Cuprins

[Introducere 3](#_Toc11797697)

[1 Descrierea problemei 4](#_Toc11797698)

[1.1 Căutare pe șiruri 4](#_Toc11797699)

[1.2 Descompunerea QR 4](#_Toc11797700)

[2 Resurse utilizate 5](#_Toc11797701)

[2.1 Subcapitol 1 5](#_Toc11797702)

[2.2 Subcapitol 2 5](#_Toc11797703)

[3 Rezolvarea problemei 6](#_Toc11797704)

[3.1 Metoda exhaustivă 6](#_Toc11797705)

[3.2 Metoda eficientă 6](#_Toc11797706)

[4 Abordarea problemei utilizând metode euristice 7](#_Toc11797707)

[4.1 Structură 7](#_Toc11797708)

[4.2 Reprezentarea datelor 7](#_Toc11797709)

[4.3 Operatori 8](#_Toc11797710)

[4.3.1 *Crossover* 8](#_Toc11797711)

[4.3.2 *Mutation* 10](#_Toc11797712)

[4.4 Funcția *fitness* 11](#_Toc11797713)

[4.5 Condiții de oprire 12](#_Toc11797714)

[5 Algoritmi euristici 14](#_Toc11797715)

[5.1 Algoritm genetic naiv 14](#_Toc11797716)

[5.1.1 *Tournament selection* 14](#_Toc11797717)

[5.1.2 *Roulette wheel selection* 15](#_Toc11797718)

[5.2 *Hill* *climbing* 15](#_Toc11797719)

[5.3 *Simulated* *annealing* 16](#_Toc11797720)

[5.4 *Building* *blocks* 17](#_Toc11797721)

[5.5 Algoritmi cu restricție 19](#_Toc11797722)

[6 Euristici în combinație cu metoda eficientă 20](#_Toc11797723)

[6.1 Integrare 20](#_Toc11797724)

[6.2 Performanțe 20](#_Toc11797725)

[7 Implementare 21](#_Toc11797726)

[7.1 Structura generală 21](#_Toc11797727)

[7.2 Fișier de configurație 21](#_Toc11797728)

[7.3 Funcții de bază 21](#_Toc11797729)

[8 Reprezentarea grafică a datelor și elemente de statistică 22](#_Toc11797730)

[8.1 Generare seturi de date 22](#_Toc11797731)

[8.2 Elemente de statistică 22](#_Toc11797732)

[8.3 Grafice 24](#_Toc11797733)

[9 Comparația algoritmilor bazată pe outputuri 25](#_Toc11797734)

[9.1 Comportamentul algoritmilor pe seturi de date reale 25](#_Toc11797735)

[9.2 Comportamentul algoritmilor pe seturi de date generate 29](#_Toc11797736)

[9.3 Avantaje versus dezavantaje 29](#_Toc11797737)

[9.4 Performanțe 29](#_Toc11797738)

# Introducere

Motivație –

Gradul de noutate –

Obiective-

Structură , titlul capitolelor și legatura dintre ele

# Descrierea problemei

Text descriere

## Căutare pe șiruri

Text

## Descompunerea QR

Text

# Resurse utilizate

Text descriere

## Subcapitol 1

Text

## Subcapitol 2

Text

# Rezolvarea problemei

Text descriere

## Metoda exhaustivă

Utilizarea elementelor de combinatorica pentru a analiza tot setul de date

## Metoda eficientă

Utilizare descompunerii QR intr-un mod eficient, astfel incat aceasta nu se va calcula pentru fiecare sub-model (explicatie)

# Abordarea problemei utilizând metode euristice

Am ales această abordare prin metode euristice în primul rând pentru că problema are un spațiu de căutare foarte vast, numărul de sub-modele crescând exponențial odată cu creșterea numărului de coeficienți, astfel crescând și timpul de execuție al programului. În al doilea rând pentru a studia și compara diferite tipuri de implementări și abordări și nu în ultimul rând pentru că aceste metode euristice sunt de actualitate, astfel lăsând loc de idei originale.

În acest capitol voi introduce elementele cheie utilizate în algoritmii euristici, voi aminti mai multe tipuri, însa din acestea le voi discuta doar pe cele utilizate în aplicație împreună cu o scurtă motivație pentru care le-am ales pe unele în defavoarea celorlalte.

## Structură

Fiind metode euristice, nu tot setul de date este evaluat, iar structura la baza unui algoritm euristic pornește de la generarea unui punct de plecare *random*. Acesta poate fi un singur individ, sub-model sau o întreagă populație de sub-modele. Un sub-model reprezintă un posibil candidat al celei mai bune soluții din spațiul de căutare.

Pentru a decide dacă un sub-model este bun se calculează o funcție de cost denumită funcție *fitness* care asociază fiecărui sub-model o valoare numerică. Această funcție poate fi minimizată sau maximizată în funcție de problema pe care se aplică un astfel de algoritm.

Trecerea de la o iterație la alta, sau cu alte cuvinte, plasarea punctului de plecare mai departe în spațiul de căutare, se face cu ajutorul operatorilor despre care voi discuta în subcapitolul 4.3. Algoritmul poate fi oprit alegând diverse metode de oprire pe care le voi descrie la finalul acestui capitol.

## Reprezentarea datelor

În algoritmii euristici cea mai des întâlnită reprezentare a datelor este reprezentarea binară, *array*-uri de biți. Deoarece setul de date utilizat are în componență o matrice de linii și coloane, fiecare coloană reprezentând o caracteristică diferită, un element de compoziție, am ales aceeași reprezentare binară. Un sub-model reprezintă o combinație a coloanelor matricei. Astfel, un sub-model va reprezenta în prealabil un șir de biți de 0 și 1, peste care am adăugat o altă reprezentare și anume, pentru fiecare bit setat pe 1, indexul acestui bit va reprezenta exact indexul coloanei alese pentru a alcătui sub-modelul.

De exemplu : având o matrice de 6 coloane numerotate de la 0 la 5, șirul 010110 este reprezentarea sub-modelului format din matricea de coloane : 1,3 și 4.

De asemenea, în algoritmii implementați, am utilizat limbajul algoritmilor genetici și anume:

* Un sub-model reprezintă o combinație de coloane, dar este utilizat și sub numele de individ, candidat, *genotype* sau *chromosome*
* Un individ este alcătuit dintr-un șir de biți setați pe 1 sau 0, acești biți mai sunt utilizați sub numele de gene

## Operatori

Operatorii sunt algoritmi aplicați peste sub-modele cu scopul de a genera alte sub-modele noi, diferite , însa cu premisa că noile sub-modele păstrează un anumit procent din material genetic al sub-modelului părinte. Mai precis, sub-modelul nou obținut va păstra șiruri de biți de 0 și 1 neschimbate. În cele mai multe cazuri acești operatori transformă sub-modelul prin procedee *random*.

### *Crossover*

Un operator des utilizat este operatorul de *crossover*. Acesta este aplicat pe doi indivizi denumiți părinți și generează alți doi indivizi sub numele de copii, care vor avea o combinație a materialului genetic a celor doi indivizi părinte. Poate fi de mai multe tipuri (*1-point crossover, n-point crossover, uniform crossover, flat crossover,* etc) însă cei utilizați în această lucrare sunt:

* *1-point crossover*
* *1-point crossover* adaptat
* *uniform crossover*
* *RRC[[1]](#footnote-1) crossover*

Metoda *1-point crossover* este cea mai simplă și comună operație de *crossover*. Se alege aleator un punct de taiere al indivizilor, acesta va fi comun atât pentru primul individ părinte cât și pentru cel de-al doilea. Primul individ copil va moșteni șirul de biți de la primul individ părinte până la punctul de tăiere, la care, se adaugă șirul de biți moștenit de la cel de-al doilea individ părinte, după punctul de tăiere. Cel de-al doilea individ copil va moșteni șirurile de biți rămase de la indivizii părinte, cele care nu au fost selectate pentru primul individ copil.

De exemplu, pentru punctul de tăiere de valoare 1 rezultate sunt:

* Părinte1 : 10|0100
* Părinte2 : 00|1011
* Copil1 : 10|1011
* Copil2 : 00|0100

Metoda *1-point crossover* adaptat este o modificare a metodei *1-point crossover*. La fel ca și la metoda de bază, se alege aleator un punct de tăiere al indivizilor, acesta va fi comun atât pentru primul individ părinte cât și pentru cel de-al doilea. Diferența este că acest punct de tăiere va respecta o condiție, și anume acesta este ales în așa măsură încât prin aplicarea metodei de *1-point crossover* adaptat, indivizii copii vor păstra același număr de gene de valoare 1 ca și indivizii părinte. Această condiție impune ca înainte de punctul de taiere cât și după acesta, numărul de gene cu valoarea 1 trebuie sa fie egal, condiție ce trebuie îndeplinită simultan de ambii indivizi părinte. Această metodă este creată de mine și am utilizat-o în Algoritmi cu restricție pe care îi voi descrie în subcapitolul 5.5.

Metoda *uniform crossover* este o metodă care nu aduce foarte multe schimbări celor doi indivizi copil, daca indivizi părinte au gene asemănătoare. La această metodă pentru fiecare dintre genele de pe aceeași poziție din indivizi părinte care sunt diferite, se va alege cu o probabilitate generată aleator ce decide dacă indivizii părinte vor face *switch* între aceste gene. Indivizii copil vor fi apoi fiecare o copie a noilor indivizi părinte. Condiția ca doua gene sa fie interschimbate este ca probabilitatea generată aleator sa fie mai mare decât o probabilitate de interschimbare stabilită în prealabil.

De exemplu, pentru o probabilitate de *switch* între indivizii părinte de 50% în care pentru fiecare din genele diferite s-a generat o probabilitate aleatorie mai mare de 50% rezultatele sunt:

* Părinte1 : 100100
* Părinte2 : 001011
* Copil1 : 001011
* Copil2 : 100100

Un alt exemplu pentru o probabilitate de *switch* între indivizii părinte de 50% în care pentru ultimele două gene diferite s-a generat o probabilitate mai mică de 50% rezultatele sunt:

* Părinte1 : 100100
* Părinte2 : 001011
* Copil1 : 001000
* Copil2 : 100111

Metoda *RRC crossover* este o metodă foarte asemănătoare celei descrise anterior, *uniform crossover.* În același fel, se generează un număr aleator pentru acele gene care diferă pentru indivizii părinte. De această dată însă, se generează o probabilitate aleatoare pentru fiecare individ copil, și nu se va face o interschimbare a acestor gene, ci în funcție de probabilitatea generată, dacă aceasta este mai mare decât probabilitatea stabilită în prealabil, va indica dacă pe acea poziție individul copil va prelua gena de la primul individ părinte sau de la cel de-al doilea.

De exemplu, pentru o probabilitate de preluare a genelor indivizilor părinte de 50% în care pentru fiecare din genele diferite s-a generat o probabiliate1 a primului individ-copil aleatoriu mai mare de 50% și o probabiliate2 a celui de-al doilea individ-copil mai mică de 50% rezultatele sunt:

* Părinte1 : 100100
* Părinte2 : 001011
* Copil1 : 100100
* Copil2 : 001011

Un alt exemplu pentru o probabilitate de preluare a genelor indivizilor părinte de 50% în care pentru fiecare din genele diferite s-a generat o probabiliate1 a primului individ copil aleatoriu mai mare de 50% și o probabiliate2 a celui de-al doilea individ copil tot mai mare de 50%, însă pentru ultimele 2 gene probabiliatea2 este mai mica de 50% rezultatele sunt:

* Părinte1 : 100100
* Părinte2 : 001011
* Copil1 : 100100
* Copil2 : 100111

Ca o mică concluzie, această ultimă metodă descrisă duce la generarea unor noi indivizi care păstrează tot mai mult structura materialului genetic ale celor doi indivizi părinte. Este o metodă care oferă rezultate mai puțin aleatoare. Din acest motiv, când generațiile devin din ce în ce mai stabile și mai evoluate, acele gene care duc la un *fitness* mai bun, vor avea o șansă mai mică de a se pierde. Această metodă de *crossover* este foarte utilă de exemplu în algoritmul genetic bazat pe *Building blocks* pe care îl voi discuta în subcapitolul 5.4 . Cu toate acestea și metodele bazate mai mult pe combinarea în mod aleator și generarea unor indivizi copil foarte diferiți de indivizii părinte pot avea un impact major în algoritmii care se opresc într-un minim local deoarece pot duce la modificare *landscape*-ului de căutare, metode cu rezultate foarte bune în algoritmul *Simulated annealing* pe care îl voi discuta în subcapitolul 5.3.

### *Mutation*

Un alt operator utilizat în obținerea noilor generații de indivizi este operatorul de mutație. O diferență majora între *crossover* și *mutation* este faptul că la generarea unui nou individ copil este nevoie de un singur individ părinte. O altă diferență o constituie faptul că cel mai des utilizate metode de mutație aduc modificări la un număr mic de gene. Ca și în cazul operatorului *crossover* există mai multe metode de mutație studiate (*insert muatation, inversion mutation, uniform mutation, etc*) însă cele utilizate de către mine sunt:

* *flip mutation*
* *interchanging mutation*
* *absolute interchanging mutation*
* *reversing mutation*

Metoda *flip mutation* este cea mai des utilizată metoda de către mine în această lucrare. Aceasta constă în generarea unui număr aleator de gene n ce constituie numărul de gene al individului părinte care vor aduce schimbări în individul copil. Așadar se generează n poziții aleatoare a căror gene își vor schimba valoare curentă cu cea opusă. Prin urmare, dacă gena de pe una din cele n poziții aleatore generate are valoarea 0 în individul părinte, aceasta va lua valoare 1 în individul copil, în caz contrar daca gena are valoare 1, aceasta va lua valoarea 0, restul genelor sunt copiate așa cum sunt în individul copil.

De exemplu, dacă n are valoarea 3 și pozițiile generate au valorile 0, 2, și 3, atunci se obține:

* Părinte : 100100
* Copil : 001000

Metoda *interchanging mutation* este o metodă care nu aduce foarte multe schimbări individului copil. Se generează doua poziții aleatoare, iar genele de pe acele poziții fac schimb de valori în individul copil, restul genelor fiind copiate în individul copil așa cum sunt. Această metodă aduce avantajul păstrării numărului de gene de valoare 1 din individul părinte în individul copil nou creat. Mai concret, păstrează același conținut, modificând doar ordinea genelor. Este o metodă pe care am utilizat-o în Algoritmi cu restricție despre care voi vorbi în subcapitolul 5.5.

De exemplu, dacă se generează aleatoriu pozițiile 0 și 5, atunci se obține:

* Părinte : 100100
* Copil : 000101

Metoda *absolute interchanging mutation* este o metodă adaptată de mine după modelul celei prezentate anterior. Această metodă urmează aceeași procedura de generare a două poziții aleatoare pentru a fi interschimbate valorile genelor cu două restricții. Una dintre ele este ca cele doua poziții *random* generate să fie diferite, iar cealaltă restricție impune ca cele doua gene de pe pozițiile generate să aibă valori diferite. Aceste condiții au fost introduse de mine pentru a genera de fiecare dată indivizi copii cu *genotype* diferit de cel al indivizilor părinte.

Metoda *reversing mutation* este o metodă care face excepție de la ideea că operatorul de mutație nu aduce foarte multe schimbări. Această metoda generează o poziție aleatoare astfel că până la această poziție individul copil copiază fiecare genă din individul părinte așa cum este, iar pentru genele ce urmează după această poziție, în individul copil se copiază valoarea opusă a genelor din individul părinte. Daca poziția generată are valoarea 0, atunci individul copil este exact opusul individului părinte.

De exemplu, dacă se generează aleatoriu poziția 2 atunci se obține:

* Părinte : 100100
* Copil : 100011

Pentru obținerea unor rezultate cât mai bune, am introdus operatori care să aibă efect mai mare în generarea noilor populații de indivizi. Cel mai des am utilizat în combinație operatorul *RRC crossover* și *flip muation*.

## Funcția *fitness*

În problemele euristice este des utilizată funcția de *fitness* sau cu alte cuvinte funcția de cost. Această funcție este o reprezentare numerică ce exprimă la ce nivel de adaptare se clasifică un individ. Pe baza acestei funcții algoritmii execută căutarea celui mai bun candidat, astfel că joacă un rol foarte important în distincția unui individ adaptat față de unul mai puțin adaptat. Prin urmare, o funcție de cost are atribuția de a clasifica cât mai bine indivizii și de aceea aceasta poate să ajungă la un nivel de complexitate mare, ceea ce induce un timp de execuție depășit.

În rezolvarea găsirii celui mai bun candidat pe problema căutării celui mai bun set de coeficienți care să aproximeze soluția sistemului, discutat în capitolul 1, am utilizat funcția de RSS[[2]](#footnote-2). Această funcție calculează distanța euclidiană de la soluția sistemului la soluția aproximată, ceea ce înseamnă că, cu cât eroarea este mai mică cu atât soluția aproximată este mai bună. Prin urmare cu cât un candidat are funcția *fitness* cu o valoare mai mică, cu atât acest candidat este mai favorabil.

Dat fiind faptul că, pentru a calcula RSS-ul unui individ nu se ține cont și de dimensiunea acestuia, pentru a obține rezultate cât mai reale, am aplicat funcția de AIC[[3]](#footnote-3). Prin urmare, pe lângă obținerea mediei de eroare, am ținut cont pentru funcția de cost și de numărul de coloane selectate al candidatului. Această funcție AIC exprimă câtă informație din setul de date a fost luată în vederea obținerii funcției de cost. Cu cât eroarea este mai mică și tot odată informațiile pierdute mai puține, cu atât funcția va da un rezultat mai bun, în cazul de față în interesul meu este obținerea unei valori cât mai mici.

În concluzie, funcția *fitness* arată în felul următor:

*AIC = n + n\*log 2π + n\*log(RSS/n) + 2\*(p + 1) ,*

unde n semnifică numărul de coloane al setului de date, iar p înseamnă numărul de coloane al candidatului.

## Condiții de oprire

Întrucât algoritmii euristici nu verifică tot setul de date, aceștia necesită o condiție de oprire. Dat fiind faptul că spațiul de căutare al acestor probleme este foarte mare, pentru oprirea algoritmului este necesară o altă condiție decât aceea că nu ar mai exista sub-seturi pentru a fi verificate. Există multe metode studiate, unele mai eficiente decât altele, însă depind foarte mult de algoritm.

Voi preciza în acest subcapitol câteva dintre procedurile de oprire, însă le voi discuta mai amănunțit în capitolul ce urmează.

Așadar, una din condițiile de oprire este atunci când algoritmul converge, ceea ce înseamnă că în iterațiile următoare algoritmul nu oferă un candidat mai adaptat față de cel deja clasificat ca cel mai bun.

O altă condiție de oprire este atunci când numărul maxim de iterații stabilit în prealabil ajunge la valoarea maximă. De asemenea, când populația de indivizi nu este prea vastă, iar procedura de selecție a generației următoare devine prea strictă, populația poate să ajungă să fie vidă și atunci algoritmul este oprit.

Un alt exemplu, strict pe algoritmul *Simulated* *annealing* pe care îl voi discuta în capitolul următor, este atunci când procedura de a accepta un alt posibil candidat pentru a fi analizat devine imposibilă, întrucât temperatura se apropie de valoarea 0, în condițiile în care acest algoritm se bazează pe faptul că cu cât temperatura este mai mare cu atât probabilitatea unui candidat de a fi ales pentru generația următoare crește.

# Algoritmi euristici

Un algoritm euristic are ca proprietate principală faptul că ajunge la un rezultat într-un timp finit. Acești algoritmi aduc soluții la probleme din clasa problemelor NP, unde spațiul de căutare este foarte mare, atenție aduc cel puțin o soluție aproximată deci nu asigură găsirea soluției perfecte . În cazul de față, pentru căutarea celei mai bune combinații de proprietăți pentru a descrie obiectivul problemei, dacă numărul de proprietăți ar fi de 500, pentru verificarea tuturor posibilităților existente, algoritmul ar trebui sa verifice 2500 de combinații pentru a determina cel mai bun sub-set de proprietăți.

## Algoritm genetic naiv

Algoritmii genetici au fost introduși încă din anii ’60 de către John Holland. Aceștia fac parte din clasa algoritmilor euristici, dar mai mult decât atât, urmăresc modelul evoluției darwiniste. Astfel că un algoritm genetic urmărește exact evoluția naturală a unei populații de indivizi pe parcursul unor generații, unde cel mai bine adaptat individ va supraviețui evoluției. Au la bază o structură bine definită care însa poate fi parametrizată și modelată în favoarea setului de date.

O structură foarte simplă a algoritmului genetic pe care am abordat-o pentru a implementa algoritmul genetic naiv poate fi descrisă în câțiva pași. Pentru început am generat aleator o populație de indivizi, numărul de indivizi fiind configurabil în funcție de numărul de observații din setul de date. Am calculat *fitness*-ul tuturor indivizilor, salvând cel mai bun candidat.

Pentru noua generație de indivizi ce urmează, am trecut indivizii printr-un proces de selecție, astfel nu toți indivizii vor produce generații noi de indivizi ci doar cei selectați. Voi descrie în următoarele două subcapitole metodele de selecție. După acest proces de filtrare al populației, am efectuat asupra indivizilor rămași, operații de *mutation* și *crossover* .

În urma acestor operații genotipul indivizilor se modifică, mutând spațiul de căutare într-o nouă direcție. Înainte de a trece la următoare iterație a algoritmului, printr-un proces simplu de comparație a funcției *fitness* am actualizat individul cel mai adaptat, doar în cazul în care noua generație a produs un individ mai adaptat decât o generație anterioară.

În fiecare iterație ce urmează, am efectuat aceiași pași și anume selecția populației, modificarea indivizilor prin intermediul operatorilor și salvarea celui mai bun individ, până ce algoritmul converge. Așadar condiția de oprire al algoritmului genetic este una din cele discutate în subcapitolul Condiții de oprire și anume atunci când pe parcursul unui număr finit de generații consecutive, cel mai bun individ nu este actualizat.

### *Tournament selection*

Metoda de selecție *Tournament selection* formează un *pool* temporar de dimensiune k pe care îl populează cu indivizi aleși în mod aleatoriu din populația curentă. Din acest *pool* alege pentru generația următoare pe cel mai bun candidat. Tot acest proces reprezintă un turneu de selecție. Cu cât turneul se repetă mai mult, cu atât generația următoare va avea un număr mai mare de candidați.

Un aspect foarte important este alegerea acestui număr k, cu cât turneul trebuie să aleagă cel mai bun candidat dintr-un număr mare de candidați al acestui *pool* temporar, cu atât crește probabilitatea ca turneul să alegă același candidat pentru noua populație, ceea ce înseamnă că populația nu va fi diversificată. Acest lucru forțează algoritmul să fie blocat în același spațiu de căutare.

- implementare

### *Roulette wheel selection*

Metoda de selecție *Roullete wheel* urmează ideea că fiecare individ are șansa să fie ales pentru viitoarea generație chiar și în cazul în care funcția *fitness* îl clasifică ca fiind cel mai nefavorabil candidat. Astfel, fiecărui individ i se calculează probabilitatea de a fi selectat pe baza evaluării funcției *fitness* în raport cu suma *fitness*-ului tuturor indivizilor din populația curentă, probabilitate care este cumulată, ceea ce înseamnă că proporția probabilității va crește pe măsură ce este calculată de la un individ la altul. În acest fel probabilitățile vor fi treptat din ce în ce mai mari și în același timp unice.

Șansa de care vorbeam mai sus este dată de faptul că în momentul stabilirii dacă un individ este ales sau nu, am generat o probabilitate *random* . Pentru o selecție realistă și unică, această probabilitate *random* trebuie să fie cuprinsă între valorile probabilității de selecție al individului curent și individul următor astfel, nici-un al element de pe ruleta nu poate îndeplini aceeași condiție. Această condiție asigură faptul că selecția este unică pentru un singur candidat exact ca indicatorul de pe o ruletă din viața reala care selectează în mod unic elementul câștigat.

- implementare

## *Hill* *climbing*

Algoritmul *Hill climbing* este un algoritm euristic clasat mai puțin calitativ din punct de vedere al soluției generate. Algoritmul pornește cu generarea unui individ *random* . Contrar algoritmului genetic, evaluează într-o iterație un singur individ, nu o populație întreagă. Așadar în fiecare iterație, algoritmul aplică pe individul curent o mutație generând un nou individ denumit *neighbor* căruia îi este evaluată funcția fitness.

Dacă individul *neighbor* este mai bine adaptat, atunci individul curent este înlocuit cu individul nou creat și algoritmul trece la iterația următoare, iar în caz contrat iterația următoare va avea ca individ pe același ca și în iterația precedentă. Numărul maxim de iterații este prestabilit, așadar algoritmul se va opri când acest prag este atins.

|  |  |
| --- | --- |
| P:\licenta\QR_SearchingAlgorithm\docs\images\hill-climbing-algorithm-in-ai.png  Figura 1 | Acest algoritm poate fi asemănat cu imaginea din Figura 1 unde sunt mai multe dealuri cu înălțimi diferite. Dacă individul generat *random* la început se afla pe unul din aceste dealuri, însă nu pe cel mai înalt, individul va urca maxim până în vârful dealului pe care se află, acest lucru poartă denumirea de maxim local. Ceea ce înseamnă că algoritmul va găsi un punct de maxim, însă există probabilitate mare ca acesta să nu fie maximul global. |

Cu toate acestea, algoritmul este foarte rapid din punct de vedere al timpului de execuție, ceea ce înseamnă că nu poate fi ignorat. Vom putea vedea în capitolul că acest algoritm, rulat pe mai multe seturi de date, oferă rezultate foarte diferite de la un caz la altul. Putem spune că are un comportament destul de imprevizibil.

## *Simulated* *annealing*

Algoritmul *Simulated annealing* a fost introdus în anii ’80 ca o îmbunătățire a algoritmului *Hill climbing.* La fel ca și în cazul algoritmului ce stă la baza acestuia, la o iterație un sigur individ este evaluat. Acestui individ curent i se aplică un operator de mutație generând individul *neighbor.* Diferența dintre cei doi algoritmi constă în faptul că atunci când individul *neighbor* nu este mai bine adaptat, acesta nu este ignorat ci i se aplică o altă condiție de a fi ales pentru iterația următoare.

Acest algoritm introduce noi termeni și anume *temperature* și *cooling rate* ale căror valori sunt prestabilite. Noul proces de acceptare stabilește dacă individul mai puțin adaptat este ales dacă funcția exponențială a raportului dintre diferența de cost dintre *fitness*-ul curent și *fitness*-ul individului *neighbor* și temperatură este mai mare decât o probabilitate generată *random* atunci individul este ales.

|  |  |
| --- | --- |
| P:\licenta\QR_SearchingAlgorithm\docs\images\Functia_exponentiala.png  Figura 2 | În imaginea din Figura 2 se poate observa graficul funcției exponențiale ce are ca bază elementul e.  Una din proprietățile funcției exponențiale este că atunci când baza este mai mare decât 1 și exponentul x mai mic decât 0, această funcția va genera valori cuprinse între 1 și valori care tind spre 0.  Datorită faptului că diferența dintre costuri va fi mereu negativă, funcția exponențială va genera valori ce pot fi raportate la o probabilitate. |

Prin urmare că cu cât algoritmul este mai predispus spre acceptarea unor indivizi mai puțin adaptați, ceea ce înseamnă o temperatură mare, și cu cât diferența de cost este mai mică, cu atât șansele individului *neighbor* de a fi selectat sunt mai mari. Elementul *cooling rate* este utilizat pentru a scădea temperatura de acceptare de la o iterație la alta. Acest algoritm este asemănat cu proprietatea unui obiect din fier de a fi mai maleabil cu cât temperatura este mai mare. Când temperatura atinge valoare 0, atunci algoritmul devine rigid și această nouă șansă a indivizilor mai puțin adaptați dispare.

În comparație cu algoritmul *Hill* *climbing*, această noua metodă de acceptare a indivizilor oferă posibilitatea algoritmului de a plasa punctul de căutare în mai multe direcții diferite, chiar și pe direcția maximului global, reducând șansele algoritmului de a se opri într-un minim local.

## *Building* *blocks*

Algoritmul *Builidng blocks* l-am implementat ca fiind o variantă mai eficientă a algoritmului genetic. Principiul de funcționare este asemănător, pornind de la o populație de indivizi generați *random*, urmând apoi procedurile de selecție și aplicare a operatorilor se obține o nouă populație. De asemenea, la fel ca și la algoritmul genetic, cel mai bun individ este actualizat dacă noua populație oferă un candidat mai adaptat.

Diferența constă în obținerea unui al doilea set de date ajutător populat cu indivizi sub numele de indivizi *schema* H sau *building blocks*. Această abordare presupune construirea unui set de *building blocks* care să reprezinte secțiuni mici de indivizi cu valoarea funcției *fitness* mai mică decât media funcției *fitness* a populației, astfel încât prin combinarea acestor scheme să se obțină cel mai bun candidat.

|  |  |
| --- | --- |
| P:\licenta\QR_SearchingAlgorithm\docs\images\1-matematica-cu-lego-8526-b.jpg  Figura 3 | Pentru a întelege mai bine conceptul de *building blcoks* am atașat în Figura 3 o imagine cu piese *lego.* Exact ca și în exemplul din imagine, schemele pot avea dimensiuni si valori diferite ale funcției de cost, cu cât aceste scheme sunt mai potrivite pentru ceea ce vrem să obținem, cu atât îmbinarea lor va oferi un rezultatul mai bun. |

Exemplu de schemă în reprezentarea binară este: 01\*\*10, aceasta reprezintă orice individ care conține coloanele 1 și 4 și nu conține coloanele 0 și 5, însă am considerat că pe problema discutată în acest document lipsa coloanelor nu este de interes, astfel că doar coloanele reprezentate de biții setați pe 1 sunt luate în considerare, așadar un exemplu de individ ce conține schema data ca exemplu poate fi orice individ ce are în componență coloanele 1 și 4 , cum ar fi 014, 124, 0124, mai exact orice sub-model reprezentat de schema \*1\*\*1\*.

În această abordare a algoritmului genetic nu am utilizat metodele de selecție descrise în subcapitolele 5.1.1 și 5.1.2, ci o nouă procedură ce implică setul auxiliar de scheme descris anterior. Astfel că, pentru generația următoare nu am selectat ce indivizi vor forma noua populație. Aceasta va conține toți indivizii populației precedente, însă nu toți suferă mutații genetice, ci doar cei care nu conțin nici-o schemă în componența lor dar și cei care deși conțin o schemă, probabilitate de a suferi modificări este mai mare. În acest fel cei mai adaptați indivizi cu valoare funcției *fitness* sub medie sunt păstrați așa cum sunt cu o probabilitate mai mare. Odată cu aceștia, se păstrează și setul de scheme adaptate.

Pentru a ști în ce măsură se va păstra setul de scheme din generația curentă în generația următoare, numărul de indivizi ce vor conține una din scheme se poate exprima printr-o funcție *expectation* E:

*E (m (H,t + 1)) ≥ Media f(H,t) / Media f(t) \* m (H,t) \* (1-(Pc \*d(H)/(L-1))*, unde

* Media f(H,t) reprezintă media *fitness*-ului indivizilor din generația t care conțin schema H
* Media f(t) reprezintă media *fitness*-ului tuturor indivizilor din generația t
* m(H,t) reprezintă numărul indivizilor din generația t ce conțin schema H
* Pc reprezintă probabilitatea de *crossover*
* d(H) reprezintă distanța dintre primul și ultimul bit setat pe 1 din schema H
* L reprezintă numărul de coloane aferent schemei H

Așadar dacă schema H este de forma \*1\*\*1\*, aceasta va avea d(H) = 3 și L=6.

Dacă aș fi ales pentru noua populație doar acei indivizi ce conțin una din scheme, atunci algoritmul ar fi avut o probabilitate mai mare să conveargă prematur. întrucât s-ar crea ceea ce se numește un *royale road*, spațiul de căutare chiar de ar duce spre indivizi adaptați peste medie, tot ar fi un spațiu restrâns de căutare. De asemenea când toți indivizii populației ar fi peste medie atunci și setul de scheme ar fi tot mai greu de îmbunătățit.

La fel ca și populația de indivizi, populația de scheme suferă modificări pe parcursul generațiilor. La fiecare generație se obțin în mod aleatoriu noi scheme care sunt peste media populației. Pe aceste scheme nu sunt aplicați operatori genetici, însă schemele sunt combinate între ele, iar dacă îmbinarea lor oferă scheme mai adaptate, acestea vor înlocui vechile scheme. Așadar algoritmul asigură îmbunătățirea schemelor la fiecare generație.

De asemenea pentru a identifica cel mai bun candidat, nu doar populația de indivizi este verificată ci și setul de scheme. În cele mai multe cazuri cel mai bun candidat este prezent în setul de scheme datorită îmbinării acestora pe parcursul generațiilor.

Tocmai datorită faptului că această abordare a algoritmului genetic duce căutarea într-un spațiu de indivizi care sunt adaptați mereu peste medie, se consideră un algoritm ideal ce nu poate exprima o evoluție realistă a generațiilor de indivizi.

## Algoritmi cu restricție

O altă abordare a algoritmilor euristici definită de mine sunt algoritmii cu restricție. Această metodă este compatibilă cu toți algoritmii descriși în acest capitol, mai puțin cu algoritmul *Building* *blocks*, acest aspect îl voi discuta în cele ce urmează.

Am ales această denumire întrucât în procesul de generare a populației am stabilit o condiție astfel încât indivizii generați să aibă un număr fix de coloane ce reprezintă un procent din numărul total de coloane al modelului. Acesta este prestabilit înainte de execuția algoritmilor. În plus, în procesul de modificare a *genotype*-ului fiecărui individ am utilizat doar operatorii genetici care asigură păstrarea numărului de coloane stabilit. Am implementat o singură combinație de operatori cu această proprietate, și anume operatorul de *crossover* 1 point adaptat împreună cu operatorul de mutație *interchanging* care sunt descriși în subcapitolul 4.3.

Această metodă nu este compatibilă cu algoritmul *Building blocks* din cauza setului auxiliar de indivizi *schema.* Cum am menționat și în subcapitolul *Building* *blocks* , setul de scheme este îmbunătățit de la o generație la alta prin îmbinarea schemelor, așadar schemele rezultate vor avea desigur un număr de coloane mai mare, iar dacă îmbinarea schemelor nu s-ar aplica atunci esența algoritmului s-ar pierde.

Desigur că limitarea numărului de coloane reduce spațiul de căutare destul de mult și mai ales pe un set de date real, este imposibil ca acest procent să fie cunoscut. Cu toate acestea este forte util în procesul de configurație al algoritmilor, atunci când aceștia din urmă sunt rulați pe fișiere de test unde cel mai bun subset este cunoscut în prealabil. Prin urmare această abordare ajută la compararea algoritmilor euristici despre care voi vorbi mai în detaliu în capitolul 9.

# Euristici în combinație cu metoda eficientă

Text descriere

## Integrare

Text

## Performanțe

Text

# Implementare

Text

## Structura generală

## Fișier de configurație

## Funcții de bază

# Reprezentarea grafică a datelor și elemente de statistică

Pentru a identifica calitatea algoritmilor prezentați în această lucrare am realizat un program care să genereze seturi de date *random*, în care se pot configura parametrii care pot influența rezultatele unui algoritm. De asemenea, pentru a face o comparație între outputuri am realizat o aplicație care utilizează elemente de statistică. Și în cele din urmă, pentru a vizualiza mai ușor eficiența și calitate algoritmilor, am realizat un program care să reprezinte grafic statisticile obținute.

## Generare seturi de date

Dat fiind faptul că algoritmii necesită un set de teste, seturile de date generate de aplicație sunt utilizate ca și *banchmark*-uri. Pentru a genera astfel de seturi de date am construit o regresie liniară. Valorile criteriilor obținute împreuna cu setul de variabile dependente este dat ca input mai departe algoritmilor genetici, iar valorile celui mai bun sub-model sunt folosite mai departe în aplicația pentru compararea cu outputurile algoritmilor euristici.

Pentru că problema pe care o rezolvă acești algoritmi este de a căuta cel mai bun sub-model de regresie, unul din parametrii de input potriviți este numărul de coloane al setului de date împreună cu numărul de coloane al celui mai bun sub-set. Am păstrat un raport de 50% dintre aceste doua valori. Cu cât modelul este mai mare, cu atât timpul de execuție va crește pentru unii algoritmi. De asemenea un model mai mare înseamnă un spațiu mai mare de căutare.

Un alt parametru semnificativ este numărul de observații, cu cât acesta este mai mare, la fel ca și în cazul numărului de coloane, acesta va crește complexitatea calculelor și timpul de execuție.

Ultimul parametru de configurație dar și printre cei mai importanți îl reprezintă valoarea deviației standard a valorilor din setul de date. Cu cât datele sunt mai distanțate de medie, cu atât cel mai bun sub-set este mai afundat în spațiul de căutare, deci mai greu de identificat.

În continuare pe baza acestor parametri am construit un set de date. Pentru început am generat o matrice de m observații și n coloane de valori cu deviația standard dată ca input. La pasul următor am generat *random* un set de indecși de coeficienți de dimensiunea numărului de coloane dat ca input al celui mai bun sub model dat. Acest set de indecși reprezintă de fapt ce coloane formează cel mai bun sub-model. Întrucât problema presupune găsirea unei soluții aproximate, în continuare am generat *random* un set de valori de eroare urmând aceeași deviație standard din input. Mai departe am calculat setul de valori al vectorului de variabile dependente după formula regresiei liniare:

*Y = A\*β + ε*

## Elemente de statistică

Pentru a stabili eficiența algoritmilor am realizat o aplicație care compară diferite elemente ale outputului algoritmilor genetici cu outputul seturilor generate de date, la care voi face referire cu numele de *banchmark*.

Unul din aceste elemente îl constituie coloanele celui mai bun sub-set. Am numit această metrică *match all columns*, care va avea valoarea 1 în cazul în care setul de coloane al algoritmilor euristici corespunde în totalitate cu setul de coloane al *banchmark*-ului, în caz contrar aceasta va avea valoarea 0.

Deoarece de multe ori algoritmii euristici oferă soluții foarte apropiate de cel al *banchmark*-ului, pentru a nu fi anulat din metrică un output din cauza unei coloane sau a mai multor coloane care sunt pe lângă valorile celui mai bun subset, următorul element din metrică îl reprezintă raportul dintre numărul de coloane găsite de algoritmii euristici față de numărul de coloane al celui mai bun subset. În acest caz, daca algoritmul găsește toate coloanele celui mai bun subset dar și alte coloane din setul mare de date, raportul va avea valoarea 1, colonele care sunt în plus nu vor penaliza soluția algoritmului. De asemenea în cazul în care algoritmul găsește doar o submulțime de coloane din cel mai bun sub-model, valoare metricii nu va fi 0 ca în cazul anterior, ci va fi valoare raportului dintre numărul de coloane care sunt la fel în cele doua soluții:

*Match some columns = |{coloanebest}∩{coloaneeuristic}| / nbest*unde,

nbest reprezintă numărul de coloane al celui mai bun sub-set.

Un alt element de comparație îl reprezintă valoare funcției AIC . Metrica *report AIC* este egală cu diferența celor doua valori AICbest  și AICeuristic în raport cu AICbest. Această metrică exprimă numeric distanța dintre eroarea sub-modelului calculată de algoritmul euristic și eroare *banchmark*-ului. Cu cât acest raport este mai mare cu atât algoritmul euristic oferă o soluție cu mai mult zgomot.

În obținerea acestei metrici am întâmpinat o mică discrepanță între valorile AIC-ului al celor doua outputuri. Atunci când soluțiile prezintă același set de coloane, așteptarea ar fi ca valorile AIC să fie de asemenea aceleași, însă acest lucru nu se întâmplă datorită faptului că sunt calculate separat, oferind aceeași soluție pentru care însă vectorul de eroare împreună cu vectorul de coeficienți diferă.

Y = A\*β + ε unde,

Prin culoarea verde am exprimat faptul că elementele ecuație sunt aceleași pentru ambele soluții, iar cele reprezentate prin culoarea roșu am exprimat faptul ca acestea pot avea valori împreună valori diferite în cele doua soluții.

Chiar dacă această discrepanță va aduna un raport de diferență chiar și în cazul în care soluțiile sunt aceleași, acesta va fi mereu mai mic decât valorile raportului atunci când soluțiile au elemente diferite. De asemenea această metrică penalizează mai mult un output euristic dacă acesta conține coloane diferite de setul coloanelor al celui mai bun sub-set, decât atunci când outputul algoritmului conține toate coloanele celui mai bun sub-set dar mai conține si alte coloane în plus.

Tot ca și metrică pentru compararea algoritmilor, este timpul de execuție. De data aceasta comparația se face exclusiv între outputurile euristice, fără un *banchmark*.

Toate aceste metrici descrise mai sus le-am utilizat pentru a crea mai multe spații de comparație. În toate cazurile, am realizat media aritmetică a metricilor pentru a stabili un output final. Așadar comparația algoritmilor se va face prin:

* modificarea numărului de seturi de date, verificarea metricilor pe mai multe seturi de date
* modificarea dimensiunii setului de date și a celui mai bun sub-set
* modificare valorii deviației standard
* modificarea configurațiilor algoritmilor euristici, diferite combinații între operatori și metode de selecție, în cazul algoritmului genetic

## Grafice

Pentru a vizualiza evoluția algoritmilor euristici, am implementat doua aplicații pentru reprezentarea outputurilor.

O aplicație afișează pe un grafic outputurile unui singur algoritm euristic, în care se poate vizualiza la ce iterație algoritmul actualizează cel mai bun individ pe măsură ce înaintează în generații împreună cu setul de coloane și numărul de coloane din set. Am utilizat acest tip de grafic pentru seturile reale de date.

Cea de-a doua aplicație am implementat-o pentru seturile generate în care am reprezentat toate metricile obținute în comparație cu un *banchmark.* În această aplicație am implementat doua tipuri de grafice, unul în care am reprezentat outputurile celor 4 algoritmi și un grafic în care am reprezentat outputurile unui singur algoritm cu toate configurațiile care pot fi setate. Acesta din urmă primește ca parametru algoritmul pentru care se vrea realizare graficului împreună cu setul de configurații.

Fiecare grafic punctează în mod diferit eficiența și evoluția algoritmilor, împreună constituind un set complet de analiză a acestor euristici.

# Comparația algoritmilor bazată pe outputuri

În acest capitol voi descrie comportamentul algoritmilor pe seturi reale de date dar și pe seturi de date generate. Pentru fiecare algoritm în parte voi selecta un set de configurații în urma căruia voi obține un grafic la care voi adăuga un set de explicații și observații.

## Comportamentul algoritmilor pe seturi de date reale

În reprezentarea outputurilor pe seturile reale de date, pentru a nu aglomera graficul foarte mult, am ales să construiesc într-o singură imagine câte 3 grafice mai mici au axay comună, pe care am plasat valorile funcției AIC. De altfel, pentru a nu aglomera graficul foarte mult, am afișat doar outputurile celui mai bun candidat pe măsura ce se îmbunătățește.

Așadar în graficul de sus al imaginii am reprezentat evoluția valorii AIC pe parcursul generațiilor algoritmului, unde am trasat o linie de la prima generație, la ultima generație în care cel mai bun subset a fost actualizat.

În graficul din mijloc am reprezentat pe axax pentru fiecare valoare AIC, numărul de coloane al sub-modelului reprezentativ unde am marcat cu o steluță roșie numărul de coloane al celui mai bun subset de la ultima actualizare.

În graficul de jos al imaginii am reprezentat pe axax combinația de coloane a sub-modelului reprezentat de funcția AIC unde am marcat cu steluțe roșii combinația de coloane al celui mai bun individ de la ultima actualizare.

Unul dintre seturile reale de date este *Los Angeles ozone pollution,* în care sunt prezente 13 variabile de regresie peste care s-au obținut 178 de observații, câte o observație pe zi.

|  |
| --- |
| Figura 4 |

Pentru fiecare set de date în parte, algoritmul poate avea un comportament bun dacă se aleg valori corespunzătoare pentru parametrii de configurație. Pentru algoritmul genetic am setat o populație de indivizi de 60% din numărul total de observații si valoarea de *converge* de 50.

|  |  |
| --- | --- |
| Figura 5 | În imaginea din Figura 5, am rulat algoritmul cu operatorii genetici *flip mutation* și *crossover 1point adaptat,* iar ca metodă de selecție am ales *Roulette wheel* . Se poate observa cum algoritmul genetic are o evoluție treptată, deoarece operatorii genetici utilizați nu aduc modificări extreme indivizilor copil. Algoritmul pornește cu o populație al cărei cel mai bun individ are un număr de 9 coloane 2,3,5,6,7,8,10,11,12 și ajung pe parcursul generațiilor la un individ format din 2 coloane chiar din primele 10 iterații. |

Se poate observa cum coloana 4 care formează cel mai bun sub-set se pierde intr-o generație însa datorita mutațiilor reapare la următorul *update* al celui mai bun sub-set. De asemenea o altă observație importantă relativă la imaginea din Figura 5 o constituie faptul că această coloană cu indexul 4 apare în componența celor mai bine clasate sub-modele. De exemplu sub valoare AIC-ului de 180, coloana 4 este cea mai răspândită. Așadar pe acest set de date poluarea cu ozon este datorată în cea mai mare proporție de media maximă/pe oră a concentrației de ozon, urmată de temperatură.

Pentru aceste seturi de date reale unde dimensiunea spațiului de căutare permite evaluarea tuturor seturilor de date într-un timp relativ scurt, am utilizat ca și *banchmark* outputul generat de metoda exhaustivă, descrisă în subcapitolul 3.1, pentru a avea convingerea că algoritmul generează cel mai bun sub-set. Desigur că acest lucru este limitat la dimensiune setului de date.

|  |  |
| --- | --- |
| Figura 6 | Figura 7 |

În imagine din Figura 6 am atașat outputul algoritmului genetic executat cu operatorii genetici *flip mutation* și *RRC crossover* cu o probabilitate de 50%*.* Se poate observa că algoritmul găsește cel mai bun subset chiar de la prima iterație, iterația 0 reprezintă generarea *random* a populației, daca cel mai bun subset este identificat la acest moment, atunci putem spune că algoritmul genetic nu are nici-un merit.

În imagine din Figura 7 am atașat outputul algoritmului genetic executat cu operatorii genetici *reversing mutation* și *crossover 1point simple.* Se poate observa că parcursul celui mai bun subset este destul de haotic, în sensul că setul de coloane diferă foarte mult de la o actualizare la alta, ceea ce demostrează că algoritmul genetic dă rezultate mai rapide când populația de indivizi este mai diversificată.

În cele ce urmează voi descrie comportamentul algoritmul *Hill climbing* pe același set de date. Am setat un număr de 1000 de iterații.

|  |  |
| --- | --- |
| Figura 8 | În imaginea din Figura 8 am rulat algoritmul cu operatorul *flip mutation*.  În graficul de jos, se poate observa cum algoritmul renunță pe rând la câte o coloană din setul inițial al celui mai bun sub-model. Așadar putem face un clasament al coloanelor din setul de date, pe ultimul loc fiind coloana 1, care reprezintă în ce lună din an au fost preluate datele.  Abia la iterația 30 algoritmul reușește să elimine și coloana 11 sin sub-set. |

|  |  |
| --- | --- |
| Figura 9 | În imaginea din Figura 9 am ales ca operator de mutație metoda *reversing.*  Se poate observa că acest algoritm din moment ce depinde doar de operatorul de mutație este important ca acesta din urmă să aibă libertatea de a modifica cât mai mult individul copil.  Deși outputul nu este printre cele obținute anterior aproximând eronat soluția problemei, se observă foarte bine modul de funcționare al operatorului de mutație. Vedem cum de la index-ul 1 genele iau valoarea opusă la următoare actualizare în cazul primei actualizări. |

În continuare, am rulat algoritmul *Simulated Annealing*. Am păstrat numărul de iterații de 1000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | Figura 10 | | Figura 11 | | Figura 12 | | În imaginile din figurile alăturate am vrut să pun în evidență pentru algoritmul *Simualted Annealing* importanța valorii temperaturii influențată de parametrul *cooling rate*.  Așadar pentru a face acest lucru, am rulat algoritmul de 3 ori cu același operator de mutație și anume *flip mutation* , însă pentru fiecare execuției am setat valori diferite pentru *cooling rate.*  În Figura 10 parametrul are valoare 0.33, în Figura 11 valoare este de 0.86 iar în Figura 12 valoarea este de 0.95. Se poate observa ca numărul de iterații este mai mare atunci când *cooling rate* este mai mare, deoarece acesta păstrează timp de mai multe iterații o temperatură mare, care să permită acceptarea indivizilor chiar și mai puțin adaptați. |

Ultimele observații le voi face pe outputurile obținute în urma execuției algoritmului *Building blocks.* Am configurat valoare *converge*-ului la 10, numărul de coloane al schemelor la 60% din totalul de coloane al setului de date, iar probabilitatea de a aplica o mutație genetică am setat-o la 30%.

|  |  |
| --- | --- |
| Figura 13 | În imaginea din Figura 13 am afișat outputul algoritmului utilizând operatorii genetici *flip muation* și *1 point crossover* adaptat.  Se poate vedea că la prima iterație a algoritmului acesta găsește doua variante diferite ale celui mai bun sub-set. Acest lucru se întâmpla deoarece cel mai bun sub-set este evaluat o data în populația de indivizi din generația curentă și a doua oară în setul de scheme.  Pe acest caz specific unde cel mai bun subset este alcătuit dintr-o singură coloană, calitatea unui astfel de algoritm nu se poate evidenția deoarece nu există o îmbinare de coloane de făcut.  Cu toate acestea este o posibilitate foarte mare ca algoritmul sa obțină cel mai bun subset din prima iterație datorită modului cum este implementată stabilirea setului de scheme. |

## Comportamentul algoritmilor pe seturi de date generate

Text

## Avantaje versus dezavantaje

Text

## Performanțe

Text

1. RRC Este o abreviere a metodei *Random Respectful Crossover* [↑](#footnote-ref-1)
2. RSS – *Residual Sum of Squares* [↑](#footnote-ref-2)
3. AIC - *Akaike information criterion* [↑](#footnote-ref-3)