دانشگاه شهید بهشتی

دانشکده مهندسی و علوم کامپیوتر

Automatic Unit Test Generation in JavaScript

گزارش پروژه عملی آزمون نرم افزار پیشرفته

کارشناسی ارشد نرم افزار

پژوهندگان

بهزاد خسروی فر

الهام استدلالی

استاد

دکتر حقیقی

زمستان 1397

هدف پروژه

هدف این پروژه تولید خودکار تست واحد (Unit Test) نرم افزار با استفاده از زبان جاوا اسکریپت نسخه ES5 در چهارچوب Jest می باشد.بنابراین نیاز به نصب node.js در سیستم مورد استفاده می باشد. این گزارش کار شامل چند بخش می باشد، بخش اول، روند انجام پروژه ای که پیاده سازی شده را شرح خواهد داد و بخش دوم، «چگونه از برنامه استفاده کنیم» [1] شرح استفاده از این برنامه را بیان می کند.

لازم به ذکر می باشد، سورس کد پیاده سازی شده برنامه بعلت حجم بالا و عدم پشتیبانی سایت LMS دانشگاه، در وب سایت گیت هاب به آدرس زیر قرار داده شده است و همچنین می توانید با توجه به بخش «چگونه از برنامه استفاده کنیم» آن را دریافت، نصب و راه اندازی کنید.

https://github.com/bezzad/UnitJest

روند انجام پروژه

روند انجام پروژه بصورت زیر می باشد که در ادامه با جزئیات شرح داده شده است.

1) معرفی Unit Test
2) انتخاب زبان برنامه نویسی جاوااسکریپت
3) انتخاب چارچوب[2] طراحی تست در زبان انتخابی
4) پیاده سازی نمونه تست در چارچوب انتخابی
5) طراحی پیش پردازشگر زبان انتخابی
6) انتخاب کتابخانه رسم گراف در زبان انتخابی
7) رسم گراف از خروجی مفسر
8) ایجاد نمونه کدهای آزمایشی جهت تولید تست برای آنها
9) ایجاد متد ارزیابی کد پوشانی برای مقادیر ورودی
10) استفاده الگوریتم ژنتیک برای تولید مقادیر تست
11) چگونه از برنامه استفاده کنیم

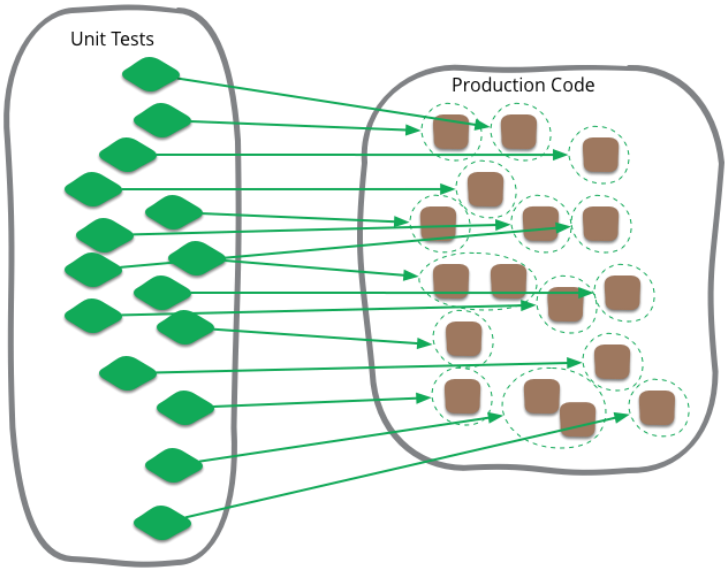
[1] How to use
[2] Framework

شرح روند انجام پروژه

1) معرفی Unit Test:

در برنامه‌نویسی، Unit Test روشی است برای آزمودن واحدهای کوچکی از کد منبع برنامه و اطمینان از درست کار کردن آن‌ها؛ در این روش، درستی هر قسمت از کد، که به آن «واحد[1]» گفته می‌شود، با استفاده از کدهای دیگری که توسط برنامه‌نویس نوشته شده ارزیابی می‌گردد.

در حالت ایده‌آل هر کدام از تست ها از بقیه مستقل است. معمولاً تست های واحد توسط توسعه‌دهندگان نرم‌افزار به کار گرفته می‌شوند. نحوه تست واحد می‌تواند از ارزیابی نتیجه روی کاغذ، تا اجرای خودکار چندین آزمایش توسط برنامه و تحلیل نتیجهٔ آن‌ها توسط خود برنامه، متغیر باشد.



یک واحد برنامه دارای خصوصیات زیر می باشد:

- Solid است و اصولا هیچ وابستگی[2] به بخش های دیگر برنامه ندارد.
- اگر ورودی آن تا ابد برابر با ورودی قبلی بود، خروجی نیز برابر با خروجی که قبلا گرفته ایم باشد. یعنی خروجی نباید رندوم باشد یا برحسب GUID یا زمان تغییر کند.

2) انتخاب زبان برنامه نویسی:

از بین زبان های زیر، در نهایت زبان جاوا اسکریپت را بدلیل آشنایی بیشتر و اینکه درگیر مباحثی مانند Object Oriented و syntax گسترده آن زبان ها نشویم و کار سریعتر پیش برود، انتخاب کردیم. همچنین جاوا اسکریپت زبانی است که برای آن Automation Unit Test وجود ندارد.

- c

[1] Unit
[2] Dependency

- c++
- c#
- java
- javascript ✓
- python
- ruby
- vb.net

3) انتخاب چارچوب طراحی تست در زبان انتخابی[1]:

زبان جاوا اسکریپت بصورت built-in چارچوبی برای «Unit Test» ندارد و از بین پکیج هایی که این امکان را در زبان جاوا اسکریپت فراهم می کنند، Mocha.js (کاملترین چارجوب که امکاناتی از قبیل Coverage را دارد)، Jest و chai.js منتخب شدند. اما jest را بدلیل سادگی استفاده و کد Coverage ای که در اختیار قرار می دهد، سرعت و تجربه کاری مان انتخاب کردیم.

- Mocha.js[2]
- Jest[3] ✓
- chai.js
- Jasmine
- YUI Test
- QUnit
- Unit.js
- Jenkins
- Puppeteer

4) پیاده سازی نمونه تست در چارچوب انتخابی:

```
1) Install node.js(minimum supported version is v6 .0 .0)
2) $ npm install jest--global
3) $ jest--init
4) $ mkdir project
5) Create test project by name project / test.js like attached file
6) $ jest test
```

بعنوان مثال، کد زیر که در فایل test.js نوشته شده است مربوط به تست یک تابع فاکتوریل می باشد:

```
function fact(num) {
    var fact = 1;
    do {
        fact *= num;
        num -= 1;
    } while (num >= 1);
```

[1] Java Script Test FrameWork
[2] https://mochajs.org
[3] https://github.com/facebook/jest , https://jestjs.io/docs/en/getting-started

```
    return fact;
}

test('fact 0 to equal 1', () => {
    expect(fact(0)).toBe(1);
});
test('fact 1 to equal 2', () => {
    expect(fact(1)).toBe(1);
});
test('fact 2 to equal 6', () => {
    expect(fact(2)).toBe(2);
});
test('fact 3 to equal 6', () => {
    expect(fact(3)).toBe(6);
});
test('fact 4 to equal 24', () => {
    expect(fact(4)).toBe(24);
});
test('fact 5 to equal 120', () => {
    expect(fact(5)).toBe(120);
});
test('fact 6 to equal 720', () => {
    expect(fact(6)).toBe(720);
});
```

که پس از اجرا آن، خروجی آن بصورت زیر می باشد:

```
E:\Test>jest test
 PASS  ./test.js
  √ fact 3 to equal 6 (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.843s, estimated 1s
Ran all test suites matching /test/i.

E:\Test>jest test
 FAIL  ./test.js
  × fact 0 to equal 1 (35ms)
  √ fact 1 to equal 2 (2ms)
  √ fact 2 to equal 6 (1ms)
  √ fact 3 to equal 6 (1ms)
  √ fact 4 to equal 24 (1ms)
  √ fact 5 to equal 120 (1ms)
  √ fact 6 to equal 720 (1ms)

  ● fact 0 to equal 1

    expect(received).toBe(expected) // Object.is equality
```

شکل 1 خروجی تست پس اجرای دستور jest test

5) طراحی پیش پردازشگر زبان انتخابی:

ما از مفسر Esprima[1] برای تفسیر کدهای تست استفاده می کنیم که خروجی آن یک درخت AST می باشد. بدلیل اینکه ممکن است کد مورد آزمایش ما دارای ساختار نامنظم و غیر استاندارد برای تحلیل نمودار و مانیتورینگ Coverage کدها دشوار باشد. بنابراین ساختار AST بدست آمده را به کتابخانه متن باز escodegen[2] می دهیم تا کد ساختار یافته ای از درخت برنامه تست ما بوجود آورد که این کد با فرمت استاندارد می باشد، بطوریکه در آن هر سطر کد مربوط به یک دستور می باشد.

حال نوبت به پرچم گذاری روی تک تک کدهای برنامه است. بعبارت دیگر، برای اینکه بدانیم برنامه در هنگام اجرا از چه مسیری عبور می کند باید بتوانیم برنامه را مانیتور کنیم. برای این کار نیاز به اضافه کردن تکه کدی به بعد هر سطر از برنامه را داریم تا به ما بگوید حالا کدام سطر اجرا شده است که به آن عمل instrument code[3] گفته می شود. عملیات فوق در فایل instrument.js پیاده سازی شده است.

6) انتخاب کتابخانه رسم گراف در زبان انتخابی:

همانطور که گفتیم، ما از مفسر Esprima با خروجی درخت AST استفاده می کنیم و سپس از روی آن گراف جریان برنامه با فرمت Graphviz[4] تولید می شود. از بین کتابخانه های رسم گراف که در زیر مشاهده می کنید، تنها کتابخانه viz.js از این فرمت پشتیبانی می کند، بنابراین این کتابخانه را انتخاب کردیم.

- http://sigmajs.org
- https://d3js.org
- https://canvasjs.com
- https://developers.google.com/chart
- http://gionkunz.github.io/chartist-js
- http://n3-charts.github.io/line-chart
- http://www.amcharts.com
- https://www.highcharts.com
- http://js.cytoscape.org
- https://github.com/anvaka/VivaGraphJS
- http://viz-js.com ✓

7) رسم گراف از خروجی مفسر:

بعنوان نمونه گراف مربوط به تابع فاکتوریل اینگونه بدست آمده است:

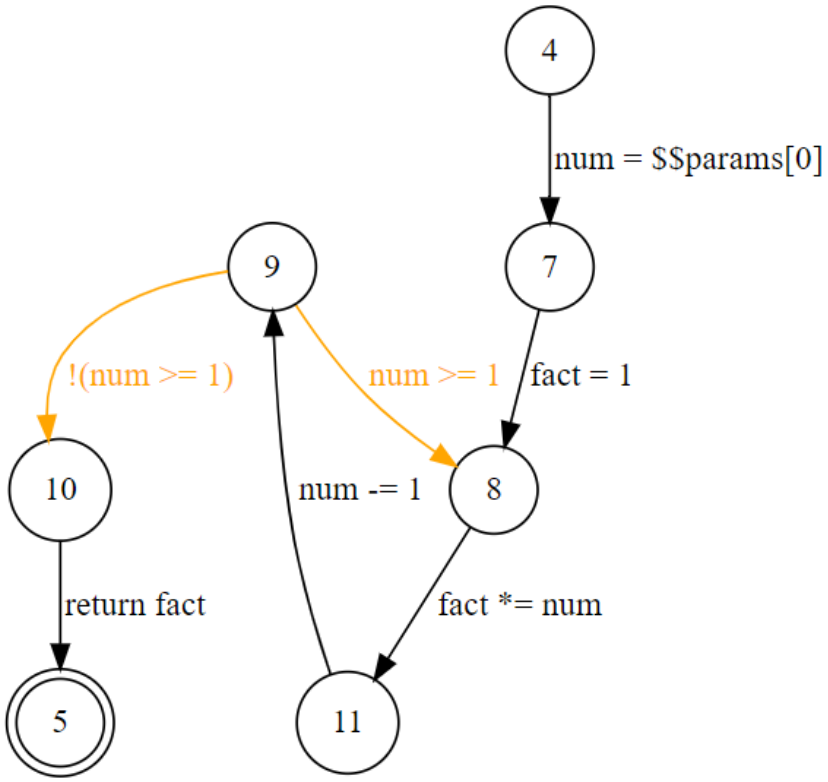
[1] https://github.com/jquery/esprima
[2] https://github.com/estools/escodegen
[3] https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)
[4] Graphviz ها اطلاعاتی از نوع dot format برای نمایش گراف جریان برنامه هستند.

کد فرمت DOT برای رسم گراف:

```
// fact
digraph control_flow_graph {
    node [shape = doublecircle] 5
    node [shape = circle]

    // Unconditional edges
    4 -> 7 [label = " num = $$params[0]"]
    7 -> 8 [label = " fact = 1"]
    8 -> 11 [label = " fact *= num"]
    10 -> 5 [label = " return fact"]
    11 -> 9 [label = " num -= 1"]

    // Conditional edges
    edge [color = orange, fontcolor = orange]
    9 -> 8 [label = " num >= 1"]
    9 -> 10 [label = " !(num >= 1)"]
}
```



شکل 2رسم گراف کنترل جریان تابع فاکتوریل با vis.js

8) ایجاد نمونه کدهای آزمایشی جهت تولید تست برای آنها:

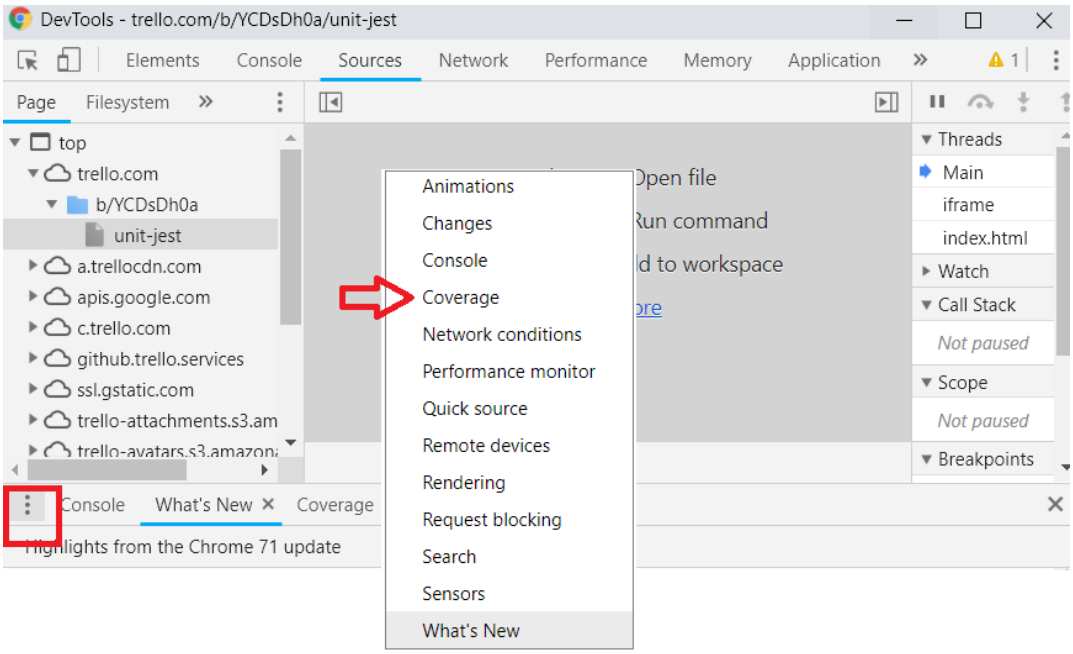
در این پروژه، ما بر روی چند کد واحد شناخته شده تست را انجام می دهیم که در پوشه test-samples قرار گرفته اند. عبارتند از:

- تابع فاکتوریل[1]
- triangle Classification (دسته بندی مثلث)

9) ایجاد متد ارزیابی کد پوشانی برای مقادیر ورودی

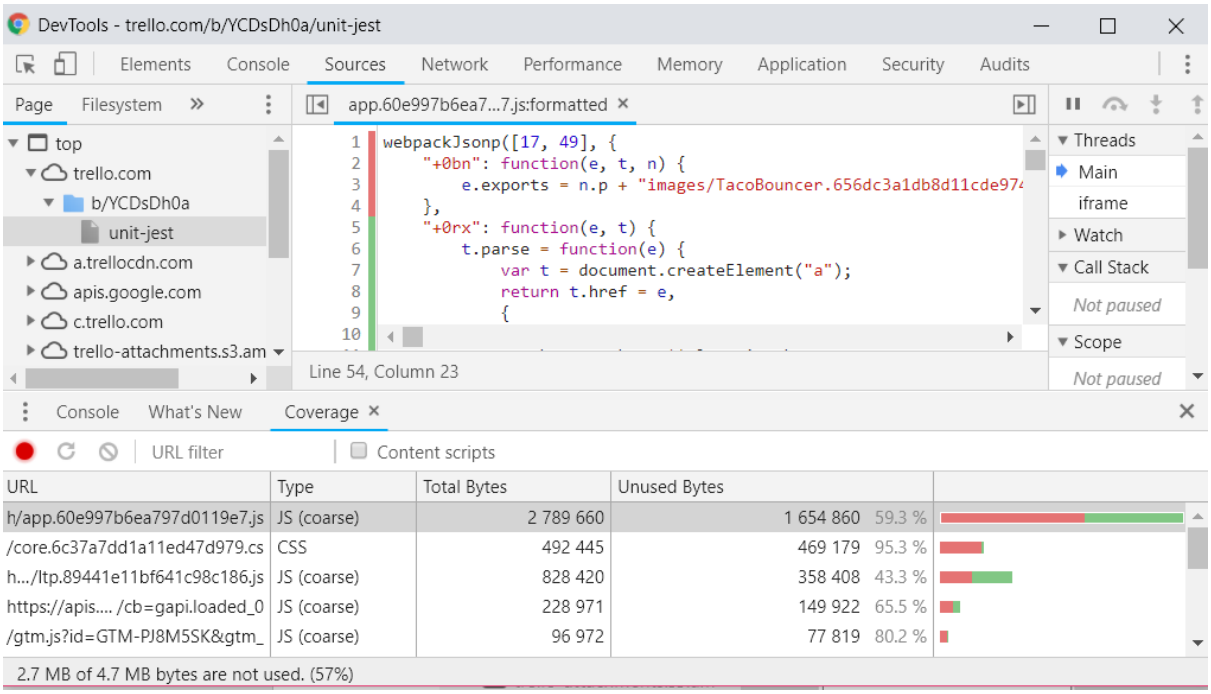
انواع کد پوشانی (Coverage) عبارتند از:

- Statement: تکه هایی از کد بعنوان block Statement شناخته شده و تمام block ها باید اجرا شود.
- Function: اجرای توابع برنامه را نشان می دهد.
- Line/Code: تک تک کدها یکبار اجرا شوند. اکثرا تست در دنیا روی Code Coverage می باشد. بعنوان مثال در مرورگر کروم از ورژن ۵۹ به بعد امکانی بنام Coverage در Chrome DevTools اضافه شده است که کد سمت font-end وب سایت مورد نظر را تحلیل کرده و میزان Coverage را در آن نمایش می دهد.
- Branch: برنامه ای که بر حسب شرط ها به شاخه هایی تقسیم می شود در کدام شاخه ها اجرا شده است.



شکل 3نحوه اضافه کردن کد پوشانی در مرورگر کروم

[1] Factorial



شکل 4نمودارهای کد پوشانی (Code Coverage) در مرورگر کروم

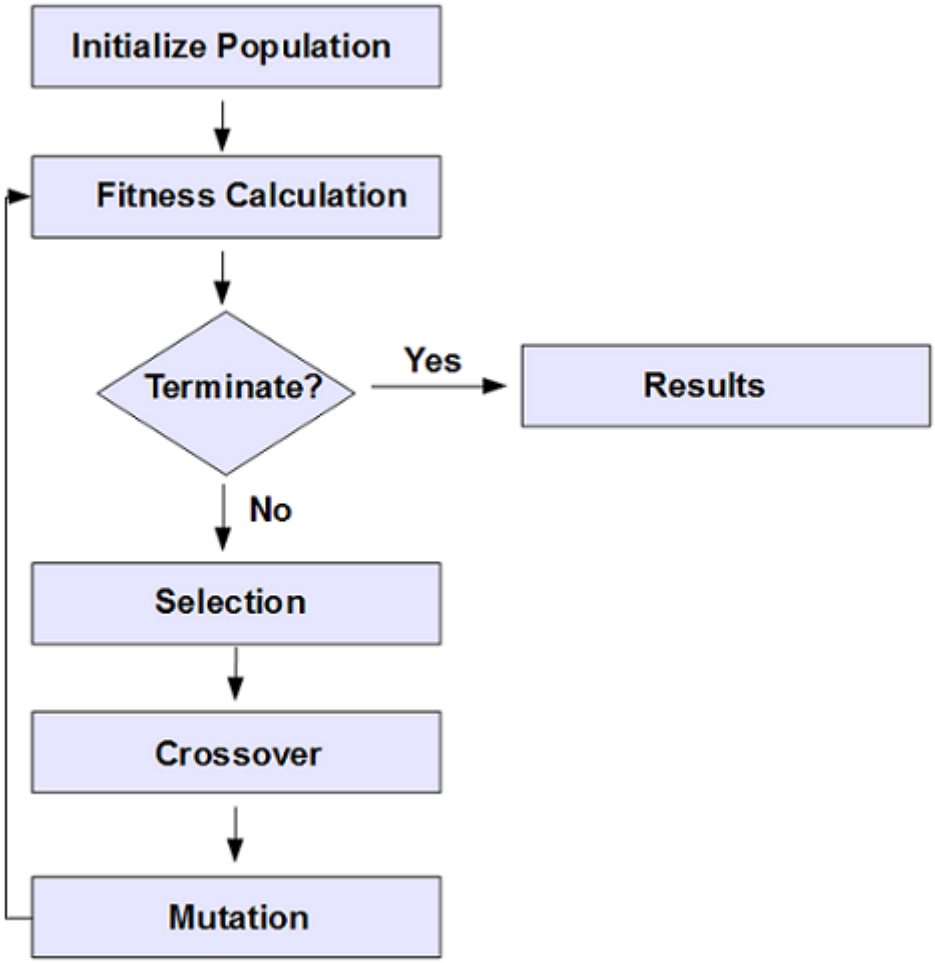
با توجه به انواع کد پوشانی می‌توان متدی را نوشت که داده‌های ورودی یک تابع یا همان پارامترهای تابع را گرفته و تحلیل کند که از چه مسیر عبور کرده و چند درصد پوشانی دارد. با توجه به مقدار پوشانی، مقادیری که دارای پوشانی بالاتری هستند امتیاز بیشتری می‌گیرند و ارزش بیشتری را در TestCaseدارند. با توجه به اینکه مسئله ی تولید داده از روی مسیرهای تست از نوع NP می‌باشد بنابراین ما این روند رو معکوس کرده و با الگوریتم ژنتیک مقادیر ورودی توابع تحت تست را تولید کرده و با این متد ارزیابی بهترین آن‌ها را برای Unit Test انتخاب می‌کنیم.

10) استفاده الگوریتم ژنتیک برای تولید مقادیر تست

با الگوریتم ژنتیک ما یک سری مقادیر ورودی توابع تحت تست را تولید می‌کنیم تا بتوانیم مسیرهایی از تست را که دارای بیشترین حد پوشانی می باشند را در بر گرفته باشیم. این الگوریتم دارای ساز و کار تکاملی می‌باشد بنابراین مقادیر اولیه بصورت تصادفی ایجاد می‌شوند و رفته رفته در هر نسل این مقادیر بهتر و بهتر می‌شوند.

در الگوریتم پیاده سازی شده ما کروموزوم‌ها را مقادیری برای پارامترها در نظر گرفته این که با هم ترکیب شده و کروموزوم‌های نسل جدید را بوجود می‌آورند و عمل جهش نیز با در نظر گرفتن درصد احتمال روی این کروموزوم‌های بوجود آمده اعمال می‌شود تا جلوی همگرایی احتمالی را بگیرد و چنانچه بعد از ۱۰۰۰نسل به جواب پوشانی بالای ۸۰٪ نرسیدیم الگوریتم متوقف می‌شود و همان مقادر کروموزوم نخبه به عنوان جواب نهایی انتخاب می‌شود. لازم به ذکر است که پوشانی کدها توسط فقط یک داده ورودی انجام نمی‌گیرد بلکه یک کروموزوم به طول ۱۰ دارای ۱۰ test case برای تابع تحت تست می‌باشد که با هم پوشانی را بوجود آورده اند. در نهایت خروجی برنامه ۱۰ عدد unit test در چهارچوب

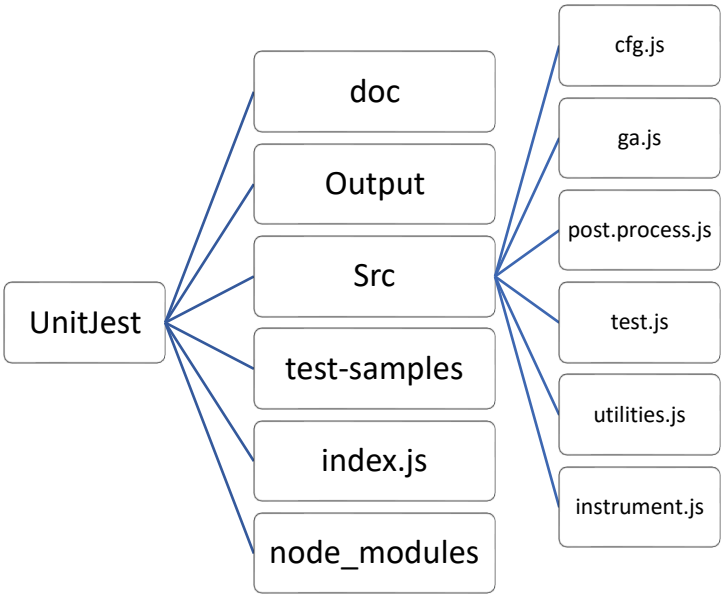
Jest می‌باشد که در این زبان قابل درک است. برای اطلاعات بیشتر به فایل ga.js مراجعه کنید تا از عملکرد این الگوریتم در برنامه ما بیشتر آشنا شوید.



شکل 5فلوچارت الگوریتم ژنتیک

11) چگونه از برنامه استفاده کنیم

ساختار کلی پروژه بصورت زیر می باشد:

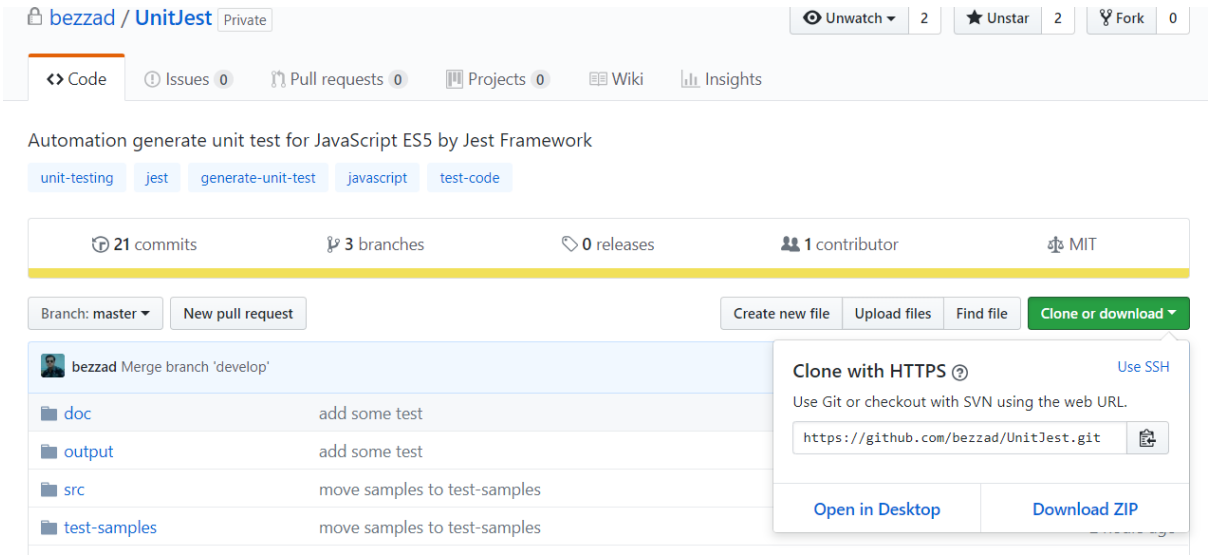


ساختار کلی فایل‌های پروژه پیاده سازی شده6 شکل

در برنامه ما **index.js** فایل اصلی[1] پروژه می باشد که با دستور node index.js اجرا می‌شود.

برای اجرای برنامه در سیستم خود به مراحل زیر توجه بفرمائید:

۱. node.js را بر روی سیستم خود نصب کنید.
۲. به آدرس https://github.com/bezzad/UnitJest در وب سایت گیت هاب رفته و از قسمت clone or download طبق شکل زیر پروژه را دانلود کنید.

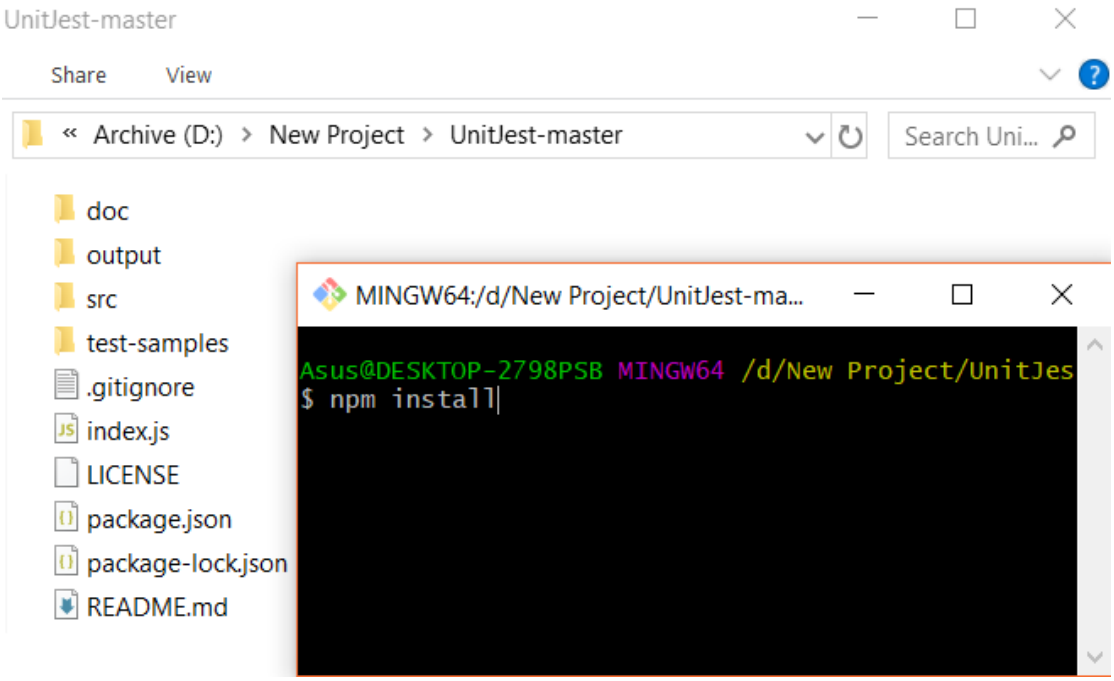


در محل دلخواهی از کامپیوتر فایل دریافتی را از حالت فشرده خارج کنید.

[1] main

۳. وارد پوشه پروژه شوید و محیط Cmd ویندوز[1] را در آن پوشه باز کنید. دستور زیر را وارد کنید تا پکیج های استفاده شده در پروژه از سایت گیت هاب بر روی کامپیوتر شما دانلود شود.

```
$ npm install
```



با اجرای دستور فوق، پوشه node_modules ساخته شده و ماژول ها در آن قرار می گیرند.

۴. بعد از اجرای برنامه پوشه output ساخته شده و داده ها در آن قرار می گیرند.(کد کامپایل شده + کدهای تست + نمودار CFG)

۵. با دستور زیر دستورات تست را اجرا می کنیم (با این دستور فایل test.js توسط فریم ورک jest اجرا می شود):

```
$ jest test.js
```

لازم به ذکر است که فایل اصلی برنامه برای اجرا پروژه index.js واقع در ابتدای پوشه پروژه می‌باشد که با دستور node index.js اجرا شده و خروجی را در پوشه output قرار می‌دهد.

[1] Powershell window

مراجع:

1. https://en.wikipedia.org/wiki/Unit_testing
2. https://github.com/facebook/jest
3. https://jestjs.io/docs/en/getting-started
4. https://github.com/jquery/esprima
5. http://viz-js.com/
6. https://en.wikipedia.org/wiki/Graphviz
7. http://web.cs.iastate.edu/~weile/cs513x/4.ControlFlowAnalysis.pdf
8. http://martintrojer.github.io/software/2011/11/02/symbolic-execution
9. https://github.com/leebyron/testcheck-js
10. https://www.cs.ccu.edu.tw/~naiwei/cs5812/st4.pdf