**Project in Data Processing 236323**

**Winter Semester 2020-2021**

Project Title: Port Graph Representation Library

Project Supervisor: Avi Mendelson

Students: Mario Barbara, Feras Bisharat

**Introduction**

Graphs are a very basic combinatorial object, used in various fields in Computer Science and Mathematics, either for theoretical work or real-life models. For this reason, it is very useful and important to be able to represent graphs as a concrete data structure (as opposed to a theoretical object).

The basic definition of a graph can be extended, specialized, or generalized such that it can represent various theoretical or real-life computational models. Port Graphs are such an extension: in a basic graph we have vertices and we connect between two vertices using edges. In a port graph, it’s a bit different: each vertex has a set of ports, and edges don’t connect between two vertices, they connect between a vertex and a port (denoted vport) to another vport. That way, we can think of ports as points in the vertex to which edges connect to.

In the example below, two edges connect vertex to vertex ; but one of them connects 's green port to 's Cyan port, while another connects the blue port of to that of .

Diagram

Description automatically generated

A real-life example of a Port Graph are logic gates. For example, if we look at an gate as a vertex, then this vertex has 3 ports: two input ports and one output port. Using this example, we can represent a logic circuit as a Port Graph, and use graph algorithms (or graph algorithms that are extended to port graphs) do analyze the logic circuit.

Our goal in this project is to implement a library that through it we can represent Port Graphs as a data structure and to analyze this data structure through graph algorithms.

**Formal Definition**

A port graph consists of the following triplet: , such that:

1. A set of vertices: .
2. Sets of ports (a set for each vertex): , where each set of ports is

.

1. A set of edges: , such that each edge is an ordered pair of the form such that . This pair indicates that there is an edge from port in vertex to the port in vertex . Formally, we have

**Other graphs/graphs libraries and comparisons**

**Bond Graph**

Bond graph is graphical representation of a physical dynamic systems with the major difference that the arcs-edges are bi-direction (exchange of physical energy). Also, it allows the conversion of the system into a state-space representation which means it represents the physical system as a set of input, output and a state variable whose values evolve over time.

**Port Graph vs Bond Graph**

Port Graph is a graph where each edge connects nods via “port labels” associated the nodes, in CS port graph is majorly used for graph rewriting in which we can create new graph out of an original graph algorithmically, Bond Graph for example are used for graphical representation of a physical dynamic systems in which we can represent any systems, although Bond Graph is based on Port Graph it has other feature that distinguish him from other graphs especially Port Graph, with Bond Graph we can assume the direction of the date flow so latter it may be corrected , feature called “half-arrow” it’s widely used in the representation of a physical dynamic systems because as with electrical circuit diagrams and free-body diagrams, the choice of positive direction is arbitrary, for example with the representation of an electrical systems we can assume the direction of the positive energy flow, unlike Port Graph in which we declare in a concreate way the direction of the flow-edge

**igraph:**

igraph is a library collection for graphs and analyzing networks. It’s open source, and is used for generating and analyzing graphs, as well as computing different properties for graphs like path length-based properties and graph components. The library is written in C but packages for Python and R also exist. igraph is mainly used for academic research in network science and related fields.

**DOT graph:**

DOT is a graph description language. DOT graphs are usually files with .gv or .dot extension. DOT is widely used because lots of programs can process DOT files.

DOT can be used to describe undirected graphs:

or directed graphs:

Furthermore, attributes can be applied to the graphs, or their nodes and edges. These attributes can describe certain aspects of the graph and its components, like color or shape.

**igraph vs DOT graph**

igraph is a library collection that allows generating different graphs and analyzing them using different functions (like calculating shortest path length for given vertices), while DOT graph is a description language that allows us to represent graphs in a generic way through files, so that other programs can read the file and get the graph from it.

Difference between igraph and DOT graph is that igraph is a library collection that we can use to build and analyze graphs, while DOT graph is a description language to represent graphs in a file in a generic way and other programs use it to analyze said graphs. Since igraph is a library collection, we can use independently to analyze and compute different properties of different graphs, while DOT graph is just a portable way to represent graphs (as a file) so that other programs can use this representation to analyze the graph through its DOT file.

**Implementation**

In this part we are going to present 3 different possible implementations of a Port Graph:

1. Port Graph as Nodes and Edges:

Let the set of vertices, the set of ports and (such that ) the set of edges (each edge consists of two pairs of edge and port). Our port graph data structures consist of the set of vertices , such that each vertex is a node that contains a set of vertices that represents the outgoing edges from .

Pros:

1. Very simple and easy to implement.
2. Very similar to the theoretical definition of port graph and easy to read.

Cons:

1. This implementation is more of a theoretical implementation and the complexity of each operation is dependent on how these sets and nodes are implemented.
2. Being simple and easy to read is a possible trade-off for performance.

Because of these cons, we’re going to present two more implementations that are different from this theoretical implementation.

1. Adjacency hash array version 1:

The first version is an adjacency **hash array** of size V x P (max) where is the number of the current vertices and is the max port number in the graph. Let the array be with a key (v,p) ,a slot represents all the edges that coming out form vertex and port , so each slot has a data struct which is a **AVL tree** for all the pairs such that there is an edge from to .

Pros:

1. Add edge operation takes in worst case.
2. Edge search operation takes in worst case.
3. Takes liner space according to the graph -
4. Add vertex operation takes in average.

Cons:

Although Edge search operation takes it’s important to optimize it, to for example, as we will see in the second version.

1. Adjacency hash array version 2:

The second version is an adjacency **hash array** of size V x P (max) , where V is the number of the current vertices and P is the max port number in the graph. Let the array be , a slot represents the edges that coming out from vertex and port , so each slot has a data struct which is an **unordered set** (hash array) for all the pairs such that there is an edge from to .

Pros:

1. Add edge operation takes on average.
2. Edge search operation takes in the worst case.
3. Add vertex operation takes on average.

Cons:

This version heavy depends on hash arrays which it’s size can be dynamically change according to the ratio between the size of the **adjacency array** and the amount of the **total adjacencies** in the graph (פקטור העומס) so in a large scale this would be overkill because this version will use more space than needed.

Final Implementation

Out Port Graph is made up from three main classes:

* + - 1. Port: this class represents the ports in Port Graph, it includes:
         1. port\_id (an int).
         2. Attribute P (a template).
      2. Vetrex: this class represents the vertices in Port Graph, it includes:
         1. vertex\_id (an int).
         2. PortMap that contains all the ports of the vertex.
         3. Attribute V (a template).

From these definition we can now define two new terms:

* + vport: a pair of Vertex and Port.
  + vport\_id: a pair of vertex\_id and port\_id.
    - 1. Edge: this class represents the edges in Port Graph. Since an edge connects between a pair of vertex and port (vport) to another vport, our Edge class includes:
         1. source vport.
         2. dest vport.
         3. edge\_id (which is a pair of vport\_ids).
         4. Attribute E (a template).

Finally, our Port Graph class includes

1. An adjacency list that maps from vertex and port id (vport\_id) to a set of edges that represent the outgoing edges from this vport. For optimization we also added another adjacency list that it used when we want our Port Graph to be transposed.
2. A vport map that maps from vertex id to the vertex itself.
3. For caching of shortest\_paths algorithms, we added two new maps, shortest\_paths\_weights and shortest\_paths that maps from a pair of vport\_id to the shortest path weight between them and the shortest path itself.