Branch and Bound e Backtracking

Lucas Saliba   Lucas Santiago de Oliveira

Novembro de 2021

Sumário

Resumo

Para começar falando sobre os algoritmos de otimização de problemas, começarei falando sobre o que é um problema linear na programação. Se considerarmos que exista um problema X que leva Y de tempo para ser resolvido e que Y cresce diretamente proporcinal ao crescimento de X, então temos um problema linear. A ideia de todos os algoritmos que serão citados nesse documento é fazer Y crescer mais lentamente. Dessa forma, se o problema X dobrar de tamanho Y não dobrará de tamanho junto.

1 *Branch and Bound*

1.1 Conceito

Branch and Bound ou ramificar e limitar é um estilo de algoritmo que separa o problema em problemas menores e tenta resolver esse subproblema da forma mais otimizada que conseguir. Caso não seja encontrado um algoritmo eficiente para resolver o problema, esse subconjunto (*disjoint set*) não será resolvido.

Ele é um paradigma de projeto de algoritmo geralmente usado para resolver problemas de otimização combinatória. Esses problemas são tipicamente exponenciais em termos de complexidade de tempo e podem exigir a exploração de todas as permutações possíveis no pior caso. *Branch and Bound* pode resolver esses problemas com relativa rapidez.

1.2 Quando usar

Ótimo uso para esse estilo de programação é em problemas de NP-Difícil [1] como:

- Problema de satisfatibilidade máxima
- Problema do caixeiro-viajante
- Filogenética computacional

Os três problemas tem o mesmo grande problema de não existir um algoritmo eficiente que o resolva diretamente. Dessa forma, *Branch and Bound* separaria esses problemas em problemas menores e tentaria resolver cada subconjunto de forma independente.

1.3 Exemplo

Vamos usar o problema da mochila abaixo para entender o *Branch and Bound*. "Dados duas matrizes de inteiros val[0..n-1] e wt[0..n-1] que representam valores e pesos associados a n itens, respectivamente. Descubra o subconjunto de valor máximo de val[] de forma que a soma dos pesos deste subconjunto seja menor ou igual à capacidade W da mochila."

Existem diferentes abordagens para resolver o problema acima, porém a solução *Branch and Bound* é o método mais adequado quando os pesos dos itens não são inteiros.

Como encontrar o limite para cada nó da mochila? A ideia é usar o fato de que a abordagem *Greedy* fornece a melhor solução para o problema da mochila fracionária. Para verificar se um determinado nó pode nos dar uma solução melhor ou não, calculamos a solução ótima (por meio do nó) usando a abordagem *Greedy*. Se a solução computada pela abordagem *Greedy* em si é mais do que a melhor até agora, então não podemos obter uma solução melhor por meio do nó.

Passo a passo do algoritmo:

- Classificar todos os itens em ordem decrescente de razão de valor por unidade de peso para que um limite superior possa ser calculado usando o *Greedy Approach*.
- Inicializar o lucro máximo, maxProfit = 0.
- Criar uma fila vazia, Q.

- Criar um nó fictício da árvore de decisão e enfileire-o em Q. O lucro e o peso do nó fictício são 0.
- Fazer o seguinte enquanto Q não estiver vazio:
    I. Extrair um item de Q. Deixar o item extraído ser u.
    II. Calcular o lucro do nó do próximo nível. Se o lucro for maior que maxProfit, atualizar o maxProfit.
    III. Limite de cálculo do nó do próximo nível. Se o limite for maior do que maxProfit, adicione o nó do próximo nível a Q.
- Considerar o caso em que o nó do próximo nível não é considerado como parte da solução e adicione um nó à fila com o nível seguinte, mas peso e lucro sem considerar os nós do próximo nível.

A seguir está a implementação em C++ da ideia acima:

```
#include <bits/stdc++.h>
using namespace std;

// Estrutura para o item que armazena o peso e correspondente

// Valor do item
struct Item
{
  float peso;
  int valor;
};

// Estrutura do nó para armazenar informa cões de decisão

// Arvore
struct Node
{
  // Nível do nó na árvore de decisão
  // Lucro dos nós no caminho da raiz para este nó (incluindo este
    nó)
```

```cpp
  // Limite superior do lucro máximo na subárvore
  int nivel, lucro, limite;
  float peso;
};

bool cmp(Item a, Item b)
{
  double r1 = (double)a.valor / a.peso;
  double r2 = (double)b.valor / b.peso;
  return r1 > r2;
}

// Retorna o limite do lucro na subárvore enraizada em u.
// Esta funcão usa principalmente a solucão Greedy para encontrar um limite superior no lucro máximo.

int limite(Node u, int n, int W, Item arr[])
{
  // se o peso superar a capacidade da mochila, devolva 0 como limite esperado
  if (u.peso >= W)
    return 0;

  // inicializar vinculado ao lucro pelo lucro atual
  int limite_lucro = u.lucro;

  // comece a incluir itens do índice 1 mais para o atual
  int j = u.nivel + 1;
  int pesoTot = u.peso;

  // verificar a condicão do índice e a capacidade da mochila
  while ((j < n) && (pesoTot + arr[j].peso <= W))
  {
    pesoTot += arr[j].peso;
    limite_lucro += arr[j].valor;
    j++;
  }

```

```
  //Se k não for n, inclua o último item parcialmente para o limite
     superior do lucro
  if (j < n)
    limite_lucro += (W - pesoTot) * arr[j].valor /
                    arr[j].peso;

  return limite_lucro;
}

// Retorna o lucro máximo que podemos obter com a capacidade W
int knapsack(int W, Item arr[], int n)
{
  // classificando item com base no valor por unidade
  sort(arr, arr + n, cmp);

  // faca uma fila para atravessar o nó
  queue<Node> Q;
  Node u, v;

  u.nivel = -1;
  u.lucro = u.peso = 0;
  Q.push(u);

  //Extraia um por um item da árvore de decisão
  // calcula o lucro de todos os filhos do item extraído e continue
     salvando maxProfit
  int maxProfit = 0;
  while (!Q.empty())
  {
    // Desenfileirar um nó
    u = Q.front();
    Q.pop();

    // Se for um nó inicial, atribua o nível 0
    if (u.nivel == -1)
      v.nivel = 0;

    // Se não houver nada no próximo nível
```

```
    if (u.nivel == n - 1)
      continue;

    // Caso contrário, se não o último nó, então incremente o nível, calcular o lucro dos nós filhos.
    v.nivel = u.nivel + 1;

    // Tomando o item do nível atual adicionar atual peso e valor do nível para o nó u peso e valor
    v.peso = u.peso + arr[v.nivel].peso;
    v.lucro = u.lucro + arr[v.nivel].valor;

    // Se o peso acumulado for menor que W e o lucro é maior do que o lucro anterior, atualizar maxprofit
    if (v.peso <= W && v.lucro > maxProfit)
      maxProfit = v.lucro;

    // Obtenha o limite superior do lucro para decidir se deve adicionar v a Q ou não.
    v.limite = limite(v, n, W, arr);

    // Se o valor vinculado for maior do que o lucro, então apenas empurre na fila para mais consideracão
    if (v.limite > maxProfit)
      Q.push(v);

    //Faca a mesma coisa, mas sem tirar o item na mochila
    v.peso = u.peso;
    v.lucro = u.lucro;
    v.limite = limite(v, n, W, arr);
    if (v.limite > maxProfit)
      Q.push(v);
  }

  return maxProfit;
}

// Programa para testar a funcão acima
```

```
int main()
{
  // Peso da mochila
  int W = 10;
  Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100}, {5, 95}, {3,
    30}};
  int n = sizeof(arr) / sizeof(arr[0]);

  cout << "Lucro máximo possível = "
       << knapsack(W, arr, n);

  return 0;
}
```

2 *Backtracking*

2.1 Conceito

2.2 Quando usar

2.3 Exemplo

Explicacao do problema Ciclo Hamiltoniano

O caminho hamiltoniano em um gráfico não direcionado é um caminho que visita cada vértice exatamente uma vez. Um ciclo hamiltoniano é um caminho hamiltoniano tal que existe uma aresta (no gráfico) do último vértice ao primeiro vértice do caminho hamiltoniano. Determine se um determinado gráfico contém Ciclo Hamiltoniano ou não. Se contiver, imprime o caminho. A seguir estão as entradas e saídas da função necessária.

Entrada do problema

Um gráfico de matriz 2D [V] [V] onde V é o número de vértices no gráfico e o gráfico [V] [V] é a representação da matriz de adjacência do gráfico. Um gráfico de valor [i] [j] é 1 se houver uma borda direta de i para j, caso contrário, gráfico [i] [j] é 0.

Saida do problema

Um caminho de matriz [V] que deve conter o Caminho Hamiltoniano. o caminho [i] deve representar o iº vértice no caminho hamiltoniano. O código também deve retornar falso se não houver um ciclo hamiltoniano no gráfico.

Algoritmo Backtracking

Cria uma matriz de caminho vazia e adiciona o vértice 0 a ela. Adiciona outros vértices, começando do vértice 1. Antes de adicionar um vértice, verifica se ele é adjacente ao vértice adicionado anteriormente e se já não foi adicionado. Se encontrarmos tal vértice, adicionamos o vértice como parte da solução. Se não encontrarmos um vértice, retornamos falso.

Exemplo de algoritmo

```python

# Algoritmo em python para resolver o problema do ciclo hamiltoniano com Backtracking

class Grafo():
    def __init__(self, vertices):
        self.grafo = [[0 for column in range(vertices)]
                            for row in range(vertices)]
        self.V = vertices

# Verifica se este vértice é um vértice adjacente do vértice adicionado anteriormente e não é incluído no caminho anterior

    def isSafe(self, v, pos, caminho):

        # Verifica se o vértice atual e o último vértice no caminho são adjacentes
        if self.grafo[ caminho[pos-1] ][v] == 0:
            return False

        # Verifica se o vértice atual ainda não está no caminho
```

```
        for vertex in caminho:
            if vertex == v:
                return False

        return True

    # Funcão recursiva para resolver o problema do ciclo hamiltoniano
    def hamCycleUtil(self, caminho, pos):

        # Caso base: se todos os vértices forem incluídos no caminho
        if pos == self.V:

            # O último vértice deve ser adjacente ao primeiro vértice no caminho para fazer um ciclo
            if self.grafo[ caminho[pos-1] ][ caminho[0] ] == 1:
                return True
            else:
                return False

        # Tenta vértices diferentes como um próximo candidato no ciclo hamiltoniano.
        # Não tentamos 0 como incluímos 0 como ponto de partida em hamCycle ()
        for v in range(1,self.V):

            if self.isSafe(v, pos, caminho) == True:

                caminho[pos] = v

                if self.hamCycleUtil(caminho, pos+1) == True:
                    return True

                #Remove o vértice atual se não leva a uma solucão
                caminho[pos] = -1

        return False
```

```

    def hamCycle(self):
        caminho = [-1] * self.V

   # Vamos colocar o vértice 0 como o primeiro vértice no caminho. Se houver um ciclo hamiltoniano então o caminho pode ser iniciado de qualquer ponto do ciclo, pois o gráfico não é direcionado
        caminho[0] = 0

        if self.hamCycleUtil(caminho,1) == False:
            print ("A solucao nao existe \n")
            return False

        self.printSolution(caminho)
        return True

    def printSolution(self, caminho):
        print ("A Solucão Existe: Seguindo ",
               "é um ciclo hamiltoniano")
        for vertex in caminho:
            print (vertex, end = " ")
        print (caminho[0], "\n")

# Codigo do condutor

# Criacao do grafo 1 de teste
g1 = Grafo(5)
g1.grafo = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
            [0, 1, 0, 0, 1,],[1, 1, 0, 0, 1],
            [0, 1, 1, 1, 0], ]

# Printa solucao
g1.hamCycle();

# Criacao do grafo 2 de teste
g2 = Grafo(5)
g2.grafo = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
```

```
        [0, 1, 0, 0, 1,], [1, 1, 0, 0, 0],
        [0, 1, 1, 0, 0], ]

# Printa solucao
g2.hamCycle();
```

3 Relação entre *Branch and Bound* e *Backtracking*

4 Relação com a Abordagem Gulosa

4.1 Conceito

As abordagens gulosas são baseadas em tentar resolver um problema sem se importar com otimizações para que o resultado seja obtido rapidamente. O primeiro passo para conseguir uma solução eficiente para um problema é conseguir apenas uma solução para o problema, mesmo que não consiga rapidamente. Discussão sobre os tipos de formas de implementar o algoritmo de Fibonacci [2].

4.2 Exemplo de código

```python
# Código de Fibonacci usando abordagem gulosa.
# Muitos elementos dentro da árvore de recursão
# serão recalculados inúmeras vezes
import time

# Código do fibonacci
def fib(numero):
    if (numero <= 2):
        return 1

    return fib(numero-1) + fib(numero-2)


def timer(fib, num):
    start = time.time()
```

```
    fib(num)
    end = time.time()
    return end - start

# Escrevendo os tempos na tela
print(f'Fibonnaci de 10: {timer(fib, 10)}')
print(f'Fibonnaci de 15: {timer(fib, 15)}')
print(f'Fibonnaci de 20: {timer(fib, 20)}')
print(f'Fibonnaci de 30: {timer(fib, 30)}')
```

4.3 Complexidade

Devido ao fato desse algoritmo criar uma árvore e deslocar dentro dela vários nós são iguais e mesmo assim recalculados. Com isso o custo é $\mathcal{O}(n^2)$.

5 Relação com Programação Dinâmica

5.1 Conceito

A ideia inicial da progamação dinâmica está baseada em armazenar todas as operações que serão repetidas durante uma recursão. Dessa forma, se durante a recursão tiver um nó igual que já foi previamente calculado, não será necessário que o cálculo seja feito novamente. Em árvores criadas como o algoritmo de fibonacci em que muitos nós são iguais, todos eles serão calculados apenas uma vez [3].

5.2 Exemplo de código

```
# Código baseado na implementacão da free code camp
# Referência: https://youtu.be/oBt53YbR9Kk?t=210
import time

# Código do fibonacci
def fib(numero):
    if (numero <= 2):
        return 1
```

```python

    return fib(numero-1) + fib(numero-2)

# Versão do código usando programacão dinâmica com memoization
def fibMemo(numero, memo={}):
    if (numero in memo):
        return memo[numero]

    if (numero <= 2):
        return 1

    memo[numero] = fibMemo(numero - 1, memo) + fibMemo(numero - 2, memo)
    return memo[numero]

def timer(fib, num):
    start = time.time()
    fib(num)
    end = time.time()
    return end - start

# Escrevendo os tempos na tela
print(f'Fibonnaci de 10: {timer(fib, 10)}')
print(f'Fibonnaci de 10 memoization: {timer(fibMemo, 10)}')

print(f'Fibonnaci de 15: {timer(fib, 15)}')
print(f'Fibonnaci de 15 memoization: {timer(fibMemo, 15)}')

print(f'Fibonnaci de 20: {timer(fib, 20)}')
print(f'Fibonnaci de 20 memoization: {timer(fibMemo, 20)}')

print(f'Fibonnaci de 30: {timer(fib, 30)}')
print(f'Fibonnaci de 500 memoization: {timer(fibMemo, 500)}')
```

5.3 Complexidade

Como o algoritmo armazena todas as equações já feitas, todas as operações são transformar de $\mathcal{O}(n^2)$ para $\mathcal{O}(n)$. Precisando apenas do tempo de encontrar o elemento dentro do dicionário do *Python* e retornando ele [4].

6 Relação com Divisão e Conquista

Referências

[1] Wikipedia, NP-Hardness, acessado em 27/11/2021.

[2] Stack Overflow, Discussion about Fibonacci's *Greedy algorithm*, acessado em 28/11/2021.

[3] Geeks For Geeks, Dynamic Programing, acessado em 28/11/2021.

[4] FreeCodeCamp, Dynamic Programing, acessado em 27/11/2021.