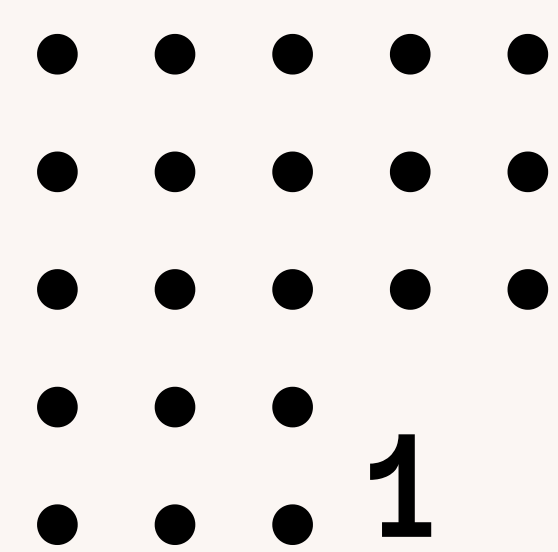
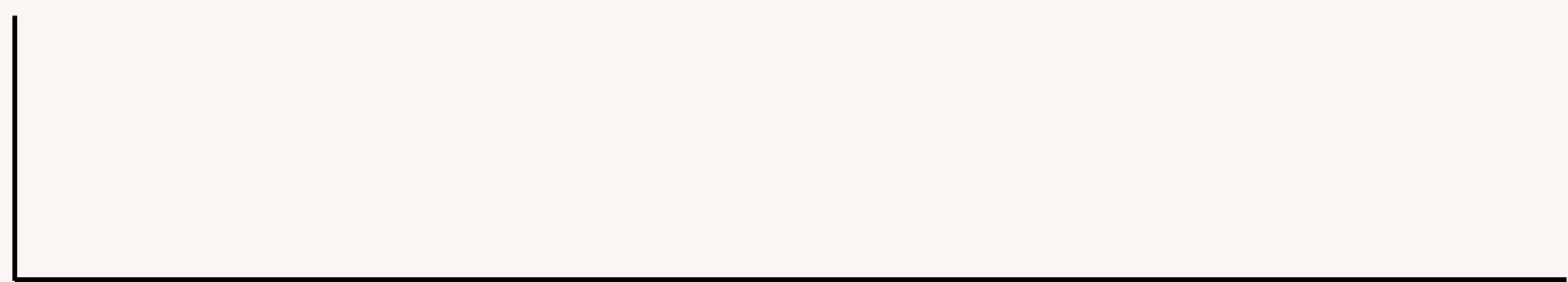




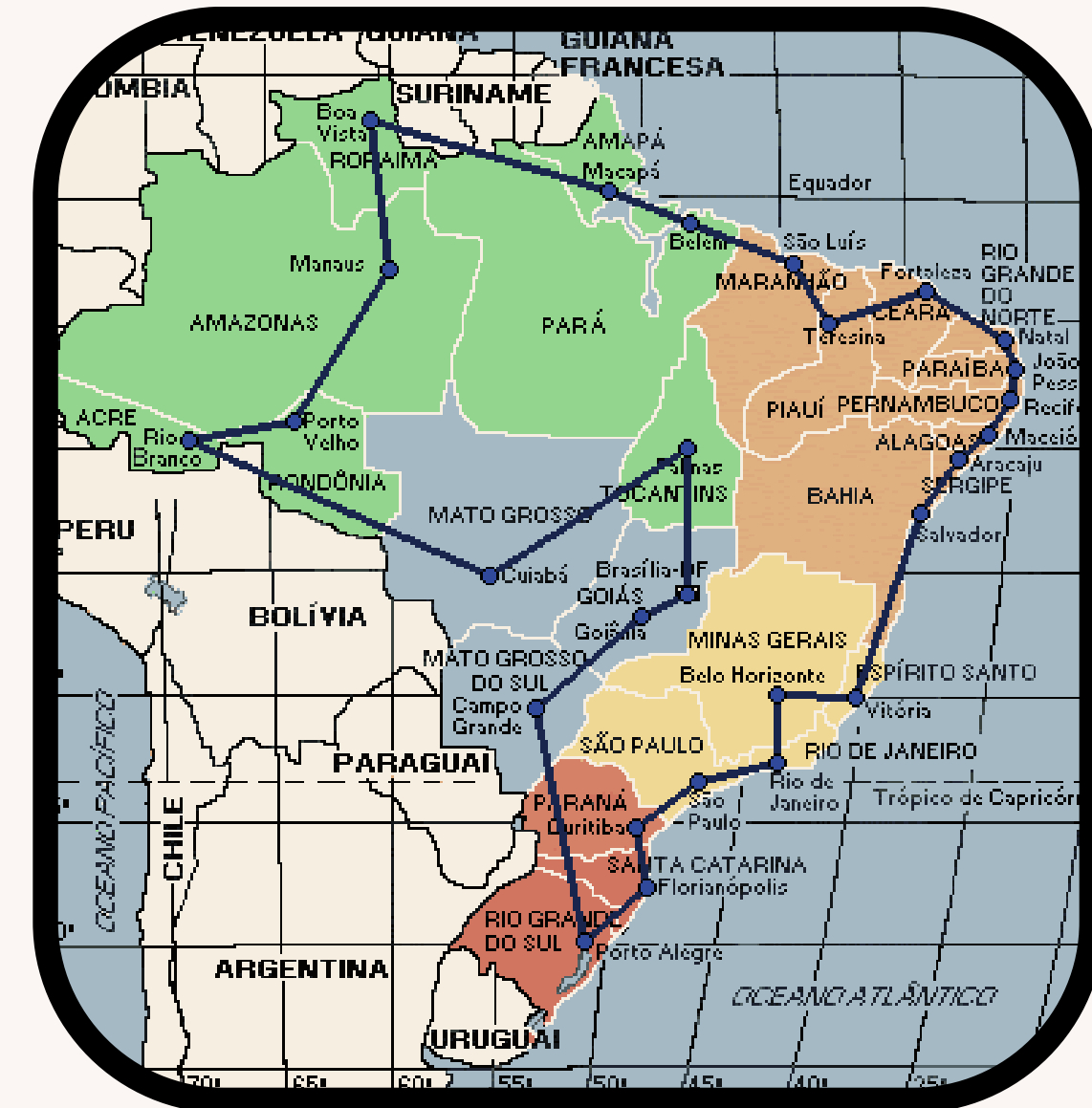
PROBLEMA DO CAIXEIRO VIAJANTE

Gilberto Alexsandro, Guilherme Miranda e Thiago Camara



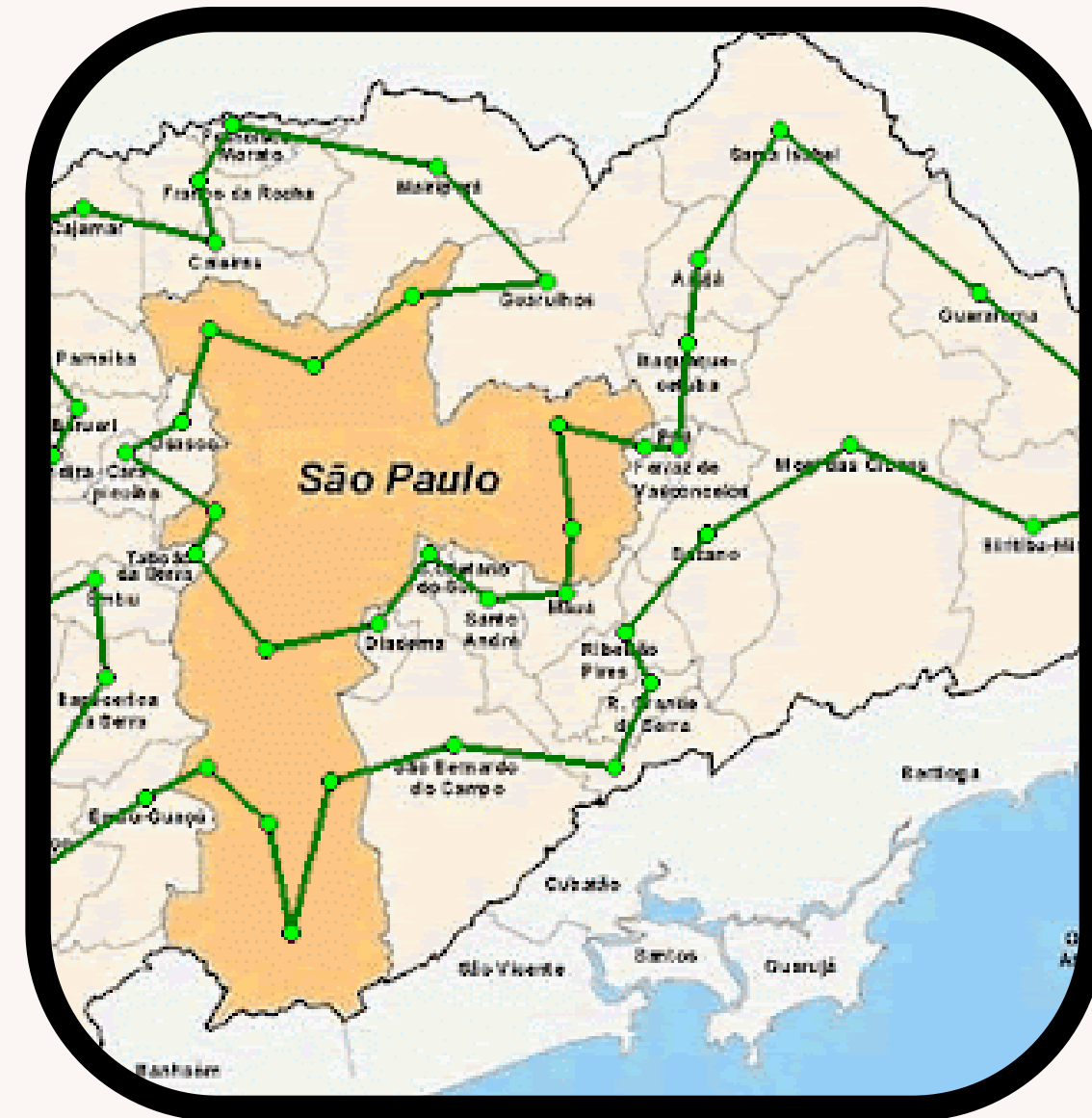
DEFININDO O PROBLEMA

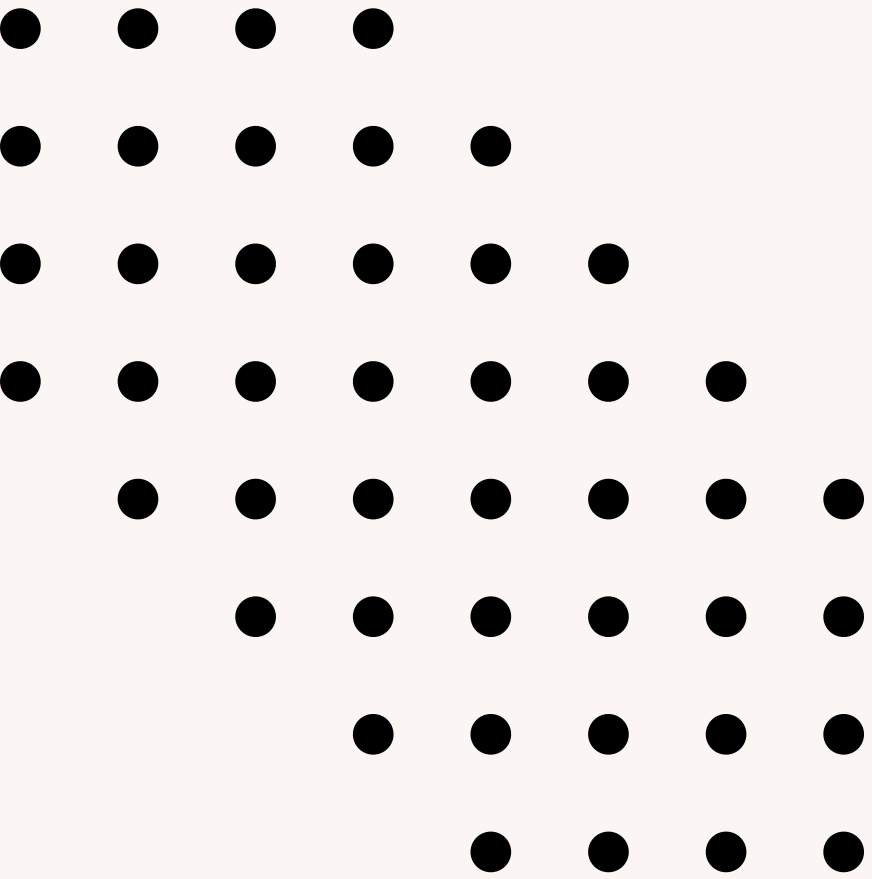
Imagine um vendedor que precisa visitar várias cidades e voltar para o ponto de partida, percorrendo a menor distância possível. O Problema do Caixeiro Viajante consiste em encontrar o caminho mais curto em um grafo completo, passando por todos os vértices exatamente uma vez e retornando ao vértice de origem. A distância entre cada par de vértices é conhecida.



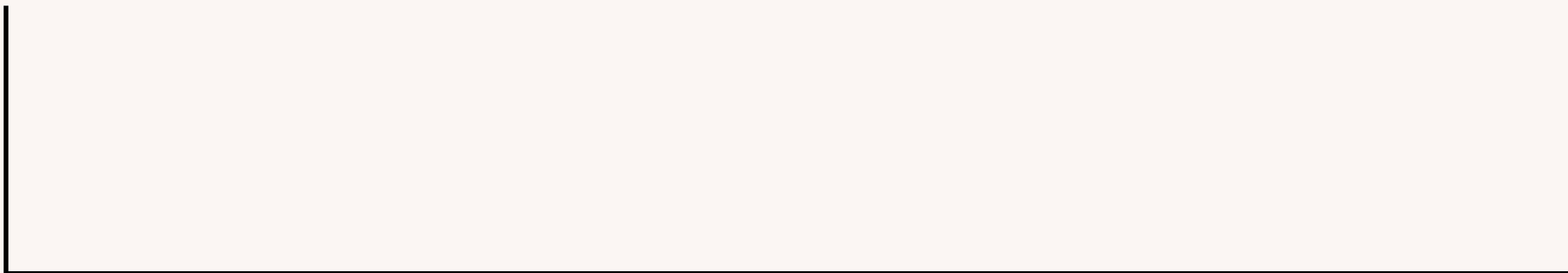
DEFININDO O PROBLEMA

O Problema do Caixeiro Viajante é um problema NP-completo: ele não tem nenhum algoritmo eficiente conhecido. Contudo, adotando-se certas premissas, há algoritmos eficientes que fornecem uma distância total não muito acima da menor possível.





Algoritmo de Força Bruta



```

1 #include <limits.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #define V 14
7 #define INF 10000000000 // Valor de custo inviável
8
9 // Função para calcular o custo total de um caminho
10 long long calcularCusto(int path[], long long grafo[V][V]) {
11     long long custoTotal = 0;
12     for (int i = 0; i < V - 1; i++) {
13         custoTotal += grafo[path[i]][path[i + 1]];
14     }
15     custoTotal += grafo[path[V - 1]][path[0]]; // Retorno à cidade inicial
16     return custoTotal;
17 }
18
19 // Função para gerar permutações e encontrar o menor custo
20 void tsp(int pos, int path[], int visited[], long long grafo[V][V],
21         long long *minCost) {
22     if (pos == V) {
23         long long custoAtual = calcularCusto(path, grafo);
24         if (custoAtual < *minCost) {
25             *minCost = custoAtual;
26         }
27         return;
28     }
29
30     for (int i = 0; i < V; i++) {
31         if (!visited[i]) {
32             visited[i] = 1;
33             path[pos] = i;
34             tsp(pos + 1, path, visited, grafo, minCost);
35             visited[i] = 0;
36         }
37     }
38 }
39

```

```

40 int main() { // Definindo um grafo de adjacência estático para 12 cidades
41     long long grafo[V][V] = {
42         {0, 20, 18, 25, 12, 22, 15, 28, 19, 24, 11, 26, 17, 21},
43         {20, 0, 23, 17, 21, 19, 26, 13, 25, 18, 22, 15, 24, 27},
44         {18, 23, 0, 20, 16, 24, 11, 27, 22, 19, 25, 17, 21, 26},
45         {25, 17, 20, 0, 19, 22, 28, 15, 24, 21, 18, 23, 12, 26},
46         {12, 21, 16, 19, 0, 14, 25, 20, 18, 24, 22, 17, 26, 19},
47         {22, 19, 24, 22, 14, 0, 17, 26, 21, 20, 18, 25, 13, 24},
48         {15, 26, 11, 28, 25, 17, 0, 20, 22, 19, 24, 21, 18, 23},
49         {28, 13, 27, 15, 20, 26, 20, 0, 24, 22, 19, 25, 17, 21},
50         {19, 25, 22, 24, 18, 21, 22, 24, 0, 20, 17, 26, 15, 23},
51         {24, 18, 19, 21, 24, 20, 24, 22, 20, 0, 23, 17, 26, 19},
52         {11, 22, 25, 18, 22, 18, 24, 19, 17, 23, 0, 21, 26, 20},
53         {26, 15, 17, 23, 26, 25, 21, 25, 26, 17, 21, 0, 19, 24},
54         {17, 24, 21, 12, 26, 19, 18, 17, 26, 26, 21, 19, 0, 23},
55         {21, 27, 26, 26, 19, 24, 23, 21, 23, 19, 20, 24, 23, 0}};
56
57     int path[V]; // Array para armazenar a sequência das cidades
58     int visited[V] = {0}; // Array para controlar as cidades visitadas
59     long long minCost = INF; // Inicializando o custo mínimo como INF
60
61     visited[0] = 1; // Começando pela cidade 0
62     path[0] = 0;
63
64     struct timespec start_time, end_time;
65
66     clock_gettime(CLOCK_MONOTONIC, &start_time);
67
68     tsp(1, path, visited, grafo, &minCost);
69
70     clock_gettime(CLOCK_MONOTONIC, &end_time);
71
72     double execution_time = (end_time.tv_sec - start_time.tv_sec) +
73         (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
74
75     printf("Custo mínimo encontrado: %lld\n", minCost);
76     printf("Tempo de execução: %f segundos\n", execution_time);
77
78     return 0;
79 }

```

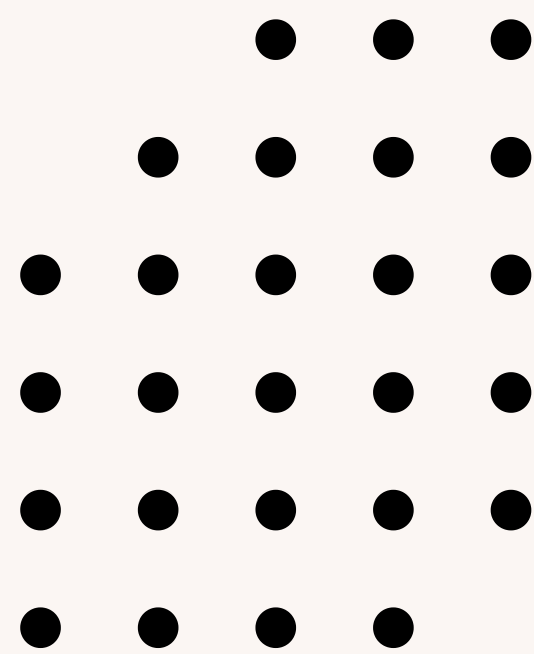
COMPLEXIDADE E CUSTO

O método do força bruta consiste em testar todas as permutações possíveis das cidades e fornece ao final uma solução exata, porém é um método não muito bom para um grande conjunto de cidades.

A complexidade é $O(V-1!)$, pois na função “tsp” do código a variável “v” é chamada recursivamente, onde “v” é o número de cidades. Isso quer dizer que com o aumento no numero de cidades, o tempo de execução cresce exponencialmente.

Além disso, possui uma complexidade temporal de $O(V)$, pois itera sobre as V cidades para calcular o custo total do caminho.

RESULTADOS



5 Cidades

- Custo mínimo encontrado: 102
- Tempo de execução: 0.000003 segundos

8 Cidades

- Custo mínimo encontrado: 104
- Tempo de execução: 0.000169 segundos

12 Cidades

- Custo mínimo encontrado: 164
- Tempo de execução: 2.392850 segundos

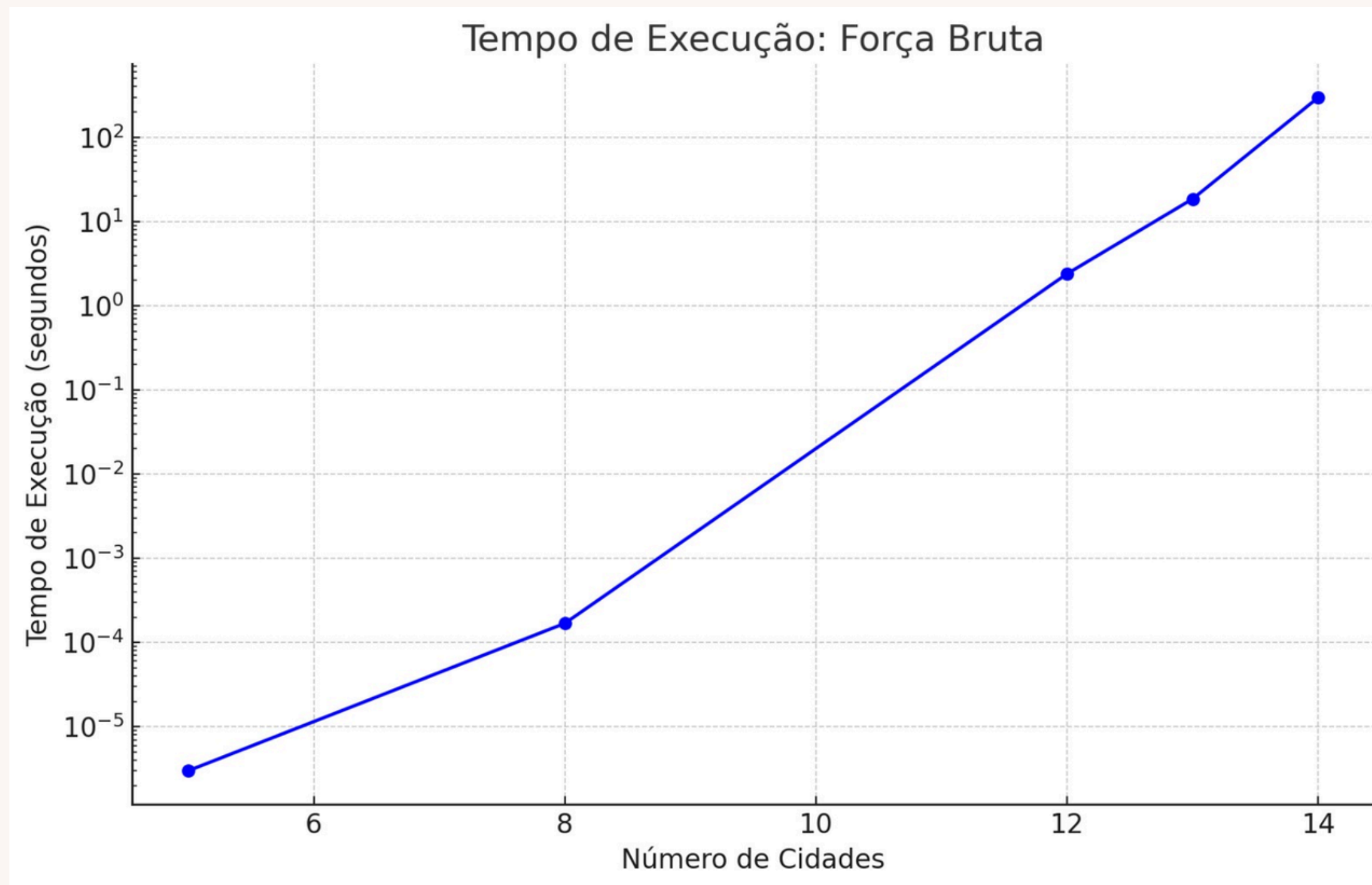
13 Cidades

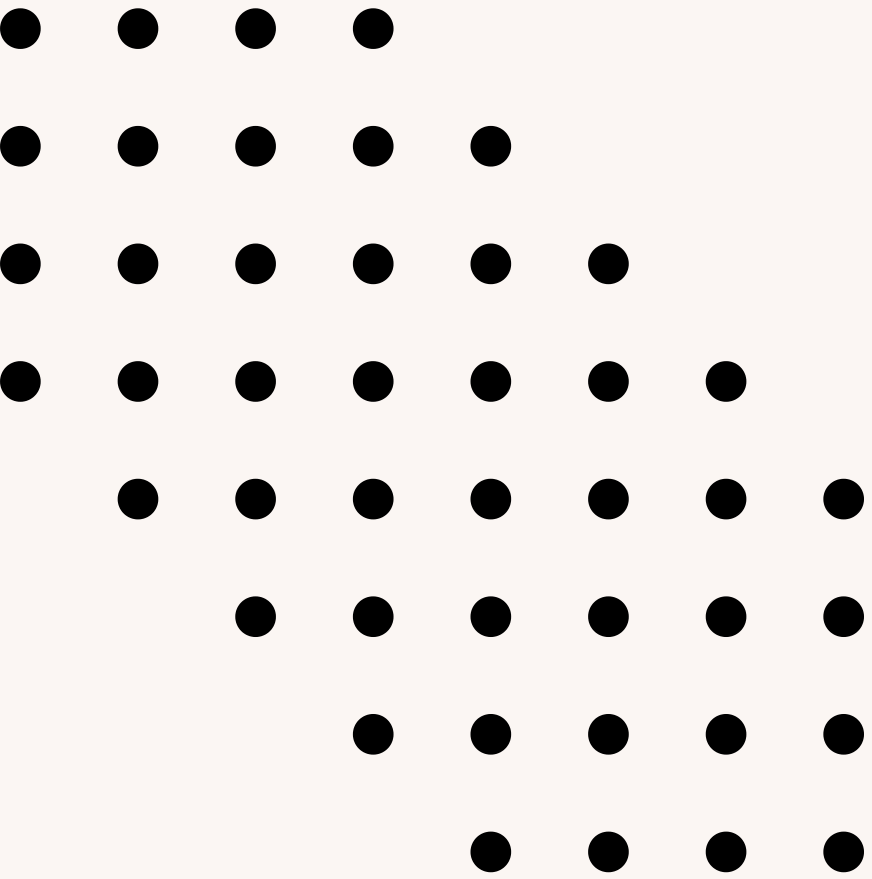
- Custo mínimo encontrado: 188
- Tempo de execução: 18.539735 segundos

14 Cidades

- Custo mínimo encontrado: 211
- Tempo de execução: 296.402887 segundos

RESULTADOS





Algoritmo de Vizinho Mais Próximo

```

1  #include <limits.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  #define V 5
7  #define INF 1000000000 // Valor de custo inviável
8
9  // Função para encontrar o vizinho mais próximo
10 int vizinhoMaisProximo(int grafo[V][V], int start, int *caminho) {
11     int visitado[V] = {0};
12     int custoTotal = 0;
13     int atual = start;
14
15     visitado[start] = 1;
16     caminho[0] = start;
17
18     for (int i = 1; i < V; i++) {
19         int proxCidade = -1;
20         int menorDist = INF;
21         // Inicializa com INF para garantir que encontramos um mínimo válido
22
23         for (int j = 0; j < V; j++) {
24             // Verifica se a cidade não foi visitada e se o caminho não é inviável (INF)
25             if (!visitado[j] && grafo[atual][j] < menorDist && grafo[atual][j] != INF) {
26                 menorDist = grafo[atual][j];
27                 proxCidade = j;
28             }
29         }
30
31         // Se não encontrou um caminho viável, o grafo pode estar desconectado
32         if (proxCidade == -1) {
33             fprintf(stderr,
34                 "Erro: Não foi possível encontrar uma cidade vizinha
35                 não visitada a partir da cidade %d.\n", atual);
36             exit(EXIT_FAILURE);
37         }
38
39         visitado[proxCidade] = 1;
40         caminho[i] = proxCidade;
41         custoTotal += menorDist;
42         atual = proxCidade;
43     }
44
45     caminho[V] = start;
46     // Retorno à cidade inicial
47     custoTotal += grafo[atual][start];
48     // Adiciona o custo de voltar à cidade inicial
49
50     return custoTotal;
51 }
52
53 int main() {
54     int grafo[V][V] = {
55         {0, 10, 20, 30, 25},
56         {10, 0, 15, 35, 20},
57         {20, 15, 0, 30, 28},
58         {30, 35, 30, 0, 22},
59         {25, 20, 28, 22, 0}
60     };
61
62     // Captura o tempo de início
63     clock_t start_time = clock();
64
65     int caminho[V + 1];
66     int custoInicial = vizinhoMaisProximo(grafo, 0, caminho);
67
68     // Captura o tempo de término
69     clock_t end_time = clock();
70
71     // Calcula o tempo de execução em segundos
72     double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
73
74     printf("Custo inicial (Vizinho Mais Próximo): %d\n", custoInicial);
75     printf("Caminho inicial: ");
76     for (int i = 0; i <= V; i++) {
77         printf("%d ", caminho[i]);
78     }
79     printf("\n");
80
81     // Exibe o tempo de execução
82     printf("Tempo de execução: %f segundos\n", execution_time);
83
84     return 0;
85 }

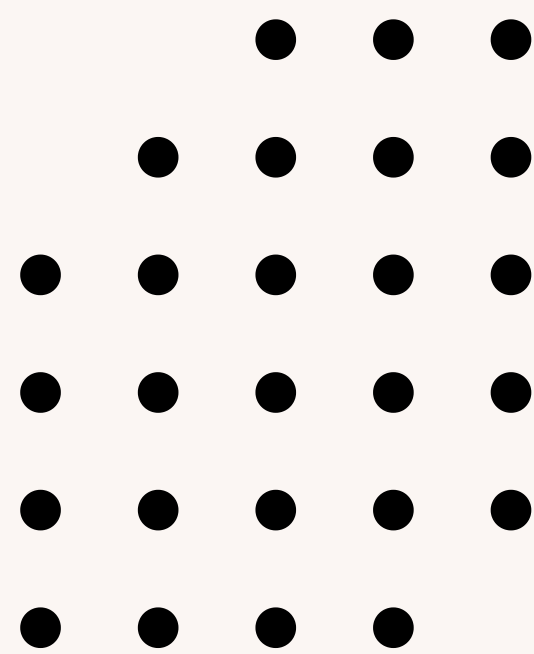
```

COMPLEXIDADE E CUSTO

O método do vizinho mais próximo (nearest neighbour) consiste em sempre se movimentar para a cidade mais próxima que ainda não foi visitada. E mesmo que não entregue uma solução exata, ele consegue entregar uma resposta aproximada e também pode trabalhar com um maior conjunto de cidades.

A complexidade é $O(V^2)$, onde V é o número de cidades. A complexidade total reflete o crescimento quadrático do número de operações à medida que o número de cidades aumenta. O tempo de execução do algoritmo cresce rapidamente, tornando-o menos eficiente para um grande número de cidades.

RESULTADOS



5 Cidades

- Custo mínimo encontrado: 105
- Caminho inicial: 0, 1, 2, 4, 3, 0
- Tempo de execução: 0.000002 segundos

8 Cidades

- Custo mínimo encontrado: 109
- Caminho inicial: 0, 7, 6, 1, 4, 5, 2, 3, 0
- Tempo de execução: 0.000002 segundos

12 Cidades

- Custo mínimo encontrado: 170
- Caminho inicial: 0, 8, 1, 10, 4, 5, 2, 7, 6, 11, 3, 9, 0
- Tempo de execução: 0.000003 segundos

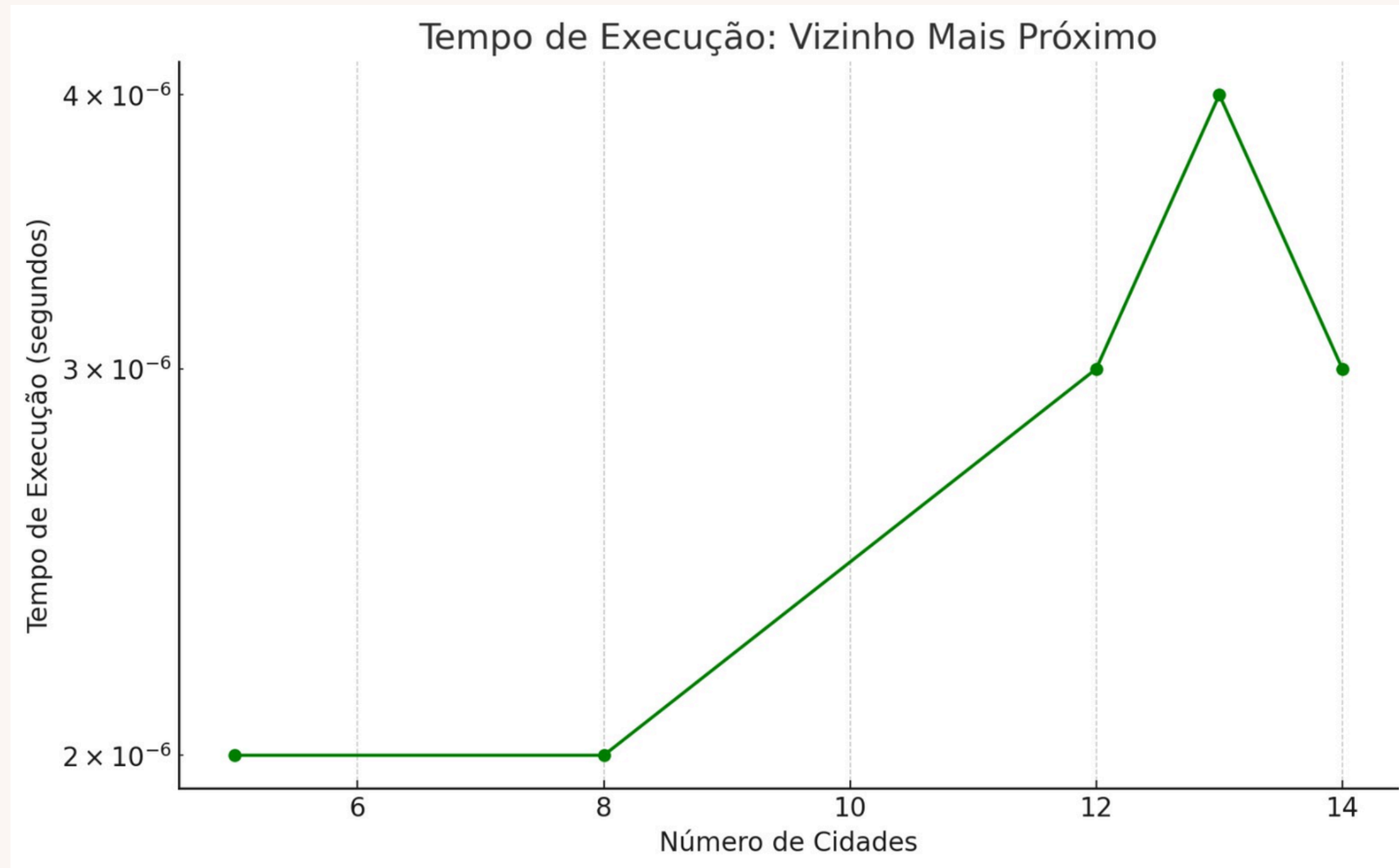
13 Cidades

- Custo mínimo encontrado: 205
- Caminho inicial: 0, 10, 8, 12, 3, 7, 1, 11, 2, 6, 5, 4, 9, 0
- Tempo de execução: 0.000004 segundos

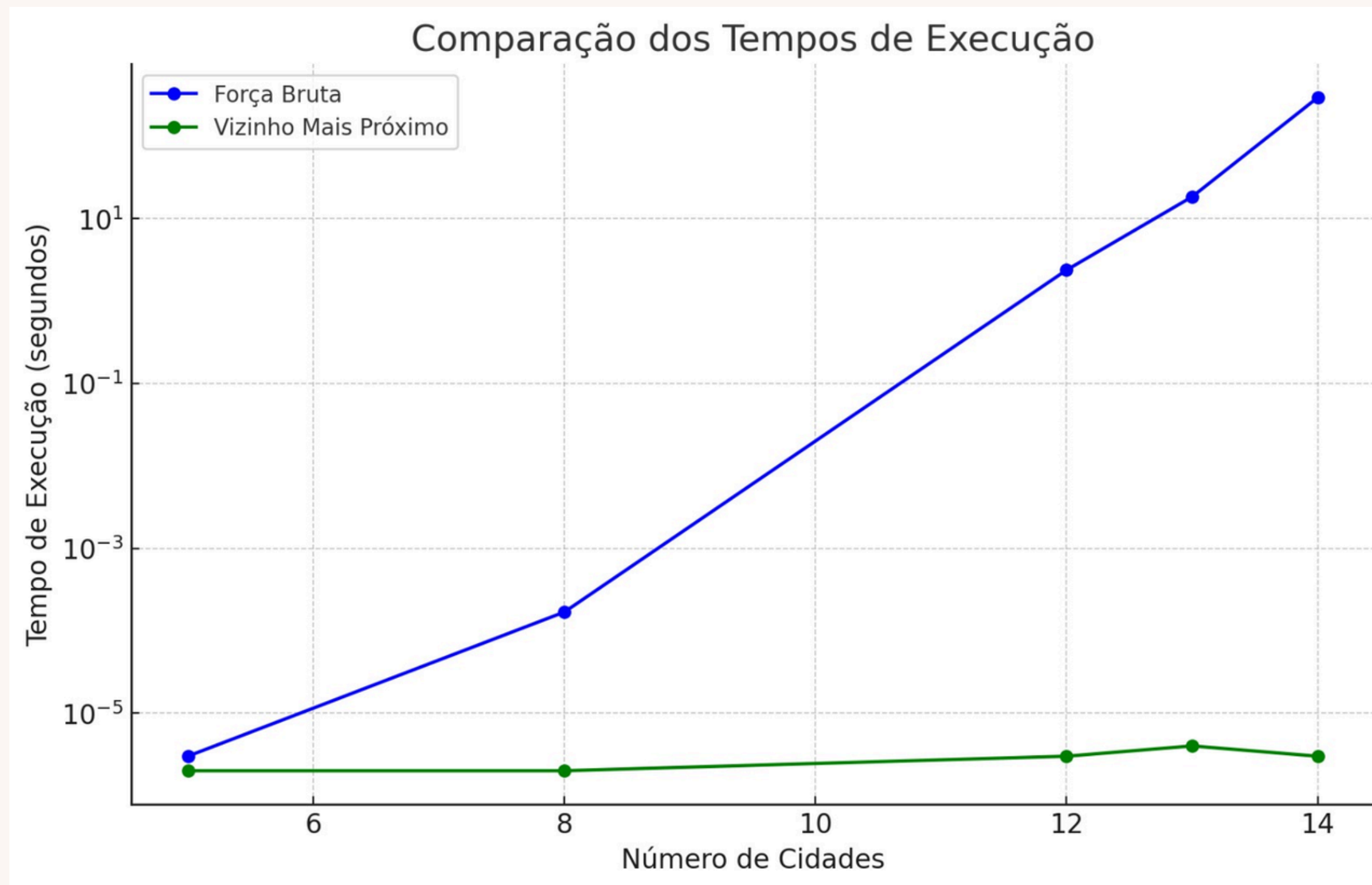
14 Cidades

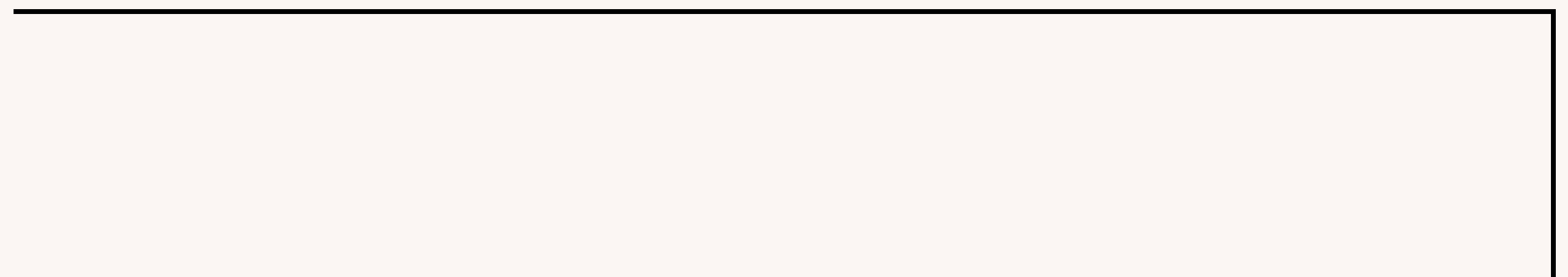
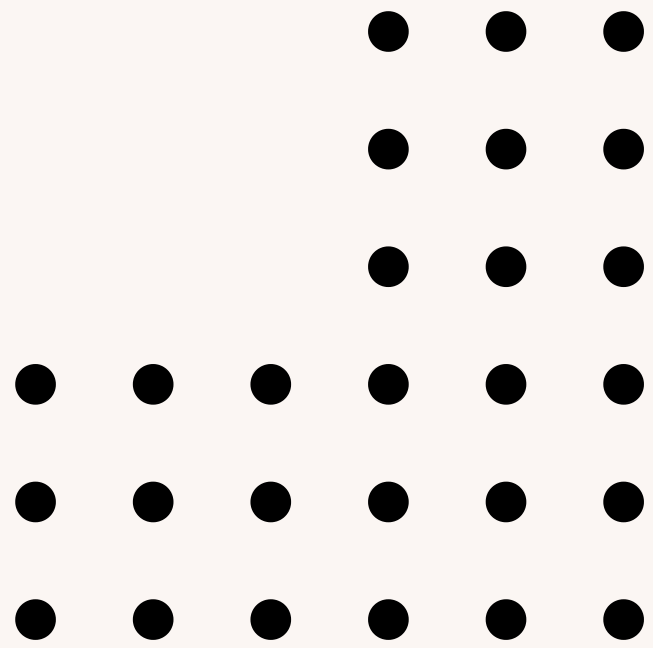
- Custo mínimo encontrado: 219
- Caminho inicial: 0, 10, 8, 12, 3, 7, 1, 11, 2, 6, 5, 4, 13, 9, 0
- Tempo de execução: 0.000003 segundos

RESULTADOS



RESULTADOS





OBRIGADO PELA ATENÇÃO!

