Trabalho Prático - Grafos

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Projeto e Análise de Algoritmos

João Francisco Barreto da Silva Martins
<joaofbsm@dcc.ufmg.br>

13 de Maio de 2019

1 Introdução

> Em um futuro distante, duas raças travam constantes batalhas por toda a galáxia, enviando frotas de naves para dominar sistemas solares inteiros. Como membro de uma dessas das frotas, sua missão atual é interceptar uma frota inimiga para um ataque surpresa...

Dado o contexto no qual nos encontramos, almejamos identificar corretamente cada uma das naves na frota inimiga a ser alvejada, cujos tipos podem ser vistos na Figura 1. Além disso, também queremos calcularmos o limite inferior para que alguma dessas naves esteja pronta para revidar nosso ataque surpresa, ou seja, o tempo até que todos os seus tripulantes se organizem, usando um sistema de teleporte interno, nos seus postos de combate corretos. Denominamos esse limite inferior como *tempo de vantagem*.

Todas as informações que temos vem de um radar que identifica a estrutura interna das naves somadas ao estado dos postos de combate obtidas ao hackearmos o computador da central de operação dos inimigos, e cabe a nós processá-las corretamente para reportarmos ao centro de inteligência da nossa própria frota.

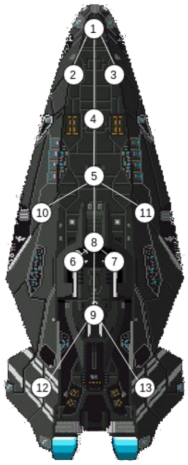
O restante desse relatório está organizado como descrito a seguir. A Seção 2 explica decisões de modelagem e implementação da solução, já descrevendo a complexidade dos métodos usados. A Seção 3 condensa as análises de complexidade previamente discutidas em complexidades de tempo e espaço que se referem a aplicação como um todo. A Seção 4 faz uma análise experimental dos tempos de execução e consumo de memória do programa para uma variedade de cenários e, por fim, a Seção 5 conclui esse documento apontando possíveis direções de melhora.

2 Modelagem e Implementação

A forma mais intuitiva de modelar esse problema é utilizando como estrutura de dados para a frota um grafo não direcionado, sem pesos nas arestas e com pesos nos vértices, onde cada nave é um componente conexo deste grafo.

A linguagem escolhida para implementação foi Python 3 na versão 3.5.2, que atualmente se encontra disponível nas máquinas Linux do Departamento de Ciência a da Computação. Apesar de ser uma linguagem consideravelmente mais lenta, ela foi escolhida pela intuitividade de sua sintaxe, o que facilitou em muito a construção da aplicação. O código fonte do trabalho se encontra hospedado no seguinte repositório: `https://github.com/joaofbsm/battleship-intel`.

(a) Reconhecimento



(b) Frigata

(c) Bombardeiro

(d) Transportador

Figura 1: Tipos de naves usadas pela frota inimiga.

2.1 Estruturas de Dados

As classes **Graph** e **Vertex**, presentes respectivamente nos módulos *graph.py* e *vertex.py* visam implementar uma estrutura de dados para grafos com listas de adjacência de forma genérica, com a classe para vértices sendo usada simplesmente para encapsular atributos, e.g., peso e grau. Já a classe **Ship** é uma abstração que encapsula os métodos para computar informação útil a respeito de cada nave da frota, os quais, por sua vez, usam métodos genéricos da primeira classe mencionada.

2.2 Formato da Entrada

A entrada descreve múltiplas naves de maneira conjunta. A primeira linha da entrada contém dois inteiros $N(10 \leq N \leq 10^5)$ e $M(8 \leq M \leq 10^6)$, representando respectivamente o número de postos de combate encontrados pelo radar e a quantidade total de teleportes possíveis entre postos de combate.

As próximas M linhas possuirão dois inteiros $a(1 \leq a \leq N)$ e $b(1 \leq b \leq N)$, representando os teleportes possíveis nas naves. Cada posto de combate a pode teleportar para b, fazendo com que b possa também teleportar para a. Desta forma, a informação sobre as arestas não estará redundante.

As próximas N linhas possuirão dois inteiros $c(1 \leq c \leq N)$ e $d(1 \leq d \leq N)$, identificando

que um tripulante no posto de combate c deve retornar a seu posto de combate correto d. Caso um tripulante esteja na posição correta, c e d serão iguais.

O programa lê da entrada padrão usando a função `create_graph_from_stdin()` presente no módulo *utils.py*, o qual contém funções utilitárias diversas.

2.3 Identificação dos Componentes Conexos

A identificação de componentes conexos num grafo não direcionado pode ser feita utilizando o algoritmo Depth-First Search (DFS). Comumente implementado de forma recursiva, nesse caso utilizamos uma versão iterativa do algoritmo para evitar que a pilha de chamadas de função do Python, que tem por padrão tamanho máximo igual a 1000. Mesmo se esse tamanho for aumentado indefinidamente utilizando o método `sys.setrecursionlimit()`, em vários dos casos extremos testados, isso leva a uma falha de segmentação (*segfault*) antes do programa terminar de executar. Um pseudoalgoritmo para o DFS iterativo pode ser encontrado no Algoritmo 1.

Algoritmo 1 Iterative DFS

```
1:  procedure DFS(G)
2:      for each vertex u ∈ G.V do
3:          u.visited = FALSE
4:      time = 0
5:      for each vertex src ∈ G.V do
6:          if not u.visited then
7:              S = ∅
8:              PUSH(S, src)
9:              while S ≠ ∅ do
10:                 time = time + 1
11:                 u = POP(S)
12:                 if not u.visited then
13:                     u.visited = TRUE
14:                     PUSH(S, u)    // Push u again so we can calculate its closing time after all the child nodes are executed
15:                     u.opening = time
16:                     for each vertex v ∈ G.Adj[u] do
17:                         if not u.visited then
18:                             PUSH(S, v)
19:                 else
20:                     u.closing = time
```

Para descobrirmos quais os vértices em cada componente basta criarmos um conjunto a cada iteração do algoritmo que passa pela condição na linha 6, e adicionar cada vértice ao conjunto quando visitado pela pela primeira vez. O método `Graph.find_connected_components()` segue esses princípios e retorna uma lista com os conjuntos de vértices para cada componente conectado.

A complexidade do algoritmo DFS iterativo é a mesma que a de sua implementação recursiva, ou seja, $O(V + E)$.

2.3.1 Otimizações

Para diminuir o tempo de execução da aplicação como um todo, reaproveitamos o caminhamento completo que fazemos no grafo na função já descrita para computar também a qual conjunto de um grafo bipartido cada vértice pertence, num esquema semelhante à coloração de grafos com duas cores: o vértice atual não pode estar no mesmo conjunto que seu pai está.

Os tempos de abertura e fechamento para cada vértice também são importantes para heurísticas ainda a serem definidas nesse documento. Por isso o algoritmo DFS foi preferido para esse passo ao invés do Breadth-First Search (BFS), que com um loop exterior também conseguiria identificar os componentes conectados.

2.4 Identificação dos Tipos das Naves

Agora que conhecemos todos os componentes conectados do grafo, ou seja suas naves, podemos prosseguir para a identificação dos tipos de cada uma. Pela descrição de cada uma das naves dada na especificação, todas são grafos bipartidos e com as seguintes peculiaridades:

- **Reconhecimento**: Árvore com vértices de grau máximo igual a 2: 2 vértices com grau 1 e $|V|$ - 2 vértices com grau 2.
- **Frigata**: Árvore. Não pode ter a mesma estrutura interna de uma nave de Reconhecimento, e portanto deve ter grau máximo maior do que 2.
- **Bombardeiro**: Grafo completamente bipartido, e que portanto contém ciclos.
- **Transportador**: Grafo cíclico onde todos os vértices tem grau igual a 2.

O método `Ship.identify_ship()` realiza uma checagem dessas condições e infere o tipo de cada nave a partir disso. Para executar a identificação são chamados dois métodos da classe **Graph**: `count_component_edges()` e `get_component_max_vertex_degree()`. Esses métodos tem complexidade $O(E)$ e $O(V)$ respectivamente, se considerarmos todas as execuções do método de identificação das naves da frota inteira, e, portanto, a complexidade final do método é $O(V + E)$.

2.4.1 Otimizações

Para otimizar a identificação das naves, aproveitamos que o grafo é não direcionado e que, portanto podemos identificar se há um ciclo no componente basicamente conferindo se $|E| > |V| - 1$. Poderíamos identificar os ciclos buscando por back edges num DFS, porém esse procedimento teria complexidade $O(V + E)$ ao invés de $O(E)$. Mesmo isso não mudando a complexidade total do método de identificação dos tipos de nave, conseguimos poupar tempo com essa alteração.

2.5 Cálculo do Tempo de Vantagem

Calcular o tempo de vantagem da frota inimiga significa calcular a menor quantidade de tempo para que tripulantes alcancem suas posições de combate corretas utilizando o sistema de transporte interno da nave. Realizar esse cálculo de forma exata é um problema muito difícil, e portanto devemos apenas calcular um limite inferior para esse tempo. Gostaríamos que esse limite fosse o mais justo possível, pois quanto mais tempo para atacarmos antes dos inimigos estarem preparados, melhor.

O problema de calcular o tempo de vantagem é equivalente ao problema de *token swapping* [1]. Ao fazermos n trocas para passarmos um combatente que está em um vértice para sua posição correta, acabaremos "embaralhando"todas as outras posições. No entanto, [1] mostra que ao realizarmos uma troca $u \to v$ para diminuir a distância entre o combatente e seu posto correto, estaremos diminuindo também a distância $v \to u$. Portanto, para termos um limite inferior razoável, basta computarmos todas as distâncias entre os combatentes e seus postos corretos, e dividirmos por 2.

Criamos heurísticas específicas para cada tipo de nave, que se aproveitam de especificidades de cada estrutura, a fim de agilizarmos o cálculo do tempo de vantagem. Além disso, para otimizar ainda mais o processo, paramos o cálculo de um tempo de vantagem assim que ele excede o menor

valor já computado para a frota até o momento. Por fim, antes de começarmos os cálculos, ordenamos crescentemente as naves por números de vértices com a função `sort_ships_by_size()` do módulo *utils.py*, pois assim possivelmente encontraremos números de vantagem pequenos logo de cara, reduzindo o tempo de execução total do programa.

2.5.1 Reconhecimento e Frigata

Como ambas as naves são árvores, podemos usar a mesma heurística para elas. Ao invés de executarmos simplesmente um algoritmo de busca de menor caminho entre dois vértices, que com o nosso contexto o BFS seria o mais eficiente, podemos usar o método de encontrar o Lowest Common Ancestor (LCA), com a otimização conhecida como Binary Lifting, que consegue encontrar o LCA em $O(lgV)$ com uma etapa de preprocessamento que custa $O(VlgV)$. Como o preprocessamento é executado apenas uma vez e o LCA é computador V vezes, a complexidade total do método é $O(VlgV)$.

A etapa de preprocessamento consiste em calcular a profundidade de cada vértice na árvore além de computar seu antecessor direto, seu antecessor dois vértices acima, seu antecessor 4 vértices pra cima, ..., até seu antecessor $2^{ceil(lgV)}$ acima. Ao agilizarmos esse cálculo utilizando programação dinâmica, conseguimos calcular todos os antecessores na árvore muito rapidamente, onde $v.ancestor[i] = ancestor[i-1].ancestor[i-1]$, iterando i de 1 até $lgV + 1$.

Em seguida, o método de LCA caminha pelas antecessores em comum, partindo da raiz, até que o antecessor dos vértices não seja mais comum, retornando o último deles. Por fim, para calcularmos a distância entre dois vértices u e v basta fazermos:

$$dist(u, v) = (u.depth - lca.depth) + (v.depth - lca.depth) \tag{1}$$

Essa heurística está implementada no método `Ship.compute_tree_advantage()`

2.5.2 Bombardeiro

Como o Bombardeiro é um grafo completamente bipartido, isso significa que a distância entre quais elementos de conjuntos diferentes é 1 e de quaisquer elementos do mesmo conjunto é 2. Sendo assim, basta utilizarmos os conjuntos já pré-computados, como explicado na Seção 2.3, e iterarmos por todos os vértices, computando em $O(1)$ as distâncias, para uma complexidade total de $O(V)$. Essa heurística está implementada no método `Ship.compute_bombardeiro_advantage()`.

2.5.3 Transportador

Para computar o tempo de vantagem para naves do tipo Transportador, devemos perceber que o fato de elas serem um ciclo faz com que existam dois caminhos de um vértice até qualquer outro: pelo lado esquerdo ou pelo lado direito. Para calcularmos então uma das distâncias subtrairmos os tempos de abertura encontrados no DFS da Seção 2.3, para, logo em seguida, subtrairmos o número de nós do valor da primeira distância, encontrando a distância entre nós tomando o outro caminho. Pegamos então a menor das duas distâncias computadas. As operações de cálculos de distância ocorrem em $O(1)$ e são executadas para todos os vértices $O(V)$, e portanto a complexidade total do método é $O(V)$. Essa heurística está implementada no método `Ship.compute_transportador_advantage()`.

3 Análise de Complexidade

3.1 Complexidade de Tempo

Usando as complexidades já descritas na Seção 2, calcularemos aqui a complexidade total do nosso código. As complexidades para cada parte do código são as seguintes:

- **Identificação dos Componentes Conexos**: $O(V + E)$
- **Identificação dos Tipos das Naves**: $O(V + E)$
- **Tempo de Vantagem Árvores**: $O(VlgV)$
- **Tempo de Vantagem Bombardeiro**: $O(V)$
- **Tempo de Vantagem Transportador**: $O(V)$

O pior dos cenários seria um grafo completamente conectado, ou seja, um único Bombardeiro. Nesse caso, $|E| = |V^2|$, e, portanto, o limite superior para a complexidade de tempo do nosso algoritmo é:

$$O(V + V^2) + O(V + V^2) + O(VlgV) + O(V) + O(V) = \mathbf{O(V^2)} \tag{2}$$

3.2 Complexidade de Espaço

No nosso código utilizamos três estruturas principais: uma lista de adjacência, e uma lista de vértices composta por objetos `Vertex()`. Todos os componentes de um objeto vértice possuem complexidade de espaço $\Theta(1)$, exceto a lista de antecessores, que contém $O(lgV)$ elementos. A complexidade de espaço para listas de adjacência é $O(V + E)$.

O pior dos cenários é novamente um grafo completamente conectado, composto por apenas um bombardeiro, e, portanto, temos:

$$O(V + V^2) + O(VlgV) = \mathbf{O(V^2)} \tag{3}$$

4 Análise Experimental

Todos os experimentos foram executados utilizando uma máquina com sistema operacional macOS Mojave Versão 10.14.4, processador Intel Core i5 de quinta geração 2.7 GHz e 8 GB de memória RAM DDR3.

Cada experimento foi executado 10 vezes e reportamos a média e desvio padrão para tempo de execução e memória consumida. Por terem ficado muito erráticos, as barras de erro com os desvios padrões foram removidas dos gráficos. Todos os gráficos possuem ambos os eixos na escala log_10 para facilitar a visualização dos comportamentos.

4.1 Execuções por Tipo de Nave

Nesta seção, fizemos experimentos com frotas compostas apenas por uma única nave de cada tipo, com quantidades de nós (vértices) em cada uma variando de acordo com potências de 10. Essa separação é importante para entender como cada uma das execuções se compara com as complexidades calculadas nas Seções 2 e 3. As Figuras 2 e 3 mostram a comparação de tempo e memória respectivamente, para cada uma das execuções.

Como esperado, as naves que tenderam a consumir mais tempo (Reconhecimento e Frigata) foram as duas que utilizam a heurística com maior complexidade ($O(VlgV)$). Já com relação ao consumo de memória, a nave Bombardeiro foi a mais cara, uma vez que representa um grafo totalmente conectado. Os dois tipos naves que precisam de calcular a matriz de antecessores para utilizar no LCA vem logo em seguida com o Transportador usando a menor quantidade de memória.

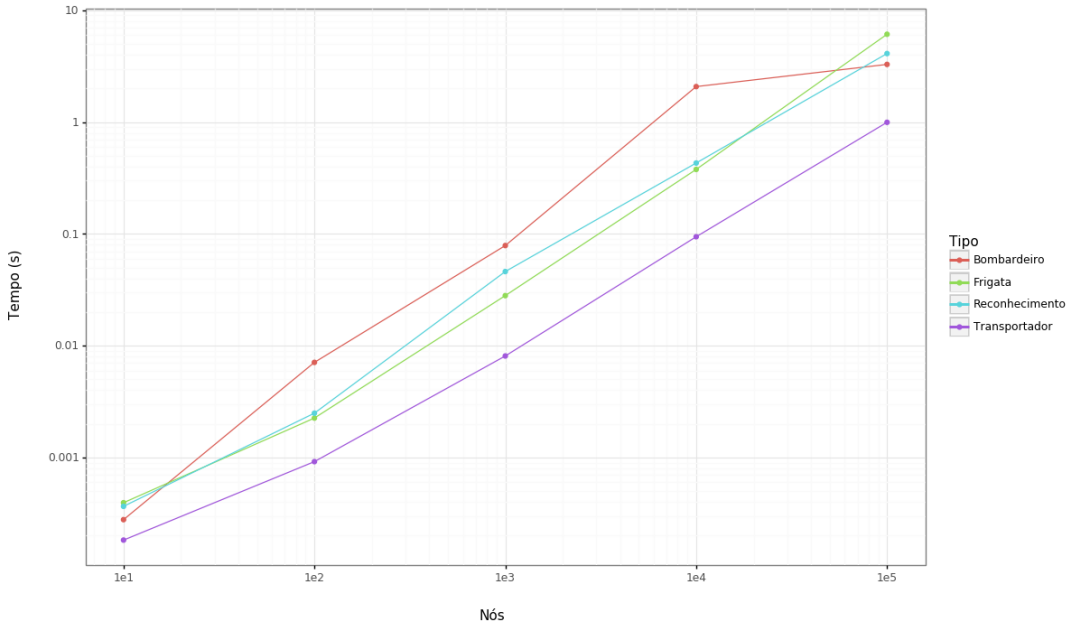


Figura 2: Tempo médio para as execuções com cada tipo de nave.

4.2 Execuções Variando Arestas e Vértices

Nesta seção, ao contrário da anterior, utilizamos frotas compostas por uma distribuição uniforme dos tipos de naves, com uma variação apenas no número de nós ou arestas totais. As Figuras 4 e 5 mostram a comparação de tempo e memória respectivamente, para cada uma das execuções.

As execuções com mais números de nós se mostraram mais caras tanto em tempo quanto em memória. Pelo fato de o acesso aos objetos que compõem cada nó não ser uma indexação trivial, e, além disso, os nós terem diversos atributos que acabam ocupando mais memória do que uma simples referência numa lista de adjacência (representação da aresta), fica evidente que esses resultados era esperados.

5 Conclusão

Nesse trabalho resolvemos o problema de identificação das naves de batalha que formam uma frota inimiga, além de também calcularmos o limite inferior para que alguma dessas naves esteja pronta para revidar nosso ataque surpresa, valor o qual denominamos *tempo de vantagem*. A frota foi modelada como um grafo não direcionado, sem pesos nas arestas e com pesos nos vértices, onde cada componente conexo representa uma das naves. A identificação de cada nave segue algumas análises simples da estrutura desses subgrafos, enquanto que o cálculo do tempo de vantagem é equivalente ao problema de *token swapping*, como descrito por [1].

A solução foi implementada em *Python 3* e possui complexidade de tempo $\mathbf{O(V^2)}$ e de espaço $\mathbf{O(V^2)}$. Diversos testes foram realizados com estruturas geradas para testar comportamentos específicos da aplicação, a qual se mostrou robusta e eficiente por conta das otimizações desenvolvidas, mesmo com os *overheads* inerentes à linguagem. Para um ganho ainda maior de eficiência, uma reimplementação em C++ seria muito interessante.

Referências

[1] K. Yamanaka, E. D. Demaine, T. Ito, J. Kawahara, M. Kiyomi, Y. Okamoto, T. Saitoh, A. Suzuki, K. Uchizawa, and T. Uno, "Swapping labeled tokens on graphs," *Theoretical Computer Science*, vol. 586, pp. 81–94, 2015.

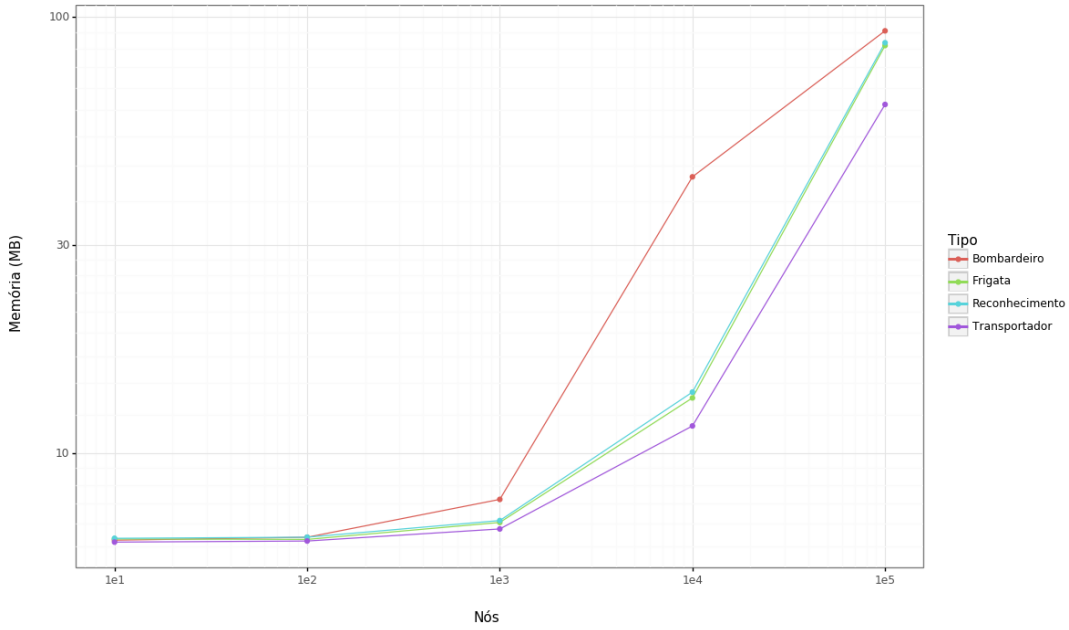


Figura 3: Consumo de memória médio para as execuções com cada tipo de nave.

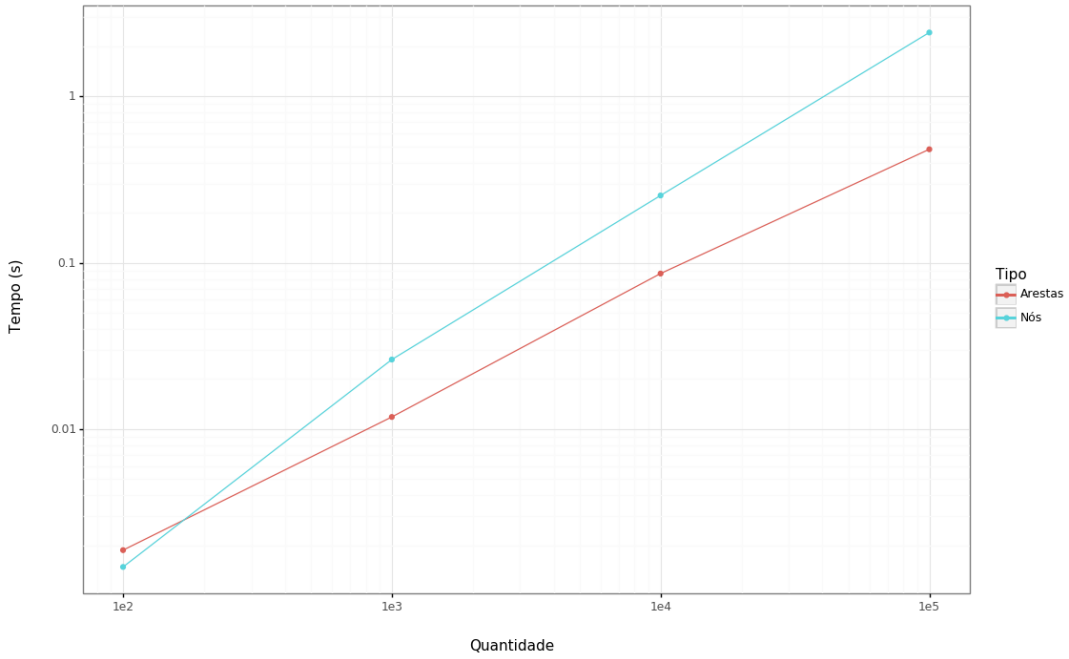


Figura 4: Tempo médio para as execuções de frotas com distribuição uniforme para cada tipo de nave.

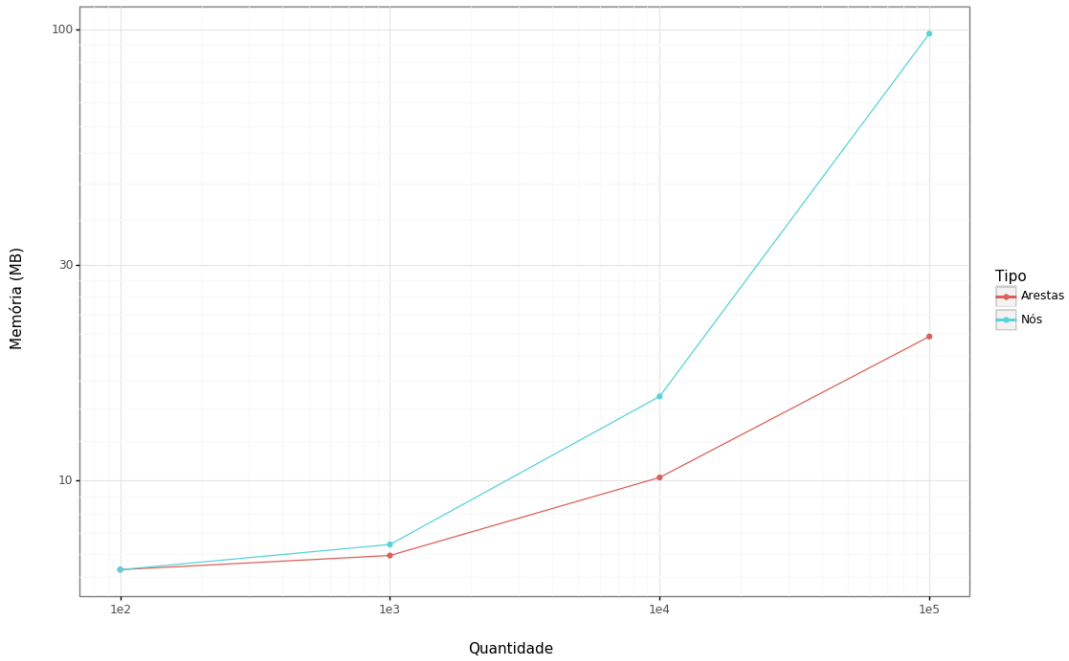


Figura 5: Consumo de memória médio para as execuções de frotas com distribuição uniforme para cada tipo de nave.