**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 is a bit different: each vertex has a set of ports, and edges do not connect between two vertices. Instead, they connect between a pair of 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) to 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. (https://en.wikipedia.org/wiki/Bond\_graph#:~:text=A%20bond%20graph%20is%20a,into%20a%20state%2Dspace%20representation., n.d.)

**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 is 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.

(https://en.wikipedia.org/wiki/Igraph, n.d.)

(https://igraph.org/, n.d.)

**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.

(https://en.wikipedia.org/wiki/DOT\_(graph\_description\_language), n.d.)

**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.

**Representation**

In this part we are going to present 3 different possible representations 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 would be represented by a 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 in any programming language because it does not include any sophisticated definitions (just sets, pairs and nodes which are not very complicated).
2. Very similar to the theoretical definition of port graph and easy to understand.

Cons:

1. This representation is more of a theoretical implementation and doesn’t take into consideration if we want to implement any operations/algorithm on this representation.
2. 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** 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.

I**mplementation**

Our concrete implementation in the C++ programming language of the Port Graph as a data structure is derived from the 3rd implementation we described above. Our 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 maps port\_id to port (std::map<int, Port>).
         3. Attribute V (a template).

From these definitions we can now define two new terms:

* + vport: a pair of Vertex and Port.
  + vport\_id: a pair of vertex\_id and port\_id.
  + pair\_vport\_id: a pair of vport\_ids.
    - 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).

We decided to make these three classes (Port, Vertex and Edge) as separate classes from the PortGraph class because we think that these classes can exist on their own and can be used independently from PortGraph.

We also are going to need these definitions for the attributes later:

* + - 1. VerticesAttributes = vector<V> (a vector of vertex attributes).
      2. PortsAttributes = vector<P> (a vector of port attributes).
      3. EdgesAttributes = vector<E> (a vector of edge attributes).

Finally, our Port Graph class includes:

1. map<vport\_id, set<Edge>> adjacency\_list: 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.
2. map<vport\_id, set<Edge>> backwards\_adjacency\_list: An adjacency list that maps from vertex and port id (vport\_id) to a set of edges that represent the incoming edges from this vport. This map was added so we can reverse the graph in time (just swap adjacency\_list with backwards\_adjacency\_list).
3. We also added an undirected version of the adjacency list that is needed in MaxFlow alogirithm.
4. We also added an adjacency list that maps vertex id to its neighbour vertices.
5. A vport map that maps from vport id to the vport itself.
6. For caching of shortest\_paths algorithms, we added two new maps, *shortest\_paths\_weights* that maps from a pair of vport ids to the shortest path’s weight between them, and *shortest\_paths* that maps from a pair of vport ids to the shortest path between (a path is a vector of edges).

**Algorithms and Operations**

In this part we are going to present the algorithms and operation that we can perform on a Port Graph. Our explanation will include a brief description on the algorithm/operation and the function name, the function’s parameter and its return value.

Firstly, to initialize a new PortGraph, we use the following constructor:

PortGraph<V, P, E> (int n\_vertices, vector<int> ports\_num, vector<edge\_id> edges\_list,

VerticesAttributes verticesAttributes, vector<PortsAttributes> portsAttributes, EdgesAttributes edgesAttributes)

The parameters:

* + - 1. V: This represents the class that you want your vertex’s attribute to be. Its default value is int.
      2. P: This represents the class that you want your port’s attribute to be. Its default value is int.
      3. E: This represents the class that you want your Edge’s attribute to be. Its default value is int.
      4. n\_vertices: The number of vertices in your PortGraph. The IDs of the vertices will be .
      5. ports\_num: A vector of size n\_vertices, each entry in this vector represents the number of ports in the appropriate vertex. The IDs of the ports in vertex of ID k will be (for example, if the first entry is 4 then the vertex of ID 0 has four ports numbered ).
      6. edges\_list: A vector of edge\_id (a pair of vport\_ids). This list represents the edges that are in the PortGraph. Each edge\_id should have valid values, meaning they should contain a pair of valid vports (vports that exist in the PortGraph): in a more mathematical language, if edge\_id is then
      7. verticesAttributes, portsAttributes, edgesAttributes: A vector of vertex/port/edge Attribute (template class V/P/E). This class represents the attribute of each vertex/port/edge. Since PortGraphs can exist without attributes, these parameters are optional.

**BFS**

Breadth-first search was implemented as an Iterator for the PortGraph class. There are two kinds of BFS Iterators: a vertex iterator and a vport iterator. The BFS iterator iterates the vertices/vports in a BFS way – it starts with a starting vport/vertex and then it iterates through all its neighbors and then to all its neighbors’ neighbors and so on…

To initialize the iterators:

BFSIterator<V,P,E> itr(vport\_id id)

BFSVertexIterator<V,P,E> itr(vertex\_id id)

To advance the iterator:

++itr

Or

itr++

(the difference is that the first one returns the new value of itr and the second one returns the previous value of itr).

If all the vports/vertices are visited then Itr returns a dummy vport/vertex iterator which can be accessed from vportEnd()/vertexEnd() which represent the end of the PortGraph’s vports/vertices (in reality it’s a vport/vertex with -1 as an id, meaning an illegal value of id for vport/vertex).

Another way to advance the iterator is to use:

Itr.next()

The difference between ++itr and itr.next() is that when all the reachable vports/vertices from the initial value of the iterator are iterated over, ++itr continues from another unvisited vport/vertex (and if there are none left then it returns vportEnd()/vertexEnd()), while itr.next() returns immediately vportEnd()/vertexEnd() (without continuing from another unvisited vport/vertex).

To iterate over the PortGraph (lets called it pg) using DFS, one can write for example:

for(BFSIterator</\*the template arguments\*/> itr(&pg,/\*starting vport id\*/); itr != pg.vportEnd() ;++itr) {

        // do something with itr.

    }

**Psudo Code:**

­For implementing BFS, we need shall need a queue and “current” vport/vertex variable.

When initializing the iterator with starting vport/vertex v:

* + - 1. Push v to the queue.
      2. Mark v as current.
      3. Mark v as visited, and all other vports/vertices as notVisited.

When advancing the iterator (using ++ operator)

* + - 1. If queue is empty then mark vportEnd()/vertexEnd() as current.
      2. Else:

v = queue.front()

queue.pop()

* + - 1. For all nv neightbor of v do:

If (nv is not\_visited)

push nv to queue.

Mark nv as visited.

* + - 1. If queue is not empty then mark queue.front() as current.
      2. If queue is empty, then

Find a non-visited vport/vertex v.

Mark it as visited and current.

Push v to queue.

* + - 1. If all vports/vertices are visited then mark vportEnd()/vertexEnd() as current.

To dereference the iterator, return current.

**DFS**

Very similar to the BFS iterator, only it iterates in a DFS way – it starts with a starting vport/vertex and and iterates over each branch of the starting node as far as possible and then backtracks.

To iterate over the PortGraph (lets called it pg) using DFS, one can write for example:

for(DFSIterator</\*the template arguments\*/> itr(&pg,/\*starting vport id\*/); itr != pg.vportEnd() ;++itr) {

        // do something with itr.

    }

**Psudo Code:**

­For implementing BFS, we need shall need a “path” (a vector of vports/vertices) and “current” vport/vertex variable.

When initializing the iterator with starting vport/vertex v:

* + - 1. Push v to the queue.
      2. Mark v as current.
      3. Mark v as visited, and all other vports/vertices as notVisited.

When advancing the iterator (using ++ operator)

* + - 1. While there is nonVisited vports/vertices do:

For all neighbors v of current

If v is notVisited then:

Mark v as visited

Push v to the end of the path

Mark v as current

Return

path.pop\_back() (delete last member).

If path is not empty, then current = path.back() (last member).

If path is empty, then choose notVisited vport/vertex, mark it as visited and push it to path.

* + - 1. Let current be vportEnd()/vertexEnd().

To dereference the iterator, return current.

**Topological Sort**

If the graph is DAG (meaning the Port Graph doesn’t have cycles) then we can perform a topological sort on it. A topological sort means a linear ordering of the vports of the Port Graph, such that if there is an edge from vport to vport , then v comes before u in the ordering.   
The function:

vector<vport\_id> topological\_sort()

Return value: a vector of vport\_id such that if there’s an edge from vport to vport then v\_id comes before u\_id in the vector. If the PortGraph isn’t DAG then we return an empty vector.

**Psudo Code:**

s

**strongly\_connected\_components**

Strongly connect component of a Port Graph is a sub Port Graph such that in each component there’s an edge between each two vports in the component.

The function:

**Reverse\_graph**

This operation transposes the Port Graph: each edge from vport u to vport v becomes an edge from vort v to vport u. This is done in O(1) time because during the construction of the Port Graph we have two adjacency list: one for the regular Port Graph and one for the transposed Port Graph. We also have a flag that tells us if the Port Graph is transposed or not.

The method:

void transposeGraph()

**Psudo Code:**

Since in the construction of the Port Graph we already had two adjacency lists, one for the Port Graph and one for the reversed Port Graph, then we just set the flag that tells us that the Port Graph is transposed.

**min\_spanning\_tree**

This method computes a minimum spanning tree of the Port Graph according to a weight function that the user provides.

The method:

double Kruskal(WeightFunction wf)

Parameters:

wf: A weight function – it takes an edge\_id and output its weight. In C++ code:

typedef double (\*WeightFunction)(edge\_id);

Return value: the weight of minimum spanning tree.

**Psudo Code:**

s

**is\_bipartite**

A bipartite Port Graph is a Port Graph that its vports can be divided into two disjoint sets U and V such that each edge connects a vport in U to a vport in V or vice versa.

The method:

bool isBipartite()

Return value: true if the Port Graph is bipartite, else the return value is false.

**Psudo Code:**

s

**is\_reachable**

This method checks if vport/vertex v is reachable from vport/vertex u, meaning that there is a path from u to v.

The methods:

bool is\_reachable(vertex\_id source, vertex\_id dest)

Parameters:

1. source: the ID of the source vertex.
2. dest: the ID of the destination vertex.

Return value: true if dest is reachable from source, else the return value is false.

bool is\_reachable(vport\_id source, vport\_id dest

Parameters:

1. source: the ID of the source vport.
2. dest: the ID of the destination vport.

Return value: true if dest is reachable from source, else the return value is false.

**Psudo Code:**

* + - 1. Initialize a BFS Iterator that starts from the source vport.
      2. In each iteration, check if the iterator now points to the dest vport.
         1. If it does then return true
         2. Else, move to the next iteration.
      3. If we reached the end of the Port Graph, then return false.

**shortestpath**

This method calculates the shortest path between two vports. The weight of the path is calculated as the sum of the weights of the edges in the path. The weights of the edges are provided by the user through a Weight Function. Note that the weights must be positive  
This function also caches the shortest paths that it calculated before so that it can returns previously calculated paths faster. If a new Weight Function is used, then the cache must be cleared so that it does not return false paths. When a new Weight Function is used then the user must pass the parameter newWeight as true so that the cache is cleared.

The method:

Path shortestPath(WeightFunction wf, vport\_id src, vport\_id dst, bool newWeights)

Parameters:

1. wf: A Weight Function that takes an edge\_id and outputs a weight (positive double).
2. source: The ID of the source vport.
3. dest: The ID of the destination vport.
4. newWeight: A flag that marks if a new Weight Function is used (only relevant if this function was called before), its default value is true. If a previously used Weight Function is used, then this flag should be false so that the function can return previously calculated paths.

Return value: A Path which is a vector of edge\_id that represents the path between source and destination vport.

typedef vector<edge\_id> Path;

If there is no path between source and dest (dest is not reachable from source) then an empty Path is returned.

**Psudo Code:**

Remember, we have two data structures (caches) that will help us with this method: *shortest\_paths\_weights* that maps from a pair of vport ids to the shortest path’s weight between them, and *shortest\_paths* that maps from a pair of vport ids to the shortest path between.

* + - 1. If newWeights = false, then check *shortest\_paths[src, dst]* if it exists(meaning we calculated it in the past) and if it does then return it, if not then continue to step 3.
      2. If newWeights = true, then clear the cache (*shortest\_paths* and *shortest\_paths\_weights)* and continue to the next step.
      3. For each vport vp do:

*shortest\_paths\_weights[src, vp] =*

*shortest\_paths[src,vp] = {}*

*shortest\_paths\_weights[src, src] =*

* + - 1. Add all vports vp to priority queue called pq (sorted in increasing order according to shortest\_path\_weights[src, vp]), and initialize a map from vport to vport called prev such that for all vport vp do:

prev[vport] = undefined

* + - 1. While pq not empty do:

vport vp = pq.top()

pq.pop()

for all neighbors n\_vp of vp do:

dist = *shortest\_paths\_weights[src, vp] + wf(vp, n\_vp)*

*if dist < shortest\_paths\_weights[src, n\_vp] then do:*

*shortest\_paths\_weights[src,n\_vp] = dist*

*prev[n\_vp] = vp*

* + - 1. For each vport vp, back track path according to prev map and build path from src to vp (if exists) and add it to *shortest\_paths[src,vp].*
      2. Return *shortest\_paths[src,dst].*

**shortestPathWeight**

This method is very similar to shortestPath method. The only difference that it returns the weight of the path and not the path itself.

The method:

double shortestPathWeight(WeightFunction wf, vport\_id src, vport\_id dst, bool newWeights)

Parameters:

1. wf: A Weight Function that takes an edge\_id and outputs a weight (positive double).
2. source: The ID of the source vport.
3. dest: The ID of the destination vport.
4. newWeight: A flag that marks if a new Weight Function is used (only relevant if this function was called before), its default value is true. If a previously used Weight Function is used, then this flag should be false so that the function can return previously calculated weights.

Return value: The weight of the shortest path from source to dest. If there is no path between source and dest (dest is not reachable from source) then DBL\_MAX is returned.

**Psudo Code:**

* + - 1. **­­** If newWeights = false, then check the *shortest\_paths\_weights[src, dst]* if it exists (meaning we calculated it in the past) and if it does then return it, if not then continue to step 3.
      2. If newWeights = true, then clear the cache (*shortest\_paths* and *shortest\_paths\_weights)* and continue to the next step.
      3. Call shortestPath(wf, src, dst, newWeights) and then return *shortest\_paths\_weights[src, dst].*

**findClique**

A clique is a Port Graph that has an edge from every vport/vertex to every other vport/vertex. This method tries to find a clique of a given size in the Port Graph and returns it if it exists.

The method (vport version):

PortGraph<V,P,E> findVportClique(int k)

Parameters:

1. k: The size of the clique.

Return value: A Port Graph that is a clique of size k if it exists or an empty Port Graph if it doesn’t exist.

The method (vertex method):

vector<vertex\_id> findVertexClique(int k)

Parameters:

1. k: The size of the clique.

Return value: A vector of vertex\_id (integer) that represent the vertices that make up the clique or an empty vector if no clique exist.

**Psudo Code:**

s

**isSubGraph**

This method checks if a given Port Graph is a sub–Port Graph of the original Port Graph, meaning that its vertices, ports and edges are subsets of the original Port Graph’s vertices, ports and edges.

The method:

bool isSubGraph(PortGraph<V,P,E>& sub\_graph,bool vertex\_attr\_check,bool ports\_attr\_check,bool edge\_attr\_check)

Parameters:

1. sub\_graph: The Port Graph that we want to check.
2. vertex\_attr\_check: If we want to check if attributes of the vertices match as well then this parameter should be true, else it should be false.
3. ports\_attr\_check: If we want to check if attributes of the ports match as well then this parameter should be true, else it should be false.
4. edge\_attr\_check: If we want to check if attributes of the edges match as well then this parameter should be true, else it should be false.

Return value: True if the sub\_graph is indeed a sub Port Graph of the original Port Graph, else it returns false.

**Psudo Code:**

s

**max\_flow**

This function calculates the maximum flow between a source vertex and a destination vertex according to a capacity function that the user provides.

The method:

int maxFlow(CapacityFunction cf, vport\_id src, vport\_id dst)

Parameters:

1. cf: A CapacityFunction that represents the capacity of each edge: takes an edge\_id and returns capacity (int).

typedef int (\*CapacityFunction)(edge\_id);

1. src: The source vport.
2. dst: The destination vport.

Return value: the maximum flow of the network between src and dst.

**Psudo Code:**

In this method we are going to need to use a capacity\_map (maps edge to int) and prev\_map (maps vport\_id to vport\_id).

* + - 1. max\_flow = 0
      2. for all edges e:

capacity\_map[e] = cf(e)

* + - 1. flow = maxFlowAux(capacity\_map, prev\_map, src, dst)
      2. while flow != 0 do:

max\_flow += flow

curr = dst

while curr != src do:

parent = prev\_map[curr]

capacity\_map[(parent, curr)]-=flow

capacity\_map[(curr, parent)]+=flow

curr = parent

flow = maxFlowAux (capacity\_map, prev\_map, src, dst)

* + - 1. return max\_flow

maxFlowAux(capacity\_map, prev\_map, vport src, vport dst):

In this helper method we are going to use a queue that stores a pair of vport and flow (int).

* + - 1. prev\_map[src] = undefined
      2. queue.push(src, )
      3. while queue is not empty do:

curr = queue.front.vport

curr\_flow = queue.front.flow

queue.pop()

for all v neighbors of curr do:

if prev\_map[v] == null & capacity\_map[(curr,v)] != 0 do:

prev\_map[v] = curr

flow = min(curr\_flow, capacity\_map[(curr, v)])

if v == dst then return flow

queue.push(v, flow)

* + - 1. return 0