Paradigmas de Resolução de Problemas

Busca Completa – *Backtracking*

Prof. Edson Alves – UnB/FGA

Sumário

1. *Backtracking*

2. Exemplos de aplicação do *backtracking*

3. Poda

Backtracking

Definição

- *Backtracking* é uma técnica de busca completa que, por meio de recursão, investiga todo o espaço de soluções
- Ela parte de uma solução, inicialmente vazia, e tenta construir uma nova solução estendendo a solução parcial um elemento por vez
- A cada passo são avaliados todos os possíveis elementos que podem integrar uma nova solução, a depender dos elementos já inseridos na solução parcial
- Quando uma solução é encontrada, ela é processada
- Em seguida, ou o algoritmo para ou ele segue para identificar todas as demais soluções, se existirem

Pseudo-código do *backtracking*

Algoritmo 1 *Backtracking*

Input: um vetor solução xs e os parâmetros P do problema

Output: O processamento de todas as soluções possíveis

```
 1: xs ← ∅
 2:
 3: function BACKTRACKING(xs, P)
 4:     if IS_SOLUTION(xs, P) then
 5:         PROCESS_SOLUTION(xs, P)
 6:     else
 7:         cs ← CANDIDADES(xs, P)
 8:         for all c ∈ cs do
 9:             PUSH_BACK(c, xs)
10:             UPDATE(P, c)
11:             BACKTRACKING(xs, P)
12:             RESTORE(P, c)
13:             POP_BACK(xs)
```

Observações sobre o *backtracking*

- No pseudocódigo apresentado, para cada algoritmo em particular é necessário implementar as funções `is_solution()`, `process_solution()` e `candidates()`
- O vetor xs pode ser tanto um `vector` do C++ ou um *array* estático de C
- A construção dos possíveis candidatos depende do estado do vetor xs e dos parâmetros P do problema
- Cada candidato deve ser inserido ao final de xs, e após o retorno da chamada recursiva, ele deve ser removido
- Assim, não é necessário fazer novas cópias de xs a cada chamada: basta passar este vetor por referência
- A inserção de um candidato em xs pode modificar os parâmetros P: esta atualização também deve ser desfeita após a chamada recursiva

Exemplos de aplicação do *backtracking*

Subconjuntos de um conjunto

- O *backtracking* pode ser utilizado para listar todos os subconjuntos de um conjunto dado
- Os parâmetros do problema são: o vetor as com os elementos do conjunto, o número de elementos N de as e o índice i do elemento de as que será considerado
- Cada subconjunto será caracterizado após todos os elementos terem sido considerados
- A cada passo, o único candidato a entrar ou não no subconjunto é o elemento a_i
- Por conta da possibilidade de a_i estar ou não no subconjunto, serão necessárias duas chamadas recursivas

Listagem de subconjuntos usando *backtracking*

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

using namespace std;

bool is_solution(int i, int N) { return i == N; }

void process_solution(const vector<int>& xs)
{
    cout << "{ ";

    for (auto x : xs)
        cout << x << ' ';

    cout << "}\n";
}

vector<int> candidates(int i, const vector<int>& as)
{
    return { as[i] };
}
```

Listagem de subconjuntos usando *backtracking*

```
22 void backtracking(vector<int>& xs, int i, int N, const vector<int>& as)
23 {
24     if (is_solution(i, N))
25         process_solution(xs);
26     else
27     {
28         auto cs = candidates(i, as);
29 
30         for (auto c : cs)
31         {
32             // Segue sem escolher c
33             backtracking(xs, i + 1, N, as);
34 
35             // Segue escolhendo c
36             xs.push_back(c);
37             backtracking(xs, i + 1, N, as);
38             xs.pop_back();
39         }
40     }
41 }
```

Listagem de subconjuntos usando *backtracking*

```
43 int main()
44 {
45     vector<int> as { 2, 3, 5, 7, 11 }, xs;
46
47     backtracking(xs, 0, (int) as.size(), as);
48
49     return 0;
50 }
```

Permutações de um conjunto de elementos

- As $N!$ permutações de um conjunto de N elementos também podem ser listadas utilizando-se o *backtracking*
- Os parâmetros do problema são: um vetor as representando os elementos do conjunto e o valor de N
- Neste caso, cada solução é composta por todos os elementos de as, em alguma ordem
- Os candidatos a serem inseridos em uma solução parcial são todos os elementos de as que ainda não estão presentes em xs
- Para simplificar o algoritmo, xs armazenará os índices, e não os elementos de as

Listagem de permutações usando *backtracking*

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

using namespace std;

const int MAX { 20 };

bool is_solution(const vector<int>& xs, const vector<int>& as)
{
    return xs.size() == as.size();
}

void process_solution(const vector<int>& xs, const vector<int>& as)
{
    cout << "(";

    for (size_t i = 0; i < xs.size(); ++i)
        cout << as[xs[i]] << (i + 1 == xs.size() ? ")\n" : ", ");
}
```

Listagem de permutações usando *backtracking*

```
20 vector<int> candidates(const vector<int>& xs, const vector<int>& as)
21 {
22     bitset<MAX> used;
23     vector<int> cs;
24
25     for (auto x : xs)
26         used[x] = true;
27
28     for (size_t i = 0; i < as.size(); ++i)
29         if (not used[i])
30             cs.emplace_back((int) i);
31
32     return cs;
33 }
```

Listagem de permutações usando *backtracking*

```
35 void backtracking(vector<int>& xs, const vector<int>& as)
36 {
37     if (is_solution(xs, as))
38         process_solution(xs, as);
39     else
40     {
41         auto cs = candidates(xs, as);
42 
43         for (auto c : cs)
44         {
45             xs.emplace_back(c);
46             backtracking(xs, as);
47             xs.pop_back();
48         }
49     }
50 }
```

Listagem de permutações usando *backtracking*

```
52 int main()
53 {
54     vector<int> as { 1, 2, 3, 4 }, xs;
55
56     backtracking(xs, as);
57
58     return 0;
59 }
```

Combinações

- A listagem de todas as combinações de N elementos de um conjunto, tomados M a M, também pode ser criada usando a técnica do *backtracking*
- O procedimento é semelhante ao da listagem de permutações
- A primeira diferença é que o problema tem um parâmetro a mais: o valor de M
- A segunda diferença reside na construção dos candidatos: devem ser considerados apenas os elementos de as que não estão presentes em xs e que são estritamente maiores do que todos os elementos já listados

Listagem de combinações usando *backtracking*

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

using namespace std;

bool is_solution(const vector<int>& xs, size_t M)
{
    return xs.size() == M;
}

void process_solution(const vector<int>& xs, const vector<int>& as)
{
    cout << "(";

    for (size_t i = 0; i < xs.size(); ++i)
        cout << as[xs[i]] << (i + 1 == xs.size() ? ")\n" : ", ");
}
```

Listagem de combinações usando *backtracking*

```
18 vector<int> candidates(const vector<int>& xs, const vector<int>& as)
19 {
20     vector<int> cs;
21
22     if (xs.empty())
23     {
24         cs.resize(as.size());
25         iota(cs.begin(), cs.end(), 0);
26     } else
27         for (size_t i = xs.back() + 1; i < as.size(); ++i)
28             cs.emplace_back(i);
29
30     return cs;
31 }
```

Listagem de combinações usando *backtracking*

```
33 void backtracking(vector<int>& xs, size_t M, const vector<int>& as)
34 {
35     if (is_solution(xs, M))
36         process_solution(xs, as);
37     else
38     {
39         auto cs = candidates(xs, as);
40 
41         for (auto c : cs)
42         {
43             xs.emplace_back(c);
44             backtracking(xs, M, as);
45             xs.pop_back();
46         }
47     }
48 }
```

Poda

Poda

- A poda (*prunning*) é uma estratégia de otimização dos algoritmos que utilizam o *backtracking*
- A poda avalia a solução parcial xs e os candidatos disponíveis cs
- Se os candidatos ainda disponíveis não são mais capazes de tornar xs uma solução válida, então o *backtracking* deve retornar imediatamente, sem realizar as chamadas recursivas
- A poda não necessariamente reduz a complexidade assintótica do algoritmo, mas impacta no valor da constante
- O ganho obtido em não processar ramos da árvore de decisão que não levam a soluções pode resultar em AC onde um *backtracking* sem poda resulta em TLE

Listagem de combinações com poda

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

using namespace std;

int chamadas, soluções;

bool is_solution(const vector<int>& xs, size_t M)
{
    return xs.size() == M;
}

void process_solution(const vector<int>&, const vector<int>&)
{
    ++soluções;
}

vector<int> candidates(const vector<int>& xs, const vector<int>& as)
{
    vector<int> cs;
```

Listagem de combinações com poda

```
21      if (xs.empty())
22      {
23          cs.resize(as.size());
24          iota(cs.begin(), cs.end(), 0);
25      } else
26          for (size_t i = xs.back() + 1; i < as.size(); ++i)
27              cs.push_back((int) i);
28
29      return cs;
30  }
31
32  void backtracking(vector<int>& xs, size_t M, const vector<int>& as, bool prunning = false)
33  {
34      ++chamadas;
35
36      if (is_solution(xs, M))
37          process_solution(xs, as);
38      else
39      {
40          auto cs = candidates(xs, as);
```

Listagem de combinações com poda

```
42          // Não há elementos o suficiente para construir uma solução
43          if (prunning and xs.size() + cs.size() < M)
44              return;
45
46          for (auto c : cs)
47          {
48              xs.push_back(c);
49              backtracking(xs, M, as, prunning);
50              xs.pop_back();
51          }
52      }
53  }
54
55  int main()
56  {
57      vector<int> as { 1, 2, 5, 10, 25, 50, 100, 200, 500, 10000 }, xs;
58
59      chamadas = soluções = 0;
60      backtracking(xs, 8, as, false);
61
```

Listagem de combinações com poda

```
62      cout << "Sem poda -- chamadas: " << chamadas << ", soluções: " << soluções << '\n';
63
64      chamadas = soluções = 0;
65      xs.clear();
66      backtracking(xs, 8, as, true);
67
68      cout << "Com poda -- chamadas: " << chamadas << ", soluções: " << soluções << '\n';
69
70      // Saída:
71      // Sem poda -- chamadas: 1013, soluções: 45
72      // Com poda -- chamadas: 375, soluções: 45
73
74      return 0;
75  }
```

Referências

1. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
2. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.