UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Verificarea unui algoritm pentru varianta discretă a Problemei Rucsacului în Dafny

propusă de

Alina-Adriana Haidău

Sesiunea: februarie, 2025

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea unui algoritm pentru varianta discretă a Problemei Rucsacului în Dafny

Alina-Adriana Haidău

Sesiunea: februarie, 2025

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 **Haidău Alina-Adriana** domiciliat în **România, jud. Suceava, com. Bălăceana, str. Principală, nr. 97**, născut la data de **16 iulie 2000**, identificat prin CNP **6000716330223**, 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 unui algoritm pentru varianta discretă a Problemei Rucsacului 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.

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 varianta discretă a Problemei Rucsacului 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 **Alina-Adriana Haidău**

Data: ..... Semnătura: .....

Cuprins

Motivație

Am ales să implementez și să demonstrez corectitudinea unui algoritm pentru varianta discretă a problemei rucsacului deoarece întotdeauna am considerat că este o problemă interesantă care de foarte multe ori are aplicabilitate practică și în viața reală. În cazul variantei discrete, abordarea bazată pe programarea dinamică a fost motivată de eficiența pe care această tehnică de rezolvare o oferă în construirea soluțiilor prin identificarea subproblemelor, minimizând astfel redundanța în calcul, dar și garantarea că la finalul algoritmului soluția obținută este mereu optimă.

De asemenea, alegerea acestui limbaj a fost motivată de faptul că nu am mai lucrat anterior în Dafny și am considerat că este o oportunitate de a-mi extinde cunoștințele și de a explora noi limbaje cu care nu am interacționat prea mult. De asemenea, am găsit acest limbaj foarte interesant datorită capacității sale de a verifica riguros proprietăți precum corectitudinea și optimalitatea.

Introducere

În acest capitol introductiv voi face o scurtă prezentare a limbajului Dafny, urmată de câteva puncte esențiale despre paradigma programării dinamice și câteva detalii despre Problema Rucsacului.

Limbajul de programare Dafny

Dafny este un limbaj de programare funcțional destinat verificării formale a corectudinii programelor. A fost creat pentru a ajuta programatorii să scrie cod care este corect din punct de vedere al specificațiilor. Un lucru foarte interesant de știut despre acest limbaj este faptul că a fost conceput astfel încât permite verificarea codului încă din faza de dezvoltare, înainte de a rula codul. Datorită faptului că verificatorul Dafny este rulat ca parte a compilatorului, orice eroare matematică sau logică va fi semnalată către programator care va trebui să ajute verificatorul prin ajustarea specificațiilor sau a logicii.

Programarea dinamică

Programarea dinamică este o metodă de rezolvare a problemelor complexe și se bazează pe conceptul de suprapunere al subproblemelor, însemnând faptul că o problemă poate fi ”spartă” în mai multe subprobleme mai mici care sunt mai ușor de rezolvat și care se repetă pe parcursul execuției algoritmului. Soluțiile acestor subprobleme sunt stocate astfel încât să nu fie necesară recalcularea lor, lucru care îmbunătățește timpul de rezolvare al algoritmului.

Principalele abordări prin care soluțiile sunt stocate când folosim programarea dinamică sunt:

- **Memoizarea (Top-Down)** este abordarea recursivă, în care rezultatele sunt salvate într-un tabel de unde vor fi accesate ulterior când este nevoie de rezultatul deja calculat;

- **Tabelizarea (Bottom-Up)** este abordarea în care se pornește de la calcularea celor mai mici subprobleme, construind treptat soluția finală din soluțiile subproblemelor mai mici.

În general, paradigma programării dinamice este folosită pentru a rezolva problemele de optimizare care au ca obiectiv obținerea celei mai bune soluții într-un cadru de optimizare, fie pentru minimizare, fie pentru maximizare.

Ce este Problema Rucsacului?

Problema rucsacului este o problemă de optimizare. Presupunând că avem un rucsac care poate susține o anumită greutate maximă numită capacitate, trebuie să alegem dintr-o mulțime mai largă de obiecte un număr limitat astfel încât valoarea totală a acestora este maximă, iar greutatea totală a obiectelor nu depășește capacitatea rucsacului. Cele mai cunoscute variațiuni ale acestei probleme sunt varianta continuă și varianta discretă.

În varianta continuă, obiectele pot fi fracționate în mai multe bucăți, permițând alegerea parțială a unui obiect. Pentru această variantă a problemei, algoritmii greedy sunt mai potriviți datorită deciziilor bazate pe un criteriu local precum raport valoare/greutate, greutate mică sau câștig mare a obiectului, astfel eliminând riscul de a pierde combinații de obiecte.

În varianta discretă fiecare obiect poate fi ales o singură dată sau deloc. Pentru varianta discretă abordările greedy pot fi aplicate, însă acestea nu pot garanta că la finalul execuției algoritmului soluția obținută este optimă. Acest lucru este datorat naturii acestor algoritmi de a alege varianta cea mai profitabilă în momentul curent și ignoră alte combinații de obiecte care individual au un profit mai mic, dar împreună ar fi mult mai profitabile, de aceea acest tip de probleme necesită o abordare mai complexă precum programarea dinamică.

Formularea problemei

Pentru a înțelege mai bine ideea problemei și cum funcționează algoritmul vom considera ca exemplu următoarea instanță a problemei, pe care am folosit-o în cadrul procesului de implementare:

Date de intrare:

• *n = 4*, unde *n* este numărul total de obiecte pe care le avem la dispoziție;

• *c = 8*, unde *c* reprezintă capacitatea totală a rucsacului;

• *gains = [1, 2, 5, 6]*, unde *gains* reprezintă un vector de lungime n cu profitul pe care îl aduce fiecare obiect;

• *weights = [2, 3, 4, 5]*, unde *weights* reprezintă un vector de lungime n cu greutatea fiecărui obiect.

Scopul algoritmului este de a alege obiectele pentru a maximiza profitul obținut respectând în același timp constrângerea de a nu depăși capacitatea rucsacului. Astfel, la fiecare pas algoritmul va trebui sa aleagă una din cele două opțiuni disponibile: fie obiectul va fi adăugat în rucsac integral, iar greutatea și profitul acestuia vor fi adăugate în soluție, fie obiectul nu va fi inclus în rucsac.

Voi explica următoarele notații care se aplică și în cadrul celorlalte capitole ale lucrării:

→ i este folosit pentru a memora indexul obiectului curent;

→ j este folosit pentru a memora valoarea curentă a capacității parțiale a rucsacului;

→ tuplul (i, j), unde $0 \leq i < n$ și $0 \leq j < c$ reprezintă o subproblemă în care considerăm i obiecte și o capacitate parțială j a rucsacului.

În continuare, voi detalia o schiță a algoritmului implementat de către mine care poate fi folosită pentru a rezolva varianta discretă a problemei rucsacului și pentru a obține profitul maxim pentru orice instanță a problemei:

1. **Inițializarea:**
   - Inițializăm o matrice $profits$ de dimensiune $(n + 1) \times (c + 1)$, unde fiecare celulă $profits[i][j]$ reprezintă profitul maxim posibil obținut pentru o subproblemă în care sunt considerate disponibile maxim i obiecte, iar rucsacul poate susține o capacitate maximă j;
   - Inițializăm prima linie din matrice, care reprezintă cazurile în care pentru orice capacitate de la 0 până la j vom considera că nu avem obiecte disponibile → profitul maxim este 0.
2. **Procesul iterativ:** Pentru fiecare obiect i, unde $1 \leq i < n+1$ și fiecare capacitate j, unde $0 \leq j < c + 1$:

(a) Capacitatea rucsacului este 0 $\rightarrow$ profitul maxim posibil 0.

(b) Obiectul nu poate fi inclus

• greutatea obiectului curent $weights[i-1]$ depășește capacitatea j, atunci profitul pentru pasul curent rămâne același ca pentru $i-1$ obiecte:

$$profits[i][j] = profits[i-1][j].$$

(c) Știind că greutatea obiectului curent $weights[i-1]$ nu depășește capacitatea j, obiectul poate fi inclus, dar:

• dacă se obține un profit mai bun decât cel anterior prin includerea obiectului atunci adăugăm câștigul obiectului curent $gains[i-1]$ la valoarea obținută pentru capacitatea rămasă $j - weights[i-1]$:

$$profits[i][j] = profits[i-1][j = weights[i-1]] + gains[i-1].$$

• dacă se obține un profit cel mult la fel de bun ca profitul anterior obținut pentru aceeași capacitate j prin adăugarea obiectului atunci se va considera profitul maxim obținut fără acest obiect:

$$profits[i][j] = profits[i-1][j].$$

3. **Rezultatul:** La finalul execuției algoritmului, profitul maxim care se poate obține utilizând toate obiectele pe care le avem la dispoziție pentru un rucsac de capacitate maximă c se află în $profits[n][c]$.

Pentru a stoca soluțiile, algoritmul folosește abordarea Bottom-Up în care punctul de start îl reprezintă cele mai mici subprobleme, mai exact cazurile de bază. După cum am văzut în schița algoritmului prezentat anterior, adeseori pe parcusul execuției vom avea nevoie de profitul calculat în pasul anterior obținut pentru $i-1$ obiecte având aceeași capacitate j. Aici intervine proprietatea **problemelor suprapuse** caracteristică programării dinamice, în care aceeași subproblemă este rezolvată de mai multe ori în timpul execuției.

De asemenea, rezolvarea problemei inițiale pentru un rucsac de capacitate c cu n obiecte poate fi construită din rezultatele obținute pentru subproblemele anterioare la care este adăugat profitul obiectului în funcție de capacitatea care permite includerea acestuia. De menționat este că la fiecare pas al procesului iterativ, mereu se va considera cel mai bun

rezultat care se poate obține bazat pe două posibile decizii:

• Dacă obiectul curent este inclus în soluție, atunci rezultatul va fi calculat din profitul obiectului la care se adaugă soluția subproblemei pentru greutatea rămasă $j - weights[i-1]$, soluție despre care știm că este optimă pentru i obiecte si capacitate $j - weights[i-1]$;

• Altfel dacă obiectul nu este inclus, soluția va rămâne aceeași ca cea pentru $i-1$ obiecte și capacitate j, soluție despre care știm că este optimă pentru $i-1$ obiecte si capacitate j.

Aici intervine cea de-a doua proprietate specifică programării dinamice, numită proprietatea de **substructură optimă** care constă în faptul că soluția globală poate fi mereu construită din soluțiile optime ale subproblemelor.

Pentru a obține profitul maxim pentru datele de intrare de mai sus putem aplica algoritmul descris și începem prin a stabili cazul de bază: pornim de la $i = 0$ (nu avem niciun obiect la dispoziție) și trecem prin fiecare valoare posibilă a capacității j. Pentru fiecare subproblemă $(0, j)$, unde $0 \leq j < c$, soluția optimă este cea de profit maxim 0:

Matricea profits pentru $i = 0$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0

Algoritmul continuă prin parcurgerea fiecărui obiect și fiecare valoare posibilă a greutății rucsacului. Pentru $i = 1$, profitul maxim care se poate obține este 1 deoarece avem un singur obiect la dispoziție indiferent de valoarea capacității rucsacului:

Matricea profits pentru $i = 1$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1

Pentru $i = 2$ considerăm doar primele două obiecte și iterăm prin valorile posibile ale capacității:

- Pentru subproblema $(2, 2)$ nu putem adăuga decât primul obiect, iar profitul optim rămâne 1;

- Pentru subproblema $(2, 3)$ putem adăuga al doilea obiect deoarece aduce un profit mai bun decât cel anterior $profits[i-1][j]$ care este 1, iar restricția de a nu depăși capacitatea ruscacului este respectată;

- Pentru subproblema $(2, 5)$ putem adăuga ambele obiecte în rucsac, iar profitul este calculat adunând câștigul obiectului cu profitul corespunzător capacității rămase după scăderea greutății obiectului, adică rezultatul subproblemei($i-1, j-weights[i-1]$) care este 1:

Matricea profits pentru $i = 2$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3

Aplicând aceeași logică pentru $i = 3$ vom obține următoarele rezultate:

• Pentru subproblema $(3, 4)$ putem adăuga al treilea obiect deoarece aduce un profit mai bun decât rezultatul suproblemei anterioare cu aceeași capacitate $(2, 4)$;

• Pentru subproblema $(3, 5)$ obținem un profit mai bun decât pentru $(2, 5)$;

• Pentru subproblema $(3, 6)$ putem adăuga primul și al treilea obiect aducând un profit mai bun decât profitul anterior pentru $(2, 6)$;

• Pentru subproblema $(3, 7)$ putem adăuga al doilea și al treilea obiect obținând un profit de valoare 7 și greutate $7 \leq j$, mai bun decât profitul anterior pentru $(2, 7)$;

• Pentru subproblema $(3, 8)$ se obține un profit mai bun decât profitul calculat pentru subproblema anterioară $(2, 6)$:

Matricea profits pentru $i = 3$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7

Asemănător și pentru $i = 4$ când avem toate obiectele la dispoziție:

• Pentru subproblema $(4, 5)$ profitul calculat este mai bun decât profitul pentru $(3, 5)$;

• Ajungem la $(4, 8)$ care este problema inițială și putem adăuga al doilea și ultimul obiect, iar profitul maxim care se poate obține pornind de la datele de intrare este 8 adunând câștigul obiectului cu rezultatul subproblemei $(3, 3)$:

Matricea profits pentru $i = 4$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

Capitolul 1

Reprezentarea datelor de intrare și de ieșire

În următoarele capitole voi detalia implementarea algoritmului, dar și modul în care Dafny evaluează corectitudinea și optimalitatea codului.

Pentru a reprezenta problema pentru care algoritmul trebuie să obțină cel mai bun profit pe setul de intrare, am ales sa definesc un tip de date ***Problem*** astfel:

```
datatype Problem = Problem(n: int, c: int, gains: seq<int>,
weights: seq<int>)
```

unde:

- câmpul ***n*** reprezintă numărul total de obiecte disponibile;
- câmpul ***c*** reprezintă capacitatea totală a rucsacului;
- câmpul ***gains*** este o secvență ce memorează profitul fiecărui obiect;
- câmpul ***weights*** este o secvență ce memorează greutatea fiecărui obiect.

Folosind cuvântul cheie ***datatype*** am definit un nou tip de date inductiv numit ***Problem*** cu un singur constructor având același nume în care câmpurile sunt separate prin virgulă.

Atât profitul, cât și greutatea fiecărui obiect pot fi accesate prin indexul corespunzător poziției din secvență. Spre exemplu, pentru a accesa greutatea celui de-al doilea obiect este folosit operatorul de indexare $weights[1]$, deoarece indexarea celor două secvențe începe de la 0. În dafny, o *secvență* este o colecție de elemente de același tip, iar în cazul acesta, de numere întregi.

Orice instanța a problemei oferită la intrare trebuie să fie validă și să respecte anumite condiții pentru ca algoritmul să poată produce o soluție corectă. Pentru a mă asigura că aceste condiții sunt îndeplinite a fost nevoie să formulez un predicat. ***Predicatele*** în Dafny sunt funcții ale căror rezultate sunt valori boolene. Acestea sunt folosite pentru a evalua proprietăți care de obicei trebuie să fie adevărate și de aceea ele ajută în procesul de verificare al corectitudinii. Astfel, am definit predicatul ***isValidProblem*** care verifică dacă avem cel puțin un obiect la dispoziție și un rucsac de capacitate mai mare decât zero, dacă știm greutatea și câștigul fiecărui obiect, iar acestea au valori pozitive diferite de zero:

```
predicate isValidProblem(p: Problem)
{
  |p.gains| == |p.weights| == p.n &&
  p.n > 0 && p.c >= 0 &&
  hasPositiveValues(p.gains) && hasPositiveValues(p.weights)
}
```

Predicatul ***hasPositiveValues*** este definit astfel:

```
predicate hasPositiveValues(arr: seq<int>)
{
  forall i :: 0 <= i < |arr| ==> arr[i] > 0
}
```

și este folosit, după cum sugerează și numele acestuia, pentru a verifica dacă toate elementele unei secvențe precum *gains* sunt pozitive.

Datele de ieșire sunt reprezentate astfel:

```
(profit: int, solution: Solution)
```

unde:

- ***profit*** reprezintă profitul maxim care se poate obține pentru o anumită instanță a problemei;
- ***solution*** reprezintă un sinonim al tipului de date `seq<int>` folosit pentru a îmbunătăți claritatea codului, este declarat anterior astfel:

```
type Solution = seq<int>
```

și este o reprezentare binară a deciziei de includere a obiectului în rucsac. Prin urmare, valoarea de 1 reprezintă faptul că obiectul aparține soluției, deci este adăugat în rucsac, iar profitul adus de acesta îmbunătățește câștigul final, iar 0 reprezintă faptul că obiectul nu este adăugat în rucsac, deci profitul obiectului nu este inclus în câștigul maxim obținut. Această secvență reprezintă soluția finală care trebuie să fie optimă pentru problema completă.

Capitolul 2

Reprezentarea soluțiilor

În cadrul problemei rucsacului putem discuta despre ce reprezintă soluția optimă, cât și soluțiile parțiale optime.

Pentru a stoca rezultatele corespunzătoare câștigurilor subproblemelor am folosit o secvență de secvențe numită `profits` de tip `seq<seq<int>>`. Deși la bază această variabilă este o secvență, ea poate fi privită ca o matrice bidimensională, iar rezultatele pot fi accesate folosind operatorul de indexare similar unui astfel de tablou: $profits[i][j]$, unde i și j reprezită mărimea subproblemei pe care încercăm să o rezolvăm.

Pe lângă matricea `profits`, această lucrare se mai bazează și pe o variabilă numită `solutions` de tip `seq<seq<seq<int>>>` care va stoca fiecare secvență binară corespunzătoare profitului stocat în matricea `profits`. Ea poate fi privită tot ca un tablou multidimensional, de aceea rezultatele sunt accesate similar matricei profits.

O ***soluție optimă*** reprezintă selecția de obiecte care maximizează profitul format din câștigul fiecărui item care poate fi adăugat în rucsac, dar care în același timp respectă constrângerile legate de capacitatea rucsacului. Această soluție adresează problema completă, deținând decizia de includere a fiecărui obiect, iar profitul calculat trebuie să aibă cea mai mare valoare pe care o putem obține pentru instanța primită. În cazul acestei lucrări, acest predicat este folosit pentru a verifica soluția finală (despre care am discutat în capitolul anterior). Pentru exprimarea acestor proprietăți am formulat predicatul ***isOptimalSolution*** astfel:

```
ghost predicate isOptimalSolution(p: Problem,
    solution: Solution)
  requires isValidProblem(p)
  requires isValidPartialSolution(p, solution)
```

```
{
  isOptimalPartialSolution(p, solution, p.n, p.c) &&
  forall s: Solution :: (((isOptimalPartialSolution(
    p, s, p.n, p.c)) ==> gain(p, solution) >= gain(p, s)))
}
```

Condițiile exprimate folosind clauza `requires` sunt numite **precondiții** și reprezintă proprietăți pe care parametrii trebuie să îi respecte înainte de intrarea în corpul unei funcții, a unei metode, leme sau predicat și care vor fi valabile până la ieșirea din acestea. Sunt esențiale pentru Dafny deoarece reduc ambiguitatea prin definirea clară a informațiilor pe care trebuie să le știm despre parametrii oferiți și permit astfel validarea corectitudinii logice ale programului, fără a fi necesară verificarea acestora pe parcurs. De asemenea, sunt folositoare și pentru a verifica dacă apelurile către componentele logice au fost conform așteptărilor.

O ***soluție optimă parțială*** reprezintă un rezultat optim intermediar calculat pentru un subset de obiecte sau pentru o capacitate parțială a rucsacului. Luând exemplul prezentat în introducere, o soluție parțială pentru subproblema $(i = 3, j = 8)$ este $[0, 1, 1]$ și aduce cel mai bun profit pentru un subset ce include doar primele trei obiecte, având la dispoziție un rucsac de capacitate maximă 8. Pentru a verifica dacă avem o astfel de soluție am implementat predicatul ***isOptimalPartialSolution***:

```
ghost predicate isOptimalPartialSolution(p: Problem,
    solution: Solution, i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= i <= p.n
  requires 0 <= j <= p.c
{
  isPartialSolution(p, solution, i, j) &&
  forall s: Solution :: (isPartialSolution(p, s, i, j) &&
    |s| == |solution| ==> gain(p, solution) >= gain(p, s))
}
```

Acest predicat verifică dacă orice altă soluție parțială pentru o subproblemă cu i obiecte și capacitate j a rucsacului va avea câștigul mai mic sau cel mult egal cu cel al soluției noastre.

O ***soluție parțială*** este o secvență binară de o lungime variabilă care nu depășește numărul de obiecte și care respectă constrângerea legată de capacitatea rucsacului. Predicatul formulat pentru a confirma o astfel de soluție este ***isPartialSolution***:

```
predicate isPartialSolution(p: Problem, solution: Solution,
        i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= i <= p.n
  requires 0 <= j <= p.c
{
  isValidPartialSolution(p, solution) && |solution| == i &&
  weight(p, solution) <= j
}
```

și vine în completarea predicatului ***isValidPartialSolution***:

```
predicate isValidPartialSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
{
  hasAllowedValues(solution) && |solution| <= p.n
}
```

Cu ajutorul celor două predicate am verificat dacă o soluție este parțială și validă: are doar elemente de 0 și 1, lungimea acesteia nu depășește numărul de obiecte și are o greutate cel mult egală cu capacitatea j a rucsacului. Predicatul apelat ***hasAllowedValues*** este definit astfel:

```
predicate hasAllowedValues(solution: Solution)
{
  forall k :: 0 <= k < |solution| ==> solution[k] == 0
        || solution[k] == 1
}
```

și asigură că toate elementele unei soluții aparțin exclusiv mulțimii $\{0, 1\}$, unde $0 -$ obiectul nu este în rucsac, $1 -$ obiectul este în rucsac.

Capitolul 3

Specificații

3.1 Predicate

Alte predicate formulate care au fost necesare în puncte diferite ale demonstrației sunt:

1. Predicatul *isSolution*:

```
predicate isSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
{
  isValidPartialSolution(p, solution) && |solution| == p.n &&
  weight(p, solution) <= p.c
}
```

Acesta verifică dacă avem o soluție validă pentru problema completă.

2. Predicatul *isValidSubproblem*:

```
predicate isValidSubproblem(p: Problem, i: int, j: int)
{
  isValidProblem(p) && 1 <= i <= p.n && 1 <= j <= p.c
}
```

Este folosit pentru a defini o subproblemă a problemei inițiale, excluzând cazurile de bază.

3. Predicatul *areValidPartialSolutions*:

```
ghost predicate areValidPartialSolutions(p: Problem, profits:
    seq<seq<int>>, solutions: seq<seq<seq<int>>>,
    partialProfits: seq<int>, partialSolutions: seq<seq<int>>,
    i: int, j: int)
  requires isValidSubproblem(p, i, j)
{
  |partialSolutions| == |partialProfits| == j &&
  (forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)) &&
  (forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k])
}
```

Este folosit pentru a valida rezultatele obținute doar pentru subproblemele pasului curent, în funcție de modificările aduse datorită creșterii numărului de obiecte disponibile.

4. Predicatul *areValidSolutions*:

```
ghost predicate areValidSolutions(p: Problem, profits:
    seq<seq<int>>, solutions: seq<seq<seq<int>>>, i: int)
  requires isValidSubproblem(p, i, p.c)
{
  i == |profits| == |solutions| && (forall k :: 0 <= k < i
  ==> |profits[k]| == |solutions[k]| == p.c + 1) &&
  (forall k :: 0 <= k < |solutions| ==>
    forall q :: 0 <= q < |solutions[k]| ==>
      isOptimalPartialSolution(p, solutions[k][q], k, q)) &&
  (forall k :: 0 <= k < |solutions| ==>
    forall q :: 0 <= q < |solutions[k]| ==>
      gain(p, solutions[k][q]) == profits[k][q])
}
```

A fost folosit pentru a verifica faptul că soluțiile parțiale ale subproblemelor sunt valide și optime: au lungime corespunzătoare și aduc cel mai bun profit pentru subproblema pe care o calculează. Acesta asigură că soluțiile obținute la fiecare pas sunt

construite pe baza soluțiilor optime ale subproblemelor anterioare.

3.2 Funcții

În Dafny, funcțiile pot fi interpretate ca funcții matematice în care corpul funcției reprezintă definiția acesteia, motiv pentru care de obicei, funcțiile nu necesită postcondiții. Sunt folosite pentru a returna un rezultat la ieșire al cărui tip este specificat în semnătura funcției, imediat după lista de parametri.

Funcțiile importante pe care le-am implementat și care au fost necesare în demonstrație sunt:

1. Funcția *gain*:

```
function gain(p: Problem, solution: Solution): int
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= p.n
{
  if |solution| == 0 then 0
        else computeGain(p, solution, |solution| - 1)
}
```

   Este folosită pentru calcularea profitului corespunzător unei soluții valide. Este folosită când se dorește câștigul întregii soluții.

2. Funcția *computeGain*:

```
function computeGain(p: Problem, solution:
    Solution, i: int) : int
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= i < |p.gains|
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= |p.gains|
  ensures computeGain(p, solution, i) >= 0
{
  if i == 0 then solution[0] * p.gains[0] else
```

```
  solution[i] * p.gains[i] + computeGain(p, solution, i - 1)
}
```

Spre deosebire de *gain*, poate primi o subsoluție al cărui profit se dorește a fi calculat prin specificarea unui index i reprezentând poziția de la final spre început a subsoluției. Este apelată de funcția *gain* pentru a obține câștigul întregii soluții și are o implementare recursivă a cărei finalitate este asigurată prin condiția $i == 0$. Această funcție oferă o informație în plus care nu reiese direct din definiția funcției, dar este adevărată, mai exact faptul că la finalul execuției rezultatul obținut va fi mereu pozitiv sau egal cu zero. Acest lucru este specificat folosind clauza `ensures`, iar adnotările de acest gen sunt numite ***postcondiții***. Ele sunt expresii logice care trebuie să fie adevărate după executarea logicii unei funcții, metode sau leme.

3. Funcția ***weight***, asemănătoare din punct de vedere al implementării cu *gain*, calculează greutatea totală a obiectelor corespunzătoare pozițiilor din soluție ale cărei valoare este 1.

4. Funcția ***computeWeight***, este asemănătoare din punct de vedere al implementării cu *computeGain*, în schimb ea calculează greutatea unei (sub)soluții. Este apelată de funcția *weight* pentru a obține greutatea întregii soluții.

5. Funcția ***sumAllGains***:

```
function sumAllGains(p: Problem, i: int) : int
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 ensures sumAllGains(p, i) >= 0
{
  if (i == 1) then p.gains[0]
    else p.gains[i - 1] + sumAllGains(p, i -1)
}
```

Rezultatul acestei funcții reprezintă suma tuturor profiturilor obiectelor.

Capitolul 4

Punctul de intrare și arhitectura codului

Execuția algoritmului propus pornește de la o metodă principală numită ***solve*** ce primește ca parametru o problemă *p: Problem*. O metodă în Dafny reprezintă o bucată de cod ce conține operații executabile și ce modifică starea variabilelor definite. Această metodă va trebui să întoarcă două rezultate, un rezultat este valoarea profitului maxim *profit: int* care se poate obține pentru instanța problemei oferită, iar cel de-al doilea rezultat este soluția *solution: Solution* corespunzătoare profitului obținut formată din elemente de 0 și 1:

```
method solve(p: Problem) returns (profit: int, solution: Solution)
  requires isValidProblem(p)
  ensures isSolution(p, solution)
  ensures isOptimalSolution(p, solution)
{
    var profits := [];
    var solutions := [];
    var i := 0;
    var partialProfits, partialSolutions :=
        solves0Objects(p, profits, solutions, i);
    profits := profits + [partialProfits];
    solutions := solutions + [partialSolutions];
    assert forall k :: 0 <= k < |partialSolutions| ==>
        isOptimalPartialSolution(p, partialSolutions[k], i, k);
    assert forall k :: 0 <= k < |solutions| ==>
        forall q :: 0 <= q < |solutions[k]| ==>
            gain(p, solutions[k][q]) == profits[k][q];
```

```
    i := i + 1;
    while i <= p.n
      invariant 0 <= i <= p.n + 1
      invariant |profits| == |solutions| == i
      invariant forall k :: 0 <= k < i ==> |profits[k]| == p.c + 1
      invariant forall k :: 0 <= k < |solutions| ==>
        |solutions[k]| == p.c + 1
      invariant forall k :: 0 <= k < |solutions| ==>
        forall q :: 0 <= q < |solutions[k]| ==>
            isOptimalPartialSolution(p, solutions[k][q], k, q)
      invariant forall k :: 0 <= k < |solutions| ==>
        forall q :: 0 <= q < |solutions[k]| ==>
            gain(p, solutions[k][q]) == profits[k][q]
    {
        partialProfits, partialSolutions :=
            getPartialProfits(p, profits, solutions, i);
        profits := profits + [partialProfits];
        solutions := solutions + [partialSolutions];
        i := i + 1;
    }
    solution := solutions[p.n][p.c];
    assert isOptimalSolution(p, solution);
    profit := profits[p.n][p.c];
}
```

Postcondițiile ***isSolution*** și ***isOptimalSolution*** ne asigură că la finalul execuției metodei obținem o soluție validă doar cu elemente de 0 și 1, de lungime egală cu numărul de obiecte, ce nu depășește capacitatea rucsacului și de profit maxim.

Începem prin a inițializa matricile $profits$ și $solutions$ cu secvențe vide, iar numărul de obiecte disponibile este contorizat prin i și este inițializat cu 0. Vom obține rezultatele pentru cazurile de bază, mai exact pentru subproblemele în care $i = 0$ (mulțimea obiectelor este vidă) și pentru fiecare capacitate parțială a rucsacului invocând o altă metodă, numită ***solves0Objects***, implementată astfel:

```
var j := 0;
while j <= p.c
  invariant 0 <= j <= p.c + 1
  invariant |partialProfits| == j
  invariant |partialSolutions| == j
  invariant forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
  invariant forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
  {
    partialProfits := partialProfits + [0];
    var currentSolution := [];
    emptySolOptimal(p, currentSolution, i, j);
    assert isOptimalPartialSolution(p, currentSolution, i, j);
    partialSolutions := partialSolutions + [currentSolution];
    j := j + 1;
  }
```

Știm că i este 0 în acest caz, mai exact nu avem niciun obiect la dispoziție, prin urmare cel mai bun profit care se poate obține este 0, iar soluțiile nu trebuie să conțină niciun element, deoarece nu avem ce să adăugăm în rucsac. Am folosit o buclă $while$ pentru a considera toate valorile posibile pe care le poate avea capacitatea rucsacului, iar la fiecare iterație soluțiile, respectiv profiturile aferente sunt memorate în $partialSolutions$, respectiv $partialProfits$.

De cele mai multe ori structurile repetitive reprezintă o problemă pentru Dafny deoarece nu știm mereu de câte ori va trebui să iterăm prin acestea, dar și datorită modului în care Dafny încearcă să demonstreze unele proprietăți care nu ar trebui să se modifice în timpul procesului iterativ. Astfel avem nevoie de niște adnotări pentru a-i specifica verificatorului ce informații se păstrează pe parcursul structurii repetitive. Pentru acest lucru este folosită clauza ***invariant***. Cu ajutorul acestor adnotări se pot specifica expresii logice care trebuie să fie adevărate pe toată durata structurii, inclusiv înainte de intrarea în buclă. Primul invariant este cel care asigură terminarea structurii, invariantul

```
invariant forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
```

este folosit pentru a garanta faptul că pe parcursul buclei toate soluțiile calculate își păstrează proprietatea de **soluție optimă parțială**, iar invariantul

```
invariant forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
```

este folosit pentru a ajuta verificatorul să înțeleagă relația dintre cele două secvențe cu rezultate, mai exact aplicând funcția $gain$ peste o soluție stocată în secvența $partialSolutions$, rezultatul acesta poate fi interpretat ca fiind cel stocat în aceeași poziție în secvența $partialProfits$. Pentru consistență, proprietățile definite de acești invarianți trebuie să fie respectate de toate soluțiile memorate pe parcurs, de aceea se vor repeta și în cadrul invarianților din celelalte metode.

De asemenea, deoarece proprietățile demonstrate în corpul buclelor sunt disponibile într-un scop limitat (de obicei chiar la blocul de instrucțiuni al structurii), cu ajutorul invarianților acestea devin adevărate și după execuția buclelor, fiind un ajutor important în verificarea postcondițiilor. Această metodă garantează astfel că soluțiile aferente cazurilor de bază sunt optime și respectă limitările impuse față de lungimea rezultatelor stocate.

După ce am obținut soluțiile cazurilor de bază, începem procesul iterativ. Metoda ***solve*** este cea care ține evidența numărului de obiecte și care salvează în matricile anterior menționate rezultatele concrete, care sunt calculate, verificate și returnate de către metoda ***getPartialProfits***, în care fiecare ramură if corespunde unei decizii de acceptare sau respingere a obiectului:

```
method getPartialProfits(p: Problem, profits: seq<seq<int>>,
 solutions : seq<seq<seq<int>>>, i: int)
  returns (partialProfits: seq<int>, partialSolutions: seq<seq<int>>)
  requires isValidProblem(p)
  requires 0 < i < p.n + 1
  requires i == |profits| == |solutions|
  requires forall k :: 0 <= k < i ==> |profits[k]| == p.c + 1
  requires forall k :: 0 <= k < i ==> |solutions[k]| == p.c + 1
  requires forall k :: 0 <= k < |solutions| ==>
   forall q :: 0 <= q < |solutions[k]| ==>
    isOptimalPartialSolution(p, solutions[k][q], k, q)
  requires forall k :: 0 <= k < |solutions| ==>
```

```
    forall q :: 0 <= q < |solutions[k]| ==>
     gain(p, solutions[k][q]) == profits[k][q]
  ensures p.c + 1 == |partialSolutions| == |partialProfits|
  ensures 0 <= |profits| <= p.n + 1
  ensures forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
  ensures forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
{
  var j := 0;
  partialProfits := [];
  partialSolutions := [];
  while j <= p.c
   invariant 0 <= j <= p.c + 1
   invariant 0 <= |profits| <= p.n + 1
   invariant j == |partialProfits| == |partialSolutions|
   invariant forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
   invariant forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
   {
     if j == 0 {
      var currentProfit, currentSolution := solvesCapacity0(p, i, j);
      partialProfits := partialProfits + [currentProfit];
      partialSolutions := partialSolutions + [currentSolution];
      assert isOptimalPartialSolution(p, currentSolution, i, j);
     } else { if p.weights[i - 1] <= j {
        if p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] >
                          profits[i - 1][j] {
          var currentProfit, currentSolution :=
            solvesAdd1BetterProfit(p, profits, solutions,
            partialProfits, partialSolutions, i, j);
          partialProfits := partialProfits + [currentProfit];
          partialSolutions := partialSolutions + [currentSolution];
```

```
            assert isOptimalPartialSolution(p, currentSolution, i, j);
          } else {
            var currentProfit, currentSolution :=
              solvesAdd0BetterProfit(p, profits, solutions,
              partialProfits, partialSolutions, i, j);
            partialProfits := partialProfits + [currentProfit];
            partialSolutions := partialSolutions + [currentSolution];
            assert isOptimalPartialSolution(p, currentSolution, i, j);
          }
        } else {
            var currentProfit, currentSolution := solvesAdd0TooBig(p,
            profits, solutions, partialProfits, partialSolutions, i, j);
            partialProfits := partialProfits + [currentProfit];
            partialSolutions := partialSolutions + [currentSolution];
            assert isOptimalPartialSolution(p, currentSolution, i, j);
          }
       }
      j := j + 1;
      }
}
```

Metoda primește ca parametri secvențele $profits$ și $solutions$ ce memorează rezultatele calculate în iterațiile anterioare pentru a evita recalcularea și returnează alte secvențe în care sunt memorate rezultatele iterației curente după ce ne asigurăm că ele sunt corecte și optime pentru subproblemele pe care le rezolvă.

Invarianții folosiți aici sunt necesari pentru verificarea cu succes a metodei și de asemenea corespund cu postcondițiile, deoarece metoda apelantă ***solve*** trebuie să știe ce fel de rezultate primește înapoi. După cum se poate observa, invarianții sunt similari cu cei descriși pentru metoda ***solves0Objects*** și au rolul de a ajuta verificatorul să demonstreze corectitudinea postcondițiilor. Fără aceștia, el nu ar știi ce modificări s-ar produce în corpul buclei asupra variabilelor folosite.

Am folosit o buclă $while$, asemănător metodei ***solves0Objects***, pentru a trece prin valorile parțiale ale capacității rucsacului, pornind de la 0 și incrementând cu 1 până la capacitatea totală a acestuia. Astfel, având metoda ***solve*** care iterează prin valorile posibile pentru i

(ce reprezintă numărul de obiecte considerate) și metoda ***getPartialProfits*** care iterează prin valorile posibile pentru j (ce reprezintă capacitatea parțială pe care o poate avea rucsacul), acoperim toate subproblemele pentru care avem nevoie ca să ajungem la soluția finală.

Știm că avem cel puțin un obiect disponibil în acest punct al algoritmului. Numărul de obiecte este fix pentru această metodă, deci trebuie să luăm în considerare doar valoarile posibile ale capacității. Astfel, avem patru cazuri pe care trebuie să le rezolvăm:

- cazul în care capacitatea parțială a ruscacului este 0

  ```
  if j == 0
  ```

  este un caz special, poate fi considerat tot un caz de bază. Capacitatea fiind 0 ar însemna că avem un rucsac ce nu poate susține niciun obiect, deci nu putem include nimic în acest punct al algoritmului. Acest caz este tratat în metoda ***solvesCapacity0***:

  ```
  currentProfit := 0;
  currentSolution := seq(i, y => 0);
  ```

  Astfel, cel mai bun profit care se poate obține este 0, iar soluțiile optime conțin doar elemente de 0 pe fiecare poziție.

- cazul în care greutatea obiectului depășește capacitatea parțială j a rucsacului

  ```
  p.weights[i - 1] > j
  ```

  aferent ultimului if din această metodă oferă iarăși o alegere relativ simplă, obiectul nu poate fi adăugat în rucsac pentru subproblema cu i obiecte și capacitate j, astfel că profitul va rămâne același ca în pasul anterior pentru aceeași capacitate, respectiv $profits[i-1][j]$, iar soluția pentru subproblema curentă va fi cea de la pasul anterior, respectiv $solutions[i-1][j]$ la care se va adăuga un 0 pentru a marca decizia luată în acest pas:

  ```
  currentProfit := profits[i - 1][j];
  currentSolution := solutions[i - 1][j];
  currentSolution := currentSolution + [0];
  ```

  este implementarea din metoda ***solvesAdd0TooBig***.

- cazul în care greutatea obiectului curent nu depășeste capacitatea rucsacului și adăugarea acestuia aduce un profit mai bun decât cel anterior pentru aceeași capacitate j:

```
if p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] >
                    profits[i - 1][j]
```

este cel care produce elementele de 1 în soluții, element care de această dată este adăugat soluției de pe poziția ce corespunde capacității rămase după ce am inclus greutatea obiectului. Profitul în acest caz este calculat adunând profitul de pe aceeași poziție a secvenței $profits$ și câștigul obiectului curent:

```
currentProfit := p.gains[i - 1] +
    profits[i - 1][j - p.weights[i - 1]];
currentSolution := solutions[i - 1][j - p.weights[i - 1]];
currentSolution := currentSolution + [1];
```

Caz tratat în metoda ***solvesAdd1BetterProfit***.

- cazul în care deși greutatea obiectului curent nu depășește capacitatea j, includerea acestuia nu produce un profit mai bun față de excluziunea lui. În acest caz, un 0 este adăugat soluției de la pasul $solutions[i-1][j]$, iar profitul rămâne același ca cel de pe poziția corespunzătoare din $profits$. Cazul este tratat în ***solvesAdd0BetterProfit*** și are implementare similară cu ***solvesAdd0TooBig***.

După cum am menționat, fiecare caz va fi tratat într-o metodă diferită, unde fiecare condiție din if din metoda curentă va deveni o precondiție a metodei în care este tratat fiecare caz de mai sus. Alte precondiții similare pentru aceste metode vor fi limitările legate de numărul de obiecte, cât și informații despre soluțiile subproblemelor calculate în iterațiile anterioare. Acestea sunt necesare ca specificații deoarece Danfy nu are acces la informațiile din metoda apelantă, fiecare metodă având o responsabilitate separată.

Capitolul 5

Leme importante

Lemele sunt declarații folosite atunci când unele proprietăți logice ale verificării sunt mult prea complexe pentru Dafny pentru a le putea demonstra singur. Acestea nu au parametri de ieșire, servind doar ca un puternic instrument de ghidare al verificatorului în demonstrarea corectitudinii programului. Sunt declarate separat și pot avea, asemănător metodelor, precondiții și postcondiții. Astfel, o lemă va fi demonstrată separat luând în calcul toate posibilitățile aplicabile pentru parametri ce îndeplinesc precondițiile, având ca scop verificarea cu succes a postcondiției, care de altfel reprezintă proprietatea pe care dorim sa o clarificăm pentru Dafny.

Voi prezenta mai departe unele dintre cele mai importante leme formulate de mine care m-au ajutat în demonstrarea corectitudinii și optimalității algoritmului.

1. Lema *computeGainAllZeros*

```
lemma computeGainAllZeros(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= |solution| <= p.n
  requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
  ensures computeGain(p, solution, i) == 0
{
  if i == 0 {
    assert  computeGain(p, solution, i) == 0;
  } else {
    computeGainAllZeros(p, solution, i - 1);
    assert computeGain(p, solution, i - 1) == 0;
```

```
    assert computeGain(p, solution, i) == 0;
  }
}
```

Această lemă este demonstrată prin inducție și folosită pentru a arăta că dacă avem o soluție ce conține doar elemente de 0, atunci câștigul produs de o astfel de soluție nu poate fi decât 0.

2. Lema ***optimalSolCapacity0***

```
lemma optimalSolCapacity0(p: Problem, solution: Solution, i: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires isPartialSolution(p, solution, i, 0)
 requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
 requires weight(p, solution) == 0
 ensures isOptimalPartialSolution(p, solution, i, 0)
{
  assert isPartialSolution(p, solution, i, 0);
  forall s: Solution |
   isPartialSolution(p, s, i, 0) && |solution| == |s|
  ensures gain(p, solution) >= gain(p, s)
  {
    assert weight(p, solution) == 0;
    assert forall k :: 0 <= k < |solution| ==> solution[k] == 0;
    computeGainAllZeros(p, solution, |solution| - 1);
    gainCapacity0(p, s, i);
    assert gain(p, solution) == 0;
    assert gain(p, s) == 0;
  }
  assert forall s: Solution :: (isPartialSolution(p, s, i, 0) &&
   |s| == |solution| ==> gain(p, solution) >= gain(p, s));
  assert isOptimalPartialSolution(p, solution, i, 0);
}
```

este folosită pentru a demonstra că soluțiile generate pentru cazul $j == 0$ sunt într-

adevăr optime pentru subproblemele în care rucsacul nu permite adăugarea niciunui obiect. Verificarea corectă a lemei începe încă de la valoarea de adevăr a precondițiilor, deoarece după cum am menționat, lemele sunt verificate separat în funcție de condițiile impuse prin precondiții. Se dorește aici să demonstrăm că pentru o problemă validă, o soluție parțială doar cu elemente de 0 și de greutate 0 este cea care produce cel mai bun profit datorită limitării impuse de către parametrul de capacitate.

Am folosit un *forall statement* cu ajutorul căruia am demonstrat că pentru toate soluțiile s care îndeplinesc proprietatea de **soluție parțială** de lungime i, câștigul acestora este cel mult egal cu cel al soluției presupuse a fi optimă. Pentru astfel de soluții am demonstrat că ele nu pot fi altele decât cele care conțin tot doar elemente de 0, iar câștigul acestora este 0.

3. Lema ***optimalSolRemove1***

```
lemma optimalSolRemove1(p: Problem, solution: Solution,
    i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i, j)
 requires solution[i - 1] == 1
 ensures isOptimalPartialSolution(p, solution[..i - 1],
    i - 1, j - p.weights[i - 1])
{
  var s := solution[..i - 1];
  weightAdd1(p, solution);
  assert isPartialSolution(p, solution[..i - 1],
    i - 1, j - p.weights[i - 1]);
  if !isOptimalPartialSolution(p, solution[..i - 1],
    i - 1, j - p.weights[i - 1]) {
    gainAdd1(p, solution);
    existsOptimalPartialSol(p, i - 1, j - p.weights[i - 1]);
    var x : Solution :| isOptimalPartialSolution(p,
        x, i - 1, j - p.weights[i - 1]);
    assert |x| == |solution[..i - 1]|;
```

```
        assert gain(p, x) > gain(p, solution[..i - 1]);
        var x1 := x + [1];
        gainAdd1(p, x1);
        weightAdd1(p, x1);
        assert isOptimalPartialSolution(p, x1, i, j);
        assert s == solution[..|solution| - 1];
        assert x == x1[..|x1| - 1];
        assert gain(p, x1) == gain(p, x) + p.gains[i - 1] >
            gain(p, s) + p.gains[i- 1] == gain(p, solution);
        assert gain(p, x1) > gain(p, solution);
        assert false;
    }
}
```

După cum am menționat anterior, una dintre proprietățile caracteristice programării dinamice este proprietatea de ***substructură optimă***, care constă în ideea că soluția optimă a unei probleme poate fi construită din soluțiile optime ale subproblemelor acesteia. Această lemă se dorește a demonstra tocmai această proprietate, dar în sens invers: având o soluție optimă pentru subproblema (i, j)

```
requires isOptimalPartialSolution(p, solution, i, j)
```

trebuie să demonstrăm că dacă din acestă soluție vom extrage ultimul obiect adăugat atunci vom obține o soluție optimă pentru o subproblemă cu un obiect mai puțin și capacitatea rămasă a rucsacului după eliminarea greutății corespunzătoare obiectului:

```
ensures isOptimalPartialSolution(p, solution[..i - 1],
        i - 1, j - p.weights[i - 1])
```

Pentru a putea demonstra această lemă, am început prin a presupune că soluția rămasă după eliminarea obiectului nu este optimă pentru subproblema $(i - 1, j - p.weights[i - 1])$, dar că există o presupusă soluție x ce aduce un profit optim pentru aceasta . Dar dacă pentru această soluție am adăuga obiectul i, am obține o soluție de lungime i cu un profit mai bun decât $solution$, ceea ce contrazice precondiția lemei, deci este presupunerea făcută la început este falsă.

4. Lema ***optimalSolAdd1***

```
lemma optimalSolAdd1(p: Problem, profit1: int, profit2: int,
    solution1: Solution, solution2: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p,
    solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution1 + [1],
    |solution1 + [1]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 > profit2
 ensures isOptimalPartialSolution(p, solution1 + [1], i, j)
{
  var s := solution1 + [1];
  if !isOptimalPartialSolution(p, s, i, j){
    existsOptimalPartialSol(p, i, j);
    var x : seq<int> :| isOptimalPartialSolution(p, x, i, j);
    assert gain(p, x) > gain(p, solution1 + [1]);
    if x[i - 1] == 0 {
      assert gain(p, x) <= profit2 by
      {
        gainAdd0(p, x);
        assert gain(p, x[..i - 1]) == gain(p, x);
        weightAdd0(p, x);
        assert weight(p, x[..i - 1]) <= j;
      }
      assert gain(p, solution1 + [1]) > profit2 by
      {
        gainAdd1(p, solution1 + [1]);
        assert gain(p, solution1 + [1]) ==
            gain(p, solution1) + p.gains[i - 1];
```

```
        }
        assert false;
      } else {
        gainAdd1Optimal(p, profit1, profit2,
          solution1, solution2, x, i, j);
        assert gain(p, x) == gain(p, solution1 + [1]);
      }
    }
}
```

este formulată pentru cazul în care trebuie să demonstrăm că adăugarea unui obiect produce o soluție parțială care este optimă, iar pentru a demonstra acest lucru am folosit metoda reducerii la absurd astfel: având două soluții optime $solution1$ și $solution2$ pentru subproblemele $(i-1, j-p.weights[i-1])$ și $(i-1, j)$, vom construi noua soluție adăugând un element de 1 în $solution1$ (soluția ce corespunde capacității rămase după includerea obiectului). Presupunând că această soluție nu este optimă pentru subproblema (i, j), înseamnă că există o soluție x ce produce un profit mai bun.

Trebuie să considerăm două cazuri pentru soluția x:

- Nu conține obiectul i $\rightarrow$ înseamnă că profitul lui x este cel mult egal cu profitul adus de soluția subproblemei anterioare $(i-1, j)$, însă știind din precondițiile

  ```
  p.gains[i - 1] + profit1 > profit2
  computeWeight(p, solution1 + [1], |solution1 + [1]| - 1) <= j
  ```

  că adăugând câștigul obiectului curent obținem un profit mai bun și nu depășim capacitatea j, deci ajungem la o contradicție, iar obiectul aparține soluției x.

- Conține obiectul i $\rightarrow$ pentru a demonstra că profitul soluției propuse este cel puțin la fel de bun ca profitul soluției x, trebuie să demonstrăm că și profitul subsoluțiilor din care extragem obiectul i sunt egale:

  ```
  assert gain(p, x) == gain(p, solution1 + [1]) by {
    gainAdd1(p, solution1 + [1]);
    gainAdd1(p, x);
    assert x == x[..i - 1] + [1];
    assert gain(p, x[..i - 1]) == gain(p, solution1) by
    {
  ```

```
        optimalSolRemove1(p, x, i, j);
        assert isOptimalPartialSolution(
                p, x[..i - 1], i - 1, j - p.weights[i - 1]);
      }
    }
```

Acest lucru este demonstrat folosind proprietatea de substructură optimă despre care am vorbit în cadrul lemei ***optimalSolRemove1***.

Astfel am demonstrat că adăugarea obiectului curent aduce un profit la fel de bun ca cel al unei soluții presupuse optime x, deci soluția propusă este și ea optimă.

5. Lema ***optimalSolAdd0***

```
lemma optimalSolAdd0(p: Problem, profit1: int, profit2: int,
    solution1: Solution, solution2: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p,
    solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution2 + [0],
    |solution2 + [0]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 <= profit2
 ensures isOptimalPartialSolution(p, solution2 + [0], i, j)
{
  if !isOptimalPartialSolution(p, solution2 + [0], i, j) {
    existsOptimalPartialSol(p, i, j);
    var x : Solution :| isOptimalPartialSolution(p, x, i, j);
    if x[i - 1] == 1 {
      var x1 := x[..i - 1];
      assert gain(p, x1) == profit1 by {
```

```
            optimalSolRemove1(p, x, i, j);
            assert x1 == x[..|x| - 1];
            assert isOptimalPartialSolution(p, x1, i - 1,
                j - p.weights[i - 1]);
          }
          gainAdd1(p, x);
          gainAdd0(p, solution2 + [0]);
          assert gain(p, x) == gain(p, x1) +
            p.gains[i - 1] <= gain(p, solution2 + [0]);
          assert false;
        }
        assert x[i - 1] == 0;
        gainAdd0Optimal(p, profit1, profit2, solution1,
            solution2, x, i, j);
        assert gain(p, x) == gain(p, solution2 + [0]);
      }
    }
```

Demonstrația lemei este similară cu cea pentru ***optimalSolAdd1***, pornind prin a presupune că ipoteza inițială este falsă, deci există o soluție x ce produce un profit mai bun. Trebuie să considerăm apartenența obiectului în x, iar în ambele cazuri vom folosi proprietatea de substructră optimă pentru a demonstra că $solution2 + [0]$ are un profit cel puțin la fel de bun ca x, deci adăugarea obiectului nu produce un profit mai bun pentru acestă subproblemă.

6. Lema ***optimalSolAdd0TooBig***

```
lemma optimalSolAdd0TooBig(p: Problem, solution: Solution,
    i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 1 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i - 1, j)
 requires computeWeight(p, solution + [0],
    |solution + [0]| - 1) <= j
```

```
 requires p.weights[i - 1] > j
 ensures isOptimalPartialSolution(p, solution + [0], i, j)
{
  var s := solution + [0];
  weightAdd0(p, s);
  if !isOptimalPartialSolution(p, s, i, j) {
    existsOptimalPartialSol(p, i, j);
    var x : Solution :| isOptimalPartialSolution(p, x, i, j);
    gainAddTooBig(p, s, i, j);
    gainAddTooBig(p, x, i, j);
    var x1 := x[..i - 1];
    assert gain(p, x1) == gain(p, x) > gain(p, s);
    assert gain(p, s) == gain(p, solution) < gain(p, x);
    assert gain(p, x1) > gain(p, solution);
    assert isPartialSolution(p, x, i, j);
    assert x[i - 1] == 0;
    computeWeightAdd0(p, x, |x| - 2);
    assert weight(p, x) == weight(p, x1);
    assert isPartialSolution(p, x1, i - 1, j);
    assert !isOptimalPartialSolution(p, solution, i - 1, j);
    assert false;
  }
}
```

Este o altă lemă demonstrată folosind reducerea la absurd, de această dată pentru cazul în care greutatea obiectului depășește capacitatea j. Pornind de la premiza că soluția $solution + [0]$ nu este optimă pentru subproblema (i, j), înseamnă că există o altă soluție mai bună. Pentru a demonstra optimalitatea soluției a fost nevoie să arăt că eliminând ultimul element (corespunzător obiectului i) se ajunge la contrazicerea precondiției ce ne garantează că avem o soluție optimă pentru $(i - 1, j)$.

7. Lema ***existsOptimalPartialSol***

```
lemma existsOptimalPartialSol(p: Problem, i: int, j: int)
 requires isValidProblem(p)
```

```
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 ensures exists s :: isOptimalPartialSolution(p, s, i, j)
{
  var k : int := 0;
  var completeSol := seq(i, y => 1);
  assert forall q :: 0 <= q < i ==> completeSol[q] == 1;
  var sum := sumAllGains(p, i);
  assert forall k :: 0 <= k < i ==> p.gains[k] > 0;
  if !exists s :: isOptimalPartialSolution(p, s, i, j) {
    var q := 0;
    var currentSol := seq(i, y => 0);
    computeWeightAllZeros(p, currentSol, |currentSol| - 1);
    computeGainAllZeros(p, currentSol, |currentSol| - 1);
    assert computeGain(p, currentSol, |currentSol|-1) == 0 >= q;
    assert sum == sumAllGains(p, i);
    while q < sum + 1
      invariant 0 <= q <= sum + 1
      invariant !exists s :: isOptimalPartialSolution(p, s, i, j)
      invariant !isOptimalPartialSolution(p, currentSol, i, j)
      invariant isPartialSolution(p, currentSol, i, j)
      invariant computeGain(p, currentSol, |currentSol|-1) >= q
    {
      assert exists s_i :: isPartialSolution(p, s_i, i, j) &&
        gain(p, s_i) > gain(p, currentSol);
      var s_i :| isPartialSolution(p, s_i, i, j) &&
        gain(p, s_i) > gain(p, currentSol);
      currentSol := s_i;
      q := computeGain(p, s_i, |s_i| - 1);
      gainUpperBound(p, s_i, i);
    }
    assert computeGain(p, currentSol, |currentSol|-1) >= sum + 1;
    gainUpperBound(p, currentSol, i);
    assert false;
```

```
    }
}
```

După cum am văzut în lemele prezentate anterior, folosite pentru a demonstra optimalitatea soluțiilor, a fost necesară presupunerea că avem la îndemână o altă soluție optimă pentru subproblema (i, j) pe care încercăm să o rezolvăm. Pentru a exprima faptul că există o astfel de soluție, am folosit cuantificatorul existențial $exists$, însă Dafny deși acoperă destul de multe cazuri de unul singur, de multe ori întâlnește probleme de indecidabilitate când acesta este folosit. Astfel, a fost nevoie să formulez o lemă ce folosește acest cuantificator și să ghidez verificatorul să demonstreze existența unei soluții optime pentru o subproblemă cu i obiecte și un rucsac de capacitate j.

Pentru a demonstra acest lucru vom presupuse prin reducere la absurd că nu există o soluție optimă pentru subproblema (i, j), ceea ce înseamnă că pentru orice soluție va exista mereu una cu un profit mai mare. Pentru a itera prin soluțiile posibile am folosit o buclă $while$, iar invarianții sunt cei care asigură verificatorul că la fiecare iterație va exista o soluție cu un profit mai bun decât cea curentă. După un număr finit de pași, vom ajunge la o soluție ce are un profit mai mare decât valoarea tuturor obiectelor, lucru care nu este posibil, deci ipoteza

```
!exists s :: isOptimalPartialSolution(p, s, i, j)
```

de la care am plecat este falsă, iar postcondiția este verificată cu succes.

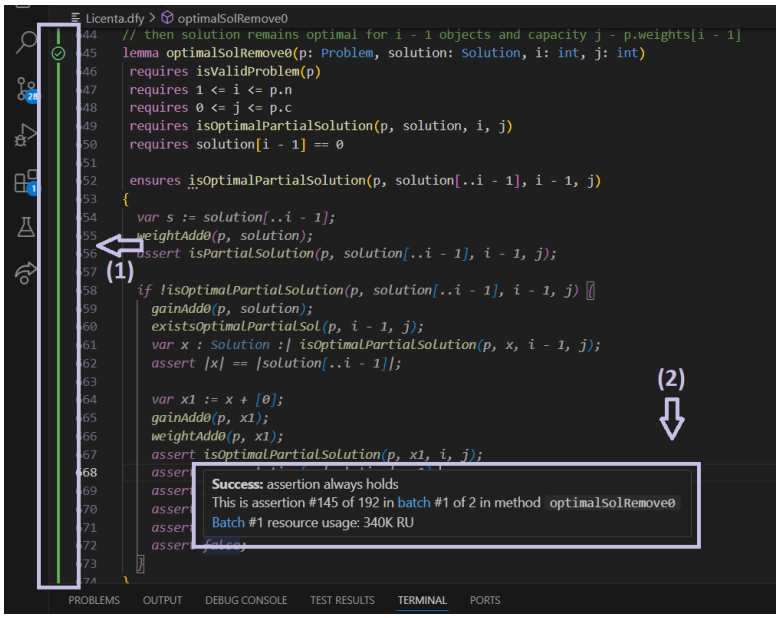
Capitolul 6

Provocări și aspecte practice ale procesului de lucru

În acest capitol aș dori să detaliez unele aspecte care consider că sunt destul de importante de menționat și unele provocări întâlnite pe parcursul procesului de dezvoltare.

Pentru realizarea proiectului am folosit Visual Studio ca mediu de dezolvare, acesta oferind o interfață intuitivă, dar și multe alte funcționalități care prin intermediul unui *Dafny Server Language* au ușurat procesul de dezvoltare, cum ar fi evidențierea sintaxei, verificarea codului pe măsură ce instrucțiunile sunt tastate și feedback visual pentru starea curentă a verificării formale a programului.

Aceasta din urmă a fost, în mod special, foarte folositoare pentru a mă asigura că verificarea formală a metodelor și a lemelor a trecut cu succes. Evidențierea verificării poate fi observată în partea din stânga a editorului (săgeata 1):



De exemplu, pentru lema ***optimalSolRemove0*** Dafny a reușit să verifice cu succes corectitudinea acesteia, iar acest lucru este evidențiat printr-o bară verticală de culoare verde pe toată lungimea declarației acesteia. În funcție de starea procesului de verificare, această bară își va schimba stilul în mod dinamic pentru a ajuta programatorul în procesul de dezvoltare al codului.

De asemenea, dacă plasăm cursorul peste o instrucțiune, o fereastră pop-up apare în care putem afla informații despre stadiul verificării acesteia (săgeata 2). Acest lucru este de ajutor mai mult în cazul invarianților pentru a afla dacă verificatorul nu poate demonstra corectitudinea acestuia înainte de execuția buclei sau în timpul acesteia.

Pentru a rula algoritmul, avem două variante:

- click dreapta în fișierul ce se dorește a fi rulat și alegem `Dafny` → `Dafny:Run` (sau `F5`)
- din linia de comandă se poate executa următoarea comanda: `dafny run Licenta.dfy` (mai multe opțiuni de control al verificării pot fi adăugate, precum `--allow-warnings`)

Una dintre cele mai comune provocări pe care le-am întâmpinat pe parcursul procesului de implementare a fost depășirea timpului de răspuns alocat pentru verificarea unei leme sau a unei metode, care poate apărea atunci când unele specificații nu sunt complete sau când logica demonstrației este mult prea complexă pentru verificator, dar există și cazuri în care acesta se pierde încercând să demonstreze lanțuri inutile de raționament. În astfel de cazuri am avut la îndemană următoarele posibilități:

- Utilizarea instrucțiunilor ***assert***: acestea sunt folosite pentru a verifica valoarea de adevăr a unei expresii logice necesare în demonstrație. Cu ajutorul acestora am verificat dacă Dafny poate aproba raționamentul pe care l-am aplicat în demonstrarea postcondițiilor deoarece de foarte multe ori a fost nevoie să ghidez verificatorul spre o anumită direcție logică, dar și unele proprietăți care trebuiau să fie adevărate după apelarea unor leme pentru a continua verificarea. Un exemplu foarte bun pentru ambele cazuri este lema ***gainAddTooBig***:

```
lemma gainAddTooBig(p: Problem, solution: Solution,
    i: int, j: int)
 ...
```

```
 requires isPartialSolution(p, solution, i, j)
 requires p.weights[i - 1] > j
 ensures solution[i - 1] == 0
 ensures gain(p, solution[..i - 1]) == gain(p, solution)
 {
    if solution[i - 1] == 1 {
      assert computeWeight(p, solution, |solution| - 1) ==
        computeWeight(p, solution, |solution[..i]| - 1) +
        p.weights[i - 1];
      assert weight(p, solution) >= p.weights[i - 1] > j;
      assert !isPartialSolution(p, solution, i, j);
      assert false;
    }
    assert solution[i - 1] == 0;
    computeGainAdd0(p, solution, |solution| - 2);
    assert gain(p, solution[..i - 1]) == gain(p, solution);
}
```

unde assert-urile din instrucțiunea if urmăresc să ajungem la o contradicție prin faptul că soluția nu are cum să fie parțială dacă greutatea obiectului depășește capacitatea j, propoziție care este falsă deoarece avem o precondiție care asigură exact acest lucru, iar instrucțiunea

```
assert gain(p, solution[..i - 1]) == gain(p, solution);
```

se asigură că după apelarea lemei computeGainAdd0 verificatorul știe că profitul unei soluții rămâne neschimbat dacă eliminăm ultimul element de 0 din soluție.

De asemenea, folosind structura `assert ... by` putem reduce sarcina verificatorului deoarece pașii din acest bloc de instrucțiuni sunt ”uitați” după verificarea condiției din assert:

```
assert gain(p, x[..i - 1]) == gain(p, solution1) by
{
  optimalSolRemove1(p, x, i, j);
  assert isOptimalPartialSolution(p, x[..i - 1],
```

```
        i - 1, j - p.weights[i - 1]);
}
```

- Utilizarea instrucțiunii ***assume***: această instrucțiune este foarte utilă pentru a determina la ce linie întâlnește Dafny probleme. O instrucțiune assume tratează condiția ca fiind adevarată, de aceea de multe ori am folosit-o pentru a afla de ce informații în plus are nevoie verificatorul sau pentru a putea continua următorii pași ai demonstrației. De exemplu, mi-a fost de ajutor în cazul lemelor în care presupun că există o soluție mai bună decât cea pentru care vreau să demonstrez că este optimă, întrucât am putut continua implementarea acestora, după care am revenit să formulez o lemă specială pentru a demonstra existența unor astfel de soluții.

  O altă utilizare a instrucțiunii assume a fost `assume false;` care oprește verificarea pentru condițiile ce urmează după aceasta și îi indică verificatorului să nu mai încerce demonstrarea acestora și să accepte orice afirmație ulterioară ca fiind adevărată. Aceasta instrucțiune a fost foarte folositoare pentru a identifica ce asserturi nu reușea Dafny să demonstreze pentru lemele mai complexe, mai ales în cazul instrucțiunilor if din cadrul lemelor unde pe fiecare ramură aveam o logică diferită și ambele puteau cauza probleme de logică.

- ***Refactorizarea codului***: în leme precum ***optimalSolAdd1*** sau ***optimalSolAdd0***, unde deși demonstrația era corectă și fiecare caz (când ultimul element este fie 0, fie 1) se verifica cu succes separat, dacă ambele erau tratate în cadrul aceleași leme obțineam acest timeout deoarece demonstrația era destul de complexă pentru Dafny. Soluția a fost astfel să tratez unul dintre cele două cazuri posibile într-o lemă separată.

Concluzii

Bibliografie

1. https://en.wikipedia.org/wiki/Dafny
2. https://dafny.org/dafny/DafnyRef/DafnyRef
3. https://www.javatpoint.com/dynamic-programming
4. https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/
5. https://sites.google.com/view/fii-pa/2024/lectures
6. https://www.w3schools.com/dsa/dsa_ref_greedy.php
7. https://medium.com/@prekshayadav0819/understanding-the-knapsack-problem-a-guide-for-beginners-d0146a59e9
8. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
9. https://dafny.org/dafny/OnlineTutorial/guide
10. https://github.com/dafny-lang/dafny/wiki/FAQ#how-does-dafny-handle-quantifiers-ive-heard-about-triggers-what-are-those
11. https://dafny.org/blog/2023/04/19/making-verification-compelling-visual-verification-feedback-for-dafny/#step-0-1
12. https://dafny.org/latest/VerificationOptimization/VerificationOptimization