit o rețea de sortare pe 16 fire cu 60 de comparatoare (care este încă recunoscut ca cel mai bun rezultat pentru o rețea de 16 fire). Pentru a obține acest rezultat a concatenat diverși comparatori unei rețele inițiale de 32 de comparatori pe care o numește Green Filter. Deși ar putea exclude posibile soluții optime, utilizarea unui prefix scade considerabil timpul de calcul necesar. În [Figura 5.7](#Figura57) este prezentat algoritmul utilizat pentru a crea o versiune generalizată a filtrului Green, potrivit pentru orice număr de fire. Pentru straturile paralele neocupate vom genera în mod aleatoriu un număr de comparatoare între și . În urma inițializării populației vom aplica operația *Edit* pentru a transforma fiecare individ într-un individ valid.

|  |
| --- |
| F = ∅ len = 1 while len < n {  for k = 1…len {  for i = k…n-len, i+=2{  F = F ∪ (i,i+len)  }  }  len = 2 \* len } |
|  |
|  |
|  |

**Figura 5.7**: filtru Green

## **5.6 Selecție**

Fiecare cromozom este selectat ca părinte cu o probabilitate proporțională cu valoarea funcției fitness. Aceasta este o schemă tipică de selecție proporțională.

### **5.6.1 Funcția fitness**

Pentru elaborarea funcției fitness am pornit de la funcția fitness determinată de Frăsinaru Cristian și Mădălina Răschip [12]. În [Formula 5.1](#Formula51) este prezentată funcția fitness propusă în această lucrare. Această funcție explorează mai mult spațiul de ieșire al unei rețele față de funcțiile utilizate în cele mai multe abordări ale problemei. Pentru a înțelege funcționarea acestei funcții trebuie să definim mulțimea pentru o rețea de sortare .

Fie o secvență de ieșire a rețelei de sortare și *j* numărul de zerouri a secvenței. Definim mulțimile și . Este clar că o rețea este rețea de sortare dacă Mulțimea pozițiilor pentru care există o secvență de ieșire neordonată este definită ca .

**Formula 5.1**: funcția fitness prezentată de Frăsinaru Cristian și Mădălina Răschip

Deși această funcție este una din cele mai bune pentru determinarea unei rețele de sortare, aceasta nu este suficient de expresivă în cadrul problemei determinării unei rețele de sortare de adâncime minimă. Pentru a rezolva această problemă am introdus o penalitate care utilizează straturile paralele suplimentare pentru a penaliza soluțiile cu un număr mai mare de straturi paralele. Această penalitate este prezentată în [Formula 5.2](#Formula52). În această formulă este adâncimea dorită, (rețea formată și din layer-ele suplimentare).  
 reprezintă numărul comparatorilor din straturile suplimentare a cromozomului.

**Formula 5.2**: Formula de penalizare utilizată împreună cu funcția fitess

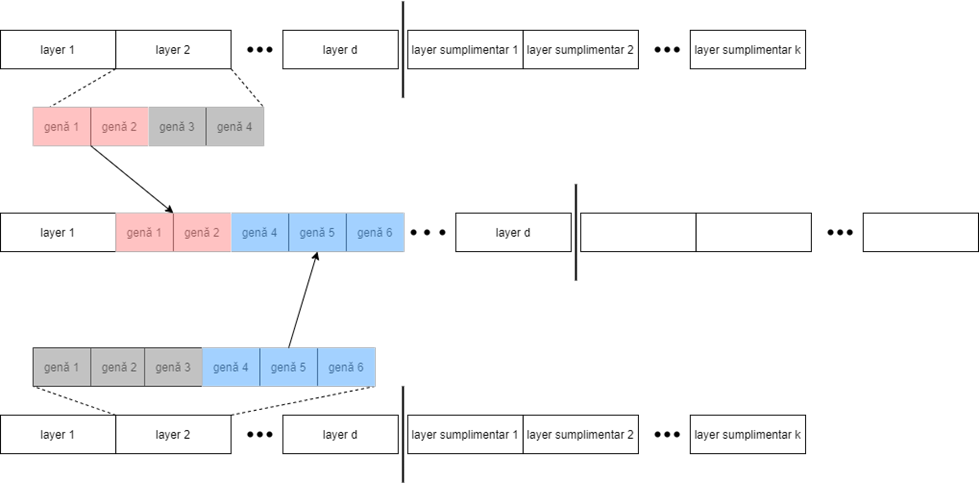
## **5.7 Crossover**

În acest algoritm genetic doar genele din straturile paralele sunt utilizate pentru crossover. Genele din straturile suplimentare sunt utilizate la evaluarea funcției fitness (partea de penalizare). Putem să numim acest algoritm un algoritm genetic hibrid Baldwinian deoarece utilizează *efectul Baldwin* : un individ nu va moșteni *cunoștința* dobândită de cei doi părinți ci doar informația genetică principală [13].

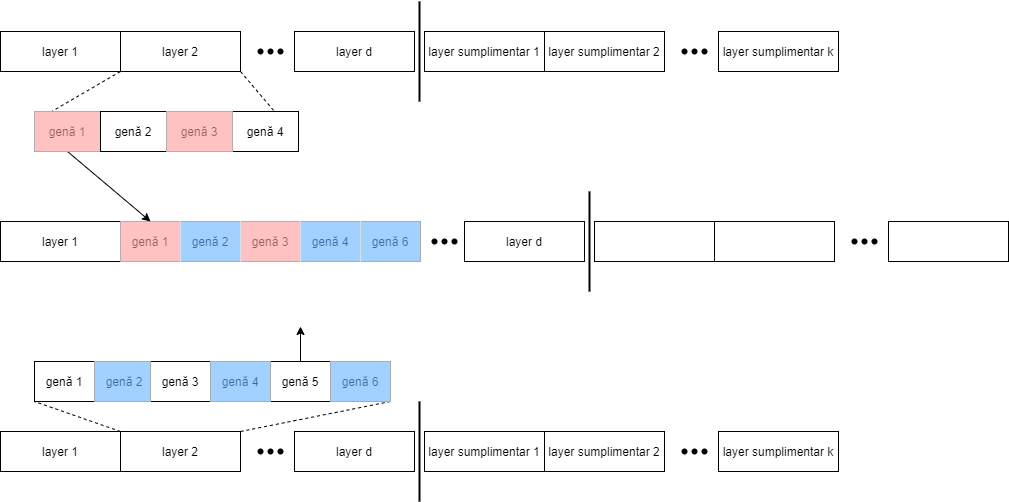
Am implementat două metode distincte de crossover : *single-point* și *uniform* care acționează pe fiecare layer în parte, dar acestea diferă de implementările standard deoarece avem o lungime variabilă a unui strat pentru doi indivizi diferiți.

În crossover-ul de tip *single-point* , prezentat în [Figura 5.8,](#Figura58) pentru fiecare strat paralel, se generează un punct de tăiere circa la aceiași poziție în cei doi părinți și se realizează încrucișarea acestora. În urma aplicării acestui operator genetic rezultă un nou individ rezultând un nou cromozom. Deoarece puține straturi noi generate sunt valide, vom elimina din fiecare strat comparatori până când nu mai există doi comparatori care să aibă un fir comun.

În crossover-ul de tip *uniform*, prezentat în [Figura 5.9](#Figura59), pentru fiecare strat paralel, se introduc în mod alternant gene din cei doi părinți pe baza unei probabilități (cu un *bias* mai mare spre părintele cu mai multă informație genetică în acel strat ), dacă numărul de gene nu este egal vom adăuga genele neatinse al părintelui cu informația genetică mai mare. Vom uiliza aceiași metodă de eliminare a comparatorilor pentru a valida noul individ.



**Figura 5.8**: Crossover de tip *single-point*



**Figura 5.9:** Crossover de tip *uniform*

## **5.8 Mutație**

Pentru mutație am implementat două metode diverse: mutația *simple* și mutația *indirect replacement.*

În prima dintre aceste mutații selectăm aleatoriu fiecare comparator, din fiecare strat, cu o probabilitate redusă (cea mai bună în practică s-a dovedit a fi 0.05) și schimbăm la întâmplare unul din firele de intrare (cel superior sau inferior cu o probabilitate de 50%). În cazul în care deja există un comparator în același strat care să partajeze firul schimbat vom conecta acesta la firul (în mod necesar) neocupat.

Mutația de tip *indirect replacement* este o mutație clasică în rezolvarea acestei probleme. Acest operator va alege un comparator din rețea și îl va șterge, apoi va alege un punct de inserție în care să plaseze un nou comparator generat aleatoriu păstrând numărul de comparatori fix.

Cele mai bune rezultate au fost obținute utilizând ambele metode cu o proporție de   
70% -30% (se va alege de mai multe ori primul tip de mutație).

## **5.9 Înlocuirea populației**

Vom înlocui cel mai rău dintre părinții selectați dacă noul individ este mai bun, din punct de vedere al fitness-ului, decât unul dintre părinți. Dacă noul individ are un fitness mai rău atunci vom înlocui cel mai rău membru al populației cu cromozomul nou creat. Prin aceasta se realizează un compromis între schema de *preselecție* și cea de tip *Genitor- replacement* care a adus rezultate promițătoare în diverse lucrări [14].

## **5.10 Optimizarea locală**

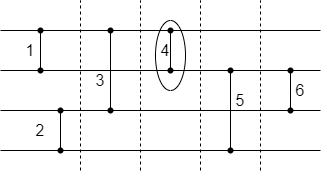
Optimizarea locală e constituită din două operații consecutive: *edit* și *repair*. Aceste operații au fost prezentate inițial de Sung-Soon Choi și Byung-Ro Moon pentru rezolvarea problemei căutării unei rețele de sortare de dimensiune minimă, dar au fost adaptate și modificate de mine pentru a rezolva problema abordată.

### **5.10.1 Edit**

Operația de edit este utilizată pentru a elimina din cromozom comparatori în funcție de un anumit criteriu. După procesul de editare, poate exista un strat paralel care să aibă niciun comparator. În acest caz, vom deplasa la stânga straturile paralele după acesta. În continuare voi prezenta cele două variante ale acestei operații și diversele puncte forte și slabe ale acestora.

*Correct wire edit* este cea mai simplă și prima variantă pe care am utilizat-o în algoritm, aceasta va elimina din fiecare strat al cromozomului orice comparator care împărtășește un fir de intrare cu un alt comparator din același strat.

Deși prima metodă a fost suficientă pentru a produce rezultate corecte, am decis utilizarea unei alte variante de editare bazată pe redundanța comparatorilor. Dacă o pereche de fire de intrare corespunzătoare unui comparator este o pereche ordonată în rețeaua obținută din comparatorii anteriori, comparatorul este redundant. Putem să găsim cu ușurință comparatorii redundanți: parcurgem toate secvențele de input pentru testarea rețelei și verificăm ce comparatori nu schimbă nicio pereche de intrare. Perechea firelor unui astfel de comparator este o pereche ordonată chiar și fără acesta. Pentru a exemplifica noțiunea de comparator redundant putem să considerăm rețeaua cu 4 fire din [Figura 5.10](#Figura510). Perechea de fire (1,2) este o pereche ordonată față de comparatoarele 1, 2 și 3; comparatorul 4 este deci redundant.



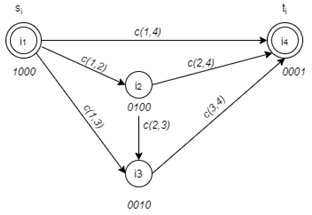
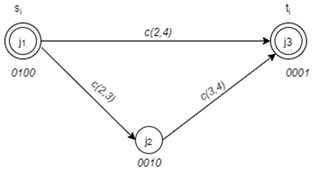
**Figura 5.10**: Comparator redundant

### **5.10.1 Repair**

Probabilitatea să obținem o rețea de sortare validă în urma crossover-ului și a mutației este foarte mică de aceea am introdus două operații consecutive care vom adăuga comparatori noi utilizând reprezentarea sub formă de graf a spațiului problemei.

Prima dintre aceste operații este ***Anexarea***. Cum am putut observa în secțiunea [5.2](#_5.2_Reprezentarea_problemei), problema poate să fie reprezentată ca un graf G ([Formula 5.1](#Formula51)) în care fiecare nod intermediar corespunde unei rețele de sortare invalide (reamintim că o rețea de sortare se numește invalidă dacă nu sortează orice secvență de intrare). Anexarea unui comparator la o rețea reprezintă deplasarea de la nodul curent din graful G prin intermediul muchiei ce corespunde comparatorului inserat. În acest proces un drum de lungime minimă va asigura folosirea unui număr minim de layere.

Deoarece spațiul problemei este foarte vast nu putem genera tot graful problemei pentru a putea determina un drum de lungime minimă vom analiza doar vecinii imediați ai nodului curent și vom alege cel mai promițător dintre ei. Vom nota mulțimea vecinilor imediați ai nodului curent *v* ca . Vom selecta nodul din care reduce cele mai multe perechi neordonate ale rețelei curente. Pentru a exemplifica vom utiliza o rețea de sortare cu 4 fire dar fără comparatori și vom reduce secvențele de input la secvențele prezentate în secțiunea 5.2, astfel avem *si* = 1000 și *sj* = 0100, reprezentarea acestora sub formă de graf individual se poate observa în [Figura 5.1](#Figura51) și [Figura 5.2](#Figura52) (Am reluat imaginile pentru simplitate). Graful complet al problemei este prezentat în [Figura 5.3](#Figura53). După cum observăm în [Tabelul 5.1](#Tabel51) comparatorul a cărei tranziție va reduce cel mai mult numărul de perechi neordonate este comparatorul (1,4). Acest comparator face parte din cel mai mic drum în graful problemei pentru care obținem o rețea de sortare cu număr minim de nivele. Acest lucru se poate vedea în [Figura 5.11](#Figura511) unde sunt evidențiate doar drumurile minime în graful G ( [Figura 5.3](#Figura53) ).

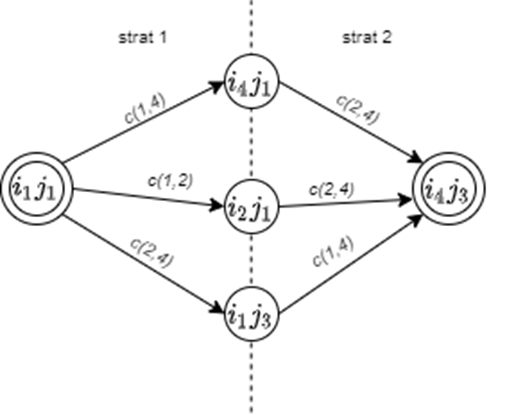


**Figura 5.1**: Graful problemei pentru *si* = 1000

**Figura 5.2**: Graful problemei pentru *sj* =0100

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| 1000 0100 | 0001 0100 | 0100 0100 | 0010 0100 | 1000 0001 | 1000 0010 |
|  | 2 | 4 | 3 | 3 | 4 |

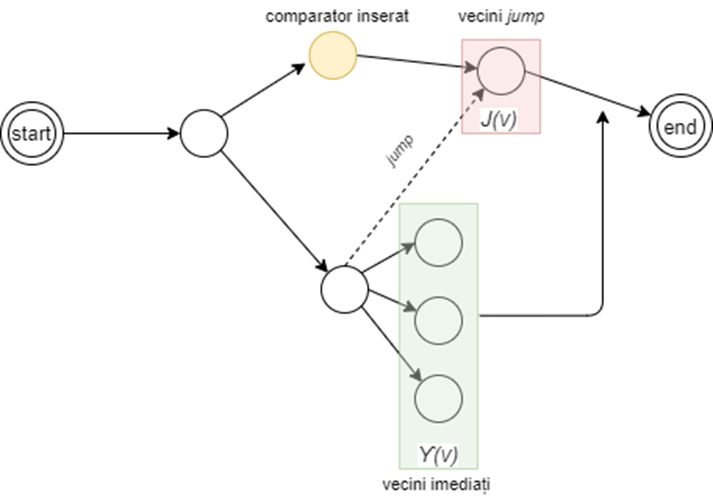
**Tabelul 5.1**: numărul perechilor neordonate



**Figura 5.11**: Drumurile de lungime minimă

Următoarea operație pe care o aplicăm este **Inserarea**. Cum am prezentat anterior, operația de Anexare va adăuga la finalul rețelei de sortare comparatori promițători, însă putem obține soluții promițătoare inserând un comparator nou după oricare genă a fiecărui strat. Pentru o rețea avem poziții în care putem insera un comparator, unde reprezintă numărul de comparatori ai rețelei.

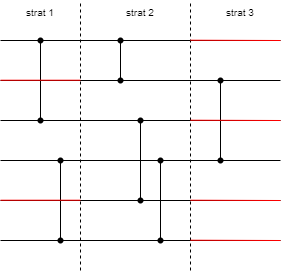
Inserarea unui comparator, într-o poziție diferită de cea finală, poate să nu realizeze o tranziție a grafului problemei spre unul din nodurile , precum se întâmpla în cazul *Anexării*, ci poate să aducă problema într-un nou nod al grafului. Putem observa această proprietate în [Figura 5.12](#Figura512). Numim nodurile la care "sărim" prin inserarea unui comparator *nodurile jump* și vom nota mulțimea acestor noduri ca , unde *v* este nodul curent.



**Figura 5.12**: Anexarea unui comparator prin *Inserare*

În cazul Inserării, pentru selectarea unui vecin promițător a unui nod *v*, putem extinde spațiul de căutare luând în considerare atât cât și . Suntem nevoiți să reducem nodurile considerate ai mulțimii deoarece este prea costisitor să parcurgem întreaga mulțime. Restricționăm căutarea selectând nodurile care nu distrug structura curentă a straturilor paralele.

Deoarece operația de Inserare se realizează după operațiile genetice de încrucișare și mutație, aceasta poate modifica semnificativ noul individ prevenind moștenirea structurii straturilor paralele ale părinților. În acest context, pentru fiecare strat paralel în parte, vom alege doar nodurile mulțimii pentru care comparatorul inserat conectează două fire neutilizate. Vom nota mulțimea acestor noduri cu . Pentru rețeaua din [Figura 5.13](#Figura513) numărul de comparatorii care realizează tranziția către un nod din pentru fiecare nod sunt 1 pentru primul strat paralel, 0 pentru stratul doi și 6 pentru ultimul strat. Pentru cazul general numărul de comparatori selectați sunt unde *unusedWire* este egal cu numărul de fire neutilizate.



**Figura 5.13:** rețea invalidă cu firele neutilizate evidențiate pentru fiecare strat evidențiate

Deoarece în urma operației de Inserare putem obține comparatori redundanți, utilizăm operația de Edit definită anterior. Lungimea unui cromozom în urma inserării poate să fie mult mai mică decât cea a cromozomului inițial, de aceea vom considera și lungimea în urma aplicării operației în selectarea de noduri promițătoare.

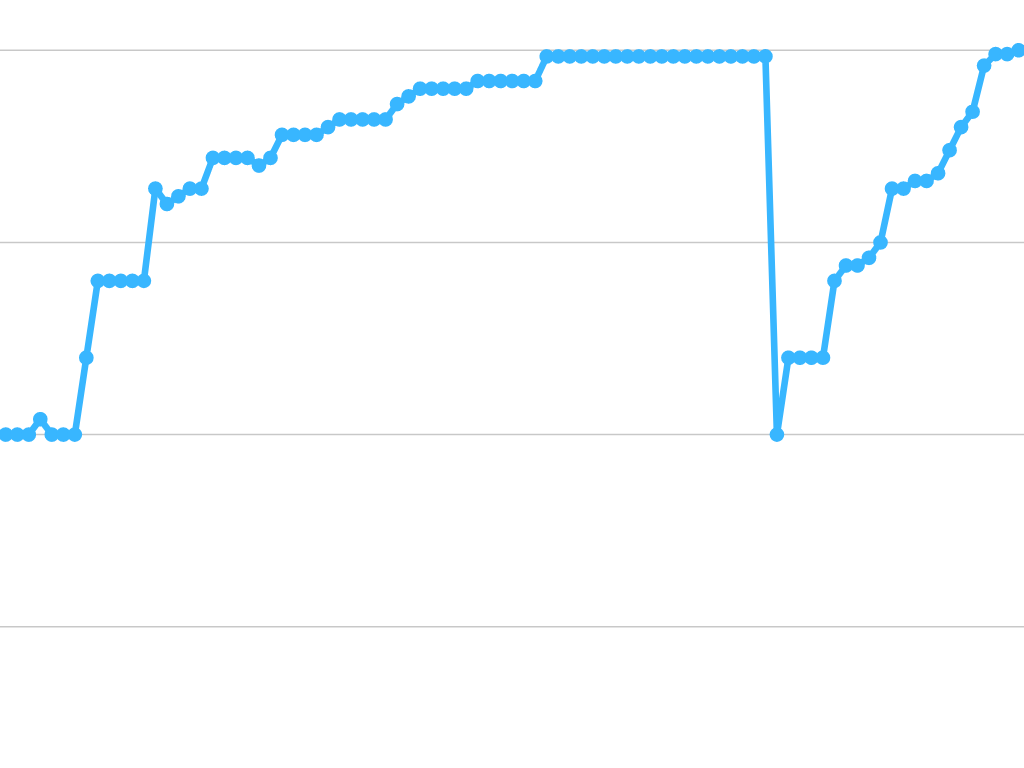
Ordinea de selecție a nodurilor promițătoare este :

1. Un nod din rețeaua rezultată să aibă un număr mai mic de straturi și să reducă numărul de perechi neordonate.
2. Un nod din rețeaua rezultată să aibă un număr mai mic de straturi și să aibă același număr de perechi neordonate.
3. Un nod din rețeaua rezultată să aibă un număr egal de straturi și să aibă același număr de perechi neordonate.
4. Nodul din care reduce un număr cât mai mare de perechi neordonate.

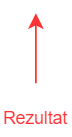
## **5.11 Helper**

Pentru a îmbunătății algoritmul genetic este utilizat un *helper*. Acest helper va acționa atunci când fitness-ul cel mai bun nu suferă modificări pentru un număr de generații(pentru intrările mai mari a fost necesar să utilizăm un helper cu 1000 de generații), atunci când este atins numărul de generații suportate, 40% din populație este eliminată și sunt adăugați cromozomi noi.

Acest algoritm a fost necesar pentru a explora spațiul de căutare și pentru depășirea platourilor de minim local. Utilitatea acestui helper este clară în graficul din [Figura 5.14](#Figura514).



**Figura 5.14**: Funcționarea helperului(9 fire, adâncime 7)



*0.5*

*0.75*

*1*

# **Capitolul 6 - Optimizare multi-obiectiv**

## **6.1 Noțiuni Fundamentale**

O problemă de optimizare multi-obiectiv poate să fie definită cum urmează:   
Cunoscând orice variabilă de decizie n-dimensională ce aparține spațiului problemei Χ, să se determine un vector , care să minimizeze un set K de funcții obiectiv . Spațiul soluției este deseori restricționat utilizând o serie de restricții și limite. Un exemplu de spațiu restricționat poate să fie , cu   
, unde și , .

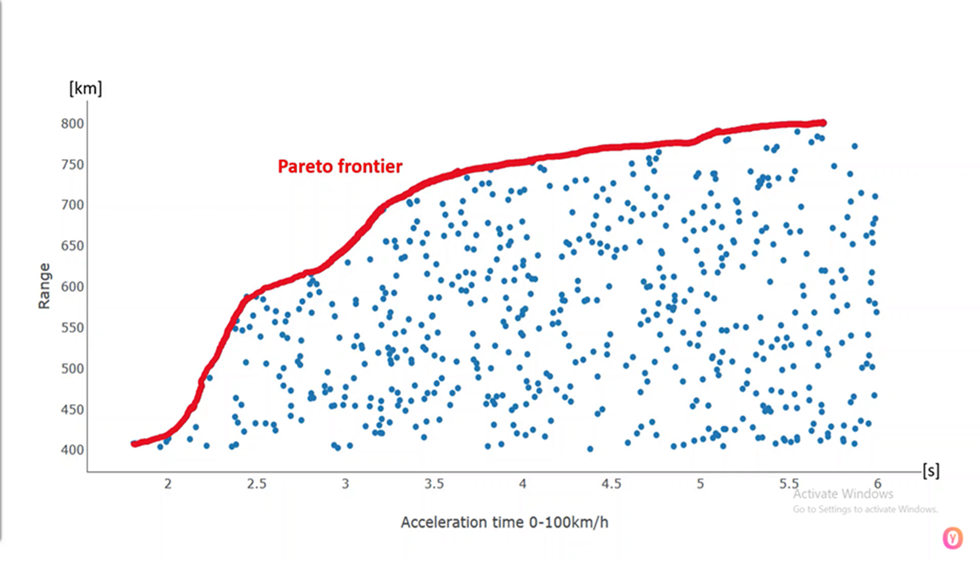
Deoarece în practică, obiectivele luate în considerare ajung să fie în conflict, nu este posibilă găsirea unei soluții ce optimizează simultan toate obiectivele introduse. Din cauza acestei afirmații, problemele de tip multi-obiectiv au fost evitate de majoritatea cercetătorilor, se prefera transformarea oricărei probleme multi-obiectiv într-o problemă cu un singur obiectiv. Problemele de optimizare multi-obiectiv au fost reluate odată cu introducerea noțiunilor de *soluție Pareto* și *front Pareto*.

O soluție rezonabilă la o problemă multi-obiectivă este investigarea un set de soluții în care fiecare soluție satisface obiectivele la un nivel acceptabil fără a fi însă dominată de o altă soluție. O soluție *x* care nu este dominată de o altă soluție *y* se numește *soluție Pareto.*

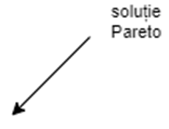
Un set de soluții Pareto se numește *set optimal Pareto*, pentru fiecare set optimal Pareto, valorile corespunzătoare funcției obiectiv în spațiul obiectivelor este denumit *front Pareto*.

Pentru a înțelege aceste concepte vom utiliza un exemplu practic: Să presupunem că se dorește lansarea pe piață a unei mașini electrice pentru care se dorește minimizarea timpului de accelerație și maximizarea autonomiei mașinii. Presupunem că putem să analiză doar trei variabile: mărimea unei roți, puterea motorului și capacitatea bateriei. În [Figura 6.1](#Figura61) se pot observa diversele elemente prezenatate anterior. Acest exemplu îl vom relua la prezentarea operațiilor specifice unui algoritm multi-obiectiv specific: *NSGA-II*.

Fiind o abordare bazată pe evoluarea unei populații, algoritmii genetici sunt foarte potriviți pentru a rezolva probleme de optimizare multi-obiectiv. Un algoritm genetic generic, cu un singur obiect, poate fi modificat pentru găsirea unor soluții nedominate fără un cost computațional mare. Capacitatea algoritmilor evolutivi de a explora simultan diferite regiuni ale spațiului soluției face posibilă determinarea unui set divers de soluții pentru probleme dificile cu spații(soluțiilor) non-convexe, discontinue și multimodale.



**Figura 6.1**: Front și soluție Pareto



## **6.2 NSGA-II**

Algoritmul NSGA-II este un algoritm genetic utilizat în rezolvarea problemelor cu obiective multiple. Acest algoritm a apărut ca îmbunătățire a algoritmului NSGA, care prezenta o serie de defecte care îl făceau inutilizabil în practică, principalele dintre acestea sunt: complexitatea computațională a operației *non-dominated sort* și lipsa elitismului în selecția populației. Acest algoritm a fost prezentat de Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal și T. Meyarivan în anul 2002 [15].

Structura algoritmului NSGA-II este prezentată în [Figura 6.2](#Figura62). Cum se poate observa operațiile principale ale acestui algoritm sunt: sortare de tip *non-dominated* și sortarea de tip *crowding distance*. Vom prezenta pe scurt cele două operații și importanța acestora în funcționarea algoritmului.

### **6.2.1 Sortarea de tip non-dominated**

Una din principalele probleme ale predecesorului acestui algoritm a fost complexitatea operației de sortare care a fost simplificată în cazul algoritmului NSGA-II.

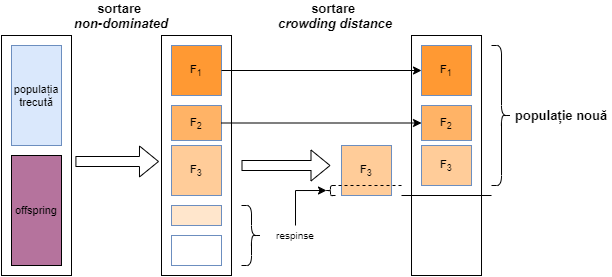
Algoritmul de sortare non-dominated presupune clasificarea soluțiilor pe fronturi. Primul front va conține soluțiile ce nu sunt dominate de o altă soluție, frontul al doilea cele ce sunt dominate de o soluție, etc.

Notăm o soluție cu valorile funcțiilor pentru cele obiective (). O soluție este dominată de o altă soluție dacă soluția este cel puțin la fel de bună pentru toate cele k obiective, dar mai bună decât pentru unul din obiective. Această proprietate este reluată în [Formula 6.1](#Formula61).



**Formula 6.1**: Soluție dominată și dominantă

În continuare este prezentat algoritmul non-dominated sort. Pentru fiecare soluție calculăm două proprietăți diferite: , numărul de soluții care domină soluția curentă, și , setul de soluții care sunt dominate. Toate soluțiile din primul front vor avea . Pentru fiecare soluție pentru care , vizităm fiecare soluție și reducem cu unu. Dacă în timpul iterației, una dintre soluții va avea , soluția va fi adăugată unui nou set U, acesta va fi cel de al doilea front. Procedura de mai sus este repetată cu fiecare membru a mulțimii U și este identificat frontul trei. Acest proces continuă până când nu sunt identificate toate fronturile. Pseudocodul acestui algoritm este prezentat în [Figura 6.3](#Figura63).



**Figura 6.2**: Schița algoritmului NSGA-II

|  |
| --- |
| for each p:P      for each q:P  if then     else if then    if then     i=1 while     for each   for each     if then    i++ |

**Figura 6.3**: Pseudocodul algoritmului noon-dominated sort

### **6.2.2 Sortarea de tip crowding distance**

Am menționat anterior că, împreună cu convergența către un set Pareto optimal, se dorește, de asemenea, ca un algoritm evolutiv multi-obiectiv să mențină o bună răspândire a soluțiilor în setul obținut. NSGA original utilizează funcția de partajare, care s-a dovedit că menține sustenabil diversitatea într-o populație, cu setarea adecvată a parametrilor asociați [16]. Atribuirea și sortare de tip crowding distance nu utilizeză parametrii suplimentari și este mai eficientă din punct de vedere computațional.

Pentru a obține o estimare a densității soluțiilor care înconjoară o anumită soluție în populație, calculăm distanța medie a două puncte de fiecare parte a acestui punct, de-a lungul fiecăruia dintre obiective. Această cantitate servește ca o estimare a perimetrului cuboidului format prin utilizarea celor mai apropiați vecini, numim această distanță: *distanța de aglomerare* [15]. Pseudocodul atribuirii aceste distanțe este prezentat în [Figura 6.4](#Figura64).

După ce tuturor membrilor populației le este atribuită această metrică de distanță, putem compara două soluții în funcție de gradul lor de proximitate cu alte soluții. O soluție cu o valoare mai mică a acestei măsuri de distanță este, într-un anumit sens, mai "aglomerată" de alte soluții [15].

|  |
| --- |
| l:=|I|  for each i: I[i].distance=0 for each obiectiv m:  I:=sort(I,m)  I[1].distance:= ∞  I[l].distance:= ∞  for i=2…(l-1): |

I[i].m se referă la valoarea funcției obiectiv m a individului i din Populația I, iar parametrii sunt valorile maxime și minime ale funcției m.

**Figura 6.4:** Procesul de atribuire crowding distance

# **Capitolul 7 – Algoritm genetic multi-obiectiv cu selecție bazată pe *novelty***

## **7.1 Prezentare Generală**

Prezentarea unei soluții diferite a fost necesară pentru a înțelege eficiența algoritmului propus de mine în comparație cu una dintre cele mai bune soluții, bazată pe algoritmi tot pe algoritmi evolutivi. Soluția pe care o propun este un algoritm genetic cu obiective *composite* și selecție în funcție de *noutate*. Algoritmul, propus inițial de Hormoz Shahrzad, Daniel Fink și Risto Miikkulainen [17], a obținut rezultate promițătoare chiar în prima lui versiune, de la care am plecat și, utilizând cele mai bune euristici(analizate de Lee Graham, Hassan Masum și Franz Oppacher [11]) precum și diverși operatori genetici specifici, am reușit să îmbunătățesc rezultatele algoritmului și să obțin soluții minime noi, ne-explorate de primul algoritm. Schița acestui algoritm este prezentată în [Figura 7.1](#Figura71). În continuare o să fie detaliate toate componentele acestui algoritm.

|  |
| --- |
| t:= 0 Creează populație inițială Pt Ct := ∅ (populația descendenților) while t < generații :  Rt:= Pt ∪ Ct  F := non-dominated sort(Rt) F=(F1,F2,…) fronturi nedominate  i:=0,Pt+1:= ∅  while |Pt+1| + |Fi| < dimensiunea populației:  Fi := crowding distance(Fi)  Pt+1:= Pt+1 ∪ Fi  i+=1  sort(Fi) utilizând crowding distance  Pt+1:= Pt+1 ∪ Fi[1:(dimensiunea populației-|Pt+1|)]  Ct+1:= creează populație nouă (crossover,mutație,selecție utilizând novelty)  t+=1 |

**Figura 7.1**: Schița algoritmului genetic multi-obiectiv cu novelty

## **7.2 Codificare**

Pentru acest algoritm am utilizat o reprezentare mai simplă[(Figura 7.2](#Figura72)), ușor de manevrat. În această reprezentare fiecare rețea de sortare este văzută ca un șir de perechi de numere, fiecare pereche de numere reprezintă un comparator. Ordinea perechilor în șir este și ordinea comparatorilor în rețeaua de sortare.



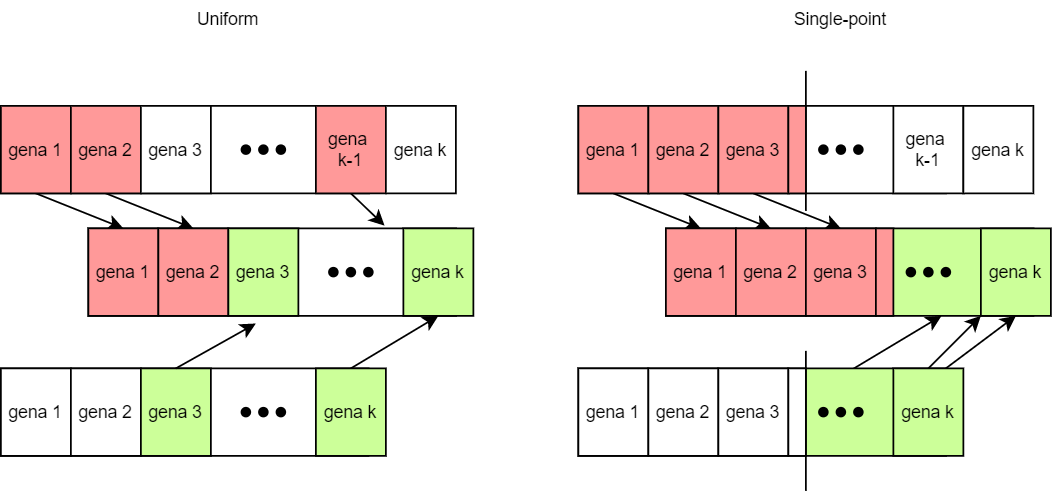
**Figura 7.2**: Codificarea unei rețele de sortare

## **7.3 Inițializarea populației**

Populația este formată din 1000 de indivizi care sunt generați aleatoriu, dar care au ca prefix filtrul Green definit anterior.

## **7.4 Crossover**

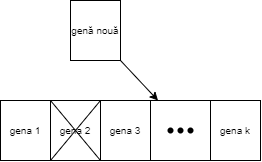
Doi operatori de crossover sunt puși la dispoziția de algoritmul genetic, crossover uniform, la nivelul comparatorilor individuali, și single-point crossover. Pentru fiecare din ei necesităm câte un parametru. Primul este parametrul obișnuit de probabilitate de încrucișare; al doilea determină proporția crossover-ului uniform față de cel single-point. Utilizând lucrarea "Statistical Analysis of Heuristics for Evolving Sorting Networks" am setat probabilitatea de crossover la 10% și proporția dintre cele două metode cu 30%-70% [11]. Operatorii de încrucișare sunt prezentați în [Figura 7.3](#Figura73).



**Figura 7.3**: Operatorii de încrucișare

## **7.5 Mutație**

Operatorul de mutație utilizat este cel *indirect replacement* utilizat și în primul algoritm. Acesta este adaptat codificării curente. Astfel, cu o probabilitate de 40% [11], vom șterge aleatoriu un comparator(o genă), vom crea un nou comparator care o să fie introdus pe o poziție aleatorie a cromozomului. Putem observa funcționarea acestui operator în [Figura 7.4](#Figura74).



**Figura 7.4**: Mutație indirect replacement

## **7.6 Funcțiile Obiectiv**

După cum a fost prezentat, cea mai importantă componentă a acestui algoritm este mulțimea funcțiilor obiectiv utilizate. Am decis utilizarea a trei obiective diferite și numărul de intrări nesortate(*b*), numărul de comparatori(*c*) și numărul de straturi paralele(*l*) ca parametrii acestor trei funcții.

Aceste obiective sunt prezentate în Formula [7.1](#Formula71),[7.2](#Formula72) și [7.3](#Formula73).

**Formula 7.1**: Prima funcție obiectiv

Ie obiectiv

**Formula 7.2**: A doua funcție obiectiv

Ie obiectiv

**Formula 7.3**: Ultima funcție obiectiv

Ie obiectiv

Prima dintre aceste funcții este preluată din cea mai bună implementare *single-objective,* este utilizată precum un filtru de greșeli: va împiedica procesul evolutiv să creeze rețele invalide, dar și rețele valide cu un număr mare de straturi și comparatori.

Al doilea obiectiv este cel mai important pentru problema pe care încercăm să o rezolvăm, acesta va încuraja găsirea unei soluții cu un număr minim de straturi paralele. Numărul de intrări sortate incorect este introdus pentru a evita rețele invalide cu un număr minim de straturi. Similar, ultima funcție va împiedica algoritmul să găsească soluții cu un număr prea mare de comparatori(importanța acestei funcții este mai mică față de funcția a doua).

## **7.7 Selecția cu “novelty”**

Selecția pe care o utilizează algoritmul este metoda turnirului cu două elemente, pornind cu un selection pool de 500 de elemente (acesta o să fie extins pentru soluții “noi”). Această metodă simplă a adus rezultate promițătoare în lucrarea [11]. Pentru a îmbunătății metoda de selecție am utilizat o măsură importantă în evoluția populație: cât de „nouă” este soluția față de restul populației. Numim această măsură: *gradul de novelty.*

Pentru a măsura cât de noi sunt soluțiile, este necesar mai întâi să le putem caracteriza comportamentul. Deși există multe modalități de a face acest lucru, un mod concis și eficient, din punct de vedere al complexității, este de a număra câte interschimbări au avut loc pe fiecare fir în sortarea tuturor intrărilor binare posibile în timpul aplicării rețelei [17].

Intuiția în spatele acestei alegere este simplă, două rețele pot să aibă o structură similară dar un comportament diferit în funcție de modul în care sortează, corect sau incorect, o aceeași intrare. Notăm cu numărul de interschimbări ale unei rețele .

Având în vedere această definire a comportamentului unei rețele de sortare, gradul de novelty al unui individ este calculat prin suma tuturor distanțelor (de comportament), între el și oricare alt individ din grupul de selecție. Această formulă este prezentată în [Formula 7.4](#Formula74).

Grupul de selecție este apoi extins: se vor adăuga încă 50 de indivizi(10% din dimensiunea grupului de selecție). Acest selection pool este apoi sortat în funcție de noutatea indivizilor, iar partea superioară, care reprezintă grupul de selecție inițial, este utilizată pentru selecție. În acest fel, soluțiile bune care sunt mai noi sunt incluse în turnir.

**Formula 7.4**: Gradul de noutate

Una dintre problemele acestei metode este că un grup de soluții depărtate de restul poate să primească grad novelty ridicat și introdus în selecție, în timp ce doar una dintre ele este necesară(destul de bună). Prin urmare, după selectarea porțiunii superioare a grupului de selecție, restul soluțiilor sortate se adaugă la grupul de selecție unul câte unul, înlocuind soluția cu cel mai mic grad de novelty([Formula 7.5](#Formula75)).

**Formula 7.5**: Soluția cu gradul de noutate minim

# **Capitolul 8 – Rezultate**

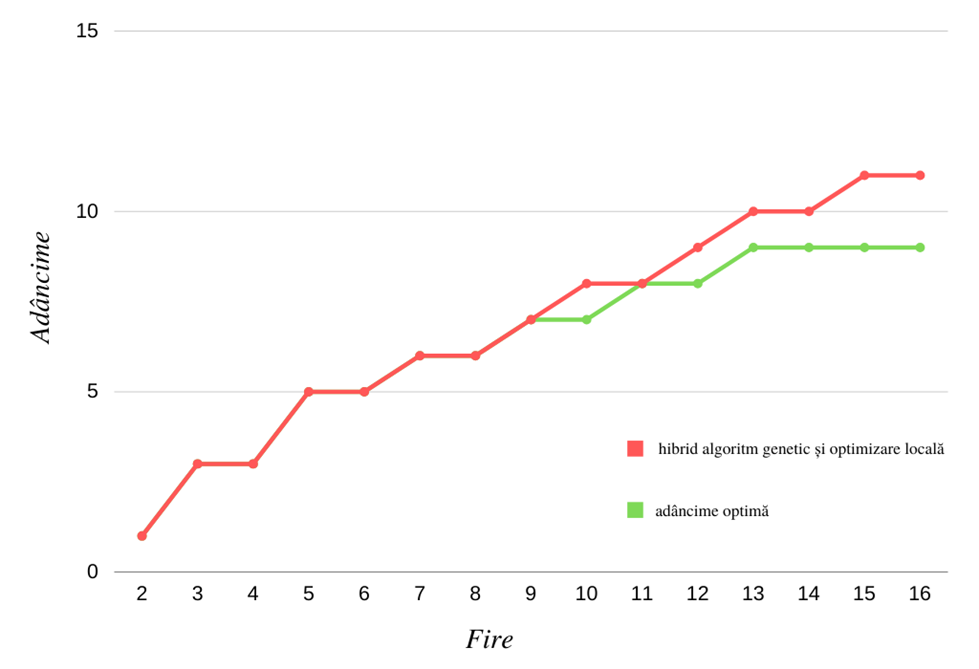
În acest capitol sunt prezentate rezultatele pentru cei doi algoritmi prezentați anterior în capitolele cinci și șapte. Experimentele au fost executate pe un Intel Core i7-8750h 2.20ghz. Pentru fiecare experiment am efectuat treizeci de rulări a algoritmului cu aceleași setări.

## **8.1 Rezultate algoritm genetic hibrid cu optimizare locală**

Cele mai bune rezultate ale acestui algoritm, în comparație cu cele mai bune rezultate cunoscute, sunt prezentate în [Figura 8.1](#Figura81). Putem observa că algoritmul obține soluții optime pentru majoritatea intrărilor, dar cu cât dimensiunea problemei crește, cu atât ne depărtăm de adâncimea minimă.

Comparând rezultatele acestui algoritm cu două dintre cele mai bune soluții pentru acest algoritm putem observa două rezultate importante.

În comparație cu algoritmul prezentat de Sung-Soon Choi *și* Byung-Ro Moon [18], algoritmul prezentat în capitolul 5 obține soluții similare, ușor de intuit deoarece este unul din principalele surse de inspirație de la care am pornit, și rezultate mai bune pentru rețele de sortare de dimensiune mai mare. Dar timpul de rulare este semnificativ mai mare decât cel al soluției propuse de Sung-Soon Choi și Byung-Ro Moon.



**Figura 8.1**: Rezultate algoritm hibrid

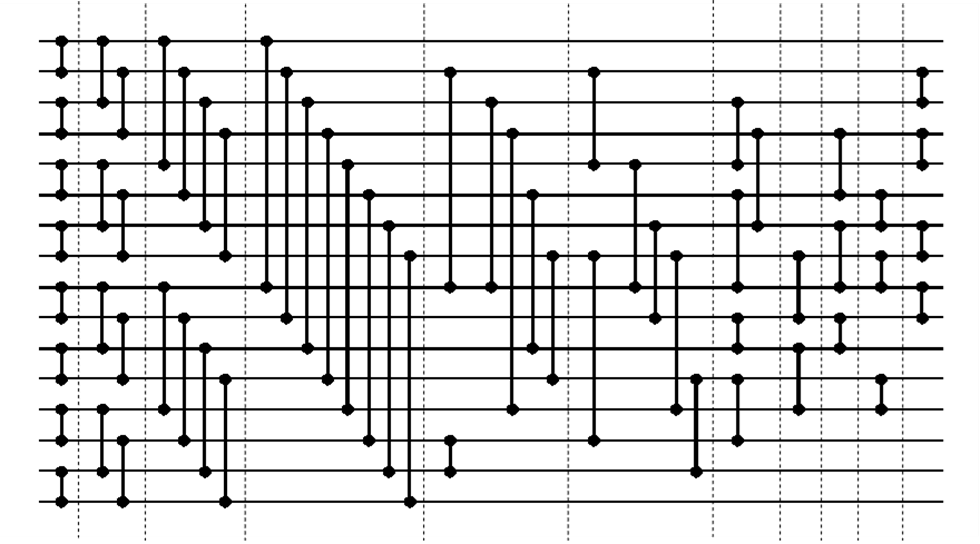
În comparație cu algoritmul prezentat de Hormoz Shahrzad, Daniel Fink și Risto Miikkulainen [17], soluția propusă obține rezulate mai bune iar timpii de rulare sunt mai mici.

Deorece această soluție este semnificativ mai rapid decât cel prezentat în capitolul șapte, am reușit rularea unor experimente pentru rețele de sortare cu mai mult de 16 fire. Aceste rezultate sunt prezentate în [Tabelul 8.1](#Tabel81), în comparție cu limitele superioare minime cunoscute până acum.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| Adâncime, algoritm prezentat | 12 | 12 | 13 | 13 | 14 | 13 | 14 | 15 |
| Adâncime optimă curentă | 10 | 11 | 11 | 11 | 12 | 12 | 12 | 12 |

**Tabel 8.1**: Rezultate pentru rețele mai mari

Una din realizările cele mai mari ale acestui algoritm a fost găsirea unei rețele de sortare de 16 fire, de dimensiune 61 și adâncime 11([Figura 8.2](#Figura82)). Acest rezultat este remarcabil deoarece este unul din cele mai bune rezultate obținute care să optimizeze și dimensiunea și adâncimea rețelei. Optimizarea rețelei pentru a obține rezultate minime pentru ambele măsuri a fost posibilă utilizând operatorul [*Redundancy Editor*](#_5.10.1_Edit), care elimină din rețea toți comparatorii redundanți.

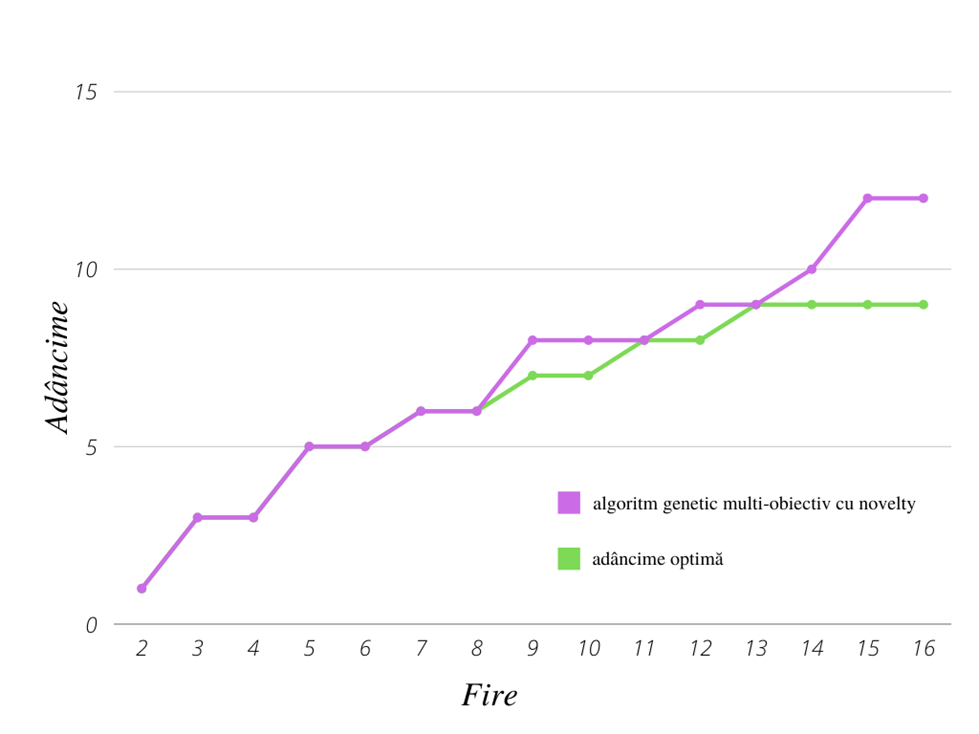


**Figura 8.2**: Rețea de sortare cu 16 fire

## **8.2 Rezultate algoritm genetic multi-obiectiv cu novelty**

În continuare sunt prezentate rezultatele pentru algoritmul genetic multi-obiectiv ce utilizează gradul de noutate. După cum se poate observa în [Figura 8.3](#Figura83), soluțiile sunt apropiate pentru rețelele de sortare de dimensiune mică, dar diferența devine tot mai mare cu cât rețeaua crește.

Una din cele mai mari probleme ale acestui algoritm este timpul de execuție semnificativ mai mare decât toate soluțiile considerate optime. Din această cauza și deoarece diferența dintre adâncimea optimă și adâncimea minimă determinată de algoritm era prea mare pentru intrări mari, am considerat că rularea algoritmului pentru rețele cu un număr de fire este intractabilă pentru acest algoritm.

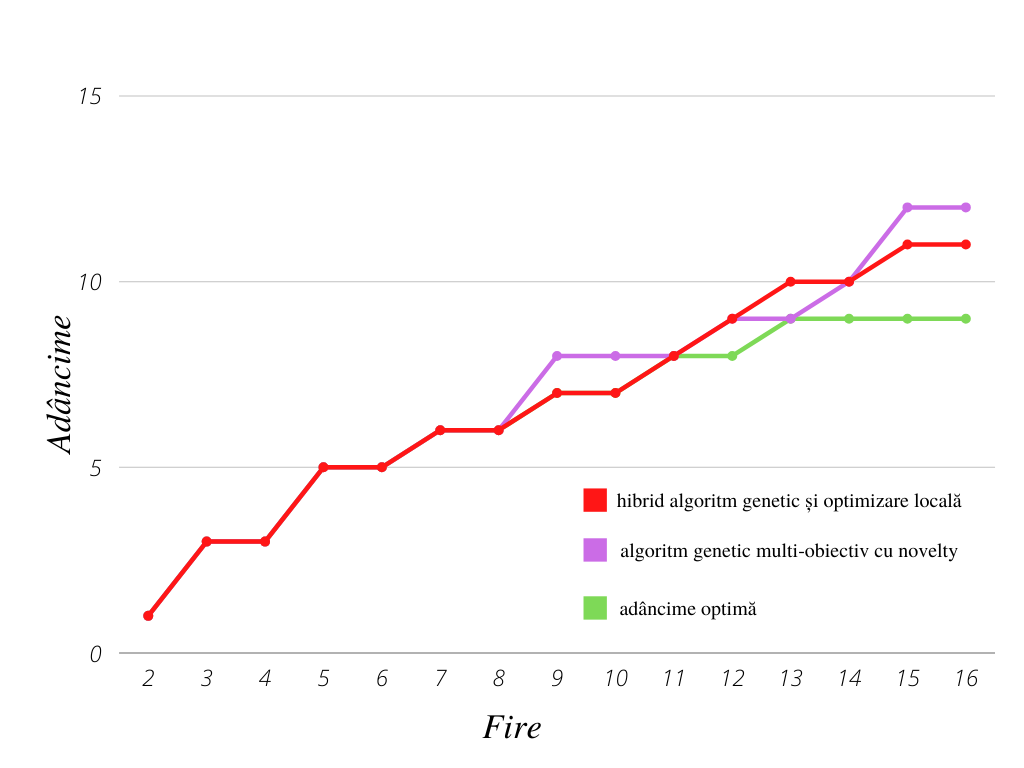


**Figura 8.3**: Rezultate algoritm genetic multi-obiectiv

## **8.3 Comparație algoritmi**

În [Figura 8.4](#Figura84) se pot observa diferențele dintre cei doi algoritmi: algoritmul genetic hibrid ce utilizează metode de optimizare locală găsește soluții mai bune decât algoritmul genetic multi-obiectiv, există însă și unele excepții precum rețeaua cu 14 fire pentru care primul dintre algoritmi nu reușește să atingă adâncimea optimă, care este însă atinsă de cel de al doilea.

Una din cele mai mari diferențe dintre cei doi algoritmi este timpul de execuție, semnificativ mai mare pentru algoritmul genetic multi-obiectiv. Pentru rețeaua de 16 fire, algoritmul multi-obiectiv necesită un timp de execuție de trei ori mai mare decât cel al algoritmului prezentat în capitolul 5.



**Figura 8.4**: Rezultate comparație soluții

# **Capitolul 9 - Concluzie**

# **Bibliografie**

|  |  |
| --- | --- |
| [1] | D. E. Knuth, în *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*, Redwood City, CA, USA, Addison Wesley Longman Publishing, 1998. |
| [2] | T. P. a. P. Steinke, „Pblib – a library for encoding pseudo-boolean constraints into cnf,” în *Theory and Applications of Satisfiability Testing – SAT 2015, ser. Lecture Notes in Computer Science, M. Heule and S. Weave*, vol. 9340, Eds. Springer International Publishing, 2015, pp. 9-16. |
| [3] | K. E. Batcher, „Sorting networks and their applications,” în *AFIPS Spring Joint Computer Conference*, 1968. |
| [4] | T. Haslop, „Minimal Depth Sorting Networks,” March 2020. [Interactiv]. Available: https://www.szi.uni-bremen.de/wp-content/uploads/2020/03/thesis\_compressed.pdf. |
| [5] | I. Parberry, „On the Computational Complexity of Optimal Sorting Network,” 1991. [Interactiv]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.9568. |
| [6] | M. Budiu, „Maşini alternante, ierarhia polinomială,” 16 noiembrie 1999. [Interactiv]. Available: https://www.cs.cmu.edu/~mihaib/articole/complex/complex-html.html#SECTION03070000000000000000. |
| [7] | I. Parberry, A Computer Assisted Optimal Depth Lower Bound for Nine-Input Sorting Networks, 1991. |
| [8] | D. Bundala și J. Závodný, Optimal Sorting Networks. Language and Automata Theory and Applications. Lecture Notes in Computer Science, 2014. |
| [9] | M. Gîrdea, „https://profs.info.uaic.ro/~marta/ga/L3/,” [Interactiv]. |
| [10] | B. Mihaela, „https://profs.info.uaic.ro/~pmihaela/GA/laborator3.html,” [Interactiv]. |
| [11] | L. Graham, H. Masum și F. Oppacher, Statistical Analysis of Heuristics for Evolving Sorting. |
| [12] | C. Frasinaru și M. Răschip, „Greedy Best-First Search for the Optimal-Size Sorting Network,” în *23rd International Conference on Knowledge-Based and Intelligent Information & Engineering*, 2019. |
| [13] | G. G. Simpson, „The Baldwin Effect,” în *Evolution 7*, 1953, pp. 110-117. |
| [14] | T. N. Bui și B. R. Moon, „Genetic algorithm and graph partitioning,” în *IEEE Trans. on Computers*, 1996, pp. 841-855. |
| [15] | D. Kalyanmoy, P. Amrit, A. Sameer și M. T., A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II, 2002. |
| [16] | F.-M. Apipie, Modelarea și predicția piețelor financiare utilizând tehnici de inteligență computațională, 2019. |
| [17] | H. Shahrzad, D. Fink și R. Miikkulainen, Enhanced Optimization with Composite Objectives and Novelty Selection. |
| [18] | S.-S. Choi și B.-R. Moon, A Hybrid Genetic Search for the Sorting Network Problem with Evolving Parallel Layers. |