Graph-lib Documentation

Rostislav Fateev

Abstract

graph-lib is a C++ graph library currently aimed at implementation of algorithms used to solve network flow problem: for a directed graph with given edge capacities find maximum possible to send flow from source vertex to sink vertex.

Task

- Design and implement graph and related foundational entities.
- Design and implement algorithms required to solve maximum network flow problem.
- Design and implement support program allowing fast and easy algorithm tests on graphs provided from file.

Algorithms

The following algorithms were implemented:

- Depth First Search - core graph traversal algorithm
  - Recursive DFS - simpliest implementation, which calls itself on undiscovered children.
  - Stack-based DFS - classic approach, uses stack to store potential undiscovered candidates.
- Breadth First Search - core graph traversal algorithm
  - Queue-based BFS - classic approach, uses queue to store potential udiscovered candidates.
  - Flow-related BFS - implementation used in Edmonds-Karp algorithm, which takes into account not only "is vertex visited" condition, but also "can edge pass through more flow" condition.

- Edmonds-Karp - one of the approaches to solve the maximum flow problem. Works by repeatedly finding the augmenting path in a graph and sending maximum allowed by edge capacity flow along it until no more path can be found.
- Dinic - more complicated, but also efficient approach. Works by repeatedly constructing a residual network - network consisting of edges, that can still pass through flow - and trying to find for each such network all possible augmenting paths to send flow along.

Program

graph-lib

Library consists of multiple modules, logically splitted to provide different functionality.

- **component** - module containing vertex and edge implementations.
  - **Vertex** - generic class representing an elementary graph theoretical entity. It is able to use identificators of various types.
  - **Edge** - generic class parametrized by Vertex identification type representing a connection between two Vertex objects. It is a six-tuple containing:
    - first - reference to the first Vertex of a connection.
    - second - reference to the second Vertex of a connection.
    - direction - specifies if connection is directed or not (by default undirected)
    - weight - specifies connection weight (by default 0)
    - capacity - specifies connection flow capacity (by default 0)
    - flow - specifies current flow passing through the connection (by default 0)
- **algorithm** - module containing all implemented algorithms. Each algorithm contains similar interface to simplify usage: constructor takes pointer to graph object as input, $run(...)$ method executes algorithm after object was constructed and $get()$ method outputs the result.
  - **DFS** - generic class parametrized by Vertex identification type encapsulating DFS algorithm behaviour. Outputs dictionary with $< Vertex, Vertex >$ entries indicating traversal order.
  - **BFS** - generic class parametrized by Vertex identification type encapsulating BFS algorithm behaviour. Outputs dictionary with $< Vertex, Vertex >$ entries indicating traversal order.

  - **Edmonds** - generic class parametrized by Vertex identification type encapsulating Edmonds-Karp algorithm behaviour. Outputs maximum possible flow that can be sent over graph's network.
  - **Dinic** - generic class parametrized by Vertex identification type encapsulating Dinic algorithm behaviour. Outputs maximum possible flow that can be sent over graph's network.

- **implementation** - module containing underlying graph functionality implementations.
  - **AdjacencyList** - generic class parametrized by Vertex identification type that implements adjacency list data structure. It is a dictionary with $< Vertex, list < Edge >>$ entries indicating all of graph's vertices and their respective list of neighbouring vertices reachable by outgoing edge. Has the same interface as Graph object.

- **display** - module containing functionality to display graph on screen.
  - **Outputter** - generic class parametrized by Vertex identification type that for an input graph displays it on the screen using Drawer and Positioner components.
  - **Drawer** - class that is resonsible for drawing Graph object or algorithm result on the screen using SFML library and coordinates computed by Positioner component.
  - **Positioner** - class that is responsible for computing vertex coordinates used by Drawer component using different techniques.

- **Graph** - generic class wrapper parametrized by Vertex identification type and implementation strucure type. It is a wrapper over implementation structure, which encapsulates actual functionality, providing a common interface using Bridge design pattern. Allows vertex and edge manipulation, quick access to vertex neighbours and quick acces to implementation structure iterators to allow bulk operations.

For details see README.md file and documentation generated in doc directory.

runner

A helper tool, which uses **graph-lib** and allows to execute algorithms on graphs provided as file input. Consists of the following modules:

- **fetcher** - module reading a graph from the input file and constructing a Graph object.

  - **Fetcher** - class which opens a file stream and parses input file constructing Graph object.

- **runner** - module executing implemented algorithms on constructed by Fetcher Graph object and outputs the result

  - **Runner** - class which launches loop, parses input commands and executes them until "quit" command is received.

For details see README.md file.

Input data

runner

File format

- Each vertex/edge is on a separate line.
- Vertices are separated from edges by a line containing $---$.
- Vertices are put before edges.
- Edge has the following strucutre <vertex1 vertex2 direction weight capacity flow>.
  - vertex1 and vertex2 data fields are required, the others are optional. Direction can be either $---$ or $--->$. If you want to specify let's say edge capacity - you must also specify all fields before it (direction and weight). Default values for each data field is:
    - direction: $---$
    - weight: 0
    - capacity: 0
    - flow: 0
  - Edge data fields are delimeted by a single space character.

Improvements

I would consider the following as possible improvements:

- Remove parametrization by implementation structure from Graph object as there is no need to distinguish $Graph < int, AdjacencyList >$ and $Graph < int, IncidenceMatrix >$.
- Add more implementation structures.

- Add Push-Relabel algorithm - one of the most efficient known techniques.
- Extend Edmonds-Karp and Dinic to allow multiple sources and sinks.
- Add algorithms for min-cost flow problem.
- Fix drawing of directed edges in Drawer component.
- Beautify drawing of algorithm results in Drawer component.

Last words

Graph theory remains an important part of mathematics and every day life, while network flow problems occupy an important place in it, hence the importance of providing a framework enabling solving problems in one of the most efficient languages in the world cannot be overstated.