# Preliminary Test Suite Reduction

Vitaly Kozyura and Sebastian Wieczorek
SAP AG
Darmstadt, Germany
v.kozyura;sebastian.wieczorek@sap.com

***Abstract*—Test suite reduction is an activity which reduces test suites while maintaining their coverage properties. This problem is equivalent to the set covering problem and therefore NP-complete. Many strategies for solving the problem are known. They are usually applied to minimizing the number of action calls within a given test suite for a certain coverage goal. While some algorithms like branch and bound compute an exact minimal solution, other algorithms like the greedy approach compute an approximation for the minimal set of actions. In this work, we deal with the problem of efficient test suite reduction in industrial practice. For this purpose, we introduce the concept of preliminary test suite reduction. Its aim is to reduce redundancy in test suites before starting the actual reduction. In the paper, we further describe experimental results that give implication on how the proposed technique can reduce the runtime of test suite reduction in the industrial practice.**

***Keywords*—*MBT; test suite reduction; industrial case study.***

## I. Introduction

Automatically generating tests suites from formal specifications as advertised by model-based testing (MBT) is regarded as a potential innovation leap in industrial software quality assurance. Most MBT approaches are running in two phases. In the first phase, vast amounts of test cases are generated for an inserted model until coverage of model entities is achieved. In the second phase, a subset of these test cases is selected with the aim to preserve the targeted coverage and therefore the assumed fault-uncovering capabilities [1]. This activity is called test suite reduction.

Fig. 1 represents a test model, that will be used as a running example in order to illustrate the reduction techniques. The test model is given in form of a finite state machine. The states are depicted as circles and the actions as named arrows. $q0$ is a start state and the state with two nested circles is an end state. A valid test is a sequence of actions starting in $q0$ and ending in the end state. As the coverage criteria we choose to cover all action names. Please note that in practice the execution of actions may be constrained by input and system data and may have additional side effects on data apart from state changes [2]. How side effects may be handled later on during test case execution is sketched in Section V-A.

Assuming that a model checking technique (breadth-first search) is used for the test generation in phase one,

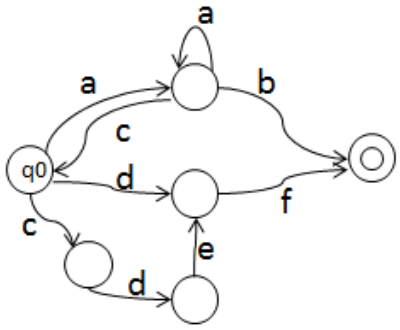


Fig. 1. Running Example.

the following test suite may be obtained:

$$[ab, df, aab, aaab, acab, acdf, cdef].$$

It can be noted that the derived test suite covers all action names, but it is not the minimal test suite (measured by the overall number of actions). How this test suite can be reduced, while preserving action name coverage will be discussed in the consequent sections.

The problem of test suite reduction is largely discussed in the literature. There are papers, where the general test suite reduction activity is described [3], [4]. Further work on how to apply 0/1-Integer linear programming to the test suite reduction problem [5] or how to improve the Greedy heuristics [6], [7], [8] can be found. In [9], [10] there are approaches using multi-objective optimization functions, whereas in [11] an approach based on genetic algorithms is introduced. Some empirical results for test suite reductions have been reported in [12].

This paper is motivated by the problem of *efficient* test suite reduction. We introduce a preliminary test suite reduction technique, aiming to reduce the runtime of the test suite reduction procedure and investigate its applicability to test suite reduction in the industrial practice of MBT.

First, we provide the industrial context for test suite reduction (Section II), then we describe the test suite reduction problem in detail (Section III). In this paper, we consider two existing approaches to the test suite reduction problem: approximative technique (described on the example of a greedy algorithm [13]) and the technique searching for an exact solution (described on the example of a branch and bound algorithm [14]).

In Section IV, we define the concept of *preliminary test suite reduction*. In essence, the goal of applying preliminary reduction is to make the overall activity of test suite

 ISBN: 978-1-61208-307-0 

reduction more efficient. In Section V we provide a few experimental results in order to illustrate the proposed technique and also to discuss its applicability for practical test suite reduction. Section VI concludes the paper.

## II. Industrial Context

In the software industry, model-based test automation is one of the most promising approaches for increasing the efficiency of testing. Various commercial vendors emerged that offer tools and consulting, maintain user groups, and organize industrial conferences. Although the various competing commercial tools utilize alternative test generation concepts like model checking, theorem proving, or random walks, the overall process of producing test suites is similar. It consists of two phases:

1) Test suite generation - deriving test cases from the model, usually until test coverage is reached.
2) Test suite reduction - calculating a subset of test cases that maintains test coverage.

Especially in industrial settings, the second phase is indispensable, because of the significant manual effort associated with test case concretization [15]. This transformation from abstract test cases to executable test scripts usually follows the keyword-driven testing principles. Keyword-driven testing uses keywords in the test cases, in addition to data. Each keyword corresponds to a fragment of a test script (the adapter code), which allows the test execution tool to translate a sequence of keywords and data values into executable tests [1].

Generated test suites usually contain a large number of redundant test cases, that would unnecessarily increase the concretization effort. For example, the initially generated test suite given in Section I contains 7 test cases and 23 actions. However, various subsets of the given test suite exist that are preserving the defined coverage of all action names. As to be shown in Section III, the optimal solution only contains 2 test cases and 6 actions.

The most common reasons for redundancy in test suites are the presence of loops in the test model as well as multiple occurrences of equal action names. The given example contains both. Further, some test generation approaches deliberately continue to create test cases despite the fact that the computed test suite already meets the coverage criteria. Often, this enables better reduction results, as it increases the variety of test cases.

In industrial practice, various additional sources of redundancy may exist that are not connected to the model structure or test generator. For example, it is often the case that after initial thorough testing, test suites with reduced coverage requirements are created to lower the execution runtime of regression tests. In order to avoid the effort of test re-generation and especially the potential test concretization of additional test cases, usually the already generated and used test suite is reduced again. Also, the merging of generated test suites with manually designed or legacy test cases often occurs in practice.

## III. Test Suite Reduction

As described, for a provided model the obtained test suite can contain a very large number of test cases. Aim of the test suite reduction is to select a subset, which preserves the targeted coverage and therefore the assumed fault-uncovering capabilities. This activity can be formulated like follows [7]:

*Given*: A test suite TS, a set of test case requirements $r_1, r_2, ..., r_n$ that must be satisfied to provide the desired testing coverage of the program, and subsets of TS, $T_1, T_2, ..., T_n$, one associated with each of the $r_i$'s such that any one of the test cases $\mathrm{tc}_j$ belonging to $T_i$ can be used to test $r_i$.

*Problem*: Find a representative set of test cases from TS that satisfies all of the $r_i$'s.

The test suite reduction problem can be considered as a hitting set problem — the problem of finding the hitting set having minimum cardinality, which is equivalent to the set cover problem and is known to be NP-complete [16]. The standard way of solving the hitting set problem is a restatement into a 0/1-Integer linear program. Afterwards, this can either be exactly solved by using a technique like branch and bound algorithm or approximately, for example by applying different variations of Greedy heuristics [17], [13]. In the following, both approaches are described.

### A. Greedy Algorithm

We use a classical implementation of the Greedy algorithm which has already been known for some time. Even though the Greedy algorithm computes an approximation, [13] showed that the result cannot become arbitrarily bad. In fact the upper bound for the performance guarantee only depends on the number of requirements.

The algorithm iteratively constructs subsets $\mathrm{TS}_i \subseteq \mathrm{TS}$, which will produce a complete test suite after termination of the algorithm. Until all requirements are met, the algorithm does the following: It computes the set of all test cases, for which the number of additional action calls is maximal. Then, it picks one of these ($tc$) at random. This test case is afterwards added to $\mathrm{TS}_{i+1} = \mathrm{TS}_i \cup \{tc\}$ and the requirements are updated appropriately.

The algorithm has a linear time complexity $O(|\mathrm{TS}|)$ with respect to the size of a test suite $|\mathrm{TS}|$ to be optimized.

Using a greedy algorithm on the initially generated test suite of the example given in Section I, the following reduced test suite can be obtained, containing 3 test cases and 10 actions:

$$[acdf, ab, cdef].$$

### B. Branch and Bound Algorithm

In order to find an exact solution to the test reduction problem, we use the branch and bound variation (Balas-algorithm) described in [14], which allows one to compute an optimal result.

The algorithm identifies all possible subsets of $\mathrm{TS} = \{\mathrm{tc}_1, \ldots, \mathrm{tc}_m\}$ with arrays $(n_1, \ldots, n_m)$. Here $n_i = 1$

 ISBN: 978-1-61208-307-0 

means, that $tc_i$ is part of the subset, while $n_i = 0$ means, that it is not. To check these arrays systematically, they are organized as a binary tree. At the root node, no decisions have been made, whereas any node on level $i$ represents a certain choice of the first $i$ bits. The node $(n_1, \ldots, n_i)$ is identified with the test suite $\{tc_j : n_j = 1\}$. During the so-called *pruning*, it is checked for each constructed node if this node can be safely discarded from the tree.

In worst case, the algorithm has an exponential run time with respect to the size of a test suite |TS| to be optimized.

In the case of using a branch and bound algorithm on the initially generated test suite of the example given in Section I, the following reduced test suite is obtained, containing 2 test cases and 6 actions:

$$[ab, cdef].$$

## IV. Preliminary Reduction

As described, test suite reduction is necessary because initially generated test suites usually contain far more test cases than necessary to achieve certain coverage goals. On the other side, also test suite reduction itself may be a costly operation. As discussed in Section III, the runtime of test suite reductions can vary from linear to exponential depending on how exact the solution should be. As test engineers expect fast feedback from test automation tools, e.g. in order to not loose their focus, each runtime improvement for test suite reduction retaining the exactness of the solution can be very valuable from the practical point of view.

In this section, we would like to introduce the concept of preliminary removing redundancy in an initial test suite in order to reduce the runtime of the actual reduction procedure. The proposed preliminary reduction is applied before the actual optimization procedure starts. We define prelimiary redundancy as follows.

*Definition*: Given a test suite TS, we say that the test case tc is *redundant* if there exists another test case $tc'$ with $|tc'| \leq |tc|$ and for each requirement $r_i$ it holds $r_i(tc) \Rightarrow r_i(tc')$.

Following algorithm can be used in order to delete the redundant test cases from the test suite before starting the actual test suite reduction.

*Algorithm*: Our preliminary reduction procedure compares the test cases from TS with each other and deletes all redundant test cases. This results in a time complexity of $O(|TS|^2)$.

Applying the above definition to the example from Section I, a given test case is redundant if another test case with equal or less actions exists that covers at least the same action names. For instance, the test *aaab* is redundant because $|ab| < |aaab|$ while both cover the same set of action names. The test suite obtained after applying the preliminary reduction to the example consequently is

$$[ab, df, acab, acdf, cdef].$$

As mentioned before, the greedy algorithm has a linear run time complexity, whereas the branch and bound algorithm has an exponential run time complexity with respect to the size of a test suite |TS| to be optimized. Based on this information, the decision should be to always use preliminary reduction when constructing an exact minimal set of test cases and to never use it when constructing an approximative solution. However, in practice there are two factors, which can influence this decision:

- The typical regions of the test suite sizes and the grades of the run time complexity curves in these regions for each reduction algorithm.
- The typical rate of redundancy in the considered test suites.

Combining this two factors can lead to the situations where either it would not be reasonable to use preliminary reduction at all (also when constructing an exact minimal set of test cases) or where it would be reasonable to use it even when constructing an approximative solution to the test suite reduction problem.

After being able to construct examples for all these situations, we decided to check the applicability of preliminary optimization for both exact and approximative approaches to the test suite reduction problem in the industrial practice. The purpose is to derive a practically applicable guidance on the basis of the experiments with typical test suites from the industrial context.

## V. Experimental Results

In this section, we present experimental results illustrating the applicability of preliminary reduction technique for optimizing the test suite reduction procedure. We start with a description of the industrial testing setup: how test models are constructed and how the tests are generated. Then, we provide the experimental results and discuss the usability of preliminary reduction in the industrial practice. Finally, possible threats to validity are described.

### A. Test Setup

The testing approach we consider in this paper leverages a keyword-driven testing framework, as described in [18]. A model editor is implemented on top of the framework in order to facilitate an automated test generation. Test models are represented by transition state machines enhanced with data flow and global data.

As an input for the experiments, we have collected a number of test models, which were designed on the basis of industrial case studies. These test models were created for system testing. In practice, system testing is based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is most appropriate to carry out the tests, as the system should be validated as a whole and only using access points that are available to the prospect user.

Keyword-based testing for UI is mostly done by utilizing capture/replay functionality, which is provided by

 ISBN: 978-1-61208-307-0 

standard test automation tools. These tools are monitoring user interactions on the interface that can reproduce the execution of the recorded sequence of events. These captured scripts commonly allow data flexibility by exchanging the concrete values (used during capturing) with variables that can be initialized independently.

Further, the recorded scripts can be combined in so-called scenarios. A scenario is a sequence of recorded scripts working on a predefined set of data. Global variables are used in scenarios to organize the data flow. Their function is to store output values of a captured script and make it available as input for another. In Fig. 2 an example of the described data flow is given. The value of the local output variable *A.out* of *Script A* is written to the global variable $X$ and later mapped to the local input variable *B.in* of *Script B*.

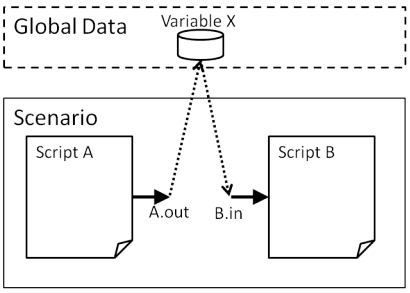


Fig. 2. Data flow in a scenario.

In order to allow a calculation of an exact minimal test suite and further to get realistic statements for the context of our work, most of the chosen samples are small- or intermediate-sized (I-IX). We also included one interesting border case example (NA), which is too large to be optimized with algorithms of the branch and bound type.

All computations were performed on an AMD Opteron (tm) Quad Core with 2.60 GHz and 32 Gigabytes of RAM.

### B. Selected Results

In Table I and Table II, we present selected results of our experiments demonstrating the applicability of the preliminary reduction for an approximative and an exact construction of the reduced test suite. In Table I, we compare approximating greedy algorithm with and without preliminary reduction and in Table II we do the comparison for the branch and bound algorithm. The tables have the following columns: number of an example (Example), size of a test suite to be optimized (|TS|), size of a test suite obtained after preliminary reduction (|TS|(PR)) , and the run time for the each algorithm with and without preliminary reduction (Time(PR) / Time). The time is measured in seconds.

### C. Discussion

As it can be seen in Table I and Table II, the preliminary reduction does not improve the approximative greedy algorithm, except for one border case (NA), where the optimal test suite (containing 3 test cases) is already obtained from 3160 test cases after the preliminary reduction. In contrast, branch and bound algorithms, which have exponential complexity, perform already better for the size of the fourth example (IV) if using the preliminary reduction.

TABLE I. Greedy algorithm with and without preliminary reduction.

| Example | \|TS\| | \|TS\|(PR) | Time | Time(PR) |
|---|---|---|---|---|
| I | 15 | 14 | 0,06 | 0,08 |
| II | 32 | 32 | 0,07 | 0.12 |
| III | 41 | 41 | 0.08 | 0.14 |
| IV | 45 | 24 | 0,09 | 0,16 |
| V | 120 | 120 | 0,10 | 1,05 |
| VI | 132 | 111 | 0,62 | 2,12 |
| VII | 289 | 203 | 1,36 | 7,63 |
| VIII | 512 | 336 | 3,67 | 20,13 |
| IX | 625 | 402 | 4,83 | 44, 37 |
| NA | 3160 | 3 | 32,13 | 3,57 |

TABLE II. Branch and bound algorithm with and without preliminary reduction.

| Example | \|TS\| | \|TS\|(PR) | Time | Time(PR) |
|---|---|---|---|---|
| I | 15 | 14 | 0,06 | 0,07 |
| II | 32 | 32 | 3,66 | 3,67 |
| III | 41 | 41 | 9,05 | 9,06 |
| IV | 45 | 24 | 0,13 | 0,11 |
| V | 120 | 120 | 77,49 | 77,50 |
| VI | 132 | 111 | 268,78 | 155,14 |
| VII | 289 | 203 | 511,02 | 403,71 |
| VIII | 512 | 336 | 1353,37 | 810,81 |
| IX | 625 | 402 | 1733,16 | 886,95 |
| NA | 3160 | 3 | - | 3,59 |

This means that in the industrial practice the most challenging test suites are located in the size region, where there is enough redundancy to make preliminary reduction an efficient technique.

In order to better understand the presented results, we also provide scatter diagrams representing the ratios for speedup or slowdown in the run time for the examples I-IX (Fig. 3). For each diagram on the x-axis, there are numbers of tests in the test suites and on the y-axis there are the corresponding ratios between run times with and without preliminary reduction (*Time(PR)/Time* ). The thick horizontal lines define the areas where *Time(PR)/Time = 1*, i.e., the preliminary reduction brings neither advantages nor disadvantages from the run time perspective. The skew lines (trend lines) represent the correlations between the test suite size and the runtime ratio described above.

From the presented diagrams, it can be seen that in the case of approximative test suite reduction it is not only unsuitable to use preliminary reduction, but the drawback is increasing with the growing size of test suites. Otherwise, for the branch and bound algorithm the trend line shows that the value of using preliminary reduction grows with the growing size of test suites.

It can be seen that it is not always reasonable to apply it for small test suites, but the larger the test suites get the more beneficial it is to apply the preliminary reduction technique. From the practical point of view, the test suites of medium or large size are crucial with respect to run time, whereas for small test suites the possible slowdown is usually not critical.

 ISBN: 978-1-61208-307-0 

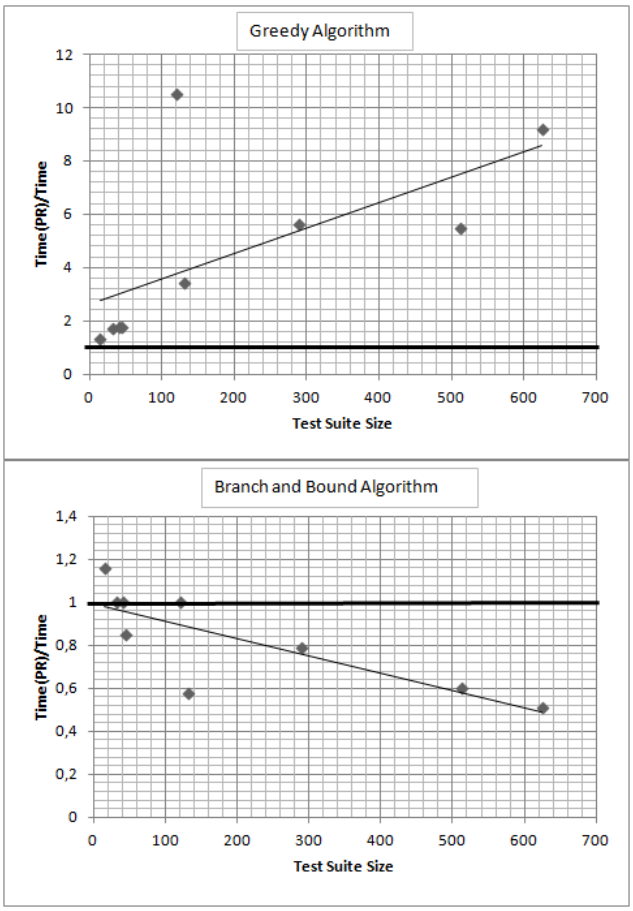


Fig. 3. Relative runtime diagrams.

Therefore, as a bottom line we deducted the following from the experiments:

For the industrial practice, it is recommended to use the preliminary reduction each time a (near) optimal solution for the test suite reduction problem is computed.

### D. Threats to Validity

We realize that a number of experiments in the area of UI-based MBT cannot serve as a proof of applicability for the whole industrial area of MBT. However, we believe that the presented results can be generalized to the practical testing of high-level usage scenarios, where UI-based testing is the most commonly used approach.

## VI. Conclusion

In this paper, we introduced the concept of preliminary test suite reduction and studied how eliminating redundant test cases can accelerate the test suite reduction algorithms.

We further described the industrial context of MBT and provided a collection of common reasons for the existence of the redundancy in test suites. The applicability of preliminary test suite reduction for the industrial practice of MBT is shown, based on a number of experiments from UI-based testing, which is a common way of testing the high-level scenarios.

We applied the preliminary optimization technique to two classical solutions of the test suite reduction problem, namely the branch and bound algorithm, which computes an exact solution in exponential time, and the greedy heuristic, which yields the best approximation possible in polynomial time. In the paper, we presented selected experimental results, which have shown that the approach pays off for branch and bound algorithms, but is rather inefficient for greedy algorithms.

From our industrial experience in MBT, we know that redundancy in the test suite is a common issue which affects test efficiency on various levels. Therefore, it can be an important aspect in practice to apply preliminary optimization in case a (near) optimal solution for the test suite reduction problem should be computed.

## References

[1] M. Utting and B. Legeard, *Practical model-based testing, a tools approach*. Morgan Kaufmann, 2007.

[2] S. Wieczorek, A. Stefanescu, and I. Schieferdecker, "Test data provision for ERP systems," in *Proc. of Int. Conf. on Software Testing (ICST'08)*. IEEE Computer Society, 2008, pp. 396–403.

[3] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, pp. 135–141, 1996.

[4] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *In Proc. Twelfth Int. Conf. Testing Computer Software*, 1995, pp. 111–123.

[5] H. S. Wang, S. R. Hsu, and J. C. Lin, "A generalized optimal path-selection model for structural program testing," *Journal of Systems and Software*, vol. 10, no. 1, pp. 55 – 63, 1989.

[6] R. Gupta and M. L. Soffa, "Compile-time techniques for improving scalar access performance in parallel memories," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 138–148, April 1991.

[7] M. J. Harrold, C. Unwersity, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, pp. 270–285, 1993.

[8] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," vol. 40, no. 5-6, pp. 347–354+, 1998.

[9] J. Black, E. Melachrinoudisl, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 106–115.

[10] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 140–150.

[11] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *Journal of Software Maintenance*, vol. 11, pp. 19–34, January 1999.

[12] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, pp. 219–249, 2002.

[13] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.

[14] J.W. Chinneck, "Practical Optimization: A Gentle Introduction," http://www.sce.carleton.ca/faculty/chinneck/po.html, 2003, chapter 13.

 ISBN: 978-1-61208-307-0 

[15] S. Wieczorek, A. Stefanescu, and I. Schieferdecker, "Model-based integration testing of enterprise services," in *Proc. of Testing: Academic & Industrial Conference - Practice and research techniques (TAICPART'09)*. IEEE Computer Society, 2009, pp. 56–60.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[17] V. V. Vazirani, *Approximation algorithms*. Springer, 2001.

[18] S. Wieczorek and A. Stefanescu, "Improving Testing of Enterprise Systems by Model-Based Testing on Graphical User Interfaces," in *2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE, 2010, pp. 352–357.

 ISBN: 978-1-61208-307-0 