O presente relatório provê detalhes sobre dois algoritmos usados para mapear um grafo: busca em profundidade (DFS) e busca em largura (BFS). Enquanto que o primeiro usa uma pilha (*First In Last Out*) para orientar o percurso do grafo, o segundo faz uso de uma fila (*First In First Out*). Além disso, dado que ambos os algoritmos possuem operações comuns, como obter os nós adjacentes a dado vértice assim como sinalizar os vértices visitados, definiu-se uma superclasse abstrata `Searchable<T>`, responsável por implementar esses métodos, em que `T` corresponde à classe armazenada pelo vértice.

```
public abstract class Searchable<T>
```

A fim de obter os vértices adjacentes, implementou-se o método `getAdjacencyUnvisitedVertex`. Inicialmente, o vértice do qual se pretende obter os vértices adjacentes é recuperado a partir do grafo - `getVertexs().indexOf(new Vertex<>(source))`. Logo após, itera-se sobre a matriz de adjacência a fim de encontrar nós adjacentes que ainda não foi visitado. Caso não haja mais nós adjacentes a serem visitados, o método retorna -1.

```
    protected Integer getAdjacencyUnvisitedVertex(T source) {
        Integer index = graph.getVertexs().indexOf(new Vertex<>(source));

        for (int i = 0; i < graph.getAdjacencyMatrix()[index].length; i+
+)
        {
            if (
                graph.getAdjacencyMatrix()[index][i] == 1 &&
                !visited.contains(graph.getVertexs().get(i).getItem())) {
                    return i;
            }
        }

        return -1;
    }
```

Além desse método, a superclasse `Searchable<T>` armazena um conjunto baseado em *hash* com a finalidade de registrar os nós que forem visitados

```
    protected Set<T> visited = new HashSet<>();
```

Por fim, a classe `Searchable<T>` define o método abstrato `search()`, a ser implementado pelas classes `DFS` e `BFS`

```
    public abstract void search();
```

No caso do DFS, aplica-se as seguintes regras

1. Se possível, visite um vértice adjacente não visitado, sinalize-o como visitado e insire-o na pilha.
2. Se não for possível seguir a primeira regra, então, se possível, remova um vértice da pilha
3. Se não for possível seguir a primeira e a segunda regra, finalize o algoritmo.

Considerando a existência de um vértice adjacente não visitado, sendo o respectivo índice obtido a partir do método `getAdjacencyUnvisitedVertex`, a primeira regra pode ser implementada como a seguir

```
    final Vertex<T> adjacencyUnvisitedVertex =
graph.getVertexs().get(adjacencyUnvisitedVertexIndex);
```

```
    visited.add(adjacencyUnvisitedVertex.getItem());

    stack.push(adjacencyUnvisitedVertex.getItem());
```

Caso não exista vértice adjacente, aplica-se a segunda regra

```
    stack.pop();
```

Se não houver mais nós adjacentes nem houver mais vértices a serem removidos da pilha, finaliza-se o algoritmo. Tal regra é monitorada por uma condicional, conforme a seguir

```
    while (!stack.empty())
```

Já para o BFS, as seguintes regras são aplicáveis

1. Visite o próximo vértice adjacente não visitado (se houver) ao vértice corrente, sinalize-o como visitado e insire-o na fila.
2. Se não for possível seguir a primeira regra, remova um vértice da fila e configure-o como o vértice corrente.
3. Se não for possível seguir a segunda regra, finalize o algoritmo.

Considerando a existência de um vértice adjacente não visitado, sendo o respectivo índice obtido a partir do método `getAdjacencyUnvisitedVertex`, a primeira regra pode ser implementada como a seguir, em que `head` corresponde ao vértice corrente. Observe que a sintaxe é parecida com a do DFS, exceto pelo fato de que os vértices adjacentes ao (vértice) corrente são inicialmente percorridos antes de seguir adiante.

```
    while ((adjacencyUnvisitedVertexIndex =
getAdjacencyUnvisitedVertex(head)) != -1) {
        final Vertex<T> adjacencyUnvisitedVertex =
graph.getVertexs().get(adjacencyUnvisitedVertexIndex);

        visited.add(adjacencyUnvisitedVertex.getItem());

        queue.offer(adjacencyUnvisitedVertex.getItem());
    }
```

Uma observação final, válida para ambos os algoritmos, é a necessidade de se definir um vértice usado como ponto de partida, podendo ser o primeiro vértice armazenado pelo grafo. Considerando esse caso, esse vértice é inicialmente marcado como visitado e adicionado à pilha ou à fila, dependendo do algoritmo usado. No exemplo abaixo, este cenário usando o DFS, que faz uso da pilha.

```
    visited.add(graph.getVertexs().get(0).getItem());

    stack.push(graph.getVertexs().get(0).getItem());
```

Referência

LAFORE, Robert. Data Structures & Algorithms in Java. 2. ed. Indianapolis: Sams, 2003.