Prioritization of Mutation Test Generation with Centrality Measure

Supachai Supmak, Yachai Limpiyakorn

***Abstract*—**Mutation testing can be applied for the quality assessment of test cases. Prioritization of mutation test generation has been a critical element of the industry practice that would contribute to the evaluation of test cases. The industry generally delivers the product under the condition of time to the market and thus, inevitably sacrifices software testing tasks, even though many test cases are required for software verification. This paper presents an approach of applying a social network centrality measure, PageRank, to prioritize mutation test generation. The source code with the highest values of PageRank, will be focused first when developing their test cases as these modules are vulnerable for defects or anomalies which may cause the consequent defects in many other associated modules. Moreover, the approach would help identify the reducible test cases in the test suite, still maintaining the same criteria as the original number of test cases.

***Keywords*—**Software testing, mutation test, network centrality measure, test case prioritization.

I. Introduction

MUTATION testing is invented to help design tests that consist of systems vulnerable to the introduced defects or anomalies. The mutation used at the primary level is unit testing. It also supports other levels, such as specification, design, integration, and system levels [1]. The method is applied in many programming languages such as C, C++, C#, Java, JavaScript, and Ruby, including specification and modeling languages.

Social Network Analysis (SNA) is a technique used for the analysis of data that is characterized by a network of connections between nodes and edges. The technique includes algorithms and measures that can reveal nodes or clusters in the network that require attention. In literature, Koochakzadeh and Alhajj [2] applied SNA technique for generating test case categories based on coverage information. A social network graph was built to identify test packages or higher groups of test cases defined as nodes connected by coverage information between them. The quality of the discovered packages was measured in terms of cohesion and coupling and compared them with the original packaging from test developers. The result showed that the technique was promising, even improving the quality of the packages.

Maitrikul and Limpiyakorn [3] proposed an approach of applying network centrality measures for GUI test case prioritization. The experiments were carried out for ranking test case importance and finding suitable parameter(s) for GUI test case prioritization. The network graph consists of nodes representing an action in a test case, while edges represent the relationship between each activity. Various network centrality measures including betweenness centrality, closeness centrality, eigenvector centrality, and page rank were selected for ranking both modified and new test cases during regression test in a large recommender system. One of the findings reported that the best measurement which would help testers catch defect earlier in each test cycle is Betweenness centrality.

Bunmapob and Limpiyakorn [4] present a visualization approach for exploring the defect data stored in a bug repository. The technique of SNA is applied to uncover the relations between defects, features, and persons. The findings would benefit for the proactive of the software process.

This paper presents a method using a social network centrality measure, PageRank, to prioritize mutation test generation. The experiment was carried out to demonstrate the benefit that would help identify the vulnerable modules which require the robust test cases. The following sections describe the background knowledge applied in this work, the proposed methodology, demonstration and result, and the conclusion.

II. Background

A. Mutation Testing

The tester always configures a set of mutation operators to express concerning string by a tester. Mutation operators are usually applied in feature-oriented programming inject faults [5]. Besides, Papadakis et al. [6] describe a set of operators that transforms syntax to define mutation testing. Each mutation operator presents a type of defect in the code. For example, the Arithmetic Operator changes the syntax of arithmetic - syntax (*) - to other arithmetic - syntax (/) -. The tester maintains the quality of test data iteratively using Mutation testing. They used test data to evaluate the program to cause each mutation to exhibit different behavior. When this happens, the mutation is thought dead and no longer needs to remain in the testing process since the fault that it represents will be detected by the same test that killed it. Moreover, the mutation has satisfied its requirement of identifying a helpful test case. A test process provides a step-by-step to follow to generate a test case. Next, mutation analysis can be applied to evaluate the test case quality. If the test case can kill mutations, it will be reliable for achieving effective test result, otherwise, the test case is vulnerable. A typical mutation analysis process is shown in Fig. 1. The bold boxes represent steps that are automated, while the

Supachai Supmak and Yachai Limpiyakorn are with the Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand (e-mail: supachai_deerse@gmail.com, Yachai.L@chula.ac.th).

other boxes represent manual steps. The Stryker tool is used to support mutators of operations, examples as listed in Table I. Figs. 2 and 3 show some mutators selected from Table I to demonstrate the objective of mutator insertion during mutation test. In Fig. 2, the Object Literal mutator is used to evaluate a test case by changing the object lines 3 - 5 in a function findDntaById to be empty. Fig. 3 (a) illustrates an original source code and (b) the original source code with the String Literal mutator generated by the Stryker tool. The mutator line, marked with '+' under line 11, will change the original source code where is a nonempty string to an empty string.

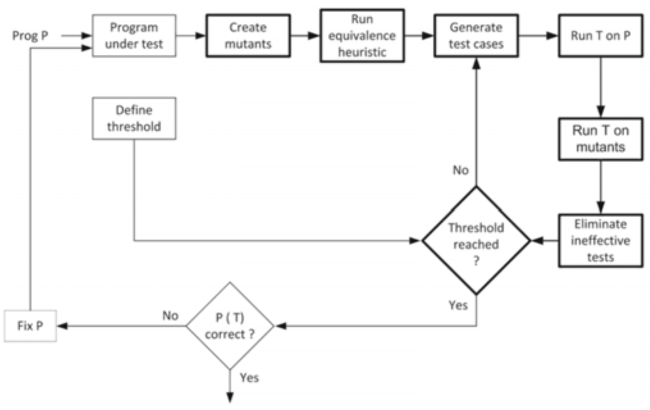
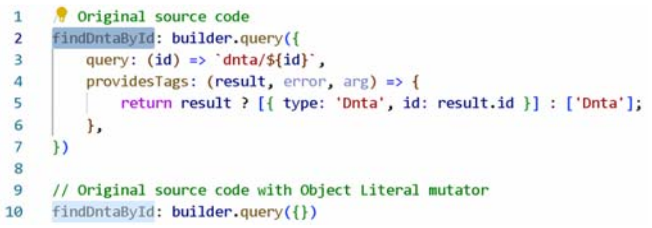
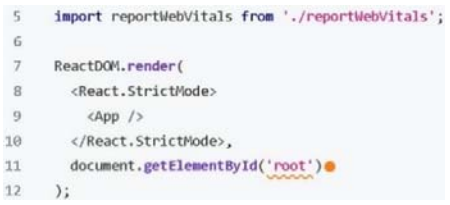


Fig. 1 Mutation testing process [4]



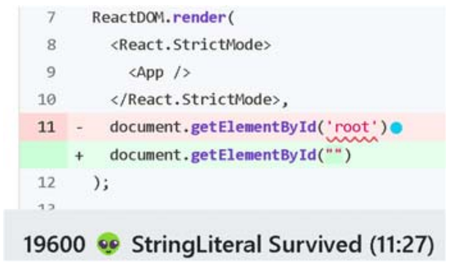
```
1   Original source code
2   findDntaById: builder.query({
3       query: (id) => `dnta/${id}`,
4       providesTags: (result, error, arg) => {
5           return result ? [{ type: 'Dnta', id: result.id }] : ['Dnta'];
6       },
7   })
8
9   // Original source code with Object Literal mutator
10  findDntaById: builder.query({})
```

Fig. 2 Example1 of original source code with mutant



```
5    import reportWebVitals from './reportWebVitals';
6
7    ReactDOM.render(
8      <React.StrictMode>
9        <App />
10     </React.StrictMode>,
11     document.getElementById('root')
12   );
```

(a)



```
7      ReactDOM.render(
8        <React.StrictMode>
9          <App />
10       </React.StrictMode>,
11   -   document.getElementById('root')
     +   document.getElementById("")
12     );
```

19600 StringLiteral Survived (11:27)

(b)

Fig. 3 Example2 of original source code with mutant

TABLE I
LIST OF SUPPORTED MUTATORS AND OPERATIONS IN STRYKER [7], [8]

Mutator	Original	Mutated
Arithmetic Operator	a + b	a - b
	a - b	a + b
	a * b	a / b
	a / b	a * b
	a % b	a % b
Array Declaration	New Array (1, 2, 3, 4)	new Array ()
	[1, 2, 3, 4]	[ ]
Block Statement	Function tryTesting () { Console.log('Test');}	Function tryTesting () {}
Boolean Literal	true	false
	false	true
Conditional Expression	! (a == b)	a == b
	while (a>b) {}	while (false) {}
Equality Operator	a < b	a <= b, a >= b
	a <= b	a < b, a > b
	a > b	a >= b, a <= b
	a >= b	a > b, a <b
	a == b	a! = b
	a! = b	a == b
	a! == b	a! = b
Logical Operator	a && b	a || b
	a ?? b	a && b
Method Expression	endWith ()	startsWith ()
	startsWith ()	endsWith ()
Object Literal	{foo: 'bar'}	{ }
Optional Chaining	foo?.bar	foo.bar
	foo?.[1]	foo[1]
	foo?.()	foo()
Regex	^abc	abc
	abc$	abc
	[abc]	[^abc]
	\d	\D
	\s	\S
	\w	\W
	a++	a
	(?=abc)	(?!abc)
String Literal	"foo" (non-emprty)	""
	s"foo ${bar}" (string interpolation)	s""
Unary Operator	+a	-a
Update Operator	a++	a--
	++a	--a

B. Network Centrality

SNA could be used to study the relationships, interactions and communications among actors in a network graph. Certain actors play key roles as the centrality in a network and they can be discovered by some well-known network centrality measures, such as degree centrality, betweenness centrality, closeness centrality and PageRank. The networks reflect both the cause of and the result of individual behavior. Analyzing these networks provides insight to better understand how individuals are connected, and how information flows.

PageRank was presented by Brin and Page [9] in 1998. The algorithm has been used by Google Search for webpage ranking, i.e., to decide which results to show at the top of its search engine listings. However, the measure encounters some challenges [10], that is, once a node becomes a high centrality, it gives all its centrality to its out-links. The centrality measure is less desirable due to not everyone known by a well-known person is well known. To mitigate this problem, one can divide the value of passed centrality by the number of outgoing links (out-degree) from that node. Each connected neighbor gets a fraction of the source node's centrality as calculated in (1):

$$C_p(v_i) = \alpha \sum_{j=1}^{n} A_{i,j} \frac{C_p(v_i)}{d_j^{out}} + \beta \qquad (1)$$

where d_j^{out} is nonzero.

III. Methodology

The proposed methodology is composed of several steps, as illustrated in Fig. 4. The application, Transport Network Operating System (TNOS) containing 258 source codes, is selected for the demonstration. The application is implemented to support customers in planning and estimating for choosing the operating vehicle routes. The application software is developed by JavaScript language and react native framework. The Stryker supports many languages, for example, JavaScript, C#, and Scala.

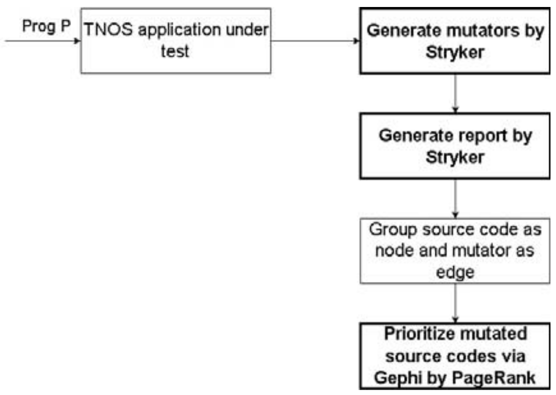
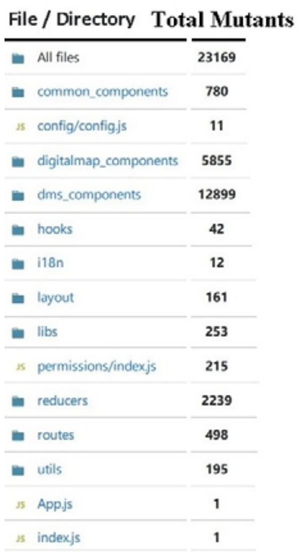

Prog P
TNOS application under test
Generate mutators by Stryker
Generate report by Stryker
Group source code as node and mutator as edge
Prioritize mutated source codes via Gephi by PageRank

Fig. 4 Overview of the proposed methodology

A. Generate Mutators and Report Using Stryker

The Stryker for JavaScript is applied to TNOS application for mutant generation. The total number of 23169 mutants associated with 258 source code was generated as reported in Fig. 5.

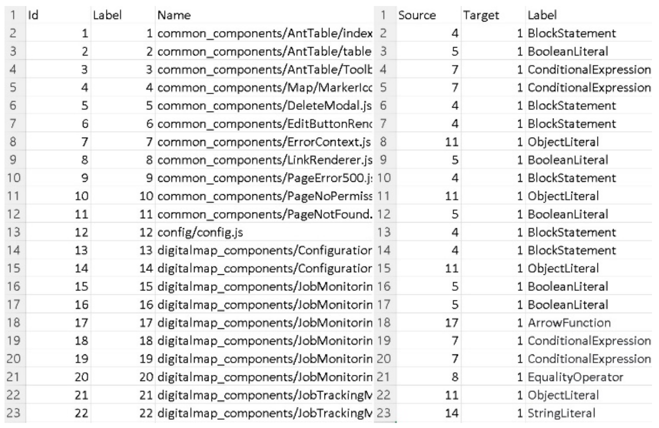


File / Directory	Total Mutants
All files	23169
common_components	780
config/config.js	11
digitalmap_components	5855
dms_components	12899
hooks	42
i18n	12
layout	161
libs	253
permissions/index.js	215
reducers	2239
routes	498
utils	195
App.js	1
index.js	1

Fig. 5 Summary report of total mutants generated for TNOS

B. Define Nodes and Associated Edges

The two data files in csv format: Node and Edge, are manually created. Excerpts of Node.csv and Edge.csv as input to Gephi are shown in Fig. 6. The Node.csv file contains list of source code associated with the source code ID and label. The Edge.csv file contains source as the id of mutator type, target as a mapping to node id, and label as a type of mutator. These two .csv files are imported into Gephi [11] which is the Open Graph Viz Platform. Gephi is used to process and visualize the network graph.



Id	Label	Name
1	1	common_components/AntTable/index
2	2	common_components/AntTable/table
3	3	common_components/AntTable/Toolb
4	4	common_components/Map/MarkerIcc
5	5	common_components/DeleteModal.js
6	6	common_components/EditButtonRenc
7	7	common_components/ErrorContext.js
8	8	common_components/LinkRenderer.js
9	9	common_components/PageError500.j:
10	10	common_components/PageNoPermiss
11	11	common_components/PageNotFound.
12	12	config/config.js
13	13	digitalmap_components/Configuratior
14	14	digitalmap_components/Configuratior
15	15	digitalmap_components/JobMonitorin
16	16	digitalmap_components/JobMonitorin
17	17	digitalmap_components/JobMonitorin
18	18	digitalmap_components/JobMonitorin
19	19	digitalmap_components/JobMonitorin
20	20	digitalmap_components/JobMonitorin
21	21	digitalmap_components/JobTrackingM
22	22	digitalmap_components/JobTrackingM

Source	Target	Label
4	1	BlockStatement
5	1	BooleanLiteral
7	1	ConditionalExpression
7	1	ConditionalExpression
4	1	BlockStatement
4	1	BlockStatement
11	1	ObjectLiteral
5	1	BooleanLiteral
4	1	BlockStatement
11	1	ObjectLiteral
5	1	BooleanLiteral
4	1	BlockStatement
4	1	BlockStatement
11	1	ObjectLiteral
5	1	BooleanLiteral
5	1	BooleanLiteral
17	1	ArrowFunction
7	1	ConditionalExpression
7	1	ConditionalExpression
8	1	EqualityOperator
11	1	ObjectLiteral
14	1	StringLiteral

Fig. 6 Node.csv and Edge.csv as input to Gephi

Table II displays the total ten types of mutators generated from the source code or node ID 49 consisting of: Block Statement, Boolean Literal, String Literal, Conditional Expression, Logical Operator, Array Declaration, Object Literal, Equality Operator, Arithmetic Operator, and Optional Chaining. The total number of 270 mutants was generated and inserted into the original source code, some of which are duplicated types.

TABLE II
List of Mutants and Its Category Inserted in Node 49

Mutator	Mutant
Block Statement	const TableComponent = (props) => {};
Boolean Literal	const [showDelete, setShowDelete] = useState(true);
String Literal	selectTableStateById(state, "")
Conditional Expression	const [dataSource, setDataSource] = useState(false);
Logical Operator	const [dataSource, setDataSource] = useState(storedDataSource && []);
Array Declaration	const [dataSource, setDataSource] = useState(storedDataSource || ["Stryker was here"]);
Object Literal	{}
Equality Operator	requiredRows >= totalRows && totalRows !== 0 ? totalRows : requiredRows;
Arithmetic Operator	let requiredRows = page / PAGE_SIZE;
Optional Chaining	ref.current.scrollTo(top);

C. Prioritize Mutated Source Code with PageRank

Based on the input data describing 258 source codes as nodes and the description of edges representing 23169 mutants generated by Stryker, the Gephi software then computed and generated the network graph using the selected centrality measure, PageRank, as illustrated in Fig. 7. The source codes associated with the first twenty highest ranking of PageRank values are displayed in Table III. The node ID 49 and 61 are ranked with the highest PageRank score. The result suggested the order of test case development for the tester. That is, the source code with the higher PageRank value, the earlier the test case of that source code will be developed and tested.

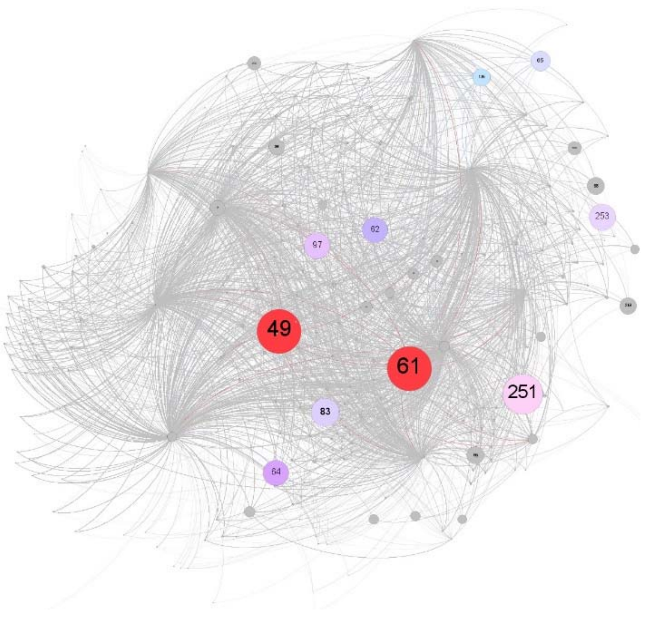
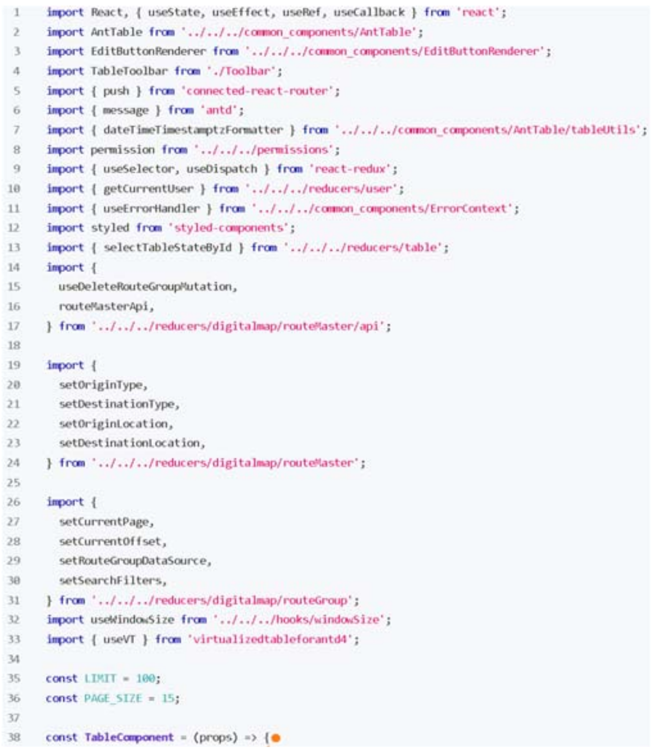


Fig. 7 SNA graph with PageRank centrality measure

TABLE III
LIST OF MUTATED SOURCE CODES WITH THE FIRST TWENTY HIGHEST RANKING OF PAGERANK VALUES

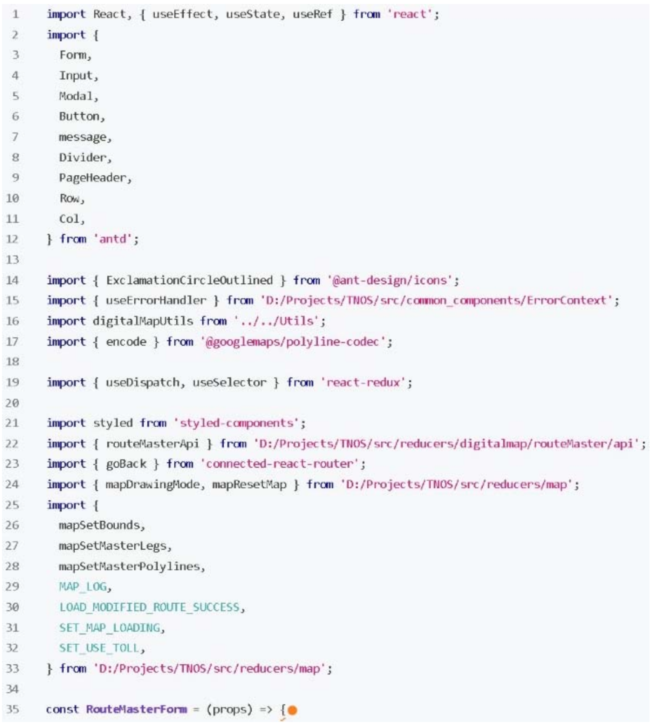
ID	Source Code	PageRank
49	digitalmap_components/RouteMaster/List/Table.js	0.004032
61	digitalmap_components/TTTJobMonitoring/List/Table.js	0.004032
251	reducers/utils/json2csv.js	0.003984
83	dms_components/DispatchBoard/List/Table.js	0.003857
253	reducers/utils/objectByString.js	0.003840
97	dms_components/Group/Details/BzpForm.js	0.003833
64	digitalmap_components/TTTJobTrackingMapview/InfoPanel.js	0.003829
62	digitalmap_components/TTTJobMonitoring/List/Toolbar.js	0.003825
65	digitalmap_components/TTTJobTrackingMapview/Mapview.js	0.003768
136	dms_components/MasterData/ReasonMaster/SubjectReason/Details/Form.js	0.003743
68	digitalmap_components/TTTJobTrackingMapview/StartMarker.js	0.003741
66	digitalmap_components/TTTJobTrackingMapview/OrderMarker.js	0.003740
249	reducers/utils/cookies.js	0.003734
35	digitalmap_components/Map/BaseMap.js	0.003723
2	common_components/AntTable/tableUtils.js	0.003716
198	libs/normalize.js	0.003803
200	libs/validation.js	0.003803
52	digitalmap_components/RouteMaster/RouteGroup/RouteCandidateTable.js	0.003790
54	digitalmap_components/RouteMaster/RouteGroup/RouteMasterTable.js	0.003790
56	digitalmap_components/RouteMaster/RouteMaster/RouteMasterForm.js	0.003787

Observing that the node with a high PageRank score means the source code that calls or imports many functions such as node ID 49 calling eighteen functions as shown in Fig. 8. While node ID 56, ranking the twentieth, merely calls twelve functions as shown in Fig. 9.



```
import React, { useState, useEffect, useRef, useCallback } from 'react';
import AntTable from '../../../common_components/AntTable';
import EditButtonRenderer from '../../../common_components/EditButtonRenderer';
import TableToolbar from './Toolbar';
import { push } from 'connected-react-router';
import { message } from 'antd';
import { dateTimeTimestamptzFormatter } from '../../../common_components/AntTable/tableUtils';
import permission from '../../../permissions';
import { useSelector, useDispatch } from 'react-redux';
import { getCurrentUser } from '../../../reducers/user';
import { useErrorHandler } from '../../../common_components/ErrorContext';
import styled from 'styled-components';
import { selectTableStateById } from '../../../reducers/table';
import {
  useDeleteRouteGroupMutation,
  routeMasterApi,
} from '../../../reducers/digitalmap/routeMaster/api';

import {
  setOriginType,
  setDestinationType,
  setOriginLocation,
  setDestinationLocation,
} from '../../../reducers/digitalmap/routeMaster';

import {
  setCurrentPage,
  setCurrentOffset,
  setRouteGroupDataSource,
  setSearchFilters,
} from '../../../reducers/digitalmap/routeGroup';
import useWindowSize from '../../../hooks/windowSize';
import { useVT } from 'virtualizedtableforantd4';

const LIMIT = 100;
const PAGE_SIZE = 15;

const TableComponent = (props) => {
```

Fig. 8 Source code of node ID 49

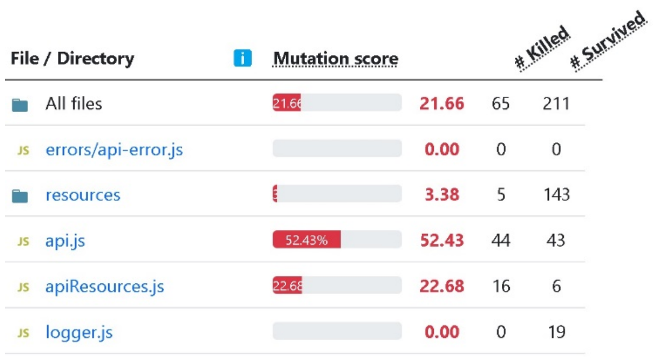


```
import React, { useEffect, useState, useRef } from 'react';
import {
  Form,
  Input,
  Modal,
  Button,
  message,
  Divider,
  PageHeader,
  Row,
  Col,
} from 'antd';

import { ExclamationCircleOutlined } from '@ant-design/icons';
import { useErrorHandler } from 'D:/Projects/TNOS/src/common_components/ErrorContext';
import digitalMapUtils from '../../Utils';
import { encode } from '@googlemaps/polyline-codec';

import { useDispatch, useSelector } from 'react-redux';

import styled from 'styled-components';
import { routeMasterApi } from 'D:/Projects/TNOS/src/reducers/digitalmap/routeMaster/api';
import { goBack } from 'connected-react-router';
import { mapDrawingMode, mapResetMap } from 'D:/Projects/TNOS/src/reducers/map';
import {
  mapSetBounds,
  mapSetMasterLegs,
  mapSetMasterPolylines,
  MAP_LOG,
  LOAD_MODIFIED_ROUTE_SUCCESS,
  SET_MAP_LOADING,
  SET_USE_TOLL,
} from 'D:/Projects/TNOS/src/reducers/map';

const RouteMasterForm = (props) => {
```

Fig. 9 Source code of node ID 56

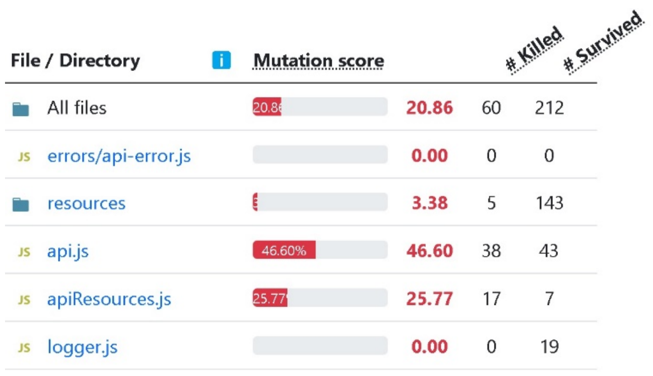
D.Perform Mutation Test

The inputs required for the Stryker mutation test consist of a set of test cases of source codes under test created by the tester, associated with the mutated source codes generated by Stryker in the previous step. Omise [12] is chosen for demonstrating mutation testing here. Omise is a payment gateway and REST API allowing integration across a variety of languages and frameworks. It consists of 19 source codes: errors/api-error.js, api.js, apiResources.js, logger.js, and the rest of 15 source codes contained in the resources folder. Account.js is the code with the highest PageRank score, while Event.js is the code with the lowest PageRank score. The results of mutation test on Account.js and Event.js are reported in Figs. 10 and 11, respectively. Compared to Event.js, Account.js achieves the higher performance, that is, higher percentage of Mutation score, higher number of killed mutants and lower number of survived. Start testing the source codes with high PageRank scores would cover#the subsequent testing those with low PageRank scores that contain the same types of mutants, resulting in decreased time and efforts.



File / Directory	Mutation score		# Killed	# Survived
All files	21.66	21.66	65	211
errors/api-error.js		0.00	0	0
resources		3.38	5	143
api.js	52.43%	52.43	44	43
apiResources.js	22.68	22.68	16	6
logger.js		0.00	0	19

Fig. 10 Mutation test result of code with highest PageRank score



File / Directory	Mutation score		# Killed	# Survived
All files	20.86	20.86	60	212
errors/api-error.js		0.00	0	0
resources		3.38	5	143
api.js	46.60%	46.60	38	43
apiResources.js	25.77	25.77	17	7
logger.js		0.00	0	19

Fig. 11 Mutation test result of code with Lowest PageRank score

IV. Conclusion

It is evident that testing is one of resource consumption activities in a software project. And it is ideal to achieve path coverage when testing. In literature, test case prioritization is one of the well-known solutions to alleviate the pain.

Mutation test is a technique used for quality assessment of test suite. It helps identify the test case vulnerability by means of mutant insertion into the original source code. The high-quality test cases are expected to kill high percentage of mutants. This paper proposed applying a network centrality measure called PageRank for test case prioritization during mutation test performed by the Stryker tool. The source code that calls many functions will earn the high PageRank score due to plenty of outgoing edges from other nodes or source codes. The software testing process starting from these source codes with high PageRank scores will yield higher test coverage since the tester can skip writing the test cases and ignore testing those lower PageRank source codes containing the same types of mutants that have been successfully killed when testing the high PageRank code formerly. Further direction would be the exploration of other network centrality measures such as Betweenness centrality for test case prioritization in mutation testing.

References

[1] A. J. Offutt, R. H. Untch, Mutation 2000: Uniting the orthogonal, Mutation Testing for the New Century, Advances in Database Systems, Springer, Boston, MA, USA, 2001, pp.34-44

[2] N. Koochakzadeh, R. Alhajj, *Social Network Analysis in Software Testing to Categorize Unit Test Cases Based on Coverage Information*, 2011 IEEE International Conference on High Performance Computing and Communications, 2011, pp. 412-416

[3] C. Maitrikul, Y. Limpiyakorn, *GUI Test Case Prioritization using Social Network Analysis*, 2022, 13th International Conference on Computer and Electrical Engineering

[4] P. Bunmapob, Y. Limpiyakorn, *Exploring Defect Data with Network Visualization*, 2022 2nd IEEE International Conference on Software Engineering and Artificial Intelligence, 2022, pp.204

[5] A. J. Offutt, R. H. Untch, Mutation 2000: Uniting the orthogonal, Mutation Testing for the New Century, Advances in Database Systems, Springer, Boston, MA, USA, 2001, pp.34-44

[6] M. Papadakis, M. Kintis, Z. Jie, J. Yue, Y. L. Traon, M. Harman, *Mutation testing advances: An analysis and survey*, 2019, pp. 275-378

[7] *Stryker Mutator*, https://stryker-mutator.io/docs/

[8] *Supported Mutators*, https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/

[9] S. Brin, L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Proc. 7th International World-Wide Web Conference*, Brisbane, Australia, 1998, pp. 107-117.

[10] R. Zafarani, M. A. Abbasi, L. Huan, *Social media mining: an introduction*. Cambridge University, 2014, pp.51-79.

[11] *Gephi*, https://gephi.org/