

Notas de Aula - AED2 – Técnicas e Análise de Algoritmos: backtracking e branch-and-bound
Prof. Jefferson T. Oliva

Nas aulas anteriores foram vistos:

- Força bruta
- Método guloso
- Divisão e conquista
- Programação dinâmica

Algoritmos força bruta testam exaustivamente todas as soluções possíveis de um problema para obter uma solução ótima. Em outras palavras, são geradas tanto soluções válidas, quanto inválidas. Por exemplo, no problema da mochila binária, seriam testadas todas as combinações possíveis de itens, desconsiderando a capacidade da mochila. Assim, para um conjunto de n itens, o custo da solução seria exponencial. Mesmo assim, apesar do custo, a solução é ótima.

Algoritmos força bruta não utiliza critérios para eliminar outras soluções que não poderão ser melhoradas. Seguindo o exemplo da mochila binária, caso uma solução ultrapasse a capacidade da mochila, ainda pode ser adicionado itens na solução (geração de outras soluções).

As soluções podem ser enumeradas de modo semelhante ao percurso em uma árvore que possua todas as recursões (ver slide 3).

Algoritmos força bruta não seguem regra fixa de computação

- Passos em direção à solução final são tentados e registrados
- Gera todos os possíveis candidatos à solução e verifica qual delas satisfaz o problema
- Geralmente simples implementar
- Força bruta é comumente utilizado em problemas de tamanho limitado ou quando um algoritmo mais eficiente é desconhecido. Um exemplo clássico de aplicação de força bruta é a ordenação, onde podem ser feitas todas as comparações possíveis em uma sequência. Outro exemplo simples é a busca sequencial em dados
- Quando a simplicidade é mais importante que a velocidade de execução, força bruta é uma solução muito útil

Por fim, uma solução força bruta pode ser muito custosa. Sendo assim, duas abordagens baseadas em força bruta foram propostas para amenizar o custo computacional:

- backtracking
- branch-and-bound

Backtracking

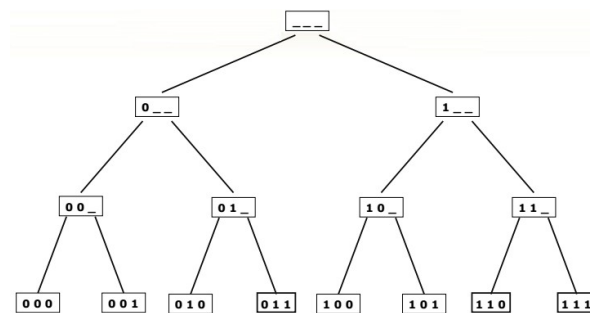
É uma abordagem que também testa todas as combinações possíveis de soluções. Diferentemente da força bruta, backtracking gera apenas soluções válidas.

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. Assim, é verificado se uma "solução parcial" possa ser completada e soluções inválidas são eliminadas sem serem completamente analisadas.

Muitos problemas podem ser modelados por meio de uma árvore, que, neste caso, representaria 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 2.



Diversos problemas famosos podem ser resolvidos através do backtracking, tais como:

- Passeio do cavalo
- Mochila
- Oito rainhas
- Labirinto
- etc

Problema do labirinto (backtracking)

Dada uma matriz $n \times m$ que representa um labirinto.

Posição inicial: $p_i = (x_i, y_i)$

Posição 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: a posição (x, y) representa uma parede
- 1: a posição (x, y) não faz parte do caminho
- i : -2: a posição (x, y) faz parte do caminho, sendo que $i \geq 0$

Ver exemplos nos slides 11–14. Códigos-fonte nos slides 15 e 16.

```
#define MAX 10
```

```
int main(){
    /*
    lab: labirinto
    n, m: número de linhas e de colunas ocupadas na matriz labirinto
    li, ci: posição inicial no labirinto
    lf, cf: posição final no labirinto
    resp: 1 se um caminho foi encontrado entre (li, ci) e (lf, cf), ou 0, caso contrário
    */
    int lab[MAX][MAX], n, m, li, ci, lf, cf, resp;

    /*Movimentos válidos no labirinto
    Exemplo: movX[0] e movY[0] indica que o movimento é para baixo
    */
    int movX[] = {0, 1, 0, -1}; // Movimentos no eixo x da matriz
    int movY[] = {1, 0, -1, 0}; // Movimentos no eixo y da matriz

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

    lab[ci][cf] = 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);
    eles printf("Solucao nao encontrada!\n");

    return 0;
}

int resolver_labirinto(int lab[MAX][MAX], int m, int n, movX[], movY[], int li, int ci, int lf, int cf){
    //l, c: posição de um passo no labirinto
    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++){
        // Posição do candidato a próximo passo
        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; // atualizar a quantidade de passos na posição l,c

            // Continuar o percurso no labirinto
            passos = resolver_labirinto(lab, m, n, movX, movY, l, c, lf, cf);
        }
    }
    return passos;
}
```

Muitas vezes pode ser desejável encontrar uma solução ótima, de acordo com algum critério de normalidade.

No problema do labirinto, por exemplo, podemos estar interessado em encontrar um caminho que contém o menor número de passos.

```
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.

Problema da mochila (backtracking)

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){
        if (b < 0)
            return 0;
        else
            return 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 acima, podemos ver que, além do custo computacional alto, também são geradas soluções inválidas, o que acarreta em desperdício de recursos.

Algumas adaptações no código evitaria a geração de soluções inválidas:

- Em vez da primeira verificação ser se i é maior ou igual a n , cuja prioridade é percorrer todos os objetos, verificar se $b < 0$. Se a mochila não pode suportar mais objetos, o algoritmo encerra a sua execução.
- Caso a nova verificação (se $b < 0$) 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 .

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

Abordagem relacionada com backtracking. A principal diferença é que em branch-and-bound é definido um limite de operações com o propósito de evitar buscas em espaços menos promissores.

Branch-and-bound:

- branch: explorar opções: testar todas as ramificações de uma solução candidata parcial.
- bound: limitar a busca: limitar a busca por soluções sempre que a solução candidata parcial for considerada infrutífera.

Branch-and-bound enumera, sistematicamente, os candidatos à solução.

Assim, as soluções parciais podem ser eliminadas nas seguintes situações:

- impossibilidade de gerar uma solução válida (igual ao que ocorre em backtracking).
- incapacidade de gerar uma solução ótima (quando é impossível encontrar um valor melhor em relação ao que já foi encontrado).
 - 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. Em outras palavras, quanto mais precisos forem estes limitantes, menos soluções parciais serão consideradas e a solução ótima será encontrada mais rapidamente.

Aplicação do branch-and-bound na solução do problema do 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;

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

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 o número de passos mínimos 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.

Aplicação do branch-and-bound na solução do problema da mochila

```
static int mochila_bnb(int c[], int p[], int n, int b, int i){
    int c1, c2;

    if ((i < n) && (b > 0)){
        c1 = mochila_bnb(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 c1 > c2 ? c1 : c2;
    }

    return 0;
}
```

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

Conclusão

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

Referências

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.