PUC Minas

BEATRIZ DE OLIVEIRA SILVEIRA
FRANCISLEY DOMINGOS MAGALHÃES
HENRIQUE JARDIM MELO
MARIA EDUARDA CHRISPIM SANTANA

TRABALHO PRÁTICO: LEILÃO DE ENERGIA

BELO HORIZONTE
2024

SUMÁRIO

1. INTRODUÇÃO

Este trabalho consiste na utilização de técnicas de projetos de algoritmos, focadas na resolução de problemas intratáveis, pertencentes à classe NP, e se concentra na maximização do lucro de uma empresa produtora de energia através de leilões, onde diversas empresas interessadas oferecem lances por lotes de energia. O objetivo é determinar a combinação ótima de lances que resulte no maior valor total de vendas, respeitando a quantidade total de energia disponível para venda.

O problema apresentado envolve a decisão de seleção de lances de forma a otimizar o retorno financeiro, considerando que cada empresa oferece um preço por megawatt de energia. Dessa forma, este trabalho tem como objetivos implementar quatro técnicas distintas para resolver o problema de seleção de lances em leilões de energia.

O relatório técnico detalha as implementações realizadas, as decisões tomadas em cada abordagem e uma análise comparativa dos resultados obtidos.

2. ESTRATÉGIAS DE SOLUÇÃO

O objetivo do projeto é projetar e implementar soluções utilizando quatro diferentes técnicas de algoritmos para resolver o problema de venda de energia, sendo elas Backtracking, Algoritmos Gulosos, Divisão e Conquista, e Programação Dinâmica.

Em relação à organização do código, as implementações foram divididas em classes separadas, cada uma responsável por uma técnica de solução diferente. Há também três classes principais da solução, sendo elas *Lance*, que representa um lance de uma empresa; *VendedorDeEnergia*, classe para gerenciar o processo de venda de energia; Main, que usa todas as classes e resolve o problema com as diferentes abordagens, inicializando os dados e executando cada algoritmo; e *Teste*, que realiza testes com conjuntos de tamanho crescente.

2.1. BACKTRACKING

A classe *SolucionadorBacktracking* utiliza a técnica de backtracking para encontrar a combinação de lances que maximiza o valor total sem exceder a energia disponível. O método *resolverBacktracking* explora todas as combinações possíveis de lances de forma recursiva, utilizando a poda para descartar combinações inviáveis.

A estratégia de poda utilizada no método *resolverBacktracking* foi a poda por capacidade de energia. A implementação consiste em, sempre que a soma das energias dos lances na combinação atual ultrapassa a *quantidadeEnergia* disponível, a função recursiva retorna imediatamente, não explorando mais combinações que incluam essa configuração, evitando, assim, a exploração de combinações que excedem a quantidade de energia disponível. Ao eliminar caminhos que não podem levar a uma solução ótima, a poda não só reduz o tempo de execução, mas também melhora a eficácia do algoritmo na busca da melhor solução possível.

```
private void resolverBacktracking(List<Lance> solucaoAtual, int valorAtual, int
energiaAtual, int index) {
        if (energiaAtual > quantidadeEnergia) {
            return;
        }

        if (valorAtual > melhorValor) {
            melhorValor = valorAtual;
            melhorSolucao = new ArrayList<>(solucaoAtual);
        }

        for (int i = index; i < lances.size(); i++) {
            Lance lance = lances.get(i);
```

```
            solucaoAtual.add(lance);
            resolverBacktracking(solucaoAtual, valorAtual + lance.getValor(),
energiaAtual + lance.getQuantidadeEnergia(), i + 1);
            solucaoAtual.remove(solucaoAtual.size() - 1);
        }
    }
```

2.2. ALGORITMO GULOSO

A classe *SolucionadorGuloso* implementa dois algoritmos gulosos para resolver o problema de seleção de lances com base na quantidade de energia disponível. O objetivo é maximizar o valor total obtido selecionando as melhores ofertas, conforme duas estratégias diferentes.

A primeira estratégia utilizada está no método *resolverPorValor.* Este método implementa a estratégia gulosa de selecionar os lances pelo valor total, escolhendo primeiro os lances de maior valor até que a quantidade de energia disponível seja esgotada. A estratégia visa obter o máximo valor total o mais rápido possível, começando pelas ofertas mais lucrativas, para maximizar o retorno financeiro.

```
public void resolverPorValor() {
        long startTime = System.currentTimeMillis();

        List<Lance> lancesOrdenados = new ArrayList<>(lances);
        Collections.sort(lancesOrdenados, new Comparator<Lance>() {
            @Override
            public int compare(Lance l1, Lance l2) {
                return Integer.compare(l2.getValor(), l1.getValor());
            }
        });

        int energiaRestante = quantidadeEnergia;
        int valorTotal = 0;
        List<Lance> solucao = new ArrayList<>();

        for (Lance lance : lancesOrdenados) {
            if (energiaRestante >= lance.getQuantidadeEnergia()) {
                energiaRestante -= lance.getQuantidadeEnergia();
                valorTotal += lance.getValor();
                solucao.add(lance);
            }
        }
    }
```

A segunda estratégia utilizada está no método *resolverPorValorPorMegawatt*. Este método implementa a estratégia gulosa de selecionar os lances pelo valor por unidade de energia (megawatt), escolhendo primeiro os lances com maior valor por

megawatt. Ao priorizar lances com maior valor por unidade de energia, essa estratégia maximiza a eficiência do uso da energia disponível, buscando otimizar o valor obtido por cada unidade de energia vendida.

```
public void resolverPorValorPorMegawatt() {
        long startTime = System.currentTimeMillis();

        List<Lance> lancesOrdenados = new ArrayList<>(lances);
        Collections.sort(lancesOrdenados, new Comparator<Lance>() {
            @Override
            public int compare(Lance l1, Lance l2) {
                double valorPorMegawatt1 = (double) l1.getValor() /
l1.getQuantidadeEnergia();
                double valorPorMegawatt2 = (double) l2.getValor() /
l2.getQuantidadeEnergia();
                return Double.compare(valorPorMegawatt2, valorPorMegawatt1);
            }
        });

        int energiaRestante = quantidadeEnergia;
        int valorTotal = 0;
        List<Lance> solucao = new ArrayList<>();

        for (Lance lance : lancesOrdenados) {
            if (energiaRestante >= lance.getQuantidadeEnergia()) {
                energiaRestante -= lance.getQuantidadeEnergia();
                valorTotal += lance.getValor();
                solucao.add(lance);
            }
        }
    }
```

2.3. DIVISÃO E CONQUISTA

A classe *SolucionadorDivisaoConquista* implementa o método de Divisão e Conquista para resolver o problema de seleção de lances de energia com o objetivo de maximizar o valor total obtido, dentro de uma quantidade específica de energia disponível.

Primeiramente, a lista de lances foi dividida em duas partes aproximadamente iguais, reduzindo a complexidade do problema original, tornando-o mais manejável. Ao dividir a lista, dois subproblemas menores são criados e são mais fáceis de resolver. Cada uma das sublistas é resolvida de forma recursiva. A mesma estratégia de divisão e conquista foi aplicada para as sublistas até que cada uma contenha apenas um lance ou não tenha lances restantes. Se a sublista contém apenas um lance, é verificado se ele pode ser incluído na solução com base na quantidade de energia disponível. Após resolver os subproblemas, as soluções

foram combinadas para obter a solução final. A solução para a sublista esquerda é calculada primeiro, e a energia restante após considerar essa solução é usada para calcular a melhor solução para a sublista direita. Os lances da sublista direita são escolhidos com base na energia que sobra após considerar os lances da esquerda.

```
private List<Lance> resolverDivisaoConquista(List<Lance> lances, int
energiaDisponivel) {

        if (energiaDisponivel <= 0 || lances.isEmpty()) {
            return new ArrayList<>();
        }

        if (lances.size() == 1) {
            Lance lance = lances.get(0);
            if (lance.getQuantidadeEnergia() <= energiaDisponivel) {
                return List.of(lance);
            } else {
                return new ArrayList<>();
            }
        }

        // Divide
        int meio = lances.size() / 2;
        List<Lance> esquerda = lances.subList(0, meio);
        List<Lance> direita = lances.subList(meio, lances.size());

        // Conquista
        List<Lance> melhorSolucaoEsquerda = resolverDivisaoConquista(esquerda,
energiaDisponivel);
        int energiaRestante = energiaDisponivel -
melhorSolucaoEsquerda.stream().mapToInt(Lance::getQuantidadeEnergia).sum();
        List<Lance> melhorSolucaoDireita = resolverDivisaoConquista(direita,
energiaRestante);

        // Combina
        List<Lance> melhorSolucaoCombinada = new
ArrayList<>(melhorSolucaoEsquerda);
        melhorSolucaoCombinada.addAll(melhorSolucaoDireita);

        return melhorSolucaoCombinada;
    }
```

2.4. PROGRAMAÇÃO DINÂMICA

A classe *SolucionadorProgramacaoDinamica* implementa uma solução baseada em programação dinâmica para encontrar a combinação de lances que maximiza o valor total obtido, dado um limite de energia disponível. Inicialmente, a classe

recebe a quantidade de energia disponível e a lista de lances, onde cada lance especifica uma quantidade de energia necessária e um valor associado. O método *resolver* começa calculando o tempo de execução e, em seguida, define uma tabela de programação dinâmica *dp*, que mantém o valor máximo acumulado para cada capacidade de energia até o limite disponível. Além disso, uma matriz *solucao* é utilizada para rastrear as escolhas feitas durante o processo, indicando se um lance foi incluído na solução ótima para cada capacidade.

A tabela *dp* é preenchida de forma iterativa, onde para cada lance disponível e cada capacidade de energia, é decidido se a inclusão do lance atual resulta em um valor total maior do que o valor anterior sem o lance. Essa decisão é armazenada na matriz *solucao*, que posteriormente permite a reconstrução da solução ótima. Após preencher a tabela, a matriz *solucao* é usada para determinar quais lances foram efetivamente escolhidos, começando da capacidade total de energia e verificando os lances incluídos.

```
public void resolver() {
        long startTime = System.currentTimeMillis();

        int n = lances.size();
        int[] dp = new int[quantidadeEnergia + 1];
        int[][] solucao = new int[n + 1][quantidadeEnergia + 1];

        for (int i = 1; i <= n; i++) {
            Lance lance = lances.get(i - 1);
            for (int j = quantidadeEnergia; j >= lance.getQuantidadeEnergia();
j--) {
                int valor = lance.getValor();
                if (dp[j] < dp[j - lance.getQuantidadeEnergia()] + valor) {
                    dp[j] = dp[j - lance.getQuantidadeEnergia()] + valor;
                    solucao[i][j] = 1;
                }
            }
        }

        List<Lance> lancesEscolhidos = new ArrayList<>();
        int w = quantidadeEnergia;
        for (int i = n; i > 0 && w > 0; i--) {
            if (solucao[i][w] == 1) {
                Lance lance = lances.get(i - 1);
                lancesEscolhidos.add(lance);
                w -= lance.getQuantidadeEnergia();
            }
        }
```

3. COMPARAÇÃO ENTRE RESULTADOS

Foram realizados testes com cada algoritmo implementado utilizando diferentes conjuntos de empresas interessadas para avaliar o desempenho dos algoritmos implementados.

O primeiro teste foi realizado com 2 conjuntos com valores já definidos, conforme apresenta a Figura 1. Cada conjunto é composto por 25 empresas interessadas, que possuem nome, quantidade e valor.

```
// Conjunto de empresas interessadas 1
List<Lance> lancesConjunto1 = new ArrayList<>();
lancesConjunto1.add(new Lance("E1", 430, 1043));
lancesConjunto1.add(new Lance("E2", 428, 1188));
lancesConjunto1.add(new Lance("E3", 410, 1565));
lancesConjunto1.add(new Lance("E4", 385, 1333));
lancesConjunto1.add(new Lance("E5", 399, 1214));
lancesConjunto1.add(new Lance("E6", 382, 1498));
lancesConjunto1.add(new Lance("E7", 416, 1540));
lancesConjunto1.add(new Lance("E8", 436, 1172));
lancesConjunto1.add(new Lance("E9", 416, 1386));
lancesConjunto1.add(new Lance("E10", 423, 1097));
lancesConjunto1.add(new Lance("E11", 400, 1463));
lancesConjunto1.add(new Lance("E12", 406, 1353));
lancesConjunto1.add(new Lance("E13", 403, 1568));
lancesConjunto1.add(new Lance("E14", 390, 1228));
lancesConjunto1.add(new Lance("E15", 387, 1542));
lancesConjunto1.add(new Lance("E16", 390, 1206));
lancesConjunto1.add(new Lance("E17", 430, 1175));
lancesConjunto1.add(new Lance("E18", 397, 1492));
lancesConjunto1.add(new Lance("E19", 392, 1293));
lancesConjunto1.add(new Lance("E20", 393, 1533));
lancesConjunto1.add(new Lance("E21", 439, 1149));
lancesConjunto1.add(new Lance("E22", 403, 1277));
lancesConjunto1.add(new Lance("E23", 415, 1624));
lancesConjunto1.add(new Lance("E24", 387, 1280));
lancesConjunto1.add(new Lance("E25", 417, 1330));
```

```
// Conjunto de empresas interessadas 2
List<Lance> lancesConjunto2 = new ArrayList<>();
lancesConjunto2.add(new Lance("E1", 313, 1496));
lancesConjunto2.add(new Lance("E2", 398, 1768));
lancesConjunto2.add(new Lance("E3", 240, 1210));
lancesConjunto2.add(new Lance("E4", 433, 2327));
lancesConjunto2.add(new Lance("E5", 301, 1263));
lancesConjunto2.add(new Lance("E6", 297, 1499));
lancesConjunto2.add(new Lance("E7", 232, 1209));
lancesConjunto2.add(new Lance("E8", 614, 2342));
lancesConjunto2.add(new Lance("E9", 558, 2983));
lancesConjunto2.add(new Lance("E10", 495, 2259));
lancesConjunto2.add(new Lance("E11", 310, 1381));
lancesConjunto2.add(new Lance("E12", 213, 961));
lancesConjunto2.add(new Lance("E13", 213, 1115));
lancesConjunto2.add(new Lance("E14", 346, 1552));
lancesConjunto2.add(new Lance("E15", 385, 2023));
lancesConjunto2.add(new Lance("E16", 240, 1234));
lancesConjunto2.add(new Lance("E17", 483, 2828));
lancesConjunto2.add(new Lance("E18", 487, 2617));
lancesConjunto2.add(new Lance("E19", 709, 2328));
lancesConjunto2.add(new Lance("E20", 358, 1847));
lancesConjunto2.add(new Lance("E21", 467, 2038));
lancesConjunto2.add(new Lance("E22", 363, 2007));
lancesConjunto2.add(new Lance("E23", 279, 1311));
lancesConjunto2.add(new Lance("E24", 589, 3164));
lancesConjunto2.add(new Lance("E25", 476, 2480));
```

Figura 1. Conjuntos de empresas interessadas 1 e 2

O segundo teste gerou conjuntos de teste de tamanho crescente para cada técnica: Backtracking, Algoritmo Guloso, Divisão e Conquista, e Programação Dinâmica, utilizando diferentes critérios de incremento e tamanho baseados no tempo de execução.

3.1 TESTE COM CONJUNTO DE VALORES DEFINIDO

Em relação aos 2 conjuntos com valores já definidos, foram realizados testes para avaliar o tempo de execução e a qualidade da solução de cada técnica. A Tabela 1 apresenta os resultados encontrados, sendo que a coluna “Algoritmo” lista os diferentes algoritmos que foram testados; as colunas “Conjunto 1 - Tempo (ms)” e “Conjunto 2 - Tempo (ms)” mostram o tempo de execução de cada algoritmo ao resolver o Conjunto de Empresas Interessadas 1 e 2, medido em milissegundos; e as colunas “Conjunto 1 - Valor Total” e “Conjunto 2 - Valor Total” apresentam o valor total obtido pela combinação de lances selecionada por cada algoritmo para o Conjunto de Empresas Interessadas 1 e 2.

Algoritmo	Conjunto 1 - Tempo (ms)	Conjunto 1 - Melhor valor	Conjunto 2 - Tempo (ms)	Conjunto 2 - Melhor valor
Backtracking	601 ms	26.725	595 ms	40.348
Guloso por Valor	2 ms	26.725	0 ms	38.673
Guloso por Valor/MW	1 ms	26.725	1 ms	39.271
Divisão e Conquista	43 ms	25.356	1 ms	38.249
Programação Dinâmica	18 ms	26.725	5 ms	40.348

Tabela 1. Comparativo entre as técnicas

O Backtracking foi a abordagem mais lenta, apresentando tempos de execução superiores a 500 milissegundos em ambos os conjuntos. Isso se deve ao fato de que ele explora exaustivamente todas as combinações possíveis de lances para encontrar a solução ótima. Embora garanta a melhor solução, o tempo de execução aumenta exponencialmente com o tamanho do conjunto de lances e a quantidade de energia. Para ambos os conjuntos, ele conseguiu a melhor combinação de lances, maximizando o valor total obtido. Caso os testes fossem realizados com conjuntos maiores, poderia ter sido computacionalmente intensivo, devido ao crescimento exponencial do número de combinações possíveis.

Já o Algoritmo Guloso é extremamente rápido, uma vez que resolveu o problema dos dois conjuntos, utilizando as duas estratégias (*resolverPorValor* e *resolverPorValorPorMegawatt*) levando tempo de 0 a 2 milissegundos. Isso ocorreu porque ele toma decisões baseadas em heurísticas locais, selecionando lances com base no maior valor ou menor energia por unidade de dinheiro, sem explorar todas as combinações possíveis. A principal vantagem é a eficiência em termos de tempo de execução.

A Divisão e Conquista equilibra o tempo de execução e qualidade da solução. Embora não seja tão rápida quanto o Algoritmo Guloso, é significativamente mais rápida que o Backtracking, uma vez que apresentou tempos de execução que variaram entre 1 à 43 milissegundos. A diferença no tempo de execução entre os conjuntos 1 e 2 mostra que o desempenho pode depender da complexidade específica dos lances e da energia disponível. A técnica é útil para uma solução rápida com qualidade razoável, mas não se compara às outras em termos de garantir a solução ótima, uma vez que, no conjunto 1 apresentou o menor “Melhor valor”.

A Programação Dinâmica oferece um bom tempo de execução e qualidade da solução em relação à técnica Divisão e Conquista, pois resolveu os problemas entre 5 e 18 milissegundos. Ela consegue resolver o problema em um tempo

relativamente curto, garantindo a solução ótima para o problema de combinação de lances, sendo eficiente pois a escolha ótima local leva a uma solução ótima global. A Programação Dinâmica alcançou a solução ótima para ambos os conjuntos, igualando-se ao Backtracking em termos de qualidade da solução. A vantagem em relação ao backtracking é o tempo de execução significativamente menor.

3.2 TESTE COM CONJUNTOS DE TAMANHO CRESCENTE

O segundo teste consistia em gerar conjuntos de teste de tamanho crescente. A técnica Backtracking gera conjuntos de teste de tamanho crescente, a partir de 10 interessadas e incrementando de 1 em 1, até atingir um tamanho T que não consiga ser resolvido em até 30 segundos pelo algoritmo. Na busca do tempo limite de 30 segundos, o teste é feito com 10 conjuntos de cada tamanho, contabilizando a média das execuções.

A técnica do Algoritmo Guloso utiliza os mesmos conjuntos de tamanho T encontrados no Backtracking. Em seguida, os tamanhos dos conjuntos aumentam de T em T até atingir o tamanho 10T, sempre executando 10 testes de cada tamanho para utilizar a média.

Na técnica Divisão e Conquista é utilizado os mesmos conjuntos de tamanho T utilizados no Backtracking e na técnica Programação Dinâmica são utilizados os mesmos conjuntos de teste do Algoritmo Guloso.

Para isso, foi criada a classe *Teste* que realiza testes para diferentes tamanhos de conjuntos de lance. Foi definido para esse caso que o tamanho T que seria atingido tem o valor de 50. Dessa forma, seriam gerados conjuntos de 10 a 50 elementos e cada teste é executado 10 vezes

A quantidade de energia que o teste gera é entre 100 e 599 MW e o valor em dinheiros é entre 100 e 1099 dinheiros.

O método *gerarLancesAleatorios* gera uma lista de lances aleatórios para um dado tamanho, e os métodos *testarBacktracking*, *testarGuloso*, *testarDivisaoConquista* e *testarProgramacaoDinamica* executam o algoritmo correspondente várias vezes e calculam o tempo médio de execução e a média dos melhores valores.

```
private static List<Lance> gerarLancesAleatorios(int tamanho) {
        Random random = new Random();
        List<Lance> lances = new ArrayList<>();
        for (int i = 1; i <= tamanho; i++) {
            int quantidade = random.nextInt(500) + 100;
            int valor = random.nextInt(1000) + 100;
            lances.add(new Lance("Empresa " + i, quantidade, valor));
        }
        return lances;
    }
```

Após a realização dos testes, foi documentado nas Tabelas 2 à 6 os resultados do Backtracking, Guloso por Valor, Guloso por Valor/MW, Divisão e Conquista, e Programação Dinâmica, respectivamente.

Tamanho	Média dos tempos de execução (ms)	Média dos melhores valores
10	0,8	2721
20	0,8	4099
30	1,1	5167
40	1,4	6273
50	1,5	7315

Tabela 2. Resultado do Backtracking com conjuntos de teste de tamanho crescente

Tamanho	Média dos tempos de execução (ms)	Média dos melhores valores
10	0	2721
20	0	4099
30	0,1	5167
40	0,1	6273
50	0,2	7315

Tabela 3. Resultado do Guloso por Valor com conjuntos de teste de tamanho crescente

Tamanho	Média dos tempos de execução (ms)	Média dos melhores valores
10	0	2721
20	0,1	4099
30	0,1	5167
40	0,1	6273
50	0,2	7315

Tabela 4. Resultado do Guloso por Valor/MW com conjuntos de teste de tamanho crescente

Tamanho	Média dos tempos de execução (ms)	Média dos melhores valores
10	0,5	2352
20	0,6	4016
30	0,5	5167
40	0,6	5271
50	0,7	6298

Tabela 5. Resultado da Divisão e Conquista com conjuntos de teste de tamanho crescente

Tamanho	Média dos tempos de execução (ms)	Média dos melhores valores
10	0,2	2721
20	0,4	4099
30	0,4	5167
40	0,5	6273
50	0,6	7315

Tabela 6. Resultado da programação Dinâmica com conjuntos de teste de tamanho crescente

A partir dos resultados, pôde-se notar que o backtracking mostrou tempos de execução mais altos em comparação com a programação dinâmica, especialmente à medida que o tamanho do problema aumenta, por ser um algoritmo exponencial, que leva mais tempo conforme o número de lances e a quantidade de energia aumentam. Os valores obtidos foram semelhantes aos da programação dinâmica, o que mostra que o backtracking também conseguiu encontrar soluções ótimas, mas com um custo maior em termos de tempo de execução.

O algoritmo guloso por valor apresentou os tempos de execução mais baixos entre os algoritmos, pois seleciona lances de maior valor primeiro sem explorar todas as possibilidades.

Já no guloso por valor/MW, os tempos de execução foram baixos, embora um pouco mais altos em comparação com a primeira estratégia do guloso, uma vez que precisa calcular e comparar essas relações.

Os tempos de execução da divisão e conquista foram maiores em comparação com os algoritmos gulosos e programação dinâmica porque a divisão e conquista precisa dividir e combinar os conjuntos de lances. Os resultados foram bons, mas

um pouco abaixo dos obtidos pela programação dinâmica e pelo backtracking, indicando que pode não encontrar a solução ótima globalmente em todos os casos.

4. CONCLUSÃO

Neste estudo, exploramos e analisamos quatro técnicas de algoritmos- Backtracking, Algoritmo Guloso, Divisão e Conquista, e Programação Dinâmica- para resolver o problema de seleção de lances em leilões de energia, visando maximizar o lucro de uma empresa produtora de energia.

A técnica de Backtracking mostrou-se eficaz em encontrar a solução ótima, mas seu tempo de execução foi significativamente maior devido à sua abordagem exaustiva, não tendo um bom uso para conjuntos de grande dimensão. Entretanto, os algoritmos gulosos mostraram ser eficientes em relação ao tempo de execução, resolvendo o problema em milissegundos, embora não garantissem a solução ótima, sendo mais adequados para situações em que é preciso ser rápido. A técnica de Divisão e Conquista apresentou um bom tempo de execução e uma boa qualidade da solução, embora nem sempre garantisse a solução ótima, sendo ideal para problemas que podem ser divididos em partes menores. A Programação Dinâmica mostrou-se ser a mais eficiente, garantindo a solução ótima e sendo recomendada para problemas de tamanho moderado que exigem precisão. Assim, cada técnica possui suas vantagens específicas, e a escolha da mais adequada depende do tamanho do problema, da necessidade de precisão e das restrições de tempo.