Amalgam - An Alloy extension for equivalence checking of models

John Call, Shreedhar Hardikar, Alisa Maas

December 21, 2013

Abstract

Model-checking, an extremely useful technique for checking the consistency of a system, is hampered by an inability to compare models directly to each other. Applications for model comparison include comparing two models meant to express the same system and comparing different versions of the same model to determine whether they remain consistent. We propose Amalgam, a dynamic, sound and complete tool for model equivalence up to a finite bound. Amalgam exhaustively generates all possible instances of two models using a trivial predicate provided by the user and searches for any instance generated by one model but not by another model, using a graph-solver in order to determine whether two instances are equivalent.

Given a set of API functions to check, we determine whether the results produced by them are equivalent to each other. If any of the functions are missing or do not appear equivalent, we report the largest subset of the API function which are equivalent to each other. If two models produce the same API-function outputs, we consider those instances equivalent. If each instance can be paired with an equivalent instance from the other model, the two models are equivalent. Our results show that two models can be compared to one another relatively quickly, with most time spent on instance generation.

1 Introduction

Our work is built on top of the Alloy 4 - which includes the Alloy modelling language and the Alloy Analyzer - a constraint solver that provides fully automatic simulation and checking. The Alloy language [2] is a simple but expressive first order relational logic which can be used to express complex structural constraints and their behaviors.

An Alloy model consists of one or more files, each containing a single module. The two main constructs of the language are signatures (sigs) and facts. Signatures define object types in the model (stored as a unary relations of atoms) and their fields create relations to other objects. Facts constrain the model by describing properties that are ensured to be true in all instances of the model. Once the model is defined, we can run commands on the Alloy Analyzer to check for the validity of certain commands (using predicates and assertions) and define special relationships between atoms (using functions). However, all
problems are solved within user-specified bounds that restrict the size of the domain, and thus makes the problem tractable.

Alloy 4 relies on Kodkod [5, 6], a constraint solver for relational logics. This solver was originally designed to improve performance of the Alloy system, as the performance previously degraded significantly as the number of constraints increased. In addition to the increase in performance, Kodkod also provides some symmetry-breaking between generated instances. Using Kodkod, the Alloy Analyzer translates the problem to be analyzed into a boolean formula. This formula is handed to a SAT solver, and the solution is translated back by the Alloy Analyzer into the language of the model, generating one or more instances for the model. These are examples (for predicates) and counter-examples (for assertions).

In practice, conceptually equivalent models may be implemented in Alloy differently. Different implementations may employ differently named sigs, or a different number of sigs and fields. Although they represent the same conceptual model, the generated instances are not trivially equivalent to each other. Amalgam is a tool that allows us to automatically compare given Alloy models for equivalence. It is a sound and complete system (up to some finite bound) which has the ability to check two models for equivalence (isomorphism) in an efficient manner. If the models are not isomorphic, it generates a counterexample proving that they are not.

2 Prior Work

2.1 Isomorphic instance detection

Some work [3] has been done on using isomorphism detection to check invariants on models in ways model-checkers cannot. Examples of this are models without bounds on the number of objects or artificially small restrictions on the size of an integer. Both of these restrictions are required for all Alloy models. By finding and removing isomorphic instances, the total number of instances that need to be generated is reduced considerably, allowing for invariants to be checked. This work is similar to Kodkod’s symmetry-breaking techniques.

2.2 Isomorphic subspace removal

There has been some other work done regarding models and isomorphism detection. In particular, work here has focused on detecting isomorphic subspaces during model generation so all models are unique[4]. Research such as this provides motivation for our research by bringing to our attention the fact that others have found isomorphism detection with models to be worthwhile.

3 Three levels of isomorphisms

We introduce three levels of isomorphisms we can consider when comparing two Alloy models - the Universal level (denoted the U-level), the Model level (M-level) and the Functional level (F-level).

At the U-level we consider all relevant sigs, along with their relationships to other sigs without taking into account any predicates or constraints on the
values. The M-Level is more specific than the U-level by also considering all the constraints. Both these levels of comparison have limited use in practice. Because of the variety of ways in which models may be implemented in Alloy, it is very difficult to compare models without a consistent interface to the implementation.

The F-level is the most restrictive level. Instead of examining all the sigs, relations and constraints, we focus on the outputs of functions. The F-level examines the results from a specified subset of functions to determine model isomorphisms. When unspecified, we assume all functions in each model should be examined. In practice, F-level isomorphism is most useful as it may be used to determine whether two models can be considered functionally equivalent, even if they are not structurally similar. This would be useful for determining if two very different representations may encode isomorphic concepts. We proceed by assuming F-level isomorphism check for the rest of the paper.

Given two Alloy models, Amalgam uses function names to match the set of functions in one model to the other. Also, instead of comparing all the functions in both the models, Amalgam only considers functions whose names contain the prefix “api.”. In case there is a function in one model that is not present in the other model, that function may be ignored for comparison.

4 An Example - Modeling a Graph

As an example, let’s consider modelling a directed graph. Fig 1 shows two possible implementation in Alloy. Each implementation defines a number of signatures and facts. In order for Amalgam to be able to compare these models, we first need to add a set of API functions. In order to run the Alloy Analyzer we also need to provide a command - the trivial predicate. Amalgam needs a Trivial predicate, containing an empty expression to generate all possible instances of the given model (also given a finite bound).

Figure 1: NodeGraph/NodeEdge example. These two Alloy models express isomorphic models in very different fashions.

As an example, let’s consider modelling a directed graph. Fig 1 shows two possible implementation in Alloy. Each implementation defines a number of signatures and facts. In order for Amalgam to be able to compare these models, we first need to add a set of API functions. In order to run the Alloy Analyzer we also need to provide a command - the trivial predicate. Amalgam needs a Trivial predicate, containing an empty expression to generate all possible instances of the given model (also given a finite bound).
Since different representations for functionally equivalent models with different structures may need varying amounts of atoms to represent the same instance, Amalgam requires the user to provide the bounds along with the trivial predicate. For example, in Fig 1, the NodeGraph.als implementation uses one atom per graph node, but the NodeEdge.als implementation uses one atom per node and one atom per edge. Hence NodeEdge instances will contain more atoms than NodeGraph instances. This implies that, in order to effectively compare the two models, we must allow for more atoms in the NodeEdge model, while generating the instances. In most cases, adding these predicates is simple given an understanding of the implementation and the expected API.

5 Implementation

A simple approach is to parse both models and find the common subset of api functions between the two models. Then for each model all instances are generated and for each instance the output of each shared function is saved. After this process, the easiest method of determining isomorphism is to check all instances in one model to find an equivalent instance in the other model. This strategy only is effective when there are a small number of generated instances with relatively few atoms in each instance. When models with a large number of instances are checked a large number of comparisons are done between instances that cannot be isomorphic. When an instance has a large number of atoms, the check for instance equivalence requires checking all permutations of the atoms.

We combat the inefficiency of this approach by reducing the number of unnecessary instance comparisons. By categorizing instances, we are able to significantly reduce the number of comparisons. Our approach for categorization is to generate a hash based on the number of atoms in an instance along with the size of each function’s output. Using this hash, we are able to create buckets for the instances. Clearly if any two instances have a different sized output for any api function then they cannot be equivalent. Similarly when two instances have a different number of atoms but the same sizes of function output, there will be two items in one instance that map to the same tuple but map to different atoms in the other instance, making them unequal. When all the instances have been generated and placed into buckets, the comparison can be performed as previously, but with significant performance gains. It is easy to show that the way our hashing system is devised, instances from either model will only fall into the same bucket if they are isomorphic (non-isomorphic instances may also happen to fall into the same bucket). Thus it suffices to check only the current bucket for isomorphic instances. This same strategy is used to check for isomorphic instances whether they are generated by the same model or the other model.

The comparison between instances are still done by checking all permutations of the atoms. Our equivalence check requires the order of the items in a tuple remains the same, but the tuples inside a function’s output can be reordered. This comparison can be converted to a graph isomorphism problem, which allows us to save time compared to checking all permutations. It also allows us to compare groups of instances at once. When doing comparisons of this type no information is returned as to which instances were successfully matched. This means that extra work must be done first to ensure that unmatched instances
are not because their equivalent instance was already matched.

One of the complications with this approach is that while Alloy’s symmetry-breaking removes most duplicate instances, it is not perfect. This means that we cannot simply generate all instances and determine if the instances are isomorphic with a single check. Instead, all duplicates must be removed before groups of instances are compared. By pairing graph isomorphism with hashing, duplicates are removed while generating instances. We check the graph of a newly generated instance against all previously generated instances with the same hash, we call this small graph comparison. Once instance generation is complete, the instances of each bucket can be grouped into a single graph and checked for isomorphism which we call large graph comparison.

**Generating the equivalent graph of an instance**

![Figure 2: An example instance with api_nodes and api_edges functions.](image)

![Figure 3: The graph representation of the instance depicted in Figure 2.](image)

To generate a graph of an instance, nodes must be created to represent the function’s output. By coloring these created nodes, we ensure that two graphs are isomorphic if and only if the instances are equivalent. Firstly, all atoms are colored the same. For each function a unique color is assigned for each location in its tuples. This coloring ensures that a node created for one function’s output cannot be matched with any other function’s output. The function output tuple is created by connecting the nodes created for each element of the tuple to the corresponding atom node along with connecting each node with the nodes that
represent the previous and next node in the current tuple. This connects each
tuple so it cannot be split up and can only be mapped to an equivalent tuple
from the other graph. Finally, when grouping instances, a special node is created
for each instance which connects to all nodes of that instance. This prevents
instances from being split up and avoids false positives.

We can optimize further when comparing one model against multiple differ-
ent models. Using the algorithm as described to this point, each comparison
would require regenerating all instances from the model. This is inefficient. To
prevent this, we allow a model to be created in a mode which is less efficient
for a one-time comparison but much more efficient if it is compared against
multiple models. This is achieved by generating all instances with all functions
and removing initial duplicates. When compared with another model, this list
of instances is pruned using the shared functions. This pruned list is saved in
case another comparison uses the same shared functions.

Finding a counter example, when using graph isomorphism to compare in-
stances, takes additional work and memory usage, as a result it is turned off
by default in Amalgam. When enabled the first instance in a bucket is used
when there are no instances in the other model with the same hash. When a
graph isomorphism check fails a brute force check between the two buckets is
performed to find the instance for which there is no equivalent instance. Finally,
counter examples are displayed using the Alloy4 visualizer.

6 Problems

Our implementation encountered its share of problems. The most difficult tech-
nical problem that we encountered involves the sheer number of instances pro-
duced. Because our implementation relies on generating every possible instance
of both models, the runtime is necessarily a function of the number of instances
in each model. One positive effect this has on our system is that the more
constraints are on a system, the more quickly Amalgam will run. However, on
simpler, less constrained systems, Amalgam can take a very long time to run.
On our NodeEdges/NodeGraph toy example, when run for 5 nodes, we found
that far over 1 million instances were generated (in fact, we were unable to
produce all possible instances without running out of memory). This may be
far too many instances for our system to handle.

However, there is very little we can do to artificially reduce the number of in-
stances. If we put an upper bound on the number of instances, our system would
be neither sound nor complete. This is because instances are not guaranteed to
be generated in any particular order, so even two equivalent models might be
deemed non-equivalent if they have a large number of instances that are gener-
ated in very different orders. Additionally, two non-equivalent instances might
have a single counter example that we could miss by imposing this restriction
on the number of instances.

The best solution we have is for the user to determine this by creating their
trivial predicate in such a way as to limit the number of instances. We also allow
them to set the bitwidth for the model, which allows users to avoid integers
from blowing up the number of instances being created. Every possible int is
generated by the model anywhere an int is permitted, and with the default
bitwidth of 4, this can cause the number of instances to be far above what
Amalgam is able to handle on a typical machine. Most of the time, users working with ints in their system will want to set their bitwidth smaller than 4. We found that models run with a bitwidth of 2 ran relatively quickly (an average of 346 ms using the searchTree models), while models with a bitwidth of 3 took longer, yet finished within a reasonable amount of time. Using a bitwidth of 4 was unsustainable in most cases. Our experiments all use a bitwidth of 2.

The Alloy Analyzer used with Alloy4 does automatically prune the instances to avoid computing multiple copies of the same instance, yet this does not mean that no isomorphic instances are produced within a single model (See figure 4). This complicates our strategy a little. Instead of simply checking that the same number of instances are generated, we have to do more work that this and check that all instances in each model match to at least one instance from another model.

Another technical hitch that we ran into was attempting to determine isomorphism without generating all possible instances. We originally planned to descend to the level of the KodKod model, and then to the CNF-level if necessary. However, we quickly realized that the KodKod API does not currently support comparing models. This may be for a very good reason. We realized that if we were able to compare at the KodKod or even the CNF level and determine easily whether two models were isomorphic, the bound on the number of atoms would disappear. In effect, we would have a sound, complete and unbounded system, which suggests that we would not be able to guarantee termination. Doing either approach would likely involve a lot of overhead in order to infer the ways in which variables relate to one another. Determining CNF formula equivalence is known to be an NP-complete problem, and it would be extremely difficult or impossible to provide a counterexample based on the CNF counterexample.

The final hurdle we had to overcome was an unforeseen complication in our sample data. We had been given old class projects which use Alloy to create several different models. There are a number of reasons why we like these data, but unfortunately, some of the data were from a time before Alloy4. Alloy4 is actually incompatible with the older version of Alloy, Alloy3. Thus, because Amalgam relies on using Alloy4, our system reports a higher level of syntax errors than actually occurred. We made an attempt to solve this problem as well, by writing a script to catch some common Alloy3/Alloy4 differences.
Our script simply replaces the Alloy3 function calls, which use () notation for functions with no parameters, and replaces them with the Alloy4 syntax. This is not comprehensive and will not catch all problems, but it did significantly increase the number of files that we were able to compare. We also found that not all of the provided files contained functions labeled with api. Most of the implementations were quite different, even though they were implementing the same models, so it was not trivial to find the equivalent api functions and add api to the model, nor was it feasible to create a generic form of each of the api functions because the implementation was usually quite different. These files were contained to a single type of model, and we did not run Amalgam on them due to the fact that every one of them would not have any api functions to compare.

7 Methodology

We have run a series of benchmarks in order to determine the speed and accuracy of our system. Our data was obtained as a series of Alloy models written originally as a homework exercises from students enrolled in CS706 at the University of Wisconsin-Madison. These models were written in several semesters, and the data was roughly organized by hash code of semester. All the data were anonymized, and we did not have access to the exact semester each model was generated in, though we were able to compare projects per semester if required.

There were four basic sets of models that we considered: binaryTree models (129 models available), searchTree models (128 models), redBlackTree models (20 models) and avlTree models (110 models). Due to a lack of api functions, all of the redBlackTree models were not suitable for use with Amalgam. We pooled all models of each type, and did not consider which semester each was from. Each model is meant to represent as general of a model as possible - though as these were written by students as homework projects, they did not all match up to our expectations. We did not have the grades available, and we did not have access to the instructor solution. Most of the models had designated api functions which were labeled as such, but some did not. Most of the models appear to have been written with Alloy4 syntax but some were written before that was available. Some students appear to have written models that depend on their previous models. Due to the way we pooled the files by model, such behavior was not supported and likely contributes to compiler errors.

In order to make these examples work with Amalgam, we modified the models as minimally as possible. Specifically, we manually added trivial predicates to each mode (running each for 4), due to the fact that each model could theoretically require extremely different trivial predicates in order to create the same instances, and we ran a script to fix up Alloy3-style function calls. In practice, we found that all trivial predicates could be identical. Amalgam does not look for a predicate called “Trivial” - rather, it runs the very first predicate it sees, so all predicates were added to the tops of the files. In our experiments, we found that some of the models did not have api functions, and some added addition functions prefixed with api not present in most models. We did not adjust for this, but Amalgam does detect this circumstance and reports this.

We ran Amalgam on all the types of models in order to determine Amalgam’s effectiveness. We consider this our typical use case, in a sense: users will create
a model and wish to determine whether it matches some specification. In this case, the specification would be in the form of an Alloy model created to serve as the solution. In another case, it might be a previous version of the Alloy model. This previous version might represent the end goal: the user might wish to make sure that the newest version is still functionally equivalent. Alternatively, it might represent a prior version containing a bug. In that case, if the two models are equivalent, this is a negative result. Although Amalgam can be used to test two apparently different models for similarities, we expect this to be an unusual use case.

We compared each Amalgam model against all other models of the same type. We determined the average amount of time it takes to generate the large and small graphs, the average (total) time to compare two models from that set, and gathered statistics about how many models in each set were isomorphic and in what ways isomorphism failed. The results can be found in Tables 1, 2 and 3. As you can see, the graph-checker itself typically catches very few non-isomorphisms. In general, our hashing strategy is able to catch most non-isomorphic models. This is a positive result, because the graph-checker can be an expensive algorithm to run, as it falls into the realm of likely NP-complete problems (no known polynomial time solution exists, but it has not been proven to be NP-complete). It also means that future optimizations to Amalgam should be done at the level of speeding up the generation of instances or the hashing algorithm itself. Picking a new graph solver might result in some performance improvement, but any improvement would not be as significant.

Table 1: Amalgam Runtime Per Model Type

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Avg Comparison Time</th>
<th>Avg time for large graphs</th>
<th>Avg time for small graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>searchTree</td>
<td>346 ms</td>
<td>1.38 ms</td>
<td>56.36 ms</td>
</tr>
<tr>
<td>binaryTree</td>
<td>328 ms</td>
<td>3.49 ms</td>
<td>105.21 ms</td>
</tr>
<tr>
<td>avlTree</td>
<td>156 ms</td>
<td>3.02 ms</td>
<td>65.99 ms</td>
</tr>
</tbody>
</table>

Overall, our comparison time per model is reasonable. We ran our comparisons without using the optimization for comparing a single model to a set of models in order to better gauge the average time a comparison takes. However, the runtime does not just depend on the speed of the algorithm itself. It also depends on how efficiently Alloy can generate instances for a particular model, and how many instances that model contains. Most of our time right now is spent generating instances, which is actually a positive result. If the Alloy Analyzer improves its instance generation time, Amalgam will also improve on any system using the new version of the Alloy Analyzer. Some of the models will take noticeably longer to generate instances. One binaryTree model took so long to compare that we examined the model itself to explain the amount of time the comparison took and found that each binaryTree model contained its entire subtree. This increased the number of items in the relation, and there was a corresponding increase in the amount of time instance generation took. Another particularly slow binaryTree model was storing an int at every node in the tree, but the bitwidth was set to 4, causing extreme slowdown.

Determining the correctness of our system was more difficult. It is possible to show that, assuming we correctly convert the instances into graphs and assuming we use a correct graph implementation, our system is both sound and
Table 2: Types of Isomorphism and Non-Isomorphism Found

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Avg instances found</th>
<th>Isomorphic</th>
<th>Not isomorphic because of hash</th>
<th>Syntax error</th>
</tr>
</thead>
<tbody>
<tr>
<td>searchTree</td>
<td>505.43</td>
<td>1106</td>
<td>2979</td>
<td>1447</td>
</tr>
<tr>
<td>binaryTree</td>
<td>358.45</td>
<td>1274</td>
<td>4069</td>
<td>708</td>
</tr>
<tr>
<td>avlTree</td>
<td>130.63</td>
<td>546</td>
<td>3190</td>
<td>1553</td>
</tr>
</tbody>
</table>

Table 3: Types of Isomorphism and Non-Isomorphism Found (Cont)

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Different Sizes</th>
<th>Graph checker rejects</th>
<th>No compatible functions</th>
<th>Not satisfable</th>
</tr>
</thead>
<tbody>
<tr>
<td>searchTree</td>
<td>0</td>
<td>101</td>
<td>279</td>
<td>0</td>
</tr>
<tr>
<td>binaryTree</td>
<td>3</td>
<td>107</td>
<td>1085</td>
<td>190</td>
</tr>
<tr>
<td>avlTree</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

complete up to some finite bound. The hashing strategy only places potentially isomorphic instances into the same bucket, meaning that by comparing only the instances that fall into the same bucket, we can’t “miss” any isomorphic instances and conclude two isomorphic instances are non-isomorphic. And because we generate all possible instances, we can’t miss a counterexample, either. This hinges, of course, on being confident in our conversion to graph strings and our graph isomorphism-checker. In order to determine that these are reliable, we randomly sampled the set of model comparisons and double-checked them by hand, checking about 10 comparisons per model type (none of these comparisons were models against themselves - we verified that all models were isomorphic to themselves separately). We were unable to check all of the models due to time constraints; however, we were not able to find any discrepancies.

8 Biases

Though we made an effort to present unbiased data, no analysis is without its limitations. By using only projects from the same class at the same school, we introduce a source of bias because the students likely were from similar backgrounds and may have taken classes with the same professors, influencing their design decisions. Since most of the students would have learned Alloy through the class project, their implementations might be more similar they otherwise would have been.

Additionally, the projects themselves may have been slightly contrived. Although each of these models are very common computer science concepts to model in a classroom setting, most of the time Alloy would not be needed to check facts about a binary tree or an avl tree, as these have already been studied. And, of course, all the models examined were implementations of a tree, which may bias our results if very different sorts of models react quite differently.

In order to put a cap on the worst possible running-time of the program, we used the NodeEdges/NodeGraph toy example, which essentially models any possible graph with the only restriction being no duplicate (directed) edges may exist. This is the most unconstrained system with only 2 sigs that we could contrive. We found that running for 4 nodes our system was feasible - running for 5 it was not. This was due to the sheer number of instances produced. We additionally noticed a large number of isomorphic instances being pruned when
run for 5, also contributing to a slower run of the system.

9 Conclusion

Overall, Amalgam provides a lightweight, easy-to-adapt system for comparing a two Alloy models, up to a finite bound. This system is both sound and complete. Amalgam’s runtime is dependent on a number of factors, including the number of instances in each model and the efficiency of each model, but in the general case, comparisons are fast and accurate. Amalgam is unable to compare two models without a finite bound, and the longer the bound on the number of atoms, the longer Amalgam will take to run (and the more accurate its results are). We expect our system to be used mainly for comparing models which are conceptually similar, intended to reflect the same concept. As such, our system might not scale well when comparing two very different models, though more work is needed to determine this. Our experiments serve as preliminary results only, and the diversity among the sample space should be improved in future studies.

10 Future Work

Future studies should compare how Amalgam performs on non-tree based systems, use more diverse models, and compare models that are not conceptually similar in order to determine whether these preliminary results are representative. Additions to Amalgam itself mainly consist of a number of new flags that the user can specify indicating different modes. Some users might be interested in determining whether one model is a subset of another - that is, all instances of model A are isomorphic to an instance from model B, but not all instances of model B are isomorphic to an instance from model A. Other options would allow the user to specify how to compare ints. Currently, Amalgam treats all ints as isomorphic, but for some examples, this may not suffice. For example, for the search tree examples, the specific ints don’t matter but their order does matter. Other possible int options that could be supported by a future version of Amalgam include comparing based on order, parity, order relative to some other integer (for example, keeping track only of whether the numbers are positive/negative), multiples of a specific number, or ignoring all ints (currently the only system we have in place).

We would also like to provide a number of language extensions in order to improve the system for end users. We have three basic language extensions that we believe would be useful to provide.

1. **API functional annotations**: As described below, one level of analysis will depend on isomorphism analysis based on a number of specified functions (relations). The specific functions to consider in the analysis can be denoted by a @api annotation before the function definition.

2. **Operator Extensions**: We would like the ability for the user to be very specific in the type of analysis that needs to be performed. We would need to add operators or overload current operators to add the ability to express complex model predicates and assertions more accurately. An
example of its usage would be to check if a subset of one model may be considered isomorphic to another model.

3. **Namespaces**: Instead of necessarily declaring all models in different files, we can allow for namespaces so that many models can be conveniently described in one file, and then compared.

Additionally, we would like to be able to allow users to tell Amalgam to attempt to automatically infer api functions. This would look like finding the largest subset of functions from each model that are isomorphic to one another. This would be an option rather than the default because it would significantly slow down the runtime, but shouldn’t require too much modification from our current system. We also believe it would be useful to have several possible graph isomorphism checkers integrated with Amalgam, so that users can specify which checker to use. If a certain isomorphism checker does not work well for a particular model, users could try again with a different checker.

11 **Acknowledgements**

The authors would like to thank Ben Liblit, for providing the Alloy models used to run our experiments, Dieter van Melkebeek, for helping us to frame our problem as a graph-isomorphism checker and pointing us to pre-existing graph-isomorphism checkers, and Adolfo Piperno, for quickly responding and fixing a bug discovered in a graph isomorphism-checker we utilized.

12 **References**