Paradigmas de Resolução de Problemas

Programação Dinâmica: Problema do Troco

Prof. Edson Alves - UnB/FGA

2020

Sumário

Definição

Definição

Problema do Troco

Seja $C = \{c_1, c_2, \ldots, c_N\}$ uma sequência ordenada de N inteiros positivos distintos e M um inteiro positivo. O problema do troco consiste em determinar um vetor de inteiros não-negativos $x = \{x_1, x_2, \ldots, x_N\}$ tal que

$$M = \sum_{i=1}^{N} x_i c_i$$

e que a soma

$$\sum_{i=1}^{N} x_i$$

seja mínima.

Características do problema do troco

- Os elementos do conjunto C são denominados *moedas*
- M é o *troco*
- O problema pode ser definido informalmente como: *Qual é o menor número de moedas necessárias para dar o troco M?*
- Se $c_1 = 1$, há solução para qualquer M
- Se C é o conjunto de moedas utilizadas no sistema financeiro da maioria dos países, o problema do troco pode ser resolvido por meio de um algoritmo guloso

Algoritmo guloso para o problema do troco

- O algoritmo guloso para o problema do troco escolhe, dentre as moedas, a maior delas (c_k) que é menor ou igual a M
- Em seguida, ele atribui a x_k o valor M/c_k e subtrai de M o valor $x_k c_k$
- O algoritmo então prossegue até que M se torne igual a zero
- Para todos os valores x_i não atribuídos durante o algoritmo, vale que $x_i = 0$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	
5	25	
4	10	
3	5	
2	2	
1	1	

$$M = 73$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	
4	10	
3	5	
2	2	
1	1	

$$M = 23$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	
3	5	
2	2	
1	1	

$$M = 23$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	
2	2	
1	1	

$$M = 3$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	0
2	2	
1	1	

$M = 3$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	0
2	2	1
1	1	

$$M = 1$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	0
2	2	1
1	1	1

$M = 0$

Implementação do algoritmo guloso para o problema do troco

```cpp
#include <bits/stdc++.h>

using namespace std;

vector<int> coin_change(int M, const vector<int>& cs)
{
    int N = (int) cs.size();
    vector<int> xs(N);

    for (int i = N - 1; i >= 0; --i)
    {
        xs[i] = M / cs[i];
        M -= (xs[i] * cs[i]);
    }

    return xs;
}

```

Implementação do algoritmo guloso para o problema do troco

```cpp
int main()
{
    vector<int> cs { 1, 2, 5, 10, 25, 50 };
    int M;

    cin >> M;

    auto xs = coin_change(M, cs);

    for (size_t i = 0; i < cs.size(); ++i)
        cout << cs[i] << ": " << xs[i] << '\n';

    cout << accumulate(xs.begin(), xs.end(), 0) << " moedas\n";

    return 0;
}
```

Incorretude do algoritmo guloso

- O algoritmo guloso para o problema do troco, porém, não produz a solução correta para todas as entradas possíveis
- Por exemplo, se $C = \{1, 4, 5\}$ e $M = 8$, o algoritmo guloso retornaria 4 moedas: uma de 5 e três de 1
- Contudo, é possível dar um troco de 8 com apenas duas moedas de 4
- Assim, para obter a solução correta para qualquer troco e qualquer conjunto de moedas, é preciso utilizar um algoritmo baseado em outro paradigma que não o guloso

Algoritmo de programação dinâmica

- A programação dinâmica pode ser utilizada para desenvolver um algoritmo correto para o problema do troco
- Seja $c(m)$ o mínimo de moedas necessárias para um troco igual a m
- O caso base acontece quando $m = 0$: neste caso, $c(0) = 0$
- As transições acontecem para cada uma das moedas $c_k \leq m$:

$$c(m) = \min\{c(m - c_{k_1}), c(m - c_{k_2}), \ldots, c(m - c_{k_r})\} + 1$$

- Isto corresponde a escolher uma moeda (o termo $+1$) que seja menor ou igual ao troco e computar o troco mínimo para o restante
- São $O(M)$ estados distintos e as transições são feitas em $O(N)$
- Portanto a complexidade do algoritmo é $O(MN)$

Algoritmo de programação dinâmica para o problema do troco

```
#include <bits/stdc++.h>

using namespace std;

const int MAX { 1010 }, oo { 1000000010 };
int st[MAX];

int coin_change(int m, const vector<int>& cs)
{
    if (m == 0)
        return 0;

    if (st[m] != -1)
        return st[m];

    auto res = oo;

    for (auto c : cs)
        if (c <= m)
            res = min(res, coin_change(m - c, cs) + 1);

```

Algoritmo de programação dinâmica para o problema do troco

```cpp
    st[m] = res;
    return res;
}

int main()
{
    memset(st, -1, sizeof st);

    int N, M;
    cin >> N >> M;

    vector<int> cs(N);

    for (int i = 0; i < N; ++i)
        cin >> cs[i];

    cout << coin_change(M, cs) << '\n';

    return 0;
}
```

Implementação *bottom-up*

- O estado e as transições apresentadas anteriormente permite uma implementação *bottom-up* do algoritmo de programação dinâmica para o problema do troco
- O natural seria um laço externo, com o troco m variando de 0 a M, e um laço interno, avaliando as N moedas
- Embora a complexidade permaneça a mesma da implementação *top-down*, esta ordem não é favorável à *cache*, por conta dos diferentes saltos associados às moedas
- É possível melhorar a performance em tempo de execução invertendo os laços: para cada moeda, deve-se avaliar todos os trocos possíveis

Implementação *bottom-up* para o problema do troco

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX { 1010 }, oo { 1000000010 };
int st[MAX];

int coin_change(int M, const vector<int>& cs)
{
    for (int m = 1; m <= M; ++m)
        st[m] = oo;

    st[0] = 0;

   for (auto c : cs)
        for (int m = c; m <= M; ++m)
            st[m] = min(st[m], st[m - c] + 1);

    return st[M];
}

```

Implementação *bottom-up* para o problema do troco

```cpp
int main()
{
    int N, M;
    cin >> N >> M;

    vector<int> cs(N);

    for (int i = 0; i < N; ++i)
        cin >> cs[i];

    cout << coin_change(M, cs) << '\n';

    return 0;
}
```

Bases canônicas

Definição

- Embora não seja um correto, o algoritmo guloso produz o resultado correto para todos os trocos possíveis para certas bases de moedas C
- Seja $G(m)$ e $D(m)$ o mínimo de moedas para o troco m computados pelo algoritmo guloso e pelo algoritmo de programação dinâmica, respectivamente
- Uma base C é dita canônica se $G(m) = D(m)$ para todos os trocos inteiros não-negativos m
- Qualquer base $C = \{1, c_2\}$ é canônica
- Se C é canônica, o ganho de performance obtido em utilizar o algoritmo guloso é notável ($O(N) \times O(NM)$ do algoritmo de programação dinâmica)

Contraexemplos

- Considere uma base $C = \{1, c_2, c_3, \ldots, c_N\}$ não-canônica
- Um contraexemplo m é um inteiro positivo tal que $G(m) > D(m)$
- Xuan Cai apresenta vários resultados relativos à bases não-canônicas e contraexemplos em seu artigo "*Canonical Coin Systems for Change-Making Problems*", de 2009
- Dentre eles há um teorema que reduz os possíveis contraexemplos ao intervalo $(c_3 + 1, c_{N-1} + c_N)$
- Outro resultado provado no artigo é que $C = \{1, c_2, c_3\}$ é não-canônica se, somente se, $0 < r < c_2 - q$, onde $c_3 = qc_2 + r$, com $r \in [0, c_2 - 1]$

Verificação de canonicidade para $N \leq 5$

- Um teorema importante associa as bases com $N = 3$ com todas as demais: se $C = \{1, c_2, c_3\}$ é não-canônico, então a base $C = \{1, c_2, c_3, \ldots, c_N\}$, com $N \geq 4$, também será não-canônico
- É provado também que as bases $C = \{1, c_2, c_3, c_4\}$ são não-canônicas se elas satisfazem exatamente uma das condições abaixo:
    1. $C = \{1, c_2, c_3\}$ é não-canônica
    2. $G((k+1)c_3) > k + 1$, onde $kc_3 < c_4 < (k+1)c_3$
- Bases $C = \{1, c_2, c_3, c_4, c_5\}$ serão não-canônicas se, e somente se, satisfazem exatamente uma das condições abaixo:
    1. $C = \{1, c_2, c_3\}$ é não-canônica
    2. $C \neq \{1, 2, c_3, c_3 + 1, 2c_3\}$
    3. $G((k+1)c_4) > k + 1$, com $kc_4 < c_5 < (k+1)c_4$

Algoritmo $O(N^3)$ para verificação de canonicidade

- David Pearson, em seu artigo "*A Polynomial-time Algorithm for the Change-Making Problem*, de 1994, apresentou um algoritmo $O(N^3)$ para a identificação do menor contraexemplo, se existir
- O algoritmo elenca $O(N^2)$ possíveis candidatos à menor contraexemplo, por meio da observação de uma relação não-trivial entre a solução gulosa $G(c_i - 1)$ e o menor contraexemplo possível
- Cada candidato pode ser verificado em $O(N)$, de modo que a solução tem complexidade $O(N^3)$

Algoritmo $O(N^3)$ para verificação de canonicidade

- Seja $\mu = (m_1, m_2, \ldots, m_N)$ a solução ótima para o menor contraexemplo μ e $(x_1, x_2, \ldots, x_N)$ a solução gulosa para $(c_i - 1)$
- Então existe um j tal que $m_k = x_k$, se $k \in [1, j-1]$, $m_j = x_j + 1$ e $m_r = 0$, se $r > j$
- Assim, para cada moeda c_i, há N candidatos a j
- A partir da solução gulosa, se constrói a possível solução ótima do menor representante
- Daí se assume que $\mu = \sum_k m_k c_k$
- Se $G(c_i - 1) > \sum_k m_k$, então μ será um contraexemplo para a base não-canônica $C = \{c_1, c_2, \ldots, c_N\}$, com $c_1 > c_2 > \ldots > c_N = 1$

Implementação do algoritmo de verificação de canonicidade

```cpp
#include <bits/stdc++.h>

using namespace std;

const int oo { 2000000007 };

vector<int> greedy(int x, int N, const vector<int>& xs)
{
    vector<int> res(N, 0);

    for (int i = 0; i < N; ++i)
    {
        auto q = x / xs[i];
        x -=  q*xs[i];

        res[i] = q;
    }

    return res;
}

```

Implementação do algoritmo de verificação de canonicidade

```cpp
int value(const vector<int>& M, int N, const vector<int>& xs)
{
    int res = 0;

    for (int i = 0; i < N; ++i)
        res += M[i]*xs[i];

    return res;
}

int min_counterexample(int N, const vector<int>& xs)
{
    if (N <= 2)
        return -1;

    int ans = oo;

    for (int i = N - 2; i >= 0; --i)
    {
        auto g = greedy(xs[i] - 1, N, xs);
        vector<int> M(N, 0);
```

Implementação do algoritmo de verificação de canonicidade

```
        for (int j = 0; j < N; ++j)
        {
            M[j] = g[j] + 1;
            auto w = value(M, N, xs);
            auto G = greedy(w, N, xs);

            auto x = accumulate(M.begin(), M.end(), 0);
            auto y = accumulate(G.begin(), G.end(), 0);

            if (x < y)
                ans = min(ans, w);

            M[j]--;
        }
    }

    return ans == oo ? -1 : ans;
}
```

Referências

1. **CAI**, Xuna. *Canonical Coin Systems for Change-Making Problems*, 2009.
2. **CORMEN**, Thomas H.; **LEISERSON**, Charles E.; **RIVEST**, Ronald; **STEIN**, Clifford. *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
4. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
5. **PEARSON**, David. *A Polynomial-time Algorithm for the Change-Making Problem*, 1994.