Força Bruta e *Backtracking*

Laboratório de Programação Competitiva I

Pedro Henrique Paiola

Rene Pegoraro

Wilson M Yonezawa

Paradigmas de Projeto de Algoritmos

- Certos problemas podem requerer **abordagens** adequadas para desenvolver/projetar um algoritmo que o resolva.
- A forma como um algoritmo aborda o problema pode levar a um desempenho ineficiente.
  - Em Programação Competitiva, se o algoritmo não seguir uma abordagem adequada, provavelmente ele excederá o tempo limite de execução (TLE)
- De forma geral, pode-se considerar os paradigmas como estratégias de como abordar o problema e modelar sua solução.

Paradigmas de Projeto de Algoritmos

- Paradigmas
  - Indução
  - Recursividade
  - Força bruta*
  - Tentativa e erro (*Backtracking*)
  - Divisão e conquista
  - Programação dinâmica
  - Algoritmos gulosos
  - Algoritmos aproximados (heurísticos)

Força Bruta

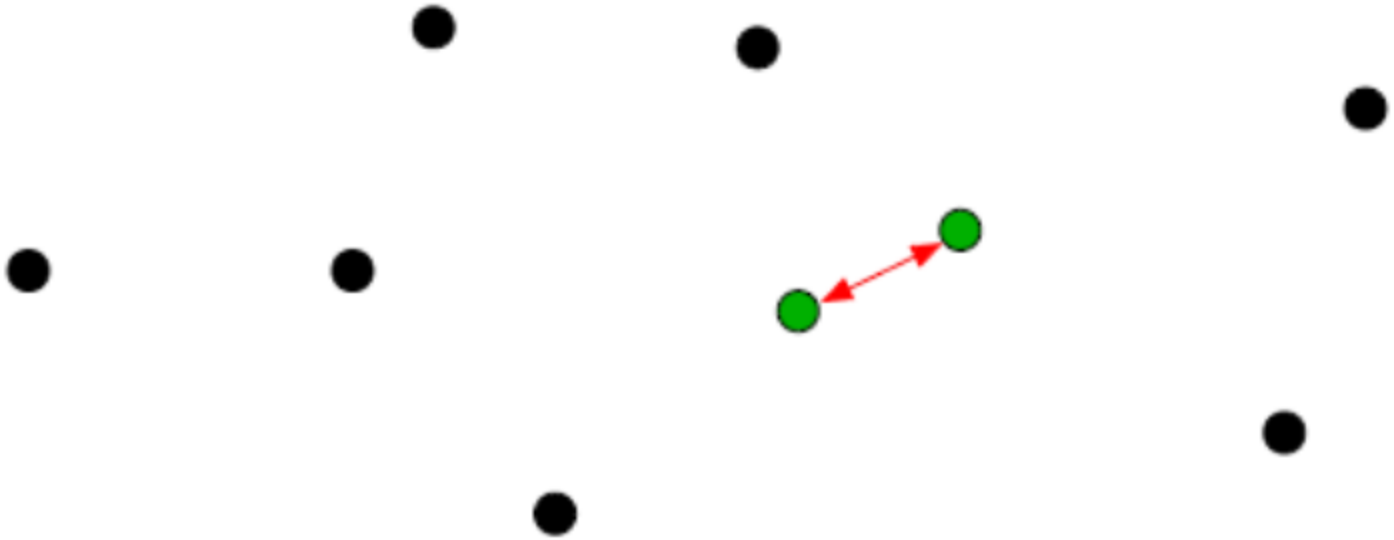
- Força bruta (ou busca exaustiva) é um tipo de algoritmo de uso geral que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema.
- É uma busca cega de todas as possibilidades de solução.
- Útil para pequenos problemas, normalmente simples de implementar. Sempre que existe uma solução, ela será encontrada.
- Grande esforço computacional. O custo computacional tende a crescer exponencialmente.
- Exemplo:
  - Busca sequencial

Força Bruta

- Método
  - Construir uma forma de listar todas as situações possíveis para o problema, de forma sistemática:
    - Todas as soluções são testadas e nenhuma é repetida
  - Avaliar as soluções uma a uma, armazenando a melhor solução até o momento.
  - Ao finalizar a busca, retornar a melhor solução.

Força Bruta

- Exemplo: menor distância entre 2 pontos
  - Dados n pontos no plano, determinar a distância mínima entre qualquer par de pontos.



$$Distancia(X_1, Y_1, X_2, Y_2) = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)}$$

Força Bruta

- Solução por força bruta: testar cada par de pontos

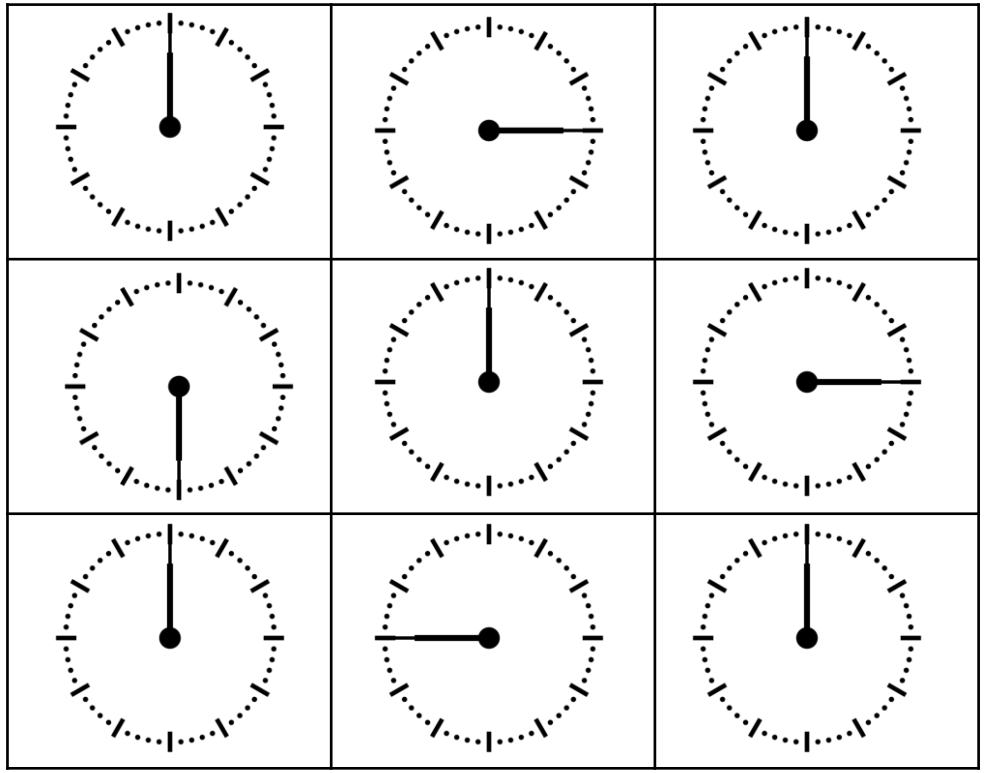
```
double menorDistancia(pair<double, double> pontos[], int n){
    double d = INF;
    for(int i = 0; i < n - 1; i++)
        for(int j = i+1; j < n; j++)
            if (distancia(pontos[i], pontos[j]) < d)
                d = distancia(pontos[i], pontos[j]);
    return d;
}

//Complexidade: O(n²)
```

Força Bruta

- Em Programação Competitiva, só se aplica Força Bruta em problemas fáceis?
  - Não necessariamente
  - Alguns problemas podem exigir uma solução mais complexa, mas com um espaço de busca pequeno, permitindo uma solução por Força Bruta
  - E/ou podem se beneficiar de uma solução um pouco mais criativa do que uma força bruta “pura”

Força Bruta - Problema dos Relógios

- Exemplo: Problema dos Relógios
  - Considere um grupo de novos relógios dispostos em uma grade 3x3. O ponteiro das horas de cada um aponta para 12, 3, 6 ou 9.
  - São definidos 9 subconjuntos, cada um contendo dois ou mais relógios.
  - Seu objetivo é manipular os relógios para que todos apontem para 12 horas.
  - Porém, para isso, você só pode escolher um dos 9 subconjuntos definidos e então rotacionar todos os relógios deste subconjunto em 90º no sentido horário.
  - **Determinar a menor sequência de movimentos para resolver o problema.** Considere que sempre existe uma solução



Força Bruta - Problema dos Relógios

- A solução por força bruta para este problema seria testar todas as possíveis combinações de movimentos, começando apenas com 1 movimento possível, até encontrar uma solução.
  - Testar todas as possibilidades com apenas 1 movimento
  - Testar todas as possibilidades com apenas 2 movimentos
  - Testar todas as possibilidades com apenas 3 movimentos
  - ...
  - Solução: k movimentos
- Essa solução tem complexidade $O(9^k)$

Força Bruta - Problema dos Relógios

- Note, porém, que a ordem dos movimentos não importa. Isso reduz a complexidade para $O(k^9)$.

Força Bruta - Problema dos Relógios

- Note, porém, que a ordem dos movimentos não importa. Isso reduz a complexidade para $O(k^9)$.
- Além disso, fazer um mesmo movimento 4 vezes é a mesma coisa de não fazer nenhum. Sendo assim, nenhum movimento deve ser realizado mais de 3 vezes.
  - Para cada movimento tenho 4 possibilidades: não fazê-lo, fazê-lo 1 vez, 2 vezes ou 3 vezes.
  - Como são 9 movimentos possíveis, o total de possibilidades é $4^9 = 262.072$ possibilidades. É perfeitamente possível avaliar 262.072 possibilidades no tempo normalmente disponibilizado.

Backtracking

- *Backtracking* é um refinamento do algoritmo de busca por força bruta, no qual boa parte das soluções podem ser eliminadas sem serem explicitamente examinadas.
- Aplicável quando as soluções candidatas podem ser construídas incrementalmente.
- A ideia central é retroceder quando detectar que a solução candidata é inviável.

Backtracking

- Cada solução possível é representada por um vetor das suas componentes $S = (c_1, c_2, \ldots, c_n)$ onde cada c_i é selecionada de um conjunto P_i das componentes possíveis para a posição i.
- Uma solução parcial é dada por:
  - $S = (c_1, c_2, \ldots, c_k)$, com $k \leq n$
- Tenta-se estender a solução acrescentando c_{k+1}.
- Se a solução não pode mais ser estendida, retrocede-se à escolha do c anterior.

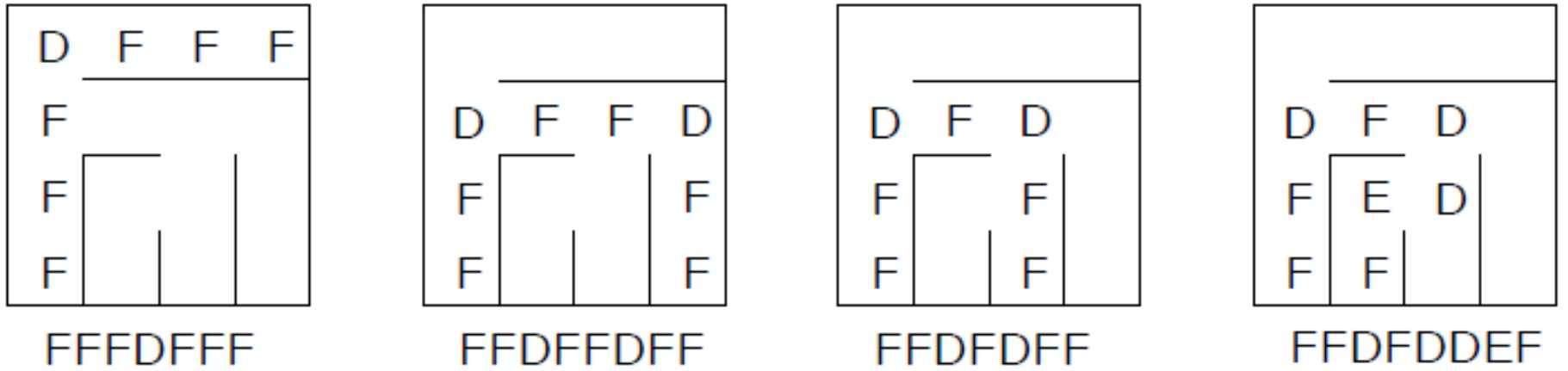
Backtracking

- Pseudocódigo:

```
backtracking(S, k){
    if (!ehValida(S, k))
        return;

    if (ehSolucao(S, k))
        processa(S, k);

    for(c_{k+1} in P_{k+1}){
        S' = S + c_{k+1};
        backtracking(S', k+1);
    }
}
```

Backtracking - Labirinto

- Exemplo: labirinto



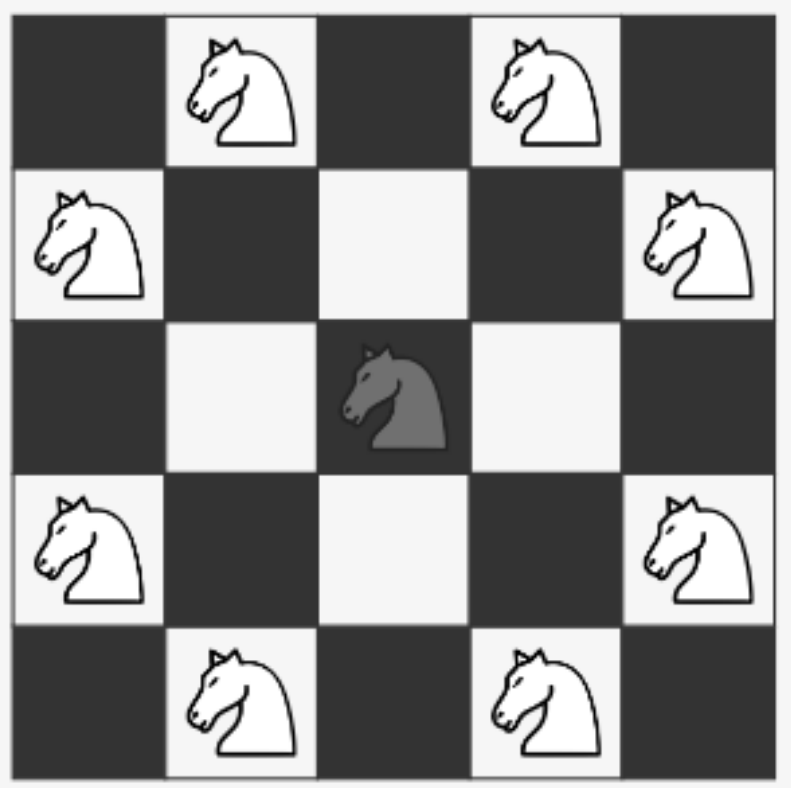
- A busca ocorre passo a passo, recursivamente, em todas as direções. Se não existe mais possibilidades, retorna-se ao nível anterior da recursão.

Backtracking - Labirinto

```
vector<char> movimentosPossiveis = {'F', 'D', 'E'}

void labirinto(vector<char> caminho) {
    if (encontrouSaida(caminho)) {
        imprimeCaminho(caminho);
        return;
    }
    for(char c : movimentosPossiveis){
        if (deslocamentoPossivel(caminho, c)) {
            caminho.push_back(c);
            labirinto(caminho);
            caminho.pop_back();
        }
    }
}
```

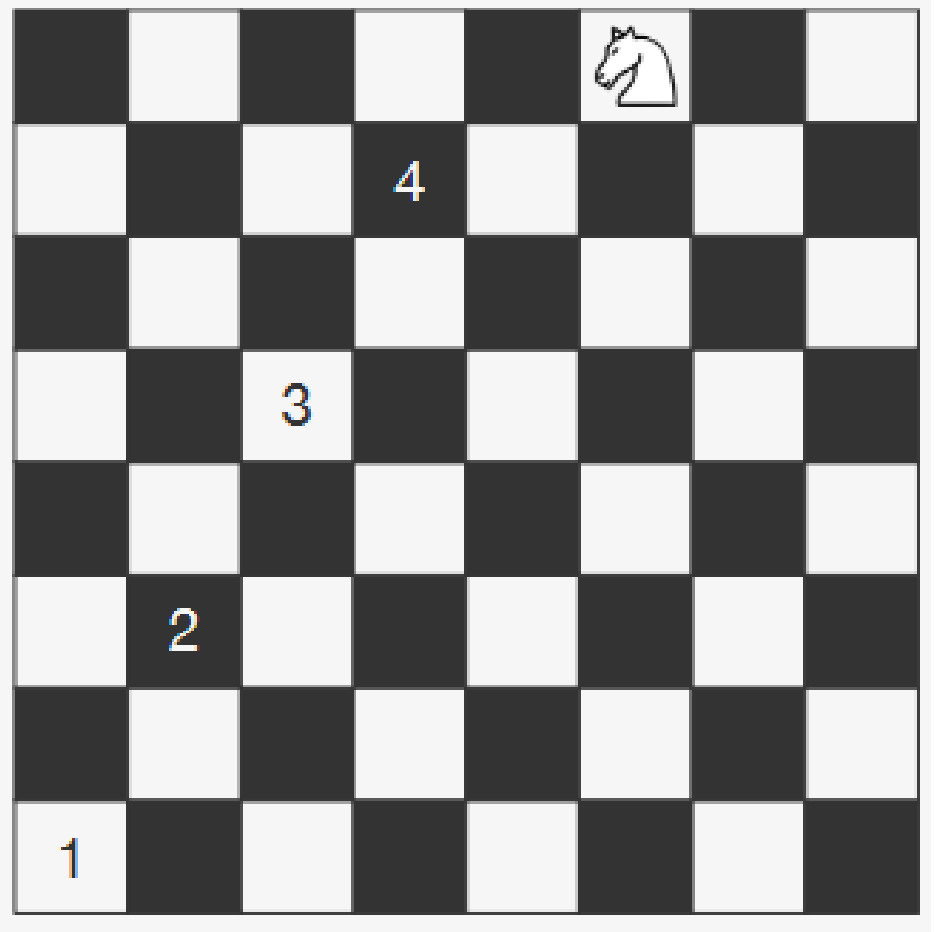
Backtracking – Passeio do Cavalo

- Dado um tabuleiro de xadrez $n \, x \, n$ e uma posição (x, y) do tabuleiro, queremos encontrar um passeio de um cavalo que visite cada casa exatamente uma vez.
- Movimento do cavalo – formato de L:
  - dois quadrados horizontalmente e um verticalmente, ou
  - dois quadrados verticalmente e um horizontalmente.



Backtracking – Passeio do Cavalo

- Vamos representar o tabuleiro em uma matriz **m**, de forma que:
  - m[l][c] = 0, se a posição (l, c) ainda não foi visitada
  - m[l][c] = i > 0, se a posição (l, c) foi visitada no passo i



Backtracking – Passeio do Cavalo

- Vamos representar o tabuleiro em uma matriz **m**, de forma que:
  - m[l][c] = 0, se a posição (l, c) ainda não foi visitada
  - m[l][c] = i > 0, se a posição (l, c) foi visitada no passo i

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

Backtracking – Passeio do Cavalo

```
int m[MAX][MAX], n;
vector<pair<int, int>> movimentos = {{2, 1}, {1, 2}, {-1, 2},
                                     {-2, 1}, {-2, -1}, {-1, -2},
                                     {1, -2}, {2, -1}};

void init(int x, int y){
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            m[i][j] = 0;
    m[x][y] = 1;
}

bool posicaoValida(int x, int y){
    return (x >= 0) && (x < n) && (y >= 0) && (y < n) && !m[x][y];
}
```

Backtracking – Passeio do Cavalo

```
int passeioCavalo(int x, int y){
    if (m[x][y] == n * n)
        return 1;
    for(auto mov : movimentos){
        int x2 = x + mov.first;
        int y2 = y + mov.second;
        if (posicaoValida(x2, y2)){
            m[x2][y2] = m[x][y] + 1;
            if (passeioCavalo(x2, y2))
                return 1;
            m[x2][y2] = 0;
        }
    }
    return 0;
}
```

Backtracking - Melhorias

- Em geral, este paradigma é mais rápido que a Força Bruta, pois eliminamos vários candidatos a solução de uma só vez.
- Porém, ele ainda pode ser lento para problemas onde temos muitas soluções parciais possíveis.
- Dicas para tentar melhorar o desempenho do algoritmo:
  - Se possível, buscar explorar ramos que pareçam mais vantajosos, com mais chance de encontrar uma solução em menos passos.
  - Se houver várias soluções e estivermos buscando a melhor possível (valor mínimo), guardar o valor da melhor solução até o momento e usá-lo para podar soluções parciais que, ainda que sejam válidas, já são piores que a melhor solução encontrada até o momento.

Backtracking – Problema do Troco

- **Problema:** dar troco de um valor **x** com o menor número de moedas possíveis.

Backtracking – Problema do Troco

- **Problema:** dar troco de um valor **x** com o menor número de moedas possíveis.
- **Solução por força bruta:** testar todas as combinações de moedas possíveis.

Backtracking – Problema do Troco

- **Problema:** dar troco de um valor **x** com o menor número de moedas possíveis.
- **Solução por *backtracking*:** refinamento da força bruta. Serão testadas diversas combinações de moedas, até encontrar uma solução que resulte em x.

Backtracking – Problema do Troco

- **Problema:** dar troco de um valor **x** com o menor número de moedas possíveis.
- **Solução por *backtracking*:** refinamento da força bruta. Serão testadas diversas combinações de moedas, até encontrar uma solução que resulte em x. Além disso, esse método admite algumas melhorias:
  - Podemos tentar adicionar na solução parcial sempre a moeda de maior valor possível. Com isso caminharemos mais rapidamente para a solução que usa o menor número de moedas.
  - Já tendo encontrado uma solução que utiliza **k** moedas, qualquer solução parcial que utilize mais que **k** moedas pode ser cortada.

Backtracking - Problema do Troco

```
vector<int> moedas = {50, 25, 10, 5, 1};
vector<int> melhorSolucao;
int qtdeMoedas = INT_MAX;
```

Backtracking - Problema do Troco

```
void troco(int x, vector<int> &solucaoAtual){
    if (solucaoAtual.size() >= qtdeMoedas)
        return;
    if (x == 0){
        melhorSolucao = solucaoAtual;
        qtdeMoedas = solucaoAtual.size();
    }
    for(int m : moedas){
        if (m > x)
            continue;
        solucaoAtual.push_back(m);
        troco(x-m, solucaoAtual);
        solucaoAtual.pop_back();
    }
}
```

Referências

Aulas de Técnicas de Programação do Prof. Dr. Rene Pegoraro.

AREFIN, Ahmed Shamsul. **Art of Programming Contest**.

http://wiki.icmc.usp.br/images/c/c3/Aula_16_-_Paradigmas_de_projeto_de_algoritmos.pdf

https://homepages.dcc.ufmg.br/~loureiro/alg/071/paa_Paradigmas_2pp.pdf

https://www.ic.unicamp.br/~zanoni/teaching/mc102/2013-2s/aulas/aula21.pdf

http://www3.decom.ufop.br/toffolo/site_media/uploads/2011-1/bcc402/slides/10._backtracking.pdf

https://www2.dc.ufscar.br/~mario/ensino/2019s1/aed1/aula25_slides.pdf

https://www.ic.unicamp.br/~rafael/cursos/1s2017/mc202/slides/unidade22-backtracking.pdf