UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Verificarea formală în Dafny a algoritmului Greedy pentru Problema Bancnotelor cu bancnote puteri ale lui 2

propusă de

Alexandra Elena Contur

Sesiunea: februarie, 2023

Coordonator științific

Conf. Dr. Ciobâcă Ștefan

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea formală în Dafny a algoritmului Greedy pentru Problema Bancnotelor cu bancnote puteri ale lui 2

Alexandra Elena Contur

Sesiunea: februarie, 2023

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 **Contur Alexandra Elena** domiciliat în **România, jud. Iași, com. Holboca, str. Dascălilor, nr. 10**, născut la data de **04 martie 2001**, identificat prin CNP **6010304226743**, 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 formală în Dafny a algoritmului Greedy pentru Problema Bancnotelor cu bancnote puteri ale lui 2** 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 formală în Dafny a algoritmului Greedy pentru Problema Bancnotelor cu bancnote puteri ale lui 2**, 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 **Alexandra Elena Contur**

Data: ..... Semnătura: .....

Cuprins

Capitolul 1

Context

1.1 Limbajul Dafny

Dafny este un limbaj de programare și verificare, capabil să verifice corectitudinea funcțională a unui program.

Verificarea este posibilă datorită caracteristicilor specifice limbajului precum precondiții, postcondiții, invariante, ș.a.m.d. De asemenea, verificatorul Dafny are grijă ca adnotările făcute să se îndeplinească, astfel acesta ne scapă de povara de a scrie cod fără erori, în schimbul scrierii de adnotări fără erori.

1.2 Algoritmul Greedy pentru Problema Bancnotelor

Problema Bancnotelor are ca scop reprezentarea unei sume într-un număr minim posibil de bancnote.

Metoda Greedy face alegerea cea mai bună la fiecare pas, construind soluția finală. Verificarea formală in Dafny a problemei bancnotelor demonstrează faptul că soluția construită este optimă pentru orice sumă dată ca input.

Reprezentarea soluției : $banknote_1 := 1 < banknote_2 < ... < banknote_n$.

Capitolul 2

Particularități ale problemei alese

2.1 Problema Bancnotelor cu bancnote puteri ale lui 2

Anterior am menționat forma generală a Problemei Bancnotelor.

Bancnotele posibile în ”Problema Bancnotelor cu bancnote puteri ale lui 2” sunt: $[1, 2, 4, 8, 16, 32]$.

2.2 Observații

Datorită bancnotelor care sunt puteri ale lui 2, dacă avem mai mult de o bancnotă de valoare mai mică decât 32, putem înlocui 2 bancnote de acea valoare cu o bancnotă de valoarea următoare și am obține o soluție cu cost mai mic.

Astfel, am descoperit proprietatea: $forall\ i :: 0 <= i <= 4 ==> s[i] <= 1$

Capitolul 3

Verificarea formală a problemei

3.1 Implementarea algoritmului

Implementarea algoritmului propriu-zis care rezolvă problema a fost primul și cel mai ușor pas.

Am început prin crearea unei bucle care la fiecare pas alegea bancnota optimă, o adăuga în secvența de bancnote considerată soluție și o scădea din sumă, fiind un algoritm tipic metodei Greedy.

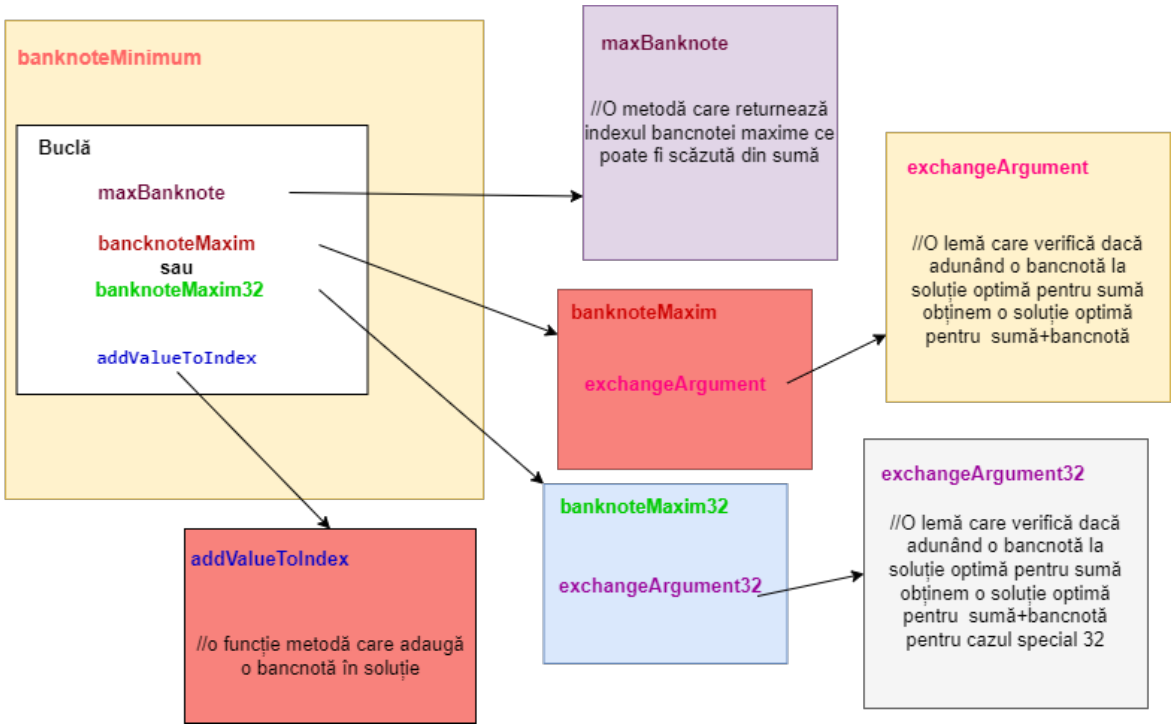
```
1   var rest:= sum;
2   solution:= [0, 0, 0, 0, 0, 0];
3   while (0 < rest)
4      decreases rest
5      {
6      index:= maxBanknote(rest);
7      var banknote:= power(2, index);
8      solution:= addValueToIndex(solution,1,index);
9      rest:= rest - banknote;
10  }
```

Acest algoritm era suficient pentru a rezolva problema, dar nu era suficient pentru a demonstra că soluția produsă este optimă.

Considerăm o soluție optimă soluția de cost minim, costul fiind numărul de bancnote.

3.2 Demonstrarea optimalității

3.2.1 Schema verificării



În schema de mai sus am exemplificat modul în care funcțiile, lemele și metodele se apelează una pe cealaltă pentru a demonstra faptul că soluția găsită este soluție optimă.

3.2.2 Condiții ca o soluția finală să fie optimă

• Pentru a avea o soluție optimă trebuie să avem o soluție validă (cu 6 elemente), care produce suma corectă și care are costul cel mai mic.

• Pentru a avea o soluție optimă finală, pe parcursul construirii soluției, în buclă, trebuie menținută proprietatea de a alege soluția optimă locală pentru rest, iar suma soluțiilor optime locale să fie soluție optimă pentru sumă.

• Soluția formată dintr-o bancnotă, cea aleasă în iterația curentă, produce o soluție optimă pentru suma de valoare bancnotă, asigurând faptul că avem o soluție optimă locală.

• O soluție optimă pentru suma x, adunată cu o soluție optimă pentru suma y creează o soluție optimă pentru suma $x + y$, asigurând faptul că suma soluțiilor locale creează soluția finală optimă.

3.3 Pașii verificării

Am implementat algoritmul Greedy care rezolva Problema Bancnotelor cu bancnote puteri ale lui 2, apoi am adăugat treptat condițiile care garantează că soluția găsită este soluție optimă.

3.3.1 banknoteMinimum

Am implementat metoda principală banknoteMinimum care returnează soluția corectă pentru o sumă dată ca input.

Aceasta se asigură că returnează o soluție validă, care produce suma corectă si optimă.

Această metodă folosește o buclă, în care, la fiecare iterație, se adaugă o bancnotă în soluție.

În primă fază calculează bancnota maximă ce poate fi dată ca rest, cu ajutorul funcției maxBanknote și funcției power, apoi folosește o lemă pentru a demonstra că soluția curentă adunată cu bancnota aleasă, va genera o soluție optimă pentru întreaga sumă, dacă este adunată cu o soluție optimă pentru restul rămas de dat - bancnota aleasă.

Această demonstrație se realizează cu ajutorul lemei banknoteMaxim sau banknoteMaxim32 în funcție de bancnota aleasă.

În final adaugăm bancnota în soluție cu ajutorul funcției addValueToIndex și scădem din restul de dat bancnota adaugată în sumă. Proprietățile necesare unei soluții optime se păstreaza pe parcursul buclei cu ajutorul invariantelor.

Voi detalia metodele și lemele mai complexe.

```
1   method banknoteMinimum(sum: int) returns(solution: seq < int > )
2       requires sum >= 0
3       ensures isValidSolution(solution)
4       ensures isSolution(solution, sum)
5       ensures isOptimalSolution(solution, sum)
6   {
7       var rest:= sum;
8       solution:= [0, 0, 0, 0, 0, 0];
9       var index:= 0;
10      assert isOptimalSolution(solution, sum - rest);
11      while (0 < rest)
```

```
12          invariant 0 <= rest <= sum
13          invariant isValidSolution(solution)
14          invariant addOptimRestEqualsOptimSum(rest, sum, solution)
15          decreases rest
16       {
17          index:= maxBanknote(rest);
18          var banknote:= power(2, index);
19          if (index != 5)
20          {
21             banknoteMaxim(rest, sum, solution, index);
22          }
23          else
24          {
25             banknoteMaxim32(rest, sum, solution);
26          }
27          solution:= addValueToIndex(solution,1,index);
28          rest:= rest - banknote;
29       }
30    }
```

3.3.2 maxBanknote

Metoda maxBanknote este folosită pentru a returna un index cu proprietatea: $bancnota_{index} <= rest < bancnota_{index+1}$.

```
1     method maxBanknote(sum: int) returns(index: int)
2       requires sum > 0
3       ensures 0 <= index <= 5
4       ensures 0 <= power(2, index) <= sum
5       ensures(index != 5 && power(2, index + 1) > sum) || index == 5
6     {
7       index:= 5;
8       if (power(2, index) > sum)
9       {
10        assert power(2, index + 1) > sum;
11        while (power(2, index) > sum && index > 0)
12          invariant power(2, index + 1) > sum
13          {
14            index:= index - 1;
15            assert power(2, index + 1) > sum;
```

```
        }
      }
       else
      {
       assert index == 5;
      }
    }
```

3.3.3 banknoteMaxim

Lema banknoteMaxim este folosită pentru a demonstra faptul că dacă adăugam o bancnotă în soluția pe care o construim, în continuare suma acestei soluții cu soluția optimă pentru $rest - bancnota$ va produce soluția optimă pentru suma întreagă.

Pentru a demonstra acest lucru avem nevoie să știm că dacă în soluția curentă, optimă pentru $rest - bancnota$ adaugăm bancnota aceasta devine optimă pentru $rest$.

Acest lucru este demonstrat cu ajutorul lemei exchangeArgument.

```
lemma banknoteMaxim(rest: int, sum: int, finalSolution: seq < int > ,
    index: int)
    requires 0 <= index <= 4
    requires power(2, index) <= rest < power(2, index + 1)
    requires isValidSolution(finalSolution)
    requires addOptimRestEqualsOptimSum(rest, sum, finalSolution)
    ensures addOptimRestEqualsOptimSum(rest - power(2, index), sum,
        finalSolution[index:= finalSolution[index] + 1])
{
    var banknote:= power(2, index);
    forall currentSolution | isValidSolution(currentSolution) &&
        isOptimalSolution(currentSolution, rest - banknote)
        ensures
            isOptimalSolution(solutionsSum(solutionsSum(currentSolution,
            finalSolution), [0, 0, 0, 0, 0, 0][index:= 1]), sum)
    {
        assert isSolution(currentSolution[index:= currentSolution[index]
            + 1], rest);
        exchangeArgument(rest, currentSolution, index);
    }
    assert forall currentSolution::isValidSolution(currentSolution) &&
        isOptimalSolution(currentSolution, rest - banknote) ==>
```

```
        isOptimalSolution(solutionsSum(solutionsSum(currentSolution,
        finalSolution), [0, 0, 0, 0, 0, 0][index:= 1]), sum);
16  }
```

3.3.4 exchangeArgument

Lema exchangeArgument presupune că nu este optimă soluția dacă adaugăm bancnota aleasă și ajunge la o contradicție. Știm că soluția e optimă pentru $rest - bancnota$, dar nu pentru rest, dacă adaugăm bancnota.

Consideră că există o altă soluție optimă pentru rest.

Verifică în presupusa soluție pentru rest că nu avem bancnota în soluție deja, altfel acea presupusă $solutie - bancnota$ are cost mai mic decât soluția noastră care e optimă pentru $rest - bancnota$, contradicție. Apoi verificăm proprietatea descrisă la observație.

Nu există două bancnote de aceeași valoare în soluție, mai mici de 32.

În acest mod știm că soluția presupusă nu poate fi optimă, deoarece, dacă respectă proprietatea suma produsă de aceasta nu poate fi egală cu rest.

Deoarece : $\bullet \sum_{k=0}^{index-1} 2^k = 2^{index} - 1 \bullet rest >= 2^{index}$ Deci presupusa soluție pentru rest nu poate fi egală decât cu cel mult rest-1, astfel nu poate fi soluție optimă și se ajunge la o contradicție.

```
1       lemma exchangeArgument(rest: int, currentSolution: seq < int > ,
            index: int)
2           requires 0 <= index <= 4
3           requires power(2, index) <= rest < power(2, index + 1)
4           requires isValidSolution(currentSolution)
5           requires isOptimalSolution(currentSolution, rest - power(2,
                index))
6           ensures isOptimalSolution(currentSolution[index:=
                currentSolution[index] + 1], rest)
7       {
8           var banknote:= power(2, index);
9           var solution:= currentSolution[index:= currentSolution[index] +
                1];
10          assert isValidSolution(solution);
11          assert isSolution(solution, rest);
12          var i:= index;
```

```
13          if (!isOptimalSolution(solution, rest))
14          {
15          var optimalSolution:| isValidSolution(optimalSolution) &&
                isSolution(optimalSolution, rest) &&
16              isOptimalSolution(optimalSolution, rest) &&
                    cost(optimalSolution) < cost(solution);
17          if (optimalSolution[index] -1 >= 0)
18          {
19              var betterSolution:= addValueToIndex(optimalSolution,-1,index);
20              assert isSolution(betterSolution, rest - banknote);
21              assert cost(betterSolution) == cost(optimalSolution) - 1;
22              assert cost(optimalSolution) - 1 < cost(currentSolution);
23              assert false;
24          }
25          else
26          {
27              while (0 < i)
28              invariant 0 <= i <= index
29              invariant forall x::index >= x >= i ==> optimalSolution[x] <= 1
30              {
31              i:= i - 1;
32              assert isOptimalSolution(optimalSolution, rest);
33              if (optimalSolution[i] > 1)
34              {
35                  var optimalSolution' :=
                        optimalSolution[i:=optimalSolution[i]-2];
36                  optimalSolution' := optimalSolution' [i + 1:=
                        optimalSolution'[i+1]+1];
37                  assert isSolution(optimalSolution', rest);
38                  assert cost(optimalSolution') == cost(optimalSolution) - 1;
39                  assert cost(optimalSolution') < cost(optimalSolution);
40                  assert false;
41              }
42              }
43              assert solutionElementsSum(optimalSolution) <= banknote - 1;
44              assert rest >= banknote;
45              assert solutionElementsSum(optimalSolution) <= rest - 1;
46              assert isOptimalSolution(optimalSolution, rest);
47              assert false;
48              }
```

```
49            }
50        }
```

3.3.5 Ultimul pas

În final, am descoperit că pentru bancnota 32 am nevoie de o verificare diferită. Voi descrie ulterior cum am tratat acest caz.

• De ce trebuie abordat diferit cazul când bancnota optimă de adăugat este 32?

Fiecare bancnotă 1,2,4,8 și 16 este mărginită superior de bancnota imediat următoare.

Felul în care abordez cazurile 1,2,4,8 și 16 constă în verificarea faptului că nu avem încă o bancnotă de aceeași valoare în soluție deja, altfel ar fi mai eficient să avem o bancnotă de valoarea imediat următoare și să scăpăm de bancnota deja existentă din soluție.

În cazul bancnotei 32 nu există o altă bancnotă de valoare mai mare cu care putem să înlocuim apariția bancnotei 32, așadar a trebuit să demonstrez în alt mod că adăugând bancnota 32 la soluția găsită până în prezent genereaz o soluție optimă pentru pentru rest.

3.4 Problema bancnotei nemărginite superior, 32

3.4.1 Cum se tratează separat cazul 32

3.4.2 banknoteMaxim32

```
1  lemma banknoteMaxim32(rest: int, sum: int, finalSolution: seq < int > )
2   requires rest >= 32
3   requires isValidSolution(finalSolution)
4   requires addOptimRestEqualsOptimSum(rest, sum, finalSolution)
5   ensures addOptimRestEqualsOptimSum(rest - 32, sum,
        solutionsSum(finalSolution, [0, 0, 0, 0, 0, 1]))
6  {
7   forall currentSolution | isValidSolution(currentSolution) &&
        isSolution(currentSolution, rest - 32)
8    ensures isSolution(solutionsSum(solutionsSum(finalSolution,
         currentSolution), [0, 0, 0, 0, 0, 1]), sum)
9   {
```

```
10    assert isSolution(solutionsSum(currentSolution, [0, 0, 0, 0, 0, 1]),
          rest);
11  }
12
13  forall currentSolution | isValidSolution(currentSolution) &&
      isOptimalSolution(currentSolution, rest - 32)
14    ensures isOptimalSolution(solutionsSum(solutionsSum(finalSolution,
          currentSolution), [0, 0, 0, 0, 0, 1]), sum)
15  {
16    forall someSolution | isValidSolution(someSolution) &&
          isSolution(someSolution, sum)
17      ensures cost(someSolution) >=
            cost(solutionsSum(solutionsSum(finalSolution, currentSolution), [0,
            0, 0, 0, 0, 1]))
18    {
19      currentSolutionHasCostMin(rest, sum, currentSolution);
20    }
21  }
22 }
```

3.4.3 currentSolutionHasCostMin

```
1     lemma currentSolutionHasCostMin(rest: int, sum: int, solution: seq <
          int > )
2   requires isValidSolution(solution)
3   requires rest >= 32
4   requires isSolution(solution, rest - 32)
5   requires isOptimalSolution(solution, rest - 32)
6   ensures isOptimalSolution(solutionsSum(solution, [0, 0, 0, 0, 0, 1]),
        rest)
7 {
8   forall someSolution | isValidSolution(someSolution) &&
        isSolution(someSolution, rest)
9     ensures cost(someSolution) >= cost(solutionsSum(solution, [0, 0, 0, 0,
          0, 1]))
10  {
11    exchangeArgument32(rest, sum, someSolution, solution);
12  }
13 }
```

3.4.4 exchangeArgument32

```
1
2  lemma exchangeArgument32(rest: int, sum: int, currentSolution: seq < int >
       , optimalSolution: seq < int > )
3     requires 32 <= rest
4     requires isValidSolution(optimalSolution)
5     requires isOptimalSolution(optimalSolution, rest - 32)
6     ensures isOptimalSolution(optimalSolution[5:= optimalSolution[5] + 1],
          rest)
7  {
8     var solution:= optimalSolution[5:= optimalSolution[5] + 1];
9     var i:= 4;
10    if (!isOptimalSolution(solution, rest))
11    {
12       if (optimalSolution[i] > 1)
13       {
14          var solution:= optimalSolution[i:= optimalSolution[i] - 2];
15          solution:= solution[i + 1:= solution[i + 1] + 1];
16          assert isSolution(solution, rest - 32);
17          assert cost(solution) == cost(optimalSolution) - 1;
18          assert cost(optimalSolution) - 1 < cost(solution);
19          assert false;
20       }
21       else
22       {
23          while (0 < i)
24             invariant 0 <= i <= 4
25             invariant forall index::4 >= index >= i ==>
                   optimalSolution[index] <= 1
26          {
27             i:= i - 1;
28             if (optimalSolution[i] > 1)
29             {
30                var solution:= optimalSolution[i:= optimalSolution[i] - 2];
31                solution:= solution[i + 1:= solution[i + 1] + 1];
32                assert isSolution(solution, rest - 32);
33                assert cost(solution) == cost(optimalSolution) - 1;
34                assert cost(optimalSolution) - 1 < cost(solution);
35                assert false;
36             }
```

```
37              assert optimalSolution[i] <= 1;
38          }
39          assert solutionElementsSum(optimalSolution) <= rest - 1;
40          assert isOptimalSolution(solution, rest);
41          assert false;
42      }
43    }
44 }
```

Concluzii

Bibliografie

1. https://github.com/alexxandra21/Licenta2023
2. https://core.ac.uk/works/69828860
3. https://arxiv.org/pdf/1412.4395.pdf
4. http://www.cse.unsw.edu.au/ se2011/DafnyDocumentation/Dafny