Universidade Federal de São João del-Rei

Hipercampos

Algoritmos e Estruturas de Dados III

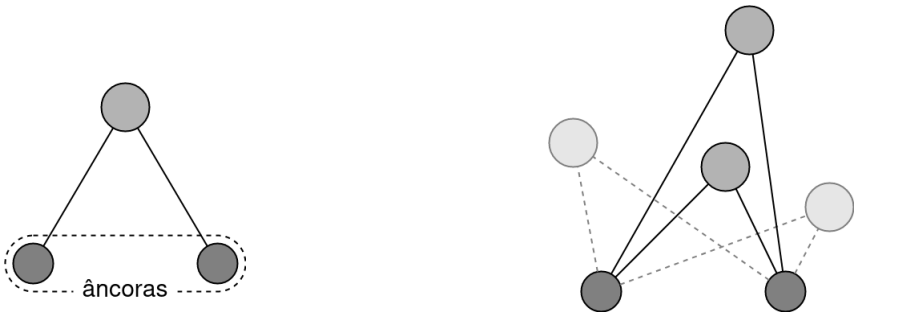
Giancarlo Oliveira Teixeira
giancarloprados@aluno.ufsj.edu.br

Wasterman Ávila Apolinário
watermanavila@gmail.com

1 Apresentação do problema

O problema é desenvolvido com base em um conjunto de pontos. São especificadas duas âncoras, que podem ser ligadas aos demais pontos por meio de arestas. Em cada **ligação**, o ponto envolvido é ligado às duas âncoras por meio de duas aresta (Figura 1a).

Dessa forma, a tarefa é desenvolver um algoritmo que encontre o número máximo de ligações que podem ser feitas de forma que as arestas que ligam diferentes pontos interceptem-se apenas nas próprias âncoras (Figura 1b).



(a) *Modelo de ligação* (b) *Exemplo de encadeamento máximo*

Figura 1: *Casos explicativos iniciais*

1.1 Motivação prática

Em primeira análise, o modelo gerado para o problema de hipercampos parece estar relacionado a casos de maximização. Contudo, com certas modificações, ele parece ser mais útil se aplicado com intuito de minimização e otimização. Isso porque um grande encadeamento de polígonos em determinado sistema implica que uma quantidade proporcionalmente grande de pontos compartilham caraceterísticas em comum - e reduzir esse compartilhamento pode ser a chave para a solução de determinadas situações.

Tenha como base a instalação de câmeras para vigilância da entrada de um banco. É natural pensar que uma maior a quantidade de dispositivos instalados ocasiona diretamente uma maior segurança. Contudo, é preciso pensar, por exemplo, que não há utilidade em dispor várias câmeras em lugares muito próximos, já que seus campos de visão apresentarão várias interseções, que, desnecessárias, aumentarão o custo do projeto e acarretarão a perda de um considerável potencial de cobertura. Como exemplo, tenha a disposição das câmeras 1 e 2 da Figura 2.

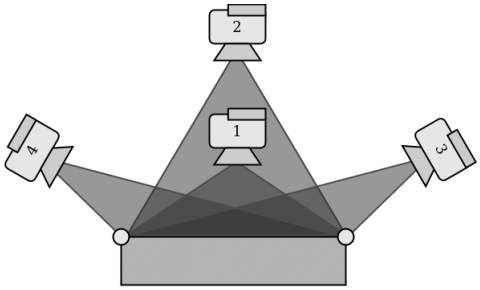


Figura 2: *Representação de algumas câmeras com base nos hipercampos*

Portanto, quanto maior for o encademanto de ligações, menos eficiente tende a ser a organização aplicada no sistema. Assim, generalizando esse conceito, algum algoritmo com funcionamento próximo ao usado no problema de hipercampos poderia ser aplicado para testar a eficiência de disposição de dispostivos em sistemas de monitoramento, produção visual e até para otimização em computação gráfica (em uma malha, vértices encapsulados por outros poderiam deixar de ser renderizados, a fim de diminuir a quantidade de processamento realizada, por exemplo). Logo, a construção de um algoritmo eficaz e minimamente eficiente para a resolução da problemática apresentada inicialmente é algo que deve ser levado em consideração - e foi a base para o desenvolvimento deste trabalho.

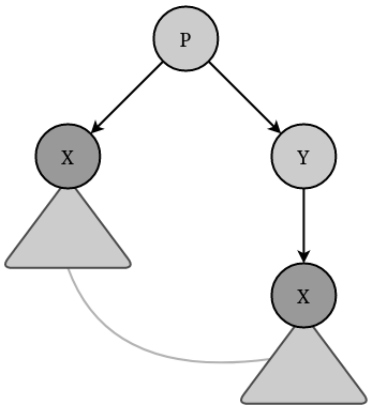
2 Propostas de solução

Em uma abordagem de força bruta, a solução consiste em testar todas as arranjos possíves de pontos e determinar qual o número máximo de ligações formadas de acordo com a restrição de interseção. Porém, é preciso levar em consideração que pode-se escolher de i $(1 \leq i \leq n)$ elementos para formar um subconjunto de pontos. Além disso, para cada quantidade escolhida, existem diversas combinações possíveis. Igualando o custo P do algoritmo à quantidade de possibilidades analisas, tem-se, então, que, para essa abordagem,

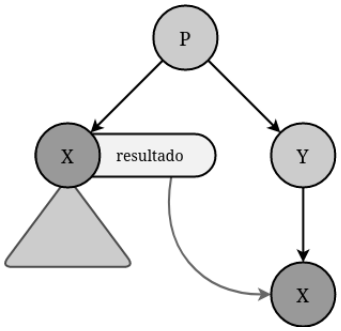
$$P(n) = \sum_{i=0}^{n} C_{n,i} - C_{n,0} == 2^n - 1 \tag{1}$$

Esse resultado é atingido usando-se a Série Binomial [Stewart, 2016, p. 686] (fazendo $x = y = 1$) e indica que, apesar de simples, essa abordagem, baseada em uma interpretação natural e imediata do problema, não é computacionalmente viável, uma vez que responde exponencialmente ao crescimento do número de pontos.

Para resolver o problema, portanto, foi necessário estruturar outro modelo. Para entender como isso foi feito, é preciso perceber que a maior parte da complexidade do algoritmo força bruta é gerada pela construção e computação de sequências repetidas. Logo, uma opção de otimização é utilizar o armazenamento e reaproveitamento de resultados já obtidos. Como mostrado na Figura 3, essa abordagem é possível de ser implementada, e o reaproveitamento de resultados foi, então, definido como o pilar de construção do programa.



(a) *Fluxo de computação sem reaproveitamento. Todo um subproblema é recomputado.*



(b) *Fluxo de computação com reaproveitamento. Um resultado já obtido é reutilizado.*

Figura 3: *Estratégias de computação. O ponto indicado por uma seta pode ser englobado pelo ponto de origem.*

3 Funcionamento do algoritmo

3.1 Programação dinâmica

Fundamentalmente, a programação dinâmica visa alcançar uma solução ótima por meio da combinação de subsoluções ótimas. Para isso, o problema original é quebrado em subproblemas, cujas soluções são armazenadas, de forma que, em vez de tornar-se refém de recomputações, o algoritmo pode simplesmente reutilizar soluções já encontradas. Teoricamente, todas as combinações possíveis ainda são testadas, mas, na prática, a computação não adquire um custo exponencial.

Como pode ser visto na Figura 3b, o problema de hipercampos possui potencial para ser resolvido com esse tipo de estratégia - e, como será ainda abordado, o resultado pode ser obtido de forma relativamente simples. Para usar esse tipo de solução, cada ponto foi representado como uma estrutura, à qual é atribuída um peso (a quantidade ótima de triângulos que podem

ser encadeados abaixo desse ponto). Dessa forma, quando for novamente verificado, esse ponto cederá diretamente esse peso, que poderá ser utilizado para outras análises.

3.2 Construção da solução

Como mostrado na Figura 3, o algoritmo poderia ser desenvolvido com uma abordagem **top-down**. Contudo, essa estratégia possui uma natureza recursiva, e, portanto, poderia ter uma análise mais complexa, já que envolveria um número maior de chamadas do sistema operacional e precisaria, normalmente, de mais recursos secundários, como memória. Portanto, decidiu-se utilizar uma abordagem **bottom up**.

Para entendê-la, basta notar que, basicamente, cada ponto pode englobar apenas pontos que estejam abaixo dele. Portanto, em vez de uma análise desordenada, é possível ordenar os pontos em ordem crescente da coordenada y e construir a solução a partir dos pontos inferiores, desenvolvendo-a à medida que o y dos pontos aumenta. Esse funcionamento é ilustrado na Figura 4 e terá seu funcionamento prático analisado na seção 5.

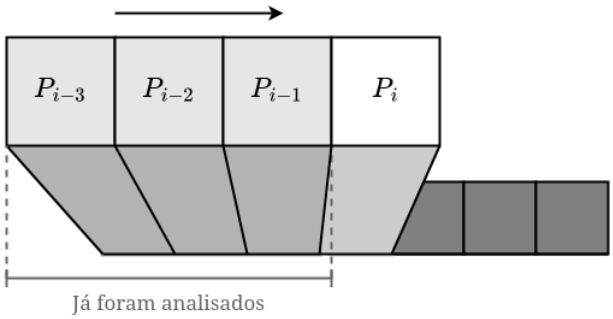


Figura 4: *Funcionamento da abordagem bottom-up*

3.2.1 Ordenação dos pontos

Como explicado anteriormente, nossa solução exige que os pontos estejam ordenados em ordem crescente. Como, por padrão, a quantidade de pontos foi limitada a 100, optamos por desenvolver essa tarefa com o uso de um algoritmo simples: o selection sort. Sua complexidade é $O(n^2)$, o que pode trazer problemas se usado em conjuntos grandes. Contudo, para a entrada especificada para nosso programa, esse custo é aceitável - e é compensado pela facilidade durante a escrita do código.

3.3 Verificação do encadeamento

Direcionando a atenção à parte mais fundamental do problema, todo o fluxo do programa depende de uma verificação: para calcular o encadeamento, é necessário assumir que um ponto pode englobar outro, e, para isso, é preciso garantir que as arestas de suas ligações não se interceptam fora das âncoras.

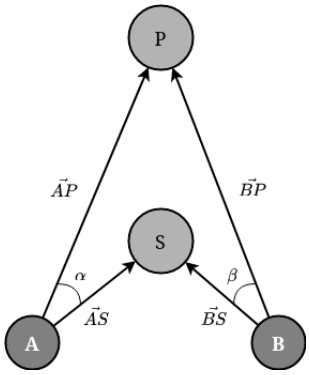


Figura 5: *Ilustração da posição relativa de vetores*

Tenha como exemplo a Figura 5. É possível notar que dados dois pontos P e S ($y_P > y_S$), S não intercepta as areastas de P apenas se estiver entre elas. Dessa forma, dadas as âncoras A e B ($x_A < x_B$), $\vec{AS}$ precisa estar à direita de $\vec{AP}$ e $\vec{BS}$ precisa estar à esquerda de $\vec{BP}$.

Para encontrar a posição relativa dos vetores, foi utilizado o cálculo do produto vetorial. Resumidamente, o vetor resultado desse produto será simulteneamente ortogonal aos outros dois vetores envolvidos. Nesse caso, dados $\vec{u} = < a, b >$ e $\vec{v} = < c, d >$, tem-se que, pelo cálculo do determinante do produto vetorial,

$$\vec{u} \times \vec{v} = < 0, 0, k > \qquad k = (ad - bc) \tag{2}$$

Pela equação do módulo do produto vetorial e pela Equação 2, tem-se que

$$|\vec{u} \times \vec{v}| = |\vec{u}| \cdot |\vec{v}| \cdot \sin\theta \tag{3}$$

$$\sqrt{0^2 + 0^2 + k^2} = |\vec{u}| \cdot |\vec{v}| \cdot \sin\theta \tag{4}$$

$$\sqrt{k^2} = |\vec{u}| \cdot |\vec{v}| \cdot \sin\theta \tag{5}$$

$$k = |\vec{u}| \cdot |\vec{v}| \cdot \sin\theta \tag{6}$$

Nessa equação, $|\vec{u}|$ e $|\vec{v}|$ são sempre positivos. Além disso, normalmente, θ é considerado como o menor ângulo positivo formado entre os vetores. Contudo, fixando esse ângulo como aquele formado de $\vec{u}$ para $\vec{v}$, ele será o único componente possivelmente negativo da equação, e é possível analisar que

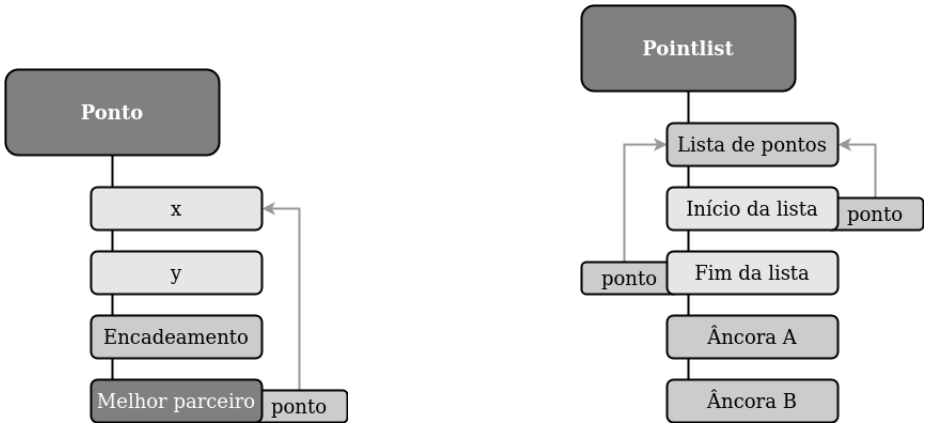
$$k > 0 \quad \longleftrightarrow \quad \theta < 0 \quad \longleftrightarrow \quad \vec{u} \text{ está à esquerda de } \vec{v} \tag{7}$$

$$k < 0 \quad \longleftrightarrow \quad \theta > 0 \quad \longleftrightarrow \quad \vec{u} \text{ está à direita de } \vec{v} \tag{8}$$

4 Estrutura do programa

4.1 Estruturas de dados

O funcionamento principal do programa é baseado em duas estruturas: uma lista de pontos e o próprio ponto. Pela brevidade, um resumo dessas construções é apresentado na Figura 6.



(a) *Estrutura do ponto* (b) *Estrutura da lista de pontos*

Figura 6: *Ilustração das estruturas utilizadas*

4.2 Rotinas

Por sua simplicidade, o programa não utiliza uma grande quantidade de rotinas. Além disso, no geral, o funcionamento de cada função é autoexplicativo. Elas são listadas na Tabela 1.

Função	Ordem de complexidade	Foco da análise
get_data (argc, argv[ ])	$O(1)$	Alocações
read_list_size (intputFile)	$O(1)$	Leituras
read_list (pointlist)	$O(n)$	Leituras
create_list (size)	$O(1)$	Alocações
erase_list (pointlist)	$O(1)$	Desalocações
gen_std_output (outputFile)	$O(1)$	Escritas
new_vector (point A, point B)	$O(1)$	Cálculos
relative_position ($\vec{u}, \vec{v}$)	$O(1)$	Comparações
sort_list (pointlist)	$O(n^2)$	Comparações
get_max_chaining_point (pointlist)	$O(n^2)$	Comparações

Table 1: *Rotinas do programa e suas ordens de complexidade*

Como as demais funções têm peso menor no programa e são autoexplicativas, a função *get_ max_ chaining_ point* será a única a receber, aqui, maior enfoque. Sua utilidade é calcular

o tamanho de encadeamento iniciado por cada ponto e indicar aquele que possui o tamanho de encadeamento máximo[1]. Para retornar esse valor, portanto, basta acessá-lo por meio da estrutura do ponto armazenado. Para encontrar esse ponto ótimo, é utilizado o Algoritmo 1.

Algoritmo 1: Algoritmo que encontra um hipercampo máximo

```
1  Function GetMaxChainingPoint(pointlist) :
2      betterChainingPoint = pointlist.begin
3      for point in pointlist do
4          betterPartner = ϕ
5          for partner < point do
6              if partner.chain > betterPartner.chain then
7                  betterPartner = partner;
8          if betterPartner.chain > betterChainingPoint.chain then
9              betterChainingPoint = betterPartner;
10     return betterChainingPoint;
```

5 Análise de Complexidade

Para realizar a análise de complexidade do algoritmo, haveriam duas abordagens gerais possíveis: analisar todas as funções utilizadas ou fazer a análise apenas daquelas com maior contribuição para o custo final. Em nosso programa, as rotinas com processamento mais expressivo são aquelas ligadas à ordenação dos pontos e computação direta dos hipercampos, e as demais são, basicamente, funções de entrada, saída e alocação de dados (tarefas cujo custo, no geral, não varia de forma relevante em relação ao tamanho da entrada). Portanto, a segunda opção de análise foi escolhida.

Em relação à análise específica das funções principais, optou-se por analisar o pior caso de execução. Essa abordagem foi escolhida não só porque é mais prática e viável, mas também porque, se não representamos com exatidão o custo mediano do algoritmo, pelo menos conseguimos garantir que ele não torna-se mais custoso que determinado limiar. Além disso, a quantidade de comparações foi tomada como medida de custo tanto para a tarefa de ordenação quanto para o cálculo dos hipercampos (cada um com suas peculiaridades).

5.1 Ordenação

Como dito anteriormente, a ordenação da lista de pontos é uma tarefa imprescindível para o funcionamento do programa e, como é uma tarefa geralmente custosa, será levada em consideração para o cáluclo da complexidade total.

Para ordenar a lista de pontos, foi utilizado o *selection sort*. Como esse algoritmo é bastante conhecido, sua análise aprofundada não será abordada neste trabalho. É importante ressaltar

[1] Mais de um ponto pode receber o número de encadeamento máximo. Nesse caso, será retornado o primeiro que for encontrado

apenas que, em relação ao número de comparações, a função de complexidade dessa rotina é dada, no pior caso, por

$$g(n) = \frac{1}{2}(n^2 - n) \tag{9}$$

[Knuth, 1994] e, portanto, sua ordem de complexidade é $O(n^2)$.

5.2 Cálculo dos hipercampos

Como mostrado brevemente na Seção 3.2, para cada ponto analisado durante a construção dos hipercampos, é iniciada uma análise para todos os seus pontos inferiores. Nesse caso, dado o ponto P_i $(1 \leq i \leq n)$ da lista ordenada, a quantidade de pontos que participam da análise será dada por

$$Q_i = i - 1 \tag{10}$$

Uma vez que a análise é feita para todos os pontos, a quantidade total de análises será igual a

$$Q(n) = \sum_{i=1}^{n} Q_i = 0 + 1 + 2 + \cdots + (n-1) = \frac{1}{2}n(n-1) \tag{11}$$

Como é possível ver, essa é a soma de uma progressão aritmética cujo número de termos é n, cujo termo inicial é $a_1 = 0$ e o termo final, $a_n = n-1$. Logo, pela fórmula de soma da progressão,

$$Q(n) = \frac{n(n-1-0)}{2} = \frac{1}{2}(n^2 - n) \tag{12}$$

Logo, a função de complexidade da função principal é dada por

$$h(n) = \frac{1}{2}(n^2 - n) \tag{13}$$

5.3 Complexidade total

As tarefas citadas até aqui são aquelas com maior contribuição do peso do algoritmo. Portanto, com as premissas feitas, consideramos que a função de complexidade do programa como um todo pode ser definida como

$$f(n) = g(n) + h(n) = 2 \cdot \frac{1}{2}(n^2 - n) \tag{14}$$

$$f(n) = n^2 - n \tag{15}$$

Por fim, a ordem de complexidade total é $O(n^2)$.

6 Análise Prática

Com o conhecimento da equação de complexidade das principais rotinas em mãos, tornou-se interessante realizar uma análise prática do desempenho do algoritmo. Para isso, foi necessário medir o tempo de execução do programa para diversas entradas diferentes, a fim de coletar dados que pudessem ser posteriormente analisados com a ajuda de ferramentas externas.

Para diferenciar o tempo gasto em diferentes etapas do programa, foram implementadas funções de medição de tempo. Foram medidos tanto o tempo total de execução quanto o tempo dedicado apenas à parte principal do programa. Com o uso da biblioteca C <*sys/time.h*> foi possível distinguir o tempo de entrada e saída, o tempo de processamento e o tempo total de execução do programa - ambos discutidos ao longo desta seção.

6.1 Metodologia

6.1.1 Obtenção de dados

Com o programa pronto para retornar o tempo em cada execução, foram desenvolvidos dois *scripts* em Python para auxiliar nos testes. Um deles atuou essencialmente como um gerador de pontos aleatórios (para cada quantidade de pontos n ($n \in \{50, 100, 150, \ldots, 10000\}$), foram gerados 10 arquivos diferentes). No total, somaram-se 2000 testes - uma quantidade que estrapola os limites definidos pelo problema a fim de permitir uma análise de complexidade próxima da assintótica.

O segundo algoritmo tem como objetivo executar o programa principal uma vez para cada arquivo de entrada gerado anteriormente. Durante cada execução do programa, são registrados em 6 arquivos de saída diferentes, referentes ao tipo de medição, o número n de entradas e o tempo gasto (em milissegundos) separados por ponto e vírgula, de maneira que cada linha seja referente a uma execução diferente do programa.

As medidas de tempo foram dividas em

- Tempo total de relógio
- Tempo total de usuário
- Tempo total de sistema
- Tempo de usuário da computação
- Tempo de sistema da computação
- Tempo de entrada e saída

Como o programa principal foi desenvolvido em C, o tempo de relógio foi obtido por meio do uso da função *gettimeofday()*, e os tempos de usuário e sistema, por meio da função *getrusage()*.

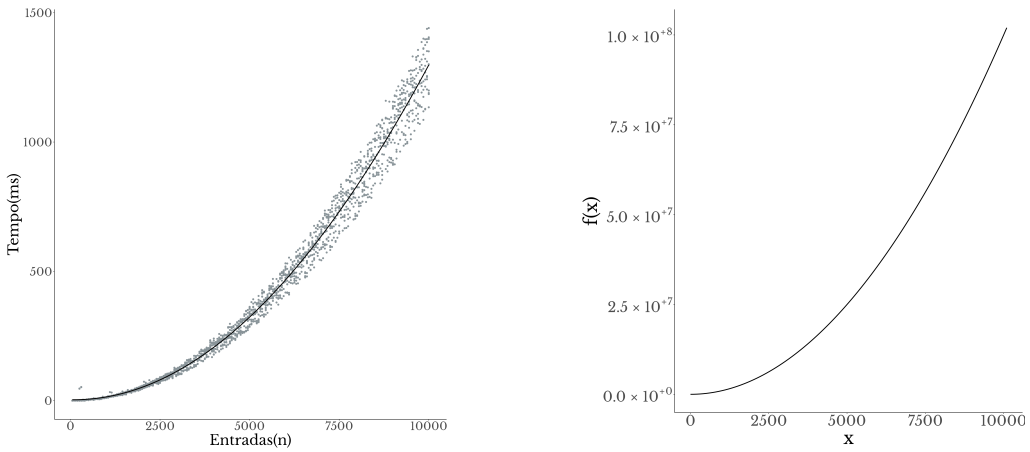
6.1.2 Tratamento dos dados

A ferramenta selecionada para o tratamento dos dados foi o RStudio. Foi desenvolvido um script em R que recebe diversos arquivos com formato pré-definido, conforme apresentado anteriormente, e gera gráficos que relacionam o número de entradas ao tempo de execução. Esses gráficos apresentam um ponto para cada dado coletado, além de uma curva de regressão que aproxima o comportamento do tempo a uma função, permitindo assim uma visualização clara dos dados e uma comparação mais fácil entre os dados.

É preciso ressaltar que, para a regressão, foi utilizado o método loess. Esse método já é implementado na biblioteca padrão da linguagem R e consegue ser mais geral do que a regressão linear padrão (baseada no cálculo dos coeficientes de uma reta). Como não é parametrizado, foi escolhido tanto por sua simplicidade quanto pela relação com a geração de polinômios (o que é consonante com a natureza da complexidade do algoritmo).

6.2 Resultados

Com os dados e gráficos disponíveis, diversas análises foram realizadas a fim de avaliar o real desempenho do algoritmo. Inicialmente, para evitar distorções na medição, foi avaliado o tempo total de usuário. Dessa forma, foi feita uma comparação entre a curva de regressão dos dados de tempo e a curva da função de complexidade encontrada, a fim de avaliar o grau de confiabilidade desta última. Os resultados, apresentados na Figura 7, mostram que, apesar da diferença de escala[2], existe uma grande semelhança entre a forma de crescimento das duas curvas, sugerindo que, assintoticamente, essa observação corrobora a afirmação da função de complexidade encontrada anteriormente.



(a) *Tempo total de usuário registrado* (b) *Função de complexidade*

Figura 7: *Comparação lado a lado entre a função calculada teoricamente e a função de tempo estimada com base na regressão*

[2]A escala logarítmica foi usada a fim de facilitar a visualização da curva.

Avançando a análise, o gráfico da Figura 8 apresenta duas curvas de regressão sobrepostas, uma referente ao tempo total de usuário e outra referente ao tempo de processamento das duas principais funções. É possível perceber que a diferença entre as medições é mínima, e, portanto, que as funções de entrada e saída do programa são praticamente dispensáveis para a análise de tempo.

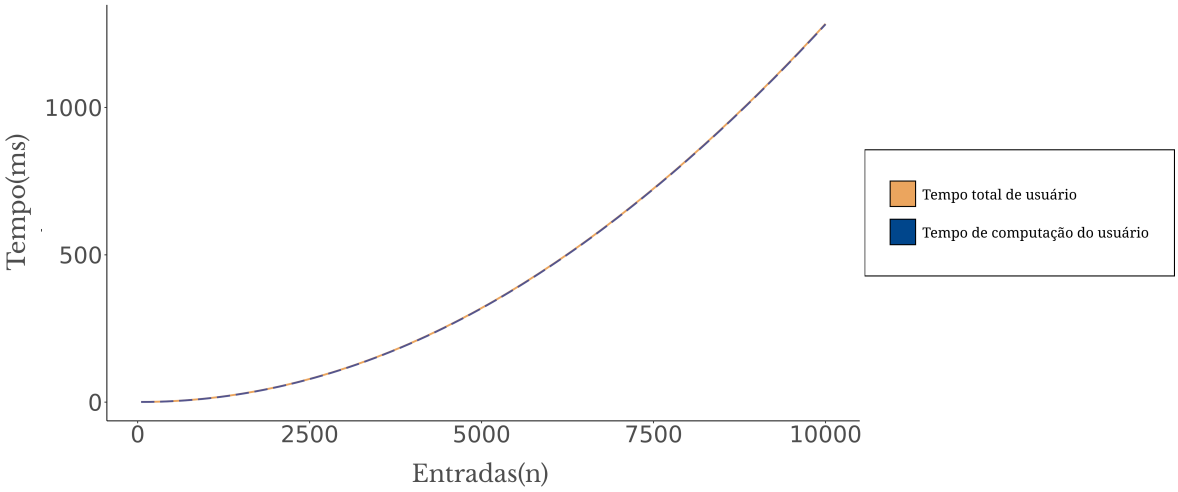


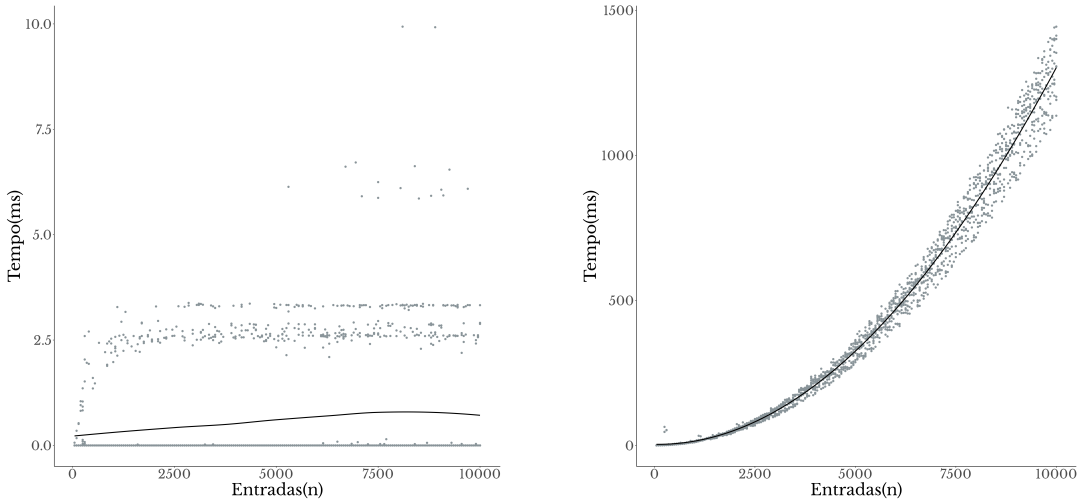
Figura 8: *Comparação entre tempo total de usuário e tempo de computação de usuário*

Visando ainda a medida de significância de diferentes componentes da execução do programa, é importante analisar os gráficos apresentados na Figura 9. Ao analisar as curvas de regressão, podemos inferir que o tempo gasto pelo programa em chamadas ao sistema foi insignificante em comparação com o tempo de processamento pelo usuário. Como fonte desse comportamento, pode ser apontada a natureza iterativa e dinâmica do programa, que reduz a presença do sistema operacional durante o funcionamento do algoritmo principal: uma vez alocadas, as estruturas principais são reutilizadas, e as funções não necessitam de gerenciamento recursivo.

Para mais, a Figura 10 apresenta diferentes curvas de aproximação de diversas informações coletadas pelo programa. Além disso, a função de complexidade também é apresentada, mas é multiplicada por uma constante pequena (0,000015), a fim de que, naturalmente representando o número de comparações, conseguisse adequar-se à avaliação do tempo.

O gráfico permite a melhor comparação entre a função de complexidade do programa e os tempo de execução na prática. Além disso, a comparação com o tempo total de relógio reforça que, como dito ateriormente, os tempos de entrada saída e chamadas de sistema têm pouca relevância quando comparados com o tempo total de usuário (que representa o processamento utilizado puramente para a ordenação dos pontos e cálculo dos hipercampos).

Por fim, é válido fazer uma última consideração sobre a relevância da ordenação dos pontos para a análise do algoritmo. Primeiramente, essa tarefa recebeu grande importância porque, sem ela,



(a) *Tempo total de sistema registrado* (b) *Tempo de relógio total*

Figura 9: *Comparação entre o tempo de sistema e o tempo de usuário*

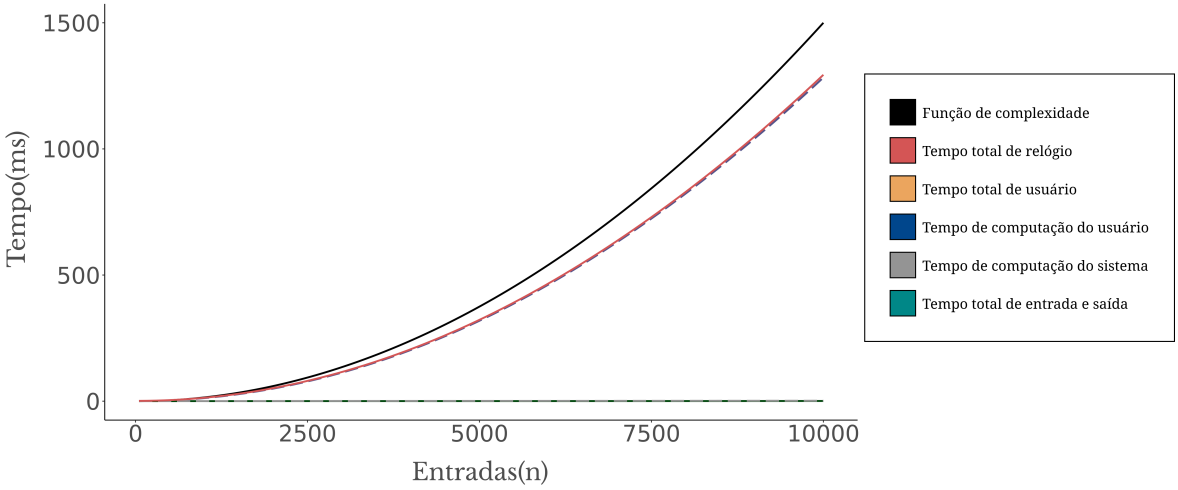


Figura 10: *Comparação entre diversos dados do programa*

a estratégia de resolução mostrada não funcionaria. Por outro lado, como mostrado na Seção 5, se a complexidade de ordenação não fosse considerada (ou, ainda, se ela fosse reduzida com o uso de algum método mais rápido, como o *quicksort*), o comportamento assintótico ainda seria $O(n^2)$ - a ordem de complexidade da função que calcula o encadeamento máximo de pontos.

7 Conclusão

Primeiramente, é válido considerar que o programa atendeu às expectativas de resolução do problema. Acima de tudo, o tempo polinomial de execução foi mantido, e a implementação foi simples.

Após a análise dos resultados, foi possível confirmar especulações feitas durante o desenvolvimento do algoritmo. O tempo de entrada e saída de dados foi pouco relevante para o desempenho final, e com a abordagem iterativa, o programa não foi refém de uma grande quantidade de chamadas de sistema. Em um método que envolvesse recursividade (como a estratégia de força bruta discutida no início da Seção 2), é provável que houvesse um *overhead* considerável durante a execução do programa. Como discutido na Seção 3.2, o bom desempenho nesse quesito era uma demanda para o desempenho da solução - e o algoritmo desenvolvido conseguiu atendê-la.

Por fim, é possível fazer considerações finais em relação ao próprio problema dos hipercampos como um todo. Sua utilidade prática, apesar de individualmente restrita, existe, e pode ser expandida, uma vez que a ideia fundamental de encadeamento pode ser usada, como dito inicialmente, para áreas como computação gráfica e gerenciamento físico de objetos. Além disso, computacionalmente, esse é um problema polinomial, mesmo que possa ser inicialmente confundido com alguma classe mais complexa de programas.

Durante a construção do programa, com o objetivo melhorá-lo, foram considerados diversos conceitos relacionados à programação, com o objetivo de aprimorá-lo. Como resultado do estudo de tópicos possíveis abordagens e técnicas de otimização, foi possível desenvolver um bom algoritmo, bem como realizar uma análise mais detalhada e precisa do problema como um todo.

References

[Cormen et al., 2009] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). *Introduction to Algorithms, third edition*. Computer science. MIT Press.

[Kernighan and Ritchie, 1988] Kernighan, B. and Ritchie, D. (1988). *The C Programming Language*. Prentice-Hall software series. Prentice Hall.

[Knuth, 1994] Knuth, D. E. (1994). *The Art of Computer Programming*. Addison Wesley, 2 edition.

[Stewart, 2016] Stewart, J. (2016). *Cálculo, volume 2*. Cengage Learning, 8 edition.