UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL PINHEIRO RODRIGUES
FILIPE PIERRI DE MENDONÇA NOBRE
LUIZ HENRIQUE GÖPFERT PALHANO LEAL

ALGORITMOS E GAMIFICAÇÃO DO PREENCHIMENTO DE GRAFOS DIRECIONADOS

RIO DE JANEIRO
2024

DANIEL PINHEIRO RODRIGUES
FILIPE PIERRI DE MENDONÇA NOBRE
LUIZ HENRIQUE GÖPFERT PALHANO LEAL

ALGORITMOS E GAMIFICAÇÃO DO PREENCHIMENTO DE GRAFOS DIRECIONADOS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Vinícius Gusmão Pereira de Sá

RIO DE JANEIRO
2024

CIP - Catalogação na Publicação

R484t

Ribeiro, Tatiana de Sousa
Titulo / Tatiana de Sousa Ribeiro. -- Rio de Janeiro, 2018.
44 f.

Orientador: Maria da Silva.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2018.

1. Assunto 1. 2. Assunto 2. I. Silva, Maria da, orient. II. Titulo.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

DANIEL PINHEIRO RODRIGUES
FILIPE PIERRI DE MENDONÇA NOBRE
LUIZ HENRIQUE GÖPFERT PALHANO LEAL

ALGORITMOS E GAMIFICAÇÃO DO PREENCHIMENTO DE GRAFOS DIRECIONADOS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em ___ de _____ de _____

BANCA EXAMINADORA:

Nome do Professor Orientador
Titulação (Instituição)

Nome do Professor1
Titulação (Instituição)

Nome do Professor2
Titulação (Instituição)

AGRADECIMENTOS

A nossas famílias: Jorge Antonio da Silva Rodrigues e Ediran Patricia Vargas Pinheiro, Daniel Augusto da Silva Nobre e Luciane Pierri de Mendonça Nobre, e Luiz Claudio Palhano Leal, Andrea Göpfert Garcia, e Maria Carolina Göpfert Palhano Leal, pelo apoio incondicional e sempre presente, tanto emocional quanto acadêmico. Sem seus incentivos e suporte, não teríamos chegado até aqui.

Ao incansável professor Vinicius Gusmão Pereira de Sá, nosso orientador durante a elaboração do trabalho e professor de muitas matérias antes disso. Não poderíamos ter escolhido outro orientador.

Finalmente, a nossos amigos do curso e outros professores do IC, que tornaram possível a graduação e estiveram presentes em momentos de dificuldade. Em particular, a Luís Fernando Garcia Jales e Thales Monteiro Pierini Macena, nossos colegas que desde o início foram insubstituíveis.

RESUMO

O trabalho aborda grafos direcionados completos e as diferentes heurísticas que podem ser utilizadas para gerá-los. Em particular, os grafos de interesse também seguem uma regra extra, de que não possuem ciclos (ou seja, são árvores). A partir de um conjunto de pontos sem arestas arbitrário, é possível, ao longo de vários passos, gerar um grafo que atenda às restrições impostas, e métodos diferentes serão mais ou menos eficientes nesse preenchimento, em termos de tempo computacional gasto. O trabalho apresenta opções de metodologias que podem ser utilizadas para esse propósito e verifica analiticamente qual delas se prova mais eficaz na redução da complexidade. Além disso, uma das heurísticas foi transformada em um aplicativo, em que os nós do grafo são associados objetos ou ideias e apresentados em pares a um usuário. O usuário, então, determina sua preferência por um nó ou pelo outro, criando, assim, uma aresta direcionada entre os dois. Dessa forma, é possível demonstrar empiricamente o funcionamento da heurística, e como a *gamificação* funciona como uma ferramenta para explicitar conceitos, realizar simulações, e auxiliar na visualização de soluções.

Palavras-chave: grafos; direcionamento; gamificação; complexidade.

ABSTRACT

The work discusses complete directed graphs and the different heuristics that may be used to generate them. In particular, the graphs of interest also follow an additional rule, that of not containing cycles (that is, being trees). From an arbitrary set of vertices without edges, it is possible, over several steps, to generate a graph which satisfies the imposed restrictions, and different methods of doing so will be more or less efficient, in terms of computational time spent. The work presents options for methods that can be used for this purpose and verifies analytically which of them proves most effective at reducing complexity. In addition, one of the heuristics was adapted into an app, in which the graph's nodes are associated with objects or ideas and presented in pairs to a user. The user, then, determines its preference for either node, and therefore creates a directed edge between the two. This way, it's possible to demonstrate empirically the heuristic's functioning, and how gamification works as a tool to show concepts, perform simulations, and aid in the visualization of solutions.

Keywords: graphs; direction; gamification; complexity.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A matemática pode ser dividida em várias áreas de conhecimento que focalizam em diferentes formas de abordar o estudo dos números. Uma dessas áreas, a teoria de grafos, se interessa por estruturas que modelam relações entre objetos, chamadas grafos. Grafos são compostos por vértices, que representam os objetos que se deseja analisar, e arestas que interligam os vértices, que, por sua vez, representam a presença ou ausência de um elo lógico que une um dado objeto a outro.

O estudo de grafos é muito útil para providenciar uma representação gráfica de conexões entre conceitos ou coisas físicas, mesmo quando essas conexões são, muitas vezes, abstratas. A teoria de grafos é usada de forma frequente e evidente em áreas da computação que são diretamente baseadas nos relacionamentos entre os componentes de um sistema, como por exemplo na Internet das Coisas ou no estudo de redes. Entretanto, ela também é encontrada em formas mais sutis, como na representação de um problema lógico ou de sistema de informações.

Mesmo fora do escopo da computação em geral, grafos podem ser utilizados para representar relacionamentos de diversas coisas. É possível, por exemplo, mostrar um mapa indicando as cidades locais e as rodovias que as interconectam. Também podemos listar várias características, e associá-las a diferentes objetos que as apresentam. Esse trabalho tem como foco um desses usos de grafos: representar os gostos de uma dada pessoa com relação a um assunto.

Dado um assunto qualquer, cada pessoa terá opiniões sobre o que são bons e maus exemplos, em sua escala pessoal de gostos, dentro desse assunto. Se o tópico fosse culinária, por exemplo, poderíamos encontrar alguém que gosta muito de pizza, um pouco de lasanha, e não gosta de salada. Caso perguntássemos sobre música, poderíamos descobrir que essa mesma pessoa prefere *jazz* a *rock* ou MPB. Essas preferências podem ser representadas por grafos, associando os exemplos aos nós e as relações de gosto às arestas direcionadas que unem esses pares de nós, indicando qual dos elementos de cada par está à frente do outro na escala de gostos daquele indivíduo.

Faremos agora uma breve revisão dos conceitos de teoria de grafos que serão relevantes ao trabalho.

1.2 GRAFOS

Um grafo G é formado por um conjunto de vértices e um conjunto de arestas. Formalmente:

Um conjunto V de vértices V(G); e
Um conjunto E de arestas E(G).

Vale ressaltar que os termos **vértice** e **nó** são equivalentes no contexto da teoria de grafos. Ambos os termos são utilizados neste trabalho.

Um dado grafo recebe uma ou mais classificações, dependendo das características que apresenta. Por exemplo, um grafo cujo conjunto de arestas é vazio é chamado de *vazio*; um grafo cujas arestas têm pesos é conhecido como *ponderado*. Os grafos em que temos interesse particular, nessa pesquisa, são grafos *completos*, *direcionados*, e *acíclicos*. Essas denominações são detalhadas mais a fundo abaixo.

1.2.1 Grafos acíclicos

Um grafo pode ser abstraído como um mapa entre ilhas, em que os vértices são as ilhas, e as arestas são pontes entre elas. Sendo assim, fica claro que é possível caminhar através do grafo, andando de vértice para vértice por meio das arestas. Um *caminho* é um conjunto de passos através do grafo que se inicia em um vértice qualquer e passa por algum número de outros vértices, sempre passando pelas arestas, até alcançar um outro vértice qualquer.

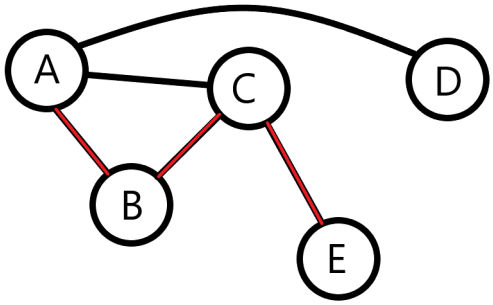

Figura 1 – Um grafo qualquer; demarcado um exemplo de caminho em vermelho

Com isso em mente, um *ciclo* em um grafo é um caminho que sai de um vértice **a**, e volta ao mesmo vértice **a**, sem passar por qualquer aresta duas vezes. Um grafo é chamado *cíclico* caso apresente um ou mais ciclos dentro de si, e, logicamente, é chamado *acíclico*, caso contrário.

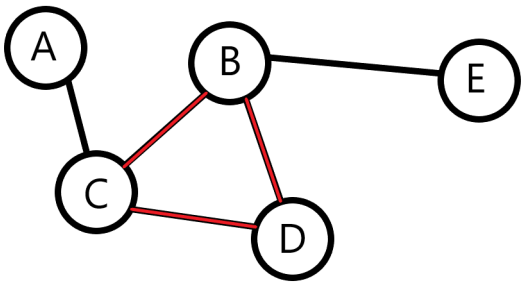

Figura 2 – Um grafo cíclico; demarcado um exemplo de ciclo em vermelho

1.2.2 Grafos direcionados

No caso dos grafos *direcionados*, suas arestas apresentam direcionamento, ou seja, para cada aresta, a ordem dos vértices importa. Num grafo direcionado, a aresta **ab** indica que o vértice **a** aponta para o vértice **b**, mas não que **b** aponta para **a**, o que seria o caso num grafo *não direcionado*.

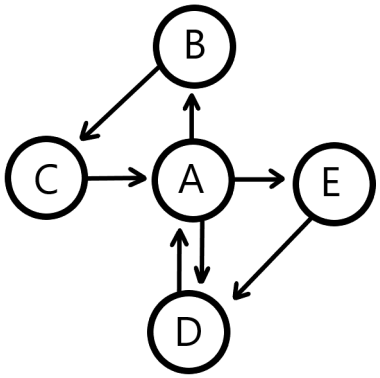

Figura 3 – Um grafo direcionado

Arestas que apontam em sentidos opostos não são equivalentes, apesar de ligarem o mesmo par de vértices, e, assim, as arestas **ab** e **ba** devem ser representadas separadamente, caso estejam presentes em um grafo. No exemplo acima, as arestas **ad** e **da** são representadas.

1.2.3 Grafos completos

Em grafos *completos*, o conjunto **E** de arestas inclui arestas para todos os pares de vértices possíveis. Em outras palavras, para qualquer par de vértices **ab**, existe ao menos uma aresta entre eles. Em geral, a presença de mais de uma aresta entre vértices é redun-

dante, e, assim, só é necessário representar uma das arestas presentes. Grafos completos também podem ser direcionados, como demonstrado pela figura abaixo.

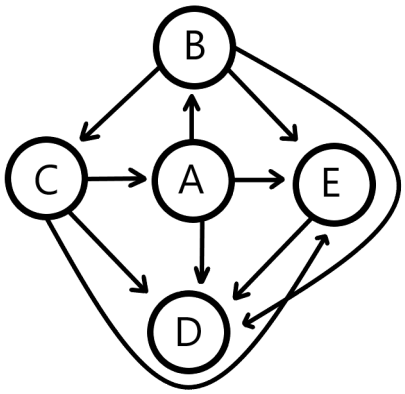

Figura 4 – Um grafo completo e direcionado

Quando um grafo é completo e direcionado, ele também pode ser chamado de *torneio*.

1.3 OBJETIVO

Se desejamos representar a rede de gostos de uma pessoa através de um grafo, então desejamos um grafo ordenado, para que possamos saber como os exemplares de um dado tópico se comparam um com o outro. Como queremos também categorizar os gostos como um todo, queremos que esse grafo também seja completo, para que seja possível comparar qualquer um objeto com qualquer outro objeto dentro do assunto. Como desejamos um grafo que seja completo e direcionado, desejamos um torneio.

Tratando-se de uma rede de preferências, também temos um detalhe a mais a manter em mente. O grafo sempre respeita o princípio da *transitividade*. Isso significa que se os gostos da pessoa forem tais que ela prefere música pop a forró, e música clássica a pop, então ela também prefere música clássica a forró. O objetivo, então, é um ou mais algoritmos que sejam capazes de realizar esse preenchimento de forma relativamente eficiente, levando em conta as características que o grafo deve atender.

Para facilitar a computação do problema, poderemos utilizar uma consequência lógica da transitividade do torneio: a *aciclicidade* do grafo. Em outras palavras, se existem as arestas **ab** e **bc**, então $\mathbf{a} > \mathbf{b}$ e $\mathbf{b} > \mathbf{c}$. Consequentemente, $\mathbf{a} > \mathbf{c}$, e a aresta **ac** também deverá ser adicionada. Não é necessário fazer a comparação entre **a** e **c** diretamente, porque $\mathbf{a} > \mathbf{c}$ não faria sentido. A prova da relação entre aciclicidade e transitividade é detalhada abaixo.

Lema 1. O torneio é acíclico se, e somente se, for transitivo.

Prova. Seja G um torneio com vértices **x**, **y**, e **z**. Se G é acíclico, então ao adicionarmos as arestas **xy** e **yz**, necessariamente não podemos adicionar a aresta **zx**, pois do contrário G não é acíclico. Como G é um torneio, então G é completo, logo é necessária uma aresta entre **z** e **x**. Como **zx** é inviável, necessariamente temos que adicionar **xz** Assim, temos que $\mathbf{x} > \mathbf{y}$ e $\mathbf{y} > \mathbf{z}$, resultando em $\mathbf{x} > \mathbf{z}$. Logo, se G é acíclico, então ele também é transitivo.

Seja G o mesmo torneio. Suponha que G é transitivo. Se adicionarmos as arestas **xy** e **yz**, temos que adicionar também a aresta **zx** para gerar um ciclo. Entretanto, essa aresta tornaria impossível adicionar a aresta **xz**, então o grafo não poderia ser transitivo. Não é possível gerar um ciclo se G é transitivo, então se G é transitivo, ele também é acíclico.

O torneio desejado é transitivo, e, por consequência, acíclico. Logo, ele será uma árvore com uma raiz e uma única folha, de tal forma que todos os nós apontarão para todos os outros nós que são inferiores a si próprios. Haverá exatamente um caminho principal que segue da raiz da árvore até a folha, que representa exatamente os gostos da pessoa.

Mas se o grafo final pode ser representado por um único caminho, com todas as outras relações podendo ser inferidas indiretamente, então não seria mais simples apenas aplicar um algoritmo de ordenação nos nós? Afinal, apesar dos nós não terem uma ordenação natural, como seria o caso se fossem apenas números sendo comparados, um algoritmo comparativo seria perfeitamente capaz de gerar essa ordenação em tempo $O(nlog(n))$.

De fato, para ordenar os vértices, seria possível aplicar um algoritmo genérico, como o *QuickSort* ou o *MergeSort*, e obter uma ordenação que corresponde à devolvida pela pessoa. Entretanto, esse trabalho seria exaustivo para a pessoa, uma vez que, pelo próprio fato dos elementos não apresentarem uma ordem natural, somos obrigados a questioná-la a respeito de cada par para fazer uso de um algoritmo qualquer, e isso se tornaria inviável para grafos muito grandes. Os algoritmos apresentados buscam fazer uso da transitividade do grafo para economizar iterações.

Além disso, algoritmos de ordenação comuns procuram ordenar todos os objetos do *dataset* que recebem. Isso normalmente é o comportamento desejado, mas como será detalhado à frente, não necessariamente pretendemos preencher o grafo por completo.

Também foi um objetivo do trabalho gerar um aplicativo que simula o comportamento de um dos algoritmos ao receber *inputs* do usuário. Esse aplicativo foi desenvolvido para demonstrar a gamificação de um processo que normalmente seria executado sem intervenção humana, e como essa gamificação pode ser aplicada em contextos de coleta de dados.

Ao unir o aplicativo e os algoritmos de preenchimento, chegamos a uma questão interessante: para jogar o jogo apresentado no aplicativo, não é necessário que o grafo esteja de fato completo, mas sim que um certo número de comparações já são possíveis. Em outras palavras, desejamos que um certo número de arestas já esteja presente no

grafo. Verificamos também como os algoritmos se comportam em termos de velocidade de preenchimento até esse certo *threshold* de comparações viáveis.

Finalmente, também consideramos a possibilidade de o usuário ter uma quantidade máxima de esforço que está disposto a fazer para jogar o jogo. Em outras palavras, o usuário realizará no máximo um certo número de comparações entre vértices **a** e **b** antes de parar. Usando esse esforço máximo como critério de parada, podemos pensar em como maximizar o número de arestas que podem ser encontradas, através das escolhas de quais vértices deverão ser comparados. Também podemos fazer uso da transitividade nesse caso, derivando relações que não sejam explicitamente delineadas pelo usuário e, assim, não necessitando comparar todos os vértices entre si.

As seções 2.2, 2.3, e 2.4 abordam os três algoritmos criados. Dentro delas, eles são analisados e sua *performance* é avaliada. O código referente a esses algoritmos se encontra no apêndice, e no GitHub do projeto. Os resultados empíricos apresentados pelos algoritmos são expostos no capítulo 3. O capítulo 4 se refere ao aplicativo, sua implementação, e resultados observados.

2 ALGORITMOS PROPOSTOS

2.1 IMPLEMENTAÇÕES DE GRAFOS E NÓS

Desejamos um algoritmo que seja capaz de preencher grafos vazios de forma que se tornem *completos*, *direcionados*, e *acíclicos*, ou seja, torneios acíclicos. Também sabemos que o grafo final é *transitivo*, por ser um torneio acíclico.

Para ser possível simular esse preenchimento, é necessário que sejamos capazes de armazenar uma estrutura de dados que represente o grafo e os nós que ele contém. Assim, poderemos tanto visualizar o progresso e o resultado, quanto realizar testes com quantidades de nós que seriam intratáveis à mão. Dessa forma, criamos, em Python 3, as classes *Graph* e *Node* para servir nossos propósitos. Os códigos das classes são detalhados no apêndice, na seção A.1. Os códigos também podem ser acessados nos nossos repositórios do GitHub: códigos teóricos e backend do aplicativo.

É importante mencionar a complexidade de adicionar uma nova aresta no grafo. Esse processo tem complexidade $O(n^2)$, e a prova disso é detalhada abaixo. Apesar disso, ele foi tratado como tendo complexidade constante na análise da complexidade dos algoritmos de preenchimento. Isso porque o que nos interessa é a complexidade desses mesmos algoritmos, e em particular a relação entre um e outro. Para obter a complexidade real dos algoritmos da forma como eles foram implementados, é necessário multiplicar as complexidades descritas em suas seções por n^2.

Lema 1. O número de operações para adicionar uma aresta válida no grafo é proporcional ao número de vizinhos dos vértices envolvidos.

Prova. Seja G um grafo incompleto com n vértices. Cada vértice possui um certo número de vizinhos de entrada e de vizinhos de saída. Sejam **a** e **b** vértices distintos e desconexos entre si. Seja L o número de vizinhos de entrada do vértice **a**, + 1. Seja M o número de vizinhos de saída do vértice **b**, + 1. Ao adicionar a aresta **a** > **b**, fazemos com que **a** passe a ser vizinho de entrada de **b** e de todos os vizinhos de saída de **b**. Assim, realizamos M operações. Esse processo é repetido para cada vértice de entrada de **a**, totalizando L vezes, uma para **a** e uma para cada um de seus vizinhos de entrada. Assim, a adição de uma aresta válida leva um total de L * M operações.

Lema 2. Dados dois vértices distintos e desconexos entre si, a contagem deles próprios mais seus vizinhos envolve no máximo todos os vértices do grafo.

Prova. Seja G o mesmo grafo incompleto com n vértices. O número máximo de vizinhos de entrada ou de saída que um vértice pode ter em um grafo incompleto é n - 2. Logo, L < n e M < n. O valor máximo que L e M podem assumir é n - 1, e o valor

mínimo é 1. Suponha que L assume seu valor máximo. Então, **a** possui n - 2 vizinhos de entrada. Sabemos que **a** e **b** são desconexos entre si. Logo, devido à transitividade do grafo, sabemos que todos os vizinhos de entrada de **a** também são desconexos de **b**. Como **a** é conexo com todos os nós do grafo, exceto **b**, então **b** não é conexo com nenhum nó. Assim, M assume seu valor mínimo de 1. O número de vértices envolvidos entre **a**, **b**, e os vizinhos dos dois é L + M, ou: $n - 1 + 1 = n$. Logo, L + M <= n.

Lema 3. A complexidade de adicionar uma aresta válida no grafo é O(n^2).

Prova. Sabemos que adicionar uma aresta no grafo incompleto G leva L * M operações. Também sabemos que L e M são números entre 1 e n - 1, e que L + M <= n. Logo, o pior caso do processo é aquele que maximiza L * M, num caso análogo ao de maximizar a área de um retângulo, dado seu perímetro máximo. Para fazer essa maximização, fazemos com que L e M sejam o mais próximo de $\frac{n}{2}$ possível. Para um número par de vértices no grafo, L = M = $\frac{n}{2}$. Para um número ímpar, um desses números recebe mais uma unidade. Dados L = M = $\frac{n}{2}$, então: $L * M = \frac{n}{2} * \frac{n}{2} = \frac{n^2}{4}$. Descartando o escalar, levamos um total de n^2 operações no pior caso.

Lema 4. A complexidade de verificar que uma aresta nova é válida é O(n^2).

Prova. Seja G o mesmo grafo incompleto. Sejam **a** e **b** os mesmos vértices desconexos entre si. Da mesma forma que adicionar uma aresta válida, verificar a validade requer L * M operações, uma vez que, para cada vértice **a** e acima dele, é necessário conferir que não há *loop* gerado adicionando uma aresta entre esse vértice e cada um dos vértices b e abaixo. Assim, o cálculo do número de operações para realizar essa verificação é o mesmo de antes: maximizar L * M, dado L + M <= n. Assim, a maximização é atingida aproximando L e M de $\frac{n}{2}$, e fazendo: $L * M = \frac{n}{2} * \frac{n}{2} = \frac{n^2}{4}$. Descartamos o escalar novamente, e levamos um total de n^2 operações no pior caso.

Como sabemos que a complexidade de adicionar uma aresta válida no grafo é de O(n^2), a complexidade de verificar que essa aresta é válida também é de O(n^2), e que esses dois processos são realizados sequencialmente, então a complexidade de *addNeighbor* é de O(n^2). Da mesma forma, como *addEdge* é, no máximo, dois *addNeighbor* sequenciais, sua complexidade também é de O(n^2).

Com essas estruturas de dados em mãos, somos capazes de simular o preenchimento de grafos usando os algoritmos detalhados abaixo, e, em particular, simular um número muito grande de vezes esse preenchimento. Assim, podemos fazer uso da Lei dos Grandes Números para garantir que o tempo médio de execução encontrado é o tempo médio de execução real. Os algoritmos utilizados são detalhados abaixo.

2.2 ALGORITMO 1 - NAIVECHOICE

Inicialmente, ao abordar um problema desse tipo, poderíamos pensar em uma solução simples, mas funcional, para ter um ponto de partida a partir do qual iterar e gerar melhorias de eficiência. No caso do preenchimento dos grafos em que nos interessamos, o objetivo final é um grafo em que sabemos as relações de todos os vértices **a** com todos os outros vértices **b**. Assim, uma proposta de algoritmo que é capaz de preencher o grafo de forma ingênua é apresentada abaixo.

Algorithm 1 Algoritmo naiveChoice

Entrada: $G \leftarrow$ Grafo vazio
Saída: $G \leftarrow$ Torneio acíclico; $N \leftarrow$ número de iterações do algoritmo

```
N ← 0
for node in G.nodes do
    for otherNode in G.nodes do
        N ← N + 1
        moedaHonesta(node, otherNode)          ▷ Decide qual nó fica na frente
        if node fica na frente then
            G.adicionarAresta(node, otherNode)
        end if
        if otherNode fica na frente then
            G.adicionarAresta(otherNode, node)
        end if
    end for
end for
return N
```

O funcionamento do algoritmo é simples de visualizar e compreender. Para cada nó no grafo, perguntamos qual a relação dele com cada outro nó do grafo, exceto si próprio. O processo para quando todos os nós tiverem sido questionados. Temos, então, um algoritmo que funciona, e que sempre para.

Notavelmente, a escolha de relacionamento entre os nós não é uma escolha no sentido tradicional, mas sim é representada por uma jogada de uma moeda honesta. Isso porque o método para a escolha de se $\mathbf{a} > \mathbf{b}$ ou $\mathbf{b} > \mathbf{a}$ é irrelevante para nossos propósitos, uma vez que não estamos interessados em alcançar um ou outro grafo específico que atenda às restrições estabelecidas. Logo, podemos usar esse modo de escolha, que é muito mais rápido do que esperar um *input* de um usuário, por exemplo. Essa filosofia também se aplica aos outros algoritmos de preenchimento.

O algoritmo itera, para cada nó do grafo, um número de vezes igual a n - 1, sendo n o número de nós no grafo. Em outras palavras, o algoritmo itera n * n - 1 vezes, para uma eficiência de O(n * n - 1), ou aproximadamente O(n^2).

Imediatamente, podemos notar ineficiências nesse algoritmo. Em primeiro lugar, ignoramos a transitividade que o grafo apresenta e isso, pareado com o fato que sempre

verificamos todas as relações **ab** e **ba** possíveis, significa que conferimos relações que já foram preenchidas anteriormente.

Além disso, partindo do mesmo problema de ignorar a transitividade, é tecnicamente possível que o grafo esteja preenchido muito antes do algoritmo terminar de executar. Na verdade, como mostraremos no capítulo 3, ele sempre preenche o grafo antes de parar. Mesmo assim, o algoritmo não tem a capacidade de perceber isso, e itera até o final independente de se mais iterações são necessárias ou não.

Com essas questões em mente, temos bons pontos de partida para melhorar o algoritmo de preenchimento.

2.3 ALGORITMO 2 - RANDOMCHOICE

A introdução de aleatoriedade no processo de preenchimento do grafo vazio apresenta vantagens sobre o método anterior. Em particular, é possível abrir mão de uma estrutura rígida de checagem ordenada, que pode restringir a busca desnecessariamente. Ao permitir que o algoritmo explore o grafo de forma não-determinística, podemos encontrar rotas que o preencham muito mais rapidamente do que o algoritmo inicial. Um algoritmo que realiza esse preenchimento de forma aleatória é apresentado abaixo.

Algorithm 2 Algoritmo randomChoice

Entrada: $G \leftarrow$ Grafo vazio
Saída: $G \leftarrow$ Torneio acíclico; $N \leftarrow$ número de iterações do algoritmo

```
N ← 0
while G incompleto do
    node ← escolherNóArbitrário(G)
    otherNode ← escolherNóArbitrário(G) ▷ Os nós têm que ser distintos; a ordem de escolha importa
    G.adicionarAresta(node, otherNode)
    N ← N + 1
    checarParada(G)
end while
return N
```

Como no caso do algoritmo ingênuo, o funcionamento do algoritmo puramente aleatório é de simples compreensão. A cada iteração, o algoritmo escolhe um par completamente arbitrário de nós distintos **a** e **b**. Então, ele tenta inserir a aresta entre esses nós, conta mais uma iteração, e verifica se o grafo está completo. Caso esteja, ele retorna.

Como discutido anteriormente, a escolha de ordenação $\mathbf{a} > \mathbf{b}$ ou $\mathbf{b} > \mathbf{a}$ é representada, nos algoritmos, por uma moeda honesta. Em outras palavras, escolhemos arbitrariamente uma ordem ou a outra, com a mesma probabilidade para ambas. Essa escolha, dentro do *randomChoice*, é implícita. Seria possível explicitar e forçar que, após a escolha de dois nós, definimos um ou o outro para ser o nó maior. Porém, a própria escolha de dois nós já

gera uma ordenação que podemos utilizar. A escolha de **ab**, e subsequente decisão entre **a** > **b** ou **b** > **a**, tem a mesma probabilidade que a escolha de **a** e então **b** ou **b** e então **a**.

Lema 1. A probabilidade de escolher **a** e então **b**, diferentes entre si, é igual à probabilidade de escolher **ab** não direcionado e então escolher **a** > **b**.

Prova. Seja G um grafo completo com n vértices. Seja P(x) a probabilidade de escolher o vértice **a** e, então, de escolher o vértice diferente **b**. Então, P(x) é dada por $\frac{1}{n} * \frac{1}{n-1} = \frac{1}{n(n-1)}$.

Seja P(y) a probabilidade de escolher a aresta não direcionada **ab**. O número de arestas em um grafo completo com n vértices é dado por $\frac{n(n-1)}{2}$. Logo, P(y) é dada por $(\frac{n(n-1)}{2})^{-1} = \frac{2}{n(n-1)}$. Para obter a escolha entre **a** > **b** e **b** > **a**, jogamos uma moeda honesta, atribuindo a mesma probabilidade para cada caso. Como são dois casos, essa probabilidade é de $\frac{1}{2}$. Assim, a probabilidade de escolher **a** > **b** é dada por P(y) * $\frac{1}{2}$, ou: $\frac{2}{n(n-1)} * \frac{1}{2} = \frac{1}{n(n-1)}$, que é exatamente igual a P(x).

Devido à natureza não determinística do algoritmo aleatório, e em contraste com o algoritmo ingênuo anterior, não é possível pará-lo em um ponto predeterminado e ter a garantia de que o grafo está preenchido. Assim, é necessário uma forma de verificar, a cada iteração, a condição de parada, parando o algoritmo caso a mesma esteja satisfeita. A função que realiza essa checagem depende da condição que desejamos testar. Abaixo, temos um exemplo de uma função que define o critério de parada como a completude do grafo.

Algorithm 3 Função auxiliar checkCompleteness

Entrada: $G \leftarrow$ Grafo qualquer
Saída: **Verdadeiro**, se o grafo for completo; **Falso**, do contrário

```
GrauMáximo ← G.grauMáximo
ListaDeGraus ← G.listaDeGraus
for grau in ListaDeGraus do
    if grau ≠ GrauMáximo then return False
    end if
end for
return True
```

Em um grafo completo, todos os vértices têm o mesmo grau, o grau máximo possível. Em um grafo com n vértices, esse grau é n - 1. Uma forma de verificar se um grafo é completo é conferindo se o grau de todos os vértices é exatamente o grau máximo; se forem, então o grafo é completo. É nessa lógica que se baseia a função *checkCompleteness*. Ela recebe como argumento um *Graph* e tira dele o grau máximo e a lista de graus dos nós. Se um dos nós não tiver o grau máximo, retorna *false*; do contrário, retorna *true*.

O mesmo não determinismo que o *randomChoice* utiliza para acelerar o preenchimento do grafo, em relação ao algoritmo ingênuo, também pode causar uma demora maior e

imprevista. Em teoria, o algoritmo pode escolher repetidamente o mesmo par de vértices por uma quantidade muito grande de iterações, e desperdiçar tempo computacional sem progredir com o preenchimento.

Na prática, a probabilidade de isso acontecer é pequena. Embora a duração do algoritmo seja variável, e o pior caso ser muito demorado, em média ele é mais eficiente do que o *naiveChoice*. Esses resultados são demonstrados no capítulo 3.

Também em relação ao pior caso, existe a questão de se o algoritmo poderia ficar em loop eternamente, ou, em outras palavras, se o algoritmo pode ficar sujeito ao problema da parada. A resposta para essa pergunta é não; o algoritmo *randomChoice* sempre leva um tempo finito para retornar. As provas dessa afirmação são detalhadas abaixo.

Lema 2. É possível representar um grafo através de uma cadeia de Markov com um número finito de estados e uma única classe comunicante.

Prova. Seja G um grafo com n vértices. Seja M uma cadeia de Markov de tempo discreto cujos estados representem todas as escolhas de pares ordenados de vértices de G. Então, M possui $n * n - 1$ estados. Para cada estado de M, a probabilidade de transição para cada outro estado, incluindo si próprio, é igual, dada por $\frac{1}{n(n-1)}$. Todos os estados são, então, acessíveis por todos os outros estados em, no máximo, n - 1 passos, e esses estados formam uma única **classe comunicante**.

Lema 3. A cadeia detalhada acima é irredutível.

Prova. Como todos os estados são acessíveis por todos os outros em um número finito de passos, no máximo n - 1, então a cadeia é **irredutível**.

Lema 4. Os estados da cadeia são recorrentes.

Prova. Suponha, por absurdo, que um dos estados de M é **transiente**, ou seja, a probabilidade da cadeia não alcançar esse estado após um número infinito de passos é 1. Entretanto, transiência é uma propriedade de classe, ou seja, todos os estados na mesma classe comunicante (nesse caso, M como um todo) são transientes. Logo, a probabilidade da cadeia nunca retornar ao mesmo estado é 1. Entretanto, como a cadeia tem um número finito de estados, então eventualmente ela é obrigada a alcançar um estado em que já esteve. Chegamos a uma contradição. Logo, todos os estados de M são **recorrentes**.

Lema 5 A cadeia ter as propriedades apresentadas acima demonstra a parada do algoritmo.

Prova. Como a cadeia é irredutível, finita, e todos os seus estados são recorrentes, então todos os estados serão visitados em um número finito de passos. Como os estados representam todos os possíveis pares ordenados de vértices do grafo, então todas as

possibilidades de aresta serão testadas em um número finito de passos. Logo, o grafo é completo em um número finito de passos.

O algoritmo aleatório itera sempre uma única vez por par. Ele demora mais ou menos dependendo apenas de quantos pares precisarem ser analisados. A ordem em que os pares de vértices são escolhidos importa, devido à transitividade das relações, o que faz com que um preenchimento demore mais ou menos que o anterior. O número de pares a serem verificados depende linearmente do número de vértices no grafo. O número de vértices que o algoritmo efetivamente verifica antes de terminar é um número finito, que não depende do número de vértices, mas sim da sorte ou azar da função de aleatoriedade. Como o algoritmo escala linearmente, sua complexidade é dada por $O(n)$.

Um aspecto do problema evidenciado pelo algoritmo aleatório é o fato que o grafo completo é uma árvore com uma única raiz e uma única folha. Em particular, o grafo possui um único caminho que percorre todos os vértices exatamente uma vez, e, por conta da transitividade, é possível derivar todas as outras relações tendo em mãos apenas esse caminho principal. Logo, o número mínimo de iterações que um algoritmo poderia ter para um grafo com n vértices é n.

2.4 ALGORITMO 3 - SNAKECHOICE

Como ter o caminho principal em mãos é o método mais eficiente para o preenchimento do grafo, em teoria um algoritmo ideal para a resolução do problema seria um que busca especificamente por esse caminho. O algoritmo *snakeChoice*, apresentado abaixo, busca unir a velocidade não-determinística do *randomChoice* e a metodicidade do *naiveChoice* para alcançar esse objetivo rapidamente.

A lógica do algoritmo *snakeChoice* é menos imediatamente evidente do que a dos outros dois propostos anteriormente, mas seu funcionamento básico também é simples. Inicialmente, são escolhidos dois nós quaisquer. Esses dois nós são comparados, e, então, um deles é escolhido para ser descartado. É sorteado um novo nó, e o nó que foi mantido é comparado com ele, e novamente um dos nós comparados é descartado aleatoriamente. Esse processo se repete até que a condição de parada seja alcançada.

Assim como no *randomChoice*, o processo de comparação e subsequente decisão de qual dos nós deve ser descartado é truncada dentro do algoritmo. Isso porque, assim como no algoritmo anterior, esse processo pode ser realizado de forma tanto implícita quanto explícita, e a forma implícita permite uma maior simplicidade na implementação e compreensão do processo.

A ideia por trás do *snakeChoice* é de procurar o caminho principal o mais rápido possível, em contraponto com simplesmente buscar pares completamente arbitrários a cada iteração. Para isso, mantemos um dos nós que já foram questionados, para que

Algorithm 4 Algoritmo snakeChoice

Entrada: $G \leftarrow$ Grafo vazio
Saída: $G \leftarrow$ Torneio acíclico; $N \leftarrow$ número de iterações do algoritmo

```
N ← 0
node ← escolherNóArbitrário(G)
otherNode ← escolherNóArbitrário(G)      ▷ Os nós têm que ser distintos; a ordem de escolha não importa
while G incompleto do
    moedaHonesta(node, otherNode)                ▷ Decide qual nó fica na frente
    if node fica na frente then
        G.adicionarAresta(node, otherNode)
        otherNode ← escolherNóArbitrário(G) ▷ Novo nó é distinto do nó que ficou na frente
    end if
    if otherNode fica na frente then
        G.adicionarAresta(otherNode, node)
        node ← escolherNóArbitrário(G) ▷ Novo nó é distinto do nó que ficou na frente
    end if
    N ← N + 1
    checarParada(G)
end while
return N
```

tenhamos sempre garantidamente uma conexão com o grafo já encontrado. Assim, não criamos classes distintas de nós que são conexos entre si, mas desconexos do resto do grafo.

Vale ressaltar que a escolha de qual nó manter ser aleatória é importante. Caso não fosse, por exemplo, se mantivéssemos sempre o nó escolhido, o algoritmo pararia quando encontrássemos a raiz da árvore. A mesma coisa aconteceria com a folha e manter sempre o nó não escolhido. Essas políticas só funcionariam caso, por pura sorte, começássemos em um extremo da árvore e seguíssemos exatamente ao longo do caminho principal.

Em termos de complexidade, o algoritmo itera uma única vez, descarta um dos nós usados, e segue para um novo par. Da mesma forma que no algoritmo aleatório, a demora maior ou menor depende apenas do número de pares que são analisados. Esse número escala linearmente com o número de nós do grafo, e, novamente, o histórico de pares escolhidos determina, através da transitividade, quantos são os pares que precisam ser analisados. Se encontrarmos $\mathbf{a} > \mathbf{b}$ e $\mathbf{b} > \mathbf{c}$, por exemplo, então não precisamos encontrar $\mathbf{a} > \mathbf{c}$ diretamente, mesmo essa aresta já contribuindo para a completude. O histórico depende apenas da função de aleatoriedade, e pode ser descartado. Como o algoritmo escala linearmente com o tamanho do grafo, sua complexidade é $\mathrm{O}(n)$.

2.5 ALGORITMOS DE ESFORÇO

Os algoritmos apresentados até agora são algoritmos cujo critério de parada é um de completude. Em outras palavras, ao atingirmos um certo *threshold* de preenchimento do grafo, seja um número arbitrário de arestas ou a completude do grafo, o algoritmo para. Por outro lado, ele só para ao atingir essa condição, não importando quantos passos levou para chegar a esse ponto.

Do ponto de vista de um computador, essa distinção não faz diferença prática. A menos que o processo entre em *loop*, ele vai parar em algum momento, e demorar mais ou menos para chegar na parada é irrelevante. Entretanto, do ponto de vista de um usuário, uma demora excessiva para o término do algoritmo é um problema, especialmente um algoritmo que requer *inputs* inseridos manualmente a cada iteração. Com isso em mente, também consideramos cenários em que o critério de parada do algoritmo é um certo número de iterações. Dessa forma, o dado que buscamos não é o tempo necessário para alcançar a parada, mas sim o número de arestas obtidas ao final da execução.

Também é de interesse como podemos maximizar esse número de arestas. Para tal, levamos em conta duas variáveis: o número de vértices a serem comparados a cada iteração, e a política de escolha desses vértices.

2.5.1 Número de Vértices Comparados

Para entender a influência do número de vértices comparados a cada vez na execução do algoritmo, podemos abstrair cada iteração como uma ordenação de $\mathbf{t}$ vértices de cada vez. Como os grafos que nos interessam são transitivos, podemos simplesmente determinar uma única ordem entre todos os $\mathbf{t}$ vértices à mão, e as relações não testadas são inferidas a partir dessa ordem. Como estamos utilizando um algoritmo de ordenação, então cada iteração leva $t * log(t)$ passos para ser resolvida. Esse é o custo de esforço que o usuário deve pagar a cada iteração; o algoritmo para quando o esforço total feito pelo usuário atinge um máximo arbitrário.

Ao ordenar $\mathbf{t}$ vértices, geramos um total de $\frac{t*(t-1)}{2}$ arestas (que podem já estar presentes no grafo principal ou não). Entretanto, como estabelecido antes, cada iteração tem um custo de $t * log(t)$ passos. O dado que nos interessa, nesse caso, é se vale mais a pena comparar os vértices par a par, pagando um custo de um único passo, mas obtendo apenas uma aresta de cada vez, ou se é mais eficaz comparar grupos de cada vez, fazendo mais esforço a cada iteração mas obtendo mais arestas de uma vez.

Também é necessário levar em conta o envolvimento do usuário nesse caso. Não queremos que ele seja forçado a ordenar conjuntos muito grandes de vértices a cada vez, porque isso levaria a uma experiência de usuário ruim. Assim, desejamos que $\mathbf{t}$ não seja muito grande. Nos testes realizados, restringimos $\mathbf{t} = 5$.

2.5.2 Políticas de Escolha de Vértices

Em termos de escolha de vértices, a grande questão é da diferença entre aleatoriedade completa ou parcial. Em particular, desejamos saber se vale mais a pena, dentro do contexto do aplicativo, manter algum grau de determinismo, para influenciar o grafo final.

A aleatoriedade completa apresenta maior potencial de tocar mais vértices do grafo. Como ela não é restrita, ela tem mais chances de escolher vértices novos, que geram arestas novas, em vez de acidentalmente refazer comparações que já foram feitas. Entretanto, em grafos muito grandes, é possível que sejam comparados muitos vértices poucas vezes, levando a um grande número de subconjuntos disjuntos de poucos vértices cada. Isso leva a uma jogabilidade diferente dentro do contexto do aplicativo, já que, a cada rodada, o usuário seria levado a comparar dois vértices que potencialmente não foram mostrados antes, e não serão mostrados novamente, em vez de determinar a opinião de um segundo usuário e gerar uma ordenação entre os tópicos apresentados.

A aleatoriedade parcial, por outro lado, restringe o algoritmo a sempre manter uma parte dos vértices que já foram comparados anteriormente, e o resto é escolhido ao acaso. Isso é a premissa do algoritmo *snakeChoice*, discutido na seção 2.4. Dessa forma, o grafo gerado ao final sempre terá um único subconjunto de vértices ordenados entre si. Vale ressaltar que esse subconjunto não necessariamente é completo, e possivelmente não alcança tantos vértices quanto um algoritmo puramente aleatório. Entretanto, ele garantidamente exibe comparações em que pelo menos um dos vértices já apareceu, ou aparecerá de novo.

3 RESULTADOS COMPUTACIONAIS

Tendo os algoritmos implementados, e seu funcionamento, em mente, podemos agora executá-los para verificar os resultados de forma empírica. Buscamos resultados comparativos tanto de quanto tempo os algoritmos levam para preencher completamente subgrafos de poucos nós, simulando a experiência do usuário com o aplicativo, quanto com alcançar um certo número de arestas em um grafo com mais nós, para verificar a velocidade com que os algoritmos atingem o *threshold* de jogabilidade num vácuo.

Entretanto, executar cada algoritmo apenas uma ou duas vezes não nos dá resultados confiáveis. Afinal, dois deles fazem uso de aleatoriedade, e, por definição, são imprevisíveis. É possível que na vez que testamos o *randomChoice*, por exemplo, ele demore uma quantidade absurda de tempo, enquanto que o *snakeChoice* encontre o caminho principal imediatamente e termine em pouquíssimos passos. Se executarmos o teste novamente, pode ser que aconteça a mesma coisa, ou um resultado completamente diferente.

Para evitar que a variância inata ao não determinismo polua nossos dados, é necessário que o teste seja executado várias vezes, para que então possamos tirar a média dos resultados. Assim, poderemos fazer uso da Lei dos Grandes Números, que nos garante que os resultados obtidos tenderão para a média real.

Dessa forma, cada algoritmo foi executado um total de 100000 (10^5) vezes. Os resultados de cada execução foram anotados, e, a partir deles, tiramos o número médio de passos que o algoritmo leva numa dada execução, quais foram o número mínimo e máximo de passos que os algoritmos levaram para terminar, e a porcentagem de execuções que atingiram um desempenho na média ou mais rápido. Cada teste de preenchimento completo foi executado um com grafos de 3, 7, e 10 vértices para cada algoritmo. Cada teste de alcançar um certo número de arestas no grafo foi executado com 15, 23, e 30 vértices. Para os propósitos do teste, esse número mínimo de arestas foi arbitrariamente decidido como 50.

Os algoritmos de esforço foram executados com 15 e 30 vértices, com grupos de 2 (par a par) e 5 vértices. Os resultados analisados são o número de arestas obtidas, o número de vértices com pelo menos uma aresta, e o tamanho médio dos subconjuntos de vértices gerados pela execução do algoritmo.

3.1 RESULTADOS - PREENCHIMENTO TOTAL

3.1.1 NaiveChoice

Os resultados apresentados pelo algoritmo ingênuo são esperados. Como sabemos que ele sempre itera um número de vezes igual ao número de vértices no grafo, para cada vértice do grafo, fica claro que o número de passos dados é sempre o quadrado desse

Quadro 1 – Resultados de Preenchimento - NaiveChoice

TOTAL DE VÉRTICES	MÍNIMO DE PASSOS	MÉDIA DE PASSOS	MÁXIMO DE PASSOS	EXECUÇÕES DE DURAÇÃO MÉDIA OU MENOR
3 Vértices	9	9	9	100%
7 Vértices	49	49	49	100%
10 Vértices	100	100	100	100%

mesmo número. Também fica evidente, por meio da completa ausência de variância nos resultados, que o algoritmo sempre executa até o final. Isso é mais uma coisa de que já estávamos cientes.

O que não fica evidente é quantas das execuções já haviam terminado de preencher o grafo, e mesmo assim continuaram iterando. Para obter esse resultado, criamos uma versão levemente modificada do *naiveChoice*, que retorna também se o grafo já estava completo antes do final da execução do algoritmo. Esse resultado aparece na tabela abaixo.

Quadro 2 – Execuções com grafo completo antes do final do algoritmo

TOTAL DE VÉRTICES	GRAFO COMPLETO ANTES DO FIM
3 Vértices	100%
7 Vértices	100%
10 Vértices	100%

Esses resultados inicialmente foram surpreendentes, mas, após consideração, eles são corretos. Pela propriedade da transitividade, não é necessário verificar todas as relações, como discutido na seção 2.4. Mas mesmo que o grafo não fosse transitivo, o *naiveChoice* sempre seria completo antes do fim da execução. Isso porque a verificação de relações entre **a** e **b** só precisa ser realizada uma vez. Em outras palavras, ao chegar no último vértice, ele já foi comparado com todos os outros durante suas respectivas iterações; assim, compará-lo novamente se torna redundante.

Esse é um ponto relevante para a melhora do algoritmo, mas, pelo próprio nome do método, fica claro que a eficácia não é o foco do *naiveChoice*. Existem várias melhoras que podem ser aplicadas a um algoritmo metódico; esta versão é apenas uma base funcional capaz de preencher o grafo sem erros, a partir da qual é possível iterar e criar versões melhores.

3.1.2 RandomChoice

Quadro 3 – Resultados de Preenchimento - RandomChoice

TOTAL DE VÉRTICES	MÍNIMO DE PASSOS	MÉDIA DE PASSOS	MÁXIMO DE PASSOS	EXECUÇÕES DE DURAÇÃO MÉDIA OU MENOR
3 Vértices	2	4	28	96,00%
7 Vértices	6	38	278	75,18%
10 Vértices	10	93	676	64,53%

Os resultados do algoritmo *randomChoice* imediatamente se mostram diferentes. O algoritmo exibe um comportamento mais rápido que o anterior, em média. Vale notar que essa média compara o desempenho do algoritmo com o desempenho do *naiveChoice*, e não com sua própria média. Além disso, devido à capacidade de determinar que o grafo já está completo sem precisar testar cada par de vértices, também se torna possível atingir a completude muito mais rápido que a média. Em particular, para 3 e 7 vértices, o algoritmo foi capaz de encontrar o caminho principal no número mínimo de passos, e quase fez o mesmo com 10 nós.

Também notavelmente, o *randomChoice* executa, em média, mais rápido que o *naiveChoice*. Não somente isso, ele atinge essa execução média, ou executa mais rápido, mais de 60% do tempo, mesmo no teste em que teve mais dificuldades, de 10 nós. Em outras palavras, a utilização do *randomChoice* se mostra uma melhora com relação ao algoritmo ingênuo.

Em contraposição, a coluna do número máximo de passos demonstra que, ocasionalmente, o algoritmo também demora muito mais do que a versão metódica, por vezes levando quase sete vezes mais tempo para terminar. Por isso, não é necessariamente óbvio que o algoritmo sempre deve ser utilizado. Em casos de tempo computacional restritivo, por exemplo, em que o tempo de execução do algoritmo não pode exceder um certo limite, pode ser arriscado fazer uso do não determinismo.

Uma forma de melhorar o algoritmo seria manter uma estrutura de dados que fosse capaz de saber com quais vértices o primeiro vértice arbitrário já tem conexão. Com isso, o algoritmo sempre poderia buscar um novo par apenas nos nós que não possuem aresta entre si, aumentando sua eficácia.

3.1.3 SnakeChoice

Quadro 4 – Resultados de Preenchimento - SnakeChoice

TOTAL DE VÉRTICES	MÍNIMO DE PASSOS	MÉDIA DE PASSOS	MÁXIMO DE PASSOS	EXECUÇÕES DE DURAÇÃO MÉDIA OU MENOR
3 Vértices	2	5	37	89,03%
7 Vértices	6	50	408	58,08%
10 Vértices	11	115	706	48,58%

O último algoritmo é, talvez, o mais surpreendente dos três. O algoritmo *snakeChoice* é melhor, em média, do que o algoritmo ingênuo, apenas para uma quantidade pequena de vértices no grafo. Ao executar os testes com 7 e 10 nós presentes, o tempo esperado de execução do algoritmo se mostra igual ou levemente pior.

Novamente, assim como no *randomChoice*, o *snakeChoice* é capaz de terminar sua execução em menos tempo do que a média. Também como no algoritmo anterior, ele termina de iterar com número de passos médio ou mais rápido cerca de 90% das execuções com 3 nós, e 60% do tempo para 7 nós. Entretanto, como seu tempo de execução esperado é maior do que o do algoritmo metódico, ele não é uma melhora tão clara quanto o algoritmo puramente aleatório. Além disso, para os testes de 10 nós, ele se mostra mais ineficiente do que o algoritmo ingênuo, executando mais rápido do que a média apenas 48% das vezes.

O número máximo de passos atingido também se mostra mais ineficiente do que o apresentado pelo *randomChoice*, passando de sete vezes mais lento do que a execução do *naiveChoice* em um dos testes. A mesma restrição do não determinismo também se aplica a esse algoritmo: se o tempo computacional gasto durante a execução não puder exceder um certo limite, é possível que o *snakeChoice* não seja a melhor escolha.

Também da mesma forma que o algoritmo anterior, o algoritmo *snakeChoice* se beneficiaria de ser capaz de escolher novos nós, durante sua execução, tais que o par formado entre o nó em mãos e o candidato seja um par sem aresta. Assim, ele sempre geraria uma nova aresta e contribuiria para o avanço em direção à completude.

De forma particular a ele, um novo critério de decisão de qual vértice deverá ser mantido em mãos para receber uma nova comparação, como por exemplo sempre manter um nó com a cardinalidade mais alta, ou sempre descartar o nó que foi mantido na iteração anterior, poderia ajudar o *snakeChoice* a terminar sua execução mais rápido.

3.2 RESULTADOS - PREENCHIMENTO PARCIAL

3.2.1 NaiveChoice

Quadro 5 – Resultados de Número de Arestas - NaiveChoice

TOTAL DE VÉRTICES	MÍNIMO DE PASSOS	MÉDIA DE PASSOS	MÁXIMO DE PASSOS
15 Vértices	14	14	55
23 Vértices	14	14	33
30 Vértices	14	14	25

Os resultados apresentados pelo *naiveChoice* durante os testes de preenchimento de um certo número de arestas são, novamente, razoavelmente previsíveis. Como o algoritmo segue um padrão determinístico de comparação, o número mínimo de passos é sempre

constante. Como o critério de parada é independente do número de vértices no grafo, então esse número mínimo também não escala com o tamanho do *dataset*.

As outras colunas, entretanto, nos são de interesse. O número máximo de passos atingido não é constante, e, em particular, o número máximo registrado de passos é menor, quanto maior o grafo. Isso se explica através de um pequeno não determinismo, presente mesmo no algoritmo ingênuo: a ordem de comparação dos vértices não é necessariamente constante.

Ao chamar um *for-each*, caso deixemos que execute até o final, não faz diferença se as comparações são feitas de **a** a **z** ou de **z** a **a**; todos os vértices serão comparados de qualquer forma. Mas quando paramos o algoritmo em um ponto arbitrário de sua execução, no caso quando o total de arestas no grafo atinge 50 ou mais, é possível que o *for-each* tenha escolhido um caminho que tenha conferido poucos vértices e já alcance o critério de parada, ou que tenha comparado um único vértice várias vezes e o número de passos registrado seja mais alto. Se deixássemos que o algoritmo terminasse, ele deterministicamente levaria n^2 passos para parar.

Notavelmente, conforme o tamanho do grafo se expande, também cai o número máximo registrado de passos que o algoritmo pode levar. Isso porque cada vértice **a** é comparado com cada outro vértice, antes de começarem as comparações do vértice **b**. Assim, o número de arestas presentes no grafo quando as novas novas comparações começam a serem feitas é maior, e o número de passos que o novo vértice potencialmente precisará dar tende a ser menor.

A média de passos ser 14 é explicada pelo fato desse número ser calculado, e então truncado. A maioria esmagadora dos testes levou 14 passos exatamente para terminar. Assim, quando a média foi calculada, ela tendeu para um número próximo de 14, mas um pouco mais alto. Ao ser guardada apenas como um número inteiro, a média se tornou apenas 14.

Um aspecto que o número de passos não aborda é a qualidade dos dados obtidos. Em particular, como o *naiveChoice* compara exatamente um vértice com todos os outros, e depois repete esse processo até alcançar o mínimo de arestas, temos muita informação sobre alguns poucos vértices e pouca sobre muitos outros. Isso não levaria a uma boa experiência durante o uso do aplicativo, porque o mesmo seria forçado a sempre incluir os vértices sobre os quais sabemos muito, levando a uma repetitividade muito grande. O que nos interessaria seria uma variedade de opções podendo ser apresentadas ao usuário, e não sempre uma comparação entre um exemplo e vários outros.

3.2.2 RandomChoice

Verificando os resultados apresentados pelo *randomChoice* no preenchimento parcial, podemos ver que os resultados são bastante similares àqueles vistos no *naiveChoice*. O

Quadro 6 – Resultados de Número de Arestas - RandomChoice

TOTAL DE VÉRTICES	MÍNIMO DE PASSOS	MÉDIA DE PASSOS	MÁXIMO DE PASSOS
15 Vértices	10	22	54
23 Vértices	11	22	39
30 Vértices	12	23	38

não-determinismo do algoritmo rapidamente fica evidente, uma vez que o mínimo e a média de passos são valores distintos entre si, diferente do algoritmo anterior.

O número mínimo de passos apresenta um escalamento com o tamanho do grafo. Porém, esse escalamento é bem pequeno, aumentando apenas em um passo a cada patamar. Isso é indicativo da política de escolha de vértices a serem testados. Como os pares são escolhidos de forma completamente arbitrária, a probabilidade de escolher um par que permita o uso da transitividade diminui proporcionalmente ao tamanho do grafo. Assim, a probabilidade de alcançar um número mínimo de passos para preencher as arestas desejadas também diminui.

A média de duração tendeu para aproximadamente 22 passos em todos os casos. Isso é um bom indicativo de que o algoritmo exibe uma certa independência do tamanho do grafo, e que pode ser utilizado mesmo em grafos maiores. O máximo de passos também tendeu a diminuir conforme mais vértices foram adicionados, o que mostra que o pior caso fica cada vez menos distante da média quanto maior for o grafo.

Novamente, a tabela do número de passos não aborda a qualidade dos dados obtidos. Dessa vez, como sabemos que o algoritmo lança mão do não-determinismo durante sua execução, podemos ter uma certeza muito maior de que as comparações serão mais variadas. Isso porque a probabilidade de escolher um único vértice para ser comparado várias vezes diminui conforme o grafo aumenta.

Entretanto, essa mesma probabilidade pequena significa que as comparações terão grandes chances de serem independentes entre si. Em outras palavras, é provável que o aplicativo seja forçado a mostrar quase sempre pares distintos de objetos, e poucas vezes uma comparação sequencial de **a** com **b** e então de **b** com **c**. Isso é um modo possível de tratar a execução do aplicativo, não necessariamente menos válida, mas também não necessariamente leva a uma experiência do usuário tão positiva, uma vez que ele descobrirá informações separadas sobre seus amigos, mais do que efetivamente listas ordenadas de preferências.

3.2.3 SnakeChoice

O preenchimento até um certo número de arestas é onde o *snakeChoice* se mostra eficaz. O número mínimo de passos apresentado por ele foi o mais baixo dos três algoritmos. O algoritmo conseguiu fazer uso da transitividade do grafo e alcançar o preenchimento

Quadro 7 – Resultados de Número de Arestas - SnakeChoice

TOTAL DE VÉRTICES	MÍNIMO DE PASSOS	MÉDIA DE PASSOS	MÁXIMO DE PASSOS
15 Vértices	10	26	67
23 Vértices	10	20	46
30 Vértices	10	18	46

de 50 arestas em 10 comparações no melhor caso, encontrando uma média de ao menos 5 arestas por passo nos casos rápidos.

O número médio de passos dados foi mais eficiente do que o *randomChoice*, mas menos do que o *naiveChoice*. Entretanto, essa média se mostrou rapidamente tendendo para baixo conforme o grafo aumenta. Essa tendência, assim como no algoritmo anterior, se explica pela política de escolha de vértices do *snakeChoice*. Como o algoritmo sempre mantém um dos vértices que foi comparado anteriormente, ele sempre consegue fazer uso da transitividade, diferente dos outros algoritmos. Além disso, como o novo vértice que substitui o descartado é escolhido ao acaso, a probabilidade de encontrar um par que não propõe uma aresta que já está presente no grafo é inversamente proporcional ao número de nós presentes no grafo total. Em outras palavras, quanto mais nós existem no grafo, maior a chance de encontrar uma aresta que não foi conferida ainda, o que acelera a execução.

Diferentemente do *randomChoice*, o *snakeChoice* garantidamente terá sempre um único grupo de vértices comparados entre si, porém não necessariamente conexo, devido à sua regra de sempre manter um dos vértices da comparação anterior. Assim, ele apresenta um contraponto ao modo de execução do aplicativo, como detalhado na seção anterior. Como o algoritmo sempre irá gerar um único grupo comparativo, os vértices que serão usados para abordar o usuário a respeito dos gostos de seus amigos poderão formar uma lista ordenada, o que leva a uma experiência de usuário mais positiva.

3.3 RESULTADOS - PREENCHIMENTO POR ESFORÇO

Todos os resultados obtidos até agora levaram em conta a condição de parada do algoritmo ser alcançar um certo número de arestas no grafo. Inicialmente, estávamos interessados em um preenchimento completo do grafo, seguido por um preenchimento parcial. Essas abordagens têm em comum o fato que desconsideram o tempo necessário para atingir a condição de parada. Em outras palavras, era irrelevante ao algoritmo se ele levava uma única iteração muito lenta ou um grande número de iterações curtas, o objetivo alcançado era o mesmo.

Embora esse modo de operar seja válido e até mesmo esperado de um computador executando o algoritmo, não necessariamente é o caso quando é um usuário que o está executando. Uma pessoa está sujeita a enfado caso o processamento leve muito tempo.

Assim, podemos considerar algoritmos que tenham como condição de parada uma certa quantidade de esforço feito pelo usuário. Após um certo número arbitrário de operações que o usuário tenha feito, o algoritmo para, e desejamos saber qual método gera a maior quantidade de dados e dados de melhor qualidade.

3.3.1 Par a par

Dentro do contexto de parar após um certo número de operações, devemos considerar quais operações serão feitas, de modo a maximizar o número de arestas que desejamos. Assim, comparamos a organização par a par de vértices com a organização de subgrupos de 5 vértices cada. Para tal, precisamos pesar o esforço gasto a cada comparação contra o número de arestas novas potenciais.

No caso par a par, a organização é trivial; uma operação é suficiente para decidir entre **ab** e **ba**. Também trivialmente descobrimos uma única aresta a partir dessa comparação, desconsiderando outras que poderiam ser geradas através da transitividade do resto do grafo. Assim, para uma operação, ou seja, 1 esforço, obtemos 1 aresta.

No caso de subgrupos, desejamos ordená-los de forma a gerar um caminho do vértice mais preferido até o vértice mais preferido, refletindo assim as preferências do usuário com o grafo. Esse processo pode ser feito a partir de um algoritmo de ordenação. *Sorting*, em média, leva $nlog(n)$ operações. Como temos 5 vértices, cada iteração custa aproximadamente 8 operações, ou 8 esforço. Além disso, o número de arestas obtidas a partir dessa ordenação é dado por $n^2 = 5^2 = 25$ arestas. Logo, para um custo de 8 esforço, obtemos potencialmente 25 arestas inéditas.

Quadro 8 – Resultados de Esforço - RandomChoice (15 Vértices)

ESFORÇO GASTO	ARESTAS ENCONTRADAS	OBJETOS ENCONTRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	89	14	9	15
100 Passos	100	14	13	15
200 Passos	103	15	14	15

Foram testadas duas metodologias de escolha de vértices para comparação, as mesmas já detalhadas em seções anteriores. O quadro acima exibe os resultados da política completamente aleatória, em que os pares vértices são escolhidos completamente ao acaso.

O algoritmo não considera a presença ou ausência de uma aresta entre os vértices escolhidos. Durante as execuções por número de arestas, isso não fazia diferença, uma vez que a escolha de um par que já tivesse ordenação era simplesmente uma iteração a mais, que não contribuía para o progresso da condição de parada. Entretanto, para a condição de esforço, é possível que a redundância cause desperdício de iterações, sem adicionar arestas novas.

Por sorte, esse fato só aparenta ser um problema se o número de vértices no grafo for muito pequeno. O número de arestas se comportou de forma razoavelmente linear conforme o esforço permitido foi aumentado, tendendo ao número de arestas presentes no grafo total.

Quadro 9 – Resultados de Esforço - SnakeChoice (15 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	79	14	11	15
100 Passos	97	14	14	15
200 Passos	103	15	14	15

A metodologia do algoritmo SnakeChoice, como antes, é de manter sempre um dos vértices de todas as comparações após a primeira, com o intuito de maximizar o grau dos vértices alcançados.

Da mesma forma que a escolha aleatória, o algoritmo pareceu se comportar de forma geralmente linear com relação aos dados que foram colhidos, sendo que maior esforço permitido levou a mais arestas.

Observando e comparando os quadros de resultados da execução dos algoritmos, fica aparente uma diferença entre as políticas de escolha de vértices. A escolha completamente aleatória cobre mais pares de vértices, mas o grafo resultante possui mais subgrupos de tamanho menor, enquanto que a escolha curada resultou em subgrupos médios de tamanho maior, ao custo de menos arestas presentes no grafo em geral.

Essa diferença indica que as políticas de escolha são mais adequadas dependendo do que desejamos obter. Caso um grafo mais cheio, mas potencialmente subdividido em grupos desconexos entre si, seja o objetivo, então é melhor realizar a escolha aleatória. Se, por outro lado, quisermos um grafo com menos informações genéricas, mas uma densidade maior das informações presentes, a escolha curada se mostra mais eficaz. Esses resultados se repetem nos testes com o grafo de 30 vértices, e delineiam ainda mais fortemente a diferença entre os dois métodos.

Quadro 10 – Resultados de Esforço - RandomChoice (30 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	208	29	7	30
100 Passos	352	29	13	30
200 Passos	408	29	20	30

Quadro 11 – Resultados de Esforço - SnakeChoice (30 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	177	24	12	30
100 Passos	308	29	16	30
200 Passos	393	29	21	30

3.3.2 Grupos de Vértices

Como discutido anteriormente, testar os vértices par a par exibe um custo de 1 esforço por 1 aresta nova em potencial, e que a testagem em grupos de vértices tem um custo de 8 esforço, mas pode retornar até 25 novas arestas. Assim, estabelecendo um limite de esforço que o usuário pode estar disposto a gastar, fica claro que o algoritmo itera menos vezes quando testamos vértices em grupos.

O lado ruim desse baixo número de iterações é que, caso as escolhas de vértices sejam muito repetitivas, o algoritmo tem menos chances de obter mais informações em iterações futuras. Em outras palavras, o algoritmo fica mais sujeito a casos extremos.

Apesar disso, o baixo número de execuções também serve para beneficiar a experiência do usuário, que responde a uma quantidade menor de perguntas um pouco mais complexas. Assim, o usuário fica menos sujeito a estafa e tem menos chances de abandonar o algoritmo, interrompendo sua execução de forma prematura.

Além disso, ao observar os resultados das execuções do algotimo de grupos, a redução do número de arestas encontradas pelo algoritmo é razoavelmente pequena. Assim, mesmo a desvantagem do uso de grupos não compromete a utilidade do algoritmo.

Quadro 12 – Resultados de Esforço - RandomGroup (15 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	84	14	10	15
100 Passos	100	14	13	15
200 Passos	104	14	14	15

Novamente, começamos a análise da execução observando a política de escolha completamente arbitrária de vértices. A cada iteração, são escolhidos cinco vértices completamente ao acaso, que ordenamos em uma lista do mais preferido ao menos preferido. Essa lista respeita decisões tomadas anteriormente, ou seja, arestas já existentes são mantidas.

Rapidamente, podemos verificar que no grafo de 15 vértices, os resultados são razoavelmente equivalentes àqueles encontrados nas comparações par a par. Como previsto, o número de arestas que o algoritmo foi capaz de encontrar foi levemente menor no caso de grupos, enquanto que o tamanho médio de subgrupos do grafo final foi levemente maior.

Quadro 13 – Resultados de Esforço - SnakeGroup (15 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	68	12	11	15
100 Passos	93	14	13	15
200 Passos	103	14	14	15

O funcionamento da escolha curada de grupos é similar à escolha curada de vértices: uma parte dos vértices que estavam presentes na rodada anterior são mantidos, e o resto são substituídos por novos vértices escolhidos ao acaso. No caso de grupos, são mantidos dois vértices a cada iteração, e três são renovados. Esses dois vértices mantidos não são necessariamente os mesmos a cada rodada.

Os resultados exibidos pela organização curada de grupos também é bastante similar aos resultados par a par. Mais uma vez verificamos uma pequena baixa no número de arestas, como esperado. Também notamos que todos os algoritmos até agora foram capazes de encontrar, pelo menos uma vez, o subgrupo máximo, que inclui todos os vértices do grafo pelo menos uma vez.

Também mais uma vez, o padrão entre a escolha aleatória e a escolha curada se repetiu. A escolha curada encontra menos arestas, e, em troca, consegue um tamanho médio de subgrupo marginalmente maior.

É interessante notar que, conforme o esforço gasto pelo usuário aumenta, as métricas dos dois métodos tendem a se aproximar. Isso porque, tendo acesso a mais iterações, por meio de mais esforço para gastar, o algoritmo tem mais chances de encontrar combinações de vértices que resultem em novas arestas. Para a escolha aleatória, isso significa encontrar arestas que juntem os subgrupos disjuntos que existiam anteriormente, enquanto que para a curada, são vértices que não tinham sido descobertos ainda.

Apesar dessa normalização tornar, pelo menos em teoria, a escolha de uma metodologia ou a outra irrelevante, é necessário levar em conta o caso de uso do algoritmo. Buscamos um processo que não cause enfado ao usuário, ou seja, uma quantidade muito grande de esforço não é ideal.

Quadro 14 – Resultados de Esforço - RandomGroup (30 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	157	21	9	26
100 Passos	216	23	17	30
200 Passos	396	29	19	30

Observando as tabelas de resultados de grupo com 30 vértices, os padrões de execuções anteriores dos algoritmos se mantêm. A escolha completamente aleatória tende a

Quadro 15 – Resultados de Esforço - SnakeGroup (30 Vértices)

ESFORÇO GASTO	ARESTAS ENCON-TRADAS	OBJETOS ENCON-TRADOS	SUBGRUPO MÉDIO	MAIOR SUBGRUPO
50 Passos	108	17	13	23
100 Passos	216	23	17	30
200 Passos	347	28	21	30

encontrar mais arestas, em média, do que a escolha curada, enquanto que o contrário é verdade para o tamanho médio de subgrupos no grafo final. Também é possível notar que a escolha par a par tende a gerar mais arestas para uma dada quantidade de esforço, e a escolha por grupos gera subgrupos maiores, em média.

Tendo esses resultados em mente, fica claro que qual algoritmo usar depende de qual métrica desejamos favorecer. O caso de uso do algoritmo determina como otimizá-lo para obter os resultados buscados com a maior eficiência possível. Nesse caso, levamos em conta um pequeno esforço feito pelo usuário, e, dependendo de se desejarmos mais arestas ou subgrupos maiores, poderemos usar uma metodologia ou outra.

4 O APLICATIVO

O aplicativo implementado é um jogo simples, em que o usuário preenche um grafo de preferências como descrito nos capítulos anteriores. Essas preferências são armazenadas, e outros jogadores deverão tentar adivinhar os gostos de seus amigos. O aplicativo demonstra tanto o preenchimento de grafos utilizando um dos algoritmos propostos, quanto a abstração de um grafo como uma rede de preferências, que eleva um grafo de apenas um conceito matemático para uma ferramenta de uso prático.

4.1 INTERFACE E UTILIZAÇÃO

Ao inicializar o aplicativo, o usuário visualiza uma tela de login. Caso já tenha uma conta, ele poderá usar as credenciais para acessá-la. Do contrário, a tela também possui uma opção de criação de conta.

Para criar uma conta nova, o usuário escolhe um nome para a conta, que não pode ser um nome de usuário que já exista, assim como uma senha e um e-mail associado. Caso o nome de usuário ou o e-mail escolhido já tenham sido utilizados, o aplicativo exibe uma mensagem de erro indicando tal para o usuário.

Figura 5 – Tela de login

Após realizar o login, o usuário é levado à tela inicial do jogo. Aqui, são apresentadas os dois modos de jogo, assim como uma tela secundária de opções para operações como alteração de senha e de nome de usuário, entre outras.

A opção "Criação de Perfil"deverá ser acessada primeiro. Essa opção apresenta os tópicos ao usuário e pede que ele escolha um deles para preencher suas preferências. Essas

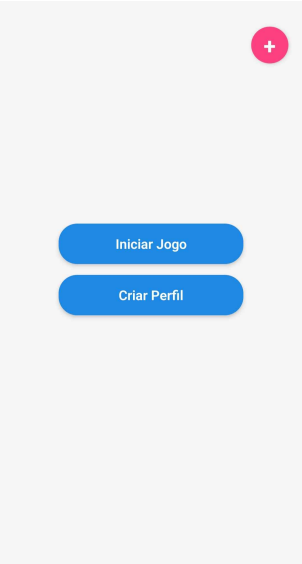

Figura 6 – Tela de escolha de modo de jogo

preferências serão, no futuro, utilizadas pelo *app* para que outros jogadores tentem adivinhar os gostos do usuário. Existem doze categorias para preencher, cada uma contendo vinte exemplos. Não é necessário preencher todas elas, mas fazê-lo melhora a jogabilidade para outros jogadores.

Ao escolher uma categoria, cinco exemplos aleatórios são mostrados ao usuário, e fica a cargo dele ordená-los como achar melhor. Isso é feito clicando nos exemplos em ordem, do menos preferido para o mais preferido. Exemplos já selecionados são passados para uma lista na tela. Para alterar a ordenação, o botão "Limpar Lista"reinicia o processo e exibe novamente quaisquer exemplos que já tivessem sido escolhidos.

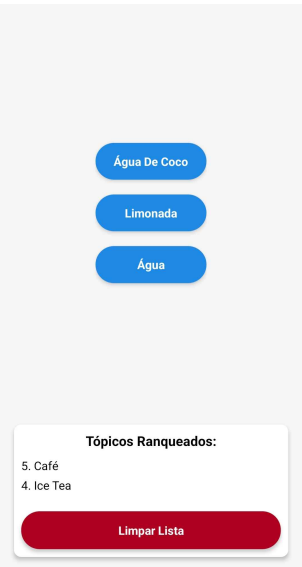

Figura 7 – Tela de escolha de preferências (criação de perfil)

Quando o usuário estiver satisfeito, ele propõe a ordenação para o aplicativo, que, por

sua vez, compara as relações com as respostas dadas anteriormente pelo usuário. Caso haja conflitos, como por exemplo afirmar inicialmente que prefere Coca-cola a vinho, e depois responder o contrário, o aplicativo exibe uma mensagem de erro e pede para que o usuário reordene os exemplos, levando em conta suas opiniões anteriores. Quando todos os conflitos forem resolvidos, ou se não houver nenhum, as novas preferências são encaminhadas para o servidor, que as armazena.

Ao terminar de ordenar os exemplos propostos, esse processo se reinicia um certo número de vezes. Dessa forma, simulamos o funcionamento do algoritmo de preenchimento RandomGroup, buscando obter arestas dentro do grafo de preferências do usuário para que possamos usá-las no outro modo de jogo.

No segundo modo de jogo, o usuário digita o nome de usuário de um amigo. Então, o usuário escolhe um tópico que tenha sido preenchido por esse amigo, para tentar adivinhar seus gostos.

Quando o jogo começa, o usuário é apresentado um par de exemplos dentro do tópico escolhido. Esse par de exemplos é um dos que o amigo preencheu, em sua instância do primeiro modo de jogo, escolhido ao acaso. O usuário deverá decidir qual dos exemplos seu amigo prefere, clicando na opção correta. Caso acerte, ele ganha um ponto e recebe um indicativo visual do acerto. Do contrário, não ganha nenhum ponto, e o indicativo agora mostra o erro.

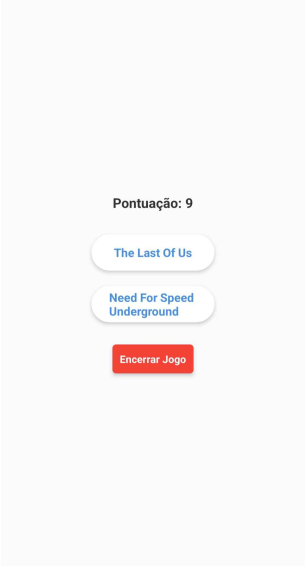

Figura 8 – Tela de jogo de adivinhação de preferências

Esse processo se repete indefinidamente, podendo ser interrompido pelo usuário a qualquer momento, com a pontuação sendo calculada como número de acertos sobre número de tentativas, e exibida como um percentual. O percentual apresenta um ponto de interrogação (?) ao lado, para indicar que a pontuação não é definitiva, até que o usuário tenha respondido um número mínimo de perguntas, inicialmente 10.

4.2 FUNCIONAMENTO DO ALGORITMO

O aplicativo foi implementado em *React Native*, uma linguagem de programação baseada em *JavaScript*, que é muito utilizada para criação de aplicativos. Ela faz a ponte entre as linguagens proprietárias utilizadas por diferentes plataformas, permitindo que apenas um código seja escrito, em vez de ser necessária uma versão para dispositivos *Android* e uma para *iOS*.

Apesar da facilidade oferecida pelo React, tivemos dificuldades inicialmente, por nunca termos desenvolvido um aplicativo nessa escala. O grande número de dependências necessárias, assim como a documentação nem sempre clara, foram obstáculos durante a fase inicial de criação do projeto. O aplicativo também faz uso de uma interface chamada *Navigator*, que gerencia a transição entre *views* (telas), com cujo *setup* não tínhamos experiência.

Também tivemos problemas com a integração com uma base de dados externa ao aplicativo, bem como a implementação dessa base. Em particular, tivemos dificuldades com o processo de autenticação de um dado usuário.

A base de dados (servidor) foi criada em *MongoDB*. O fato de essa ferramenta ser não relacional foi atraente num primeiro momento, uma vez que, dessa forma, a implementação inicial poderia ser mais simples. Assim como no caso do React, também nunca tínhamos tido necessidade de uma base de dados nesse escopo, então a facilidade aparente foi um fator que contribuiu para a escolha.

Um empecilho encontrado foi a necessidade de hospedar o *backend* inteiramente de forma externa, e acessá-lo apenas através de APIs RESTful. Como não tínhamos realizado um projeto deste escopo anteriormente, tivemos que aprender como esse processo é feito. Em particular, foi importante determinar quais são os módulos necessários para a integração do aplicativo com o MongoDB, e o modo correto de instalar e chamar esses módulos.

5 CONCLUSÃO

O preenchimento de um grafo vazio é um problema que rapidamente se torna intratável manualmente, conforme mais vértices são adicionados. Isso é duplamente verdade quando o grafo final precisa, necessariamente, atender a restrições com relação a como as arestas podem ser posicionadas dentro de si. Assim, é necessário o uso de algoritmos que sejam capazes de realizar esse preenchimento. Dessa forma, é importante procurar a eficiência nesses algoritmos, para minimizar o tempo computacional gasto processando o preenchimento com quantidades grandes de vértices envolvidos.

Abordamos, ao longo desse trabalho, três problemas relacionados a esse preenchimento de grafos. Inicialmente, desejamos um preenchimento completo, que incluísse arestas para todos os vértices do grafo. Em seguida, buscamos um preenchimento parcial, que preenchesse apenas um número mínimo de arestas necessário para o funcionamento do aplicativo associado. Finalmente, limitamos o número de vezes que o algoritmo foi permitido iterar, e procuramos maximizar o número de arestas que conseguíssemos encontrar com essa limitação.

As variáveis que determinam se um algoritmo é bom ou ruim são dependentes do objetivo do algoritmo. Como cada um desses problemas requer um processamento levemente diferente, as métricas buscadas por cada um levam a otimizações distintas. Em outras palavras, um algoritmo que funcione melhor para um dos problemas pode ser menos ideal para outro, enquanto uma solução que funcione bem para o outro talvez seja ineficiente, se usada para o terceiro.

Fica claro, também, que esse princípio de otimização para um objetivo ou para outro também se aplica para outros algoritmos com outros propósitos, em termos gerais. É importante considerar os recursos que podem ser utilizados, como tempo de computação, memória, ou até mesmo número de iterações máximas, para garantir os resultados desejados.

5.1 TRABALHOS FUTUROS

Os problemas abordados nesse trabalho são apenas alguns exemplos de tópicos que podem ser explorados dentro do contexto de torneios transitivos e redes de preferências. Em trabalhos futuros, seria possível utilizar os dados providenciados pelo usuário para inferir padrões de preferências, utilizando o estudo de probabilidades para tentar determinar preferências desse usuário mesmo que elas não tenham sido obtidas, direta ou indiretamente. Também podemos pensar em mais métricas que podem ser estudadas durante o uso de um dos algoritmos apresentados, como por exemplo como o preenchimento se comporta com grupos de vértices com mais de 5 nós, ou com políticas de escolhas de vértices diferentes

das utilizadas.

Fora do escopo dos problemas e soluções propostas, também é possível pensar em utilizações diferentes para torneios transitivos e seu preenchimento. Em vez de representar os gostos de um usuário, também poderíamos utilizá-las para simular eventos que fluem uns para outros, ou *rankings* mais generalizados. Torneios transitivos e suas propriedades são ferramentas poderosas, que podem ser aplicadas a vários contextos fora da teoria de grafos pura.

REFERÊNCIAS

BENDER, E. A.; WILLIAMSON, S. G. **Lists, Decisions and Graphs With an Introduction to Probability**. San Diego: University of California, 2010.

DOBROW, R. P. **Introduction to Stochastic Processes with R**. Hoboken: John Wiley & Sons, Inc, 2016.

KARP, R. M. On-line algorithms versus off-line algorithms: How much is it worth to know the future? **International Computer Science Institute**, California, 1992.

KIRK, D. **Contemporary Mathematics**. Houston: OpenStax and Rice University, 2023.

APÊNDICES

APÊNDICE A – CÓDIGOS UTILIZADOS (ALGORITMOS DE PREENCHIMENTO)

Código 1 – A.1: Classes Graph e Node

```
1  import copy
2
3  '''
4  Classe Grafo, para guardar os nós e representar as relações de
       superioridade. Representa um grafo direcionado.
5
6  Cada nó representa um exemplo dentro da categoria relevante (por exemplo
       , comidas: pizza, lasanha, macarrão, sushi, etc.).
7
8  Um nó A é superior a outro nó B se, e somente se, existir caminho de A
       para B. Não é possível existir ciclo.
9  '''
10 class Graph:
11     def __init__(self):
12         #Nós do grafo são guardados em um dicionário para fácil acesso.
       O dicionário é formatado como {string : Node}
13         self.nodeByName = {}
14         #Nós do grafo também são guardados em uma lista para acesso da
       função randomChoice. Guardados pelo nome, em string
15         #Não é acessado por nenhum outro lugar
16         self.nodesList = []
17         #Tamanho do grafo é guardado como um inteiro. Acessado para
       calcular o grau máximo
18         self.graphSize = 0
19
20     #Retorna um dado nó. Retorna o tipo Node, caso o node exista; não
       retorna nada, do contrário.
21     def getNode(self, nodeName):
22         if nodeName not in self.nodeByName:
23             print("\nNode does not exist.")
24         else:
25             return self.nodeByName[nodeName]
26
27     #Adiciona um nó ao grafo, caso já não esteja lá. (Alterações a um nó
       devem ser feitas usando os métodos da classe Nó)
28     def addNode(self, nodeName, incomingNeighbors, outgoingNeighbors):
29         if nodeName in self.nodeByName:
30             print("\nNode already exists.")
31         else:
32             newNode = Node(nodeName, incomingNeighbors,
```

```
    outgoingNeighbors)
33              self.nodeByName.update({newNode.name : newNode})
34              self.nodesList.append(newNode.name)
35              self.graphSize += 1
36
37      #Remove um nó do grafo, caso ainda esteja lá.
38      def removeNode(self, nodeName):
39          if nodeName not in self.nodeByName:
40              print("\nNode does not exist.")
41          else:
42              self.graphSize -= 1
43              self.nodesList.remove(nodeName)
44              self.nodeByName.pop(nodeName)
45              for node in self.nodeByName.values():
46                  if nodeName in node.incomingNeighbors:
47                      node.incomingNeighbors.remove(nodeName)
48                  if nodeName in node.outgoingNeighbors:
49                      node.outgoingNeighbors.remove(nodeName)
50
51      #Exibe um único nó do grafo, caso exista.
52      def showNode(self, nodeName):
53          if nodeName not in self.nodeByName:
54              print("\nNode does not exist.")
55          else:
56              node = self.getNode(nodeName)
57              print("\nNode name: " + node.name)
58              print("\nIncoming neighbors:")
59              for neighborName in node.incomingNeighbors:
60                  print("\n" + neighborName)
61              print("\nOutgoing neighbors:")
62              for neighborName in node.outgoingNeighbors:
63                  print("\n" + neighborName)
64
65      #Exibe o grafo todo em forma de texto.
66      def showGraph(self):
67          if len(self.nodeByName) == 0:
68              print("\nGraph is empty.")
69          else:
70              for node in self.nodeByName:
71                  self.showNode(node)
72
73      #Gera uma lista que contém os graus dos vértices ordenados.
74      def getDegreeList(self):
75          if len(self.nodeByName) == 0:
76              print("\nGraph is empty.")
77          else:
78              ret = []
```

```
79              for vertex in self.nodeByName.values():
80                  vertex.updateDegree()
81                  ret.append(vertex.degree)
82              ret.sort()
83              return ret
84
85      #Cria uma cópia temporária do próprio grafo para adicionar arestas temporárias. Caso a operação de adição de aresta tenha sucesso,
86      #faz a adição de aresta no grafo real. Do contrário, aborta o processo
87      #Caso o processo tenha sucesso, também atualiza o grau dos vértices do grafo
88      #Recebe o nome de dois vértices e cria a aresta entre eles, além de todas as outras que são consequência
89      def addEdge(self, greaterNode, lesserNode):
90          tempGraph = copy.deepcopy(self)
91          if tempGraph.getNode(greaterNode).addNeighbor(tempGraph.getNode(lesserNode), self):
92              self.getNode(greaterNode).addNeighbor(self.getNode(lesserNode), self)
93              for vertex in self.nodeByName.values():
94                  vertex.updateDegree()
95
96      #Reinicializa o grafo para um estado sem arestas
97      def resetGraph(self):
98          for node in self.nodeByName.values():
99              node.resetNode()
100
101 '''
102 Classe Nó, representa os nós dentro do grafo. Cada nó tem um nome e um conjunto (set) de vizinhos de entrada e de saída.
103
104 Os vizinhos de entrada são aqueles que apontam para o nó; os de saída são aqueles para os quais o nó aponta.
105
106 Os vizinhos de entrada são superiores ao nó; o nó é superior aos seus vizinhos de saída.
107
108 Vizinhos de entrada são chamados incomingNeighbors; vizinhos de saída são chamados outgoingNeighbors.
109 '''
110 class Node:
111     def __init__(self, name, incomingNeighbors, outgoingNeighbors):
112         #Nome do nó; guardado como uma string
113         self.name = name
114         #Vizinhos de entrada do nó; guardado como um conjunto de strings
115         self.incomingNeighbors = incomingNeighbors
```

```
116         #Vizinhos de saída do nó; guardado como um conjunto de strings
117         self.outgoingNeighbors = outgoingNeighbors
118         #Cardinalidade do nó; guardada como um inteiro
119         self.degree = 0
120 
121     #Adiciona um novo vizinho de saída ao nó. Ao fazer isso:
122     #Cada vizinho de entrada do nó atual e o próprio nó atual passam a
        ser vizinhos de entrada do novo nó; ciclo aborta o processo.
123     #Se o novo nó ainda tiver vizinhos de saída, chama a função por
        recorrência para os vizinhos de saída do novo nó.
124     #Notavelmente, implementado desse jeito, todos os nós sempre têm um
        caminho de tamanho 1 entre si e seus vizinhos de baixo. Isso
125     #torna óbvio que, se A>B>C, então A>C. A>C fica explicitado.
126     #Recebe um Node a ser adicionado e o Graph que contém os nós;
        retorna False, caso o processo tenha sido abortado; retorna True, do
        contrário.
127     def addNeighbor(self, newNeighbor, currentGraph):
128         #Verifica que não encontrou loop
129         if (self.name in newNeighbor.outgoingNeighbors) or (newNeighbor.
        name in self.incomingNeighbors):
130             #print("\nLoop detected. Aborting new neighbor.")
131             return False
132         #Adiciona si mesmo ao novo vizinho
133         newNeighbor.incomingNeighbors.add(self.name)
134         self.outgoingNeighbors.add(newNeighbor.name)
135         #Adiciona todos os vizinhos acima de si ao novo vizinho
136         for neighbor in self.incomingNeighbors:
137             upperNeighbor = currentGraph.getNode(neighbor)
138             upperNeighbor.outgoingNeighbors.add(newNeighbor.name)
139             newNeighbor.incomingNeighbors.add(upperNeighbor.name)
140         #Caso o novo vizinho não tenha filhos, acaba aqui
141         if (len(newNeighbor.outgoingNeighbors) == 0):
142             return True
143         #Recorre até o final da árvore
144         for neighbor in newNeighbor.outgoingNeighbors:
145             isWorking = self.addNeighbor(currentGraph.getNode(neighbor),
        currentGraph)
146             if not isWorking:
147                 return False
148         return True
149 
150     #Mostra o grau do vértice
151     def showDegree(self):
152         print("\nCardinalidade do vértice " + self.name + ":")
153         print(self.degree)
154 
155     #Atualiza o grau do vértice
```

```
156     def updateDegree(self):
157         self.degree = len(self.incomingNeighbors) + len(self.
        outgoingNeighbors)
158 
159     #Reinicializa o nó para um nó vazio
160     def resetNode(self):
161         self.incomingNeighbors = set()
162         self.outgoingNeighbors = set()
163         self.updateDegree()
```

Graph é a classe que representa o grafo em si. Ela contém os nós do grafo, representados por Nodes, em um dicionário para acesso rápido de cada nó, a partir do nome do nó. Esse é o dicionário *nodeByName*. Os nós também são armazenados em uma lista, chamada *nodesList*, que pode ser iterada pelos algoritmos de preenchimento. Finalmente, *graphSize* é um inteiro armazena o tamanho do grafo, um número usado para calcular o grau máximo de um nó qualquer.

Algumas funções de Graph são razoavelmente padrão para uma estrutura de dados complexa. Os métodos *getNode*, *addNode*, e *removeNode* permitem operações básicas de acesso, inclusão, e remoção de elementos do grafo. O método *showNode* mostra um nó requisitado em forma de texto, para propósitos de monitoramento e exibição de resultados. De forma mais genérica, *showGraph* mostra todos os nós do grafo em sequência, iterando por cada um e os exibindo individualmente.

Para propósitos de trabalho com grafos, também foram necessárias funções mais especializadas. Uma forma de determinar a completude do grafo é verificar se todos os nós têm o maior grau possível, e, assim, *getDegreeList* retorna uma lista iterável que permite ao código conferir rapidamente se todos os nós do grafo atendem a essa condição. O maior grau possível, em um grafo completo e sem repetição desnecessária de arestas, é n - 1, sendo n o tamanho do grafo.

O método *resetGraph* remove todas as arestas do grafo, mas mantém os nós inseridos. Isso retorna o grafo para um estado inicializado, e o permite ser preenchido novamente. Esse método é crucial para os testes de repetição, em que cada algoritmo é testado um número muito grande de vezes em sequência para determinar seu tempo médio de execução.

Finalmente, o método *addEdge* complementa o método *addNeighbor*, de Node, detalhado abaixo. Esse é o método que efetivamente realiza o preenchimento do grafo, recebendo um par de vértices e tentando fazer a adição da aresta entre eles, além de todas as arestas adicionais que essa relação geraria. Caso seja uma aresta válida, ou seja, não quebra nenhum dos pré-requisitos do grafo, a aresta é adicionada. Entretanto, caso essa aresta fosse causar alguma incongruência, como por exemplo um *loop*, o processo é abortado.

Em complemento a Graph, que representa um grafo, Node representa um nó que será

guardado no grafo. Essa estrutura possui os atributos *name*, o próprio nome do nó, *incomingNeighbors* e *outgoingNeighbors*, conjuntos pareados que representam os vizinhos de entrada e saída do nó, e *degree*, um inteiro que armazena o grau do nó, ou seja, o número de arestas conectadas a ele.

Como uma classe complementar a Graph, sem intenções de ser utilizada por si, Node não apresenta os mesmos métodos de exibição e acesso de dados. A única exceção é *showDegree*, que exibe o grau do nó para propósitos de verificação.

Esse mesmo grau precisa ser atualizado a cada vez que o nó recebe um ou mais novos vizinhos, e para isso serve a função *updateDegree*.

Durante os testes de repetição, em que o grafo é preenchido por um mesmo algoritmo repetidamente para verificar o tempo médio de execução desse algoritmo, o grafo precisa ser reinicializado. Isso é feito através da função *resetGraph*, de Graph. Essa função chama, para cada nó do grafo, a função *resetNode* daquele nó, que remove todas as arestas associadas ao nó e atualiza seu grau.

O método principal da classe Node é *addNeighbor*. Esse método recebe um novo vizinho e o grafo em que os dois nós se encontram, e adiciona uma aresta apontando de si para esse novo vizinho. Caso essa adição necessite mais arestas sendo adicionadas, por transitividade, *addNeighbor* também faz essas adições. O método também verifica recursivamente se as arestas que está adicionando são válidas: se não forem, por exemplo ao tentar fazer um nó apontar para seu vizinho de entrada, ou ao criar um *loop*, o processo é abortado.

Código 2 – A.2: Algoritmo NaiveChoice

```
#Pergunta para cada vértice a relação com cada outro vértice
def naiveChoice(thisGraph):
    nIterations = 0
    #Para cada nó do grafo
    for node in thisGraph.nodesList:
        #Pergunta para cada outro nó do grafo a relação entre os dois
        for neighbor in thisGraph.nodesList:
            #A cada iteração, aumenta em 1 o número de iterações
            nIterations += 1
            #Exceto ele próprio
            if node == neighbor:
                continue
            #Moeda honesta é usada para simular a escolha entre um ou
    outro
            if random.randint(0, 1):
                thisGraph.addEdge(node, neighbor)
            else:
                thisGraph.addEdge(neighbor, node)

    #Ao final, retorna quantos passos deu
```

```
20      return nIterations
```

Código 3 – A.3: Algoritmo RandomChoice

```
1  #Sempre escolhe dois vértices completamente aleatórios a cada rodada
2  #Caso a relação entre os vértices já exista, simplesmente segue em
       frente
3  def randomChoice(thisGraph):
4      nIterations = 0
5      isDone = False
6      while not isDone:
7          #Escolhe dois nós aleatórios; garantidamente são nós distintos
8          nodes = copy.deepcopy(thisGraph.nodesList)
9          firstNode = random.choice(nodes)
10         nodes.remove(firstNode)
11         secondNode = random.choice(nodes)
12
13         #Como os dois nós escolhidos são aleatórios, também podemos
       presumir que
14         #a ordenação deles é exatamente essa
15         thisGraph.addEdge(firstNode, secondNode)
16
17         #A cada iteração, aumenta em 1 o número de iterações
18         nIterations += 1
19
20         #Ao terminar a iteração atual, checa a completude do grafo
21         isDone = checkCompleteness(thisGraph)
22
23     #Ao final, retorna quantos passos deu
24     return nIterations
```

Código 4 – A.4: Algoritmo SnakeChoice

```
1  #Pega dois vértices arbitrários e pergunta a relação entre eles; mantém
       o que foi escolhido em mãos
2  #e pega outro arbitrário para substituir o que não foi escolhido
3  #Notavelmente, um usuário faria a escolha de A ou B, mas, para propó
       sitos práticos, a simulação disso
4  #é feita por aleatoriedade
5  def snakeChoice(thisGraph):
6      nIterations = 0
7      isDone = False
8      #Escolhe dois nós aleatórios; garantidamente são nós distintos
9      nodes = copy.deepcopy(thisGraph.nodesList)
10     firstNode = random.choice(nodes)
11     secondNode = random.choice(nodes)
12     while firstNode == secondNode:
13         secondNode = random.choice(nodes)
```

```
14
15      while not isDone:
16          #Joga uma moeda honesta para escolher um dos nós para ser o nó
        maior
17          if random.randint(0, 1):
18              #Se for o primeiro, joga o segundo fora e itera
19              thisGraph.addEdge(firstNode, secondNode)
20              secondNode = random.choice(nodes)
21              while firstNode == secondNode:
22                  secondNode = random.choice(nodes)
23          else:
24              #Se for o segundo, joga o primeiro fora e itera
25              thisGraph.addEdge(secondNode, firstNode)
26              firstNode = random.choice(nodes)
27              while firstNode == secondNode:
28                  firstNode = random.choice(nodes)
29
30          #A cada iteração, aumenta em 1 o número de iterações
31          nIterations += 1
32
33          #Ao terminar a iteração atual, checa a completude do grafo
34          isDone = checkCompleteness(thisGraph)
35
36      #Ao final, retorna quantos passos deu
37      return nIterations
```

Código 5 – A.5: Algoritmo ExtraNaiveChoice

```
1 #NaiveChoice, mas conta se a iteração terminou antes do algoritmo ou não
2 def extraNaiveChoice(thisGraph):
3      nIterations = 0
4      hasOvershot = False
5      #Para cada nó do grafo
6      for node in thisGraph.nodesList:
7          #Pergunta para cada outro nó do grafo a relação entre os dois
8          for neighbor in thisGraph.nodesList:
9              #Se terminou, mas não parou, então passou do prazo
10             if checkCompleteness(thisGraph):
11                 hasOvershot = True
12             #A cada iteração, aumenta em 1 o número de iterações
13             nIterations += 1
14             #Exceto ele próprio
15             if node == neighbor:
16                 continue
17             #Moeda honesta é usada para simular a escolha entre um ou
       outro
18             if random.randint(0, 1):
19                 thisGraph.addEdge(node, neighbor)
```

```
20              else:
21                  thisGraph.addEdge(neighbor, node)
22
23      #Ao final, retorna quantos passos deu e se completou o grafo antes
        de terminar
24      return nIterations, hasOvershot
```

Código 6 – A.6: Função CheckCompleteness

```
1 #Função auxiliar de checagem de completude do grafo
2 #Quando o grafo é completo, todas os graus são iguais a N - 1; se um for
      diferente, não está completo
3 def checkCompleteness(candidateGraph):
4     degrees = candidateGraph.getDegreeList()
5     highestDegree = candidateGraph.graphSize - 1
6     for degree in degrees:
7         if degree != highestDegree:
8             return False
9     return True
```

Código 7 – A.7: Função averageRunsGenerator

```
1 #Executa o algoritmo escolhido um número grande de vezes para normalizar
      o número médio de
2 #iterações que cada um requer. Retorna a média de iterações
3 def generateAverageRuns(thisGraph):
4     #Número de vezes que o algoritmo é executado em sequência
5     nIterations = 100000
6     #Armazena os resultados para permitir o cálculo do tempo médio de
      execução
7     iterationsList = []
8     #Guarda o tempo mínimo de execução do algoritmo
9     lowIterations = 1000000
10    #Guarda o tempo máximo de execução do algoritmo
11    highIterations = -1
12    #Guarda o número de vezes que o algoritmo foi na média ou mais rá
      pido
13    avgOrBetter = 0.0
14    #Guarda o número de vezes que o naiveChoice passou do prazo
15    totalOvershoots = 0.0
16    #Guarda se o naiveChoice passou do prazo
17    currentOvershoot = False
18
19    #Executa o algoritmo escolhido nIterations vezes
20    #print("Algoritmo escolhido: NaiveChoice")
21    print("Algoritmo escolhido: RandomChoice")
22    #print("Algoritmo escolhido: SnakeChoice")
23    #print("Algoritmo escolhido: extraNaiveChoice")
```

```
24      print("Número de nós:")
25      print(thisGraph.graphSize)
26      for i in range(nIterations):
27          #Executar somente um de cada vez; comentar os outros
28          #currentNumber = naiveChoice(thisGraph)
29          currentNumber = randomChoice(thisGraph)
30          #currentNumber = snakeChoice(thisGraph)
31          #currentNumber, currentOvershoot = extraNaiveChoice(thisGraph)
32
33          #Após calcular o número de iterações, reseta o grafo e anota
        quantas foram as iterações
34          thisGraph.resetGraph()
35          iterationsList.append(currentNumber)
36          if currentNumber <= lowIterations:
37              lowIterations = currentNumber
38          if currentNumber >= highIterations:
39              highIterations = currentNumber
40          if currentOvershoot:
41              totalOvershoots += 1
42          if i % (nIterations / 10) == 0:
43              print("\nWorking...")
44
45      #Calcula a média de iterações do algoritmo
46      iterationsAverage = sum(iterationsList) / len(iterationsList)
47      iterationsAverage = math.floor(iterationsAverage)
48      #Calcula quantas vezes o algoritmo atingiu a média ou melhor
49      for current in iterationsList:
50          if current <= iterationsAverage:
51              avgOrBetter += 1
52      avgOrBetterResults = avgOrBetter / nIterations
53      #Calcula quantas vezes o naiveChoice passou do prazo
54      if totalOvershoots >= 1:
55          totalOvershoots = totalOvershoots / nIterations
56
57      return iterationsAverage, lowIterations, highIterations,
        avgOrBetterResults, totalOvershoots
58
59  averageIterations, lowIterations, highIterations, avgOrBetterTime,
        naiveOvershoots = generateAverageRuns(newGraph)
60  print("\nMédia de iterações do algoritmo escolhido: ")
61  print(averageIterations)
62  print("\nTempo baixo de execução:")
63  print(lowIterations)
64  print("\nTempo alto de execução:")
65  print(highIterations)
66  print("\nProporção de vezes que foi na média ou mais rápido:")
67  print(avgOrBetterTime)
```

```
68 if naiveOvershoots >= 1:
69     print("\nProporção de vezes que naiveChoice completou o grafo e
       continuou executando:")
70     print(naiveOvershoots)
```