

MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2014

Roteiro

1 Força Bruta

2 Backtracking

3 Branch and Bound

Força Bruta

- Força bruta (ou busca exaustiva) é um tipo de algoritmo de uso geral que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema.
- Esse tipo de algoritmo geralmente possui uma implementação simples e sempre encontra uma solução se ela existir. Entretanto, seu custo computacional é proporcional ao número de candidatos à solução que, em muitos problemas reais, tende a crescer exponencialmente.
- A força bruta é tipicamente usada em problemas cujo tamanho é limitado ou quando não se conhece um algoritmo mais eficiente.
- Também pode ser usada quando a simplicidade da implementação é mais importante do que a velocidade de execução, como nos casos de aplicações críticas em que os erros de algoritmo possuem sérias consequências.

Força Bruta - Clique

- Considere um conjunto P de n pessoas e uma matriz M de tamanho $n \times n$, tal que $M[i, j] = M[j, i] = 1$, se as pessoas i e j se conhecem e $M[i, j] = M[j, i] = 0$, caso contrário.
- Problema: existe um subconjunto C (Clique), de r pessoas escolhidas de P , tal que qualquer par de pessoas de C se conhecem?
- Solução de força bruta: verificar, para todas as combinações simples (sem repetições) C de r pessoas escolhidas entre as n pessoas do conjunto P , se todos os pares de pessoas de C se conhecem.

Força Bruta - Clique

- Considere um conjunto P de 8 pessoas representado pela matriz abaixo (de tamanho 8×8):

x	1	2	3	4	5	6	7	8
1	1	0	1	1	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	1
4	1	0	1	1	1	1	1	1
5	1	1	0	1	1	0	0	0
6	1	0	1	1	0	1	1	1
7	1	0	1	1	0	1	1	0
8	0	1	1	1	0	1	0	1

- Existe um conjunto C de 5 pessoas escolhidas de P tal que qualquer par de pessoas de C se conhecem?

Força Bruta - Clique \times Combinação Simples

Existem $C(8, 5) = \frac{8!}{5!(8-5)!} = 56$ combinações simples de 5 elementos escolhidos dentre um conjunto de 8 elementos. Essas combinações são listadas a seguir:

1 2 3 4 5	1 2 4 6 8	1 3 5 7 8	2 3 5 6 8
1 2 3 4 6	1 2 4 7 8	1 3 6 7 8	2 3 5 7 8
1 2 3 4 7	1 2 5 6 7	1 4 5 6 7	2 3 6 7 8
1 2 3 4 8	1 2 5 6 8	1 4 5 6 8	2 4 5 6 7
1 2 3 5 6	1 2 5 7 8	1 4 5 7 8	2 4 5 6 8
1 2 3 5 7	1 2 6 7 8	1 4 6 7 8	2 4 5 7 8
1 2 3 5 8	1 3 4 5 6	1 5 6 7 8	2 4 6 7 8
1 2 3 6 7	1 3 4 5 7	2 3 4 5 6	2 5 6 7 8
1 2 3 6 8	1 3 4 5 8	2 3 4 5 7	3 4 5 6 7
1 2 3 7 8	1 3 4 6 7	2 3 4 5 8	3 4 5 6 8
1 2 4 5 6	1 3 4 6 8	2 3 4 6 7	3 4 5 7 8
1 2 4 5 7	1 3 4 7 8	2 3 4 6 8	3 4 6 7 8
1 2 4 5 8	1 3 5 6 7	2 3 4 7 8	3 5 6 7 8
1 2 4 6 7	1 3 5 6 8	2 3 5 6 7	4 5 6 7 8

Força Bruta - Clique

Note que todos os pares de pessoas do subconjunto $C = \{1, 3, 4, 6, 7\}$ se conhecem:

x	1	3	4	6	7
1	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
6	1	1	1	1	1
7	1	1	1	1	1

Como enumerar todas as combinações simples de r elementos de um conjunto de tamanho n ?

Força Bruta - Combinação Simples

```
#include <stdio.h>
#include <stdlib.h>

void combinacao_simples(int n, int r, int temp[],
                        int next, int size);

void imprime_vetor(int vetor[], int n) {
    int i;

    for (i = 0; i < n; i++)
        printf("%d ", vetor[i]);

    printf("\n");
}
```


Força Bruta - Combinação Simples

```
int main() {
    int n, r, *temp;

    printf("Entre com o valor de n: ");
    scanf("%d", &n);

    printf("Entre com o valor de r: ");
    scanf("%d", &r);

    temp = malloc(r * sizeof(int));

    combinacao_simples(n, r, temp, 1, 0);

    free(temp);

    return 0;
}
```

Força Bruta - Combinação Simples

```
void combinacao_simples(int n, int r, int temp[],
                       int next, int size) {
    int i;

    if (size == r)
        imprime_vetor(temp, r);
    else
        for (i = next; i <= n; i++) {
            temp[size] = i;
            combinacao_simples(n, r, temp, i + 1, size + 1);
        }
}
```

Força Bruta - Ciclo Hamiltoniano

- Considere um conjunto de n cidades e uma matriz M de tamanho $n \times n$ tal que $M[i, j] = 1$, se existe um caminho direto entre as cidades i e j , e $M[i, j] = 0$, caso contrário.
- Problema: existe uma forma de, saindo de uma cidade qualquer, visitar todas as demais cidades, sem passar duas vezes por nenhuma cidade e, ao final, retornar para a cidade inicial?
- Note que, se existe uma forma de sair de uma cidade X qualquer, visitar todas as demais cidades (sem repetir nenhuma) e depois retornar para X , então existe uma forma de fazer o mesmo para qualquer outra cidade do conjunto, já que existe um Ciclo Hamiltoniano (uma forma circular de visitar todas as cidades) e qualquer cidade do ciclo pode ser usado como ponto de partida.

Força Bruta - Ciclo Hamiltoniano

- Como vimos, qualquer cidade pode ser escolhida como cidade inicial. Sendo assim, vamos escolher, arbitrariamente a cidade n como ponto de partida.
- Solução de força bruta: testar todas as permutações das $n - 1$ primeiras cidades, verificando se existe um caminho direto entre a cidade n e a primeira da permutação, assim como um caminho entre todas as cidades consecutivas da permutação e, por fim, um caminho direto entre a última cidade da permutação e a cidade n .
- Ciclo Hamiltoniano: $n \rightsquigarrow [p_1 \rightsquigarrow p_2 \rightsquigarrow p_3 \rightsquigarrow \dots \rightsquigarrow p_{n-1}] \rightsquigarrow n$.

Força Bruta - Ciclo Hamiltoniano

- Considere um conjunto de 8 cidades representado pela matriz abaixo (de tamanho 8×8):

x	1	2	3	4	5	6	7	8
1	0	0	1	0	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	0
4	0	0	1	1	0	0	1	0
5	1	1	0	1	1	0	0	0
6	0	0	1	1	0	0	1	1
7	1	0	0	1	0	1	1	1
8	0	1	1	1	0	1	0	1

- Existe uma forma de, a partir da cidade 8, visitar todas as demais cidades, sem repetir nenhuma e, ao final, retornar para a cidade 8?

Força Bruta - Ciclo Hamiltoniano \times Permutação

Existem $P(7) = 7! = 5040$ permutações das 7 primeiras cidades da lista original:

1 2 3 4 5 6 7	...	7 6 5 2 3 4 1
1 2 3 4 5 7 6	3 6 4 5 1 7 2	7 6 5 2 4 1 3
1 2 3 4 6 5 7	3 6 4 5 2 1 7	7 6 5 2 4 3 1
1 2 3 4 6 7 5	3 6 4 5 2 7 1	7 6 5 3 1 2 4
1 2 3 4 7 5 6	3 6 4 5 7 1 2	7 6 5 3 1 4 2
1 2 3 4 7 6 5	3 6 4 5 7 2 1	7 6 5 3 2 1 4
1 2 3 5 4 6 7	3 6 4 7 1 2 5	7 6 5 3 2 4 1
1 2 3 5 4 7 6	3 6 4 7 1 5 2	7 6 5 3 4 1 2
1 2 3 5 6 4 7	3 6 4 7 2 1 5	7 6 5 3 4 2 1
1 2 3 5 6 7 4	3 6 4 7 2 5 1	7 6 5 4 1 2 3
1 2 3 5 7 4 6	3 6 4 7 5 1 2	7 6 5 4 1 3 2
1 2 3 5 7 6 4	3 6 4 7 5 2 1	7 6 5 4 2 1 3
1 2 3 6 4 5 7	3 6 5 1 2 4 7	7 6 5 4 2 3 1
1 2 3 6 4 7 5	3 6 5 1 2 7 4	7 6 5 4 3 1 2
1 2 3 6 5 4 7	...	7 6 5 4 3 2 1

Como enumerar todas as permutações de n valores distintos?

Força Bruta - Permutação (versão 1)

```
#include <stdio.h>
#include <stdlib.h>

void permutacao(int n, int temp[], int size);

void imprime_vetor(int vetor[], int n) {
    int i;

    for (i = 0; i < n; i++)
        printf("%d ", vetor[i]);

    printf("\n");
}
```

Força Bruta - Permutação (versão 1)

```
int main() {
    int n, *temp;

    printf("Entre com o valor de n: ");
    scanf("%d", &n);

    temp = malloc(n * sizeof(int));

    permutacao(n, temp, 0);

    free(temp);

    return 0;
}
```


Força Bruta - Permutação (versão 1)

```
void permutacao(int n, int temp[], int size) {
    int i, j, used;

    if (size == n)
        imprime_vetor(temp, n);
    else
        for (i = 1; i <= n; i++) {
            used = 0;

            for (j = 0; (j < size) && !used; j++)
                if (temp[j] == i)
                    used = 1;

            if (!used) {
                temp[size] = i;
                permutacao(n, temp, size + 1);
            }
        }
}
```

Força Bruta - Permutação (versão 2)

```
#include <stdio.h>
#include <stdlib.h>

void permutacao(int n, int temp[], int used[], int size);

void imprime_vetor(int vetor[], int n) {
    int i;

    for (i = 0; i < n; i++)
        printf("%d ", vetor[i]);

    printf("\n");
}
```

Força Bruta - Permutação (versão 2)

```
int main() {
    int i, n, *temp, *used;

    printf("Entre com o valor de n: ");
    scanf("%d", &n);

    temp = malloc(n * sizeof(int));
    used = malloc(n * sizeof(int));

    for (i = 0; i < n; i++)
        used[i] = 0; /* Indica que o elemento i + 1 nao foi usado */

    permutacao(n, temp, used, 0);

    free(temp);
    free(used);

    return 0;
}
```

Força Bruta - Permutação (versão 2)

```
void permutacao(int n, int temp[], int used[], int size) {
    int i;

    if (size == n)
        imprime_vetor(temp, n);
    else
        for (i = 0; i < n; i++)
            if (!used[i]) {
                used[i] = 1;
                temp[size] = i + 1;
                permutacao(n, temp, used, size + 1);
                used[i] = 0;
            }
}
```

Força Bruta - Exercícios

Exercício

Implemente um programa que resolva o problema da Clique, usando força bruta.

Exercício

Implemente um programa que resolva o problema do Ciclo Hamiltoniano, usando força bruta.

Força Bruta - Exercícios

Exercício

Implemente um programa que enumere todas as combinações com repetições de tamanho r dentre um conjunto de n elementos.

Exercício

Implemente um programa que enumere todos os arranjos simples (sem repetições) de tamanho r dentre um conjunto de n elementos.

Exercício

Implemente um programa que enumere todos os arranjos com repetições de tamanho r dentre um conjunto de n elementos.

Força Bruta - Exercícios

Exercício

Dado um inteiro n , gere todas as possíveis senhas formadas por:

- *n dígitos*
- *n dígitos ou letras minúsculas*
- *n dígitos ou letras minúsculas ou letras maiúsculas*

Backtracking

- *Backtracking* refere-se a um tipo de algoritmo para encontrar todas (ou algumas) soluções de um problema computacional, que incrementalmente constrói candidatas de soluções e abandona uma candidata parcialmente construída tão logo quanto for possível determinar que ela não pode gerar uma solução válida.
- *Backtracking* pode ser aplicado para problemas que admitem o conceito de “solução candidata parcial” e que exista um teste relativamente rápido para verificar se uma candidata parcial pode ser expandida para uma solução válida.
- Quando aplicável, *backtracking* é frequentemente muito mais rápido que algoritmos de enumeração total (força bruta), já que ele pode eliminar um grande número de soluções inválidas com um único teste.

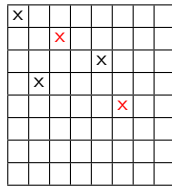
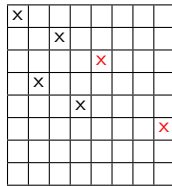
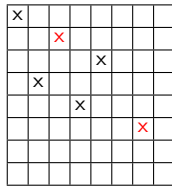
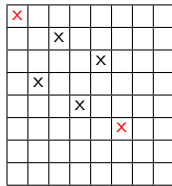
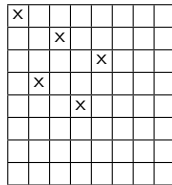
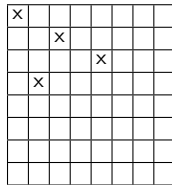
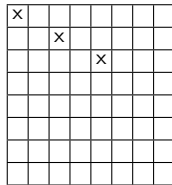
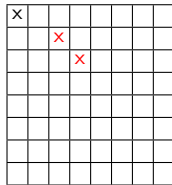
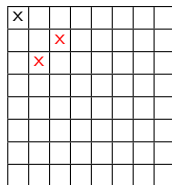
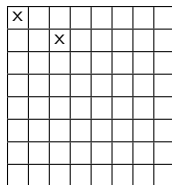
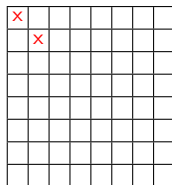
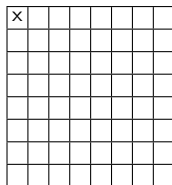
Backtracking

- Enquanto algoritmos de força bruta geram possíveis soluções e só depois verificam se elas são válidas, *backtracking* só gera soluções válidas.
- Alguns exemplos famosos de uso de *backtracking*:
 - ▶ Problema das Oito Rainhas
 - ▶ Passeio do Cavalo
 - ▶ Labirinto

Backtracking - Problema das Oito Rainhas

- O problema consiste em dispor oito rainhas em um tabuleiro de xadrez de dimensões 8×8 , de forma que nenhuma delas seja atacada por outra. Para tal, é necessário que duas rainhas quaisquer não estejam em uma mesma linha, coluna ou diagonal.
- Podemos representar uma solução candidata como um vetor `rainhas` de 8 posições, de tal forma que a rainha i é posicionada na linha i e na coluna `rainhas[i]`, já que duas rainhas nunca serão posicionadas na mesma linha.
- Como duas rainhas também nunca serão posicionadas na mesma coluna, o vetor `rainhas`, quando completo, representará uma permutação dos inteiros de 1 a 8.
- Um algoritmo de força bruta gerará todas as permutações e verificará se as rainhas não possuem conflitos (estão na mesma diagonal).
- Um algoritmo de *backtracking* apenas gerará as permutações que representam soluções válidas. Assim que um conflito for detectado, o algoritmo dará um passo para trás (*backtrack*) e tentará reposicionar a última rainha (que gerou o conflito).

Backtracking - Problema das Oito Rainhas



Backtracking - Problema das Oito Rainhas

- O problema das oito rainhas possui 92 soluções distintas, as quais podem ser obtidas a partir de um conjunto de 12 soluções únicas por meio de operações de simetria (reflexão e rotação).
- O problema pode ser generalizado de tal forma que o objetivo seja posicionar n rainhas num tabuleiro de dimensões $n \times n$, de tal forma que nenhuma delas seja atacada pela outra.

Backtracking - Problema das Oito Rainhas

- 12 soluções básicas:

		x			
				x	
	x				
					x
x			x		
				x	

			x		
	x				
		x			
				x	
	x				
					x
x				x	

		x			
	x				
					x
		x			
				x	
					x
x				x	

		x			
				x	
					x
	x				
x					
					x
				x	

		x			
				x	
x					x
			x		
					x
			x		
x					

			x		
	x				
					x
		x			
					x
x					
	x				

				x	
					x
			x		
x					
		x			
					x
				x	

			x		
x					
				x	
					x
		x			
					x

		x			
				x	
x					x
				x	
					x
x					

				x	
	x				
x					x
			x		
					x
	x				

			x		
					x
x					
					x
				x	
	x				
					x

				x	
			x		
x					x
					x
	x				
				x	

Backtracking - Problema das n Rainhas

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void nRainhas(int rainhas[], int n, int size);
int valida(int rainhas[], int k);

void imprime_vetor(int vetor[], int n) {
    int i;

    for (i = 0; i < n - 1; i++)
        printf("%d ", vetor[i]);

    printf("\n");
}
```

Backtracking - Problema das n Rainhas

```
int main() {
    int n, *rainhas;

    printf("Numero de rainhas: ");
    scanf("%d", &n);

    rainhas = malloc(n * sizeof(int));

    /* Inicialmente nenhuma das n rainhas esta posicionada */
    nRainhas(rainhas, n, 0);

    free(rainhas);

    return 0;
}
```

Backtracking - Problema das n Rainhas

```
void nRainhas(int rainhas[], int n, int size) {
    int i;

    if (size == n)
        imprime_vetor(rainhas, n);
    else
        for (i = 1; i <= n; i++) {
            rainhas[size] = i;
            if (valida(rainhas, size))
                nRainhas(rainhas, n, size + 1);
        }
}
```


Backtracking - Problema das n Rainhas

```
int valida(int rainhas[], int k) {
    int i;

    for (i = 0; i < k; i++)
        /* Se duas rainhas estao na mesma coluna... */
        if ((rainhas[i] == rainhas[k]) ||
            /* ... ou estao na mesma diagonal ... */
            (abs(rainhas[i] - rainhas[k]) == (k - i)))
            /* ... entao a solucao eh invalida */
            return 0;

    /* Solucao valida */
    return 1;
}
```

Backtracking - Passeio do Cavalo

- Dado um tabuleiro de $n \times n$ posições, o cavalo se movimenta segundo as regras do xadrez. A partir de uma posição inicial (x, y) , o problema consiste em encontrar, se existir, um passeio do cavalo com $n^2 - 1$ movimentos tal que todas as posições do tabuleiro sejam visitadas uma única vez.
- O tabuleiro pode ser representado por uma matriz M de tamanho $n \times n$. A situação de cada posição do tabuleiro pode ser representada por um inteiro para registrar a evolução das posições visitadas pelo cavalo:

$$M[x, y] = \begin{cases} 0, & \text{se posição } (x, y) \text{ não foi visitada} \\ 1 \leq i \leq n^2, & \text{se } (x, y) \text{ foi visitada no } i\text{-ésimo passo} \end{cases}$$

Backtracking - Passeio do Cavalo

- A figura abaixo mostra uma solução para um tabuleiro de dimensões 8×8 , cuja posição inicial do passeio é $(1, 1)$:

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

Backtracking - Passeio do Cavalo

```
#include <stdio.h>
#define MAX 10

struct Move {
    int x, y;
};
typedef struct Move Move;

int passeio(int n, int x, int y, int M[MAX][MAX], Move move[]);

void imprime_matriz(int M[MAX][MAX], int n) {
    int x, y;

    for (x = 0; x < n; x++) {
        for (y = 0; y < n; y++)
            printf(" %3d ", M[x][y]);
        printf("\n");
    }
}
```

Backtracking - Passeio do Cavalo

```
int main() {
    int M[MAX][MAX], x, y, n, startX, startY;

    /* Define os movimentos do cavalo */
    Move move[8] = {{ 2, 1}, { 1, 2}, {-1, 2}, {-2, 1},
                  {-2,-1}, {-1, -2}, { 1, -2}, { 2, -1}};

    printf("Entre com o valor de n: ");
    scanf("%d", &n);

    printf("Entre com a linha inicial do cavalo: ");
    scanf("%d", &startX);

    printf("Entre com a coluna inicial do cavalo: ");
    scanf("%d", &startY);

    ...
}
```

Backtracking - Passeio do Cavalo

```
...

/* Inicializacao do tabuleiro */
for (x = 0; x < n; x++)
    for (y = 0; y < n; y++)
        M[x][y] = 0;

/* Define a posicao inicial do cavalo */
M[startX - 1][startY - 1] = 1;

/* Verifica se ha uma solucao valida */
if (passeio(n, startX - 1, startY - 1, M, move))
    imprime_matriz(M, n);
else
    printf("Nao existe solucao.\n");

return 0;
}
```

Backtracking - Passeio do Cavalo

```
int passeio(int n, int x, int y, int M[MAX][MAX], Move move[]) {
    int k, nextX, nextY;

    if (M[x][y] == n * n) return 1;

    /* Testa todos os movimentos a partir da posicao atual do cavalo (x,y) */
    for (k = 0; k < 8; k++) {
        nextX = x + move[k].x;
        nextY = y + move[k].y;

        /* Verifica se o movimento eh valido e gera uma solucao factivel */
        if ((nextX >= 0) && (nextX < n) && (nextY >= 0) && (nextY < n) &&
            (M[nextX][nextY] == 0)) {
            M[nextX][nextY] = M[x][y] + 1;

            if (passeio(n, nextX, nextY, M, move))
                return 1;
            else
                M[nextX][nextY] = 0; /* Libera a posicao do tabuleiro */
        }
    }

    return 0;
}
```

Backtracking - Labirinto

- Dado um labirinto representado por uma matriz de tamanho $n \times m$, uma posição inicial $p_i = (x_i, y_i)$ e uma posição final $p_f = (x_f, y_f)$, tal que $p_i \neq p_f$, determinar se existe um caminho entre p_i e p_f .
- Podemos representar o labirinto como uma matriz M tal que:

$$M[x, y] = \begin{cases} -2, & \text{se a posição } (x, y) \text{ representa uma parede} \\ -1, & \text{se a posição } (x, y) \text{ não pertence ao caminho} \\ i \geq 0, & \text{se a posição } (x, y) \text{ pertence ao caminho} \end{cases}$$

- Neste caso, vamos supor que o labirinto seja cercado por paredes, eventualmente apenas com exceção do local designado como saída.

Backtracking - Labirinto

- A figura abaixo mostra um labirinto de tamanho 8×8 :

X	X	X	X	X	X	X	X
X	•						X
X	X		X				X
X			X	X	X		X
X		X	X				X
X		X				X	X
X				X			X
X	X	X	X	X	X	○	X

- Legenda:
 - ▶ X: parede/obstáculo
 - ▶ •: posição inicial
 - ▶ ○: posição final (saída do labirinto)

Backtracking - Labirinto

- Caminho encontrado usando a seguinte ordem de busca:
 - ▶ para esquerda
 - ▶ para baixo
 - ▶ para direita
 - ▶ para cima

X	X	X	X	X	X	X	X
X	00	01					X
X	X	02	X				X
X	04	03	X	X	X		X
X	05	X	X				X
X	06	X	10	11	12	X	X
X	07	08	09	X	13	14	X
X	X	X	X	X	X	15	X

Backtracking - Labirinto

- Caminho encontrado usando a seguinte ordem de busca:

- ▶ para direita
- ▶ para baixo
- ▶ para esquerda
- ▶ para cima

X	X	X	X	X	X	X	X
X	00	01	02	03	04	05	X
X	X		X			06	X
X			X	X	X	07	X
X		X	X		09	08	X
X		X			10	X	X
X				X	11	12	X
X	X	X	X	X	X	13	X

Backtracking - Labirinto

```
#include <stdio.h>
#define MAX 10

struct Move {
    int linha, coluna;
};
typedef struct Move Move;

int labirinto(int M[MAX][MAX], Move move[], int Li, int Ci, int Lf, int Cf);

void imprimeLabirinto(int M[MAX][MAX], int n, int m) {
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            if (M[i][j] == -2) printf(" XX");
            if (M[i][j] == -1) printf("  ");
            if (M[i][j] >= 0) printf(" %02d", M[i][j]);
        }
        printf("\n");
    }
}
```

Backtracking - Labirinto

```
void obtemLabirinto(int M[MAX][MAX], int *n, int *m,
                   int *Li, int *Ci, int *Lf, int *Cf) {
    int i, j, d;

    scanf("%d %d", n, m); /* Dimensoes do labirinto */
    scanf("%d %d", Li, Ci); /* Coordenadas da posicao inicial */
    scanf("%d %d", Lf, Cf); /* Coordenadas da posicao final (saida) */

    /* Labirinto: 1 = parede ou obstaculo
                 0 = posicao livre */
    for (i = 0; i < *n; i++)
        for (j = 0; j < *m; j++) {
            scanf("%d", &d);
            if (d == 1)
                M[i][j] = -2;
            else
                M[i][j] = -1;
        }
}
```

Backtracking - Labirinto

```
int main() {
    int M[MAX][MAX], resposta, n, m, Li, Ci, Lf, Cf;

    /* Define os movimentos validos no labirinto */
    Move move[4] = {{0, +1}, {+1, 0}, {0, -1}, {-1, 0}};

    /* Obtem as informacoes do labirinto */
    obtemLabirinto(M, &n, &m, &Li, &Ci, &Lf, &Cf);

    M[Li - 1][Ci - 1] = 0; /* Define a posicao inicial no tabuleiro */

    /* Tenta encontrar um caminho no labirinto */
    resposta = labirinto(M, move, Li - 1, Ci - 1, Lf - 1, Cf - 1);

    if (resposta == 0) {
        printf("Nao existe solucao.\n");
    } else {
        printf("Existe uma solucao em %d passos:\n", resposta);
        imprimeLabirinto(M, n, m);
    }

    return 0;
}
```

Backtracking - Labirinto

```
int labirinto(int M[MAX][MAX], Move move[],
              int Li, int Ci, int Lf, int Cf) {
    int L, C, k, passos;

    if ((Li == Lf) && (Ci == Cf)) return M[Li][Ci];

    /* Testa todos os movimentos a partir da posicao atual */
    for (k = 0; k < 4; k++) {
        L = Li + move[k].linha;
        C = Ci + move[k].coluna;

        /* Verifica se o movimento eh valido e gera uma solucao factivel */
        if (M[L][C] == -1) {
            M[L][C] = M[Li][Ci] + 1;

            passos = labirinto(M, move, L, C, Lf, Cf);

            if (passos > 0) return passos;
        }
    }

    return 0;
}
```

Backtracking - Labirinto

- Exemplo de entrada:

```
8 8
2 2
8 7
1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 1
1 0 0 0 1 0 0 1
1 0 0 1 1 0 0 1
1 0 1 1 0 0 0 1
1 0 0 1 1 1 1 1
1 0 0 0 0 0 0 1
1 1 1 1 1 1 0 1
```


Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX    08 XX
XX      XX XX    09 XX
XX     XX XX 12 11 10 XX
XX      XX XX XX XX XX
XX
XX XX XX XX XX XX    XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 13 08 XX
XX      XX XX 12 09 XX
XX    XX XX 12 11 10 XX
XX      XX XX XX XX XX
XX                                XX
XX XX XX XX XX XX      XX
```

Backtracking - Labirinto

- Exemplo de saída:

Existe uma solucao em 13 passos:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 13 08 XX
XX 04 03 XX XX 12 09 XX
XX 05 XX XX 12 11 10 XX
XX 06 07 XX XX XX XX XX
XX    08 09 10 11 12 XX
XX XX XX XX XX XX 13 XX
```

Backtracking - Exercícios

Exercício

Implemente um programa que resolva o problema das n Rainhas usando força bruta.

Exercício

Implemente um programa que resolva o problema da Clique usando backtracking.

Exercício

Implemente um programa que resolva o problema do Ciclo Hamiltoniano usando backtracking.

Em todos os casos, compare o desempenho do programa que usa a técnica de força bruta com o programa que usa *backtracking*.

Backtracking - Problemas de Otimização

- Muitas vezes não estamos apenas interessados em encontrar uma solução qualquer, mas em encontrar uma solução ótima (segundo algum critério de otimalidade pré-estabelecido).
- Por exemplo, no problema do labirinto, ao invés de determinar se existe um caminho entre o ponto inicial e o final (saída), podemos estar interessados em encontrar uma solução que use o menor número possível de passos.

Exercício

Modifique o programa visto anteriormente, que verifica se existe uma solução para um dado labirinto, de forma que ele encontre uma solução ótima para o problema (menor número possível de passos).

Backtracking - Labirinto

```
#include <stdio.h>
#define MAX 10

struct Move {
    int linha, coluna;
};
typedef struct Move Move;

void imprimeLabirinto(int M[MAX][MAX], int n, int m) {
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            if (M[i][j] == -2) printf(" XX");
            if (M[i][j] == -1) printf("  ");
            if (M[i][j] >= 0) printf(" %02d", M[i][j]);
        }
        printf("\n");
    }
}
```

Backtracking - Labirinto

```
void obtemLabirinto(int M[MAX][MAX], int *n, int *m,
                   int *Li, int *Ci, int *Lf, int *Cf) {
    int i, j, d;

    scanf("%d %d", n, m); /* Dimensoes do labirinto */
    scanf("%d %d", Li, Ci); /* Coordenadas da posicao inicial */
    scanf("%d %d", Lf, Cf); /* Coordenadas da posicao final (saida) */

    /* Labirinto: 1 = parede ou obstaculo
                 0 = posicao livre */
    for (i = 0; i < *n; i++)
        for (j = 0; j < *m; j++) {
            scanf("%d", &d);
            if (d == 1)
                M[i][j] = -2;
            else
                M[i][j] = -1;
        }
}
```

Backtracking - Labirinto

```
int main() {
    int M[MAX][MAX], min, n, m, Li, Ci, Lf, Cf;

    /* Define os movimentos validos no labirinto */
    Move move[4] = {{0, +1}, {+1, 0}, {0, -1}, {-1, 0}};

    /* Obtem as informacoes do labirinto */
    obtemLabirinto(M, &n, &m, &Li, &Ci, &Lf, &Cf);

    M[Li - 1][Ci - 1] = 0; /* Define a posicao inicial no tabuleiro */

    /* Tenta encontrar um caminho no labirinto */
    min = n * m;
    labirinto(M, move, Li - 1, Ci - 1, Lf - 1, Cf - 1, &min);

    if (min == n * m) {
        printf("Nao existe solucao.\n");
    } else {
        printf("Existe uma solucao em %d passos:\n", min);
        imprimeLabirinto(M, n, m);
    }
    return 0;
}
```


Backtracking - Labirinto

```
void labirinto(int M[MAX][MAX], Move move[],
              int Li, int Ci, int Lf, int Cf, int *min) {
    int L, C, k;

    if ((Li == Lf) && (Ci == Cf)) {
        if (M[Lf][Cf] < *min)
            *min = M[Li][Ci];
    } else {
        /* Testa todos os movimentos a partir da posicao atual */
        for (k = 0; k < 4; k++) {
            L = Li + move[k].linha;
            C = Ci + move[k].coluna;

            /* Verifica se o movimento eh valido e pode gerar uma solucao otima */
            if ((M[L][C] == -1) || (M[L][C] > M[Li][Ci] + 1)) {
                M[L][C] = M[Li][Ci] + 1;

                labirinto(M, move, L, C, Lf, Cf, min);
            }
        }
    }
}
```

Backtracking - Labirinto

- Exemplo de entrada:

```
8 8
2 2
8 7
1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 1
1 0 0 0 1 0 0 1
1 0 0 1 1 0 0 1
1 0 1 1 0 0 0 1
1 0 0 1 1 1 1 1
1 0 0 0 0 0 0 1
1 1 1 1 1 1 0 1
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX    08 XX
XX      XX XX    09 XX
XX     XX XX 12 11 10 XX
XX      XX XX XX XX XX
XX
XX XX XX XX XX XX    XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 13 08 XX
XX      XX XX 12 09 XX
XX    XX XX 12 11 10 XX
XX      XX XX XX XX XX
XX                                XX
XX XX XX XX XX XX      XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 11 08 XX
XX      XX XX 10 09 XX
XX    XX XX 12 11 10 XX
XX      XX XX XX XX XX
XX                                XX
XX XX XX XX XX XX      XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 09 08 XX
XX      XX XX 10 09 XX
XX     XX XX 12 11 10 XX
XX      XX XX XX XX XX
XX
XX XX XX XX XX XX XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 07 08 XX
XX      XX XX 08 09 XX
XX    XX XX 10 09 10 XX
XX      XX XX XX XX XX
XX                                XX
XX XX XX XX XX XX      XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 07 08 XX
XX 04 03 XX XX 08 09 XX
XX 05 XX XX 10 09 10 XX
XX 06 07 XX XX XX XX XX
XX    08 09 10 11 12 XX
XX XX XX XX XX XX 13 XX
```


Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 07 08 XX
XX 04 03 XX XX 08 09 XX
XX 05 XX XX 10 09 10 XX
XX 06 07 XX XX XX XX XX
XX 09 08 09 10 11 12 XX
XX XX XX XX XX XX 13 XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 07 08 XX
XX 04 03 XX XX 08 09 XX
XX 05 XX XX 10 09 10 XX
XX 06 07 XX XX XX XX XX
XX 07 08 09 10 11 12 XX
XX XX XX XX XX XX 13 XX
```

Backtracking - Labirinto

- Exemplo de saída:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 07 08 XX
XX 02 03 XX XX 08 09 XX
XX 03 XX XX 10 09 10 XX
XX 04 05 XX XX XX XX XX
XX 07 06 07 08 09 10 XX
XX XX XX XX XX XX 11 XX
```

Backtracking - Labirinto

- Exemplo de saída:

Existe uma solucao em 11 passos:

```
XX XX XX XX XX XX XX XX
XX 00 XX 04 05 06 07 XX
XX 01 02 03 XX 07 08 XX
XX 02 03 XX XX 08 09 XX
XX 03 XX XX 10 09 10 XX
XX 04 05 XX XX XX XX XX
XX 05 06 07 08 09 10 XX
XX XX XX XX XX XX 11 XX
```

Branch and Bound

- *Branch and Bound* refere-se a um tipo de algoritmo usado para encontrar soluções ótimas para vários problemas de otimização, especialmente em otimização combinatória.
- Problemas de otimização podem ser tanto de maximização (por exemplo, maximizar o valor de uma solução), quanto de minimização (por exemplo, minimizar o custo de uma solução).
- *Branch and Bound* consiste em uma enumeração sistemática de todos os candidatos à solução, com eliminação de uma candidata parcial quando uma destas duas situações for detectada (considerando um problema de minimização):
 - ▶ A candidata parcial é incapaz de gerar uma solução válida (teste similar realizado pelo método de *backtracking*).
 - ▶ A candidata parcial é incapaz de gerar uma solução ótima, considerando o valor da melhor solução encontrada até então (limitante superior) e o custo ainda necessário para gerar uma solução a partir da solução candidata atual (limitante inferior).

Branch and Bound

- O desempenho de um programa de *Branch and Bound* está fortemente relacionado à qualidade dos seus limitantes inferiores e superiores: quanto mais precisos forem estes limitantes, menos soluções parciais serão consideradas e mais rápido o programa encontrará a solução ótima.
- O nome *Branch and Bound* refere-se às duas fases do algoritmo:
 - ▶ *Branch*: testar todas as ramificações de uma solução candidata parcial.
 - ▶ *Bound*: limitar a busca por soluções sempre que detectar que o atual ramo da busca é infrutífero.

Branch and Bound - Labirinto

- Podemos alterar o programa visto anteriormente para encontrar um caminho ótimo num labirinto usando a técnica de *Branch and Bound*.
- Podemos inicialmente notar que se um caminho parcial já usou tantos passos quanto o melhor caminho completo previamente descoberto, então este caminho parcial pode ser descartado.
- Mais do que isso, se o número de passos do caminho parcial mais o número de passos mínimos necessários entre a posição atual e a saída (desconsiderando eventuais obstáculos) for maior ou igual ao número de passos do melhor caminho previamente descoberto, então este caminho parcial também pode ser descartado.

Backtracking - Labirinto

```
void labirinto(int M[MAX][MAX], Move move[],
               int Li, int Ci, int Lf, int Cf, int *min) {
    int L, C, k;

    if ((Li == Lf) && (Ci == Cf)) {
        if (M[Lf][Cf] < *min)
            *min = M[Li][Ci];
    } else {
        /* Testa todos os movimentos a partir da posicao atual */
        for (k = 0; k < 4; k++) {
            L = Li + move[k].linha;
            C = Ci + move[k].coluna;

            /* Verifica se o movimento eh valido e pode gerar uma solucao otima */
            if ((M[L][C] == -1) || (M[L][C] > M[Li][Ci] + 1)) {
                M[L][C] = M[Li][Ci] + 1;

                labirinto(M, move, L, C, Lf, Cf, min);
            }
        }
    }
}
```


Branch and Bound - Labirinto (versão 1)

```
void labirinto(int M[MAX][MAX], Move move[],
              int Li, int Ci, int Lf, int Cf, int *min) {
    int L, C, k;

    if ((Li == Lf) && (Ci == Cf)) {
        if (M[Lf][Cf] < *min)
            *min = M[Li][Ci];
    } else {
        /* Testa todos os movimentos a partir da posicao atual */
        for (k = 0; k < 4; k++) {
            L = Li + move[k].linha;
            C = Ci + move[k].coluna;

            /* Verifica se o movimento eh valido e pode gerar uma solucao otima */
            if ((M[L][C] == -1) || (M[L][C] > M[Li][Ci] + 1)) {
                M[L][C] = M[Li][Ci] + 1;
                if (M[L][C] < *min)
                    labirinto(M, move, L, C, Lf, Cf, min);
            }
        }
    }
}
```

Branch and Bound - Labirinto (versão 2)

```
void labirinto(int M[MAX][MAX], Move move[],
              int Li, int Ci, int Lf, int Cf, int *min) {
    int L, C, k;

    if ((Li == Lf) && (Ci == Cf)) {
        if (M[Lf][Cf] < *min)
            *min = M[Li][Ci];
    } else {
        /* Testa todos os movimentos a partir da posicao atual */
        for (k = 0; k < 4; k++) {
            L = Li + move[k].linha;
            C = Ci + move[k].coluna;

            /* Verifica se o movimento eh valido e pode gerar uma solucao otima */
            if ((M[L][C] == -1) || (M[L][C] > M[Li][Ci] + 1)) {
                M[L][C] = M[Li][Ci] + 1;
                if (M[L][C] + abs(L - Lf) + abs(C - Cf) < *min)
                    labirinto(M, move, L, C, Lf, Cf, min);
            }
        }
    }
}
```

Branch and Bound - Caixeiro Viajante

- Considere um conjunto de n cidades e uma matriz C (não necessariamente simétrica), de tamanho $n \times n$, tal que $C[i, j] > 0$ indica o custo de, saindo da cidade i , chegar até a cidade j .
- Problema: qual é a forma, de custo mínimo, de, saindo de uma cidade qualquer, visitar todas as demais cidades, sem passar duas vezes por nenhuma cidade e, ao final, retornar para a cidade inicial?

Exercício

Escreva um programa de força bruta para resolver o problema do Caixeiro Viajante.

Exercício

Escreva um programa de branch and bound para resolver o problema do Caixeiro Viajante.

Branch and Bound - Clique Máxima

- Considere um conjunto P de n pessoas e uma matriz M de tamanho $n \times n$, tal que $M[i, j] = M[j, i] = 1$, se as pessoas i e j se conhecem e $M[i, j] = M[j, i] = 0$, caso contrário.
- Problema: qual é o maior subconjunto C (Clique Máxima) de pessoas escolhidas de P , tal que qualquer par de pessoas de C se conhecem?

Exercício

Escreva um programa de força bruta para resolver o problema da Clique Máxima.

Exercício

Escreva um programa de backtracking para resolver o problema da Clique Máxima.

Exercício

Escreva um programa de branch and bound para resolver o problema da Clique Máxima.