# Problema Damelor

Bușecan Andrei, Sălnicean Răzvan, Terpe Victor, Titea Ruben

Ianuarie 2025

## Contents



# 1 Introducere

Echipa noastră a ales această temă deorece am mai întâlnit-o și anul trecut la cursul de proiectare al algoritmilor, ni s-a părut o temă interesantă și am dorit să vedem cum poate fi rezolvată prin altă metodă.

Problema damelor este o problemă care este folosiă în cea mai mare măsură de către profesorii din licee sau universități pentru a-i învăța pe elevi și studenți programare și algoritmi.

Enunțul problemei este unul care este foarte ușor de înțeles, și anume trebuie așezate un număr de n dame pe o tablă de șah care are dimensiuni de NxN în

așa mod încât aceste dame să nu se atace între ele. Se consideră că o soluție a fost găsită când nu există două sau mai multe regine pe același rand, coloană sau diagonală.

Problema are mai multe modalități prin care poate să fie rezolvată, printre care amintim metoda backtracking care este cel mai des întâlnită pentru rezolvarea acestei probleme, metoda greedy, programarea dinamică, algoritmi genetici etc.

Pe parcursul acestui proiect, vom rezolva această problemă utilizând un algoritm genetic.

Algoritmii genetici sunt o categorie de algoritmi care se bazează pe teoria evoluției propusă de Charles Darwin. Sunt folosiți pentru a rezolva și a imbunatăți problemele complexe. Ei simulează procesul de selecție naturală prin care cei mai adaptați indivizi dintr-o populație sunt selectați pentru a-și transmite genele generației următoare.

# 2 Cerințe

Prin implementarea acestui algorim dorim să obținem cel puțin o soluție care rezolvă problema pentru o tablă de dimensiune N.

**Generarea populației inițiale**

- Algoritmul trebuie să genereze o populație inițială cu un anumit număr de indivizi.
- Indivizii vor reprezenta o posibilă soluție și nu vor exista doua regine pe o linie sau pe o coloană.

**Funcția de fitness**

- Pentru fiecare individ generat va fi calculată funcția de fitness pentru a arăta cât de bună este soluția sa.
- Fiind o problemă de minim, un fitness mai mic va indica o soluție mai bună.

**Selecția indivizilor**

- Algoritmul trebuie să aibă o metodă de selecție pentru a alege indivizii pentru reproducere.

**Crossover sau Recombinare**

- Utilizând indivizii selectați se va realizea recombinarea acestora pentru a obține doi copii.
- Dacă indivizii obținuți în urma recombinării nu respectă condițiile impuse la început, și anume daca vor exista 2 regine pe aceeași linie sau coloană, cromozomii vor fi corectați.

**Mutația**

- Se va efectua mutație în mod aleator asupra indivizilor pentru a menține diversitatea și pentru a ieși din posibilele minime locale.

**Formarea următoarei generații**

- Când se trece la o nouă generație indivizii care au fost selectați pentru crossover(părinții) vor fi înlocuiți de copii.

# 3 Aspecte teoretice și State of the Art

După cum am menționat în Introducere, algoritmii genetici sunt inspirați din evoluție, iar în această secțiune dorim să analizăm mai în detaliu acest lucru.

În natură există mai multe colective de indivizi care fac parte din aceeași specie. Deși fac parte aceleași categorii, fiecare individ are caracteristici și trăsături diferite, pe care le-a primit în urma intervenției unor factor externi. Acești factori ajută la evoluția populației pentru a se putea adapta unor noi condiții de supraviețuire. Desigur, aceste modificări ale indivizilor pot să ducă la o schimbare în bine, o schimbare în rău sau una neutră. În cadrul populației indivizii concurează între ei pentru resursele indispensabile cum ar fi hrana și adăpostul, și de asemenea atragerea altor indivizi pentru a se reproduce. [1] Din aceast motiv, indivizii care au trăsături superioare celorlalți, de cele mai multe ori, au mai multe șanse să supraviețuiască și să se înmulțească.

Fiecare inivid își transmite trăsăturile generației următoare, iar în fiecare generație acest proces se repetă ajungându-se de-a lungul mai multor generații ca trăsăturile benefice să fie cumulate și evidențiate din ce în ce mai mult în copii. [2] În momentul transmiterii trăsăturilor sau genelor, noul individ format poate să sufere mutații, adică una sau mai multe trăsături obținute de la părinți să fie alterate, fapt care se datorează mai multor factori, cum ar fi erori in replicarea ADN-ului, radiații, viruși etc. Toate acestea contribuie la evoluția unei populații.

În cele de mai sus am explicat ce înseamnă evoluția, iar acum vom arăta cum simulează algoritmii genetici procesul evolutiv.

În algoritmii genetici, un individ se mai numește și cromozom, acesta reprezentând o posibilă soluție pentru problema care se dorește a fi rezolvată. Inivizii au nevoie de o anumită structură pentru a putea fi interpretați și folosiți de calculator. În funcție de natura problemei, individul este reprezentat în mod diferit. Câteva reprezentări ale inidvizilor sunt: șirurile de biți(0 sau 1), tablouri de numere întregi sau numere reale, arbori, grafuri etc.

Algoritmii genetici încep prin generearea unei mulțimi de indivizi în mod random, care este denumită populație inițială. Variațiile dintre indivizii obținuți fac ca unii să fie mai buni(au o mai bună soluție pentru problemă) decât alții. [3] Pentru a verifica cât de adaptat este individul, adică cât de bună este soluția sa, se folosește o funcție numită funcție de fitness. Acestă funție returnează o valoare calculată pe baza tuturor informațiilor care sunt în cromozom.

Din populația inițială se selectează invizi pentru a se recombina. Selecția poate fi realizată prin mai multe moduri:

**Selecția Random:** din populație se aleg în mod aleator un număr dorit de indivizi.

**Selecția Turnir:** pentru această selecție se specifică un număr N de indivizi care să participe în confruntare. Cei N indivizi sunt aleși în mod random din populație și comparați între ei, câștigător fiind cel care are fitness-ul cel mai bun.

**Roata Norocului:** Se calculează suma fitness-ului tuturor indivizilor din populație, iar fiecare individ primește o probabilitate de a fi selectat proporțională cu fitness-ul său (se împărțește fitness-ul individului la fitness-ul total al populației). Roata este împărțită în segmente astfel înât indivdul cu probabilitate mai mare va avea o bucată mai mare. Se alege un număr random între 0 și 1(se învârte roata) iar individul căruia îi corespunde segmentul pe care cade numărul este selectat.

Pe lângă acestea trei, mai sunt și alte modalități de selecție: elitism, selecție bazată pe rang, selecție prin trunchiere etc. [4]

Următoarea etapă într-un algoritm genetic este recombinarea sau crossover-ul. Această operație presupune combinarea informațiilor genetice ale părinților pentru a forma noi indivizi care pot să aibă o soluție mult mai bună. Există două tipuri principale de crossover: crossover cu un singur punct de tăiere și crossover cu cu două sau mai multe puncte de tăiere. Primul tip funcționează prin alegerea în mod random a unui punct din cromozomii părinților selectați pentru reproducere, iar genele de la aces punct până la finalul cromozomului sunt interschimbate între cei doi părinți și rezultând astfel 2 cromozomi noi, adică copii [5]. Cel de-al doilea tip implică selectarea a 2 sau mai multe puncte de tăiere din cromozomii părinților, și la fel se interschimbă genele cuprinse între aceste puncte rezultând noi indivizi.

Mutația este un operator genetic care modifică genele indivizilor din populație pentru a introduce diversitate genetică. Această modificare nu este făcută pe toți indivizii, ci doar pe cei aleși în mod aleatoriu. Mutația este folosită pentru a se preveni convergența prematură a populației către soluții suboptimale sau locale [6], explorându-se mai eficient spațiul soluțiilor. Este nevoie de diversitate genetică pentru ca să se poată găsi soluții mai bune.

Acest proces se repetă până când o soluție este găsită sau condiția de oprire impusă este satisfăcută. Mai jos este o imagine care reprezintă acest proces.

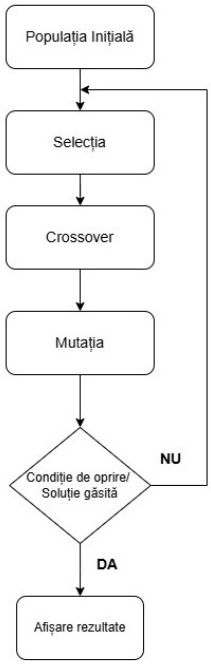


Figure 1: Procesul Algoritmului Genetic.

În ultima perioadă au fost aduse îmbunătățiri algoritmilor genetici, pentru a-i face mai eficienți și pentru a-i putea aplica în diferite domenii.

**Integrarea cu învățarea automată**

- Algoritmii genetici au fost combinați cu învățarea automată pentru a se obține niște modele hibride prin care să se optimizeze performanțele calculatoarelor. Cercetările făcute au demonstrat că rețelele evolutive de porți logice probabilistice și deterministe numite Markov Brains pot să se adapteze de la o generație la alta, fiind îmbunătățită capacitatea lor de a învăța singure pe durata vieții. [7]

**Algoritmi genetici multi-obiectiv (MOGA)**

- NSGA-II(Nondominated Sorting Genetic Algorithm II) este un algoritm folosit pentru a găsi soluții optime pentru mai multe obiective diferite în același timp, cum ar fi costul și performanța. Cercetătorii au adus îmbunătățiri acestui algoritm pentru descoperirea biomarkerilor. Testele au fost făcute pe date genetice ale unor pacienți cu cancer la sân, iar rezultatele obținute arată că metodele folosite au o acuratețe foarte mare. [8]

Aplicații ale algoritmilor genetici:
**Domeniul educațional**

- Un dezavantaj al sistemului educațional este că toți studenții sunt obligați să urmeaze aceeași metodă, deși vin din medii diferite, nu au toți același grad de cunoștințe, având preferințe și nevoi de învățare diferite. Așadar, metodele tradiționale de predare nu sunt potrivite pentru toți cursanții, iar ca încercare de rezolvarea a acestei probleme a fost propus un algoritm genetic pentru a construi căi de învățare personalizate pentru fiecare student. Rezultatele obținute în urma utilizării modelului cu algoritmi genetici arată că cea mai slabă cale de învățare obținută este mai bună decât metodele tradiționale. [9]

**Prelucearea imaginilor**

- O alta aplicație a algoritmilor genetici poate fi în domeniul prelucrarii imaginilor unde ne ajută în refacerea imaginii originale prin îndepărtarea zgomotelor(cum ar fi urmărirea imaginilor în mișcare). Se folosește imaginea ca un individ iar un pixel ca fiind un cromozon, în acest domeniu recombinarea se realizează interschimbând blocuri de NxM cromozomi. [10]

**Domeniul financiar**

- În domeniul pieței bursiere și cel financiar algoritmii genetici sunt folosiți pentru a aproxima cea mai eficientă valoare a unei anumite acțiuni în care o persoană ar dori sa investească. [11]

# 4 Implementare

Pentru implementarea algoritmului gentic care să rezolve problema damelor am folosit limbajul de programare Python deoarece am văzut că este folosit foarte mult în inteligență artificială și are o sintaxă foarte simplă care ne ajută să ne concentrăm mai mult pe logica algoritmului decât pe alte detalii de limbaj.

## 4.1 Reprezentarea individului

În cadrul algoritmului noi am reprezentat un individ printr-o clasă numită Individ care are ca atribut un tablou(listă) de dimensiune n numit cromozom, iar acesta corespunde cu dimensiunile unei table de șah. Pozițiile tabloului reprezintă coloanele tablei, iar valorile din tablou reprezintă liniile tablei. Asfel o poziție și valoarea care îi corespunde arată locul de pe tală în care există o regină. Având această reprezentarea nu vor exista două regine pe aceeași coloană, iar adăugând și condiția ca valorile să fie unice am făcut ca să nu existe două regine sau mai multe nici pe aceeași linie. De asemenea, clasa mai are un atribut numit fitness și o metodă pentru a se calcula fitness-ul fiecărui individ.

## 4.2 Funcția de fitness

La problema damelor am identificat că funcția de fitness trebuie să calculeze numărul de atacuri dintre cele n regine. Fiind o problemă de minim, un număr mai mic returnat în urma calucării fitness-ului va indica faptul că inividul respectiv este mai bun. Prin reprezentarea individului am redus verificările necesare pentru a calcula atacurile, deoarece pe linii și coloane e imposibil să existe mai mult de o regină. Așadar mai trebuie verificate atacurile pe diagonală, lucru pe care noi l-am făcut printr-o funcție în care iterăm cromzomul și verificăm dacă diferența absolută(pentru a include daigonala prinicipală și pe cea secundară) dintre linii și coloane e egală. Dacă acest lucru e adevărat atunci creștem numărul de atacuri, iar la finalul iterării returnăm numărul de atacuri găsit. Funcția aceasta este apelată din metoda indivului pentru a seta atributul fitness.

## 4.3 Generarea populației inițiale

Pentru a genera populația inițială de indivizi am folosit două funcții:

- **Generare_cromozomi():** Funcția ia 2 parametrii, unul cu dimensiunea tablei, iar celălalt pentru pentru a știi câți cromozomi să se genereze. Cromozomii generați vor fi distincți, iar aici am verificat condiția ca valorile acestora să fie distincte. La final se returnează cromozomii generați.
- **Generare_populație_inițială():** Aici am folosit funcția de generare a cromozomilor pentru a genera numărul dorit de cromozomi distincți, iar apoi pentru fiecare cromozom am creat obiecte de tip Individ, care sunt returnate la final într-o listă ce reprezintă populația inițială.

## 4.4 Selecția

Selecția indivizilor am făcut-o utilizând metoda turnirului. Funcția pentru selecția turnir(`selectie_turnir()`) ia trei parametrii: populația din care se selectează(*populatie*), numărul de indivizi care vor participa în confruntare(*dim_turnir*) și numărul de indivizi care se dorște să se obțină(*dim_populatie*). Sunt selectați *dim_turnir* indivizi în mod aleatoriu din *populatie*, iar cel care are cel mai mic fitness este ales câștigător și adăugat într-o lista care ține indivizii castigatori(*populatie_obtinuta*), care este returnată la final. Pentru a selecta părinții pentru recombinare am folosit o altă funcție(`selectare_părinți()`) care din populația dată selectează o pereche de 2 părinți folosind metoda turnir, am verificat și ca perechea să nu conțină același părinte de două ori. La final se returnează o listă de perechi de doi părinți.

## 4.5 Crossover

Această etapă a algoritmului cuprinde a fost implementată folositnd 4 funcții:

- **Crossover():** Această funcție ia cromozomii a doi părinți ca și parametrii, se alege ca punct de tăiere un număr cuprins între 0 și lungimea cromozomului unui părinte în mod random, apoi se crează doi copii: primul ia genele părintelui de la tăietură până la capăt și pe ale celui de-al doilea părinte de la început până la punctul de tăiere, iar al doilea copil este format din genele rămase de la ambii părinți. La final returnează cei doi cromozomi obținuți.
- **Există_duplicate():** Verifică dacă în cromozomul unui copil există două valori la fel.
- **Generare_copii():** În această funcție, pentru fiecare pereche de părinți este apelată funcția crossover, iar apoi se crează obiecte de tipul Individ pentru cromozomii returnați și obiectele sunt puse într-o listă care e returnată la sfârșit.

**Corectează_cromozom():** Dacă cromozomul copilului are duplicate, atunci se apelează această funcție având cromozomul ca parametru pentru a-l corecta.

```
n = len(copil)
lipsa = []
for i in range(n):
    if i not in set(copil):
        lipsa.append(i)
```

Pentru a putea corecta cromozomul trebuie să aflăm care valori lipsesc din el. Așa că se itereză cromozomul și valorile care nu sunt prezente se adaugă în lista *lipsa*.

```
pozitii = {}
for pozitie, element in enumerate(copil):
    if element in pozitii:
        pozitii[element].add(pozitie)
    else:
        pozitii[element] = {pozitie}
```

Mai departe trebuie să știm care sunt pozițiile pe care un elemnt apare de mai multe ori. Am folosit un dicționar pentru a stoca pozițiile fiecărui element. Se iterează cromozomul și se verifică dacă elementul există în dicționar. Dacă elementul este deja prezent, se adaugă poziția curentă în setul de poziții care este asociat elementului. Dacă elementul nu există, se creează o nouă cheie în dicționar pentru acel element, iar valoarea asociată va fi poziția sa curentă. De exemplu, dacă avem cromozomul [1,2,2,3] atunci dicționarul va arăta asfel: `pozitii = { {1:{0}, 2:{1,2}, 3:{3} }`

```
    k=0
    for element, valoare in pozitii.items():

        if(len(valoare) > 1):
            i = random.choice(list(valoare))
            copil[i] = lipsa[k]
            k += 1
    return copil
```

În această parte, se caută care element se află pe mai multe poziții, iar pe una dintre pozițiile sale se pune un element care lipsește. Variabila k ține cont de poziția elementelor din lista *lipsa*. Iterăm dicționarul și verificăm dacă setul de poziții are o lungime mai mare decât unu. Dacă rezultatul verificării este adevărat, atunci se ia una dintre pozițiile din set și se înlocuiește elementul de pe acea poziție din cromozom cu o valore lipsă, iar k e incrementată. La final se returnează cromozomul corectat al copilului.

## 4.6 Mutația

Mutația în cadrul algoritmului este realizată prin interschimbarea a două gene din cromozom, alese în mod aleatoriu. Implementarea am făcut-o prin alte două funcții:

- **Mutație_Cromozom():** Ia un individ ca parametru, extrage cormozomul din acesta, apoi generează două numere random distincte care reprezintă două poziții. Elementele care se află în cromozom pe aceste poziții sunt interschimbate, iar la final se actualizează individul cu noul cromozom obținut.

- **Mutație():** Această funcție ia ca parametrii populația și o rată de mutație care reprezintă probabilitatea ca un individ să fie mutat. Se interează lista populației și pentru fiecare individ se generează un numar random între 0 și 1, iar dacă acest număr este mai mic decât rata de mutația atunci individul este mutat prin apelarea funcției Mutație Cromozom(). După muție, fitness-ul individului este recalculat deoarece a fost schimbată structura cromzomului, iar acum reprezintă o altă solție, care poate să fie mai bună sau mai slabă.

**Alte funcții folosite:**

- **Verifică_Soluții():** Verifică dacă într-o generație există soluții și dacă sunt le returnează.

- **Există Duplicate Copil():** Această funcție am folosit-o pentru a testa dacă toți copii generați au cromozomi cu valori unice.

- **Factorial():** Pentru a calcula factorialul numărului care reperzintă dimensiunea tablei.

## 4.7 Punctul de pornire al algoritmului

Punctul de începere al algoritmului este funcția `start()`. Aici sunt inițializate variabilele necesare:

- `nr_indivizi:` numărul de indivizi pentru populația inițială.
- `n:` dimensiunea tablei.
- `max_inidvizi:` numărul maxim de indivizi distincți ce poate fi generat; este obținut prin apelarea funcției factorial().
- `rata_mutatie:` procentajul indivizilor care vor fi mutați.
- `dim_turnir:` cati indivizi vor participa în selecția turnir.
- `dim_populatie:` numarul de indivizi care sunt selectați din populația inițială pentru a forma noua populație; noi am pus ca acesta să fie jumătate din `nr_indivizi`.
- `generatia_maxima:` numărul de generații pentru care algoritmul va fi rulat.
- `generatie:` contor care tine evidenta generațiilor.
- `cel_mai_bun_individ:` aici va fi stocat cel mai bun individ generat de-a lungul algoritmului, pentru a-l putea afișa la final dacă nu a fost găsită nicio soluție după maximul de generații care a fost impus.
- `populatie_initiala:` stochează lista de indivizi generați în urma apelării funcției `generare_populație_inițială()`.
- `populatie:` lista de indivizi selectați prin metoda turnir din `populatie_initiala`.

Funcția `start()` verifică ca `nr_indivizi` să fie mai mic decât `max_inidvizi`, iar dacă acest lucru nu este respectat algoritmul nu va porni.

Am folosit o buclă while pentru a putea trece de la o generație la alta. Bucla se oprește dacă a fost găsită cel puțin o soluție sau dacă s-a ajuns la numărul maxim de generații. În buclă se selectează părinții din populație, din acești părinți se generează copii prin crossover, iar apoi se apelează funcția de mutație pentru copiii obținuți. După mutație se actualizează variabila `cel_mai_bun_individ` dacă există un individ printre copii care are un fitness mai bun decât individul care este deja în variabilă. Se incrementează contorul generațiilor, iar copii devin noua populație. După încheierea buclei, se stochează soluțiile găsiteîn urma apelării funcției verifica_solutii(): în variabila `solutii`.

Dacă există soluții, se afișează toate, iar dacă nu există, este afișată cea mai bună soluție găsită(`cel_mai_bun_individ`).

# 5 Testare

Pentru a testa funcționarea algoritmului am testat fiecare funcție în parte. După ce am implementat o funcție, am apelat-o cu parametrii care sunt necesari, iar apoi am afișat rezultatul pe care l-a returnat. Am comparat rezultatul obținut cu rezultatul care trebuia să fie returnat și asfel am putut să identficăm dacă există o eroare care trebuie corectată. De exemplu pentru corectarea cromozomului, am apelat funcția cu un cromozom care avea valori duplicate și am verificat ca acesta să fie corectat.

```
print(corecteaza_cromozom([1,2,2,3]) -> [1,2,0,3] sau [1,0,2,3]
```

Pentru a testa dacă funcțiile algoritmul funcționează corect împreună, am dat valori variabilelor din funcția start() apoi am rulat algoritmul. Așa am reușit să aflăm unele erori de logică pe care le-am avut. Am verificat soluțiile returnate de algoritm desenând pe hârtie tabela de șah și punând reginele pe pozițiile indicate. Deoarece această metodă a fost una destul de costisitoare, am configurat o interfață care să deseneze soluțiile pe tabelă la încheierea algoritmului genetic.

# 6 Rezultate

Rezultatele obținute în urma implementării algoritmului sunt următoarele:

### Rezultate în raport cu Cerințele:

Algoritmul generează o populație inițială în care indivizii nu au două sau mai multe regine pe linie sau pe coloană. Funcția de fitness a fost calculată pentru fiecare individ, și chiar recalculată după mutație. Fitness-ul a fost implementat astfel încât o valoare mai mică să indice o soluție mai bună. Pentru selecție a fost implementată metoda turnir. A fost realizat și crossover-ul pentru câte o pereche de doi părinți, iar dacă cromozomul copiilor are duplicate este corectat. Mutația a fost efectuată în mod aleatoriu, iar la trecerea la o nouă generație părinții sunt înlocuiți de copii. Deoarece timpul de așteptare pentru a obține cel puțin o soluție pentru dimensiunile mari ale tablei de șah este unul foarte mare am pus o condiție ca algoritmul să se oprească după un anumit număr de generații și să afișeze cea mai bună soluție gasită până în acel moment. Deci această cerință nu a fost satisfăcută în totalitate.

### Performanța algoritmului:

Timpul de execuție al algoritmului a variat în funcție de numărul de indivizi, rata de mutație și dimensiunea tablei de șah.

Pentru 1000 de indivizi, limita de 200 de generații și rata de mutație de 0.5 în 10 rulări am avut următoarele rezultate:

- **Tablă cu N=8:** S-a găsit o medie de 5.6 soluții într-un timp mediu de aproximativ 0.066 secunde. 9 rulări au găsit soluțiile în generația 1, iar 1 rulare în generația 2.
- **Tablă cu N=18:** S-a găsit o medie de 1 soluție într-un timp mediu de aproximativ 2.026 secunde. Generațiile în care au fost găsite soluțiile au fost variate, cea mai mică fiind generația 57, iar cea mai mare generația 127 .
- **Tablă cu N=26:** S-a găsit o medie de 0.9 soluții într-un timp mediu de aproximativ 5.318 secunde. Generațiile în care au fost găsite soluțiile au fost variate, cea mai mică fiind generația 67, cea mai mare generația 158, iar pentru o rulare nu a fost găsită o soluție în cele 200 de generații(cea mai bună soluție având fitness 1).

# 7 Conculzii

În cadrul acestui proiect, am implementat un algoritm genetic pentru a rezolva problema damelor, și am demostrat asfel că acestă metodă poate fi folosită pentru a găsi soluții pentru probleme complexe. Algoritmul a fost capabil să genereze soluții valide pentru mai multe dimensiuni ale tablei de șah, și a satisfăcut aproape toate cerințele din secțiunea *Cerințe*.

Interfața pe care am adăugat-o a fost un ajutor foarte mare pentru a putea vizualiza și verifica soluțiile generate.

Pentru dimensiuni mici algoritmul se descurcă foarte bine, găsind soluții într-un timp foarte scurt, însă pentru dimensiuni mai mari nu este la fel de bun, așa că ar putea să fie îmbunătățit.

Așadar, rezultatele pe care le-am obținut au arătat faptul că algoritmul are o capacitate bună de a explora spațiul soluțiilor și de a afla soluții foarte bune într-un timp destul de rezonabil.

# 8 Funcționarea ca echipă

Echipa noastră a fost formată din următorii membri: Bușecan Antonio Andrei, Sălnicean Răzvan, Terpe Victor Cristian și Titea Dan Ruben. Pentru acest proiect noi am lucrat mai mult la facultate în timpul laboratoarelor și în timpul liber dintre cursuri, așa că am lucrat împreună la cod, iar pe GitHub fiecare a încărcat partea la care a contribuit mai mult.

Ruben a creat repo-ul și a făcut partea de generare a populației inițiale. La începutul proiectului reprezentarea individului am făcut-o fără condiția ca elementele din cromozom să fie unice, așadar pentru funcția de fitness trebuia să se calculeze atacurile pe rânduri, lucru pe care Victor l-a implementat. Mai apoi am schimbat reprezentarea și contribuția lui Victor la funcția de fitness a fost ștearsă. Pentru calculul atacurilor pe diagonală, am luat mai mulți cromozomi pe care i-am desenat pe foaie și am încercat să ne dăm seama care ar fi formula. Aici Andrei a avut o contribuție mai mare deorece el a realizat că există o legătură între diferența dintre linii și coloane și atacuri. După mai mult timp în care ne-am gândit, am reușit să facem formula și să o implementăm în cod. Ulterior, Ruben a mai îmbunătățit-o. Răzvan a făcut funcția prin care se selectează părinții din populație. Apoi Andrei a făcut funcția pentru crossover în care a creat cei doi copii din părinți, iar Victor a implementat o funcție pentru a verifica dacă există duplicate într-un cromozom și a adăugat această verificare în funcția de crossover pentru a verifica cromozomii copiilor. Pe urmă Ruben a realizat funcția pentru a corecta un cromzom, și a adăugat-o în crossover pentru a corecta copii în caz că aceșia au duplicate. Răzvan a implementat funcția care generează copii folosind funcția crossover și formând noi obiecte de tip Individ din cromozomii obținuți. În final Ruben a adăugat funcțiile de factorial(pentru a verifica maximul de indivizi ce pot fi generați), exista_duplicate_copil(pentru a verifica toți copii din populație dacă au duplicate în cromozom), partea de mutație(funcțiile mutatie_cromzom si mutatie), funcția de start, selecția turnir și interfața(am luat-o de pe internet și am integrat-o în algoritmul nostru).

La documentație de asemenea, am lucrat toți. Pentru aceasta am vorbit pe Discord și am schimbat rolurile de editor din Overleaf pentru ca cine avea o idee să o poată scrie. Partea de Aspecte Teoretice a fost cea mai grea, deoarece a trebuit să căutăm și să citim foarte multe articole, iar de multe ori după ce citeam constatam că textul respectiv nu prea are legătură cu ceea ce doream să scriem noi.

În concluzie, în cadrul acestui proiect am învățat un concept nou, și anume algoritmii genetici, despre care nu am mai auzit niciodată înainte de a începe acest semestru.

# References


[1] SN Sivanandam, SN Deepa, SN Sivanandam, and SN Deepa. *Genetic algorithms*. Springer, 2008.

[2] T Ryan Gregory. Understanding natural selection: essential concepts and common misconceptions. *Evolution: Education and outreach*, 2:156–175, 2009.

[3] Stephanie Forrest. Genetic algorithms. *ACM computing surveys (CSUR)*, 28(1):77–80, 1996.

[4] Saneh Lata, Saneh Yadav, and Asha Sohal. Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering Science*, 07 2017.

[5] Kalyanmoy Deb. Genetic algorithm in search and optimization: the technique and applications. In *Proceedings of International Workshop on Soft Computing and Intelligent Systems,(ISI, Calcutta, India)*, pages 58–87. Proceedings of International Workshop on Soft Computing and Intelligent ..., 1998.

[6] Siew Mooi Lim, Abu Bakar Md Sultan, Md Nasir Sulaiman, Aida Mustapha, and Kuan Yew Leong. Crossover and mutation operators of genetic algorithms. *International journal of machine learning and computing*, 7(1):9–12, 2017.

[7] Leigh Sheneman and Arend Hintze. Evolving autonomous learning in cognitive networks. *Scientific reports*, 7(1):16712, 2017.

[8] Luca Cattelani and Vittorio Fortino. Improved nsga-ii algorithms for multi-objective biomarker discovery. *Bioinformatics*, 38(Supplement_2):ii20–ii26, 2022.

[9] Lumbardh Elshani and Krenare Pireva Nuçi. Constructing a personalized learning path using genetic algorithms approach. *arXiv preprint arXiv:2104.11276*, 2021.

[10] Vasile Buzuloiu and Constantin Vertan. Aplicaţii ale algoritmilor genetici în teoria transmisiunii informaţiei, prelucrarea imaginilor şi în ergonomie. *Revista Română de Informatică și Automatică*, 6(3):39–45, 1996.

[11] Li Lin, Longbing Cao, Jiaqi Wang, and Chengqi Zhang. The applications of genetic algorithms in stock market data mining optimisation. *Management Information Systems*, 2004.