Algoritmos e Estruturas de Dados II – Trabalho 2

Alessandro Borges de Souza, Henrique Baptista de Oliveira

Escola Politécnica – PUCRS

11 de novembro de 2021

Introdução

Dentro do escopo da disciplina de Algoritmos e Estruturas de Dados II, o problema demonstrado no enunciado deste trabalho pode ser resumido assim: Dado uma sequência de tarefas ordenadas em um grafo direcional, realize todas as tarefas utilizando o menor número possível de agentes (no contexto do enunciado, minions) seguindo as seguintes regras:

1. Cada tarefa só pode ser realizada por um agente;

2. Uma tarefa é liberada para realização quando todas as outras tarefas que apontam para ela são concluídas (pré-requisitos), ou quando não é apontada por ninguém;

3. Diversos agentes podem realizar diversas tarefas ao mesmo tempo;

4. Quando uma tarefa está livre para ser realizada e há um agente ocioso, ele é instantaneamente enviado para a realização da mesma;

5. Quando há mais tarefas livres do que agentes disponíveis, será priorizado a realização das que venham primeiro na ordem alfabética;

6. Quando um agente inicia uma tarefa, ele só poderá alterar de tarefa assim que a mesma tiver sido concluída (não pode sair “no meio”).

Porém, as regras supracitadas servem para gerar um algoritmo que realiza apenas uma “sessão de treinamento”, ou seja, ele realizará o circuito com uma quantidade pré-determinada de agentes uma vez, obtendo o tempo de realização do circuito para X agentes. Para exemplificar, a imagem a seguir demonstra um grafo direcional como circuito sendo realizado e em seguida, o mesmo circuito, porém com menos agentes para demonstrar o impacto no tempo de execução.

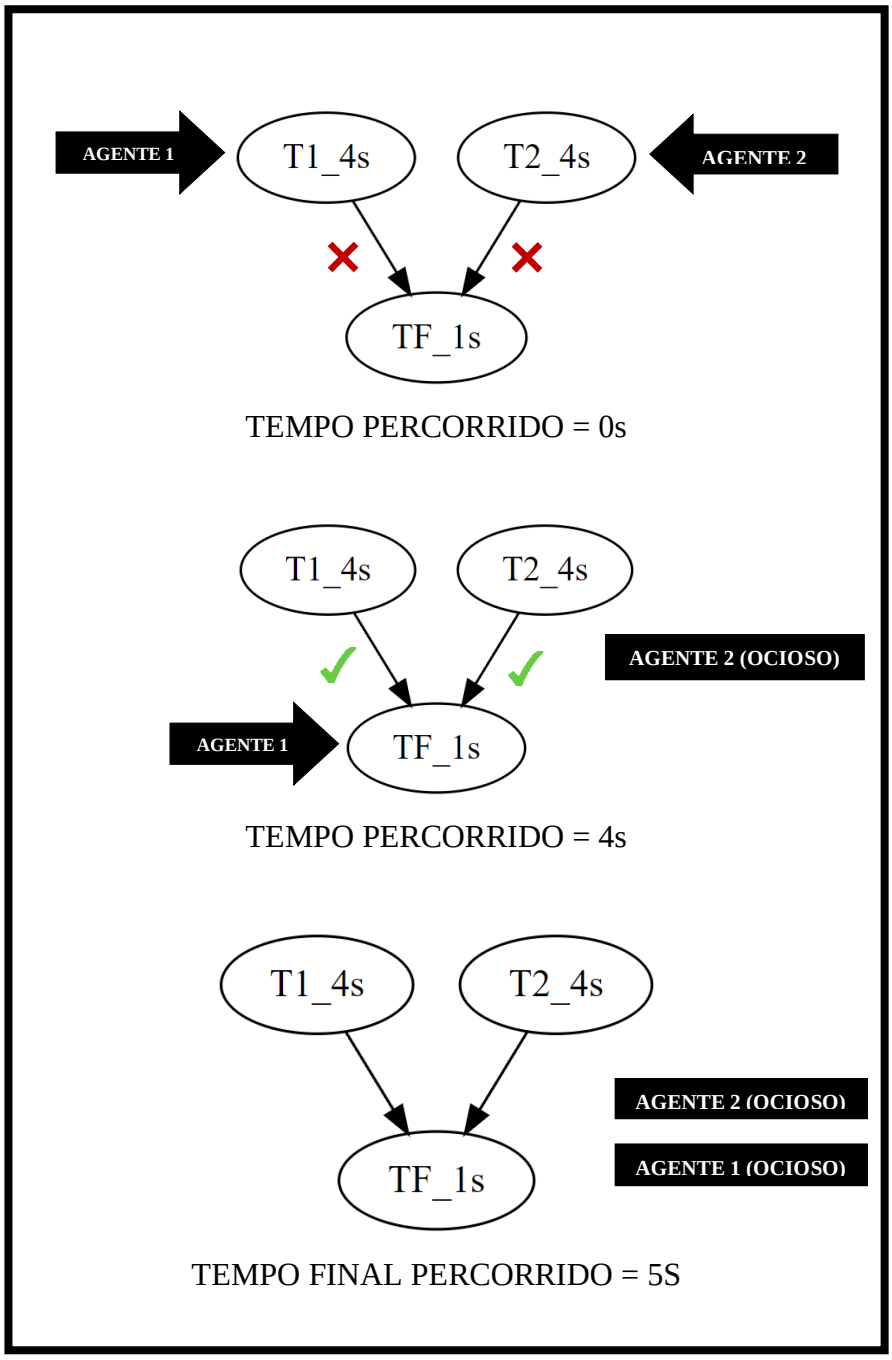


Figura 1: *Exemplo de circuito realizado por dois agentes*

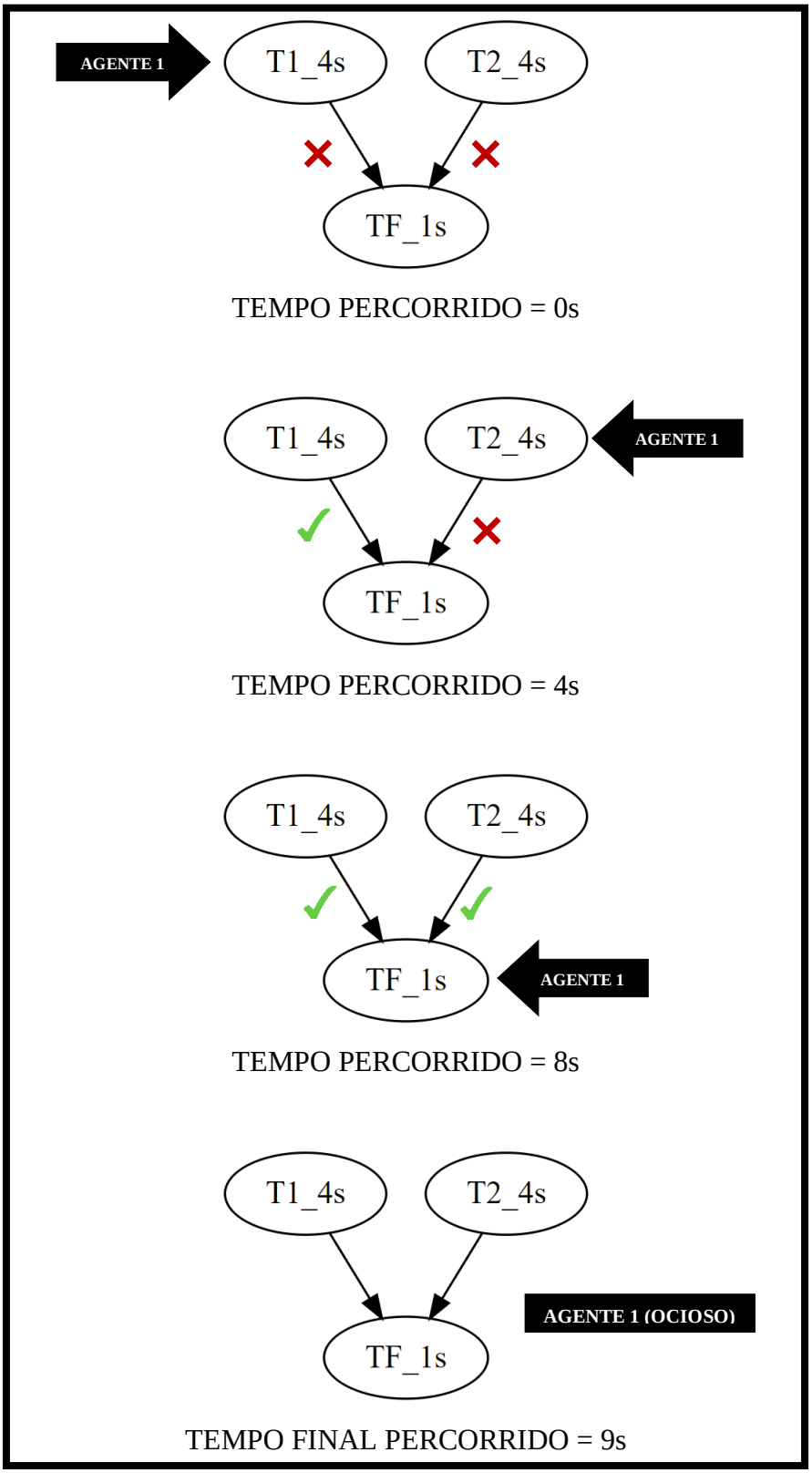


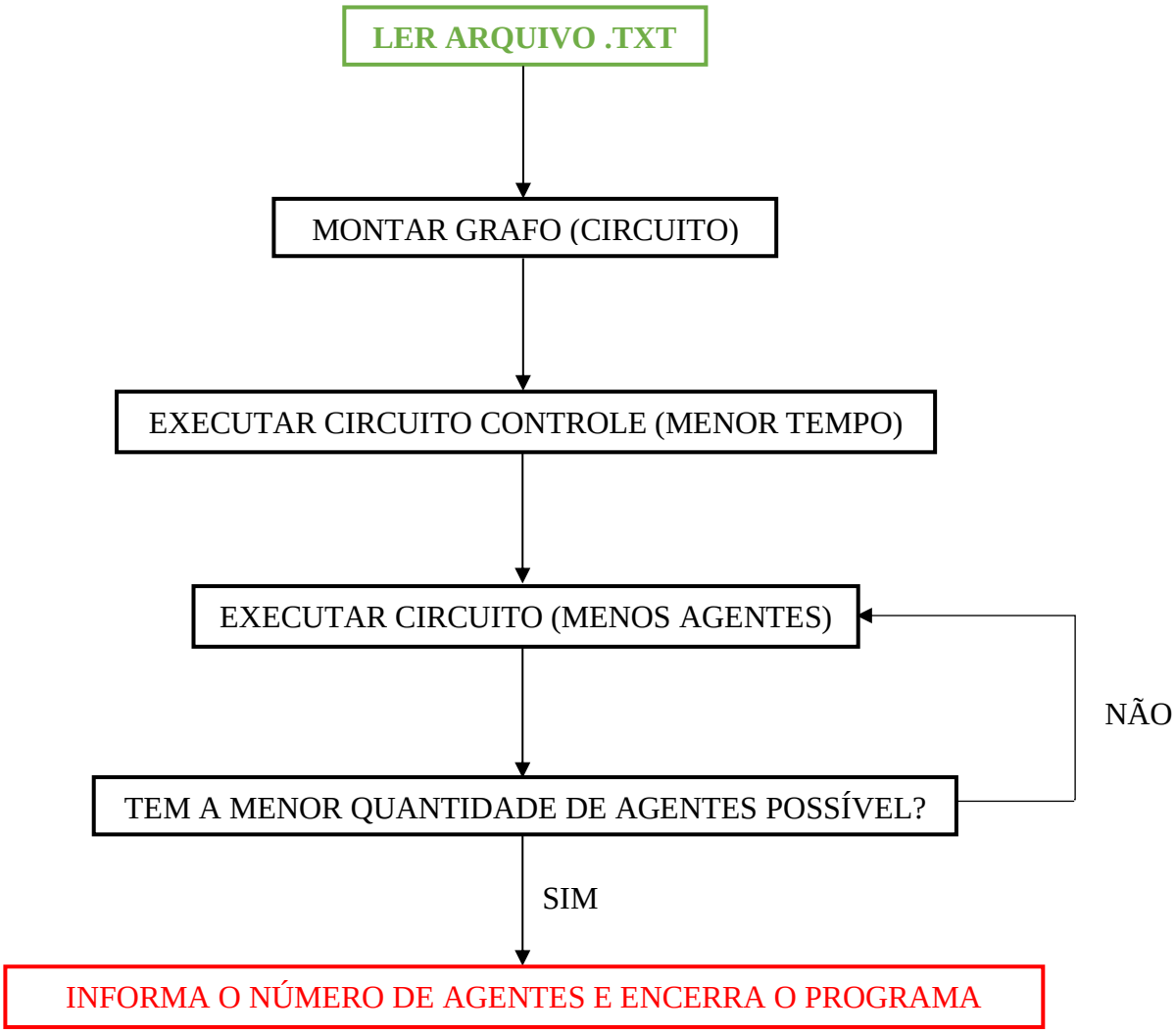
Figura 2: *Exemplo de circuito realizado por um agente*

Como visto acima, a quantidade de agentes influencia diretamente no tempo de execução do circuito, portanto o problema a ser resolvido é, qual a quantidade mínima de agentes (minions) para que o circuito seja completo no menor tempo possível? Para resolver tal questão, deve-se realizar um algoritmo que descubra essa quantidade mínima de agentes, para tanto deverá realizar o circuito diversas vezes com quantias de agentes

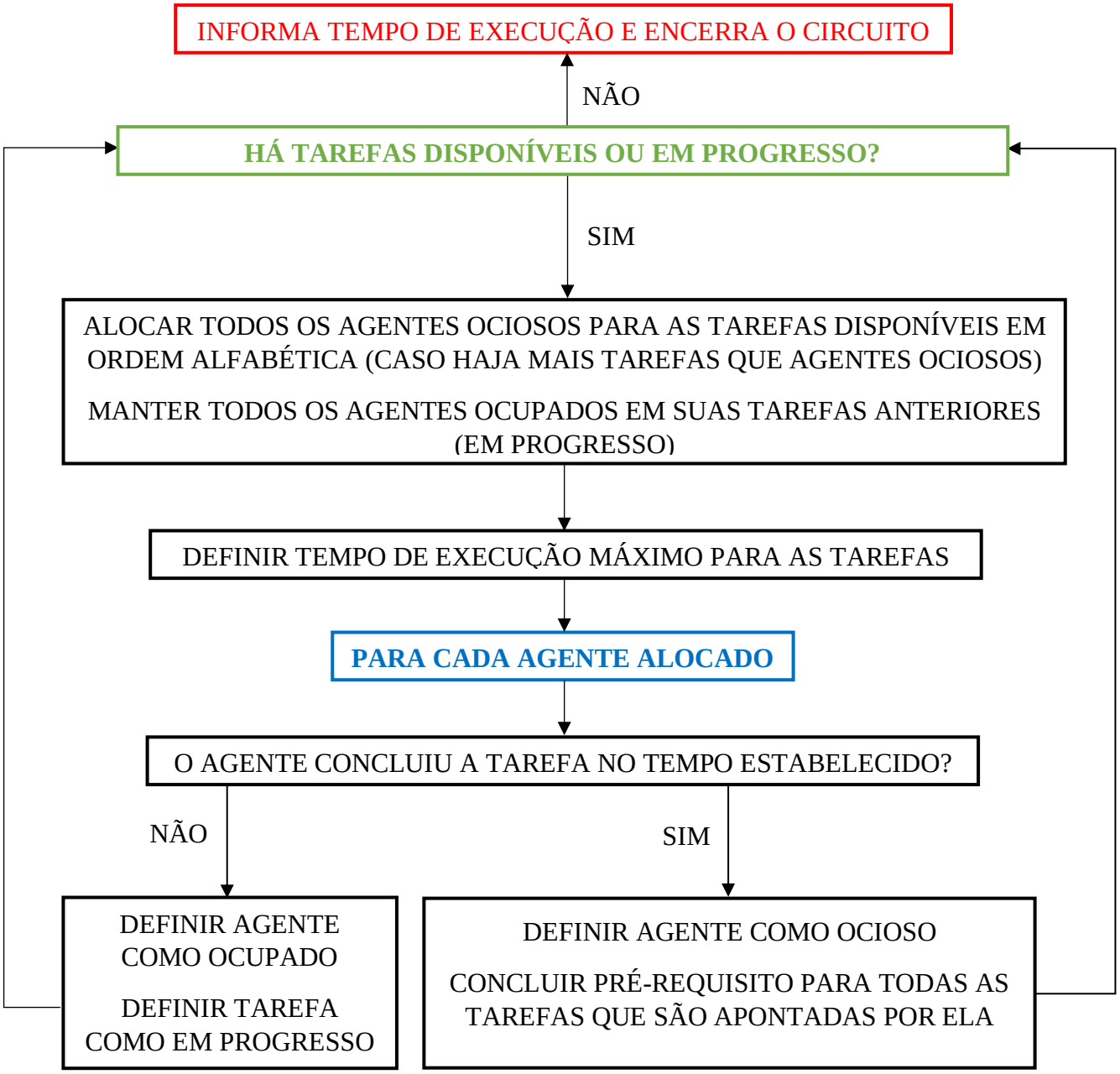
diferentes, sendo o problema a forma mais eficiente para chegar no melhor resultado (mínimo tempo e mínimo agentes) com a menor quantidade de realização de circuitos.

Além disso, analisando os exemplos de circuito acima é possível afirmar que se cada tarefa possuísse um agente que realizasse única e exclusivamente a sua tarefa, o tempo de execução seria mínimo. Portanto com apenas uma execução do circuito com o número de agentes igual ao número de tarefas, é possível descobrir o menor tempo de execução, assim servindo como circuito "controle". Restando agora descobrir a menor quantidade de agentes que gera o mesmo resultado de tempo do circuito "controle".

Dado as definições iniciais do problema, as regras de operação e alguns exemplos de funcionamento, é possível realizar uma modelagem inicial que sirva como guia para a implementação do algoritmo. Abaixo serão demonstrados dois guias em forma de fluxograma sendo um deles da obtenção do melhor resultado e o outro do processo de execução do circuito.



Fluxograma 1: *Lógica base para descobrimento da quantidade mínima de agentes (minions) para realização do circuito com tempo mínimo.*



Fluxograma 2: *Lógica base da realização do circuito de tarefas no grafo, seguindo as regras definidas.*

Portanto, durante a realização deste trabalho foram feitas três abordagens diferentes para a realização dos fluxogramas supracitados e do problema proposto, cada um com suas peculiaridades e alguns deles com suas limitações. Além disso é possível perceber que algumas questões ainda ficaram em aberto como por exemplo, como saber se o circuito tem o menor tempo dentre todos, como é definido o tempo máximo para execução das tarefas e como saber se uma tarefa está disponível ou não? Todas essas perguntas foram respondidas de formas diferentes nas implementações desse algoritmo e serão apresentadas e respondidas a seguir.

Primeira Solução

Como ponto de partida a primeira solução foi pensada para realizar as soluções propostas para os casos mais simples dos arquivos de texto, portanto sua eficiência para casos maiores e mais complexos não foram levados em conta e com isso essa solução se torna funcional, porém altamente limitada, devido ao tempo de processamento em casos maiores, sendo assim os passos para o seu desenvolvimento foram os seguintes.

Os algoritmos implementados nessa solução foram realizados na linguagem Java, que por padrão não tem suporte nativo a estrutura Grafo, sendo assim foi necessário antes de efetivamente iniciar o algoritmo de resolução do problema, programar uma estrutura própria que implementasse o grafo, sendo esse um detalhe importante para a eficiência do código que posteriormente demonstrou ser o grande "gargalo" na eficiência dessa solução.

O primeiro passo dessa solução foi inicializar um grafo vazio (sem vértices e arestas) e então começar a leitura dos dados provenientes do arquivo de texto, que possuem o seguinte padrão.

Figura 3: *Demonstrando significado dos dados provenientes dos arquivos de texto*

Iniciando uma leitura do arquivo, linha-a-linha o algoritmo foi responsável por verificar se ambos os vértices informados em uma linha já estavam alocados no grafo como vértices e caso negativo, seriam então adicionados. Posteriormente a essa verificação a adição da aresta direcional era realizada de forma que todo vértice sabe para quem está apontando, mas nenhum vértice sabe se está sendo apontado. Assim após realizar esse ciclo linha-a-linha foi possível montar o grafo com todos os vértices e arestas necessários. Para demonstrar essa lógica, um algoritmo parecido com o abaixo deveria ser realizado:

```
 1 enquanto (houverLinhas){
 2
 3    linha.separar(elementosLinha)
 4    //Separa os dados da linha nos dois vértices informados no arquivo
 5
 6    se (vertice1 não está no grafo) grafo.adicionaVertice(vertice1)
 7    se (vertice2 não está no grafo) grafo.adicionaVertice(vertice2)
 8
 9    grafo.adicionaAresta(vertice1,vertice2)
10    //Adiciona uma aresta que aponta do vertice1 para o vertice2
11 }
```

Após a leitura dos dados e do grafo devidamente montado o código é redirecionado para a secção responsável por realizar o circuito com diversas quantidades de agentes (minions) a fim de descobrir o tempo mais curto com a menor quantidade de agentes. Como solução inicial e pouco sofisticada, o algoritmo iniciava o primeiro treinamento com o número de agentes igual ao número de vértices, gerando tempo de controle e então realizava um loop de treinamentos cada vez com um agente a menos até que o tempo de execução parasse de diminuir e começasse a crescer, assim indicando que o valor ideal de agentes sem alteração no tempo foi alcançado. O algoritmo abaixo representa essa lógica, que apesar de funcional é pouco sofisticada e ineficiente:

```
quantidadeMinions = quantidadeVertices
//Caso de menor tempo garantido é quando cada tarefa tem um
//agente exclusivo

enquanto (tempoDeExecução igual tempoControle){

   treinarMinions(quantidadeMinions)
   //Realiza o circuito com a quantidade de agentes definida

   se (tempoDeExecução maior tempoControle) fim
   //Se o tempo de execução for maior que o tempo de controle
   //o valor ideal de agentes foi encontrado e a aplicação se encerra

   quantidadeMinions = quantidadeMinions - 1
   //Novo valor de agentes é ele mesmo menos um
}
```

Então com o loop de treinamentos encerrado é possível determinar qual é a quantidade mínima de agentes para realizar um circuito com o mesmo tempo que o controle, bastando agora apenas informar o resultado através de uma mensagem no terminal, por exemplo. Porém antes de encerrar a primeira solução é necessário explicar a principal e mais importante tarefa desse algoritmo, o treinamento dos agentes (minions).

O método encontrado para realizar um circuito inteiro foi através da recursão sendo que cada nível nessa recursão representa um “turno” de tempo de execução, ou seja, todas as tarefas que foram realizadas nesse turno, teriam seu tempo de execução reduzido baseado no tempo do “turno”.

Nessa solução a forma de verificar se uma tarefa estava apta a ser realizada foi através da verificação de todas as outras tarefas procurando por alguma que apontasse para ela, caso negativo, a tarefa então era alocada para a lista das tarefas pendentes. Como é possível perceber essa lógica, apesar de funcional, faz com que o grafo tenha que ser modificado a cada operação, mais especificamente “desmontado”, ou seja, à medida que a tarefa era realizada o grafo ia perdendo vértices e arestas até que não houvesse mais vértices para serem verificados, dessa forma causando um processo muito custoso para ser realizado, se tornando impraticável nos casos com grafos maiores, além disso a verificação em todos o vértices do grafo também gerou forte impacto negativo no tempo de processamento. De forma simplificada a ideia exposta segue o algoritmo a seguir:

```
1 para cada (tarefa na listaTodasTarefas){
2    se (tarefa não é apontadaPorOutraTarefa()){
3          listaTarefasDisponiveis.adiciona(tarefa)
4    }
5    //apontadaPorOutraTarefa() verifica se qualquer outro vértice
6    //no grafo aponta para tarefa
7 }
```

Após verificar quais tarefas estavam livres para realização elas foram organizadas em ordem alfabética em uma fila de aguardo para a realização pelos agentes, porém antes de efetivamente iniciar, essa fila era "furada" pelas tarefas em progresso, que obrigatoriamente deveriam continuar sendo realizadas até o fim e caso sobrasse agentes esses pegariam as tarefas restantes na fila até que, ou acabasse as tarefas e agentes ficassem ociosos, ou acabassem os agentes antes das tarefas, nesse caso resultando nas tarefas se manter na fila para a próximo "turno" de treinamento, porém isso não significa que possuíam alguma prioridade, sendo reorganizadas com as novas tarefas liberadas, novamente em ordem alfabética antes da escolha dos agentes no novo "turno". Sendo essa lógica em código algo como:

```
1 filaTarefasDisponiveis.ordenar()
2 //Ordena todas as tarefas na fila em ordem alfabética
3
4 para cada (tarefa na listaTarefasEmProgresso){
5    filaTarefasDisponiveis.furaFila(tarefa)
6    //Coloca a tarefa em primeiro lugar na fila
7 }
```

Na fase em que os agentes selecionavam suas tarefas todas eram inspecionadas a fim de encontrar a tarefa com o menor tempo de execução, sendo esse tempo declarado o tempo do "turno", assim permitindo que a regra de realização simultânea de tarefas fosse respeitada. Logo em seguida todas as tarefas se iniciavam e a partir daí poderiam ter dois resultados, como demonstrado no fluxograma, ou seja, poderiam zerar o seu tempo de execução, que é concluir a tarefa, ou o tempo de o "turno" não ser o suficiente para zerar o tempo da tarefa, assim tendo que se manter no estado de em progresso. Sendo a lógica de obter o tempo do turno a seguinte:

```
1 para cada (tarefa na listaTarefasIniciadas){
2    //tarefa.pegarTempo() pega o tempo para realizar a tarefa
3    //menorTempo é o tempo do turno
4    se (tarefa.pegarTempo() menor menorTempo){
5          menorTempo = tarefa.pegarTempo()
6    }
7 }
```

No primeiro caso todas as arestas relacionadas com o vértice da tarefa eram removidas e em sequência o vértice era removido, de forma a não bloquear mais as tarefas para as quais ela apontava. Já no segundo caso o valor do vértice era atualizado com o novo tempo, porém devido a limitações da implementação, era necessário remover o vértice e suas arestas e adicionar um novo com o valor atualizado e replicar todas as arestas do antigo no novo, dessa forma diminuindo substancialmente a performance do algoritmo em casos maiores, porém funcional. E após atualizar o tempo essa tarefa que não foi concluída era adicionada na lista das tarefas em progresso, fazendo com que no próximo "turno" sua realização fosse continuada de forma ininterrupta. Dessa forma ambos os casos são verificados seguindo esse conceito:

```
1 para cada (tarefa na listaTarefasIniciadas){
2
3    se ((tarefa.pegarTempo() – menorTempo) maior 0){
4          tarefa.atualizaNovoTempo()
5          listaTarefasEmProgresso.adiciona(tarefa)
6          //Atualiza a tarefa com o novo tempo e adiciona na lista
7          //das tarefas em progresso (para receber prioridade depois)
8    }
9
10   ou {
11         tarefa.remover()
12         //Remove a tarefa do grafo, junto com seu vértice e arestas
13   }
14 }
```

Por fim o tempo do turno era contabilizado na memória para que ao final do circuito fosse possível saber o tempo total de execução das tarefas, ou seja, a soma dos turnos. Assim a primeira solução já era capaz de entregar os resultados esperados, porém devido a sua implementação com algumas redundâncias lógicas e com um método de descoberta do melhor tempo rudimentar, fez com que não funcionasse bem com casos maiores que 100 vértices, dessa forma se tornando inadequado para a resolução desse problema.

Segunda Solução

A segunda solução teve como objetivo solucionar a questão da eficiência na quantidade de realizações do circuito para descobrir a quantidade mínima de agentes necessárias para obter o menor tempo. Enquanto na parte do treinamento poucas alterações de implementação foram feitas, na secção de busca de melhor desempenho não houve só uma alteração na implementação, mas também uma alteração na lógica de buscar para obtenção de resultados. As figuras a seguir demonstrarão as diferenças do método de busca antigo para o novo.

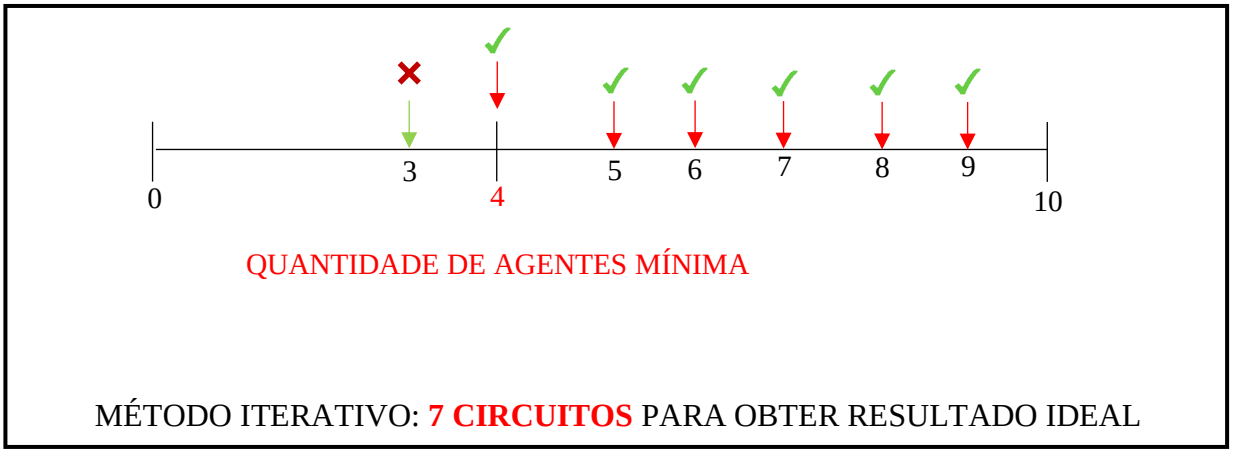


Figura 4: *Método iterativo de obtenção da melhor quantidade de agentes*

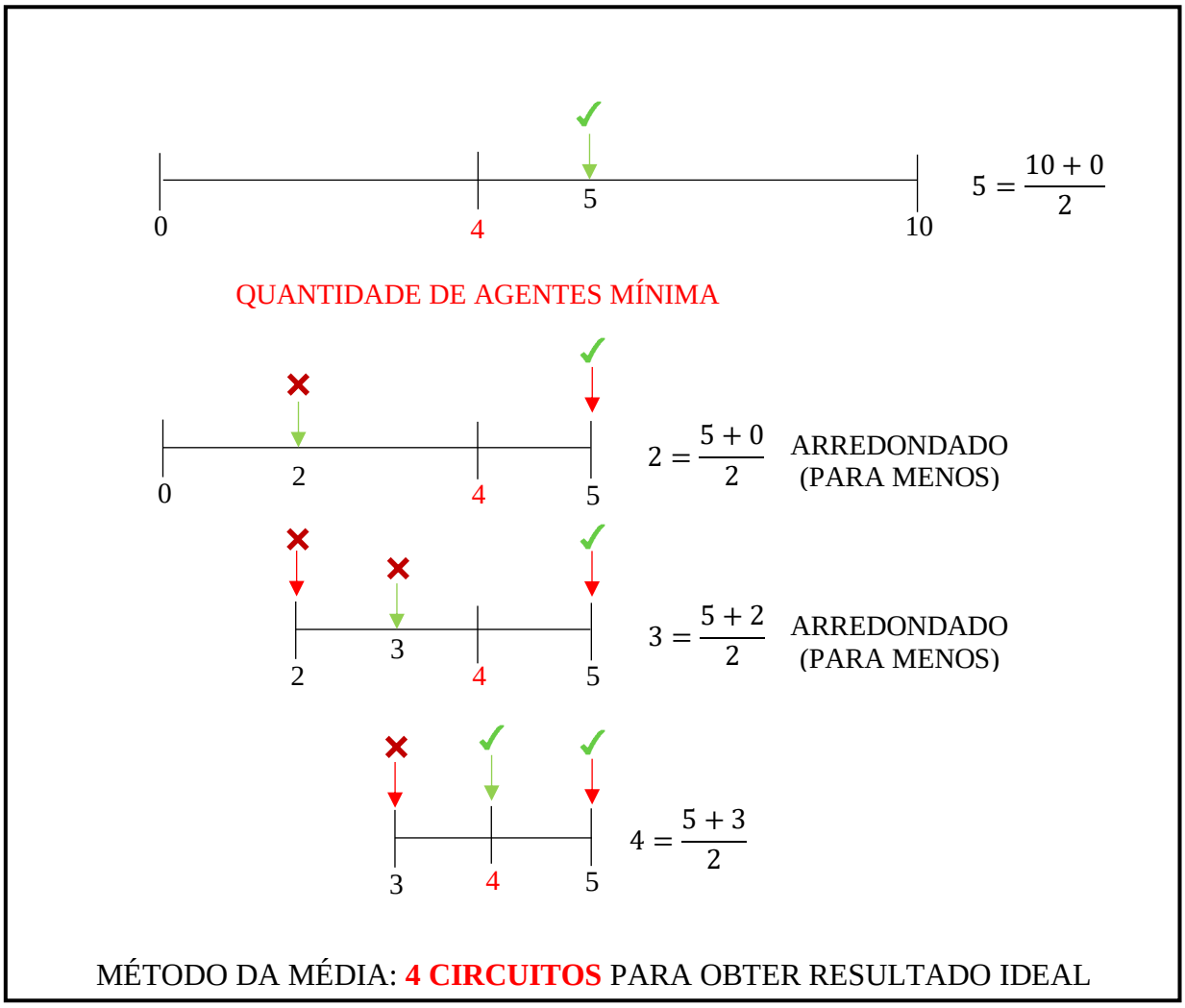


Figura 4: *Método da média para obtenção da melhor quantidade de agentes*

Como demonstrado acima o método da média realiza muito menos circuitos para chegar no mesmo resultado, além disso o crescimento da realização de circuitos no método iterativo é de O(n), ou seja, se esse exemplo fosse com 100 vértices, teria que ser feito 96 circuitos ao passo que o método da média que tem crescimento logarítmico, deveria realizar apenas 6 circuitos para chegar no resultado. A seguir será demonstrado em código a lógica do método da média:

```
enquanto (ehMenor for falso){

   treinarMinions(quantidadeMinions)

   se (tempoTreino maior tempoControle){
         quantidadeMinima = quantidadeMinions
   }

   ou se (tempoTreino igual tempoControle e
   quantidadeMinions menor quantidadeMaximma){

   quantidadeMaxima = quantidadeMinions
   }

   quantidadeMinions = (quantidadeMinima + quantidadeMaxima)/2
   //Em caso de decimais o valor é arredondado para baixo

   se (quantidadeMinions igual quantidadeMinima){
         ehMenor = verdadeiro
         sai
   }
}
```

Porém como na primeira solução, o algoritmo ainda possui problemas para a resolução de circuitos com grafos maiores e apesar de ficar mais rápido após a redução na quantidade de circuitos feitos, o tempo de resolução de cada circuito ainda é alto e ineficiente e apesar de alguns ajustes e refinamentos, ainda só serve como uma solução intermediária, que apesar de não entregar os resultados esperados de forma integral, consegue com parte do seu código já entregar um resultado satisfatório, como na secção de realização de diversos treinos.

Terceira Solução

Na terceira e última solução diversas mudanças estruturais foram realizadas de modo a aumentar a performance do algoritmo. Em linhas gerais a lógica do código se manteve igual, porém com uma drástica redução nas redundâncias e falhas lógicas no código que faziam com que fosse necessário mais esforço para se chegar ao mesmo lugar. A principal mudança que mudou toda a estrutura foi a reestruturação da API que implementa o grafo, deixando de ser uma estrutura feita de forma local e mais rudimentar, para uma API externa que foi utilizada durante as aulas dessa disciplina e que possui um nível de eficiência excepcional. Além disso para acompanhar essa nova implementação do grafo, que por questões de eficiência não permite a remoção de um vértice, uma nova abordagem teve de ser feita, visto que não era mais possível o “desmonte” do grafo.

Para acompanhar essa nova estrutura o primeiro passo foi adaptar o processo de criação do grafo, que ao invés de armazenar os dados da tarefa, armazena apenas um número que representa o seu vértice, sendo então necessária a criação de estruturas de mapa (chave-valor) para que a partir do número do vértice fosse possível localizar e armazenar uma tarefa correspondente. Assim aproveitando a reestruturação, as tarefas viraram objetos que armazenavam três valores, o seu nome, que é único e servia como

identificador da tarefa, o tempo de execução da tarefa (quando em 0 estava completa) e a quantidade de outras tarefas que apontam para ela (indegree), dessa forma uma tarefa sabe quando está livre para ser processada sem ter que “desmontar” o grafo. As figuras abaixo demonstram a diferença dos grafos com a informação direta ou com a informação relacionada a um número:

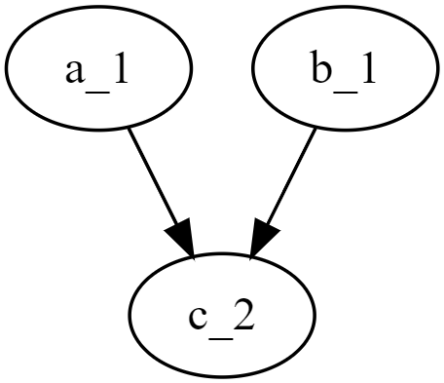


Figura 5: *Método de grafo com informações diretas nele*

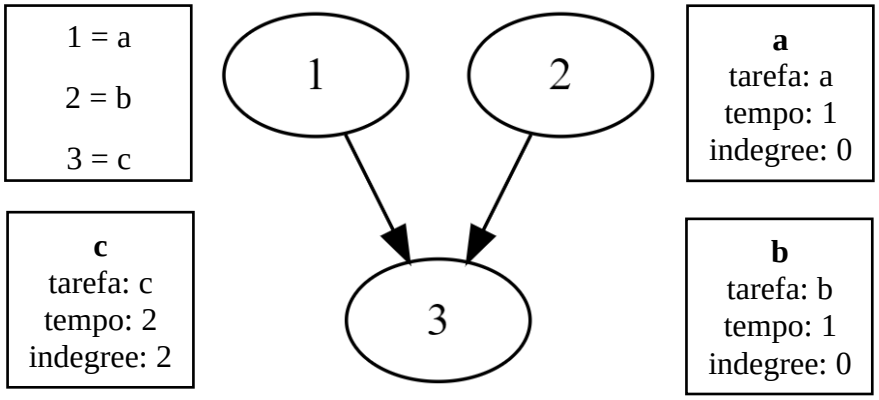


Figura 6: *Método de grafo com informações que referenciam o valor de um vértice*

Dessa forma a estrutura original do grafo foi preservada reduzindo drasticamente o custo operacional para a realização de um circuito, visto que após gerado o grafo é apenas consultado e não mais modificado. As operações de tarefa concluída que anteriormente deveriam ser desmontadas do grafo agora simplesmente reduzem o indegree de uma tarefa e quando essa chega em 0, significa que está disponível para realização, assim não sendo mais necessário percorrer todas as tarefas em busca de alguma que aponte para ela.

Portanto seguindo as lógicas base da primeira solução, com o método de procura da menor quantidade de agentes da segunda solução juntamente com as diversas melhoras e alterações estruturais foi possível alcançar um algoritmo com os resultados esperados em tempo satisfatório.

Resultados

Apesar das três soluções entregarem os mesmos resultados, nessa apresentação serão expostos apenas os da terceira solução devido ao melhor tempo de resolução para os casos de grafos maiores, sendo todo o cálculo de eficiência de algoritmo sendo baseado nesses resultados para análise e conclusão finais. Seguem os resultados abaixo:

- ***Caso oito_enunciado:***
  Quantidade de Minions Ideal: 3
  Tempo de Treinamento: 365
  Tempo de execução em segundos 0.08s

- ***Caso dez:***
  Quantidade de Minions Ideal: 5
  Tempo de Treinamento: 803
  Tempo de execução em segundos 0.08s

- ***Caso trinta:***
  Quantidade de Minions Ideal: 9
  Tempo de Treinamento: 1437
  Tempo de execução em segundos 0.08s

- ***Caso cinquenta:***
  Quantidade de Minions Ideal: 15
  Tempo de Treinamento: 1824
  Tempo de execução em segundos 0.09s

- ***Caso cem:***
  Quantidade de Minions Ideal: 22
  Tempo de Treinamento: 2245
  Tempo de execução em segundos 0.17s

- ***Caso quinhentos:***
  Quantidade de Minions Ideal: 55
  Tempo de Treinamento: 5250
  Tempo de execução em segundos 0.44s

- ***Caso mil:***
  Quantidade de Minions Ideal: 99
  Tempo de Treinamento: 6938
  Tempo de execução em segundos 0.84s

- ***Caso mil_e_quinhentos:***
  Quantidade de Minions Ideal: 76
  Tempo de Treinamento: 9448
  Tempo de execução em segundos 1.13s

- ***Caso dois_mil:***
  Quantidade de Minions Ideal: 84
  Tempo de Treinamento: 13064
  Tempo de execução em segundos 1.43s

- ***Caso tres_mil:***
  Quantidade de Minions Ideal: 64
  Tempo de Treinamento: 20135
  Tempo de execução em segundos 2.15s

- ***Caso quatro_mil:***
  Quantidade de Minions Ideal: 59
  Tempo de Treinamento: 24494
  Tempo de execução em segundos 2.80s

- ***Caso cinco_mil:***
  Quantidade de Minions Ideal: 96
  Tempo de Treinamento: 32430
  Tempo de execução em segundos 3.90s

Conclusão

Com os resultados do algoritmo final (solução 3) em mãos, é possível utilizarmos seus dados para realizar alguns gráficos e dessa forma conseguir tirar algumas conclusões através de suas análises. A seguir serão demonstrados diversos gráficos com dados provenientes dos resultados e de informações obtidas na depuração do algoritmo implementado (solução 3):

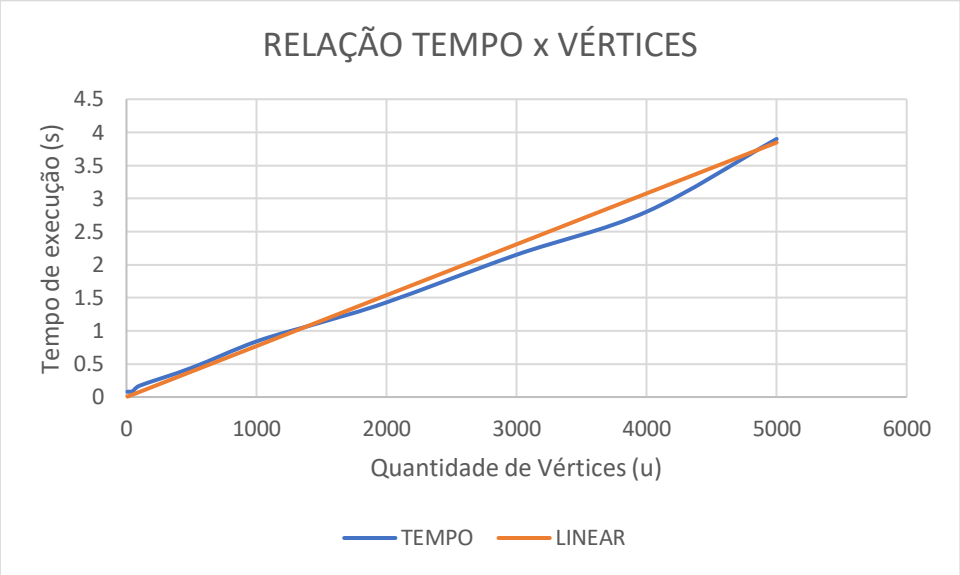


Gráfico 1: *Gráfico comparando tempo de execução com quantidade de vértices, gerando um crescimento linear O(n).*

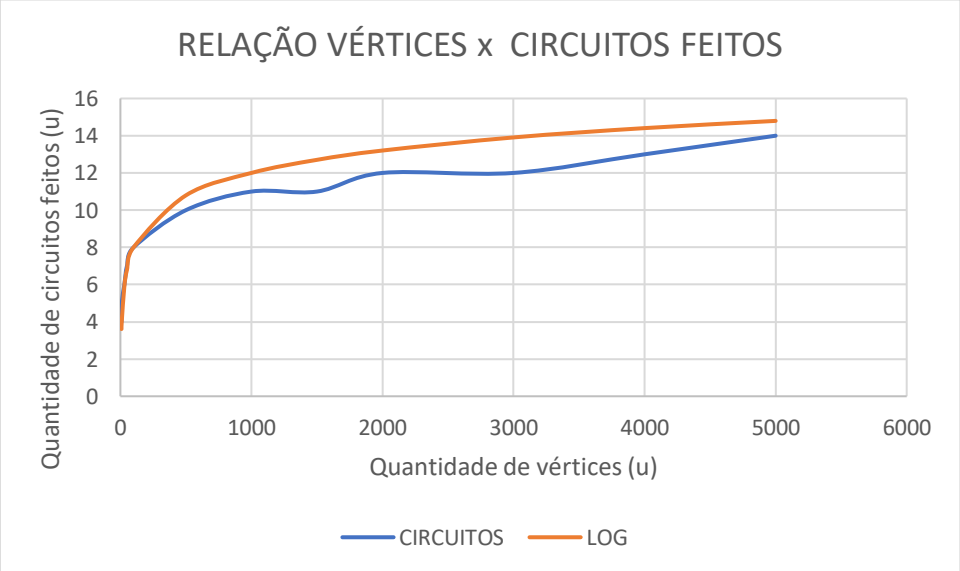


Gráfico 2: *Gráfico comparando quantidade de vértices com a quantidade de vezes que um circuito foi feito, gerando um crescimento logarítmico O(log(n)).*

Com os gráficos demonstrados acima é possível concluir que o algoritmo possui um crescimento O(n), ou seja, linear, dessa forma à medida que se aumenta o tamanho do grafo, o tempo para sua resolução e descobrimento do valor de agentes (minions) ideal também aumenta. É possível perceber também no segundo gráfico que graças ao método de descobrimento pela média é possível reduzir o crescimento na quantidade de circuitos realizados para uma taxa de O(log(n)), dessa forma auxiliando de forma importante para com que o crescimento desse algoritmo não fosse exponencial, visto que a quantidade de circuitos não cresce de forma expressiva, apenas o tamanho individual de cada circuito.

Dessa forma encontrou-se um resultado de algoritmo eficiente e balanceado, sendo possível prever com facilidade o uso computacional necessário para casos maiores. Para exemplificar esse ganho de performance, segue abaixo um gráfico vértice por tempo, comparando o tempo de execução da solução 3 utilizando o método da média e o mesmo algoritmo da solução 3, alterando apenas o modo para o método iterativo:

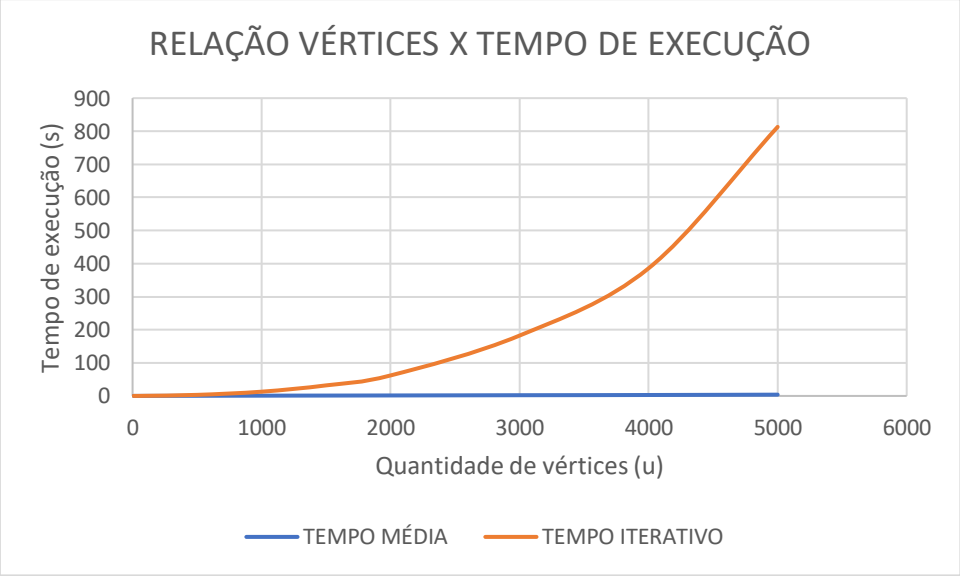


Gráfico 3: *Gráfico comparando quantidade de vértices com o tempo de execução do método da média (logarítmico) e do método iterativo (exponencial).*

Sobre as soluções iniciais (1 e 2), apesar de atenderem os objetivos esperados para o problema, o seu tempo de execução chegou a níveis alarmantes, como por exemplo o caso de o sistema demorar mais de trinta minutos para realizar o caso de 5000 vértices. Contudo essas soluções tiveram papel fundamental no processo de refinamento do algoritmo para que se chegasse a uma solução consistente e viável, que ainda possuía traços das versões iniciais.

Vale ressaltar que devido a natureza recursiva do algoritmo, na sua implementação em Java, foi necessário aumentar o limite padrão da stack da JVM (Máquina virtual Java), de 1MB (default) para 2MB, através do seguinte argumento a ser adicionado nas configurações da JVM:

-Xss2m

Assim conclui-se que a solução proposta é viável e conseguiu resolver todos os problemas requisitados dentro de um tempo aceitável e com eficiência geral O(n), tendo apenas como contraponto a configuração extra para sua execução (caso seja utilizado Java).

Referências

[1] B. OLIVEIRA, João; COHEN, Marcelo. Escrevendo seu relatório. Moodle PUCRS, 2017. Disponível em: https://moodle.pucrs.br/pluginfile.php/2329756/mod_resource/content/2/artdemo2.pdf. Acesso em dia: 12 de Novembro de 2021.

[2] Oracle. Digraph. Disponível em: https://docs.oracle.com/cd/E92951_01/wls/WLAPI/weblogic/management/provider/internal/DiGraph.html
Acesso em dia: 12 de Novembro de 2021.

[3] HowToDoInJava. Java deep copy using in-memory serialization. Disponível em: https://howtodoinjava.com/java/serialization/how-to-do-deep-cloning-using-in-memory-serialization-in-java/
Acesso em dia: 12 de Novembro de 2021.

[4] SEDGEWICK, Robert. WAYNE, Kevin. Directed Graphs. Disponível em: https://algs4.cs.princeton.edu/42digraph/
Acesso em dia: 12 de Novembro de 2021.

[5] Apache Software Foundation. BidiMap. Disponível em: http://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/BidiMap.html
Acesso em dia: 12 de Novembro de 2021

[6] GeeksForGeeks. Comparator Interface in Java with Examples. Disponível em: https://www.geeksforgeeks.org/comparator-interface-java/
Acesso em dia: 12 de Novembro de 2021.

[7] BMC. Increasing the JVM memory allocation and thread stack size. Disponível em: https://docs.bmc.com/docs/ars1805/increasing-the-jvm-memory-allocation-and-thread-stack-size-804712666.html
Acesso em dia: 12 de Novembro de 2021.