Problema Canibalilor

Cătălina Racolța
Tuns Andrei
Sima Alin
Vancea Paul

Ianuarie 2025

Contents

1 Introducere, motivația alegerii temei

Această problemă și anume cea a canibalilor a fost aleasă în principiu de către Cătălina, deoarece este o fană a jocurilor, iar problema canibalilor poate fi implementată și sub formă de joc. Împreună cu colegii de proiect, am dorit să creăm o aplicație în Java care să o rezolve. În problemă este vorba despre 3 canibali și 3 misionari care se află pe un mal, aceștia având la dispoziție o barcă. Scopul jocului este de a trece toate persoanele pe malul opus, fără ca vreun misionar să fie mâncat de canibali. Acest lucru este posibil doar dacă numărul de misionari de pe un mal este mai mare sau egal decât cel al canibalilor.

Problema canibalilor este una de genetică, care poate fi implementată la un nivel mai complex dar și la un nivel mai superficial. Complexitatea problemei crește de îndată ce problema se abordează precum un algoritm genetic.

2 Context, cerințe, ce dorim să obținem

2.1 Context

O problemă de genetică are un rol evolutiv, având proprietatea de adaptabilitate. Aceasta va fi mereu în continuă dezvoltare, adaptându-se în mod iterativ până ajunge la o soluție suficient de bună din punct de vedere a timpului de rulare cât și a pașilor parcurși.

Algoritmii genetici copiază teoria evoluției unei specii pe parcursul mai multor generații. Această teorie are două proprietăți de bază:

1. Supraviețuirea unor indivizi bine adaptați mediului în care trăiesc având un avantaj față de cei care duc lipsa unor trăsături avantajoase evoluției.

2. Modificări genetice aleatorii care introduc noi caracteristici în populația care evoluează deja.

2.2 Cerințe

Pașii necesari pentru rezolvara unei probleme de genetică sunt:

1. **Generarea populației inițiale**

   Generarea indivizilor din populația inițială se poate face total aleator sau respectând anumite condiții date de cerințele problemei. În cazul problemei canibalilor, generarea se va face total aleator.

2. **Selecția individului**

   Selecția individului se poate face prin mai multe metode, cum ar fi: complet aleator, metoda turnirului, roata norocului, etc. În problema aleasă de noi, am folosit metoda turnirului. Pentru metoda turnirului se procedează astfel: se aleg aleator un număr de indivizi, din care se alege individul cu fitness-ul cel mai bun.

3. **Încrucișare**

   Încrucișarea se face pe baza celor doi părinți aleși după selecția individului, rezultând în cazul nostru doi copii, dar în alte situați poate rezulta un singur copil sau chiar mai mulți. Există mai multe tipuri de încrucișare, cea mai folosită fiind metoda cu un singur punct de tăiere pe care am folosit-o și noi.

4. **Mutația**

   Mutația este folosită pentru a diversifica populația. Noi am folosit mutația pentru a explora noi soluții și pentru a nu ajunge în minime locale. Prin mutație se modifică aleator o parte a cromozomului.

5. **Populația**

   În problema canibalilor, noi am făcut ca populația actuală să fie mereu înlocuită de noua populație, astfel copii nu se întâlnesc cu părinții lor niciodată.

2.3 Ce dorim să obținem

Ne dorim obținerea unui algoritm genetic capabil să rezolve problema canibalilor într-un timp optim și să ajungă la o soluție optimizată și viabilă. O soluție viabilă ar presupune ca toți canibalii și misionarii să fie mutați pe malul opus, fără a fi mâncați misionarii.

Optimizarea acestei soluții reprezintă eliminarea traseelor inutile care conțin stări repetitive, stări în care misionarii ajung mâncați de canibali, stări care ar fi imposibile, spre exemplu nefiind posibil să muți mai mulți canibali decât există pe unul dintre maluri. Cu ajutorul acestor optimizări ne dorim să obținem niște soluții ideale.

3 Aspecte teoretice relevante și state-of-the-art

Există mai multe metode de rezolvare a problemei canibalilor și a misionarilor. Noi am realizat un algoritm genetic pentru rezolvare, dar mai există și algoritmii de căutare care se bazează pe explorarea soluțiilor folosind un arbore prin căutarea în adâncime, lățime, etc. Algoritmii se aseamănă prin faptul că ambii caută într-un spațiu de soluții pentru a ajunge la o soluție optimă, folosesc anumite funcții pentru a decide dacă soluția este ceea ce ne dorim și dacă respectă condițiile impuse de problemă. Chiar dacă algoritmii au unele asemănări există și diferențe notabile între cele două metode de rezolvare a problemei. Diferențele ar fi: algoritmul genetic se bazează pe evoluția soluțiilor, în timp ce algoritmul de căutare se bazează pe explorarea nodurilor arborelui pentru a ajunge la o soluție, algoritmii genetici lucrează cu o populație de soluții la un moment dat, în timp ce algoritmii de căutare lucrează la un singur traseu la un moment dat, revenind la o stare anterioară pentru a explora și alte soluții posibile. Algoritmii genetici sunt preferați atunci când spațiul de soluții este foarte mare, deoarece

explorarea unui spațiu mare de soluții folosind un algoritm de căutare pe un arbore poate deveni foarte costisitoare mai ales din punct de vedere al timpului de execuție, dar și a memoriei utilizate.

Primul pas în funcționarea unui algoritm genetic este generarea unei populații inițiale, fiecare individ reprezentând o posibilă soluție a problemei. O populație inițială diversificată crește posibilitatea de a găsi o soluție bună. Totuși o selecție prea dură a indiviziilor poate duce rapid către o populație foarte asemănătoare, reducând diversitatea. [1]

Numărul de părinți utilizați în procesorul de recombinare este de obicei doi, în majoritatea cazurilor doi părinți sunt optimi pentru rezolvarea problemei. Există și cazuri în care un număr mai mare de părinți îmbunătățește performanța. Chiar dacă în natură nu există acest concept, de utilizare a mai multor părinți pentru recombinare, în cadrul algoritmilor genetici s-a studiat și această posibilitate pentru a se observa impactul acestei metode asupra rezolvarea problemei. [2]

Algoritmii genetici hibrizi au primit o mare atenție în ultimii ani și sunt din ce în ce mai folosiți pentru a rezolva probleme din lumea reală. Un algoritm genetic este capabil să includă alte tehnici în cadrul său pentru a realiza o metodă hibridă de rezolvare a problemei care conține ce este mai bun din ambele lumi. [3]

Menținerea diversității într-un algoritm genetic este foarte importantă, deoarece aceasta ajută la continua evoluție, evitand minime locale. Introducerea diferitelor metode pentru menținerea diversitații, duce la optimizarea algoritmului până într-un punct favorabil. Evitarea plafonării poate duce la o multitudine de posibilitați când vine vorba de evoluție.[4]

Identificarea ratelor de încrucișare și mutație , explicație pentru cum se pot evita minimele locale datorită mutației. Cum putem aborda aceste probleme într-un mod în care ținem cont de indicii de performanță, pentru a putea identifica care ar fi metodele optime de mutație și încrusișare. Existența mai multor metode pentru identificarea unei rate optime pentru problema noastră conferă un avantaj în optimizarea algoritmului.[5]

Funcțiile de fitness pot să varieze în funcție de problemă, și putem aplica diferite penalizari, care îmbunătațesc eficiența algoritmului genetic. Testarea efectelor asupra perfomanței algoritmului se face în funcție de rata de schimbare în funcția de fitness.[6]

Operațiunea de încrucișare este una dintre pașii necesari pentru optimizarea unui algoritm genetic. Aceasta necesită unul sau mai multe puncte de tăiere pentru a putea împărți genele părinților selectați către generația următoare. Tipul de încrucișare pe care îl folosim în problema canibalilor este Single-Point Crossover, adică tăierea printr-un singur loc și încrucișarea genei, ceea ce oferă un avantaj în performanța algoritmului nostru. Uneori, utilizarea mai multor puncte de tăiere poate duce la reducerea performanței algoritmului. Deși uneori multi-point crossover poate fi util, pentru problema canibalilor am dedus că performanța maximă este oferită de încrucișarea cu un singur punct de tăiere. [7]

Algoritmii genetici sunt un instrument pentru rezolvarea problemelor com-

plexe. Aceștia sunt utilizați nu numai pentru modelarea fenomenelor naturale, ci și pentru optimizare, modelare, proiectare și previziuni în domenii precum știința, tehnologia și viața de zi cu zi. Algoritmii genetici fi personalizați pentru o sarcină specifică, pot fi utilizați împreună cu alte metode pentru a crea metode hibride. [8]

Algoritmii genetici sunt o tehnologie populară pentru optimizarea și modelarea sistemelor evolutive, care este puternică pentru explorarea spațiilor mari și complexe ale soluțiilor. Deși au multe avantaje, cum ar fi flexibilitatea și capacitatea de a găsi rapid soluții promițătoare, au și dezavantaje, cum ar fi costurile ridicate de calcul, provocările de configurare a parametrilor și problemele de reprezentare a soluțiilor.[9]

Comparand diferitele metode de selectie, am ajuns la concluzia ca problema canibalilor ar putea fi optimizata cu ajutorul unei selectii elitiste, datorita necesitatii de gasire unei solutii maximale. Cealalta metoda posibila cum ar fi selectia ruletei nu ajuta la gasirea celui mai bun rezultat in mod constant, in cazul problemei canibalilor.[10]

4 Implementarea aspectelor teoretice în cadrul proiectului

4.1 Reprezentarea individului

Un individ este reprezentat de o listă de stări care reprezintă drumul realizat cu barca de pe un mal pe celălalt pentru a trece toți canibalii și misionarii. În barcă pot exista maxim 2 persoane și cel puțin 1 persoană deoarece barca nu se poate deplasa singură, acestea fiind: 2 canibali, 2 misionari, 1 misionar, 1 canibal sau 1 misionar și 1 canibal. Numărul necesar de pași pentru a trece toate persoanele pe celălalt mal este de 11.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Individ{
    private static final int NR_MISIONARI = 3;
    private static final int NR_CANIBALI = 3;
    private static final int MAX_PASI = 11;
    private static final Random random = new Random();
    // Mutari posibile: {canibali, misionari}
    private static final int[][] MUTARI_POSIBILE = {{1, 0}, {0, 1}, {1,
        1}, {2, 0}, {0, 2}};

    private List<Stare> individ;
    private double fitness;

    public Individ(){
```

```
        this.individ = null;
        this.fitness = 0;
    }

    public static List<Stare> genereazaIndivid() {
        List<Stare> individ = new ArrayList<>();
        for (int i = 0; i < MAX_PASI; i++)
        {
            int[] mutare =
                MUTARI_POSIBILE[random.nextInt(MUTARI_POSIBILE.length)];
            individ.add(new Stare(mutare[0], mutare[1]));
        }
        return individ;
    }

    public void setIndivid(List<Stare> individ) {
        this.individ = individ;
    }
    public List<Stare> getIndivid() {
        return individ;
    }
    public int getNrMisionari() {
        return NR_MISIONARI;
    }
    public int getNrCanibali() {
        return NR_CANIBALI;
    }
    public double getFitness() {
        return fitness;
    }
    public void setFitness(double fitness){
        this.fitness = fitness;
    }
    public void afisareIndivid(){
        for (Stare stare : individ) {
            stare.afisareStare();
        }
        System.out.println();
    }
}
```

O stare este reprezentată de două variabile care semnifică numărul de canibali și/sau misionari aflați în barcă la un moment dat.

```
public class Stare{
    int CB, MB;
    public Stare(int CB, int MB){
        this.CB = CB;
```

```
        this.MB = MB;
    }

    //Afisare stare
    public void afisareStare(){
        System.out.print("(" + CB + " : " + MB + "), ");
    }
    //Getters and Setters
    public int getCB() {
        return CB;
    }

    public void setCB(int CB){
        this.CB = CB;
    }

    public int getMB(){
        return MB;
    }

    public void setMB(int MB){
        this.MB = MB;
    }
}
```

4.2 Generarea populației inițiale

Fiecare individ din populația inițială este generat total aleator. Lungimea individului generat este de 11, fiind numărul necesar de pași pentru găsirea soluției. Stările posibile sunt memorate într-o matrice bidimensională constantă din care se alege aleator o stare care se adaugă la individ.

```
    public static List<Stare> genereazaIndivid(){
        List<Stare> individ = new ArrayList<>();
        for (int i = 0; i < MAX_PASI; i++){
            int[] mutare =
                MUTARI_POSIBILE[random.nextInt(MUTARI_POSIBILE.length)];
            individ.add(new Stare(mutare[0], mutare[1]));
        }
        return individ;
    }
```

4.3 Funcția de fitness

Fitness-ul individului este calculat pe baza restricțiilor impuse de problemă. La început barca se află pe malul stâng. Pentru fiecare stare se verifică dacă: mutarea este posibilă sau nu, adică dacă de pe un mal se încearcă mutarea unui

număr mai mare de canibali sau misionari decât sunt pe acel mal, mutarea este identică cu cea precedentă, penalizându-se dacă se repetă stările, sau acordându-se puncte dacă stările diferă. Când barca ajunge pe celălalt mal, se actualizează starea bărcii. După ce toate stările au fost parcurse se verifică câte persoane au fost mutate pe celălalt mal și se acordă un punctaj în funcție de numărul acestora.

```
public static void calculeazaFitness(Individ individ){
    int nrMisionariStanga = NR_MISIONARI;
    int nrCanibaliStanga = NR_CANIBALI;
    int nrMisionariDreapta = 0;
    int nrCanibaliDreapta = 0;
    boolean barcaStanga = true; // Barca incepe pe malul stang
    double fitness = 0;

    int canibaliMutatiAnterior = 0;
    int misionariMutatiAnterior = 0;

    for (Stare stare : individ.getIndivid()){
        int canibaliMutati = stare.getCB();
        int misionariMutati = stare.getMB();
        // Actualizare stari bazate pe directia barcii
        if (barcaStanga){
            if (canibaliMutati > nrCanibaliStanga || misionariMutati
                > nrMisionariStanga){
                fitness += PUNCTE_MUTARE_IMPOSIBILA;
            }
            else{
                fitness += PUNCTE_MUTARE_POSIBILA;
            }
            if(canibaliMutati==canibaliMutatiAnterior &&
                misionariMutati==misionariMutatiAnterior){
                fitness += PUNCTE_STARI_IDENTICE;
            }
            else{
                fitness += PUNCTE_STARI_DIFERITE;
            }
            if
                (canibaliMutati+nrMisionariDreapta>misionariMutati+nrCanibaliDreapta){
                fitness += PUNCTE_MISIONARI_MANCATI;
            }
            canibaliMutatiAnterior=canibaliMutati;
            misionariMutatiAnterior=misionariMutati;
            nrCanibaliStanga -= canibaliMutati;
            nrMisionariStanga -= misionariMutati;
            nrCanibaliDreapta += canibaliMutati;
            nrMisionariDreapta += misionariMutati;
        }
        else{
```

```
            if (canibaliMutati > nrCanibaliDreapta || misionariMutati
                > nrMisionariDreapta){
                fitness += PUNCTE_MUTARE_IMPOSIBILA;
            }
            else{
                fitness += PUNCTE_MUTARE_POSIBILA;
            }
            if(canibaliMutati==canibaliMutatiAnterior &&
                misionariMutati==misionariMutatiAnterior){
                fitness += PUNCTE_STARI_IDENTICE;
            }
            else{
                fitness += PUNCTE_STARI_DIFERITE;
            }
            if
                (canibaliMutati+nrMisionariStanga>misionariMutati+nrCanibaliStanga){
                fitness += PUNCTE_MISIONARI_MANCATI;
            }
            canibaliMutatiAnterior=canibaliMutati;
            misionariMutatiAnterior=misionariMutati;
            nrCanibaliDreapta -= canibaliMutati;
            nrMisionariDreapta -= misionariMutati;
            nrCanibaliStanga += canibaliMutati;
            nrMisionariStanga += misionariMutati;
        }
        // Barca isi schimb pozitia
        barcaStanga = !barcaStanga;
    }
    fitness +=
        PUNCTE_PERSOANA_MUTATA*(nrMisionariDreapta+nrCanibaliDreapta);
    individ.setFitness(fitness);
}
```

4.4 Selecția individului

Pentru selecția individului am ales metoda turnirului care presupune selectarea aleatoare a câtorva indivizi din populație, din care se alege un individ. Funcția de fitness fiind o funcție de maxim, se alege individul cu fitness-ul cel mai mare. Numărul de indivizi din care se face selecția este de 4, care poate fi modificat în funcție de evoluția problemei și de performanța acesteia.

```
public static Individ selectieTurnir(List<Individ> populatie, int k){
    // Alegem un subset de k indivizi
    List<Individ> turnir = new ArrayList<>();
    for (int i = 0; i < k; i++){
        int indexAleator = random.nextInt(populatie.size());
        turnir.add(populatie.get(indexAleator));
    }
```

```
        // Gasim cel mai bun individ din turnir (cel cu fitness-ul cel
            mai mare)
        turnir.sort((individ1, individ2) ->
            Double.compare(individ2.getFitness(),
            individ1.getFitness()));
        Individ celMaiBun = turnir.getFirst();
        return celMaiBun;
    }
```

4.5 Încrucișarea

Prin încrucișarea părinților are loc nașterea a doi copii. Se alege aleator un punct de tăiere care poate fi maxim dimensiunea părinților minus unu, pentru a nu copia pur și simplu părinții fără să aibă loc procesul de încurcișare. Primul copil moștenește prima parte din gena unui părinte și a doua parte de la celălalt copil, și al doilea copil moștenește celelalte părți de la cei doi părinți.

```
    public static List<Individ> incrucisare(Individ parinte1, Individ
        parinte2)
    {
        // Incrucisarea a doi parinti
        List<Stare> copilIndivid1 = new ArrayList<>();
        List<Stare> copilIndivid2 = new ArrayList<>();
        Individ copil1 = new Individ();
        Individ copil2 = new Individ();
        int punctIncrucisare =
            random.nextInt(parinte1.getIndivid().size()-1);
        for (int i = 0; i < parinte1.getIndivid().size(); i++){
            if (i < punctIncrucisare){
                copilIndivid1.add(parinte1.getIndivid().get(i));
                copilIndivid2.add(parinte2.getIndivid().get(i));
            }
            else{
                copilIndivid1.add(parinte2.getIndivid().get(i));
                copilIndivid2.add(parinte1.getIndivid().get(i));
            }
        }
        copil1.setIndivid(copilIndivid1);
        copil2.setIndivid(copilIndivid2);
        return List.of(copil1, copil2);
    }
```

4.6 Mutația

Procesul de mutație are loc prin generarea unui număr aleator care se verifică dacă este mai mic decât un număr stabilit care reprezintă procentul de mutație ales în problemă. Dacă numărul este mai mic, se alege tot aleator o stare din

individid care va fi schimbată cu o altă stare care se verifică să fie diferită față de starea care va fi schimbată.

```
public static void mutatie(Individ individ, int sansaMutatie){
    // Mutatie a unui individ
    int sansa = random.nextInt(100);
    if (sansa <= sansaMutatie){
        int pozitieMutatie =
            random.nextInt(individ.getIndivid().size()-1);
        int[] mutare =
            MUTARI_POSIBILE[random.nextInt(MUTARI_POSIBILE.length)];
        while (individ.getIndivid().get(pozitieMutatie).getCB() ==
            mutare[0] &&
        individ.getIndivid().get(pozitieMutatie).getMB() ==
            mutare[1]){
            mutare =
                MUTARI_POSIBILE[random.nextInt(MUTARI_POSIBILE.length)];
        }
        individ.getIndivid().set(pozitieMutatie, new Stare(mutare[0],
            mutare[1]));
    }
}
```

5 Testare și validare

După terminarea codului, am început să facem debugging de mai multe ori, începând cu 100 de indivizi și am observat că fitness-ul a rămas blocat la o anumită valoare, ceea ce înseamnă că funția a ajuns într-un minim local, iar după ce am mărit numarul de indivizi la 1000, am observat că fitness indivizilor a început să crească.

Procedura de încrucișare am inițializat-o folosind un singur punct de tăiere, care ne-a oferit niște rezultate satisfăcătoare. Mai apoi am încercat să mărim numarul punctelor de tăiere pentru a crește performanța algoritmului, dar folosirea mai multor puncte de tăiere, nu a oferit nici un fel de îmbunătățire programului nostru. De aceea ne-am decis să rămânem la încrucișare folosind un singur punct de tăiere.

Rata de mutație a rasei inițiale este de 5 la sută și am încercat mărirea acesteia, ducând la o populație foarte haotică și distrugând indivizii prea mult, soluțiile bune ajungând să fie stricate de mutația prea mare.

Am încercat să acordăm diferite punctaje și penalizări în funcție de stările individului pentru a putea vedea cum influențează algoritmul general și pentru a încerca să ieșim din minima locală, unde soluțiile posibile ajung să stagneze.

Pentru generarea unui individ din populația inițială am ales să-l generăm total aleator, deoarece soluțiile care cuprind stări imposibile, ajung să fie eliminate în procesul de selecție datorită fitness-urilor mai mici pe care le au.

6 Rezultate obținute

În aplicația noastră am reușit:

1. Generarea populației inițiale printr-o metodă total aleator, care va fi folosită pentru pornirea algoritmului de rezolvare a problemei.
2. Să implementăm metoda turnirului, care ne-a ajutat să găsim individul cu fitness-ul cel mai bun.
3. La încrucișare am folosit metoda cu un singur punct de tăiere, care ne-a dat ca rezultat câte doi copii.
4. Să aplicăm mutația asupra populației, pentru a ne asigura că nu ajungem cu aceasta într-o minimă locală și ne-a mai ajutat la explorarea de noi soluții.
5. Pentru fiecare populație deja existentă, am făcut ca populația nouă să o înlocuiască, iar astfel copiii nu ajung să își întâlnească părinții.

Pe viitor am mai putea încerca:

1. Să modificăm funcția de fitness și să o mai optimizăm pentru a ne ajuta în calcularea unui fitness mai reprezentativ pentru individul respectiv.
2. Folosirea de noi metode prin care să evităm ca algoritmul să ajungă în minimă locală.

7 Concluzii

Aplicația a fost realizată majoritar de Alin, cu modificări făcute de Andrei, iar Paul și Cătălina au lucrat la partea de testare.

Documentația a fost realizată majoritar de către Cătălina și Paul, care au primit unele indicații de la Sima și Tuns.

Unit testing-ul a fost realizat de Paul și Cătălina, ajutați de Alin și Andrei.

În realizarea algoritmului am întâmpinat diferite dificultăți. Primul obstacol și lucru asupra căruia trebuia să ne hotărâm era reprezentarea individului în cadrul algoritmului. După mai multe discuții și căutări cu privire la problemă am decis ca individul să fie reprezentat sub formă de listă care conține traseul efectuat.

O altă problemă întâlnită a fost determinarea funcției de fitness care joacă un rol important în găsirea soluției. Am încercat diferite metode pentru acordarea de puncte care să favorizeze indivizii mai buni pentru a avea șanse mai mari de a fi ales în procesul de încrucișare. Valorile folosite în calcularea fitness ului au fost schimbate de mai multe ori pentru a încerca să aflăm o acordare care să reflecte cât mai corect cât de bun este individul respectiv.

Implementarea funcțiilor de încrucișare și mutație a fost destul de ușoară, fără a fi nevoie de prea multe modificări asupra lor. Acestea și-au îndeplinit scopul în cadrul algoritmului în procesul de găsire a soluției.

Paul și Cătălina au întâmpinat probleme la realizarea unit testing-ului, deoarece nu aveau conștințele necesare care să îi ajute la o rezolvare cât mai bună și corectă a problemelor.

În mare parte, toți din grupă ne-am ajutat de viziualizarea unor videoclipuri pe YouTube, am apelat și la Copilot și ne-am ajutat reciproc. De fiecare dată când am lucrat la proiect, am intrat cu toții pe discord, pentru a ne putea împărtăși părerile și pentru a ne sfătui.

Am mai întâmpinat toți probleme la Overleaf, deoarece a fost prima dată când am lucrat în el și ne-a luat ceva pană am găsit comenzile necesare pentru documentația noastră.

Prezentarea în Canva a fost realizată de către Tuns, cu mici indicații de la Sima, Cătălina și Paul.

Procentaj de lucru în proiect: Alin 50%, Andrei 20%, Cătălina 15%, Paul 15%.

References

[1] Pedro A Diaz-Gomez and Dean F Hougen. Initial population for genetic algorithms: A metric approach. In *Gem*, pages 43–49. Citeseer, 2007.

[2] Agoston E Eiben, P-E Raue, and Zs Ruttkay. Genetic algorithms with multi-parent recombination. In *International conference on parallel problem solving from nature*, pages 78–87. Springer, 1994.

[3] Tarek A El-Mihoub, Adrian A Hopgood, Lars Nolle, and Alan Battersby. Hybrid genetic algorithms: A review. *Eng. Lett.*, 13(2):124–137, 2006.

[4] Deepti Gupta and Shabina Ghafir. An overview of methods maintaining diversity in genetic algorithms. *International journal of emerging technology and advanced engineering*, 2(5):56–60, 2012.

[5] Ahmad Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Hammouri, and VB Surya Prasath. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information*, 10(12):390, 2019.

[6] S Kazarlis and Vassilios Petridis. Varying fitness functions in genetic algorithms: Studying the rate of increase of the dynamic penalty terms. In *International conference on parallel problem solving from nature*, pages 211–220. Springer, 1998.

[7] Padmavathi Kora and Priyanka Yadlapalli. Crossover operators in genetic algorithms: A review. *International Journal of Computer Applications*, 162(10), 2017.

[8] Wojciech Paszkowicz. Genetic algorithms, a nature-inspired tool: survey of applications in materials science and related fields. *Materials and Manufacturing Processes*, 24(2):174–197, 2009.

[9] Aymeric Vie, Alissa M Kleinnijenhuis, and Doyne J Farmer. Qualities, challenges and future of genetic algorithms: a literature review. *arXiv preprint arXiv:2011.05277*, 2020.

[10] Saneh Lata Yadav and Asha Sohal. Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering, Science and Mathematics*, 6(3):174–180, 2017.