Welcome to the BDAthlon at IWBDA 2018! Before your team gets started hacking, please take 10-15 minutes to read through the BDAthlon rules and descriptions of the challenges.

Submission Instructions

All teams' solutions must be submitted to the appropriate BDAthlon repository on GitHub by 8:00 AM on Aug. 1st. For example, if team "Penguin Style" is submitting a solution to Challenge 1, they must upload their solution's code and associated materials to the repository 2018-PenguinStyle-1. In addition to code, each solution must include a README file that contains a description of the project and instructions on how to build and run it. Finally, each solution must include an Open Source Initiative approved license (see Intellectual Property).

Software Development

Teams are permitted to use any software development tool, game engine, IDE, or programming language that they prefer. All code submitted as part of a solution must be written during the BDAthlon (8 AM on July 31st to 8 am on Aug. 1st). The only exception is that teams are permitted to use any publicly available library, API, or framework, provided that it is subject to an Open Source Initiative approved license (see Intellectual Property). Teams must document in their project's README file all external libraries, APIs, and frameworks used in their project.

Judging

All solutions will be judged by a subset of the IWBDA organizing committee and scored according to the general criteria below, in addition to the more specific functional criteria listed for each challenge.

1. Originality - Does the software have unique features?
2. User Experience - Is the software intuitive to use?
3. Technical Proficiency - Does the software reflect good engineering practices (e.g. modularity)?

Intellectual Property[1]

By submitting a Solution in this Contest, entrants warrant and represent that their Solution, including the programming and related material, is open source and is released subject to the Open Source Initiative approved licenses (https://opensource.org/licenses) and not subject to the proprietary rights of any person or entity.

Warranty[1]

Participants warrant that their Solutions are their own original work and, as such, they are the sole and exclusive owner and rights holder of the submitted Solution and that they have the right to submit the Solution in the Contest and grant all required licenses. Each entrant agrees not to submit any Solution that (1) infringes any third party proprietary rights, intellectual property rights, industrial property rights, personal or moral rights or any other rights, including without limitation, copyright, trademark, patent, trade secret, privacy, publicity or confidentiality obligations; or (2) otherwise violates the applicable state, federal, local or provincial law.

[1] Adapted from the rules of the Google I/O Hackathon Challenge

1. Genetic Circuit Design Automation

Since the inception of synthetic biology as a field nearly two decades ago, genetic circuits have been the focus of significant research interest, beginning with transcriptional logic circuits such as the genetic toggle switch and repressilator and more recently other types of circuits implemented with recombinase logic and deactivated CRISPR-Cas9. These circuits can provide programmable control over biological functions such as sensing and responding to the environment, both of which are key requirements for engineering biological systems that can operate under a range of complex conditions. One of the longstanding goals of biodesign automation research has been to develop software tools that are capable of automatically designing genetic circuits to meet a logical specification or other type of specification of biological behavior. One prominent example of such a tool that is backed by a large amount of experimental data is Cello [1], which was recently developed for the design of transcriptional genetic logic circuits from a library of sensors and genetic NOT gates.

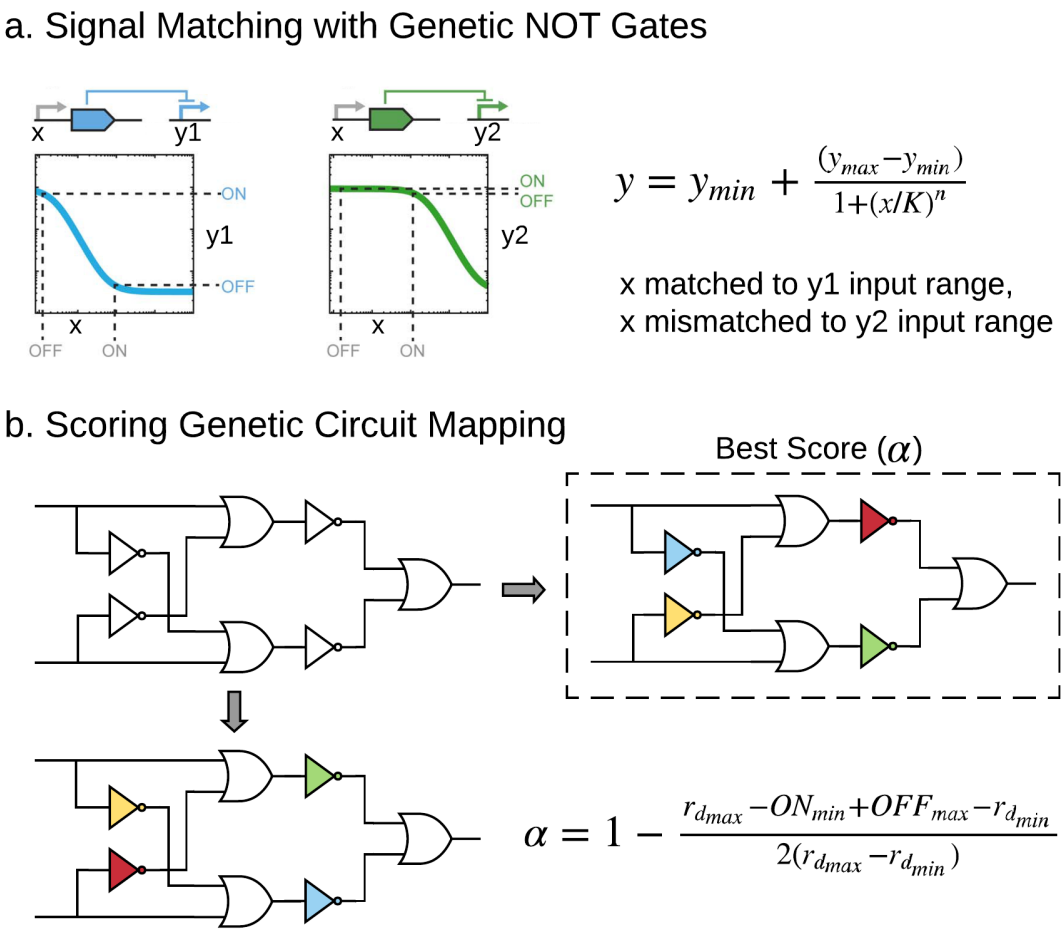

Figure 1: **Genetic technology mapping.** (a) Two genetic NOT gates are shown with their response functions. When the same OFF and ON inputs x are applied to both gates, output y1 changes but output y2 does not. Unlike electronic logic gates, genetic logic gates must have their input signals matched to their response functions for optimal function. Response function parameters include min and max outputs y_{min} and y_{max}, cooperativity/curve slope n, and input for half max output K. (b) Two alternate mappings are shown for the netlist of a genetic circuit with XOR functionality. Of the two mappings, one is selected as having the best score α. α captures how well a mapped circuit realizes its specified Boolean function and by extension how well its gates' response functions are matched to their input signals. The parameters for computing α include the min and max outputs $r_{d_{min}}$ and $r_{d_{max}}$ among available gates, the smallest circuit output intended to be "ON" ON_{min}, and the largest circuit output intended to be "OFF" OFF_{max}.

The goal of this challenge is to develop an algorithm that automates one of the final steps of the Cello workflow: selecting genetic logic gates from a library to implement or "color" a circuit netlist, a process also known as genetic technology mapping. Figure 1 illustrates signal matching, a key consideration for genetic technology mapping, and describes how a metric α [2] can be used to score a circuit mapping and evaluate how well its signals are matched.

Given data from one of the three netlist files (majority_netlist, rule_30_netlist, and multiplexer_netlist.json) and the genetic gate library file (genetic_gate_library.json) that we have provided to you, your algorithm must produce a circuit mapping that optimizes α by mapping each NOT gate in the netlist to a library gate. OR gates can be omitted in this challenge because our gate library consists solely of transcriptional NOT gates with outputs that are assumed to combine linearly when applied as the inputs to another NOT gate. The software that implements your algorithm must export a JSON file formatted the same as the example majority_mapping.json, where each NOT gate has been given a "mapping" property that specifies which library gate has been mapped to it. You can compute α for different circuit mappings using the Python library that we have provided and compare them, but your software must designate a single circuit mapping as its optimized output each time it is executed. It is okay if your software designates a different single circuit mapping each time it is executed, for example if it includes random selection of library gates.

There is one additional constraint to which your circuit mapping must adhere. Namely, your circuit mapping must not include two gates whose "factors" properties list the same transcription factor. Because all the gates of a genetic circuit and their transcription factors co-exist in the same liquid solution, if two gates use the same transcription factor, it generally results in unintended regulation between genes and therefore unintended connections between gates in the genetic circuit. This phenomenon is referred to as crosstalk in the genetic circuit literature.

Your algorithm and implementing software will be judged primarily on the basis of the following criteria:

Feature	**Criteria**
Quality	The best algorithms will on average produce circuit mappings with a higher α than those produced by other algorithms.
Efficiency	The best algorithms will on average produce circuit mappings in a shorter amount of time than that taken by other algorithms.
Versatility	The best algorithms will efficiently produce circuit mappings of high quality for all three circuit netlists provided for this challenge (majority, rule 30, and multiplexer).
Reporting	The best software will provide informative statistics related to its performance when requested, such as total execution time and statistics on α.

2. Genetic Circuit Tuning

Even after genetic technology mapping, it may be discovered that a genetic circuit design does not satisfy its original functional specification. In such cases, it may be necessary to expand one's library of available genetic gates by engineering the very parts that make these gates up, including promoters, ribosome binding sites (RBSs), and protein coding sequences. In general, promoter and and RBS engineering are less complex and easier to accomplish than protein engineering, but they only allow us to modify certain

parameters in the response functions of transcriptional genetic NOT gates such as those used by Cello. For example, increasing or decreasing the strength of a NOT gate's promoter increases or decreases both y_{min} and y_{max}, respectively. Increasing the strength of a NOT gate's RBS, however, decreases K, while decreasing its strength increases K.

The goal of this challenge is to develop an algorithm that, given a circuit mapping, determines which gate promoters and RBSs to tune and by how much to tune them to improve α. The algorithm, however, should also seek to minimize the total number and magnitude of changes made to improve α, since solutions that require less DNA engineering are more desirable from a cost perspective. Your algorithm does not need to determine specific changes to be made to a DNA sequence. Instead, you can assume that multiplying a NOT gate's promoter strength by a factor of z multiplies both its y_{min} and y_{max} parameters by factor of z, and that multiplying a NOT gate's RBS strength by a factor of z divides its K parameter by a factor of z.

Given data from one of the three example mapping files (majority_mapping, rule_30_mapping, and multiplexer_mapping.json) and the genetic gate library file (genetic_gate_library.json) that we have provided to you, your algorithm must produce a list of changes to be made to the promoters and RBSs of gates in the gate library to optimize α for the circuit mapping. The software that implements your algorithm must export this list of changes as a JSON file formatted the same as the example gate_tuning.json. You can compute α for different circuit tunings using the Python library that we have provided and compare them, but your software must designate a single circuit tuning as its optimized output each time it is executed. It is okay if your software designates a different single circuit tuning each time it is executed, for example if it includes random selection of gate parameters to modify.

Your algorithm and implementing software will be judged primarily on the basis of the following criteria:

Feature	**Criteria**
Quality	The best algorithms will on average produce circuit tunings with a higher α than those produced by other algorithms and with fewer and smaller gate parameter changes than other algorithms.
Efficiency	The best algorithms will on average produce circuit tunings in a shorter amount of time than that taken by other algorithms.
Versatility	The best algorithms will efficiently produce circuit tunings of high quality for all three example circuit mappings provided for this challenge (majority, rule 30, and multiplexer).
Reporting	The best software will provide informative statistics related to its performance when requested, such as total execution time and statistics on α and number of gate parameter changes made.

References

[1] Alec A. K. Nielsen et al., *Genetic circuit design automation*, Science, vol. 352, 2016.

[2] P. Vaidyanathan et al., *A framework for genetic logic synthesis*, Proceedings of the IEEE, vol. 103, 2015.