ACS SyntheticBiology

Research Article

pubs.acs.org/synthbio

Double Dutch: A Tool for Designing Combinatorial Libraries of Biological Systems

Nicholas Roehner,[*,†] Eric M. Young,[‡] Christopher A. Voigt,[‡] D. Benjamin Gordon,[‡] and Douglas Densmore[†]

[†]Department of Electrical and Computer Engineering, Boston University, Boston, Massachusetts 02215, United States

[‡]Department of Biological Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, United States

Supporting Information

ABSTRACT: Recently, semirational approaches that rely on combinatorial assembly of characterized DNA components have been used to engineer biosynthetic pathways. In practice, however, it is not practical to assemble and test millions of pathway variants in order to elucidate how different DNA components affect the behavior of a pathway. To address this challenge, we apply a rigorous mathematical approach known as design of experiments (DOE) that can be used to construct empirical models of system behavior without testing all variants. To support this approach, we have developed a tool named Double Dutch, which uses a formal grammar and heuristic algorithms to automate the process of DOE library design. Compared to designing by hand, Double Dutch enables users to more efficiently and scalably design libraries of pathway variants that can be used in a DOE framework and uniquely provides a means to flexibly balance design considerations of statistical analysis, construction cost, and risk of homologous recombination, thereby demonstrating the utility of automating decision making when faced with complex design trade-offs.

Engineered biological systems are sensitive to the tuning of many interdependent variables, such as gene expression levels. By varying the genetic components that make up these systems, we can modulate gene expression levels and other parameters to learn how these systems respond to these changes and other inputs.

Historically, the engineering of biosynthetic pathways has been accomplished using both rational and combinatorial approaches. Most rational approaches use metabolic modeling to identify pathway targets for optimization via gene knockouts,[1] protein engineering,[2] and codon optimization, among other techniques. Combinatorial approaches, on the other hand, use techniques such as chemical mutagenesis, multiplex automated genomic engineering,[3] and global transcription machinery engineering[4,5] to generate libraries of pathway variants and screen them for desired phenotypes.

Recently, semirational approaches that rely on combinatorial assembly have been used to engineer libraries of biosynthetic pathway variants from characterized DNA components.[5−11] These approaches are attractive because they can require less data on system parameters than other rational approaches, they can leverage predictive models, and they can take full advantage of economies of scale for DNA assembly. The development of methods for designing and characterizing novel DNA components (such as promoters,[12−15] ribosome binding sites (RBS),[16,17] terminators,[18] and combinations thereof[19]) and rapidly assembling them into multigene constructs[20,21] has made it conceivable to design very large component-based libraries that can sample millions of pathway variants.

In practice, however, it is not practical to assemble and test millions of variants in order to elucidate how different DNA components affect the behavior of a pathway. Fortunately, statistics provides the means to construct empirical models of system behavior without testing all variants, using a rigorous mathematical approach known as design of experiments (DOE).[22] In classical DOE, a factorial design uses a limited set of designs to test how one or more factors affect a system of interest when taking on different combinations of quantitative levels selected beforehand. In order for DOE to be used in the context of genetic design, a natural choice is to consider the genes in a system of interest as the factors of a factorial design and the genetic components that can control their expression as the levels of the design.

Designing a set of DNA constructs that satisfy a factorial design is nontrivial. From here on, we call this the assignment problem, in which the same factorial design can be assigned different DNA components that vary in terms of accurately matching desired expression levels, DNA homology due to component reuse, and DNA synthesis cost. How to weight each of these design concerns is nonobvious, and this weighting may change from project to project. Clearly, a tool is needed that can quantitatively evaluate factorial designs based on these

Special Issue: IWBDA 2015

Received: November 13, 2015
Published: April 25, 2016

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

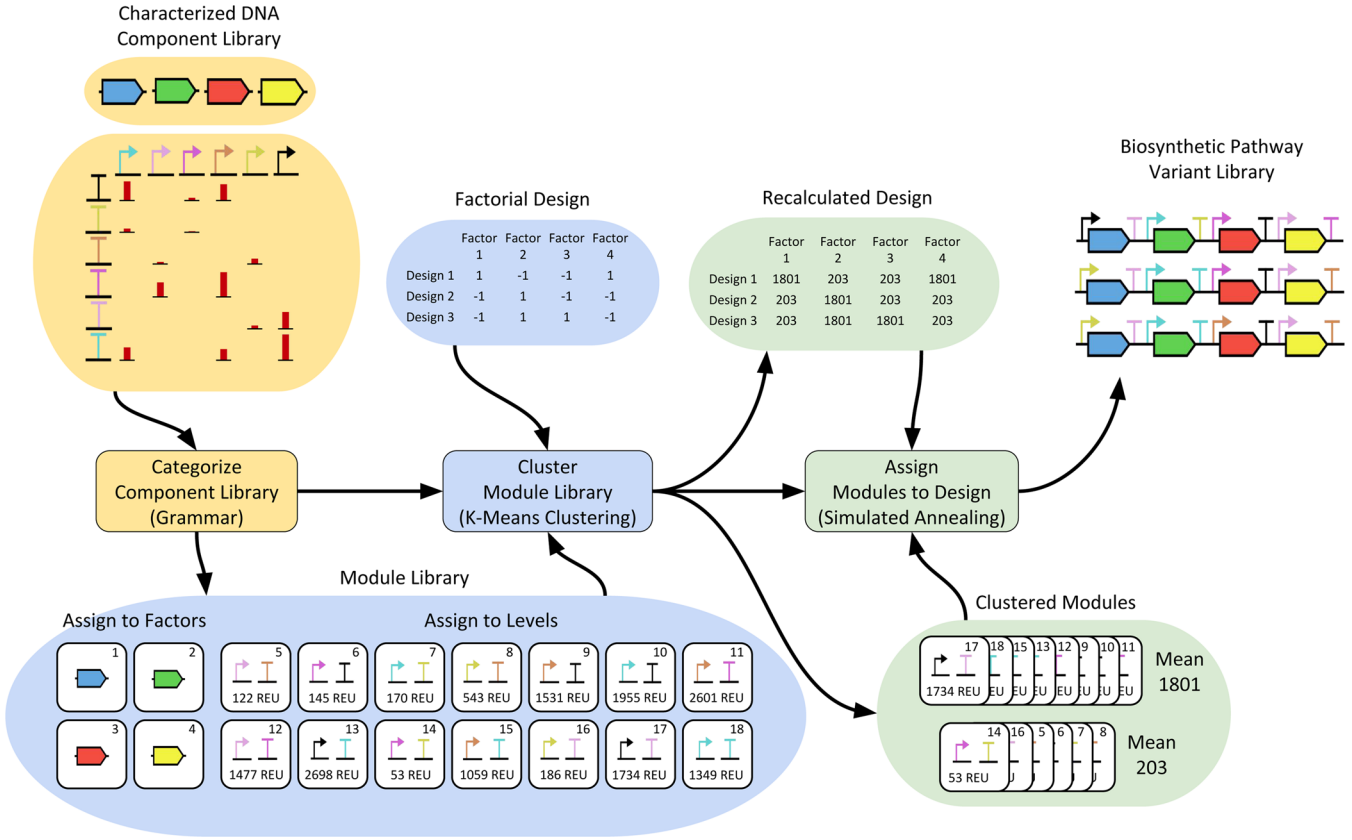


Figure 1. Workflow for designing libraries of biosynthetic pathway variants in Double Dutch (top) and examples of data involved in this workflow (bottom). Listed in parentheses under each step of the workflow is the method by which that step is carried out. The input data to each step are highlighted with the same color as that step. Each CDS, promoter, and terminator are represented using symbols from the SBOL Visual standard,[27] while the expression strengths for pairings of specific promoters and terminators are represented using bar graphs or numbers with relative expression units (REU). In this example of the workflow, once the parametrized modules are clustered, their mean parameter values become the new low and high levels of the factorial design. The parametrized modules are then assigned by Double Dutch from their clusters to the corresponding levels of the design (unparameterized modules are previously assigned by the user to the factors of the design). The final result is an optimized library of biosynthetic pathway variants based on a factorial design.

biological and economical concerns, which are among the three major challenges of the assignment problem.

The first challenge is the selection of components for assignment. Large data sets that contain information on hundreds to thousands of DNA component combinations must be categorized as eligible for assignment to either the factors of a design or to the levels that each factor takes on. This categorization step may change between projects and need to be repeated for different use cases (for example, varying pathway promoters versus RBSs). The second challenge is matching design levels to DNA components. In order to facilitate their reuse across different disciplines, factorial designs are typically written in a canonical form that represents levels using small integer values that are not always representative of the parameters associated with DNA components. Hence, most factorial designs must be recalculated to make it possible to match their levels to DNA components. The third challenge is the balanced consideration of experimental concerns during the assignment process. For example, due to the risk of homologous recombination, reuse of DNA components within a design must be controlled,[23,24] which favors using a large variety of components. On the other hand, reagent costs and handling considerations favor using the fewest unique components possible. As a result, many possible assignments must be evaluated to find those that balance these competing concerns.

To address these challenges, we have developed an approach to designing libraries of pathway variants based on factorial designs and implemented it in a web application named Double Dutch. This approach leverages a formal grammar and heuristic algorithms (k-means clustering[25] and simulated annealing[26]) in conjunction with a weighted cost function to search the space of library designs and find those that are near-optimal with respect to matching desired expression levels and minimizing pathway homology and DNA synthesis costs.

METHODS

Overview of Double Dutch Workflow. Figure 1 illustrates the overall workflow for designing libraries of biosynthetic pathway variants in Double Dutch. This workflow consists of three major steps: categorization of a DNA component library into parametrized and unparameterized modules, clustering of the parametrized modules, and assignment of modules from these clusters to the levels of a factorial design.

In order to engage with the Double Dutch workflow, users must first upload data on their characterized DNA components and factorial designs in the form of comma-separated value (CSV) files. The CSV file for a factorial design is expected to

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

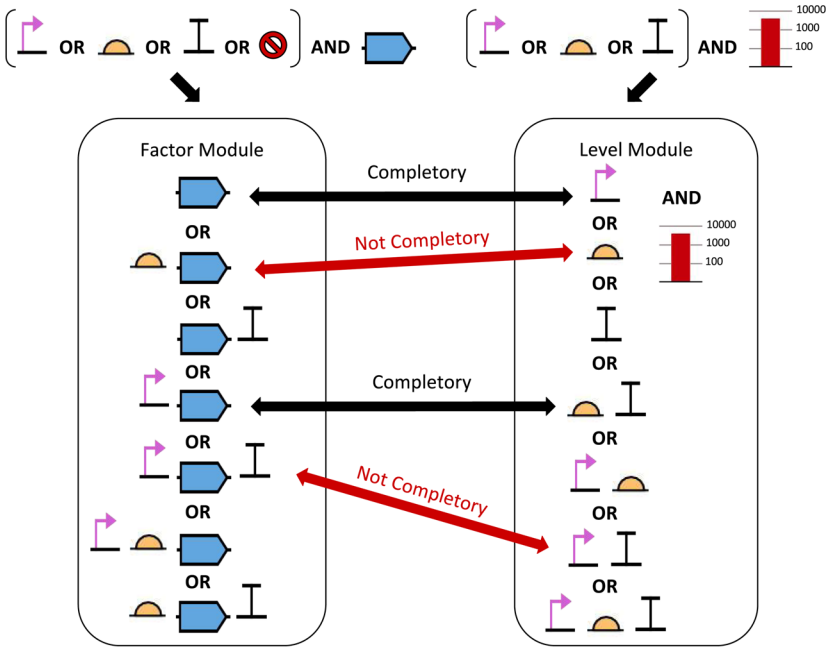


Figure 2. Visualization of Double Dutch grammar. The rules of this grammar specify that one or more DNA components forming part of a transcriptional unit map to a module. Furthermore, any module that includes a CDS is a "factor module" (left) that is eligible for assignment to a factor in a factorial design, and any module that lacks a CDS and has a parameter (red bar graph) is a "level module" (right) that can be assigned to a level that a factor takes on. The only other restriction on module assignment is that level modules can be assigned only to the levels of factors that have "completory" factor modules assigned to them. In other words, the contents of coassigned level modules and factor modules must be capable of forming complete transcriptional units that include one promoter, up to one RBS, one CDS, and one terminator. The black and red arrows indicate examples of completory and not completory modules, respectively.

contain a matrix of real numbers. In the Double Dutch workflow, files for basic factorial designs (including full factorial, Plackett–Burman,[28] Box–Behnken,[29] and 2^{k-p} fractional factorial[30] designs) are generated by Double Dutch from templates described in the literature, but Double Dutch can also import files for other factorial designs generated by statistics software packages, such as JMP.[31] Each CSV file for a DNA component library is expected to contain a table of component identifiers and their associated properties (for example, role, sequence, and strength) or a matrix of parameters for component combinations in which each row and column are labeled with a component identifier and role. Examples of these files are available in the Supporting Information and at www.doubledutchcad.org.

DNA Component Categorization. In order to determine which DNA components are eligible for assignment to the factors and levels of a factorial design, Double Dutch implements a formal grammar (see Supporting Information). This grammar effectively consists of a set of rules for combining a parameter and/or set of DNA components into a "module" that is eligible for assignment to a factor or level. Figure 2 provides a visual representation of Double Dutch grammar.

A module assigned to a factor contains components that are common to a gene in every pathway variant, such as a coding sequence (CDS). Modules assigned to levels contain components that vary in a gene across different pathway variants and modulate its expression, such as promoters. Here, we focus on assigning modules that contain CDSs to factors and modules that contain promoters, terminators, and expression strengths to levels. However, Double Dutch is capable of supporting other use cases, such as those shown in Figure 2.

Problem Formulation. The following definitions provide a more explicit formulation of the Double Dutch module assignment problem.

Definition 1. A DNA component library is a set P of sets of DNA component identifiers. Each component identified by these sets must have a role that is permitted by Double Dutch grammar: promoter, RBS, CDS, or terminator.

Definition 2. A factor module library is a pair $\langle N, \psi \rangle$, including a set N of factor module identifiers, and a mapping ψ: $N \rightarrow P$ from each identified factor module to a set of DNA component identifiers. The components identified in this way must constitute a valid factor module as defined by Double Dutch grammar.

Definition 3. A level module library is a triple $\langle M, \mu, \phi \rangle$, including a set M of level module identifiers, a mapping μ: $M \rightarrow \mathbb{R}$ from each identified level module to a real number parameter, and a mapping ϕ: $M \rightarrow P$ from each identified level module to a set of DNA component identifiers. The components identified in this way must constitute a valid level module as defined by Double Dutch grammar.

Definition 4. A factorial design is a two-dimensional array D of real numbers, where each row represents a design and each column represents a factor that is common to each design in the overall factorial design. Each number $D_{i,j}$ in the matrix then represents the level that the column's factor takes on in the row's design variant.

Definition 5. A factor module assignment is a sequence I of factor module identifiers. The length of I is equal to the number of columns in the factorial design D. In a valid factor module assignment, each factor module must contain the same number of components with the same nonrepeated roles.

Definition 6. A level module assignment is a sequence G of sequences of level module identifiers. The length of G is equal

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507–517

to the number of columns or factors in the factorial design D, while each child sequence G_j has a length equal to the number of unique numbers or levels in the jth column of D. In a valid level module assignment, each level module must contain the same number of components with the same nonrepeated roles. In addition, the components of each level module must be completory with respect to the components of its associated factor module. That is, the roles of the DNA components identified in each set $\psi(I_j)$ must be capable of forming a complete transcriptional unit with the components identified in each set $\phi((G_j)_i)$ (see Figure 2). These rules constrain the calculation of assignment costs as described later on.

Definition 7. An assignment cost function π outputs a real number cost s and takes as input a factorial design D, a DNA component library P, a level module library $\langle M, \mu, \phi\rangle$, a level module assignment G, and a triple W of real number weights. As described later, π is implemented in Double Dutch by Algorithms 4–6.

Definition 8. A solution to the Double Dutch module assignment problem is to find the level module assignment G that minimizes the Double Dutch assignment cost function π for a given factorial design D, DNA component library P, factor module library $\langle N, \psi\rangle$, level module library $\langle M, \mu, \phi\rangle$, and factor module assignment I.

Module Clustering. The first step in the Double Dutch approach to solving the module assignment problem is to use k-means clustering to cluster the library of level modules available for assignment to the factorial design and then to use linear regression to recalculate the levels of the design. These are preprocessing heuristics that make level matching more efficient during module assignment by eliminating the need to search the entire level module library for a match to a design level. They also enable calculation of the level matching cost for a module assignment, as implemented in the Double Dutch assignment cost function (see Algorithm 4).

Algorithm 1 is the main algorithm for module clustering. It takes as input a factorial design D, a library of level modules μ, and a number of clustering trials n, and it outputs clusters of level modules C and the recalculated factorial design D. The factorial design is encoded as a two-dimensional array of real numbers, while the clusters of level modules are encoded as a sequence of sequences of sets of module identifiers. Each subsequence represents the level modules clustered for assignment to the levels for one factor in the design. A sequence of sequences is used instead of a two-dimensional array because it is possible that the grid of level module clusters is ragged. That is, there might be a different number of levels per factor in the design and, hence, a different number of module clusters for assignment to these levels.

Algorithm 1: Cluster Modules

Input: Factorial design D; library of level modules $\langle M, \mu, \phi\rangle$; number of clustering trials encoded as an integer n
Output: Clusters of level modules encoded as a sequence C of sequences of sets of module identifiers; recalculated factorial design D

1 $C \leftarrow \langle\rangle$
2 **for** $j \leftarrow 0 \ldots$ (number of columns in D) $- 1$ **do**
3 $L \leftarrow$ sequence of unique numbers in jth column of D sorted smallest to largest
4 $s \leftarrow -1$
5 **for** $0 \ldots n - 1$ **do**
6 $\langle C'_j, A'\rangle \leftarrow$ K-Means-Cluster-Modules($\langle M, \mu, \phi\rangle$, length of L) where A' is the sequence of cluster means sorted smallest to largest
7 $s' \leftarrow$ Score-Clusters($C'_j, A', \langle M, \mu, \phi\rangle$)
8 **if** $s < 0 \vee s' < s$ **then**
9 $\langle s, C_j, A\rangle \leftarrow \langle s', C'_j, A'\rangle$
10 $\langle \lambda, e\rangle \leftarrow$ Calculate-Linear-Regression(L, A), where $\lambda : \mathbb{R} \rightarrow \mathbb{R}$ is the regression function and e is the standard error of regression
11 **for** $i \leftarrow 0 \ldots k - 1$ **do**
12 $O_i \leftarrow \log_{10}(A_i)$
13 $\langle \lambda', e'\rangle \leftarrow$ Calculate-Linear-Regression(L, O)
14 **if** $e < e'$ **then**
15 **for** $i \leftarrow 0 \ldots$ (number of rows in D) $- 1$ **do**
16 $D_{i,j} \leftarrow \lambda(D_{i,j})$
17 **for** $i \leftarrow 0 \ldots k - 1$ **do**
18 $A_i \leftarrow \lambda(L_i)$
19 **else**
20 **for** $i \leftarrow 0 \ldots$ (number of rows in D) $- 1$ **do**
21 $D_{i,j} \leftarrow 10^{\lambda'(D_{i,j})}$
22 **for** $i \leftarrow 0 \ldots k - 1$ **do**
23 $A_i \leftarrow 10^{\lambda'(L_i)}$
24 $C_j \leftarrow$ Re-Cluster-Modules($\langle M, \mu, \phi\rangle, A$)
25 **return** $\langle C, D\rangle$

Lines 3–9 of Algorithm 1 perform k-means clustering on the library of level modules, preparing them for assignment to levels L of the design factor currently under consideration (the jth column of D). During clustering, each module is added to the cluster with the ith mean A_i, the cluster that minimizes the distance criterion $|\mu(m) - A_i|$. While k-means clustering always converges to a solution, it may not always converge to same solution when given the same input. Hence, k-means clustering is performed n time,s and the results are scored to determine the best sequence of module clusters to store as the jth entry of C.

Once k-means clustering is complete, Algorithm 2 scores the resulting module clusters using a normalized distance criterion.

Algorithm 2: Score Clusters

Input: Clusters of level modules encoded as a sequence C (length k) of sets of module identifiers; cluster means encoded as a sequence A (also length k) of real numbers; library of level modules $\langle M, \mu, \phi\rangle$
Output: Score for clusters of level modules encoded as a real number s

1 $s \leftarrow 0$
2 **for** $i \leftarrow 0 \ldots k - 1$ **do**
3 $V \leftarrow \{\}$
4 **for** $m \in C_i$ **do**
5 Add $|\mu(m) - A_i|$ to V
6 $s' \leftarrow 0$
7 **for** $v \in V$ **do**
8 $s' \leftarrow s' + (v - \min V)$
9 $s \leftarrow s + s'/(\max V - \min V)$
10 **return** s

Namely, each module is scored using the clustering distance criterion $|\mu(m) - A_i|$, and each module cluster is scored as the sum of the differences between its modules' scores and the minimum score among its modules, divided by the difference between the maximum and minimum scores among its modules. Normalization is necessary to prevent the score for a cluster that contains modules associated with larger parameters from dominating the combined score for all k module clusters, which is obtained via a simple summation.

Once the best sequence of module clusters C_j has been determined, lines 10–23 of Algorithm 1 handle the process of recalculating the canonical integer levels of the jth factor in the design so that they fall within the range of the parameters associated with the clustered modules. Doing so enables these modules to later be assigned to the design levels on the basis of the quantitative differences between their parameters and the level values. There are two primary steps in the level recalculation process: the first step is to calculate both linear and log−linear regressions that relate the design levels to the cluster means. The second step is to use the regression with the smallest standard error (the best fit) to recalculate the design levels. These recalculated levels should be similar in magnitude to the cluster means, but since they are not always equal, Algorithm 1 concludes by replacing the cluster means with the recalculated levels and redividing the modules among the

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507–517

clusters in accordance with the clustering distance criterion. This accounts for the edge case in which a module is on the boundary of one cluster and should shift to another cluster following the adjustment of the cluster means.

Module Assignment. Once the modules have been clustered, Double Dutch uses simulated annealing to optimally assign modules in accordance with a cost function. A module assignment is a mapping between one module from each cluster and its corresponding level and is costed as the weighted sum of three design concerns: level matching, pathway homology, and DNA synthesis cost. Double Dutch attempts to manage these competing concerns according to user-defined weights and find the module assignment with the smallest total weighted cost.

Algorithm 3 outputs a module assignment G and takes as input the output of Algorithm 1 (a recalculated factorial design D and clusters of level modules C), a level module library μ, cost function weights W, level frequency ratios H, and several parameters for simulated annealing, including an initial temperature t', a number of iterations k, a annealing constant b, and a number of trials n'.

Algorithm 3: Assign Modules

Input: Factorial design D; library of DNA components P; library of level modules $\langle M, \mu, \phi \rangle$; clusters of level modules encoded as a sequence C of sequences of sets of module identifiers; cost function weights W; level frequency ratios encoded as a sequence H of sequences of real numbers; initial annealing temperature encoded as a real number t'; number of iterations per annealing encoded as an integer k; annealing modifier constant encoded as a real number b; number of annealing trials encoded as an integer n'

Output: Level module assignment G

```
1  n ← 0
2  while n < n′ do
3      G ← ⟨⟩
4      for j ← 0 . . . (length of C) − 1 do
5          G_j ← ⟨⟩
6          for i ← 0 . . . (length of C_j) − 1 do
7              (G_j)_i ← random module identifier m ∈ (C_j)_i
8      s ← Calculate-Assignment-Cost(G, D, P, ⟨M, μ, φ⟩, C, W, H)
9      if n = 0 then
10         G″ ← G
11         s″ ← s
12     t ← t′
13     while t ≥ 1 do
14         j ← random integer [0, length of G − 1]
15         i ← random integer [0, length of G_j − 1]
16         G′ ← copy G
17         (G′_j)_i ← random module identifier m ∈ (C_j)_i
18         s′ ← Calculate-Assignment-Cost(G′, D, P, ⟨M, μ, φ⟩, C, W, H)
19         if s′ ≤ s ∨ random real number [0, 1] ≤ e^{b∗(t′/t)∗(s−s′)} then
20             s ← s′
21             G ← G′
22         t ← (1/t′)^{1/k} ∗ t
23     if s ≤ s″ then
24         s″ ← s
25         G″ ← G
26     n ← n + 1
27 return G″
```

Lines 3−8 of Algorithm 3 initialize the module assignment G by randomly selecting one module from each module cluster and then calculating the cost of this assignment. If this is the first trial of simulated annealing, then the initial module assignment G is stored as the current best module assignment G'' in lines 9−11.

Next, lines 12−22 handle the process of simulated annealing. During each iteration of this process, G is copied to G' and one copied assigned module $(G'_j)_i$ is randomly changed or "mutated" to another module in the corresponding cluster $(C_j)_i$. This mutation is then accepted and copied back into the current module assignment G if one of two conditions is met. First, if G' has a lower cost than G, then the mutation is accepted. Second, if G' has a higher cost than G, then the mutation is accepted only if a specific probabilistic expression is satisfied. The probability that this expression is satisfied is dependent on the current temperature t and inversely dependent on the difference in cost between G and G'. Accepting worse assignments in this manner can help prevent the search for the assignment with the smallest cost from stopping at a local minimum. Since the temperature decreases with each iteration of simulated annealing, the probability of accepting worse module assignments generally decreases over time.

Parameters t' and b influence how the probability of accepting worse assignments changes with decreasing temperature, with b also determining the range of initial probabilities. Parameters k and n, on the other hand, change the runtime of simulated annealing by determining the number of mutations during annealing and the number of times the entire process is repeated starting from a random initial assignment, respectively. The worst-case runtime for simulated annealing in Double Dutch scales as the product of n, k, and the size of the design in terms of its number of factors and unique levels per factor. The latter term exists because each mutation step in a given trial results in the recalculation of the assignment cost, which has a runtime that depends on the size of the factorial design.

Line 18 of Algorithm 3 is used to calculate the cost of the mutated module assignment G'. The cost function weights W modify the contribution of the cost of each design concern to the total cost, while the level frequency ratios H can modify how these costs are calculated. H captures how frequently a factor in the design takes on a particular level relative to a factor in a full factorial design. For example, $(H_j)_i = 2$ would indicate that the ith level of the jth factor appears twice as frequently in this design as it would in a full factorial design. Consequently, if a module assigned to this level has a good match, then it should be weighted accordingly during the calculation of the total level matching cost.

Algorithm 4 calculates the level matching cost of a module assignment. In general, this process involves summing the differences between the parameters of the assigned modules in G and the corresponding design levels L of D. These differences are the level matching costs of the individual assigned modules. In order to prevent the costs of modules with larger parameters from dominating the level matching cost, they must be normalized using the smallest and largest costs of other modules in the same cluster. Lines 5−7 of Algorithm 4 calculate the costs V of all modules in the cluster $(C_j)_i$, while line 8 calculates the cost v of the module $(G_j)_i$ assigned from this

Algorithm 4: Calculate Level Matching Cost

Input: Level module assignment G; factorial design D; clusters of level modules encoded as a sequence C of sequences of sets of module identifiers; library of level modules $\langle M, \mu, \phi \rangle$; level frequency ratios encoded as a sequence H of sequences real numbers

Output: Level matching cost encoded as a real number s

```
1  s ← 0
2  for j ← 0 . . . (number of columns in D) − 1 do
3      L ← sequence of unique numbers in jth column of D sorted smallest to largest
4      for i ← 0 . . . (length of L) − 1 do
5          V ← {}
6          for m ∈ (C_j)_i do
7              Add |μ(m) − L_i| to V
8          v ← |μ((G_j)_i) − L_i|
9          s ← s + (H_j)_i ∗ (v − min V)/(max V − min V)
10 s″ ← 0
11 for j ← 0 . . . (length of H) − 1 do
12     for i ← 0 . . . (length of H_j) − 1 do
13         s″ ← s″ + (H_j)_i
14 s ← s/s″
15 return s
```


DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

cluster. Line 9 then adds the product of $(H_j)_i$ and v to the total level matching cost s but not before normalizing v by subtracting from it the minimum module cost min V and dividing it by the difference in min V and max V. In principle, min V and max V can determined and memoized earlier during cluster scoring in Algorithm 2, and this is what is done in practice in Double Dutch. Once the total level matching cost has been calculated, lines 10−14 normalize this cost to between 0 and 1 by dividing it by the sum of the level frequency ratios H (the cost of the module assigned from each cluster $(C_j)_i$ has a maximum possible value of 1, which would then be multiplied by $(H_j)_i$).

Next, Algorithm 5 is used to calculate the DNA synthesis cost of a module assignment. Lines 1−8 calculate the base

Algorithm 5: Calculate DNA Synthesis Cost

Input: Level module assignment G; library of DNA components P; library of level modules $\langle M, \mu, \phi \rangle$
Output: Level matching cost encoded as a real number s

```
1  s ← 0
2  R ← {}
3  for j ← 0 ... (length of G) − 1 do
4      for i ← 0 ... (length of G_j) − 1 do
5          for p ∈ φ((G_j)_i) do
6              if p ∉ R then
7                  s ← s + 1
8                  Add p to R
9  s″ ← 0, where s″ is the maximum DNA synthesis cost
10 s′ ← 0, where s′ is the minimum DNA synthesis cost
11 z ← size of φ(m) for any m ∈ M
12 for j ← 0 ... (length of G) − 1 do
13     s″ ← s″ + z ∗ length of G_j
14     z′ ← z + length of G_j − 1
15     if s′ < z′ then
16         s′ ← z′
17 R ← {}
18 for P′ ∈ P do
19     for p ∈ P′ do
20         Add p to R if p ∉ R
21 if s″ > size of R then
22     s″ ← size of R
23 s ← (s − s′)/(s″ − s′)
24 return s
```

DNA synthesis cost, which is the total number of unique DNA components among the assigned modules in G. The DNA synthesis cost is then normalized to between 0 and 1 using estimates of the maximum and minimum DNA synthesis costs s'' and s', respectively. These estimates are calculated by lines 9−22. To estimate the maximum cost of DNA synthesis, line 13 adds to s'' the product of the number of components per level module z and the length of G_j (equivalent to the number of levels for the jth factor of the design). This is done under the assumption that the components being tallied are distinct from those previously tallied in this manner. Once this process has been repeated for every factor in the design, lines 17−22 limit s'' to the actual number of unique components available in O if the former exceeds the latter. To estimate the minimum DNA synthesis cost, lines 14−16 use z to calculate z', which captures the minimum number of unique components necessary to implement the modules assigned to the levels of one factor in the design. For example, if a given factor has two levels and z = 2, then a minimum of three unique components is necessary to implement the two modules assigned to these levels, since the two components in the first module must differ by one component from those in the second if we assume that different levels of the same factor must be implemented by nonidentical sets of DNA components. Since the grammar of Double Dutch prohibits the repetition of components of the same role, this must be accomplished by using at least one unique component per level rather than by varying the order of components of the same role.

Lastly, Algorithm 6 is used to calculate the pathway homology cost of a module assignment. The basic strategy of

Algorithm 6: Calculate Pathway Homology Cost

Input: Level module assignment G; library of level modules $\langle M, \mu, \phi \rangle$; level frequency ratios encoded as a sequence H of sequences real numbers
Output: Level matching cost encoded as a real number s

```
1  Q ← {}, where Q is a set of component frequencies encoded as sequences of integers
2  κ : O → Q
3  for j ← 0 ... (length of G) − 1 do
4      for i ← 0 ... (length of G_j) − 1 do
5          for p ∈ φ((G_j)_i) do
6              if κ(p) is undefined then
7                  Add sequence of 0s with same length as G to Q
8                  κ(p) ← last sequence added to Q
9              κ(p)_j ← κ(p)_j + (H_j)_i
10 Q″ ← {}, where Q″ is a set of max comp. freq. encoded as sequences of integers
11 z ← size of φ(m) for any m ∈ M
12 for z′ ← 0 ... z − 2 do
13     Add sequence of 0s with same length as G to Q″
14 U ← ⟨⟩
15 for j ← 0 ... (length of H) − 1 do
16     for i ← length of U ... (length of H_j) − 1 do
17         U_i ← Sequence of 0s with same length as G
18     for i ← 0 ... (length of H_j) − 1 do
19         for F″ ∈ Q″ do
20             F″_j ← F″_j + (H_j)_i
21         (U_i)_j ← (U_i)_j + (H_j)_i
22 Add contents of U to Q″
23 Q′ ← Calculate-Min-Component-Frequencies(G, C)
24 s ← Sum-Component-Frequencies(Q)
25 s″ ← Sum-Component-Frequencies(Q″)
26 s′ ← Sum-Component-Frequencies(Q′)
27 s ← (s − s′)/(s″ − s′)
28 return s
```

this algorithm is to sum the frequencies of each DNA component in the module assignment G (as modified by the level frequency ratios H). This sum is related to the number of repeated components or instances of homology in the pathway variants that result from the module assignment, but it can be calculated more efficiently than the latter. Lines 1−9 calculate the component frequencies per factor Q from the module assignment G, while lines 10−22 and 23 estimate the maximum and minimum possible component frequencies by factor (Q'' and Q'), respectively. Q'' is estimated under the assumptions that the modules assigned to the levels of each individual factor share all but one component and that identical modules are assigned to the jth levels of each factor. Q' is estimated only under the first assumption. This assumption alone would still result in a minimum pathway homology cost of 0, since components are not repeated across factors in the design and hence are not repeated across genes in the pathway variants. If this assumption would require too many unique components, then it is instead assumed that components are repeated as sparingly as possible between the assigned modules with the same indices j and i in G as the indices of the smallest level frequency ratios in H (these ratios are preranked, so they do not need to be sorted every time the pathway homology cost is computed).

Algorithm 7 is used in lines 25−27 to calculate the pathway homology cost s of the module assignment and the maximum and minimum possible costs s'' and s' from the component frequencies. s is then normalized using s'' and s' on line 28. Note that Algorithm 7 sums only the component frequencies from the sequences $F \in Q$ that contain more than one nonzero entry, that is the frequencies of components that appear in modules assigned to the levels for more than one factor. In

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

Algorithm 7: Sum Component Frequencies

Input: DNA Component frequencies encoded as a set Q of sequences of real numbers
Output: Non-normalized pathway homology cost encoded as a real number s

```
s ← 0
x ← 0
for F ∈ Q do
    v ← 0
    y ← 0
    for j ← 0 . . . length of F do
        if F_j > 0 then
            v ← v + F_j
            y ← y + 1
    if y > 1 then
        s ← s + v
        x ← x + 1
if x > 1 then
    s ← s − (x − 1)
return s
```

addition, given two module assignments with the same total component frequencies (for example, two components each repeated once versus one component repeated three times), the algorithm rewards the assignment that exhibits greater diversity in its repeated components, since this leads combinatorially to fewer pathway variants that contain homology.

RESULTS

Cost Function. The Double Dutch cost function allows users to specify the relative importance of level matching, homology, and DNA synthesis in designing a library of pathway variants based on a factorial design. Users may alter the weights of each design concern, even down to zero if they do not want Double Dutch to optimize that concern. In addition, users may "flip" the DNA synthesis concern so that Double Dutch attempts to maximize the total number of unique components in the final library, thereby optimizing library diversity instead of controlling synthesis costs. Ultimately, changing the weights of the cost function results in different assignment costs for each design concern and different libraries of pathway variants. Figure 3 illustrates this concept for the assignment of modules from a library described in the next section to a four-factor Plackett–Burman design. Ordinarily, the assignment costs in this figure would be normalized, weighted, and summed. Here, the raw costs are shown to motivate a more intuitive understanding of how different weightings affect the cost distribution.

In this example, doubling the DNA synthesis weight halves the number of unique DNA components in the optimized library, but it repeats 17 components in the pathway variants and doubles the total difference between the parameters of the assigned modules and the levels of the factorial design. Note that in this case Double Dutch assigns only the black or olive promoter to every gene in every pathway variant. Quadrupling the level matching weight, on the other hand, roughly halves the total difference between module parameters and design levels, but it repeats seven components in the pathway variants (the black promoter and black terminator). The number of unique components in the library is also decreased by one-quarter, but this is expected given that DNA synthesis and pathway homology are partly in opposition as design concerns. In both cases, optimization comes at the expense of increasing the assignment cost of at least one other design concern.

Performance. This section evaluates the performance of simulated annealing in Double Dutch and compares it to the performance of purely random assignment. Figure 4 presents the average module assignment costs resulting from 100 trials of these heuristics as applied to full factorial designs that contain 5, 10, and 15 factors and 2, 3, and 5 levels per factor. These values were chosen with respect to the size of the module library tested here and the following facts: to date, the largest engineered biosynthetic pathways have contained on the order of 20 genes,[32] while the most expressive engineered bacterial systems have spanned between 4–5 orders of magnitude in gene expression.[8] Modules were assigned from a library containing 1221 combinations of 37 promoters and 33

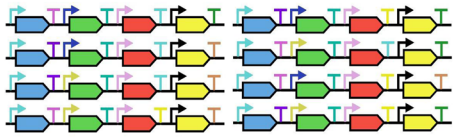
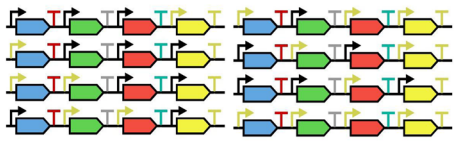
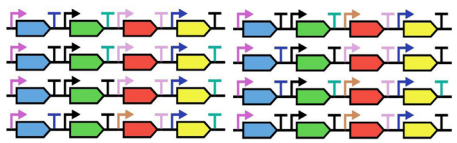
Cost Function Weights			Raw Assignment Costs			
Level Matching	DNA Synthesis	Pathway Homology	Total Level Difference	# of Unique Components	# of Repeated Comp. in Pathways	Optimized Library of Biosynthetic Pathway Variants
1	1	1	1487.6	12	0	
1	2	1	3235.12	6	17	
4	1	1	819.94	9	7	

Figure 3. Example of how changing the cost function weights results in different assignment costs for each design concern and pathway libraries for a four-factor Plackett–Burman design (see Supporting Information). The first three columns list each cost function weight by the design concern that it affects, while the second three columns list the resulting raw (unweighted, non-normalized) assignment costs of the design concerns. Lastly, the third column displays the optimized libraries of biosynthetic pathway variants resulting from each module assignment. Different promoter and terminator assignments are represented using different colors. Increasing the weights for design concerns other than pathway homology optimize those concerns, but this causes the gold and black promoters and black terminator to be repeated within each pathway variant.

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507–517

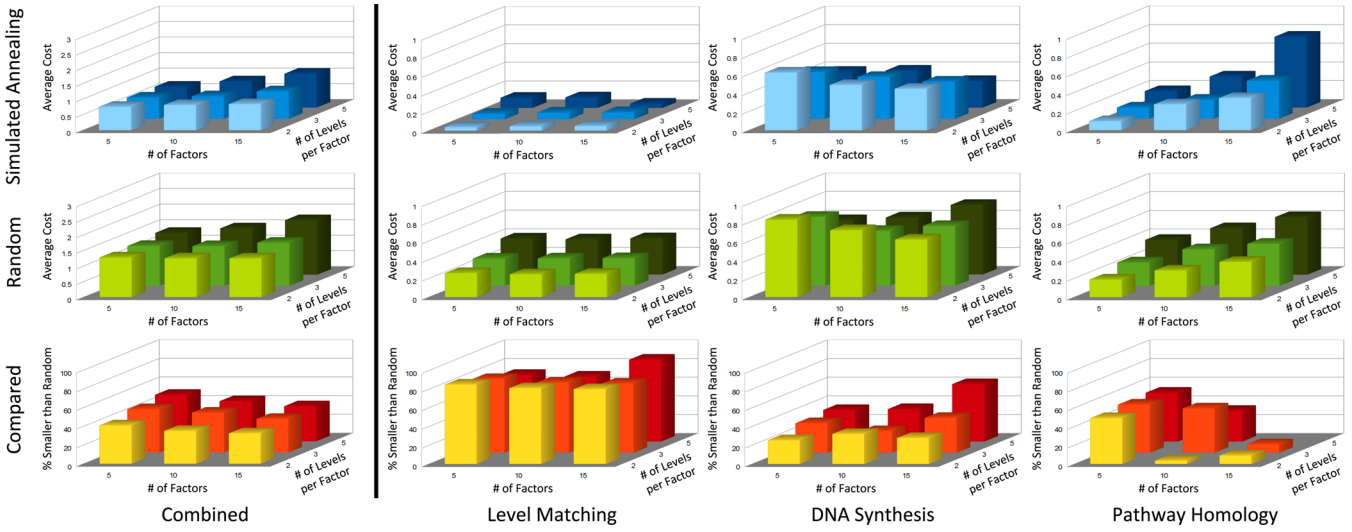


Figure 4. Comparison of the average module assignment costs for 100 trials of simulated annealing and purely random assignment. These include the average costs for assignment via simulated annealing (blue), the average costs for random assignment (green), and the percentages by which the former are smaller than the latter (red-yellow) for nine different design sizes. Each column beyond the first contains the average assignment costs and percentages for an individual design concern (level matching, DNA synthesis, or pathway homology), while the first column contains the average module assignment costs and percentages for the combination of all three concerns. Note that the percentage for pathway homology for the largest design is not shown but has a value of −22.

terminators (see Supporting Information) that were each characterized for their expression of green fluorescent protein (GFP) in yeast. (Expression strength was measured as the geometric mean of absolute GFP fluorescence during the mid log phase of growth.) In addition, all three design concerns were weighted equally. Since each concern has a worst possible cost of 1, their combination has a worst possible cost of 3. While the results presented here are somewhat specific to this module library, the general trends discussed in this section are expected to hold for other module libraries and factorial designs.

In general, the average cost of module assignment for simulated annealing and random assignment increases linearly as the number of factors and levels in the design increases. The average cost of module assignment for simulated annealing, however, is between 30 and 50% smaller than that for random assignment for all design sizes tested, and these differences are statistically significant (that is, there are greater than 2 standard errors separating each pair of average costs). Experimentally, the cost difference between randomly designed libraries and libraries designed with simulated annealing would be confirmed through their construction, screening for product yield, and statistical analysis. We would expect the libraries designed with simulated annealing to be cheaper to construct in terms of reagent and handling costs (DNA synthesis concern), result in fewer failed pathway variants (pathway homology concern), and be more amenable to analysis using techniques from DOE (level matching concern).

Finally, while random assignment is faster than simulated annealing on average (milliseconds versus seconds per trial), the time taken for simulated annealing scales linearly with design size and is on average less than 5 s per trial for the chosen simulated annealing parameters (see Methods), as shown in Figure 5. All module assignments in this article were made with $n = 100$, $k = 10^4$, $t' = 10^9$, and $b = 0.1$. There may exist other parameter sets that can be considered more optimal with respect to runtime versus quality of assignment found, but this set was found to work well enough for the purposes of this article.

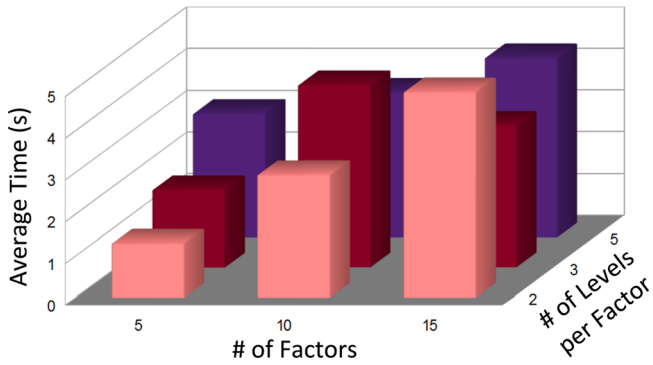


Figure 5. Average module assignment times (per trial) for 100 trials of simulated annealing. In general, the average time taken for assignment via simulated annealing increases linearly with the size of the design, but it is less than 5 s per trial for the largest designs with 15 factors.

Figure 4 also shows how the average cost of each design concern is affected by changes in design size. In particular, the average DNA synthesis cost is least affected by the size of a design, with 2.2-fold separation between the smallest and largest average costs for simulated annealing and 1.4-fold separation for random assignment. The average level matching cost is the next most affected, with 3-fold separation for simulated annealing and 1.6-fold separation for random assignment, followed by the average pathway homology cost, with 7.8-fold separation for simulated annealing and 3.2-fold separation for random assignment. The average level matching cost increases only with the number of levels per factor in the design, while the average pathway homology cost generally increases with the overall size of the design. Interestingly, the average DNA synthesis cost generally decreases with design size for simulated annealing, but it increases with design size for random assignment after an initial decrease.

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

The effect of design size on the average pathway homology cost is not unexpected, since any finite module library cannot continue to implement larger and larger designs without assigning modules that contain nonunique DNA components and introduce pathway homology. The effect of design size on the average level matching cost, on the other hand, can be explained as follows: as the number of levels per factor in the design increases, the modules available for assignment are divided between a larger number of clusters such that each cluster contains fewer modules. Consequently, the modules in each cluster have parameters that are less representative of the cluster mean. Since the level matching cost is based on the difference between the assigned modules' parameters and the cluster means, this results in a larger average level matching cost. Lastly, the effect of design size on the average DNA synthesis cost is the result of random sampling. Namely, as a design increases in size, a module assignment represents a larger and larger random sample of the module library until it is very likely that at least one of every unique DNA component in the library is present among the assigned modules, which maximizes the cost of DNA synthesis. Simulated annealing, however, optimizes module assignment and undoes this random sampling effect for the design sizes tested here. The reason that the average DNA synthesis cost initially decreases is partly an artifact of how this cost is calculated. The number of unique DNA components in the module assignment is partly normalized through its division by the maximum number of unique DNA components that can possibly be found in the assignment (see Methods). As the design increases in size, this maximum number grows more quickly than the average number of unique components found in the assignment until it reaches the total number of unique DNA components present in the module library. Table 1 summarizes the general effects of design size on each concern and their probable causes.

Table 1. Summary of Effects of Design Size on Each Concern and Their Probable Causes

	level matching	DNA synthesis	pathway homology
effect as no. of factors increases		cost decreases, then increases	cost increases
effect as no. of levels increases	cost increases	cost decreases, then increases	cost increases
cause of effects	small cluster size	normalization artifact, law of large numbers	finite module library

Figure 4 also shows which concerns are optimized by simulated annealing over random assignment. Most notably, the majority of the benefit to the average combined cost that comes from performing simulated annealing instead of random assignment can be seen in the optimization of the average level matching cost. In addition, these results indicate that the optimization of the average DNA synthesis cost is partly in opposition to the optimization of the average pathway homology cost. This is an unsurprising result since an assignment that reuses as many DNA components as possible would maximize pathway homology. In general, these results show that simulated annealing optimizes the average combined cost for smaller designs by minimizing level matching and pathway homology at the expense of increasing DNA synthesis and for larger designs by minimizing level matching and DNA synthesis at the expense of increasing pathway homology. The latter effect is most evident for the largest 15 factor design, for which the average combined cost is smaller for simulated annealing than for random assignment, despite the fact that the average pathway homology cost for simulated annealing is 22% larger than for random assignment.

Ultimately, the design size that results in the greatest optimization by simulated annealing over random assignment has the fewest factors and the most levels per factor. This is because having fewer factors makes the design smaller and easier for simulated annealing to uniquely implement without introducing pathway homology, and having more levels per factor enables simulated annealing to reuse more DNA components across the levels that a single factor takes on (that is, across different variants of the same gene), thereby decreasing DNA synthesis without increasing pathway homology.

DISCUSSION

Double Dutch enables users to design combinatorial component-based libraries of biosynthetic pathway variants based on quantitative factorial designs. Double Dutch uses k-means clustering and simulated annealing to automate the design of these libraries and to balance competing concerns, including level matching, pathway homology, and DNA synthesis. Compared to designing such libraries by hand, Double Dutch enables users to more efficiently and scalably design DOE libraries that are less likely to fail due to homologous recombination and that are cheaper to synthesize in terms of their base-level DNA components, such as promoters, RBSs, and terminators.

Besides automation, one of the most significant contributions that Double Dutch makes to library design is the introduction of a cost function. This cost function makes it possible for users to quantitatively compare different library designs and weight how important each design concern is to them. In addition, the cost function enables users to see how modifying a design affects its total cost and divides this cost among the different design concerns. The latter feature enables users to explore any design trade-offs that may exist between these concerns. Our framework allows for the cost function to be extended to account for new experimental concerns and other design concerns.

Possible future directions include incorporating Double Dutch into a workflow for optimizing biological designs that leverages a laboratory inventory management system, database search engine, statistics software packages, and other combinatorial design tools, including Eugene.[33,34] As part of this effort, Double Dutch would be extended to allow users to import component data from other applications built on the Clotho software environment[35] and from repositories supporting the Synthetic Biology Open Language (SBOL),[36−38] such as JBEI-ICE,[39] the iGEM Registry,[40] and the SBOL Stack.[41]

Furthermore, the core Double Dutch algorithms could be integrated with a machine learning based-approach. Data from testing designs would be fed to learning algorithms to select the factor and level requirements for subsequent factorial designs. Double Dutch could then be iteratively run with these requirements to explore increasingly narrow library design spaces. Learned data could be captured for future users, and the reduced design spaces would allow for faster run times and more exhaustive explorations.

Finally, while Double Dutch is currently used to design libraries of biosynthetic pathway variants, it could also

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

potentially be extended to design libraries of genetic circuit variants. In order to make this use case possible, Double Dutch would need to be extended to take as inputs an abstract genetic regulatory network[42] and a database of regulatory relationships, in addition to a factorial design and DNA component library. If users were given the ability to modify Double Dutch's grammar, then it would enable the automated design of libraries containing transcriptional units that include DNA components with novel roles.

■ ACCESS

Double Dutch can be accessed via its Web site, www.doubledutchcad.org. The source JavaScript and HTML for the Double Dutch Web site and Pathway Designer web application (the latter of which implements Algorithms 1−7) are available via the GitHub Web site at www.github.com/CIDARLAB/doubledutch. The Double Dutch Web site is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License, and the Double Dutch Pathway Designer web application is licensed under a BSD 3-Clause License.

■ ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge on the ACS Publications website at DOI: 10.1021/acssynbio.5b00232.

PDF file of the formal grammar used by Double Dutch; CSV files for the DNA component library (promoters and terminators) used in this article and the level module assignments to the full factorial designs described in Performance. Also included in a directory named "plackett_burman" are CSV files for the example on modifying the weights of the Double Dutch cost function, namely, the four-factor Plackett−Burman design, the level module assignments to this design for three different sets of weights, and the resulting libraries of pathway variants (ZIP)

■ AUTHOR INFORMATION

Corresponding Author

*E-mail: nicholasroehner@gmail.com.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

This material is based on work supported by U.S. Defense Advanced Research Projects Agency (DARPA) Living Foundries award HR0011-15-C-0084. The authors would like to thank the members of the MIT-Broad Foundry for their helpful discussions in relation to this project.

■ REFERENCES

(1) Kim, T. Y., Park, J. M., Kim, H. U., Cho, K. M., and Lee, S. Y. (2015) Design of homo-organic acid producing strains using multi-objective optimization. *Metab. Eng. 28*, 63−73.

(2) Flowers, D., Thompson, R. A., Birdwell, D., Wang, T., and Trinh, C. T. (2013) SMET: Systematic Multiple Enzyme Targeting - a method to rationally design optimal strains for target chemical overproduction. *Biotechnol. J. 8*, 605−618.

(3) Wang, H. H., Isaacs, F. J., Carr, P. A., Sun, Z. Z., Xu, G., Forest, C. R., and Church, G. M. (2009) Programming cells by multiplex genome engineering and accelerated evolution. *Nature 460*, 894−898.

(4) Alper, H., Moxley, J., Nevoigt, E., Fink, G. R., and Stephanopoulos, G. (2006) Engineering yeast transcription machinery for improved ethanol tolerance and production. *Science 314*, 1565−1568.

(5) Santos, C. N. S., Xiao, W., and Stephanopoulos, G. (2012) Rational, combinatorial, and genomic approaches for engineering L-tyrosine production in Escherichia coli. *Proc. Natl. Acad. Sci. U. S. A. 109*, 13538−13543.

(6) Zou, R., Zhou, K., Stephanopoulos, G., and Too, H. P. (2013) Combinatorial engineering of 1-deoxy-D-xylulose 5-phosphate pathway using Cross-Lapping In Vitro Assembly (CLIVA) method. *PLoS One 8*, e79557.

(7) Nowroozi, F. F., Baidoo, E. E. K., Ermakov, S., Redding-Johanson, A. M., Batth, T. S., Petzold, C. J., and Keasling, J. D. (2014) Metabolic pathway optimization using ribosome binding site variants and combinatorial gene assembly. *Appl. Microbiol. Biotechnol. 98*, 1567−1581.

(8) Farasat, I., Kushwaha, M., Collens, J., Easterbrook, M., Guido, M., and Salis, H. M. (2014) Efficient search, mapping, and optimization of multi-protein genetic systems in diverse bacteria. *Mol. Syst. Biol. 10*, 731.

(9) Smanski, M. J., Bhatia, S., Zhao, D., Park, Y., Woodruff, L. B. A., Giannoukos, G., Ciulla, D., Busby, M., Calderon, J., Nicol, R., Gordon, D. B., Densmore, D., and Voigt, C. A. (2014) Functional optimization of gene clusters by combinatorial design and assembly. *Nat. Biotechnol. 32*, 1241.

(10) Ng, C. Y., Farasat, I., Maranas, C. D., and Salis, H. M. (2015) Rational design of a synthetic Entner-Doudoroff pathway for improved and controllable NADPH regeneration. *Metab. Eng. 29*, 86.

(11) Zhou, H., Vonk, B., Roubos, J. A., Bovenberg, R. A. L., and Voigt, C. A. (2015) Algorithmic co-optimization of genetic constructs and growth conditions: application to 6-ACA, a potential nylon-6 precursor. *Nucleic Acids Res.*, 10560−10570.

(12) Cox, R. S., III, Surette, M. G., and Elowitz, M. B. (2007) Programming gene expression with combinatorial promoters. *Mol. Syst. Biol. 3*, 145.

(13) Davis, J. H., Rubin, A. J., and Sauer, R. T. (2011) Design, construction and characterization of a set of insulated bacterial promoters. *Nucleic Acids Res. 39*, 1131−1141.

(14) Blount, B. A., Weenink, T., Vasylechko, S., and Ellis, T. (2012) Rational diversification of a promoter providing fine-tuned expression and orthogonal regulation for synthetic biology. *PLoS One 7*, e33279.

(15) Alper, H., Fischer, C., Nevoigt, E., and Stephanopoulos, G. (2005) Tuning genetic control through promoter engineering. *Proc. Natl. Acad. Sci. U. S. A. 102*, 12678−12683.

(16) Salis, H. M., Mirsky, E. A., and Voigt, C. A. (2009) Automated design of synthetic ribosome binding sites to control protein expression. *Nat. Biotechnol. 27*, 946−950.

(17) Na, D., and Lee, D. (2010) RBSDesigner: software for designing synthetic ribosome binding sites that yields a desired level of protein expression. *Bioinformatics 26*, 2633−2634.

(18) Chen, Y.-J., Liu, P., Nielsen, A. A. K., Brophy, J. A. N., Clancy, K., Peterson, T., and Voigt, C. A. (2013) Characterization of 582 natural and synthetic terminators and quantification of their design constraints. *Nat. Methods 10*, 659−664.

(19) Kosuri, S., Goodman, D. B., Cambray, G., Mutalik, V. K., Gao, Y., Arkin, A. P., Endy, D., and Church, G. M. (2013) Composability of regulatory sequences controlling transcription and translation in Escherichia coli. *Proc. Natl. Acad. Sci. U. S. A. 110*, 14024−14029.

(20) Weber, E., Engler, C., Gruetzner, R., Werner, S., and Marillonnet, S. (2011) A modular cloning system for standardized assembly of multigene constructs. *PLoS One 6*, e16765.

(21) Coussement, P., Maertens, J., Beauprez, J., Van Bellegem, W., and De Mey, M. (2014) One step DNA assembly for combinatorial metabolic engineering. *Metab. Eng. 23*, 70−77.

(22) Khuri, A. I., and Mukhopadhyay, S. (2010) Response surface methodology. *WIREs Comp. Stats 2*, 128−149.

(23) Shen, P., and Huang, H. V. (1986) Homologous recombination in Escherichia Coli: dependence on substrate length and homology. *Genetics 112*, 441−457.

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517

(24) Hua, S.-b., Qiu, M., Chan, E., Zhu, L., and Luo, Y. (1997) Minimum length of sequence homology required for in vivo cloning by homologous recombination in yeast. *Plasmid 38*, 91−96.
(25) Hartigan, J. A., and Wong, M. A. (1979) Algorithm AS 136: a k-means clustering algorithm. *Applied Statistics 28*, 100−108.
(26) Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983) Optimization by simulated annealing. *Science 220*, 671−680.
(27) Quinn, J. Y., et al. (2015) SBOL Visual: A Graphical Language for Genetic Designs. *PLoS Biol. 13*, e1002310.
(28) Plackett, R. L., and Burman, J. P. (1946) The design of optimum multifactorial experiments. *Biometrika 33*, 305−325.
(29) Box, G. E. P., and Behnken, D. W. (1960) Some new three level designs for the study of quantitative variables. *Technometrics 2*, 455−475.
(30) Box, G. E. P., and Hunter, J. S. (1961) The 2^{k-p} fractional factorial designs part I. *Technometrics 3*, 311−351.
(31) Sall, J., Lehman, A., Stephens, M. L., and Creighton, L. (2012) *JMP Start Statistics: A Guide to Statistics and Data Analysis Using JMP*, SAS Institute.
(32) Galanie, S., Thodey, K., Trenchard, I. J., Interrante, M. F., and Smolke, C. D. (2015) Complete biosynthesis of opioids in yeast. *Science 349*, 1095−1100.
(33) Bilitchenko, L., Liu, A., Cheung, S., Weeding, E., Xia, B., Leguia, M., Anderson, J. C., and Densmore, D. (2011) Eugene - a domain specific language for specifyng and constraining synthetic biological parts, devices, and systems. *PLoS One 6*, e18882.
(34) Oberortner, E., and Densmore, D. (2015) Web-based software tool for constraint-based design specification of synthetic biological systems. *ACS Synth. Biol. 4*, 757−760.
(35) Xia, B., Bhatia, S., Bubenheim, B., Dadgar, M., Densmore, D., and Anderson, J. C. (2011) Developer's and user's guide to Clotho v2.0: a software platform for the creation of synthetic biological systems. *Methods Enzymol. 498*, 97−135.
(36) Galdzicki, M., et al. (2014) The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology. *Nat. Biotechnol. 32*, 545−550.
(37) Roehner, N., Oberortner, E., Pocock, M., Beal, J., Clancy, K., Madsen, C., Misirli, G., Wipat, A., Sauro, H., and Myers, C. J. (2015) Proposed data model for the next version of the Synthetic Biology Open Language. *ACS Synth. Biol. 4*, 57−71.
(38) Bartley, B., Beal, J., Clancy, K., Misirli, G., Roehner, N., Oberortner, E., Pocock, M., Bissell, M., Madsen, C., Nguyen, T., Zhang, Z., Gennari, J. H., Myers, C., Wipat, A., and Sauro, H. (2015) Synthetic Biology Open Language (SBOL) version 2.0.0. *J. Integr. Bioinform. 12*, 272.
(39) Ham, T. S., Dmytriv, Z., Plahar, H., Chen, J., Hillson, N. J., and Keasling, J. D. (2012) Design, implementation and practice of JBEI-ICE: an open source biological part registry platform and tools. *Nucleic Acids Res. 40*, e141.
(40) Smolke, C. D. (2009) Building outside of the box: iGEM and the BioBricks Foundation. *Nat. Biotechnol. 27*, 1099−1102.
(41) Madsen, C., Misirli, G., Pocock, M., Hallinan, J., and Wipat, A. (2015) SBOL Stack: The One-Stop-Shop for Storing and Publishing Synthetic Biology Designs, *7th International Workshop on Bio-Design Automation*, Seattle, WA.
(42) 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.

DOI: 10.1021/acssynbio.5b00232
ACS Synth. Biol. 2016, 5, 507−517