UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ROMÂNĂ

LUCRARE DE LICENŢĂ

Rezolvarea unei probleme de tip planificare constrânsă de resurse multi-calificate utilizând algoritmi genetici

Conducător ştiinţific
Conf. Dr. Suciu Mihai

Absolvent
Dreţcanu Mihai

2023

ABSTRACT

A project is a complex action undertaken by an individual or group of individuals in order to reach a set of proposed goals. In the context of a project, some activities are defined such that their completion contributes towards accomplishing the targeted goals. The completion of an activity can be constrained by some factors, for example time, resources or knowledge, depending on the project. As such, the more factors that are taken in account, the more complex a project grows. This makes it harder for said project to be managed, hindering the chance to accomplish its objectives.

For that reason, it's important to have a good, well-thought plan to follow, or at least to consult, when trying to complete a project. The plan represents a way of defining how the project should be done from our perspective, from the usage and distribution of resources to the time the activities should start. As projects represent an important aspect of modern day, over the years people have formulated numerous problems related to project scheduling, a category of such problems being the Multi-Skilled Resource-Constrained Project Scheduling Problem, abbreviated MSRCPSP. The goal of this thesis is to study one such problem and propose a solution using genetic algorithms.

In the introduction we'll discuss the history and context of the project scheduling problem and its evolution towards the apparition of MSRCPSP. We'll give a formal definition of the problem that we're going to solve and we'll study the state of the art of the MSRCPSP category in general. In the second chapter we'll give a theoretical approach to the problem in the form of genetic algorithms. We'll explain what a genetic algorithm is and we'll rephrase the problem as a combinatorial search and optimisation problem, paving the way for a genetic solution. We'll use a standard solution representation for which valid operators are already known. We'll also propose a new solution representation for which we'll discuss the theoretical benefits. We'll discuss some links between the two representations while also proving the validity of both representations and their proposed operators.

In the third chapter we present a generic genetic algorithm implementation in Python and a general methodology of using said implementation to develop genetic solutions. We use this as a starting point for implementing the algorithms that solve the problem proposed before. In the fourth chapter we discuss experimental results, showing that there are cases where our implementations succeed in finding good solutions, and cases where they fail. We also find that there are cases when one algorithm outperforms the other. The fifth chapter presents an API which integrates the developed algorithms. In the last chapter we discuss overall conclusions and possible extensions to the work done in this thesis.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Cuprins

Listă de figuri

Listă de tabele

Capitolul 1

Despre problema abordată

Gestionarea proiectelor este un proces complex ce presupune luarea unor decizii în cadrul derulării unui proiect astfel încât șansa de a atinge anumite obiective propuse să fie maximizată. Printre aceste obiective se pot număra minimizarea duratatei sau a costurilor proiectului, maximizarea calității sau utilizarea în mod echilibrat a resureslor umane. Proiectul progresează prin realizarea unor activități a căror execuție poate fi condiționată de anumite aspecte precum context, resursele alocate, calificările angajaților sau completarea în prealabil a altor activități din cadrul proiectului. Cum îndeplinirea acestor activități este un factor ce influențează în mod direct atingerea obiectivelor, planul de execuție al acestora reprezintă o decizie improtantă în gestionarea proiectului.

Procesul de realizare a planului începe prin identificarea activităților ce se vor desfășura în cadrul proiectului. Se stabilesc diverse aspecte relative la activități, precum durata acestora, calificările necesare și dependențele față de alte activități, urmată de stabilirea resurselor și oamenilor disponibili proiectului. Odată cunoscute detaliile despre activități, o persoană din cadrul proiectului, adesea managerul de proiect, este responsabilă cu a alege o planificare a activităților și resureslor astfel încât obiectivele propuse să fie atinse. Atingerea tuturor obiectivelor în cadrul unui proiect este ceva idealist, ce nu poate fi mereu realizată. Din cauza aceasta, se prioritizează anumite obiective și se încearcă găsirea unui plan ce este cel mai aproape de îndeplinirea acestora. Astfel, cu cât proiectele sunt mai complexe, cu atât găsirea unei planificări adecvate este mai grea.

1.1 Istoria problemelor de planificare

Pe parcursul timpului s-a încercat enunțarea formală a acestor probleme de planificare a proiectelor, creând o întreagă categorie denumită Project Scheduling Problem. Scopul acestor probleme este găsirea unor planificări adecvate, conceptul de

planificare variând în funcție de problemă și complexitatea acesteia. Cel mai simplu exemplu pentru o problemă de planificare îl reprezintă cazul în care realizarea activităților este constrânsă doar de dependența de alte activități și durata de execuție. Această problemă a fost enunțată și rezolvată în anii 50 în două articole diferite, prin dezvoltarea unor metode numite Critical Path Method de către James E. Kelley și Morgan R. Walker [KW59] și Program Evaluation And Review Technique de către Malcolm et al. [MRCF59]. Ulterior, în anii 60 a apărut interesul pentru planificarea de proiecte constrânsă de resurse [Wie64], ce este o extensie a problemei menționate anterior, adăugând conceptul de alocare de resurse și disponibilitatea acestora. Această problemă este NP-hard [BLK83], astfel că pentru găsirea unor soluții se practică aplicarea de euristici. În general, majoritatea problemelor definite în continuare sunt extensii directe ale problemei de planificare constrânsă de resurse. O analiză mai detaliată a extensiilor existente a fost realizată de Hartmann și Briskorn în lucrarea [HB10].

Către finalul anilor 1990 apare conceptul de mod de realizare al unei activități, existând mai multe modalități de executare în funcție de resursele alocate. În 1998, Hartmann et al. realizează un sumar al problemei și o analiză a metodelor existente [HD98]. La începutul anilor 2000 a apărut o nouă extensie, și anume planificarea de proiecte constrânsă de resurse multi-calificate, cunoscută în literatură ca Multi-Skilled Resource-Constrained Project Scheduling Problem, abreviată în continuare ca MSRCPSP. Se consideră că prima enunțare a unei probleme de acest tip ar fi fost dată în lucrarea lui Hegazy et al. [HSEC00], în care se presupune că fiecare resursă are o singură calificare bine definită, dar ce poate fi folosită pentru a substitui o altă calificare. Această enunțare poate fi privită similar cu problema de planificare cu multiple moduri de execuție. Ulterior apar noi enunțări ale problemei, introducându-se și conceptul de resursă multi-calificată propriu-zisă ce înlocuiește conceptul de subsituție. Cum această problemă este o extensie a problemei de planificare constrânsă de resurse, aceasta este la rândul ei o problemă NP-hard, chiar mai complexă.

1.2 Formularea problemei abordate

Vom enunța problema pe care se va focusa lucrarea. Problema începe prin definirea unui set de activități $Activities = \{A_1, A_2, ..., A_n\}$, $n = |Activities|$, ce trebuie realizate. Între acestea există dependențe, anumite activități necesitând completarea altora înainte de a putea fi începute. Spre exemplu, pentru a putea începe A_{k_3}, se poate impune realizarea activităților A_{k_1} și A_{k_2}. Astfel, problema poate fi reprezentată sub forma unui graf orientat unde nodurile sunt elementele mulțimii $Activities$, iar arcele reprezintă dependențele dintre activități. Vom nota mulțimea acestor arce prin

$Dependencies$. Relația $(A_{k_1}, A_{k_3}) \in Dependencies$ semnifică faptul că pentru a putea începe activitatea A_{k_3}, este necesar ca activitatea A_{k_1} să se fi încheiat deja. Se poate observa că dacă graful prezintă cicluri, problema nu poate fi rezolvată, niciuna din activitățile ciclului neputând fi începută. Din această cauză, în continuare se va considera că graful ce corespunde problemei este aciclic, deci problema permite soluții din perspectiva planificării activităților.

Pentru realizarea unei activități A_k este necesară o anumită durată de timp ce o vom nota prin $Duration(A_k)$, unde $Duration : Activities \mapsto \mathbb{R}+ \cup \{0\}$, durată prestabilită și ce nu se modifică pe durata proiectului. Unitatea de timp aleasă este arbitrară, putând fi reprezentată spre exemplu de oră, săptămână, lună etc. Se consideră că, o dată începută, o activitate nu poate fi întreruptă și este dusă la bun sfârșit. Astfel, se pot modela relații despre momentele de începere și de finalizare a activităților. Dacă notăm prin $StartTime(A_k)$ și $FinishTime(A_k)$ momentul de start, respectiv de terminare al activității A_k relativ la începerea proiectului într-o anumită planificare, unde $StartTime$, $FinishTime : Activities \mapsto \mathbb{R}+ \cup \{0\}$, apare relația $FinishTime(A_k) = StartTime(A_k) + Duration(A_k)$. Se poate observa că, datorită acestei relații, este necesară doar cunoașterea uneia dintre funcțiile $StartTime$ și $FinishTime$. Totodată, se observă că momentul de start al unei activități trebui să fie mai mare ca momentul de terminare al oricărei activități de care depinde:

$$\forall A_{k_1}, A_{k_2} \in Activities$$

$$(A_{k_1}, A_{k_2}) \in Dependencies \Rightarrow FinishTime(A_{k_1}) \leq StartTime(A_{k_2})$$

O planificare a activităților poate fi reprezentată de o funcție $StartTime$ ce respectă constrângerea menționată anterior. Formularea realizată pe moment reprezintă una din enunțările ce descriu problema clasică de planificare de proiect.

Vom defini un set de calificări, denumite și skill-uri, $Skills = \{S_1, S_2, ..., S_m\}$, $m = |Skills|$. Pentru realizarea unei activități este necesar un anumit număr natural de unități din fiecare calificare. Vom nota prin $SkillRequirements : Activities \times Skills \mapsto \mathbb{N}$ necesitatea de resurse, valoarea lui $SkillRequirements(A_k, S_i)$ semnificând numărul de unități din calificarea S_i necesitată de activitatea A_k. Daca valoarea este 0, calificarea respectivă nu este necesară execuției activității.

Vom considera un set de resurse disponibile $Resources = \{R_1, R_2, ..., R_l\}$, $l = |Resources|$. Aceste resurse pot avea una sau mai multe din calificările definite anterior. Vom introduce funcția $SkillMasteries : Resources \times Skills \mapsto \{0, 1\}$, astfel încât $SkillMasteries(R_j, S_i) = 1$ dacă resursa R_j stăpânește calificarea S_i și 0 în caz contrar. Prin asignarea unei resurse la o anumită activitate se poate progresa spre îndeplinirea unei necesități dintr-o anumită calificare. Pentru a reprezenta alocarea de resurse în cadrul unei activități, vom defini funcția $ResourcesAssigned$:

$Activities \times Skills \mapsto P(Resources)$, unde $P(Resources)$ reprezintă mulțimea părților sau mulțimea submulțimilor lui $Resources$. $ResourcesAssigned(A_k, S_i)$ semnifică resursele alocate activității A_k pentru îndeplinirea calificării S_i. Vom impune condiția

$$R_j \in ResourcesAssigned(A_k, S_i) \Rightarrow SkillMasteries(R_j, S_i) = 1$$

pentru a asigura faptul că pentru o calificare sunt alocate doar resurse ce o stăpânesc. Numărul căutat de unități dintr-o anumită calificare poate fi modelat prin relația $|ResourcesAssigned(A_k, S_i)| = SkillRequirements(A_k, S_i)$.

Vom enunța constrângeri legate de asignarea resurselor. La un moment dat, o resursă nu poate fi asignată decât unei singure activități, pentru o singură calificare. O dată asignată unei activități, resursa menține atât activitatea la care a fost asignată cât și calificarea pentru care a fost asignată până la finalizarea activității.

Introducem funcția $AllResourcesAssigned : Activities \mapsto P(Resources)$, pentru a reprezenta toate resursele ce sunt alocate activității A_k. Valoarea funcției poate fi calculată prin relația

$$AllResourcesAssigned(A_k) = \bigcup_{i=1}^{m} ResourcesAssigned(A_k, S_i)$$

În continaure putem modela constrângerile enunțate anterior utilizând notațiile curente. Prin condiția

$$\forall A_{k_1}, A_{k_2} \in Activities, A_{k_1} \neq A_{k_2},$$

$$[StartTime(A_{k_1}), FinishTime(A_{k_1})) \cap [StartTime(A_{k_2}), FinishTime(A_{k_2})) \neq \emptyset$$

$$\Rightarrow AllResourcesAssigned(A_{k_1}) \cap AllResourcesAssigned(A_{k_2}) = \emptyset$$

se impune ca, dacă două activități diferite au timpi comuni în care sunt executate, să nu existe resurse comune. Cum în cadrul unei activități A_k fiecare resursă este alocată unei singure calificări, mulțimile $ResourcesAssigned(A_k, S_i)$ obținute prin varierea lui S_i vor fi disjuncte. Asta înseamnă că numărul total de resurse asignate pentru întreaga activitate va fi egal cu suma numărului de resurse alocat fiecărei calificări în parte în contextul activității. Acest lucru poate fi exprimat prin relația:

$$|AllResourcesAssigned(A_k)| = \sum_{i=1}^{m} |ResourcesAssigned(A_k, S_i)|$$

Faptul că resursa rămâne alocată aceleași activități și pe aceași calificare pe toată durata derulării activității este deja asigurată de modul în care a fost definită funcția de alocare de resurse și condițiile modelate anterior.

O planificare de resurse este reprezentată de o funcție $ResourcesAssigned$ ce respectă constrângerile menționate anterior. Este de notat că există posibilitatea ca pentru o anumită planificare în timp a activităților să nu existe nicio alocare validă de resurse și vice-versa. În continuare vom presupune că există cel puțin o alocare validă de resurse relativ la cel puțin o planificare de activități.

Scopul problemei este găsirea unei planificări, mai exact a funcțiilor $StartTime$ și $ResourcesAssigned$, astfel încât timpul de execuție al proiectului să fie minim. Vom nota acest timp prin $ProjectDuration$, modul de calcul fiind dat de

$$ProjectDuration = \max_{1 \leq k \leq n} (FinishTime(A_k))$$

Vom relua pe scurt enunțarea problemei. Se dau un set de activități $Activities$, un set de calificări $Skills$ și un set de resurse $Resources$ a căror calificări sunt redate prin $SkillMasteries$. Cunoscându-se dependențele între activități $Dependencies$, duratele de execuție prin $Duration$ și necesitatea fiecărei calificări $SkillRequirements$, să se găsească funcțiile $StartTime$ și $ResourcesAssigned$ ce respectă constrângerile de dependență, de necesitate de calificări și de utilizarea de resurse, astfel încat $ProjectDuration$ să fie minim.

1.3 Statutul curent al problemei și motivație

Așa cum a fost menționat în introducere, MSRCPSP reprezintă o categorie largă de probleme. Formularea oferită anterior reprezintă o enunțare simplă și de bază ce se regăsește în majoritatea problemelor din această categorie. Adăugându-se și modificându-se anumite constrângeri se poate ajunge la noi formulări ce au fost rezolvate pe parcursul timpului. Astfel, în literatură apar numeroase lucrări ce menționează rezolvarea MSRCPSP, rezolvând de fapt una dintre numeroasele enunțări ale acestei probleme.

În lucrarea lui Hegazy et al. [HSEC00] fiecare resursă stăpânește exact o calificare, cele două concepte fiind interschimbabile. Se utilizează ideea de reguli de substituție, acestea reprezentând capacitatea unei calificări de a fi înlocuită prin alte calificări. Spre exemplu, o unitate de calificare S_{i_1} poate fi substituită cu trei unități de calificare S_{i_2}. Aceste reguli sunt stabilite înainte de începerea proiectului, astfel acestea fac parte din datele de intrare. Autorii au abordat această problemă utilizând euristici.

Yannibelli și Amandi[YA11] adaugă conceptul de context de lucru și eficacitate a unei resurse. Fiecare resursă prezintă o anumită eficacitate într-un anumit context dat de atribute precum activitatea executată, calificarea utilizată în cadrul activității și persoanele asignate celorlalte calificări necesare. Aceste valori sunt folosite mai

departe pentru a calcula eficacitatea planificării alese. Astfel, pe lângă datele de intrare necesare problemei de bază, apare nevoia de a ști eficacitatea fiecărei resurse în fiecare context posibil. Totodată, problema introduce ca obiectiv maximizarea eficacității unei planificări. Soluția oferită utilizează algoritmi genetici.

Alt concept important îl reprezintă nivelul de stăpânire al unei calificări, fiind un mod de a cuantifica îndemânarea unei resurse în domeniu. Spre exemplu, în cadrul unei companii pot exista mai multe persoane cu calificarea de software developer ce sunt diferențiati ca experiență prin titluri de tipul "Junior" și "Senior". O persoană ce este apreciată ca "Senior" ar putea ajuta în realizarea mai rapidă a unei activități ce-i necesită expertiza, accelerând progresul proiectului. Printre lucrările ce abordează această idee este cea a lui Chen și Zhang [CZ13], ce utilizează totodată conceptul de cost al unei resurse și penalitate. Obiectivul acestora a fost minimizarea costurilor unui proiect, iar soluția propusă este una de tip Ant Colony Optimization.

O altă problemă cu ierarhii de stăpânire a unei calificări a fost rezolvată de Jakob Snauwaert și Mario Vanhoucke [SV21] utilizând algoritmi genetici și căutare locală pentru a minimiza durata proiectului. Spre deosebire de Chen și Zhang, stăpânirea unei calificări cu un grad mai mare nu accelerează durata de execuție, fiind în schimb o constrângere pentru executarea activității. Tot prin algoritmi genetici au abordat Enrique Alba și J. Francisco Chicano [AF07] o problemă de categorie MSRCPSP ce introduce conceptul de dedicație a unui angajat. Aceasta depinde de factori precum salariul și durata din timpul total de lucru alocate proiectului.

Lucrarea lui Hamidreza Maghsoudlou et al.[MANN16] adaugă conceptul de moduri de execuție, apărut anterior la finalul anilor 1990. Astfel, apare posibilitatea de a executa o activitate prin apelul la diferite combinații de calificări. În această lucrarea s-a lucrat multi-obiectiv, scopul fiind cel de a minimiza timpul de execuție al proiectului, de a minimiza costul de execuție și de a maximiza calitatea activităților. Rezolvarea acesteia a fost realizată printr-un algoritm de tip Invasive Weed Optimisation.

O enunțare ce renunță la constrângerea ca o resursă să fie prezentă pe toată durata execuției unei activități este dată în lucrarea lui Drezet și Billaut [DB08]. În aceasta, se permite ca sub anumite condiții o resursă să fie schimbată în timpul derulării activității. Rezolvarea propusă se face printr-o combinație de euristici cu Tabu-Search.

Pe lângă a rezolva problema, s-a încercat dezvoltarea unor modalități de a defini limite inferioare pentru problemele din această categorie ce au ca obiectiv minimizarea duratei proiectului. O astfel de metodă este propusă de către Néron [Nér02] pentru una din problemele de bază, ulterior fiind oferită o nouă metodă pentru problemele extinse cu ierarhie de stăpânire pentru calificări în lucrarea lui Bellenguez și Néron [BN04]. Prin aceste limite se oferă un punct de reper relativ la posibilitatea

de îmbunătățire a soluțiilor găsite. Ulterior, aceiași autori au propus o soluție de tip Branch-and-Bound pentru problemă [BN07].

Aceste exemple au ca scop ilustrarea varietății problemelor de tip MSRCPSP și a metodelor prin care acestea au fost rezolvate. Problema de bază, deși ar fi cea mai simplă, este tratată în literatură prin intermediul problemelor ce o extind, dintre care cele menționate anterior. Analiza acestei probleme de bază poate oferi informații despre comportamentul extensiilor, ajutând în găsirea unor noi soluții pentru acestea. Totodată, această problemă este aplicabilă unui context real, unde se dorește găsirea unor planificări considerând doar datele de intrare specificate în această formulare, neavându-se mai multe informații precum eficacitatea unei resurse într-un context de lucru sau costurile exacte. Astfel, abordarea acestei enunțări particulare este motivată de utilitatea problemei în sine, cât și de absența unei literaturi ample despre aceasta.

Capitolul 2

Propunerea unei soluții

2.1 Formularea ca problemă de căutare și optimizare

Reprezentarea curentă a unei planificări prezintă două dezavantaje. Vom considera o activitate A_{k_0} a cărei moment de terminare corespunde momentului de terminare a proiectului și de care nicio altă activitate nu depinde, altfel spus $EndTime(A_{k_0}) = ProjectDuration$ și $\forall A_k \in Activities, (A_{k_0}, A_k) \notin Dependencies$. Dacă o planificare $StartTime$ este validă, putem construi o nouă planificare validă $StartTime_0$ ce îndeplinește condițiile $\forall A_k \in Activities \setminus \{A_{k_0}\}$, $StartTime_0(A_k) = StartTime(A_k)$ și $StartTime_0(A_{k_0}) = EndTime(A_{k_0}) + x, x \in \mathbb{R}+$. Faptul că momentul de start al activității A_{k_0} este după momentul de terminare al proiectului în planificărea inițială ne garantează că restul activităților sunt încheiate și că resursele necesare sunt disponibile, afirmând valididatea planificării. Prin acest proces putem genera o infinitate de soluții, deci spațiul soluțiilor este unul infinit. Totodată, este evident faptul că $StartTime$ este o planificare mai bună ca $StartTime_0$, terminând proiectul mai din timp. Astfel, spațiul curent prezintă o infinitate de soluții ce sunt evident mai slabe ca altele. Din această cauză, pentru rezolvarea acestor probleme se practică restrângerea spațiului planificărilor la unul finit, soluțiile mai slabe fiind candidați buni pentru eliminare.

Vom denumi spațiul menționat anterior $ActivitiesPlanifications$ și spațiul tuturor alocărilor de resurse valide $ResourcesPlanifications$. Cea din urmă este setul tuturor funcțiilor de forma

$$ResourcesAssigned : Activities \times Skills \mapsto P(Resources)$$

ce respectă constrângerile problemei. Cum $Activities$, $Skills$ și $Resources$ reprezintă mulțimi finite, și $Activities \times Skills$ și $P(Resources)$ vor fi mulțimi finite. Deoarece funcțiile $ResourcesAssigned$ au atât domeniul cât și codomeniul mulțimi finite, putem concluziona că există un număr finit de moduri de a le defini, deci mulțimea

$ResourcesPlanifications$ este finită.

O soluție la problemă va fi un elemente al mulțimii $ActivitiesPlanifications \times ResourcesPlanifications$. $ActivitiesPlanifications$ și $ResourcesPlanifications$ sunt mulțimi finite, $ActivitiesPlanifications$ prin modul în care am definit-o anterior iar $ResourcesPlanifications$ datorită demonstrației precedente. Concluzionăm atunci că spațiul definit este unul finit. Problema poate fi exprimată combinatorial și interpretată ca o problemă de căutare și optimizare, spațiul de căutare fiind cel menționat anterior, iar funcția de optimizat fiind durata proiectului, ce dorim să o minimizăm.

Odată exprimate în această maniera, problemele din această categorie sunt rezolvate utilizând algoritmi specifici problemelor de căutare și optimizare, precum modalitățile menționate în capitolul anterior 1.3. În continuare vom studia o soluție ce utilizează algoritmi genetici.

2.2 Algoritmi genetici

Algoritmii genetici reprezintă metode metaeuristice de rezolvare a problemelor de căutare și optimizare, enunțați pentru prima dată în lucrarea lui Holland [Hol92]. Aceștia sunt inspirați din natură, mai exact din modelul evolutiv al lui Darwin, ce descrie modul în care o specie se dezvoltă. Indivizii populației sunt caracterizați prin trăsături ce sunt codificate prin materialul lor genetic. Acestea le oferă anumite avantaje și dezavantaje în cadrul supraviețuirii în mediul în care trăiesc. Indivizii mai slabi decedează, iar cei mai puternici supraviețuiesc și se reproduc. Prin reproducere descendenții moștentesc o parte din materialul genetic al părinților, și astfel din trăsăturile lor. Datorită acestui fapt, dacă doi părinți sunt bine adaptați mediului, și copilul poate fi bine adaptat, existând chiar șanse de îmbunătățire. Înainte de a se naște urmașii, materialul genetic este încă instabil și se pot întâmpla mutații ce modifică neașteptat trăsăturile individului. Prin repetarea acestui proces, generațiile avansează și se adaptează în timp la mediu.

Aplicând procedeul definit anterior pentru o problemă dată, se obține un algoritm genetic. Se consideră mediul ca fiind problema aleasă, iar populația ca fiind potențiale soluții. Se generează o populație inițială ce este evaluată într-un mod calitativ raportat la cât de bine rezolvă problema propusă, valoarea calculată fiind cunoscută sub conceptul de fitness. Se avansează în mod repetat generațiile până ce un anumit criteriu de oprire este îndeplinit. Pentru a avansa generațiile, se selectează anumiți indivizi ai populației curente ce sunt utilizați pentru a genera noi soluții. Generarea de soluții noi se face prin intermediul operatorului de încrucișare, cunoscut și drept "crossover", descendenții obținuți fiind ulterior supuși la un operator de mutație ce le poate modifica reprezentarea. Repetând acest proces de generare de soluții de un număr dat de ori, formăm o subpopulație de descendeți în cadrul

generației curente. Noii candidați sunt apoi evaluați și integrați în cadrul populației utilizând o strategie de înlocuire a indivizilor existenți. Procesul de înlocuire poate lipsi, în acest caz considerându-se că toți descendenții curenți, împreună cu soluțiile deja existente, vor face parte din generația următoare. Pentru flexibilitate, se va considera algoritmul cu strategie de înlocuire. Această procedură de algoritm genetic poate fi vizualizată în figura 2.1.

```
procedure GeneticAlgorithm()
        population <- generateInitialPopulation()
        evaluate(population)
        while stop criterion not met
                advanceGeneration()
        endwhile
endprocedure

procedure advanceGeneration()
        generationOffsprings <- []
        for i <- 1, numberOfReproductions
                parents <- select(population)
                offsprings <- cross(parents)
                offsprings <- mutate(offsprings)
                generationOffsprings.appendAll(offsprings)
        endfor
        evaluate(generationOffsprings)
        population <-replace(population,generationOffsprings)
endprocedure
```

Figura 2.1: Pseudocod pentru un algoritm genetic generic

Problemele de căutare și optimizare pot fi astfel ușor translatate într-un mod formal care să le permită rezolvarea utilizând astfel de algoritmi. Mulțimea tuturor soluțiilor posibile reprezintă spațiul de căutare, în timp ce valoarea acordată de funcția de evaluare reprezintă obiectivul ce se dorește a fi optimizat. Dacă problema este una de maximizare, un fitness mai bun este unul mai mare. Dacă problema este una de minimizare, un fitness mai bun este unul mai mic. Vom caracteriza acum elementele utilizate în pseudocod.

În funcție de problemă, ce se înțelege prin soluție poate varia, putând prespune concepte abstracte. Din această cauză, se utilizează reprezentări ale soluțiilor, numite și codificări. Deobicei acestea se aleg astfel încât să fie mai ușor să se aplice operatorii genetici. Printre cele mai utilizate reprezentări sunt cea de permutare și cea de vectori de valori, unde o soluție este un vector de dimensiune fixă n. Pentru aceste codificări există numeroase aplicații de mutație și încrucișare bine cunoscute în literatură.

Populația reprezintă o mulțime de soluții pe care se lucrează. Aceasta se modifică prin utilizarea metodelor menționate, convergând pe parcurs către anumite optime. Cu cât populația este de dimensiune mai mare, cu atât șansele de a favoriza diversitatea cresc, dar și viteza de găsire a noi soluții optime este mai lentă. Din cauza acestor motive, se preferă ca mărimea populației să fie echilibrată astfel încât nici găsirea optimelor să nu fie înceată, și nici diversitatea să nu fie impactată negativ. În contextul algoritmilor genetici, diversitatea se traduce prin acoperirea unui

spațiu cât mai larg al spațiului reprezentărilor.

Metoda de generare a populației are rolul de a crea soluții inițiale fezabile pentru problemă. Deobicei se preferă ca această metode să genereze soluțiile aleator și uniform, astfel încât să se acopere cât mai multe zone ale spațiului de căutare și a asigura astfel diversitate în cadrul populației. Se mai practică și introducerea unor soluții predefinite în populația inițială, ce sunt stabilite printr-o presupunere informată, observându-se că în anumite probleme ajută la găsirea unor soluții mai bune.

Metoda de evaluarea are ca scop măsurarea calității unei soluții în raport cu problema studiată. Cum se lucrează cu o reprezentare a soluției, aceastea trebuie decodificată. Este important ca evaluarea să fie eficientă ca timp, fiind aplicată pentru fiecare soluție generată. În general, se observă că dificultatea utilizării algoritmilor genetici este dată de durata de execuție a funcției de evaluare.

Criteriul de oprire reprezină o condiție de stop a algoritmului. Condițiile de stop sunt în general atingerea unui anumit număr de generații, trecerea unei anumite durate de timp de la începerea execuției algoritmului sau atingerea unui anumit fitness țintă.

Numărul $numberOfReproductions$ reprezintă numărul de reproduceri ce vor fi realizate în cadrul unei generații. Acesta, împreună cu numărul de urmași generați în cadrul unei încrucișări, determină numărul total de de descendenți generați în cadrul unei generații prin formula $numberOfOffsprings \cdot numberOfReproductions$. Cum în general prin procesul de încrucișare se generează doi descendenți, formula devine $2 \cdot numberOfReproductions$.

Procesul de selecție alege din populația curentă anumiți indivizi prin care se vor produce mai departe noi soluții. Modul de selecție introduce deobicei un factor aleator pentru a da o șansă întregii populații de a fi aleasă pentru încrucișare. Motivul pentru această alegere este încurajarea diversității populației, dând șansa și genelor mai slabe să supraviețuiască. Aceste gene mai slabe au posibilitatea de a conduce la zone neexplorate unde se pot găsi noi soluții optime. Printre cele mai utilizate metode de încrucișare din literatură sunt selecția de tip ruletă și selecția de tip turneu. Selecția de tip ruletă acordă fiecărui individ o șansă de a fi ales direct proporțională cu fitness-ul său în raport cu restul populației. Astfel, indivizii mai buni au o șansă mai mare de a fi aleși, crescând șansa ca descendenții să fie mai adecvați problemei. Selecția de tip turneu alege în mod uniform și aleator o parte a populației ca și participanți de turneu peste care se aplică un proces de determinare a unui câștigător. Pentru desemnarea câștigătorului se pot aplica mai multe modalități, cum ar fi o selecție de tip ruletă, șansa fiecărui candidat de a fi ales fiind raportată la fitness-ul său, sau alegerea directă a celui mai bun candidat. Alegerea câștigătorului în sine poate fi privită și ea ca o selecție aplicată pe o populație mai

mică. Selecțiile de tip turneu pot încuraja o diversitate mai mare decât selecțiile de tip ruletă, fiecare individ având o șansă egală de a fi ales pentru a participa mai departe la turneu. O selecție de tip turneu de dimensiune 2 unde câștigătorul este cel cu fitness mai bun poate fi văzută în figura 2.2.

Figura 2.2: Selecție de tip turneu binar
[AESEM20]

Încrucișarea reprezintă modul în care generăm noi soluții fezabile utilizând anumite soluții deja existente, denumite părinți. Încrucișarea standard utilizează doi părinți și generează doi descendenți, dar se poate opta și pentru varierea numerelor folosite. În particular, cazul cu un părinte și un descendent poate fi privit ca un proces de tip mutație. În continuare vom lucra doar cu cazul standard.

Pentru reprezentările vectoriale, cele mai populare metode sunt cele prin tăiere, în care soluțiile părinte sunt împărțite în subvectori și reasamblate. Procedeul standard începe prin alegerea unui punct aleator k. Cei doi părinți sunt împărțiți în două porțiuni după această poziție, generându-se un descendent prin ansamblarea primului subvector dintr-un părinte cu cel de-al doilea subvector din cel de-al doilea părinte. Prin inversarea ordinii părinților și aplicarea aceleași logici se poate obține un al doilea descendent în cadrul aceleași încrucișări. Acest procedeu poate fi generalizat pentru r puncte, creându-se astfel $r + 1$ bucăți ce sunt reasamblate prin alternare, generând de asemeni doi descendeți. În literatură, aceste metode sunt cunoscute și ca "One-Point Crossover", în general "K-Point Crossover", unde "K", este egal cu r-ul utilizat anterior. Un alt procedeu de încrucișare este cel de încrucișare uniformă. În cadrul acestuia, pentru fiecare poziție k, descendentul are șansa egală de a moșteni valoarea de pe aceiași poziție de la unul din cei doi părinți. Un al doilea descendent poate fi obținut prin utilizarea părților respinse de către descendentul descris anterior. Încrucișarea uniformă poate fi vazută și ca o încrucișare de tip K-Point Crossver cu numărul de tăieturi K generat aleator la fiecare aplicare a operatorului. Aceste procedee de încrucișare pot fi vizualizate în figura 2.3.

Pentru reprezentările de tip permutație se poate realiza de asemeni un procedeu de tip tăietură, cu condiția de a păstra proprietatea de permutare. Se consideră în continuare o valoare aleatoare k. Pentru crearea unui descendent, se copiază primele k valori dintr-unul dintre părinți. Se iterează cel de-al doilea părinte. De fiecare dată când găsim o valoare ce nu este deja utilizată, o inserăm în descendent. Lista

generată va fi o permutare validă. Schimbând rolurile părinților se poate genera concomitent o a doua soluție validă. Și acest procedeu poate fi generalizat pentru r tăieturi. Se pornește de la unul din părinți și descompunerea sa în $r + 1$ fragmente. Fragmentele cu număr de ordine impar se copiază în exactitate de la primul părinte. Pentru fragmentele cu număr de ordine par se ordonează elementele din fragmentul primului părinte după ordinea în care acestea apar în permutarea completă a celui de-al doilea părinte.

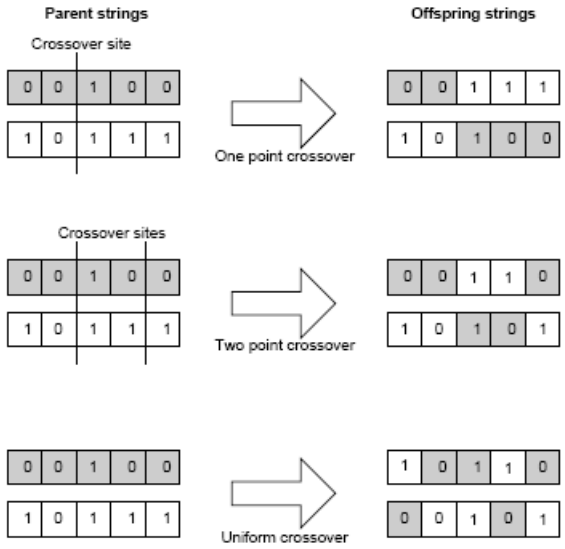


Figura 2.3: Operatori de încrucișare pentru reprezentări vectoriale [MQAV16]

Pentru fiecare individ pe care este aplicat, operatorul de mutație perturbă soluția pentru a genera una nouă validă. În contextul evoluției, mutația încearcă să introducă sau reintroducă o calitate în fondul genetic al unei specii. Pentru un algoritm genetic, acest lucru înseamnă explorarea unor noi zone ale spațiului de căutare. Se consideră important ca operatorul de mutație să fi capabil ca, prin aplicări repetate asupra unei soluții, să poată parcurge întreg spațiul de lucru. Astfel se garantează că orice soluție poate fi atinsă. Cum operatorul poate reprezenta un salt brusc și haotic, se încearcă aplicarea acestuia în mod limitat. Deobicei se introduce o probabilitate de mutație pentru a controla utilizarea acestuia. Pentru codificările de tip vector de valori, cea mai simplă mutație este reprezentată de schimbarea valorii de pe o poziție aleatoare k din vector cu o nouă valoare validă poziției respective. Pentru reprezentările de tip permutare, o mutație poate fi reprezentată de interschimbarea valorilor de pe două poziții aleatoare i, j.

Metoda de înlocuire utilizează indivizii populației curente și descendenții lor pentru a alege elementele viitoarei generații prin intermediul unor strategii alese. Cel mai simplu model ar fi cel în care următoarea generație este formată doar din descendenți, având ca dezavantaj faptul că se pot pierde pe parcurs soluțiile optime.

Pentru a asigura păstrarea minimelor și maximelor găsite se poate aplica o strategie elitistă, reunindu-se populația curentă cu descendeții ei și păstrându-se doar soluțiile cu fitness-ul cel mai bun. Prin această abordare sunt șanse ca populația să conveargă rapid la o vecinătate. În acest caz, o mare parte a spațiului soluțiilor rămâne neexplorat, putându-se rata anumite soluții mai bune. Altă strategie ar fi generarea unui număr mai mic de descendenți decât indivizi într-o populație, înlocuind în generația curentă cei mai slabi indivizi cu soluțiile noi. Astfel, fitness-ul celei mai bune soluții din generația curentă va fi garantat cel puțin la fel de bun ca cel mai bun fitness al generației precedente. Totodată, populația e mai permisivă, asigurând un grad mai mare de diversitate ca metoda elitistă. Această strategie este de tip stare stabilă.

Dintre aceste metode, criteriul de oprire, selecția și metoda de înlocuire nu depind de problemă, putând fi reutilizate de la un algoritm la altul. În schimb, metodele de generarea a populației, evaluare, încrucișare și mutație sunt dependente de problema rezolvată și reprezentarea utilizată.

2.3 Propunerea unor abordări genetice

2.3.1 Reprezentare și evaluare

În secțiunea 2.1 am transformat mulțimea soluțiilor problemei de planificare și am enunțat-o ca o problemă de căutare și optimizare. Astfel, problema ar putea fi rezolvată prin utilizarea algoritmilor genetici. Vom căuta și alege mai întâi un spațiu $ActivitiesPlanifications$ adecvat, acesta fiind încă nedeterminat. Pentru simplitate dar fără a pierde din generalitate, când ne referim la $ActivitiesPlanifications$ vom considera că planificarea de resurse este una fixată.

Vom introduce conceptul de listă de ordine de planificare a activităților. Aceasta este o permutare a elementelor mulțimii $Activities$ și reprezintă ordinea în care activitățile vor fi programate în cadrul proiectului. Vom nota această permutare cu $ActivitiesOrder$, $ActivitiesOrder \in Sym(Activities)$, unde $ActivitiesOrder(o)$ este a o-a activitatea ce va fi planificată. O proprietate necesară ca ordinea să fie validă este ca, dacă o activitate A_{k_1} este necesară începerii unei activități A_{k_2}, aceasta să fie programată înaintea ei. Acestă constrângere se modelează ca:

$$\forall o_1, o_2, 1 \leq o_1 < o_2 \leq n$$

$$(ActivitiesOrder(o_2), ActivitiesOrder(o_1)) \notin Dependencies$$

În termeni uzuali, pentru o activitate A_k, toate activitățile de care depinde se află la stânga ei, iar toate activitățile care depind de ea se află la dreapta ei. Acesta poate re-

prezenta un criteriu intuitiv de verificare că o permutare reprezintă o listă de ordine validă. Vom nota mulțimea tuturor ordonărilor valide prin $ValidActivitiesOrders$. Cum $ValidActivitiesOrders \subseteq Sym(Activities)$, iar $Sym(Activities)$ este finită, concluzionăm că și $ValidActivitiesOrders$ este implicit finită.

Permutarea $ActivitiesOrder$ poate fi privită ca o coadă de așteptare. Vom memora momentele de terminare ale activităților, respectiv momentele în care o resursă este din nou în circulație. Se ia elementul din vârful cozii și se verifică care este cel mai devreme moment în care condițiile permit executarea sa. Condițiile sunt reprezentate de dependențele față de celelalte activități și disponibilitatea resurselor. Astfel, timpul de start va fi maximul acestor valori. Odată determinat, se calculează și setează pentru activitate timpul de finisare. Se updatează pentru resursele utilizate timpul în care devin din nou valabile ca fiind momentul de terminare al activității curente. Procesul se repetă până când coada nu mai conține niciun element. Un exemplu concret poate fi vizualizat în figura 2.4. Utilizând acest proces se poate determina o unică funcție $StartTime$, deci o planificare. Afirmația inversă nu este adevărată, existând atât planificări ce nu pot fi transpuse, cât și planificări ce au mai multe transpuneri. Pentru a exemplifica, presupunem că nu există necesitatea de resurse sau calificări.

```
Activities={A1;A2;A3;A4}
Skills={S1}
Resources={R1}
SkillMasteries(R1,S1)=1

Dependencies={(A1;A2)}
Duration(A1)=3;Duration(A2)=1;Duration(A3)=2;Duration(A4)=2;
SkillRequirements(A1,S1)=1;SkillRequirements(A2,S1)=1;SkillRequirements(A3,S1)=1;SkillRequirements(A4,S1)=0

ActivityOrder=[A1,A4,A3,A2]

[A1,A4,A3,A2]     | [A4,A3,A2]        | [A3,A2]           | [A2]              | []
RefreshTime(R1)=0 | RefreshTime(R1)=3 | RefreshTime(R1)=3 | RefreshTime(R1)=5 | RefreshTime(R1)=6
StartTime(A1)  =0 | StartTime(A1)  =0 | StartTime(A1)  =0 | StartTime(A1)  =0 | StartTime(A1)  =0
StartTime(A2)  =0 | StartTime(A2)  =0 | StartTime(A2)  =0 | StartTime(A2)  =0 | StartTime(A2)  =5
StartTime(A3)  =0 | StartTime(A3)  =0 | StartTime(A3)  =0 | StartTime(A3)  =3 | StartTime(A3)  =3
StartTime(A4)  =0 | StartTime(A4)  =0 | StartTime(A4)  =0 | StartTime(A4)  =0 | StartTime(A4)  =0
EndTime(A1)    =0 | EndTime(A1)    =3 | EndTime(A1)    =3 | EndTime(A1)    =3 | EndTime(A1)    =3
EndTime(A2)    =0 | EndTime(A2)    =0 | EndTime(A2)    =0 | EndTime(A2)    =0 | EndTime(A2)    =6
EndTime(A3)    =0 | EndTime(A3)    =0 | EndTime(A3)    =0 | EndTime(A3)    =5 | EndTime(A3)    =5
EndTime(A4)    =0 | EndTime(A4)    =0 | EndTime(A4)    =2 | EndTime(A4)    =2 | EndTime(A4)    =2
```

Figura 2.4: Decodificarea unei liste de ordine

Pentru cazul $Activities = \{A1, A2\}$, $Dependencies = \{(A1, A2)\}$ avem o singură listă de ordine validă, $[A1, A2]$, planificarea asociată fiind dată prin $StartTime(A1) = 0$ și $StartTime(A2) = Duration(A1)$. Se observă ca există planificarea $StartTime_0$ cu $StartTime_0(A1) = 0$ și $StartTime_0(A2) = Duration(A1)+1$ ce este validă. Pentru datele curente există o unică listă de ordine validă ce ne dă planificarea $StartTime$, ce este evident diferită de $StartTime_0$. Deci, $StartTime_0$ nu are reprezentări sub formă de listă de ordine, singura listă validă ducând la o altă planificare.

Considerăm datele $Activities = \{A1, A2\}$, $Dependencies = \emptyset$. În acest caz avem

două liste de ordine valide, $[A1, A2]$ și $[A2, A1]$. Utilizând procesul definit mai sus, vom obține pentru ambele aceiași planificare $StartTime$ cu $StartTime(A1) = 0$ și $StartTime(A2) = 0$. Deci, este posibil ca pentru o planificare să avem reprezentări multiple sub formă de listă de ordine.

Vom defini un proces prin care putem rafina o soluție validă $StartTime$. Ordonând crescător activitățile după momentul de start asociat, obținem o listă de ordine validă $ActivitiesOrder$ cu proprietatea

$$\forall o_1, o_2, 1 \leq o_1 < o_2 \leq n$$

$$StartTime(ActivitiesOrder(o_1)) \leq StartTime(ActivitiesOrder(o_2))$$

Generând o planificare $StartTime_0$ din lista de ordine $ActivitiesOrder$, procesul de construcție garantează că fiecare activitate va fi executată în cel mai devreme moment posibil atunci când este planificată. Deci momentul de start al fiecărei activități va fi cel puțin la fel de bun ca cel din soluția inițială $StartTime$. Prin enunțarea acestei metode de rafinare am demonstrat că pentru orice planificare $StartTime$ există o planificare cel puțin la fel de bună $StartTime_0$ ce este reprezentabilă ca listă de ordine. Mulțimea tuturor listelor de ordine valide $ValidActivitiesOrders$ va conține atunci reprezentări pentru cele mai bune soluții la problema de planificare.

Vom alege în continuare $ActivitiesPlanifications$ ca fiind spațiul determinat de soluțiile ce sunt reprezentabile în $ValidActivitiesOrders$, asigurând proprietățile căutate.

Prin lista de ordine am reușit să reducem spațiul soluțiilor la unul finit, totodată asigurând că cele mai bune soluții pot fi găsite. Dezavantajul acestei reprezentări este faptul că există elemente redundante, două liste putând duce la aceiași planificare. Cu cât o soluție permite mai multe reprezentări, cu atât domină mai mult noul spațiu, având șanse mai mari de a se ivi în cadrul unei căutări.

Lista de ordine este una din metodele standard de reprezentare a unei planificări de activități, fiind utilizată în numeroase rezolvări ale problemei, dar sub alte denumiri precum listă de activități [YA11] [SV21] sau listă de sarcini [CZ13]. Vom propune o nouă reprezentare pentru planificarea activităților inspirată din lucrarea lui Snauwaert si Vanhoucke[SV21]. În aceasta se utilizează conceptul de listă de prioritate pentru alocarea de resurse. Ideea de bază este utilizarea unor reguli pentru a stabili ce resursă este alocată atunci când o activitate este începută. Vom introduce conceptul de listă de priorități pentru planificarea de activități.

O listă de priorități este o permutare $ActivitiesPriority \in Sym(Activities)$. Vom nota mulțimea tuturor astfel de liste prin $ValidActivitiesPriorities$. Spre deosebire de lista de ordine, nu vom impune nicio condiție în plus pentru a considera o listă de priorități validă, deci $ValidActivitiesPriorities = Sym(Activities)$. Cum

$Sym(Activities)$ este finită, și $ValidActivitiesPriorities$ va fi o mulțime finită. Acestă listă va dicta care este prioritatea fiecărei activități în planificare. Pentru p_1, p_2 cu proprietatea $p_1 < p_2$, vom spune că activitatea $ActivitiesPriority(p_1)$ are prioritate mai mare ca $ActivitiesPriority(p_2)$. Faptul că o activitate are prioritate mai mare nu înseamnă neapărat ca va fi planificată mai rapid ca orice activitate cu prioritate mai mică. Spre exemplu, dacă $(ActivitiesPriority(p_2), ActivitiesPriority(p_1)) \in Dependencies$ este evident că nu putem planifica activitatea $ActivitiesPriority(p_1)$ înaintea lui $ActivitiesPriority(p_2)$.

Vom defini un proces de decodificare a listei de priorități. Asemeni listei de ordine, vom memora momentele de terminare ale activităților și momentele în care resursele sunt din nou disponibile. Se iterează lista de priorități până când găsim o activitate ce poate fi efecutată, altfel spus, toate activitățile de care este dependentă sunt deja planificate. Îi programăm execuția pentru cel mai devreme moment posibil, prin același proces definit la lista de ordine, apoi scoatem elementul din listă. Procedura se repetă până când lista de priorități nu mai conține elemente. Acest proces poate fi vizualizat în figura 2.5.

```
Activities={A1;A2;A3;A4}
Skills={S1}
Resources={R1}
SkillMasteries(R1,S1)=1

Dependencies={(A1;A2)}
Duration(A1)=3;Duration(A2)=1;Duration(A3)=2;Duration(A4)=2;
SkillRequirements(A1,S1)=1;SkillRequirements(A2,S1)=1;SkillRequirements(A3,S1)=1;SkillRequirements(A4,S1)=0

ActivityOrder=[A2,A4,A1,A3]

[A2,A4,A1,A3]     | [A2,A1,A3]        | [A2,A3]           | [A3]              | []
RefreshTime(R1)=0 | RefreshTime(R1)=0 | RefreshTime(R1)=3 | RefreshTime(R1)=4 | RefreshTime(R1)=6
StartTime(A1)  =0 | StartTime(A1)  =0 | StartTime(A1)  =0 | StartTime(A1)  =0 | StartTime(A1)  =0
StartTime(A2)  =0 | StartTime(A2)  =0 | StartTime(A2)  =0 | StartTime(A2)  =3 | StartTime(A2)  =3
StartTime(A3)  =0 | StartTime(A3)  =0 | StartTime(A3)  =0 | StartTime(A3)  =0 | StartTime(A3)  =4
StartTime(A4)  =0 | StartTime(A4)  =0 | StartTime(A4)  =0 | StartTime(A4)  =0 | StartTime(A4)  =0
EndTime(A1)    =0 | EndTime(A1)    =0 | EndTime(A1)    =3 | EndTime(A1)    =3 | EndTime(A1)    =3
EndTime(A2)    =0 | EndTime(A2)    =0 | EndTime(A2)    =0 | EndTime(A2)    =4 | EndTime(A2)    =4
EndTime(A3)    =0 | EndTime(A3)    =0 | EndTime(A3)    =0 | EndTime(A3)    =0 | EndTime(A3)    =6
EndTime(A4)    =0 | EndTime(A4)    =2 | EndTime(A4)    =2 | EndTime(A4)    =2 | EndTime(A4)    =2
[]                | [A4]              | [A4,A1]           | [A4,A1,A2]        | [A4,A1,A2,A3]
```

Figura 2.5: Decodificarea unei liste de priorități

Se observă că lista formată din activtățile efectuate plasate în ordinea în care au fost eliminate reprezintă o listă de ordine. Extrăgând din procesul de decodificare doar porțiunea ce alege următoarea activitate ce va fi programată, obținem un proces de transformare a unei liste de priorități în listă de ordine. Datorită acestuia, putem defini un procedeu alternativ de determinare a planificării asociate listei de priorități. Pornind de la o listă de priorități, generăm lista de ordine asociată pe care o decodificăm, obținând planificarea căutată. Astfel, planificarea găsită va corespunde listei de ordine asociate.

Din moment ce o listă de ordine este o permutare de activități, este posibilă in-

terpretarea acesteia ca listă de priorități. Vom studia comportamentul permutărilor ce reprezintă liste de ordine valide din perspectiva listelor de priorități. Aplicând transformarea din listă de priorități în listă de ordine vom ajunge înapoi la lista de ordine de la care am plecat, și deci la aceiași planificare. Putem concluziona astfel că orice listă de ordine este reprezentabilă ca listă de priorități, cea mai simplă reprezentare fiind chiar lista în sine.

Cum orice listă de priorități este reprezentabilă ca listă de ordine și orice listă de ordine este reprezentabilă ca listă de priorități, concluzionăm ca spațiul soluțiilor de asemeni va coincide. Cum spațiul listelor de ordine este inclus în spațiul listelor de priorități, listele de priorități pot fi văzute ca o generalizare a listelor de ordine. Astfel, $ValidActivitiesPriorities$ poate fi utilizat pentru a reprezenta în totalitate și cu exactitate spațiul soluțiilor $ActivitiesPlanifications$.

Putem defini un proces de verificare pentru listele de ordine. Aplicând transformarea peste o permutare ce nu reprezintă o listă de ordine validă rezultatul nu va coincide cu lista inițială, acesta fiind o listă de ordine validă. În schimb, aplicând transformarea peste o listă de ordine validă, vom ajunge mereu la lista de la care am plecat. Astfel, putem verifica că o listă de ordine este validă prin aplicarea transformării și compararea rezultatului cu lista inițială.

```
Activities={A1;A2;A3} Skills={S1} Resources={R1} SkillMasteries(R1,S1)=1

Dependencies={(A1;A2)} Duration(A1)=3;Duration(A2)=1;Duration(A3)=2
SkillRequirements(A1,S1)=1;SkillRequirements(A2,S1)=0;SkillRequirements(A3,S1)=1

We consider that R1 is assigned for every activity requiring S1
ValidActivitiesOrders={[A1;A2;A3],[A1;A3;A2],[A3;A1;A2]}

Schedule1:                              | Schedule2:
Associated Orders: [A1;A2;A3],[A1;A3;A2]  | Associated Orders: [A3;A1;A2]
Refresh(R1)=[0;3;5]                     | Refresh(R1)=[0;3;5]
StartTime(A1)=0;  EndTime(A1)=3         | StartTime(A1)=2;  EndTime(A1)=5
StartTime(A2)=3;  EndTime(A2)=4         | StartTime(A2)=5;  EndTime(A2)=6
StartTime(A3)=3;  EndTime(A3)=5         | StartTime(A3)=0;  EndTime(A3)=2

Schedule1 has 2 representations, Schedule2 has 1 representation in ValidActivitiesOrders - ratio of 2/1

ValidActivitiesPriorities
Transformation of Priority Lists into Order Lists and the associated Schedule
[A1;A2;A3] -> [A1;A2;A3] -> Schedule1
[A1;A3;A2] -> [A1;A3;A2] -> Schedule1
[A2;A1;A3] -> [A1;A2;A3] -> Schedule1
[A2;A3;A1] -> [A3;A1;A2] -> Schedule2
[A3;A1;A2] -> [A3;A1;A2] -> Schedule2
[A3;A2;A1] -> [A3;A1;A2] -> Schedule2

Schedule1 has 3 representations, Schedule2 has 3 representations in ValidActivitiesPriorities ratio of 1/1
```

Figura 2.6: Caz favorabil pentru lista de prioritate

Cum $ValidActivitiesPriorities$ este o generalizare a $ValidActivitiesOrders$ sunt păstrate și anumite particularități legate de maparea planificărilor la reprezentări. Înafara spațiului definit, o planificare poate în continuare să nu fie reprezentabilă sau să aibă asociate reprezentări multiple. În contextul lui $ActivitiesPlanifications$ persistă doar faptul că o soluție poate avea codificări multiple, chiar mai multe ca în $ValidActivitiesOrders$. Putem observa totuși un avantaj generat de această pro-

prietate. În $ValidActivitiesOrders$ exista posiblitatea ca o planificare să fie mai predominantă, și deci să defavorizeze soluțiile cu mai puține reprezentări în procesul de căutare. Mulțimea $ValidActivitiesPriorities$ introduce noi reprezentări ce sunt asociate planificărilor, aducând posibilitatea de eliminare a dominanțelor. Un astfel de caz poate fi vizualizat în figura 2.6.

Totuși, comportamentul listelor de prioritate nu este mereu favorabil. Introducerea de noi soluții poate să nu aibă impact asupra dominanței soluțiilor, sau chiar să o influențeze negativ. Exemple pentru astfel de cazuri sunt redate în figurile 2.7 și 2.8.

```
Activities={A1;A2;A3} Skills={S1} Resources={R1} SkillMasteries(R1,S1)=1

Dependencies={(A1;A2)} Duration(A1)=3;Duration(A2)=1;Duration(A3)=2
SkillRequirements(A1,S1)=0;SkillRequirements(A2,S1)=1;SkillRequirements(A3,S1)=1

We consider that R1 is assigned for every activity requiring S1
ValidActivitiesOrders={[A1;A2;A3],[A1;A3;A2],[A3;A1;A2]}

Schedule1:                    | Schedule2:
Associated Orders: [A1;A2;A3]  | Associated Orders: [A1;A3;A2],[A3;A1;A2]
Refresh(R1)=[0;4;6]            | Refresh(R1)=[0;2;4]
StartTime(A1)=0;  EndTime(A1)=3 | StartTime(A1)=0;  EndTime(A1)=3
StartTime(A2)=3;  EndTime(A2)=4 | StartTime(A2)=3;  EndTime(A2)=4
StartTime(A3)=4;  EndTime(A3)=6 | StartTime(A3)=0;  EndTime(A3)=2

Schedule1 has 1 representation, Schedule2 has 2 representations in ValidActivitiesOrders - ratio of 1/2

ValidActivitiesPriorities
Transformation of Priority Lists into Order Lists and the associated Schedule
[A1;A2;A3] -> [A1;A2;A3] -> Schedule1
[A1;A3;A2] -> [A1;A3;A2] -> Schedule2
[A2;A1;A3] -> [A1;A2;A3] -> Schedule1
[A2;A3;A1] -> [A3;A1;A2] -> Schedule2
[A3;A1;A2] -> [A3;A1;A2] -> Schedule2
[A3;A2;A1] -> [A3;A1;A2] -> Schedule2

Schedule1 has 2 representations, Schedule2 has 4 representations in ValidActivitiesPriorities - ratio of 1/2
```

Figura 2.7: Caz fără impact pentru lista de prioritate

```
Activities={A1;A2;A3} Skills={S1} Resources={R1} SkillMasteries(R1,S1)=1

Dependencies={(A1;A2)} Duration(A1)=3;Duration(A2)=1;Duration(A3)=2
SkillRequirements(A1,S1)=1;SkillRequirements(A2,S1)=1;SkillRequirements(A3,S1)=1

We consider that R1 is assigned for every activity requiring S1
ValidActivitiesOrders={[A1;A2;A3],[A1;A3;A2],[A3;A1;A2]}

Schedule1:                    | Schedule2:                    | Schedule3:
Associated Orders: [A1;A2;A3]  | Associated Orders: [A1;A3;A2]  | Associated Orders: [A3;A1;A2]
Refresh(R1)=[0;3;4;6]          | Refresh(R1)=[0;3;5;6]          | Refresh(R1)=[0;2;5;6]
StartTime(A1)=0;  EndTime(A1)=3 | StartTime(A1)=0;  EndTime(A1)=3 | StartTime(A1)=2;  EndTime(A1)=5
StartTime(A2)=3;  EndTime(A2)=4 | StartTime(A2)=5;  EndTime(A2)=6 | StartTime(A2)=5;  EndTime(A2)=6
StartTime(A3)=4;  EndTime(A3)=6 | StartTime(A3)=3;  EndTime(A3)=5 | StartTime(A3)=0;  EndTime(A3)=2

Schedule1 has 1 representation, Schedule2 has 1 representation,
Schedule3 has 1 representation in ValidActivitiesOrders - ratio of 1/1/1

ValidActivitiesPriorities
Transformation of Priority Lists into Order Lists and the associated Schedule
[A1;A2;A3] -> [A1;A2;A3] -> Schedule1
[A1;A3;A2] -> [A1;A3;A2] -> Schedule2
[A2;A1;A3] -> [A1;A2;A3] -> Schedule1
[A2;A3;A1] -> [A3;A1;A2] -> Schedule3
[A3;A1;A2] -> [A3;A1;A2] -> Schedule3
[A3;A2;A1] -> [A3;A1;A2] -> Schedule3

Schedule1 has 2 representation, Schedule2 has 1 representations,
Schedule3 has 3 representations in ValidActivitiesPriorities - ratio of 2/1/3
```

Figura 2.8: Caz negativ pentru lista de prioritate

Pentru a reprezenta o alocare de resurse, vom utiliza reprezentarea standard de listă de asignări. Vom nota această mulțime $ValidResourcesAllocations$. O soluție va fi o listă $ResourcesAllocations$ cu proprietatea că cel de-al k-lea element reprezintă o mapare a skill-urilor la resursele utilizate pentru o activitate A_k, modelată prin

$$ResourcesAllocations(k) = ResourcesAllocation_k$$

$$ResourcesAllocation_k(S_i) = ResourcesAssigned(A_k, S_i), \forall S_i \in Skills$$

Obținem astfel un proces de a genera pentru orice planificare de resurse o listă de alocări și vice-versa. Astfel, toate asignările de resurse sunt reprezentabile în acest spațiu.

Alte reprezentări utilizate sunt cele de liste de priorități pentru asignare de resurse [SV21]. Ideea de bază este că, atunci când trebuie să alocăm unei activități resursele necesare, acestea vor fi alese în funcție de prioritatea din listă și de regulile de prioritate alese. Se poate utiliza o listă comună de priorități pentru toate activitățile, cu costul de a limita spațiul de căutare, sau se poate atașa câte o listă separată fiecărei activități, crescând complexitatea de memorie la $\Theta(PopulationSize \cdot l^2)$, unde $PopulationSize$ este numărul de indivizi din populație, iar l este numărul de resurse. În ambele situații există posibilitatea ca o planificare de resurse să aibă mai multe reprezentări valide. Din cauza dezavantajelor menționate anterior, nu vom folosi reprezentarea de tip listă de priorități, ci reprezentarea standard de listă de asignări.

Astfel, avem două domenii de reprezentări cu care vom lucra, anume

$$ValidActivitiesPriorities \times ValidResourcesAllocations$$

$$ValidActivitiesOrders \times ValidResourcesAllocations$$

ambele acoperind în completitudine spațiul de căutare $ActivitiesPlanifications \times ResourcesPlanifications$. Pentru ambele mulțimi, un element este o pereche formată dintr-o permutare de activități $ActivitiesList$ și o listă de asignări de resurse $ResourcesAllocations$.

Vom defini metodele de evaluarea și vom oferi complexitățile lor asimptotice de timp pentru a ne face o idee despre durata de evaluare a unui candidat, aceasta fiind importantă așa cum a fost menționat în 2.2. Pentru a evalua o reprezentare, se folosește metoda de decodificare aferentă tipului de reprezentare, generând o planificare $StartTime$. Cum din aceasta putem deduce $EndTime$, se poate utiliza formula prezentată în 1.2 pentru a afla $ProjectDuration$. Complexitatea de timp aferentă evaluării unei liste de ordine este $O(n(n + l))$, unde n este numărul de activități iar l numărul de resurse. Notația nu este Θ deoarece este posibil ca nicio

activitate să nu necesite nicio resursă existentă. Transformarea unei liste de priorități în listă de ordine presupune o complexitate de timp de $O(n^2)$. Nici aici notația nu este Θ datorită cazului în care toate activitățile sunt independente una de alta. Deci, complexitatea de timp finală va rămâne în continuare $O(n(n+l))$ și pentru lista de priorități.

2.3.2 Generarea populației inițiale

Vom defini proceduri de generare pentru listele de activități și pentru listele de asignări de resurse. Pentru listele de priorități se poate utiliza un proces simplu de generare aleatoare a unei permutări, unde toate permutările să aibă șansa egală de a fi generate. Cum lista de priorități este o permutare simplă, fără alte constrângeri, acest proces este îndeajuns și ne garantează că putem genera orice element al spațiului. Generând aleator o listă de priorități, putem aplica apoi procesul de transformare în listă de ordine. Astfel am obținut o modalitate de generare aleatoare și pentru liste de ordine.

Vom descrie în continuare o procedură alternativă pentru liste de ordine. Se memorează activitățile executabile, dar ce nu sunt încă programate. Inițial, aceasta este formată din activitățile ce nu sunt dependente de alte activtăți. La fiecare iterație scoatem aleator una dintre aceste activități. Actualizăm lista de activități executabile adăugând noile activități ce pot fi executate. Acest proces se repetă până când nu mai avem activități rămase, ordinea în care am ales activitățile determinând lista de ordine. Ni se garantează că putem ajunge la orice listă de ordine. Intuitiv, dacă dorim să ajungem la o listă de ordine dată l, este îndeajuns ca la fiecare pas al generării să alegem activitatea din l a cărei poziție corespunde pasului curent. Această metodă de generare este descrisă și în lucrarea lui Yannibelli și Amandi[YA11]. Un exemplu concret împreună cu probabilitățile reprezentărilor finale este dat in figura 2.9.

Se observă că nu avem șanse egale de a obține fiecare reprezentare. Aceiași situație o avem pentru metoda inițială, în acest caz șansa de generare fiind direct proporțională cu numărul de reprezentări ce lista de ordine îl are în formatul de listă de priorități. Prima metodă de generare ar fi mai bună în cazul în care reprezentarea de listă de prioritate este avantajată. Cum ambele metode pot genera soluții mai variate sau mai puțin variate în funcție de context, vom merge pe cea de a doua, fiind mai bine documentată pentru reprezentarea de listă de ordine.

Pentru a genera liste de asignări de resurse vom itera activitățile și vom genera pentru fiecare în parte o asignare validă. Pentru o activitate se crează o listă de calificări cu proprietatea că fiecare calificare apare de atâtea ori cât necesită acitivitatea. Se aplică un proces de backtracking pentru a găsi o listă de resurse cu proprietatea

că nicio resursă nu se repetă și că resursa de pe poziția i stăpânește calificarea de pe poziția i în lista creată anterior. Ca soluțiile generate să fie aleatoare, lista de calificări și lista de resurse valide ce va fi iterată pentru poziția i vor fi ordonate aleator.

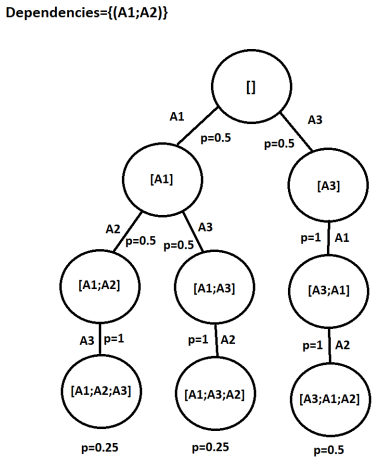


Figura 2.9: Generare constructivă de liste de ordine

Acest procedeu ne garantează acoperirea spațiului reprezentărilor. Cum generarea unei asignări nu influențează generarea o alteia, este îndeajuns să arătăm că orice asignare validă pentru anumite necesități poate fi generată. Exprimăm necesitățile sub forma de listă precizată anterior. Dacă dorim o anumită asignare validă, o reprezentăm și pe aceasta sub formă de listă de resurse. Cum asignarea este validă, garantat putem reordona elementele listei astfel încât fiecare resursă să fie pe o poziție corectă din punct de vedere al stăpânirii calificării asociate. Utilizând această listă pentru a impune ordinea de alegere a resureselor în procesul de backtracking definit anterior, ne garantează că ajungem la asignarea căutată. Astfel, orice asignare validă poate fi generată prin acest proces. Totuși, nici acest procedeu nu va garanta că toate reprezentările au șansă egală de a fi generate. În ciuda dezavantajului, vom folosi această procedură.

2.3.3 Încrucișare și mutație

Pentru procesul de încrucișare pe listele de activități, vom considera pentru ambele reprezentări o metodă comună. Vom utiliza încrucișarea de tip tăietură într-un punct descrisă în secțiunea 2.2, generând noi permutări ale activităților. Pentru reprezentarea ca listă de priorități, este evident faptul că rezultatul va fi tot un element valid, neexistând constrângeri în plus peste permutări. Vom demonstra că pentru lista de ordine rezultatul rămâne valid.

Considerăm o listă de ordine l, obținută din părinții l_1 și l_2 după pozitiția k. Vom aplica procesul de verificare definit în 2.3.1. Considerând l ca simplă permutare, formăm lista de ordine asociată. Este evident că primele k elemente vor coincide cu

cele din lista inițială, acestea provenind din l_1 ce este deja o listă de ordine validă. Iterăm restul permutării inițiale. Pentru a programa activitatea ce urmează $l(k+1)$, este necesar să fim siguri că toate activitățile de care depinde au fost programate deja. Comform procesului de încrucișare, $l(k+1)$ va fi primul element din l_2 ce nu a fost programat deja în l. Cum l_2 este o listă de ordine, toate activitățile de care depeninde $l(k+1)$ se vor afla la stânga acestuia. Cum acestea au fost deja programate, înseamnă că putem să-l programăm și pe $l(k+1)$. Activitatea $l(k+2)$ va fi următoarea activitate din l_2 ce nu a fost deja programată. Aplicând același raționament, deducem că o putem programa imediat. Acest proces se repetă pentru restul elementelor din lista l. Lista de ordine rezultată va fi chiar l. Din această cauză, l este listă de ordine, demonstrând corectitudinea operatorului.

Se pot utiliza și procedee de tip încrucișare cu r tăieturi, acestea generând în continuare liste de ordine și priorități valide. Pentru simplitate, vom opta pentru varianta cu o tăietură.

Pentru procesul de încrucișare la nivel de liste de asignări de resurse aplicăm procedeul de tip încrusișare uniformă pentru vectori de valori. Fie r_1, r_2 doi părinți si r descendentul. Pentru o poziție k avem asignările pentru activitatea A_k. Cum r_1 și r_2 sunt valide, $r_1(k)$ și $r_2(k)$ vor fi asignări valide pentru A_k. Pentru r, $r(k)$ va fi moștenit aleator de la unul dintre cei doi, deci $r(k)$ va fi valid. Cum acest lucru se întâmplă pentur toți k, r va fi o listă de asignări validă.

Procesul de mutație utilizat pentru listele de priorități va fi cel standard pentru permutări, rezultatul fiind tot o permutare și deci o listă de priorități validă. Cum acesta este un operator standard pentru mutația permutărilor, este bine cunoscut faptul că prin aplicarea sa repetată se poate ajunge la orice altă permutare validă. Această metodă nu poate fi aplicată și pe o listă de ordine, fiind evident faptul că ar putea produce soluții invalide dacă sunt interschimbate două activități între care se află dependență. Vom defini o nouă metodă de mutație.

Considerăm o listă de ordine l și o poziție k astfel încât $l(k)$ să nu fie necesară completării lui $l(k+1)$. Definim l_r ca fiind permutarea obținută prin interschimbarea lui $l(k)$ cu $l(k+1)$. Aplicăm procesul de verificare a listelor de ordine. Vom genera lista de ordine asociată lui l_r. În lista rezultată, primele $k-1$ elemente vor corespunde listei inițiale, acestea fiind deja ordonate. Pentru planificarea imediată a elementului $l_r(k) = l(k+1)$ trebuie să ne asigurăm că toate activitățile de care depinde au fost organizate deja. Cum l este listă de ordine validă, înseamnă că activitățile necesare lui $l(k+1)$ pot fi doar $l(1), l(2), ..., l(k)$. Cum știm garantat că $l(k+1)$ nu depinde de $l(k)$, lista este redusă la $l(1), l(2), ..., l(k-1)$. Acestea corespund activităților $l_r(1), l_r(2), ..., l_r(k-1)$, ce sunt deja programate, însemnând că putem planifica $l_r(k) = l(k+1)$ imediat. Utilizând o logică asemanătoare, pentru elementul $l_r(k+1) = l(k)$ observăm de asemeni că toate activitățile de care ar fi

putut depinde au fost deja planificate, deci putem planifica imediat activitatea. Pentru $p > k + 1$ ce are asociată valoarea $l_r(p) = l(p)$ putem constata că elementele ce se află la stânga lui $l_r(p)$ în l_r sunt aceleași cu elementele aflate la stânga lui $l(p)$ în l. Continuând procesul de creare a listei de ordine, pentru fiecare element rămas putem aplica această observație și să deducem că toate dependențele activității sunt satisfăcute, deci o putem planifica imediat. Astfel, lista de ordine rezultată este lista de la care am început. Conform procedeului de verificare, l_r este o listă de ordine. Astfel, aplicând acest proces peste o listă de ordine, rezultatul va fi de asemeni o listă de ordine.

Vom demonstra că dintr-o listă de ordine oarecare l_i putem ajunge prin aplicarea repetata a operației anterioare la orice altă listă de ordine validă l_f. Iterăm lista de ordine l_i. Presupunem că primele $k - 1$ elemente ale listelor coincid. Asta înseamnă că toate dependențele lui $l_f(k)$ se află printre elementele $l_f(1) = l_i(1), l_f(2) = l_i(2), ...,$ $l_f(k-1) = l_i(k-1)$. Fie $k_0 > k$ poziția lui $l_f(k)$ în l_i. Asta înseamnă că, în lista inițială l_i, elementele $\{l_i(k), l_i(k+1), l_i(k+2), ..., l_i(n)\} \setminus \{l_i(k_0)\}$ nu sunt necesare completării lui $l_i(k_0)$. Aplicând repetat procesul de mutație definit anterior, putem muta $l_i(k_0)$ pe poziția k. Acum, primele k elemente coincid. Aplicând repetat această metodă, ajungem de la lista l_i la lista l_f. Astfel, este garantat că de la orice listă de ordine validă putem ajunge la oricare altă listă de ordine validă prin aplicarea repetată a mutației. Acest lucru, împreună cu demonstrarea anterioară a faptului că aplicarea operatorului ne garantează că rezultatul este tot o listă de ordine validă ne asigură că operatorul de mutație definit este corect.

Utilizând mutația anterioară, vom defini o mutație mai puternică. Alegem aleator o poziție k în cadrul listei de ordine l. Găsim limitele c_1 și c_2 astfel încât pentru orice element c, $c1 \leq c \leq c2$ să nu existe dependențe directe între $l(c)$ și $l(k)$. Alegem o nouă valoare aleatoare c_0 pe intervalul determinat de c_1 și c_2. Dacă $c_0 < c$, vom aplica repetat operatorul de mutație definit anterior pentru a-l muta pe $l(c)$ de la dreapta la stânga, până ce acesta ajunge pe poziția c_0. Asemănător, daca $c_0 > c$, îl vom muta pe $l(c)$ de la stânga la dreapta prin mutații repetate. Dacă $c_0 = c$, lista rămâne neschimbată. Cum rezultatul este obținut prin aplicarea repetată a unei operații de mutație validă, rezultatul va fi tot o listă de ordine validă. Totodată, se observă că pentru $c_0 = c+1$ și $c_0 = c-1$ operatorul este chiar mutația anterioară. Astfel, pornind de la o listă oarecare este garantat că putem ajunge la oricare altă listă prin aplicarea repetată a mutatorului nou definit. Datorită acestora, procesul definit reprezintă o metodă de mutație validă. Acesta coincide cu procesul de mutație definit de Yannibelli în lucrarea [YA11]. Vom lucra cu acest operator de mutație pentru listele de ordine.

Pentru mutația listelor de asignări vom utiliza o mutație standard pentru vectori de valori. Vom alege aleator o activitate și vom genera o nouă alocare folosind

procesul de generare de asignări definit în secțiunea 2.3.2. Astfel, se garantează că rezultatul mutației rămâne în continuare unul valid.

Pornind de la o listă de asignări validă oarecare a_i putem ajunge la oricare altă listă de asignări validă a_f. Aplicăm de n ori operatorul de mutație peste a_i. Pentru cea de-a k-a aplicație a operatorului impunem regenerarea asignării resurselor pentru activitatea A_k, adică regenerarea lui $a_i(k)$. Cum procesul de generare poate produce orice asignare de resurse validă lui A_k, impunem ca acesta să genereze asignarea de resurse $a_f(k)$ ce este de asemeni validă. Prin acest proces, la final vom ajunge chiar la lista de asignări a_f. Deci, aplicând repetat procesul de mutație definit, putem ajunge de la orice listă validă la asignări la oricare altă listă validă de asignări. Cum am demonstrat anterior inclusiv că pentru orice listă de asignări validă mutația produce de asemeni o listă de asignări validă, operatorul de mutație este unul valid.

Datorită procesului de transformare a listelor de priorități în liste de ordine, se pot formula și alți operatori de mutație și încrucișare pentru listele de ordine. Interpretând lista de ordine ca liste de priorități, putem aplica pe aceasta orice operatori de încrucișare și mutație ce funcționează pe această reprezentare. Rezultatele obținute sunt supuse procedeului de transformare a listelor de priorități în liste de ordine. Astfel, rezultatele vor fi mereu liste de ordine valide. În continuare ne vom concentra pe metodele deja definite, deci nu vom utiliza astfel de operatori.

Aplicarea mutației se va face restrâns, controlarea acesteia făcându-se printr-un proces aleator. Vom asocia astfel fiecărui operator de mutație o probabilitate de a fi aplicat. Se observă cum procesul de încrucișare este realizat la nivel de listă de asignări, dar nu și la alocările individuale. Pentru a remedia acest fapt se poate utiliza o probabilitate mai mare de mutație pentru listele de asignări.

2.3.4 Metode de selecție și înlocuire, criteriu de oprire

Vom utiliza o metodă de selecție de tip turneu. Vom alege aleator doi indivizi ai populației curente ce vor forma populația de turneu. Dintre cei doi, cel cu fitness mai bun, în cazul nostru mai mic, va fi selectat ca și câștigător. Aplicând procedeul de două ori, generăm o pereche de părinți. Cum a fost descris în secțiunea 2.2, această alegere va favoriza diversitatea. Procesul de înlocuire va fi unul de tip stare stabilă, asigurând atât diversitatea cât și păstrarea celor mai bune soluții găsite până în momentul de față. Numărul de indivizi înlocuiți va fi egal cu numărul de soluții generate într-o generație. Parametrii reprezentați de mărimea populației și numărul de iterații sunt variabili, astfel vor fi aleși ulterior. Criteriul de oprire se stabilește în funcție de necesitate. Criteriul principal folosit va fi unul bazat pe trecerea unei durat de timp prestabilite.

Capitolul 3

Implementarea abordării

După cum a fost prezentat în 2.2, algoritmii genetici sunt ușor parametrizabili, fiecare operator fiind reutilizabil într-o anumită măsura. Atunci când se dorește compararea a doi operatori în cadrul aceleiași probleme acest lucru se dovedește important, eliminând necesitatea de a duplica cod. Astfel, se pot încapsula comportamentele operatorilor în obiecte ce sunt apoi utilizate în cadrul algoritmilor. Din acest motiv s-au considerat ca limbaje de programare Java și Python, amândouă prezentând calificări pentru dezvoltarea sub paradigma orientată pe obiecte. Realizând un simplu algoritm genetic în ambele limbaje, am observat cum Python permite o dezvoltare mai rapidă, datorită atât a simplității sale în sintaxă, cât și a faptului că inclusiv funcțiile sunt obiecte. Din acest considerent, continuarea dezvoltării algoritmilor a fost făcută în Python, iar lucrarea va face referire doar la această implementare.

3.1 Structurarea proiectului

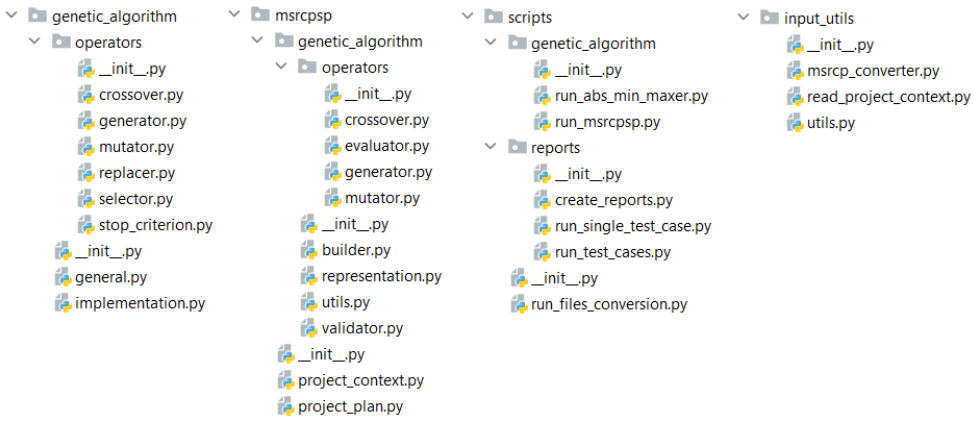


Figura 3.1: Structurarea pachetelor și modulelor

Proiectul este structurat pe pachete, aspectele necesare fiecăruia fiind separate în module. Structura generală de referință a proiectului poate fi văzută în figura

3.1. Pachetul ”genetic_algorithm” conține tot ce este necesar unui punct de start în dezvoltarea unui algoritm genetic. Acesta nu depinde de alte pachete. Modulul său principal este ”implementation” unde se află implementarea în sine de algoritm genetic generic. Modulul ”general” conține aspecte de uz comun algoritmului și operatorilor, precum conceptul de candidat și constante reprezentative pentru problemele de minimizare și maximizare. Fiecare modul al subpachetului ”operators” este dedicat unui anumit tip de operator necesar algoritmilor genetici. Aceștia implementează diverse metode ce nu depind de problema rezolvată, aplicabilitatea lor putând fi limitată doar de reprezentarea aleasă în cazul operatorilor de generare, mutație și încrucișare.

Cel de-al doilea pachet important îl reprezintă ”msrcpsp”, ce conține tot ceea ce ține de problemă și rezolvarea ei. La rădăcina pachetului se află două module ce conțin clase ce modelează informațiile problemei și ale unei soluții la problemă. În subpachetul ”genetic_algorithm” se găsesc toate aspectele referitoare la soluționarea problemei utilizând algoritmi genetici. Modulul ”representation” reunește aspectele de reprezentare pentru rezolvarea problemei. Modulului ”validator” expune funcții pentru validarea reprezentărilor găsite. Modulul ”builder” conține metode ce construiesc algoritmi genetici ce soluționează problema abordată, facilitând asamblarea acestora. ”utils” este un modul utilitar, cu metode de interes comun pentru operatorii genetici, validatori și algoritm în general. Toate implementările concrete pentru operatorii algoritmului se află în subpachetul ”operators”. În general, majoritatea modulelor subpachetului ”msrcpsp.genetic_algorithm” sunt dependente de pachetul ”genetic_algorithm” și de modulele aflate la rădăcina pachetului ”msrcpsp”.

Acestea reprezintă pachetele ce realizează implementarea algoritmului în sine. Pe lângă acestea, în contextul testării, experimentării și utilizării algoritmului au mai fost create 2 pachete. Primul este ”input_utils” ce conține utilitare pentru convertirea fișierelor din formatul de bază în formatul propriu și citirea fișierelor convertite. Pachetul ”scripts” conține scripturi ce rulează codurile de convertire, scripturi pentru executarea algoritmilor genetici, și scripturi dedicate realizării și vizualizării de rapoarte în contextul experimentării.

3.2 Implementarea unui algoritm genetic generic

Pentru a putea implementa un algoritm genetic generic, vom introduce anumite concepte ce vor facilita dezvoltarea și crearea de algoritmi concreți. Vom folosi în continuare ca punct de plecare pseudocodul din figura 2.1 prezentată anterior. Separăm conceptul de candidat de cel de reprezentare. Vom considera candidatul ca fiind format dintr-o reprezentare și fitness-ul acesteia. Cum Python este slab-tipizat, acest lucru permite ca reprezentarea să poată avea orice tip, asigurând un

nivel de flexibilitate ridicat. Introducem și conceptul de context ca fiind ”datele mediului”, altfel spus datele la care se raportează reprezentările atunci când se operează pe ele. Un exemplu ar fi găsirea rădăcinii unei funcții definite pe un interval oarecare. În acest caz, contextul poate conține informații referitoare la funcție și interval. Ultimele două concept introduse sunt cel de tip de problemă - maximizare sau minimizare - și cel de generator de numere aleatoare. Astfel, putem oferi descrieri pentru parametrii ce vor fi utilizați în continuare. Aceștia se regăsesc în figura 3.2, această secțiune de documentație fiind prezentă în modulul ”genetic_algorithm.implementation”.

```
# nc means non-conforming - that means that the method doesn't respect arguments constraints
# the functions should not modify the state of their input arguments, the only exception being Random objects
# __Context__ -> an entity that represents the extra needed data for the problem - can be any type, even none
# __Representation__ -> an entity that represents the chosen representation - also called individual
# __ProblemType__ -> MAXIMIZING_PROBLEM or MINIMIZING_PROBLEM
SAMPLE_ARGS = {
    PROBLEM_TYPE: None,  # either MAXIMIZING_PROBLEM or MINIMIZING_PROBLEM, depending on problem
    POPULATION_SIZE: None,  # the size of the initial population
    REPRODUCTIONS_PER_GENERATION: None,  # the number of reproductions that should be done per generation
    RANDOM: None,  # the random number generator used
    CONTEXT: None,  # the context of the problem - it can be any data type
    GENERATOR_FUN: None,  # generate(Random,__Context__) : __Representation__
    # generates a valid solution representation
    EVALUATOR_FUN: None,  # evaluate(__Representation__,__Context__): number
    # calculates the fitness of the representation
    STOP_CRITERION: None,  # should_stop(GeneticAlgorithm):boolean
    # it returns true if the stop criterion for an algorithm is reached, or false otherwise
    SELECTOR_FUN: None,  # select(list[Candidate],Random,__ProblemType__): list[Candidate]
    # randomly selects some candidates from a pool sorted from best to worst
    CROSSOVER_FUN: None,  # cross(list[__Representation__],Random,__Context__): list[__Representation__]
    # crosses some representations and returns the offsprings
    MUTATOR_FUN: None,  # mutate(__Representation__,Random,__Context__): __Representation__
    # mutates a given representation
    REPLACER_FUN: None  # replace(list[Candidate],list[Candidate],Random,__ProblemType__): list[Candidate] np
    # the first list represents the current population. the second list are the current offsprings. both lists
    # are sorted best to worst. the function returns the candidates that form the new generation.
}
```

Figura 3.2: Argumentele unui algoritm genetic

După cum se observă, funcția de generare returnează o singură reprezentare, fiind responsabilitatea algoritmului de a genera atâtea elemente cât are nevoie. De asemeni, funcția de generare primește ca și parametrii generatorul de numere aleatoare și contextul problemei, același fiind cazul și pentru funcțiile de încrucișare și mutație. Funcția de mutație primește de acum un singur parametru, fiind apelată manual de către algoritm pentru fiecare descendent generat. Funcția de evaluare are ca argumente o reprezentare și un context, returnând fitness-ul reprezentării pentru problemă. Astfel, va fi responsabilitatea algoritmului să creeze candidații din reprezentări și rezultatele evaluărilor. Criteriul de oprire este o funcție booleană ce primește ca argument chiar algoritmul genetic. Astfel, avem acces la datele interne ale algoritmului și putem accesa liber populația sau numărul de generații pentru a

decide dacă este momentul să ne oprim. Funcțiile de selecție și înlocuire primesc pe lângă populațiile de candidați generatorul de numere aleatoare și tipul problemei. Tipul problemei este necesar în cazul nevoii de a comapara în mod direct fitness-urile, pentru a decide când unul este mai bun ca altul. Factorul aleator este necesar selecției pentru a asigura diversitatea populației. În timp ce nu sunt implementați operatori de înlocuire ce utilizează factorul aleator, există posibilitatea de utilizare. Astfel, am optat pentru păstrarea acestuia ca argument.

Se consideră că listele de candidați, anume populația și argumentele funcțiilor de selecție și înlocuire, sunt ordonate astfel încât primele soluții sunt cele mai bune, acest lucru fiind asigurat de către algoritm.

În continuare vom discuta despre alte aspecte de notat, sau practici bune, atunci când se dezvoltă un algoritm utilizând schema oferită. În contextul implementării algoritmului, este un stil de lucru bun ca funcțiile utilizate să fie pure. Asta înseamnă că argumentele de intrare nu-și schimbă starea în urma execuției, iar același set de date de intrare generează același rezultat. Singura excepție de la această regulă o prezintă factorul aleator, care la fiecare apel de generare își va schimba starea. Este de notat că rămâne validă consistența datelor de ieșire pentru aceleași date de intrare, pentru aceiași stare internă generatorul aleator returnând același set de valori. O convenție de numire pentru fiecare funcție este utilizarea denumirii din documentație sufixată de o scurtă descriere, cum ar fi "cross_lists_one_point". În cazul în care funcția nu respectă signatura oferită, utilizarea aceasta fiind de fapt făcută prin apelul ca delegat de către alt operator, metoda poate fi prefixată cu "nc" de la "non-confirming".

Există trei metode principale prin care s-ar putea implementa un operator. O primă modalitate este implementarea directă a funcției și trimiterea acesteia mai departe ca parametru. Acest lucru este posibil datorită faptului că inclusiv funcțiile sunt obiecte în Python. Totuși, există cazuri în care ar fi favorabilă utilizarea unei clase. Spre exemplu, dacă dorim să avem un criteriu generic de oprire a cărui condiție de stop este atingerea unei durate parametrizabile. Astfel, durata ar putea fi parametru al clasei. Clasa va conține o metodă ce respectă contractul dat de documentație, iar ca parametru se trimite referință la funcția obiectului instațiat. Există totuși o alternativă mai simplă, dată de utilizarea unor funcții ce pot returna alte funcții. Funcția principală primește aceleași argumente ce le-ar primi și constructorul clasei. Putem definii în interiorul funcției principale o funcție ce respectă contractul operatorului ce dorim să-l implementăm. Parametrii dați ca argument funcției de creare pot fi folosiți de către funcția interioară asemeni unor variabile statice. Returnând funcția interioară, obținem un procedeu de a crea funcții parametrizabile ce respectă contractul operatorilor. Cum în general singurul scop al operatorilor este să fie apelați, neavând alte funcționalități, acest procedeu poate fi avantajos

pentru a dezvolta mai rapid algoritmii. Această metodă este foarte asemănătoare cu modul în care limbajul React funcționează. Un exemplu poate fi văzut în figura 3.3. Cum decizia aleatoare de a realiza o mutație sau nu este prezentă în toți algoritmii genetici, am separat procesul de decizie de procesul de mutație propriu-zisă. Astfel, se pot crea operatori de mutație fără a adăuga în aceștia partea de decizie, decorând ulterior operatorii prin apelul la această funcție.

```
# creates a function that applies a mutation with a given probability
def create_probabilistic_mutator_function(mutator_function, mutation_probability: float):
    def mutate(candidate, random: Random, context):
        if random.random() < mutation_probability:
            return mutator_function(candidate, random, context)
        return candidate

    return mutate
```

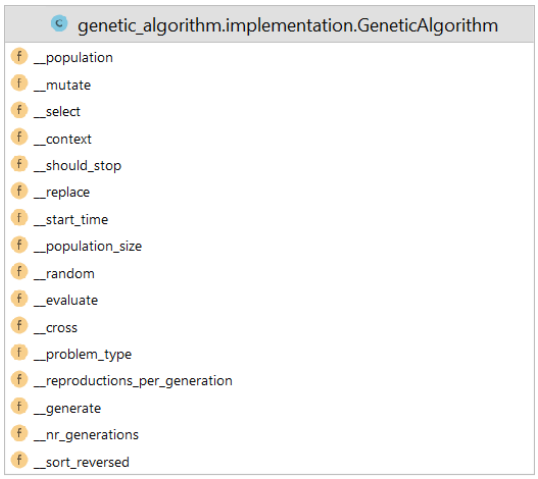
Figura 3.3: Creator de funcție de mutație probabilistică

Utilizând principiile menționate anterior, am realizat atât implementarea unor operatori de bază, cei din pachetul ”genetic_algorithm.operators”, cât și implementarea operatorilor specifici problemei studiate.

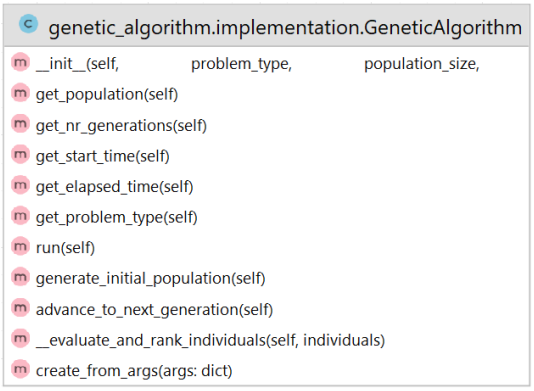
Cum algoritmul genetic este o entitatea complexă cu numeroase câmpuri parmterizabile, vom încapsula comportamentul algoritmului într-o clasă a cărei structură poate fi văzută în figura 3.4. Parametrii primiți de constructorul obiectelor de această clasă sunt cei menționați în figura 3.2. În timp ce algoritmii pot fi creați prin intermediul constructorului prin trimiterea manuală a fiecărui argument, este mai recomandată utilizarea metodei statice ”create_from_args” ce primește ca parametru un dicționar făcut pe structura prezentată anterior. Totodată, există metodă ce permite crearea unui astfel de dicționar în locul unui algoritm genetic, având aceiași signatură. Această abordare permite o flexibilitate mai mare, putându-se face metode ce generează astfel de dicționare în locul algoritmilor genetici, ulterior modificându-se parametrii dupa propria nevoie. Spre exemplu, se poate face o metodă ce generează argumentele unui algoritm ce rezolvă o problemă. Pentru a măsura performanța operatorului de evaluare, se înlocuiește metoda de evaluare cu una decorată ce scrie într-un fișier durata de execuție a fiecărei evaluări. Dacă metoda ar fi returnat în mod direct algoritmul, nu ar fi fost posibilă această modificare deoarece câmpul ce conține metoda este privat și neaccesabil în alt mod.

Vom prezenta parametrii ce nu corespund direct unor elemente din dicționar. ”__start_time” reprezintă momentul în care algoritmul a fost pornit, anume momentul de apel al metodei de inițializare. ”__nr_generations” corespunde numărului de generații trecute de la prima generație, iar ”__population” corespunde populației curente. Considerând generațiile indexate de la 0, ”__nr_generations” poate fi privit ca indexul generației curente. ”__sort_reversed” este un câmp ajutător, utilizat pentru a determina ordinea elementelor în cadrul populației. Acesta este determinat de

valoarea câmpului "__problem_type".



(a) Câmpuri



(b) Metode

Figura 3.4: Clasa de algoritm genetic

Printre funcții, se observă prezența unor metode de accesare a anumitor câmpuri de interes, cum ar fi populația și numărul de generații. De asemeni am inclus o metodă ce returnează timpul trecut de la pornirea algoritmului. Acestea sunt de ajutor în implementarea criteriilor de oprire, precum cel generațional sau constrâns de timp. În timp ce rularea algoritmului se poate face prin apelul metodei "run", se poate realiza și o rulare manuală prin apelul metodei de generare a populației inițiale și apelul repetat al funcției de avansare a generației. Acestea sunt metode publice pentru a permite un mai bun control asupra algoritmului, în cazul în care utilizarea criteriului de oprire nu este o opțiune. Un astfel de exemplu este oprirea algoritmului pe parcurs și continuarea sa într-un moment mai târziu în cadrul unei aplicații la linia de comandă.

3.3 Implementarea algoritmilor asociați problemei

În continuare, toate clasele utilizate au ca scop strict reunirea mai multor date, din acest motiv câmpurile lor fiind publice și neconținând metode. Realizăm clase pentru a reprezenta datele problemei și soluțiile acesteia, structura acestora fiind independentă de metoda de soluționare. Cum a fost explicat anterior, soluția de bază reprezintă o mapare a activităților relativ la timpii de start, în timp ce în contextul algoritmilor genetici soluția va fi o listă de activități. Astfel, separăm problema în sine de modul ei de rezolvare. Clasele pentru aceste entități sunt la rădăcina pachetului "msrcpsp", în modulele aferente. În timp ce clasele deja definite ar oferi îndeajunsă informație pentru a deduce restul valorilor specifice problemei, precum care sunt resursele ce stăpânesc o anumită calificare sau care sunt timpii de terminare în ca-

drul soluției, observăm că ar fi în avantajul oricărei rezolvări a problemei accesul direct la aceste informații. Astfel, pe lângă entitățile de bază există entități soră ce sunt mai bogate ca informație. În aceleași module sunt metode pentru transformarea entităților de bază în entități detaliate. Diagramele pentru clasele menționate pot fi vizualizate în figurile 3.5 și 3.6 .

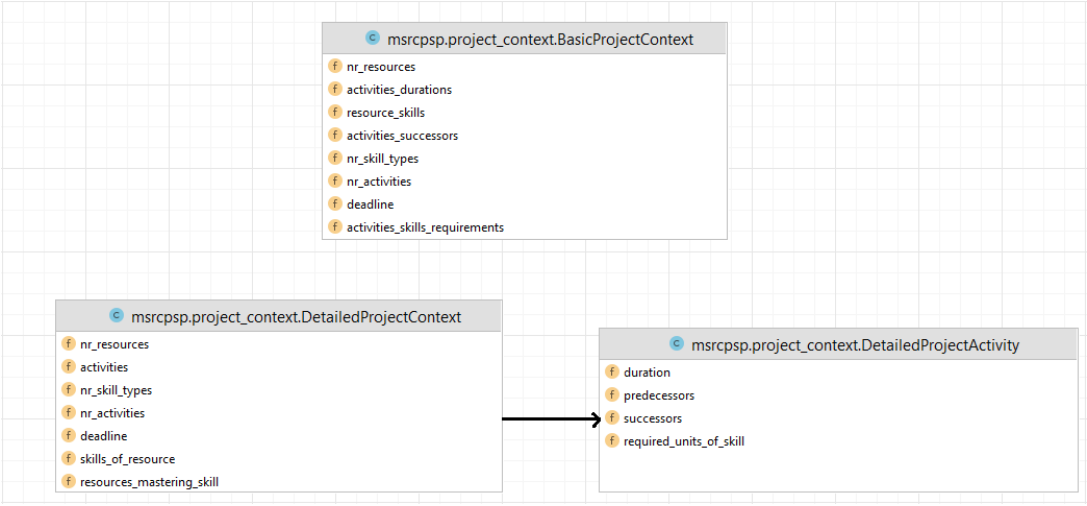


Figura 3.5: Clase pentru contextul problemei

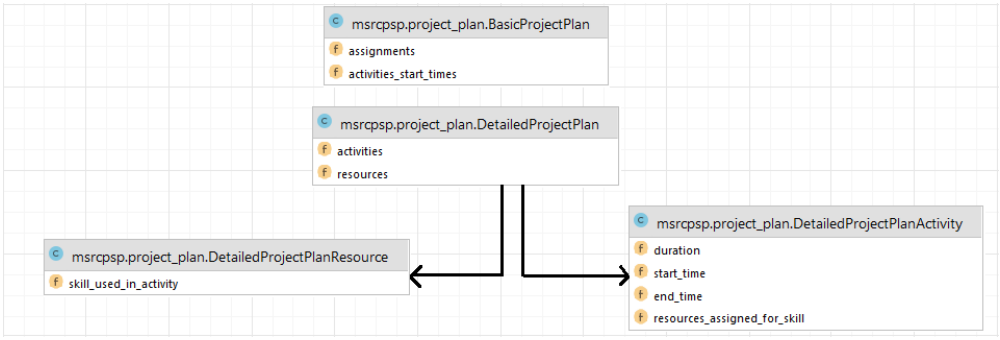


Figura 3.6: Clase pentru soluția problemei

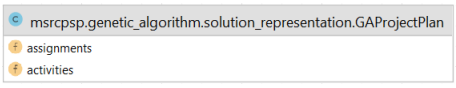


Figura 3.7: Clasă pentru reprezentarea soluției în algoritm genetic

În continuare, avem nevoie de reprezentarea unei soluții. Cum singura diferență dintre cele două reprezentări studiate anterior este la nivelul interpretării listei de activități, putem folosi o singură clasă pentru amândouă. Transformarea reprezentărilor din ordine în prioritate este necesară evaluării și validării soluțiilor, astfel metoda ce realizează acest lucru se află în modulul de utilitare. Tot aici se găsesc metode de transformare a soluțiilor din reprezentarea asociată unui algoritm genetic în reprezentarea de bază prezentată anterior în figura 3.6.

Cum reprezentarea soluției poate fi privită ca două reprezentări separate, cea de listă de activități și cea de listă de ordine, pentru implementarea operatorilor genetici am optat pentru realizarea de funcții ce operează separat peste cele două. Astfel decuplăm operatorii aplicați reprezentării principale de operatorii aplicați reprezentărilor primitve, oferind un grad mai mare de reutilizabilitate. Asemeni modului de lucru prezentat în capitolul anterior, în unele cazuri vom folosi funcții ce crează metodele necesare. Spre exemplu, în cazul operatorului de mutație vom utiliza o funcție ce primește ca parametrii metodele de mutație aplicate peste activități, respectiv alocări. Această funcție poate fi văzută în figura 3.8. Modulul "validator" conține metode pentru validarea reprezentărilor. Acestea validează separat listele de activități și listele de alocări, astfel că putem aplica un procedeu ca cel descris anterior pentru crearea validatorilor.

```
def create_mutator_function(activities_mutator_function, assignments_mutator_function):
    def mutate(individual: GAProjectPlan, random: Random, project_context: DetailedProjectContext) -> GAProjectPlan:
        result = GAProjectPlan()
        result.activities = activities_mutator_function(individual.activities, random, project_context)
        result.assignments = assignments_mutator_function(individual.assignments, random, project_context)
        return result

    return mutate
```

Figura 3.8: Funcție constructor pentru operator de mutație

Generarea unui algoritm genetic ce rezolvă o problemă de tip MSRCPSP se realizează cu ajutorul modulului "builder". În timp ce crearea de algoritm este personalizabilă, sunt aleși ca parametrii default valorile pentru care s-a observat cea mai bună performanță pentru algoritmi în general. Construcția de algoritm permite și injectarea de validatori în metodele ce crează și modifică reprezentările soluțiilor prin setarea unui parametru de tip boolean. Validarea a fost utilizată doar pentru a crește nivelul de certitudine că implementările operatorilor nu au fost greșite, astfel că au fost folosite doar într-un context restrâns. Validarea nu a fost făcută în contextul experimental, acest proces putând impacta negativ performanța.

3.4 Date de intrare, preprocesare, rapoarte, rulare manuală

Datele de intrare se iau din fișiere de format "msrcp" despre care vom discuta mai mult în capitolul de experimentare. Acestea conțin numeroase date redundante problemei noastre, astfel că am propus un alt format de fișier asemănător cu cel inițial. Descrieri amănunțite ale acestor două formate se pot găsi în folderul "data/problem_input_data". În pachetul "input_utils" se găsește un modul cu o metodă de conversie ce primește ca parametrii un nume de fișier sursă de format "msrcp" și un nume de fișier destinație pentru conversie. Tot în același pachet se află

un modul cu o metodă pentru citirea fișierelor de format propriu ce returnează un obiect de tip BasicProjectContext. Acesta poate fi ulterior convertit într-un context detaliat ce este pasat algoritmului în sine. Un script ce rulează aceste conversii se află în modulul "run_files_conversion" din pachetul "scripts". În continuare, rularea algoritmilor se face folosind fișierele convertite ca date de intrare.

Logica pentru rapoarte se află în pachetul "scripts.reports". Cum Python realizează anumite cache-uri pentru rezultatele obținute, o a doua rulare a algoritmului va fi accelerată, obținând evident rezultate mai bune. Astfel, am decis că în contextul creării de rapoarte de bază, rularea unui algoritm se face fără a exista alte rulări precedente ale algoritmilor. Pentru executarea unui caz de testare se rulează modulul "run_single_test_case", trimițând ca argumente la linia de comandă fișierul de input, fișierul de output și reprezentarea aleasă. Un raport va fi reprezentat de un fișier de format "csv" ce conține anumite caracteristici ale fiecărei generații. Pentru a realiza aceste rapoarte, am decorat criteriul de oprire deja existent astfel încât acesta să salveze în fișier datele la nivelul fiecărei generații.

Modulul "run_test_cases" este responsabil de generarea tuturor rapoartelor de bază necesare. Acesta rulează pentru fiecare set de date de intrare ambele implementări de algoritmi, fiecare de câte un număr de ori dat. Pentru a evita apariția pe parcurs a accelerării, realizarea de rapoarte este delegată modulului prezentat anterior prin lansarea unui proces separat ce îl rulează.

Vizualizarea rapoartelor se poate face prin rularea modulului "create_reports". Acesta se folosește de fișierele "csv" generate anterior pentru a forma grafice ce ajută în a pune în perspectivă diferite aspecte ale fiecărei rulări. Totodată, aici se pot prelucra rapoartele de bază pentru a obține unele noi, cu informații derivate din cele inițiale. Pentru implementarea acestui modul s-au folosit librăriile "pandas" și "matplotlib".

Pachetul "scripts.genetic_algorithm" conține două module pentru rularea manuală a algoritmilor genetici. Unul dintre acestea conține o implementare a unei probleme simple, având ca unic scop validarea deciziilor de proiectare luate pe parcurs. Cel de-al doilea modul rulează algoritmii genetici implementați pentru MSRCPSP, acesta fiind utilizat pentru a studia comportamentul înainte de faza experimentală.

Capitolul 4

Rezultate experimentale

În contextul experimentării, am folosit cazuri de testare dintr-un set de benchmark generat de Snauwaert și Vanhoucke în lucrarea [SV23]. Aceste teste se află într-un format special de fișier, "msrcp", conținând îndeajunse informații astfel încât să poată fi utilizate și pentru alte extensii ale problemei. Cum a fost precizat anterior, acestea au fost convertite într-un format propriu ce conține strict informațiile necesare problemei abordate. Pe lângă acestea se regăsește și un optim aflat prin rularea unui program CPLEX pentru o durată de 60 de secunde chiar pentru problema noastră, acesta fiind inclus de asemenea în fișierul inițial. În timp ce ar fi fost de preferat rularea programului CPLEX pe propriul dispozitiv, acest lucru nu poate fi realizat din cauza absenței surselor programului cât și a lipsei de experiență în utilizarea CPLEX. Vom folosi totuși acest optim ca punct de reper pentru calitatea soluțiilor obținute de algoritmii noștrii, focusându-ne totodată pe compararea reciprocă a acestora.

Pentru rularea testelor, am optat pentru o populație de 300 de indivizi cu 145 de reproduceri pe generație pentru un număr total de 290 de descendenți. Cum metoda de înlocuire este de tip steady-state, asta înseamnă că cele mai bune 10 soluții vor rămâne de la o generație la alta. Vom alege ca și criteriu de oprire trecerea unei durate de 60 de secunde, asemănător cu soluția CPLEX. Probabilitățile de mutație alese vor fi de 0.1 pentru lista de activități și 0.25 pentru lista de asignări de resurse. Cum a fost precizat anterior în 2.3.3, este de preferat ca probabilitatea de mutație pentru listele de asignări de resurse să fie mai mare, astfel valoarea aleasă. Pentru generatorul de numere aleatoare am utilizat un seed pentru care s-a observat obținerea unor rezultate favorabile în general. Parametrii propuși au fost aleși atât din considerente teoretice, cât și printr-o experimentare manuală.

Rularea testelor va produce fișiere de format "csv" ce conțin pe fiecare linie numărul generației curente, momentul în timp la care a fost atinsă generația raportat la începerea execuției algoritmului, cel mai bun fitness respectiv fitness-ul mediu al generației curente. Se merită menționat cum, ignorând criteriul de oprire

dat de depășirea timpului limită, algoritmul este determinist, seed-ul generatorului de numere aleatoare fiind fixat înainte de începerea rulării. Astfel, în contextul aceluiași test, prin realizarea a două rulări cu același algoritm se vor genera pentru fiecare generație același cel mai bun fitness și fitness mediu. Diferența va fi dată de timpii la care generațiile sunt atinse și numărul total de generații parcurse. Din acest considerent, vom utiliza numărul de generații pentru a măsura performanța computațională a algoritmilor.

Din setul de date propus am ales 10 cazuri astfel încât să putem ilustra atât situații favorabile, cât și nefavorabile asociate algoritmilor noștrii. Testele alese au un număr de 32 de activități, 4 calificări, iar numărul de resurse este variabil, primele 5 teste având câte 6-7 resurse, iar ultimele 5 câte 10-11 resurse. Vom rula algoritmii pentru fiecare caz în parte de câte 10 ori pentru a putea determina numărul mediu de generații și deviația standard a acestora. Specificațiile dispozitivului utilizat pot fi vizualizate în tabelul 4.1, iar rezultatele obținute pot în tabelul 4.2.

Processor	11th Gen Intel(R) Core(TM) i7-11800H 2.30GHz 2.30 GHz
RAM	16,0 GB (15,7 GB usable)
OS	Windows 10 Pro Version 22H2
System type	64-bit operating system, x64-based processor

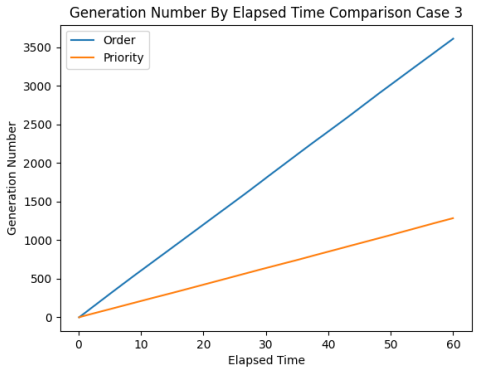
Tabela 4.1: Specificațiile dispozitivului

Test Case	Order	Priority
Case 1	3818.9±23.784238478454597	2106.3±18.264993840677857
Case 2	3713.1±25.508625992005136	1664.6±11.884443613396463
Case 3	3609.5±8.868483523128404	1296.2±7.249827584156742
Case 4	3711.1±23.3471625685007	2360.5±8.261355820929152
Case 5	3729.5±16.835973390332974	2182.4±11.119352499134111
Case 6	3707.1±20.225973400556025	1688.6±11.038115781237304
Case 7	3722.1±18.76406139405859	1743.3±10.964032105024135
Case 8	3689.1±20.724140512938042	1517.8±5.844655678480984
Case 9	3608.6±15.723867208800765	1676.7±7.988116173416608
Case 10	3631.2±19.701776569639602	1558.0±6.0332412515993425

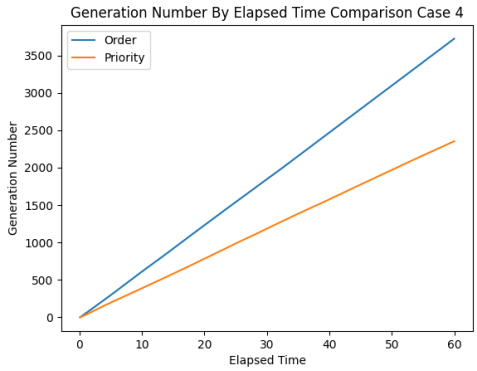
Tabela 4.2: Numărul mediu de generații și deviația standard pentru rularea fiecărui caz de testare

Se observă cum reprezentarea sub formă de listă de ordine permite un număr de generații mediu mai mare ca reprezentarea ca listă de priorități. Acest fapt este cauzat de faptul că evaluarea unei liste de priorități presupune transformarea acesteia în listă de ordine. Se observă totodată cum, în general, listele de ordine au o deviație standard mai mare ca listele de priorități. O explicație ar fi faptul că, permițând

un număr mai mare de generații într-o anumită perioadă de timp, cu cât procesul este mai mult pus în așteptare de către procesor, impactul asupra numărului total de generații va fi mai mare spre deosebire de lista de priorități. Relativ la cazurile de testare, se observă cum, crescând numărul de resurse, scade numărul mediu de generații pentru ambele reprezentări. Altă particularitate este reprezentată de varierea numărului de generații între cazurile de testare. În timp ce reprezentarea ca listă de ordine pare să aibă numărul de generații cuprins între 3600 și 3900, reprezentarea de listă de priorități variază de la 1200 până la 2400. O explicație ar fi în continuare modul în care este realizată evaluarea soluțiilor în sine, performanța transformării unei liste de priorități în listă de ordine fiind afectată de contextul problemei, mai exact dependențele dintre activități. O reprezentare grafică a numărului de generații parcurse după o anumită perioadă de timp în anumite cazuri poate fi văzut în figura 4.1. În timp ce în general panta pentru lista de ordine este aproximativ aceiași în toate cazurile, pentru lista de priorități variază. Panta poate fi privită ca un mod de a măsura rata de creștere a numărului de generații în raport cu timpul.



(a) Cazul 3



(b) Cazul 4

Figura 4.1: Numărul de generații parcurse după o anumită perioadă de timp

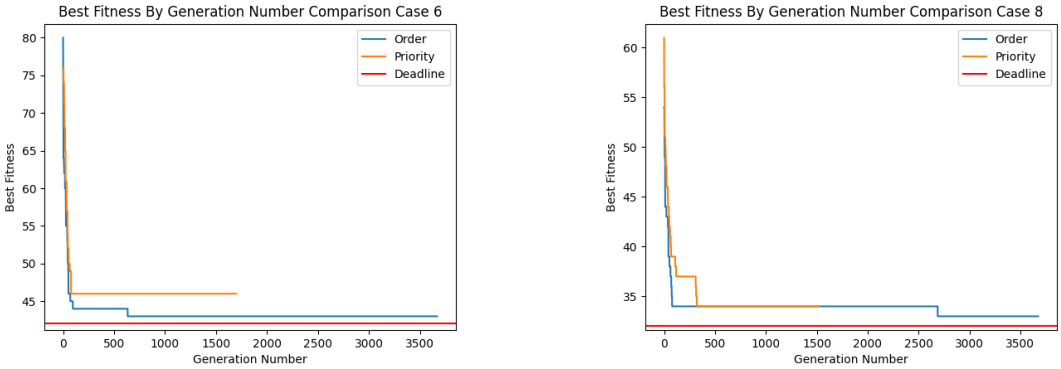
Utilizând tabelul prezentat anterior 4.2 putem alege rulări reprezentative pentru fiecare caz de testare și algoritm, utilizându-le pe cele a căror număr de generații se află în intervalele de încredere aferente. Acest fapt se datorează observației făcute inițial despre determinismul rulărilor din punct de vedere al rezultatelor numerice obținute pentru valorile de fitness. Astfel, valorile utilizate pentru comparație se găsesc în tabelul 4.3.

Pentru cazurile 1, 2 și 9 se observă cum algoritmii obțin rezultate considerabil mai bune ca cele oferite de CPLEX. În același timp, în cazuri precum 6 și 8, algoritmii nu reușesc să atingă țintele. Analizând graficele 4.2 pare că în anumite cazuri aceștia nu s-ar fi blocat în minime locale, deci există posibilitatea de îmbunătățire prin alocarea a mai mult timp. În anumite situații, unul din algoritmi dă greși, iar celălalt reușește să depășească limita propusă. Cazul 7 ilustrează cum lista de ordine

obține un nou minim, dar lista de priorități eșuează. În cazul 4, lista de priorități depășește ținta, în timp ce lista de ordine nu reușește decât să o atingă.

Test Case	Target Value	Order Best Fitness	Priority Best Fitness	Order Avg. Fitness	Priority Avg. Fitness
Case 1	78.000000	67.000000	67.000000	69.603333	69.010000
Case 2	84.000000	78.000000	79.000000	79.786667	80.923333
Case 3	70.000000	70.000000	72.000000	71.383333	73.310000
Case 4	76.000000	76.000000	72.000000	77.806667	75.000000
Case 5	71.000000	64.000000	68.000000	65.686667	70.346667
Case 6	42.000000	43.000000	46.000000	44.790000	48.200000
Case 7	46.000000	46.000000	50.000000	47.810000	53.020000
Case 8	32.000000	33.000000	34.000000	34.550000	36.280000
Case 9	70.000000	59.000000	58.000000	60.733333	60.120000
Case 10	57.000000	56.000000	57.000000	58.020000	59.943333

Tabela 4.3: Fitness-uri reprezentative generațiilor finale fiecărui caz de testare



(a) Cazul 6

(b) Cazul 8

Figura 4.2: Evoluția celui mai bun fitness o dată cu generațiile

Comparând fitness-uril relativ la numărul generației, putem valorifica calitatea unei iterații a celor doi algoritmi. Totuși, din punct de vedere practic ne interesează mai mult valorile obținute după trecerea unei anumite perioade de timp. În general, în ambele cazuri se observă că pentru același număr de generații iterate, respectiv durată de timp trecută, se obțin rezultate mai bune prin utilizarea listelor de ordine. În figurile 4.3 4.4 se observă evoluția în timp a fitness-ului mediu pentru cazuri în care domină atât lista de priorități, cât și lista de ordine. În cazul 4 lista de priorități pierde din fitness-ul mediu apoi recapătă avantajul, sugerând ieșirea dintr-un minim local ce a condus la găsirea ulterioară a unor soluții mai bune. O situație asemănătoare este reprezentată în figura 4.5 ascoiată cazului 7, unde inițial în avantaj se află lista de priorități, ulterior fiind depășita semnificativ de lista de ordine. În general, lista de ordine aduce rezultate mai bune ca lista de priorități

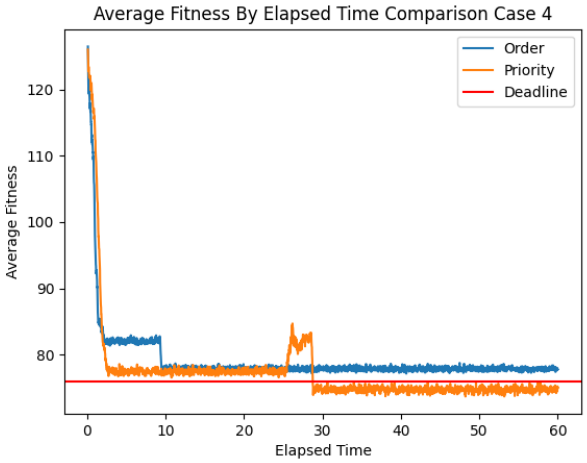


Figura 4.3: Evoluția fitness-ului mediu o dată cu timpul în cazul 4

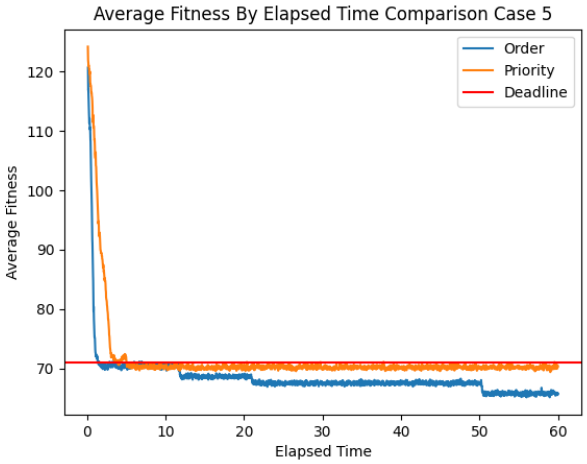


Figura 4.4: Evoluția fitness-ului mediu o dată cu timpul în cazul 5

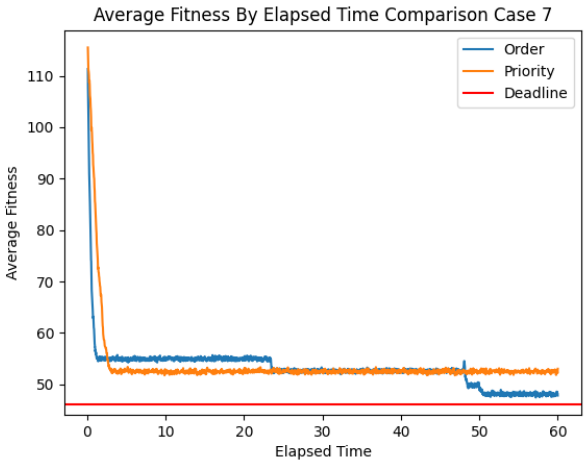


Figura 4.5: Evoluția fitness-ului mediu o dată cu timpul în cazul 7

Capitolul 5

Aplicație practică

Cum am verificat că algoritmii propuși ne oferă soluții satisfăcătoare, aceștia pot fi integrați la nivelul unei aplicații. În introducere am prezentat cum problema de planificare a apărut ca urmare a unor nevoi reale de a reduce timpii de execuție a proiectelor de diferite tipuri. Persoanele cele mai interesate de rezolvarea unei asemenea probleme ar fi cele responsabile cu planificarea proiectelor, în general managerii de proiect. Vom denumi soluția propusă "Telescope Planner".

Cum algoritmii genetici necesită multă putere computațională, aceștia ar trebui rulați pe dispozitive performante ce nu sunt neapărat la îndemâna utilizatorului obișnuit. Algoritmul poate rula pe un server remote, comunicarea cu acesta făcându-se prin intermediul unui API. API-ul expune un endpoint prin care primește cereri de a găsi o planificare adecvată pentru un anumit context de proiect dat. Cum modul de a reprezenta structura unui proiect poate varia inclusiv în cadrul unei firme, formatul unei cereri va fi unul simplist, considerându-se că activitățile sunt numere de la 0 la $n-1$, unde n este numărul de activități. Aceiași logică este aplicată și pentru resurse și calificări. Totodată, o cerere conține un timp limită de execuție măsurat în secunde. Nu vom oferi alți parametrii de rulare din considerentul că din exterior nu este cunoscută implementarea, astfel, spre exemplu, ideea de reprezentare a soluției nu are sens pentru cel ce apelează API-ul. Aceste decizii ar trebui făcute intern, la nivelul serverului. Cererile vor fi trimise ca json, comunicarea făcându-se prin protocolul HTTP, cu un apel POST la endpoint-ul "/plans". Un exemplu de cerere poate fi văzut în figura 5.1. În sursele oferite am inclus și o colecție postman ce conține un astfel de exemplu la API.

Serverul este proiectat pe o arhitectură stratificată simplă cu 2 straturi. Primul strat este reprezentat de controller-ul ce primește cererile sub forma unor obiecte DTO. Acesta conține un validator pentru cereri utilizat la fiecare apel al API-ului și o referință la serviciul de planificare. Controler-ul transformă obiectele DTO în obiecte de domeniu, trimițându-le mai departe la serviciu. Obiectele returnate de serviciu sunt folosite pentru a crea DTO-uri ce sunt trimise ca răspuns. Serviciul de

```
{
    "time_limit": 1,
    "project_context": {
        "nr_skills": 4,
        "resources": [
            {
                "skills": [
                    0
                ]
            },
            {
                "skills": [
                    1,
                    2
                ]
            },
            {
                "skills": [
                    2
                ]
            }
        ],
        "activities": [
            {
                "duration": 5,
                "successors": [],
                "skill_requirements": {}
            },
            {
                "duration": 10,
                "successors": [
                    0
                ],
                "skill_requirements": {
                    "2": 2
                }
            }
        ]
    }
}
```


Figura 5.1: Corpul unei cereri la API

planificare reprezintă cel de-al doilea strat, aici realizându-se în sine rulările algoritmului genetic. Pentru fiecare cerere se realizează un nou algoritm genetic, fiecare cu parametrii pentru care se știe că rezultatele obținute sunt cele mai optime.

Implementarea acestui server a fost realizată în Python, utilizând FastAPI pentru controller și comunicare și uvicorn pentru rularea serverului. Datorită modului de funcționare a FastAPI, controller-ul este reprezentat chiar de modulul ce pornește serverul în sine. Astfel, serviciul și validatorul sunt obiecte instațiate la nivelul modulului. Abstractizând modulul ce rulează serverul ca și controller, obținem arhitectura dată de figura 5.2. Serverul implementat este concurent, astfel se pot realiza mai multe cereri deodată.

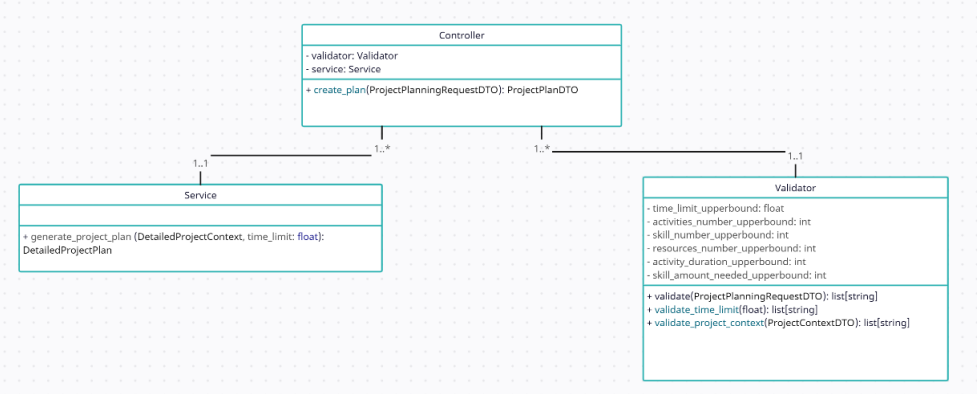


Figura 5.2: Arhitectura serverului

În continuare, vom implementa un client de tip CLI ce să apeleze acest API. Cum aplicația este la nivel de consolă, accesarea feature-urilor se face prin introducerea de comenzi. Lista de comenzi poate fi vizualizată în interiorul aplicației. Feature-ul principal oferit este cel de generare de planificare. Utilizatorul introduce o cale către un fișier de același format propriu cu care s-a lucrat pe parcursul lucrării. Acesta introduce calea către un fișier destinație unde va fi salvată planificarea și durata de timp pe care este dispus să o aștepte pentru realizarea planificării. O dată trimisă cererea, aplicația citește datele din fișierul sursă și le trimite mai departe la server. Serverul răspunde cu planificarea ce este salvată în fișier, utilizatorul fiind anunțat de succesul operației. În cazul în care datele oferite sunt invalide, utilizatorului i se afișează un mesaj de eroare pentru a-l anunța de acest lucru.

Cum majoritatea fișierelor pentru problemele de acest tip sunt de format "msrcp", am introdus o funcționalitate ce permite convertirea unui fișier din formatul inițial în cel propiu. Utilizatorul introduce calea pentru fișierul sursă și fișierul destinație. O dată trimisă cererea, aplicația realizează conversiile necesare și înștiințează utilizatorul legat de succesul operației. În cazul în care datele sunt eronate, utilizatorul este informat de acest lucru.

Vom utiliza de asemeni o arhitectură stratificată pe 2 nivele, simplitatea aplicației permițând acest lucru. Nivelul superior este reprezentat de interfața CLI, aceasta primind datele și comenzile de la utilizator. Interfața comunică cu un serviciu ce conține două metode publice pentru a oferi funcționalitățile menționate. Serviciul mai departe poate comunica cu serverul pentru rezolvarea problemei de planificare. Cum cele două programe trebuie să cunoască pentru comunicare atât tipul de mesaj trimis cât și tipul de mesaj primit, clasele obiectelor de tip DTO sunt incluse într-un pachet pentru module comune. Ca și librării au fost folosite "requests" pentru cererile HTTP și fastapi pentru conversiile din DTO-uri în json și vice-versa.

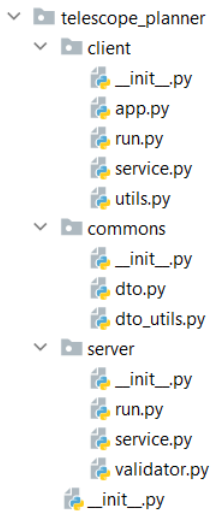


Figura 5.3: Structurarea pachetelor și modulelor

Programele au fost dezvoltate în pachete separate, neexistând dependențe între ele. Cum a fost menționat anterior, comunicarea a fost izolată într-un pachet separat unde ar fi puse toate elemente comune. Modulele pentru convertiri de DTO-uri comunică cu modulele ce conțin reprezentările datelor problemei, astfel acestea devenind o dependență și pentru pachetele client și server. Pachetul server comunică cu pachetul ce conține rezolvarea prin algoritmi genetici a problemei, iar pachetul client comunică cu pachetul de utilitare pentru citirea datelor din fișiere. Structura pachetelor și a modulelor poate fi văzută în figura 5.3.

Capitolul 6

Concluzii și discuții

În această lucrare am analizat o problemă de tip MSRCPSP, prelucrând-o astfel încât să o putem rezolva utilizând algoritmi genetici. În acest context am oferit două reprezentări pentru soluții, una standard și una nou propusă, pentru care am oferit abordări genetice concrete. Testând experimental implementările acestor algoritmi, am obținut rezultate satisfăcătoare, reușind în general să depășim sau să atingem anumite fitness-uri țintă. Comparând rezultate s-a observat faptul că reprezentarea standard este mai avantajoasă în majoritatea situațiilor propuse. Totuși, în general, reprezentarea nouă obține soluții aproape la fel de bune, existând inclusiv cazuri în care reușește să fie mai performantă ca reprezentarea standard. Astfel, ambele soluții reprezintă modalități valide de a rezolva problema.

În subsecțiunea 2.3.1 am propus reprezentările și am discutat despre cum acestea acoperă spațiul soluțiilor. Motivul pentru care am decis să continuăm cu listele de priorități este modul în care acestea acoperă spațiul mai uniform în anumite cazuri. Astfel, se merită studiată distribuția soluțiilor relativ la spațiile de reprezentare, putându-se descoperi criterii după care să se decidă înainte de rularea algoritmului care reprezentare ar fi mai optimă.

Așa cum a fost menționat când au fost propuși operatorii genetici 2.3.3, datorită procesului de transformare din listă de priorități în listă de ordine, există posibilitatea de a oferi noi operatori genetici pentru listele de ordine. Aceștia ar putea conduce mai departe la noi soluții mai performante decât cele oferite până acum, astfel îmbunătățind soluția deja existentă. Totodată, se poate încerca reprezentarea asignărilor de resurse utilizând listele de priorități. Combinarea acesteia cu noua reprezentare propusă pentru listele de activități ar oferi un grad de flexibilitate mai mare în cadrul reprezentării unei soluții Cum listele de priorități pot fi considerate generalizări ale listelor de ordine, anumite proprietăți descoperite pentru listele de proprietăți ar putea fi transpuse mai departe în listele de ordine. Astfel, o analiză mai amănunțită a listelor de priorități ar fi în general benefică, oferind o nouă perspectivă asupra problemei.

Bibliografie

[AESEM20] A. Ayoub, M. El-Shorbagy, Islam Eldesoky, and A. Mousa. *Cell Blood Image Segmentation Based on Genetic Algorithm*, pages 564–573. 03 2020.

[AF07] Enrique Alba and J. Francisco Chicano. Software project management with gas. *Information Sciences*, 177(11):2380–2401, 2007.

[BLK83] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[BN04] Odile Bellenguez and Emmanuel Néron. Lower bounds for the multi-skill project scheduling problem with hierarchical levels of skills. In *International conference on the practice and theory of automated timetabling*, pages 229–243. Springer, 2004.

[BN07] Odile Bellenguez and Emmanuel Néron. A branch-and-bound method for solving multi-skill project scheduling problem. *RAIRO - Operations Research*, 41:155 – 170, 04 2007.

[CZ13] Wei-Neng Chen and Jun Zhang. Ant colony optimization for software project scheduling and staffing with an event-based scheduler. *IEEE Transactions on Software Engineering*, 39(1):1–17, 2013.

[DB08] L.-E. Drezet and J.-C. Billaut. A project scheduling problem with labour constraints and time-dependent activities requirements. *International Journal of Production Economics*, 112(1):217–225, 2008. Special Section on Recent Developments in the Design, Control, Planning and Scheduling of Productive Systems.

[HB10] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010.

[HD98] Sönke Hartmann and Andreas Drexl. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks*, 32(4):283–297, 1998.

[Hol92] John H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992.

[HSEC00] Tarek Hegazy, Abdul Shabeeb, Emad Elbeltagi, and Tariq Cheema. Algorithm for scheduling with multiskilled constrained resources. *Journal of Construction Engineering and Management-asce - J CONSTR ENG MANAGE-ASCE*, 126, 12 2000.

[KW59] James E. Kelley and Morgan R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), page 160–173, New York, NY, USA, 1959. Association for Computing Machinery.

[MANN16] Hamidreza Maghsoudlou, Behrouz Afshar-Nadjafi, and Seyed Taghi Akhavan Niaki. A multi-objective invasive weeds optimization algorithm for solving multi-skill multi-mode resource constrained project scheduling problem. *Computers & Chemical Engineering*, 88:157–169, 2016.

[MQAV16] Usama Mehboob, Junaid Qadir, Salman Ali, and Athanasios Vasilakos. Genetic algorithms in wireless networking: Techniques, applications, and issues. *Soft Computing*, 20, 06 2016.

[MRCF59] D. G. Malcolm, J. H. Roseboom, C. E. Clark, and W. Fazar. Application of a technique for research and development program evaluation. *Operations Research*, 7(5):646–669, 1959.

[Nér02] Emmanuel Néron. Lower bounds for the multi-skill project scheduling problem. In *Proceeding of the eighth international workshop on project management and scheduling*, pages 274–277. Citeseer, 2002.

[SV21] Jakob Snauwaert and Mario Vanhoucke. A new algorithm for resource-constrained project scheduling with breadth and depth of skills. *European Journal of Operational Research*, 292(1):43–59, 2021.

[SV23] Jakob Snauwaert and Mario Vanhoucke. A classification and new benchmark instances for the multi-skilled resource-constrained project scheduling problem. *European Journal of Operational Research*, 307(1):1–19, 2023.

[Wie64] Jerome D. Wiest. Some properties of schedules for large projects with limited resources. *Operations Research*, 12(3):395–418, 1964.

[YA11] Virginia Yannibelli and Analía Amandi. A knowledge-based evolutionary assistant to software development project scheduling. *Expert Systems with Applications*, 38(7):8403–8413, 2011.