Implementation of algorithms and data structures

4. seminar

Jirka Fink

https://ktiml.mff.cuni.cz/~fink/

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Summer semestr 2021/22
Last change 9. března 2022

Fuzz testing

Description (Wikipedia)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Fuzz testing

Description (Wikipedia)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Motivation

- For advanced algorithm, we are unable to create unit tests for all possible cases.

Fuzz testing

Description (Wikipedia)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Motivation

- For advanced algorithm, we are unable to create unit tests for all possible cases.
- Testing data consistency helps to fix a bug, but we have to be able to cause an error.

Fuzz testing

Description (Wikipedia)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Motivation

- For advanced algorithm, we are unable to create unit tests for all possible cases.
- Testing data consistency helps to fix a bug, but we have to be able to cause an error.
- Therefore, we provide our program random data, both correct and incorrect.

Fuzz testing

Description (Wikipedia)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Motivation

- For advanced algorithm, we are unable to create unit tests for all possible cases.
- Testing data consistency helps to fix a bug, but we have to be able to cause an error.
- Therefore, we provide our program random data, both correct and incorrect.

Simple example

```
elements = list(range(start, stop, step))
random.shuffle(elements)
TEST(sorted(elements) == list(range(start, stop, step)))
```

Fuzz testing and probability of finding an error

Question

What is the chance that a fuzz test finds an error?

Fuzz testing and probability of finding an error

Question

What is the chance that a fuzz test finds an error?

Infinite monkey theorem

- A monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type any given text, such as the complete works of William Shakespeare.

Fuzz testing and probability of finding an error

Question

What is the chance that a fuzz test finds an error?

Infinite monkey theorem

- A monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type any given text, such as the complete works of William Shakespeare.
- The probability that monkeys filling the entire observable universe would type a single complete work is so tiny that the chance of it occurring during a period of time hundreds of thousands of orders of magnitude longer than the age of the universe is extremely low (but technically not zero).

Fuzz testing and probability of finding an error

Collorary and practice

- A random generator finds a sequence of operations leading to an existing error after sufficient amount of time, assuming the generator can create such a sequence.

Fuzz testing and probability of finding an error

Collorary and practice

- A random generator finds a sequence of operations leading to an existing error after sufficient amount of time, assuming the generator can create such a sequence.
- The necessary amount of time is increasing with the length of input pretty fast.

Fuzz testing and probability of finding an error

Collorary and practice

- A random generator finds a sequence of operations leading to an existing error after sufficient amount of time, assuming the generator can create such a sequence.
- The necessary amount of time is increasing with the length of input pretty fast.
- Most bugs in practice can be cause by many inputs which reduced the time for testing.

Collorary and practice

- A random generator finds a sequence of operations leading to an existing error after sufficient amount of time, assuming the generator can create such a sequence.
- The necessary amount of time is increasing with the length of input pretty fast.
- Most bugs in practice can be cause by many inputs which reduced the time for testing.
- Even fuzz testing cannot find all bugs,
  e.g. using formal verification a bug was found in the standard implementation of Timsort and the smallest input was of size 67 108 864.

Fuzz testing

Example

```
t := TestedTree()
for i in 1:10...0 do
    t.insert(random())
    t.test_data_consistency()
for i in 1:10...0 do
    t.find(random())
for i in 1:10...0 do
    t.delete(random())
    t.test_data_consistency()
```

Fuzz testing

Example

```
t := TestedTree()
for i in 1:10...0 do
    t.insert(random())
    t.test_data_consistency()
for i in 1:10...0 do
    t.find(random())
for i in 1:10...0 do
    t.delete(random())
    t.test_data_consistency()
```

Why this is not sufficient?

- Data can be stored correctly even if the tree is loosing elements.

Example

```
t := TestedTree()
for i in 1:10 . . . 0 do
    t.insert(random())
    t.test_data_consistency()
for i in 1:10 . . . 0 do
    t.find(random())
for i in 1:10 . . . 0 do
    t.delete(random())
    t.test_data_consistency()
```

Why this is not sufficient?

- Data can be stored correctly even if the tree is loosing elements.
- A bug may be caused by an insertion after some deletions.

Fuzz testing

Example

```
t := TestedTree()
for i in 1:10 . . . 0 do
    t.insert(random())
    t.test_data_consistency()
for i in 1:10 . . . 0 do
    t.find(random())
for i in 1:10 . . . 0 do
    t.delete(random())
    t.test_data_consistency()
```

Why this is not sufficient?

- Data can be stored correctly even if the tree is loosing elements.
- A bug may be caused by an insertion after some deletions.
- The correctness of query operations should be tested but how do we determine the correct answer?

Test storing expected elements

Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

Test storing expected elements

Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

Approach

- Choose an appropriate container from the standard library, e.g. array or a hash table

Test storing expected elements

Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

Approach

- Choose an appropriate container from the standard library, e.g. array or a hash table
- Create for testing an auxiliary class containing an instance of
  - our tested data structure and
  - selected (referee) container.

Test storing expected elements

Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

Approach

- Choose an appropriate container from the standard library, e.g. array or a hash table
- Create for testing an auxiliary class containing an instance of
  - our tested data structure and
  - selected (referee) container.
- Ensure that both data structures contains the same elements:
  - New elements are inserted into both data structures
  - Deleted elements are removed from both data structures

Test storing expected elements

Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

Approach

- Choose an appropriate container from the standard library, e.g. array or a hash table
- Create for testing an auxiliary class containing an instance of
  - our tested data structure and
  - selected (referee) container.
- Ensure that both data structures contains the same elements:
  - New elements are inserted into both data structures
  - Deleted elements are removed from both data structures
- Query operations compare results from both data structures
  - The referee data structure does not need to efficient.
  - It is sufficient that it provides correct answers.

Test storing expected elements

Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

Approach

- Choose an appropriate container from the standard library, e.g. array or a hash table
- Create for testing an auxiliary class containing an instance of
  - our tested data structure and
  - selected (referee) container.
- Ensure that both data structures contains the same elements:
  - New elements are inserted into both data structures
  - Deleted elements are removed from both data structures
- Query operations compare results from both data structures
  - The referee data structure does not need to efficient.
  - It is sufficient that it provides correct answers.
- We also tests that both data structures contains the same elements using e.g.
  - simultaneous traverses of both data structures if elements are stored in a sorted order
  - traversing elements in one data structure and marking them in the other

Example of a testing class

```
class tester {
  my_buggy_tree examinee;
  set referee;

  void insert(element e) {
    examinee.insert(e);
    referee.insert(e);
    compare_content();
  }

  void compare_content() {
    iterator e = examinee.begin();
    for(auto r : referee) { // Iterate elements in both trees
      TEST(e != examinee.end());
      TEST(e.key() == r.key()); // Comparing data in elements
      e = e.next();
    }
    TEST(n == examinee.end());
    TEST(examinee.size() == referee.size());
  }

  void find(key k) {
    TEST(examinee.find(e) == referee.find(e));
  }
};
```

Data generation

Goal

- Create data for insertion, deletions, queries ...
- Choose order of these operations

Data generation

Goal

- Create data for insertion, deletions, queries . . .
- Choose order of these operations

Simple sequetial data generation

- Insert (e.g.) even numbers
- Try to find all number, including number out of range
- Try to delete all numbers
- Results of these operations are known, so we also verify our testing class

Data generation

Goal

- Create data for insertion, deletions, queries . . .
- Choose order of these operations

Simple sequetial data generation

- Insert (e.g.) even numbers
- Try to find all number, including number out of range
- Try to delete all numbers
- Results of these operations are known, so we also verify our testing class

Příklad

```
void test_sequence(int length) {
  tester t;
  for(int i = 1; i <= length; i++)
    t.insert(2*i);
  for(int i = 0; i <= 2*length+5; i++)
    TEST(t.find(i) == (iseven(i) && 1 <= i <= 2*length));
  for(int i = 0; i <= 2*length+5; i++)
    t.remove(i);
}
```

Random data generation

Methods

- Insert random elements
- Search for random elements which usually are not found
- Search for elements randomly selected from the referee which has to be found
- Delete both random elements and randomly selected from the referee

Random data generation

Methods

- Insert random elements
- Search for random elements which usually are not found
- Search for elements randomly selected from the referee which has to be found
- Delete both random elements and randomly selected from the referee

Příklad

```
void test_random_data(int length) {
  tester t;
  for(int i = 0; i < length; i++)
    t.insert(random()); // Duplicit keys may be inserted
  for(key k : t.referee) {
    TEST(t.find(k)); // Test a key should be stored
    t.find(random()); // Test a key which is not most likely stored
  }
  while(!t.empty()) {
    TEST(t.remove(random(t.referee))); // Random element from referee
    t.remove(random());
  }
}
```

Calling functions in a random order

Motivation

Some errors may be caused by sequence alternating insertions and deletions

Calling functions in a random order

Motivation

Some errors may be caused by sequence alternating insertions and deletions

Example of approach

```
void test_random_order(int length) {
  tester t;
  for(int i = 0; i < length; i++) {
    switch(random() % 7) {
      case 0: t.insert(random());
      case 1: TEST(!t.insert(random(t.referee)));
      case 2: t.find(random());
      case 3: TEST(t.find(random(t.referee)));
      case 4: t.remove(random());
      case 5: TEST(t.remove(random(t.referee)));
      case 6: ...
    }
  }
}
```

Fuzz testing of advanced algorithm

Questions

- How to obtain a correct solution for a random input?

Fuzz testing of advanced algorithm

Questions

- How to obtain a correct solution for a random input?
- How to test correctness if there are multiple correct solutions?
  - E.g. there may be many shortest paths between given pair of vertices.

Fuzz testing of advanced algorithm

Questions

- How to obtain a correct solution for a random input?
- How to test correctness if there are multiple correct solutions?
    - E.g. there may be many shortest paths between given pair of vertices.

How to obtain a correct solution?

- Use available libraries if possible
    - E.g. sorting or red-black trees are implemented in standard libraries

Fuzz testing of advanced algorithm

Questions

- How to obtain a correct solution for a random input?
- How to test correctness if there are multiple correct solutions?
  - E.g. there may be many shortest paths between given pair of vertices.

How to obtain a correct solution?

- Use available libraries if possible
  - E.g. sorting or red-black trees are implemented in standard libraries
- A given problem may have other slower but easier algorithms
  - E.g. for testing Strassen algorithm, use definition of matrix multiplication

Fuzz testing of advanced algorithm

Questions

- How to obtain a correct solution for a random input?
- How to test correctness if there are multiple correct solutions?
  - E.g. there may be many shortest paths between given pair of vertices.

How to obtain a correct solution?

- Use available libraries if possible
  - E.g. sorting or red-black trees are implemented in standard libraries
- A given problem may have other slower but easier algorithms
  - E.g. for testing Strassen algorithm, use definition of matrix multiplication
- Faster algorithms are often obtained by improving slower ones
  - Keep the slower algorithm for testing

Fuzz testing of advanced algorithm

Questions

- How to obtain a correct solution for a random input?
- How to test correctness if there are multiple correct solutions?
  - E.g. there may be many shortest paths between given pair of vertices.

How to obtain a correct solution?

- Use available libraries if possible
  - E.g. sorting or red-black trees are implemented in standard libraries
- A given problem may have other slower but easier algorithms
  - E.g. for testing Strassen algorithm, use definition of matrix multiplication
- Faster algorithms are often obtained by improving slower ones
  - Keep the slower algorithm for testing
- Use theoretical knowledge of a problem or an algorithm

Optimization problem

The task is to find a solution satisfying given constraints and minimizing or maximizing given objective function.

- Shortest path
- Minimum spanning tree
- Maximum network flow

Testing optimization algorithms

Optimization problem

The task is to find a solution satisfying given constraints and minimizing or maximizing given objective function.

- Shortest path
- Minimum spanning tree
- Maximum network flow

Verifying correctness of solutions of a random input

- Test satisfaction of all constraints

Testing optimization algorithms

Optimization problem

The task is to find a solution satisfying given constraints and minimizing or maximizing given objective function.

- Shortest path
- Minimum spanning tree
- Maximum network flow

Verifying correctness of solutions of a random input

- Test satisfaction of all constraints
- Test the optimality if possible

Shortest path problem

Output of Dijkstra's algorithm

For every vertex, a predecessor on a shortest path and its length is provided.

Output of Dijkstra's algorithm

For every vertex, a predecessor on a shortest path and its length is provided.

Veriying correctness of a solution

- Feasibility: Traverse a path from a goal vertex using predecessors until the starting vertex is reached and verify that a vertex and its predecessor are connected by an edge and that lengths are correct.

Shortest path problem

Output of Dijkstra's algorithm

For every vertex, a predecessor on a shortest path and its length is provided.

Veriying correctness of a solution

- Feasibility: Traverse a path from a goal vertex using predecessors until the starting vertex is reached and verify that a vertex and its predecessor are connected by an edge and that lengths are correct.
- Optimality: Use the following theorem.

Shortest path problem

Output of Dijkstra's algorithm

For every vertex, a predecessor on a shortest path and its length is provided.

Veriying correctness of a solution

- Feasibility: Traverse a path from a goal vertex using predecessors until the starting vertex is reached and verify that a vertex and its predecessor are connected by an edge and that lengths are correct.
- Optimality: Use the following theorem.

Theorem

Let G be a directed graph, l_{uv} be length of edge uv and s be a starting vertex. Then d_u are lengths of a shortest path from s to all vertices u if and only if

- $d_s = 0$
- $d_v \leq l_{vu} + d_u$ holds for every edge uv
- for every vertex v except s there exists an edge uv such that $d_v = l_{vu} + d_u$.

Shortest path problem: Testing optimality

Theorem

Let G be a directed graph, l_{uv} be length of edge uv and s be a starting vertex. Then d_u are lengths of a shortest path from s to all vertices u if and only if

- $d_s = 0$
- $d_v \leq l_{vu} + d_u$ holds for every edge uv
- for every vertex v except s there exists an edge uv such that $d_v = l_{vu} + d_u$.

Theorem

Let G be a directed graph, l_{uv} be length of edge uv and s be a starting vertex. Then d_u are lengths of a shortest path from s to all vertices u if and only if

- $d_s = 0$
- $d_v \leq l_{vu} + d_u$ holds for every edge uv
- for every vertex v except s there exists an edge uv such that $d_v = l_{vu} + d_u$.

Testing optimality

```
void dijkstra_test(Graph *g, int start, int *dist) {
  vector<bool> has_predecessor(g->nv, false);
  has_predecessor[start] = true;
  TEST(dist[start] == 0);
  for(u : g->vertices)
    for(v : g->out_neighbors(u)) {
      TEST(dist[v] <= g->length(v,u) + dist[u]);
      if(dist[u] + g->length(u,v) == dist[v])
        has_predecessor[v] = true;
    }
  for(u : g->vertices)
    TEST(has_predecessor[u]);
}
```

How would you test correctness of solutions for these problems

- Sorting

How would you test correctness of solutions for these problems

- Sorting
- Heap

How would you test correctness of solutions for these problems

- Sorting
- Heap
- Searching a substring in a string

How would you test correctness of solutions for these problems

- Sorting
- Heap
- Searching a substring in a string
- Connected components

How would you test correctness of solutions for these problems

- Sorting
- Heap
- Searching a substring in a string
- Connected components
- Minimum spanning tree

How would you test correctness of solutions for these problems

- Sorting
- Heap
- Searching a substring in a string
- Connected components
- Minimum spanning tree
- Maximum flow

How would you test correctness of solutions for these problems

- Sorting
- Heap
- Searching a substring in a string
- Connected components
- Minimum spanning tree
- Maximum flow
- Maximum matching

How would you test correctness of solutions for these problems

- Sorting
- Heap
- Searching a substring in a string
- Connected components
- Minimum spanning tree
- Maximum flow
- Maximum matching
- Travelling salesman problem

Tasks for next week

The first assignment: Left-leaning Red-black trees

- Design and implement API
- Design and implement data representation
- Write unit tests with small and also large number of elements
- Write test checking correctness of data representation
- Implement operations insert and find k-th element
- Implement operation delete
- Write fuzz tests