Parallelizing the graph isomorphism portion of an automatic reaction mechanism generation algorithm

Geoffrey M. Oxberry

May 15, 2009

1 Problem: RMG is slow in running certain case studies

Reaction Mechanism Generator (or RMG) [13, 20, 3] is a software package that, given a starting set of chemical species and reaction conditions (species concentrations, temperature, pressure, etc.), automatically generates a chemical reaction mechanism that models the chemistry that takes place under those conditions. RMG yields reasonably accurate, first principles reaction mechanism models that can be used in a variety of applications, such as modeling pollutant emissions to inform environmental policy decisions, modeling combustion chemistry taking place in a laboratory experiment, predicting the effects of varying fuel and oxidant composition on engine operation, and others. While RMG generates models in less than a day for some case studies, for large case studies that potentially involve thousands of species and tens of thousands of reactions (such as diesel combustion, which involves modeling diesel as the combination of several surrogates with anywhere from 8 to 22 carbon atoms), RMG requires a few days or longer to generate a predicted reaction mechanism. One of the goals of the RMG project is to automate the process of generating a chemical reaction mechanism so that it can be used to automatically construct and solve a PDE model describing the fluid mechanics and the chemistry of combustion in an engine. To be practical, this whole process must take no more than a few days [14]. Given that solution of the PDE model in these applications is frequently a costly endeavor, RMG should generate a reaction mechanism within a day to meet the stated time constraint.

Currently, the main obstacle to meeting the stated time constraint of one day is algorithmic in nature. RMG generates reaction mechanisms using an iterative process. During the current iteration, RMG has a core set of species, which it uses in combination with a set of programmed reaction rules to determine possible chemical reactions and new species for inclusion in the reaction mechanism being generated. Chemical reactions in organic chemistry at sites on a reactant molecule; for combustion chemistry, the number of possible reaction sites scales combinatorially with the number of carbon atoms in the main fuel species in the reaction mechanism. (See [6, 7] for examples of large chemical reaction mechanisms that are considered the state-of-the-art in predictive modeling of combustion chemistry.) One aspect of the reaction mechanism generation algorithm involves solving multiple instances of a variant of the graph isomorphism (GI) problem for each species added to the reaction mechanism. If a different reaction occurs at each site, then the number of species added to the mechanism by proposed reactions scales combinatorially. As a result of the combinatorial scaling of the RMG algorithm with number of carbon atom, in reaction mechanisms derived from fuels containing a large number of carbon atoms, such as diesel combustion, RMG must solve in excess of tens of thousands of instances of the GI problem. In addition, the algorithm should scale combinatorially with the initial number of species, since each reaction can add new species, yielding additional possible opportunities for reactions, each of which can add new species, and so on. Therefore, if the time it takes to solve the GI problem portion of the reaction mechanism generation algorithm can be decreased, then the execution time of RMG on large case studies could be decreased substantially.

To decrease the time it takes to solve the GI problem within RMG, I propose two changes to the software. First, it is known that the (serial) algorithm used to solve instances of the GI problem in RMG is a brute-force algorithm. Several publications (such as [21, 17, 15, 4]) propose algorithms that solve instances of the GI problem with as good or better asymptotic scaling and demonstrated performance improvements (the improvements are for test cases over a range of graph sizes from approximately 10 to 1000; see [11] for these results) over the brute-force algorithm currently implemented in RMG. Replacing the brute-force serial algorithm with a more efficient serial algorithm has the potential to significantly decrease the execution time of RMG on case studies that generate reaction mechanisms with a large number of species. Second, it is known that the some of the instances of the GI problem solved in RMG are loosely coupled to one another, or completely decoupled altogether. Consequently, it is possible to leverage task parallelism to create a parallel implementation of the RMG algorithm that yields a modest decrease in the execution time of RMG relative to the serial implementation.

Since RMG is a large software package written in a combination of Java, Fortran 77, and C++, it would have been difficult to carry out these changes in the production version of RMG before the end of the Spring 2009 semester. As a result, I instead created a prototype version of RMG containing a skeletal version of the reaction mechanism generation algorithm that contains only the functionality relevant to solving the GI problem as it relates to RMG. I implemented this RMG prototype in Python, since this language has a robust set of libraries and easy-to-learn syntax, both of which support the rapid construction of software. Python also has several sets of libraries that provide bindings to MPI, which enabled me to construct a parallel implementation of the RMG prototype; Java lacks good MPI bindings, and was another reason I decided not to implement these proposed changes directly in the production version of RMG. This project demonstrates that task parallelism can be used to speed up RMG in some cases. As a result, it is worth investigating further to see if the serial reaction mechanism generation algorithm should be replaced with the reaction mechanism generation algorithm within my RMG prototype, adapted for a production version of RMG. Further investigations include additional case studies of intermediate size, further debugging, and exploiting additional opportunities for parallelization. Such a parallel implementation of this algorithm potentially could be added to RMG, and then could be used in combustion research within the Green group to better and more quickly model large combustion case studies, such as diesel combustion, and combustion of 1,3-butanol.

2 Implementing a more efficient serial graph isomorphism algorithm in RMG

Graph isomorphism (GI) is the problem of determining whether two graphs are ”the same”. Suppose we define an undirected graph $G = (V, E)$ as an ordered pair of a set of nodes V and a set of edges E, where edges are denoted by ordered pairs of nodes (for example, if $v, w \in V$, an edge between v and w is denoted (v, w), or equivalently, (w, v), since this graph is undirected). Graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a function $f : V \rightarrow V'$ such that $e = (v, w) \in E$ iff $e' = (f(v), f(w)) \in E'$. Put another way, graphs G and G' are isomorphic if there is a connectivity-preserving bijective mapping between the nodes of G and the nodes of G'. RMG takes the GI problem one step further in that graphs in RMG, called ChemGraphs, represent chemical species, and each node and edge has attributes associated with them. In ChemGraphs, nodes represent atoms, and have attributes of atom type and electronic state; edges represent bonds, and have the attribute of bond type. A more complete description of ChemGraphs can be found in [20]. In this case, two ChemGraphs G and G' are isomorphic if there is a connectivity-preserving bijective mapping between the nodes of G and the nodes of G' that also respects node labels and edge labels. The notation associated with this notion of attributed (or colored) graph isomorphism can be quite heavy, consequently, I eschew a formal description of attributed graph isomorphism here; for an example of this notation, see [18].

GI (ignoring graph attributes) is a difficult problem in the theory of computation, in that it is in class NP, and thus known any known deterministic algorithms that solve this problem are exponential-time algorithms [19]. In certain cases, GI is theoretically tractable, in that placing restrictions on GI yields a problem in class P. For example, if we are comparing two graphs of bounded degree, then GI is in class P [16]. This particular restriction applies to GI in the context of RMG, where the graphs represent the connectivity of atoms within a particular chemical species; since the largest known number of ligands around a central atom is 7 (as predicted by VSEPR theory and observed in iodine heptafluoride), the nodes of the graphs in RMG can be assumed to have degree no greater than 7. Furthermore, it is known that in the case of randomly generated graphs, graph isomorphism can be solved fairly quickly for most inputs [8]. Finally, note that most graphs in RMG are ”small”, in that they contain no more than 100 carbon atoms, and therefore no more than 500 nodes (as a fairly conservative upper bound). Therefore, solving the GI problem without considering attributes on graphs resembling chemical species is generally a practical task; the main obstacle in RMG, aside from adding attributes, is that the GI problem must be solved a large number of times.

Adding attributes to the GI problem poses no problem for all of the GI algorithms considered here. The attributed GI problem (or the colored GI problem, or the colored graph matching problem) seems to differ from the GI problem mostly in implementation details; clearly the GI problem is no easier to solve than the attributed GI problem, since the GI problem is merely a special case of the attributed GI problem.

The attributed GI problem could be solved in RMG in several ways. The original author of RMG, Dr. Jing Song, chose to implement a brute-force algorithm to solve the GI problem by implementing a labeled graph data structure in Java and then adopting a depth-first graph traversal algorithm out of a Java algorithms textbook [20, 12] to search for isomorphisms. This approach has the virtue of being the simplest possible GI algorithm, requiring the least amount of storage (because an adjacency list representation was used for graphs) at the cost of pruning the search space of possible isomorphisms in the crudest way (possible isomorphisms are no longer considered at the first discrepancy between nodes in the case of labels). Ullmann suggests a brute-force GI algorithm and a more efficient GI algorithm that prunes the search space of possible isomorphisms using additional rules [21]; both of these algorithms require the construction of matrix data structures that are much larger than the corresponding adjacency list representations of graphs in RMG. (Since the number of vertices in a chemical graph, $|V|$, is generally much larger than the number of edges, $|E|$, an adjacency list representation requires $O(|V|)$ space, whereas a matrix representation requires $O(|V|^2)$ space. Since RMG will store thousands of species, the adjacency list representation is preferred to save on storage.) Ullmann’s algorithm (we will use this name to refer to his more efficient algorithm) can both be modified to account for node and edge attributes, but this possibility was not considered in this report. McKay has implemented an algorithm called Nauty that transforms graphs into canonical graph representations that can be compared easily [17]. Junttila and Kaski updated Nauty to include checks on node and edge labels [15]; their algorithm is called BLISS. While BLISS is a viable algorithm for this project, I could not find a Python implementation of BLISS that used the node and edge label-checking features of the algorithm, and the level of graph theory required to successfully write my own implementation was beyond my capabilities. Cordella, *et al* have developed an algorithm called VF2 that prunes the search space of possible isomorphisms using more rules than Ullmann [4]. Unlike Ullmann's algorithm, VF2 does not require the construction of a matrix; the adjacency list representation of the graph can be used directly. In comparisons of these algorithms, VF2 shows the best overall performance characteristics and asymptotic scaling [11, 4], although care should be taken to point out that the authors of VF2 were also the authors of the comparison studies, and that the comparison studies were generally skewed towards graphs larger than the graphs that will be encountered in RMG.

The VF2 algorithm was chosen to solve the GI problem both because of its performance characterstics and because I found a Python API and C library (bundled together, they are called *igraph* [1, 5]) that implements the VF2 algorithm in a way that also checks both node and edge attributes. I believe that implementing this algorithm will yield better performance over the existing brute-force algorithm used in RMG for two reasons. First, the VF2 algorithm should have better performance characteristics, if the literature is to be believed; intuitively, I believe this is true because it prunes the search space of possible isomorphisms more extensively and using a greater number of heuristics than the brute-force GI algorithm implementation currently in RMG. (That being said, the overhead required to initialize the algorithm could outweigh its more sophisticated method of pruning the search space.) Second, the VF2 algorithm was coded by a computer scientist in C using more sophisticated and efficient programming techniques than the Java implementation of the brute-force GI algorithm, which was written by a graduate student who did not specialize in graph theory and the implementation of graph theoretic algorithms. The Java implementation of the brute-force GI algorithm was essentially copied out of a textbook, and as such, does not seem to include tweaks that could enhance the performance of the algorithm, as opposed to the C implementation of VF2, which seems to include these tweaks (and in fact, the igraph authors reference them in comments accompanying their commits to the igraph version control repository [1]).

3 Many instances of the graph isomorphism problem are solved during the reaction mechanism generation algorithm

Consider the following high-level/pseudocode description of the reaction mechanism generation algorithm:

1. Initialize reaction system with species given as input assigned to a group called the *core species*. Core species, and all reactions among them, comprise the reaction mechanism that will be generated as output from RMG.

2. Loop until termination:

   (a) Loop over all reaction family rules with core species as reactants to generate ChemGraphs of *postulated species* to be added to the mechanism:

      i. Check the ChemGraphs of the postulated species generated from the current rule for forbidden configurations using subgraph isomorphism. Discard any postulated species that have forbidden configurations (along with their associated reactions) and preserve the remainder.

      ii. Check that ChemGraphs of these postulated species among themselves for duplication using graph isomorphism. (Species are represented by ChemGraphs; since the postulated species are generated automatically as ChemGraphs according to a set of rules, it is possible that different rules yield different ChemGraphs for the same species. Since we want each species to be included in the reaction mechanism once, we need to check for duplication.) Discard any postulated species that duplicates another postulated species, reassigning labels/pointers for duplicated species appropriately to include the reactions found in step 2a.

      iii. Check the ChemGraphs of these postulated species for duplication against the *edge species* or core species using graph isomorphism. Discard any postulated species that duplicates a core species or edge species, reassigning labels/pointers for duplicated species appropriately to include the reactions found in step 2a.

Edge species are species that are produced in small amounts, relative to user-specified tolerances on rates of production of species. Core species can react among themselves to produce core species, or they could also produce edge species that are not part of the core species at the current iteration of the algorithm, but could be included in the core species later on. In this way, the set of core species (and thus the reaction mechanism) can grow at each iteration.

    iv. Move any remaining postulated species generated from the current rule to the set of edge species and add the associated reactions found in step 2a.

    v. Use subgraph isomorphism to query chemical properties databases in RMG for the chemical properties of the postulated species just added to the set of edge species. All of the chemical properties are stored with references to subgraphs of ChemGraphs (corresponding to functional groups of molecules), hence the need for subgraph isomorphism.

(b) Integrate the set of ODEs corresponding to the reaction mechanism consisting of core species, edge species, and all associated reactions at the initial conditions (species concentrations, temperatures, pressures) specified by the user. Integration should occur until the rate of production of an edge species exceeds the user-specified rate of production threshold for inclusion into the core set of species, or until a user-specified termination criterion is met. The termination criterion could be integration until a specific end point in time, or until a set amount of a species (usually a fuel species) has reacted, or until a steady state has been reached.

(c) If a species was moved from the set of edge species to the set of core species in step 2b, then return to step 2a. Otherwise, the termination criterion has been met, in which case RMG stops and outputs the reaction mechanism consisting of the set of core species and all reactions between core species.

As indicated in the high-level description of the algorithm, a GI algorithm is used to check for graph isomorphism in steps 2(a)ii and 2(a)iii; in total, the GI algorithm must be called roughly ((# postulated species in current iteration of the inner loop) · (# core and edge species in current iteration of the outer loop)) times for each iteration of the inner loop within the outer loop. Based on conversations with the current developers of RMG, postulated species are currently added by a reaction one or two at a time, while the set of core and edge species could be on the order of magnitude of tens of thousands at later steps in the algorithm. Subgraph isomorphism must also be used multiple times for each species to query chemical properties databases. Consequently, it is important to reduce the amount of time needed to carry out graph isomorphism comparisons by implementing a faster GI algorithm (if possible) and to parallelize the algorithm wherever possible, since the graph isomorphism comparisons between postulated species and core/edge species in step 2(a)iii are task parallel.

Two significant obstacles prevented me from rapidly making the necessary modifications within RMG itself. First, knowledge of the current internal workings of RMG is frighteningly incomplete. Dr. Jing Song's PhD thesis [20] fails to mention graph isomorphism testing in the description of the reaction mechanism generation algorithm in Chapter 7 of her thesis, although she mentions implementing graph isomorphism (referred to as ”graph equivalence”) in Chapter 3. Documentation of development decisions in RMG is limited to Dr. Song's thesis and recent additions by the RMG version 3.0 development team. While the current RMG development team is aware of the problem of the lack of documentation (and its consequences), the current state of affairs requires me to rely heavily on the current RMG development staff for RMG lore rather than consulting the documentation or the code itself (which also happens to be in a language I don't know, and does not consistently use good software development or coding practices). Second, RMG is a really large code in Java, which makes it difficult to understand, difficult to modify without inadvertently breaking key features within the code, and difficult to parallelize.

To address the first obstacle, I met regularly with the RMG team to refine my understanding of the algorithms in RMG. In addition, I've been active in proposing better documentation and development practices. Given that we as a group are exploring the possibility of migrating the RMG code from Java to Python, a move towards better software development and coding practices could prove valuable since these practices are most effective when carried out from the very beginning of a project, rather than implemented later on, after significant development has taken place; for this reason, I made a conscious effort to follow the best software development practices I know of when I code for this project.

To address the second obstacle, I constructed a prototype of the serial reaction mechanism generation algorithm in Python. The Python prototype was based on the following algorithm, which is a version of the reaction mechanism generation algorithm described above, with most of the domain-specific functionality stripped out (i.e., no ODE integration, no chemical properties database queries, no reaction rate rules, etc.) except for graph isomorphism:

1. Load a master database of ChemGraphs from existing RMG output.
2. Randomly select a part of this database be our initial set of core species. Permute all of the graphs within the database, to decrease the chance of finding trivial graph isomorphisms (that is, isomorphic graphs with identical node IDs).
3. Loop until the number of iterations exceeds the number of species in the master database, or until the set of core species equals the set of species in the master database:
   - (a) To mimic production of postulated species, randomly select ChemGraphs from the master database with duplicates. Since the postulated species in this prototype take the roles of both the postulated and edge species in the production-scale algorithm, and the number of edge species at the current iteration should be roughly proportional to the number of core species at the current iteration, the number of postulated species generated in this step will be proportional to the number of core species currently in the model.
   - (b) Use the GI algorithm to test for duplicates among the postulated species, discarding duplicates.
   - (c) Use the GI algorithm to check that postulated species do not duplicate any of the core species. (Edge species will not be considered here.) Discard duplicates.
   - (d) Add the remaining postulated species to the set of core species.

This prototype algorithm replicates the key parts of the reaction mechanism generation algorithm that I want to investigate (namely, the graph isomorphism portion) while eliminating the parts I don't have the time to implement. As a result, the prototype was easier to implement than modifying RMG directly, at the cost of somewhat inaccurately replicating the behavior of the actual reaction mechanism generation algorithm. Since most of the time in the production-scale reaction mechanism generation algorithm is spent carrying out GI comparisons or subgraph isomorphism comparisons, stripping out the ODE integration step shouldn't affect the results I'd obtain in a (relative) timing comparison between serial and parallel implementations; I wouldn't be parallelizing the ODE integration in RMG. Eliminating subgraph isomorphism comparisons should affect the results slightly, in that I'll be carrying out fewer subgraph isomorphism comparison per iteration of the prototype algorithm than I would per iteration of the outermost loop in the RMG reaction mechanism generation algorithm. Eliminating chemical properties database queries (aside from the subgraph isomorphism) shouldn't affect my serial-parallel timing comparison results, even though these queries are parallelizable; most of the time in the RMG reaction mechanism generation algorithm is spent performing GI comparisons (or subgraph isomorphism comparisons). Grouping together edge species and postulated species in the RMG reaction mechanism generation algorithm and reclassifying all of these species as postulated species in the prototype algorithm should affect the results, in that I will be making a different number of GI comparisons in the prototype algorithm as opposed to the RMG reaction mechanism generation algorithm due to the reclassification of species. As stated earlier, RMG generates postulated species one or two at a time, and then compares them against all current core and edge species. Here, I'll be generating lots of postulated species at a time, in an amount proportional to the current number of core species. This difference is an attempt to simulate many iterations of generating postulated species because I don't have reaction families over which I can loop, and this modification also lends itself to later parallelization. Since the total number of postulated species generated in an outer loop iteration of the RMG reaction mechanism generation algorithm is roughly proportional to the number of species in the set of core species at that outer loop iteration, the prototype algorithm attempts to replicate this behavior so that the number of GI comparisons in the prototype algorithm is roughly equal to (or greater than) the number of GI comparisons in the RMG reaction mechanism generation algorithm. Absolute timings for the prototype algorithm will be different than timings for the RMG serial algorithm due to differences in languages and implementation. It would have been interesting to time the prototype algorithm using the VF2 GI algorithm and the RMG brute-force GI algorithm, but I could not carry out this comparison due to time constraints.

4 Potential opportunities for parallelization of RMG's reaction mechanism generation algorithm

Revisiting the high-level/pseudocode description of the reaction mechanism generation algorithm, below is one proposed parallel algorithm for reaction mechanism generation within RMG. Note the following potential avenues for parallelism, and that the algorithm has been slightly reorganized:

1. Initialize reaction system with species on the first processor used for an RMG job, given as input assigned to a group called the *core species*. Communicate this set of core species to all other processors. Core species, and all reactions among them, comprise the reaction mechanism that will be generated as output from RMG.
2. Loop until termination:
   (a) Generate ChemGraphs of *postulated species* to be added to the mechanism using all possible reaction family rules with core species as reactants, tagging the postulated species ChemGraphs with their corresponding reactions accordingly.
   (b) Check the ChemGraphs of the postulated species generated from the current rule for forbidden configurations using subgraph isomorphism. Discard any postulated species that have forbidden configurations (along with their associated reactions) and preserve the remainder. Parallelize this by scattering the postulated species across the available processors so that processor loads are approximately balanced.
   (c) Continuing the previous step, check the ChemGraphs of these postulated species for duplication against the *edge species* or core species using graph isomorphism. Discard any postulated species that duplicates a core species or edge species, reassigning labels/pointers for duplicated species appropriately to include the reactions found in step 2a. This step should be carried out

in parallel by continuing the previous step; if a postulated species does not have any forbidden configurations, it should remain on the same processor to be checked for duplication against the edge species or core species. Any postulated species that are not duplicated should be sent back to the first processor as soon as this information is known.

(d) On the first (rank 0) processor, check that ChemGraphs of these postulated species among themselves for duplication using graph isomorphism as they arrive. (Again, species are represented by ChemGraphs; since the postulated species are generated automatically as ChemGraphs according to a set of rules, it is possible that different rules yield different ChemGraphs for the same species. Since we want each species to be included in the reaction mechanism once, we need to check for duplication.) Discard any postulated species that duplicates another postulated species, reassigning labels/pointers for duplicated species appropriately to include the reactions found in step 2a. It may be possible to parallelize this step more efficiently by subdividing the list of postulated species, checking for isomorphism, and then merging.

(e) Move any remaining postulated species generated from the current rule to the set of edge species and add the associated reactions found in step 2a. Communicate these updates to each processor.

(f) Use subgraph isomorphism to query chemical properties databases in RMG for the chemical properties of the postulated species just added to the set of edge species. Queries can be parallelized by species, since each species has its own chemical properties. (All of the chemical properties are stored with reference to subgraphs of ChemGraphs (corresponding to functional groups of molecules), hence the need for subgraph isomorphism.) Communicate the chemical properties back to the first processor; do not store any of these properties on other processors, since the other processors only need the sets of ChemGraphs corresponding to postulated, edge, and core species.

(g) On the first processor, integrate the set of ODEs corresponding to the reaction mechanism consisting of core species, edge species, and all associated reactions at the initial conditions (species concentrations, temperatures, pressures) specified by the user. Integration should occur until the rate of production of an edge species exceeds the user-specified rate of production threshold for inclusion into the core set of species, or until a user-specified termination criterion is met. The termination criterion could be integration until a specific end point in time, or until a set amount of a species (usually a fuel species) has reacted, or until a steady state has been reached.

(h) If a species should be moved from the set of edge species to the set of core species in step 2g, update the set of edge species and core species accordingly on each processor, and then return to step 2a. Otherwise, the termination criterion has been met, in which case RMG stops and outputs the reaction mechanism consisting of the set of core species and all reactions between core species.

Note in the algorithm above that many of the graph isomorphism and subgraph isomorphism tasks are task-parallelizable by species; this property is fortuitous, since most of the time in the RMG reaction mechanism generation algorithm is spent on graph isomorphism and subgraph isomorphism. Each postulated species must be checked against forbidden configurations and against all core and edge species for duplication; the results of these tasks do not depend on the order in which postulated species are checked. After checking among the postulated species for duplicates, chemical properties database queries are also task-parallelizable because results are independent of the order in which species are queried for their properties. ODE integration isn't a time-intensive step in this algorithm, although it could conceivably be parallelized if we wanted to integrate the same set of ODEs for multiple initial conditions; this situation occurs frequently in RMG, but will not be considered further.

Note also that the parallelism will incur communication costs due to the updates to the core set of species (scattering data among processorsm, gathering data from processors, and broadcasts). To economize on communication, it would be preferable to transmit information to the first processor used by an RMG job while the first processor is also processing data, thus hiding the cost of communication, and it would also be preferable to strike a balance between sending small packets of information whose communication costs are dominated by overhead and sending large packets of information whose communication costs are a bottleneck to data processing in the algorithm. However, for simplicity, I avoided hiding communication costs in the prototype. These issues should be explored when refining the parallel prototype in future work.

The obstacles that prevented me from rapidly modifying the serial reaction mechanism generation algorithm also prevented me from rapidly parallelizing RMG. As stated earlier, RMG lacks documentation, and it is a big code written in Java. Addressing and overcoming these issues involves using the same approach of constructing a prototype that simulates the relevant features of the RMG reaction mechanism generation algorithm. This prototype will be based on the serial prototype of the RMG reaction mechanism generation algorithm, and will incorporate parallelism in a fashion that approximates how parallelism could be incorporated into a production-scale implementation of the RMG reaction mechanism generation algorithm. To minimize the time it took to implement this prototype, I coded it in Python 2.5, using the Python module *mpi4py* [2, 9, 10] to interface Python to MPI to implement parallelism and communication steps. This parallel Python prototype was based on the following algorithm:

1. Load a master database of ChemGraphs from existing RMG output.
2. Randomly select a part of this database be our initial set of core species. Permute all of the graphs within the database, to decrease the chance of finding trivial graph isomorphisms (that is, isomorphic graphs with identical node IDs). Synchronize this database with all processors.
3. Loop for a fixed number of iterations:
    (a) To mimic production of postulated species, randomly select ChemGraphs from the master database on the first processor with duplicates. Since the postulated species in this prototype take the roles of both the postulated and edge species in the production-scale algorithm, and the number of edge species at the current iteration should be roughly proportional to the number of core species at the current iteration, the number of postulated species generated in this step will be proportional to the number of core species currently in the model.
    (b) Use the GI algorithm to test for duplicates among the postulated species, discarding duplicates.
    (c) Distribute the postulated species among all processors and use the GI algorithm to check that postulated species do not duplicate any of the core species. (Edge species will not be considered here.) Discard duplicates. Any postulated species should be sent back to the first processor (head node).

(d) Add the remaining postulated species to the set of core species and synchronize the database across all processors.

As in the serial case, this parallel prototype algorithm approximately replicates the key parts of the RMG reaction mechanism generation algorithm I want to investigate (the graph isomorphism portion) while abstracting away the parts that I don't have the time to implement. Many of the comments in the comparison of the serial prototype algorithm to the serial production-scale reaction mechanism generation algorithm were written with the goal of accurately approximating the results I would get if I could directly compare the execution time of RMG to the execution time of a parallelized version of RMG. All of these comments still apply (such as why eliminating ODE integration will not change relative timings much in a serial to parallel comparison, why eliminating subgraph isomorphism probably won't change relative timings much in a serial to parallel comparison, etc.). The only aspects of a comparison of relative timings that weren't addressed in the discussion of the serial prototype are those aspects of the parallel prototype that involve communication. Since I do not have any subgraph isomorphism functionality in my prototype code, I cannot replicate the communication that takes place in the production-scale parallel algorithm involving subgraph isomorphism steps. I do, however, replicate in the prototype parallel algorithm all of the other communication that occurs in the production-scale parallel algorithm. Consequently, I believe that the prototype parallel algorithm replicates to the best of my ability, given existing time constraints, the functionality of a hypothetical production-scale parallel algorithm in such a way that a relative comparison of timings between the serial prototype and the parallel prototype will yield a good approximation of the results I would get if I were to carry out a relative comparison of timings between a serial version of RMG and a parallel version of RMG.

5 Results and Discussion

In order to determine whether or not parallelization of RMG would reduce appreciably the execution time of RMG, I implemented both the serial and parallel prototypes proposed in this technical report. Both prototypes were implemented in Python 2.5 on a 64-node cluster, pharos.mit.edu. Each node of the cluster

Database Size	Initial Set Of Core Species	# Iterations	Final # Core Species	Time (serial)	Time (parallel)
21	10	11	21	0.055 s	0.222 s
119	59	13	119	64.4 s	32.2 s
2105	1052	100	1448	169.7 s	673.9 s

Table 1: Table of execution times for implementations of both serial and parallel prototypes. Note that the parallel prototype outperforms the serial prototype on the intermediate test case, having a database size of 119 species, whereas the serial prototype outperforms the parallel prototype in the two other case studies.

consists of 8 GB of memory and two quad-core Xeon processors, both running at either 2.33 GHz or 2.66 GHz (depending on the node), connected by gigabit Ethernet interconnects. MPICH version 1.0.8 was used as a source of MPI libraries, which were called from Python through the module mpi4py [2, 9, 10]. Both serial and parallel implementations were run by submitting jobs through a PBS queue system; due to the distribution of jobs on the cluster, specific nodes were not reserved for timings, and so timing results are approximate. Eight processors were used for parallel simulations.

One would expect that when starting with small initial set of core species, the serial implementation of RMG would be faster than the parallel implementation; this can be seen in results in Table 1 when starting with 10 initial species out of a set of 21 species. In slightly larger test cases, one would hope that the benefit of distributing the comparisons between postulated species and core species over many processors outweighs the cost incurred due to communications; this result is seen in Table 1 when starting with 59 species out of a set of 119 species. Run-time errors occurred during the execution of the parallel implementation on a very large test case of 1052 initial species out of a set of 2105 species using the termination criterion indicated in the pseudocode for the parallel prototype; as a result, results were obtained for the serial and parallel implementation for the very large test case by terminating the algorithm at 100 iterations. Here, the serial implementation is faster than the parallel implementation, as can be seen in Table 1. In this case, the expense of communication outweighs the benefits of distributing computations.

Although there isn't enough data in Table 1 to make any definitive conclusions at this time, I suspect that the primary reason that increasing the database size and initial set of core species eventually causes the serial prototype to run more quickly than the parallel prototype is due to the way that mpi4py passes Python objects in my current implementation of the parallel prototype. In my current implementation of the parallel prototype, I elected to use the simpler of the two available communication APIs in mpi4py. This

API uses the cPickle module in Python to convert any nonstandard Python objects (i.e., user-defined classes and non-native data types) into strings to facilitate communication, and this conversion process can require significant CPU resources. Since all of the graphs are implemented as user-defined classes (in this case, as igraph.Graph objects), pickling must occur any time that ChemGraph-like objects are passed. In the 119 species database test case, the cost of pickling was probably not prohibitive, and only 50-100 graphs were scattered and gathered per iteration. In the 2105 species database test case, however, the cost of pickling was probably prohibitive, since 1000-1500 graphs were scattered and gathered per iteration. In order to make more definitive conclusions about the benefits of parallelism, further case studies must be run in order to get a better idea of how the implementations of the serial and parallel prototypes scale with database size. In addition, it would be worthwhile to profile the serial and parallel code to get a better idea of where CPU effort is being spent in each code, and what the limiting steps are in each code; this investigation has not yet been carried out since Python debugging and profiling tools were not available on pharos at the time the project was completed.

6 Obstacles

Several obstacles were encountered over the course of completing this project. First, my laptop died about three weeks into the project, which required me to find an alternate computer for prototyping. Then, after that, I had to install both the mpi4py and igraph libraries on pharos. Both libraries were difficult to install on pharos. In the case of mpi4py, installation required recompilation of the MPICH libraries with different compiler flags, followed by modifying the path, the linker path, and adding environment variables; a similar installation on my laptop (for prototyping purposes) was only partially successful. The installation of igraph also required the modification of the path, the linker path, and adding environment variables. After successfully installing both libraries, my serial code worked, but my parallel code did not. In order to get my parallel code to work, I had to modify my PBS job scripts and additional environment variables in order to ensure that libraries were recognized on cluster nodes, in addition to the head node. After this step, I had spent about a week only on installing and debugging software installation issues. Finally, I had issues debugging my parallel code. Since I was using Python on pharos when most pharos users run C, C++, or Fortran 77 code, no Python debuggers were installed on the nodes, making it difficult to diagnose MPI and memory run-time errors that occurred when running the very large test case. In addition, no Python profilers were installed on pharos, making it impossible to assess how CPU effort was being distributed when running each code. These technical issues delayed the completion of the parallel prototype and should be taken into account when considering the development of a parallelized, Python version of RMG to run on pharos.

7 Conclusions and future work

Based on timing results from running Python implementations of both serial and parallel prototypes of RMG, parallelizing RMG has the potential to reduce the execution time of RMG and enable the automatic generation of chemical reaction mechanisms derived from fuels with a larger number of carbon atoms. In order to investigate more fully the benefits of parallelizing RMG, the parallel prototype must first be debugged more extensively using an MPI debugger and memory profiler. After that, I would run several additional case studies of sizes intermediate to the largest and smallest case studies (in terms of species database size) in order to get a better idea of how the parallel code execution time scales with problem size; executing the parallel code on additional case studies with a profiler would be especially helpful in order to better understand how much time is spent on computation versus communication, and which parts of the parallel code are computation or communication-intensive.

Depending on the results of additional case studies after debugging and profiling, I'd consider changing the current mpi4py API calls from the more compute-intensive calls that use cPickle to calls that use NumPy's array data structures to more effectively buffer communication using MPI. In order to make best use of these calls, it would be necessary to dig through the C language implementation of igraph to determine the underlying C/Python data structures that make up the objects I use to represent ChemGraphs (in other words, I need to determine how igraph.Graph objects are represented in terms of underlying C data structures, like integers, floats, doubles, etc.). This information could be used to create a derived data type in MPI to further improve the buffering of communication within MPI calls, as well as reduce the computational burden of communication (recall that cPickle is computationally intensive, whereas the NumPy method of passing data using mpi4py is computationally cheaper). Another possible opportunity for speed gains could arise from rearranging the steps in the parallel prototype algorithm by checking the postulated species against the core species first, and then checking the postulated species against themselves; this rearrangement would increase the amount of data that needs to be communicated by scattering and gathering, but it would decrease the effort needed to check postulated species against themselves, since presumably, fewer postulated species would need to be checked after comparing them to the core species. Checking the postulated species against themselves for duplicates could be parallelized as well. Finally, it would be worthwhile to construct a serial prototype that uses the current naïve graph isomorphism algorithm currently used in RMG in order to determine definitively that the VF2 algorithm performs more efficiently and quickly than the naïve graph isomorphism algorithm.

8 Acknowledgments

I'd like to thank the current RMG development team of Gregory Magoon, Michael Harper, C. Franklin Goldsmith, Josh Allen, Sandeep Sharma, and Dr. Richard West. Special thanks go to Dr. West for notifying me of the igraph library, which has made this project considerably easier to tackle. I'd also like to thank the Ghoniem group at MIT for helping me to make arrangements to use the shared Ghoniem/Green group cluster, on which I ran my project for serial and parallel timings. In particular, I'd like to thank Ray Speth, who administers the Green/Ghoniem cluster, for his help in installing libraries, setting environment variables, setting up scripts, and answering many other questions I had in order to get my code working. Without his help, this project would probably remain unfinished. I'd like to thank Professor William H. Green for his help on clarifying the role of RMG in Green group research, and Professor Paul I. Barton for his patience and guidance while working on this project. I'd also like to gratefully acknowledge support under a DOE CSGF fellowship provided under grant DE-FG02-97ER25308, since this draft will also be turned into a technical report, as it pertains to Green group research.

References

[1] igraph website. http://igraph.sourceforge.net/. 6

[2] MPI4Py website. http://code.google.com/p/mpi4py/. 15, 17

[3] RMG website. http://rmg.sourceforge.net/. 1

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, October 2004. 2, 6

[5] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. 6

[6] H. J. Curran, P. Gaffuri, W. J. Pitz, and C. K. Westbrook. A comprehensive modeling study of n-heptane oxidation. *Combustion and Flame*, 114:149–177, 1998. 2

[7] H. J. Curran, P. Gaffuri, W. J. Pitz, and C. K. Westbrook. A comprehensive modeling study of iso-octane oxidation. *Combustion and Flame*, 129:253–280, 2002. 2

[8] T. Czajka and G. Pandurangan. Improved random graph isomorphism. *Journal of Discrete Algorithms*, 6(1):85–92, 2008. 4

[9] L. Dalcín, R. Paz, and M. Storti. Mpi for Python. *Journal of Parallel and Distributed Computing*, 65:1108–1115, June 2005. 15, 17

[10] L. Dalcín, R. Paz, M. Storti, and J. D'Elía. Mpi for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68:655–662, October 2008. 15, 17

[11] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, Ischia, May 2001. 2, 6

[12] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, 2005. 5

[13] W. H. Green. *Chemical Engineering Kinetics*, volume 32 of *Advances in Chemical Engineering*, chapter Predictive Kinetics: A new approach for the 21st century, pages 1–50,313. Elsevier, 2007. 1

[14] W. H. Green. Building and solving accurate combustion chemistry simulations. *Journal of the Combustion Society of Japan*, 50(151):19–28, February 2008. 1

[15] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, New Orleans, January 2007. Society for Industrial and Applied Mathematics. 2, 5

[16] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982. 4

[17] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981. 2, 5

[18] B. T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12:307–323, March/April 2000. 4

[19] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, Massachusetts, 2006. 4

[20] J. Song. *Building Robust Chemical Reaction Mechanisms: Next Generation Automatic Model Construction Software*. PhD thesis, Massachusetts Institute of Technology, 2004. 1, 4, 5, 9

[21] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association of Computing Machinery*, 23(1):31–42, January 1976. 2, 5

A Appendix: Python code

RMG utilities library:

```
def LoadGraphsFromFileIntoList(graphFileName):
    """
    Purpose: Imports ChemGraphs from text file and converts them into a list
    of graphs encoded as a list of text strings. (Note that this is not a
    list of Graph objects, which are distinct.)

    Input: graphFileName = text string for a file name

    Output: graphDataList = list of graphs

    Outcomes: Returns a list of graph objects, each object encoded as a list.

    Also assumes that the text file takes the format:
    <graph object>
    \n
    \n
    <graph object>
    \n
    \n
    (repeated ’til end-of-file)

    """

#Open file of ChemGraphs
    graphFile = open(graphFileName,’r’);

#Set up list of graph data.
#Stores list of ChemGraph data...these aren’t the actual graphs yet.
    graphDataList = [];
#Stores a list, containing the label of the graph, followed by the graph.
#This list will be known as the ’current graph member.’
    currentGraphMember = [];
#Boolean variable
    isSecondBlankLine = False;

#Read each line in the file of ChemGraphs
    for fileLine in graphFile:
#If I have a blank line, then check to see if this is the first blank
#line in a sequence or the second blank line in a sequence.
        if fileLine == ’\n’:
#If it isn’t the second line, update the second line counter to reflect that
#the next blank line is the second one.
            if not isSecondBlankLine:
                isSecondBlankLine = True;
#Otherwise, it is a second line, so updated the second line counter to reflect
#that the next blank line is the first blank line in a two line sequence.
            else:
                graphDataList += [currentGraphMember];
                currentGraphMember = [];
                isSecondBlankLine = False;
#Otherwise, add the line to the currentGraphMember.
        else:
            currentGraphMember.append(fileLine.strip('\n'));

    return graphDataList

#Testing:
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.LoadGraphsFromFileIntoList('RMGDictionarySmallTestCase.txt')
#>>> for line in listOfGraphs[0]:
# print line
#
#
#HXD13(1)
#1  C 0 {2,D} {7,S} {8,S}
#2  C 0 {1,D} {3,S} {9,S}
#3  C 0 {2,S} {4,D} {10,S}
#4  C 0 {3,D} {5,S} {11,S}
#5  C 0 {4,S} {6,S} {12,S} {13,S}
#6  C 0 {5,S} {14,S} {15,S} {16,S}
#7  H 0 {1,S}
#8  H 0 {1,S}
#9  H 0 {2,S}
#10  H 0 {3,S}
#11  H 0 {4,S}
#12  H 0 {5,S}
#13  H 0 {5,S}
#14  H 0 {6,S}
#15  H 0 {6,S}
#16  H 0 {6,S}

def ConvertGraphMember(graphObject):
    """
    Purpose: Take a graph member (from the list data structure in
    ImportChemGraphsFromFile) that is stored in list format and
    converts it into a graph object via igraph.

    Input: graphObject = list containing a graph member object in format
    [<tag>,
     <node 1 listing>,
     <node 2 listing>,
     <node 3 listing>,...,etc.]

    Output: graphTitle = title of the chemistry graph (some text string
            describing the species)
            chemGraph = chemistry graph, in igraph format

    Outcomes: returns a list: [graphTitle, chemGraph]

    Assertion checks: Needs igraph module!
    """

#Import igraph module
    import igraph

#TODO(goxberry@mit.edu): Remove dead code from the copying...
#Copy the input argument to avoid pass-by-reference badness that occurred
#during testing...since in production, I generally won’t need repeat
#invocations of this function on the same object, I could probably get away
# without doing a deep copy of the list. In debugging, however, I will
#repeatedly invoke this function on the same argument, so I need a deep copy
#    import copy
#    graphMember = copy.deepcopy(graphObject)
    graphMember = graphObject

#Need to create data structures to hold data
#First element of the graph member is the title; grab it and pop it off.
#    graphTitle = graphMember.pop(0)
    graphTitle = graphMember[0]
    nodeList = []
    atomList = []
    elecStateList = []
    nodeColorList = []
    edgeList = []
    bondTypeList = []
    edgeColorList = []

#The igraph indexing convention starts with nodes at 0
    iGraphIndexConvention = 1

#For each of the remaining lines in the graph member list, we need to do
#some list processing
#    for line in graphMember:
    for line in graphMember[1:]:
    #Split on whitespace
        lineList = line.split()
    #Pop off node identity and attributes
        currentNode = (int(lineList.pop(0)) - iGraphIndexConvention)
        nodeList += [currentNode]
        currentAtomType = lineList.pop(0)
        currentElecState = lineList.pop(0)
        atomList += [currentAtomType]
        elecStateList += [currentElecState]
        nodeColorList += [NodeHashFunction(currentAtomType,currentElecState)]
    #Now all we have left is a list of edge specifiers of the form
    #’{<node label>, <bond type>}’, so we iterate over the strings in
    #the list, strip out the braces, split on the commas, and pull out
    #the node label and the bond type.
        for bond in lineList:
            bond = bond.strip(’{}’)
            bondAttrib = bond.split(’,’)
            destinationNode = int(bondAttrib[0]) - iGraphIndexConvention
            bondType = bondAttrib[1]
            #igraph doesn’t cull out duplicate edges in its data structures,
            #so to avoid duplicate edges, we only store edges of the form
            #(x, y), where x and y are natural numbers, and x < y.
            if (destinationNode > currentNode):
                edgeList += [(currentNode,destinationNode)]
                bondTypeList += [bondType]
                edgeColorList += [EdgeHashFunction(bondType)]

#Now we use all of the information above to instantiate a Graph object.
    chemGraph = igraph.Graph(len(nodeList))
    #print edgeList
    chemGraph.add_edges(edgeList)
    chemGraph.vs["atom"] = atomList
    chemGraph.vs["elecState"] = elecStateList
    chemGraph.vs["color"] = nodeColorList
    chemGraph.es["bond"] = edgeList
    chemGraph.es["edgeColor"] = edgeColorList

    return [graphTitle,chemGraph]
#Testing:
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.LoadGraphsFromFileIntoList(’RMGDictionarySmallTestCase.txt’)
#chemGraphTest = rmgUtils.ConvertGraphMember(listOfGraphs[0])
#>>> chemGraphTest.get_edgelist()
#[(0, 1), (0, 6), (0, 7), (1, 2), (1, 8), (2, 3), \
#(2, 9), (3, 4), (3, 10), (4, 5), (4, 11), (4, 12), (5, 13), (5, 14), (5, 15)]

def ImportChemGraphsFromFile(graphFileName):
    """
    Purpose: Imports ChemGraphs from text file and converts them into a list
    of Graph objects.

    Input: graphFileName = text string for a file name

    Output: graphDataList = list of graphs

    Outcomes: Returns a list of graph objects.

    Assertion checks: Needs igraph module!

    Also assumes that the text file takes the format:
    <graph object>
    \n
    \n
    <graph object>
    \n
    \n
    (repeated ’til end-of-file)
    """

    #Debugged to ensure that the list of graph members properly
    #includes node and edge attributes.
    listOfGraphMembers = LoadGraphsFromFileIntoList(graphFileName)
    listOfGraphs = []
    for species in listOfGraphMembers:
        newGraph = ConvertGraphMember(species)
        listOfGraphs += [newGraph]

    return listOfGraphs


def NodeHashFunction(atomType,elecState):
    """
    Purpose: Takes an atom type and an electronic state, both strings,
    and hashes them to a single integer (like a C-type integer). The idea
    is to convert multiple vertex labels to a single integer label for use
    in the VF2 graph isomorphism algorithm. The integer is treated like a
    16-bit bitfield.

    Input: atomType = string that contains the atom type
           elecState = string that contains the electronic state of an atom.

    Output: nodeLabel = label for this node (also known as a node "color")

    Assertion checks: none.

    """

    #Constants that denote the width of each bitfield
    #There are 117 elements in the periodic table.
    atomBitfieldWidth = 7;
    #RMG 3.0 currently lists 6 different atomic electronic states.
    elecStateBitfieldWidth = 3;

    #Constants for atom types; the constant = atomic # - 1. The symbol used
    #for each atom is from the periodic table of elements; additional
    #representations may be added.
    listOfAtomTypes = {’H’:0,
                       ’He’:1,
                       ’Li’:2,
                       ’B’:3,
                       ’C’:5,
                       ’N’:6,
                       ’O’:7,
                       ’F’:8,
                       ’Ne’:9}

    #Constants for electronic state types; the constant really just refers
    #to the order in which they were listed in Jing Song’s PhD thesis.
                        #’0’ = nonradical
    listOfElecStates = {’0’:1,
                        #’1’ = monoradical
                        ’1’:2,
                        #’2’ = biradical
                        '2':3,
                        #'2T' = triplet
                        '2T':4,
                        #'2S' = singlet
                        '2S':5,
                        #'3' = triradical
                        '3':6}

    #Assign the first atomBitfield width bits of the nodeLabel bitfield to
    #the binary integer represented by the constants in the list of atom types.
    #Assign the second atomBitfield width bits of the nodeLabel bitfield to
    #the binary integer represented by the constants in the list of electronic
    #state types. Note that errors will be returned if either the atomType
    #of the elecState isn't one of the listed choices.
    nodeLabel = listOfAtomTypes[atomType] + \
        (listOfElecStates[elecState] << atomBitfieldWidth)

    return nodeLabel

#Tests:
#>>> import rmgUtils
#>>> rmgUtils.NodeHashFunction('C','0')
#133
#>>> 5 + (1 << 7)
#133

def EdgeHashFunction(bondType):
    """
    Purpose: Takes a bond type and an electronic state, both strings,
    and hashes them to a single integer (like C-type integer). The idea is
    to convert (possibly multiple) edge labels to a single integer label for
    use in the VF2 graph isomorphism algorithm. The integer is treated like a
    16-bit bitfield.

    Input: bondType = string that contains the bond type

    Output: edgeLabel = label for this edge (also known as an edge "color")

    Assertion checks: none.
    """

    #Constants that denote the width of each bitfield
    bondBitfieldWidth = 3;

    #Constants for bond types; the constant really just refers to the order
    #in which they were listed in Jing Song's PhD thesis.
                       #'S' = single bond
    listOfBondTypes = {'S':0,
                       #'D' = double bond
                       'D':1,
                       #'T' = triple bond
                       'T':2,
                       #'Dcis' = cis double bond
```

```
                        'Dcis':3,
                        #'Dtrans' = trans double bond
                        'Dtrans':4,
                        #'B' = benzene/aromatic bond
                        'B':5}

    #Assign the first atomBitfield width bits of the nodeLabel bitfield to
    #the binary integer represented by the constants in the list of atom types.
    edgeLabel = listOfBondTypes[bondType]

    return edgeLabel

#Tests:
#>>> import rmgUtils
#>>> rmgUtils.EdgeHashFunction('D')
#1
#>>> rmgUtils.EdgeHashFunction('Dcis')
#3

def SelectSubsetOfGraphsNoRepeats(listOfGraphs,subsetSize):
    """
    Purpose: Given a list of Graphs in listOfGraphs and the size of the
    desired randomly selected subset (w/o repeats), this function returns
    a randomly selected subset of the desired size.

    Input: listOfGraphs = a list of Graphs (w/ titles), obtained by running
    ImportChemGraphsFromFile
    subsetSize = size of the subset of listOfGraphs to be selected randomly
    w/o repeats

    Output: subsetList = list of Graphs (w/ titles) selected; this list
    will be a subset of listOfGraphs

    Assertion checks: subsetSize <= len(listOfGraphs) in order to make sense;
    a subset can't be larger than the host set. Also, subsetSize >= 0, since
    a subset can't have a negative number of elements. Won't test if
    subsetSize is nonintegral; won't test if subsetSize >= 0.

    """
    #Import the random number generator to determine subset indices.
    import random

    #Check that the size of the subset is smaller than the size of the list
    #of graphs.
    assert subsetSize <= len(listOfGraphs)

    #Generate the list of indices for the subset.
    subsetIndices = random.sample(xrange(len(listOfGraphs)),subsetSize)

    #Pull out from listOfGraphs the elements corresponding to the indices in
    #the list of subset indices
    subsetList = []
    for index in subsetIndices:
        subsetList += [listOfGraphs[index]]

    return subsetList

#Testing
#>>> import rmgUtils
#>>> listOfGraphs = \
#    rmgUtils.ImportChemGraphsFromFile('RMGDictionarySmallTestCase.txt')
#>>> len(listOfGraphs)
#21
#>>> listOfGraphs
#[['HXD13(1)', <igraph.Graph object at 0x963102c>],\
# ['CH4(2)', <igraph.Graph object at 0x96310ac>],\
# ['H2(3)', <igraph.Graph object at 0x963112c>],\
# ['CH3J(24)', <igraph.Graph object at 0x96311ac>],\
# ['C5H7J(30)', <igraph.Graph object at 0x963122c>],\
# ['C5H7J(84)', <igraph.Graph object at 0x96312ac>],\
# ['C5H7J(85)', <igraph.Graph object at 0x963132c>],\
# ['C2H6(36)', <igraph.Graph object at 0x96313ac>],\
# ['C10H14(80)', <igraph.Graph object at 0x963142c>],\
# ['HJ(25)', <igraph.Graph object at 0x96314ac>],\
# ['C5H6(53)', <igraph.Graph object at 0x963152c>],\
# ['C6H10(79)', <igraph.Graph object at 0x96315ac>],\
# ['C10H14(81)', <igraph.Graph object at 0x963162c>],\
# ['C10H14(82)', <igraph.Graph object at 0x96316ac>],\
# ['C5H7J(203)', <igraph.Graph object at 0x963172c>],\
# ['C3H3J(339)', <igraph.Graph object at 0x96317ac>],\
# ['C2H4(638)', <igraph.Graph object at 0x963182c>],\
# ['C4H6(748)', <igraph.Graph object at 0x96318ac>],\
# ['C5H8(55)', <igraph.Graph object at 0x963192c>],\
# ['C8H10(751)', <igraph.Graph object at 0x96319ac>],\
# ['C5H6(124)', <igraph.Graph object at 0x9631a2c>]]
#>>> subsetList = rmgUtils.SelectSubsetOfGraphsNoRepeats(listOfGraphs,5)
#>>> subsetList
#[['C2H4(638)', <igraph.Graph object at 0x963182c>],\
# ['C5H7J(85)', <igraph.Graph object at 0x963132c>],\
# ['C8H10(751)', <igraph.Graph object at 0x96319ac>],\
# ['CH3J(24)', <igraph.Graph object at 0x96311ac>],\
# ['HJ(25)', <igraph.Graph object at 0x96314ac>]]

def SelectSubsetOfGraphsInclRepeats(listOfGraphs,subsetSize):
    """
    Purpose: Given a list of Graphs in listOfGraphs and the size of the
    desired randomly selected subset (w/ repeats), this function returns
    a randomly selected subset of the desired size.

    Input: listOfGraphs = a list of Graphs (w/ titles), obtained by running
    ImportChemGraphsFromFile
    subsetSize = size of the subset of listOfGraphs to be selected randomly
    w/ repeats

    Output: subsetList = list of Graphs (w/ titles) selected; this list
    will be a subset of listOfGraphs
    Assertion checks: subsetSize <= len(listOfGraphs) in order to make sense;
    a subset can’t be larger than the host set. Also, subsetSize >= 0, since
    a subset can’t have a negative number of elements. Won’t test if
    subsetSize is nonintegral; won’t test if subsetSize >= 0.

    """
    #Import the random number generator to determine subset indices.
    import random

    #Check that the size of the subset is smaller than the size of the list
    #of graphs.
    assert subsetSize <= len(listOfGraphs)

    #Generate the list of indices for the subset.
    subsetIndices = []
    for element in xrange(subsetSize):
        subsetIndices += [random.randint(0,len(listOfGraphs)-1)]

    #Pull out from listOfGraphs the elements corresponding to the indices in
    #the list of subset indices
    subsetList = []
    for index in subsetIndices:
        subsetList += [listOfGraphs[index]]

    return subsetList
#Testing code:
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.ImportChemGraphsFromFile(’RMGDictionarySmallTestCase.txt’)
#>>> subsetList = rmgUtils.SelectSubsetOfGraphsInclRepeats(listOfGraphs, 10)
#>>> subsetList
#[[’C5H6(124)’, <igraph.Graph object at 0x92688ac>],\
# [’C10H14(81)’, <igraph.Graph object at 0x92684ac>],\
# [’CH3J(24)’, <igraph.Graph object at 0x925ed2c>],\
# [’C5H6(124)’, <igraph.Graph object at 0x92688ac>],\
# [’CH3J(24)’, <igraph.Graph object at 0x925ed2c>],\
# [’C2H6(36)’, <igraph.Graph object at 0x926822c>],\
# [’C5H7J(203)’, <igraph.Graph object at 0x92685ac>],\
# [’C5H8(55)’, <igraph.Graph object at 0x92687ac>],\
# [’H2(3)’, <igraph.Graph object at 0x925ec2c>],\
# [’C4H6(748)’, <igraph.Graph object at 0x926872c>]]

def SelectPermutedSubsetOfGraphsNoRepeats(listOfGraphs,subsetSize):
    """
    Purpose: Given a list of Graphs in listOfGraphs and the size of the
    desired randomly selected subset (w/o repeats), this function returns a
    randomly selected subset of the desired size; each graph will be permuted.

    Input: listOfGraphs = a list of Graphs (w/ titles), obtained by running
    ImportChemGraphsFromFile
    subsetSize = size of the subset of listOfGraphs to be selected randomly
    w/o repeats
    Output: subsetList = list of Graphs (w/ titles) selected; this list
    will be a subset of listOfGraphs

    Assertion checks: subsetSize <= len(listOfGraphs) in order to make sense;
    a subset can't be larger than the host set. Also, subsetSize >= 0, since
    a subset can't have a negative number of elements. Won't test if
    subsetSize is nonintegral; won't test if subsetSize >= 0. These assertion
    checks are built into SelectSubsetOfGraphsNoRepeats, so they won't be
    repeated here.

    """

    import igraph
    import random
    #Copy the input argument to avoid pass-by-reference badness that occurred
    #during testing...since in production, I generally won't need repeat
    #invocations of this function on the same object, I could probably get away
    # without doing a deep copy of the list. In debugging, however, I will
    #repeatedly invoke this function on the same argument, so I need a deep copy
    #import copy
    #graphList = copy.deepcopy(listOfGraphs)
    graphList = listOfGraphs

    #Select a subset of graphs and then permute each one in sequence.
    subsetList = SelectSubsetOfGraphsNoRepeats(graphList,subsetSize)
    #Use indices here because I want to assign in place; using a generator
    #won't let me assign in place!
    for index in xrange(len(subsetList)):
        #Make an easy species alias; this serves as a pointer
        species = subsetList[index][1]

        #Generate list to use in permuting vertices.
        numVertices = species.vcount()
        vertexPermutation = random.sample(xrange(numVertices),numVertices)
        subsetList[index][1] = PermuteGraphVerticesAndLabels(species,
                                                             vertexPermutation)

    return subsetList
#Tested. TODO(goxberry@mit.edu): Add test code here.



def SelectPermutedSubsetOfGraphsInclRepeats(listOfGraphs,subsetSize):
    """
    Purpose: Given a list of Graphs in listOfGraphs and the size of the
    desired randomly selected subset (w/ repeats), this function returns a
    randomly selected subset of the desired size; each graph will be permuted.

    Input: listOfGraphs = a list of Graphs (w/ titles), obtained by running
    ImportChemGraphsFromFile
    subsetSize = size of the subset of listOfGraphs to be selected randomly
    w/ repeats
    Output: subsetList = list of Graphs (w/ titles) selected; this list
    will be a subset of listOfGraphs

    Assertion checks: subsetSize <= len(listOfGraphs) in order to make sense;
    a subset can’t be larger than the host set. Also, subsetSize >= 0, since
    a subset can’t have a negative number of elements. Won’t test if
    subsetSize is nonintegral; won’t test if subsetSize >= 0. These assertion
    checks are built into SelectSubsetOfGraphsNoRepeats, so they won’t be
    repeated here.

    """

    import igraph
    import random
    #Copy the input argument to avoid pass-by-reference badness that occurred
    #during testing...since in production, I generally won’t need repeat
    #invocations of this function on the same object, I could probably get away
    # without doing a deep copy of the list. In debugging, however, I will
    #repeatedly invoke this function on the same argument, so I need a deep copy
#    import copy
#    graphList = copy.deepcopy(listOfGraphs)
    graphList = listOfGraphs

    #Select a subset of graphs and then permute each one in sequence.
    subsetList = SelectSubsetOfGraphsInclRepeats(graphList,subsetSize)
    #Use indices here because I want to assign in place; using a generator
    #won’t let me assign in place!
    for index in xrange(len(subsetList)):
        #Make an easy species alias; this serves as a pointer
        species = subsetList[index][1]

        #Generate list to use in permuting vertices.
        numVertices = species.vcount()
        vertexPermutation = random.sample(xrange(numVertices),numVertices)
        subsetList[index][1] = PermuteGraphVerticesAndLabels(species,
                                                             vertexPermutation)

    return subsetList


def PermuteGraphVerticesAndLabels(graphObject, vertexPermutation):
    """
    Purpose: Permutes the vertices of a graph and its labels.

    Input: graphObject = object of type igraph Graph
           vertexPermutation = list of length graphObject.vcount();
           some permutation of range(graphObject.vcount())

    Output: permutedGraphObject = graphObject, with the vertices, vertex
            labels, and edge labels all permuted in a consistent way

    Assertion checks: None, although I should check that graphObject
```

```
            is a igraph.Graph object, and that vertexPermutation is some
            permutation of range(graphObject.vcount())

    """

    #TODO(goxberry@mit.edu): Debugging code; copy species w/o
    #permutation and test for isomorphism to ensure permutation
    #of attributes carried out correctly.
    #graphObjectNoPermutation = copy.deepcopy(graphObject)

    #Copy vertex attribute lists.
    atomList = graphObject.vs['atom']
    elecStateList = graphObject.vs['elecState']
    nodeColorList = graphObject.vs['color']
    bondList = graphObject.es['bond']
    edgeColorList = graphObject.es['edgeColor']

    #Grab the adjacency list in edge format for later
    edgeAdjList = graphObject.get_edgelist()

    #Vertex permutation scrubs a graph of vertex_attributes
    #and edge attributes! However, it does properly permute
    #vertices and edges. So, after permuting the graph, we need to
    #permute the copied vertex and edge attribute lists and copy them
    #back to the graph.
    #First, permute the vertices of the graph and the vertex attributes.
    permutedGraphObject = \
        graphObject.permute_vertices(vertexPermutation)
    atomList = listPermute(atomList,vertexPermutation)
    elecStateList = listPermute(elecStateList,vertexPermutation)
    nodeColorList = listPermute(nodeColorList,vertexPermutation)
    permutedGraphObject.vs['atom'] = atomList
    permutedGraphObject.vs['elecState'] = elecStateList
    permutedGraphObject.vs['color'] = nodeColorList

    #Grab the new adjacency list in edge format and use to find a
    #permutation vector that will permute edge attributes in the right
    #way. Then permute the edge attribute vectors properly.
    newEdgeAdjList = permutedGraphObject.get_edgelist()
    edgePermutation = findEdgePermutation(edgeAdjList,
                                          newEdgeAdjList,
                                          vertexPermutation)
    bondList = listPermute(bondList,edgePermutation)
    edgeColorList = listPermute(edgeColorList,edgePermutation)
    permutedGraphObject.es['bond'] = bondList
    permutedGraphObject.es['edgeColor'] = edgeColorList

    #TODO(goxberry@mit.edu): Remove debugging code
#    print graphObject.vs['color']
#    print vertexPermutation
#    print permutedGraphObject.vs['color']
    #Works when labels are ignored...
#    print permutedGraphObject.isomorphic_vf2(graphObject,\
#           permutedGraphObject.vs['color'],
#           graphObject.vs['color'],
#           permutedGraphObject.es['edgeColor'],
#           graphObject.es['edgeColor'])


    return permutedGraphObject
#Tested. TODO(goxberry@mit.edu): Add test here.

def listPermute(inputList,permutation):
    """
    Purpose: Quick hack job, unoptimized function that takes a given
    input list and a permutation vector and returns the permuted list.

    Inputs: inputList = the list to be permuted
            permutation = a list of indices; these must be integers
            between 0 and len(inputList) - 1 inclusive, without repeats;
            in other words, it must be a permutation of range(len(inputList)).
            I want element index to be moved to element permutation[index]
            (or, schematically:
            inputList[element] -> permutedList[permutation[element]])

    Output: permutedList = a list consisting of the elements of inputList,
            but with the order permuted

    Assertion checks: None, but there should be checks to ensure that
            permutation is a permutation of the list range(len(inputList)) and
            that len(permutation) == len(inputList).

    """

    permutedList = range(len(inputList))
    for element in xrange(len(inputList)):
        permutedList[permutation[element]] = inputList[element]

    return permutedList
#Testing:
#>>> import igraph
#>>> import rmgUtils
#>>> methane1 = igraph.Graph(5)
#>>> methane1.add_edges([(0,1),(0,2),(0,3),(0,4)])
#>>> methane1.vs['color'] = [1,0,0,0,0]
#>>> methane2 = methane1.permute_vertices([4,0,1,2,3])
#>>> methane2.vs['color'] = [0,0,0,0,1]
#>>> methane1.isomorphic_vf2(methane2,methane1.vs['color'],methane2.vs['color'])
#True
#>>> permutation = [4,0,1,2,3]
#rmgUtils.listPermute(methane1.vs['color'],permutation)
#[0, 0, 0, 0, 1]

def findEdgePermutation(oldEdgeAdjList, newEdgeAdjList, vertexPermutation):
    """
    Purpose: Another quick hack job. Function takes the old and new edge
    adjacency lists and figures out the mapping isomorphism that takes
    the old adjacency list to the new adjacency list.

    Inputs: oldEdgeAdjList = the adjacency list (in edge form) of a
            graph before it is permuted in
            SelectPermutedSubsetOfGraphsNoRepeats
            newEdgeAdjList = the adjacency list (in edge form) of a
            graph after it is permuted in
            SelectPermutedSubsetOfGraphsNoRepeats
            vertexPermutation = list that describes how vertices of
            graph before permutation are mapped to vertices after permutation

    Output: edgePermutation = vector of length len(oldeEdgeAdjList) that
            maps elements of oldEdgeAdjList to newEdgeAdjList

    Assertion checks: None, but there should be checks to ensure that
            len(oldEdgeAdjList) == len(newEdgeAdjList), and to ensure
            that neither oldEdgeAdjList nor newEdgeAdjList have any
            repeated entries.

    """

    permutation = []
    for element in oldEdgeAdjList:
        (firstNode,secondNode) = element
        newFirstNode = vertexPermutation[firstNode]
        newSecondNode = vertexPermutation[secondNode]
        if newFirstNode > newSecondNode:
            (newFirstNode,newSecondNode) = (newSecondNode,newFirstNode)
        permutation += [newEdgeAdjList.index((newFirstNode,newSecondNode))]

    return permutation

def IsSpeciesInList(species,speciesList):
    """
    Purpose: Given a species in the format:
    [<graph title>,<igraph.Graph object>]

    and a list of species in the format:
    [[<graph title>,<igraph.Graph object>],
     [<graph title>,<igraph.Graph object>],
     [<graph title>,<igraph.Graph object>],
     :
     :
     :
     [<graph title>,<igraph.Graph object>]]

     determine if the species in the first argument is isomorphic to
     any species in the list.

    Input: species = species to be checked for membership in a list of species
           (the 'core' species)
           speciesList = list of species that we search for graph
```

```
            isomorphisms (comparing against the graph in species)

    Output: isInCore = Boolean result; True if the graph component of species
    is isomorphic to the graph component of any species in speciesList.
    index = returns index of the isomorphic graph in speciesList if
    isInCore == True; returns -1 if isInCore == False

    Assertion checks: None; should check that the arguments have the proper
    format.

    """

    import igraph
    speciesGraph = species[1]

    for index in xrange(len(speciesList)):
        compoundGraph = speciesList[index][1]
        sameNumberVertices = (speciesGraph.vcount() == compoundGraph.vcount())
        isIsomorphic =  sameNumberVertices and \
            speciesGraph.isomorphic_vf2(compoundGraph,speciesGraph.vs['color'],
                                        compoundGraph.vs['color'])
        #print 'same # Vertices = ',sameNumberVertices
        #print 'is isomorphic = ',isIsomorphic
        if isIsomorphic:
        #    print ''
            return (True,index)

    #print ''
    return (False,-1)
#Testing: (Should return true because a species from the subset list
#          should be isomorphic to some species in the whole dictionary file.)
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.ImportChemGraphsFromFile('RMGDictionarySmallTestCase.txt')
#>>> subsetList = rmgUtils.SelectPermutedSubsetOfGraphsNoRepeats(listOfGraphs,5)
#>>> rmgUtils.IsSpeciesInList(subsetList[0],listOfGraphs)
#True

def GrowCoreSpeciesSerial(coreSpecies,speciesDatabase):
    """
    Purpose: Implements the steps within the loop of the RMG serial prototype
    algorithm. Specifically, it

    a)Randomly selects graphs from the master database (dictionary file) of
    species imported from RMG output.
    b)Uses the graph isomorphism algorithm to test for duplicates among
    postulated species, discarding duplicates.
    c)Uses the graph isomorphism algorithm to test for duplicates between
    postulated and core species.
    d)Add remaining postulated species to set of core species

    Inputs: coreSpecies = list of species currently in the core of the
            mechanism
            speciesDatabase = list of all species imported from an RMG
            dictionary file; we sample from this list in the RMG
            prototype to grow the core species

    Outputs: newCoreSpecies = list of species in the core after considering
             new postulated species for addition

    Assertion checks: None; this function was written for refactoring purposes;
    it made testing much easier, and makes the code easier to read and debug.
    All assertion checks should be performed by the functions called within
    this function.

    """
    import random
    #import math

    #Randomly selects graphs from the master database of species imported
    #from RMG output. The # of species selected from the master database
    #is proportional to the number of species currently in the core. The
    #species selected are called the postulated species.
    sizeOfCore = len(coreSpecies)
    fractionOfCoreSelected = .5
    sizeOfSubset = int(fractionOfCoreSelected * sizeOfCore)
    #If there is an error in this subroutine, look in the routine called
    #in the line below...
    postulatedSpecies = SelectSubsetOfGraphsInclRepeats(speciesDatabase,
                                                         sizeOfSubset)

    #Use the graph isomorphism algorithm to test for duplicates among
    #postulated species, discarding duplicates.
    postulatedSpecies = checkPostulatedSpeciesSerial(postulatedSpecies)

    #Use the graph isomorphism algorithm to test for duplicates between
    #postulated and core species.
    postulatedSpecies = checkCoreSpeciesSerial(postulatedSpecies,
                                               coreSpecies)

    #Add remaining postulated species to set of core species.
    newCoreSpecies = coreSpecies + postulatedSpecies

    return newCoreSpecies
#Testing code:
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.ImportChemGraphsFromFile('RMGDictionarySmallTestCase.txt')
#>>> coreSpeciesList = \
#rmgUtils.SelectPermutedSubsetOfGraphsNoRepeats(listOfGraphs,10)
#>>> print coreSpeciesList
#[['CH4(2)', <igraph.Graph object at 0x8c6202c>],\
# ['C5H7J(30)', <igraph.Graph object at 0x8c585ac>],
# ['H2(3)', <igraph.Graph object at 0x8c5872c>],
# ['C5H7J(203)', <igraph.Graph object at 0x8c5862c>],
# ['C2H4(638)', <igraph.Graph object at 0x8c58c2c>],
# ['HJ(25)', <igraph.Graph object at 0x8c58d2c>],
# ['CH3J(24)', <igraph.Graph object at 0x8c589ac>],
# ['C5H8(55)', <igraph.Graph object at 0x8c586ac>],
# ['C3H3J(339)', <igraph.Graph object at 0x8c58e2c>],
# ['C5H7J(85)', <igraph.Graph object at 0x8c58cac>]]
#>>> newCoreSpeciesList = \
#rmgUtils.GrowCoreSpeciesSerial(coreSpeciesList,listOfGraphs)
#>>> len(coreSpeciesList)
#10
#>>> len(newCoreSpeciesList)
#12
#>>> print newCoreSpeciesList
#[['CH4(2)', <igraph.Graph object at 0x8c6202c>],
# ['C5H7J(30)', <igraph.Graph object at 0x8c585ac>],
# ['H2(3)', <igraph.Graph object at 0x8c5872c>],
# ['C5H7J(203)', <igraph.Graph object at 0x8c5862c>],
# ['C2H4(638)', <igraph.Graph object at 0x8c58c2c>],
# ['HJ(25)', <igraph.Graph object at 0x8c58d2c>],
# ['CH3J(24)', <igraph.Graph object at 0x8c589ac>],
# ['C5H8(55)', <igraph.Graph object at 0x8c586ac>],
# ['C3H3J(339)', <igraph.Graph object at 0x8c58e2c>],
# ['C5H7J(85)', <igraph.Graph object at 0x8c58cac>],
# ['C2H6(36)', <igraph.Graph object at 0x8c58b2c>],
# ['C10H14(81)', <igraph.Graph object at 0x8c620ac>]]

def checkPostulatedSpeciesSerial(postulatedSpecies):
    """
    Purpose: Uses graph isomorphism algorithm to test for duplicates among
    postulated species and discards duplicates.

    Inputs: postulatedSpecies = list of postulated species

    Outputs: postulatedSpeciesNoRepeats = list of postulated species with
    duplicates culled.

    Assertion checks: None; should check for proper format of input.

    """

    import igraph

    #Deep copy for debugging purposes...
    #import copy
    #postSpecies = copy.deepcopy(postulatedSpecies)
    #Need to keep in mind that mutable types are passed into functions
    #like pointers and can be modified by the function into which it is
    #passed; this is true here. However, since we're using this function
    #to modify the postulated species, it's okay that we're scrambling it.
    #Just keep this in mind for testing purposes...
    postSpecies = postulatedSpecies
    postulatedSpeciesNoRepeats = []

    #While we still have postulated species to check...
    while (len(postSpecies) > 0):
        #Pop the current species and check to see if it is repeated
        currentSpecies = postSpecies.pop(0)
        (isSpeciesRepeated,repeatIndex) = IsSpeciesInList(currentSpecies,
                                                          postSpecies)

        #Add the species to the list of postulated species w/o repeats
        postulatedSpeciesNoRepeats += [currentSpecies]

        #If the species was repeated in the list, delete the repeat from the
        #list and continue searching for repeats until all repeats are purged
        #from the list.
        while isSpeciesRepeated:
            del postSpecies[repeatIndex]
            (isSpeciesRepeated,repeatIndex) = \
                IsSpeciesInList(currentSpecies,postSpecies)

    return postulatedSpeciesNoRepeats
#Testing code:
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.ImportChemGraphsFromFile('RMGDictionarySmallTestCase.txt')
#>>> subsetList = rmgUtils.SelectSubsetOfGraphsInclRepeats(listOfGraphs, 10)
#>>> subsetList
#[['C5H6(124)', <igraph.Graph object at 0x92688ac>],\
# ['C10H14(81)', <igraph.Graph object at 0x92684ac>],\
# ['CH3J(24)', <igraph.Graph object at 0x925ed2c>],\
# ['C5H6(124)', <igraph.Graph object at 0x92688ac>],\
# ['CH3J(24)', <igraph.Graph object at 0x925ed2c>],\
# ['C2H6(36)', <igraph.Graph object at 0x926822c>],\
# ['C5H7J(203)', <igraph.Graph object at 0x92685ac>],\
# ['C5H8(55)', <igraph.Graph object at 0x92687ac>],\
# ['H2(3)', <igraph.Graph object at 0x925ec2c>],\
# ['C4H6(748)', <igraph.Graph object at 0x926872c>]]
#>>> rmgUtils.checkPostulatedSpeciesSerial(subsetList)
#[['C5H6(124)', <igraph.Graph object at 0x916df2c>],\
# ['C10H14(81)', <igraph.Graph object at 0x916db2c>],\
# ['CH3J(24)', <igraph.Graph object at 0x916dc2c>],\
# ['C2H6(36)', <igraph.Graph object at 0x916dcac>],\
# ['C5H7J(203)', <igraph.Graph object at 0x916dd2c>],\
# ['C5H8(55)', <igraph.Graph object at 0x916ddac>],\
# ['H2(3)', <igraph.Graph object at 0x916de2c>],\
# ['C4H6(748)', <igraph.Graph object at 0x916deac>]]

def checkCoreSpeciesSerial(postulatedSpecies,coreSpecies):
    """
    Purpose: Uses graph isomorphism algorithm to test for postulated species
    that are duplicated in the core species.

    Inputs: postulatedSpecies = list of postulated species
            coreSpecies = list of core species

    Outputs: postulatedSpeciesNoRepeats = list of postulated species
    that are not duplicated in the core.

    Assertion checks: None; should check for proper format of input, also
    should check that coreSpecies has no repeats. However, since we have
    implemented this algorithm so that coreSpecies should never have
    repeats, this invariant should be preserved by this function.

    """

    import igraph

    #Deep copy for debugging purposes...
    #import copy
    #postSpecies = copy.deepcopy(postulatedSpecies)
    #Need to keep in mind that mutable types are passed into functions
    #like pointers and can be modified by the function into which it is
    #passed; this is true here. However, since we’re using this function
    #to modify the postulated species, it’s okay that we’re scrambling it.
    #Just keep this in mind for testing purposes.
    postSpecies = postulatedSpecies
    postulatedSpeciesNoRepeats = []

   #While we still have postulated species to check...
    while (len(postSpecies) > 0):
        #Pop the current species and check to see if it is repeated in
        #the core
        currentSpecies = postSpecies.pop(0)
        (isSpeciesRepeated,repeatIndex) = IsSpeciesInList(currentSpecies,
                                                          coreSpecies)

        #If the current species isn’t repeated in the list of core species,
        #add it to the list of postulated species w/o repeats
        if not isSpeciesRepeated:
            postulatedSpeciesNoRepeats += [currentSpecies]

    return postulatedSpeciesNoRepeats
#Testing code:
#>>> import rmgUtils
#>>> listOfGraphs = \
#rmgUtils.ImportChemGraphsFromFile(’RMGDictionarySmallTestCase.txt’)
#>>> coreSpeciesList = \
#rmgUtils.SelectPermutedSubsetOfGraphsNoRepeats(listOfGraphs,10)
#
#>>> postulatedSpeciesList = \
#rmgUtils.SelectSubsetOfGraphsInclRepeats(listOfGraphs,5)
#>>> print coreSpeciesList
#[[’CH4(2)’, <igraph.Graph object at 0x8c6202c>],\
# [’C5H7J(30)’, <igraph.Graph object at 0x8c585ac>],
# [’H2(3)’, <igraph.Graph object at 0x8c5872c>],
# [’C5H7J(203)’, <igraph.Graph object at 0x8c5862c>],
# [’C2H4(638)’, <igraph.Graph object at 0x8c58c2c>],
# [’HJ(25)’, <igraph.Graph object at 0x8c58d2c>],
# [’CH3J(24)’, <igraph.Graph object at 0x8c589ac>],
# ['C5H8(55)', <igraph.Graph object at 0x8c586ac>],
# ['C3H3J(339)', <igraph.Graph object at 0x8c58e2c>],
# ['C5H7J(85)', <igraph.Graph object at 0x8c58cac>]]
#>>> print postulatedSpeciesList
#[['C10H14(81)', <igraph.Graph object at 0x8c580ac>],
# ['HJ(25)', <igraph.Graph object at 0x8b79eac>],
# ['C8H10(751)', <igraph.Graph object at 0x8c5842c>],
# ['C5H6(124)', <igraph.Graph object at 0x8c584ac>],
# ['C5H7J(203)', <igraph.Graph object at 0x8c581ac>]]
#>>> rmgUtils.checkCoreSpeciesSerial(postulatedSpeciesList,coreSpeciesList)
#[['C10H14(81)', <igraph.Graph object at 0x8c5882c>],
# ['C8H10(751)', <igraph.Graph object at 0x8c587ac>],
# ['C5H6(124)', <igraph.Graph object at 0x8c588ac>]]

def RMGSerialPrototype(dictionaryFileName):
    """
    Purpose: Implements the RMG serial prototype algorithm in the progress
    report.

    Input: dictionaryFileName = text file containing text encodings of
    ChemGraphs in the following format:

    <graph object>
    \n
    \n
    <graph object>
    \n
    \n
    (repeated 'til end-of-file)

    Output: coreSpecies = set of core species at last iteration.

    Assertion checks: None; this function is essentially the main driver
    function, so no assertion checks are needed at this level of abstraction.
    Instead, all of the functions called in this function should have
    the appropriate assertion checks.

    """

    #Load the master database of ChemGraphs from RMG output.
    speciesMasterDatabase = ImportChemGraphsFromFile(dictionaryFileName)
    numSpeciesInDatabase = len(speciesMasterDatabase)

    #Randomly select a part of this database to be our initial set of core
    # species. In order to decrease our chance of finding trivial graph
    # isomorphisms, permute all graphs in the database.
    initFracOfDatabaseInCore = .4
    initCoreSize = int(numSpeciesInDatabase * initFracOfDatabaseInCore)
    coreSpecies = \
        SelectPermutedSubsetOfGraphsNoRepeats(speciesMasterDatabase,
                                              initCoreSize)

    #Grow core species until all species included or until maximum
    #number of iterations is reached
    maxItersDueToRunTimeErrors = 100
    maxIters = min(len(speciesMasterDatabase),maxItersDueToRunTimeErrors)
    iterCount = 0

    #Set up Boolean conditions for max # iters reached or all species incl.
    isIterCountLessThanMax = (iterCount < maxIters)
    areSomeSpeciesNotInCore = (len(coreSpecies) < numSpeciesInDatabase)
    while areSomeSpeciesNotInCore and isIterCountLessThanMax:
        #Perform one iteration of growing the core
        coreSpecies = GrowCoreSpeciesSerial(coreSpecies,speciesMasterDatabase)

        #Update iteration count and Boolean conditions
        iterCount += 1
        isIterCountLessThanMax = (iterCount < maxIters)
        areSomeSpeciesNotInCore = (len(coreSpecies) < numSpeciesInDatabase)

    print 'Output Summary:'
    print 'iterCount = ',iterCount
    print 'Max iterations reached? ',not(isIterCountLessThanMax)
    print '# of species in RMG dictionary file = ',numSpeciesInDatabase
    print '# of species in core @ end = ',len(coreSpecies)
    print 'Are all species from dictionary file in core? ',\
        not(areSomeSpeciesNotInCore)
    print ''
    return coreSpecies
#Testing: Since all functions within this function have been tested, all
#testing should achieve here is to debug for syntax at runtime and
#look to see if the results vaguely make sense

def GrowCoreSpeciesParallel(coreSpecies,speciesDatabase,
                            comm):
    """
    Purpose: Implements the steps within the loop of the RMG parallel prototype
    algorithm. Specifically, it

    a)Randomly selects graphs from the master database (dictionary file) of
    species imported from RMG output.
    b)Distributes the postulated species evenly among all processors and use
    the graph isomorphism algorithm to check that postulated species do
    not duplicate any of the core species. Dsicare duplicates and send
    remaining postulated speices back to first processor to check for
    duplication among postulated species.
    c)Uses the graph isomorphism algorithm to test for duplicates between
    postulated and core species.
    d)Add remaining postulated species to set of core species on first
    processor.

    Inputs: coreSpecies = list of species currently in the core of the
            mechanism
            speciesDatabase = list of all species imported from an RMG
            dictionary file; we sample from this list in the RMG
            prototype to grow the core species
            comm = MPI communicator used to pass msgs

    Outputs: newCoreSpecies = list of species in the core after considering
             new postulated species for addition

    Assertion checks: None; this function was written for refactoring purposes;
    it made testing much easier, and makes the code easier to read and debug.
    All assertion checks should be performed by the functions called within
    this function.

    """
    import random
    import cPickle
    from mpi4py import MPI

    headNodeRank = 0
    procRank = comm.rank
    commSize = comm.size

    if procRank == 0:
    #Randomly selects graphs from the master database of species imported
    #from RMG output. The # of species selected from the master database
    #is proportional to the number of species currently in the core. The
    #species selected are called the postulated species.
        sizeOfCore = len(coreSpecies)
        fractionOfCoreSelected = .5
        sizeOfSubset = int(fractionOfCoreSelected * sizeOfCore)
        postulatedSpecies = SelectSubsetOfGraphsInclRepeats(speciesDatabase,
                                                        sizeOfSubset)
    #Use the graph isomorphism algorithm to test for duplicates among
    #postulated species, discarding duplicates.
        postulatedSpecies = checkPostulatedSpeciesSerial(postulatedSpecies)

    #   For debugging purposes... <debug>
    #   print 'Postulated species:'
    #   print postulatedSpecies
    #   </debug>

    else:
        postulatedSpecies = []

    #Algorithm works up to here, but there's an error in this function below
    #this point; right now, ALL postulated species are being added...

    #Uses graph isomorphism algorithm to test for postulated species
    #that are duplicated in the core species.
    mergedPostulatedSpecies = comm.bcast(postulatedSpecies,headNodeRank)
    splitPostulatedSpecies = ListSplitForScatter(mergedPostulatedSpecies,comm)

    #For debugging purposes... <debug>
    #if comm.rank == 0:
    #    print 'The postulated species are:'
    #    print splitPostulatedSpecies
```

```
    #</debug>

    postulatedSpeciesToCheck = comm.scatter(splitPostulatedSpecies,headNodeRank)

    checkedPostulatedSpecies = checkCoreSpeciesSerial(postulatedSpeciesToCheck,
                                                      coreSpecies)

    #Gather all parts of the checked postulated species and then concatenate
    #the pieces; we can't do this using an MPI reduced because it won't work.
    #(I've already tried.)
    postulatedSpeciesGathered = comm.gather(checkedPostulatedSpecies,
                                            headNodeRank)

    #After gathering, we need to flatten the postulatedSpeciesGathered...
    if (postulatedSpeciesGathered == None):
        postulatedSpeciesGathered = []
    postulatedSpeciesReduced = sum(postulatedSpeciesGathered,[])

    #For debugging: <debug>
    #if procRank == 0:
    #     print 'Gathered the following postulated species after checking'
    #     print 'against core species:'
    #     print postulatedSpeciesGathered
    #     print ''
    #     print 'Added the following postulated species:'
    #     print postulatedSpeciesReduced
    #     print ''
    #</debug>

    if procRank == 0:
    #Add remaining postulated species to set of core species.
        newCoreSpecies = coreSpecies + postulatedSpeciesReduced
        return newCoreSpecies
    else:
        return []


def RMGParallelPrototype(dictionaryFileName):
    """
    Purpose: Implements the RMG parallel prototype algorithm in the progress
    report.

    Input: dictionaryFileName = text file containing text encodings of
    ChemGraphs in the following format:

    <graph object>
    \n
    \n
    <graph object>
    \n
    \n
    (repeated 'til end-of-file)
    Output: coreSpecies = set of core species at last iteration.

    Assertion checks: None; this function is essentially the main driver
    function, so no assertion checks are needed at this level of abstraction.
    Instead, all of the functions called in this function should have
    the appropriate assertion checks.

    """
    import igraph
    import cPickle
    from mpi4py import MPI

    #Initialize a communicator and figure out the processor rank.
    comm = MPI.COMM_WORLD
    processorRank = comm.Get_rank()
    headNodeRank = 0
    #print 'This processor is processor ',processorRank,'!\n'

    #On the first processor...
    if processorRank == 0:

#        print 'Initialized communicator successfully!'

        #Load the master database of ChemGraphs from RMG output.
        headNodeSpeciesMasterDatabase = ImportChemGraphsFromFile(
            dictionaryFileName)
        headNodeNumSpeciesInDatabase = len(headNodeSpeciesMasterDatabase)

        #Randomly select a part of this database to be our initial set of core
        # species. In order to decrease our chance of finding trivial graph
        # isomorphisms, permute all graphs in the database.
        initFracOfDatabaseInCore = .4
        initCoreSize = int(headNodeNumSpeciesInDatabase *
                           initFracOfDatabaseInCore)
        headNodeCoreSpecies = \
            SelectPermutedSubsetOfGraphsNoRepeats(headNodeSpeciesMasterDatabase,
                                                  initCoreSize)

        #Grow core species until all species included or until maximum
        #number of iterations is reached
        headNodeMaxIters = min(len(headNodeSpeciesMasterDatabase),
                               100)
    else:
        headNodeCoreSpecies = []
        headNodeSpeciesMasterDatabase = []
        headNodeMaxIters = 0
        headNodeNumSpeciesInDatabase = 0

#Broadcast the core species to all processors using mpi4py.Comm.bcast;
#this is the 'slow' way, but it is also much simpler.

    #Need to sync up coreSpecies, max # iterations, number of species in
    #master database, iteration count, and Boolean conditions
```

```
   comm.barrier()
#   if comm.rank == 0:
#       print 'Reached first barrier!'

   coreSpecies = comm.bcast(headNodeCoreSpecies,headNodeRank)
#   if comm.rank == 0:
#       print 'Broadcast coreSpecies!'
   speciesMasterDatabase = comm.bcast(headNodeSpeciesMasterDatabase,
                                      headNodeRank)
#   if comm.rank == 0:
#       print 'Broadcast speciesMasterDatabase!'
   maxIters = comm.bcast(headNodeMaxIters,headNodeRank)
#   if comm.rank == 0:
#       print 'Broadcast maxIters!'
   numSpeciesInDatabase = comm.bcast(headNodeNumSpeciesInDatabase,headNodeRank)
#   if comm.rank == 0:
#       print 'Broadcast numSpeciesInDatabase!'
   iterCount = 0

   #Set up Boolean conditions for max # iters reached or all species incl.
   isIterCountLessThanMax = (iterCount < maxIters)
   areSomeSpeciesNotInCore = (len(coreSpecies) < numSpeciesInDatabase)

   if comm.rank == 0:
       print 'Entering loop successfully!'

   #Entering the loop, all loop quantities should be synced up.

   #For debugging...<debug>
   #if processorRank == 0:
   #       print 'At iteration %s, the core species list is:' %(iterCount)
   #       print coreSpecies
   #</debug>

   while areSomeSpeciesNotInCore and isIterCountLessThanMax:
       #Perform one iteration of growing the core using the
       #parallel algorithm; species are updated in parallel, with
       #all of the necessary communications being carried out within
       #GrowCoreSpeciesParallel, which saves me from throwing around
       #even more MPI primitives in this driver...
       coreSpecies = \
           GrowCoreSpeciesParallel(coreSpecies,speciesMasterDatabase,comm)

       #For debugging... <debug>
       #if processorRank == 0:
       #    print 'At iteration %s, the core species list is:' %(iterCount)
       #    print coreSpecies
       #</debug>

       #Sync up coreSpecies, update iteration count and Boolean conditions;
       #at the end of the loop for a given iteration, iterCount and the
       #Boolean conditions should all have the same value (locally) on every
       #processor.
```

```
        comm.barrier()
#        if ((comm.rank == 0) and (iterCount % 20 == 0)):
#            print 'Before broadcast during iteration ',iterCount
        coreSpecies = comm.bcast(coreSpecies,headNodeRank)
#        if ((comm.rank == 0) and (iterCount % 20 == 0)):
#            print 'After broadcast during iteration ',iterCount
        iterCount += 1
        isIterCountLessThanMax = (iterCount < maxIters)
        areSomeSpeciesNotInCore = (len(coreSpecies) < numSpeciesInDatabase)

    if processorRank == 0:
        print 'Output Summary:'
        print 'iterCount = ',iterCount
        print 'Max iterations reached? ',not(isIterCountLessThanMax)
        print '# of species in RMG dictionary file = ',numSpeciesInDatabase
        print '# of species in core @ end = ',len(coreSpecies)
        print 'Are all species from dictionary file in core? ',\
            not(areSomeSpeciesNotInCore)
        print ''
        print coreSpecies
        return coreSpecies
#Testing: Since all functions within this function have been tested, all
#testing should achieve here is to debug for syntax at runtime and
#look to see if the results vaguely make sense

def ListSplitForScatter(listToSplit,comm):
    """Purpose: Split listToSplit into commSize lists of roughly equal length.
                Earlier elements in the list will be longer than later elements
                of the list. Idea is to have a function that will split up
                lists into the format required by mpi4py.Comm.scatter.

    Inputs: listToSplit = the list to be split
            comm = communicator

    Outputs: splitList = the list, split into roughly equal parts.
    """
    #We break up the splitList into two sublists:
    #The longer list part will have longer lists as elements, and it contains
    # a number of elements equal to ''whatever is left'' (i.e., the remainder)
    #after dividing the length of the list to split by the number of
    #processors in commSize.
    #The shorter list part will have shorter lists as elements; it consists
    #of commSize - longerListPart elements.
    commSize = comm.size
    lenListToSplit = len(listToSplit)
    shortStride = int(lenListToSplit/commSize)
    longerListPart = lenListToSplit % commSize
    longStride = shortStride + (longerListPart > 0)
    shorterListPart = commSize - longerListPart

    splitList = [listToSplit[i*longStride:(i+1)*longStride]
                 for i in range(longerListPart)]
    numEltsInLongerPart = longerListPart * longStride
    splitList.extend([listToSplit[(numEltsInLongerPart+(i*shortStride)):
                              (numEltsInLongerPart+((i+1)*shortStride))]
                 for i in range(shorterListPart)])

    #For debugging:
    #print ''
    #print 'lenListToSplit = ',lenListToSplit
    #print 'shortStride = ',shortStride
    #print 'longStride = ',longStride
    #print 'shorterListPart = ',shorterListPart
    #print 'longerListPart = ',longerListPart
    #print 'numEltsInLongerPart =', numEltsInLongerPart
    #print ''

    return splitList
```

Serial driver script:

```
#!/usr/bin/env python
import rmgUtils
import time
import random
random.seed(0)
startTime = time.time()
#rmgUtils.RMGSerialPrototype('RMGDictionarySmallTestCase.txt')
#rmgUtils.RMGSerialPrototype('RMGDictionaryLargeTestCase.txt')
rmgUtils.RMGSerialPrototype('RMGDictionaryVeryLargeTestCase.txt')
endTime = time.time()
print 'Time taken by serial prototype = %s s' %(endTime - startTime)
```

Parallel driver script:

```
#!/usr/bin/env python
import rmgUtils
from mpi4py import MPI
import random
random.seed(0)
comm = MPI.COMM_WORLD
startTime = MPI.Wtime()
#rmgUtils.RMGParallelPrototype('RMGDictionarySmallTestCase.txt')
#rmgUtils.RMGParallelPrototype('RMGDictionaryLargeTestCase.txt')
rmgUtils.RMGParallelPrototype('RMGDictionaryVeryLargeTestCase.txt')
endTime = MPI.Wtime()
if comm.rank == 0:
    print 'Time taken by parallel prototype = %s s' %(endTime - startTime)

#Common test code:
#---------------------------------------------------------------------
#from mpi4py import MPI
#import numpy
#Initialize a communicator and figure out the processor rank.
#comm = MPI.COMM_WORLD
#processorRank = comm.rank
```

```
#commSize = comm.size
#headNodeRank = 0

#Example scatter: size restriction; need to create a list of comm.size
#elts
#----------------------------------------------------------------------
#stuff = range(16)
#stride = int(len(stuff)/commSize)
#if ((len(stuff)/commSize) - int(len(stuff))/
#packedStuff = [stuff[(i*stride):((i*stride)+stride)] for i in range(commSize)]
#newStuff = comm.scatter(packedStuff,headNodeRank)
#packedNewNewStuff = comm.gather(newStuff,headNodeRank)
#newNewStuff = sum(packedNewNewStuff,[])

#if processorRank == 0:
#    newNewStuff = sum(packedNewNewStuff, [])
#    for elem in newNewStuff:
#        print elem, ' is on processor ',processorRank
#    print packedNewNewStuff

#Example broadcast: no size restrictions...
#-----------------------------------------
#stuff = range(16)
#newStuff = comm.bcast(stuff,headNodeRank)

#if processorRank == 2:
#    print newStuff, 'is on processor ',processorRank
```