Soluție pentru planificarea în timp real în sisteme distribuite utilizând algoritmi genetici

Damian Cristian SILIȘTEANU[1], Bogdan Costel MOCANU[1], Mihnea Horia VREJOIU[2], Florin POP[2,1]

[1] Facultatea de Automatică și Calculatoare, Universitatea Politehnica din București

[2] Institutul Național de Cercetare-Dezvoltare în Informatică – ICI București

damian.silisteanu@stud.acs.upb.ro, bogdan_costel.mocanu@upb.ro, mihnea.vrejoiu@ici.ro, florin.pop@ici.ro / florin.pop@upb.ro

Rezumat: În ultima vreme, sistemele distribuite și-au demonstrat eficiența în procesarea unui număr mare de task-uri într-un timp cât mai redus. În acest context, planificatorul are cea mai mare influență pentru că el gestionează modul în care task-urile respective sunt procesate pe diferite resurse. Ne-am propus tratarea unei probleme de planificare având ca entități principale task-urile și resursele. Între aceste două entități am stabilit anumite constrângeri pentru a modela o problemă reală, iar pentru rezolvarea problemei de planificare am optat pentru dezvoltarea unui algoritm genetic. Pentru evaluarea performanțelor algoritmului am analizat modul în care acesta reușește să diminueze timpul de procesare și în același timp să îndeplinească constrângerile stabilite. De asemenea, am analizat impactul adăugării de resurse suplimentare în planificare pentru aceleași task-uri.

Cuvinte cheie: sisteme distribuite, sarcini, resurse, planificator, constrângeri, optimizare, algoritmi genetici.

Solution for real-time scheduling in distributed systems using genetic algorithms

Abstract: In recent times, distributed systems have demonstrated their efficiency in processing a large number of tasks in as little time as possible. In this context, the scheduler has the most influence because it manages how the respective tasks are processed on various resources. We set out to treat a planning problem with tasks and resources as main entities. Between these two entities we established certain constraints to model a real problem, and to solve the planning problem we opted for the development of a genetic algorithm. To evaluate the performance of the algorithm, we analyzed how it manages to reduce the processing time and at the same time fulfill the set constraints. We also analyzed the impact of adding additional resources in the schedule for the same tasks.

Keywords: distributed systems, tasks, resources, scheduler, constraints, optimization, genetic algorithms.

1. Introducere

Provocarea găsirii unor metode pentru eficientizarea timpilor de procesare pentru oricare sarcină a existat de-a lungul evoluției tehnicii de calcul. Astfel, inclusiv în cazul calculatoarelor monoprocesor s-a pus problema dezvoltării unui mod de gestiune eficientă a procesării task-urilor / thread-urilor pe respectivul procesor. Astfel au apărut algoritmii de planificare pentru arhitecturile monoprocesor. Odată cu apariția arhitecturilor multiprocesor, problema planificării a căpătat noi valențe. Pentru a dezvolta un planificator pentru acest tip de arhitectură, problema trebuie împărțită în două etape (Krishna & Shin, 1997): (i) găsirea unui mod de distribuire a task-urilor între procesoare și, apoi, (ii) folosirea unuia dintre algoritmii de planificare specifici monoprocesor pentru a gestiona task-urile alocate fiecărui procesor. În jurul problemei planificării s-a dezvoltat un întreg domeniu de cercetare. Astăzi, sistemele de calcul au devenit foarte complexe și se confruntă cu procesarea unui număr mare de task-uri, problema planificării fiind de mare actualitate.

Sistemele distribuite și supercalculatoarele sunt utilizate din ce în ce mai mult și sunt într-o continuă evoluție. Unul dintre avantajele acestora în procesarea de task-uri intensive computațional este evidențiat prin timpii de execuție reduși, în obținerea cărora planificatorul are un rol important. Planificarea în sistemele distribuite este o problemă NP-hard. De aceea, pentru optimizare se face de cele mai multe ori apel la algoritmi de aproximare. Important este ca acești algoritmi de planificare să reușească să atingă stadiul de echilibru Nash (Nash, 1950) pentru problemele propuse. Astfel, se pune problema menținerii unei balanțe echilibrate între timpii de execuție și costuri, întrucât sistemele distribuite sunt în general accesibile prin intermediul unui furnizor, care asigură

https://doi.org/10.33436/v32i3y202203

accesul utilizatorilor contra cost. În acest context, un mod de planificare cât mai inteligent aduce beneficii pe de o parte consumatorului, cel care își dorește ca task-urile sale să fie cât mai rapid procesate la un cost cât mai mic, iar pe de altă parte și furnizorului de servicii, care își dorește să aibă clienți și să obțină un profit pe seama serviciilor pe care le pune la dispoziție. Este vital ca prețul plătit de utilizator să fie atractiv și la îndemâna acestuia și, în același timp, pe lângă acoperirea costului generat de utilizarea și menținerea resurselor, acest preț trebuie să asigure și un profit rezonabil pentru furnizor. Scăderea timpului de execuție influențează pozitiv ambele aceste aspecte, precum și experiența utilizatorului.

Problema pe care ne-am propus să o rezolvăm se apropie prin caracteristici de problemele de planificare din sistemele distribuite reale. Principalele componente ale problemei sunt sarcinile, ca abstractizare a task-urilor ce pot fi trimise către un sistem distribuit de către utilizatori și resursele, ca abstractizare a calculatoarelor ce fac parte din sistemul distribuit. Pentru o apropiere cât mai mare de problema reală, am impus anumite constrângeri de memorie și capacitate asupra task-urilor și resurselor, iar pentru îndeplinirea execuției task-urilor în timp util am stabilit o constrângere a momentului de finalizare a fiecărui task. Focusul principal în cadrul problemei este modul prin care se poate micșora timpul de execuție al task-urilor, cu respectarea constrângerilor legate de memorie și capacitate și cu o eventuală încălcare cât mai mică a termenului maxim de finalizare a fiecărui task.

Soluția propusă implică realizarea planificării cu ajutorul unui algoritm genetic. Algoritmii genetici sunt o metodă inteligentă de determinare euristică a soluției aproximative. Acești algoritmi au fost dezvoltați, pornind de la ideea modelului genetic și modelului de selecție naturală, pentru a determina soluții optime ale problemei și pentru a explora inteligent spațiul de căutare al soluției, cu aplicații importante în planificare. Pentru gestionarea problemei utilizând un algoritm genetic am codificat o soluție posibilă ca o înșiruire a indicilor task-urilor, în ordinea în care acestea trebuie procesate pe o anumită resursă. Soluția globală conține un număr de liste de indici de această formă egal cu numărul de resurse, reprezentând task-urile alocate fiecărei resurse în ordinea de procesare a acestora. Întrucât algoritmul genetic se bazează ca metodologie pe aproximarea soluției, acesta are nevoie de stabilirea unui mod de evaluare a posibilelor soluții. În acest sens, am utilizat modul recompensă-penalizare pentru a direcționa explorarea de către algoritm a spațiului de soluții pentru găsirea uneia optime.

Rezultatul aplicării algoritmului genetic implementat este o posibilă planificare a task-urilor pe resursele respective. Această planificare este analizată verificându-se în primul rând să nu se încalce niciuna din constrângerile problemei. Este calculat timpul cu care au fost eventual încălcate termenele limită de procesare a task-urilor, dacă acest lucru s-a întâmplat. Se identifică timpul maxim de funcționare a resurselor și costul total de procesare a task-urilor. Algoritmul și modul de simulare al problemei fac posibilă și analiza situației în care se adaugă un număr arbitrar de resurse suplimentare pentru același set de task-uri, în ceea ce privește reducerea timpului maxim de funcționare a resurselor și costul de procesare. S-a observat faptul că adăugarea de resurse în planificare este benefică atâta timp cât costurile de utilizare pentru aceste resurse nu sunt prea mari.

Algoritmul dezvoltat și analiza realizată cu ajutorul acestuia asupra echilibrului dintre timpul de utilizare a resurselor și costul de procesare a task-urilor pot servi unui furnizor de servicii ce oferă acces utilizatorilor la un sistem distribuit. Furnizorul poate lua o decizie în urma simulării rulării algoritmului, dacă ar fi profitabil să pună la dispoziția anumitor utilizatori un număr mai mare de resurse și, mai exact, ce resurse ar fi bine să introducă în planificare. În afara simulării desfășurării planificării task-urilor pe resurse, algoritmul propus poate servi și ca algoritm de planificare propriu-zis.

În continuare, articolul este structurat după cum urmează. În secțiunea 2 sunt trecute în revistă soluțiile de planificare în timp real și situația actuală în utilizarea acestora. Secțiunea 3 este dedicată prezentării diverselor aspecte teoretice ale problemei, precum și a detaliilor legate de dezvoltarea și implementarea algoritmului genetic propus pentru rezolvarea acesteia, a modelului utilizat și a constrângerilor impuse. Secțiunea 4 conține rezultatele experimentale obținute și interpretarea acestora. În fine, în secțiunea 5 sunt schițate câteva concluzii și idei de continuare a cercetării.

2. Soluții utilizate pentru planificare

Algoritmii de planificare reprezintă baza domeniului planificării. Aceștia au fost dezvoltați pornind de la cele mai simple idei de gestiune a task-urilor și resurselor până la idei specifice cu aplicabilitate într-un anumit context țintă (Brucker, 2006). Algoritmii de planificare pot fi împărțiți în două categorii, non-preemptivi și preemptivi. Algoritmii non-preemptivi, odată ce au lăsat un proces să utilizeze procesorul, acesta este lăsat să ruleze până ce își termină execuția. Algoritmii preemptivi introduc prioritatea ca specificație a proceselor, planificându-l pentru rulare pe procesor pe cel cu prioritatea cea mai mare. Sistemul în care este utilizat algoritmul trebuie să asigure posibilitatea ca orice proces să poată fi oprit din execuție și trecut în așteptare în orice moment de timp în cazul apariției în lista de task-uri a unui proces cu prioritate mai mare. În cele ce urmează prezentăm succint câțiva algoritmi de planificare remarcabili analizați, care, din punct de vedere metodologic, au stat la baza algoritmului propus.

First Come First Serve (FCFS) (Hotovy, 1996). Principiul de funcționare al acestui algoritm este extrem de simplu, deci este și foarte ușor de implementat. Se urmărește ca procesele care solicită primele procesorul să fie planificate primele, gestiunea lor realizându-se cu ajutorul unei cozi. Din cauză că acest tip de algoritm este non-preemptiv, procesorul nu este eliberat până când nu termină execuția curentă. Din acest motiv, nu este un algoritm foarte eficient, mai ales când multe procese (task-uri) mici sunt ținute în coada de așteptare. Acest neajuns generează timpi de așteptare mari, lucru care nu este tocmai ideal pentru un sistem de timp real.

Priority Based Scheduling (PBS) (Haupt, 1989). Acest tip de algoritm poate fi și preemptiv și non-preemptiv, lucru ce conferă o oarecare flexibilitate la implementare. Ideea de gestiune a proceselor în cadrul algoritmului este de a le atribui acestora o prioritate și de a aloca pe procesor procesul cu prioritatea cea mai mare. Dacă există mai multe task-uri cu aceeași prioritate este nevoie de implementarea altui algoritm pentru a decide ce task va prelua procesorul. De regulă, se folosește un algoritm FCFS pentru simplitate. Implementarea preemptivă este ideală pentru sistemele în timp real deoarece, chiar dacă un task cu prioritate scăzută este deja alocat pe procesor, el poate fi preemptat și procesul cu prioritate ridicată îi va lua locul. Un dezavantaj al acestui tip de algoritm este faptul că un task cu prioritate scăzută riscă să rămână în coada de așteptare la infinit dacă apar numai task-uri cu prioritate mai ridicată.

Shortest Remaining Time First (SRTF) (Schrage, 1968). Algoritmul se bazează pe principiul că procesorul va fi alocat task-ului a cărui execuție se va termina în timpul cel mai scurt. Prin urmare, pentru ca acest algoritm să poată fi implementat, trebuie cunoscut timpul necesar până la finalizare pentru fiecare task și, totodată timpul total de rulare a fiecarui task pe procesor. Problema care poate să apară este că, dacă vor fi primite succesiv numai task-uri cu timpi mici de execuție, task-urile cu timp mai mare nu vor fi alocate pe procesor niciodată, sau foarte rar („starvation").

Round Robin (RR) (Hahne, 1986). Acest algoritm este, de asemenea, unul simplu de înțeles și ușor de implementat. Fiecare proces va primi, pe rând, o cuantă egală de timp pe procesor. Acest tip de algoritm este ideal pentru multitasking deoarece fiecare proces va fi alocat pe procesor la un moment dat. Deoarece procesele vor rula pe rând, se poate aborda alocarea lor fără a folosi priorități și problema de „process starvation" dispare. Însă, există și câteva dezavantaje care trebuie amintite. Această metodă se bazează mult pe „context switching", respectiv pe oprirea procesului ce rulează și alocarea pe procesor a altui proces. În acest mod se pierde destul de mult timp când se alocă procesorul de la un proces la altul. Cu cât cuanta de timp este mai mică cu atât overhead-ul provocat de „context switching" este mai mare, iar identificarea unei cuante de timp ideale este un proces dificil în sine.

Multilevel Queue Scheduling Algorithm (Omar et al., 2021). În cazul în care procesele sunt ușor de clasificat, se poate folosi o nouă clasă de algoritmi de planificare. Practic se încearcă o hibridizare de algoritmi de planificare folosind mai multe cozi. Fiecare dintre aceste cozi folosește, de regulă, unul din algoritmii prezentați anterior. În cadrul algoritmului, fiecare coadă va avea o prioritate și, prin urmare, până când coada cu prioritate maximă nu este goală, toate celelalte cozi vor fi în așteptare. Complexitatea acestui tip de planificare este dată de numărul cozilor și complexitatea algoritmilor folosiți pentru fiecare coadă. Se poate afirma că, dacă este implementată

corect, această metodă este cea mai completă și performantă cale de a aloca procesele pe procesor. Din păcate există și dezavantaje, dintre care cel mai evident constă în existența riscului ca o coadă de prioritate ridicată să primească în continuu procese, iar celelalte cozi cu prioritate scăzută să aștepte timp îndelungat golirea cozii prioritare.

Sistemele în timp real sunt sisteme ce lucrează cu task-uri a căror rulare corectă nu depinde doar de corectitudinea logică, ci și de timpul de răspuns. Din acest punct de vedere sistemele în timp real se împart în două categorii (Laplante, 2004):

- *Soft real-time systems*: În cadrul acestor sisteme, un termen limită poate fi ratat. Acest lucru este permis pentru că task-ul respectiv poate fi reprogramat sau îi este permis să fie finalizat după timpul limită.
- *Hard real-time systems*: În cadrul acestor sisteme, un termen limită neîndeplinit poate genera pierderi uriașe. Task-urile trebuie să fie efectuate la un moment specific de timp și răspunsul sistemului trebuie să fie cât mai rapid.

În sistemele în timp real, planificatorul joacă cel mai important rol. Obiectivul principal al acestuia este de a reduce timpul de răspuns al tuturor task-urilor. Dacă se utilizează un planificator preemptiv, task-urile trebuie să aștepte până ce task-ul curent își termină de executat cuanta. În cazul unui planificator non-preemptiv, chiar dacă se asociază task-ul cel mai urgent cu prioritatea cea mai mare, acesta tot trebuie să aștepte până la finalizarea task-ului curent. Problema este că task-ul curent poate fi lent sau de prioritate scăzută și poate duce la o întârziere îndelungată a task-ului urgent. Abordarea consacrată în cazul planificării pentru sistemele în timp real este de a combina cele două strategii de planificare, preemptivă si non-preemptivă. Acest lucru se realizează prin introducerea întreruperilor pe o perioadă de timp în sistemele bazate pe prioritate. În această situație procesul curent este întrerupt pentru o perioadă de timp dacă în coada de procese apare un proces cu prioritate mai mare, ce trebuie alocat cât mai repede pe procesor. În Figura 1 este reprezentat, pe baza analizei algoritmilor de planificare existenți, modul de clasificare al planificatoarelor sistemelor în timp real.

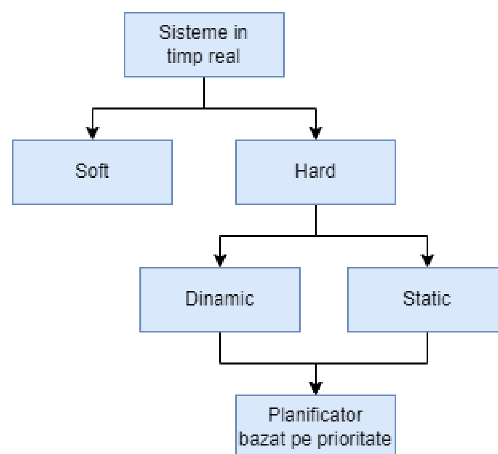


Figura 1. Criteriile de clasificare ale planificatoarelor sistemelor în timp real

În funcție de modul de implementare și constrângerile impuse, algoritmii de planificare pot fi clasificați (Dhall & Liu, 1978) în:

- *Algoritmi statici bazați pe tabele*: Acești algoritmi realizează o analiză statică asociată cu planificarea și rețin planificările avantajoase. Acest lucru poate ajuta la identificarea unui task cu care trebuie începută execuția.
- *Algoritmi statici preemptivi bazați pe prioritate*: Față de categoria anterioară acești algoritmi realizează analiza statică pentru a obține o metodă de atribuire a priorității task-urilor în planificarea preemptivă.
- *Algoritmi dinamici de planificare*: În acest caz, planificările realizabile sunt identificate în timpul rulării. Se alege un interval fix și un proces este executat doar dacă își îndeplinește constrângerile în acest interval.

- *Algoritmi dinamici bazați pe minimizarea efortului*: Acești algoritmi consideră ca principală constrângere îndeplinirea termenului limită. Sunt folosiți în foarte multe dintre sistemele în timp real pentru că, dacă un task își depășește termenul, este abandonat.

Problema planificării în sistemele distribuite și în sistemele Cloud este una NP-hard, fiind dificil să se obțină o soluție optimă exactă. Tendința actuală este de a folosi algoritmi inteligenți de optimizare pentru a aproxima soluția optimă. Cel mai adesea, în rezolvarea problemelor de planificare, se folosesc algoritmi euristici pentru căutarea soluției optime. Totuși, pe baza analizei făcute, putem afirma că acești algoritmi devin însă ineficienți în cazul datelor de intrare complexe.

Algoritmii genetici s-au dovedit adecvați pentru rezolvarea problemelor de planificare (Constantinescu, 2004; Şerban & Carp, 2021), aceştia căutând o soluție optimă prin aproximare dintr-o populație de soluții, spre deosebire de metodele euristice care folosesc o singură soluție. Beneficiul existenței unei multitudini de soluții posibile este că, prin analiza acestora, algoritmul poate alege evolutiv soluțiile cele mai potrivite. Un avantaj al algoritmilor genetici în rezolvarea problemelor de planificare este modul simplu de orientare către soluția optimă pe baza unor criterii stabilite. De exemplu, se poate dori micșorarea timpului total de procesare, a consumului de energie al resurselor sau asigurarea îndeplinirii anumitor constrângeri ce apar între task-uri și resurse. Există numeroase articole de specialitate care analizează utilizarea algoritmilor genetici în diverse probleme de planificare pentru sisteme distribuite (Wang et al., 2008), sisteme Cloud (Naithani, 2018) și supercalculatoare (Abdulal et al., 2009), iar tendințele actuale de cercetare în domeniul planificării pun din ce în ce mai mult accent pe acest tip de algoritmi (Omara & Arafa, 2008; Bacalhau et al., 2021; Barredo & Puente, 2022).

3. Metoda și algoritmul propus

În această lucrare ne-am propus dezvoltarea unui algoritm genetic pentru o problemă de planificare care, prin modul de parametrizare, se apropie de problemele reale din sistemele distribuite din zilele noastre. Constrângerile pe care le-am impus între resurse și task-uri modelează posibilitatea procesării și respectarea caracteristicilor resurselor. În continuare prezentăm în detaliu diversele aspecte teoretice ale problemei, entitățile și constrângerile implicate, precum și componentele și metodologia de dezvoltare și implementare a algoritmului ce o rezolvă.

Task-urile sunt abstractizarea sarcinilor, reprezentând entitățile ce trebuie planificate pentru procesare. Prin caracteristicile lor, task-urile modelează entități reale din diverse probleme de planificare. Astfel, un task T_i (task-ul cu indicele i) se caracterizează prin următoarele:

- p_i = timpul necesar execuției;
- d_i = momentul de timp până la care trebuie să își încheie execuția;
- in_i = spațiul necesar stocării datelor de intrare de care are nevoie;
- out_i = spațiul necesar stocării datelor de ieșire.

Resursele sunt abstractizarea calculatoarelor, reprezentând entitățile ce realizează procesarea task-urilor. Prin caracteristicile lor, resursele modelează câteva aspecte cheie ale dispozitivelor implicate. Astfel, o resursă, R_i, (resursa cu indicele i) se caracterizează prin:

- mem_i = spațiul de stocare ce poate să îl pună la dispoziție;
- cap_i = capacitatea de procesare măsurată în unități de timp;
- $cost_i$ = costul de utilizare pe unitate de timp.

Dorim ca problema propusă să se apropie cât mai mult de condițiile unei probleme de planificare reale ce poate apărea în sistemele distribuite. În acest sens, am decis stabilirea unor constrângeri rezonabile între task-uri și resurse. Astfel, constrângerile pentru ca un task T_i să poată fi alocat pe o resursă R_j sunt:

- spațiul total de stocare ocupat de datele de intrare și datele de ieșire ale task-ului T_i să aibă o dimensiune mai mică sau egală cu spațiul de stocare ce poate să îl pună la dispoziție resursa R_j: $in_i + out_i \leq mem_j$;
- suma timpilor de procesare ai task-urilor ce sunt alocate pe resursa R_j să nu depășească capacitatea resursei respective: $\sum(p_i) \leq cap_j$.

Pentru procesarea fără penalizare a oricărui task *i*, condiția este ca timpul până la preluarea spre execuție a task-ul respectiv, $delay_i$, însumat cu timpul necesar procesării acestuia, să nu depășească momentul de timp până la care trebuie să își încheie execuția: $delay_i + p_i \leq d_i$.

În cadrul problemei propuse, parametrizată prin entitățile și constrângerile prezentate mai sus, dorim minimizarea timpului maxim de utilizare a resurselor și analizarea penalizărilor de depășire în procesare a termenelor limită ale task-urilor. Timpul maxim de utilizare a resurselor T_{max} este dat de maximul dintre timpii de rulare ai resurselor: $T_{max} = max\{ T_j \} \ | \ j = 1 \div n$, unde *n* este numărul de resurse și T_j este timpul de utilizare al resursei *j*.

Aceste constrângeri modelează problemele de planificare în care un task nu poate fi atribuit oricărei resurse și, în același timp, o resursă nu are o capacitate infinită de procesare, aceasta devenind indisponibilă după ce și-a atins capacitatea de procesare. Astfel de constrângeri sunt stabilite de furnizorul care oferă acces la sistemul distribuit, iar constrângerea legată de îndeplinirea la timp a task-urilor este stabilită de utilizatorul sistemului respectiv.

Un algoritm genetic este un model informatic care emulează modelul biologic evoluționist (Holland, 1992) pentru a rezolva probleme de optimizare și căutare. Algoritmii genetici folosesc, ca reprezentare a soluțiilor posibile ale problemei, șiruri cu informație codificată, numite cromozomi. O entitate din cadrul algoritmului, ce poartă denumirea de individ, are în componența sa un astfel de cromozom și are asociat un mod de evaluare al soluției care o conține. Algoritmul modelează spațiul de căutare al soluției prin păstrarea unui set de astfel de entități, denumit populație. În cadrul algoritmului este pus la dispoziție și un set de operatori de natură biologică definiți asupra populației respective. Cu ajutorul acestor operatori, algoritmii genetici manipulează cele mai promițătoare entități, evaluate conform unei funcții obiectiv, căutând soluții mai bune. În cazul nostru, problema de planificare vizează modul în care task-urile pot fi gestionate pentru a fi alocate optim pe resurse. Prin urmare, am ales ca reprezentare a soluției codificate înșiruirea indicilor task-urilor ce sunt alocate pe o anumită resursă, în ordinea în care acestea trebuie procesate. Indicii task-urilor nu se repetă, iar lista task-urilor alocate pe fiecare resursă este ordonată crescător după indicii resurselor. Figura 2 exemplifică reprezentarea cromozomului pentru o problemă de planificare cu 3 resurse și 8 task-uri.

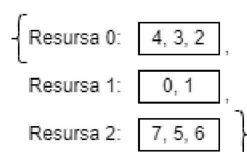


Figura 2. Reprezentarea cromozomului

În Figura 3 este reprezentată schema generală a unui algoritm genetic. Algoritmul începe prin inițializarea populației și evaluarea acesteia cu ajutorul funcției de fitness (funcția obiectiv, *F*). Pe baza evaluării realizate se verifică și condiția de terminare a algoritmului. Dacă această condiție nu este satisfăcută, se continuă cu procesul de obținere a noii populații, iar dacă este satisfăcută, algoritmul se oprește și returnează soluția aproximativă găsită. Pentru realizarea unei noi populații, populația actuală trece printr-un proces de prelucrare în trei pași: selecție, încrucișare și mutație. Selecția realizează identificarea indivizilor cu cel mai mare potențial și organizează un mod în care aceștia vor fi folosiți drept părinți pentru indivizii din populația viitoare. Încrucișarea realizează schimbul de informație între doi părinți ce au fost selectați, pentru a obține un nou individ. Mutația este procesul prin care informația din cromozomii noilor indivizi se poate modifica, simulând procesul biologic de mutație a genelor. La finalul mutației, noua populație este definitivată și se aplică asupra ei funcția obiectiv. Procesul continuă până ce condiția de terminare este satisfăcută.

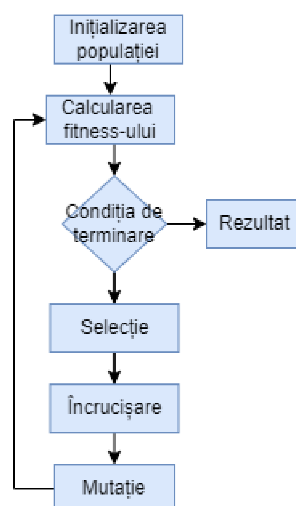


Figura 3. Pașii din cadrul algoritmilor genetici

Inițializarea populației. În principiu, în algoritmii genetici sunt folosite două metode de inițializare a populației:

- *inițializarea aleatoare*: Se bazează pe așezarea în populația inițială a unor soluții generate aleator;
- *inițializarea euristică*: Se bazează pe așezarea în populația inițială a unor soluții generate pe baza unei euristici cunoscute pentru problema abordată.

Am observat experimental că o inițializare euristică a populației conduce la soluții foarte asemănătoare între ele. Populația inițială beneficiază de valori bune ale funcției de fitness, dar acest fapt nu are un efect semnificativ asupra generațiilor următoare. Aceste soluții euristice micșorează diversitatea populației. Însă, pentru ca algoritmul să aibă capacitatea de a explora semnificativ spațiul de căutare, acesta are nevoie de soluții cât mai diverse. Această diversitate necesară este asigurată de o inițializare aleatoare a populației.

Calcularea fitness-ului. O funcție de fitness evaluează soluțiile, valoarea produsă de aceasta indicând cât de „bună" este soluția respectivă. Funcția trebuie să aibă următoarele caracteristici:

- *Rapid de calculat*: Funcția de fitness trebuie să își termine evaluarea cât mai repede pentru că, în cadrul algoritmului genetic, aceasta este evaluată de foarte multe ori. O funcție ce produce o încărcare computațională mare scade performanțele algoritmului.
- *Evaluare calitativă*: Funcția de fitness trebuie să furnizeze o evaluare calitativă pentru cât de bună este soluția.

Funcția de fitness utilizată de noi este definită prin formula:

$$F = a\sum_{t,r}\frac{M_{t,r}}{M^{r}_{max}} + b\sum_{t,r}\frac{C_{t,r}}{C^{r}_{max}} + c\sum_{t,r}\frac{D_{t,r}}{D^{r}_{max}} + d\,\frac{1}{nT_{max}}\sum_{t,r}T_{t,r}$$

$$a = 10, b = 2, c = \frac{2}{1000}, d = 1/2$$

unde $M_{t,r}$ reprezintă memoria consumată pentru execuția task-ului t pe resursa r, $C_{t,r}$ este capacitatea de procesare necesară pentru execuția task-ului t pe resursa r, $D_{t,r}$ este deadline-ul pentru task-ul t cu resursa r, iar $T_{t,r}$ este timpul de execuție a task-ului t folosind resursa r. Prin intermediul funcției de fitness implementate se realizează modelarea constrângerilor problemei și minimizarea timpului maxim de procesare al resurselor. Algoritmul urmărește maximizarea funcției de fitness (funcție obiectiv) pentru apropierea de o soluție optimă.

Condiția de terminare. Condiția de terminare este foarte importantă în cadrul algoritmilor genetici. Am observat că, inițial, algoritmul produce soluții din ce în ce mai bune foarte repede, în curs de câteva iterații. Această tendință se saturează în cursul iterațiilor târzii, când îmbunătățirea

soluțiilor este foarte mică. Se dorește ca algoritmul să își termine rularea atunci când soluția găsită este aproape de cea optimă. Cele mai utilizate condiții de terminare sunt: (i) după ce s-a realizat un număr stabilit de generații; (ii) când nu s-a mai realizat nicio îmbunătățire în populație de un număr de iterații; (iii) când se îndeplinește o marjă de eroare setată. Pentru a ne asigura că algoritmul nu va ajunge în situații de suprareglaj (overfitting), am optat pentru determinarea experimentală a unui număr maxim de generații după realizarea cărora algoritmul să se termine.

Operatorii algoritmului genetic asigură transformările efectuate asupra populației pentru a obține o nouă populație din aceasta. Operatorii joacă un rol vital în componența algoritmului pentru că, atât diversitatea cât și convergența către optim a soluției sunt direct influențate de aceștia.

Selecția. Este procesul prin care sunt alese soluții din generația curentă ce urmează să fie folosite în cadrul încrucișării pentru a crea noi soluții. Aceste soluții alese poartă denumirea de părinți. Acest proces de alegere al părinților influențează vital rata de convergență a algoritmului către soluții mai bune. În același timp, selecția este utilizată și pentru a preveni posibila acaparare de către o soluție foarte bună a întregii populații, ceea ce ar duce la diminuarea diversității populației și o convergență prematură a algoritmului. Metodele de selecție analizate de noi sunt:

- *Selecție de tip turneu*: În cadrul acestei metode se stabilește un număr de indivizi ce sunt selectați aleator din populația curentă. Dintre aceștia, cel mai bun din punctul de vedere al funcției de fitness devine părinte.
- *Selecție de tip ruletă*: Este o metodă de selecție proporționată pe baza fitness-ului. Fiecare individ are o șansă de încrucișare proporțională cu fitness-ul său. Părinții sunt aleși dintre toți indivizii pe baza probabilității fiecăruia. Această metodă pune accent pe soluțiile mai bune, dezvoltând soluții și mai bune în timp. Am păstrat această metodă de selecție în cadrul algoritmului datorită performanțelor obținute.

Încrucișarea. Procesul de încrucișare este inspirat din procesul biologic de reproducere (Holland, 1992) În acest proces, mai mult de un părinte este selectat pentru a obține un nou individ prin combinarea genelor părintești. Acest operator este de obicei aplicat cu o probabilitate mare. Tipurile de încrucișare pe care le-am analizat sunt:

- *Încrucișare într-un singur punct*: Se alege un punct aleator pe reprezentarea genelor celor doi părinți și se interschimbă informația din cei doi cromozomi de până la, sau de după acel punct. Astfel se obțin doi noi indivizi.
- *Încrucișare în mai multe puncte*: Această metodă este o generalizare a încrucișării într-un singur punct. Segmentele din gene ce se află între punctele alese se interschimbă.

În cadrul algoritmului nostru, metoda de încrucișare aleasă pornește de la cea într-un singur punct, dar nu păstrează informația ce se interschimbă între cromozomi exact în starea în care se afla în părinți. În exemplul din Figura 4 se observă că încrucișarea poate genera repetarea sau dispariția unor indici în codificarea soluției. Noi am asigurat unicitatea și prezența tuturor indicilor în soluție.

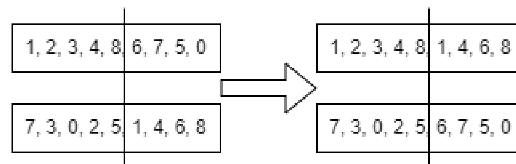


Figura 4. Exemplu de încrucișare într-un singur punct

Mutația. În termeni simpli, în cadrul procesului de mutație se produce o mică modificare asupra cromozomului pentru a obține o nouă soluție. Acest proces ajută la păstrarea diversității. Totuși, este aplicat cu o probabilitate mică în cadrul algoritmului pentru a nu dezechilibra rata de convergență. Tipurile de mutații analizate au fost:

- *Interschimbare*: În acest proces se aleg două poziții aleatoare din cadrul codificării soluției și se interschimbă valorile de la pozițiile respective.
- *Amestecare*: În acest proces se alege o porțiune din cromozom și genele din această zonă sunt amestecate.

Am ales ca metodă de mutație interschimbarea, ușor modificată, astfel încât accentul să se pună pe migrarea unui task din planificarea unei resurse în planificarea altei resurse. Figura 5 exemplifică modul cum are loc mutația. Task-ul cu indicele 4 este ales din planificarea resursei cu indicele 0 și este mutat pe o poziție aleatoare în planificarea resursei cu indicele 1. Tot în cadrul procesului de mutație este asigurată unicitatea și prezența tuturor indicilor de task-uri în planificare.

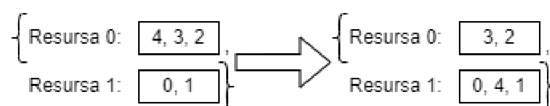


Figura 5. Exemplu de realizare a mutației

Implementarea algoritmilor genetici este un proces destul de complex. Pentru dezvoltarea algoritmului propus, noi am optat pentru limbajul Python, pentru care există o mare varietate de biblioteci utile pentru algoritmul nostru genetic, în care este nevoie de gestionarea de probabilități, de generarea de numere aleatoare, de manipularea rapidă a diverselor structuri de date cu ajutorul cărora sunt implementate diversele componente. Un alt avantaj al limbajului este posibilitatea de programare atât structurată, cât și orientată pe obiecte, ceea ce a permis modelarea diverselor caracteristici și dependențe din schema algoritmului cu ajutorul claselor. Existența listelor ca tip de date (clasă) și a metodelor disponibile pentru operarea cu acestea este foarte utilă. De asemenea, de utilitate în manipularea eficientă a tipurilor de date sunt funcțiile anonime, sau funcțiile map și filter, puse la dispoziție de limbajul Python. Algoritmul și modul de simulare al problemei au condus la o implementare structurată în mai multe fișiere, după scopul acestora: implementarea efectivă a algoritmului, funcții de verificare a constrângerilor și calcularea anumitor părți din funcția de fitness, modulul de simulare, modulul de generare a task-urilor și resurselor.

Pe baza metodologiei prezentate anterior, detaliem în continuare implementarea propriu-zisă a algoritmului, începând cu modul de generare a datelor de intrare. Datele de intrare generate constau în caracteristicile task-urilor și resurselor. În cadrul acestei etape, pe lângă generarea parametrilor ce definesc cele două entități ale problemei de planificare, am realizat și anumite verificări că datele generate permit rezolvarea problemei. Astfel, am verificat că task-ul generat are posibilitatea de a fi alocat pe un procent de 50% din resursele generate din punctul de vedere al constrângerii de memorie. De asemenea, am verificat dacă suma capacităților de procesare ale resurselor este mai mare decât suma timpilor de procesare ai task-urilor.

Generarea task-urilor constă în fixarea celor patru parametri ce definesc un task: timpul de procesare, termenul limită (deadline-ul), spațiul necesar stocării datelor de intrare și, respectiv, de ieșire. Am decis ca timpul de procesare să fie între 5 și 50 de secunde pentru ca planificatorul, care nu este preemptiv, să permită unui task să fie rulat integral la un moment dat. Deadline-ul este generat pe baza numărului de task-uri și resurse pentru a avea valori cât mai apropiate de cazurile reale. Nu avem cum să stabilim deadline-uri foarte drastice dacă avem un număr foarte mare de task-uri ce trebuie executate cu puține resurse. Din acest motiv, am decis ca deadline-urile să fie alese ca un moment de timp între capacitatea minimă și cea maximă de procesare a task-urilor de către numărul dorit de mașini, care au fost determinate experimental astfel încât task-urile generate să aibă posibilitatea de a-și îndeplini deadline-ul generat. Valorile spațiilor de stocare a datelor de intrare și de ieșire ale task-urilor au fost generate aleator, între 500 și 4000 de MB.

Generarea resurselor constă în găsirea modului de obținere a celor trei parametri ce definesc o resursă: dimensiunea memoriei pe care o pune la dispoziție, capacitatea de procesare pe care o are și costul de utilizare pe secundă al resursei. Dimensiunea memoriei, ca sumă a spațiului necesar datelor de intrare și de ieșire, este generată aleator între 6000 și 10000 MB. Cele două valori sunt alese astfel pentru a permite generarea de valori diverse în jurul maximului de memorie ce poate fi necesar unui task, estimat la 8000 MB. Capacitatea de procesare a resursei este generată în raport cu numărul de task-uri și resurse ce iau parte la planificare, pentru a garanta posibilitatea de procesare a tuturor task-urilor. De exemplu, dacă avem 100 de task-uri și 20 de resurse acestea au o capacitate pentru a putea planifica toate task-urile. Dacă, în acest caz, numărul de resurse este scăzut la 10, capacitatea de procesare a resurselor trebuie să crească, pentru a reuși să planificăm toate task-urile. Limitele între care se generează aleator capacitatea de procesare a resursei au fost determinate experimental după încercări repetate de calibrare a problemei.

Pentru a reprezenta cromozomul, respectiv posibilele soluții, am utilizat o listă de liste de clase, așa cum este redat în Figura 6.

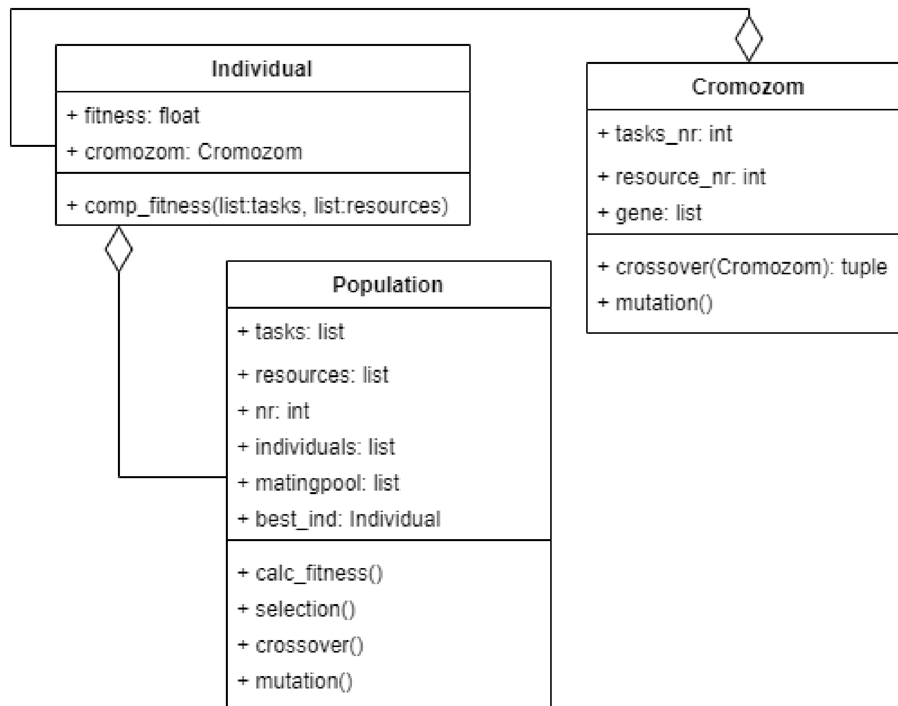


Figura 6. Diagrama claselor utilizate

Clasa Cromozom. Câmpul "gene" din cadrul clasei Cromozom păstrează această listă de liste. Listele componente corespund indicilor resurselor și stochează indicii task-urilor în ordinea în care s-a decis planificarea acestora pe resursa respectivă. Clasa Cromozom păstrează în interiorul său numărul de task-uri și numărul de resurse și implementează metodele de mutație și încrucișare cu un alt cromozom. Constructorul clasei Cromozom realizează, pe baza numărului de task-uri și de resurse primite, inițializarea aleatoare a listei ce reprezintă soluția. Lista de liste "gene" conține codificarea soluției. Din aceasta se alege aleator o listă și se adaugă în ea un indice de task ales aleator. Apoi indicele este eliminat dintre posibilele alegeri pentru a se asigura unicitatea sa. Clasa Cromozom realizează implementarea propriu-zisă a operatorilor de încrucișare și mutație. Încrucișarea are nevoie și de materialul genetic al celuilalt părinte, de aceea funcția primește ca parametru încă un cromozom. Funcția de mutație realizează modificări în cadrul cromozomului individului și nu are nevoie de niciun parametru.

Clasa Individ. Această clasă modelează indivizii din cadrul populației prezente. Individul este caracterizat de câmpul „fitness", care păstrează valoarea asociată acestuia la ultima evaluare a funcției de fitness. Am păstrat în interiorul clasei referința la cromozomul ce este legat de individ, care este evaluat de funcția de fitness. Funcția de fitness este implementată în cadrul clasei Individ și modelează constrângerile problemei. La începutul funcției am verificat dacă în soluție există vreo resursă ce nu a primit niciun task și în acest caz valoarea fitness-ului este lăsată setată la 0. Acest lucru încurajează alocarea cât mai rapidă a cel puțin unui task pe fiecare resursă. În continuare, se analizează încălcarea constrângerilor stabilite pentru problema abordată:

- *Constrângerea de memorie*: Se verifică câte task-uri au fost așezate pe resurse ce nu pot satisface nevoia spațiului de memorie dorit. O funcție "check_mem_problem" returnează procentul de task-uri ce au depășit această constrângere cu semn negativ.
- *Constrângerea de capacitate*: Se verifică dacă există în planificarea analizată resurse ce și-au depășit capacitatea de execuție prin suma timpilor de procesare a task-urilor alocate. O funcție "check_cap_problem" returnează procentul de resurse ce au depășit această constrângere cu semn negativ.

- *Constrângerea îndeplinirii deadline-urilor*: Se verifică timpul total de depășire al deadline-urilor task-urilor. O funcție "check_deadlines" returnează suma acestor timpi.

După stabilirea constrângerilor ce au fost încălcate și calcularea indicilor de încălcare, se stabilește recompensa ce se folosește în calcularea efectivă a valorii fitness-ului. Se stabilesc limitele de acceptare ale penalizărilor și se realizează ponderarea acestor valori de penalizare. Am dorit respectarea strictă a constrângerii de memorie. De aceea, această constrângere a primit ponderea cea mai mare în penalizare, respectiv 50%. Funcția de fitness are de obicei valori între 0 și 1. Astfel, am stabilit ponderea penalizării de memorie la -0,5 din valoarea maximă acceptată. Analog, am calculat ponderile celorlalte două constrângeri prioritizând îndeplinirea deadline-urilor în defavoarea constrângerii de capacitate. Aceste două constrângeri sunt legate între ele, pentru că, prin încercarea îndeplinirii deadline-urilor, se asigură (și) nedepășirea capacității resurselor. Aceste penalizări trebuie să fie minimizate în funcția de fitness și, deci, în formula efectivă a funcției, apar ca scăzute din valoarea maximă posibilă a acesteia (valoarea 1). Un al doilea termen al expresiei funcției de fitness vizează minimizarea timpului maxim de utilizare a resurselor. Acest termen l-am gândit pe principiul recompensei. Am normalizat timpul maxim de utilizare a resurselor aducând valorile dintr-un interval nedefinit în intervalul (0, 1) prin raportarea la timpul total de procesare a task-urilor și la numărul de resurse disponibile. Acest termen garantează că scăderea timpului maxim duce la o recompensă mai mare. În fine, am ponderat la 50% acest al doilea termen pentru ca funcția de fitness să fie influențată mai mult de primul termen, care minimizează penalizările, respectiv în proporție de 75%.

Clasa Populație. Această clasă modelează generațiile de indivizi din cadrul algoritmului genetic. Atributele clasei sunt lista de task-uri, lista de resurse, numărul de indivizi din cardul unei generații, lista cu indivizii din populația curentă, lista cu indivizii selectați pentru încrucișare și individul cu fitness-ul cel mai mare din generația curentă. Clasa asigură prin constructor inițializarea aleatoare a indivizilor ce alcătuiesc prima generație. În același timp, se inițializează și celelalte atribute ale clasei. Metodele de calculare fitness, selecție, încrucișare și mutație asigură aplicarea operatorilor pe toți indivizii ce alcătuiesc generația. În cadrul metodelor de încrucișare și mutație din această clasă, sunt modelate probabilitățile cu care se realizează acești operatori. De exemplu, pentru modelarea probabilității de 80% în realizarea încrucișării, am folosit o listă cu valori întregi de la 0 la 5 și, alegând aleator unul din elementele din listă, se verifică dacă acesta este mai mic decât cea mai mare valoare din listă, 4. Dacă această condiție este îndeplinită, înseamnă că încrucișarea se poate realiza. Analog am gestionat probabilitatea realizării mutației.

Operatorii sunt, după funcția de fitness, foarte importanți în cadrul algoritmului genetic. În diagrama de clase se observă că am ales să implementăm acești operatori secvențial pe drumul de la populație către cromozomi. Clasa care gestionează populația implementează propriu-zis doar operatorul de selecție, celelalte două funcții realizând doar anumite etape din cadrul operatorilor pe care îi implementează sau, pur și simplu, realizând apelul funcției ce implementează operatorul din clasa Cromozom pe cromozomul conținut de fiecare individ din populație.

Selecția. Este realizată în întregime în clasa care gestionează populația, acest operator având efect asupra întregii populații, pregătind-o de încrucișare. Am realizat selecția prin metoda aleasă în capitolul anterior, mai exact selecția de tip ruletă. Procesul începe prin determinarea valorii maxime a fitness-ului indivizilor din generația curentă. Dacă această valoare maximă este nulă înseamnă că toți indivizii încalcă constrângerea de a distribui cel puțin un task pe fiecare resursă. În acest caz, am optat pentru popularea listei "matingpool" cu indivizi generați aleator. Această listă conține indivizii selectați pentru încrucișare. În continuare, având o valoare maximă pozitivă și nenulă a fitness-ului se poate realiza o normare a tuturor valorilor indivizilor prin împărțirea la această valoare maximă. Astfel, valorile fitness-ului pentru toți indivizii sunt în intervalul [0, 1]. Acesta este momentul în care fiecare individ din generația curentă primește o probabilitate de încrucișare. Probabilitatea fiecărui individ este direct proporțională cu valoarea fitness-ului său. Astfel, am calculat numărul de indivizi de același fel ce trebuie introduși în lista din care se vor alege aleator părinții pentru a garanta probabilitatea de încrucișare. Pentru un adaos de diversitate, am introdus în lista pregătită pentru încrucișare un individ generat aleator, cu o probabilitate foarte mică de alegere pentru încrucișare, dar care ajută la diminuarea riscului de convergență prematură.

Încrucișarea. Este realizată ca etapă inițială în clasa “Population”, prin alegerea aleatoare a doi părinți din lista cu indivizii pregătiți în urma procesului de selecție. Etapa finală de combinare efectivă a informației din cromozomii celor doi părinți este realizată în clasa “Cromozom”. În metoda “crossover”, din cadrul clasei “Population”, se aleg aleator părinții din “matingpool”, se modelează probabilitatea de realizare a încrucișării între cei doi părinți și se apelează efectiv metoda de “crossover” din clasa “Cromozom”. Tot aici se garantează păstrarea constantă a numărului de indivizi din populație. În metoda “crossover” din cadrul clasei “Cromozom” se realizează procesul de combinare a informației. Pentru a reda mai explicit metoda de încrucișare implementată, vom utiliza exemplul din Figura 7.

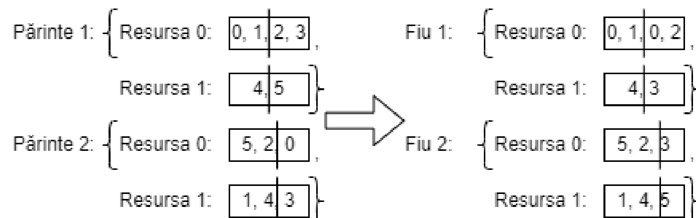


Figura 7. Exemplu de realizare a încrucișării

În partea stângă sunt reprezentați cromozomii celor doi părinți. Încrucișarea pe care am implementat-o pornește de la ideea încrucișării într-un singur punct. În interiorul cromozomului, informația se combină în cadrul listelor ce reprezintă planificări ale aceleiași resurse. Prima dată se realizează schimbul de informație între listele corespunzătoare resursei cu indicele 0 din cromozomii celor doi părinți. În exemplu, se alege punctul de mijloc pe ambele liste. Se păstrează neschimbată informația din prima jumătate a primului părinte și se atribuie fiului 1. Apoi, se completează gena respectivă cu indici aleatori de task-uri din a doua jumătate a listei corespunzătoare din părintele 2. De aceea, după punctul de încrucișare, în fiul 1 apare indicele 0 din nou. Pentru că lista nu și-a atins dimensiunea inițială se completează cu indici aleatori din a doua jumătate a listei specifice resursei 0 din primul părinte. Astfel indicele 2 apare în fiul 1. Dacă lista își atingea mărimea inițială prin completări din al doilea părinte, procesul s-ar fi oprit. Am ales ca operația de încrucișare să producă doi noi indivizi. În acest sens, procesul prezentat anterior se întâmplă din nou, dar, de data aceasta, se inversează ordinea părinților. Astfel am obținut gena specifică resursei 0 din fiul 2. Acest proces de încrucișare a informației din cromozomii părinților specific resurselor se repetă pentru toate resursele care iau parte la planificare. Se poate observa că încrucișarea astfel implementată poate genera duplicarea sau dispariția unor indici de task-uri. Algoritmul nostru nu permite astfel de situații, corectarea fiind realizată în procesul de mutație.

Mutația. Este realizată, propriu-zis, în clasa “Cromozom” prin metoda “mutation”. Procesul de mutație are două etape. Într-o primă etapă se realizează înlocuirea aparițiilor suplimentare a indicilor task-urilor în cadrul cromozomului cu indicii care lipsesc. Acest lucru este vital pentru funcționarea algoritmului așa cum ne dorim. Task-urile nu au cum să dispară din planificare și nici cum să apară de mai multe ori. Pentru fiii obținuți în urma încrucișării la pasul anterior sunt identificați indicii task-urilor ce lipsesc din întregul cromozom. În exemplul din Figura 8, în cazul fiului 1 lipsește indicele 5, iar în cazul fiului 2 lipsește indicele 0. Prin parcurgerea tuturor genelor din cromozom, se identifică a doua apariție a oricărui indice și se înlocuiește cu un indice aleator din setul de indici lipsă.

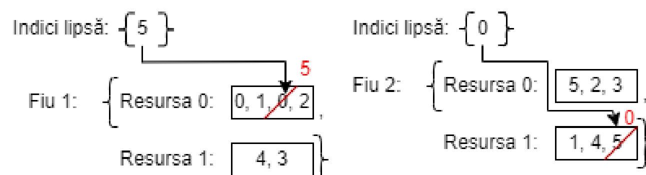


Figura 8. Exemplu de realizare a primei etape din cadrul mutației

Într-o a doua etapă, se realizează un schimb de task între planificările a două resurse diferite. Acest proces face posibilă migrarea unui task dintr-o planificare în alta și sporește diversitatea. Această etapă se realizează cu o probabilitate de 20%. Se aleg aleator două resurse.

Din planificarea primei resurse se alege aleator un task și se elimină. Task-ul ales este introdus în planificarea celei de-a doua resurse alese pe o poziție aleatoare. În Figura 9 este prezentat un exemplu. Se poate observa că task-ul cu indicele 5 din prima resursă aleasă este mutat pe poziția cu indicele 2 în cadrul planificării celei de-a doua resurse alese.

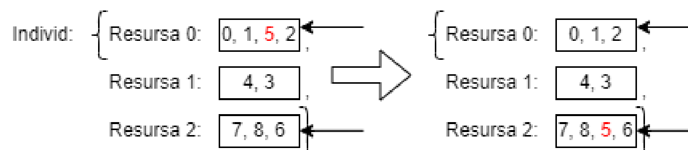


Figura 9. Exemplu de realizare a celei de-a doua etape din cadrul mutației

4. Rezultate experimentale

Am dezvoltat un mediu de simulare cu ajutorul căruia să analizăm performanța algoritmului, acesta putând servi atât pentru analiza unei viitoare configurații de resurse, cât și ca algoritm de planificare propriu-zis. Pentru a păstra compatibilitatea cu algoritmul, am utilizat tot limbajul Python. Acesta facilitează și obținerea timpului de rulare al algoritmului, precum și desenarea graficelor cu datele extrase în timpul rulării.

În programul realizat pentru simulare stabilim numărul de task-uri și numărul de resurse. Apoi, pentru a analiza rezultatele, generăm setul de date de intrare o singură dată și rulăm algoritmul pe acest set de un număr de ori ales. Algoritmul, aproximând ca rezultat soluția optimă, reușește să producă rezultate diferite. Aceste rezultate sunt preluate și comparate după criteriile de performanță, fiind păstrată planificarea cea mai bună. După acest număr de rulări ale algoritmului, se calculează, pentru planificarea cea mai bună, costul de utilizare al resurselor. Pentru că dorim o analiză comparativă care să ne indice un mod de îmbunătățire al planificării și al costului de utilizare, mai introducem în planificare și un număr de resurse suplimentare. Cu noile resurse adăugate la setul de date de intrare inițial, rulăm algoritmul de același număr de ori. În urma rulărilor, se produce o nouă schemă de planificare și un nou cost de utilizare a resurselor. Astfel, prin acest program am reușit să analizăm atât performanțele algoritmului cât și calitatea rezultatului produs.

Metodele de evaluare a rezultatelor obținute și a performanțelor algoritmului utilizate au vizat atât analiza algoritmilor genetici, cât și pe cea a algoritmilor de planificare. Primul pas este stabilirea metricilor de performanță pe baza cărora se face analiza rezultatelor.

Pentru algoritmul genetic am stabilit următorii parametri: (i) numărul de generații: 200; (ii) numărul de indivizi din populație: 100; (iii) probabilitatea de realizare a încrucișării: 80%; (iv) probabilitatea de realizare a mutației: 20%. Am analizat performanțele algoritmului genetic prin evoluția, de-a lungul generațiilor, a valorilor funcției de fitness, a valorilor penalizărilor și a valorilor timpului maxim de utilizare a resurselor.

Din punct de vedere al algoritmului de planificare, am analizat rezultatul obținut de algoritm comparativ cu valoarea ideală a timpului maxim de utilizare. Timpul mediu, obținut prin împărțirea timpului total de procesare a task-urilor la numărul de resurse, este timpul ideal. Ca valoare, acesta nu poate fi atins de către algoritm din cauza modului de formulare a problemei, dar este un termen de comparație pertinent pentru rezultatul obținut de algoritmul nostru. Pentru analiza comparativă a costurilor de utilizare obținute în urma adăugării de resurse la configurația planificării de bază, am realizat două tipuri de grafice ce ilustrează atât diferențele de costuri, cât și evoluția costului, în încercarea de a găsi un număr optim de resurse ce pot fi adăugate în planificare pentru a avea un impact benefic.

Ca în cazul oricărui algoritm, o metrică bună de performanță este timpul de rulare al acestuia, pentru care am realizat Tabelul 1, unde valorile care apar sunt în secunde.

Un prim pas în analiza rezultatelor este realizarea graficelor funcției de fitness și ale penalizărilor de memorie, de capacitate și de depășire deadline.

Tabelul 1. Timpii de rulare ai algoritmului măsurați în secunde

	Număr task-uri						
Număr resurse	100	200	300	400	500	600	1000
10	5.7	10.8	17.0	24.9	33.9	44.8	102.3
13	5.9	10.9	17.2	24.97	34.92	44.83	105.4
16	6.0	11.3	17.9	25.4	34.72	45.38	103.6
25	6.59	11.5	18.4	25.8	35.1	46.5	104.8
40	7.5	12.7	19.2	27.2	36.9	47.5	105.5
50	7.9	13.6	19.7	27.4	37.4	48.2	105.4

Pentru a ilustra modul în care acestea evoluează, prezentăm în Figurile 10 și 11 câteva exemple obținute la diferite rulări ale algoritmului. În graficul cu evoluția valorilor funcției de fitness se observă că, de-a lungul iterațiilor, algoritmul reușește să maximizeze aceste valori, apropiindu-se de maximul funcției (1,5). Totodată, se observă scăderea valorilor penalizărilor cu creșterea valorilor funcției de fitness. Cu precădere, se micșorează valoarea penalizării de memorie și de depășire a deadline-urilor pentru că acestea sunt prioritizate.

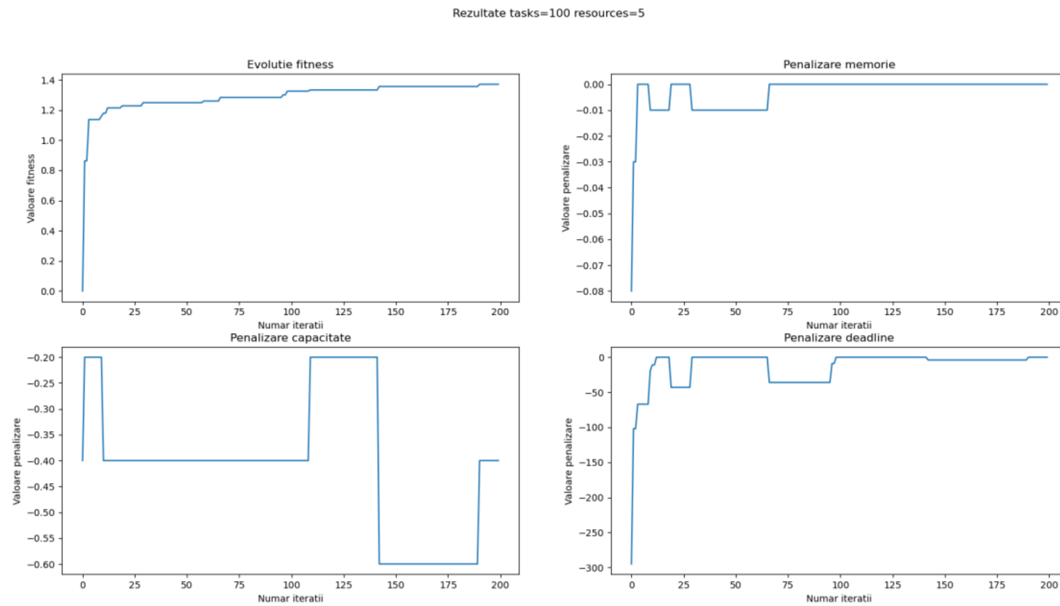


Figura 10. Graficele fitness-ului și penalizărilor pentru 100 de task-uri și 5 resurse

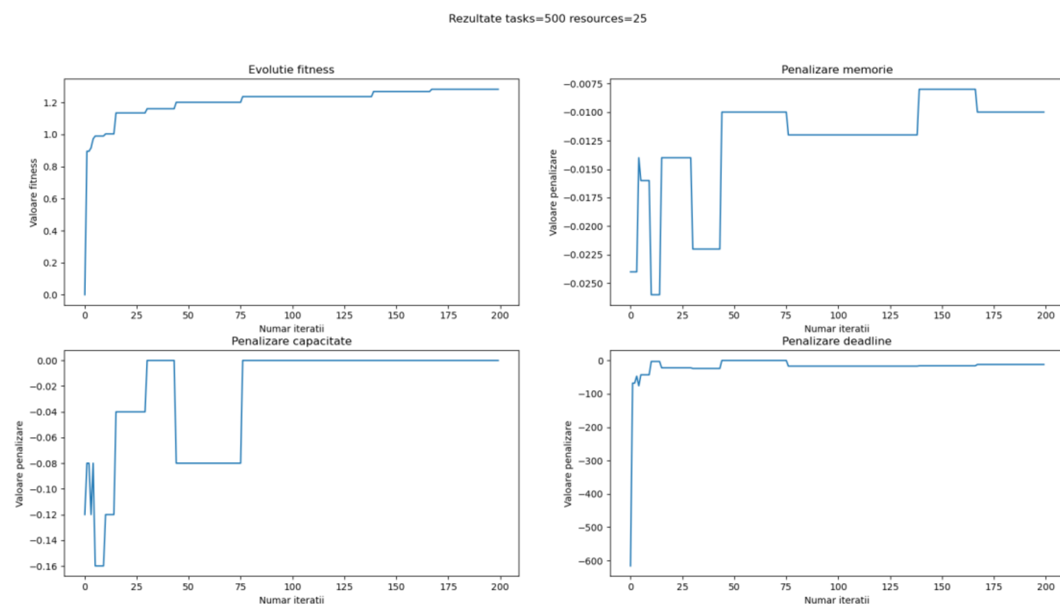
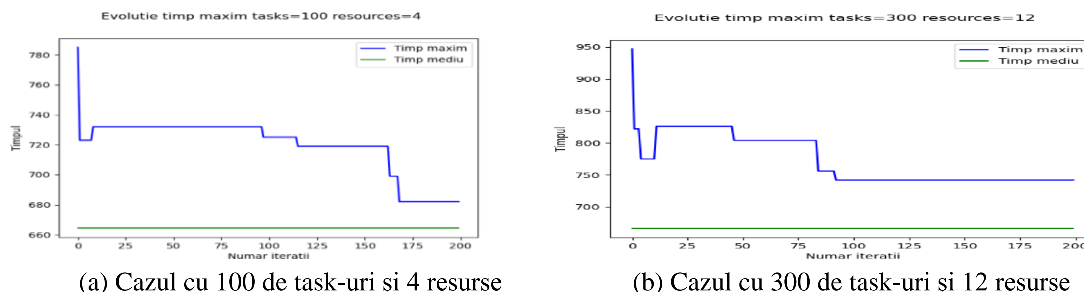


Figura 11. Graficele fitness-ului și penalizărilor pentru 500 de task-uri și 25 resurse

În graficele din Figura 11, se observă aceeași tendință de minimizare a penalizărilor, doar că penalizarea de depășire a deadline-urilor converge mult mai repede către 0.

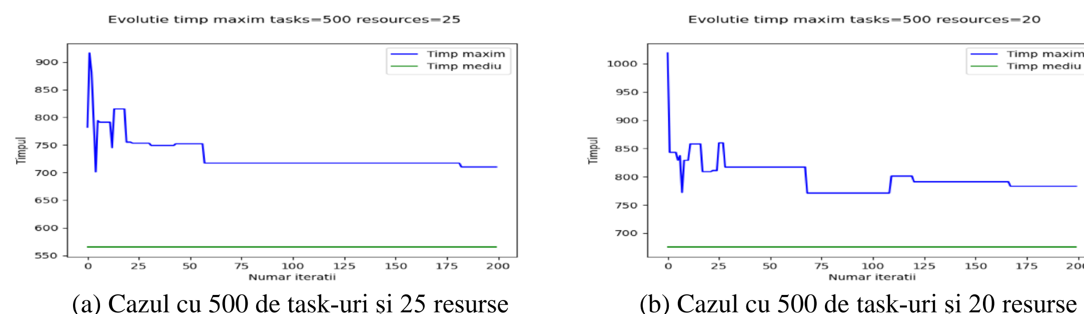
În Figura 12, cele două grafice prezintă evoluția timpului maxim de utilizare a resurselor pentru două cazuri de configurații de task-uri și resurse. Se observă că tendința algoritmului este de a apropia timpul maxim de timpul mediu. Acest timp mediu reprezintă un ideal, foarte puțin probabil de atins, din cauza modului în care este definită problema.



(a) Cazul cu 100 de task-uri și 4 resurse (b) Cazul cu 300 de task-uri și 12 resurse

Figura 12. Evoluția timpului maxim de utilizare a resurselor

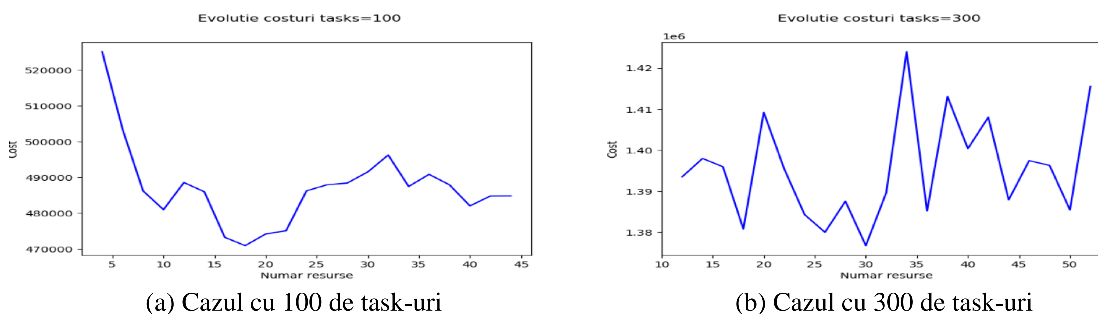
În Figura 13 se poate observa aceeași tendință de minimizare a timpului maxim de utilizare a resurselor pentru cazul în care numărul de task-uri și numărul de resurse este mai mare.



(a) Cazul cu 500 de task-uri și 25 resurse (b) Cazul cu 500 de task-uri și 20 resurse

Figura 13. Evoluția timpului maxim de utilizare a resurselor pentru mai multe task-uri și resurse

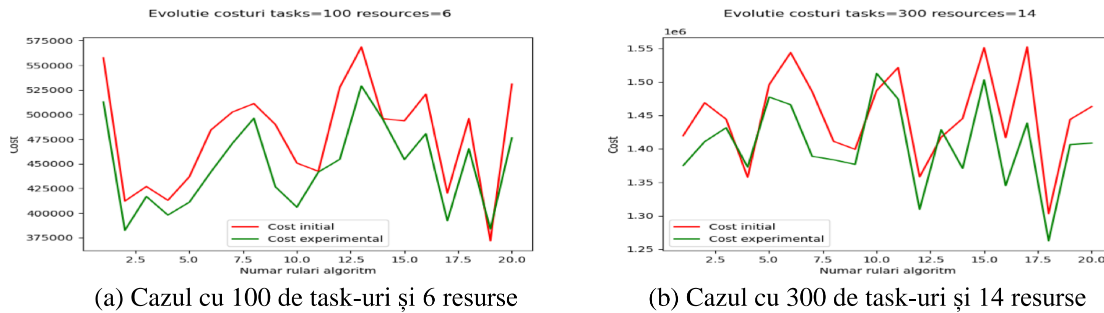
În strânsă legătură cu timpul de utilizare al resurselor este costul total de utilizare. Acest cost este obținut prin înmulțirea costului resursei cu timpul de procesare al task-urilor alocate pe aceasta. Astfel, am realizat un studiu asupra modului în care adăugarea de noi resurse în cadrul unei planificări influențează acest cost de utilizare. În Figura 14 se poate observa că adăugarea de noi resurse în planificare este cea mai benefică atunci când se adaugă un număr mic. În graficul 14b, variațiile mari ale costului se datorează naturii resurselor noi adăugate. Dacă acestea au costuri mari, nu este benefică utilizarea lor în planificare. Astfel, prin această analiză, am observat că, pentru îmbunătățirea costului, trebuie adăugat un număr mic de resurse, iar acestea să aibă costuri medii de utilizare per unitate de timp.



(a) Cazul cu 100 de task-uri (b) Cazul cu 300 de task-uri

Figura 14. Evoluția costului de utilizare odată cu adăugarea treptată de noi resurse

Pentru a verifica observațiile menționate mai sus, am analizat 20 de cazuri de planificări. În Figura 15, costul inițial este obținut în urma planificării standard, iar costul experimental este obținut prin realizarea planificării pe aceleași date de intrare, dar cu adăugarea a două resurse suplimentare, de cost mediu, la cele considerate anterior. Din grafice se poate observa că acest cost experimental este, în majoritatea cazurilor, mai mic decât costul inițial.



(a) Cazul cu 100 de task-uri și 6 resurse (b) Cazul cu 300 de task-uri și 14 resurse

Figura 15. Analiza comparativă a costului inițial și a costului experimental

5. Concluzii și direcții de continuare

Algoritmii clasici de planificare au performanţe bune în sistemele pentru care au fost proiectați. La momentul actual, sistemele de calcul evoluează foarte rapid și la o scară foarte mare. Astfel, astăzi se pune accentul pe sistemele distribuite (Lindsay et al., 2021), sistemele Cloud și supercalculatoare. În aceste sisteme, problema planificării este NP-hard. De aceea, algoritmii de planificare fac adesea apel la metode de aproximare a soluției optime.

Problema abordată conține aspecte ale problemelor de planificare reale actuale. Luând în considerare acest lucru și analizând tendințele de cercetare curente în domeniul planificării, am decis să dezvoltăm un algoritm genetic pentru rezolvarea problemei. Prin analiza algoritmului, am observat că acesta reușește să determine o soluție aproximativă, fără a încălca constrângerile problemei. Când configurația problemei nu permite o soluție apropiată de optimul ideal, algoritmul obține ca rezultat soluția cea mai optimistă.

Soluția propusă și dezvoltată poate servi atât ca algoritm de planificare, cât și ca metodă de analiză. Scopul analizei este determinarea diverselor modificări ce ar trebui aduse în configurația resurselor pentru o rezolvare mai profitabilă.

Direcțiile de cercetare pe care le avem în vedere în continuare în ceea ce privește planificarea pentru procesarea inteligentă, sunt: (i) obținerea unor seturi de date reale și analiza rezultatelor obținute pe acestea; (ii) optimizarea algoritmului pentru a micșora timpul de răspuns în cazul datelor de intrare ce au cu două ordine de mărime mai mult; (iii) adăugarea constrângerilor de dependență între task-uri; (iv) adăugarea constrângerii momentului la care task-urile devin disponibile procesării.

Toate aceste aspecte apropie și mai mult problema abordată de problemele de planificare reale din sistemele de calcul distribuite actuale. Cele două noi constrângeri pe care dorim să le luăm în calcul au un impact important asupra modului în care se realizează planificarea. Pentru a depăși și aceste constrângeri, este nevoie de combinarea algoritmului genetic cu metodele clasice.

Menţiuni

Lucrarea de faţă are la bază parte din activităţile şi rezultatele proiectului PN 1937-0601, derulat la ICI Bucureşti în cadrul Programului Naţional Nucleu 2019-2022, finanţat de Ministerul Cercetării, Inovării şi Digitalizării. De asemenea, rezultatele prezentate au fost obținute cu sprijinul Ministerului Investițiilor și Proiectelor Europene prin Programul Operațional Sectorial Capital Uman 2014-2020, Contract nr. 62461/03.06.2022, cod SMIS 153735, proiect implementat în UPB.

BIBLIOGRAFIE

1. Abdulal, W., Al Jadaan, O., Jabas, A. & Ramachandram, S. (2009). Genetic algorithm for grid scheduling using best rank power. *In IEEE 2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pp. 181-186.

2. Bacalhau, E. T., Casacio, L. & Tavares de Azevedo, A. (2021). *New hybrid genetic algorithms to solve dynamic berth allocation problem*. Expert Systems with Applications 167: 114198.

3. Barredo, P. & Puente, J. (2022). Robust Makespan Optimization via Genetic Algorithms on the Scientific Workflow Scheduling Problem. In *International Work-Conference on the Interplay Between Natural and Artificial Computation*, Springer, Cham, pp. 77-87.

4. Brucker, P. (2006). *Scheduling algorithms*. Springer-Verlag.

5. Constantinescu, R. (2004). Algoritmi de planificare a execuției proceselor. *Revista Română de Informatică şi Automatică*, *14*(1) *(Romanian Journal of Information Technology and Automatic Control)*, 40-47.

6. Dhall, S. K. & Liu, C. L. (1978). On a real-time scheduling problem. *Operations research, 26*(1), 127-140.

7. Hahne, E. L. (1986). *Round robin scheduling for fair flow control in data communication networks*. Massachusetts Inst of Tech Cambridge Lab for Information and Decision Systems.

8. Haupt, R. (1989). A survey of priority rule-based scheduling. *Operations-Research-Spektrum, 11*(1), 3-16.

9. Holland, J. H. (1992). Genetic algorithms. *Scientific american, 267*(1), 66-73.

10. Hotovy, S. (1996). Workload evolution on the Cornell theory center IBM SP2. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 27-40

11. Krishna, C. M. & Shin, K. G. (1997). *Real-time systems*. McGraw Hill Higher Education.

12. Laplante, P. A. (2004). *Real-time systems design and analysis*. New York: Wiley.

13. Lindsay, D., Gill, S. S., Smirnova, D. et al. (2021). The evolution of distributed computing systems: from fundamental to new frontiers. *Computing 103*, 1859-1878.

14. Naithani, P. (2018). Genetic Algorithm Based Scheduling To Reduce Energy Consumption In Cloud. *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pp. 616-620.

15. Nash Jr, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the national academy of sciences, 36*(1), 48-49.

16. Omar, H. K., Jihad, K. H. & Hussein, S. F. (2021). Comparative analysis of the essential CPU scheduling algorithms. *Bulletin of Electrical Engineering and Informatics 10*(5): 2742-2750.

17. Omara, F. A. & Arafa, M. M. (2009). Genetic algorithms for task scheduling problem. In *Foundations of Computational Intelligence, Vol. 3*, pp. 479-507. Springer, Berlin, Heidelberg.

18. Schrage, L. (1968). A proof of the optimality of the shortest remaining processing time discipline. *Operations Research, 16*(3), 687-690.

19. Șerban, C. & Carp, D. (2021). Using Genetic Algorithms to Solve Discounted Generalized Transportation Problem. *Studies in Informatics and Control, 30*(3), ISSN 1220-1766*,* 29-38.

20. Wang, Y.-H., Li, H. & Liu, H.-W. (2008). Multi-Agent and Hybrid Genetic Algorithm Approach for Distributed Jobshop Scheduling. In *IEEE 2008 International Conference on Apperceiving Computing and Intelligence Analysis,* pp. 404-407.

Damian Cristian SILIȘTEANU este student la masterul de Sisteme Software Avansate în cadrul Universității Politehnica din București. Principalele sale domenii de interes în activitatea de cercetare sunt: sisteme distribuite, metode de optimizare Bio-inspired, sisteme embedded, protocoale de comunicație.

Damian Cristian SILIȘTEANU is a student pursuing a Master degree in Advanced Software Systems at the University Politehnica of Bucharest. His main topics of interest in the research activity are: distributed systems, Bio-inspired optimization methods, embedded systems, communication protocols.

Bogdan Costel MOCANU este şef de lucrări și post-doctorand la Departamentul de Calculatoare și Tehnologia Informației, Universitatea Politehnica din Bucureşti. Domeniile sale principale de interes sunt: sisteme distribuite, sisteme Peer-to-Peer, cloud computing, gestionare Big Data, planificare în timp real, securitatea datelor.

Bogdan Costel MOCANU is Assistant Professor and Postdoc at the Department of Computer and Information Technology, University Politehnica of Bucharest. His main topics of interest are: distributed systems, Peer-to-Peer systems, cloud computing, Big Data management, real-time scheduling, data security.

Mihnea Horia VREJOIU este cercetător ştiinţific gr. III la ICI Bucureşti. Domeniile sale principale de expertiză şi interes cuprind: vederea artificială (prelucrare şi analiză de imagini, OCR, recunoaştere numere de înmatriculare), învăţare automată (clasificatoare, memorii asociative) și optimizare.

Mihnea Horia VREJOIU is a Senior Scientific Researcher at ICI Bucharest. His main topics of expertise and interests include: Artificial Vision (Image Processing and Analysis, Pattern Recognition, OCR, License Plate Recognition), Machine Learning (classifiers, associative memories) and optimization.

Florin POP este profesor la Departamentul de Calculatoare şi Tehnologia Informației, Universitatea Politehnica din București. De asemenea, este cercetător științific gr. I la ICI București. Principalele sale domenii de interes sunt: sisteme distribuite, grid şi cloud computing, sisteme peer-to-peer, gestionare Big Data, agregare date, tehnici de regăsire și clasificare a informațiilor, metode de optimizare Bio-inspired.

Florin POP is Professor at the Department of Computer and Information Technology of University Politehnica of Bucharest. He is also a 1st degree Scientific Researcher at ICI Bucharest. His main domains of interest are: distributed systems, grid and cloud computing, peer-to-peer systems, Big Data management, data aggregation, information retrieval and classification techniques, Bio-inspired optimization methods.