Lab4 Documentation

Release 2023

Gabor Ioana

May 22, 2023

CONTENTS

`class graph.Graph`

Represents a directed graph.

`add_edge`(*edge: tuple*)

Adds an edge to the graph. Preconditions: both of its vertices must exist, but the edge must not be present in the current graph.

:param edge:tuple of 2 vertices, the added edge :return: nothing :raises Exception: if one of its vertices doesn't exist, if the edge already exists or if the cost is not integer

`add_vertex`(*vertex: str*)

Adds a vertex to the graph. Preconditions: the vertex must not exist in the current graph.

Parameters
`vertex` – given vertex

Returns

Raises
`Exception` – if the vertex already exists

`classmethod build_graph_from_given_vertices_and_edges`(*vertices*, *edges*)

Builds a graph from a list of vertices and a list of edges with costs.

Parameters

- `vertices` – list of vertices
- `edges` – list of tuples of 2 parameters - (first vertex, second vertex)

Returns
created graph

Raises
`Exception` – if the list of received edges is invalid or if the vertices are not all strings

`classmethod build_random_graph`(*number_of_vertices: int*, *number_of_edges: int*)

Builds a random graph with a specified number of vertices and number of edges. Preconditions: the number of edges should be smaller than or equal to the square of the number of vertices.

Parameters
`number_of_vertices` – integer

:param number_of_edges:integer :return: created graph :raises Exception: if the number of edges is too large for a graph with distinct edges

`classmethod create_copy`(*graph*)

Creates a deepcopy of a graph

Parameters
`graph` –

Returns
created graph

`get_all_edges`()

Gets the list of all the edges in this graph.

Returns
list of tuple of 2 elements: a tuple for the edge (another tuple of 2 vertices) and an integer

`get_all_vertices()`

Gets the list of all the vertices in this graph.

Returns
list of vertices

`get_in_degree_of_vertex(`*vertex: str*`)`

Gets the in-degree of a vertex. The in-degree is the number of vertices that directly reach
this vertex.

Parameters
`vertex` – given vertex

Returns
integer - the in degree

Raises
`Exception` – if the vertex does not exist

`get_inbound_neighbours(`*vertex*`)`

Gets the inbound neighbours of a vertex.

Parameters
`vertex` – given vertex

Returns
list of outbound neighbours

Raises
`Exception` – if the vertex does not exist

`get_isolated_vertices()`

Gets the list of all the isolated vertices

Returns
list of vertices

`get_number_of_vertices()`

Gets the number of vertices in this graph.

Returns
integer

`get_out_degree_of_vertex(`*vertex*`)`

Gets the out-degree of a vertex. The out-degree is the number of vertices that are directly reachable from this vertex.

Parameters
`vertex` – given vertex

Returns
integer - the out degree

Raises
`Exception` – if the vertex does not exist

`get_outbound_neighbours(`*vertex*`)`

Gets the outbound neighbours of a vertex.

Parameters
`vertex` – given vertex

Returns
list of outbound neighbours

Raises
`Exception` – if the vertex does not exist

`is_edge`(*edge: tuple*)

Checks if an edge is part of this graph. Preconditions: the edge must be a tuple of 2 vertices.

Parameters
`edge` – tuple of 2 edges

Returns
boolean

Raises
`Exception` – if the edge is not a tuple of 2 vertices

`is_vertex`(*vertex: str*)

Checks if a vertex is part of this graph.

Parameters
`vertex` – given vertex

Returns
boolean

`remove_edge`(*edge: tuple*)

Removes an edge.

Parameters
`edge` – tuple of 2 vertices

Returns
nothing

Raises
`Exception` – if the edge doesn't exist

`remove_vertex`(*vertex: str*)

Removes a vertex.

Parameters
`vertex` – given vertex

Returns
nothing

Raises
`Exception` – if the vertex doesn't exist

`topological_sort`()

Performs a topological sort of the vertices of the DAG, using predecessor counting

Returns
list of sorted vertices

Raises
`Exception` – if the graph is not a DAG

`class graph_utils.GraphUtils`

Helper methods used for reading and writing graphs to files, in normal or modified format.

`static read_graph_modified_format(`*filename*`)`

Reads a graph in "modified format" from a given filename.

Preconditions: the file must exist, the filename must end with "modified.txt" and the graph should be in the valid format.

The "modified" format must obey the following rules: on the first line of the file, there are two numbers, separated by space: the number of vertices(n) and the number of edges(m). On the second line, there is the list of isolated vertices. On the following m lines, there are three numbers that describe each of the m edges: the starting vertex, the ending vertex and the cost of the edge.

Parameters
`filename` – string

Returns
Graph

Raises
`Exception` – if the graph is invalid, if the file doesn't exist, if the filename doesn't end with

"modified.txt" or if other file-related errors occurred

`static read_graph_normal_format(`*filename*`)`

Reads a graph in "normal format" from a given filename.

Preconditions: the file must exist, the graph must be in the valid format.

The "normal" format must obey the following rules: on the first line of the file, there are 2 integers, separated by space: the number of vertices (n) and the number of edges (m). On the next m lines, there are three numbers that describe each of the m edges: the starting vertex, the ending vertex and the cost of the edge.

Parameters
`filename` – string

Returns
Graph

Raises
`Exception` – if the graph is invalid, if the file doesn't exist or if other file-related errors occured.

`static write_graph_modified_format(`*filename: str*, *graph*`)`

Writes a graph in "modified format" to a given file.

Parameters

- `filename` – string
- `graph` – Graph

Raises
`Exception` – if the filename doesn't end with "modified.txt" or if other file-related errors occurred.

`static write_graph_normal_format(`*filename*, *graph*`)`

Writes a graph in "normal format" to a given file.

Parameters

- `filename` – string

- **graph** – Graph

Raises
`Exception` – if output-related errors occurred

`class scheduling_problem.`**`SchedulingProblem`**

Static methods used for solving the Scheduling problem.

`static` **`read_activities_from_file`**(*filename*)

Reads the activities from the file and builds the corresponding graph, adding 2 additional activities, of cost 0: start and finish

Parameters
`filename` – string

Returns
graph

`static` **`solve`**(*graph:* Graph)

Given a graph of activities, solves the scheduling problem. :param graph: :return: tuple: the total duration of the project, the earliest starting times, the latest starting times, the critical vertices :raises Exception: if the graph is not a DAG

PYTHON MODULE INDEX

INDEX