Matemática Discreta

# GRAFO DIRIGIDO

Perez Molina, Tomás

# Índice

# 1. Consigna

1. Especificar un grafo dirigido y no ponderado.
2. Realizar y probar la implementación de un grafo dirigido y no ponderado con lista de aristas. Calcular el orden de cada operación.
3. Para probar el buen funcionamiento de la clase anterior:
   (i) Hacer un método que permita cargar datos a un grafo (este método debe permita cargar los valores de los nodos y las aristas).
   (ii) Hacer un método que genere un grafo en forma aleatoria.
   (iii) Hacer un método que muestre por pantalla al grafo (puede ser un listado con los valores de los nodos y otro con las aristas o bien el dibujo del grafo).
4. Implementar los tres algoritmos que permiten recorrer un grafo (búsqueda plana, BFS y DFS)
5. Modificando adecuadamente los métodos anteriores, escribir y probar algoritmos que:
   (i) Obtengan la cantidad de vértices fuentes y la cantidad de sumideros.
   (ii) Verificar si es débilmente conexo.
   (iii) Dados dos vértices verificar si existe un camino de longitud 2 entre ambos.
   (iv) Implementar el algoritmo de Warshall.

# 2. Especificación

## 2.1. DirectedGraph

*Description:* Represents a directed non-weighted graph.

**Digraph**: Max Order → Digraph
**Order:** O(1)
**Description:** Creates a new empty digraph with the given max order limit. If no max order is given, the graph has unlimited max order.
**Precondition:** Max Order > 0
**Postcondition:** A Digraph is created.
**Classification:** Constructor

**add_vertex**: Digraph X Data → void
**Order:** O(1)
**Description:** Adds a new vertex to the graph and stores the given data.
**Precondition:**

- Digraph must exist.
- Data must exist.

**Postcondition:** New vertex added to the given graph, storing the given data.
**Classification:** Modifier

**remove_vertex**: Digraph X Key → Data
**Order:** O(n)
**Description:** Finds the vertex referenced by the given key in the digraph and removes it. Returns the data stored in the vertex.
**Precondition:**

- Digraph must exist.
- Key must reference an existing vertex in the digraph.

**Postcondition:** Vertex removed, and its data returned
**Classification:** Modifier

**key_of**: Digraph X Data → Key
**Order:** O(n)
**Description:** Finds the key referencing the vertex storing the given data in the graph.
**Precondition:**

- Digraph must exist.
- The data must be stored in the digraph.

**Postcondition:** Key referencing the vertex storing the given data is returned.
**Classification:** Analyzer

**add_edge**: Digraph X From Key X To Key → void
**Order:** O(n)
**Description:** Adds a new edge connecting the vertices referenced by the given keys. The edge goes from the vertex referenced by From Key to the vertex referenced by To Key.
**Precondition:**

- Digraph must exist.
- From Key and To Key must reference vertices in the digraph.

**Postcondition:** New edge added to the graph, that goes from the vertex referenced by From Key to the vertex referenced by To Key.
**Classification:** Modifier

**remove_edge**: Digraph X From Key X To Key → void
**Order:** O(n)
**Description:** Finds the edge connecting the vertices referenced by From Key and To Key and removes it.
**Precondition:**

- Digraph must exist.
- From Key and To Key must reference a existing vertices in the digraph.
- An edge going from the vertex referenced by From Key to the vertex referenced by To Key must exist.

**Postcondition:** Edge removed
**Classification:** Modifier

**get_vertex**: Digraph X Key → Data
**Order:** O(1)
**Description:** Finds the vertex referenced by Key in the digraph and returns the data stored in it
**Precondition:**

- Digraph must exist.
- Key must reference an existing vertex in the digraph.

**Postcondition:** Return the data stored in the vertex referenced by the given key.
**Classification:** Analyzer

**get_adjacency_list**: Digraph X Key → Key List
**Order:** O(n)
**Description:** Returns the list of all vertices adjacent to the one referenced by the given key in the graph.
**Precondition:**

- Digraph must exist.
- Key must reference an existing vertex in the digraph.

**Postcondition:** Return a list of keys referencing all adjacent vertices to the one referenced by the given key.
**Classification:** Analyzer

**edge_exists**: Digraph X From Key X To Key → Boolean
**Order:** O(n)
**Description:** Verifies whether an edge from the vertex referenced by From Key to the vertex referenced by To Key exists.
**Precondition:**

- Digraph must exist.
- From Key and To Key must reference a existing vertices in the digraph.

**Postcondition:**

- True if the edge exists.
- False if the edge does not exist.

**Classification:** Analyzer

**random_digraph**: Vertex Amount X Edge Amount → Digraph
**Order:** O(n)
**Description:** Creates a random digraph with the given amount of vertices and edges.
**Precondition:**

- Vertex Amount > 0
- Edge Amount > 0

**Postcondition:** A random digraph with the given amount of vertices and edges is created.
**Classification:** Constructor.

## 2.2. Graph Related Algorithms

*Description:* Algorithms related to graph theory

**plain_search**: Graph → Key List
**Order:** O(n)
**Description:** Given a graph returns a list of keys referencing all its vertices in no particular order.
**Precondition:** Graph must exist
**Postcondition:** Return a list of keys referencing all vertices in the graph.
**Classification:** Analyzer

**dfs**: Graph → Key List
**Order:** O($n^2$)
**Description:** Given a graph returns a list of keys referencing all its vertices using Depth First Search, starting from the vertex first inserted in the graph.
**Precondition:** Graph must exist
**Postcondition:** Return a list of keys referencing all vertices in the graph using Depth First Search.
**Classification:** Analyzer

**bfs**: Graph → Key List
**Order:** O($n^2$)
**Description:** Given a graph returns a list of keys referencing all its vertices using Breath First Search, starting from the vertex first inserted in the graph.
**Precondition:** Graph must exist
**Postcondition:** Return a list of keys referencing all vertices in the graph using Breath First Search.
**Classification:** Analyzer

**number_of_sources**: Graph → Number
**Order:** O(n)
**Description:** Given a graph returns the number of source vertices.
**Precondition:** Graph must exist
**Postcondition:** Return the amount of source vertices in the graph
**Classification:** Analyzer

**number_of_sinks**: Graph → Number
**Order:** O(n)
**Description:** Given a graph returns the number of sink vertices.
**Precondition:** Graph must exist
**Postcondition:** Return the amount of sink vertices in the graph
**Classification:** Analyzer

**is_weakly_connected**: Graph → Boolean
**Order:** O($n^2$)
**Description:** Given a graph, checks whether it is weakly connected.
**Precondition:** Graph must exist
**Postcondition:**

- True if the graph is weakly connected.
- False if the graph is not weakly connected.

**Classification:** Analyzer

**path_exists**: Graph x From Key x To Key X Length X Transitive Closure → Boolean
**Order:** $O(n^2)$
**Description:** Checks if a path between the vertices referenced by From Key and To Key exist.

- A length or a transitive closure can be given, not both.
- If a length is given, the path can be no longer than the length.
- If a Transitive Closure is given the path is checked in it, achieving O(1) performance.

**Precondition:**

- Digraph must exist.
- From Key and To Key must reference a existing vertices in the digraph.
- If given, length $> 0$.

**Postcondition:**

- True if a path exists.
- False if a path does not exist.

**Classification:** Analyzer

**warshall**: Graph → Transitive Closure
**Order:** $O(n^3)$
**Description:** Given a graph calculates its transitive closure using Warshall's.
**Precondition:** Graph must exist
**Postcondition:** The transitive closure must have the capability of calculating whether there is a path between two vertices in O(1) time.
**Classification:** Analyzer

**save**: Graph X Name X View X Format $\rightarrow$ void
**Order:** O(n)
**Description:** Given a graph, stores its graphic representation in the given format with the given name. A view of the file is given if View is True.
**Precondition:**

- Graph must exist
- Name must be a valid file name
- If given, view must be True or False (default False)
- If given, format must be a valid file format for the Graphviz program (default 'pdf').

**Postcondition:**

- A file containing DOT code for Graphviz and an image in the given format are created.
- If View is True, the created image is displayed.

**Classification:** Analyzer