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.

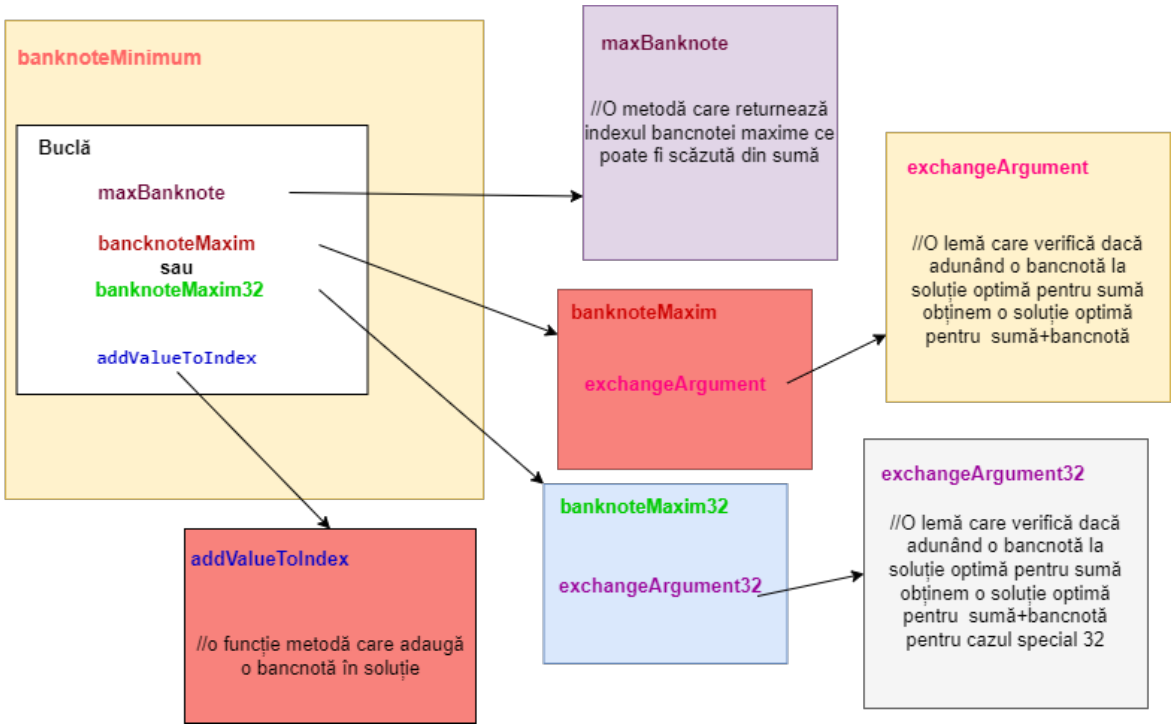
```
var rest:= sum;
solution:= [0, 0, 0, 0, 0, 0];
while (0 < rest)
   decreases rest
   {
   index:= maxBanknote(rest);
   var banknote:= power(2, index);
   solution:= addValueToIndex(solution,1,index);
   rest:= rest - banknote;
}
```

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

3.3.1 banknoteMinimum

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.

```
method banknoteMinimum(sum: int) returns(solution: seq < int > )
   requires sum >= 0
   ensures isValidSolution(solution)
   ensures isSolution(solution, sum)
   ensures isOptimalSolution(solution, sum)
{
   var rest:= sum;
   solution:= [0, 0, 0, 0, 0, 0];
   var index:= 0;
   assert isOptimalSolution(solution, sum - rest);
   while (0 < rest)
      invariant 0 <= rest <= sum
      invariant isValidSolution(solution)
      invariant addOptimRestEqualsOptimSum(rest, sum, solution)
      decreases rest
   {
      index:= maxBanknote(rest);
      var banknote:= power(2, index);
      if (index != 5)
      {
         banknoteMaxim(rest, sum, solution, index);
      }
      else
      {
         banknoteMaxim32(rest, sum, solution);
      }
      solution:= addValueToIndex(solution,1,index);
      rest:= rest - banknote;
   }
```

```
}
```

3.3.2 maxBanknote

Pentru a alege bancnota cea mai mare am creat funcția maxBanknote.

```
method maxBanknote(sum: int) returns(index: int)
  requires sum > 0
  ensures 0 <= index <= 5
  ensures 0 <= power(2, index) <= sum
  ensures(index != 5 && power(2, index + 1) > sum) || index == 5
{
  index:= 5;
  if (power(2, index) > sum)
  {
    assert power(2, index + 1) > sum;
    while (power(2, index) > sum && index > 0)
      invariant power(2, index + 1) > sum
      {
        index:= index - 1;
        assert power(2, index + 1) > sum;
      }
  }
    else
  {
    //we know the index is 5
    assert index == 5;
  }
}
```

3.3.3 banknoteMaxim

```
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);
}
```

3.3.4 exchangeArgument

```
    lemma exchangeArgument(rest: int, currentSolution: seq < int >
        , index: int)
        requires 0 <= index <= 4
        requires power(2, index) <= rest < power(2, index + 1)
        requires isValidSolution(currentSolution)
        requires isOptimalSolution(currentSolution, rest - power(2,
            index))
        ensures isOptimalSolution(currentSolution[index:=
            currentSolution[index] + 1], rest)
    {
```

```
var banknote:= power(2, index);
var solution:= currentSolution[index:=
   currentSolution[index] + 1];
assert isValidSolution(solution);
assert isSolution(solution, rest);
var i:= index;
if (!isOptimalSolution(solution, rest))
{
var optimalSolution:| isValidSolution(optimalSolution) &&
   isSolution(optimalSolution, rest) &&
   isOptimalSolution(optimalSolution, rest) &&
      cost(optimalSolution) < cost(solution);

assert cost(solution) == cost(currentSolution) + 1;
assert isOptimalSolution(optimalSolution, rest);

if (optimalSolution[index] -1 >= 0)
{
   assert optimalSolution[index]-1 >= 0;
   var betterSolution:=
      addValueToIndex(optimalSolution,-1,index);
   assert isSolution(betterSolution, rest - banknote);
   assert cost(betterSolution) == cost(optimalSolution) - 1;
   assert cost(optimalSolution) - 1 < cost(currentSolution);
   assert false;
}
else
{
   while (0 < i)
   invariant 0 <= i <= index
   invariant forall x::index >= x >= i ==>
      optimalSolution[x] <= 1
   {
   i:= i - 1;
   assert isOptimalSolution(optimalSolution, rest);
   if (optimalSolution[i] > 1)
```

```
        {
            var optimalSolution' :=
                optimalSolution[i:=optimalSolution[i]-2];
            optimalSolution' := optimalSolution' [i + 1:=
                optimalSolution'[i+1]+1];
            assert isSolution(optimalSolution', rest);
            assert cost(optimalSolution') ==
                cost(optimalSolution) - 1;
            assert cost(optimalSolution') < cost(optimalSolution);
            assert false;
        }
        }
        assert solutionElementsSum(optimalSolution) <= banknote
            - 1; assert rest >= banknote; assert
            solutionElementsSum(optimalSolution) <= rest - 1;
            assert isOptimalSolution(optimalSolution, rest);
            assert false;
    }
    }
}
```

3.3.5 addValueToIndex

```
function method addValueToIndex(solution: seq<int>, value:
    int, index: int): seq<int>
    requires 0 <= index <= 5
    requires isValidSolution(solution)
    requires solution[index] + value >= 0
    ensures isValidSolution(solution)
    ensures solutionElementsSum(solution) + value*power(2,
        index) ==
        solutionElementsSum(solution[index:=solution[index]+value])
{
solution[index:= solution[index] + value]
}
```

3.3.6 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 abordata diferit-

3.4 Problema bancnotei nemărginite superior, 32

3.4.1 Cum se tratează separat cazul 32

3.4.2 banknoteMaxim32

```
lemma banknoteMaxim32(rest: int, sum: int, finalSolution: seq < int
   > )
 requires rest >= 32
 requires isValidSolution(finalSolution)
 requires addOptimRestEqualsOptimSum(rest, sum, finalSolution)
 ensures addOptimRestEqualsOptimSum(rest - 32, sum,
    solutionsSum(finalSolution, [0, 0, 0, 0, 0, 1]))
{
 forall currentSolution | isValidSolution(currentSolution) &&
    isSolution(currentSolution, rest - 32)
  ensures isSolution(solutionsSum(solutionsSum(finalSolution,
     currentSolution), [0, 0, 0, 0, 0, 1]), sum)
 {
  assert isSolution(solutionsSum(currentSolution, [0, 0, 0, 0, 0,
     1]), rest);
 }

 forall currentSolution | isValidSolution(currentSolution) &&
    isOptimalSolution(currentSolution, rest - 32)
  ensures
     isOptimalSolution(solutionsSum(solutionsSum(finalSolution,
     currentSolution), [0, 0, 0, 0, 0, 1]), sum)
 {
```

```
    forall someSolution | isValidSolution(someSolution) &&
       isSolution(someSolution, sum)
     ensures cost(someSolution) >=
        cost(solutionsSum(solutionsSum(finalSolution,
        currentSolution), [0, 0, 0, 0, 0, 1]))
    {
     currentSolutionHasCostMin(rest, sum, currentSolution);
    }
  }
}
```

3.4.3 currentSolutionHasCostMin

```
    lemma currentSolutionHasCostMin(rest: int, sum: int, solution:
       seq < int > )
  requires isValidSolution(solution)
  requires rest >= 32
  requires isSolution(solution, rest - 32)
  requires isOptimalSolution(solution, rest - 32)
  ensures isOptimalSolution(solutionsSum(solution, [0, 0, 0, 0, 0,
     1]), rest)
{
  forall someSolution | isValidSolution(someSolution) &&
     isSolution(someSolution, rest)
   ensures cost(someSolution) >= cost(solutionsSum(solution, [0, 0,
      0, 0, 0, 1]))
  {
   exchangeArgument32(rest, sum, someSolution, solution);
  }
}
```

3.4.4 exchangeArgument32

```
lemma exchangeArgument32(rest: int, sum: int, currentSolution: seq <
   int > , optimalSolution: seq < int > )
   requires 32 <= rest
   requires isValidSolution(optimalSolution)
   requires isOptimalSolution(optimalSolution, rest - 32)
   ensures isOptimalSolution(optimalSolution[5:= optimalSolution[5]
      + 1], rest)
{
   var solution:= optimalSolution[5:= optimalSolution[5] + 1];
   var i:= 4;
   if (!isOptimalSolution(solution, rest))
   {
      if (optimalSolution[i] > 1)
      {
         var solution:= optimalSolution[i:= optimalSolution[i] - 2];
         solution:= solution[i + 1:= solution[i + 1] + 1];
         assert isSolution(solution, rest - 32);
         assert cost(solution) == cost(optimalSolution) - 1;
         assert cost(optimalSolution) - 1 < cost(solution);
         assert false;
      }
      else
      {
         while (0 < i)
            invariant 0 <= i <= 4
            invariant forall index::4 >= index >= i ==>
               optimalSolution[index] <= 1
         {
            i:= i - 1;
            if (optimalSolution[i] > 1)
            {
               var solution:= optimalSolution[i:= optimalSolution[i]
                  - 2];
               solution:= solution[i + 1:= solution[i + 1] + 1];
               assert isSolution(solution, rest - 32);
               assert cost(solution) == cost(optimalSolution) - 1;
```

```
                assert cost(optimalSolution) - 1 < cost(solution);
                assert false;
            }
            assert optimalSolution[i] <= 1;
        }
        assert solutionElementsSum(optimalSolution) <= rest - 1;
        assert isOptimalSolution(solution, rest);
        assert false;
    }
  }
}
```

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