UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Verificarea problemei de selecție a activităților cu profit maxim în Dafny

propusă de

Roxana Mihaela Timon

Sesiunea: iulie, 2024

Coordonator științific

Conf. Dr. Ciobâcă Ștefan

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea problemei de selecție a activităților cu profit maxim în Dafny

Roxana Mihaela Timon

Sesiunea: iulie, 2024

Coordonator științific

Conf. Dr. Ciobâcă Ștefan

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Ciobâcă Ștefan.

Data: .... Semnătura: ....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Timon Roxana Mihaela** domiciliat în **România, jud. Vaslui, sat. Valea-Grecului, str. Bisericii, nr. 24**, născut la data de **09 iulie 2000**, identificat prin CNP **6000709375208**, absolvent al Facultății de informatică, **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 problemei de selecție a activităților cu profit maxim în Dafny** elaborată sub îndrumarea domnului **Conf. Dr. Ciobâcă Ștefan**, 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.

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 problemei 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 sa fac această temă deoarece Dafny era un limbaj de programare nou pentru mine si am considerat a fi o provocare. Știam doar că acesta este folosit pentru a asigura o mai mare siguranță si corectitudine, putând fi aplicat in industria aerospațiala, în industria medicala, la dezvoltarea sistemelor financiare, în securitate și criptografie. Totodată, prin demonstrarea corectitudinii unui algoritm in Dafny puteam să-mi folosesc pe langă cunostințele informatice si pe cele de matematică, de care am fost mereu atrasă. Pentru a crește gradul de complexitate al lucrării am decis sa folosesc ca și tehnică de proiectare a aloritmilor 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.

Intenție

În cadrul lucrării voi discuta despre limbajul de programare Dafny, surpinzând particularitătile sale, despre problema de selecție a activităților cu profit maxim, despre programarea dinamică și avantajele sale în comparație cu alte tehnici de proiectare a algoritmilor. Voi prezenta, de asemenea, demonstrația de corectitudine a problemei de selecție a activităților cu profit maxim folosind programarea dinamică în Dafny.

Introducere

Lucrarea este structură în 3 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.
- **Problema de selecție a activităților cu profit maxim**. Acest capitol conține prezentarea algoritmului de selecție a activităților cu profit maxim, demonstrația de corectitudine cu ajutorul limbajului de programare Dafny și tehnica de lucru abordată.

Capitolul 1

Dafny

1.1 Prezentare generală

Dafny este un limbaj imperativ de nivel înalt cu suport pentru programarea orientată pe obiecte. Metodele realizate în Dafny au precondiții, postcondiții și invarianți care sunt verificate la compilare, bazându-se pe soluționatorul SMT Z3. În cazul în care o postcondiție nu poate fi stabilită (fie din cauza unui timeout, fie din cauza faptului că aceasta nu este valabilă), 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 unui cod corect, în sensul de a nu avea erori de execuție, dar și corect în sensul de a face ceea ce programatorul a intenționat să facă.[1] Dafny, ca platforma 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.[2]

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. 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. [4]

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.[4] Avantajele programării dinamice fața de tehnica greedy sunt:

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

- **Flexibilitate**: Programarea dinamică poate fi utilizată pentru o gamă mai largă de probleme, inclusiv cele care implică restricții mai complexe sau soluții care necesită evaluarea mai multor posibilități. În comparație, algoritmii greedy sunt adesea limitați la problemele care pot fi rezolvate prin luarea deciziilor locale în fiecare pas.

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 care returneaza pentru o listă de activități distincte (difera prin cel putin un timp) caracterizate prin timp de început, timp de încheiere și profit, ordonate dupa timpul de încheiere, o secventă de activități care nu se suprapun doua cate doua și care produc un profit maxim.

```
Input: O secventa de activitati caracterizate prin
{ timp de inceput, timp de incheiere,  profit}
   Activitate 1:  {1, 2, 50}
   Activitate 2:  {3, 5, 20}
   Activitate 3:  {6, 19, 100}
   Activitate 4:  {2, 100, 200}
Output: Profit-ul maxim este 250, pentru solutia optima formata din
activitatea 1 si activitatea 4.
```

2.3 Pseudocod

```

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] = [activitati[0]] //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)

        // Calculeaz  profitul maxim incluzand activitatea curenta si excluzand-o
        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 Cum funcționeaza programarea dinamică în cazul problemei de selectie a activitatilor

Pentru problema de selectie a activitatilor se poate folosi programarea dinamică, deoarece aceasta poate fi împarțită în subprobleme, respectiv la fiecare pas putem forma o soluție parțiala optimă, de al carei profit ne putem folosi la urmatorul pas.

2.4.1 Solutia partiala si solutia partiala optima

O solutie partiala trebuie sa contina activitati care nu se suprapun si sa aiba o lungime prestabilita. O solutie partiala este si optima, daca profitul acesteia este mai mare sau egal decat al oricarei alte solutii partiale de aceeasi lungime.

Pentru datele de intrare precizate de mai sus se obtin urmatoarele valori:
La primul pas:

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

La al 2-lea pas:

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

La al 3-lea pas:

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

La al 4-lea pas:

1. soluția parțială ce conține Activitatea 4 este formata din Activitatea 4 si Activitatea 1.

2. profitul pentru aceasta soluție parțiala este strict mai mare decat 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 si avand profitul optim 250.

2.4.2 Proprietatea de substructura optimă în cazul problemei de selecție a activităților

În cazul problemei de selecție a activităților, proprietatea de substructură optimă este identificata atunci cand eliminam o activitate dintr-o soluție parțială optimă si obținem tot o soluție parțiala optimă. Altfel, dacă acest proprietate nu ar fi valabila, înseamna ca soluția parțială optimă pe care o descompunem nu este cu adevărat optimă.

Capitolul 3

Verificarea problemei de selecție a activităților in Dafny

3.1 Reprezentarea datelor de intrare si a celor de ieșire

3.1.1 Date de intrare și predicate specifice

Pentru a reprezenta o activitate am folosit un tuplu.

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

Problema de selectie a activitatilor are ca date de intrare un singur parametru:

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

Un predicat este o funcție care returnează o valoare booleana.[3] Acestea sunt folosite ca precondiții și postcondiții în metode. Predicatele pe care le-am folosit la validarea datelor de intrare sunt:

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

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

Predicatul **validJobsSeq(jobs: seq<Job>)** asigură faptul ca secvența de activități primită ca data de intrare conține doar activitati valide, ceea ce înseamnă că timpul de început este anterior timpului de încheiere și că profitul asociat fiecărei activități este un număr pozitiv.

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


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

Predicatul **sortedByActEnd(s: seq<Job>)** ne asigură că in secvența de activități primită activitatile sunt sortate din punct de vedere al timpului de încheiere.

```
predicate  distinctJobs(j1: Job, j2: Job)
  requires validJob(j1) && validJob(j2)
{
  j1.jobStart != j2.jobStart || j1.jobEnd != j2.jobEnd
}


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

Pentru a ne asigura că secvența de activități primită ca input nu conține activități identice am folosit predicatul **distinctJobsSeq(s: seq<Job>)**.

Aceste 3 predicate le-am unit intr-unul singur, **validProblem**.

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

```
    && distinctJobsSeq(jobs)
}
```

3.1.2 Variabile folosite pentru rezolvare

- **solution** de tipul *seq<int>* : reprezintă soluția parțială optima obtinuta la fiecare iteratie
- **partialSol** de tipul *seq<int>* : reprezintă o soluție parțială
- **dp** de tipul *seq<int>*: ce conține toate profiturile optime obținute la fiecare pas
- **allSol** de tipul *seq<seq<int>>* : reprezintă soluțiile parțiale optime obținute la fiecare pas

Predicatele pe care le-am folosit pentru a verifica corectitudinea valorilor variabilelor intermediare folosite sunt:

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

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])
}

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)
}
```

Cu ajutorul acestui predicat, **isPartialSol(partialSol: seq<int>, jobs: seq<Job>, length: int)** am verificat ca o soluție parțială să aibă lungimea dorita, să conțină doar valori de 0 și 1, iar activitățile care corespund acestui vector caracteristic să nu se suprapună.

```
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
}

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 **isOptParSol(partialSol: seq<int>, jobs: seq<Job>, length: int)** are rolul de a verifica dacă o soluție parțiala este și optimă. Pentru ca aceasta să fie optimă trebuie să îndeplinească urmatoarele condiții:

1. să fie o soluție parțială
2. orice altă soluție parțială are un profit mai mic decât soluția parțială optimă.

```
ghost predicate isOptParSolDP(partialSol: seq<int>, jobs: seq<Job>,
 length : int, dp:int)
  requires validJobsSeq(jobs)
  requires 1 <= |partialSol|
```

```
  requires 1 <= length <= |jobs|
{
  |partialSol| == length && isOptParSol(partialSol, jobs, length)
   && HasProfit(partialSol, jobs, 0,  dp)
}

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])
}
```

Pentru a putea verifica dacă allSol conține doar soluții parțiale optime, am folosit predicatul **OptParSolutions(allSol: seq<seq<int>>, jobs: seq<Job>, dp:seq<int>, index: int)**. Acest predicat verifică pentru o secvență de secvențe:

1. fiecare secventă să aibă lungimea dorită
2. fiecare secventă să fie o soluție parțială optimă
3. fiecare secventă să aibă profitul dorit (optim)

3.2 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 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)
{
```

```
    var maxProfit, partialSolWithI := 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, partialSolWithI);
      assert isOptParSol(solution, jobs, i + 1);

    }
    allSol := allSol + [solution];
    i := i + 1;
  }

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

Metoda care imi genereaza solutia partiala ce contine activitatea i, este **MaxProfitWithJobI**. In Dafny, metodele sunt subrutine care pot fi executate în timpul execuției și pot avea efecte secundare, cum ar fi modificarea obiectelor heap.[2] Aceasta metoda primeste ca parametri:

1. **jobs** : secventa de activitati (problema)
2. **i** : pozitia (din secventa de activitati primita ca data de intrare) pe care se afla activitatea cu care vrem sa formam o solutie partiala
3. **dp** : secventa cu profiturile optime obtinute la pasii anteriori

4. **allSol**: secventa cu solutiile partiale optime obtinute la pasii anteriori

Si returneaza:

1. **maxProfit**: profitul pentru solutia partiala ce contine activitatea de pe pozitia i din secventa de activitati primita ca data de intrare

2. **optParSolWithI**: solutia partiala optima ce contine activitatea de pe pozitia 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 PositiveProfitsDP(dp)
  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;
}
```

3.2.1 Detalii de implementare

Cum functioneaza algoritmul:

1. **Date de intrare**: primim ca date de intrare variabila activitati care reprezinta o secventa de activitati caracterizate prin timp de inceput, timp de incheiere si profit, distincte (difere prin cel putin un timp), sortate dupa timpul de incheiere

2. **Programare dinamica**:

   La fiecare iteratie stocam solutiile optime ale subproblemelor si profiturile acestora.

   - variabila solution, care reprezinta solutia partiala optima la fiecare pas, respectiv solutia optima finala

- variabila dp, care va contine profiturile optime obtinute la fiecare iteratie
- variabila allSol care va contine toate solutiile partiale optime obtinute la fiecare pas

3. **Iteratii**:

- **La prima iteratie** solution = [1], dp = activitati[1].profit, allSol = [[1]]
- **Selectăm activitatea de pe pozitia i**, unde i = 1... |activitati| - 1, în ordinea în care au fost declarate în secvența de intrare
- **Formăm soluția parțială ce conține activitatea de pe pozitia i**
  - **profitul** pentru solutia partiala cu acitivitatea i are valoarea initiala **activitati[i].profit**
  - **cautam o activitate j**, unde 0 <= j < i, care nu se suprapune cu activitatea i, adica activitati[j].timpDeIncheiere <=activitati[i].timpDeInceput
  - **daca j** >= **0** atunci solutia partiala cu activitatea de pe pozitia i va fi formata din allSol[j]+ 0...0 + 1, unde:
    * allSol[j] contine solutia partiala optima cu activitatile de pana la pozitia j inclusiv
    * 0-urile reprezinta, **daca exista**, activitatile dintre j si i care se suprapun cu activitatea de pe pozitia i
    * 1 - inseamna ca activitatea i este selectata

    iar **profitul** pentru solutia partiala ce contine activitatea i este egal cu **dp[j] + activitati[i].profit**
  - **daca j** == **-1** atunci solutia partiala cu activitatea de pe pozitia i va fi formata din 0...0 + 1, unde:
    * 0-urile reprezinta, activitatile din fata activitatii i care se suprapun cu aceasta.
    * 1-inseamna ca activitatea de pe pozitia i a fost selectata

    iar **profitul** ramane egal cu **activitati[i].profit**
- **Comparam** profitul solutiei partiale ce contine activitatea de pe pozitia i cu profitul care s-ar obtine fara activitatea curenta (dp[i-1]) si **actualizam** valoarea variabilelor dp si solution

    - daca dp[i-1] >= profitul solutiei partiale cu activitatea i, atunci dp[i] = dp[i-1] si solution = solution + [0]
    - daca dp[i-1] $<$ profitul solutiei partiale cu activitatea i, atunci dp[i] = profitul solutiei partiale cu activitatea i si solution = solutia partiala ce contine activitatea i
  - **Actualizam valoarea variabila allSol** care retine solutiile partiale optime obtinute la fiecare iteratie, allSol = allSol + [solution]

4. **Datele de iesire ale problemei** sunt reprezentate de solutia optima **sol** si de profitul maxim, **profit** $=$ **dp[|activitati| - 1]**

3.2.2 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 rezolvator SMT care verifică dacă acestea sunt valide.[2]

Preconditiile reprezintă conditii care trebuie sa fie indeplinite la intrarea intr-o metoda sau functie.

Postconditiile reprezinta conditii care trebuie sa fie indeplinite la iesirea dintr-o metoda sau functie.

Invariantii, la fel ca si preconditiile si postconditiile, reprezinta conditii care trebuie sa fie indeplinite la intrarea in bucla, in timpul buclei si la iesirea din aceasta.

In cazul problemei de selectie a activitatilor preconditiile care trebuie indeplinite sunt urmatoarele:

```
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)
}
```

Astfel, secvența de activități primită ca input trebuie să conțină doar activități valide, distincte și sortate dupa timpul de încheiere (problema valida). La final, când algoritmul se termină, soluția returnată trebuie să îndeplinească urmatoarele postcondiții: ca aceasta reprezinta o solutie si ca aceasta este optima.

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

O solutie inseamna ca secventa returnata sa indeplineasca proprietatea de solutie partiala, dar cu lungimea egala cu cea a secventei de activitati primita ca data de intrare.

La fiecare iterație a algoritmului, următoarele proprietati 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
  && isPartialSolution(partialSol, jobs, i) ==>
  HasLessProfit(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 si:

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

care ne asigura ca variabila solution retine la fiecare pas atat o solutie partiala, cat si o solutie partiala optima.

3.3 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.[2] Una din funcțiile pe care am folosit-o cel mai des este funcția: *PartialSolutionPrefixProfit(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)
}
```

3.3.1 Date de ieșire și predicate specifice

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

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


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)
}
```

Predicatul **isSolution(solution: seq<int>, jobs: seq <Job>)** este asemănător cu predicatul **isPartialSolution(partialSol: seq<int>, jobs: seq<Job>, length: int)**, doar că după cum se poate observa din antetul acestuia, lipseste variabila length, deoarece acum lungimea soluției trebuie sa fie egală cu lungimea secvenței de activități primită ca data de intrare. Același lucru se aplică și pentru predicatul **isOptimalSolution(solution: seq<int>, jobs: seq<Job>)** care verifică:

1. secvența primită să fie o soluție
2. orice altă secvență care este o soluție să aibă un profit mai mic decât aceasta

3.3.2 Descrierea datelor de iesire

Variabila de ieșire, denumită ”sol”, este reprezentată printr-un vector caracteristic al secvenței de activități primita 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.

3.4 Leme importante in demonstrarea corectitudinii

Unul din punctele complexe in dezvoltarea codului de verificare pentru problema de selectie a activitatilor a fost atunci când a trebuit să recurg la leme pentru a demonstra anumite proprietăți. Lemele reprezintă blocuri de instructiuni care au ca scop demonstrarea unor teoreme pe care Dafny nu reuseste sa le demonstreze de unul singur. [3]

Lemma **OtherSolHasLessProfitThenMaxProfit2** am folosit-o pentru a demonstra ca orice alta **solutie partiala** care:

1. **contine activitatea de pe pozitia i**
2. orice activitatea aflata pe oricare din pozitiile *j + 1, ..., i - 1* **se suprapune cu activitatea i**, unde $0 <= j < i$
3. activitatea de pe pozitia **j nu se suprapune** cu activitatea de pe pozitia **i**

are un **profit mai mic sau egal** decat cel al solutiei partiale care indeplineste aceleasi proprietati mentionate mai sus, doar ca pentru aceasta in plus **substructura formata din activitatile de pe pozitia 0 pana la pozitia j este optima**.

```
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;

  //presupunem contrariul
  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; //(2)

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

    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);

}
```

Aceasta lema am folosit-o in cadrul metodei **OptParSolWhenNonOverlapJob** (linia 79) pe care o apelez in metoda **MaxProfitWithJobI** pe ramura cu *j >= 0*, adica atunci cand gasim o activitate pe o pozitie j in fata activitatii de pe pozitia 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 aceasta lema am presupus ca ar exista o alta solutie partiala cu aceleasi proprietati care ar avea un profit mai mare decat solutia partiala optima obtinuta. Astfel, in urma calculelor, am gasit ca exista o alta solutie partiala de lungime j + 1 al carui profit este mai mare decat cel al solutiei partiale optime de lungime j + 1, ceea ce este imposibil, deoarece ar contrazice ipoteza de la care am plecat. In acest mod, am identificat si **proprietatea de substructura optima**, care se ascunde in aceasta lema.

```
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|);
}
```

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.

De asemenea, consider că și lemele pe care le-am demonstrat folosind inducția au fost la fel de complexe, 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);
  }
}
```

In cadrul acestei leme se poate observa cum am demonstrat proprietatea de asociativitate a functiei PartialSolutionPrefixProfit, care imi calculeaza profitul pentru o solutie partiala, astfel incat daca adaugam un element la capatul unei solutii partiale, valoarea profitului acesteia creste cu valoarea profitului activitatii adaugate.

3.5 Mod de lucru

In timpul dezvoltarii codului, un aspect important a fost modul de lucru. Pasii pe care i-am urmat pentru a-mi usura munca au fost:

1. instructiunea **assume false** pentru a prespune anumite bucati de cod a fi false (Dafny nu mai incerca demonstrarea lor), incat sa evit situatiile in care metodele deveneau fragile (duc la timeout) si nu mai puteam identifica la care linie este o eroare

2. utilizarea **assert-urilor** pentru a vedea exact de la ce linie o proprietatea nu mai este indeplinita sau daca am reusit sa demonstrez o anumita proprietate

3.5.1 Timeout

De-a lungul procesului de dezvoltare a codului, au existat situații în care anumite metode si leme au cauzat depășirea timpului de așteptare (timeout). Acest lucru se intampla atunci cand acestea contin prea multe instructiuni care trebuie demonstrate, avand un grad de complexitate crescut. Pentru a evita timeout-urile, solutia a constat in refactorizarea metodelor si lemelor care aveau aceasta problema, extragand partile de cod critice pe care le-am demonstrat separat intr-o alta lema. Prin această abordare, nu doar că am reușit să reduc timpul de demonstrare, dar am reusit și să îmbunătățesc claritatea și modularitatea codului meu, făcându-l mai ușor de întreținut și de înțeles în viitor.

Concluzii

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Nunc mattis enim ut tellus elementum sagittis vitae et. Placerat in egestas erat imperdiet sed euismod. Urna id volutpat lacus laoreet non curabitur gravida. Blandit turpis cursus in hac habitasse platea. Eget nunc lobortis mattis aliquam faucibus. Est pellentesque elit ullamcorper dignissim cras tincidunt lobortis feugiat. Viverra maecenas accumsan lacus vel facilisis volutpat est. Non odio euismod lacinia at quis risus sed vulputate odio. Consequat ac felis donec et odio pellentesque diam volutpat commodo. Etiam sit amet nisl purus in. Tortor condimentum lacinia quis vel eros donec. Phasellus egestas tellus rutrum tellus pellentesque eu tincidunt. Aliquam id diam maecenas ultricies mi eget mauris pharetra. Enim eu turpis egestas pretium.

Bibliografie

[1] 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.

[2] 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.

[3] 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.

[4] D. Lucanu and Ş. Ciobâcă. Lecture 11: Dynamic programming.