UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Verificarea unui algoritm pentru problema de selecție a activităților cu profit maxim în Dafny

propusă de

Roxana Mihaela Timon

Sesiunea: iulie 2024

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea unui algoritm pentru problema de selecție a activităților cu profit maxim în Dafny

Roxana Mihaela Timon

Sesiunea: iulie 2024

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Ștefan Ciobâcă.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Timon Roxana Mihaela** domiciliat în , născut la data de , identificat prin CNP , absolvent al **Universității "Alexandru-Ioan Cuza" din Iași**, **Facultatea de informatică** specializarea **informatică**, promoția 2022, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Verificarea unui algoritm pentru problema de selecție a activităților cu profit maxim în Dafny** elaborată sub îndrumarea domnului **Conf. Dr. Ștefan Ciobâcă**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop. Declar că lucrarea de față are același conținut cu lucrarea în format electronic pe care profesorul îndrumător a verificat-o prin intermediul software-ului de detectare a plagiatului.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul **Verificarea unui algoritm pentru problema de selecție a activităților cu profit maxim în Dafny**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Roxana Mihaela Timon**

Data: ..... Semnătura: .....

Cuprins

Motivație

Am ales să fac această temă deoarece Dafny era un limbaj de programare nou pentru mine și am considerat a fi o provocare. Știam doar că acesta este folosit pentru a asigura o mai mare siguranță și corectitudine, putând fi aplicat în industria aerospațială, în industria medicală, la dezvoltarea sistemelor financiare, în securitate și criptografie. Totodată, prin demonstrarea corectitudinii unui algoritm în Dafny puteam să-mi folosesc pe lângă cunoștințele informatice și pe cele de matematică, de care am fost mereu atrasă. Pentru a crește gradul de complexitate a lucrării am decis să folosesc ca și tehnică de proiectare a algoritmilor programarea dinamică. Astfel, având posibilitatea sa înțeleg mai bine cum funcționează programarea dinamică și să demonstrez că, cu ajutorul ei, se obține o soluție optimă.

Introducere

În cadrul lucrării voi prezenta verificarea în Dafny a unui algoritm pentru problema de selecție a activităților cu profit maxim folosind programarea dinamică.

Lucrarea este structură în trei capitole:

- **Dafny**. În acest capitol voi prezenta particularitățile limbajului de programare Dafny.
- **Programarea dinamică**. În acest capitol voi reaminti despre programarea dinamică ca tehnică de proiectare a algoritmilor, despre avantajele sale în comparație cu alte tehnici de proiectare, precum Greedy. Voi prezenta problema de selecție a activităților cu profit maxim, un algoritm bazat pe programare dinamică care rezolvă această problemă și cum funcționează programarea dinamică în cazul acestei probleme.
- **Problema de selecție a activităților cu profit maxim**. Acest capitol conține implementarea și verificarea în Dafny a unui algoritm bazat pe programare dinamică care rezolvă problema de selecție a activităților.

Capitolul 1

Dafny

1.1 Prezentare generală

Dafny este un limbaj imperativ de nivel înalt cu suport pentru programarea orientată pe obiecte. Subprogramele în Dafny se numesc metode. Metodele implementate în Dafny pot avea precondiții, postcondiții și invarianți care sunt verificate la compilare, bazându-se pe solverul SMT Z3 [de Moura and Bjørner, 2008]. În cazul în care o postcondiție nu poate fi demonstrată (fie din cauza unui timeout, fie din cauza faptului că aceasta nu este validă), compilarea eșuează. Prin urmare, putem avea un grad ridicat de încredere într-un program verificat cu ajutorul sistemului Dafny. Acesta a fost conceput pentru a facilita scrierea de programe fără erori de execuție și corecte în sensul că respectă specificația.

Dafny [Leino, 2021], ca platformă de verificare, permite programatorului să specifice comportamentul dorit al programelor sale, astfel încât acesta să verifice dacă implementarea efectivă este conformă cu acel comportament. Cu toate acestea, acest proces nu este complet automat iar uneori programatorul trebuie să ghideze verificatorul [Leino, 2021].

Următoarele implementări au fost verificate în Dafny:

- algoritmul DPLL [Andrici and Ciobâcă, 2019];
- structurilor de date liniare mutabile și algoritmi bazați pe iteratori [Blázquez et al., 2023].

Iată un exemplu de cod preluat din manualul Dafny:

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
  requires 0 < y
  ensures less < x < more
{
  more := x + y;
  less := x - y;
}
```

Acestă metoda primește ca parametri variabilele `x` si `y`, unde `y` trebuie sa respecte precondiția `requires 0 < y`, adică y sa fie strict pozitiv. Aceasta metodă calculează suma și diferența dintre `x` și `y` pe care le returnează în variabilele `more` și `less`. Postcondiția care trebuie sa fie îndeplinită la ieșirea din metodă este `ensures less < x < more`.

Capitolul 2

Programarea dinamică

Programarea dinamică este o tehnică de proiectare a algoritmilor utilizată pentru rezolvarea problemelor de optimizare. Pentru a rezolva o anumită problemă folosind programarea dinamică, trebuie să identificăm în mod convenabil mai multe subprobleme. După ce alegem subproblemele, trebuie să stabilim cum se poate calcula soluția unei subprobleme în funcție de alte subprobleme mai mici. Principala idee din spatele programării dinamice constă în stocarea rezultatelor subproblemele pentru a evita recalcularea lor de fiecare dată când sunt necesare. În general, programarea dinamică se aplică pentru probleme de optimizare pentru care algoritmii greedy nu produc în general soluția optimă [Lucanu and Ciobâcă].

2.1 Avantajele programării dinamice față de celelalte tehnici de proiectare

În ceea ce privește programarea dinamică și tehnica greedy, în ambele cazuri apare noțiunea de subproblemă și proprietatea de substructură optimă. De fapt, tehnica greedy poate fi gândită ca un caz particular de programare dinamică, unde rezolvarea unei probleme este determinată direct de alegerea greedy, nefiind nevoie de a enumera toate alegerile posibile [Lucanu and Ciobâcă]. Avantajele programării dinamice fața de tehnica greedy sunt:

- **Optimizare Globală**: : Programarea dinamică are capacitatea de a găsi soluția optimă globală pentru o clasă mai mare de probleme, în timp ce algoritmii greedy pot fi limitați la luarea deciziilor locale care pot duce la o soluție suboptimală.

Cu toate acestea, algoritmii greedy pot fi mai potriviți pentru problemele care permit luarea de decizii locale și producerea rapidă a unei soluții aproximative.

2.2 Ce este problema de selecție a activităților cu profit maxim ?

Problema de selecție a activităților cu profit maxim este o problema de optimizare. Dându-se o listă de activități distincte caracterizate prin timp de început, timp de încheiere și profit, ordonate după timpul de încheiere se cere o secventă de activități care nu se suprapun două câte două și care produc un profit maxim. Spre exemplu, pentru datele de intrare din Tabela 2.1: profitul maxim este 250, pentru soluția optimă formată din Activitatea 1 și Activitatea 4.

Timp de început	Timp de încheiere	Profit
1	2	50
3	5	20
6	19	100
2	100	200

Tabela 2.1: Date de intrare

2.3 Cum funcționează programarea dinamică în cazul problemei de selecție a activităților

Pentru problema de selecție a activităților se poate folosi programarea dinamică, deoarece aceasta poate fi împărțită în subprobleme, respectiv la fiecare pas putem forma o soluție parțiala optimă, de al cărei profit ne putem folosi la următorul pas. O **subproblemă** este caracterizată de un index în secvența de activități. Semnificația indexului, în subproblemă, este că sunt disponibile doar activitățile de la 0 la `index - 1` din secvența de activități primită ca date de intrare.

De exemplu, o subproblema pentru datele de intrare de mai sus cu indexul 2, este formată din activitățile de pe poziția 0 până la poziția 2 : **Activitatea 1, Activitatea 2 și Activitatea 3**.

2.4 Pseudocod

Iată cum putem rezolva în pseudocod algoritmul pentru problema de selecție a activităților cu profit maxim folosind programarea dinamică.

```
struct Activitate {
    timp de inceput: int
    timp de incheiere : int
    profit : int
}
```

```
function planificareActivitatiPonderate(activitati):
    // activitatile sunt sortate crescator dupa timpul de incheiere
    // initializam un vector pentru a stoca profiturile maxime pentru fiecare activitate, fie dp un vector de dimensiune n, unde n este numarul de activitati
    dp[0] = activitati[0].profit
    solutie[0] = [1] //un vector binar 0 - nu am ales activitatea si 1 - am ales activitatea
    solutiiOptime = solutie //stocam solutiile optime la fiecare pas
    // Programare dinamica pentru a gasi profitul maxim
    pentru i de la 1 la n-1:
        // Gaseste cea mai recenta activitate care nu intra in conflict cu activitatea curenta
        solutiaCuActivitateaI = [1] // selectam activitatea curenta
        ultimaActivitateNeconflictuala = gasesteUltimaActivitateNeconflictuala(activitati, i)
        profitulCuActivitateaCurenta = activitati[i].profit
        daca ultimaActivitateNeconflictuala != -1:
            profitulIncluderiiActivitatiiCurente += dp[ultimaActivitateNeconflictuala]
            solutiaCuActivitateaI = solutiiOptime[ultimaActivitateNeconflictuala] + solutiaCuActivitateaI
        dp[i] = maxim(profitulIncluderiiActivitatiiCurente, dp[i-1])

        daca profitulIncluderiiActivitatiiCurente > dp[i-1]:
            solutie = solutieCuActivitateaI
        else
            solutie = solutie + [0]
        solutiiOptime = solutiiOptime + solutie
    // Returneaza solutia si profitul maxim
    return solutie, dp[n-1]

function gasesteUltimaActivitateNeconflictuala(activitati, indexCurent):
    pentru j de la indexCurent-1 la 0:
        daca activitati[j].timpDeIncheiere <= activitati[indexCurent].timpDeInceput:
            return j
    return -1
```

2.4.1 Soluția parțială și soluția parțială optimă

O soluție parțială conține doar activitățile din subproblemă care nu se suprapun. O soluție parțială este optimă, dacă profitul acesteia este mai mare sau egal decât al oricărei alte soluții

parțiale pentru aceeași subproblemă. Pentru a reprezenta o soluție parțiala folosim un vector caracteristic, unde 0 înseamna ca activitatea nu a fost selectată, iar 1 că activitatea a fost selectată.

Spre exemplu, o soluția parțială pentru subproblema ce conține toate activitățile primite ca date de intrare este formată din Activitatea 1, Activitatea 2, Activitatea 3 și are valoarea [1, 1, 1, 0]. În schimb, soluția optimă pentru aceasta subproblemă, [1, 0, 0, 1] este formată din Activitatea 1 și Activitatea 4.

Pentru datele de intrare menționate de mai sus, subproblemele și soluțiile optime sunt :
Pentru prima subproblemă (în care este disponibilă doar prima activitate):

1. soluția parțiala optimă este formata din Activitatea 1.
2. iar profit-ul maxim este 50.

Pentru a doua subproblemă (în care sunt disponibile primele două activități):

1. deoarece Activitatea 1 nu se suprapune cu Activitatea 2 se obține soluția parțială optimă formata din Activitatea 1 și Activitatea 2.
2. profitul optim la pasul 2 este 50 + 20 = 70, care este mai mare decât cel anterior (condiție necesară).

Pentru a treia subproblemă (în care sunt disponibile primele trei activități):

1. soluția parțială optimă este formată din Activitatea 1, Activitatea 2 și Activitatea 3, deoarece Activitatea 3 nu se suprapune cu activitatea 2 (înseamnă că nu se suprapune cu soluția parțiala de la al 2-lea pas, fiind ordonate după timpul de sfârșit) și le putem concatena.
2. profitul pentru această soluție parțiala este strict mai mare decât cel de la pasul 2, noul profit optim devenind 170.

Pentru a patra subproblemă (în care sunt disponibile toate activitățile):

1. soluția parțială ce conține Activitatea 4 este formata din Activitatea 4 și Activitatea 1.
2. profitul pentru aceasta soluție parțiala este strict mai mare decât profitul optim anterior = 170, noul profit optim devenind 250.

Astfel, soluția problemei este cea de la ultimul pas, fiind formata din Activitatea 1 și Activitatea 4 cu profitul optim 250.

În cazul problemei de selecție a activităților cu profit maxim, **proprietatea de substructură optimă** înseamnă dacă eliminăm o activitate dintr-o soluție parțială optimă, atunci obținem tot o soluție parțială optimă.

Capitolul 3

Implementarea și verificarea în Dafny a algoritmului

3.1 Reprezentarea datelor de intrare și predicate specifice

Pentru a defini o activitate am folosit un tuplu, reprezentat în Dafny prin următorul tip algebric de date

```
datatype Job = Tuple(jobStart: int, jobEnd: int, profit: int)
```

unde:

- `Job` este numele tipului de date,
- `Tuple` este singurul constructor,
- iar `jobStart, jobEnd, profit` sunt destructori.

Problema de selecție a activităților are ca date de intrare un singur parametru:

- `jobs` de tipul `seq<Job>` : reprezintă secvența de activități

Tipul de date `Seq` reprezintă o secvență (sau listă) de elemente. Secvențele în Dafny sunt imuabile, ceea ce înseamnă că, odată create, nu pot fi modificate. Cu toate acestea, se pot crea noi secvențe prin operații pe secvențe existente.

Un **predicat** este o funcție care returnează o valoare booleană[Koenig and Leino, 2012]. Acestea sunt folosite ca precondiții și postcondiții în metode. Predicatele pe care le-am folosit la validarea datelor de intrare sunt:

- Predicatul `validJob(jobs: Job)` verifică dacă numerele din tuplu pot reprezenta timpul de început, timpul de final și profitul unei activități. Am ales ca timpul de început să fie strict mai mic decât timpul de încheiere, deoarece nu permitem activități de dimensiune 0. Profitul am ales sa fie pozitiv, deoarece activitățile cu un profit negativ ar aduce un profit mai mic.

```
predicate validJob(job: Job)
{
  job.jobStart < job.jobEnd && job.profit >= 0
}
```

- Predicatul `validJobsSeq(jobs: seq<Job>)` verifică proprietatea de `validJob` pentru toată secvența de activități primită ca date de intrare.

```
predicate validJobsSeq(jobs: seq<Job>)
{
  forall job :: job in jobs ==> validJob(job)
}
```

- Predicatul `JobComparator (job1: Job, job2: Job)` compară două activități după timpul de încheiere.

```
predicate JobComparator(job1: Job, job2: Job)
{
  job1.jobEnd <= job2.jobEnd
}
```

- Predicatul `sortedByActEnd(s: seq<Job>)` ne asigură că secvența de activități conține activități sortate după timpul de încheiere. Acest lucru este important pentru a putea împărți problema în subprobleme.

```
predicate sortedByActEnd(s: seq<Job>)
  requires validJobsSeq(s)
{
  forall i, j :: 0 <= i < j < |s| ==> JobComparator(s[i], s[j])
}
```

- Predicatul `distinctJobs(j1: Job, j2: Job)` compară două activități și asigură că ele diferă prin cel puțin un timp. Nu comparăm profitul deoarece permitem activități cu același profit.

```
predicate  distinctJobs(j1: Job, j2: Job)
  requires validJob(j1) && validJob(j2)
```

```
{
  j1.jobStart != j2.jobStart || j1.jobEnd != j2.jobEnd
}
```

- Predicatul `distinctJobsSeq(s: seq<Job>)` verifică proprietatea de `distinctJobs` pentru toată secvența de activități.

```
predicate distinctJobsSeq(s: seq<Job>)
  requires validJobsSeq(s)
{
  forall i, j :: 0 <= i < j < |s| ==> distinctJobs(s[i], s[j])
}
```

- Predicatul `validProblem(jobs: seq<Job>)` este folosit pentru a defini ce înseamnă date de intrare valide. Pentru a defini predicatul `validProblem(jobs: seq<Job>)` am folosit predicatele `validJobsSeq(jobs: seq<Job>)`, `sortedByActEnd(s: seq<Job>)`,`distinctJobsSeq(s: seq<Job>)`. Am ales `1 <= |jobs|` pentru a nu permite date de intrare triviale.

```
predicate validProblem(jobs: seq<Job>)
{
  1 <= |jobs| && validJobsSeq(jobs) && sortedByActEnd(jobs)
   && distinctJobsSeq(jobs)
}
```

3.2 Reprezentarea soluțiilor parțiale și predicate specifice

O soluție parțială este un vector caracteristic pentru un prefix al secvenței de activități primită la intrare, alcătuit doar din valori binare, unde 1 reprezintă că activitatea a fost selectată, iar 0 reprezintă că aceasta nu a fost selectată.

Predicatele pe care le-am folosit pentru validarea soluțiilor parțiale sunt:

- Predicatul `overlappingJobs(j1: Job, j2: Job)` verifică dacă activitățile se suprapun. Două activități se suprapun, dacă prima se termină înainte ca cealaltă să înceapă și invers.

```
predicate overlappingJobs(j1:Job, j2:Job)
  requires validJob(j1)
  requires validJob(j2)
{
  j1.jobEnd > j2.jobStart &&  j2.jobEnd > j1.jobStart
}
```

- Predicatul `hasNoOverlappingJobs(partialSol:seq<int>, jobs:seq<Job>)` verifică pentru o soluție parțială, să nu conțină activități care să se suprapună.

```
predicate hasNoOverlappingJobs(partialSol: seq<int>,
    jobs: seq<Job>)
  requires validJobsSeq(jobs)
{
  |partialSol| <= |jobs|  && forall i, j :: 0 <= i < j < |partialSol|
   ==> (partialSol[i] == 1 && partialSol[j] == 1)
    ==> !overlappingJobs(jobs[i], jobs[j])
}
```

- Cu ajutorul predicatului `isPartialSol(partialSol: seq<int>, jobs: seq<Job>, length: int)` am verificat proprietatea de soluție parțială: secvența să conțină doar valori de 0 și 1, activitățile care corespund acestui vector caracteristic să nu se suprapună și să aibă lungimea dorită.

```
predicate isPartialSol(partialSol: seq<int>, jobs: seq<Job>,
length: int)
  requires validJobsSeq(jobs)
{
  |partialSol| == length && forall i :: 0 <= i <= |partialSol| - 1
    ==> (0 <= partialSol[i] <= 1) &&
     hasNoOverlappingJobs(partialSol, jobs)
}
```

3.2.1 Funcții importante folosite

Funcțiile reprezintă un element important pe care le-am folosit la verificarea corectitudinii algoritmului ales. Funcțiile în Dafny sunt asemănătoare funcțiilor matematice, fiind constituite dintr-o singură instrucțiune al cărui tip de return este menționat în antetul acestora. Acestea sunt prezente doar la compilare și nu pot avea efecte secundare [Blázquez et al., 2023]. Una din funcțiile pe care am folosit-o cel mai des este funcția:

```
function PartialSolProfit(solution: seq<int>, jobs: seq<Job>, index: int): int
```

cu ajutorul căreia calculez profitul unei soluții parțiale, după formula

$$\sum_{i=0}^{|partialSol|} partialSol[i] * jobs[i].$$

```
function PartialSolProfit(solution: seq<int>, jobs: seq<Job>, index: int)
    :int
```

```
  requires 0 <= index <= |solution| <= |jobs|
  decreases |solution| - index
  ensures PartialSolProfit(solution, jobs, index) ==
    if index == |solution| then 0 else
        solution[index] * jobs[index].profit +
            PartialSolProfit(solution, jobs, index + 1)
{

  if index == |solution| then 0 else
    solution[index] * jobs[index].profit +
     PartialSolProfit(solution, jobs, index + 1)
}
```

3.3 Reprezentarea soluțiilor parțiale optime

O soluție parțială optimă este o soluție parțială care aduce un profit maxim.
Pentru a valida soluțiile parțiale am folosit următoarele predicate:

- Predicatul `hasLessProf` verifică profitul unei secvențe să fie mai mic sau egal cu un `maxProfit`.

```
predicate HasLessProf(partialSol: seq<int>, jobs: seq<Job>,
maxProfit: int,position: int)
  requires validJobsSeq(jobs)
  requires 0 <= position < |partialSol| <= |jobs|
{
  PartialSolProfit(partialSol, jobs, position) <= maxProfit
}
```

- Predicatul `isOptParSol` verifică dacă o soluție parțiala este și optimă. O soluție parțială este optimă dacă orice altă soluție parțială pentru aceeași subproblemă are un profit mai mic sau egal decât profitul acesteia. Predicate `ghost` sunt folosite pentru specificare și verificare, dar nu au impact asupra execuției, fiind șterse la compilare.

```
ghost predicate isOptParSol(partialSol: seq<int>, jobs: seq<Job>,length: int)
  requires validJobsSeq(jobs)
  requires 1 <= |jobs|
  requires length == |partialSol|
  requires 1 <= |partialSol| <= |jobs|
{
  isPartialSol(partialSol, jobs, length) &&
  forall otherSol :: isPartialSol(otherSol, jobs, length)
   ==> HasLessProf(otherSol, jobs,
```

```
    PartialSolProfit(partialSol, jobs, 0), 0)
}
```

- Predicatul `isOptParSolDP` verifică o secvență să fie o soluție parțială optimă pentru o subproblemă și să aibă profitul optim.

```
ghost predicate isOptParSolDP(partialSol: seq<int>, jobs: seq<Job>,
 length : int, profit:int)
  requires validJobsSeq(jobs)
  requires 1 <= |partialSol|
  requires 1 <= length <= |jobs|
{
  |partialSol| == length && isOptParSol(partialSol, jobs, length)
   && HasProfit(partialSol, jobs, 0,  profit)
}
```

- Predicatul `OptParSolutions` verifică pentru o secvență de secvențe:

  1. fiecare secvență să corespundă subproblemei pe care o rezolvă
  2. fiecare secvență să fie o soluție parțială optimă
  3. fiecare secvență să aibă profitul optim

```
ghost predicate OptParSolutions(allSol: seq<seq<int>>, jobs: seq<Job>,
 dp:seq<int>, index: int)
  requires validJobsSeq(jobs)
  requires |dp| == |allSol| == index
  requires 1 <= index <= |jobs|
{
  forall i : int :: 0 <= i < index ==> |allSol[i]| == i + 1
   && isOptParSolDP(allSol[i], jobs, i + 1, dp[i])
}
```

3.4 Punctul de intrare în algoritm

Variabilele locale folosite pentru rezolvarea algoritmului pentru problema de selecție a activităților cu profit maxim sunt:

- `solution` de tipul *seq<int>*: reprezintă soluția parțială optimă obținută la fiecare iterație
- `dp` de tipul *seq<int>*: conține toate profiturile optime obținute la fiecare pas
- `allSol` de tipul *seq<seq<int>>*: conține soluțiile parțiale optime obținute la fiecare pas

- `partialSol` de tipul *seq<int>*: reprezintă o soluție parțială

Iată punctul de intrare în algoritm:

```
method WeightedJobScheduling(jobs: seq<Job>) returns (sol: seq<int>,
 profit : int)
  requires validProblem(jobs)
  ensures isSolution(sol, jobs)
  ensures isOptimalSolution(sol, jobs)
{
  var dp :seq<int> := [];
  var dp0 := jobs[0].profit;
  dp := dp + [dp0];
  var solution : seq<int> := [1];
  var i: int := 1;
  var allSol : seq<seq<int>> := [];
  allSol := allSol + [[1]];

  assert |solution| == 1;
  assert |allSol[0]| == |solution|;
  assert 0 <= solution[0] <= 1;

  assert isPartialSol(solution, jobs, i);
  assert validJob(jobs[0]); //profit >=0
  assert isOptParSol(solution, jobs, i);

  while i < |jobs|
    invariant 1 <= i <= |jobs|
    decreases |jobs| - i
    invariant i == |dp|
    invariant 1 <= |dp| <= |jobs|
    decreases |jobs| - |dp|
    invariant isPartialSol(solution, jobs, i)
    invariant |solution| == i
    invariant i == |allSol|
    decreases |jobs| - |allSol|
    decreases |jobs| - |allSol[i-1]|
    invariant isPartialSol(allSol[i-1], jobs, i)
    invariant HasProfit(solution, jobs, 0, dp[i - 1])
    invariant HasProfit(allSol[i - 1], jobs, 0 , dp[i - 1])
    invariant OptParSolutions(allSol, jobs, dp, i)
    invariant isOptParSol(allSol[i - 1], jobs, i)
    invariant forall partialSol :: |partialSol| == i
```

```
      && isPartialSol(partialSol, jobs, i) ==>
       HasLessProf(partialSol, jobs, dp[i - 1], 0)
    invariant isOptParSol(solution, jobs, i)
  {
    var maxProfit, optParSolWithI :=
        MaxProfitWithJobI(jobs, i, dp, allSol);

    if dp[i-1] >= maxProfit
    {
      solution, dp :=
        leadsToOptWithoutJobI(jobs, dp, allSol, i, maxProfit, solution);
      assert isOptParSol(solution, jobs, i + 1);
    }
    else
    {
      solution, dp :=
        leadsToOptWithJobI(jobs, dp, allSol, i, maxProfit, optParSolWithI);
      assert isOptParSol(solution, jobs, i + 1);
    }
    allSol := allSol + [solution];
    i := i + 1;
  }

  sol := solution;
  profit := dp[|dp|-1];
}
```

3.4.1 Precondiții, postcondițiilor și invarianți

Metodele și funcțiile Dafny sunt adnotate cu precondiții, postcondiții, invarianți și afirmații (assert-uri), iar verificatorul generează condiții de verificare care sunt trimise unui solver SMT care verifică dacă acestea sunt valide [Leino, 2021]. **Precondițiile** reprezintă condiții care trebuie sa fie îndeplinite la intrarea într-o metodă sau funcție. **Postcondițiile** reprezinta condiții care trebuie sa fie îndeplinite la ieșirea dintr-o metodă sau funcție. **Invarianții** reprezintă condiții care trebuie să fie îndeplinite la intrarea în bucla, în timpul buclei și la ieșirea din aceasta. În cazul problemei de selecție a activităților avem o singură precondiție care trebuie sa fie îndeplinită, precondiția `validProblem(jobs: seq<Job>)`. Secvența de activități primită trebuie să respecte următoarea precondiție: `validProblem(jobs)`. Asta înseamnă să conțină doar activități valide, distincte și sortate după timpul de încheiere.

```
method WeightedJobScheduling(jobs: seq<Job>) returns
 (sol: seq<int>, profit : int)
     requires validProblem(jobs)
     ensures isSolution(sol, jobs)
     ensures isOptimalSolution(sol, jobs)

 predicate validProblem(jobs: seq<Job>)
{
  1 <= |jobs| && validJobsSeq(jobs) && sortedByActEnd(jobs)
   && distinctJobsSeq(jobs)
}
```

La final, când algoritmul se termină, soluția returnată trebuie să îndeplinească următoarele postcondiții:

- `isSolution(solution: seq<int>, jobs: seq<Job>)`
- `isOptimalSolution(solution: seq<int>, jobs: seq<Job>)`

despre care ofer mai multe detalii în capitolul Date de ieșire și predicate specifice.

La fiecare iterație a algoritmului, următoarele proprietăți sunt invarianți:

```
while i < |jobs|
    invariant 1 <= i <= |jobs|
    decreases |jobs| - i
    invariant i == |dp|
    invariant 1 <= |dp| <= |jobs|
    decreases |jobs| - |dp|
    invariant isPartialSol(solution, jobs, i)
    invariant |solution| == i
    invariant i == |allSol|
    decreases |jobs| - |allSol|
    decreases |jobs| - |allSol[i-1]|
    invariant isPartialSol(allSol[i-1], jobs, i)
    invariant HasProfit(solution, jobs, 0, dp[i - 1])
    invariant HasProfit(allSol[i - 1], jobs, 0 , dp[i - 1])
    invariant allSol[i - 1] == solution
    invariant OptParSolutions(allSol, jobs, dp, i)
    invariant isOptParSol(allSol[i - 1], jobs, i)
    invariant forall partialSol :: |partialSol| == i
     && isPartialSol(partialSol, jobs, i) ==>
      HasLessProf(partialSol, jobs, dp[i - 1], 0)
    invariant forall i :: 0 <= i < |dp| ==> dp[i] >= 0
    invariant isOptParSol(solution, jobs, i)
```

Un invariant foarte important este:

```
invariant forall partialSol :: |partialSol| == i
  && isPartialSol(partialSol, jobs, i) ==>
  hasLessProf(partialSol, jobs, dp[i - 1], 0)
```

deoarece cu ajutorul lui ne asigurăm că la fiecare pas variabila `dp` reține profitul optim.

La fel de importante sunt și:

```
invariant isPartialSol(solution, jobs, i)
invariant isOptParSol(solution, jobs, i)
```

care ne asigură că variabila `solution` reține la fiecare pas o soluție parțială optimă.

3.5 Detalii de implementare

Metoda care îmi generează soluția parțială optimă ce conține activitatea `i`, este `MaxProfitWithJobI`. În Dafny, metodele sunt subrutine care pot fi executate în timpul execuției și pot avea efecte secundare, cum ar fi modificarea obiectelor din heap [Leino, 2021]. Această metodă primește ca parametri:

1. `jobs` : secvența de activități (problema);
2. `i` : poziția (din secvența de activități primită ca date de intrare) pe care se află activitatea cu care vrem să formăm o soluție parțială optimă;
3. `dp` : secvența cu profiturile optime obținute la pașii anteriori;
4. `allSol`: secvența cu soluțiile parțiale optime obținute la pașii anteriori.

Și returnează:

1. `maxProfit`: profitul pentru soluția parțială optimă ce conține activitatea de pe poziția i din secvența de activități primită ca date de intrare;
2. `optParSolWithI`: soluția parțială optimă ce conține activitatea de pe poziția i.

```
method MaxProfitWithJobI(jobs: seq <Job>, i: int, dp: seq<int>,
 allSol :seq<seq<int>>) returns (maxProfit:int, optParSolWithI: seq<int>)
  requires validProblem(jobs)
  requires 1 <= i < |jobs|
  requires |allSol| == i
  requires |dp| == i
```

```
  requires OptParSolutions(allSol, jobs, dp, i)
  ensures isPartialSol(optParSolWithI, jobs, i + 1)
  ensures maxProfit == PartialSolProfit(optParSolWithI, jobs, 0)
  ensures partialSolutionWithJobI(optParSolWithI, jobs, i)
  ensures forall partialSol :: |partialSol| == i + 1 &&
    partialSolutionWithJobI(partialSol, jobs, i) ==>
        HasLessProf(partialSol, jobs, maxProfit, 0)
{

  var max_profit := 0;
  var partialSolutionPrefix : seq<int> := [];
  var partialSolution : seq<int> := [];
  var j := i - 1;
  var length := 0;

  while j >= 0 && jobs[j].jobEnd > jobs[i].jobStart
    invariant - 1 <= j < i
    invariant forall k :: j < k < i ==> jobs[k].jobEnd > jobs[i].jobStart
    invariant forall k :: j < k < i ==> validJob(jobs[k])
    invariant forall k :: j < k < i ==> JobComparator(jobs[k], jobs[i])
    invariant forall k :: j < k < i ==> jobs[k].jobEnd > jobs[k].jobStart
    invariant forall k :: j < k < i ==> overlappingJobs(jobs[k], jobs[i])
  {
    j := j - 1;
  }

  assert  j != -1 ==> !overlappingJobs(jobs[j], jobs[i]);

  if j >= 0
  {
    max_profit, partialSolution, length :=
     OptParSolWhenNonOverlapJob(jobs, i, dp, allSol, j);
  }
  else
  {
    max_profit, partialSolution, length :=
     OptParSolWhenOverlapJob(jobs, i, dp);
  }
  assert isPartialSol(partialSolution, jobs, length);
  assert forall partialSol :: |partialSol| == i + 1
    && partialSolutionWithJobI(partialSol, jobs, i)
        ==> HasLessProf(partialSol, jobs, max_profit, 0) ;
```

```
  maxProfit := max_profit;
  optParSolWithI := partialSolution;
}
```

Iată cum funcționează algoritmul:

1. **Date de intrare**: primim ca date de intrare variabila `jobs` care reprezintă o secvență de activități caracterizate prin timp de început, timp de încheiere și profit, distincte (diferite prin cel puțin un timp), sortate după timpul de încheiere.

2. **Programare dinamică**:
   La fiecare iterație stocam soluțiile optime ale subproblemelor și profiturile acestora.

   - variabila `solution`, care reprezintă soluția parțială optimă obținută la fiecare pas, respectiv soluția optimă finală
   - variabila `dp`, care va conține profiturile optime obținute la fiecare iterație
   - variabila `allSol` care va conține toate soluțiile parțiale optime obținute la fiecare pas

3. **Iterații**:

   - **La prima iterație** `solution = [1], dp = jobs[1].profit, allSol = [[1]]`
   - **Selectăm activitatea de pe poziția `i`**, unde `i = 1... |jobs| - 1`, în ordinea în care au fost declarate în secvența de intrare
   - **Formăm soluția parțială optimă ce conține activitatea de pe poziția `i`:**
     - **profitul** pentru soluția parțială optimă cu activitatea `i` are valoarea inițială `jobs[i].profit`
     - **caută o activitate `j`**, unde `0 <= j < i`, care nu se suprapune cu activitatea `i`, adică `jobs[j].jobEnd <= jobs[i].jobStart`
     - **dacă `j >= 0`** atunci soluția parțială optimă cu activitatea de pe poziția i va fi formata din `allSol[j]+ 0...0 + 1`, unde:
       * `allSol[j]` conține soluția parțială optimă cu activitățile de pana la poziția j inclusiv
       * 0-urile **dacă există**, activitățile dintre `j` și `i` care se suprapun cu activitatea de pe poziția `i`
       * 1 înseamnă că activitatea `i` este selectată

       iar **profitul** (`maxProfit`) pentru soluția parțială optimă ce conține activitatea `i` devine `dp[j] + jobs[i].profit`

- **dacă** `j == -1` atunci soluția parțială optimă cu activitatea de pe poziția i va fi formată din `0...0 + 1`, unde:
  - 0-urile reprezinta, activitățile din fața activității i care se suprapun cu aceasta.
  - 1- înseamnă că activitatea de pe poziția i a fost selectată

  iar **profitul**(`maxProfit`) rămâne egal cu **jobs[i].profit**

- **Comparăm** profitul soluției parțiale optime (`maxProfit`) ce conține activitatea de pe poziția `i` cu profitul care s-ar obține fără activitatea curentă (`dp[i-1]`) și **actualizăm** valoarea variabilelor `dp` și `solution`
  - dacă `dp[i-1]` >= **profitul** soluției parțiale optime cu activitatea i (`maxProfit`), atunci `dp[i] = dp[i-1]` și `solution = solution + [0]`
  - dacă `dp[i-1]` < **profitul** soluției parțiale optime cu activitatea i (`maxProfit`), atunci `dp[i] = maxProfit` (profitul soluției parțiale optime cu activitatea i) și `solution` = soluția parțială optimă ce conține activitatea i (`optParSolWithI`)
- **Actualizăm valoarea variabilei `allSol`** care reține soluțiile parțiale optime obținute la fiecare iterație, `allSol = allSol + [solution]`

4. **Datele de ieșire ale problemei** sunt reprezentate de soluția optimă `sol := solution` și de profitul maxim, `profit` = `dp[|dp| - 1]`

3.6 Date de ieșire și predicate specifice

Algoritmul care rezolvă problema de selecție a activităților cu profit maxim prezentat mai sus are următoarele date de ieșire:

- `sol` de tipul *seq<int>* : reprezintă soluția optimă finală pentru secvența de activități primită
- `profit` de tipul *int* : reprezintă profitul maxim al soluției optime finale

Predicatele folosite la validarea datele de ieșire sunt:

- Predicatul `isSolution(solution: seq<int>, jobs: seq<Job>)` verifică dacă o secvență este soluție parțială și o soluție pentru secvența de activități primită ca date de intrare.

```
predicate isSolution(solution: seq<int>, jobs: seq <Job>)
  requires validJobsSeq(jobs)
{
```

```
  isPartialSol(solution, jobs, |jobs|)
}
```

- Predicatul `isOptimalSolution(solution: seq<int>, jobs: seq<Job>)` care verifică dacă o secvență este o soluție optimă.
  O soluție este optimă dacă pentru aceeași problemă, nu există o altă soluție care să aibă un profit mai bun.

```
ghost predicate isOptimalSolution(solution: seq<int>, jobs: seq<Job>)
  requires validJobsSeq(jobs)
  requires 1 <= |jobs|
  requires |solution| == |jobs|
{
  isSolution(solution, jobs) &&
  forall otherSol :: isSolution(otherSol, jobs) ==>
   PartialSolProfit(solution, jobs, 0) >=
    PartialSolProfit(otherSol, jobs, 0)
}
```

Variabila de ieșire, `sol`, este reprezentată printr-un vector caracteristic al secvenței de activități primită ca date de intrare. Acest vector conține doar valori de 0 și 1, unde 0 înseamnă că o activitate nu a fost selectată, în timp ce 1 indică faptul că activitatea respectivă a fost selectată.

3.7 Leme importante în demonstrarea corectitudinii

Unul din punctele complexe în dezvoltarea codului de verificare pentru problema de selecție a activităților a fost atunci când a trebuit să recurg la leme pentru a demonstra anumite proprietăți. Lemele reprezintă blocuri de instrucțiuni care au ca scop demonstrarea unor teoreme pe care Dafny nu reușește să le demonstreze de unul singur.

Am folosit lema `OtherSolhasLessProfThenMaxProfit2` pentru a demonstra că orice **soluție parțială**:

1. care **conține activitatea de pe poziția `i`**
2. pentru care activitățile de pe pozițiile `j + 1, ..., i - 1` **se suprapune cu activitatea `i`**, unde `0 <= j < i`
3. activitatea de pe poziția `j` **nu se suprapune** cu activitatea de pe poziția `i`

are un **profit mai mic sau egal** decât cel al soluției parțiale care îndeplinește aceleași proprietăți menționate mai sus, și pentru care `substructura [0..j] este optimă`.

```
lemma OtherSolHasLessProfThenMaxProfit2(partialSol: seq<int>, jobs : seq<Job>,
 i: int, j : int, max_profit : int, allSol : seq<seq<int>>, dp: seq<int>)
  requires validJobsSeq(jobs)
  requires 1 <= |jobs|
  requires 0 <= j < i < |jobs|
  requires |allSol| == |dp| == i
  requires OptParSolutions(allSol, jobs, dp, i)
  requires isOptParSol(allSol[j], jobs, j + 1)
  requires max_profit == dp[j] + jobs[i].profit
  requires forall k :: j < k < i ==> overlappingJobs(jobs[k], jobs[i])
  requires !overlappingJobs(jobs[j], jobs[i])
  requires |partialSol| == i + 1
  requires partialSolutionWithJobI(partialSol, jobs, i)
  requires PartialSolProfit(partialSol, jobs, i) == jobs[i].profit;
  requires isPartialSol(partialSol, jobs, i + 1)
  requires forall k :: j < k < i ==> partialSol[k] == 0
  ensures HasLessProf(partialSol, jobs, max_profit, 0)
{
  var k : int := i - 1;
  assert |partialSol| == i + 1;
  assert j <= k < i;
  //assert !exists k' :: k < k' < i;
  assert forall k' :: k < k' < i ==> partialSol[k'] == 0;
  assert PartialSolProfit(partialSol, jobs, i) == jobs[i].profit;
  ComputeProfitWhenOnly0BetweenJI(partialSol, jobs, i, j);
  assert PartialSolProfit(partialSol, jobs, j + 1) == jobs[i].profit;
  if !HasLessProf(partialSol, jobs, max_profit, 0)
  {
    var profit' := PartialSolProfit(partialSol, jobs, 0);
    assert max_profit == dp[j] + jobs[i].profit;

    assert HasMoreProfit(partialSol, jobs, max_profit, 0);
    assert !HasLessProf(partialSol, jobs, max_profit, 0);

    assert partialSol[..j+1] + partialSol[j+1..] == partialSol;

    SplitSequenceProfitEquality(partialSol, jobs, 0, j + 1);
    EqOfProfitFuncFromIndToEnd(partialSol, jobs, 0);
    EqOfProfFuncUntilIndex(partialSol, jobs, 0, j + 1);
    EqOfProfitFuncFromIndToEnd(partialSol, jobs, j + 1);

    assert PartialSolProfit(partialSol[..j + 1], jobs, 0) +
```

```
      PartialSolProfit(partialSol, jobs, j + 1) ==
       PartialSolProfit(partialSol, jobs, 0);
    assert PartialSolProfit(partialSol, jobs, j + 1) == jobs[i].profit;

    var partialSol' :seq<int> := partialSol[..j + 1];
    assert isPartialSol(partialSol', jobs, j + 1);
    var profit := PartialSolProfit(partialSol', jobs, 0);

    assert |partialSol'| == j + 1;
    assert profit + jobs[i].profit == profit';
    assert profit + jobs[i].profit > max_profit;
    assert profit > max_profit - jobs[i].profit;
    assert profit > dp[j];
    HasMoreProfThanOptParSol(allSol[j], jobs, partialSol');
    assert !isOptParSol(allSol[j], jobs, j + 1); //contradictie
    //assume false;
    assert false;
  }
  assert HasLessProf(partialSol, jobs, max_profit, 0);
}
```

Am folosit această lemă în cadrul metodei **OptParSolWhenNonOverlapJob** pe care o apelez în metoda **MaxProfitWithJobI** pe ramura cu `j >= 0`, atunci când găsim o activitate pe o poziție `j`, unde `j < i` în fața activității de pe poziția i, care nu se suprapune cu aceasta.

```
method  OptParSolWhenNonOverlapJob(jobs: seq <Job>, i: int, dp: seq<int>,
 allSol :seq<seq<int>>, j : int)
 returns (maxProfit:int, partialSolution: seq<int>, length: int)
  requires validProblem(jobs)
  requires 0 <= j < i < |jobs|
  requires |allSol| == i
  requires |dp| == i
  requires OptParSolutions(allSol, jobs, dp, i)
  requires !overlappingJobs(jobs[j], jobs[i]);
  requires jobs[j].jobEnd <= jobs[i].jobStart
  requires forall k :: j < k < i ==> overlappingJobs(jobs[k], jobs[i])
  requires !overlappingJobs(jobs[j], jobs[i]);
  ensures isPartialSol(partialSolution, jobs, i + 1)
  ensures partialSolutionWithJobI(partialSolution, jobs, i)
  ensures maxProfit == PartialSolProfit(partialSolution, jobs, 0)
  ensures forall partialSol :: |partialSol| == i + 1 &&
   partialSolutionWithJobI(partialSol, jobs, i) ==>
   HasLessProf(partialSol, jobs, maxProfit, 0)
```

```
  ensures length == i + 1;
{
  var partialSolutionPrefix : seq<int> := [];
  var max_profit : int := 0 ;
  length := 0;

  partialSolutionPrefix := allSol[j];
  length := length + |allSol[j]|;

  assert forall i :: 0 <= i <= length - 1 ==>
   0 <= partialSolutionPrefix[i] <= 1;
  assert hasNoOverlappingJobs(partialSolutionPrefix, jobs);

  max_profit := max_profit + dp[j];

  var nr_of_zeros := i - |allSol[j]|;

  while nr_of_zeros > 0
    decreases nr_of_zeros
    invariant 0 <= nr_of_zeros <= i - |allSol[j]|
    decreases i - length
    invariant |allSol[j]| <= length <= i
    invariant |partialSolutionPrefix| == length
    invariant forall k :: 0 <= k <= length - 1 ==>
     0 <= partialSolutionPrefix[k] <= 1
    invariant length < |jobs|;
    invariant length == i - nr_of_zeros
    invariant hasNoOverlappingJobs(partialSolutionPrefix, jobs)
    invariant forall k :: j < k < |partialSolutionPrefix| ==>
        partialSolutionPrefix[k] == 0
    invariant max_profit == PartialSolProfit(partialSolutionPrefix, jobs, 0)
  {
    AssociativityOfProfitFunc(partialSolutionPrefix, jobs, 0, 0);
    assert max_profit == PartialSolProfit(partialSolutionPrefix, jobs, 0);
    partialSolutionPrefix :=  partialSolutionPrefix + [0];
    assert length + nr_of_zeros < |jobs|;
    length := length + 1;
    nr_of_zeros := nr_of_zeros - 1;
    assert max_profit == PartialSolProfit(partialSolutionPrefix, jobs, 0);
  }

  assert length == i;
```

```
  assert |partialSolutionPrefix| == i ;

  assert forall k :: j < k < i ==> partialSolutionPrefix[k] == 0;
  assert forall k :: j < k < i ==> overlappingJobs(jobs[k], jobs[i]);
  assert isPartialSol(partialSolutionPrefix, jobs, i);

  assert hasNoOverlappingJobs(partialSolutionPrefix + [1], jobs);

  AssociativityOfProfitFunc(partialSolutionPrefix, jobs, 1, 0);
  partialSolutionPrefix := partialSolutionPrefix + [1];
  length := length + 1;
  max_profit := max_profit + jobs[i].profit;

  assert isPartialSol(partialSolutionPrefix, jobs, i + 1);
  assert max_profit == PartialSolProfit(partialSolutionPrefix, jobs, 0);
  forall partialSol | |partialSol| == i + 1
    && partialSolutionWithJobI(partialSol, jobs, i)
    ensures HasLessProf(partialSol, jobs, max_profit, 0)
  {

    OnlY0WhenOverlapJobs(partialSol, jobs, i, j);
    assert forall k :: j < k < i ==> partialSol[k] == 0;
    ProfitLastElem(partialSol, jobs, i);
    assert PartialSolProfit(partialSol, jobs, i) == jobs[i].profit;
    OtherSolHasLessProfThenMaxProfit2
        (partialSol, jobs, i, j, max_profit, allSol, dp);
  }
  maxProfit := max_profit;
  partialSolution := partialSolutionPrefix;
}
```

Pentru a demonstra această lemă am folosit metoda reducerii la absurd. Presupunând că ar exista o altă soluție parțiala cu aceleași proprietăți care ar avea un profit mai mare decât soluția parțială optimă obținută, am arătat că există o alta soluție parțiala de lungime `j + 1` al cărei profit este mai mare decât cel al soluției parțiale optime de lungime `j + 1`, ceea ce este imposibil, deoarece ar contrazice ipoteza de la care am plecat. În acest mod, am identificat și **proprietatea de substructura optimă**, care se ascunde în această lemă.

În lema `HasMoreProfThanOptParSol` am confirmat că în cazul în care există o altă soluție parțială cu un profit mai mare decât cel al soluției optime, atunci soluția considerată optimă nu poate fi cu adevărat optimă.

```
lemma HasMoreProfThanOptParSol(optimalPartialSol: seq<int>, jobs: seq<Job>,
```

```
 partialSol: seq<int>)
  requires validJobsSeq(jobs)
  requires 1 <= |optimalPartialSol| <= |jobs|
  requires |optimalPartialSol| == |partialSol|
  requires isPartialSol(partialSol, jobs, |partialSol|)
  requires isOptParSol(optimalPartialSol, jobs, |optimalPartialSol|)
  requires PartialSolProfit(partialSol, jobs, 0) >
   PartialSolProfit(optimalPartialSol, jobs, 0)
  ensures !isOptParSol(optimalPartialSol, jobs, |optimalPartialSol|)
{
  var other_profit := PartialSolProfit(partialSol, jobs, 0);
  var optParSolProfit := PartialSolProfit(optimalPartialSol, jobs, 0);
  assert forall otherSol:: isPartialSol(otherSol, jobs, |optimalPartialSol|)
   ==> HasLessProf(otherSol, jobs, optParSolProfit, 0);
  assert other_profit > optParSolProfit;
  assert !isOptParSol(optimalPartialSol, jobs, |optimalPartialSol|);
}
```

De asemenea, consider că și lemele pe care le-am demonstrat prin inducția au fost importante, precum `AssociativityOfProfitFunc`.

```
lemma AssociativityOfProfitFunc(partialSolPrefix : seq<int>, jobs: seq<Job>,
 val: int, index: int)
  requires 1 <= |jobs|
  requires validJobsSeq(jobs)
  requires 0 <= index <= |partialSolPrefix|
  requires 0 <= val <= 1
  requires 0 <= |partialSolPrefix| < |jobs|
  decreases |partialSolPrefix| - index
  ensures PartialSolProfit(partialSolPrefix, jobs, index)
    + val * jobs[|partialSolPrefix|].profit ==
          PartialSolProfit(partialSolPrefix + [val], jobs, index)
{
  if |partialSolPrefix| == index {

  }
  else
  {
    AssociativityOfProfitFunc(partialSolPrefix , jobs, val, index + 1);
  }
}
```

În cadrul acestei leme se poate observa cum am demonstrat proprietatea de asociativi-

tate a funcției `PartialSolutionPrefixProfit`, care îmi calculează profitul pentru o soluție parțială. Dacă adăugam un element la sfârșitul unei soluții parțiale, valoarea profitului acesteia crește cu valoarea profitului activității adăugate.

În metoda principală `WeightedJobScheduling` apelez două metode importante:

- `leadsToOptWithoutJobI`. Metoda `leadsToOptWithoutJobI` returnează soluția parțială optimă în cazul în care soluția parțială optimă ce conține activitatea `i` are profitul mai mic sau egal decât `dp[i-1]`, profitul care s-ar obține fără adăugare activității `i`. Pe această ramură se consideră construirea soluției parțiale optime fără a alege activitatea de pe poziția `i`.

```
method leadsToOptWithoutJobI(jobs: seq<Job>, dp:seq<int>,
allSol: seq<seq<int>>, i: int, maxProfit: int,
 optParSol: seq<int>) returns
 (optimalPartialSolution: seq<int>, profits:seq<int>)
  requires validProblem(jobs)
  requires 1 <= i < |jobs|
  requires |dp| == |allSol| == i
  requires |optParSol| == i
  requires isOptParSol(optParSol, jobs, i)
  requires dp[i - 1] == PartialSolProfit(optParSol, jobs, 0)
  requires forall partialSol :: |partialSol| == i + 1
    && partialSolutionWithJobI(partialSol, jobs, i)
        ==> HasLessProf(partialSol, jobs, maxProfit, 0)
  requires forall partialSol :: |partialSol| == i
    && isPartialSol(partialSol, jobs, i)
        ==> HasLessProf(partialSol, jobs, dp[i - 1], 0);
  requires dp[i - 1] >= maxProfit
  ensures isPartialSol(optimalPartialSolution, jobs, i + 1)
  ensures isOptParSol(optimalPartialSolution, jobs, i + 1)
  ensures profits == dp + [dp[i-1]]
  ensures |profits| == i + 1
  ensures HasProfit(optimalPartialSolution, jobs, 0, profits[i])
  ensures forall partialSol :: |partialSol| == i + 1
   && isPartialSol(partialSol, jobs, |partialSol|)
    ==> HasLessProf(partialSol, jobs, profits[i], 0);
{
  assert dp[i-1] >= maxProfit;
  profits := dp + [dp[i-1]];
  assert dp[i - 1] == PartialSolProfit(optParSol, jobs, 0);
```

```
  AssociativityOfProfitFunc(optParSol, jobs, 0, 0);
  optimalPartialSolution := optParSol + [0];
  assert dp[i - 1] == PartialSolProfit(optimalPartialSolution, jobs, 0);

  assert isPartialSol(optimalPartialSolution, jobs, i + 1);
  assert partialSolutionWithoutJobI(optimalPartialSolution, jobs, i);
  optimalPartialSolutionWithoutJobI(i, jobs, maxProfit, dp, profits);

  assert isOptParSol(optimalPartialSolution, jobs, i + 1);
  assert forall partialSol :: |partialSol| == i + 1 &&
   isPartialSol(partialSol, jobs, |partialSol|)
        ==> HasLessProf(partialSol, jobs, profits[i], 0);
}
```

Lema `optimalPartialSolutionWithoutJobI` demonstrează știind că soluția parțială optimă ce conține activitatea `i` aduce un profit mai mic decât dacă nu s-ar alege activitatea de pe poziția `i`, că fără activitatea de pe poziția `i` se obține o soluție parțială optimă pentru subproblema pe care o rezolvă.

```
lemma optimalPartialSolutionWithoutJobI(i: int, jobs: seq<Job>,
 maxProfit: int, dp: seq<int>, profits: seq<int>)
  requires validJobsSeq(jobs)
  requires 1 <= |jobs|
  requires 1 <= i < |jobs|
  requires |dp| == i
  requires |profits| == i + 1
  requires profits[i] == dp[i - 1]
  requires dp[i - 1] >= maxProfit
  requires forall partialSol :: |partialSol| == i + 1
    && partialSolutionWithJobI(partialSol, jobs, i)
        ==> HasLessProf(partialSol, jobs, maxProfit, 0)
  requires forall partialSol :: |partialSol| == i
    && isPartialSol(partialSol, jobs, i)
        ==> HasLessProf(partialSol, jobs, dp[i - 1], 0);
  ensures forall partialSol :: |partialSol| == i + 1
    && isPartialSol(partialSol, jobs, |partialSol|)
        ==> HasLessProf(partialSol, jobs, profits[i], 0)
{
  forall partialSol | |partialSol| == i + 1
    && isPartialSol(partialSol, jobs, |partialSol|)
    ensures HasLessProf(partialSol, jobs, profits[i], 0)
```

```
    if partialSol[i] == 1
    {
      assert maxProfit <= dp[i - 1];
      assert profits[i] == dp[i - 1];
      assert maxProfit <= profits[i];
    }
    else
    {
      assert partialSol[i] == 0;
      assert partialSol[..i] + [0] == partialSol;
      NotAddingAJobKeepsSameProfit(partialSol[..i], jobs, 0);
      assert PartialSolProfit(partialSol[..i], jobs, 0)
       == PartialSolProfit(partialSol, jobs, 0);
      assert isPartialSol(partialSol[..i], jobs, |partialSol[..i]|);
      assert forall partialSol :: |partialSol| == i
        && isPartialSol(partialSol, jobs, i)
            ==> HasLessProf(partialSol, jobs, dp[i - 1], 0);
      assert PartialSolProfit(partialSol[..i], jobs, 0) <= dp [i - 1];
      assert PartialSolProfit(partialSol, jobs, 0) <= dp[i - 1];
      assert profits[i] == dp[i - 1];
      assert PartialSolProfit(partialSol, jobs, 0) <= profits[i];
    }
  }
}
```

- și `leadsToOptWithJobI`. Metoda `leadsToOptWithJobI` returnează soluția parțială optimă în cazul în care soluția parțială optimă ce conține activitatea i are profitul strict mai mare decât `dp[i-1]`, profitul care s-ar obține fără adăugare activității `i`. Pe această ramura soluția parțială optimă este cea care conține activitatea `i`.

```
method leadsToOptWithJobI(jobs: seq<Job>, dp:seq<int>,
 allSol: seq<seq<int>>,i: int, maxProfit: int,
 optParSolWithJobI: seq<int>)
 returns (optimalPartialSolution: seq<int>, profits:seq<int>)
  requires validProblem(jobs)
  requires 1 <= i < |jobs|
  requires |dp| == |allSol| == i
  requires isPartialSol(optParSolWithJobI, jobs, i + 1)
  requires partialSolutionWithJobI(optParSolWithJobI, jobs, i)
  requires maxProfit == PartialSolProfit(optParSolWithJobI, jobs, 0);
  requires forall partialSol :: |partialSol| == i + 1
   && partialSolutionWithJobI(partialSol, jobs, i) ==>
```

```
    HasLessProf(partialSol, jobs, maxProfit, 0)
  requires forall partialSol :: |partialSol| == i
    && isPartialSol(partialSol, jobs, i) ==>
     HasLessProf(partialSol, jobs, dp[i - 1], 0)
  requires dp[i - 1] < maxProfit
  ensures isPartialSol(optimalPartialSolution, jobs, i + 1)
  ensures |profits| == i + 1
  ensures profits == dp + [maxProfit]
  ensures isOptParSol(optimalPartialSolution, jobs, i + 1)
  ensures HasProfit(optimalPartialSolution, jobs, 0, profits[i])
  ensures forall partialSol :: |partialSol| == i + 1
    && isPartialSol(partialSol, jobs, i + 1) ==>
     HasLessProf(partialSol, jobs, profits[i], 0);
{

  profits := dp + [maxProfit];
  assert optParSolWithJobI[i] == 1;
  optimalPartialSolution := optParSolWithJobI;
  assert isPartialSol(optimalPartialSolution, jobs, i + 1);
  assert PartialSolProfit(optimalPartialSolution, jobs, 0) == maxProfit;
  assert partialSolutionWithJobI(optimalPartialSolution, jobs, i);
  optimalPartialSolutionWithJobI(i, jobs, maxProfit, dp);
  assert forall partialSol :: |partialSol| == i + 1
   && isPartialSol(partialSol, jobs, i + 1)
        ==> HasLessProf(partialSol, jobs, profits[i], 0);
  assert isOptParSol(optimalPartialSolution, jobs, i + 1);
}
```

În cadrul acestei metode am folosit o lemă importantă, `optimalPartialSolutionWithJobI`. Această lemă demonstrează că soluția parțială optimă ce conține activitatea `i` este soluție parțială optimă pentru subproblema pe care o rezolvă.

```
lemma optimalPartialSolutionWithJobI(i: int, jobs: seq<Job>,
 maxProfit: int, dp: seq<int>)
  requires validJobsSeq(jobs)
  requires 1 <= |jobs|
  requires |dp| == i
  requires 1 <= i < |jobs|
  requires dp[i - 1] < maxProfit
  requires forall partialSol :: |partialSol| == i
    && isPartialSol(partialSol, jobs, i)
```

```
            ==> HasLessProf(partialSol, jobs, dp[i - 1], 0)
    requires forall partialSol :: |partialSol| == i + 1
      && partialSolutionWithJobI(partialSol, jobs, i)
            ==> HasLessProf(partialSol, jobs, maxProfit, 0)
    ensures forall partialSol :: |partialSol| == i + 1 &&
     isPartialSol(partialSol, jobs, |partialSol|)
     ==> HasLessProf(partialSol, jobs, maxProfit, 0);
  {
    forall partialSol | |partialSol| == i + 1 &&
     isPartialSol(partialSol, jobs, |partialSol|)
      ensures HasLessProf(partialSol, jobs, maxProfit, 0);
    {
      if partialSol[i] == 1
      {
        assert forall partialSol :: |partialSol| == i + 1
         && partialSolutionWithJobI(partialSol, jobs, i)
          ==> HasLessProf(partialSol, jobs, maxProfit, 0);
      }
      else{
        NotAddingAJobKeepsSameProfit(partialSol[..i], jobs, 0);
        assert partialSol == partialSol[..i] + [0];
        assert PartialSolProfit(partialSol, jobs, 0)
          == PartialSolProfit(partialSol[..i], jobs, 0);
        SubSeqOfPartialIsAlsoPartial(partialSol, jobs, i);
        assert isPartialSol(partialSol[..i], jobs, |partialSol[..i]|);
        assert PartialSolProfit(partialSol[..i], jobs, 0) <= dp[i - 1];
        assert PartialSolProfit(partialSol, jobs, 0)  <= dp[i - 1];
        assert dp[i - 1] < maxProfit;
        assert HasLessProf(partialSol, jobs, maxProfit, 0);
      }
    }
  }
```

Un rol important în demonstrarea acestor leme l-a avut invariantul

```
invariant forall partialSol :: |partialSol| == i  && isPartialSol(partialSol, jobs, i)
==> HasLessProf(partialSol, jobs, dp[i - 1], 0)
```

din funcția principală `WeightedJobScheduling`, care ne asigură că secvența de profituri `dp` conține pentru fiecare subproblemă profiturile optime.

3.8 Mod de lucru

De-a lungul procesului de dezvoltare a codului, au existat situații în care anumite metode și leme au cauzat depășirea timpului de așteptare (timeout). Acest lucru se întâmplă atunci când acestea conțin prea multe instrucțiuni care trebuie demonstrate, având un grad de complexitate crescut sau fac parte dintr-o clasă de probleme care nu poate fi rezolvată de solverul Z3. Pentru a evita timeout-urile, soluția a constat în creșterea modularității prin refactorizarea metodelor și lemelor care aveau această problemă, extrăgând părțile de cod critice pe care le-am demonstrat separat într-o alta lemă. Prin această abordare, nu doar că am reușit să reduc timpul de demonstrare, dar am reușit și să îmbunătățesc claritatea și modularitatea codului meu, făcându-l mai ușor de întreținut și de înțeles în viitor.

În timpul dezvoltării codului, un aspect important a fost modul de lucru. Pașii pe care i-am urmat pentru a-mi ușura munca au fost:

1. utilizarea instrucțiunii `assume false` pentru a presupune că toate condițiile de verificare de după sunt adevărate (Dafny nu mai încearcă demonstrarea lor), încât să evit situațiile în care metodele deveneau fragile (duc la timeout) și nu mai puteam identifica la care linie este o eroare. De cele mai multe ori, am folosit `assume false` în cazul if-urilor unde aveam două fire de execuție pentru a vedea exact pe care din cele două nu se respectă invarianții, deoarece de cele mai multe ori obțineam timeout și era dificil de identificat unde este problema. După ce pe o ramură obțineam un cod verificat, știam sigur că dacă problema încă este prezentă, trebuie să continui cu demonstrația pe cealaltă. De exemplu, în metoda principală a fost nevoie să presupun pe rând cele doua ramuri ale if-ului fiind adevărate, pentru a putea să lucrez pe fiecare în parte, încât să văd exact ce nu este bine și de ce.

```
method WeightedJobScheduling(jobs: seq<Job>) returns (sol: seq<int>,
 profit : int)
  requires validProblem(jobs)
  ensures isSolution(sol, jobs)
  ensures isOptimalSolution(sol, jobs)
{
  ....
  while i < |jobs|
    ...
    invariant isOptParSol(solution, jobs, i)
  {
    var maxProfit, optParSolWithI :=
      MaxProfitWithJobI(jobs, i, dp, allSol);
    if dp[i-1] >= maxProfit
```

```
    {
      solution, dp :=
       leadsToOptWithoutJobI(jobs, dp, allSol, i, maxProfit, solution);
      assert isOptParSol(solution, jobs, i + 1);
    }
    else
    {
      assume false;
      solution, dp :=
       leadsToOptWithJobI(jobs, dp, allSol, i, maxProfit, optParSolWithI);
      assert isOptParSol(solution, jobs, i + 1);
    }
    allSol := allSol + [solution];
    i := i + 1;
  }
  sol := solution;
  profit := dp[|dp|-1];
}
```

2. utilizarea `assert-urilor` pentru a vedea exact de la ce linie o proprietatea nu mai este îndeplinită sau dacă am reușit să demonstrez o anumită proprietate. Assert-urile au fost instrucțiunile pe care le-am utilizat cel mai des pe parcursul dezvoltării codului în Dafny. Acestea m-au ajutat să verific proprietățile care trebuiau îndeplinite. În cazul predicatele mai complexe, definite print-o conjuncție cu mai multe propoziții, de cele mai multe ori foloseam assert pentru fiecare proprietate în parte, pentru a vedea care din ele nu este îndeplinită. De exemplu, în cazul predicatului `isPartialSol`, de fiecare dată când făceam o modificare asupra unei soluții parțiale, verificam separat toate proprietățile sale.

```
predicate isPartialSol(partialSol: seq<int>, jobs: seq<Job>, length: int)
  requires validJobsSeq(jobs)
{
  |partialSol| == length &&
  forall i :: 0 <= i <= |partialSol| - 1 ==> (0 <= partialSol[i] <= 1)
   && hasNoOverlappingJobs(partialSol, jobs)
}
```

În metoda `MaxProfitWithJobI` după ce am modificat variabila `partialSolution` am verificat dacă aceasta îndeplinește proprietatea de soluție parțială.

```
method MaxProfitWithJobI(jobs: seq <Job>, i: int, dp: seq<int>,
 allSol :seq<seq<int>>)
 returns (maxProfit:int, optParSolWithI: seq<int>)
```

```
  ...
  ensures forall partialSol :: |partialSol| == i + 1 &&
   partialSolutionWithJobI(partialSol, jobs, i) ==>
    HasLessProf(partialSol, jobs, maxProfit, 0)
{

  var max_profit := 0;
  var partialSolutionPrefix : seq<int> := [];
  var partialSolution : seq<int> := [];
  var j := i - 1;
  var length := 0;

  while j >= 0 && jobs[j].jobEnd > jobs[i].jobStart //
    ...
    invariant forall k :: j < k < i ==> overlappingJobs(jobs[k], jobs[i])
  {
    j := j - 1;
  }
  assert  j != -1 ==> !overlappingJobs(jobs[j], jobs[i]);
  if j >= 0
  {
    max_profit, partialSolution, length :=
     OptParSolWhenNonOverlapJob(jobs, i, dp, allSol, j);
  }
  else
  {
    max_profit, partialSolution, length :=
    OptParSolWhenOverlapJob(jobs, i, dp);
  }
  assert forall k :: 0 <= k < length ==> 0 <= partialSolution[k] <= 1;
  assert hasNoOverlappingJobs(partialSolution, jobs);
  assert |partialSolution| == length;
  assert isPartialSol(partialSolution, jobs, length);
  ...
}
```

Au fost și cazuri în care a trebuit să folosesc assert-uri deoarece Dafny are nevoie de indicii să demonstreze proprietăți care păreau evidente pentru o persoană, dar care fac parte dintr-o clasă de formule nedecidabilă. De exemplu assert-ul `assert partialSol == partialSol[..i] + [0]` din următoarea bucată de cod este necesar:

```
forall partialSol | |partialSol| == i + 1 &&
 isPartialSol(partialSol, jobs, |partialSol|)
    ensures HasLessProf(partialSol, jobs, maxProfit, 0);
```

```
{
  if partialSol[i] == 1
  {
    ...
  }
  else{
    NotAddingAJobKeepsSameProfit(partialSol[..i], jobs, 0);
    assert partialSol == partialSol[..i] + [0];
    ...
    assert HasLessProf(partialSol, jobs, maxProfit, 0);}
```

Concluzii

În cadrul acestei lucrări de licență, am explorat și demonstrat în Dafny corectitudinea algoritmului de selecție a activităților cu profit maxim, utilizând programare dinamică. Prin intermediul limbajului imperativ Dafny, am reușit să implementez algoritmul și să verific formal corectitudinea acestuia.

Utilizarea Dafny a fost esențială pentru a garanta că implementarea respectă toate cerințele de corectitudine, prin verificarea semiautomată a invarianților și a proprietăților specificate. Rezultatele obținute au confirmat faptul că algoritmul este corect.

3.9 Statistici

Implementarea este structurată într-un singur script:

Nume	Număr de linii	Scop
day.dfy	928	Verificarea algoritmului

Liniile de cod de mai sus, includ specificații, leme, assert-uri ajutătoare, comentarii și linii de delimitare. Dezvoltarea conține un total de 34 metode, leme, funcții și predicate de verificat. Verificarea durează aproximativ 8 secunde pe un calculator modern (2.60GHz 11th Gen Intel(R) Core(TM) i5-1145G7, 16 GB) cu un singur fir de execuție.

Iată timpii de verificare pentru metodele, lemele și predicatele folosite la verificarea algoritmului:

Tabela 3.1: Timpii de verificare

Verificare	**Timp (s)**
Running abstract interpretation	0.0605648
validProblem	0.115
distinctJobsSeq	0.033
sortedByActEnd	0.037

Verificare	**Timp (s)**
hasNoOverlappingJobs	0.039
PartialSolProfit	0.042
isPartialSol	0.044
isOptParSol	0.051
HasProfit	0.056
isOptParSolDP	0.051
OptParSolutions	0.055
isSolution	0.048
isOptimalSolution	0.047
containsOnlyZeros	0.031
partialSolutionWithJobI	0.041
partialSolutionWithoutJobI	0.045
HasLessProf	0.041
HasMoreProfit	0.036
ProfitParSolStartFinishPos	0.037
AssociativityOfProfitFunc (well-formedness)	0.061
AssociativityOfProfitFunc (correctness)	0.049
EqOfProfitFuncFromIndToEnd (well-formedness)	0.038
EqOfProfitFuncFromIndToEnd (correctness)	0.040
EqOfProfFuncUntilIndex (well-formedness)	0.036
EqOfProfFuncUntilIndex (correctness)	0.047
SplitSequenceProfitEquality (well-formedness)	0.034
SplitSequenceProfitEquality (correctness)	0.425
ProfitLastElem (well-formedness)	0.041
ProfitLastElem (correctness)	0.043
ComputeProfitWhenOnly0BetweenJI (well-formedness)	0.043
ComputeProfitWhenOnly0BetweenJI (correctness)	2.988
HasMoreProfThanOptParSol (well-formedness)	0.046
HasMoreProfThanOptParSol (correctness)	0.047
OtherSolHasLessProfThenMaxProfit2 (well-formedness)	0.096
OtherSolHasLessProfThenMaxProfit2 (correctness)	0.190
OnlY0WhenOverlapJobs (well-formedness)	0.045
OnlY0WhenOverlapJobs (correctness)	0.045
OptParSolWhenNonOverlapJob (well-formedness)	0.067

Verificare	Timp (s)
OptParSolWhenNonOverlapJob (correctness)	0.321
OtherSolHasLessProfThenMaxProfit (well-formedness)	0.044
OtherSolHasLessProfThenMaxProfit (correctness)	0.059
OptParSolWhenOverlapJob (well-formedness)	0.080
OptParSolWhenOverlapJob (correctness)	0.392
MaxProfitWithJobI (well-formedness)	0.056
MaxProfitWithJobI (correctness)	0.135
NotAddingAJobKeepsSameProfit (well-formedness)	0.039
NotAddingAJobKeepsSameProfit (correctness)	0.093
SubSeqOfPartialIsAlsoPartial (well-formedness)	0.039
SubSeqOfPartialIsAlsoPartial (correctness)	0.040
optimalPartialSolutionWithJobI (well-formedness)	0.055
optimalPartialSolutionWithJobI (correctness)	0.158
leadsToOptWithJobI (well-formedness)	0.081
leadsToOptWithJobI (correctness)	0.117
optimalPartialSolutionWithoutJobI (well-formedness)	0.052
optimalPartialSolutionWithoutJobI (correctness)	0.122
leadsToOptWithoutJobI (well-formedness)	0.085
leadsToOptWithoutJobI (correctness)	0.107
WeightedJobScheduling (well-formedness)	0.062
WeightedJobScheduling (correctness)	0.268
Main (correctness)	0.068
Total	**7.9045648**

Putem observa că timpul de verificare este foarte bun, valoarea cea mai mare fiind 2 secunde în cazul lemei `ComputeProfitWhenOnly0BetweenJI`. Această lema demonstrează pentru o soluție parțială care conține activitatea de pe poziția `i`, că profitul de la poziția `j + 1` până la poziția `i` este `jobs[i].profit`, știind că nicio activitate nu a fost selectată pe aceste poziții (avem doar 0-uri).

```
lemma ComputeProfitWhenOnly0BetweenJI(partialSol: seq<int>, jobs: seq<Job>, i: int,
j: int)
  requires validJobsSeq(jobs)
  requires 0 <= j < i < |partialSol| <= |jobs|
  requires |partialSol| == i + 1
  requires isPartialSol(partialSol, jobs, |partialSol|)
```

```
  requires partialSolutionWithJobI(partialSol, jobs, i)
  requires PartialSolProfit(partialSol, jobs, i) == jobs[i].profit
  requires forall k :: j < k < i ==> partialSol[k] == 0
  ensures PartialSolProfit(partialSol, jobs, j + 1) == jobs[i].profit
  decreases i - j
{
  if j == i - 1
  {
    assert PartialSolProfit(partialSol, jobs, j + 1) == jobs[i].profit;
  }
  else
  {
    ComputeProfitWhenOnly0BetweenJI(partialSol, jobs, i, j + 1);
  }

}
```

Iată rezumatul întregii dezvoltări:

Categorie	**Număr**
Metode	7
Predicate	21
Funcții	2
Leme	14
Precondiții	148
Postcondiții	49
Invarianți	46
Assert-uri	129
Comentarii	155

Procesul de dezvoltare a codului a durat aproximativ 7 luni, unde prima lună a fost dedicată introducerii în limbajul de programare Dafny. Pe parcurs am învățat foarte multe lucruri noi: începând cu structuri de date din Dafny, ce sunt invarianții, care sunt limitele pentru invarianți, ce sunt metodele, funcțiile și predicatele și continuând până la cum să evit construirea unor metode fragile, cum să folosesc assert-urile și assume-urile încât să-mi dau seama ce nu reușește Dafny să demonstreze. Lemele au fost la început partea puțin mai dificilă, deoarece nu reușeam să le generalizez.

Dacă ar fi ceva de îmbunătățit la Dafny, eu consider că ar fi de mare folos să ofere sugestii, exemple în cazurile în care ceva lipsește sau nu este bine.

Pe baza acestei lucrări, consider că există oportunități semnificative pentru cercetări viitoare, cum ar fi extinderea tehnicii pentru probleme mai complexe sau adaptarea algoritmului pentru a lucra într-un context paralel sau distribuit. De asemenea, integrarea unor alte limbaje și instrumente de verificare formală ar putea oferi perspective noi și valoroase.

Codul sursă este disponibil la adresa `https://github.com/roxana413/Licenta`.

În concluzie, am demonstrat că algoritmul de selecție a activităților poate fi implementat corect folosind programarea dinamică și verificarea formală în Dafny, contribuind astfel la îmbunătățirea metodelor și tehnologiilor utilizate în dezvoltarea de algoritmi siguri și fiabili.

Bibliografie

C. Andrici and Ş. Ciobâcă. Verifying the DPLL algorithm in Dafny. In M. Marin and A. Craciun, editors, *Proceedings Third Symposium on Working Formal Methods, FROM 2019, Timişoara, Romania, 3-5 September 2019*, volume 303 of *EPTCS*, pages 3–15, 2019. doi: 10.4204/EPTCS.303.1. URL https://doi.org/10.4204/EPTCS.303.1.

J. Blázquez, M. Montenegro, and C. Segura. Verification of mutable linear data structures and iterator-based algorithms in Dafny. *J. Log. Algebraic Methods Program.*, 134:100875, 2023. doi: 10.1016/J.JLAMP.2023.100875. URL https://doi.org/10.1016/j.jlamp.2023.100875.

L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

J. Koenig and K. R. M. Leino. Getting started with Dafny: A guide. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 152–181. IOS Press, 2012. doi: 10.3233/978-1-61499-028-4-152. URL https://doi.org/10.3233/978-1-61499-028-4-152.

R. Leino. *Dafny: A Language and Program Verifier for Functional Correctness*. Microsoft Research, 2021. URL https://dafny.org/latest/DafnyRef/DafnyRef. Accessed: June 2024.

D. Lucanu and Ş. Ciobâcă. Lecture 11: Dynamic programming. URL https://sites.google.com/view/fii-pa/2022/lectures. Accessed: June 2024.