

Técnicas e Análise de Algoritmos: *backtracking* e *branch-and-bound*

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados II (AE23CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

- *Backtracking*
 - Labirinto
 - Mochila
- *Branch-and-Bound*
 - Labirinto
 - Mochila

Introdução

- Algoritmos força bruta testam exaustivamente todas as soluções possíveis de um problema para obter uma solução ótima
 - Não utiliza critérios para eliminar outras soluções que não poderão ser melhores que a obtida no estágio considerado
 - As soluções podem ser enumeradas de modo semelhante ao percurso em uma árvore que possua todas as soluções
 - Muitas vezes, o conjunto de soluções cresce exponencialmente!



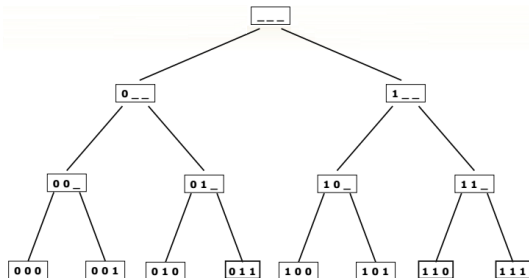
- Algoritmos força bruta não seguem regra fixa de computação
- Um algoritmo força bruta gera todos os possíveis candidatos da solução e verifica qual deles satisfaz o problema
 - Geralmente possui uma implementação simples
 - Encontra soluções, caso exista
 - Comumente utilizado em problemas com tamanho limitado ou quando um algoritmo mais eficiente é desconhecido
 - Muito útil quando a simplicidade da solução é mais importante do que a velocidade de execução

- *Backtracking*
- *Branch-and-bound*

Backtracking

- Método que gera apenas soluções válidas
- Aplicado em problemas cuja solução pode ser definida a partir de uma sequência de passos
- Quando um conflito é detectado, o algoritmo dará um passo para trás (*backtrack*) e tentará um "outro caminho" para encontrar a solução
- Comumente mais rápido que algoritmos força bruta, caso aplicável
 - Soluções inválidas são eliminadas sem serem completamente analisadas

- Muitos problemas podem ser modelados por uma árvore que representa todas as sequências de decisões possíveis
 - Exemplo: encontrar todos os números binários de 3 bits em que a soma de 1's seja maior ou igual a 2
 - Espaço de busca para o exemplo:



- Como reduzir o espaço de busca?

- Alguns problemas famosos em que o *backtracking* é utilizado:
 - Passeio do cavalo
 - Oito rainhas
 - Labirinto
 - Mochila

- Dada uma matriz $n \times m$ que representa um labirinto
- Dadas as posições inicial $p_i = (x_i, y_i)$ e final $p_f = (x_f, y_f)$
- O objetivo é verificar se existe um caminho entre p_i e p_f
- A matriz que representa o labirinto pode conter um dos seguintes valores:
 - -2: se a posição (x, y) representa uma parede
 - -1: se a posição (x, y) não faz parte do caminho
 - i : se a posição (x, y) , tal que $i \geq 0$, faz parte do caminho

- Exemplo de labirinto 8×8 , onde I é a posição final e F é a posição final

-2	-2	-2	-2	-2	-2	-2	-2
-2	I	-1	-1	-1	-1	-1	-2
-2	-2	-1	-2	-1	-1	-1	-2
-2	-1	-1	-2	-2	-2	-1	-2
-2	-1	-2	-2	-1	-1	-1	-2
-2	-1	-2	-1	-1	-1	-2	-2
-2	-1	-1	-1	-2	-1	-1	-2
-2	-2	-2	-2	-2	-2	F	-2

- Em uma posição no labirinto, apenas os seguintes movimentos podem ser executados
 - Direita
 - Esquerda
 - Cima
 - Baixo

- Exemplo de caminho encontrado na matriz

-2	-2	-2	-2	-2	-2	-2	-2
-2	0	1	-1	-1	-1	-1	-2
-2	-2	2	-2	-1	-1	-1	-2
-2	4	3	-2	-2	-2	-1	-2
-2	5	-2	-2	-1	-1	-1	-2
-2	6	-2	10	11	12	-2	-2
-2	7	8	9	-2	13	14	-2
-2	-2	-2	-2	-2	-2	15	-2

- Outro exemplo de caminho encontrado na matriz

-2	-2	-2	-2	-2	-2	-2	-2
-2	0	1	2	3	4	5	-2
-2	-2	-1	-2	-1	-1	6	-2
-2	-1	-1	-2	-2	-2	7	-2
-2	-1	-2	-2	-1	9	8	-2
-2	-1	-2	-1	-1	10	-2	-2
-2	-1	-1	-1	-2	11	12	-2
-2	-2	-2	-2	-2	-2	13	-2

- Para este labirinto, existem várias soluções

Backtracking

Labirinto

```
#define MAX 10

int main(){
    int lab[MAX][MAX], n, m, li, ci, lf, cf, resp;

    /*Movimentos válidos no labirinto*/
    int movX[] = {0, 1, 0, -1};
    int movY[] = {1, 0, -1, 0};

    /*inicializar as variáveis relacionadas ao labirinto*/
    iniciar_labirinto(lab, m, n, li, ci, lf, cf);

    lab[li][ci] = 0; // posição inicial no labirinto

    resp = resolver_labirinto(lab, m, n, movX, movY, li, ci, lf, cf);

    if (resp > 0)
        imprimir_labirinto(lab, m, n);
    else
        printf("Solucao nao encontrada!\n");
    return 0;
}
```

Backtracking

Labirinto

```
int resolver_labirinto(int lab[MAX][MAX], int m, int n, movX[], movY[], int li,
    int ci, int lf, int cf){
    int l, c, i, passos = 0;

    if ((li == lf) && (ci == cf)) return lab[li][ci];

    /*Todos os movimentos a partir da posição inicial são testados*/
    for (i = 0; i < 4; i++){
        l = li + movX[i];
        c = ci + movY[i];

        /*O movimento é verificado e caso seja válido, uma solução é gerada*/
        if ((l >= 0) && (l < m) && (c >= 0) && (c < n) && (lab[l][c] == -1)){
            lab[l][c] = lab[li][ci] + 1;

            passos = resolver_labirinto(lab, m, n, movX, movY, l, c, lf, cf);
        }
    }

    return passos;
}
```


- Muitas vezes não é desejável encontrar apenas uma solução qualquer
 - Pode ser desejável encontrar uma solução ótima, de acordo com algum critério de otimalidade
- No problema do labirinto, por exemplo, podemos estar interessado em encontrar um caminho que contém o menor número de passos

- Possível adaptação no algoritmo

```
void resolver_labirinto(int lab[MAX][MAX], int m, int n, movX[], movY[],
    int li, int ci, int lf, int cf, int *min){
    int l, c, i;

    if ((li == lf) && (ci == cf)){
        if (lab[lf][cf] < *min)
            *min = lab[lf][cf];
    }else{
        /*Todos os movimentos a partir da posição inicial são testados*/
        for (i = 0; i < 4; i++){
            l = li + movX[i];
            c = ci + movY[i];

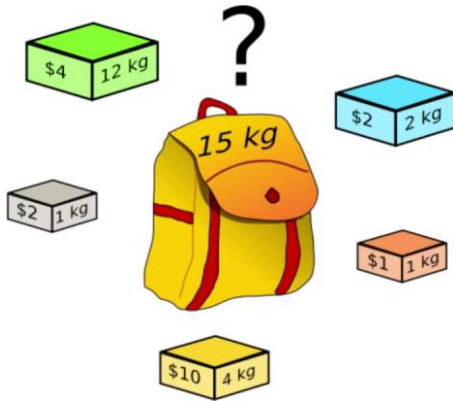
            /*O movimento é verificado e caso seja válido, uma solução é gerada*/
            if ((l >= 0) && (l < m) && (c >= 0) && (c < n) && (lab[l][c] == -1) ||
                (lab[l][c] > lab[li][ci] + 1)){
                lab[l][c] = lab[li][ci] + 1;

                resolver_labirinto(lab, m, n, movX, movY, l, c, lf, cf, min);
            }
        }
    }
}
```

- Apesar do algoritmo encontrar a solução ótima, ainda todas as outras soluções são testadas

Backtracking

Mochila



- Lembra do algoritmo para a solução do problema da mochila por força bruta?

```
static int mochila_fb(int c[], int p[], int n, int b, int
i, int max){
    int c1, c2;
    if (i >= n)
        return b < 0 ? 0 : max;
    else{
        c1 = mochila_fb(c, p, n, b, i + 1, max);
        c2 = mochila_fb(c, p, n, b - p[i], i + 1, max + c[i]);
        return c1 > c2 ? c1 : c2;
    }
}

int mochila(int c[], int p[], int n, int b){
    return mochila_fb(c, p, n, b, 0, 0);
}
```

- Na implementação apresentada no slide anterior, podemos ver que, além do custo computacional alto, também são geradas soluções inválidas
- Algumas adaptações no código evitaria a geração de soluções inválidas
 - Em vez da primeira verificação ser se i é maior ou igual a n , verificar se $b < 0$
 - Caso a nova verificação resultar em "verdadeiro", retorne 0
 - A próxima verificação é se $i < n$: caso a resposta seja positiva, são feitas chamadas recursivas como realizadas na solução por força-bruta
 - Se as verificações acima falharem, basta retornar max (valor máximo acumulado)

- Solução do problema da mochila por *backtracking*

```
static int mochila_bkt(int c[], int p[], int n, int b, int
i, int max){
    int c1, c2;
    if (b < 0)
        return 0;
    else if (i < n){
        c1 = mochila_bkt(c, p, n, b, i + 1, max);
        c2 = mochila_bkt(c, p, n, b - p[i], i + 1, max +
c[i]);
        return c1 > c2 ? c1 : c2;
    }else
        return max;
}

int mochila(int c[], int p[], int n, int b){
    return mochila_bkt(c, p, n, b, 0, 0);
}
```

Branch-and-Bound

- Relacionada com *backtracking*
- *Branch-and-bound* é um método de exploração mais sofisticada
 - *Branch*: explora opções
 - *Bound*: limite quantitativo (tem o propósito de evitar buscas em espaços menos promissores)

- *Branch-and-bound* realiza enumeração sistemática dos candidatos à solução
- Candidatas (a solução) parciais são eliminadas quando ocorre uma das seguintes situações:
 - Incapacidade de gerar uma solução válida (similar ao *backtracking*)
 - Incapacidade de gerar uma solução ótima, considerando
 - O valor da melhor solução encontrada até então (limitante superior)
 - O custo ainda necessário para gerar uma solução a partir da candidata atual (limitante inferior)
- O desempenho de uma solução *branch-and-bound* está fortemente relacionado à qualidade dos seus limitantes inferiores e superiores

- Solução *backtracking* otimizada do problema

```
void resolver_labirinto(int lab[MAX][MAX], int m, int n, movX[], movY[],
    int li, int ci, int lf, int cf, int *min){
    int l, c, i;

    if ((li == lf) && (ci == cf)){
        if (lab[lf][cf] < *min)
            *min = lab[lf][cf];
    }else{
        /*Todos os movimentos a partir da posição inicial são testados*/
        for (i = 0; i < 4; i++){
            l = li + movX[i];
            c = ci + movY[i];

            /*O movimento é verificado e caso seja válido, uma solução é gerada*/
            if ((l >= 0) && (l < m) && (c >= 0) && (c < n) && ((lab[l][c] == -1) ||
                (lab[l][c] > lab[li][ci] + 1))){
                lab[l][c] = lab[li][ci] + 1;

                resolver_labirinto(lab, m, n, movX, movY, l, c, lf, cf, min);
            }
        }
    }
}
```

- O código apresentado anteriormente pode ser alterado para que o caminho ótimo seja encontrado utilizando *branch-and-bound*
- Se um caminho parcial utilizou tantos passos quanto o melhor caminho já encontrado, o mesmo pode ser descartado
 - Caso o número de passos do caminho parcial mais a diferença entre a posição atual e a saída for maior ou igual ao número de passos do melhor caminho já encontrado, então esse caminho parcial também pode ser descartado

Branch-and-Bound

Labirinto

```
void resolver_labirinto(int lab[MAX][MAX], int m, int n, movX[], movY[],
    int li, int ci, int lf, int cf, int *min){
    int l, c, i;

    if ((li == lf) && (ci == cf)){
        if (lab[lf][cf] < *min)
            *min = lab[lf][cf];
    }else{
        /*Todos os movimentos a partir da posição inicial são testados*/
        for (i = 0; i < 4; i++){
            l = li + movX[i];
            c = ci + movY[i];

            /*O movimento é verificado e caso seja válido, uma solução é gerada*/
            if ((l >= 0) && (l < m) && (c >= 0) && (c < n) && ((lab[l][c] == -1) ||
                (lab[l][c] > lab[li][ci] + 1))){
                lab[l][c] = lab[li][ci] + 1;

                if (lab[l][c] + abs(l - lf) + abs(c - cf) < *min)
                    resolver_labirinto(lab, m, n, movX, movY, l, c, lf, cf, min);
            }
        }
    }
}
```

- Solução *backtracking*

```
static int mochila_bkt(int c[], int p[], int n, int b, int i, int max){
    int c1, c2;
    if (b < 0)
        return 0;
    else if (i < n){
        c1 = mochila_bkt(c, p, n, b, i + 1, max);
        c2 = mochila_bkt(c, p, n, b - p[i], i + 1, max + c[i]);
        return c1 > c2 ? c1 : c2;
    }else
        return max;
}
```

- Solução *branch-and-bound*

```
static int mochila_bnb(int c[], int p[], int n, int b, int i){
    if ((i < n) && (b > 0)){
        c1 = mochila_bkt(c, p, n, b, i + 1);
        if (b - p[i] >= 0)
            c2 = c[i] + mochila_bnb(c, p, n, b - p[i], i + 1);
        else
            c2 = 0;
        return max(c1, c2);
    }
    return 0;
}

int mochila(int c[], int p[], int n, int b){
    return mochila_bnb(c, p, n, b, 0);
}
```

- Método guloso
 - Considera a alternativa mais promissora e não explora as outras
 - As decisões não são reconsideradas
 - Rápida execução
 - Pode não gerar solução ótima
 - Simples implementação
 - Não necessita memória auxiliar

- Divisão e conquista
 - Fácil implementação
 - Paralelizável
 - Simplificação de problemas complexos
 - Necessita memória auxiliar (recursão)
 - Pode não gerar solução ótima

- Programação dinâmica
 - Difícil implementação
 - Explora todas as alternativas de forma eficiente
 - Em cada passo, as decisões podem ser revogadas
 - Solução pode ser lenta
 - Solução ótima
 - Necessita memória auxiliar (tabela para armazenamento de resultados parciais)

- *Backtracking*
 - Fácil implementação
 - Explora apenas as alternativas que geram soluções válidas
 - Busca exaustiva
 - Em cada passo, as decisões podem ser revogadas
 - Solução pode ser lenta
 - Solução ótima
 - Necessita memória auxiliar (recursão)

- *Branch-and-bound*
 - Explora apenas as alternativas que geram soluções válidas, mas considerando limitantes
 - Em cada passo, as decisões podem ser revogadas
 - Solução pode ser lenta
 - Busca "semi-exaustiva"
 - A solução ótima pode não ser encontrada dependendo dos limitantes
 - Necessita memória auxiliar (recursão)
 - Definir limitantes pode ser difícil



Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Introduction to Algorithms.
Third edition, The MIT Press, 2009.



Dias, Z.
Força Bruta, Backtracking e Branch and Bound. MC102 –
Algoritmos e Programação de Computadores.
Slides. Ciência de Computação. IC/Unicamp, 2013.



Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.



Rosa, J. L. G.
Paradigmas e Técnicas de Projetos de Algoritmos. SCC-201 –
Introdução à Ciência da Computação II.
Slides. Ciência de Computação. ICMC/USP, 2016.



Szwarcfiter, J.; Markenzon, L.
Estruturas de Dados e Seus Algoritmos.
LTC, 2010.



Toffolo, T.
Backtracking. Algoritmos e Programação Avançada. BCC402 –
Algoritmos e Programação Avançada.
Slides. Ciência de Computação. Decom/UFOP, 2011.



Ziviani, N.
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.