Project in Data Processing 236323

Winter Semester 2020-2021

Project Title: Port Graph Representation Library

Project Supervisor: Avi Mendelson

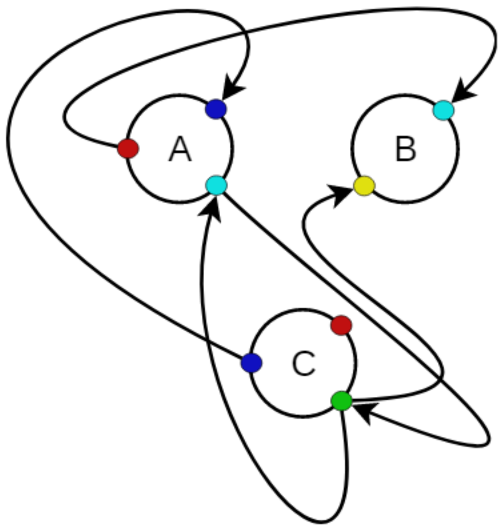
Students: Mario Barbara, Feras Bisharat

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 C to vertex A; but one of them connects A's green port to C's Cyan port, while another connects the blue port of C to that of A.



(https://app.box.com/s/7zef5rtolj19zmi3xokxaqhk6xd5rxlf, n.d.)

A real-life example of a Port Graph are logic gates. For example, if we look at an AND 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 G consists of the following triplet: $G = ((V, P), E)$, such that:

1. A set of vertices: $V = \{v_1, v_2, \dots, v_n\}$.
2. Sets of ports (a set for each vertex): $P = \{P_1, P_2, \dots, P_n\}$, where each set of ports is $P_i = \{p_1, p_2, \dots, p_{n_i}\}$.
3. A set of edges: $E = \{e_1, e_2, \dots, e_m\}$, such that each edge e_k is an ordered pair of the form $((v_i, p_i), (v_j, p_j))$ such that $p_i \in P_i, p_j \in P_j$. This pair indicates that there is an edge from port p_i in vertex v_i to the port p_j in vertex v_j . Formally, we have $E \subseteq (V \times P) \times (V \times P)$.

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 later 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:

$graph\ g\ \{$

$A - C - B$

$B - D$

$\}$

or directed graphs:

$graph\ e\ \{$

$A -> C -> B$

$B -> D$

$\}$

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.

Goal

Our goal in this project is to implement a basic Port Graph library in the C++ programming language that will include the Port Graph as a concrete data structure that will support several operations:

1. Building a Port Graph.
2. Modifying an existing Port Graph: adding a new vertex/edge or deleting a vertex/edge…
3. Performing graph algorithms that will be extended to Port Graph, like:
    a. BFS, DFS iterators,
    b. Topological sort,
    c. Minimum spanning tree,
    d. Shortest Path,
    e. Is Reachable,
    f. Induced Graph,
    And many more.

Representations

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

1. Port Graph as Nodes and Edges:

   Let $V = \{v_1, v_2, \dots, v_n\}$ the set of vertices, $P = \{p_1, p_2, \dots, p_k\}$ the set of ports and $E = \{((u_1, q_1), (w_1, r_1)), ((u_2, q_2), (w_2, r_2)), \dots, ((u_m, q_m), (w_m, r_m))\}$ (such that $u_1, u_2, \dots, u_m \in V, q_1, q_2, \dots, q_m, r_1, r_2, \dots, r_m \in P$) 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 V, such that each vertex v_i is a node that contains a set of vertices E_i that represents the outgoing edges from v_i.

   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 and exist in any programming language).
   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 matrix** 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 matrix be $adj[]$with a key (v,p) ,a slot $adj[(v,p)]$ represents all the edges that coming out form vertex v and port p , so each slot $adj[(v,p)]$ has a data struct which is a **AVL tree** for all the pairs (u,k) such that there is an edge from (v,p) to (u,k).

   Pros:

   1. Add edge operation takes $O(log(|V| \cdot |P|))$ in worst case.
   2. Edge search operation takes $O(log\ (|V| \cdot |P|))$ in worst case.
   3. Takes liner space according to the graph - $O(|V| \cdot |P| + |E|)$
   4. Add vertex operation takes $O(1)$ in average.

   Cons:

   Although Edge search operation takes $O(log(|V| \cdot |P|))$ it's important to optimize it, to $O(1)$ for example, as we will see in the second version.

3. Adjacency hash array version 2:

   The second version is an adjacency **hash matrix** 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 $adj[]$, a slot

$adj[(v, p)]$ represents the edges that coming out from vertex v and port p, so each slot $adj[(v, p)]$ has a data struct which is an **unordered set** for all the pairs (u, k) such that there is an edge from (v, p) to (u, k) .

Pros:

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

Cons:

This version heavy depends on hash arrays which its size can be dynamically change according to the ratio between the size of the **adjacency matrix** 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.

Implementation

Our concrete implementation in the C++ programming language of the Port Graph as a class is derived from the 3rd implementation we described above with more added class members. Our Port Graph class is made up from three main classes:

1. Port: this class represents the ports in Port Graph, it includes:
    a. port_id (an int).
    b. Attribute P (a template).
2. Vetrex: this class represents the vertices in Port Graph, it includes:
    a. vertex_id (an int).
    b. PortMap that maps port_id to port (std::map<int, Port>).
    c. 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.

3. 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:
    a. source vport.
    b. dest vport.
    c. edge_id (which is a pair of vport_ids).
    d. Attribute E (a template).

It is important to note that all these classes are templated (because they include a template member – the attribute) but they can be used without the attribute – just set the template to int and ignore it.

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, and hence they need a separate id for each of them (the ID of the edge can be derived from the source and dest vports but for convenience purposes we added the ID separately).

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 $O(1)$ 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 algorithm.
4. map<vertex_id, vertex> vertex_map: A map that maps vertex_id to the vertex itself.

5. We also added an adjacency list that maps vertex id to its neighbour vertices' ids.
6. We also added a reversed version of 4 so we can reverse the graph in $O(1)$ time (just swap the two maps).
7. 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 $0, 1, 2, \dots, n\_vertices - 1$.
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 $0, 1, 2, \dots, ports\_num[k] - 1$ (for example, if the first entry is 4 then the vertex of ID 0 has four ports numbered $0, 1, 2, 3$).
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 $((vertexId_1, PortId_1), (vertexId_2, PortId_2))$ then
$$0 \le vertexId_1, vertexId_2 \le n\_vertices - 1$$
$$0 \le PortId_1 \le ports\_num[vertexId_1] - 1$$
$$0 \le PortId_2 \le ports\_num[vertexId_2] - 1$$
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.
    }
```

Pseudo 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. $Else:$
            $v\ =\ queue.front()$
            $queue.pop()$
    3. $For\ all\ nv\ neightbor\ of\ v\ do:$
            $If\ (nv\ is\ not\_visited)$
                    $push\ nv\ to\ queue.$
                    $mark\ nv\ as\ visited.$
```

```
4.  $If\ queue\ is\ not\ empty\ then\ mark\ queue.front()\ as\ current.$
5.  $If\ queue\ is\ empty, then$
        $Find\ a\ non-visited\ vport/vertex\ v.$
        $Mark\ it\ as\ visited\ and\ current.$
        $Push\ v\ to\ queue.$
6.  $If\ all\ vports/vertices\ are\ visited\ then\ mark\ vportEnd()/vertexEnd()\ as\ current.$
```

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 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.
    }
```

Pseudo 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.  $While\ there\ is\ nonVisited\ vports/vertices\ do:$
        $For\ all\ neighbors\ v\ of\ current$
            $If\ \ v\ is\ notVisited\ then:$
                $Mark\ v\ as\ visited$
                $Push\ v\ to\ the\ end\ of\ the\ path$
                $Mark\ v\ as\ current$
                $Return$
        $path.pop\_back()\ (delete\ last\ member).$
        $If\ path\ is\ not\ empty, then\ current\ =\ path.back()\ (last\ member).$
        $If\ path\ is\ empty, then\ choose\ notVisited\ vport/vertex, mark\ it\ as\ visited$
        $and\ push\ it\ to\ path.$
2.  $Let\ current\ be\ vportEnd()/vertexEnd().$
```

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 v to vport u, then v comes before u in the ordering, for Implementing this method we will need a data struct to keep track with each vport's indegree let's call it $indgree_{vports}$, which maps vports to the indegree of the corresponding port.
The function:

```
vector<vport_id> topological_sort()
```

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

Pseudo Code:

```
1. $indegree_{vports} \leftarrow \emptyset, order \leftarrow \emptyset$
2. $foreach\ edge\ e = \{u, v\}\ do:$
        $indegree_{vports}[v] += 1$
3. $foreach\ vport\ vp\ do:$
        $if\ vp.indegree\ is\ 0:$
            $order.push\_back(vp)$
4. $foreach\ edge\ e = \{u, v\}\ do:$
        $indegree_{vports}[v] -= 1$
        $if\ v.indegree\ is\ 0:$
            $order.push\_back(v)$
5. $if\ order.size = vports.Num : return\ order$
6. $else : return\ empty\ set\ \emptyset$
```

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()
```

Pseudo 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. For implementing this method, we will need Union Find data structute, in which every set's ID is a vport ID, to keep track with each connected component. For maintaining the struct will we use the following operations:

1. Find(u): return set's ID storing u.

2. Unite(u, v): replace the sets storing u and v with their union and updating the set ID with $\min\{find(u), find(v)\}$

The method:

```
double findMST(WeightFunction wf)
```

Parameters:

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

```
typedef double (*WeightFunction)(edge_id,E /* E: type of edge attribute*/);
```

Return value: a pair with a type of <double, PortGraph>, in which the first is the weight of the MST and the second is the MST represented as a port graph.

Pseudo Code:

```
1. $mst_{weight} \leftarrow 0, mst \leftarrow \emptyset$
2. $edges_w \leftarrow \emptyset$
3. $foreach\ edge\ e\ do:$
    $edges_w \leftarrow edges_w \cup \{e, e_w\} // e_w \coloneqq wf(e.ID, e.Attribute)$
4. $sort(edges_w)$
5. $UnionFindDS\ components(PG_{vports})$
5. $foreach\ edge\ e = \{u, v\}\ in\ edges_w\ do:$
    $if\ components.Find(u) \neq components.Find(v):$
        $mst.AddVport(u.ID, u.Attribute)$
        $mst.AddVport(u.ID, u.Attribute)$
        $mst.addEdge(e.ID, e.Attribute)$
        $mst_{weight} += e_w$
```

```
                    $components.Unite(u,v)$
6. $retrun\ \{mst_{weight}, mst\}$
```

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. Graphs in general could be unconnected, thus we need to check every connected component if it's a bipartite or not, so for this case we will implement a helper function called isBipartiteUtil, which will check each connected component and the main method isBipartite will keep track for each such component.

The method:

```
bool isBipartiteUtil(vport_id src, map<vport_id,int>& colorArr)
```

Parameters:

1. Src: the root of the current connected component in which the function if it can be represented as a bipartite graph or not.
2. ColorArr: a data srtuct which map each vport(id) to a color representing the current status of this vport, we will use the following colors:
   1. Color "-1": which will represent a vport that haven't been checked.
   2. Color "0": which will represent a vport included in set U.
   3. Color "1": which will represent a vport included in set V.

Return value: true if the connected component with the root 'src' represent a bipartite graph, else the return value is false.

Pseudo Code:

```
1. $colorArr[src] = 1$
2. $queue \leftarrow \{src\}$
3. $while\ queue\ isn't\ empty\ do:$
      $u \leftarrow queue.popFront()$
      $if\ isNeighbours(u,u):\ return\ false$
      $foreach\ vport\ v\ do:$
            $if\ isNeighbours(u,v)\ \textbf{and}\ colorArr[v]\ is\ \text{-1}:$
                  $colorArr[v] = 1 - colorArr[u]$
                  $queue \leftarrow queue \cup \{v\}$
            $else\ if\ isNeighbours(u,v)\ \textbf{and}\ colorArr[v] = colorArr[u]:$
```

```
                    $return\ false$
4. $return\ true$
```

The method:

```
bool isBipartite()
```

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

Pseudo Code:

```
1.$map\ colorArr\ \leftarrow \emptyset$
2.$foreach\ vport\ vp\ do:$
      $colorArr[vp] \leftarrow -1$
3.$valid \leftarrow true$
4.$foreach\ vport\ vp\ do:$
      $if\ colorArr[vp]\ is\ -1:$
            $valid\ \leftarrow vailed\ \textbf{and}\ isBipartiteUtil(vp,\ colorArr)$
5.$return\ valid$
```

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.

Pseudo Code:

```
1. $Initialize\ a\ BFS\ Iterator\ that\ starts\ from\ the\ source\ vport.$
2. $In\ each\ iteration, check\ if\ the\ iterator\ now\ points\ to\ the\ dest\ vport.$
        a. $If\ it\ does\ then\ return\ true$
        b. $Else, move\ to\ the\ next\ iteration.$
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 newWeigh
ts)
```

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.

Pseudo 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. $For\ each\ vport\ vp\ do:$
```

```
        $shortest\_paths\_weights[src, vp] = \infty$
        $shortest\_paths[src, vp] = \{\}$
        $shortest\_paths\_weights[src, src] = 0$
4.  $Add\ all\ vports\ vp\ to\ priority\ queue\ called\ pq\ (sorted\ in\ increasing\ order\ according\ to$
    $shortest\_path\_weights[src, vp]), and\ initialize\ a\ map\ from\ vport\ to\ vport\ called\ prev$
    $such\ that\ for\ all\ vport\ vp\ do:$
        $prev[vport] = undefined$
5.  $While\ pq\ not\ empty\ do:$
        $vport\ vp = pq.top()$
        $pq.pop()$
        $for\ all\ neighbors\ n\_vp\ of\ vp\ do:$
                $dist = shortest\_paths\_weights[src, vp] + wf(vp, n\_vp)$
                $if\ dist < shortest\_paths\_weights[src, n\_vp]\ then\ do:$
                        $shortest\_paths\_weights[src, n\_vp] = dist$
                        $prev[n\_vp] = vp$
6.  $For\ each\ vport\ vp, back\ track\ path\ according\ to\ prev\ map\ and\ build\ path\ from\ src\ to$
    $vp\ (if\ exists)\ and\ add\ it\ to\ shortest\_paths[src, vp].$
7.  $Return\ shortest\_paths[src, dst].$
```

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.

Pseudo 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.

For Implementing both versions we will use a helper function called **findClique,** which returns true if it can construct by brute force, a clique of size K with the vports/vertices in 'candidates' set. If so, the nodes of such clique will be stored in result parameter, else the function returns false and result parameter will be an empty set.

The method:

```
Template<T> /* with the way the function is being used, T type will be int
(vertex_id) or vport_id
bool findClique(vector<T>& candidates, vector<T>& result, int k)
```

Parameters:

1. candidates: the nodes which construct the clique
2. result: the result of the current "clique".
3. K: The size of the clique.

Pseudo Code:

```
1. $if\ result.size = K: return\ true$
2. $if\ candidates.size = 0 : return\ false$
3. $u\ = candidates.back()$
4. $candidates.remove(u)$
5. $if\ findClique(candidates, result, K)\ is\ true : return\ true$
6. $inClique\ \leftarrow true$
7. $foreach\ node\ v\ in\ result\ do :$
       $inClique = inClique\ \textbf{and}\ isNeighbours(u, v)\ and\ isNeigbours(v, u)$
8. $if\ InClique\ is\ false :$
       $candidates.Add(u)$
       $return\ false$
9. $if\ findClique(candidates, result\ \cup \{u\}, K) is\ true : return\ true$
10. $candidates.Add(u), return\ false$
```

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.

Pseudo Code:

```
1. $result \leftarrow \emptyset, candidates \leftarrow \emptyset$
2. $foreach\ vport\ vp\ do:$
        $if\ vp.Outdegree \geq K-1\ \textbf{and}\ vp.Indegree \geq K-1:$
              $candidates.add(vp)$
3. $if\ findClique(candidates, resukt, K)\ is\ false: return\ empty\ PortGrpah$
4. $else:$ // A clique of size k can be found in the graph
5. $PortGraph\ Clique \leftarrow \emptyset$
6. $foreach\ two\ vports\ u, v\ in\ result\ such\ that\ u \neq v\ do:$
              $CLique.AddVport(u.ID, u.Attribute)$
              $CLique.AddVport(v.ID, v.Attribute)$
              $Clique.AddEdge(\{u,v\}.ID, \{u,v\}.Attribute)$
              $Clique.AddEdge(\{v,u\}.ID, \{v,u\}.Attribute)$
8. $return\ Clique$
```

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.

Pseudo Code:

```
1. $result \leftarrow \emptyset, candidates \leftarrow \emptyset$
2. $foreach\ vertex\ v\ do:$
```

```
    $if\ v.Outdegree \geq K-1$ **and** $v.Indegree \geq K-1:$
        $candidates.add(v)$
3. $if\ findClique(candidates, resukt, K)\ is\ false : return\ empty\ set$
4. $else$ : // A clique of size k can be found in the graph
5. $Set < vport_{id} > \ Clique \ \leftarrow \emptyset$
6. $foreach\ two\ vertex\ v\ in\ result\ do:$
        $foreach\ port\ p\ in\ v.Ports\ do:$
            $Clique.Add(\{v, p\}.ID\ )$
7. $return\ Clique$
```

isSubGraph

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

The method:

```
bool isSubGraph(PortGraph<V,P,E>& that_graph,bool vertex_attr_check,bool ports
_attr_check,bool edge_attr_check)
```

Parameters:

1. that_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 original Port Graph is indeed a sub Port Graph of the given that_grraph, else it returns false.

Pseudo Code:

```
1. $foreach\ vertex\ v\ in\ original_{PG}\ do:$
    $if\ v\ can't\ be\ found\ in\ that_{PG}: return\ false$
    $if\ VertexAttrCheck\ is\ true$ **and** $v.Attribute\ \neq v_{that_{PG}}.Attribute:$
        $return\ false$
2. $foreach\ vport\ vp\ in\ original_{PG}\ do:$
    $if\ vp\ can't\ be\ found\ in\ that_{PG}: return\ false$
    $if\ VportAttrCheck\ is\ true\ \textbf{and}\ vp.Attribute\ \neq vp_{that_{PG}}.Attribute:$
        $return\ false$
3. $foreach\ edge\ e\ in\ original_{PG}\ do:$
    $if\ e\ can'tbe\ found\ in\ that_{PG}: return\ false$
    $if\ EdgeAttrCheck\ is\ true\ \textbf{and}\ e.Attribute\ \neq e_{that_{PG}}.Attribute:$
        $return\ false$
4. $retun\ true$
```

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);
   ```

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

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

Pseudo 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. $for\ all\ edges\ e:$
    $capacity\_map[e] = cf(e)$
3. $flow = maxFlowAux(capacity\_map, prev\_map, src, dst)$
4. $while\ flow\ != 0\ do:$
    $max\_flow += flow$
    $curr = dst$
    $while\ curr\ != src\ do:$
        $parent = prev\_map[curr]$
        $capacity\_map[(parent, curr)] -= flow$
        $capacity\_map[(curr, parent)] += flow$
        $curr = parent$
```

```
        $flow = maxFlowAux(capacity\_map, prev\_map, src, dst)$
5.  $return\ max\_flow$
```

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

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.  $queue.push(src, \infty)$
3.  $while\ queue\ is\ not\ empty\ do$:
        $curr = queue.front.vport$
        $curr\_flow = queue.front.flow$
        $queue.pop()$
        $for\ all\ v\ neighbors\ of\ curr\ do$:
            $if\ prev\_map[v] == null\ \&\ capacity\_map[(curr, v)]\ != 0\ do$:
                $prev\_map[v] = curr$
                $flow = min(curr\_flow, capacity\_map[(curr, v)])$
                $if\ v == dst\ then\ return\ flow$
                $queue.push(v, flow)$
4.  $return\ 0$
```

InducedGraph

Formally an induced subgraph of a graph is another graph, formed from a subset of the “vertices” of the graph and all the edges connecting pairs of “vertices” in that subset. for implementing such function and for the purposes of the project we will implement vertex, vport and edge version of induced sub-port graph.

The method (**vport version**):

Let $pred: vport \rightarrow bool$, a subset S which will form the vports of the induced graph is defined as follows : $S = \left\{ vp \in PortGraph_{vports} \,\middle|\, pred(vp) = true \right\}$, and the edges for such graph will be: $induced_{edges} = \left\{ e = (u \rightarrow v) \in PortGraph_{edges} \,\middle|\, u, v \in S. \right.$

```
PortGraph<V,P,E> inducedGraph(PredFunction pred)
```

Parameters:

1. Pred: a function the indicates which vports are included in the subset based on their id and attribute value, in C++ code:

```
typedef bool (*PredFunction)(vport_id,V,P);
//where V and P are the attribute’s type of the vertices and vports
```

Return value: vport induced port graph, where the vports of the graph are S and edges are pairs of vports in the subset.

Pseudo Code:

```
1. $induced_{PG} \leftarrow \emptyset,\ subset \leftarrow \emptyset$
```

```
2. $foreach\ vport\ vp\ do:$
        $if\ pred(vp.ID, vp.Attribute):$
            $subset.Add(vp)$
3. $foreach\ vport\ u\ in\ subset\ do:$
        $induced_{PG}.AddVport(u.ID, u.Attribute)$
        $foreach\ vport\ v\ in\ subset\ such\ that\ u \neq v\ do:$
            $induced_{PG}.AddVport(v.ID, v.Attribute)$
            $if\ isNeighbours(u, v):$
                $Edge\ e = new\ Edge(u, v, attribute_{u \to v})$
                $induced_{PG}.AddEdge(e.ID, e.Attribute)$
4. $return\ induced_{PG}$
```

The method (**vertex version**):

Let $pred: vertex \to bool$, a subset S which will form the vertices of the induced graph is defined as follows : $S = \{\, v\ \in PortGraph_{vertcies} \mid pred(vp) = true \,\}$, and the edges of such graph will be :

$$induced_{edges} = \{e = \{(u, p_u), (v, p_v)\} \in PortGraph_{edges} \mid u, v\ \in S\ and\ p_u \in u.Ports\ and\ p_v \in v.Ports\}$$

```
PortGraph<V,P,E> inducedGraph(PredFunction pred)
```

Parameters:

1. Pred: a function that indicates which vertices are included in the subset based on their id and attribute value, in C++ code :

```
typedef bool (*PredFunction)(vertex_id,V);
//where V is the attribute’s type of the vertices
```

Return value: vertex induced port graph, where the vertices of the graph are S and edges are pairs of vports in the subset, as defined above.

Pseudo Code:

```
1. $induced_{PG} \leftarrow \emptyset,\ subset \leftarrow \emptyset$
2. $foreach\ vertex\ v\ do:$
        $if\ pred(v.ID, v.Attribute):$
            $subset.Add(v)$
3. $foreach\ vertex\ u\ in\ subset\ do:$
        $induced_{PG}.AddVertex(u.ID, u.Attribute, u.Ports)$
        $foreach\ vertex\ v\ in\ subset\ such\ that\ u \neq v\ do:$
        $induced_{PG}.AddVertex(v.ID, v.Attribute, v.Ports)$
        $foreach\ p_u \in u.Ports\ \textbf{and}\ p_v \in v.Ports\ \textbf{and}\ isNeighbours(\{u, p_u\}, \{v, p_v\})$ :
            $Edge\ e = new\ Edge(\{u, p_u\}, \{v, p_v\}, attribute)$
            $induced_{PG}.AddEdge(e.ID, e.Attribute)$
4. $return\ induced_{PG}$
```

The method (**edge version**):

Let $pred: edge \rightarrow bool$, a subset S which will form the edges of the induced graph is defined as follows : $S = \{ e \in PortGraph_{edges} \mid pred(e) = true \}$, and the vports of the graph will be defined as follows : $induced_{vports} = \{u \in PortGraph_{vports} | \exists\ v \in PortGraph_{vports}\ and\ \{u, v\} \in S \}$

```
PortGraph<V,P,E> inducedGraph(PredFunction pred)
```

Parameters:

1. Pred: a function the indicates which edges are included in the subset based on their id and attribute value, in C++ code:
   ```
   typedef bool (*PredFunction)(edge_id, E);
   //where E is the attribute's type of edges
   ```

Return value: vport induced port graph, where the vports of the graph are S and edges are pairs of vports in the subset.

Pseudo Code:

```
1. $induced_{PG} \leftarrow \emptyset$
2. $foreach\ edge\ e = \{u, v\}\ do$ :
    $if\ pred(e.ID, e.Attribute)$ :
        $induced_{PG}.AddVport(u.ID, u.Attribute)$
        $induced_{PG}.AddVport(v.ID, v.Attribute)$
        $induced_{PG}.AddEdge(e.ID, e.Attribute)$
3. $return\ induced_{PG}$
```

Diff (between two port graphs)

With this method we tried to give an approximation for a diff between the original port graph and an another giver port graph, formally $diff_{PG} = original_{PG} \setminus other_{PG}$, thus $diff_{PG}$ has no vertex, vport or edge which is a part of $other_{PG}$ port graph.

Note: by using this method the original port graph and the other given port graph must be same template type, which means both has to have the same vertex, vport and edge attribute template type.

The method:

```
PortGraph<V,P,E> diff(PortGraph<V,P,E>& other_pg)
```

Parameters:

1. other_pg: The Port Graph that has the same type which we want to “subtract” by.

Return value: diff port graph which satisfies the definition mentioned above.

Pseudo Code:

```
1. $diff_{PG} \leftarrow original_{PG}$
2. $foreach\ edge\ e\ in\ other_{PG}\ do:$
      $diff_{PG}.removeEdge(e)$
3. $foreach\ vport\ \{v,p\}\ in\ diff_{PG}\ do:$
      $if\ \{v,p\}.degree\ is\ 0:$
            $diff_{PG}.removePort(\{v,p\})$
4. $foreach\ vertex\ v\ in\ diff_{PG}\ do:$
      $if\ v.degree\ is\ 0:$
            $diff_{PG}.removeVertex(v)$
5. $return\ diff_{PG}$
```