Ciência da Computação

Algoritmos e Estruturas de Dados III

Documentação Trabalho Prático 1 - Hipercampos

Davi dos Reis de Jesus, Gabriel de Paula Meira

2023.1

1 Introdução

O problema que precisa ser solucionado é um problema elementar de geometria computacional, cujo foco principal está no desenvolvimento e análise da eficiência de diferentes algoritmos.

O objetivo do trabalho prático é conseguir formar a maior quantidade de triângulos possível, entre as âncoras e os pontos inseridos, e, para que esses triângulos sejam válidos, não pode haver uma intersecção entre os lados dos triângulos formados, conforme ilustrado na Figura 1. Os dados do problema são recebidos do usuário usando um arquivo de texto e o resultado é salvo em outro arquivo, como pode ser observado na seção seguinte.

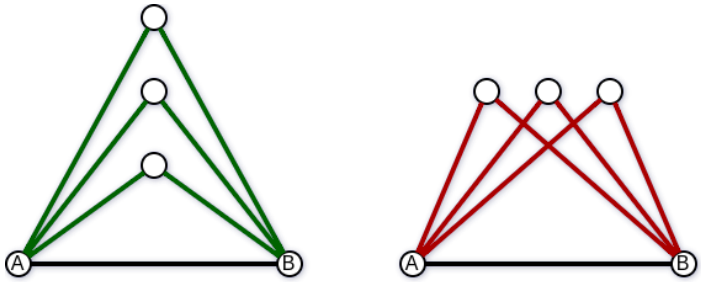

Figura 1: Exemplos de sequências válidas e inválidas

A eficiência de um algoritmo está ligada não somente ao tempo que a resposta é encontrada, mas em como será o comportamento para diferentes conjuntos de candidatos à solução, variando em quantidade e organização dos dados iniciais.

Faz-se necessário, portanto, desenvolver um programa que encontre a solução de maneira otimizada ao mesmo tempo que garante a facilidade de manutenção do código, o que pode ser alcançado por meio de boas práticas de programação e documentação adequada, assim como demonstrado adiante.

2 Estruturando a solução

É necessário, antes de iniciar a programação, planejar o funcionamento do programa em cada uma das etapas, a fim de garantir a qualidade do software do início ao fim do projeto, facilitando o processo de depuração e manutenção.

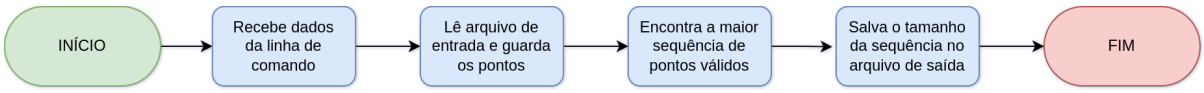

Figura 2: Fluxograma geral de funcionamento do programa

O programa foi dividido em quatro etapas principais, como ilustrado na Figura 2, com o objetivo de modularizar as tarefas em arquivos menores que são compilados utilizando o arquivo de instruções **Makefile**.

O controle de versionamento do software foi gerenciado de forma eficiente e segura usando o **Git**, a fim de permitir uma melhor colaboração entre a dupla e possibilitando a realização de alterações simultâneas sem conflitos, além de oferecer um histórico completo das modificações realizadas ao longo de todo o desenvolvimento.

2.1 Receber dados da linha de comando

A função **getopt()**, da biblioteca padrão do C, torna possível realizar a leitura dos argumentos da linha comando, dessa forma, os pontos usados no programa são lidos de um arquivo de entrada fornecido pelo usuário ao executar o código com a *flag* **-i**. De maneira semelhante, o arquivo de saída é informado utilizando a *flag* **-o**, resultando no seguinte formato:

```
{executavel} -i {arquivoEntrada} -o {arquivoSaida}
```

Para operações realizadas em outras *branches* do projeto, como a exibição gráfica dos pontos e o monitoramento do tempo de execução, são usadas as *flags* opcionais **-p** e **-r**, respectivamente, auxiliando na etapa de testes.

2.2 Ler arquivo de entrada e guardar os pontos

Logo após obter o endereço do arquivo de entrada, o código efetua a leitura dos dados em cada linha do arquivo usando a função **fscanf()**, pertencente à biblioteca **stdio.h**. A entrada é padronizada com a primeira linha contendo o número total de pontos seguido da posição no eixo das abscissas dos pontos-âncora A e B. As demais linhas possuem as coordenadas X e Y de cada ponto que deve ser analisado.

2.3 Encontrar a maior sequência de pontos válidos

Existem múltiplas formas de determinar a maior sequência de pontos que são válidos entre si, por essa razão foram desenvolvidos dois algoritmos funcionais para resolver esse problema gerando o mesmo resultado, exemplificado pela Figura 3.

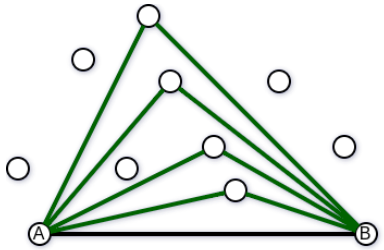

Figura 3: Formação de uma sequência

O primeiro método consiste em analisar todas as sequências de pontos válidos que podem ser formadas e, ao final da busca, determinar a maior de todas. Essa técnica é mais simples, porém mais custosa, tendo em vista que todos os possíveis candidatos à solução são testados.

Em contrapartida, o segundo algoritmo consiste em analisar as maiores sequências formadas iniciando em cada ponto, combinando o ponto válido com sua maior sequência local a fim de encontrar a maior sequência global.

2.4 Salvar o tamanho da sequência no arquivo de saída

Com o resultado obtido da etapa anterior, o próximo e último passo é salvar, usando a função **fprintf()** também da biblioteca **stdio.h**, o número inteiro que define o tamanho da sequência no arquivo de saída, cujo endereço fora fornecido pelo usuário no momento da execução do código.

3 Pontos e Vetores

Os pontos e os vetores são os elementos fundamentais para o funcionamento do algoritmo, portanto, estruturou-se um tipos abstratos de dados para cada um, facilitando o armazenamento das informações sobre esses elementos e as operações que podem ser realizadas.

```
// pointlib.h

typedef int Coordinate;

typedef struct {
    Coordinate x, y;
} Point;

typedef struct {
    Coordinate i, j;
} Vector;
```

Em um plano cartesiano, os componentes geométricos possuem coordenadas, indicando a posição relativa dos pontos em relação à âncora mais à esquerda. Dessa forma, consegue-se ter uma referência da localização de cada ponto inserido, tornando possível a resolução do problema.

3.1 Verificação dos pontos

A verificação dos pontos é feita pelo método matemático chamado produto vetorial, em que se analisa o seno do ângulo resultante entre vetores. De acordo com as propriedades desse método, quando é realizada a operação entre um vetor referência e um vetor que está à esquerda desse vetor referência, o resultado da operação será um valor maior que zero e, se o vetor estiver à direita, o resultado será um valor menor que zero.

Portanto, para resolver o problema proposto, o produto entre o vetor que representa o lado esquerdo do triângulo formado ($\vec{AC}$) e o vetor que liga o ponto P ao ponto A ($\vec{AP}$) deve resultar em um valor negativo e, de maneira análoga, o produto entre o vetor que representa o lado direito do mesmo triângulo ($\vec{BC}$) e o vetor que liga o mesmo ponto P a B ($\vec{BP}$), deve resultar em um valor positivo. Assim, haverá a comprovação geométrica de que o ponto analisado está "dentro"do triângulo formado (A, B, C).

A Figura 4 contém as três possibilidades para a verificação de um ponto, sendo, na ordem, o ponto estar à esquerda dos dois vetores, entre os dois e à direita.

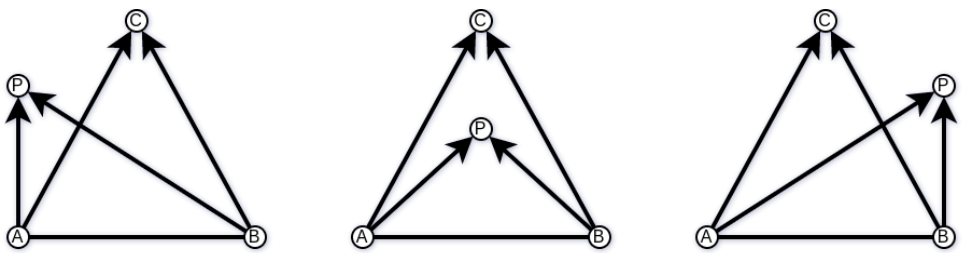

Figura 4: As três possibilidades de posições relativas ente dois pontos

4 Sequências

Após definir como será realizada a validação dos pontos, resta apenas criar uma função para encontrar a maior sequência de pontos válidos, sendo a principal etapa de todo o programa. Como dito anteriormente, foram elaborados dois algoritmos capazes de solucionar o problema, o primeiro, com uma lógica mais simples iterativa recursiva e o segundo mais complexo apenas iterativo, cada um com suas vantagens e desvantagens.

4.1 Algoritmo de força bruta (com *backtracking*)

A solução mais elementar para encontrar a maior sequência é testar todas as sequências possíveis, armazenando aquela de maior tamanho. Esse método, conhecido como "busca por força bruta", é muito utilizado em diversos campos da computação, tornando possível encontrar, com exatidão, a resposta para um problema em um grupo limitado de candidatos à solução.

O fluxograma lógico da função é descrito na Figura 5, onde há um ciclo que se inicia em um conjunto N de pontos, realizando uma nova chamada da função para todos os N-1 pontos restantes. Vale mencionar que para o funcionamento do algoritmo não é necessário que os pontos estejam ordenados por algum critério.

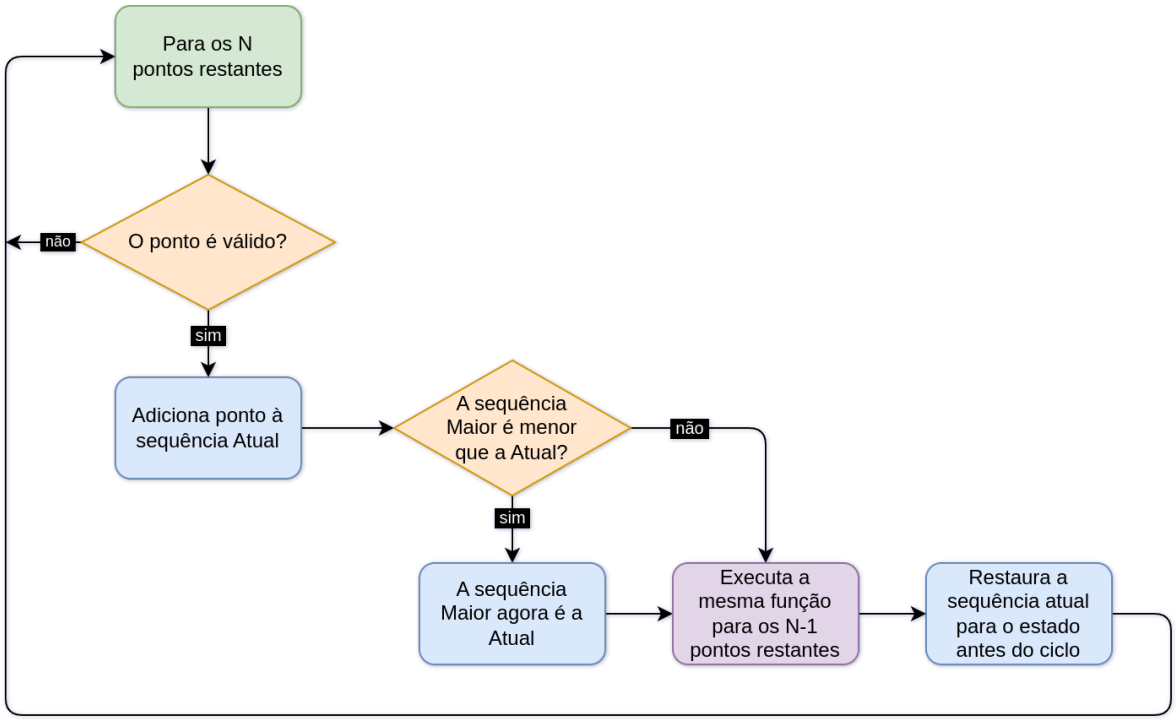

Figura 5: Fluxograma do Algoritmo de Força Bruta

Enquanto um algoritmo de força bruta testa todas as sequências possíveis sem exceção, o uso do *backtracking* permite descartar automaticamente uma sequência assim que um elemento inválido é encontrado, melhorando a eficiência do programa.

Porém uma desvantagem desse método é a grande quantidade de chamadas recursivas, muitas vezes de forma desnecessária, tendo em vista que sequências grandes incluem, dentro de si, muitas sequências menores, levando a um aumento drástico do tempo de execução, o que torna o algoritmo ineficiente para grandes conjuntos de pontos.

4.1.1 Poda (*Branch and Bound*)

Uma possível modificação na implementação do algoritmo de força bruta, que muda de maneira significativa o desempenho do programa, é adicionar uma verificação que ajuda a evitar comparações desnecessárias. Para tal, é necessário interromper a execução de uma busca em uma subsequência que, devido à quantidade de pontos restantes, não consegue superar a maior sequência registrada até o momento, assim como mostra a Figura 6.

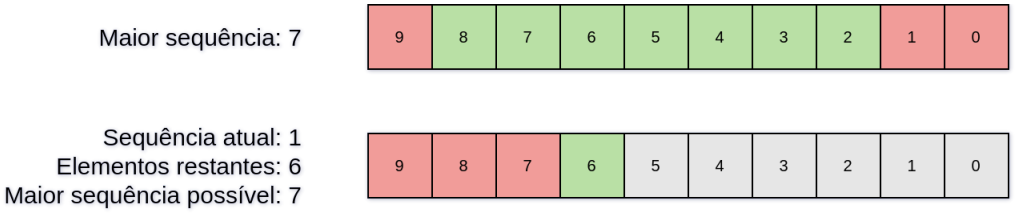

Figura 6: Representação do funcionamento da poda

Essa lógica, apesar de parecer complexa, é implementada de maneira simples, com apenas uma estrutura condicional que irá executar o comando **break** para interromper o laço repetição caso a condição seja satisfeita. No exemplo abaixo, o tamanho da sequência atual somado ao número de pontos restantes deve ser maior que o o tamanho da maior sequência para continuar o processo de busca.

```
// ...
    for (int i = n; i >= 0; i--) {
        if (longestPath->length >= activeSequence->length + (i + 1)) {
            break;
        }
// ...
```

Apesar da aplicação da técnica *Branch and Bound* não reduzir a ordem de complexidade do pior caso do algoritmo, a melhoria proporcionada em relação ao tempo de execução é facilmente perceptível, como demonstrado nos testes da seção 6.1.1.

4.2 Algoritmo de programação dinâmica

Um algoritmo de programação dinâmica consiste em formular soluções locais ótimas em cada etapa da busca com o objetivo de encontrar uma solução global ótima. Trazendo para o contexto do problema, esse método é aplicado para encontrar as maiores sequências formadas por cada ponto e no final formar a sequência que conecta mais pontos válidos.

A grande vantagem dessa implementação é a eficiência computacional, uma vez que, diferente do método de força bruta, não trabalha testando todas as possíveis soluções, mas sim formando a melhor solução a partir de cada iteração, realizando o processo descrito no fluxograma da Figura 7.

É importante destacar que, para o funcionamento do algoritmo, os pontos necessariamente precisam estar ordenados em relação à altura (coordenada Y), isso pode ser feito no momento em que os pontos são recebidos do arquivo de entrada.

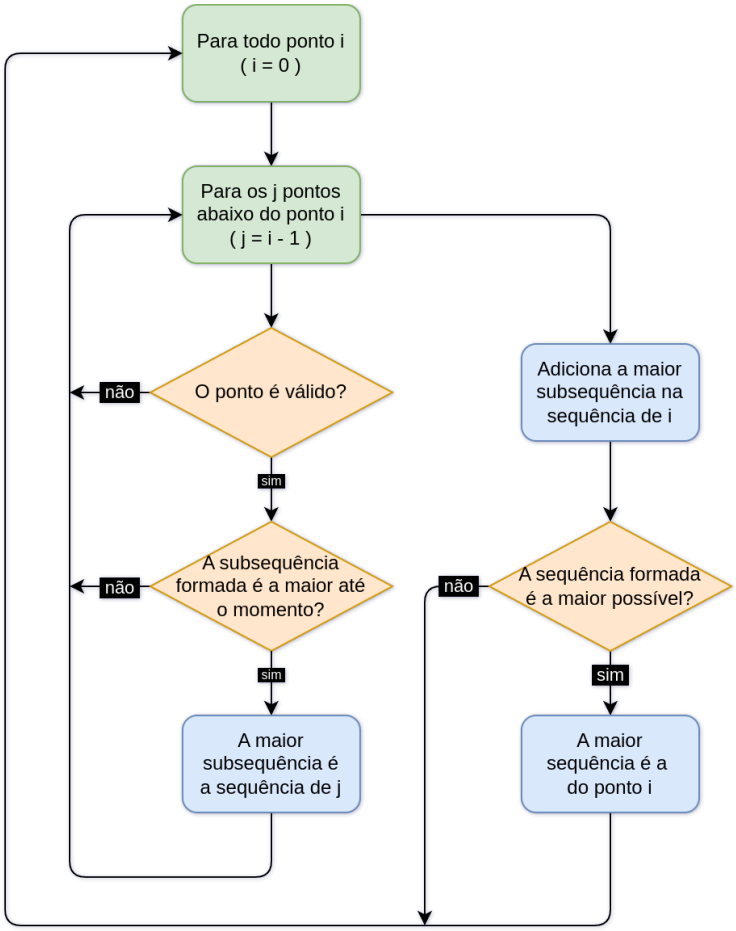

Figura 7: Fluxograma do Algoritmo de Programação Dinâmica

5 Análises matemáticas

De acordo com a seção anterior, a diferença dos dois algoritmos criados, além da lógica para encontrar a solução, se evidencia nos tempos de execução, tendo em vista que testar todas as possibilidades, apesar da simplicidade de funcionamento, acaba não sendo vantajoso em determinadas situações.

5.1 Força bruta

O algoritmo de força bruta realiza comparações que variam a depender da disposição dos pontos, possuindo então casos melhores, piores e médios, assim como mostra a Tabela 1.

CASO	TOTAL DE COMPARAÇÕES	COMPLEXIDADE
Melhor	$N^2/2 - N/2$	$O(N^2)$
Pior	$2^N - 1$	$O(2^N)$
Médio/Esperado	$2^{N/2} - 1$	$O(2^N)$

Tabela 1: Tabela de complexidade do algoritmo de força bruta

5.1.1 Melhor caso

O melhor caso do algoritmo de força bruta ocorre quando não há pontos válidos entre si, formando uma sequência de tamanho 1. Para poder afirmar isso em um conjunto de N pontos, para cada ponto são realizadas comparações com os N-1 pontos restantes, que resultam no somatório abaixo:

$$0 + 1 + 2 + ... + N - 2 + N - 1 = \sum_{i=0}^{N-1} a_i$$

Essa soma pode ser classificada como uma progressão aritmética, cuja fórmula da soma dos elementos é dada por:

$$S_N = \frac{N(a_1 + a_N)}{2}$$

Substituindo os elementos na fórmula, obtém-se a soma total de todas as comparações realizadas e, assim, a função de complexidade:

$$S_N = \frac{N(0 + N - 1)}{2} = \frac{N^2}{2} - \frac{N}{2} = O(N^2)$$

5.1.2 Pior caso

Dado um conjunto de N pontos, para cada ponto há a possibilidade de pertencer ou não à maior sequência, ou seja, existem duas possibilidades para cada ponto, resultando em:

$$2 \times 2 \times 2 \times ... \times 2 = \prod_{i=1}^{N} 2 = 2^N$$

Esse é o número total de sequências possíveis de serem formadas para encontrar maior sequência no pior caso, que ocorre quando todos os pontos são válidos entre si. No entanto, um ponto forma uma sequência consigo mesmo de tamanho 1, o que garante a existência de ao menos uma sequência global.

$$2^N - 1 = O(2^N)$$

Devido a essa quantidade de comparações, o algoritmo de força bruta é classificado como exponencial. Isso significa que o tempo de execução do algoritmo aumenta exponencialmente à medida que o tamanho do conjunto de pontos de entrada aumenta.

5.1.3 Caso médio/esperado

Classificar o caso médio é uma tarefa difícil para esse tipo problema, tendo em vista que a disposição dos pontos não é aleatória. Porém, é possível admitir que os pontos tenham uma probabilidade igual a 0.5 de serem válidos ou não uns com os outros, portanto, um caso médio seria aquele no qual a maior sequência em um conjunto de N elementos tenha tamanho igual à metade de N, realizando o total de comparações subtraído de uma unidade, como no pior caso.

$$2 \times 2 \times 2 \times ... \times 2 = \prod_{i=1}^{N/2} 2 = 2^{N/2}$$

$$2^{N/2} - 1 = \sqrt{2^N} - 1 = O(2^N)$$

5.2 Algoritmo de programação dinâmica

Ao contrário do anterior, o algoritmo de programação dinâmica não possui diferentes casos de execução, dessa forma, para encontrar a maior sequência em um conjunto de N pontos são realizadas comparações entre cada ponto I e todos os pontos J abaixo de I, resultando na soma de todos os elementos entre 0 e N-1:

$$0 + 1 + 2 + ... + N - 2 + N - 1 = \sum_{i=0}^{N-1} a_i$$

Como visto anteriormente, essa progressão aritmética pode ser simplificada como:

$$\frac{N^2}{2} - \frac{N}{2} = O(N^2)$$

A complexidade quadrádica, que independe da disposição dos pontos, se mostra muito mais eficiente que a exponencial obtida no cálculo da complexidade envolvendo força bruta, tendo em vista que o número de pontos definidos no arquivo de entrada pode chegar a 100. A comparação é simples:

$$100^2 < 2^{100}$$

$$10^4 < 1.2676506 \times 10^{30}$$

Na próxima seção são descritos os testes realizados que comprovam a larga diferença de execução entre os algoritmos desenvolvidos.

6 Testes de Software

Os testes estiveram presentes em todas as etapas do desenvolvimento do software, desde o funcionamento das funções mais elementares até o tempo de execução da função para encontrar a solução do problema, garantindo uma aplicação performática e de fácil manutenção.

6.1 Monitoramento do tempo de execução

Para medir o tempo de execução total da função que encontra a maior sequência de pontos válidos, a biblioteca **getrusage.h** foi escolhida pois fornece diversas informações úteis, incluindo o tempo total de processamento do programa. Isso é uma grande vantagem em relação à biblioteca **gettimeofday.h**, que fornece apenas o tempo físico gasto, podendo apresentar inconsistências na medição.

Um gerador aleatório de pares de números inteiros foi usado para facilitar o processo de geração das entradas, possibilitando uma maior gama de testes e resultados mais fidedignos.

O tempo decorrido de cada execução foi armazenado em um arquivo de saída, que acrescenta os resultados de cada execução sem apagar os dados previamente inseridos. Cada linha contém o total de pontos, o tamanho da maior sequência e o tempo decorrido, nesta exata ordem.

```
// Arquivo de saída dos resultados obtidos

100 16 0.021185
100 17 0.136639
100 20 0.248232
100 19 0.165011
```

6.1.1 Resultados e análises

Os programas desenvolvidos, após finalizados, foram testados visando comparar a maneira em que o tempo de execução varia de acordo ao total de pontos. O primeiro teste usou o pior caso do algoritmo de força bruta, que, como mostra os dados do gráfico (Figura 8), se torna inviável para quantidades acima de 30 pontos, demorando mais de 60 segundos para computação do resultado de cada entrada.

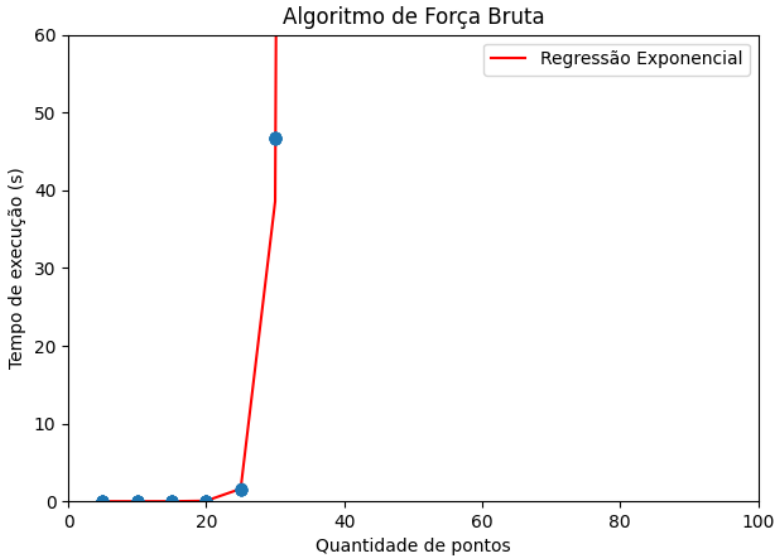

Figura 8: Gráfico dos tempos de execução do algoritmo de força bruta

Conforme apresentado na seção 4.1.1, a técnica de poda pode ser utilizada para melhorar significativamente o desempenho do algoritmo anterior, tornando-o viável para conjuntos grandes de pontos. No entanto, a efetividade desse método depende da disposição dos pontos no espaço, variando muito os tempos de execução, como é possível observar na Figura 9.

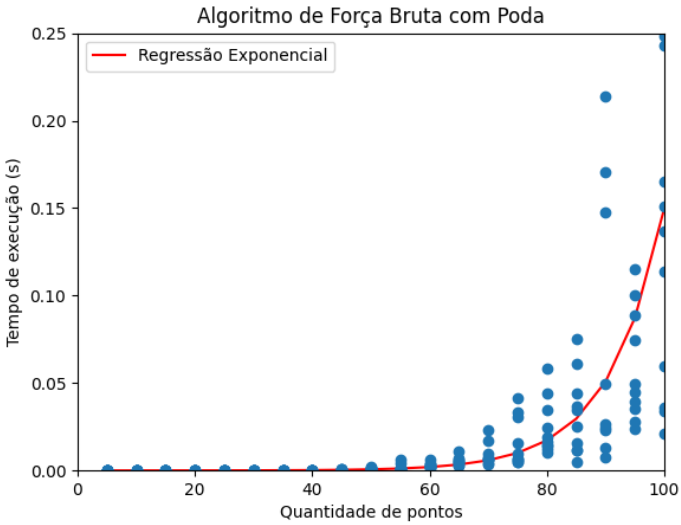

Figura 9: Gráfico dos tempos de execução do algoritmo de força bruta com poda

Por outro lado, o gráfico das execuções do algoritmo de programação dinâmica (Figura 10) se mostra mais uniforme, visto que sua complexidade é quadrádrica. Importante destacar que nos testes realizados o tempo total de execução não excedeu 0,15 milissegundos.

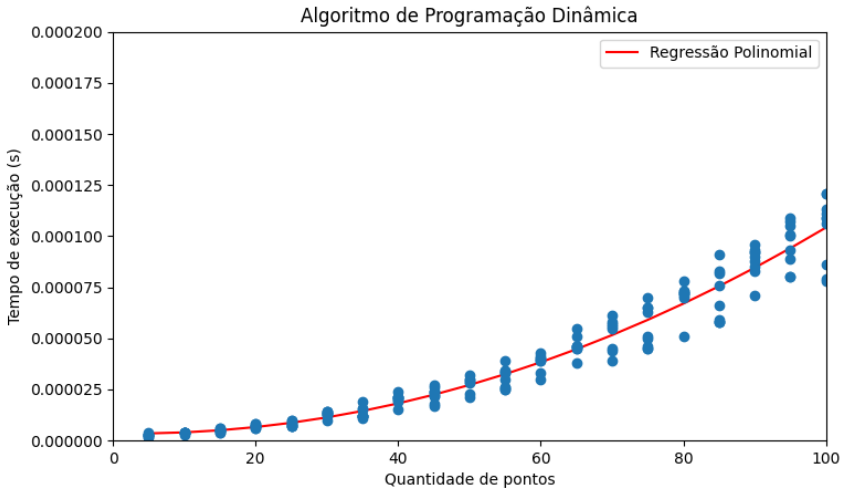

Figura 10: Gráfico dos tempos de execução do algoritmo de programação dinâmica

6.2 Visualização gráfica dos pontos

A biblioteca **SDL** (Simple DirectMedia Layer), não nativa do C, é uma biblioteca de código aberto que dá suporte ao desenvolvimento multi-plataforma de softwares que exigem a renderização de elementos, como interfaces gráficas e jogos.

Dessa forma, conhecendo a maior sequência de pontos válidos, é possível exibir graficamente os pontos e as retas formadas em uma janela de renderização, como exibido na Figura 11, tornando a aplicação muito mais completa.

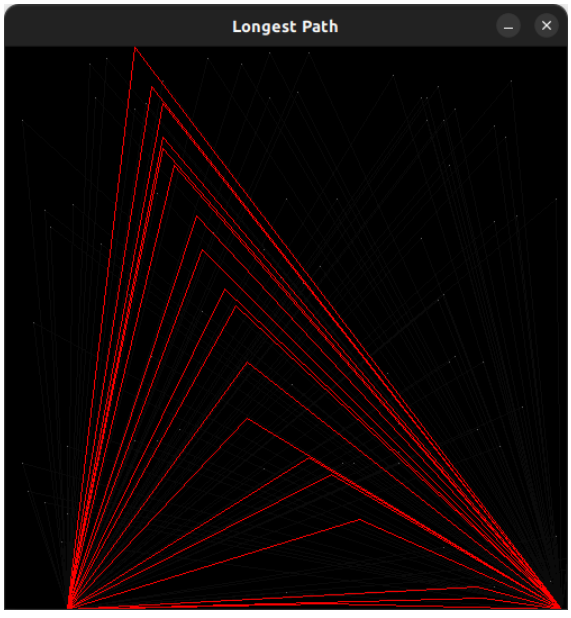

Figura 11: Janela de visualização dos pontos

7 Conclusão

Ao comprovar as análises matemáticas realizadas previamente durante os testes, fica evidente a importância de considerar várias soluções para um mesmo problema. Isso se deve ao fato de que um algoritmo pode enfrentar um caso de execução muito desfavorável, podendo tornar-se inviável devido ao tempo total decorrido para encontrar a resposta.

Para o problema em específico, o algoritmo de força bruta, apesar de ser a solução mais simples, não consegue lidar com um conjunto grande de pontos a depender de sua distribuição, podendo demorar séculos (literalmente) para resolver um problema que o algoritmo de programação dinâmica consegue resolver de forma instantânea. No entanto, é importante ressaltar que a busca por força bruta não deve ser totalmente descartada. Com a implementação de melhorias adicionais, como a técnica de poda descrita na seção 4.1.1, o programa se torna capaz de lidar com um número maior de pontos, apresentando a solução em um tempo satisfatório.

Em todas as áreas do desenvolvimento de software, garantir a rapidez e eficiência dos algoritmos é uma tarefa crucial de todo programador. A redução do tempo de processamento não é importante apenas para melhorar a experiência do usuário com a aplicação, mas também pode reduzir significativamente os custos de operação. Quanto mais eficiente um algoritmo, menor é a quantidade de recursos necessários para executá-lo.

Além disso, é importante destacar que a geometria computacional possui diversas aplicações práticas, tais como na modelagem de objetos tridimensionais, computação gráfica, dentre outras. Portanto, o estudo e desenvolvimento de algoritmos eficientes para esse tipo de problema pode ter impactos significativos em diferentes áreas da ciência e tecnologia.

Referências

[1] Thomas H. Cormen. *Algoritmos: Teoria e prática*. LTC, 2012.

[2] Orunmila. Modular code and how to structure an embedded C project. https://www.microforum.cc/blogs/entry/46-modular-code-and-how-to-structure-an-embedded-c-project/, 2019.

[3] Jayme L. Szwarcfiter and Lilian Markenzon. *Estruturas de Dados e Seus Algoritmos 3ª edição*, volume 53. GEN, 2015.

[4] Nishtha Thakur. getopt() function in c to parse command line arguments. https://www.tutorialspoint.com/getopt-function-in-c-to-parse-command-line-arguments, 2019.