UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Un studiu comparativ al algoritmilor euristici pentru selecția unui sub-model de regresie

propusă de

Paula Roxana Tanasă

Sesiunea: *Iulie, 2019*

Coordonator științific

Conf. Dr. Cristian Gațu

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Un studiu comparativ al algoritmilor euristici pentru selecția unui sub-model de regresie

Paula Roxana Tanasă

Sesiunea: *Iulie, 2019*

Coordonator științific

Conf. Dr. Cristian Gațu

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conţinutului lucrării de licenţă

Subsemntatul(a) ……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …….. specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul:

_____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării falsificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie

răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „*Un studiu comparativ al algoritmilor euristici pentru selecţia unui sub-model de regresie*", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

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

Iaşi, 26*.06.2019*

Absolvent *Paula Roxana Tanasă*

_____

Introducere

Pentru lucrarea de licență mi-am propus să realizez o aplicație care să compare câteva implementări diferite de algoritmi euristici care rezolvă problema căutării celui mai bun subset de regresie liniară. Pentru a genera soluții pentru un set de date real, aplicația primește ca input tipul de algoritm pentru care se dorește studierea comportamentului, împreună cu un set restrâns de configurații și fișierul cu setul de date, iar outputul va fi un grafic reprezentativ al rezultatului generat de acesta. Pentru a studia comportamentul algoritmilor pe un set de date generat cu anumite configurații stabilite de mine, am implementat un script în *Python* care apelează aplicația cu configurații diferite, în urma cărora se va obține un grafic reprezentativ al statisticilor pe rezultatele obținute de algoritmii euristici.

Studiul pe algoritmi euristici a început încă din anii ’50 nu este un subiect nou pentru zilele noastre, cu toate acestea am realizat o aplicație *software* în limbajul C în așa fel încât să fie ușor de înțeles și de configurat, de asemenea datorită graficelor implementate este ușor de făcut o comparație între algoritmi.

În primul capitol voi vorbi despre regresia liniară și modul de aplicare a descompunerii QR utilizată în stabilirea calității unui algoritm euristic.

În următorul capitol voi prezenta resursele software utilizate în implementarea aplicației.

În cel de-al treilea capitol voi prezenta o abordarea naivă a căutării celui mai bun subset, pe care am utilizat-o pentru a verifica outputul algoritmilor euristici dar și pentru a pune în evidență eficiența acestora.

În capitolul 4 voi prezenta tot ce înseamnă concepte și elemente de bază ale acestui tip de algoritmi.

În capitolul 5 voi descrie fiecare algoritm în parte, împreună cu elementele care se aplică în mod individual, unde va fi cazul.

În capitolul 6 voi prezenta modul de implementare al aplicației împreună cu setul de configurații pentru rularea algoritmilor. De asemenea, voi atașa o diagramă reprezentativă.

În capitolul 7 voi prezenta câteva elemente de statistică utilizate în comparația algoritmilor, precum și modul de reprezentare al rezultatelor algoritmilor pe grafice.

În prima parte a capitolului 8 voi discuta despre rezultatele algoritmilor pe un set real de date, iar în continuare voi prezenta comportamentul acestora pe diferite seturi de date generate. Mai întâi voi face *tuning* pe fiecare algoritm în parte după care voi compara toți algoritmii fiecare având configurația pentru care acesta generează cele mai bune rezultate.

Cuprins

1 Descrierea problemei

În acest capitol voi discuta despre problema pe care am ales-o pentru a compara rezultatele algoritmilor euristici.

1.1 Regresie liniară

Regresia liniara este o problemă des utilizată în statistică dar și în algoritmi euristici sau *machine learning*. Aceasta este utilizată pentru a oferi o predicție asupra valorii outputului, atunci când noi seturi de date sunt introduse în studiul unui model.

Așadar această problemă primește ca input un set de variabile de mediu care în cele mai multe cazuri reprezintă un set de factori ce influențează un anumit comportament. Depinde foarte mult de problema pe care o rezolvă, fie vrem să aproximăm modul de funcționare al componentelor unei biciclete electrice în așa fel încât acesta să aibă o autonomie cât mai îndelungată, fie vrem să aflăm cum sunt aduse modificări ale ADN-ului uman ce cauzează boala de cancer, domeniul este la libera imaginație. De asemenea, în input există și un set de variabile dependente, care reprezintă numeric comportamentul modelului. Având aceste informații regresia liniară găsește o corelație în așa fel încât să poată asocia o predicție a comportamentului în condițiile în care primește un nou set de valori pentru variabilele de mediu. Această corelație este reprezentată printr-un set de coeficienți de valori numerice.

Ca și reprezentare fiecare set de variabile de forma $x_{00}*\beta_0 + x_{01}*\beta_1 +...+x_{0m}*\beta_m + \varepsilon_0 = y_0$ reprezintă o observație, unde $x_{00}... x_{0m}$ reprezintă primul set de variabilele de mediu, $\beta_0...\beta_m$ reprezintă vectorul de coeficienți. Întregul set de date poate fi reprezentat ca o matrice de n astfel de observații. Așadar forma generală a unei combinații liniare este:

$$Y = X*\beta + \varepsilon$$

Deoarece acest tip de problemă face doar o predicție, este calculat și un set de valori numit vectorul de eroare. Așadar, pentru fiecare coeficient aproximat, există o valoare de eroare asociată acestuia. Pentru a stabili calitatea unei predicții, putem să analizăm acest set de erori. Una dintre cele mai utilizate metode este ceea ce se numește *Residual sum of squares*, prescurtat RSS, care face o media a erorii, mai exact calculează distanța euclidiană dintre soluția aproximată și soluția dată ca input. Cu cât această distanță este mai mică, cu atât predicția este mai bună. RSS se calculează după formula:

$$RSS = \sum_{i=1}^{n} (Y - X * \beta')^2,$$

unde β' reprezintă vectorul de coeficienți estimat.

Având această posibilitate de a calcula o predicție, mai departe algoritmii euristici sunt utilizați pentru a căuta cel mai bun set de coeficienți dintr-un număr mare de predicții, imposibil de a fi atins cu o un algoritm exhaustiv într-un timp convenabil. Această căutare euristică pornește cu un vector de coeficienți absolut *random* selectat, care generează o predicție la fel, absolut *random*, pentru care calculează valoarea RSS. În continuare prin

diferite procedee de a trece la un alt set de coeficienți, se vor obține tot mai multe valori de RSS din care algoritmul euristic va alege acel set care generează cea mai mică valoare RSS.

Acest proces de căutare poate fi influențat prin diferite metode care oferă posibilitatea de a evalua posibile soluții din *clustere* diferite sau metode care influențează căutarea pe un singur *cluster* de soluții, aceste metode diferă de la un algoritm la altul. În general o evaluare mai amplă a spațiului de soluții posibile oferă aproximări mai puțin eronate, însă cu costul de a crește timpul de căutare.

1.2 Descompunerea QR

În acest subcapitol voi descrie legătura dintre regresia liniară și descompunerea QR. Așadar regresia liniara oferă un set de coeficienți care aproximează soluția sistemului de regresie liniară cu un vector asociat de eroare. Pentru implementare am calculat acest vector de eroare folosindu-mă de librăria *GNU*, prin utilizarea descompunerii setului de date ce corespunde variabilelor independente notat cu matricea $X_{m\times n}$, unde m reprezintă numărul de observații, iar n reprezintă numărul de coloane, adică numărul de factori peste care se fac observațiile. Cu toate că am utilizat librăria pentru a obține setul de erori, voi descrie în cele ce urmează cum este obținut.

Așadar, descompunerea QR reprezintă o descompunere a datelor din setul de intrare sub o formă care facilitează calculele matematice pentru a căuta cel mai bun set de coeficienți. Așadar această descompunere este de forma:

$$X = Q \times R,$$

unde $Q_{m\times m}$ este o matrice ortogonală ceea ce înseamnă că principala proprietate de interes este faptul că $Q \times Q^T = I$, iar $R_{n\times n}$ este o matrice superior triunghiulară. Așadar combinația liniară poate fi scrisă sub forma:

$$Y = Q \times R \times \beta + \varepsilon,$$

În continuare putem renunța la vectorul eroare ε pentru a defini forma generală a funcției RSS. Așadar dacă înmulțim ecuația cu transpusa matricei Q, vom obține:

$$Q^T \times Y = R \times \beta \equiv Q^T \times Y = Y',$$

unde primele n linii din Y' corespund vectorului Y_1, iar restul până la m corespund vectorului Y_2. Așadar,

$$\beta' = R^{-1} \times Q^T \times Y \equiv \beta' = R^{-1} \times Y'_1$$

Astfel că RSS, calculat cu ajutorul descompunerii QR, este de forma:

$$Err = Q^T \times (Y - X * \beta') = Q^T * (Y - Q \times R \times R^{-1} \times Y_1) = Y'_1 - Y_1 = Y_2,$$

ceea ce rezultă faptul că, RSS poate fi foarte simplu calculat după ce se obține înmulțirea $Q^T \times Y$, în urma căreia se obține noul vector Y', pentru care se calculează suma pătratelor a ultimelor m valori.

2 Resurse utilizate

În acest capitol voi menționa librăriile utilizate în implementare.

2.1 Descompunere QR

Datorită faptului că am ales ca și limbaj de programare C, pentru a obține într-un mod mai simplu datele necesare peste care am realizat comparația algoritmilor genetici, am utilizat funcțiile descrise la capitolul Linear Algebra din pachetul oferit de *GNU Scientific Library*. Pentru a putea utiliza aceste metode am compilat librăriile *cyggsl.dll* și *cyggslcblas.dll* cu colecția de *tool*-uri *Cygwin*, compatibilă cu compilatorul gcc.

2.2 Realizarea graficelor

Pentru analiza outputurilor obținute de algoritmii euristici, am realizat câteva grafice descriptive utilizând pachetul *Matplotlib* oferit de limbajul de programare *Python*.

3 Rezolvarea problemei

3.1 Introducere

În acest capitol voi descrie modul de rezolvare al problemei căutării celui mai bun subset dintr-un set de valori prin metoda exhaustivă.

3.2 Descrierea metodei exhaustive

Pentru a obține cel mai bun subset de valori, este nevoie de a cunoaște și evalua tot spațiul de date. Pentru a face acest lucru, se iterează prin toate subseturile posibile, în timp ce actualul subset *best* este comparat și actualizat în condițiile în care la una din aceste iterații se obține o soluție cu o eroare mai mică decât cea a subsetului *best*.

Așadar, pentru a itera prin toate subseturile posibile, diferite ca și conținut de coloane, am utilizat elemente de combinatorică. Așadar am obținut toate combinațiile posibile de indecși ale coloanelor, combinații de dimensiuni de la 1 până la numărul total de coloane. În continuare am creat câte o matrice de criterii corespondentă combinației de indecși de coloane din setul de date din input. Spre exemplu subsetul {1,4,6} va constitui matricea cu coloanele 1, 4 și 6 din datele de input. În continuare pe fiecare matrice corespondentă unei combinații de coloane, am obținut valoarea RSS[1] discutată în subcapitolul 1.2 și mai apoi valoarea funcției AIC[2] pe care le voi discuta în capitolul 4.4.

3.3 Concluzie

Metoda exhaustivă, deși oferă cel mai bun subset la finalul execuției, consumă foarte mult timp de execuție atunci când numărul de coloane crește, așadar pentru un set de 5 criterii vom avea doar 2^5-1 ceea ce înseamnă 31 de subseturi diferite, însă pentru un set de 30 de criterii ar fi un număr de 2^{30}-1 adică 1 073 741 824 de posibile soluții.

Cu toate acestea, am utilizat această metodă ca și *benchmark* pentru seturile de date reale, cu un număr de coloane mai mic de 20, pentru a evalua calitatea outputului oferit de algoritmii euristici.

[1] *Residual sum of sqaures*
[2] *Akaike information criterion*

4 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, însă 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.

4.1 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.

4.2 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

4.3 Operatori

Operatorii sunt algoritmi aplicați peste sub-modele cu scopul de a genera alte sub-modele noi, diferite , însă 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*.

4.3.1 *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*[3] *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.

[3] RRC Este o abreviere a metodei *Random Respectful Crossover*

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 probabiliate$_1$ a primului individ-copil aleatoriu mai mare de 50% și o probabiliate$_2$ 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 probabiliate$_1$ a primului individ copil aleatoriu mai mare de 50% și o probabiliate$_2$ a celui de-al doilea individ copil tot mai mare de 50%, însă pentru ultimele 2 gene probabiliatea$_2$ 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.

4.3.2 *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 aleatoare 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*.

4.4 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. 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[4]. 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:

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

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

4.5 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.

[4] AIC - *Akaike information criterion*

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.

5 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 2^{500} de combinații pentru a determina cel mai bun sub-set de proprietăți.

5.1 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 însă 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.

5.1.1 *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 aleagă 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.

5.1.2 *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.

5.2 *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.

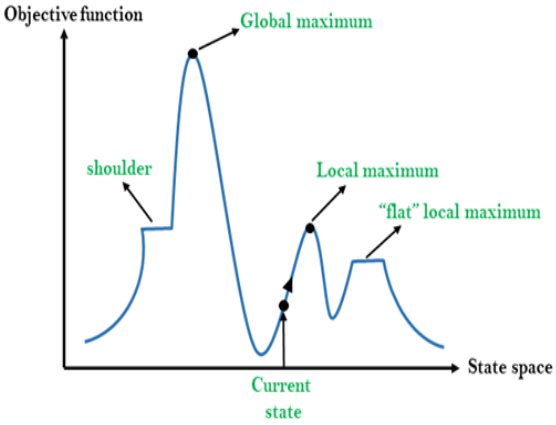


Figura 1 – O reprezentare a spațiului de căutare pentru algoritmul *Hill Climbing*

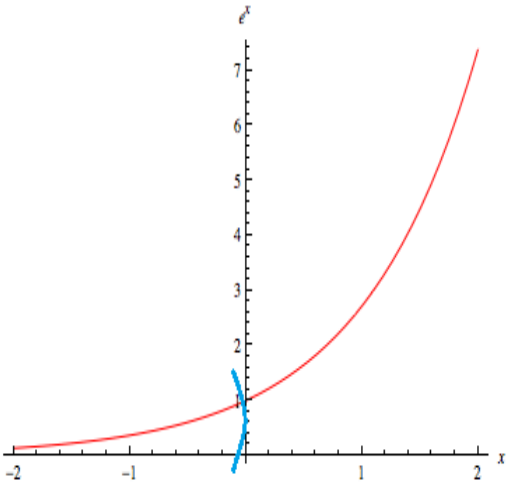
Acest algoritm poate fi asemănat cu imaginea din Figura 1 unde sunt mai multe dealuri cu înălțimi diferite. Dacă individul generat *random* la început se afla pe unul din aceste dealuri, însă nu pe cel mai înalt, individul va urca maxim până în vârful dealului pe care se află, acest lucru poartă denumirea 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 8 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.

5.3 *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.



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.

Figura 2 – Reprezentarea grafică a funcției exponețiale pentru baza e și exponent negativ

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.

5.4 *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.

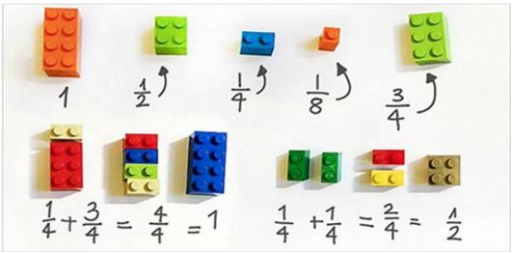


Figura 3 – Un set de piese de lego ce pot fi asemănate cu strategia de *building blocks*

Pentru a înțelege 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:

$$E\,(m\,(H,t+1)) \geq Media\ f(H,t) / Media\ f(t) * m\,(H,t) * (1-(P_c * d(H)/(L-1))$$, 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
- P_c 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.

5.5 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 8.

6 Implementare

În acest capitol voi prezenta structura aplicației și comportamentul acesteia cu ajutorul unor diagrame. Tot în acest capitol voi prezenta fișierul de configurație al algoritmilor euristici și un set de funcții importante.

6.1 Structura generală

Pentru a avea o aplicație modulară și ușor de înțeles am împărțit-o în 5 fișiere dedicate pentru un anumit set de operații după cum se poate vedea în Figura 4.

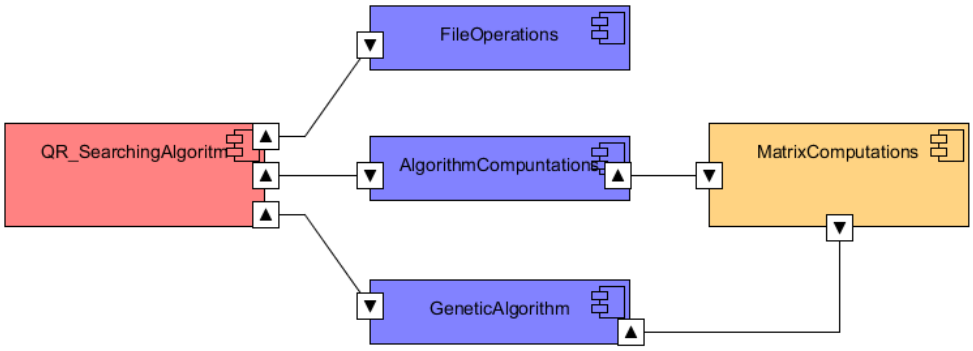


Figura 4 - Diagramă de compoziție

Așadar în fișierul *QR_SearchingAlgoritm* am implementat o funcție de apel a algoritmilor în funcție de parametrii de intrare. În fișierul *FileOperations* am adăugat un set de operații pentru a valida fișierul de input, de asemenea tot aici obțin dimensiunea datelor și construiesc matricea de input. Metoda exhaustivă am implementat-o în fișierul AlgorithmComputations împreună cu calcularea funcției RSS. În fișierul MatrixComputations am implementat atât operații pe matrici cât și pe vectori utilizate deseori în obținerea subseturilor de date, iar în fișierul GeneticAlgorithm am implementat tot ce tine de operații utilizate în algoritmii euristici de la operatori genetici la implementarea algoritmilor. Aplicația urmează pașii următori:

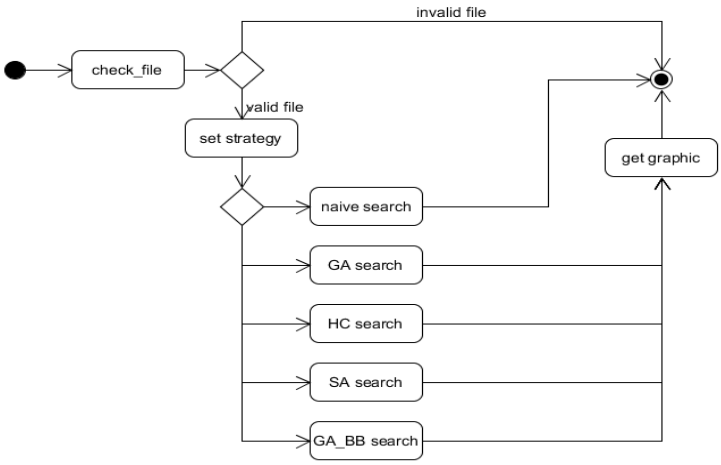


Figura 5 - Diagramă de activitate

6.2 Fișier de configurație

În fișierul de configurație am definit un set de parametrii pentru fiecare algoritm euristic în parte, cu câteva excepții unde anumiți parametrii sunt valabili pentru maxim 2 algoritmi. Am ales să îi configurez dintr-un fișier separat și nu în setul de parametrii de intrare deoarece nu necesită să fie modificați la fiecare rulare a aplicației.

```
/*==========Genetic Algorithm==========*/
#define PERCENTAJE_OF_CHROMOSOMES 100u //70
#define CONVERGE 100u
#define FIXED_NR_GENES 0u
#define PERCENTAJE_OF_GENES 50u//55u
/*=====================================*/
```

Figura 6 - Setul de parametrii de configurație pentru GA

După cum se poate observa în imaginea din Figura 6, pentru algoritmul genetic am definit un procent aferent numărului de indivizi dintr-o populație. Acest procent este calculat în funcție de numărul de observații din setul de date.

Al doilea parametru este valoare de *converge* utilizat în condiția de oprire a algoritmului. Parametrul *fixed_number_genes* determină dacă indivizi din populația generata vor avea un număr fix de coloane, iar ultimul parametru calculează acest număr fix de coloane, în funcție de numărul total de coloane din setul de date. Ultimii doi parametrii sunt utilizați în metoda descrisă în subcapitolul 5.5.

```
/*=====================================*/
/*==========Tournament Selection==========*/
/*=====================================*/
#define PERCENTAJE_OF_TOURNAMENT_K 30u
/*=====================================*/
```

Figura 7 - Configurație *Tournament Selection*

Tot pentru algoritmul genetic am definit un parametru de configurație pentru metoda de selecție *tournament* după cum se poate vedea în imaginea din Figura 7.

Prin acest parametru setez procentul de indivizi din populația curentă care vor participa într-un turneu de selecție dintre care cel mai bun candidat este ales.

```
/*=====================================*/
/*==========Simulated Annealing==========*/
/*=====================================*/
#define TEMP 1000000 //10000 100000 100000 // 10000000
#define COOLING_RATE 0.30//0.003 0.98 0.330 0.729 0.85/ 0.86 /0.95 /0.75
/*This parameter is also used in Hill Climbing*/
#define NR_ITERATIONS 100u
/*=====================================*/
```

Figura 8 - Configurație Algoritmul *Simulated Anealing*

În imaginea din Figura 7 am definit pentru algoritmul *simulated anealing* valoarea temperaturii și a gradului de răcire a acesteia de la o iterație la alta. De asemenea am definit un număr de iterații pe care l-am utilizat și în cadrul algoritmului *hill climbing* în stabilirea condiției de oprire.

```
/*=====================================*/
/*==========BUilding BLocks==========*/
/*=====================================*/
#define PERCENTAJE_OF_SCHEMAS 90u //70
#define OPERATORS_PROBABILITY 20
/*=====================================*/
```

Figura 9 - Configurație Algoritmul *Building Blocks*

În ceea ce privește algoritmul *Building Blocks*, am definit un procent pentru a stabili numărul de scheme utilizate. În funcție de acesta am calculat și numărul de coloane aferent schemelor la momentul generări lor.

Cel de-al doilea parametru definit reprezintă probabilitatea de aplicare a operatorilor genetici pe acei indivizi adaptați peste medie. Restul parametrilor cum ar fi, valoare de *converge* și numărul de indivizi ai populație este setat cu parametrii descriși la algoritmul genetic.

6.3 Funcţii de bază

În acest subcapitol voi prezenta câteva funcții mai importante utilizate în implementarea algoritmilor euristici.

```
typedef struct{
    boolean  selected;
    double   fitness_value;
    double   RSS;
    double   selection_probability;
    gsl_vector*  columns;
    gsl_vector* bit_columns;

}T_INDIVIDUAL;
```

Figura 10 - Structura unui individ

În imaginea din Figura 10 se poate observa modul de reprezentare al unui individ. Câmpul *selected* este utilizat doar în algoritmul genetic și *building blocks* în procedurile de selecție. De asemenea, am păstrat doua tipuri de reprezentare a datelor, o data cu un set de vectori ce păstrează indecșii coloanelor și a doua oară cu un set de vectori ce păstrează coloanele corespondente setului de indecși.

În continuare voi descrie metodele de selecție utilizate în algoritmii euristici. Acestea având un rol important în evoluția indivizilor.

```
do {

probability_selection(population, *size);
population_selected(population, temp_pool, *size, &temp_pool_size);

if (temp_pool_size)

{
    for (uint16 i = 0; i < temp_pool_size; i++) {
        if (temp_size < *size) {

            individual_init(&temp_population[temp_size]);
            copy_individual_into_population(temp_population, temp_pool,
                    temp_size, i);
            ++temp_size;
        } else {
            break;
        }
    }

}

    shuffle_array(population, *size);


} while (temp_size < *size);
```

Figura 11 - Selecția *Roultte Wheel*

În imaginea din Figura 11 se poate vedea implementare metodei de selecție *Roullete Wheel*. Pentru început am calculat pentru fiecare individ probabilitatea de a fi selectat. În continuare am adăugat într-o populație temporară indivizi selectați pentru generația următoare.
După fiecare iterație am schimbat poziția indivizilor din populația inițială pentru a obține alte poziționări pe ruletă.
Am repetat acești pași până am obținut o populație de același număr de indivizi.

```
for (uint16 i = 0; i < size; i++) {//size; i++) {
    copy_population(temp_population, population, size);

    if(shuffle_array(temp_population, size) == data_no_error)
    {
        //get best k individuals from tournament
        uint16 index1 = get_index_of_BEST(temp_population, k);

        copy_individual_into_population(pool_population, temp_population, i,
                index1);
    }

}
```

Figura 12 - Selecția *Tournament*

În imaginea din Figura 12 se poate vedea implementare metodei de selecție *Tournament* unde pentru fiecare turneu amestec populația de indivizi și din primii k-indivizi îl selectez pe cel mai bun pentru a-l adăuga noii populații de indivizi.

```
if (FALSE != individual_match_schema(&population[i], &schemas[j])) {

    r = (double) (rand() / (double) (RAND_MAX));

    if (r < OPERATORS_PROBABILITY) {
        population[i].selected = TRUE;
    } else {
        population[i].selected = FALSE;
    }
    break;
```

Figura 13 - Selecție *Building blocks*

În imaginea din Figura 13 am prezentat metoda de stabilire a unui individ pentru a trece în generația următoare în funcție de setul de coloane, unde am setat și selecția acestui în ceea ce privește aplicare operatorilor genetici.

7 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 calitatea algoritmilor, am realizat un program care să reprezinte grafic statisticile obținute.

7.1 Generare seturi de date

Dat fiind faptul că algoritmii necesită un set de teste, seturile de date generate de aplicație sunt utilizate ca și *benchmark*-uri. Pentru a genera astfel de seturi de date am construit o regresie liniară. Valorile criteriilor obținute împreună 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:

$$Y = A*\beta + \varepsilon$$

7.2 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 *benchmark*.

Deoarece de multe ori algoritmii euristici oferă soluții foarte apropiate de cel al *benchmark*-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 foarte mult 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:

$$Match\ columns = |\{coloane_{best}\} \cap \{coloane_{euristic}\}| \ / \ n_{best} \text{ unde,}$$

n_{best} reprezintă numărul de coloane al celui mai bun sub-set.

În condițiile în care dintr-un număr de 20 de coloane, din care 5 coloane sunt asociate celui mai bun model, dacă algoritmul identifică aceste 5 coloane dar și restul de 15 coloane metrica *match columns* va atașa acestui rezultat valoare 1 ce reprezintă raportul de 5/5, considerând-ul un individ adaptat. Pentru a evita acest lucru, am adăugat o metrică *missmatch columns* ce reprezintă raportul dintre numărul de coloane identificate de algoritmul euristic care nu fac parte din setul de coloane al celui mai bun subset.

$$Missmatch\ columns = n_{euristic} - |\{coloane_{best}\} \cap \{coloane_{euristic}\}| / \ n_{euristic} \text{ unde,}$$

$n_{euristic}$ reprezintă numărul total de coloane al setului identificat de algoritmul euristic

Un alt element de comparație îl reprezintă valoare funcției AIC . Metrica *report AIC* este egală cu diferența celor doua valori AIC_{best} și $AIC_{euristic}$ în raport cu AIC_{best}. Această metrică exprimă numeric distanța dintre eroarea sub-modelului calculată de algoritmul euristic și eroare *benchmark*-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.

$$Y = A*\beta + \varepsilon \text{ unde,}$$

Prin culoarea verde am exprimat faptul că elementele ecuației sunt aceleași pentru ambele soluții, iar cele reprezentate prin culoarea roșu am exprimat faptul că acestea pot avea 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 *benchmark*.

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

7.3 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 *benchmark*. 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.

8 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.

8.1 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 axa_y 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 axa_x 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 axa_x 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.

```
Column    Contains
   1      Month: 1 = January, ..., 12 = December.
   2      Day of month.
   3      Day of week: 1 = Monday, ..., 7 = Sunday.
   4      Daily maximum one-hour-average ozone reading (parts per million)
          at Upland, CA.
   5      500 millibar pressure height (m) measured at Vandenberg AFB.
   6      Wind speed (mph) at Los Angeles International Airport (LAX).
   7      Humidity (%) at LAX.
   8      Temperature (degrees F) measured at Sandburg, CA.
   9      Temperature (degrees F) measured at El Monte, CA.
  10      Inversion base height (feet) at LAX.
  11      Pressure gradient (mm Hg) from LAX to Daggett, CA.
  12      Inversion base temperature (degrees F) at LAX.
  13      Visibility (miles) measured at LAX.
```

Figura 14 – Seminficația coloanelor din setul de date reale Ozone.txt

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.

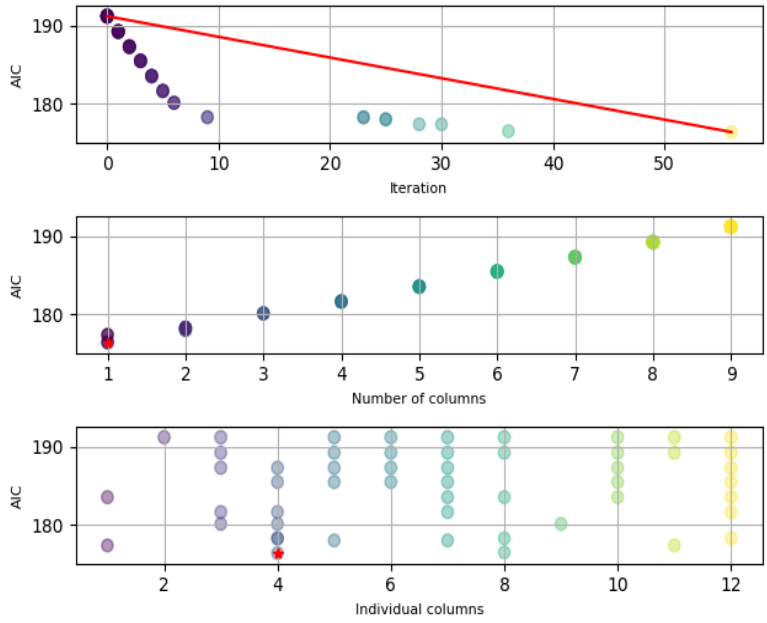


Figura 15 – Outputul Algoritmului genetic pe setul real de date rulat cu operatorii genetici *flip mutation* și *crossover 1point adaptat.*

În imaginea din Figura 15, 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 însă datorită mutațiilor reapare la următorul *update* al celui mai bun sub-set. De asemenea o altă observație importantă relativă la imaginea din Figura 15 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 *benchmark* outputul generat de metoda exhaustivă, descrisă în subcapitolul 3.2, pentru a avea convingerea că algoritmul generează cel mai bun sub-set. Desigur că acest lucru este limitat la dimensiune setului de date.

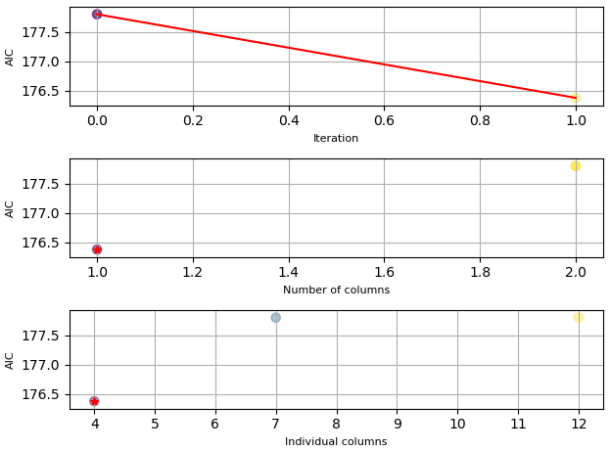


Figura 16 - Outputul Algoritmului genetic pe setul real de date rulat cu operatorii genetici *flip mutation* și RRC *crossover*

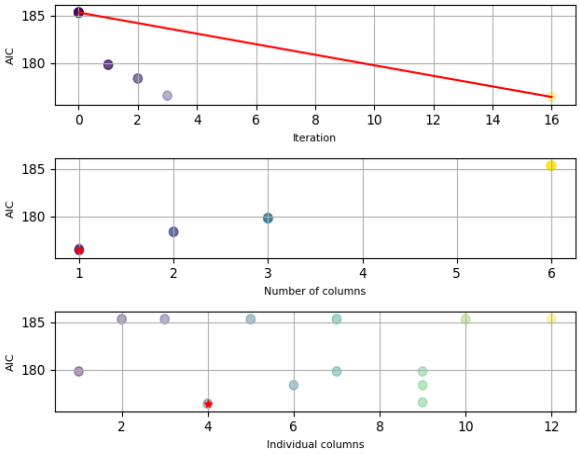


Figura 17 - - Outputul Algoritmului genetic pe setul real de date rulat cu operatorii genetici *reversing mutation* și *crossover 1-point simple*

În imaginea din Figura 16 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 imaginea din Figura 17 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 demonstrează 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.

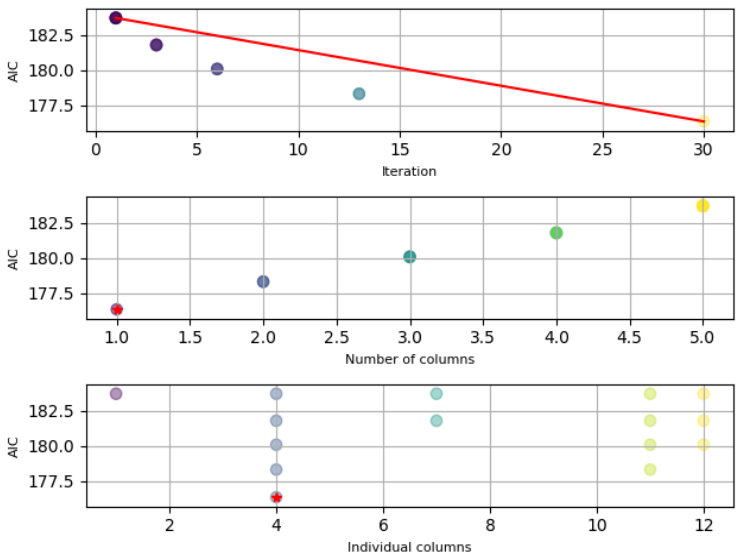


Figura 18 - Outputul Algoritmului *Hill Climbing* pe setul real de date rulat cu operatorul genetic *flip mutation*

În imaginea din Figura 18 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.

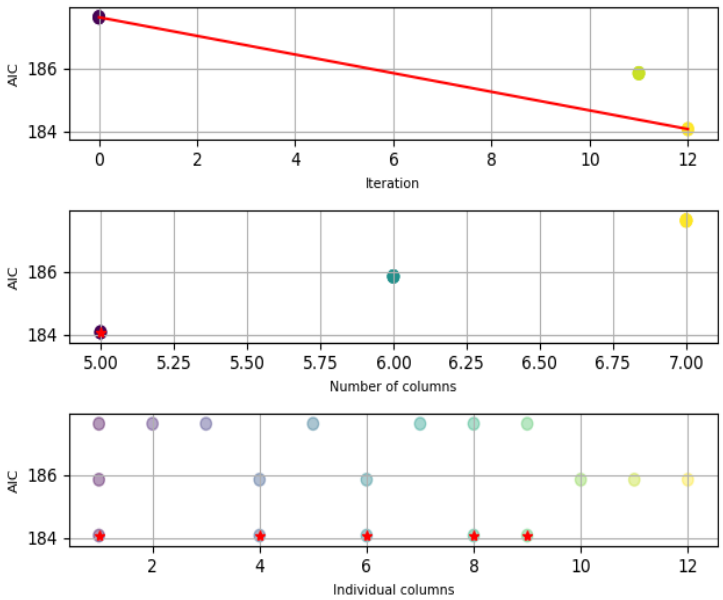


Figura 19 - Outputul Algoritmului *Hill Climbing* pe setul real de date rulat cu operatorul genetic *reversing mutation*

În imaginea din Figura 19 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.

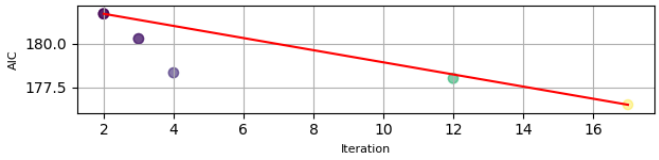


Figura 20 - Outputul Algoritmului *Simulated Anealing* pe setul real de date rulat cu operatorul genetic *flip mutation și cooling_rate 0.33*

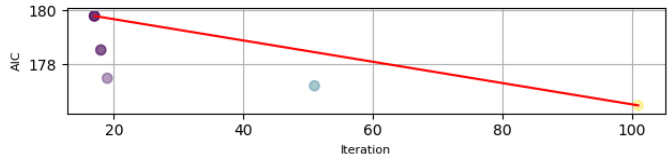


Figura 21 - Outputul Algoritmului *Simulated Anealing* pe setul real de date rulat cu operatorul genetic *flip mutation și cooling_rate 0.86*

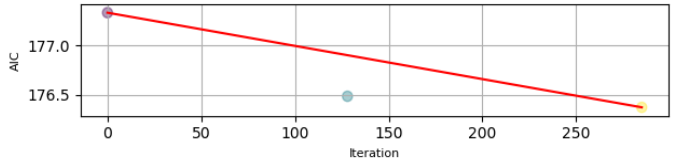


Figura 22 - Outputul Algoritmului *Simulated Anealing* pe setul real de date rulat cu operatorul genetic *flip mutation și cooling_rate 0.95*

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 20 parametrul *coolin rate* are valoare 0.33, în Figura 21 valoare este de 0.86 iar în Figura 22 valoarea este de 0.95.

Se poate observa ca numărul de iterații în care algoritmul actualizează cel mai bun candidat 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 valoarea *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%.

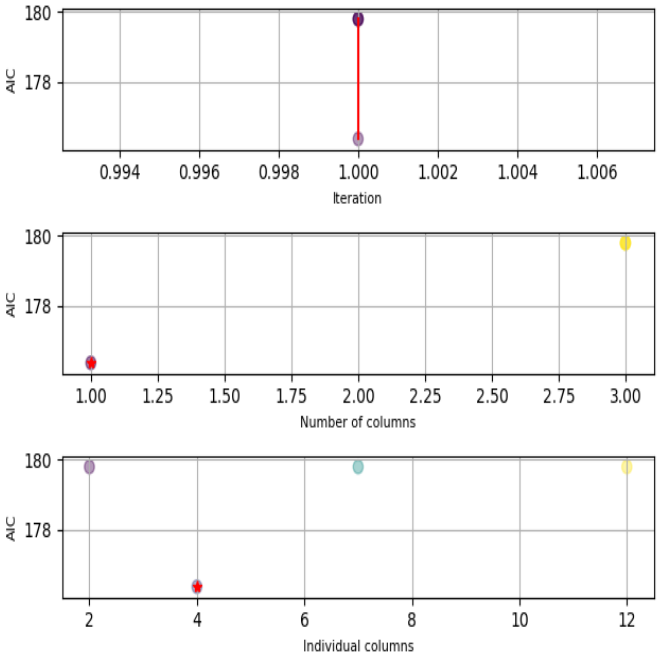


Figura 23 - Outputul Algoritmului *Buildinf Blocks* pe setul real de date rulat cu operatorii genetici *flip mutation 1 point crossover* adaptat

În imaginea din Figura 23 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.

8.2 Comportamentul algoritmilor pe seturi de date generate

8.2.1 *Tuning algorithms*

Pentru a face o comparație cât mai corectă pe algoritmii euristici discutați în această lucrare, am încercat să aduc pe fiecare algoritm în parte, la cel mai bun stadiu în care acesta poate exista. Acest lucru l-am efectuat prin modificarea tuturor parametrilor ce influențează comportamentul algoritmului. În cele ce urmează, voi atașa pe rând, pentru fiecare algoritm în parte, câteva dintre rezultatele cele mai semnificative pe care le-am obținut.

În ceea ce privește algoritmul genetic, înainte de a discuta despre toți parametrii setați, voi discuta despre modurile în care algoritmul genetic poate fi executat. Spre deosebire de *hill climbing* unde pentru a trece de la o soluție la alta se apelează doar la o operatorul de mutație, algoritmul genetic combină 3 nivele de modificare a soluțiilor:

{*flip, interchanging, reversing* }*{*1-point, 1-point* adaptat, *uniform, RRC}**

{ *roullete wheel* , *tournament*}

Nr. config.	Operator mutație	Operator *crossover*	Selecție
0	*flip*	*1-point*	*roulette wheel*
1	*flip*	*1-point* adaptat	*roulette wheel*
2	*flip*	*uniform*	*roulette wheel*
3	*flip*	RRC	*roulette wheel*
4	*interchanging*	*1-point*	*roulette wheel*
5	*interchanging*	*1-point* adaptat	*roulette wheel*
6	*interchanging*	*uniform*	*roulette wheel*
7	*interchanging*	RRC	*roulette wheel*
8	*reversing*	*1-point*	*roulette wheel*
9	*reversing*	*1-point* adaptat	*roulette wheel*
10	*reversing*	*uniform*	*roulette wheel*
11	*reversing*	RRC	*roulette wheel*
12	*flip*	*1-point*	*tournament*
13	*flip*	*1-point* adaptat	*tournament*
14	*flip*	*uniform*	*tournament*
15	*flip*	RRC	*tournament*
16	*interchanging*	*1-point*	*tournament*
17	*interchanging*	*1-point* adaptat	*tournament*
18	*interchanging*	*uniform*	*tournament*
19	*interchanging*	RRC	*tournament*
20	*reversing*	*1-point*	*tournament*
21	*reversing*	*1-point* adaptat	*tournament*
22	*reversing*	*uniform*	*tournament*
23	*reversing*	RRC	*tournament*

Tabel 1

Fiecare iterație atașată pe axa Ox a graficelor reprezintă un set unic de forma {*mutation, crossover, selection*} din Tabel 1. Așadar, primele 12 iterații corespund metodei de selecție *roulette wheel* cu toate combinațiile de operatori, iar ultimele 12 iterații corespund metodei de selecție *tournament*.

Pentru algoritmul genetic am început cu execuția acestuia pe un set de date cu un număr de 100 de observații peste 20 de factori a căror valori au foste generate cu o deviație standard de 0.01 pentru care cel mai bun subset constituie 50% din numărul total de factori.

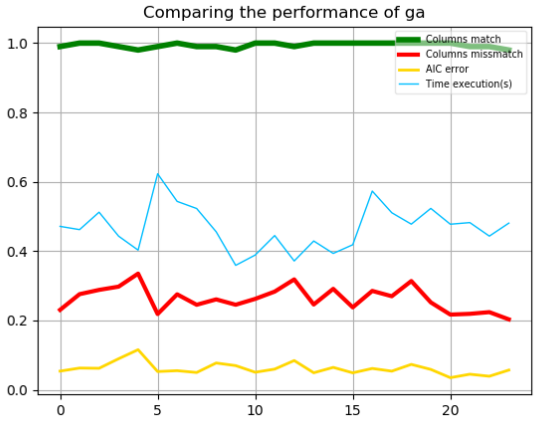


Figura 24 - Tuning GA , populație 10%

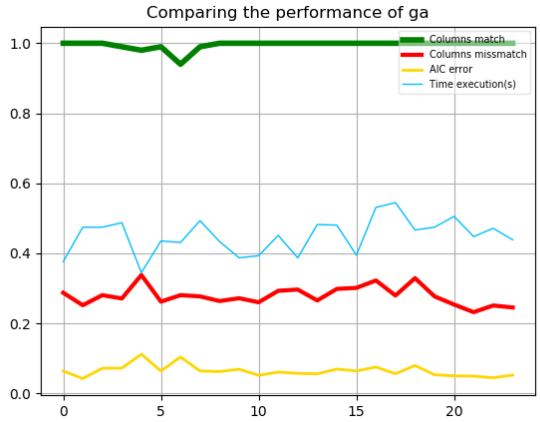


Figura 25 - Tuning GA , populație 30%

În imaginile din Figura 24 și Figura 25 am configurat populația indivizilor la un procent de 30% din numărul de observații, cu valoarea de *converge* de 100. În Figura 24, am setat pentru *tournament selection* un procent de 10% din populație pentru fiecare turneu, iar în Figura 25, un procent de 30%.

Din cele doua imagini se poate observa că metoda de selecție *tournament* generează soluții cu erori mai mici. Cele mai bune combinații de operatori sunt generate de configurațiile de la iterațiile 0, 1, 10 respectiv 12, 13, 20, descrise în tabelul Tabel 1 . Se poate observa că valoarea erorii minime este atinsă atunci când *match*-ul de coloane este maxim, iar *missamtch*-ul este minim. De asemenea, se poate observa că odată ce am mărit numărul de indivizi care participă la turneu, algoritmul genetic identifică în fiecare configurație a selecției turneu toate coloanele din setul celui mai bun candidat.

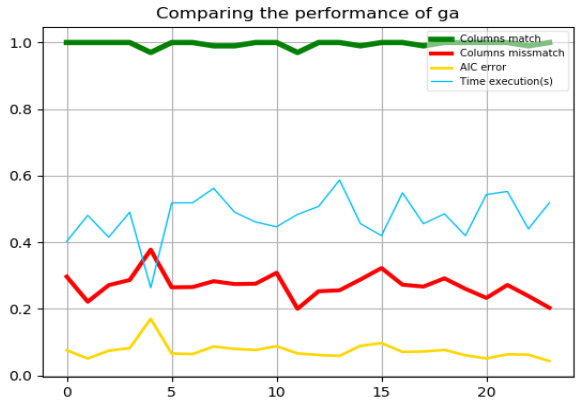


Figura 26 - Tuning GA , populație 50%

În imaginea din Figura 26 am setat pentru numărul de candidați participanți la un turneu un procent de 50% din populație. Deși cel mai bun individ din turneu va avea o valoare de cost mai calitativă decât în cazurile anterioare, deoarece va fi mai adaptat decât cel puțin 50% din populație, acest lucru va restrânge diversitatea populație. În cazul cel mai favorabil, algoritmul va alege pentru generația următoare, maxim 50% de candidați diferiți din generația curentă.

Se poate observa, ca și în cazul în care numărul de candidați din turneu este de 10%, algoritmul nu găsește în toate cazurile toate coloanele celui mai bun subset. Cu alte cuvinte turneul trebuie sa fie configurat la o valoare care să confere un procent de diversitate nici prea mare, dar nici să creeze o populație de indivizi foarte buni care se repetă excesiv.

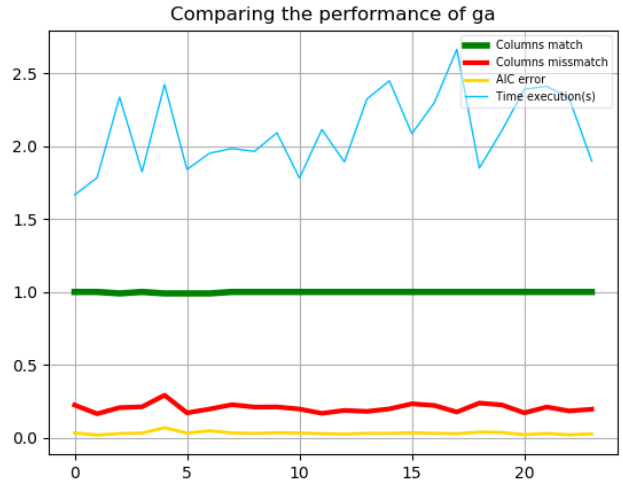


Figura 27 -Tuning GA , populație 100%

Pentru imaginea din Figura 27 am configurat numărul de indivizi din populație la 100%. Pe lângă acest lucru am setat un număr fix de 10 coloane pentru fiecare individ în momentul generării populației, valoarea de *converge* am setat-o ca și în cazurile discutate mai sus la valoare 100 și procentul de indivizi pentru turneu la valoare de 30%.
În urma acestor setări se poate observa că timpul de execuție este chiar și de 4 ori mai mare când populația este mai încărcată.

O altă observație este faptul că la configurațiile 5 și 17, unde operatorii nu modifică numărul de coloane, valoarea erorii este printre cele mai mici. Nu este minimă deoarece metricile aplicate penalizează mai mult o soluție a algoritmului genetic dacă acesteia îi lipsesc coloane din cel mai bun subset, decât dacă soluția conține toate coloanele celui mai bun subset dar conține și alte coloane în plus. De exemplu dacă avem :

$$I_{best} = \{1, 2, 3, 4, 5\},$$

$$I_{eurisitc1} = \{1, 2, 3, 4\} \text{ și}$$

$$I_{eurisitc2} = \{1, 2, 3, 4, 5, 6\}$$

eroare mai mică va fi a individului $I_{eurisitc2}$.

În continuare voi discuta despre configurarea algoritmului *hill climbing*. Am păstrat aceleași setări pentru seturile de date generate, cu excepția numărului de observații pe care l-am setat pe valoarea 150, deoarece timpul de execuție permite acest lucru.

Dat fiind faptul că acest algoritm generează un singur individ pe o generație, nu se poate aplica decât operatorul de tip mutație genetică. Așadar am comparat outputurile algoritmului selectând câte una din mutații pentru fiecare rulare, pentru prima iterație mutația *flip,* a doua iterație cu mutația *interchanging* și pentru ultima iterație, mutația *reversing.*

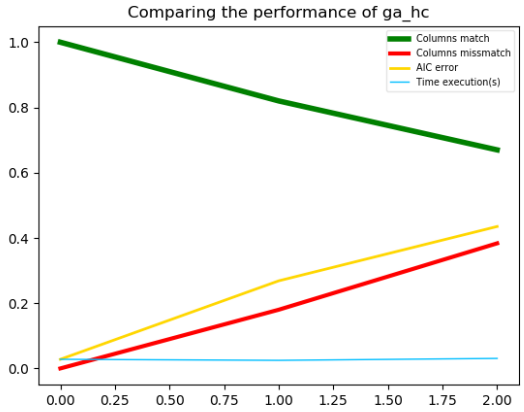


Figura 28 - *Tuning* HC , 50 de iterații

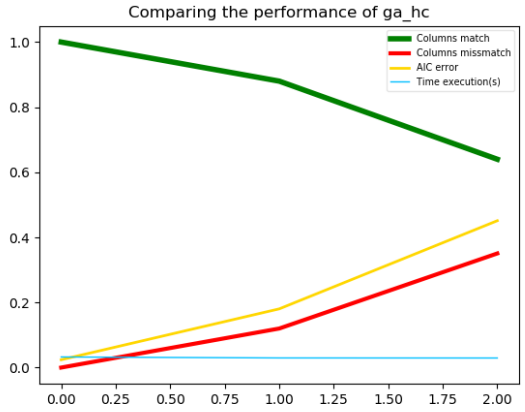


Figura 29 - *Tuning* HC , 100 de iterații

În imaginea din Figura 28 am rulat algoritmul *hill climbing* pe 50 de iterații iar în imaginea din Figura 29, am crescut numărul de iterații la 100. Pe ambele cazuri se observă că mutația *flip* generează cel mai bun subset, întrucât *columns match* are valoare 1 iar *columns missmatch* este 0. Numărul crescut de iterații aduce îmbunătățirii si pentru mutația *interchanging*, cu toate acestea timpul de execuție rămâne la aceleași valori.

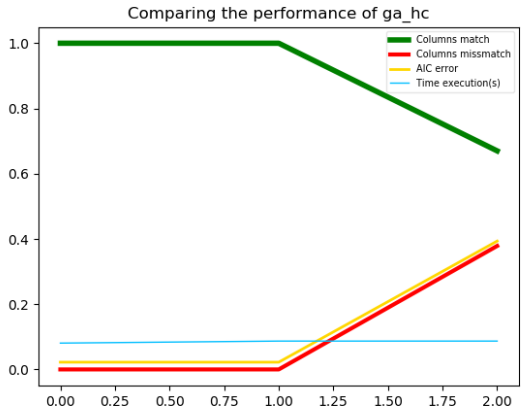


Figura 30 - *Tuning* HC , 1000 de iterații

În imaginea din Figura 30 am crescut și mai mult numărul de iterații la valoarea de 1000, ceea ce înseamnă de 20 de ori mai mult decât în primul exemplu pentru a observa dacă și mutația *reversing* aduce rezultate mai bune, însă se poate observa că la ultima iterații comportamentul este aproximativ la fel, doar valoarea erorii descrește cu aproximativ 0,1.
Deși am mărit numărul de iterații, timpul de execuție crește foarte puțin ceea ce permite ca numărul de iterații să fie setat cu o valoare foarte mare.

Prin urmare pentru ca algoritmul *hill climbing* să aducă soluții bune, contează extrem de mult operatorul de mutație, deoarece putem observa că sunt diferențe foarte mari între outputuri,

spre deosebire de algoritmul genetic unde diferențele sunt mai puțin sesizabile. Algoritmul are un comportament haotic, în acest sens, de punctat este faptul că algoritmul *hill climbing* rulat cu mutația *reversing* generează o soluție cu o eroare mai mare decât oricare din soluțiile generate de algoritmul genetic, pe când atunci când *hill climbing* este rulat cu mutația *flip*, acesta generează setul de coloane al celui mai bun candidat, lucru care nu se întâmplă în exemplele discutate la algoritmul genetic.

În cele ce urmează voi discuta despre modul de configurare al algoritmului *Simulated Annealing* pentru care am păstrat setările aplicate pentru algoritmul *hill climbing* pentru generarea seturilor de date.

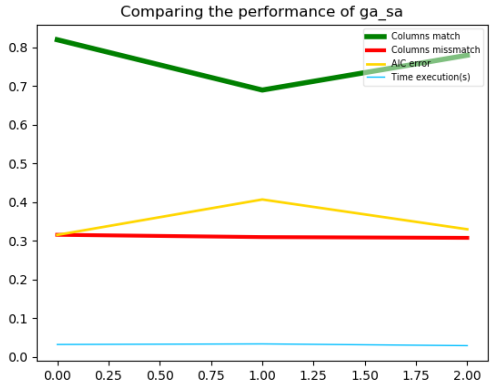


Figura 31 - *Tuning* SA , *cooling rate* 0.99

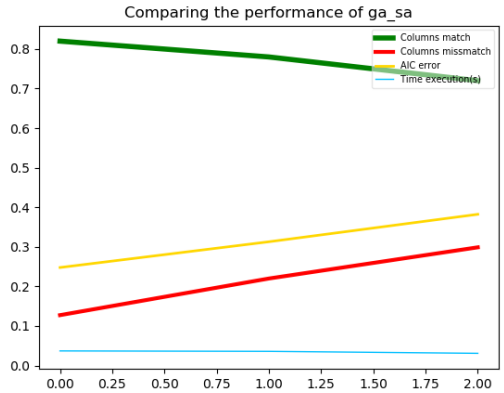


Figura 32 - *Tuning* SA , *cooling rate* 0.85

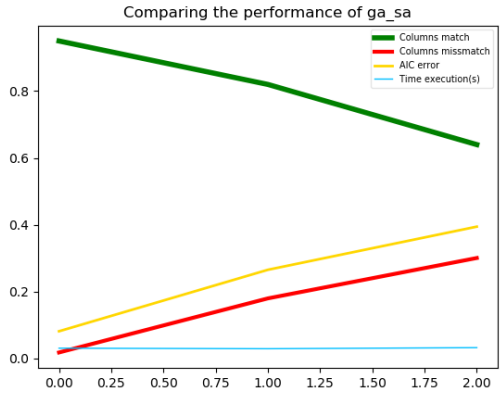


Figura 33 - *Tuning* SA , *cooling rate* 0.75

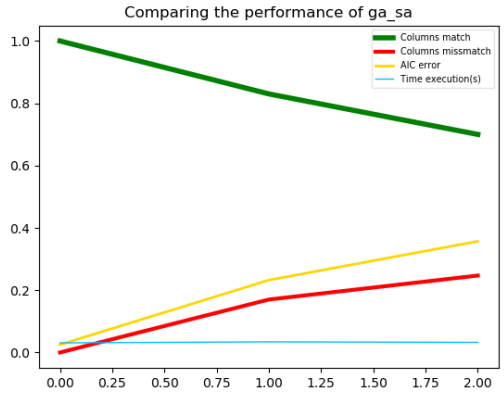


Figura 34 - *Tuning* SA , *cooling rate* 0.50

În imaginile din figurile de mai sus am setat numărul de iterații cu valoarea 100 și temperatura de 1000000, însă pentru fiecare imagine am configurat *colling rate*-ul cu valori diferite descrescătoare din setul de valori (0.99, 0.85, 0.75, 0.50) în ordinea figurilor. Se observă că gradul de răcire al temperaturii afectează foarte mult comportamentul algoritmului, în imaginea din Figura 31, Figura 32 chiar și Figura 33, deși temperatura este mare și implicit și gradul de acceptare al indivizilor mai puțin adaptați ce oferă o diversitate mai mare a indivizilor evaluați, se poate vedea că algoritmul nu reușește să găsească cel mai bun subset. Outputul se îmbunătățește în imaginea din Figura 34, unde *cooling rate*-ul a fost configurat la 0.50.

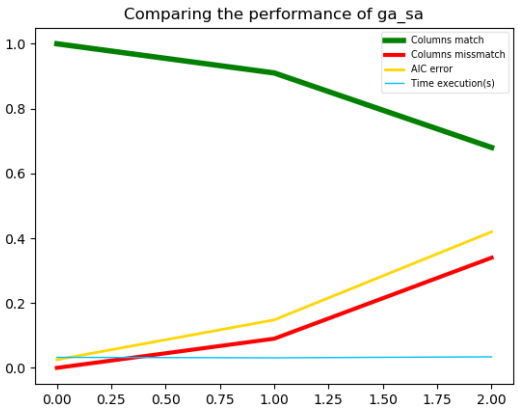


Figura 35 - *Tuning* SA , *cooling rate* 0.25, iterații 100

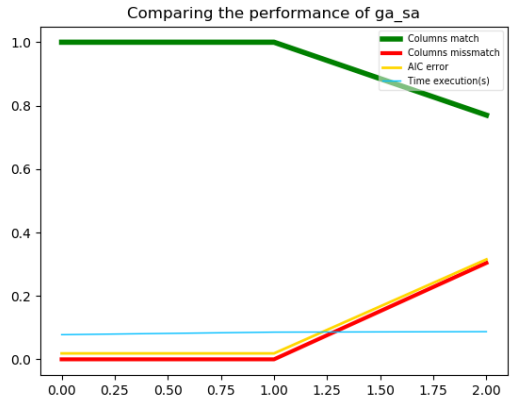


Figura 36 - *Tuning* SA , *cooling rate* 0.25, iterații 1000

În imaginea din Figura 35 și Figura 36, am scăzut și mai mult parametrul de răcire, la valoare de 0.25. Pentru Figura 36 am schimbat și numărul de iterații de la 100 la 1000. Comparativ cu algoritmul *hill climbing*, algoritmul *simulated annealing* se comportă puțin mai bine pe mutația *reversing*, acesta din urmă generează o eroare de 0.3, pe când *hill climbing* cu aceeași mutație generează o eroare de 0.4, cum se poate vedea în imaginea din Figura 30.

Building Blocks este ultimul algoritm pentru care am analizat parametrii de configurație în urmă cărora algoritmul să obțină cele mai bune outputuri. Pentru seturile generate de date am păstrat aceleași setări menționate la algoritmul genetic. De asemenea configurația operatorilor genetici este descrisă în tabelul Tabel 1 până la configurația numărul 12 fără metoda de selecție. Numărul de *converge* este de 100 pentru toate exemplele pe care le voi discuta.

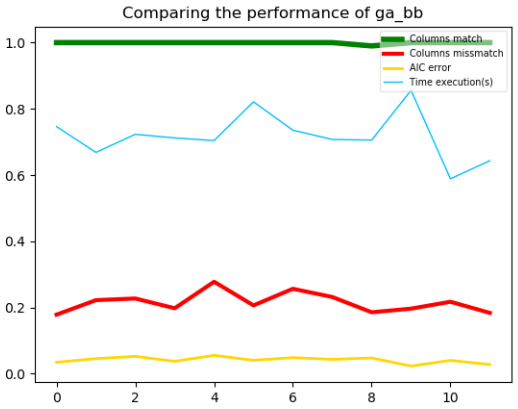


Figura 37 - *Tuning* BB , *size schema* 20%, pc, pm = 50%

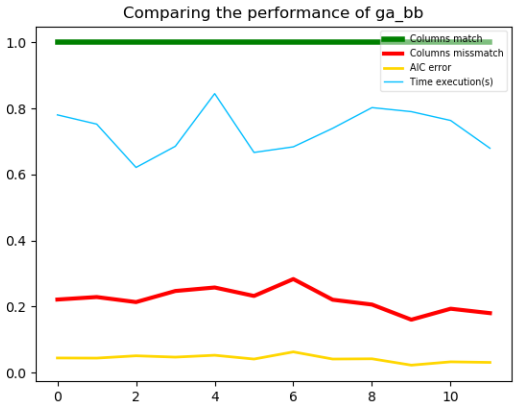


Figura 38 - *Tuning* BB , *size schema* 30%, pc, pm = 50%

În imaginea din Figura 37 am setat un procent de 30% din numărul total de coloane din setul de date ce reprezintă numărul de scheme. Cu cât acest procent este mai mare, cu atât numărul de coloane pentru fiecare schemă este mai mic. În imaginea din Figura 38, am setat acest procent cu 50%. În ambele cazuri, am setat un procent de 20% de modificare al indivizilor ce conțin una din scheme. Se poate observa că atunci când schemele au un număr de coloane mai

mic, nu există fluctuații în valorile exprimate prin *columns match* , de asemenea eroarea minimă este vizibilă tot în imaginea din Figura 38, în configurația cu numărul 9.

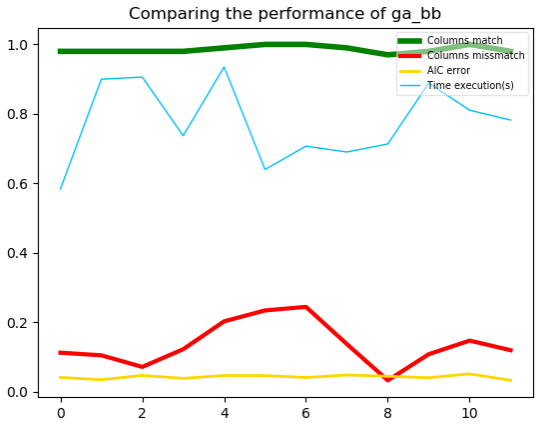


Figura 39 - *Tuning* BB , *size schema* 90%, pc, pm = 10%

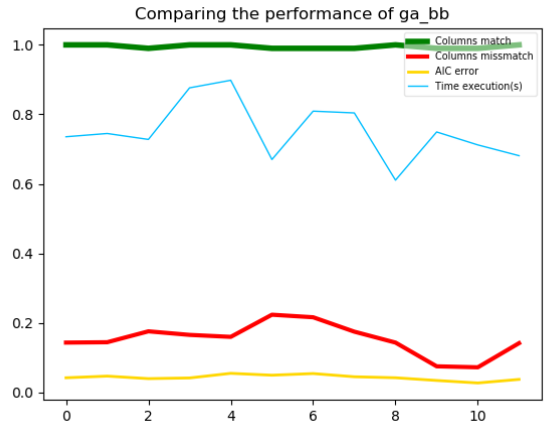


Figura 40 - *Tuning* BB , *size schema* 90% , pc, pm = 20%

În imaginea din Figura 39 am setat un procent de 10% pentru aplicarea operatorilor genetici, iar în imaginea din Figura 40 am setat un procent de 20%. Pentru ambele imagini am setat numărul schemelor la un procent de 90% din numărul total de coloane din setul de date. Un procent atât de mare pe un număr de coloane de 20, va rezulta faptul că fiecare schemă va fi compusă dintr-o singură coloană. Deși în imaginea din Figura 39 se observă cum *columns missmatch* tinde către 0 , valoare pentru *columns match* nu este maximă, pentru puține configurații *columns match* este 1. Un procent de 10% pentru operatorii genetici reduce prea mult modificarea indivizilor ce conțin una din scheme. Aceștia sunt bine adaptați însă necesită un minim de modificare pentru a deveni și mai buni. Acest lucru de poate observa în imaginea din Figura 40 pentru configurațiile 9 și 10.

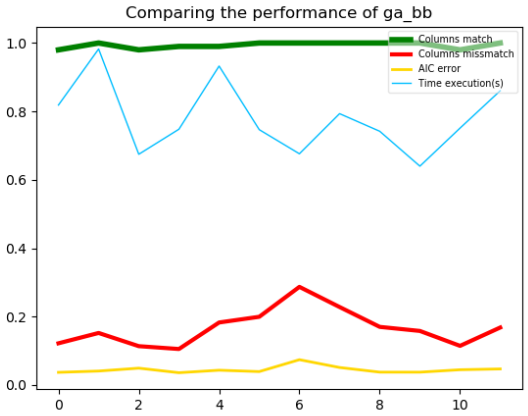


Figura 41 - *Tuning* BB , *size schema* 90%, pc, pm = 1%

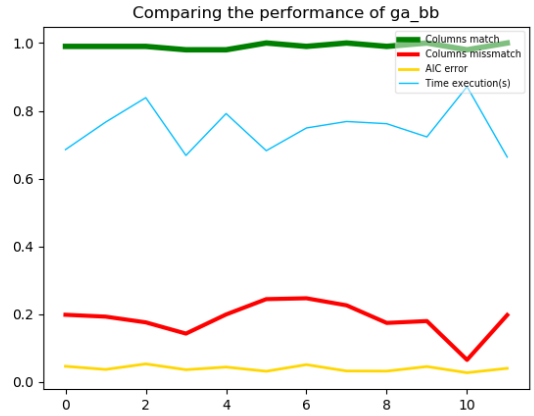


Figura 42 - *Tuning* BB , *size schema* 90%, pc, pm = 50%

În imaginea din Figura 41 cât și din Figura 42 am păstrat procentul de 90% pentru numărul de scheme, dar am modificat probabilitatea de aplicare a operatorilor genetici. Așadar, pentru imaginea din Figura 41 am configurat un procent de 1%, iar în imaginea din Figura 42 acest

procent l-am configurat cu valoarea de 50, pentru a pune în evidență efectul aplicării operatorilor genetici prea mult sau deloc. Dacă acest procent este mic, atunci indivizii care conțin una din scheme nu vor evalua pe parcursul generațiilor, iar daca acest procent este prea mare, atunci schemele se vor pierde pe parcursul generațiilor, în acest caz algoritmul va produce rezultate *random*.

8.2.2 Comparație algoritmi

În acest subcapitol voi puncta câteva observații în ceea ce privește comparația algoritmilor euristici implementați în aceasta lucrare. Pentru început voi menționa configurațiile ales și planurile de comparație.

Datorită implementării aplicației pentru *tuning algorithms*, am reușit pentru fiecare algoritm euristic în parte să obțin o combinație de configurații care să ofere printre cele mai bune rezultate pe plan individual. Așadar configurațiile sunt următoarele:

- Algoritmul genetic:
  - mutația *reversing,* operatorul 1-*point crossover, tournament selection*
  - *converge* 100
  - procentul de indivizi dintr-o populație de 100%
  - procentul indivizilor aleși pentru turneu de 30%
- *Hill climbing*:
  - mutația *flip*
  - 100 de iterații
- *Simulated annealing*:
  - mutația *flip*
  - 100 de iterații
  - *cooling rate* 0.25
  - temperatura de 1000000
- *Building blocks*:
  - mutația *reversing,* operatorul de *crossover* RRC
  - *converge* 100
  - procentul de indivizi dintr-o populație de 100%
  - procentul numărului de scheme de 90%
  - probabilitatea de 20% de a aplica operatori genetici peste indivizii ce conțin una din scheme

Folosind aceste configurații, vom vedea o comparație peste acești algoritmi, modificând doar seturile de date de input. Am utilizat ca și *template* configurația cu un număr de 100 de observații, cu numărul total de coloane de 20, numărul de coloane al celui mai bun subset de 10, deviația standard a datelor generate de 0.01 și un număr de 10 fișiere generate. Peste acest *template* am modificat pe rând valorile fiecărui membru pentru a obține seturi diferite de date pentru a compara aproximările algoritmilor euristici.

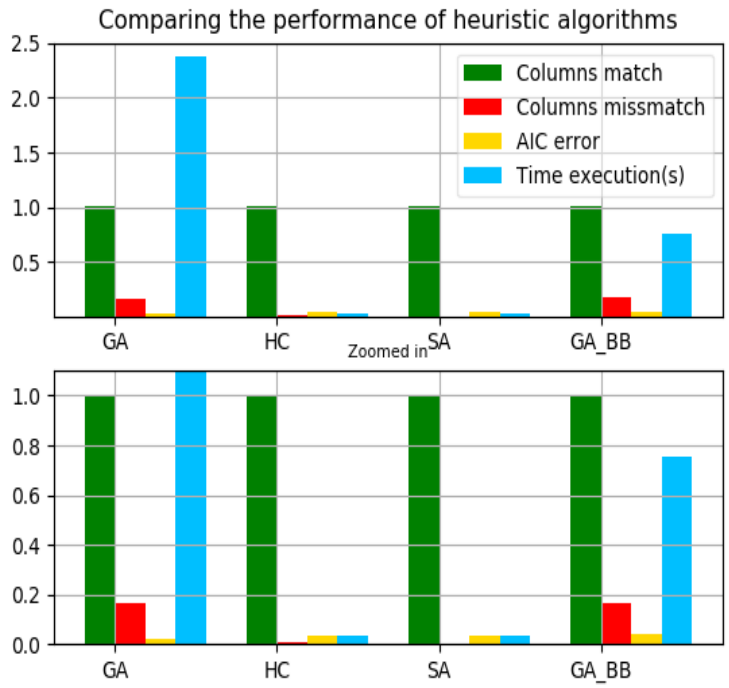


Figura 43 – *Comparing Algorithms*, config 1.1

Pentru început am rulat algoritmii cu configurația *template*. În imaginea din Figura 43 am modificat numărul de indivizi dintr-o populație la 80% din numărul de observații.
În această configurație toți algoritmii găsesc cel mai bun subset, însă *simulated annealing* nu generează nici-o soluție cu erori, pe toate cele 10 seturi de date, algoritmul generează cel mai bun subset, urmat de *hill climbing*.
În ceea ce privește algoritmul genetic, acesta oferă soluții un pic mai bine adaptate față de *building blocks*, însă referitor la timpul de execuție, acesta consumă cel mai mult timp.

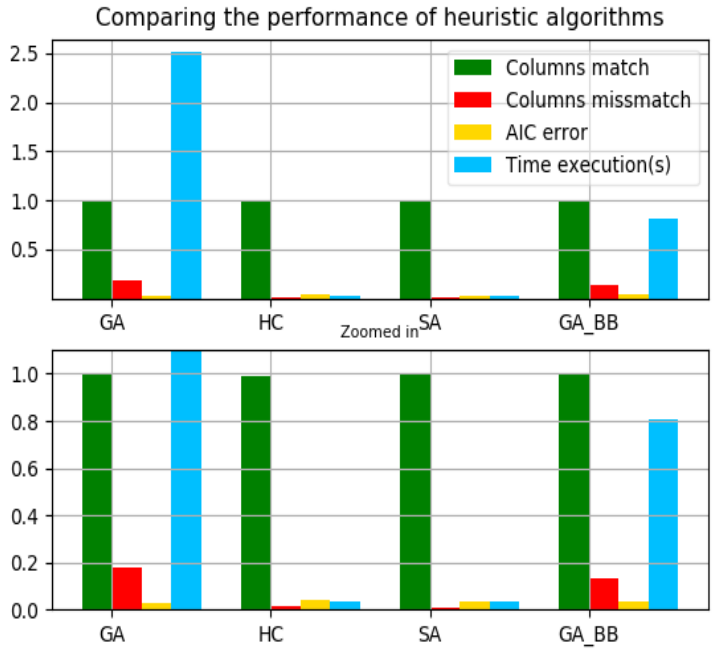
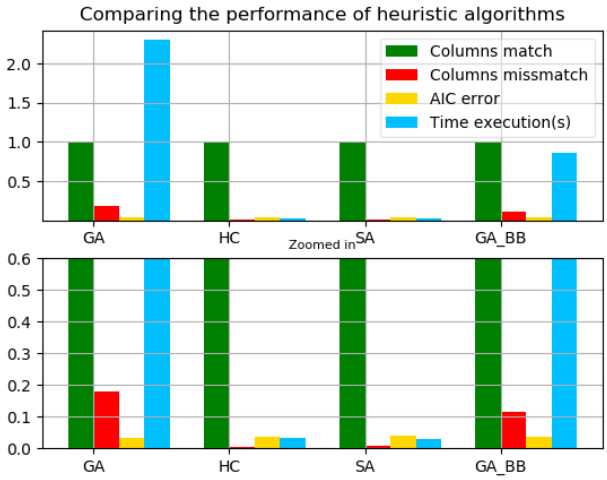


Figura 44 - *Comparing Algorithms*, config 1.2

În imaginea din Figura 44 am setat un număr de 50 de fișiere și numărul de indivizi dintr-o populație la 100%.
După cum se vede, algoritmul *hill climbing* este singurul pentru care *columns match* nu are valoare maximă.
O altă observație este faptul că algoritmul *building blocks* oferă soluții mai puțin eronate față de algoritmul genetic.
Cu toate acestea, pentru toți cei 4 algoritmi, valoarea de eroare este asemănătoare.



În final, pentru imaginea din Figura 45, am setat un număr de 100 de fișiere.
Raportul dintre algoritmul genetic și *building blocks* se păstrează, însă în ceea ce privește algoritmul *hill climbing*, acesta dovedește un comportament imprevizibil. Pe această configurație se poate observa că oferă cele mai bune soluții.

Figura 45 *Comparing Algorithms*, config 1.3

În continuare voi prezenta outputurile algoritmilor în condițiile în care numărul de observații crește, restul configurațiilor sunt în conform cu *template*-ul, mai puțin *cooling rate* pentru care am setat valoarea 30.

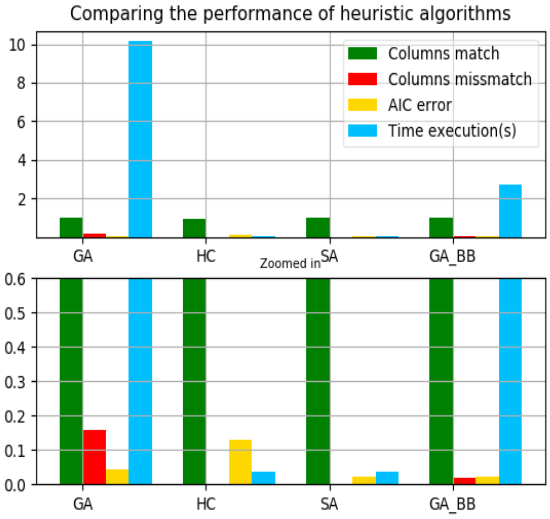


Figura 46 - *Comparing Algorithms*, config 2.1

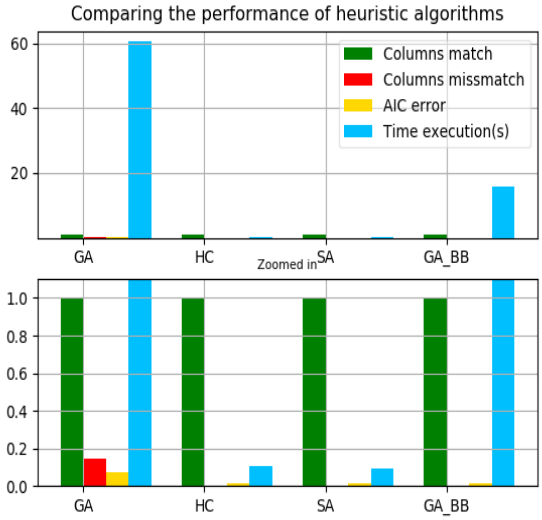
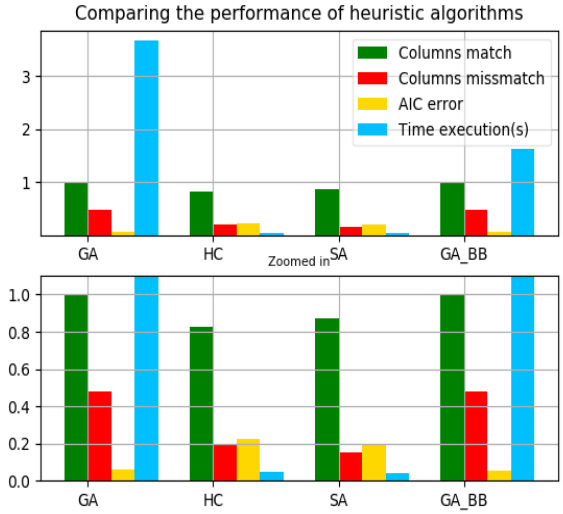


Figura 47 *Comparing Algorithms*, config 2.2

În imaginea din figura Figura 46 am setat un număr de 200 de observații, iar în imaginea din Figura 47, am crescut numărul la 500. Pe aceste configurații se poate observa o evoluție interesantă a algoritmului *building blocks* care reușește să ofere pentru fiecare fișier soluția cea mai bună. Așadar o observație importantă o constituie faptul că algoritmul *building blocks*

funcționează foarte bine în condițiile în care numărul de observații din setul de date de intrare este cât mai mare. O altă observație este asupra outputului algoritmului *hill climbing,* care în imaginea din Figura 46 oferă cele mai eronate soluții.

În continuare, pe același *template*, am modificat numărul total de coloane și numărul de coloane al celui mai bun candidat, între care am păstrat un raport de 50%.



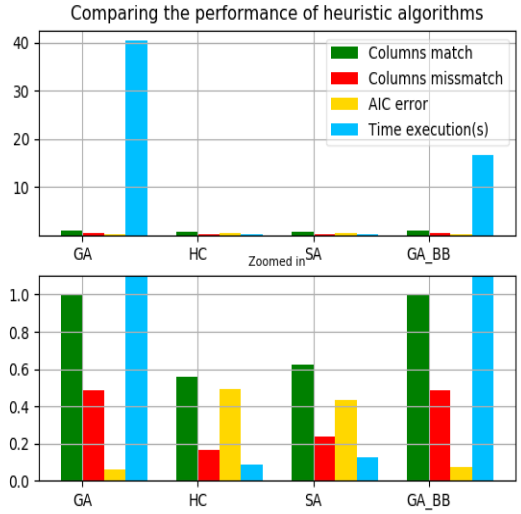


Figura 48 - *Comparing Algorithms*, config 3.1

Figura 49 - *Comparing Algorithms*, config 3.2

În imaginea din Figura 48 precum și în imaginea din Figura 49 se observă că algoritmii *simulated annealing* și *building blocks* nu reușesc să cuprindă în soluțiile generate toate coloanele celui mai bun subset, și din cauza acestui fapt, eroarea a crescut până la 0.5. În imaginea din Figura 49 se poate observa cât de mult a crescut timpul de execuție până la 40s, odată ce a crescut și cu numărul de indivizi din populație.

În cele din urmă am rulat algoritmii pe un set de configurații în care am modificat valoarea parametrului *standard deviation*. Această metrică este utilizată pentru a afunda soluția celui mai bun subset într-o populație cu valori foarte diferite de la un individ la altul. O observație importantă este faptul că acest parametru nu ar trebui sa ia valori foarte mari, deoarece suntem în cadrul unei probleme de regresie liniară, unde datele trebuie sa fie corelate.

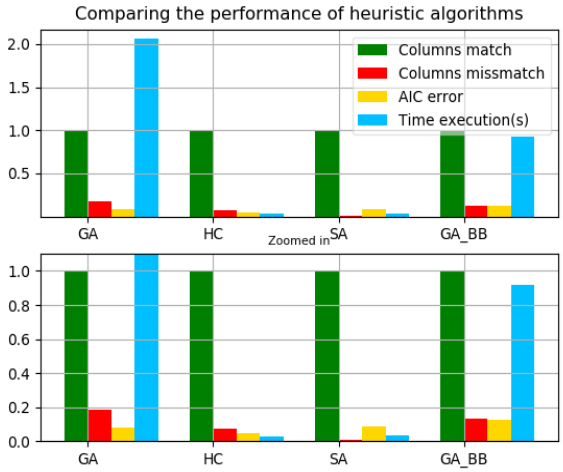


Figura 50 - *Comparing Algorithms*, config 4.1

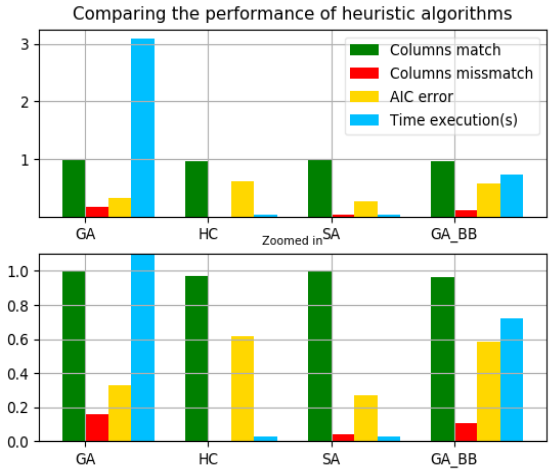


Figura 51 - *Comparing Algorithms*, config 4.2

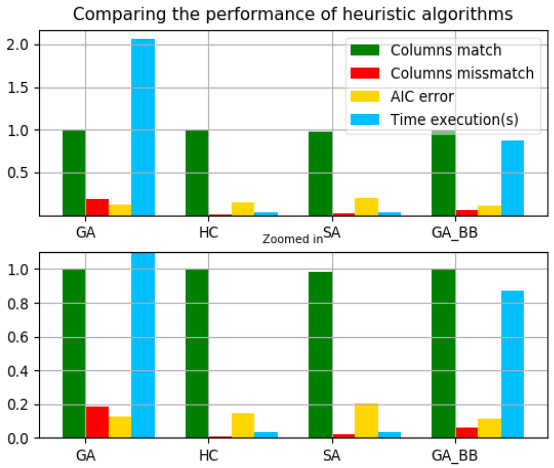


Figura 52 - *Comparing Algorithms*, config 4.3

În imaginea din Figura 50 am setat deviația standard de 0.3, iar în imaginea din Figura 51, am setat acest parametru cu valoarea 0.5. Se poate observa diferența adusă de acest parametru pe outputurile algoritmilor.
În condițiile în care datele sunt împrăștiate față de medie, algoritmul *hill climbing* nu reușește să obțină outputuri semnificative. Cu toate că algoritmul are valoarea 0 pentru *columns missmatch* , acesta pierde din coloanele celui mai bun subset.
În imaginea din Figura 52 am setat deviația standard cu valoare de 0.9, cu toate că așteptarea ar fi ca rezultatele să fie mai eronate, se întâmplă contrariul.

8.3 Avantaje versus dezavantaje

Având în vedere outputurile și observațiile discutate în subcapitolele precedente, dintre algoritmii implementați nu pot alege unul care să fie general valabil pentru orice set de date, care să rezolve orice tip de problemă.

Cu toate acestea am putut observa că algoritmul genetic pe toate configurațiile stabilite a generat soluții care conțin cel mai bun subset. Cu costul de a consuma un timp mai mare și cu costul de a oferi pe lângă coloanele celui mai bun subset și pe altele, acest algoritm produce rezultate bune, eroare având în cele mai multe cazuri valoare minimă, ceea ce oferă acestui algoritm credibilitate.

Algoritmul *hill climbing*, implementat cu o mutație care să permită individului curent de a-și modifca *genotype*-ul la nivel de mai mulți biți și nu *step by step*, poate genera soluții foarte bune. Deși acest algoritm oferă soluții diferite de la o rulare la alta, ceea ce îl face practic este timpul de execuție mic.

Algoritmul *simulated annealing* vine ca o îmbunătățire pentru *hill climbing*, așadar acesta poate fi utilizat unde seturile de date nu conțin foarte multe coloane și dorim un timp mic de așteptare.

Algoritmul *building blocks* oferă soluții bune atunci când numărul de observații este mare, de asemenea la fel ca și algoritmul genetic, acesta oferă de cele mai multe ori aproximări care cuprind toate coloanele celui mai bun subset.

9 Concluzii

Așadar am implementat o aplicație software în realizarea unui studiu comparativ al algoritmilor euristici pentru selecția unui sub-model de regresie cu o vizualizare grafică a comportamentului acestora atât pe set real de date cât și pe seturi generate. Aplicația este compatibilă pe platforma *Windows*, odată ce este instalat pachetul *Cygwin* împreună cu *Python 3.7.*

Pentru obținerea rezultatelor potrivite pe seturile de date, operatorii genetici și metodele de selecție pot fi implementate în foarte multe moduri. Și tocmai datorită faptului că algoritmii au un parcurs *random* este greu de stabilit dacă algoritmul produce un rezultat mai bun datorită schimbării parametrilor. Cu toate acestea, datorită implementării conceptului de *tuning* pe fiecare algoritm în parte, am înțeles comportamentul algoritmilor și am reușit să fac o comparație între aceștia.

9.1 Îmbunătățirea obținerii funcției fitness

În studiul algoritmilor euristici foarte importantă este stabilirea și obținerea funcției de cost a unui candidat. În această lucrarea am utilizat descompunerea QR pentru fiecare candidat în parte în vederea obținerii funcției RSS utilizată apoi în descrierea funcției de cost. Acest lucru poate fi îmbunătățit prin aplicarea descompunerii QR o singură data pe candidatul format din toate coloanele din setul de date. Urmând apoi, din acest model să se elimine coloane pentru obținerea tuturor combinațiilor de coloane, și calcularea funcției RSS prin aplicarea procedurilor de rotație si de retriangularizare asupra matricei R.

Ce m-a împiedicat în realizarea metodei eficiente pentru obținerea funcției RSS a fost faptul că eliminarea de coloane produce rezultate respectând o anumită ordine, spre exemplu dacă modelul nostru este format din 5 coloane {1, 2, 3, 4, 5} și vreau să obțin funcția RSS pentru candidatul cu coloanele {1, 3, 4, 5} atunci ar trebui să elimin coloana 2 dar acest lucru îmi va genera și valorile funcției RSS pentru {1, 3 ,4}, {1, 3} și {1} ce pot fi salvate și utilizate ulterior. Această strategie poate fi implementată mai ușor pentru metoda exhaustivă unde căutarea se face ordonat pe toate sub-modelele setului de date, însă această metodă eficientizată îmbinată cu algoritmi euristici unde căutarea și evaluare algoritmilor se face *random* nu este atât de intuitivă.

9.2 Îmbunătățirea comunicației dintre programe

Datorită faptului că am avut nevoie de câteva *script*-uri în *python* care să comunice cu aplicația C, am implementat un mic *toolchain* pentru a îmbina toate aplicațiile în obținerea outputului. Așadar pentru obținerea rezultatelor pe seturi generate, am apelat aplicația C din scriptul *python* pentru generarea statisticilor. În ceea ce privește rularea algoritmilor pe seturi reale de date, am îmbinat seria de comenzi într-un *batch* file.

Din cauza faptului că aplicația poate fi parametrizată în multe feluri în *script*-uri diferite din acest *toolchain* de funcționalități, aș îmbunătății modul de configurare al acestora, aș face un fișier global în format *json* cu toate configurațiile împărțite pe funcționalitate. De asemenea aș îmbunătății comunicația dintre aplicațiile *python* și C.

10 Bibliografie

1. *Parallel algorithms for computing all possible subset regression models using the QR decomposition.* **Cristian Gatu, Erricos J. Kontoghiorghes.** 2002.

2. **D.M. SMITH, J.M. BREMNER.** *All possible subset regressions using the QR decomposition* . 1998.

3. **JohnKONTOGHIORGHE, CristianGATU and Erricos.** *Branch-and-Bound Algorithms for Computing the Best-Subset Regression Models.*

4. **Carr, Jenna.** *An Introduction to Genetic Algorithms.* May 16, 2014.

5. **Schaalje, Alvin C. Rencher and G. Bruce.** *LINEAR MODELS IN.*

6. **Nitasha Soni, Dr Tapas Kumar.** *Study of Various Mutation Operators in Genetic.* 2014.

7. **Sheth, A.J. Umbarkar and P.D.** *CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW.* 2015.

8. **Azencott, R.** *Simulated annealing: parallelization techniques.* 1992.

9. **Spall, J. C.** *Introduction to stochastic search and optimization.* 2003.

10. **Forrest, Stephanie.** *Fitness Landscapes: Royal Road Functions.*

11. **Stefan Edelkamp, Stefan Schrödl.** *Heuristic Search, Theory and Applications.* 2012.

12. **Brian W. Kernighan, Dennis M. Ritchie.** *The C Programming language Second Edition.* 1988.

13. **GNU.** GNU Scientific Library Release 2.5. [Interactiv] Jun 14, 2018. https://www.gnu.org/software/gsl/.

14. Python documentation. [Interactiv] 2019. https://docs.python.org/3/.

15. Matplotlib documentation. [Interactiv] 2019. https://matplotlib.org/.