PUCRS – Escola Politécnica
Ciência de Dados e Inteligência Artificial – 2022/2

Trabalho 2: Algoritmos e Estruturas de Dados II

Thiago de Almeida Macedo – 21104690

Porto Alegre - RS

Introdução

Este relatório tem como objetivo descrever o trabalho 2 da disciplina de Algoritmos e Estruturas de Dados II, explicando o problema que foi abordado e a solução elaborada, mostrando detalhes de sua implementação em Python e demostrando sua eficácia através de casos de testes.

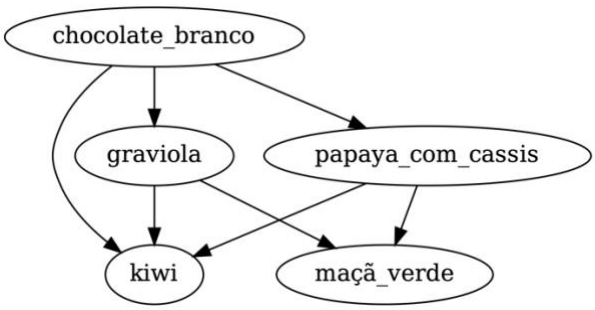
Enunciado do Trabalho

O trabalho se chamava "A Sorveteria dos Horrores" e consistia em uma sorveteria que, por causa do dono e de suas regras preocupantes, limitou consideravelmente a quantidade de combinações de sabores de sorvete possíveis. As regras estabelecidas são as seguintes:

- A sorveteria tem copinhos para 2 e 3 bolas de sorvete.
- O dono nunca coloca um sabor forte (ex.: chocolate mega-ultra-power-100%) em cima de um sabor suave (ex.: iogurte), pois argumenta que não é possível sentir o sabor mais fraco.
- O dono tem uma lista de quem é mais forte do que quem para sua orientação.
- Ele não aceita pedidos com sabor repetido.

Devido suas regras controversas, a família do dono encomendou a criação de um programa que, a partir da lista da ordem dos sorvetes, contasse quantos copinhos de sorvete distintos poderiam ser feitos. Ou seja, quantos combinações de 2 e de 3 sabores de sorvete poderiam ser vendidos pela sorveteria. A lista de sabores elaborada pelo dono e seu desenho correspondente estão organizados no seguinte formato:

```
chocolate_branco -> papaya_com_cassis
chocolate_branco -> graviola
chocolate_branco -> kiwi
papaya_com_cassis -> maçã_verde
papaya_com_cassis -> kiwi
graviola -> maçã-verde
graviola -> kiwi
```



A Solução Desenvolvida

Para resolver o problema proposto, utilizei um grafo dirigido para representar os sabores e suas respectivas ordens na sorveteria. Nesse grafo, cada vértice é um sabor de sorvete e cada aresta liga dois sabores, saindo de um sabor e apontando para um outro que é considerado mais fraco.

Para criar o grafo, o arquivo foi lido e, a partir dele, foi criada uma lista de adjacência, representando os vértices e arestas. Depois que o grafo foi montado, entrou em ação o algoritmo que contava quantas combinações de 2 e 3 sabores eram possíveis de criar.

Para encontrar todas as combinações de 2 sabores, eu desenvolvi um algoritmo que, para cada vértice no grafo, contava quantos vértices eram possíveis de se alcançar. Esse método consistia em utilizar o Breadth First Search (BFS) para checar se havia um caminho entre dois

vértices. Se sim, essa seria uma combinação de dois sabores possível. Ao repetir isso para todos os vértices, o número total de combinações era alcançado.
O algoritmo funcionava da seguinte maneira:

1. Para cada vértice origem no grafo:
    a. Para cada vértice destino no grafo:
        i. Se existe um caminho entre os vértices origem e o destino: **soma um**.
2. Retorna a contagem total.

Para encontrar todas as combinações de 3 sabores, implementei uma versão um pouco mais complicada do que a utilizada para encontrar as de 2 sabores. O algoritmo funciona da mesma forma, mas, ao invés de somar um quando existe um caminho entre a origem e o destino, ele conta a quantidade de vértices que se é possível alcançar a partir do destino. Ou seja, para cada combinação de dois sabores, ele procura quantos vértices são alcançáveis a partir do segundo sabor. Ele funcionava da seguinte maneira:

1. Para cada vértice origem no grafo:
    a. Para cada vértice destino no grafo:
        i. Se existe um caminho entre os vértices origem e destino:
            1. Para cada próximo destino no grafo:
                a. Se existe um caminho entre o destino 1 e o destino 2: **soma um**.

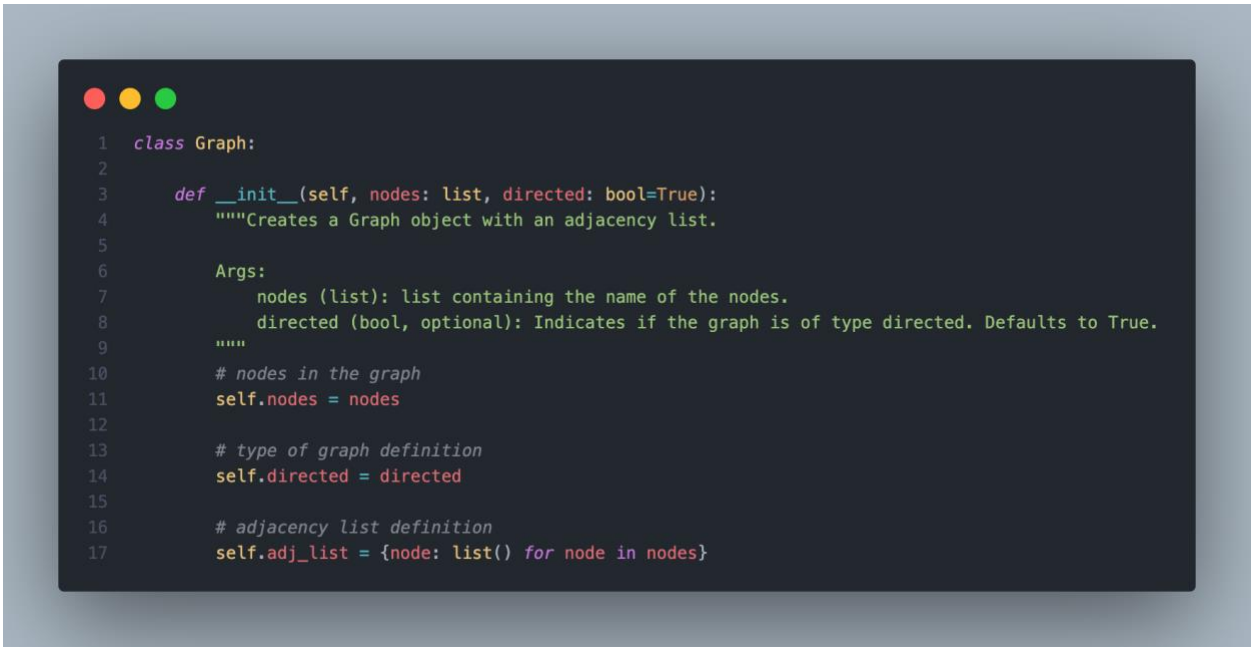
Obs.: Como não é possível montar copinhos com sabores repetidos, os vértices de destino do grafo são todos os vértices do grafo, exceto a origem.

Obs. 2: O algoritmo BFS foi utilizado para checar se existia um caminho entre dois vértices no grafo. O método implementado para sua execução retorna “True”, se existia um caminho e “False” caso contrário.

Além do retornar o número de possíveis combinações de 2 e 3 sabores, o código também escreve em um arquivo todas elas, especificando o nodo de origem, o destino intermediário e o destino final (caso tenha).

Principais Classes e Métodos Utilizados

1. Classe ***Graph***: o objeto grafo foi implementado através de uma lista de adjacência, utilizando-se um dicionário do Python.

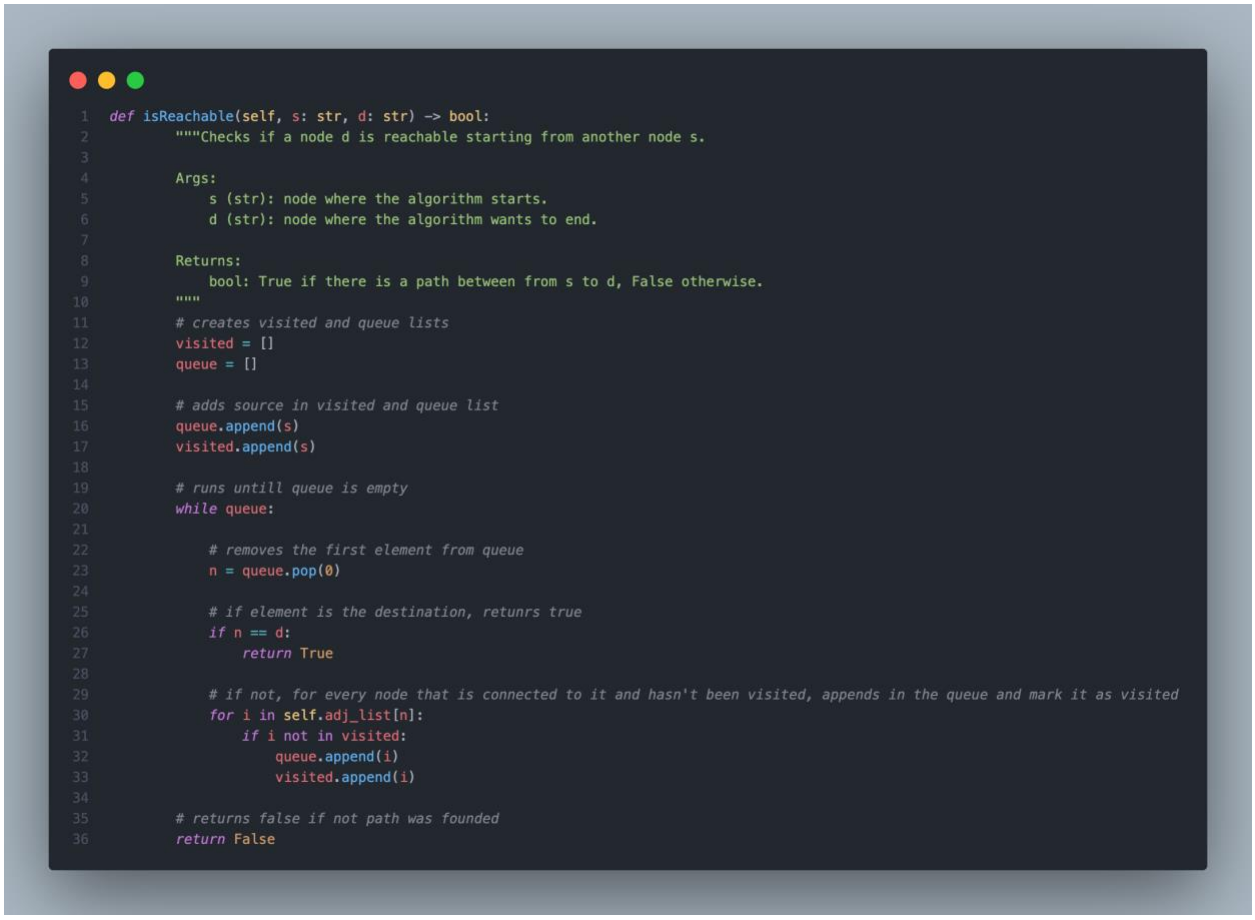


```python
class Graph:

    def __init__(self, nodes: list, directed: bool=True):
        """Creates a Graph object with an adjacency list.

        Args:
            nodes (list): list containing the name of the nodes.
            directed (bool, optional): Indicates if the graph is of type directed. Defaults to True.
        """
        # nodes in the graph
        self.nodes = nodes

        # type of graph definition
        self.directed = directed

        # adjacency list definition
        self.adj_list = {node: list() for node in nodes}
```

2. Método ***add_edge:*** adiciona uma aresta entre dois vértices do grafo. Faz isso colocando o nome do vértice destino na lista de adjacência do vértice de origem.

```python
def add_edge(self, node1: str, node2: str):
    """Connects two nodes with an edge:
        - If graph is directed: node1 points to node2.
        - If graph is not directed: node1 points to node2 and node2 points to node1.


    Args:
        node1 (str): Name of the node1.
        node2 (str): Name of the node2.
    """
    # edge from node1 to node2
    self.adj_list[node1].append(node2)

    # edge from node2 to node1 (only if not directed)
    if not self.directed:
        self.adj_list[node2].append(node1)
```

3. Método ***isReachable:*** Implementação do BFS que retorna “True” se existe um caminho de um vértice origem para um vértice destino.



```python
def isReachable(self, s: str, d: str) -> bool:
        """Checks if a node d is reachable starting from another node s.

        Args:
            s (str): node where the algorithm starts.
            d (str): node where the algorithm wants to end.

        Returns:
            bool: True if there is a path between from s to d, False otherwise.
        """
        # creates visited and queue lists
        visited = []
        queue = []

        # adds source in visited and queue list
        queue.append(s)
        visited.append(s)

        # runs untill queue is empty
        while queue:

            # removes the first element from queue
            n = queue.pop(0)

            # if element is the destination, retunrs true
            if n == d:
                return True

            # if not, for every node that is connected to it and hasn't been visited, appends in the queue and mark it as visited
            for i in self.adj_list[n]:
                if i not in visited:
                    queue.append(i)
                    visited.append(i)

        # returns false if not path was founded
        return False
```

4. Método ***n_two_nodes_edges:*** Retorna a quantidade de combinações de dois sabores.

```python
def n_two_nodes_edges(self) -> int:
    """Calculates the amount of combinations that can be made with two different connected nodes
       in the graph. Uses bfs to check if there's a path between two nodes and sums 1 if it does.


    Returns:
        int: Amount of connections.
    """
    # open output file
    f = open('2FlavorCombinations.txt', 'w')

    # number of edges
    n_conncetions = 0

    # for every source in the graph
    for source in self.adj_list.keys():
        # for every destination in the graph
        for destination in self.adj_list.keys():
            # if destination is not the source
            if destination != source:
                # if the destination in reachable from the source
                if self.isReachable(source, destination):
                    # adds a new connection
                    n_conncetions += 1
                    # prints connection in the output file
                    f.write(f'{source} -> {destination}\n')

    f.close()

    return n_conncetions
```

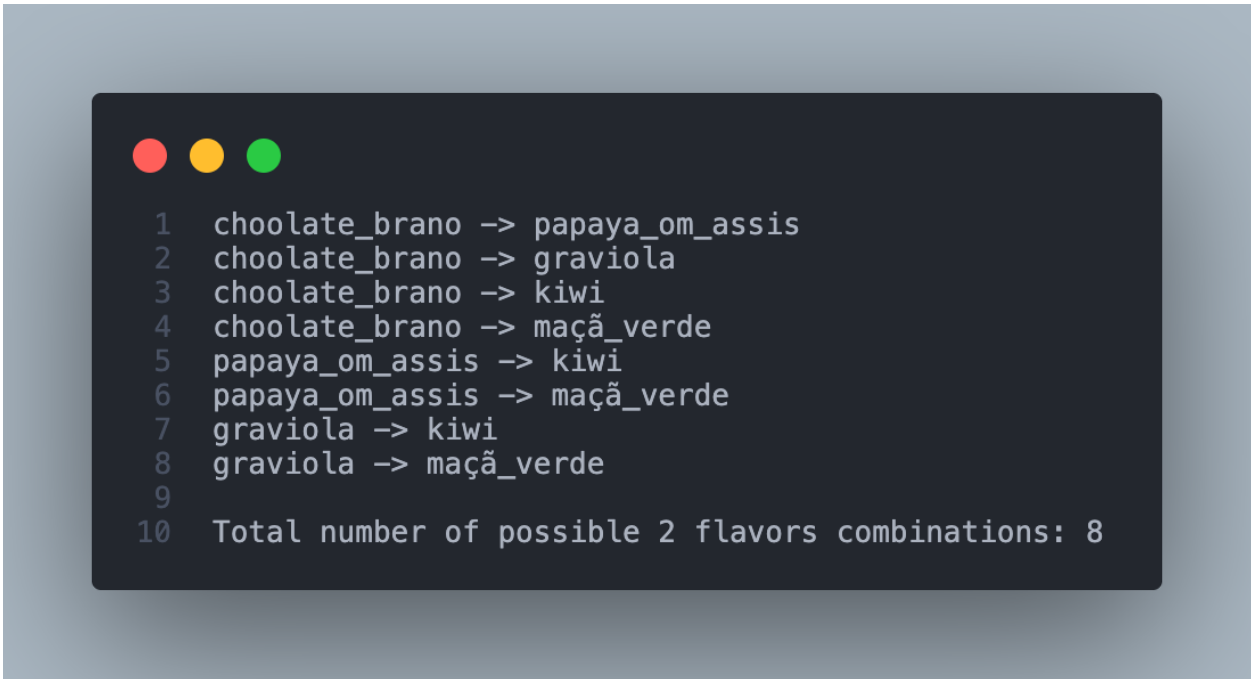
5. Método ***n_three_nodes_edges:*** Retorna a quantidade de combinações de três sabores.

```python
def n_three_nodes_edges(self) -> int:
        """Calculates the amount of combinations that can be made with three connected nodes
           Uses bfs to check if there's a path between two nodes and if it does, checks how many
           nodes can be reached by it. Presents the total number if the end.

        Returns:
            int: amount of combinations.
        """
        # creates new output file
        f = open('3FlavorCombinations.txt', 'w')

        # number of connections
        n_conncetions = 0

        # for each source in the graph
        for source in self.adj_list.keys():
            # for each destination in the graph
            for destination in self.adj_list.keys():
                # if source is not the destination
                if destination != source:
                    # if the destination can be reached from the source
                    if self.isReachable(source, destination):
                        # for each next_destination in the graph
                        for next_destination in self.adj_list.keys():
                            # if next_destination is not the source or the previous destination
                            if next_destination not in [source, destination]:
                                # if the next destination can be reached from the previous destionation
                                if self.isReachable(destination, next_destination):
                                    # writes in the output file
                                    f.write(f'{source} -> {destination} -> {next_destination}\n')
                                    # adds a new connections
                                    n_conncetions += 1

        # write number of combinations and closes the file
        f.write(f'\nTotal number of possible 3 flavors combinations: {n_conncetions}')
        f.close()

        return n_conncetions
```

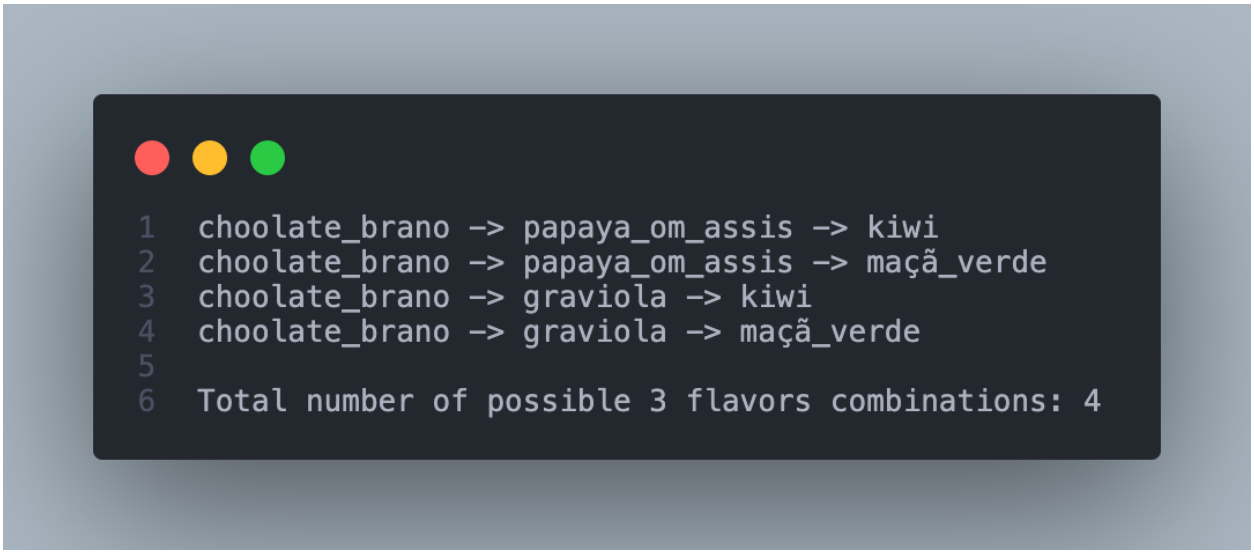
Casos de Teste

Considerando o exemplo dado no enunciado do trabalho, ao utilizar o algoritmo naquela situação, são encontradas 8 combinações possíveis de 2 sabores e 4 combinações possíveis de 3 sabores. As combinações são as seguintes:

2 Sabores:



3 Sabores:



Conclusão

O trabalho 2 da disciplina de Algoritmos e Estrutura de Dados II buscou explorar os conceitos de grafos e seus respectivos algoritmos de caminhamento. A solução também está disponível no GitHub.