ACS SyntheticBiology

Research Article

pubs.acs.org/synthbio

Directed Acyclic Graph-Based Technology Mapping of Genetic Circuit Models

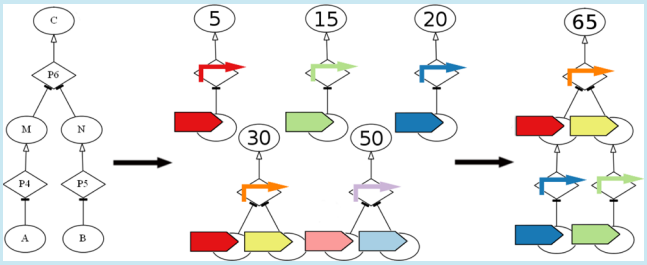
Nicholas Roehner*,† and Chris J. Myers*,‡

†Department of Bioengineering, University of Utah, Salt Lake City 84112, United States

‡Department of Electrical and Computer Engineering, University of Utah, Salt Lake City 84112, United States

Supporting Information

ABSTRACT: As engineering foundations such as standards and abstraction begin to mature within synthetic biology, it is vital that genetic design automation (GDA) tools be developed to enable synthetic biologists to automatically select standardized DNA components from a library to meet the behavioral specification for a genetic circuit. To this end, we have developed a genetic technology mapping algorithm that builds on the directed acyclic graph (DAG) based mapping techniques originally used to select parts for digital electronic circuit designs and implemented it in our GDA tool, iBioSim. It is among the first genetic technology mapping algorithms to adapt techniques from electronic circuit design, in particular the use of a cost function to guide the search for an optimal solution, and perhaps that which makes the greatest use of standards for describing genetic function and structure to represent design specifications and component libraries. This paper demonstrates the use of our algorithm to map the specifications for three different genetic circuits against four randomly generated libraries of increasing size to evaluate its performance against both exhaustive search and greedy variants for finding optimal and near-optimal solutions.



KEYWORDS: *genetic design automation, technology mapping, part selection, SBML, SBOL, iBioSim*

As engineering foundations such as standards and abstraction begin to mature within synthetic biology,[1,2] one limiting factor for the realization of more complex genetic circuits is the availability and effectiveness of *genetic design automation* (GDA) tools that use these foundations to separate the design of genetic circuits from their physical construction. In particular, it is vital that GDA tools be developed to enable synthetic biologists to construct standardized descriptions for the structure and function of genetic components[3−5] and to automatically select these components from a library to meet the behavioral specification for a genetic circuit. By encoding knowledge in tools and standardized data, GDA can lower the barrier to entry for new designers and promote the reuse of experimentally proven genetic components. Through intelligent automation, GDA can make the design of more complex genetic circuits tractable and decrease the length of design and test cycles. Toward this end, we have developed a *genetic technology mapping* algorithm that builds on the *directed acyclic graph* (DAG) based mapping techniques originally used to select parts for digital electronic circuit designs[6] and implemented it in our GDA tool iBioSim.[7,8]

While iBioSim is one of the first GDA tools to adapt DAG-based mapping techniques from *electronic design automation* (EDA), it is by no means the first GDA tool to adopt terminology, concepts, and methods from electronic circuit design in general. Furthermore, synthetic biology as a whole has encompassed research into genetic circuits for which high-level function is commonly described in terms of digital logic gates,[9−11] an abstraction borrowed from electronic circuit design. While this paper does not attempt to estimate the percentage of all known genetic circuits that behave as digital logic gates, it is worth noting that even randomly synthesized genetic circuits have been found to exhibit phenotypic behavior resembling digital logic.[12] Owing to the relative simplicity of digital logic and the existence of many tools and techniques that make use of it in other engineering disciplines, there is a clear opportunity for GDA tools to support the *digital logic abstraction* in synthetic biology. Of equal interest are modifications of digital logic techniques to better reflect biological reality, for instance multivalued logic methods that can represent more than two states for a biological signal[13] and mixed-signal techniques that can combine discrete and continuous descriptions of biological behavior.[14−16]

At present, our DAG-based approach to genetic technology mapping introduces digital logic from EDA to GDA. Accordingly, our behavioral representations for genetic circuits and components are regulatory DAGs with a particular logical interpretation that determines, for instance, whether a promoter activated by two different transcription factors is an AND

Special Issue: IWBDA 2013

Received: September 9, 2013
Published: March 20, 2014

motif or an OR motif. While such DAGs do not fully capture the great diversity of genetic structure and function that is found in nature and synthetic biology, they do capture digital logic operations such as NOT and NOR. Hence, these DAGs are sufficient for representing the abstract behavior if not the actual mechanism underlying genetic circuits for which the measured relationship between steady-state input and output is fairly sigmoidal and, therefore, amenable to abstraction as digital logic. In light of these considerations, the approach described in this paper is best used to design genetic circuits that implement combinational logic, such as composite genetic logic gates and multiplexers for coordinating all-or-nothing responses to multiple environmental signals,[17−21] rather than to design analog genetic circuits for more graded responses to environmental signals[22] or metabolic pathways for optimal flux of chemically manufactured materials.[23,24]

Within the domain of synthetic biology that adheres to digital logic, our approach focuses on addressing one of the major differences between GDA and EDA, namely that genetic components can produce signals with a variety of molecular identities, unlike electronic components. When composing genetic components to form a genetic circuit design, the molecular identities of their signals must be accounted for to ensure proper connections between components and avoid undesirable cross-talk. The implication of this restriction is that each choice of a genetic component for a genetic circuit design precludes the choice of other components that would introduce cross-talk. Hence, optimally mapping from a behavioral specification to a genetic circuit design can be a very computationally expensive problem, one in which every valid combination of genetic components that can possibly produce the specified behavior may have to be explored in order to ensure that the optimal circuit design is found. This paper attempts to adapt computer science techniques commonly used in EDA such as *dynamic programming* and *branch-and-bound* in order to lower the average time taken to find this optimum.

For GDA, the hypothetical quality of a genetic circuit design is often defined with respect to circuit parameters of interest such as number of genes, length in base pairs, signal delay, or signal stability. For the purposes of design, these circuit parameters can be derived from standardized descriptions of a circuit's constituent genetic components, including descriptions of their structure, models of their behavior, and existing characterization data. The actual quality of a given genetic circuit design, however, can only be determined by constructing the genetic circuit and testing it in a lab. This paper focuses for simplicity on minimizing the total length in base pairs of our genetic circuit designs, as this parameter is at least partly related to other parameters such as delay in transcription/translation and cost for *de novo* synthesis. Our approach, however, is not limited to optimizing sequence length and can easily be extended to handle other circuit parameters as well.

Other considerations for GDA besides the molecular identity of signals include matching the time scales at which components operate, matching the input/output ranges of components, and making sure that components are otherwise compatible with their host or platform for deployment. Previous research on the problem of genetic technology mapping has resulted in GDA tools that handle some or all of these design considerations, including the Genetic Engineering of Living Cells (GEC),[25] MatchMaker,[26] Cello,[27] and the Synthetic Biology Reusable Optimization Methodology (SBROME).[28] Of these considerations, iBioSim currently only accounts for the molecular identity of signals to ensure input/output compatibility and avoid cross-talk, but unlike other GDA tools, it uses a *cost function* to guide the DAG-based selection of genetic components. A cost function is an unambiguous, mathematical description that implicitly denotes the relative importance of different circuit parameters and explicitly relates them to the hypothetical quality of a genetic circuit design. Consequently, a cost function provides an extensible basis for computing and comparing solutions to the genetic technology mapping problem and enables the application of dynamic programming and branch-and-bound techniques to speed up its solution.

Lastly, the implementation of our approach in iBioSim also integrates genetic technology mapping with standards for genetic structure and function. These standards include the *Systems Biology Markup Language* (SBML),[29] an established standard for describing biochemical models that is supported by over 250 software tools, and the *Synthetic Biology Open Language* (SBOL),[30,31] a nascent standard for describing genetic components that has growing support among GDA tools.[32−35] The incorporation of standards into our approach provides for the possibility of greater interoperability between GDA tools in the future.

RESULTS AND DISCUSSION

Our approach to genetic technology mapping can be divided into three stages: graph construction, partitioning and decomposition, and matching and covering. Figure 1 presents an overview of this process as applied to automate the design of a genetic multiplexer,[21] a genetic circuit that can be used to share multiple incoming signals with another genetic circuit and thereby reduce resource requirements. During the first stage of our approach, regulatory DAGs are constructed from a model specification written in SBML and a library of SBML models annnotated with SBOL DNA components. Each DAG constructed from a library model is also assigned a cost that is a function of the combined length in base pairs of DNA components annotating the model. During the second stage, the specification DAG is partitioned into a set of rooted DAGs (also called trees) and is decomposed alongside the library DAGs to a logically equivalent canonical form. Lastly, during the third stage, the library DAGs are matched to each node in the specification DAG and lower bounds on the costs of solutions starting at each node are calculated via *dynamic programming*. Matches are then selected using a *branch-and-bound* approach to obtain a solution set of library DAGs and their corresponding SBOL-annotated SBML models that may be composed to satisfy or "cover" the original specification for a minimal cost.

Before describing the three stages of our approach in greater detail, let us first outline two major assumptions that are inherent in our approach as it currently stands. First, our approach assumes that the genetic components in a library have a sigmoidal relationship between steady-state input and output, such that it is possible to distinguish between high and low inputs/outputs to a component and assign a logical semantics to their relationship such as OR or AND. This is not an entirely unfounded assumption given previous research into the design and construction (even random construction[12]) of genetic components[10,11,17] and circuits[18,20,21] with steady-state phenotypic behaviors resembling digital logic. Second, our approach assumes that it is possible to connect genetic components on the basis of whether the molecular identities of their input and output signals are the same, but in doing so, our approach

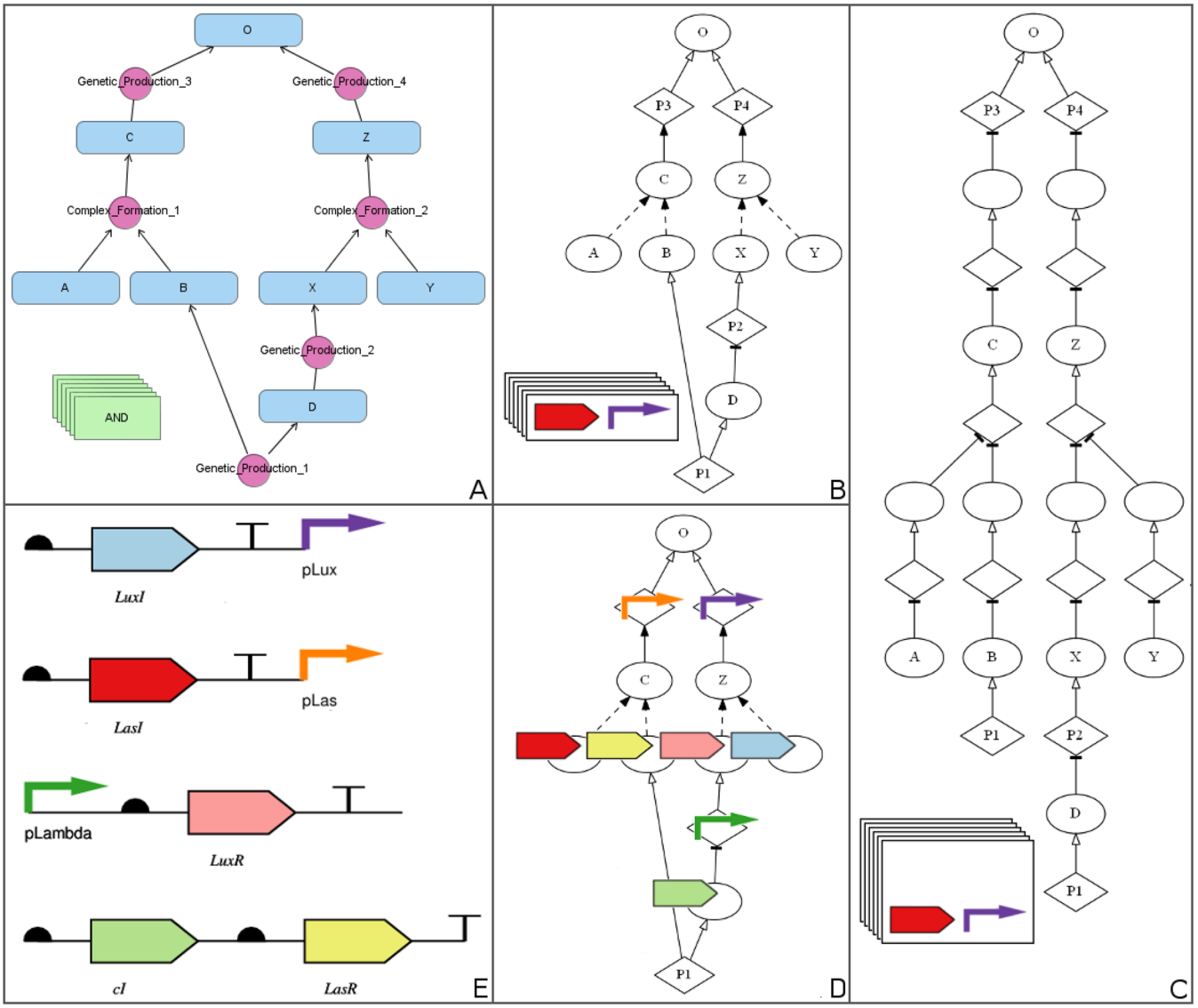


Figure 1. Overview of our genetic technology mapping process as applied to automate the design of a genetic multiplexer.[21] (A) iBioSim is used to build a SBML model of an abstract genetic multiplexer and a library of SBOL-annotated SBML models of genetic components that implement digital logic operations. The model of the genetic multiplexer contains species (blue ellipses) that are modifiers (lines) or reactants/products (arrows) of genetic production and complex formation reactions (purple circles). (B) A regulatory DAG specification and library of SBOL-labeled DAGs are constructed from the SBML model specification and library. See Figure 2 for a key describing the nodes and edges of a regulatory DAG. (C) The DAG specification is partitioned and decomposed alongside the library DAGs to faciliate matching and covering. (D) The DAG specification is matched and covered to obtain an optimal solution set of library DAGs. (E) Composite SBOL DNA components are inferred from the structure of the cause-and-effect relationships between DNA components as encoded by the solution.[5]

neglects other considerations for connecting components, such as determining whether the high and low output signals for one component constitute high and low input signals for its connected component, or whether two connected components are compatible in the context of their intended host. In the future, if our approach is extended to explicitly address these other considerations for component compatibility, then our approach can also leverage our cost function framework to guide the search for a solution that maximizes component compatibility, rather than just ensure that component compatibility is satisfied.

Graph Construction. The genetic circuit models for our specification and library are written in SBML and are composed of *species* and *reactions*. Within these models, species are interpreted as the small molecules or protein transcription factors that relay signals within genetic circuits. Reactions, on the other hand, are interpreted as the combined genetic processes of transcription and translation initiated at a promoter or as complex formations involving small molecules and/or transcription factors. In order to facilitate the interpretation of SBML in this genetic context, the species and reactions in our genetic circuit models are annotated with the appropriate *Systems Biology Ontology* (SBO) terms. In particular, reactions are annotated with the term *complex formation* or *genetic production*. Within genetic production reactions, species modifiers are annotated with the term *stimulator* or *inhibitor*. Furthermore, in order to identify the SBOL DNA components that are described by our genetic circuit models, the species and reactions in these models are annotated using a previously developed methodology for

dx.doi.org/10.1021/sb400135t | *ACS Synth. Biol.* 2014, 3, 543−555

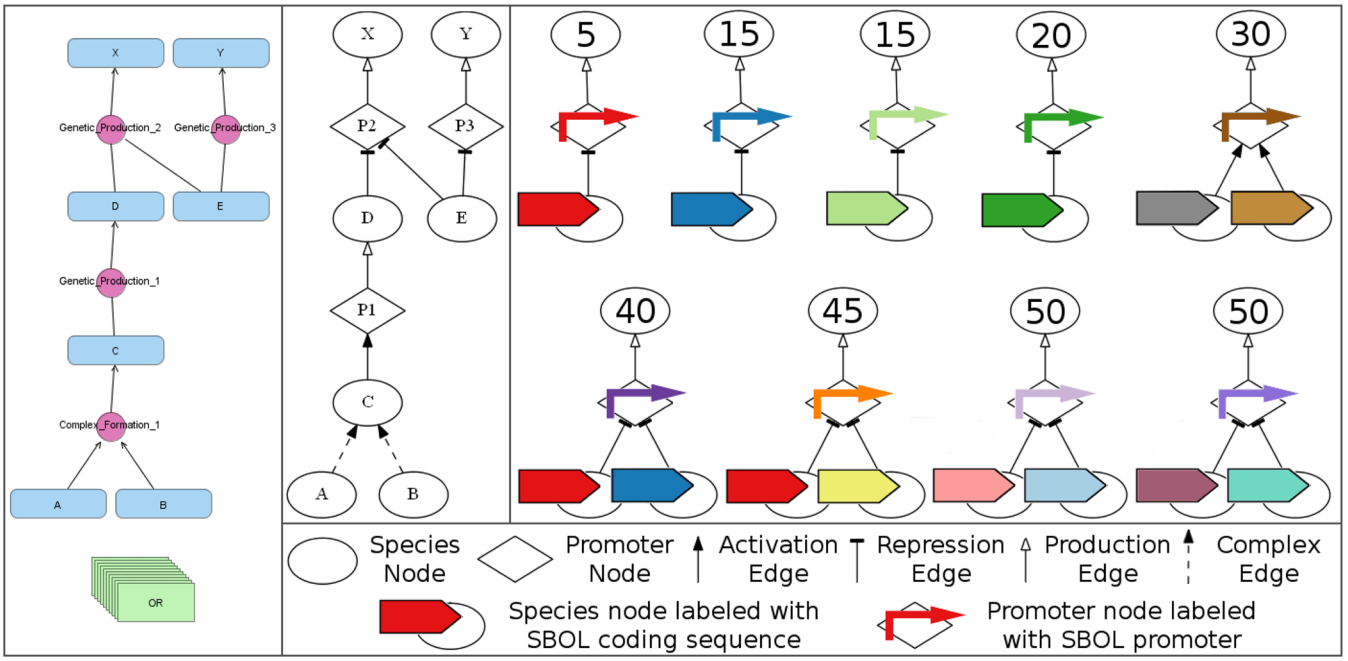


Figure 2. Construction of specification and library DAGs (middle and right) from the SBML models for a simple, abstract genetic circuit and a library of genetic logic gates (left). The colored symbols belong to the SBOL Visual extension[36] and represent SBOL DNA components labeling the nodes of our library DAGs. The numbers on the library DAGs indicate the cost associated with choosing that DAG to cover part of the specification DAG. Normally, each library DAG would have a cost equal to the combined length in base pairs of its associated DNA components, but this example uses smaller, simpler costs to make it easier to follow along later during matching and covering.

SBML-to-SBOL annotation.[5] Figure 2 displays the regulatory DAGs constructed from the SBML models for a simple, abstract genetic circuit and a library of genetic logic gates. A later section returns to the DAGs for this genetic circuit and library in the Matching and Covering section as a part of a basic example of our approach.

During the construction of a regulatory DAG from a SBML model, species nodes and promoter nodes are created for each species and genetic production reaction, respectively. Directed edges are created between species nodes and promoter nodes in accordance with the structure of the genetic production reactions. In particular, edges are drawn from the species nodes for the modifiers of genetic production reactions to the promoter nodes for these reactions, and edges are drawn from the promoter nodes for genetic production reactions to the species nodes for these reactions' products. The former are labeled as activation or repression edges, depending on whether the reaction modifer is annotated with the SBO term stimulator or inhibitor, while the latter are labeled as production edges. Directed edges are also created between species nodes in accordance with the structure of any complex formation reactions. Specifically, edges are drawn from the species nodes for reactants in complex formation reactions to the species nodes for the products in these reactions and are labeled as complex formation edges.

Once a regulatory DAG is constructed, all of its nodes are labeled with the *Uniform Resource Identifiers* (URIs)[37] for any SBOL DNA components annotating the nodes' corresponding SBML elements. This labeling is later used during matching and covering to determine whether the choice of a particular DAG would introduce cross-talk to a solution. This paper omits the labeling for ribosome binding sites and terminators to simplify the presentation of our approach. These DNA components are still included in the files for our test cases (see Supporting Information section). Next, each DAG is assigned a cost as a function over the parameters of its associated DNA components. This paper uses a very simple cost function over a parameter that is easily obtained: the combined length in base pairs of the DNA components associated with a DAG. At the very least, the length in base pairs of a circuit can be partially correlated with other design parameters of interest such as delay in transcription/translation and cost for *de novo* synthesis. In the future, however, we are keenly interested in extending our cost function with other relevant parameters such as the high and low levels of input and output signals for genetic components, the noise associated with these levels, and the degree of input/output compatibility when two components are connected.

Finally, to distinguish between regulatory DAGs and make them amenable to logical decomposition, let us assign one possible logical semantics to the genetic regulatory motifs present in these DAGs. Under this semantics, a promoter node with a single repressor is an inverter motif while a promoter with two repressors is a NOR motif. Inverter and NOR motifs produce output only when no inputs are present. Similarly, a promoter with one activator is a buffer motif while a promoter with two activators is an OR motif. Buffer and OR motifs produce output when one or more inputs are present. Lastly, a promoter with a complex activator is an AND motif while a promoter with a complex repressor is a NAND motif. An AND motif produces output only when both inputs are present while a NAND motif produces output so long as both inputs are not present.

Note that this particular logical interpretation enables the regulatory DAGs of our approach to capture the abstract behavior if not the exact mechanism of genetic circuits that implement combinational logic. For example, in Figure 1, part of the solution for the genetic multiplexer is an AND motif that includes

dx.doi.org/10.1021/sb400135t | *ACS Synth. Biol.* 2014, 3, 543−555

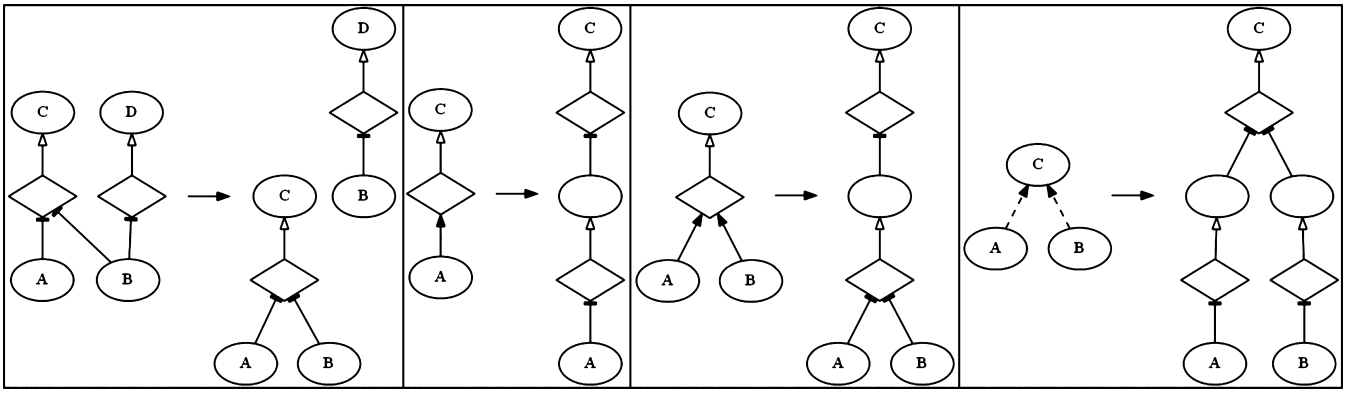


Figure 3. Examples of partitioning a DAG so that it has no nodes with more than one than one outgoing edge (left) and decomposing a buffer motif to two inverter motifs in series (center left), an OR motif to a NOR motif followed by an inverter motif (center right), and an AND motif to two inverter motifs in parallel followed by a NOR motif (right).

complex formation between LuxR and LuxI. Strictly speaking, it is the enzymatic product of LuxI, AHL, that forms a complex with LuxR and then activates the promoter pLux, but this additional mechanistic detail is not necessary to express the abstract logical relationship between the inputs and output of the underlying genetic circuit.

Partitioning and Decomposition. During partitioning (see Figure 3), the specification DAG is split at nodes with more than one outgoing edge into n rooted DAGs, where n is the total number of outgoing edges at these nodes. Partitioning enables the use of dynamic programming to calculate lower bounds on the costs of solutions starting from each node during matching. These lower bounds can then be used during covering to terminate the search for suboptimal solutions and thereby speed up the process of finding optimal solutions for each partition. The trade-off is that there is no guarantee of global optimality when the covered partitions are composed to form a final solution. As with many heuristics, the hope is that the nonglobally optimal solution is found more quickly and that it is still of fairly high quality.

Decomposition, on the other hand, increases the number of matches that can be made between the library DAGs and each node in the specification DAG, thereby increasing the number of possible solutions during covering and potentially improving the quality of our final solution. During decomposition (see Figure 3), the specification and library DAGs are transformed to a logically equivalent canonical form. In this canonical form, the DAGs only contain inverter and NOR motifs. Consequently, if the library contains at least as many inverter and NOR motifs as the decomposed specification DAG, then it is guaranteed that there is a complete solution that satisfies the specification DAG. This is particularly useful when a genetic regulatory motif is not shared by the library and specification DAGs prior to decomposition. For example, if the specification DAG contains a complex activator but the library DAGs do not, then this AND motif must be decomposed to logically equivalent motifs that are shared by the library DAGs in order to facilitate a complete solution. While there are other possible canonical forms based on different pairings of logical motifs (such as inverter and NAND motifs), our approach decomposes to inverter and NOR motifs based in part on their prevalence in online repositories such as the iGEM Registry of Standard Biological Parts.[38]

Matching and Covering. Our approach to matching the library and specification DAGs builds upon that taken in Keutzer's foundational technology mapping system for electronic circuits, DAGON.[6] Like DAGON, our tool also uses the Aho−Corasick algorithm[39] with little modification to match strings of characters encoding paths through our library DAGs with strings encoding paths through our specification DAG. Through string representation and the construction of a discrete finite automaton, the Aho−Corasick algorithm achieves a worst-case runtime that scales linearly with the size of the specification and independently of the size of the library.

During matching, the nodes of the specification DAG are traversed in a topological order. At each node, the Aho−Corasick algorithm is used to match the subtree rooted at that node to the library DAGs. The library DAGs that match at a node are then ordered such that the resulting sequence begins with the DAG that is part of the minimal cost solution starting at that node. Note that the minimal cost solution is determined without giving consideration to the possibility of genetic cross-talk due to shared transcription factors, as the cost of this solution is meant to serve as a lower bound on the cost of any solution starting at that node

For example, consider node O of the specification DAG in Figure 4. There are four library DAGs that match the subtree rooted at this node, namely the four inverter motifs at the bottom of the library and the OR motif (decomposed to a NOR motif followed by an inverter motif) in the upper right corner. Of the inverter motifs, the motif on the far left has the lowest cost of 5. If this motif is selected to cover node O, then it would also cover down to node C. Since the previously determined lower bound at node C is 50, the lower bound at node O resulting from choosing this inverter motif would equal 55. It is possible, however, to obtain a better lower bound than this by selecting the OR motif which has a cost of 30 and covers down to nodes M and N. These nodes both have lower bounds of 5, so the lower bound at node O resulting from choosing the OR motif would equal 40. Hence, the OR motif is positioned first in the list of matching library DAGs at node O and its corresponding lower bound is entered into the table in Figure 4.

Once matches and lower bounds have been determined, the process of selecting matches to form a valid, optimal cover begins at the root of the specification and proceeds in a depth-first fashion. Our covering algorithm, however, must take into account the possibility of genetic cross-talk resulting from shared transcription factors. Due to this possibility, a particular matching library DAG may be selected only once during a cover. Furthermore, each covering decision has implications

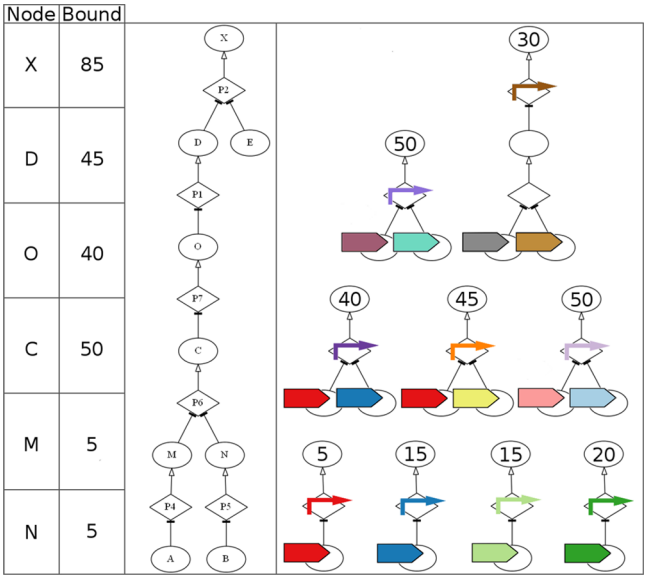


Figure 4. Partitioned, decomposed specification and library DAGs from Figure 2. Includes the results of calculating lower bounds on the costs of solutions starting at each node in the specification DAG and covering down to its leaves. For example, after iterating through each possible match to the subtree rooted at node O, it is determined that the minimal lower bound for a solution starting at this node that ignores cross-talk is obtained by selecting the OR motif in the upper right corner. This motif has a cost of 30 and covers down to nodes M and N, which each have a previously determined lower bound of 5 since matching proceeds in topological order. These lower bounds are added to the cost of the OR motif to obtain a lower bound of 40 at node O.

that may effect future covering decisions, which prevents an optimal solution from being found after a single traversal of the specification. To address this problem, our covering algorithm incorporates recursive backtracking and effectively becomes a branch-and-bound algorithm.

In our branch-and-bound algorithm, whenever a *dead-end* (i.e., a state in which selecting any available match introduces cross-talk) or solution is encountered, the traversal of the specification backtracks to the last node at which a match was selected. The next best match is then selected provided that the best-case estimate for the cost of a solution starting at this node does not exceed the cost of the best solution found so far. The best-case estimate is calculated by summing the total cost of the currently selected matches plus the lower bound at this node. While branch-and-bound guarantees that the optimal solution is eventually found, it also has a worst-case runtime that scales exponentially with the size of the input specification. In practice, the bounding aspect of branch-and-bound can improve the average runtime by pruning the search for suboptimal solutions. Figure 5 demonstrates the application of branch-and-bound covering to the specification and library DAGs from Figure 2.

Finally, at the terminus of matching and covering, the SBOL-annotated SBML models corresponding to the DAG matches that make up the best solution found are composed to form a composite SBML model annotated with composite SBOL. The composite SBML model encodes the behavior for the designed genetic circuit as a whole, while the composite SBOL annotating the SBML encodes structural information on the DNA components which make up the designed genetic circuit, including their DNA sequence and sequence annotations. This composite genetic circuit design that contains both structural and functional information on DNA can then serve to inform subsequent analysis, optimization, and verification steps, as well as, the eventual physical construction of the design.

Test Cases. To evaluate the performance of our branch-and-bound approach to genetic technology mapping, we have used it to map three genetic circuit specifications against four semirandomly generated libraries of increasing size, then compared the results with those for a naive exhaustive search that tries all possible solutions and a greedy variant of our approach that returns the first solution found. The greedy variant orders matches based on the lower bounds of costs for solutions that start with them, but quits when the first solution is found and does not use these lower bounds to inform the search to find better solutions. Note that the specifications mapped in this section are not meant to describe genetic circuits that have a particular real-world application. Rather, they are meant to serve as general examples that vary in size and incorporate a range of regulatory motifs for the purpose of benchmarking our approach.

As for the test libraries, each one is generated such that the DAGs constructed from it conform to a roughly uniform distribution of inverter, buffer, OR, NOR, AND, and NAND motifs. Furthermore, each model in a library is annotated with the URIs for DNA components having random, Gaussian-distributed lengths and is made to share a URI for a coding sequence DNA component with four percent of the other models in the library. Annotating the test libraries in this manner simulates the likely reuse of DNA components across library motifs and larger modules and the resulting cross-talk relationships between them.

Table 1 displays the results of mapping the specification model for a genetic *AND-OR-inverter* (AOI) against libraries containing 25, 50, 100, and 200 models. The AOI is shown in Figure 6 and has an effective size of 11 nodes that must be matched after its decomposition.

When mapping the genetic AOI, it appears that the solution times for our branch-and-bound approach scale very well relative to the size of the library, unlike the solution times for exhaustive search. As expected, the sizes of the solutions found with both approaches are the same, which indicates that branch-and-bound does find the best possible solution but in much less time. As for the greedy variant, its overall performance when mapping the AOI is nearly identical to that of branch-and-bound. Normally, it is expected that the greedy variant produces a lower quality solution in less time than branch-and-bound, but in this case, the best possible solution is found first, and it appears that branch-and-bound leverages this solution to terminate almost immediately after finding it. This rapid termination is likely facilitated by the fact that the theoretically optimal solution for mapping against each library is within 400 base pairs of the best possible solution. Lastly, the quality of the best possible solution increases with library size, likely owing to the greater absolute number and, therefore, diversity of motifs that are left to select from larger libraries after a percentage has been ruled out due to considerations of cross-talk during covering.

Next, Table 2 displays the results of mapping the specification model for a genetic NAND-NOR cascade against the same libraries used to generate results in Table 1. The NAND-NOR cascade can be seen in Figure 6 and has a decomposed size of 16 nodes.

When mapping the genetic NAND-NOR cascade, it appears that the solution times for branch-and-bound increase by one

dx.doi.org/10.1021/sb400135t | *ACS Synth. Biol.* 2014, 3, 543−555

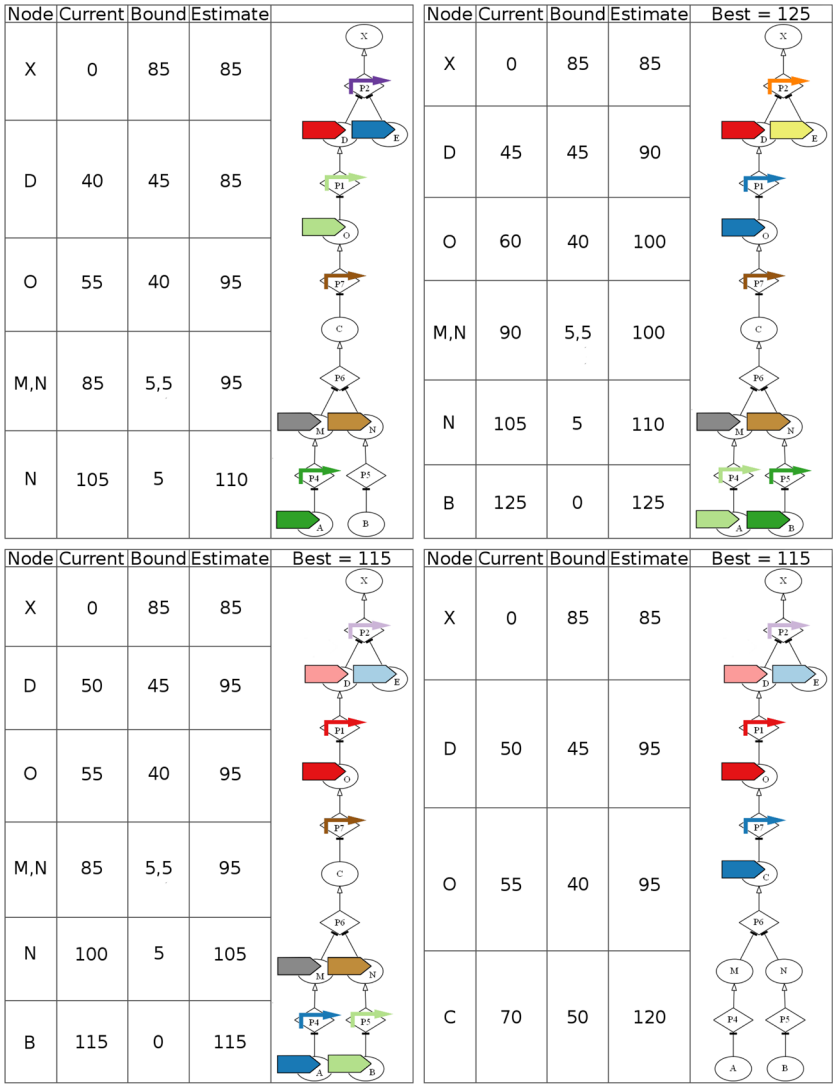


Figure 5. Covering with the specification and library DAGs from Figure 2. During the first traversal (upper left), a dead-end is encountered when selecting any of the remaining inverter motifs from the library to cover the rest of the specification would result in unintended cross-talk. After backtracking and encountering several other dead-ends, the next best match at node X is selected and the resulting traversal yields a complete solution with a cost of 125 (upper right). Backtracking to look for better solutions yields a solution with a cost of 115 when the next best match at node X is selected, which becomes the current best solution (lower left). Finally, when backtracking to node O and selecting the next best match, the search for better solutions is halted because the best-case estimate for solutions starting from the next node C is 120, which is greater than the cost for the best solution found so far. The best-case estimate for a partial solution is calculated by summing its current cost and the lower bound for the remaining uncovered portion of the specification.

Table 1. Solution Times in Seconds (Left) and Sizes in Base Pairs (Right) for the Genetic AOI (Motif Size of 11 Nodes Following Decomposition) Using Three Different Algorithms and Four Libraries of Different Sizes[a]

	library size				library size			
algorithm	25	50	100	200	25	50	100	200
exhaustive search	0.2	1	60	>1 h	3662	2871	2946	n/a
branch-and-bound	0.01	0.01	0.02	0.02	3662	2871	2946	2913
greedy variant	0.01	0.01	0.02	0.02	3662	2871	2946	2913

[a]Exhaustive search is run for up to 1 h before timing out and failing to find a solution.

to 2 orders of magnitude compared with the solution times for the AOI, but it also appears that these times still scale fairly well as library size increases. Exhaustive search, on the other hand, times out when applied against all but the smallest of libraries. As for the greedy variant, it is now consistently faster than branch-and-bound, but produces lower quality solutions. In the case of the library with size 50, however, the solution produced by the greedy variant is only two base pairs worse than that produced by branch-and-bound, suggesting that there are still conditions under which the greedy variant can produce a nearly optimal solution for larger specifications. Also, once again the quality of the best possible solution increases with library size.

Lastly, Table 3 displays the results of mapping the specification model for a genetic *OR-AND-invert* (OAI) cascade

dx.doi.org/10.1021/sb400135t | *ACS Synth. Biol.* 2014, 3, 543−555

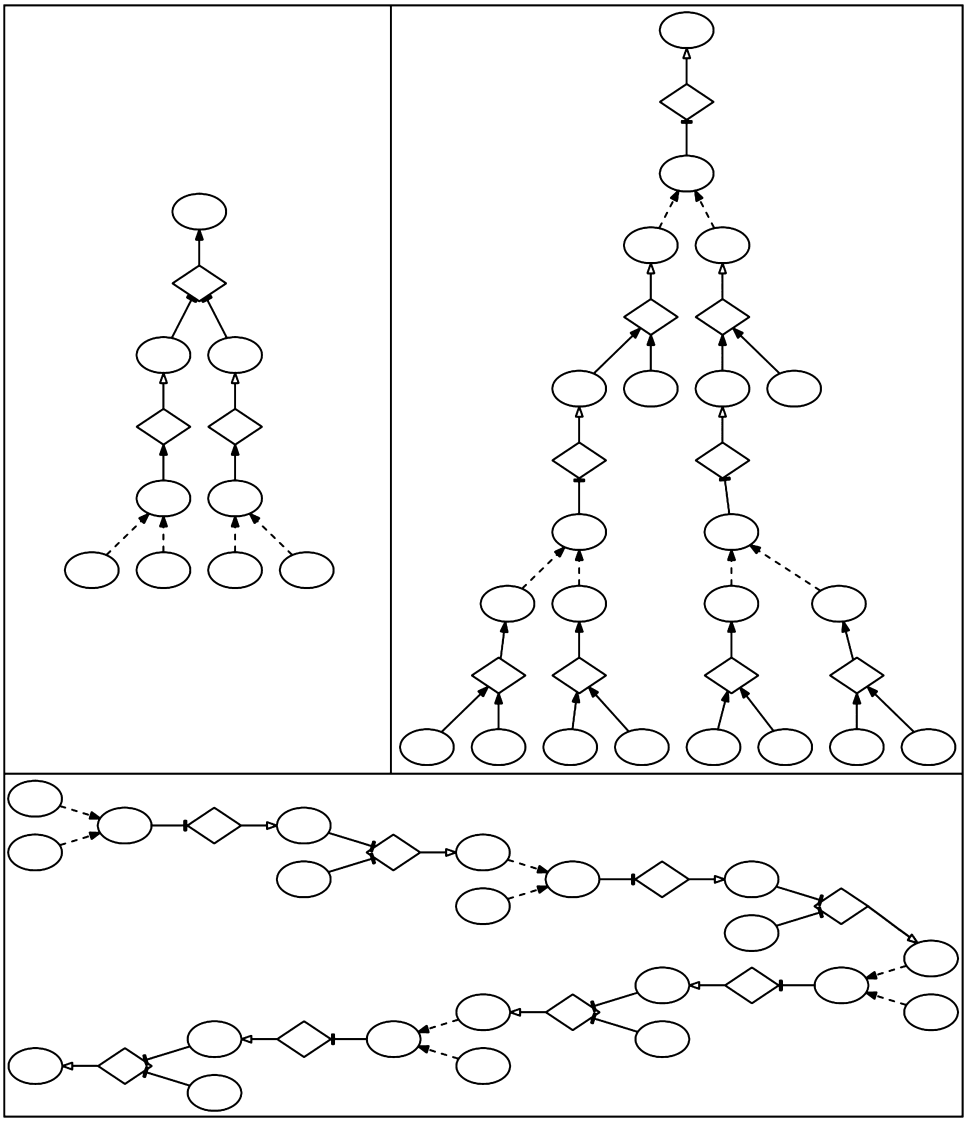


Figure 6. Test case specification DAGs. These include regulatory DAGs for a genetic AOI (upper left), a genetic NAND-NOR cascade (bottom), and a genetic OAI cascade (upper right). The sizes of these DAGs following their decomposition are 11, 16, and 24 nodes, respectively.

Table 2. Solution Times in Seconds (Left) and Sizes in Base Pairs (Right) for the Genetic NAND-NOR Cascade (Motif Size of 16 Nodes Following Decomposition) Using Three Different Algorithms and Four Libraries of Different Sizes[a]

	library size				library size			
algorithm	25	50	100	200	25	50	100	200
exhaustive search	1	>1 h	>1 h	>1 h	11178	n/a	n/a	n/a
branch-and-bound	0.2	1	0.7	1	11178	10931	10592	8270
greedy variant	0.02	0.03	0.04	0.06	13219	10933	11107	8482

[a]Exhaustive search is run for up to 1 h before timing out and failing to find a solution.

against the same libraries as before. The OAI cascade is depicted in Figure 6 and has a size of 24 nodes following decomposition.

When mapping the genetic OAI cascade, it appears once again that the solution times for branch-and-bound increase by one to 2 orders of magnitude when compared with the previous smaller specification, yet scale tolerably for larger library sizes. Interestingly, the longest solution time occurs when mapping against a library of size 50 and is an order of magnitude larger than when mapping against the larger libraries. One likely cause for this result is an increase in the cost gap between the first and best solutions found for this size of library, which can make bounding less effective at halting the search for suboptimal solutions and pruning the decision tree. The cause of this cost gap may be that the motifs that make up low-cost solutions close to the theoretically optimal solution (i.e., motifs that appear early in the sequence of matches at each node) are motifs that interfere with each other, a condition made more likely by library sizes that are small enough to increase the probability that any two motifs interfere with each other, but not so small that many solutions are ruled out due to cross-talk and pruning of the decision tree is achieved regardless of bounding.

dx.doi.org/10.1021/sb400135t | ACS Synth. Biol. 2014, 3, 543−555

Table 3. Solution Times in Seconds (Left) and Sizes in Base Pairs (Right) for the Genetic OAI Cascade (Motif Size of 24 Nodes Following Decomposition) Using Three Different Algorithms and Four Libraries of Different Sizes[a]

	library size				library size			
algorithm	25	50	100	200	25	50	100	200
exhaustive search	>1 h	>1 h	>1 h	>1 h	n/a	n/a	n/a	n/a
branch-and-bound	5	100	10	40	13836	12518	11377	9335
greedy variant	2	0.03	0.06	0.02	14774	15357	11603	9592

[a]Exhaustive search was run for up to 1 h before timing out and failing to find a solution.

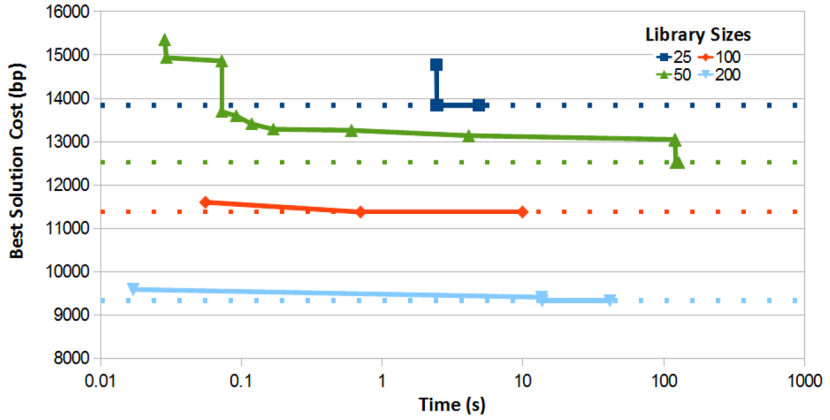


Figure 7. Best solution cost in base pairs versus time in seconds when applying our branch-and-bound approach to the genetic OAI cascade and four libraries of different sizes. Each dotted line represents the cost for the best possible solution when mapping against a particular library.

Figure 7 visualizes the dynamics of our branch-and-bound approach as applied to the solution of the OAI cascade. For library sizes other than 50, branch-and-bound converges to the best possible solution cost within two to three solutions and the majority of its run time is spent determining that there are no better solutions. For the size 50 library, on the other hand, 11 solutions are found before the best possible solution. While finding the first nine solutions takes less than 4 s of run time and represents a 15% gain in solution quality, the last three solutions require an additional 1.5 min to determine and achieve a less than 5% gain in solution quality.

These results, taken together with preliminary observations that default, exact branch-and-bound begins to time out when mapping specification models larger than the OAI cascade, suggest that an iterative greedy variant should be used to find near-optimal solutions much more quickly for larger specification models. While a fixed number of iterations may suffice for large specifications and small or large libraries, additional heuristics may be required when mapping large specifications against midsized libraries. These heuristics could include a dynamic cap on the number of iterations that takes effect when the average increase in solution quality bottoms out over time, or perhaps a less conservative approach to bounding that only explores a potential solution if its best-case estimate cost is less than the best solution cost so far plus an extra factor that depends on previous changes in solution quality.

Discussion. We have developed and implemented an algorithm for DAG-based genetic technology mapping in our GDA tool, iBioSim. It is among the first genetic technology mapping algorithms to adapt techniques from electronic circuit design, in particular the use of a cost function to guide the search for an optimal solution. It also part of an overall approach to genetic technology mapping that perhaps makes the greatest use of standards such as SBML[29] and SBOL[30,31] to represent design specifications and component libraries.

Previous heuristic-based approaches to genetic technology mapping include MatchMaker[26] and SBROME.[28] MatchMaker seeks to map a *abstract genetic regulatory network* (AGRN) against a feature database that contains information on qualitative regulatory relationships such as repression and activation, which is analogous to mapping a regulatory DAG specification against a DAG library in our approach, but Matchmaker works at a slightly finer grain of modularity in that individual features such as promoters and coding sequences are selected by themselves rather than as part of a larger motif. Representing the library as a graph of regulatory interactions between features in this way has the benefit of obviating the need to construct motifs for every possible combination of features, but it can also increase the number of feature combinations that must be searched to obtain the best possible solution. To address this challenge, MatchMaker takes a minimized AGRN as input and uses a timed heuristic search that selects features using knowledge of their regulatory interactions with the rest of the database to find solutions quickly. Solutions are obtained, however, without a cost function to bias toward finding good solutions first. Rather, preminimization of the AGRN specification is used to find smaller solutions in general and those found are ranked with regards to the degree that their features are compatible in terms of input/output signal ranges.

SBROME, on the other hand, takes a more similar approach to our own with regards to library organization in that parts belong to modules that can be used to cover a circuit in fewer matches. For the purposes of matching and covering, SBROME uses a greedy approach that prefers larger, experimentally characterized modules and finds a fixed number of solutions while using look-ahead information to avoid exhaustively searching the exponentially large solution space, thereby finding more promising solutions quickly. Again, however, solutions are not obtained with a more general cost function to evaluate the

dx.doi.org/10.1021/sb400135t | *ACS Synth. Biol.* 2014, 3, 543−555

quality of a solution or take advantage of information on what the theoretically optimal solution would be barring considerations of cross-talk. Both SBROME and Matchmaker are effective tools for genetic technology mapping, but their respective approaches can still potentially benefit from being augmented with the mathematical framework of a cost function.

In light of the results presented in this paper and the previous comparisons to other heuristic-based approaches, there is a case to be made for DAG-based genetic technology mapping that uses a cost function to relate component and circuit parameters to solution quality, rank matches between the specification and library based on their inclusion in a theoretically optimal solution, bias toward the discovery of near-optimal solutions first, and prune the search for suboptimal solutions. The average run time for an exact branch-and-bound approach scales very well with increasing library size for specification sizes less than 25 nodes, while preliminary results suggest that a greedy branch-and-bound approach applied for a fixed number of iterations can find near-optimal solutions to larger specifications, especially in conjunction with larger, more diverse libraries that help to mitigate the effects of cross-talk. Medium library sizes (relative to a given specification and degree of cross-talk between library components), on the other hand, may require additional heuristics when mapped against larger specifications to more quickly determine that a near-optimal solution has been found.

Furthermore, while our approach currently only handles genetic circuit specifications that are directed and acyclic in nature (i.e., no feedback), in the future we intend to handle cyclic specifications by extending our partitioning step with existing algorithms for efficiently cutting cyclic directed graphs into DAGs. This is a very important step for GDA as it would enable the automated design of genetic circuits to move beyond combinational logic and into the realm of *asynchronous state machines*. Furthermore, it should also prove to be interesting to try different cost functions that make use of information on DNA components other than length in base pairs alone and explore how the formalism of a cost function can make it quantitatively clear how important is one desired trait for a design when compared to another. The length metric is appealing because it is easily determined and is at least partly related to delay in transcription/translation and the fiscal cost associated with physical construction of a genetic circuit. In the future, however, we would also like calculate cost based on metrics that are more directly related to a genetic circuit's correct function, such as the degree to which the high and low levels of input and output signals for two connected components are compatible when noise is taken into account.

Lastly, while our approach currently uses a stylized form of SBML annotated with SBO terms in order to encode the regulatory interactions between SBOL DNA components from which our regulatory DAGs are constructed, in the future we intend to represent these regulatory interactions natively in SBOL and instead use SBML primarily as a means of mathematically modeling these interactions for simulation and analysis following technology mapping. Drawing such boundaries between different formats for representation and mapping one's own data to them are major challenges of working with standards, but they are necessary to avoid duplication of effort and enable complementary exchange of information. Moving qualitative descriptions of regulation from stylized SBML into the SBOL standard enables different GDA tools to agree on the intended regulatory interactions between the components of a genetic circuit design, but ultimately choose their own modeling standard to mathematically describe these interactions. We would also like to use SBOL to represent nongenetic components of design such as small molecules or light, thereby increasing the structural range by which our approach is able to represent genetic circuit designs in a standardized manner. Facilitating these eventual changes to our approach is a current development effort within the SBOL community to introduce regulatory interactions and generalized components outside of DNA to the SBOL standard.

iBioSim is available for download at http://www.async.ece.utah.edu/iBioSim/.

METHODS

During the final stage of genetic technology mapping, Algorithm 1 and Algorithm 2 handle the processes of matching and covering, respectively. These algorithms differ from their typical EDA implementations in the following ways: Algorithm 1,

Algorithm 1: Matching

Input: Specification DAG $G = \langle V, E \rangle$ where V is a set of nodes and E is a set of edges, DFA D constructed from DAG library L using the Aho-Corasick string matching algorithm

Output: Specification DAG G where each node in V contains a sequence of library DAGs L_i that match the subtree rooted at that node and a sequence of lower bounds on the costs of solutions beginning with each match

```
/* || is a composition operator that joins two sequences     */
/* ⟨⟩ are opening/closing sequence brackets                  */
C ← leaves(G)
while |C| > 0 do
    for L_i ∈ DETERMINE-MATCHES(D, paths(G, C_0)) do
        matches(C_0) ← matches(C_0)||⟨L_i⟩
        b ← CALCULATE-COST(L_i)
        for v ∈ WALK-PATHS(paths(L_i), G, C_0) do
            b ← b + lowerBounds(v)_0
        lowerBounds(C_0) ← lowerBounds(C_0)||⟨b⟩
    QUICKSORT-MATCHES(matches(C_0), lowerBounds(C_0))
    N ← successors(C_0)
    C ← sub(C, 1, |C| − 1)
    if |C| = 0 ∧ |N| > 0 then
        C ← C||N
```

in addition to determining the matches at each node and calculating lower bounds on the solutions to which they belong, also sorts the matches at each node in accordance with their lower bounds. Sorting in this manner is necessary to enable effective bounding while covering. Once one match at a node has been ruled out as belonging to a suboptimal solution, all other matches that come after it in sequence can be ruled out as well and backtracking can be initiated. Algorithm 2, rather than traversing the specification a single time to determine the optimal solution, must instead potentially traverse the specification many times to explore every possible solution due to cross-talk considerations. Hence, Algorithm 2 is extended with the ability to backtrack whenever a complete solution is found, cross-talk prevents a solution from being found, or the best-case estimate of the final solution cost exceeds the cost of the best solution found so far.

Matching. Algorithm 1 handles the process of matching library DAGs to the subtrees rooted at each node in the specification DAG and calculating lower bounds on the costs of solutions that start with each match. During this process, nodes are matched and bound in a topological order, that is, starting from the leaf nodes of the specification DAG that have no incoming edges and ending at the root. In this way, previously calculated lower bounds can be reused to calculate later lower bounds, a dynamic programming approach that enables bounding to be performed with a worst-case runtime that

Algorithm 2: Covering

Input: Specification DAG $G = \langle V, E \rangle$
Output: Sequence solutions B where the best solution is located at index 0 and each solution is a sequence of library DAGs L_i

```
S ← ⟨⟩
P ← ⟨⟩
C ← ⟨root(G)⟩
B ← ⟨⟩
repeat
    if |matches(C_0)| = 0 then
        C ← sub(C, 1, |C| − 1)
        if |C| = 0 ∧ |P| > 0 then
            if |B| = 0 ∨ cost(S) < cost(B_0) then
                B ← ⟨S⟩||B
            S ← sub(S, 0, |S| − 1)
            resetBranch(P_0)
            P ← sub(P, 1, |P| − 1)
    else if branch(C_0) then
        if ¬CROSS-TALK(currentMatch(C_0), S) then
            if SOLUTION-IN-BOUND(S, C, B_0) then
                uncovered(C_0) ← sub(C, 1, |C| − 1)
                N ← WALK-PATHS(paths(L_i), G, C_0)
                S ← S||⟨currentMatch(C_0)⟩
                P ← ⟨C_0⟩||P
                C ← sub(C, 1, |C| − 1)
                C ← N||C
            else
                resetBranch(C_0)
                C ← ⟨⟩
    else
        C ← ⟨⟩
    if |C| = 0 ∧ |P| > 0 then
        S ← sub(S, 0, |S| − 1)
        C ← C||uncovered(P_0)
        C ← ⟨P_0⟩||C
        P ← sub(P, 1, |P| − 1)
until |C| = 0
return B
```

scales as $|G||L|$, where $|G|$ is the size of the specification and $|L|$ is the size of the library. As for matching, a DFA constructed via the Aho-Corasick string matching algorithm is used to match the subtrees rooted at each node in the specification simultaneously against all library DAGs, thereby achieving a worst-case runtime that scales independently of the size of the library.

Functions called by Algorithm 1 include DETERMINE-MATCHES, CALCULATE-COST, WALK-PATHS, and QUICKSORT-MATCHES. The DETERMINE-MATCHES function takes as input the Aho−Corasick DFA D and a sequence of strings representing paths through the specification DAG G that begin at the current node C_0 and proceed back to the leaves of G. This string representation of the subtree rooted at C_0 is provided as input to D, which outputs the sequence of library DAGs matching the subtree.

The CALCULATE-COST function, as its name suggests, calculates the cost for a matching library DAG L_i in accordance with the cost function, which in this paper sums the nucleotide counts for DNA components labeling the DAG. The WALK-PATHS function then uses a sequence of strings representing paths through L_i to walk back through G from C_0 and identify the nodes in G that are at the boundary of the subtree covered by L_i. Together, CALCULATE-COST and WALK-PATHS work to determine the lower bounds on the costs of solutions that start with each match and cover all the way back to the leaves of G.

Finally, the function QUICKSORT-MATCHES uses the Quicksort algorithm to sort the sequence of matching library DAGs at each node in accordance with the corresponding sequence of lower bounds on the costs of solutions starting with these matches. Note that ordering the matches at each node in this manner makes the overall worst-case runtime for matching scale as $|G||L|\log(|L|)$. The reason ordering is performed is to enhance the efficiency of covering, which has a worse worst-case runtime that scales as $|L|^{|G|}$. By biasing toward the discovery of better solutions earlier, ordering is expected to increase the efficacy with which these solutions' costs are used to bound the search for suboptimal solutions.

Primitive routines called by Algorithm 1 include the graph routines *leaves* and *paths*, the node routines *predecessors*, *matches*, and *lowerBounds*, and the sequence routine *sub*. The graph routine *leaves* returns the leaf nodes for the given graph, that is, the nodes with no incoming edges, while *paths* returns strings encoding each possible path from the given node (or root node if none is given) back to the leaves of the given graph. These strings consist of alternating letters and numbers that represent the types and cardinality of the nodes and edges.

Next, the node routine *successors* returns all nodes with incoming edges that point from the given node. The routines *matches* and *lowerBounds*, on the other hand, return sequences of library DAGs found to match the given node and sequences of the lower bounds on costs of solutions beginning with these matches, respectively. Lastly, *sub* returns a subsequence of the given sequence that starts at the indicated index and has the indicated length. This routine is used during Algorithm 1 to effectively delete the first node in a sequence by replacing the sequence with a subsequence that starts at index one and has a length equal to that of the sequence minus one.

Covering. Algorithm 2 handles the process selecting matches to form the best solution for the entire specification, starting with matches at the root node of the specification and covering back through the specification to its leaves in a depth-first fashion. Despite the use of bounding to terminate the search for suboptimal solutions, the worst-case runtime for covering still scales as $|L|^{|G|}$, since in the absolute worst-case there is no solution at all due to the constraints of genetic cross-talk and every possible combination of library DAGs that nearly covers the specification is tested without the aid of bounding. It is also possible that the best solution exists but has a cost that differs dramatically from the theoretically optimal cost for a solution that does not account for cross-talk, in which case bounding by comparing the two does not discriminate suboptimal solutions until many choices have been made and fewer branches are pruned from the decision tree as a result.

Functions called by Algorithm 2 include CROSS-TALK, SOLUTION-IN-BOUND, and WALK-PATHS. The first two functions serve to verify whether the current matching library DAG L_i under consideration can be part of a valid, potentially optimal solution. Specifically, the CROSS-TALK function checks whether or not L_i shares any signaling DNA components such as promoters or coding sequences with the current solution S as a whole, while the SOLUTION-IN-BOUND function determines whether the sum of the cost of S and the lower bounds on the currently selected matches at nodes C is less than the cost of the best solution B_0 found so far. The WALK-PATHS function, on the other hand, works as previously described but is used in Algorithm 2 to determine which nodes should be considered next after a match at the current node C_0 has been selected to be part of S.

Previously undescribed primitive routines called by Algorithm 2 include the graph routine *root*, the solution routine *cost*, and the node routines *currentMatch*, *branch*, *resetBranch*, and *uncovered*. The graph routine *root* returns the root node of the given graph, that is, the node with no outgoing edges. Next, the

dx.doi.org/10.1021/sb400135t | ACS Synth. Biol. 2014, 3, 543−555

solution routine *cost* returns the total cost for the given solution. Finally, the first three node routines keep track of and modify which match is currently selected at the given node. In particular, *currentMatch* returns the currently selected match, while *branch* increments the index that indicates which match is selected and *resetBranch* sets this index to its starting value. The *uncovered* routine, on the other hand, keeps track of which nodes were in C alongside the given node when it was the current node under consideration (C_0). This routine enables Algorithm 2 to recover the state of C when backtracking to a previously considered node.

■ ASSOCIATED CONTENT

Supporting Information

Test case files. This material is available free of charge *via* the Internet at http://pubs.acs.org.

■ AUTHOR INFORMATION

Corresponding Authors

*E-mail: n.roehner@utah.edu.

*E-mail: myers@ece.utah.edu.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

We thank Andrew Fisher and Zhen Zhang for helpful discussions related to the material presented in this paper. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0916042 and CCF-1218095. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

■ REFERENCES

(1) Endy, D. (2005) Foundations for engineering biology. *Nature 438*, 449−453.

(2) Arkin, A. (2008) Setting the standard in synthetic biology. *Nat. Biotechnol. 26*, 771−774.

(3) Cooling, M. T., Rouilly, V., Misirli, G., Lawson, J., Yu, T., Hallinan, J., and Wipat, A. (2010) Standard Virtual Biological Parts: A repository of modular modeling components for synthetic biology. *Bioinformatics 26*, 925−931.

(4) Misirli, G., Halliman, J. S., Yu, T., Lawson, J. R., Wimalaratne, S. M., Cooling, M. T., and Wipat, A. (2011) Model annotation for synthetic biology: Automating model to nucleotide sequence conversion. *Bioinformatics 27*, 973−979.

(5) Roehner, N., and Myers, C. J. (2013) A methodology to annotate systems biology markup language models with the synthetic biology open language. *ACS Synth. Biol.*, DOI: 10.1021/sb400066m.

(6) Keutzer, K. (1987) DAGON: Technology binding and local optimization by DAG matching. *Proceedings of the 24th ACM/IEEE Design Automation Conference*, 341−347.

(7) Myers, C. J., Barker, N., Jones, K., Kuwahara, H., Madsen, C., and Nguyen, N.-P. D. (2009) iBioSim: A tool for the analysis and design of genetic circuits. *Bioinformatics 25*, 2848−2849.

(8) Madsen, C., Myers, C., Patterson, T., Roehner, N., Stevens, J., and Winstead, C. (2012) Design and test of genetic circuits using iBioSim. *IEEE Design Test 29*, 32−39.

(9) Buchler, N. E., Gerland, U., and Hwa, T. (2003) On schemes of combinatorial transcription logic. *Proc. Natl. Acad. Sci. U.S.A. 100*, 5136−5141.

(10) Wang, B., Kitney, R. I., Joly, N., and Buck, M. (2013) Engineering modular and orthogonal genetic logic gates for robust digital-like synthetic biology. *Nat. Commun. 2*, 508.

(11) Tamsir, A., Tabor, J. J., and Voigt, C. A. (2011) Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'. *Nature 469*, 212−215.

(12) Guet, C. C., Elowitz, M. B., Hsing, W., and Leibler, S. (2002) Combinatorial synthesis of genetic networks. *Science 296*, 1466−1470.

(13) Chaouiya, C. (2008) Qualitative modeling of biological regulatory networks combining a logical multi-valued formalism and Petri nets. *9th International Workshop on Discrete Event Systems (WODES)*, 263−268.

(14) Batt, G.; Ropers, D.; Jong, H.; Geiselmann, J.; Page, M.; Schneider, D. (2005) Qualitative analysis and verification of hybrid models of genetic regulatory networks: nutritional stress response in Escherichia coli. In *Hybrid Systems: Computation and Control* (Morari, M. and Thiele, L., Ed.), pp 134−150, Springer, Berlin Heidelberg.

(15) Fromentin, J., Eveillard, D., and Roux, O. (2010) Hybrid modeling of biological networks: Mixing temporal and qualitative biological properties. *BMC Syst. Biol. 4*, 79.

(16) Clewley, R. (2012) Hybrid models and biological model reduction with PyDSTool. *PLoS Comput. Biol.*, DOI: 10.1038/msb4100173.

(17) Anderson, J. C., Voigt, C. A., and Arkin, A. P. (2007) Environmental signal integration by a modular AND gate. *Mol. Syst. Biol.*, DOI: 10.1038/msb4100173.

(18) Tabor, J. J., Salis, H. M., Simpson, Z. B., Chevalier, A. A., Levskaya, A., Marcotte, E. M., Voigt, C. A., and Ellington, A. (2009) A synthetic genetic edge detection program. *Cell 137*, 1272−81.

(19) Salis, H., Tamsir, A., and Voigt, C. (2009) Engineering bacterial signals and sensors. *Contrib. Microbiol. 16*, 194−225.

(20) Moon, T. S., Clarke, E. J., Groban, E. S., Tamsir, A., Clark, R. M., Eames, M., Kortemme, T., and Voigt, C. A. (2011) Construction of a genetic multiplexer to toggle between chemosensory pathways in *Escherichia coli*. *J. Mol. Biol. 406*, 215−227.

(21) Pasotti, L., Quattrocelli, M., Galli, D., Angelis, M. G. D., and Magni, P. (2011) Multiplexing and demultiplexing logic functions for computing signal processing tasks in synthetic biology. *Biotechnol. J. 6*, 784−795.

(22) Daniel, R., Rubens, J. R., Sarpeshkar, R., and Lu, T. K. (2013) Synthetic analog computation in living cells. *Nature 497:7451*, 619−623.

(23) Ro, D.-K., Paradise, E. M., Ouellet, M., Fisher, K. J., Newman, K. L., Ndungu, J. M., Ho, K. A., Eachus, R. A., Ham, T. S., Kirby, J., Chang, M. C. Y., Withers, S. T., Shiba, Y., Sarpong, R., and Keasling, J. D. (2006) Production of the antimalarial drug percursor artemisinic acid in engineered yeast. *Nature 440*, 940−943.

(24) Atsumi, S., and Liao, J. C. (2008) Metabolic engineering for advanced biofuels production from *Escherichia coli*. *Curr. Opin. Biotechnol. 19*, 414−419.

(25) Pedersen, M., and Phillips, A. (2009) Towards programming languages for genetic engineering of living cells. *J. R. Soc. Interface 6*, S437−S450.

(26) Yaman, F., Bhatia, S., Adler, A., Densmore, D., and Beal, J. (2012) Automated selection of synthetic biology parts for genetic regulatory networks. *ACS Synth. Biol. 1*, 332−344.

(27) Yordanov, B., Appleton, E., Ganguly, R., Gol, E. A., Carr, S. B., Bhatia, S., Haddock, T., Belta, C., and Densmore, D. (2012) Experimentally driven verification of synthetic biological circuits. *Design, Automation, and Test in Europe Conference and Exhibition (DATE)*, 236−241.

(28) Huynh, L., Tsoukalas, A., Koppe, M., and Tagkopoulos, I. (2013) SBROME: A scalable optimization and module matching framework for automated biosystems design. *ACS Synth. Biol. 2*, 1073−1089.

(29) Hucka, M., et al. (2003) The Systems Biology Markup Language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics 19*, 524−531.

(30) Galdzicki, M. et al. (2012) *Synthetic Biology Open Language (SBOL)* Version 1.1.0. BBF RFC 87; http://hdl.handle.net/1721.1/73909.

(31) Galdzicki, M. (2013) SBOL: A community standard for communicating designs in synthetic biology. *Figshare*, DOI: 10.6084/m9.figshare.762451.
(32) Chandran, D., Bergmann, F. T., and Sauro, H. M. (2009) TinkerCell: Modular CAD tool for synthetic biology. *J. Biol. Eng. 3*, 19.
(33) Czar, M. J., Cai, Y. Z., and Peccoud, J. (2009) Writing DNA with GenoCAD™. *Nucleic Acids Res. 37*, W40−W47.
(34) Densmore, D.; Van Devender, A.; Johnson, M.; Sritanyaratana, N. (2009) A platform-based design environment for synthetic biological systems. *The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations*, New York, pp 24−29.
(35) Chen, J., Densmore, D., Ham, T. S., Keasling, J. D., and Hillson, N. J. (2012) DeviceEditor visual biological CAD canvas. *J. Biol. Eng. 6*, 1.
(36) Quinn, J.; Beal, J.; Bhatia, S.; Cai, P.; Chen, J.; Clancy, K.; Hillson, N. J.; Galdzicki, M.; Maheshwari, A.; Umesh, P.; Pocock, M.; Rodriguez, C.; Stan, G.-B.; Endy, D. (2013) *Synthetic Biology Open Language Visual (SBOL Visual),* Version 1.0.0. BBF RFC 93; http://hdl.handle.net/1721.1/78249.
(37) Berners-Lee, T.; Fielding, R.; Masinter, L. (2005) *Uniform Resource Identifier (URI): Generic Syntax*, IETF RFC 3986; The Internet Society; http://tools.ietf.org/html/rfc3986.
(38) iGEM Registry. (2003); http://parts.igem.org (accessed 08/24/13).
(39) Aho, A. V., and Corasick, M. J. (1975) Efficient string matching: an aid to bibliographic search. *Commun. ACM 18*, 333−340.