Automated Test Mapping and Coverage for Network Topologies – Source Code

P. E. Strandberg

Summer 2018

1 About

This text describes the source code accompanying "Automated Test Mapping and Coverage for Network Topologies" by P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and W. Afzal, accepted for publication at ISSTA 2018. The software was developed at Westermo Research and Development AB and should be read in conjunction with this paper. The source code in this bundle explores the subgraph isomorphism problem.

2 License

Copyright (C) 2018 P. E. Strandberg, Westermo Research and Development AB

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

3 Maintenance

This software may or may not be maintained by the authors. Do not get upset if there are no bug-fixes or enhancements to the software!

4 Code walk-through

There are five source code files, all written for Python 2.7.

4.1 network.py

This file contains two classes and unit tests implemented as doctests. The classes are `GraphException` and `Graph`.

A reader might ask: Why not just use a graph from a standard graph-library such as networkx? We opted for a minimal class developed in house in order to avoid additional dependencies.

- GraphException: A class for exceptions related to Graphs, raised for example when a node is added twice to the same Graph.
- Graph: A class for generating graphs. You can add nodes and edges, and investigate neighbours. You can get the degree sequence and adjacency matrix of a Graph, and so on.

  A Graph can also export itself to dot-format, so that pretty figures can be generated with Graphviz.

4.2 tree.py

This file contains the class `Tree` and unit tests in the form of doctests. The tree contains some data in the node, and children (other trees).

The purpose of the tree is to represent mappings: such as a failed mapping, a completed mapping, or several mappings. By traversing from the root node to a leaf such a mapping is identified.

Again: export to dot format is possible (and encouraged!).

4.3 testdata.py

This file contains a class for test data, in short: a bunch of graphs that can be used to try out the mapper.

4.4 mapper.py

This file contains the class `Mapper`, to be used for mapping a graph onto another graph. This is the whole purpose of the paper. There are shortcomings in all algorithms, so feel free to extend, improve and reject this class.

4.5 demo.py

The demo maps the graph H4 onto the graph G0 – this is the same example as in the paper. If you have a hard time following the example, just read the paper and look at the pictures.

5 Example

This is a walk-through of the second demo, `demo2.py`. First we import the mapper and the graph classes:

```
from mapper import Mapper
from network import Graph
```

Now we create a simple graph G with only five nodes in a ring.

```
G = Graph()
G.add_node(41)
G.add_node(42)
G.add_node(43)
G.add_node(44)
G.add_node(45)

G.add_edge(41, 42)
G.add_edge(42, 43)
G.add_edge(43, 44)
G.add_edge(44, 45)
G.add_edge(45, 41)
```

Next we create the graph H with only three nodes in a bus.

```
H = Graph()
H.add_node("A")
H.add_node("B")
H.add_node("C")

H.add_edge("A", "B")
H.add_edge("B", "C")
```

We map it once and print the nodes that were used. The expected output is: “In the first mapping, we used: [41, 42, 45]”.

```
M1 = Mapper(G, H, find_all=False)
(hmap, gmap, _) = M1.solutions[-1]

print "In the first mapping, we used: %s" % sorted(gmap)
```

Now we iterate a few times to map the same graph H onto the same graph G, while remembering previous mappings and avoiding them.

```
old_ones = list()
old_ones.append(tuple(M1.solutions[-1]))
```

```
iterations = 7

while 0 < iterations:
    iterations -= 1
    M = Mapper(G, H, find_all=False, check_old=True, old_ones=old_ones)
    old_ones.append(tuple(M.solutions[-1]))
```

OK, now we have mapped H onto G a couple of times, lets look at how the mapper mapped:

```
g_used = dict()

for old_one in old_ones:
    (_, gmap, _) = old_one

    for node in gmap:
        if node not in g_used:
            g_used[node] = 0
        g_used[node] += 1

print "In total we used the nodes in G this many times:"
for node in G.get_nodes():
    print "  node %s was used %s times" % (node, g_used[node])
```

The expected output of the last example is: “In total we used the nodes in G this many times:

- node 41 was used 5 times
- node 42 was used 6 times
- node 43 was used 5 times
- node 44 was used 4 times
- node 45 was used 4 times”

6 About Westermo

Westermo designs and manufactures data communications products for mission-critical systems in physically demanding environments. The products are used both in social infrastructure, such as transport, water and energy supplies, as well as in process industries, such as mining and petrochemical. Westermo was founded in 1975, currently has 200+ employees and spend 14% on R&D. Westermo has sales and support offices in 11 countries and sales partners in many others.

The software testing process at Westermo includes continuous integration and automated nightly testing on a system level. On a typical night thousands of test cases are automatically executed, on more than 15 test systems, many software versions and more than 30 different hardware variants.

Read more about Westermo at https://www.westermo.com