Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis

René Just[1,2] & Gregory M. Kapfhammer[3] & Franz Schweiggert[2]

[1]University of Washington, USA

[2]Ulm University, Germany

[3]Allegheny College, USA

23rd International Symposium on Software Reliability Engineering

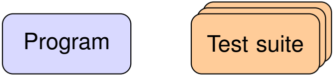
November 28, 2012

Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

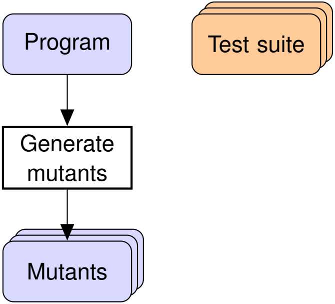
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)



Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

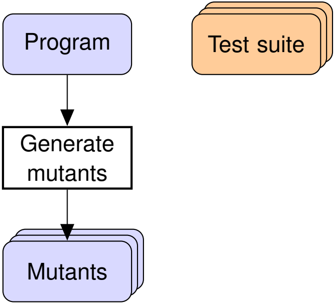


Mutation Analysis Background

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

Original

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

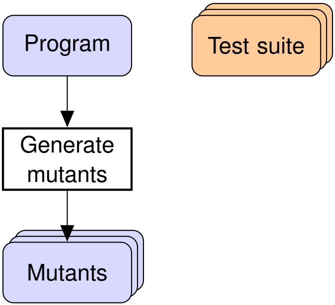


Mutation Analysis Background

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

Original

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)



Mutation Analysis Background

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

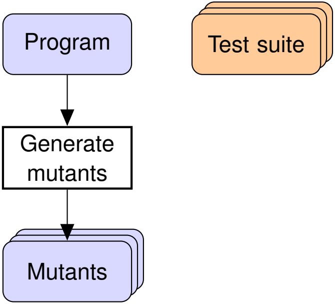
Original

```
public int max(int a, int b){
   int max = a;
   if (b>=a){
      max=b;
   }
   return max;
}
```

Contains a small syntactic change

Mutant

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)



Mutation Analysis Background

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

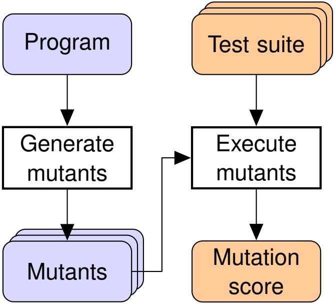
Original

```
public int max(int a, int b){
   int max = a;
   if (b>=a){
      max=b;
   }
   return max;
}
```

Contains a small syntactic change

Mutant

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)



Mutation Analysis is Expensive

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

Original

Mutation Analysis is Expensive

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

Original

```
if (b <  a)
if (b <= a)
if (b >= a)
if (b != a)
if (b == a)
```

Mutation Analysis is Expensive

```
public int max(int a, int b){
    int max = a;
    if (b>a){
        max=b;
    }
    return max;
}
```

Original

```
if (b <  a)
if (b <= a)
if (b >= a)
if (b != a)
if (b == a)
```

Many mutants can be generated for large programs

Mutation Analysis is Expensive

```
public int max(int a, int b){
    int max = a;
    if (b>a){
        max=b;
    }
    return max;
}
```

Original

```
if (b < a)
if (b <= a)
if (b >= a)
if (b != a)
if (b == a)
```

Large programs include comprehensive test suites

Many mutants can be generated for large programs

Mutation Analysis is Expensive

```
public int max(int a, int b){
   int max = a;
   if (b>a){
      max=b;
   }
   return max;
}
```

Original

```
if (b <  a)
if (b <= a)
if (b >= a)
if (b != a)
if (b == a)
```

Large programs include comprehensive test suites

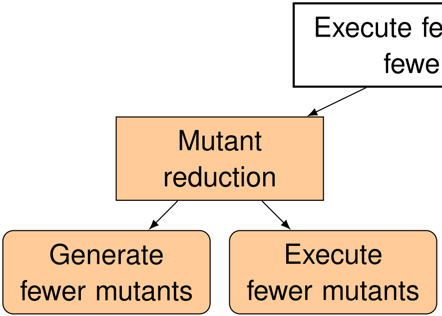
Many mutants can be generated for large programs

Executing the entire test suite for all mutants in large programs is prohibitive!

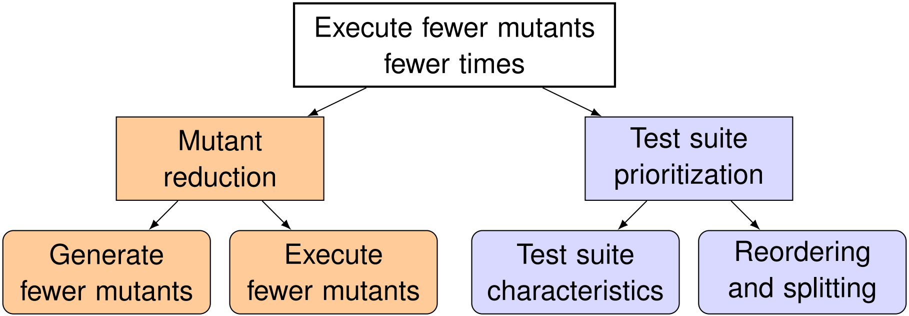
Overview: Efficient Mutation Analysis

$\boxed{\text{Execute fewer mutants fewer times}}$

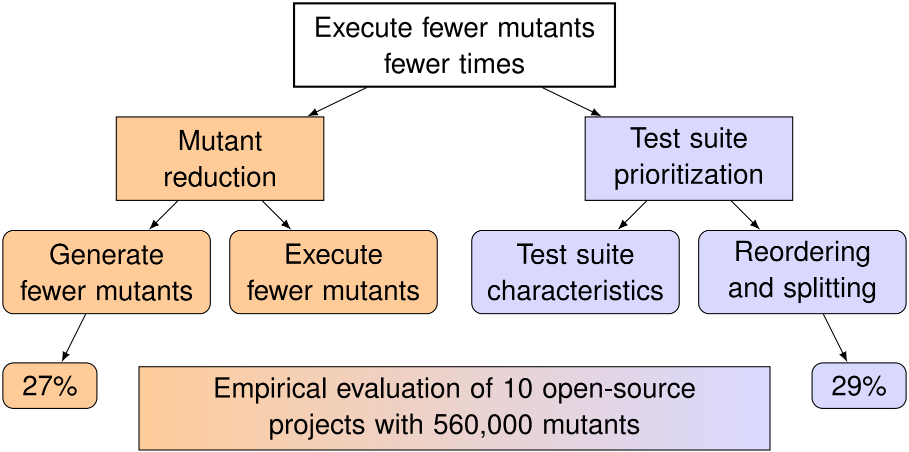
Overview: Efficient Mutation Analysis



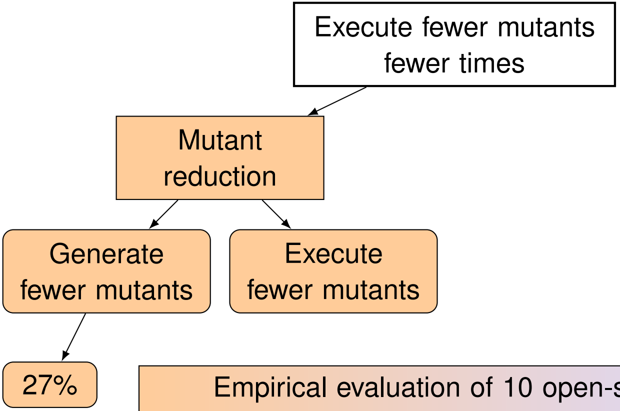
Overview: Efficient Mutation Analysis



Overview: Efficient Mutation Analysis



Reduction of Mutants



Empirical evaluation of 10 open-source projects with 560,000 mutants

Reduce Number of Generated Mutants

Mutation operators may introduce redundancy:

- Redundant mutants are subsumed by other mutants
  - `a + b` $\mapsto$ `a - b` (replace binary operator)
  - `a + b` $\mapsto$ `a + (-b)` (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants

Reduce Number of Generated Mutants

Mutation operators may introduce redundancy:

- Redundant mutants are subsumed by other mutants
  - `a + b` $\mapsto$ `a - b` (replace binary operator)
  - `a + b` $\mapsto$ `a + (-b)` (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants

Number of generated mutants reduced by 27%

Reduce Number of Generated Mutants

Mutation operators may introduce redundancy:

- Redundant mutants are subsumed by other mutants
  - `a + b` $\mapsto$ `a - b` (replace binary operator)
  - `a + b` $\mapsto$ `a + (-b)` (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants

Number of generated mutants reduced by 27%

More than 410,000 generated mutants remaining

Reduce Number of Generated Mutants

Mutation operators may introduce redundancy:

- Redundant mutants are subsumed by other mutants
  - `a + b` $\mapsto$ `a - b` (replace binary operator)
  - `a + b` $\mapsto$ `a + (-b)` (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants

Number of generated mutants reduced by 27%

More than 410,000 generated mutants remaining

Executing all non-redundant mutants is still prohibitive!

Reduce Number of Executed Mutants

Exploit necessary conditions:

- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Reduce Number of Executed Mutants

Exploit necessary conditions:

- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Total reduction of executed mutants of more than 50%

Reduce Number of Executed Mutants

Exploit necessary conditions:

- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Total reduction of executed mutants of more than 50%

Mutation analysis runtime still up to 13 hours

Reduce Number of Executed Mutants

Exploit necessary conditions:

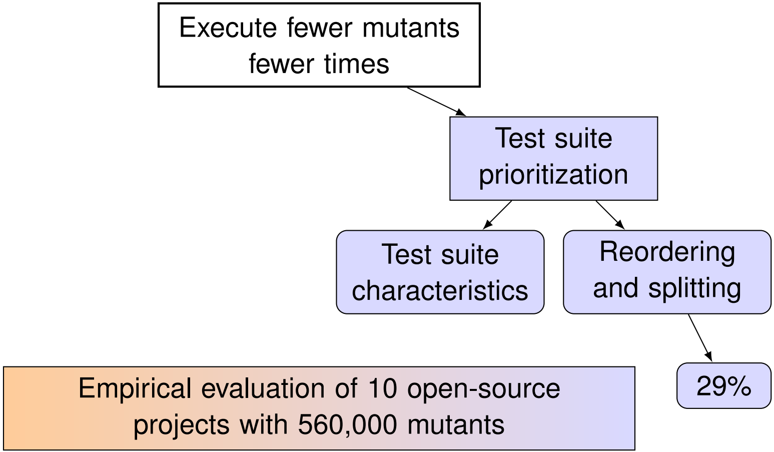
- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Total reduction of executed mutants of more than 50%

Mutation analysis runtime still up to 13 hours

Further optimizations beyond the reduction of mutants are necessary!

Optimized Workflow for Mutation Analysis



Motivating Example for Reordering

Mutants:
1, 2, 3, 4, 5

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

Motivating Example for Reordering

	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Mutants: 1, 2, 3, 4, 5			
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

| t_1 | t_2 | t_3 | : |

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

t_1 t_2 t_3 :	1	2	3	4	5

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

t_1 t_2 t_3 :						
1	2	3	4	5	3	4

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

t_1 t_2 t_3 :								
	1	2	3	4	5	3	4	3

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

$t_1 t_2 t_3$: | 1 | 2 | 3 | 4 | 5 | 3 | 4 | 3 |

$t_3 t_2 t_1$:

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

$t_1\,t_2\,t_3$: 1 | 2 | 3 | 4 | 5 | 3 | 4 | 3

$t_3\,t_2\,t_1$: 1 | 2 | 3

Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

$t_1 t_2 t_3$: 1 2 3 4 5 3 4 3

$t_3 t_2 t_1$: 1 2 3 1 4 5

Motivating Example for Reordering

	Test case t_1: 5 seconds	Test case t_2: 2 seconds	Test case t_3: 1 second
Mutants: 1, 2, 3, 4, 5			
Covered:	1, 2, 3, 4, 5	1, 3, 4, 5	1, 2, 3
Detected:	1, 2, 5	1, 4	3

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

$t_1\,t_2\,t_3$:	1	2	3	4	5	3	4	3
$t_3\,t_2\,t_1$:	1	2	3	1	4	5	2	5

Motivating Example for Splitting

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds
Covered:	1, 2, 3, 4, 5
Detected:	1, 2, 5

Motivating Example for Splitting

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_1': 3 seconds	Test case t_1'': 2 seconds
Covered:	1, 2, 3, 4, 5	1, 2, 3, 4	2, 3, 4, 5
Detected:	1, 2, 5	1, 2	2, 5

Motivating Example for Splitting

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_1': 3 seconds	Test case t_1'': 2 seconds
Covered:	1, 2, 3, 4, 5	1, 2, 3, 4	2, 3, 4, 5
Detected:	1, 2, 5	1, 2	2, 5

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

t_1	:	1	2	3	4	5

Motivating Example for Splitting

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_1': 3 seconds	Test case t_1'': 2 seconds
Covered:	1, 2, 3, 4, 5	1, 2, 3, 4	2, 3, 4, 5
Detected:	1, 2, 5	1, 2	2, 5

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

t_1	:	1	2	3	4	5
t_1' t_1''	:					

Motivating Example for Splitting

Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_1': 3 seconds	Test case t_1'': 2 seconds
Covered:	1, 2, 3, 4, 5	1, 2, 3, 4	2, 3, 4, 5
Detected:	1, 2, 5	1, 2	2, 5

- Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

t_1 :	1	2	3	4	5
t_1' t_1'' :	1	2	3	4	

Motivating Example for Splitting

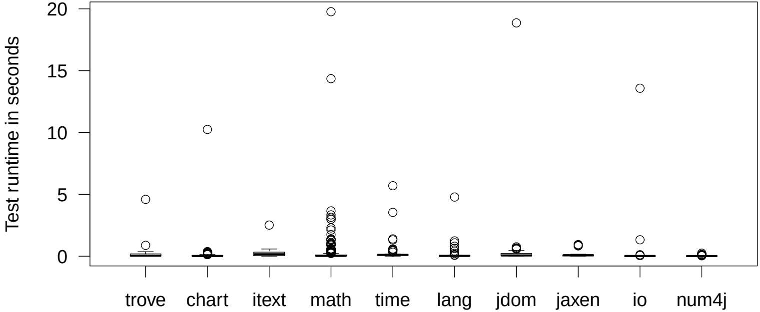
Mutants: 1, 2, 3, 4, 5	Test case t_1: 5 seconds	Test case t_1': 3 seconds	Test case t_1'': 2 seconds
Covered:	1, 2, 3, 4, 5	1, 2, 3, 4	2, 3, 4, 5
Detected:	1, 2, 5	1, 2	2, 5

- Once a mutant is detected, it is not executed again!

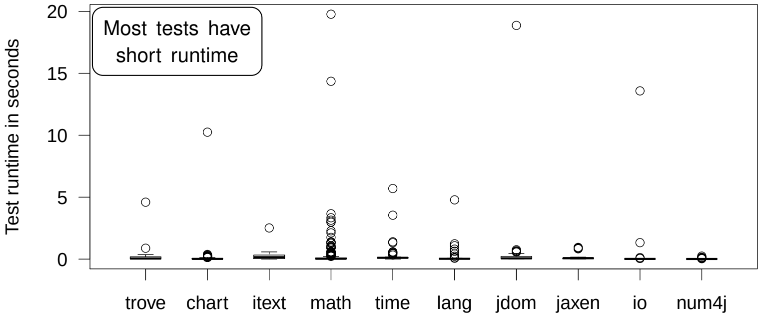
Executed mutants and total runtime:

t_1 :	1	2	3	4	5			
t_1' t_1'' :	1	2	3	4	3	4	5	

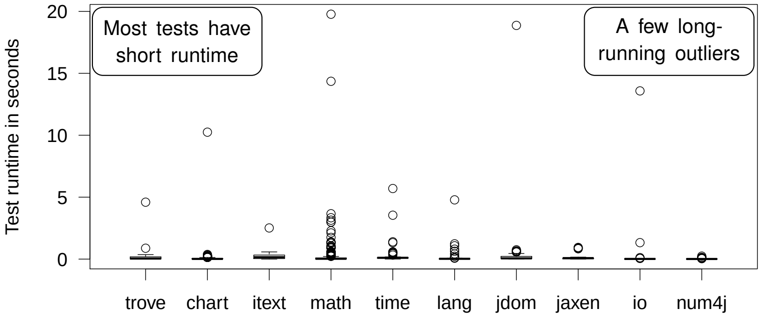
Runtime Distribution of Tests within Test Suites



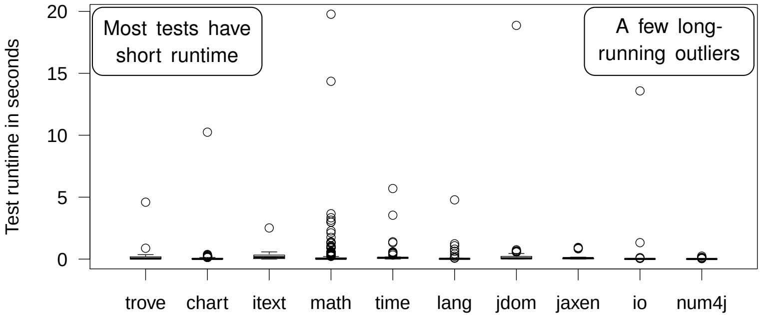
Runtime Distribution of Tests within Test Suites



Runtime Distribution of Tests within Test Suites



Runtime Distribution of Tests within Test Suites



A few tests constitute most of the total runtime:
Reduce number of executions for these tests

Mutation Coverage Overlap

- Overlap measures the similarity of a test case with its enclosing test suite
- Pair-wise comparison of test cases is infeasible

Mutation Coverage Overlap

- Overlap measures the similarity of a test case with its enclosing test suite
- Pair-wise comparison of test cases is infeasible

Definition: *Overlap* $O(t_i, T), \ t_i \in T$

$$O(t_i, T) := \begin{cases} 1, & |Cov(t_i)| = 0 \\ \frac{|Cov(t_i) \cap Cov(T \setminus t_i)|}{|Cov(t_i)|}, & |Cov(t_i)| > 0 \end{cases}$$

Mutation Coverage Overlap

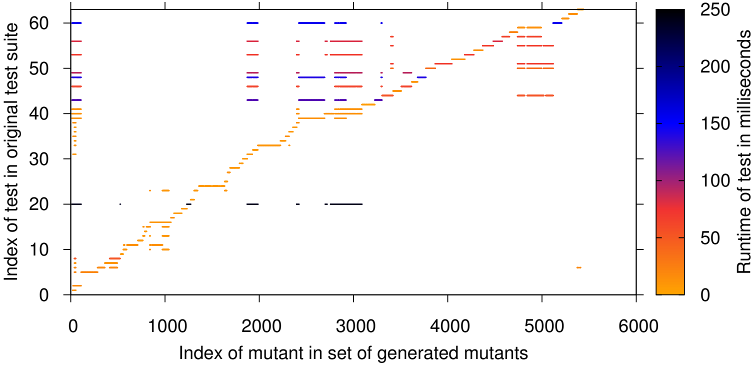
- Overlap measures the similarity of a test case with its enclosing test suite
- Pair-wise comparison of test cases is infeasible

Definition: *Overlap* $O(t_i, T), \ t_i \in T$

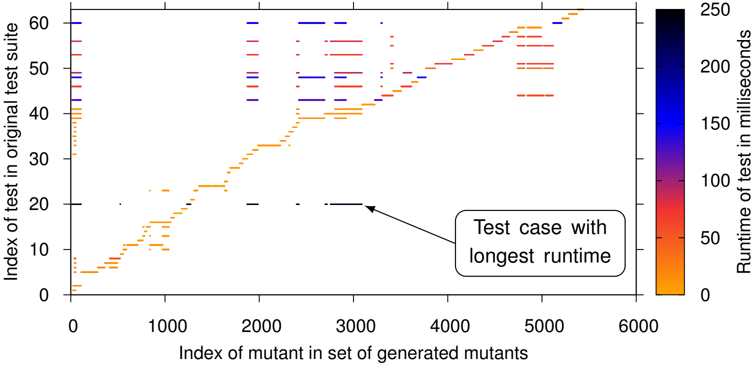
$$O(t_i, T) := \begin{cases} 1, & |Cov(t_i)| = 0 \\ \frac{|Cov(t_i) \cap Cov(T \setminus t_i)|}{|Cov(t_i)|}, & |Cov(t_i)| > 0 \end{cases}$$

Most of the test cases exhibit high overlap:
Does test runtime correlate with overlap?

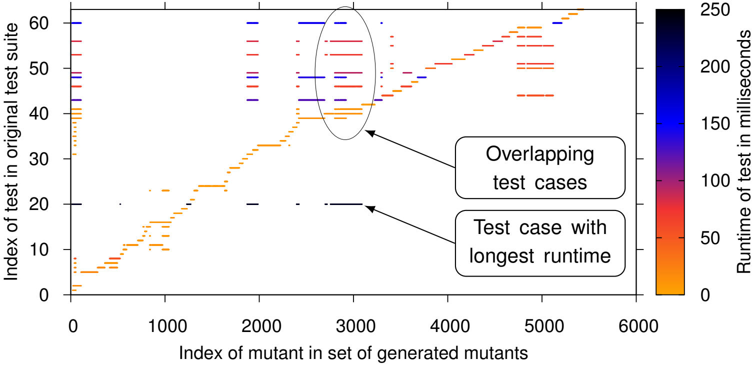
Correlation of Test Runtime and Mutation Coverage



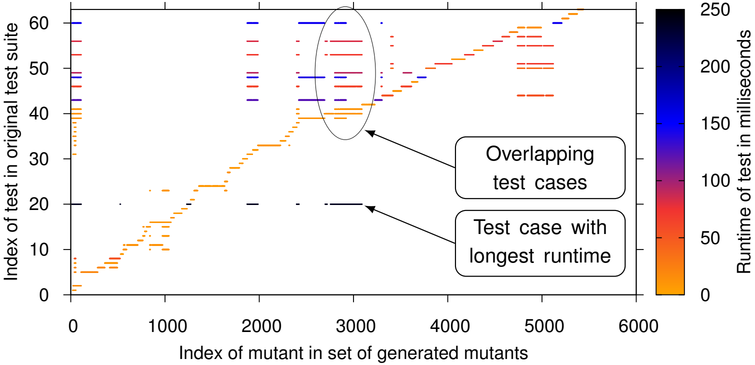
Correlation of Test Runtime and Mutation Coverage



Correlation of Test Runtime and Mutation Coverage

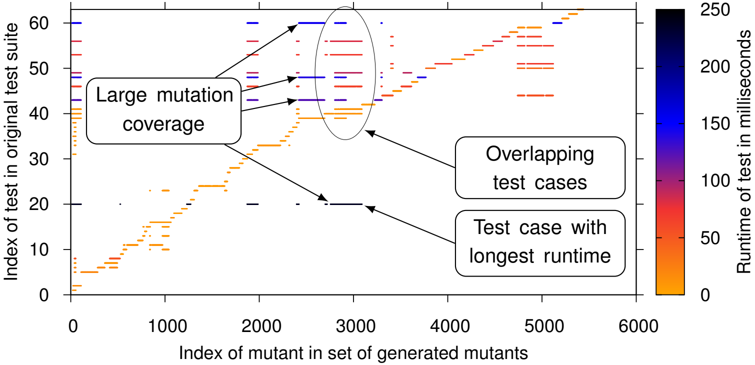


Correlation of Test Runtime and Mutation Coverage



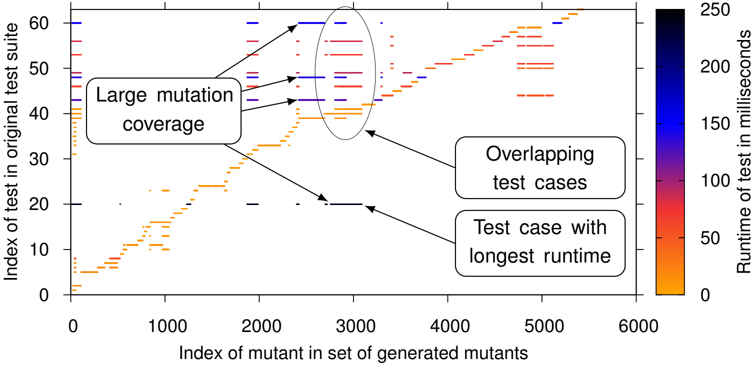
Reorder to exploit mutation coverage overlap

Correlation of Test Runtime and Mutation Coverage



Reorder to exploit mutation coverage overlap

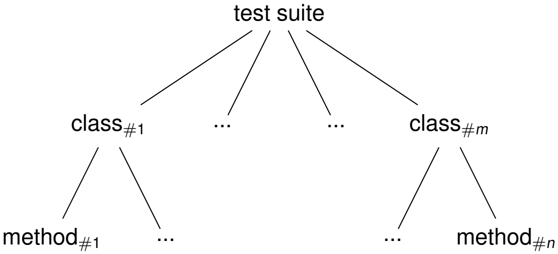
Correlation of Test Runtime and Mutation Coverage



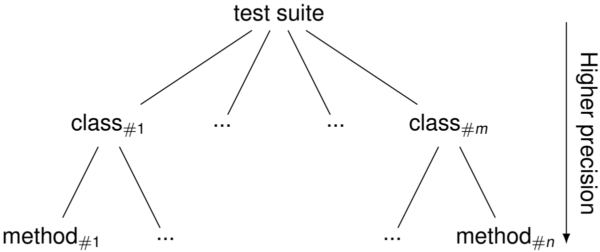
Reorder to exploit mutation coverage overlap

Split test cases to increase coverage precision

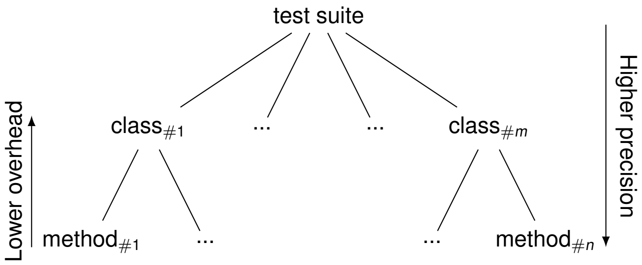
Mutation Coverage of Test suites



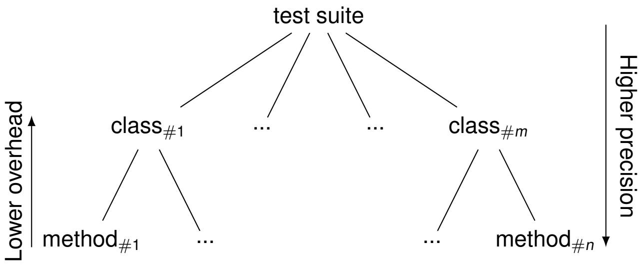
Mutation Coverage of Test suites



Mutation Coverage of Test suites



Mutation Coverage of Test suites

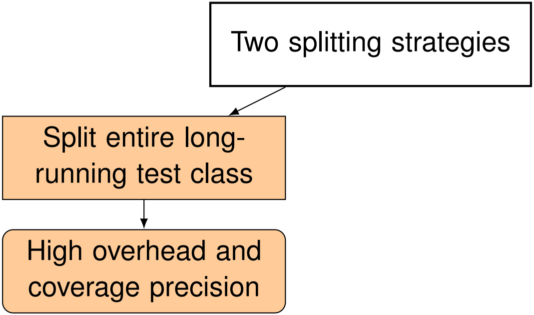


Only split long-running test classes

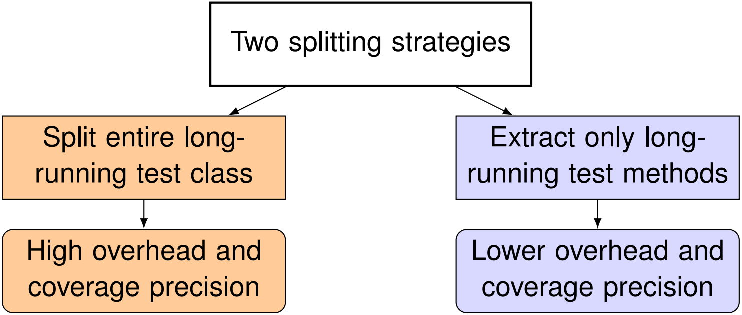
Splitting Test Classes

Two splitting strategies

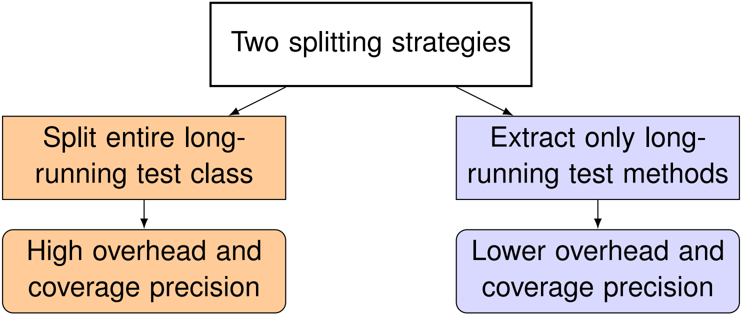
Splitting Test Classes



Splitting Test Classes

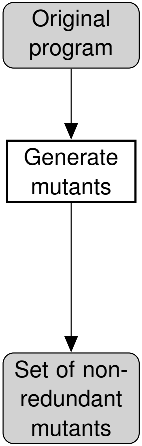


Splitting Test Classes

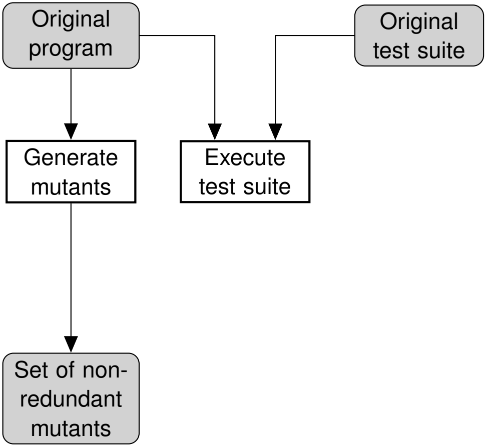


Trade-off between overhead and precision:
Splitting based on threshold for test runtime

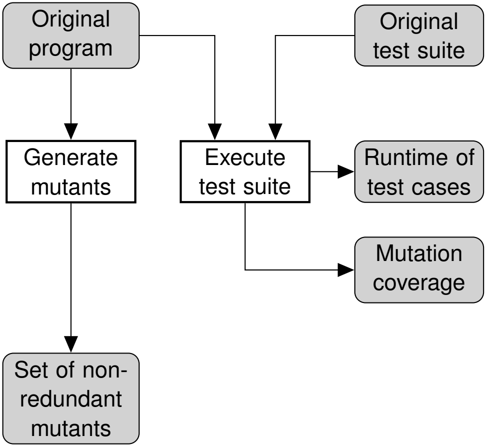
Optimized workflow



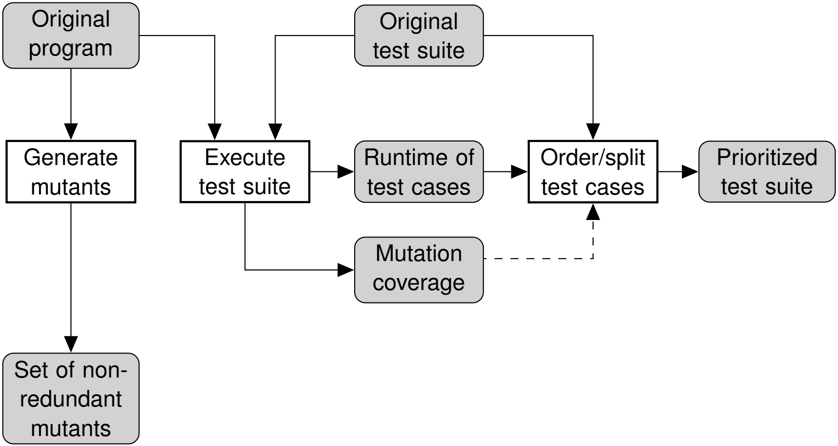
Optimized workflow



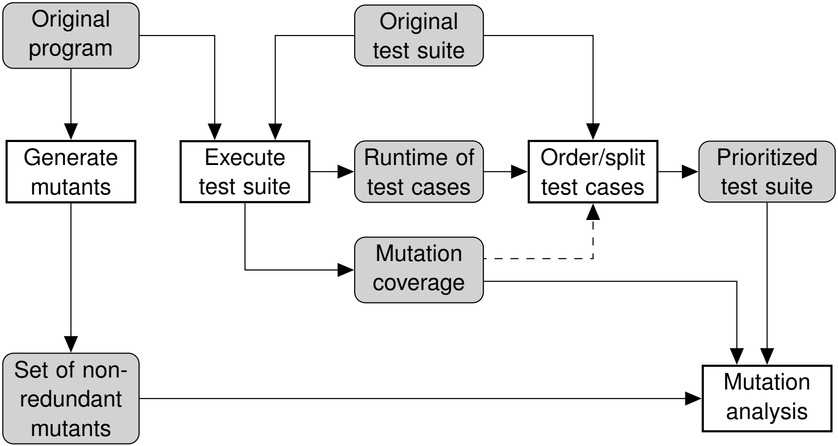
Optimized workflow



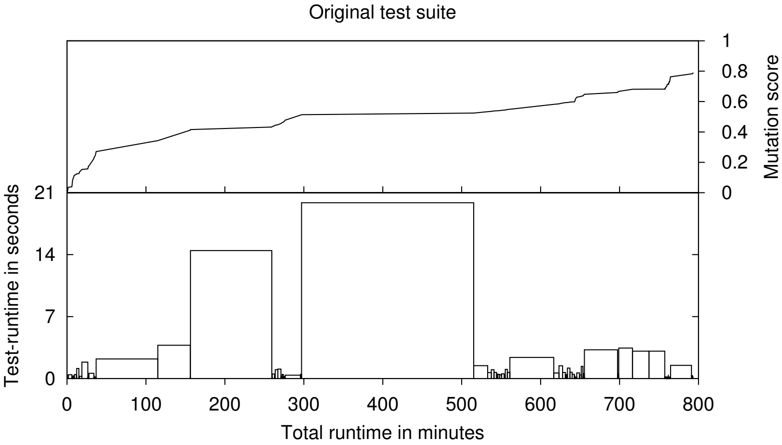
Optimized workflow



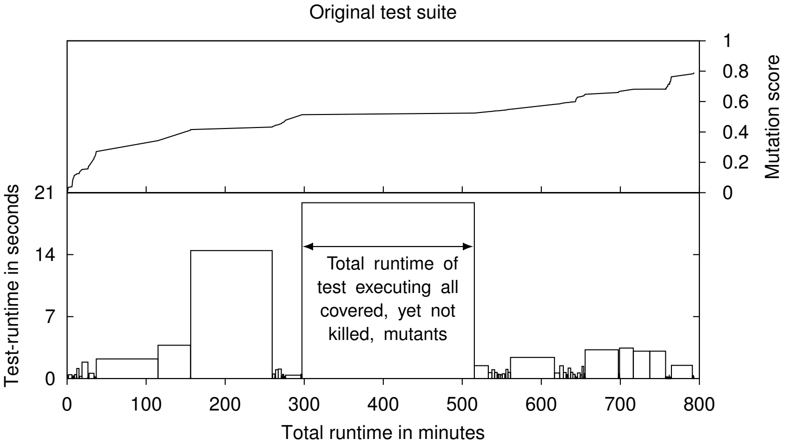
Optimized workflow



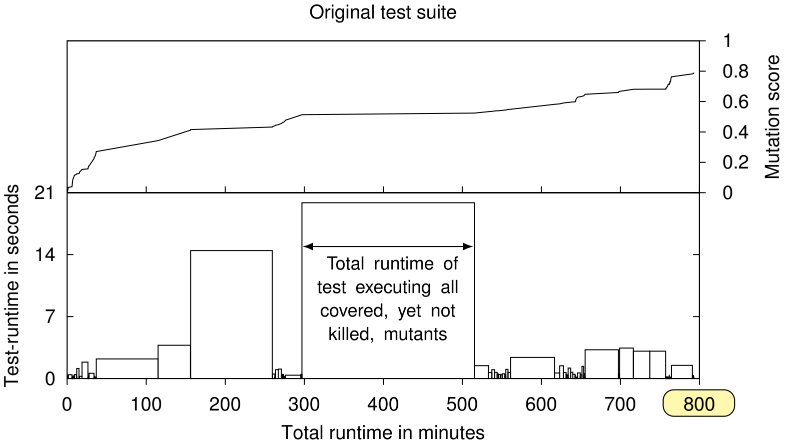
Example with Original Test Suite



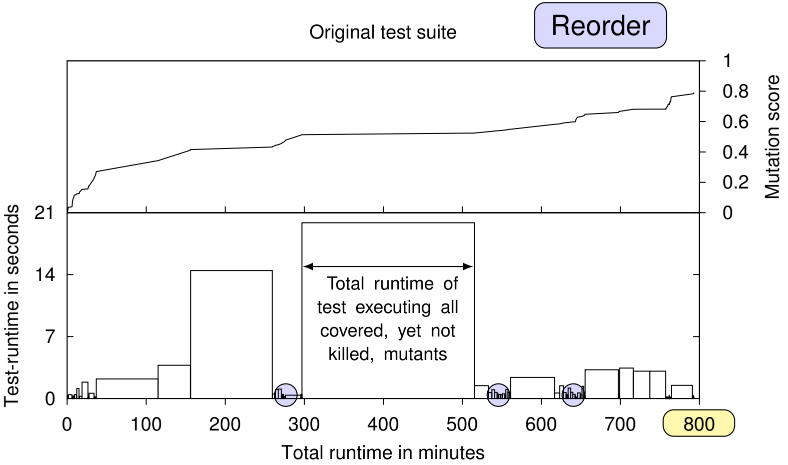
Example with Original Test Suite



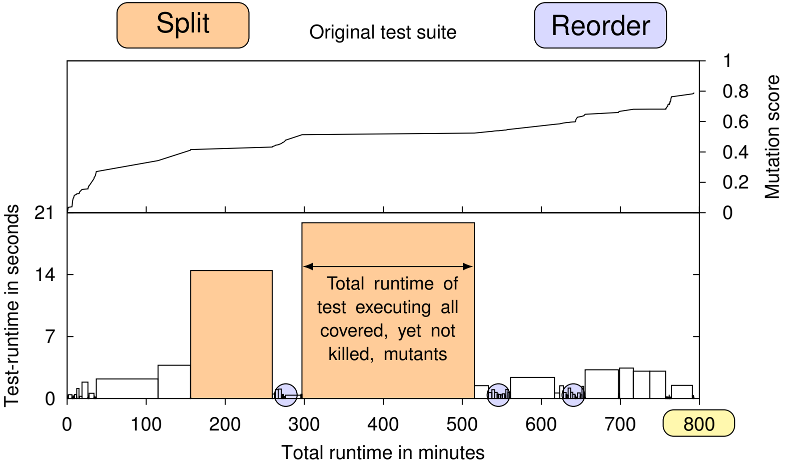
Example with Original Test Suite



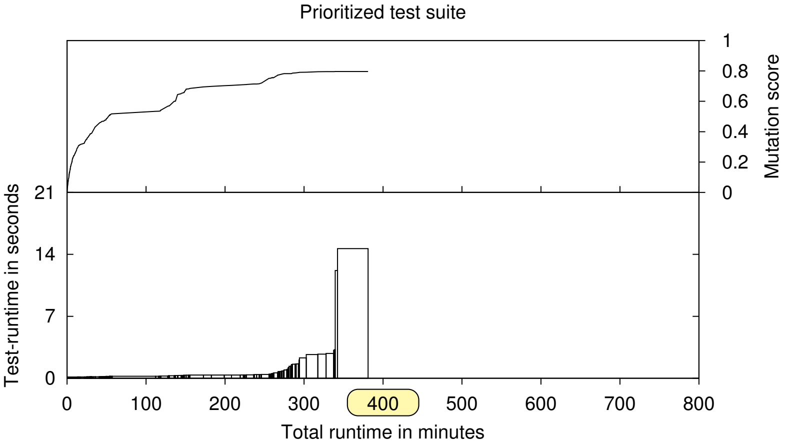
Example with Original Test Suite



Example with Original Test Suite



Example with Prioritized Test Suite



Empirical Results

Reordering:

- Reordering decreases the runtime by 20%

Splitting strategies:

- Extracting long test methods reduces the runtime by 29%
- Splitting entire test classes increases the runtime by 27%

Splitting may increase runtime if:

- Test suite has a very low mutation detection rate
- Test methods exhibit huge mutation coverage overlap

Empirical Results

Reordering:

- Reordering decreases the runtime by 20%

Splitting strategies:

- Extracting long test methods reduces the runtime by 29%
- Splitting entire test classes increases the runtime by 27%

Splitting may increase runtime if:

- Test suite has a very low mutation detection rate
- Test methods exhibit huge mutation coverage overlap

Prioritizing test suites improves the efficiency of mutation analysis by 29% on average!

Related Work

Reduction of generated mutants:

- Sufficient mutation operators
  - Offutt et al., TOSEM'96
  - Namin et al., ICSE'08
- Non-redundant mutation operators
  - Kaminski et al., AST'11
  - Just et al., Mutation'12

Mutation-based test suite optimization:

- Test case prioritization
  - Elbaum et al. TSE'02
  - Do and Rothermel, TSE'06

Related Work

Reduction of generated mutants:

- Sufficient mutation operators
  - Offutt et al., TOSEM'96
  - Namin et al., ICSE'08
- Non-redundant mutation operators
  - Kaminski et al., AST'11
  - Just et al., Mutation'12

Still contain redundancies

Mutation-based test suite optimization:

- Test case prioritization
  - Elbaum et al. TSE'02
  - Do and Rothermel, TSE'06

Related Work

Reduction of generated mutants:

- Sufficient mutation operators
  - Offutt et al., TOSEM'96
  - Namin et al., ICSE'08
- Non-redundant mutation operators
  - Kaminski et al., AST'11
  - Just et al., Mutation'12

Still contain redundancies

Used in empirical study

Mutation-based test suite optimization:

- Test case prioritization
  - Elbaum et al. TSE'02
  - Do and Rothermel, TSE'06

Related Work

Reduction of generated mutants:

- Sufficient mutation operators
  - Offutt et al., TOSEM'96
  - Namin et al., ICSE'08
- Non-redundant mutation operators
  - Kaminski et al., AST'11
  - Just et al., Mutation'12

Still contain redundancies

Used in empirical study

Mutation-based test suite optimization:

- Test case prioritization
  - Elbaum et al. TSE'02
  - Do and Rothermel, TSE'06

Do not address efficiency

Conclusions

Reduction of mutants:

- Non-redundant operators reduce number of mutants by 27%

Test suite characteristics:

- Most of the tests exhibit mutation coverage overlap
- Notable difference in runtime of tests

Optimized workflow:

- Exploits mutation coverage overlap and runtime differences
- Further reduces total runtime of mutation analysis by 29%

Conclusions

Reduction of mutants:

- Non-redundant operators reduce number of mutants by 27%

Test suite characteristics:

- Most of the tests exhibit mutation coverage overlap
- Notable difference in runtime of tests

Optimized workflow:

- Exploits mutation coverage overlap and runtime differences
- Further reduces total runtime of mutation analysis by 29%

Non-redundant operators and optimized workflow implemented in the MAJOR mutation system