Relatório do Trabalho Prático de Leilão de Energia

Lucas Cabral Soares, Lucca Oliveira Vasconcelos, Maria Eduarda Amaral, Vitor Lagares

1. Introdução

Este relatório apresenta a análise e a resolução de um problema de otimização utilizando quatro técnicas algorítmicas distintas: backtracking, algoritmo guloso, divisão e conquista e programação dinâmica. O problema em questão envolve uma empresa produtora de energia que possui uma quantidade X de energia, medida em megawatts (MW), para vender. O objetivo da empresa é maximizar o valor total obtido nas vendas de energia através de leilões.

Nos leilões, diversas empresas interessadas em adquirir energia fazem lances para comprar um lote de K megawatts, oferecendo um valor V por esse lote. Cada empresa interessada comprará apenas um lote do tamanho exato da oferta. Dessa forma, a empresa produtora de energia deve selecionar de maneira otimizada os lances que maximizarão seu lucro total, respeitando a quantidade X de energia disponível para venda.

Neste contexto, as técnicas algorítmicas aplicadas são descritas da seguinte forma:

Backtracking: Explora todas as combinações possíveis de lances, retornando a solução ótima. Embora seja uma abordagem que garante encontrar a melhor solução, sua complexidade pode ser proibitiva para grandes valores de X e K.

Algoritmo Guloso: Baseia-se em escolher a opção localmente ótima em cada etapa com a esperança de encontrar a solução globalmente ótima. Esta técnica pode não garantir a solução ótima em todos os casos, mas é eficiente em termos de tempo de execução.

Divisão e Conquista: Divide o problema original em subproblemas menores e resolve cada um independentemente, combinando depois as soluções para obter a solução final. Esta técnica busca melhorar a eficiência ao reduzir o tamanho do problema em cada etapa.

Programação Dinâmica: Utiliza a abordagem bottom-up para resolver subproblemas sobrepostos e armazena os resultados desses subproblemas para evitar cálculos redundantes. É particularmente eficaz para problemas de otimização onde subproblemas menores se repetem.

Este relatório detalha a implementação de cada uma dessas técnicas, discute suas eficiências e limitações, e apresenta os resultados obtidos com a aplicação dos algoritmos ao problema proposto. O objetivo é oferecer uma visão abrangente das diferentes abordagens para a resolução de problemas de otimização e suas respectivas performances

2. Desenvolvimento

A resolução do problema de otimização de vendas de energia foi abordada utilizando as técnicas descritas anteriormente. Nesta seção apresenta-se os algoritmos que foram desenvolvidos para cada técnica, assim como os testes realizados para encontrar a melhor solução de cada algoritmo.

2.1 Algoritmos

2.1.1 Backtracking

O algoritmo de Backtracking implementado utiliza uma abordagem recursiva para resolver um problema de otimização onde o objetivo é maximizar o lucro obtido através de uma combinação de lances, respeitando uma restrição de energia disponível.

```java
public void resolver(List<Lance> Lances, int energia) {
    maiorLucro = 0;
    solucao = new ArrayList<>();
    backtrack(Lances, new ArrayList<>(), energia, 0, 0);
}

private void backtrack(List<Lance> Lances, List<Lance> solucaoAtual, int
energiaRestante, int lucroAtual, int comeco) {
    if (energiaRestante < 0) {
        return;
    }

    if (lucroAtual > maiorLucro) {
        maiorLucro = lucroAtual;
        solucao = new ArrayList<>(solucaoAtual);
    }

    for (int i = comeco; i < Lances.size(); i++) {
        Lance Lance = Lances.get(i);
        if (Lance.energia <= energiaRestante) {
            solucaoAtual.add(Lance);
            backtrack(Lances, solucaoAtual, energiaRestante -
Lance.energia, lucroAtual + Lance.valor, i + 1);
            solucaoAtual.remove(solucaoAtual.size() - 1);
        }
    }
}
```

Código 1 – Algoritmo implementado de backtracking

Primeiramente, o algoritmo utiliza duas variáveis principais: **`maiorLucro`** e **`solucao`**. **`maiorLucro`** é inicializado com zero para acompanhar o maior lucro encontrado até o momento, enquanto **`solucao`** armazena a lista de lances que resulta nesse maior lucro. Essas variáveis são cruciais para acompanhar e retornar a solução ótima encontrada.

A estratégia de Backtracking é central neste algoritmo. A função **`backtrack`** é recursiva e responsável por explorar todas as possíveis combinações de lances. A cada passo da recursão, um lance é adicionado à **`solucaoAtual`** se ele puder ser incorporado sem ultrapassar a energia disponível (**`energiaRestante`**). Após adicionar um lance, a função continua recursivamente para explorar todas as outras combinações possíveis, atualizando o lucro acumulado (**`lucroAtual`**) e verificando se a solução atual supera **`maiorLucro`**.

Para melhorar a eficiência, o algoritmo utiliza técnicas de poda. Em particular, antes de adicionar um lance à **`solucaoAtual`**, verifica-se se a energia necessária para este lance excede a energia restante disponível. Se isso ocorrer, a recursão para aquele caminho é interrompida, evitando a exploração de caminhos sem futuro. Além disso, sempre que uma nova solução com um lucro maior do que **`maiorLucro`** é encontrada, **`maiorLucro`** é atualizado e a lista **`solucao`** é substituída pela **`solucaoAtual`** atual. Isso garante que o algoritmo explore apenas combinações que tenham potencial para superar a melhor solução encontrada até então.

O loop **`for`** dentro de **`backtrack`** itera sobre os lances disponíveis a partir do índice `comeco`, permitindo que o algoritmo explore cada lance como parte da solução atual. Após explorar um lance, ele é removido de **`solucaoAtual`** (**`backtracking`**), permitindo a exploração de outras combinações.

Por fim, o algoritmo define condições de parada claras: a recursão termina quando todos os lances foram considerados (quando **`comeco`** ultrapassa o tamanho da lista de lances) ou quando não há energia restante suficiente para adicionar mais lances à **`solucaoAtual`**. Nesses casos, a solução atual é avaliada em relação à melhor solução encontrada até então.

Análise de Complexidade

O espaço utilizado pelo algoritmo é dominado pelo armazenamento da solução ótima, que é mantida na lista `solucao`. Como o número de elementos em `solucao` é no máximo igual ao número de elementos em `Lances`, o espaço é O(n), onde n é o número de Lances.

A complexidade de tempo do algoritmo é exponencial em relação ao número de Lances, devido ao fato de que para cada Lance, duas escolhas são consideradas (adicionar ou não adicionar). Assim, a complexidade de tempo é **O(**2^n**)**, onde n é o número de Lances. Isso ocorre porque o algoritmo explora todas as combinações possíveis de Lances que podem ser incluídos na solução, resultando em uma árvore de recursão com 2^n possíveis caminhos.

2.1.2 Algoritmo Guloso

```
public static List<Lance> resolverPorEnergiaProValor(List<Lance> lances,
int energia) {
    //Organiza os lances em ordem decrecente em relação valor/energia
    Collections.sort(lances, Comparator.comparingDouble(b -> -(double)
b.valor / b.energia));
```

```
    List<Lance> lancesSelecionados = new ArrayList<>();
    int totalValor = 0;
    for (Lance lance : lances) {
        //Se consegue colocar o lance com a enegia disponivel coloca
        if (energia >= lance.energia) {
             energia -= lance.energia;
            totalValor += lance.valor;
            lancesSelecionados.add(lance);
        }
        //Para quando tiver 10 ou menos de energia restante
        if (energia <= 10)
            break;
    }
    return lancesSelecionados;
}
```

Código 2 – Algoritmo Guloso: estratégia de valor/energia

O primeiro algoritmo guloso seleciona lances com base na relação valor/energia, priorizando lances que proporcionam o maior valor por unidade de energia. A ideia é maximizar a eficiência da energia usada.

Primeiro, os lances são ordenados em ordem decrescente de valor por unidade de energia. Em seguida, os lances são iterados na ordem ordenada, e cada lance é adicionado à solução se a energia disponível permitir. O algoritmo retorna a lista de lances quando a energia restante for 10 ou menos. Considerando que a ordenação tem uma complexidade de O (n log n), a seleção (o loop para inserir os lances) é O (n), logo o algoritmo é O (n log n).

```
public static List<Lance> resolverPorMaiorValor(List<Lance> lances, int
energia) {
    //Organiza os lances em ordem decrecente em relação valor
    Collections.sort(lances, Comparator.comparingInt(b -> -b.valor));
    List<Lance> lancesSelecionados = new ArrayList<>();
    int totalValor = 0;
    for (Lance lance : lances) {
        if (energia >= lance.energia) {
            energia -= lance.energia;
            totalValor += lance.valor;
            lancesSelecionados.add(lance);
        }
        if (energia <= 10)
            break;
    }
```

```
    return lancesSelecionados;
}
```

Código 3 – Algoritmo Guloso: estratégia de lances com maior valor

Já o segundo algoritmo guloso seleciona lances com base no valor absoluto, priorizando lances com o maior valor, independentemente da energia que consomem.

O algoritmo funciona de maneira muito parecida sendo que a grande diferença é o quesito de ordenação que no caso é o valor absoluto dos lances em ordem decrescente. Por conta disso a complexidade é a mesma.

2.1.3 Divisão e Conquista

```
public static List<Lance> resolver(List<Lance> lances, int energiaMaxima)
{

   lances.sort((l1, l2) -> Double.compare((double) l2.valor / l2.energia,
(double) l1.valor / l1.energia));
    List<Lance> lancesSelecionados = new ArrayList<>();
    divisaoConquista(lances, 0, lances.size() - 1, energiaMaxima,
lancesSelecionados);
    return lancesSelecionados;
}
```

Código 4 – Algoritmo chamador do algoritmo divisão e conquista

O *Código 4* inicializa uma lista vazia **`lancesSelecionados`** para armazenar os lances que serão selecionados. Além disso, ordena os lances por valor por unidade de energia de forma decrescente. Em seguida, chama a função **`divisaoConquista`** para iniciar o processo de divisão e conquista, passando a lista de lances, os índices iniciais e finais da lista, o limite máximo de energia e a lista `lancesSelecionados` como parâmetros.

```
private static void divisaoConquista(List<Lance> lances, int esquerda, int
direita, int energiaRestante, List<Lance> lancesSelecionados) {
```

```
    if (esquerda > direita || energiaRestante <= 0) {
        return;
    }


    int indiceMaximoLucro = esquerda;
    for (int i = esquerda + 1; i <= direita; i++) {
        if ((double) lances.get(i).valor / lances.get(i).energia >
(double) lances.get(indiceMaximoLucro).valor /
lances.get(indiceMaximoLucro).energia) {
            indiceMaximoLucro = i;
        }
    }

    Lance lanceSelecionado = lances.get(indiceMaximoLucro);


    if (lanceSelecionado.energia <= energiaRestante) {
        lancesSelecionados.add(lanceSelecionado);
        energiaRestante -= lanceSelecionado.energia;
    }


    divisaoConquista(lances, esquerda, indiceMaximoLucro - 1,
energiaRestante, lancesSelecionados);
    divisaoConquista(lances, indiceMaximoLucro + 1, direita,
energiaRestante, lancesSelecionados);
}
```

Código 5 – Algoritmo de divisão e conquista

O código 5 chamado **`divisaoConquista`** é a implementação principal do algoritmo de divisão e conquista, cujo objetivo é resolver o problema de seleção de lances com uma restrição de energia máxima. Ela recebe como entrada a lista de **`lances`**, os índices **`esquerda`** e **`direita`** que delimitam o intervalo atual de lances a serem considerados, a **`energiaRestante`** que representa a energia disponível até o momento, e a lista **`lancesSelecionados`** onde os lances escolhidos serão armazenados.

A recursão possui um critério de parada que é quando **esquerda** ultrapassa **direita.** O que significa que não há lances a serem considerados no intervalo atual ou se **`energiaRestante`** for menor ou igual a zero.

O loop Itera sobre os lances do índice **`esquerda`** a **`direita`** para encontrar o lance com o maior valor por unidade de energia.

Se o lance selecionado (lances.get(indiceMaximoLucro)) pode ser incluído na energia disponível (lanceSelecionado.getEnergia() <= energiaRestante), ele é adicionado à lista lancesSelecionados. A energia disponível é então reduzida pelo consumo do lance do **energiaRestante -= lanceSeleciono.getEnergia().**

O algoritmo resolve de forma recursiva dois sub-problemas: Um subproblema que abrange o intervalo à esquerda do lance selecionado (de **esquerda a `indiceMaximoLucro - 1`) e** outro subproblema que abrange o intervalo à direita do lance selecionado (**indiceMaximoLucro + 1 a direita)**.

Ao dividir o problema em subproblemas menores e resolver cada um de forma recursiva, o algoritmo pode encontrar uma solução ótima que maximiza o lucro total sob análise do valor de “dinheiros” dos lances selecionados.

A complexidade de tempo para esse algoritmo é O(n log(n)), onde n é o número de lances totais. Isso porque o algoritmo divide de forma recursiva o problema em dois subproblemas pela metade do tamanho, parecido com o **Merge Sort**.

2.1.4 Programação Dinâmica

```java
package algoritmos;

import java.util.ArrayList;
import java.util.List;

import entity.Lance;

public class ProgramacaoDinamica {

    public static class Solucao {
        public int maiorLucro;
        public List<Lance> lancesSelecionados;

        public Solucao(int maiorLucro, List<Lance> lancesSelecionados) {
            this.maiorLucro = maiorLucro;
            this.lancesSelecionados = lancesSelecionados;
```

```
        }
    }

    public static Solucao getSolucao(List<Lance> lances, int energia) {
        int n = lances.size();
        int[] lucrosMax = new int[energia + 1];
        int[][] escolhidos = new int[n + 1][energia + 1];

        for (int i = 1; i <= n; i++) {
            for (int j = energia; j >= lances.get(i - 1).energia; j--) {
                if (lucrosMax[j] < lucrosMax[j - lances.get(i -
1).energia] + lances.get(i - 1).valor) {
                    lucrosMax[j] = lucrosMax[j - lances.get(i -
1).energia] + lances.get(i - 1).valor;
                    escolhidos[i][j] = 1;
                }
            }
        }

        List<Lance> solucao = new ArrayList<>();
        int j = energia;
        for (int i = n; i > 0; i--) {
            if (escolhidos[i][j] == 1) {
                solucao.add(lances.get(i - 1));
                j -= lances.get(i - 1).energia;
            }
        }

        return new Solucao(lucrosMax[energia], solucao);
    }

    public static void main(String[] args) {
        List<Lance> lances = new ArrayList<>();
        lances.add(new Lance(500, 500));
        lances.add(new Lance(500, 510));
        lances.add(new Lance(400, 520));
        lances.add(new Lance(300, 400));
        lances.add(new Lance(200, 220));
        lances.add(new Lance(900, 1110));

        int energia = 1000;
        Solucao solucao = getSolucao(lances, energia);

        System.out.println("Maior lucro: " + solucao.maiorLucro);
```

```
        System.out.println("Solução:");
        for (Lance lance : solucao.lancesSelecionados) {
            System.out.println(lance);
        }
    }
}
```

Código 6 – Implementação de Programação Dinâmica

O algoritmo de programação dinâmica apresentado tem o objetivo de maximizar o lucro selecionando lances sem exceder uma capacidade máxima de energia. Ele faz isso criando duas tabelas: uma para armazenar o lucro máximo possível para cada nível de energia e outra para rastrear quais lances foram escolhidos. O algoritmo itera sobre cada lance e nível de energia, atualizando as tabelas com os valores ótimos baseados no lucro adicional que cada lance pode trazer. Após processar todos os lances, o algoritmo reconstrói a solução ótima percorrendo as tabelas de trás para frente e coletando os lances que foram selecionados. Por fim, ele retorna o maior lucro possível e a lista de lances que compõem essa solução.

A classe **Solucao** Inicializa os atributos **maiorLucro** e **lancesSelecionados** que armazenam, respectivamente, o maior lucro possível e a lista de lances que foram selecionados para alcançar esse lucro.

O método **getSolucao** recebe uma lista de lances e um valor máximo de energia com o objetivo de encontrar a combinação de lances que maximiza o lucro sem exceder a energia disponível. Para isso, o algoritmo inicializa uma tabela **lucrosMax** que armazena o lucro máximo possível para cada nível de energia, até o limite especificado. A matriz **escolhidos** é usada para rastrear quais lances foram escolhidos para formar o lucro máximo em cada nível de energia. O algoritmo itera sobre todos os lances e todas as possíveis energias, atualizando as tabelas com os valores ótimos e reconstrói a solução ótima percorrendo a matriz de trás para frente e coletando os lances selecionados.

O algoritmo possui uma complexidade de tempo de $O(n \cdot E)$, considerando que **n** é o número de lances e **E** é a quantidade máxima de energia disponível. Isso ocorre porque o algoritmo itera sobre todos os lances com um loop externo e cada lance itera sobre os níveis de energia com um loop interno.

2.2 Testes

Para gerar os testes foi feito uma classe geradora de lances:

```java
public class GeradorLances {

   public static List<Lance> gerarLancesAleatorios(int quantidade, int
energiaMax, long seed) {
        Random random = new Random(seed);
        List<Lance> lances = new ArrayList<>();
        for (int i = 0; i < quantidade; i++) {
            int energy = random.nextInt(energiaMax) + 1;
            int value = random.nextInt(1000) + 1;
            lances.add(new Lance(energy, value));
        }
        return lances;
    }
}
```

Código 7 – Algoritmo que gera lances aleatórios

A função **gerarLancesAleatorios** recebe como parâmetro a quantidade de lances que se deseja criar, a energia máxima disponível e uma semente para gerar os números aleatórios. A energia de um lance pode ir de 0 até a energia máxima e o valor pode ser de 0 a 1000.

Para os testes de **backtracking** e **divisão e conquista**, conforme o enunciado do exercício, foram executados 10 testes por tamanho de conjunto de lances que vão de conjuntos de 10 até T elementos. Para os algoritmos **gulosos** e **programação dinâmica** os mesmos testes foram feitos, mas ao chegar a T o tamanho dos conjuntos de lances foi aumentando em T em T ao invés de 1 em 1 até chegar em 10T.

O T foi escolhido como 130, após observações iniciais do algoritmo de backtracking que começa a ficar extremamente demorado depois desse ponto, apesar de não estourar o limite de 30 s estipulado no enunciado. Sendo assim, por questão de tempo, esse valor foi assumido.

A cada teste é gerado uma lista de lances baseado no tamanho atual do conjunto e no iterador de número de testes para ser a semente, esse iterador sempre vai de 0 até 9. Portanto, mesmo que o conjunto aumente e novos números sejam introduzidos, muitas vezes as respostas encontradas podem tender a ter termos parecidos para diferentes tamanhos de conjuntos.

O algoritmo desejado é executado é o seu tempo de execução é calculado. Médias do **tempo de execução**, **energia total das soluções** e **valor total das soluções** são feitas das 10 execuções de um tamanho de conjunto e são registradas em um arquivo CVS.

```java
public class TesteBackTracking {
 public static void main(String[] args) {
        int tamanhoInicial = 10;
        int incremento = 1;
        int numTestes = 10;
        int energia = 1000;
        long limiteTempo = 30000; // 30 segundos em milissegundos
        int T = 130;

        try (FileWriter writer = new
FileWriter("resultadosBackTracking.csv")) {
            // Cabeçalho para os dados de saída
            writer.append("Tamanho DuraçãoMédia(ms)
MédiaValorTotalSoluções MédiaEnergiaTotalSoluções\n");

            for (int tamanho = tamanhoInicial; tamanho <= 130 ; tamanho +=
incremento) {
                List<Long> duracoes = new ArrayList<>();
                List<Lance> melhorSolucaoGeral = new ArrayList<>();
                int maxLucroGeral = 0;
                List<List<Lance>> todasSolucoes = new ArrayList<>();

                for (int teste = 0; teste < numTestes; teste++) {
                    List<Lance> lances = gerarLancesAleatorios(tamanho,
energia, teste);
                    BackTracking solucao = new BackTracking();

                    long inicioTempo = System.currentTimeMillis();
                    solucao.resolver(lances, energia);
                    long fimTempo = System.currentTimeMillis();

                    long duracao = fimTempo - inicioTempo;
                    duracoes.add(duracao);

                    List<Lance> solucaoAtual = solucao.getSolucao();
                    todasSolucoes.add(solucaoAtual);

                    if (solucao.getMaiorLucro() > maxLucroGeral) {
                        maxLucroGeral = solucao.getMaiorLucro();
                        melhorSolucaoGeral = solucaoAtual;
                    }
                }
```

```
                // Calcular a duração média para cada 10 execuções
                long duracaoTotal = 0;
                for (long duracao : duracoes) {
                    duracaoTotal += duracao;
                }
                double duracaoMedia = duracaoTotal / (double) numTestes;

                // Calcular a energia total e o valor da melhor solução
geral
                int energiaTotalMelhor = 0;
                int valorTotalMelhor = 0;
                for (Lance lance : melhorSolucaoGeral) {
                    energiaTotalMelhor += lance.energia;
                    valorTotalMelhor += lance.valor;
                }

                // Calcular a média dos valores e energias entre as 10
soluções
                double[] medias = calcularMedias(todasSolucoes);
                double mediaValorTotal = medias[0];
                double mediaEnergiaTotal = medias[1];

                System.out.println("Tamanho: " + tamanho);
                System.out.println("Duração média: " + duracaoMedia + "
ms");
                System.out.println("Energia total da melhor solução: " +
energiaTotalMelhor);
                System.out.println("Valor total da melhor solução: " +
valorTotalMelhor);
                System.out.println("Média do valor total das soluções: " +
mediaValorTotal);
                System.out.println("Média da energia total das soluções: "
+ mediaEnergiaTotal);
                System.out.println();

                // Formatar a saída em CSV e escrever no arquivo
                writer.append(String.format("%d %.2f %.2f %.2f\n",
                        tamanho, duracaoMedia, mediaValorTotal,
                        mediaEnergiaTotal));

                // Verificar se a duração média excede o limite de tempo
                if (duracaoMedia > limiteTempo) {
                    writer.append("Tamanho " + tamanho + " excedeu o
limite de 30 segundos\n");
                    break;
                }
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static double[] calcularMedias(List<List<Lance>>
todasSolucoes) {
        double somaValorTotal = 0;
        double somaEnergiaTotal = 0;
        int numSolucoes = todasSolucoes.size();

        for (List<Lance> solucao : todasSolucoes) {
            int valorTotal = 0;
            int energiaTotal = 0;

            for (Lance lance : solucao) {
                valorTotal += lance.valor;
                 energiaTotal += lance.energia;
            }

            somaValorTotal += valorTotal;
            somaEnergiaTotal += energiaTotal;
        }

        double mediaValorTotal = somaValorTotal / numSolucoes;
        double mediaEnergiaTotal = somaEnergiaTotal / numSolucoes;

        return new double[]{mediaValorTotal, mediaEnergiaTotal};
    }
}
```

Código 8 – Teste do algoritmo BackTracking

3. Resultados

Nesta seção, apresenta-se uma análise dos testes realizados. Explora-se a eficácia de cada abordagem, considerando diferentes conjuntos de dados e cenários variados, com o objetivo de identificar padrões de desempenho e determinar a adequação de cada algoritmo para diferentes condições de entrada. Finalmente, discutimos os resultados obtidos, destacando as soluções ótimas encontradas e comparando o desempenho dos algoritmos frente aos critérios de eficiência e precisão definidos para o problema em questão.

Os resultados gerados após a execução do algoritmo foram armazenados em um arquivo .csv chamado **`resultados<nome-do-algoritmo>.csv.`**

Cada arquivo contém as seguintes informações: tamanho dos conjuntos de dados de lances utilizados nos testes (**Tamanho**), tempo médio de execução dos testes em milissegundos (DuraçãoMédia), média do valor total das soluções encontradas nos testes (MédiaValorTotalSoluções), média da energia total consumida pelas soluções encontradas nos testes (MédiaEnergiaTotalSoluções). Cada linha de dados representa os resultados de um teste realizado com um tamanho específico de conjunto de dados de lances.

Nas seções a seguir é mostrado um fragmento de cada tabela de cada algoritmo gerada no Excel com tamanho de 10 a 20. As tabelas completas podem ser acessadas aqui.

3.1 Backtracking

A duração média de execução aumentou conforme o tamanho dos conjuntos de entrada cresceu, o que é esperado para um algoritmo de backtracking devido à sua complexidade exponencial. Nos primeiros tamanhos de entrada, o tempo de execução foi insignificante, mas a partir do tamanho 56, a duração começou a crescer de forma mais acentuada. A partir do tamanho 90, a duração média ultrapassou 50 ms, chegando a 2148,5 ms no tamanho 130. O valor total médio das soluções também mostrou uma tendência de aumento com o crescimento do tamanho do conjunto de entrada, indicando que o algoritmo conseguiu encontrar soluções mais lucrativas à medida que mais lances foram considerados. No início, os valores totais médios estavam em torno de 2000 dinheiros, aumentando gradualmente até 9097,2 dinheiros no tamanho 130. A média da energia total das soluções variou conforme os tamanhos dos conjuntos de entrada, mantendo-se geralmente próxima da capacidade máxima de energia (1000 megawatts), demonstrando que o algoritmo foi eficaz em utilizar a maior parte da energia disponível para maximizar o lucro, com a energia total média variando entre 876 e 991,2 megawatts. O resultado mostra que o tempo de execução aumenta exponencialmente com o crescimento do tamanho do conjunto de entrada. Embora a estratégia de poda tenha contribuído para a eficiência do algoritmo, a escalabilidade é um desafio devido à complexidade inerente do backtracking. O algoritmo foi capaz de encontrar soluções cada vez mais lucrativas conforme o número de interessados aumentava, indicando sua eficácia em explorar o espaço de solução. Além disso, o algoritmo utilizou a maior parte da energia disponível de maneira eficiente, maximizando o lucro das vendas. Resolvemos deixar o tamanho do conjunto de entrada até 130 porque, após esse ponto, a duração para executar

era muito alta, indicando a necessidade de técnicas mais avançadas ou heurísticas para problemas de maior escala.

No entanto, foram realizados testes adicionais para avaliar o tempo de execução e o comportamento do algoritmo até atingir o limite de 30 segundos. Observou-se que o algoritmo chegou a rodar por até uma hora para concluir. Apesar do tempo prolongado, tanto o valor total das soluções quanto a energia utilizada aumentaram, indicando um potencial de melhoria na qualidade das soluções encontradas.

Os testes adicionais demonstraram que o algoritmo de backtracking pode encontrar soluções cada vez melhores à medida que o tempo de execução aumenta, resultando em valores totais mais altos e maior utilização de energia. Porém, a escalabilidade limitada e o tempo de execução prolongado destacam a necessidade de explorar métodos mais eficientes para resolver problemas de grande porte.

Backtracking	Tamanh	DuraçãoMédia(ms)	MédiaValorTotalSoluções	MédiaEnergiaTotalSoluções
	10	0,1	2112,2	876
	11	0	2206	873,8
	12	0	2300	899,8
	13	0,1	2395,7	935,2
	14	0	2589,1	921,8
	15	0	2589,1	921,8
	16	0	2649,2	918
	17	0,1	2688	921,8
	18	0	2730	915,8
	19	0	2871,5	954,1
	20	0,1	3021,3	963,8

Tabela 1 – Tabela dos resultados do backtracking

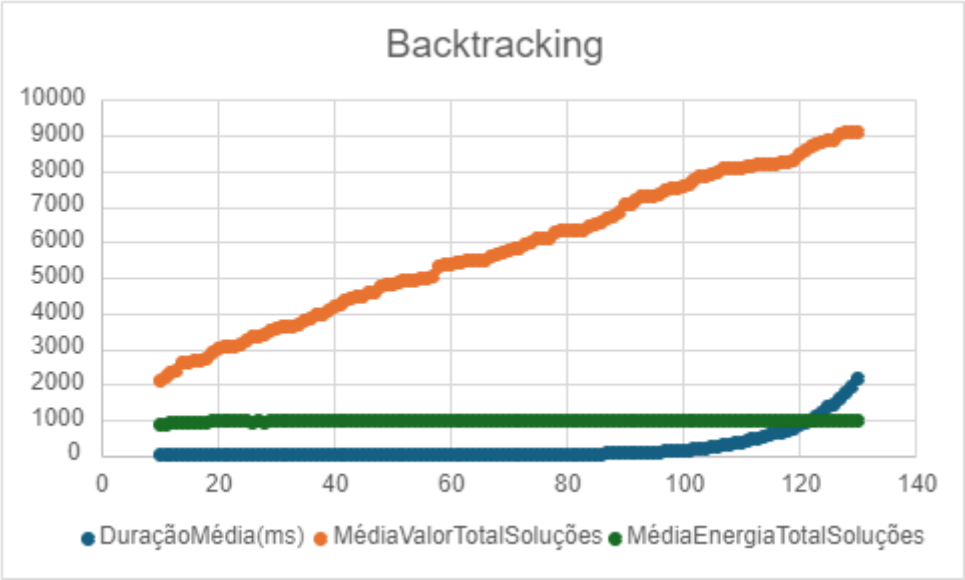

Gráfico 1 – Gráfico dos resultados do backtracking

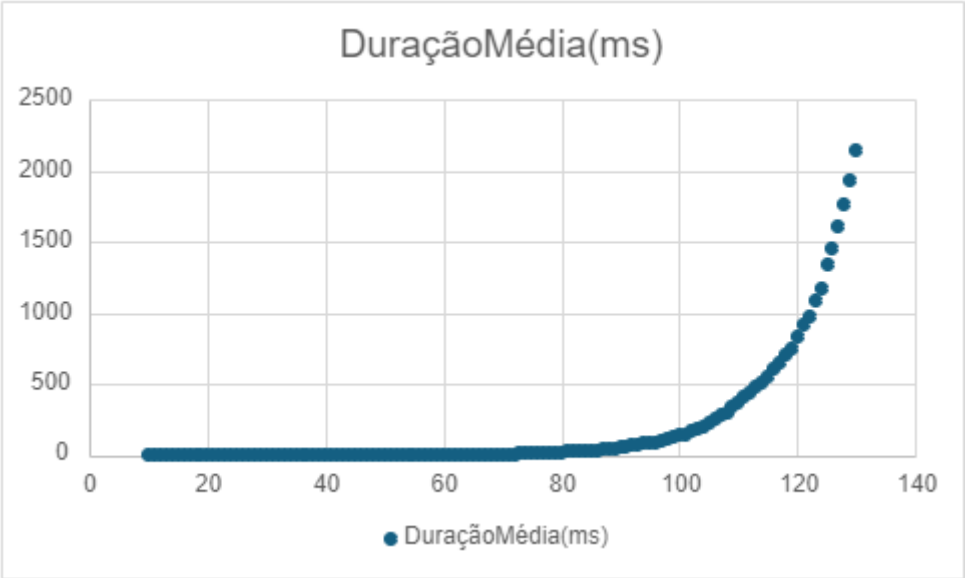

Gráfico 2 – Gráfico de duração do backtracking

3.2 Algoritmo Guloso

Para os algoritmos gulosos pode se ver que o critério guloso de **Valor/Energia** tem resultados muito melhores em questão de valor total. Isso ocorre, pois, a estratégia de **maior valor** é muito suscetível a lances de valores altos, mas que também ocupam muito espaço, enquanto a estratégia de valor/energia dá uma boa estimativa para maximizar a quantidade de lucro pela energia disponível.

Apesar disso, ambos algoritmos são extremamente rápidos, o que é de se esperar considerando a simplicidade de seus critérios e sua fácil implementação.

Uma observação interessante no gráfico do algoritmo guloso de **maior valor** é o fato de ele apresentar um padrão de zig zag. Apesar de muita especulação do grupo, não foi descoberto um motivo muito aparente para isso, uma teoria foi a forma de como as sementes dos testes são feitas, porém não parece muito coeso. Por outro lado, imaginamos que seja só a aparição de um elemento que exalte a fraqueza do algoritmo.

Guloso	Tamanho	DuraçãoMédia(ms)	MédiadoValorTotaldasSoluções	MédiadaEnergiaTotaldasSoluções
	10	0,5	2084,9	852,9
	11	0	2178,7	850,7
	12	0	2220,9	860,5
	13	0	2321,2	899,5
	14	0	2526,1	844
	15	0	2526,1	844
	16	0	2597,3	866,5
	17	0,1	2636,1	870,3
	18	0	2641,9	844
	19	0	2832,9	902,5
	20	0,1	2969,6	907,2

Tabela 2 – Tabela dos resultados do Guloso 1 (Valor/Energia)

Guloso	Tamanho	DuraçãoMédia(ms)	MédiadoValorTotaldasSoluções	MédiadaEnergiaTotaldasSoluções
	10	0,2	1974,7	880,1
	11	0	1998,5	871,2
	12	0	1861,1	915,2
	13	0	1946,1	936,7
	14	0	1946,1	936,7

	15	0	1946,1	936,7
	16	0	1953,5	949,1
	17	0	1953,5	949,1
	18	0	1946	947,2
	19	0	2125,8	940,3
	20	0	2112,5	940,1

Tabela 3 – Tabela dos resultados do Guloso 2(Maior Valor)

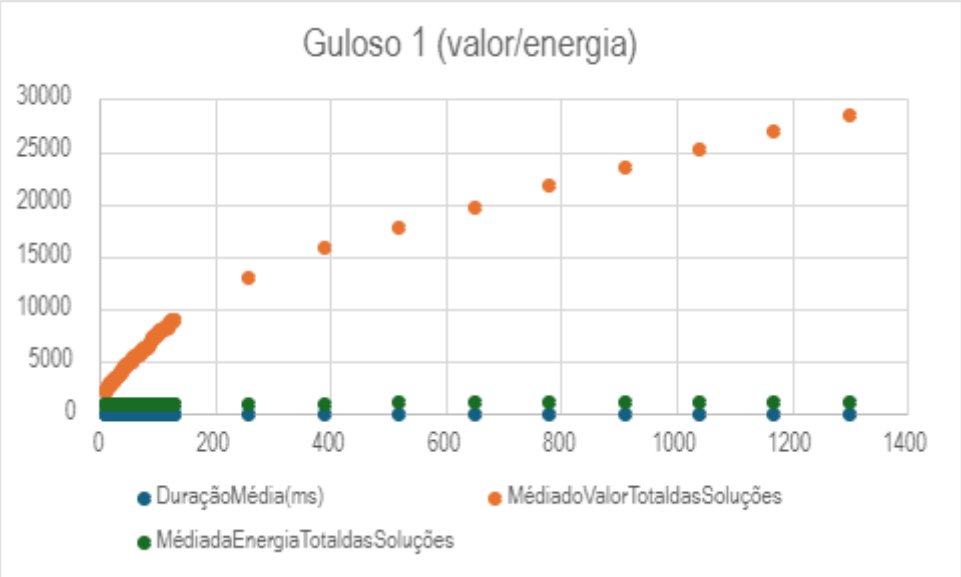

Gráfico 3 – Gráfico dos resultados do Guloso 1(Valor/Energia)

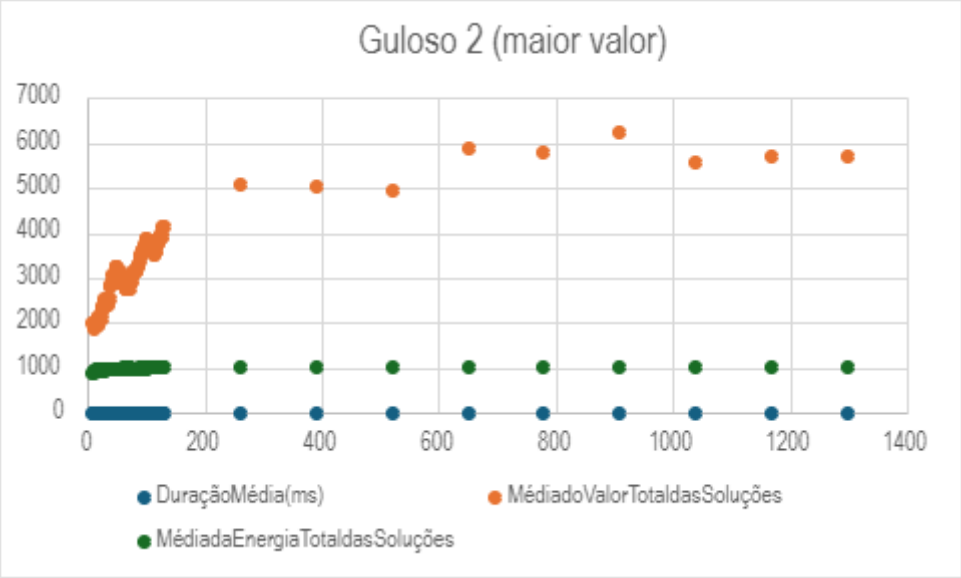

Gráfico 4 – Gráfico dos resultados do Guloso 2(Maior Valor)

3.3 Divisão e Conquista

Os resultados gerados após a execução do algoritmo foram armazenados em um arquivo .csv chamado **`resultadosDivisaoConquista.csv.`**

Divisão e Conquista	Tamanho	DuraçãoMédia(ms)	MédiaValorTotalSoluções	MédiaEnergiaTotalSoluções
	10	0,1	2084,9	852,9
	11	0,1	2178,7	850,7
	12	0	2220,9	860,5
	13	0	2321,2	899,5
	14	0,1	2526,1	844
	15	0	2526,1	844
	16	0	2597,3	866,5
	17	0,1	2636,1	870,3
	18	0,1	2641,9	844
	19	0,1	2832,9	902,5
	20	0	2969,6	907,2

Tabela 3 – Tabela dos resultados da divisão e conquista

Na tabela 3, é observado que a duração média de execução do algoritmo, medida em milissegundos (ms), é predominantemente baixa, com a maioria dos casos registrando tempos de execução de apenas 0,1 ms. Isso sugere que o algoritmo é eficiente e capaz de processar rapidamente conjuntos de dados mesmo à medida que o tamanho da amostra aumenta.

A energia total média utilizada nas soluções selecionadas apresenta variações, mas tende a se manter relativamente estável ao longo dos diferentes tamanhos de amostra. Isso sugere uma eficiência consistente na energia disponível.

Como dito anteriormente, a complexidade desse algoritmo é $O\,(n \log(n))$, onde n é o tamanho do problema. Isso significa que, idealmente, à medida que o tamanho do problema aumenta, o tempo de execução do algoritmo cresce de maneira

logarítmica, o que é muito eficiente comparado a abordagem backtracking mostrada no presente trabalho.

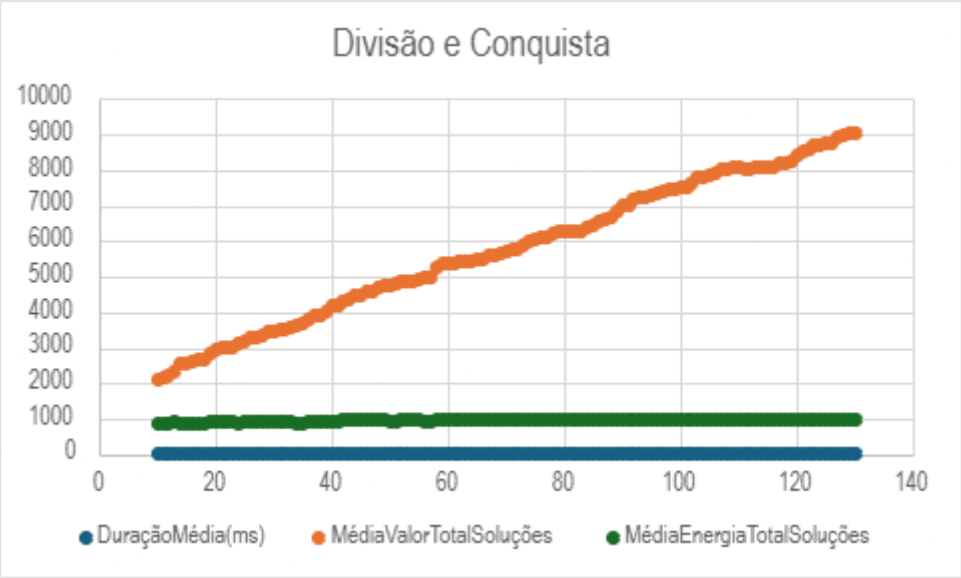

Gráfico 5 – Gráfico dos resultados da divisão e conquista

3.4 Programação Dinâmica

Os resultados obtidos da execução do teste do algoritmo de Programação Dinâmica, armazenados em **resultadosProgramacaoDinamica.csv**, foram transformados em uma tabela e em seguida em um gráfico, que podem ser observados abaixo.

Conforme foi dito anteriormente, a complexidade desse algoritmo é O(n*E), no entanto, para o teste executado, o valor de E permanece constante, fazendo com que o algoritmo se comporte com complexidade de O(n). É possível observar esse comportamento através do gráfico levando em conta que o tempo médio de execução aumenta de forma linear, como é esperado para esse tipo de complexidade.

Além disso, pode-se perceber que o algoritmo oferece resultados eficientes de modo geral, encontrando soluções com uso de energia próximo do valor máximo, melhorando ao passo que a quantidade de lances aumenta. Os valores totais dos lances analisados também mostram resultados relativamente bons, que aumentam conforme mais lances são adicionados.

Tamanho	DuraçãoMédia(ms)	MédiadoValorTotaldasSoluções	MédiadaEnergiaTotaldasSoluções
10	0,7	2112,2	876
11	0,2	2206	873,8
12	0,2	2300	899,8

13	0,2	2395,7	935,2
14	0,2	2589,1	921,8
15	0,1	2589,1	921,8
16	0	2649,2	918
17	0	2688	921,8
18	0	2730	915,8
19	0,2	2871,5	954,1
20	0	3021,3	963,8

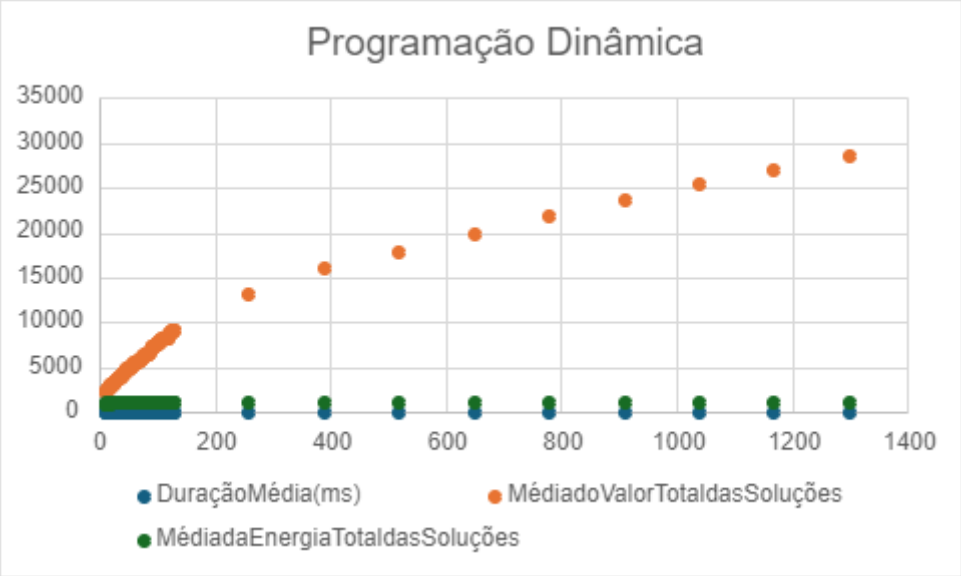

Gráfico 6 – Gráfico dos resultados de Programação Dinâmica

4. Conclusão

Este trabalho prático apresentou uma análise detalhada de quatro técnicas algorítmicas distintas para resolver o problema de otimização de vendas de energia em leilões: backtracking, algoritmo guloso, divisão e conquista e programação dinâmica. Cada técnica foi implementada e testada para avaliar sua eficácia e eficiência em maximizar o lucro da empresa produtora de energia ao vender lotes de megawatts em leilões.

Os resultados demonstraram que cada técnica possui suas próprias vantagens e limitações.

Backtracking: Embora esta técnica garanta a solução ótima, sua complexidade exponencial torna-a impraticável para grandes conjuntos de dados, como evidenciado pelo tempo de execução significativamente aumentado com o

crescimento dos tamanhos dos conjuntos de lances. No entanto, a estratégia de poda implementada ajudou a melhorar a eficiência.

Algoritmo Guloso: Duas abordagens gulosas foram testadas, uma baseada na relação valor/energia e outra no valor absoluto dos lances. Ambas mostraram uma execução extremamente rápida devido à sua simplicidade. A abordagem valor/energia apresentou resultados superiores em termos de lucro total, demonstrando ser uma heurística eficiente para maximizar o lucro utilizando a energia disponível de forma mais eficaz.

Divisão e Conquista: Esta técnica mostrou-se eficiente, com tempos de execução baixos mesmo para conjuntos de dados maiores. A abordagem de dividir o problema em subproblemas menores permitiu uma resolução rápida, mantendo uma utilização eficiente da energia disponível.

Programação Dinâmica: Com uma complexidade de tempo linear em relação ao número de lances, esta técnica mostrou-se eficaz na obtenção de soluções próximas ao ótimo, utilizando a maior parte da energia disponível. O aumento linear do tempo de execução conforme o tamanho dos dados aumentava confirmou a previsibilidade e a escalabilidade do algoritmo.

A análise comparativa dos algoritmos revelou diferenças significativas em termos de eficiência e eficácia. O backtracking, apesar de garantir soluções ótimas, mostrou-se impraticável para conjuntos de dados maiores devido ao aumento exponencial do tempo de execução. Em contraste, os algoritmos gulosos apresentaram tempos de execução extremamente rápidos. No entanto, a abordagem valor/energia do algoritmo guloso foi mais eficaz em maximizar o lucro comparado ao maior valor absoluto, que, embora rápido, produziu resultados menos consistentes. A divisão e conquista equilibrou bem eficiência e qualidade da solução, mantendo baixos tempos de execução e altos valores totais. A programação dinâmica destacou-se por sua capacidade de fornecer soluções próximas do ótimo de maneira eficiente, com um comportamento linear em relação ao número de lances, sendo altamente eficaz para problemas de maior escala.

Em termos de utilização da energia disponível, todos os algoritmos foram eficientes, com a média da energia total das soluções sendo próxima da capacidade máxima. A escolha do algoritmo ideal depende do balanço desejado entre a precisão da solução e o tempo de execução, com programação dinâmica e divisão e conquista sendo as mais indicadas para grandes conjuntos de dados e algoritmos gulosos para soluções rápidas e suficientemente boas.