graphe

Morten Jagd Christensen

Jun 28, 2023

PRACTICAL

graphe is a Python library for creating, traversing and visualising graphs. Most algorithms are inspired by the Algorithms Course[1] by Robert Sedgewick. The Python classes are reimplementations of Java code.

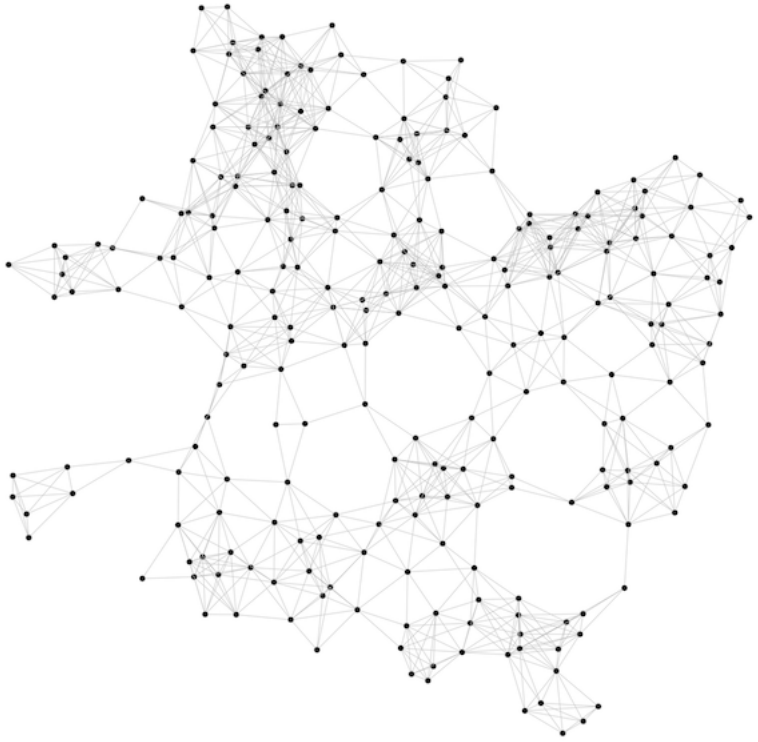


Fig. 1: Graphviz visualisation of an undirected graph loaded from 'mediumG.txt', courtesy R. Sedgewick, Princeton.

Note: This is a project under unpredictable development. Class names and code organisation is subject to (breaking) changes.

[1] Algorithms - https://algs4.cs.princeton.edu

CHAPTER

ONE

INSTALL

1.1 Installation

To use graphe, first install it using pip:

```
(.venv) $ pip install graphe
```

1.2 Local Install

To install locally from source

```
$ pip install .
```

CHAPTER

TWO

GRAPH

class Graph(*init*)

Creates a Graph object

Parameters

init – if integer (V) initialize empty Graph with V vertices. If string (filename) load and populate from file.

add_edge(*v*, *w*)

Connects vertices v and w, both must be smaller than V

Parameters

- **v** – vertice id
- **w** – vertice id

adj(*v*)

Return a list of vertices adjacent to v

Parameters

v – vertice id

Return type

array of vertice ids

to_string()

Create a string representation of the Graph

Return type

string

2.1 BFSearch

class BFSearch(*G*, *s*)

Does a breadth first search of *G* from *s*.

Parameters

- **G** – Graph object
- **s** – Vertex id of the starting point for search

`bfs`(*G*, *s*)

Performs the breadth first search - called from constructor, should *not* be called directly

Parameters

- `G` – Graph object created by Graph or SymbolGraph
- `s` – Vertex id of the starting point for search

`has_path_to`(*v*)

Does (G,s) have a path to vertex v?

Parameters
`v` – Vertex id

Return type
Boolean

`path_to`(*v*)

Make one path to v from s

Parameters
`v` – Vertex id

Return type
array of vertex ids connecting v to s

`count`()

Number of visited nodes when exploring (G, s)

Return type
number of visited nodes

2.2 DFSearch

`class DFSearch`(*G*, *s*)

Does a depth first search of *G* from *s*.

Parameters

- `G` – Graph object
- `s` – Vertex id of the starting point for search

`dfs`(*G*, *v*)

Performs the depth first search - called from constructor, should *not* be called directly

Parameters

- `G` – Graph object created by Graph or SymbolGraph
- `v` – Vertex id of the starting point for search

`has_path_to`(*v*)

Does (G,s) have a path to vertex v?

Parameters
`v` – Vertex id

Return type
Boolean

`path_to`(*v*)

Make one path to v from s

Parameters
v – Vertex id

Return type
array of vertex ids connecting v to s

`count`()

Number of visited nodes when exploring (G, s)

Return type
number of visited nodes

CHAPTER

THREE

GRAPH APPLICATIONS

These are examples of applications that uses graphs for their implementation.

CHAPTER

FOUR

DIGRAPH CLASSES

4.1 Digraph

class Digraph(*init*)

Creates a Digraph object

Parameters
init – if integer (V) initialize empty Digraph with V vertices. If string (filename) load and populate from file.

add_edge(*v*, *w*)

Connects vertices v and w, both must be smaller than V

Parameters

- **v** – vertice id
- **w** – vertice id

adj(*v*)

Return a list of vertices adjacent to v

Parameters
v – vertice id

Return type
array of vertice ids

reverse()

Return the reversed Digraph

Return type
Digraph

to_string()

Create a string representation of the Digraph

Return type
string

4.2 Directed Cycle

Detects cycles in Digraphs

class DirectedCycle(*DG*)

Parameters
DG – Digraph object

has_cycle()

Return type
boolean

get_cycle()

Return type
list of vertices in the cycle

4.3 DepthFirstOrder

Performs a depth first search with support for returning visited nodes in pre-order, post-order and reverse post-order.

class DepthFirstOrder(*DG*)

Parameters
DG – Digraph object

get_pre()

Return type
vertice list in pre-order

get_post()

Return type
vertice list in post-order

get_reverse_post()

Return type
vertice list in reverse post-order

4.4 DirectedDFSearch

class DirectedDFSearch(*DG*, *s*)

Does a depth first search of *Digraph* from *s*.

Parameters

- **DG** – Digraph object
- **s** – Vertex id of the starting point for search

dfs(*DG*, *v*)

Performs the depth first search - called from constructor, should *not* be called directly

Parameters

- **DG** – Digraph object
- **v** – Vertex id of the starting point for search

has_path_to(*v*)

Does (DG, s) have a path to vertex v?

Parameters
v – Vertex id

Return type
Boolean

path_to(*v*)

Make one path to v from s

Parameters
v – Vertex id

Return type
array of vertex ids connecting v to s

count()

Number of visited nodes when exploring (Digraph, s)

Return type
number of visited nodes

4.5 Regex

An implementation of regular expressions. Constructs a NFA Digraph of the regular expression, then simulates the NFA using repeated depth first searches.

class Regex(*expression*)

Parameters
expression – regular expression

`match`(*text*)

Parameters
`text` – text string to be matched against the regex

Return type
boolean

4.6 SymbolDigraph

Support Digraph objects with named edges.

`class SymbolDigraph`(*filename*)

Parameters
`filename` – file to read

`graph`()

Return type
Digraph object

`node_names`()
List of node names corresponding to vertice ids

Return type
array of node names

4.7 Topological Sort

Performs topological sort. Since this can only work on DAGs this code raises an exception of a cycle is found.

`class Topological`(*DG*)

Parameters
`DG` – Digraph

`get_order`()

Return type
vertices in topological order

CHAPTER

FIVE

DIGRAPH APPLICATIONS

These are examples of applications that uses Digraph algorithms for their implementation.

5.1 Regex

Use non deterministic finite state automaton (NFA) simulation to implement minimalistic regular expressions. The supported regex commands are

```
contatenation    AB
grouping         (ABC)
or               A|B
closure          A*B
wildcard         A.B
```

Regex does not support common regex features such as ranges [], repeats {}, '?' or '+' as these are not fundamental

```
[A-E]*  -> (A|B|C|D|E)*
(AB){3} -> ABABAB
(AB)+   -> AB(AB)*
(CD)?   -> (|(CD))
```

Examples

```
CGC(CCG|CAG)*AGT        Genomic repeats
.*ion                   'ending' in ion (crossword)
.*Needle.*              search for Needle (grep)
(0|(1(01*(00)*0)*1)*)*  binary numbers which are a multiple of 3
```

class Regex(*regex*)

Construct the NFA Digraph of epsilon transitions for the regular expression.

match(*text*)

matches text against the given regexp by repeatedapplication of DirectedDFSearch to explore the reachable states.

Parameters
text – text string to match with regex

Return type
boolean

5.2 Topological Search

Finds one solution (of potentially several) to the 'scheduling' problem where some tasks must be preceded by others.

CHAPTER

SIX

UTILITIES

6.1 Draw

class Draw(*digraph=False*)

Prepares for drawing a graph/digraph. Creates a graphviz object and sets the initial graph attributes

set_names(*names*)

Provide figure with names (from SymbolGraph) instead of ids

Parameters
names – array [] of names for each vertex id

get_name(*v*)

Use when drawing the figure, should not normally be called direcctly.

Parameters
v – vertex id

Return type
string

node_attr(***kwargs*)

Set graphviz attributes for nodes

Parameters
****kwargs** – List of graphviz keywords (e.g. color='black')

edge_attr(***kwargs*)

Set graphviz attributes for edges

Parameters
****kwargs** – List of graphviz keywords (e.g. penwidth='0.75')

draw(*G*, *path=[]*)

Draws the graph using the configured attributes and, optionally, showing the provided path

Parameters

- **G** – Graph object
- **path** – list of vertices on the path

CHAPTER

SEVEN

FILE FORMATS

This section describes the formats used for loading graphs from file.

7.1 Graph/Digraph

The format consist of two lines with number of vertives (V) and edges (E) followed by E lines of edges between vertices. The format is the same for Graph and Digraph.

```
$ cat tinyG.txt
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

7.2 SymbolGraph

This format simply consists of lines containing named edges. In the below example the space character ' ' is used as separator.

```
$ cat routes.txt
JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
ORD DFW
ORD PHX
```

(continues on next page)

(continued from previous page)

```
ATL HOU
DEN PHX
PHX LAX
JFK ORD
DEN LAS
DFW HOU
ORD ATL
LAS LAX
ATL MCO
HOU MCO
LAS PHX
```

7.3 SymbolGraphII

This format simply consists of lines containing named edges.

The below example has multiple nodes per line and is using the '/' as separator.

A line like

    A/B/C/D

Is interpreted as adding three edges to the graph: A->B, A->C, A->D

```
$ cat jobs.txt
Algorithms/Theoretical CS/Databases/Scientific Computing
Introduction to CS/Advanced Programming/Algorithms
Advanced Programming/Scientific Computing
Scientific Computing/Computational Biology
Theoretical CS/Computational Biology/Artificial Intelligence
Linear Algebra/Theoretical CS
Calculus/Linear Algebra
Artificial Intelligence/Neural Networks/Robotics/Machine Learning
Machine Learning/Neural Networks
```

CHAPTER

EIGHT

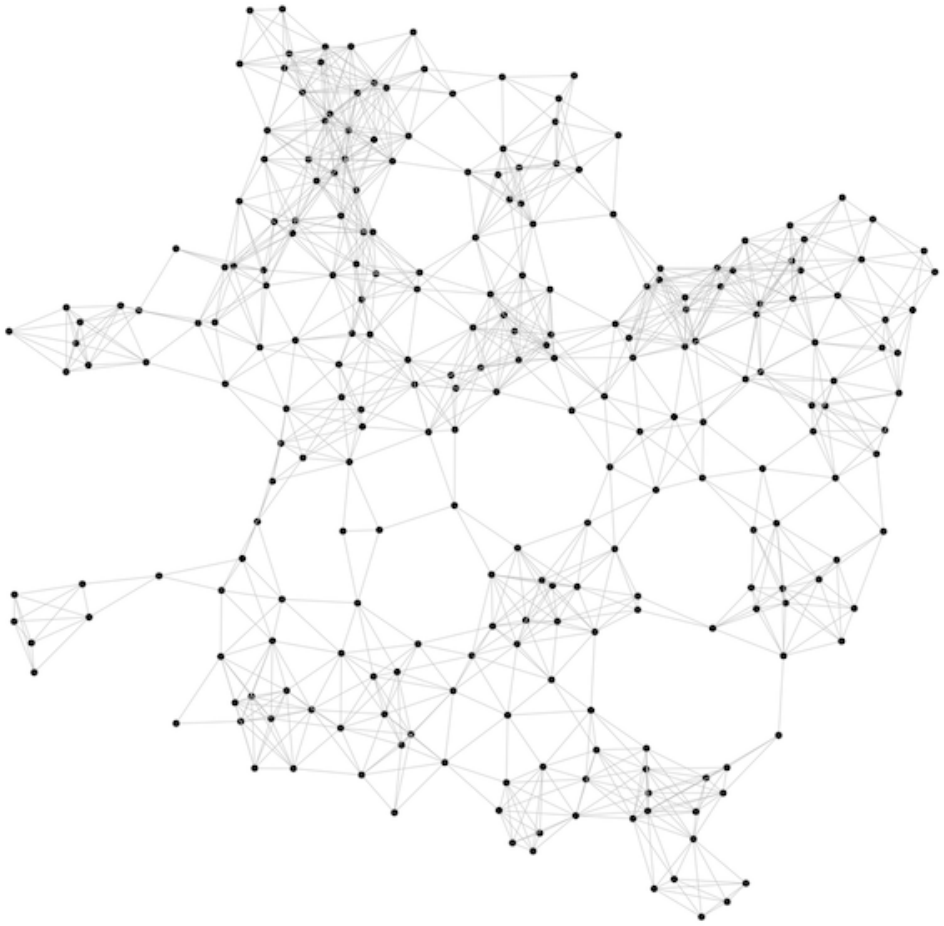
USAGE

8.1 Creating a Graph object

```
from graphe.graph import graph
from graphe import draw

G = graph.Graph('mediumG.txt')

fig = draw.Draw()
fig.node_attr(label='')
fig.draw(G)
```



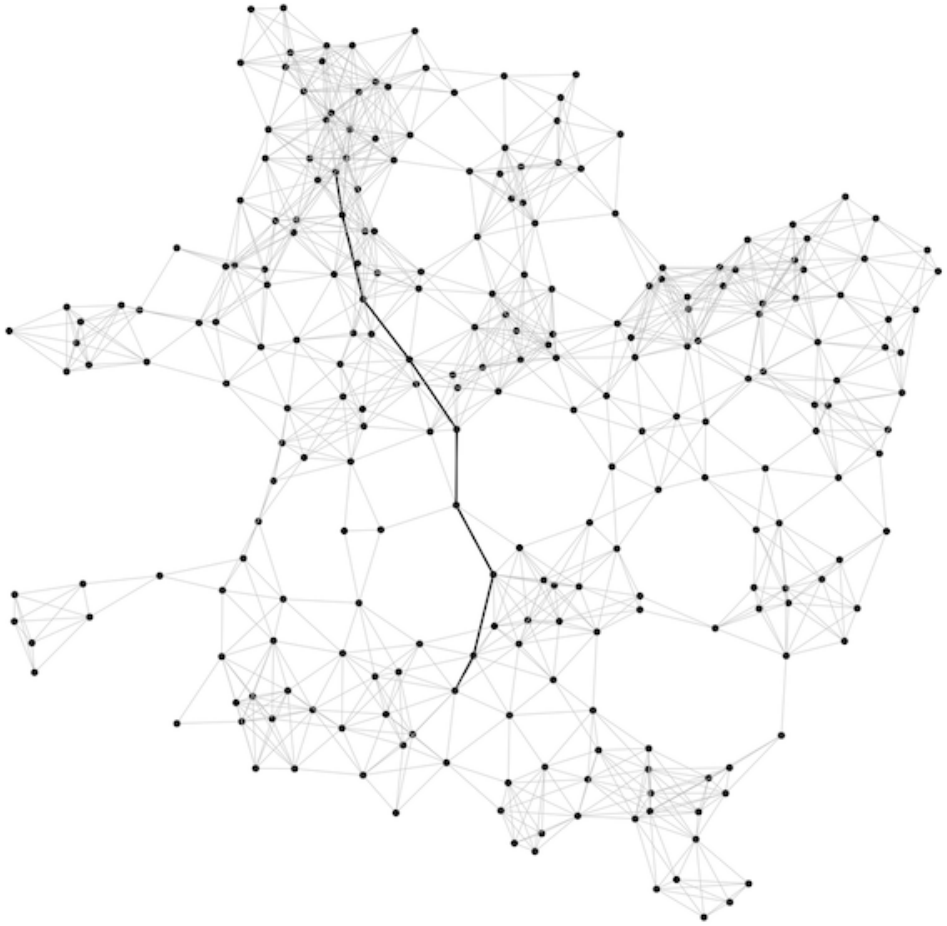
8.2 Breadth-first search

```
from graphe.graph import graph
from graphe.graph import bfs
from graphe import draw

G = graph.Graph('mediumG.txt')

bfs = bfs.BFSearch(G, 0)  # make tree with root on vertex 0
bfpath = bfs.path_to(200) # find path to vertex 200 from 0

fig = draw.Draw()
fig.node_attr(label='')
fig.draw(G, bfpath)
```



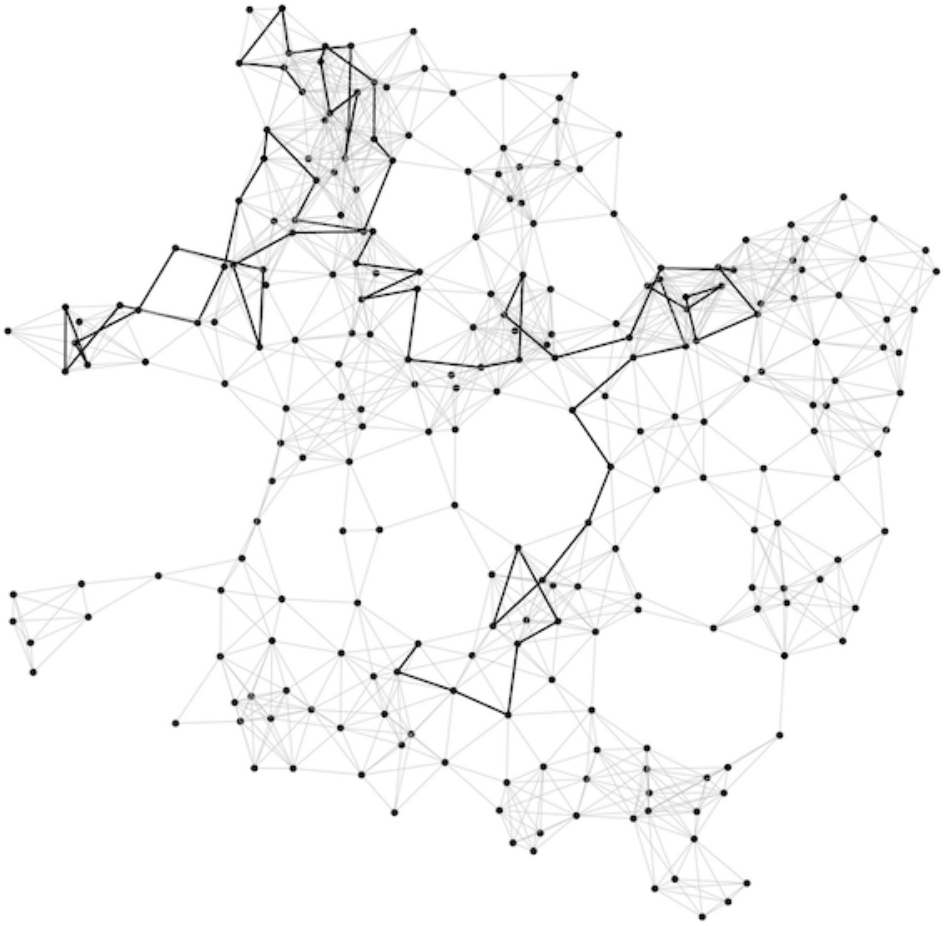
8.3 Depth-first search

```
from graphe.graph import graph
from graphe.graph import dfs
from graphe import draw

G = graph.Graph('mediumG.txt')

dfs = dfs.DFSearch(G, 0)
dfpath = dfs.path_to(200)

fig = draw.Draw()
fig.node_attr(label='')
fig.draw(G, dfpath)
```



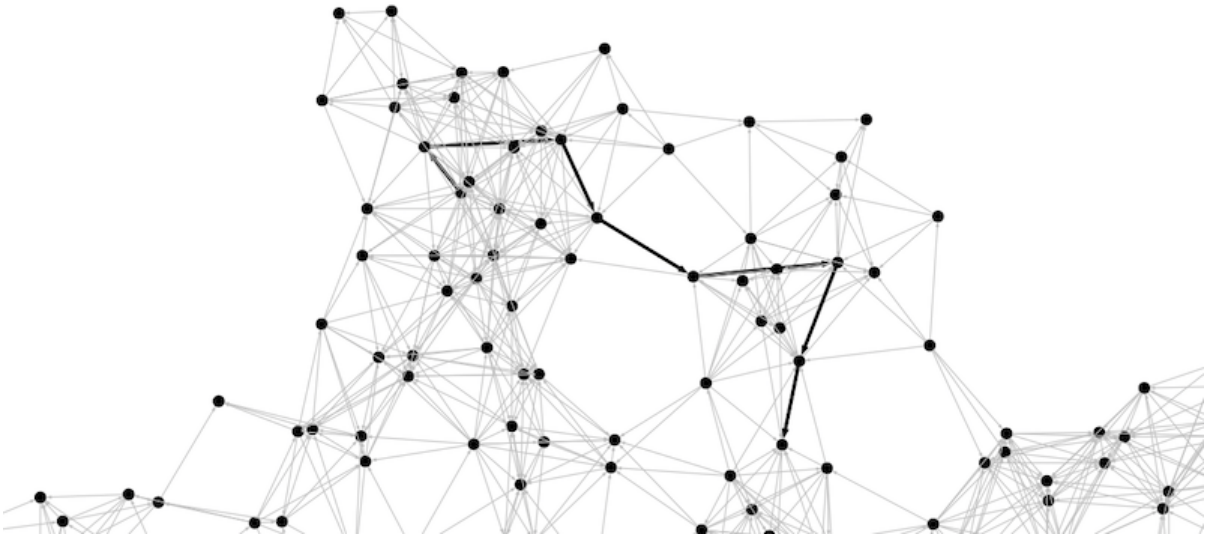
8.4 Directed Depth-first search

```
from graphe.digraph import digraph
from graphe.digraph import digraphdfs
from graphe import draw

DG = digraph.Digraph('mediumG.txt')

dfs = digraphdfs.DirectedDFSearch(DG, 0)
dfpath = dfs.path_to(197)

fig = draw.Draw(digraph=True)
fig.node_attr(label='')
fig.edge_attr(color='gray', arrowsize='0.2', penwidth='0.75')
fig.draw(DG, dfpath)
```

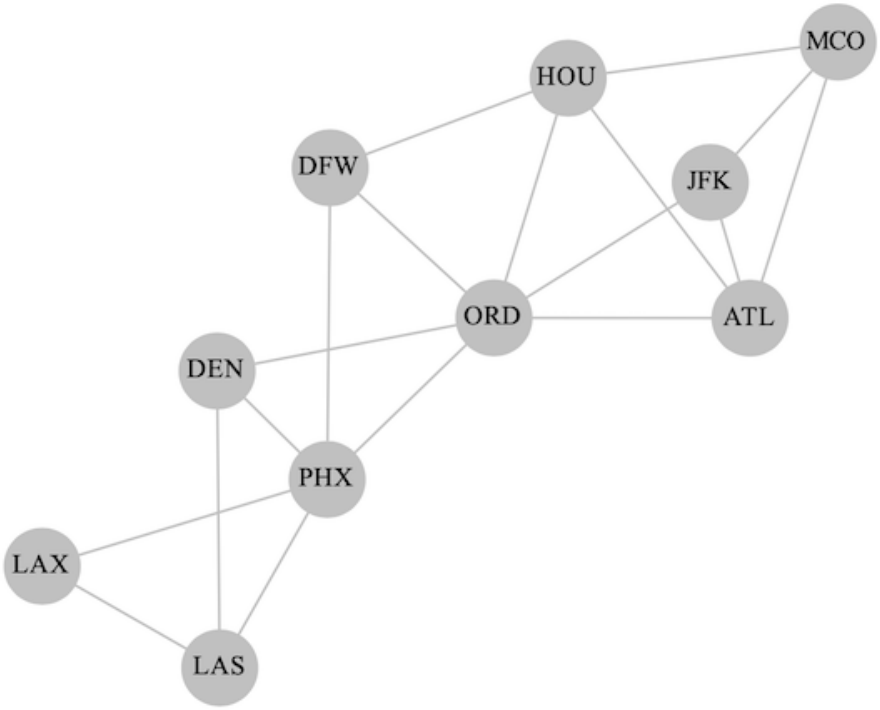


8.5 SymbolGraph

```
from graphe.digraph import symboldigraph
from graphe import draw

SG = symbolgraph.SymbolGraph('routes.txt')

fig = draw.Draw()
fig.set_names(SG.node_names())
fig.node_attr(width='0.3', height='0.3', shape='circle', style='filled',
              color='gray', fontcolor='black', fontsize='8')
fig.draw(SG.graph())
```

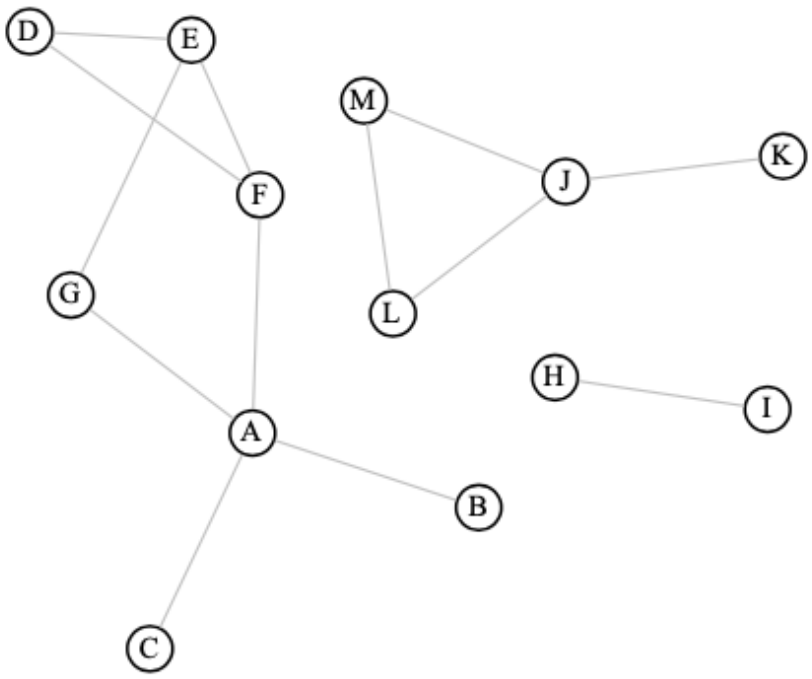


When plotting you can manually add node name

```
node_names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M']

G = graph.Graph('tinyG.txt')

fig = draw.Draw()
fig.set_names(node_names)
fig.node_attr(style='', fontcolor='black', fontsize='10')
fig.draw(G)
```

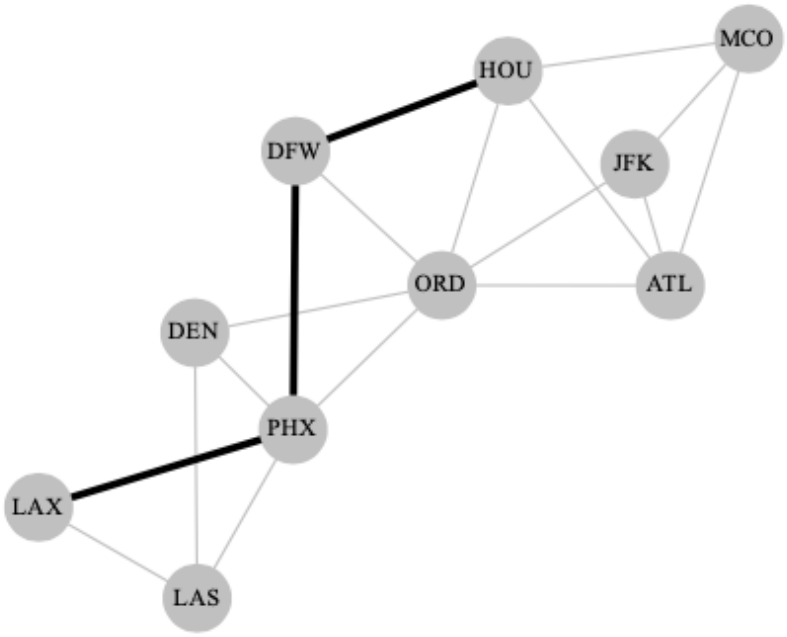


And you can do breadth first search on SymbolGraph

```
SG = symbolgraph.SymbolGraph('routes.txt')

b = bfs.BFSearch(SG.graph(), SG.ST['LAX'])
path = b.path_to(SG.ST['HOU'])

fig = draw.Draw()
fig.set_names(SG.node_names())
fig.node_attr(width='0.3', height='0.3', shape='circle', style='filled',
              color='gray', fontcolor='black', fontsize='8')
fig.draw(SG.graph(), path)
```

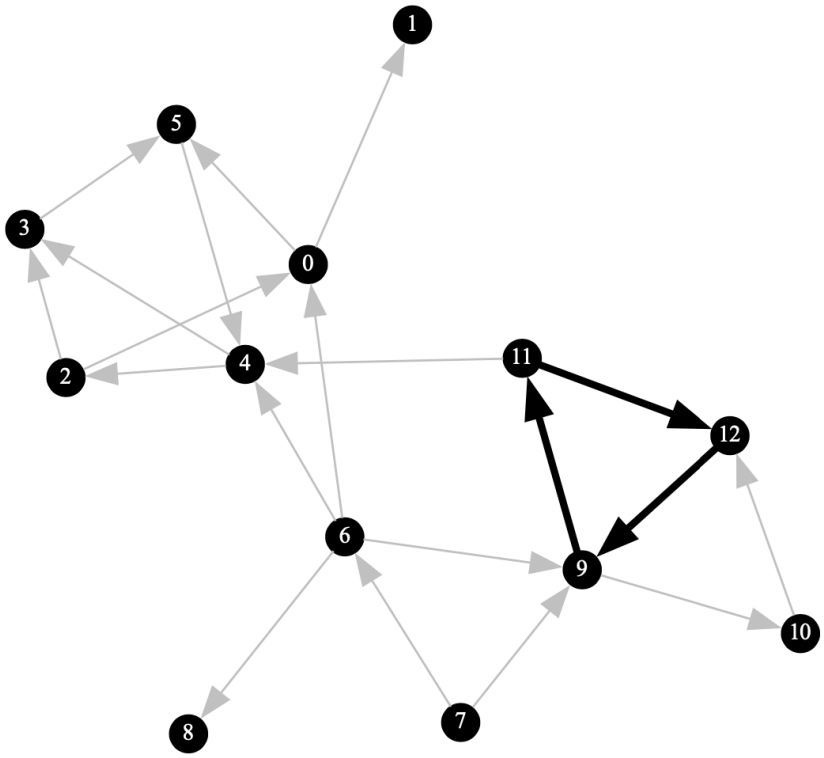


8.6 Digraph

```
from graphe.digraph import digraph
from graphe import draw

DG = digraph.Digraph('tinyDG.txt')

fig = draw.Draw(digraph=True)
fig.node_attr(fontsize='8')
fig.draw(DG, [11, 12, 9, 11])
```


1
5
3
0
2
4
11
12
6
9
10
8
7

INDEX