GraphLib

Graph library using the C++ STL

Ayoub Ouarrak

Dipartimento di Matematica e Informatica
Università degli studi di Parma
2013/2014

Contents

1 Class definition

1.1 Representation of nodes and edge

The nodes is represented by a vector of strings, the single edge is represented by a pair of string and the edges is a vector of pair.

```
/** eg. <v. u> */
typedef std::pair<std::string, std::string> link;

/** eg. {v1, v2, v3, ...} */
std::vector<std::string> _node;

/** eg. {<v1, u1>, <v2, u2>, ...}   */
std::vector<link>           _edge;

/** eg. {<v1,u1> = 1, <v2,u2> = 1  ,...}   */
std::map<link, double>    _edgeWeight;
```

1.2 Constructors

GraphLib offer the possibility to define directed/undirected graph using the static bool variable, it is also the possibility to generate a random graph.

```
/**
   Generate random graph

   @param maxNode    max node of the generated graph
   @param graphType directed/undirect graph
   @return Graph
*/
static Graph generateRandomGraph(int, bool graphType = directed);


/**
   @param   graphType   directed/undirect graph
*/
explicit Graph(bool graphType = directed);


/**
   Add Node using regex eg. G("A-Z"), G(1-5), G(12-82)

   @param regex      regex eg. A-Z, 1-6
   @param edgeType   random/circular edges generation
   @param graphType directed/undirect graph
*/
Graph(std::string regex, int edgeMode, bool graphType = directed);
```

```
/**
    Copy contructor

    @param  Graph graph to copy
*/
Graph(const Graph&);
```

1.3 Public Method

GraphLib offer the principal graph operation like add node, add edge and other utility function amd graph algorithm.

- Return a graph that is the transpose of *this
  cost: *O(E)*

  ```
  Graph transpose();
  ```

- DFS traversal of the vertices reachable from v. It uses recursive _DFSUtil()
  cost: *O(V+E)*

  ```
  void DFS(std::string sourceNode);
  ```

- Breadth First Traversal for a Graph
  cost: *O(V+E)*

  ```
  void BFS(std::string sourceNode);
  ```

- Solves the all-pairs shortest path problem using Floyd Warshall algorithm
  cost: *O(V^3)*

  ```
  void floydWarshell(double** graph);
  ```

- Assigns colors (starting from 0) to all vertices and prints the assignment of colors (greedy algorithm)

  ```
  void coloring();
  ```

- Draw the graph using html/javascript
  cost(worst-case): *O(V+E)*

  ```
  void draw() const;
  ```

- Print the graph on the standard output
  cost: *O(V+E)*

  ```
  void print(std::ostream&) const;
  ```

- Add node to the graph
  cost: *O(1)*

```
void addNode(std::string node);
```

- Remove node from the graph (and the edges connected to the node)
  cost(worst-case): *O(V+E)*

```
void removeNode(std::string node);
```

- Add edge to the graph (create the nodes if not present in the graph)
  cost: *O(1)*

```
void addEdge(std::string fromNode,
             std::string toNode,
             double cost = 1);
```

- Remove edge from the graph
  cost(worst-case): *O(V+E)*

```
void removeEdge(std::string from, std::string toNode);
```

- Set weight to edge(fromNode, toNode)
  cost(worst-case): *O(V)*

```
void setWeight(std::string fromNode,
               std::string toNode,
               double cost);
```

- Control if graph has negative weight
  cost(worst-case): *O(E)*

```
bool hasNegativeWeigth() const;
```

- Control if graph is cyclic
  cost: *O(V+E)*

```
bool isCyclic() const;
```

- Method to check if all non-zero degree vertices are connected
  cost: *O(V+E)*

```
bool isConnected() const;
```

- The function returns one of the following values
  0 → If grpah is not Eulerian
  1 → If graph has an Euler path (Semi-Eulerian)
  2 → If graph has an Euler Circuit (Eulerian) .

```
int isEulerian() const;
```

- Convert adjacent list into a matrix
  cost: *O(V*E)*

```
int** fromListADJToMatrixADJ();
```

- Return matrix of edge's weight, necessary for the floydWarshell algorithm
  cost: *O(V^2)*

  ```
  double** weightMatrix();
  ```

- Min rank of a graph
  cost: *O(V)*

  ```
  unsigned minRank() const;
  ```

- Max rank of a graph
  cost: *O(V)*

  ```
  unsigned maxRank() const;
  ```

- Adjacent list
  cost: *O(E)*

  ```
  std::list<std::string> adjacent(std::string v) const;
  ```

- Control if exist edge(fromNode, toNode)
  cost: *O(E)*

  ```
  inline bool
  hasEdge(std::string fromNode, std::string toNode) const;
  ```

- Control if the graph is oriented
  cost: *O(1)*

  ```
  inline bool
  isOriented() const;
  ```

- Control if minRank = maxRank
  cost: *O(V)*

  ```
  inline bool
  isRegular() const;
  ```

- Control if node
  cost: *O(V)*

  ```
  inline bool
  exist(std::string) const;
  ```

- Return the nodes vector
  cost: *O(1)*

  ```
  inline unsigned
  nodes() const;
  ```

- Return the edges vector
  cost: *O(1)*

```
inline unsigned
edges() const;
```

- Return the rank of node
  cost: *O(E)*

  ```
  inline unsigned
  rank(std::string v) const;
  ```

- Return the weight of edge(fromNode, toNode)
  cost: *O(E)*

  ```
  inline double
  weight(std::string fromNode, std::string toNode) const;
  ```

2 Class implementation

```
using namespace GraphLib;

typedef std::pair<std::string, std::string> link;
typedef std::map<std::string, bool> mapStringBool;

int  Graph::random = 0;
int  Graph::circular = 1;
bool Graph::directed = true;
bool Graph::undirected = false;

Graph::Graph(bool graphType) {
   direct = graphType;
}

Graph::Graph(const Graph& G) {
   _node = G._Node();
   _edge = G._Edge();
   _edgeWeight = G._EdgeWeight();
}

Graph::Graph(std::string regex, int edgeType, bool graphType) {
  direct = graphType;
  if(utility::checkIfInterval(regex)) {
    if(regex.length() == 3) {
      std::vector<char> tmp = utility::regexChar(regex);
      std::vector<char>::const_iterator it;
      for(it = tmp.begin(); it != tmp.end(); ++it)
        _node.push_back(utility::to_string(*it));
    }
    else if(regex.length() > 3) {
      std::vector<int> tmp = utility::regexInt(regex);
```

```
      std::vector<int>::const_iterator it;
      for(it = tmp.begin(); it != tmp.end(); ++it)
        _node.push_back(utility::to_string(*it));
    }
  }
  _generateEdge(edgeType);
}

void Graph::_generateEdge(int edgeType) {
  switch(edgeType) {
    /** random */
    case 0: {
      srand(time(NULL));
      for(unsigned i = 0; i < nodes(); ++i) {
        int randNode1 = rand() % nodes();
        int randNode2 = rand() % nodes();
        double randWeight = rand() % 100;
        if(randNode1 != randNode2)
          addEdge(_node.at(randNode1), _node.at(randNode2), randWeight);
      }
    }
    /** circular */
    case 1: {
      std::string initialNode = _node.at(0);
      std::vector<std::string>::const_iterator it;
      for(it = _node.begin(); it != _node.end(); ++it) {
        if(it + 1 != _node.end())
          addEdge(*it, *(it + 1));
        else
          addEdge(*it, initialNode);
      }
    }
  }
}

Graph Graph::generateRandomGraph(int maxNode, bool graphType) {
  srand(time(NULL));
  int fromInt = rand() % maxNode;
  int toInt = rand() % maxNode;
  std::string ivt = std::min(utility::to_string(fromInt),
                             utility::to_string(toInt)) +
                    "-" +
                    std::max(utility::to_string(toInt),
                             utility::to_string(fromInt));

  Graph G(ivt, Graph::random, graphType);
  return G;
}

Graph Graph::transpose() {
```

```
  Graph G;
  for(auto e = _edge.begin(); e != _edge.end(); ++e) {
    G.addEdge(e->second, e->first, weight(e->first, e->second));
  }
  return G;
}

void Graph::addNode(std::string node) {
  _node.push_back(node);
}

void Graph::removeNode(std::string node) {
  std::vector<std::string>::iterator v;
  std::vector<link>::iterator e;
  std::vector<link> edgeToRemove;
  v = std::find(_node.begin(), _node.end(), node);
  if(v != _node.end()) {
    /** remove the edge connected to the node */
    for(e = _edge.begin(); e != _edge.end(); ++e) {
      if(direct) {
        if(e->first == node || e->second == node)
          edgeToRemove.push_back(std::make_pair(e->first, e->second));
      }
      else {
        if(e->first == node)
          edgeToRemove.push_back(std::make_pair(e->first, e->second));
      }
    }
    /** removing edges */
    for(e = edgeToRemove.begin(); e != edgeToRemove.end(); ++e)
      removeEdge(e->first, e->second);
    _node.erase(v);
  }
}

void Graph::addEdge(std::string fromNode,
                    std::string toNode,
                    double cost) {
  if(!exist(fromNode))
    addNode(fromNode);

  if(!exist(toNode))
    addNode(toNode);

  if(!hasEdge(fromNode, toNode)) {
    if(direct) {
      _edge.push_back(std::make_pair(fromNode, toNode));
      _edgeWeight[std::make_pair(fromNode, toNode)] = cost;
    }
    /** undirected graph */
```

```
    else {
     _edge.push_back(std::make_pair(fromNode, toNode));
     _edge.push_back(std::make_pair(toNode, fromNode));
     _edgeWeight[std::make_pair(fromNode, toNode)] = cost;
     _edgeWeight[std::make_pair(toNode, fromNode)] = cost;
    }
  }
}


void Graph::removeEdge(std::string fromNode, std::string toNode) {
  if(hasEdge(fromNode, toNode) && exist(fromNode) && exist(toNode)) {
    if(direct) {
      _edge.erase(std::find(_edge.begin(), _edge.end(),
                            std::make_pair(fromNode, toNode)));
      _edgeWeight.erase(std::make_pair(fromNode, toNode));
    }
    /** undirected graph */
    else {
      _edge.erase(std::find(_edge.begin(), _edge.end(),
                            std::make_pair(fromNode, toNode)));
      _edge.erase(std::find(_edge.begin(), _edge.end(),
                            std::make_pair(toNode, fromNode)));
      _edgeWeight.erase(std::make_pair(fromNode, toNode));
      _edgeWeight.erase(std::make_pair(toNode, fromNode));
    }
  }
}

void Graph::setWeight(std::string fromNode,
                      std::string toNode,
                      double cost) {
  if(exist(fromNode) && exist(toNode)) {
    if(direct) {
      _edgeWeight[std::make_pair(fromNode, toNode)] = cost;
    }
    /** undirected Graph */
    else {
      _edgeWeight[std::make_pair(fromNode, toNode)] = cost;
      _edgeWeight[std::make_pair(toNode, fromNode)] = cost;
    }
  }
}

void Graph::print(std::ostream& os) const {
  std::vector<std::string>::const_iterator V;
  std::vector<link>::const_iterator E;
  os << "Node : { ";
  for(V = _node.begin(); V != _node.end(); ++V) {
    os << *V;
```

```
    if(V + 1 != _node.end())
      os << " , ";
  }

  os << " }" << std::endl << "Edge : { " << std::endl;

  for(E = _edge.begin();  E != _edge.end(); ++E)
    os << "\t( "
       << E->first  << " , " << E->second
       << " ) "
       << " weight: " << _edgeWeight.at(*E) << std::endl;

  os << std::endl << "}" << std::endl;
}

std::list<std::string> Graph::adjacent(std::string v) const {
  std::list<std::string> adj;
  std::vector<link>::const_iterator E;
  for(E = _edge.begin(); E != _edge.end(); ++E) {
    if(E->first == v)
      adj.push_back(E->second);
  }
  return adj;
}

unsigned Graph::minRank() const {
  unsigned min;
  std::vector<std::string>::const_iterator v = _node.begin();
  min = rank(*v);

  for(v = _node.begin() + 1; v != _node.end(); ++v) {
    if(v != _node.end())
      if(rank(*v) < min)
        min = rank(*v);
  }
  return min;
}

unsigned Graph::maxRank() const {
  unsigned max;
  std::vector<std::string>::const_iterator v = _node.begin();
  max = rank(*v);

  for(v = _node.begin() + 1; v != _node.end(); ++v) {
    if(v != _node.end())
      if(rank(*v) > max)
        max = rank(*v);
  }
  return max;
}
```

```
bool Graph::hasNegativeWeigth() const {
  std::map<link, double>::const_iterator w;
  for(w = _edgeWeight.begin(); w != _edgeWeight.end(); ++w) {
    if(w->second < 0)
      return true;
  }
  return false;
}

void Graph::_generateHtmlPage() const {
  // html code
}


void Graph::_generateJavascriptPage() const {
  // javascript code
}


void Graph::draw() const {
  _generateHtmlPage();
  _generateJavascriptPage();
  /** execute default browser */
  system("xdg-open html/G.html &");
}


bool Graph::isCyclic() const {
  mapStringBool visited;
  mapStringBool recStack;
  std::vector<std::string>::const_iterator i;
  for(i = _node.begin(); i != _node.end(); ++i) {
    visited[*i] = false;
    recStack[*i] = false;
  }


  for(i = _node.begin(); i != _node.end(); ++i) {
    if(_isCyclicUtil(*i, visited, recStack))
      return true;
  }
  return false;
}

bool Graph::_isCyclicUtil(std::string v,
                          mapStringBool visited,
                          mapStringBool recStack) const {
  if(visited[v] == false) {
    /** Mark the current node as visited and part of recursion stack */
```

```
    visited[v]= true;
    recStack[v]= true;

    /** Recur for all the vertices adjacent to this vertex */
    for(auto i = adjacent(v).begin(); i != adjacent(v).end(); ++i) {
      if(!visited[*i] && _isCyclicUtil(*i, visited, recStack))
        return true;
      else if(recStack[*i])
        return true;
    }
  }
  /** remove the vertex from recursion stack */
  recStack[v]= false;
  return false;
}

void Graph::coloring() {
  Graph Gt = this->transpose();

  /** remove common edges */
  for(auto e = this->_edge.begin(); e != this->_edge.end(); ++e)
    Gt.removeEdge(e->first, e->second);

  /** temporaly turn graph into undirected (if not) */
  for(auto e = this->_edge.begin(); e != this->_edge.end(); ++e)
    this->addEdge(e->first, e->second, 1);

  std::map<std::string, int> result;
  /** Assign the first color to first vertex */
  result[*(_node.begin())] = 0;

  /** Initialize remaining V-1 vertices as unassigned */
  for(auto u = _node.begin() + 1; u != _node.end(); ++u)
    result[*u] = -1;  // no color is assigned to u

  /** Assign colors to remaining V-1 vertices */
  for(auto u = _node.begin() + 1; u != _node.end(); ++u) {
    /** Process all adjacent vertices and flag their colors
        as unavailable */
    std::list<std::string> adj = adjacent(*u);

    signed color = -1;
    bool found = false;
    while(!found) {
      ++color;
      found = true;
      for(auto v = adj.begin(); v != adj.end(); ++v) {
        if(color == result[*v]) {
          found = false;
          break;
```

```
        }
      }
    }
    /** Assign the found color */
    result[*u] = color;
  }

  for(auto u = _node.begin(); u != _node.end(); ++u)
    std::cout << "Vertex " << *u
              << " ---> Color " << result[*u] << std::endl;

  for(auto e = Gt._edge.begin(); e != Gt._edge.end(); ++e)
    this->removeEdge(e->first, e->second);
}

void Graph::_DFSUtil(std::string v, mapStringBool& visited) const {
  /** Mark the current node as visited and print it */
  visited[v] = true;
  /** Recur for all the vertices adjacent to this vertex */
  std::list<std::string>::iterator i;
  std::list<std::string> adj = adjacent(v);
  for(i = adj.begin(); i != adj.end(); ++i)
    if(!visited[*i])
      _DFSUtil(*i, visited);
}

void Graph::_DFSUtil2(std::string v, mapStringBool& visited) const {
  /** Mark the current node as visited and print it */
  visited[v] = true;
  std::cout << v << " ";

  /** Recur for all the vertices adjacent to this vertex */
  std::list<std::string>::iterator i;
  std::list<std::string> adj = adjacent(v);
  for(i = adj.begin(); i != adj.end(); ++i)
    if(!visited[*i])
      _DFSUtil2(*i, visited);
}

/**
  DFS traversal of the vertices reachable from v.
*/
void Graph::DFS(std::string sourceNode) {
  /** Mark all the vertices as not visited */
  mapStringBool visited;
  for(auto u = _node.begin(); u != _node.end(); ++u)
    visited[*u] = false;

  // Call the recursive helper function to print DFS traversal
  _DFSUtil2(sourceNode, visited);
```

```
}


/**
  Breadth First Traversal for a Graph
*/
void Graph::BFS(std::string sourceNode) {
  /** Mark all the vertices as not visited */
  mapStringBool visited;
  for(auto u = _node.begin(); u != _node.end(); ++u)
    visited[*u] = false;

  /** Create a queue for BFS */
  std::list<std::string> queue;

  /** Mark the current node as visited and enqueue it */
  visited[sourceNode] = true;
  queue.push_back(sourceNode);

  /** 'i' will be used to get all adjacent vertices of a vertex */
  std::list<std::string>::iterator i;

  while(!queue.empty()) {
    /** Dequeue a vertex from queue and print it */
    sourceNode = queue.front();
    std::cout << sourceNode << " ";
    queue.pop_front();

    /** Get all adjacent vertices of the dequeued vertex s
        If a adjacent has not been visited, then mark it visited
        and enqueue it */
    std::list<std::string>::iterator i;
    std::list<std::string> adj = adjacent(sourceNode);
    for(i = adj.begin(); i != adj.end(); ++i) {
      if(!visited[*i]) {
        visited[*i] = true;
        queue.push_back(*i);
      }
    }
  }
}

bool Graph::isConnected() const {
  /** Mark all the vertices as not visited */
  std::map<std::string, bool> visited;
  std::vector<std::string>::const_iterator u;
  for(u = _node.begin(); u != _node.end(); ++u)
    visited[*u] = false;

  /** Find a vertex with non-zero degree */
```

```
  for(u = _node.begin(); u != _node.end(); ++u)
    if(adjacent(*u).size() != 0)
      break;
  std::string strU = *u;
  unsigned lastNode = atoi(strU.c_str());
  /** If there are no edges in the graph, return true */
  if(lastNode == nodes())
    return true;

  /** Start DFS traversal from a vertex with non-zero degree */
  _DFSUtil(*u, visited);

  /** Check if all non-zero degree vertices are visited */
  for(auto v = _node.begin(); v != _node.end(); ++v)
    if(visited[*v] == false && adjacent(*v).size() > 0)
      return false;
  return true;
}

int Graph::isEulerian() const {
  /** Check if all non-zero degree vertices are connected */
  if(isConnected() == false) {
    std::cout << "not connected" <<std::endl;
    return 0;
  }

  /** Count vertices with odd degree */
  int odd = 0;
  for(auto v = _node.begin(); v != _node.end(); ++v)
    if(adjacent(*v).size() & 1)
      odd++;

  /** If count is more than 2, then graph is not Eulerian */
  if(odd > 2)
    return 0;

  /** If odd count is 2, then semi-eulerian.
      If odd count is 0, then eulerian
      Note that odd count can never be 1 for undirected graph */
  return (odd)? 1 : 2;
}


int** Graph::fromListADJToMatrixADJ() {
  /** matrix allocation */
  int** ADJMatrix = new int*[nodes()];
  for(unsigned i = 0; i < nodes(); ++i)
    ADJMatrix[i] = new int[nodes()];

  /** initialize matrix */
```

```
  for(auto i = _node.begin(); i != _node.end(); ++i)
    for(auto j = _node.begin(); j != _node.end(); ++j)
      ADJMatrix[atoi((*j).c_str())][atoi((*i).c_str())] = 0;

  for(auto j = _node.begin(); j != _node.end(); ++j) {
    std::list<std::string>::iterator k;
    std::list<std::string> adj = adjacent(*j);
    for(k = adj.begin(); k != adj.end(); ++k) {
      if(direct)
        ADJMatrix[atoi((*k).c_str())][atoi((*j).c_str())] = 1;
      else {
        ADJMatrix[atoi((*j).c_str())][atoi((*k).c_str())] = 1;
        ADJMatrix[atoi((*k).c_str())][atoi((*j).c_str())] = 1;
      }
    }
  }

  return ADJMatrix;
}


/**
  @return matrix of edge's weight
*/

double** Graph::weightMatrix() {
  /** matrix allocation */
  int i, j, k;
  double** wMatrix = new double*[nodes()];
  for(unsigned i = 0; i < nodes(); ++i)
    wMatrix[i] = new double[nodes()];

  /** initialize matrix */
  for(auto ii = _node.begin(); ii != _node.end(); ++ii) {
    i = atoi((*ii).c_str());
    for(auto jj = _node.begin(); jj != _node.end(); ++jj) {
      j = atoi((*jj).c_str());
      if(*ii == *jj)
        wMatrix[j][i] = 0;
      else
        wMatrix[j][i] = INF;
    }
  }

  for(auto jj = _node.begin(); jj != _node.end(); ++jj) {
    j = atoi((*j).c_str());
    std::list<std::string>::iterator kk;
    std::list<std::string> adj = adjacent(*jj);

    for(kk = adj.begin(); kk != adj.end(); ++kk) {
```

```
      k = atoi((*kk).c_str());
      if(direct) {
        wMatrix[k][j] = weight(*jj, *kk);
      }
      else {
        wMatrix[j][k] = weight(*jj, *kk);
        wMatrix[k][j] = weight(*jj, *kk);
      }
    }
  }
  return wMatrix;
}

/**
  Solves the all-pairs shortest path problem using
  Floyd Warshall algorithm
*/
void Graph::floydWarshell(double** graph) {

  int i, j, k;
  int** dist = new int*[nodes()];
  for(unsigned i = 0; i < nodes(); ++i)
    dist[i] = new int[nodes()];

  for(auto ii = _node.begin(); ii != _node.end(); ++ii) {
    i = atoi((*ii).c_str());
    for(auto jj = _node.begin(); jj != _node.end(); ++jj) {
      j = atoi((*jj).c_str());
      dist[i][j] = graph[i][j];
    }
  }

  for(auto kk = _node.begin(); kk != _node.end(); ++kk) {
    /** Pick all vertices as source one by one */
    k = atoi((*kk).c_str());
    for(auto ii = _node.begin(); ii != _node.end(); ++ii) {
      i = atoi((*ii).c_str());
      for(auto jj = _node.begin(); jj != _node.end(); ++jj) {
        j = atoi((*jj).c_str());
        if(dist[i][k] + dist[k][j] < dist[i][j])
          dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }

  // Print the shortest distance matrix
  _printSolutionFloydWarshell(dist);
}

/**
```

```
  A utility function to print solution
*/
void Graph::_printSolutionFloydWarshell(int** dist) {
  std::cout << "Following matrix shows the shortest distances"
            << std::endl << " between every pair of vertices \n";
  int i, j;
  for(auto ii = _node.begin(); ii != _node.end(); ++ii) {
    i = atoi((*ii).c_str());
    for(auto jj = _node.begin(); jj != _node.end(); ++jj) {
      j = atoi((*jj).c_str());
      if(dist[i][j] == INF)
        std::cout << "INF\t";
      else
        std::cout << dist[i][j] << "\t";
    }
    std::cout << std::endl;
  }
}
```

2.1 Utility library

The utility library is necessary for the Grahlib library

```
namespace utility {

/**
  Convert to string
  @param n type to convert to string
*/
template<typename T> std::string to_string(const T& n) {
  std::ostringstream stm;
  stm << n;
  return stm.str();
}

/**
  Transform string 2-4 to pair <2, 4>
  @param s string to transform
  @return pair of int
*/
std::pair<int, int> interval(std::string s) {
  // tokenize the string
  int i = s.length() - 1;
  unsigned power = 0;
  int toInt = 0;
  int fromInt = 0;

  while(s.at(i) != '-') {
    toInt += (s.at(i--) - 48) * pow(10, power++);
  }
  power = 0;
```

```
  --i;
  while(i != -1) {
    fromInt += (s.at(i--) - 48) * pow(10, power++);
  }
  return std::make_pair(fromInt, toInt);
}

/**
  a-d -> [a, b, c, d]  | A-Z  -> [A, B, C, ..., Z]
  @param s string to parse
  @return vector of char
*/
std::vector<char> regexChar(std::string s) {
  std::vector<char> tmp;
  int head = s.at(0);
  int tail = s.at(2);

  while(head != tail) {
    tmp.push_back(head++);
  }
  tmp.push_back(head);
  return tmp;
}

/**
  1-4 -> [1,2,3,4]   34-101  -> [34,35,36,...,100,101]
  @param s string to parse
  @return vector of int
*/
std::vector<int> regexInt(std::string s) {
  // generate the number from (fromInt) to (toInt)
  std::vector<int> tmp;
  std::pair<int , int> intval(interval(s));
  for(int i = intval.first; i <= intval.second; ++i) {
    tmp.push_back(i);
  }
  return tmp;
}

/**
  Check of s is a valid interval
  @param s interval to check
  @return bool
*/
bool checkIfInterval(std::string s) {
  std::pair<int, int> intval(interval(s));
  std::string fromInt = to_string(intval.first);
  std::string toInt = to_string(intval.second);
  std::string::const_iterator fromIt = fromInt.begin();
  std::string::const_iterator toIt = toInt.begin();
```

```
  while(fromIt != fromInt.end() && std::isdigit(*fromIt)) ++fromIt;
  while(toIt != toInt.end() && std::isdigit(*toIt)) ++toIt;
  return (!fromInt.empty() && fromIt == fromInt.end()) &&
         (!toInt.empty() && toIt == toInt.end());
}

/** namespace utility */
}
```

References

[1] J. Kleinberg, E. Tardos, Algorithm Design, Addison-Wesley, 2005;

[2] P. Crescenzi, G. Gambosi, Strutture di dati e algoritmi, Pearson, 2012, (seconda edizione);

[3] http://www.geeksforgeeks.org/;