Otimizando Vendas de Energia

Explore as estratégias para maximizar os lucros na venda de energia elétrica por meio de leilões.

Integrantes do Grupo:

- Emmanuel Viglioni
- Lucas Machado de Oliveira Andrade
- Marcelo Aguilar Araújo D'Almeida
- Paulo Victor Pimenta Rubinger

Backtracking: Solução Exaustiva

1

Identificar Lances

Analisar todos os lances recebidos no leilão.

2

Avaliar Combinações

Testar diferentes combinações de lances para encontrar a solução ótima.

3

Poda de Soluções

Aplicar uma estratégia de poda para descartar soluções não promissoras.

```
private static void backtrack(int capacidade, List<Oferta> ofertas, int indice, List<Oferta> selecionadas,
                              int valorAtual, Resultado melhorResultado) {
    // Atualiza o melhor resultado se o valor atual for maior
    if (valorAtual > melhorResultado.getValorMaximo()) {
        melhorResultado.setValorMaximo(valorAtual);
        melhorResultado.setOfertasSelecionadas(new ArrayList<>(selecionadas));
    }

    // Loop para considerar cada oferta a partir do indice atual
    for (int i = indice; i < ofertas.size(); i++) {
        Oferta oferta = ofertas.get(i);
        // Verifica se a oferta atual pode ser considerada (poda)
        if (oferta.getMegawatts() <= capacidade) {
            // Adiciona a oferta atual à lista de selecionadas e atualiza a capacidade e o valor atual
            selecionadas.add(oferta);
            capacidade -= oferta.getMegawatts();
            valorAtual += oferta.getValor();

            // Continua a busca recursivamente
            backtrack(capacidade, ofertas, indice: i + 1, selecionadas, valorAtual, melhorResultado);

            // Remove a oferta atual para explorar outras combinações
            selecionadas.remove( index: selecionadas.size() - 1);
            capacidade += oferta.getMegawatts();
            valorAtual -= oferta.getValor();
        }
    }
}
```

Algoritmo Guloso: Soluções Rápidas

Estratégia 1

Ordenação pelo maior valor da oferta, de forma decrescente

Estratégia 2

Ordenação pelo maior valor do megawatt (V/M), de forma decrescente

Comparação

Avaliar o desempenho das diferentes estratégias gulosas.

Algoritmo Guloso

```java
public class AlgoritmoGuloso1 {  2 usages  marceloaaguilar +1

    public static Resultado calcular(int capacidade, List<Oferta> ofertas) {  1 usage  marceloaaguilar +1

        // Iniciando valor total
        int valorTotal = 0;

        // Iniciando arraylist de ofertas
        ArrayList<Oferta> ofertasSelecionadas = new ArrayList<~>();

        // início tempo de execução
        long inicio = System.nanoTime();

        // Ordenar os lances por valor total (V) em ordem decrescente
        Collections.sort(ofertas, new Comparator<Oferta>() {  marceloaaguilar
            @Override  marceloaaguilar
            public int compare(Oferta o1, Oferta o2) { return o2.getValor() - o1.getValor(); }
        });

        // For para percorrer todas as ofertas ordenadas e adicionar ao nosso ArrayList
        for (Oferta oferta : ofertas){
            if (oferta.getMegawatts() <= capacidade){
                ofertasSelecionadas.add(oferta);
                capacidade -= oferta.getMegawatts();
                valorTotal += oferta.getValor();
            }
        }

        // fim tempo de execução
        long fim = System.nanoTime();
        // calculo tempo de execução
        long tempoExecucao = fim - inicio;

        // Retorna resultado final com todos os dados
        return new Resultado(valorTotal, ofertasSelecionadas, tempoExecucao);
    }
}
```

```java
public class AlgoritmoGuloso2 {  2 usages  Marcelo D'almeida

    public static Resultado calcular(int capacidade, List<Oferta> ofertas) {  1 usage  Marcelo D'almeida

        // Iniciando valor total
        int valorTotal = 0;

        // Iniciando arraylist de ofertas
        ArrayList<Oferta> ofertasSelecionadas = new ArrayList<~>();

        // início tempo de execução
        long inicio = System.nanoTime();

        // Ordenar os lances por valor por megawatt (V/K) em ordem decrescente
        Collections.sort(ofertas, new Comparator<Oferta>() {  Marcelo D'almeida
            @Override  Marcelo D'almeida
            public int compare(Oferta o1, Oferta o2) {
                double v1 = (double) o1.getValor() /o1.getMegawatts();
                double v2 = (double) o2.getValor() / o2.getMegawatts();
                return Double.compare(v2, v1);
            }
        });

        // For para percorrer todas as ofertas ordenadas e adicionar ao nosso ArrayList
        for (Oferta oferta : ofertas){
            if (oferta.getMegawatts() <= capacidade){
                ofertasSelecionadas.add(oferta);
                capacidade -= oferta.getMegawatts();
                valorTotal += oferta.getValor();
            }
        }

        // fim tempo de execução
        long fim = System.nanoTime();
        // calculo tempo de execução
        long tempoExecucao = fim - inicio;

        // Retorna resultado final com todos os dados
        return new Resultado(valorTotal, ofertasSelecionadas, tempoExecucao);
    }
```

Divisão e Conquista: Abordagem Recursiva

1

Ordenação

Ordenação inicial das ofertas pelo valor por megawatt (de forma decrescente)

2

Dividir

Separar o problema em subproblemas menores.

3

Memoização

Verificar se já existe uma solução para o subproblema

4

Resolver

Resolver cada subproblema de forma independente.

5

Combinar

Reunir as soluções dos subproblemas para obter a solução final.

Divisão e Conquista

```java
private static int calcularRecursivo(int capacidadeRestante, List<Oferta> ofertas, int indice,  3 usages  Emmanuel *
                                     List<Oferta> ofertasSelecionadas, Map<String, ResultadoParcial> memo) {
    // Caso base: sem capacidade restante ou sem ofertas
    if (capacidadeRestante <= 0 || indice >= ofertas.size()) {
        return 0;
    }

    // Chave para memoização
    String chave = capacidadeRestante + "-" + indice;

    // Verificar se já temos o resultado memoizado
    if (memo.containsKey(chave)) {
        ResultadoParcial resultadoParcial = memo.get(chave);
        ofertasSelecionadas.clear();
        ofertasSelecionadas.addAll(resultadoParcial.ofertasSelecionadas);
        return resultadoParcial.valor;
    }

    // Obter a oferta atual
    Oferta ofertaAtual = ofertas.get(indice);

    // Listas para armazenar as ofertas selecionadas em ambos os cenários
    List<Oferta> ofertasComAtual = new ArrayList<>(ofertasSelecionadas);
    int valorComAtual = 0;
    // Caso a oferta atual possa ser incluída (não ultrapassa a capacidade restante)
    if (ofertaAtual.getMegawatts() <= capacidadeRestante) {
        ofertasComAtual.add(ofertaAtual);
        valorComAtual = ofertaAtual.getValor() +
                calcularRecursivo( capacidadeRestante: capacidadeRestante - ofertaAtual.getMegawatts(),
                        ofertas,  indice: indice + 1, ofertasComAtual, memo);
    }
```

```java
    // Valor sem incluir a oferta atual
    List<Oferta> ofertasSemAtual = new ArrayList<>(ofertasSelecionadas);
    int valorSemAtual = calcularRecursivo(capacidadeRestante, ofertas,  indice: indice + 1, ofertasSemAtual, memo);

    // Selecionar a opção com maior valor
    int valorMaximo;
    List<Oferta> melhoresOfertas;
    if (valorComAtual > valorSemAtual) {
        ofertasSelecionadas.clear();
        ofertasSelecionadas.addAll(ofertasComAtual);
        valorMaximo = valorComAtual;
        melhoresOfertas = ofertasComAtual;
    } else {
        ofertasSelecionadas.clear();
        ofertasSelecionadas.addAll(ofertasSemAtual);
        valorMaximo = valorSemAtual;
        melhoresOfertas = ofertasSemAtual;
    }

    // Armazenar o resultado memoizado
    memo.put(chave, new ResultadoParcial(valorMaximo, melhoresOfertas));

    return valorMaximo;
```

Programação Dinâmica: Otimização Iterativa

Construir Tabela

Criar uma tabela para armazenar soluções parciais.

Preencher Tabela

Preencher a tabela com base em soluções anteriores.

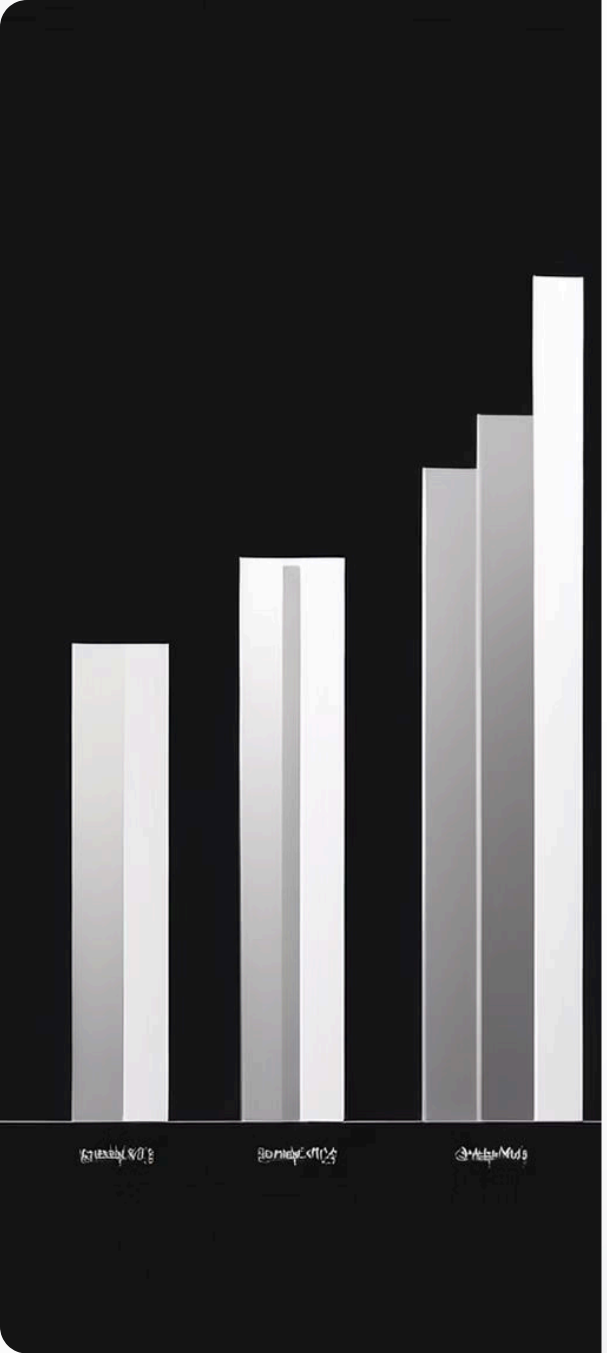
Obter Solução

Recuperar a solução ótima a partir da tabela.

Programação Dinâmica

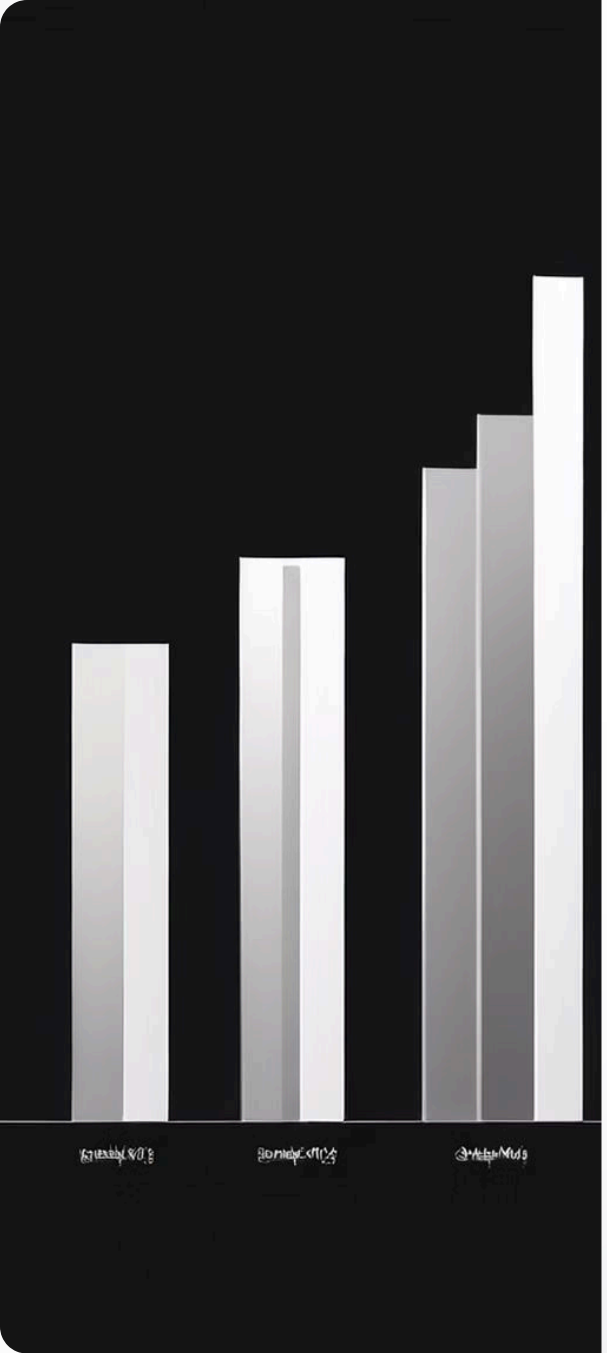
```java
private static void preencherTabela(int[][] tabela, int capacidade, List<Oferta> ofertas) {  1 usage  Andrade
    // for para percorrer cada oferta (Linha da tabela)
    for (int i = 1; i <= ofertas.size(); i++) {

        // obtem dados da linha selecionada
        Oferta oferta = ofertas.get(i - 1);
        int mw = oferta.getMegawatts();
        int valor = oferta.getValor();

        // for para percorrer cada megawatt de capacidade máxima (Coluna da tabela)
        for (int j = 0; j <= capacidade; j++) {
            /*
             * Caso a quantidade de megawatts da tabela seja superior ou igual a quantidade de megawatts da oferta
             * deve ser selecionado o maior valor considerando:
             *      O valor já existe da oferta anterior tabela[i - 1][j]
             *      O valor ao adiciona a nova oferta data a limitação de megawatts tabela[i - 1][j - mw] + valor
             * Caso a quantidade de megawatts da tabela seja inferior a quantidade de megawatts da oferta não
             * é possível inserir a nova oferta, ou seja, deve ser copiado o valor anterior
            */
            if (mw <= j) {
                tabela[i][j] = Math.max(tabela[i - 1][j], tabela[i - 1][j - mw] + valor);
            } else {
                tabela[i][j] = tabela[i - 1][j];
            }
        }
    }
}
```

```java
private static List<Oferta> buscarOfertasSelecionadas(int[][] tabela, int capacidade, List<Oferta> ofertas) {  1 usage
    List<Oferta> ofertasSelecionadas = new ArrayList<>();
    int valorTabela = tabela[ofertas.size()][capacidade];
    int colunaTabela = capacidade;

    /*
     * Inicia-se o for da último oferta e irá subindo na tabela dinâmica
     * Caso o valor encontrado na tabela for 0, significa que chegou-se na base e o for pode ser interrompido
     */
    for (int i = ofertas.size(); i > 0 && valorTabela > 0; i--) {
        /*
         * Caso tenha alteração da oferta atual com a oferta anterior da tabela, significa que a oferta faz parte
         * do resultado
        */
        if (valorTabela != tabela[i - 1][colunaTabela]) {
            // obtem a oferta e adiciona na solução
            Oferta oferta = ofertas.get(i - 1);
            ofertasSelecionadas.add(oferta);
            // valor da tabela é redefinido para retirar a influência da oferta atual, que já está na solução
            valorTabela -= oferta.getValor();
            // reduz a coluna da tabela para retirar a influência da oferta atual, que já está na solução
            colunaTabela -= oferta.getMegawatts();
        }
    }

    return ofertasSelecionadas;
}
```



Comparação de Desempenho

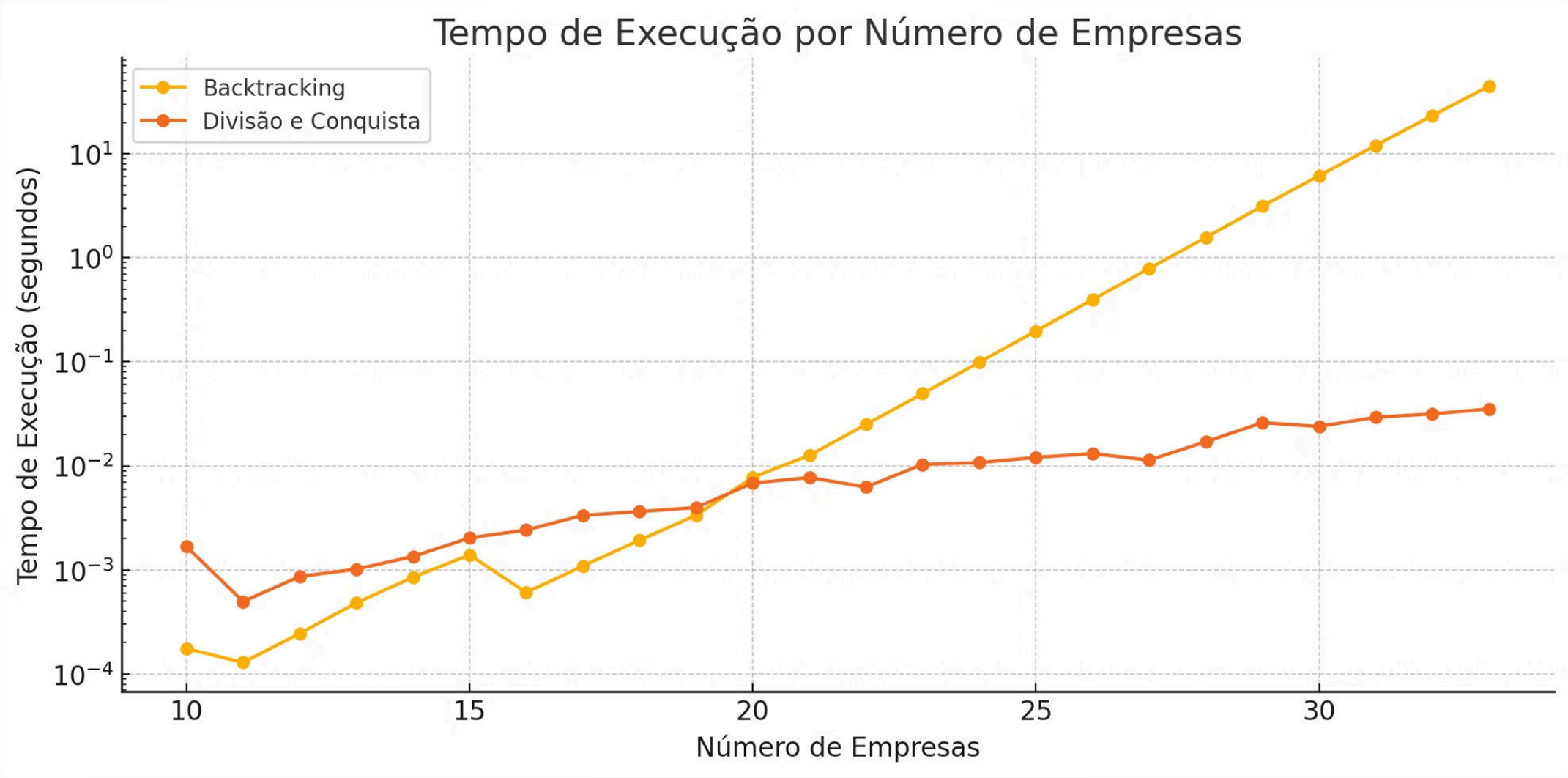
Qntd. Empresas	Algoritmo	Tempo Médio	Valor Total	Qntd. Vendida
33	Backtracking	44.56 segundos	28567 dinheiros	8000MW
33	Divisão e Conquista	43.848 ms	28567 dinheiros	8000MW
33	Algoritmo Guloso 1	0.016 ms	28317 dinheiros	7777MW
33	Algoritmo Guloso 2	0.006 ms	28197 dinheiros	7715MW
33	Programação Dinâmica	0.972 ms	28567 dinheiros	8000MW



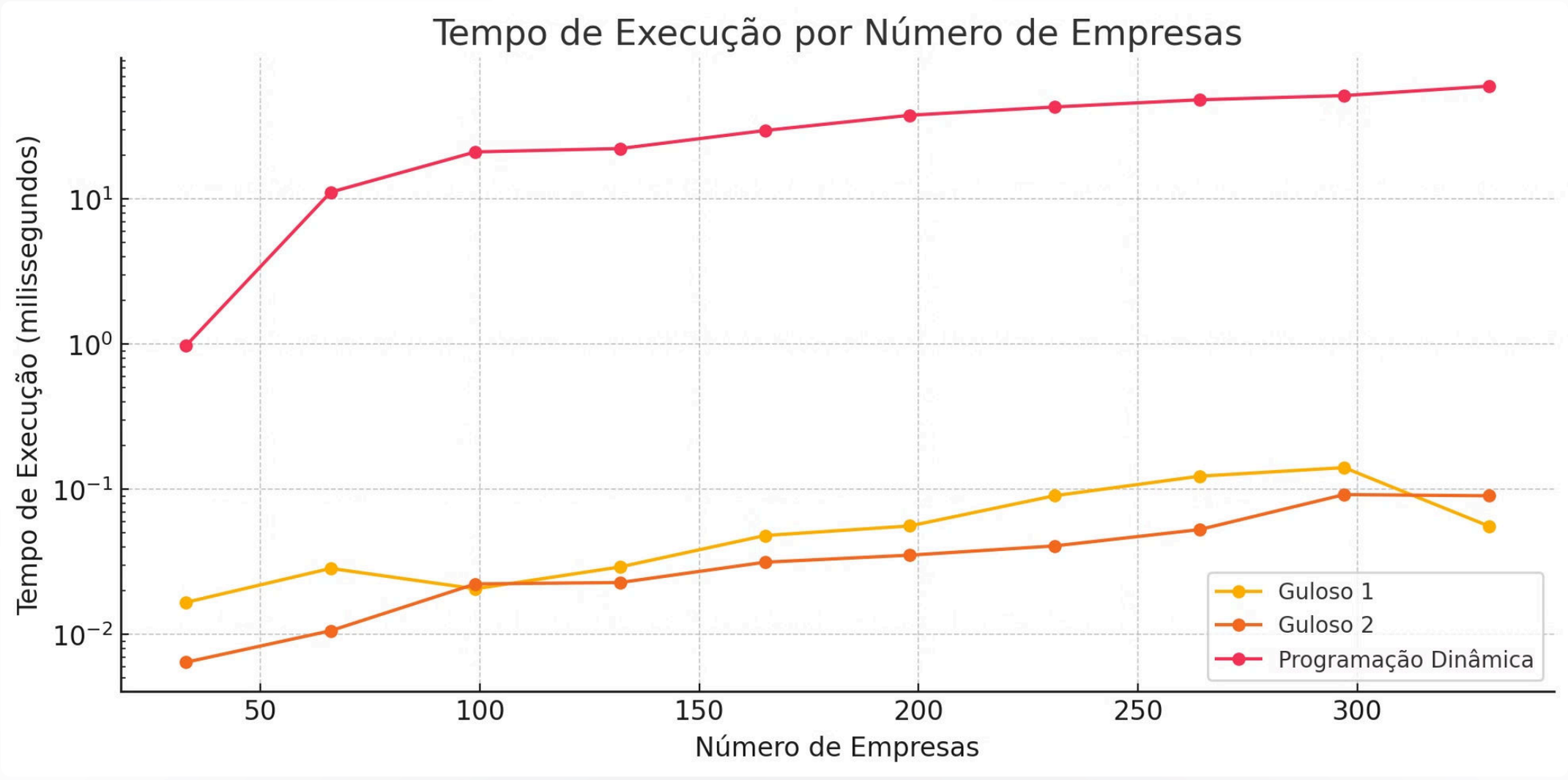
Comparação de Desempenho Massivo

Qntd. Empresas	Algoritmo	Tempo Médio
33	Backtracking	44.56 segundos
33	Divisão e Conquista	43.848 ms
330	Algoritmo Guloso 1	0.0557 ms
330	Algoritmo Guloso 2	0.090 ms
330	Programação Dinâmica	5.985 ms

Resultados



Resultados



Conclusão

1 **Tempo de Execução**

Guloso possui melhor tempo e backtracking o pior

2 **Qualidade da Solução**

Todos conseguiram resultado ótimo com exceção do guloso

3 **Escalabilidade**

Programação dinâmica possui melhor relação tempo com solução ótima