1 Diversity score

We calculated a diversity score equal to the average phenotypic distance between each pair of viable individuals in the final population. For each function, this diversity is normalized by dividing it by the largest diversity measured for a run on that function. Normalization is performed because some functions have more potential for diversity than others.

2 Finding canonical devices

Simplicity search is a Truth-Seq-Er run with an additional objective: minimize the number of H-segments that bind to each other. In exploratory work (not shown), this new objective considered *all* possible H-segments interactions. However, this did not produce acceptable results. Results were significantly improved when only OBS and negator *self-interactions* were considered. It appears that these interactions are largely *non-essential* unlike the interactions between two *different* segments. These self-interactions are represented by a new fitness term, f^{SI}, which is equal to sum of the diagonal entries of the SPMMS.

In addition to the self-interactions fitness term, three other minor changes are made to Truth-Seq-Er. Instead of initializing a random population, simplicity search begins with the *final population* of a Truth-Seq-Er run. Furthermore, in order to compensate for a new fitness term being added, the two performance scores (f^{ON}, f^{OFF}) are averaged into a single *switch* score: f^{sw}. Finally, the viability nullification parameters are set more aggressively. The fitness parameters of simplicity search are shown in table 1. For each 3-input function, the canonical device is the viable individual with the least number of segment interactions from the final simplified population.

Number of generations	200
Population size	300
Mutation rate	4
Objective scores	Switch, self-interactions, novelty
Viability nullification?	Yes
(Start, End) threshold values	(0.9, 0.95)
Threshold Breakpoint (generation, value)	N/A
Novelty neighborhood size	30

Table 1 Simplicity search fitness parameters.

3 Canonical devices for functions f-7-3I, f-25-3I, f-27-3I, f-129-3I, f-135-3I

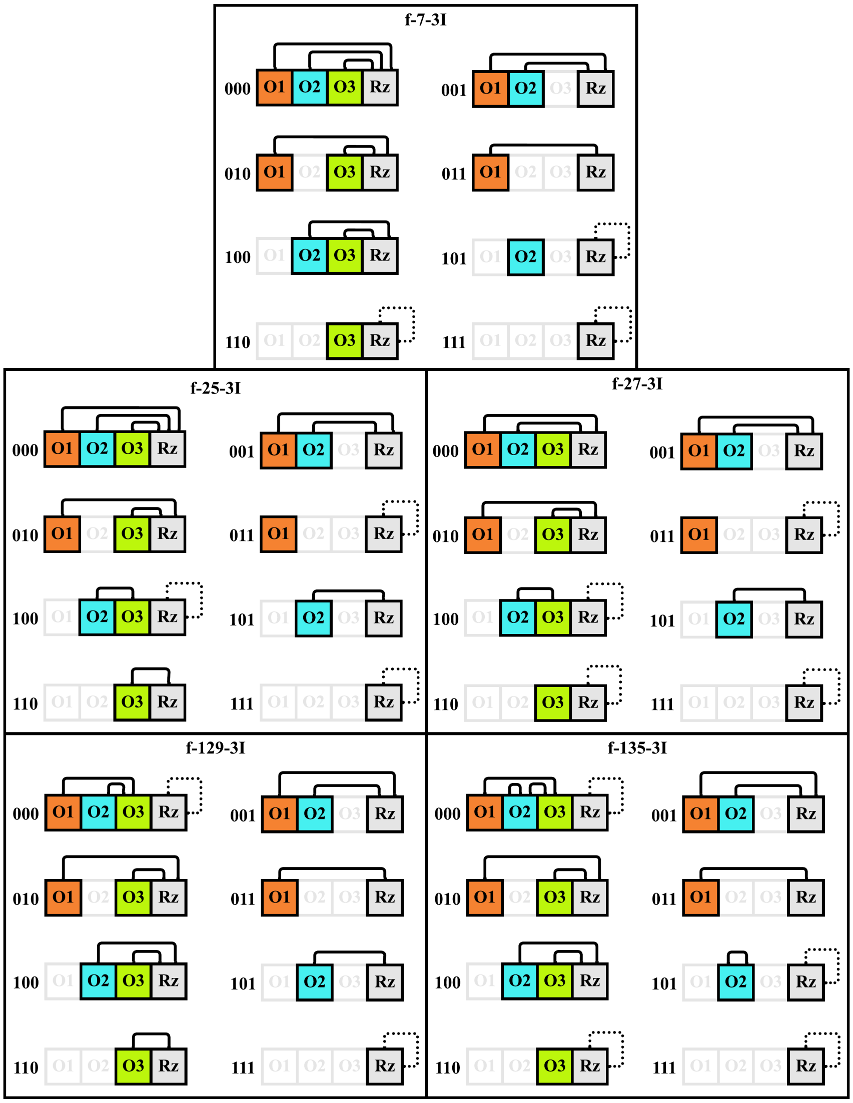

Fig. 1 Segment structures of canonical devices.

4 Simulation

In the main text, we generated segment structures from a given mechanism graph by hand. Here we present an algorithm called *simulation* which automates the process. It uses exhaustive search to calculate the stability of all candidate structures and it returns the most stable one. The pseudocode for simulation is shown in Algorithm 1.

Algorithm 1: Simulate mechanism graph

```
Inputs: ($E$, $V$, $B$) // The edges, nodes, and bundles of the mechanism graph
        $W$ // A dictionary storing the weight assigned to each edge
        $V_O$ // The set of occupied nodes (OBSs bound to their respective input)
// Generate the set of candidate structures
$edges \leftarrow$ the set of all $e \in E$ such that $e$ is not incident on any $node \in V_O$
$provisional\_candidates \leftarrow$ powerset($edges$) // All possible subsets of $edges$
// Remove invalid structures from the set of candidates
$candidates \leftarrow$ copy $provisional\_candidates$
for each $c$ in $provisional\_candidates$ do
    for each $node$ in $V$ do
        $node\_bundles \leftarrow \emptyset$
        $node\_edges \leftarrow$ the set of all edges $\in edges$ that are incident on $node$
        // If multiple edges are incident on a node, they must share a bundle
        // If at least one node violates this condition, the structure is invalid
        if $|node\_edges| > 1$ then
            for each $edge$ in $node\_edges$ do
                $edge\_bundles \leftarrow$ all bundles that include edge
                $node\_bundles \leftarrow node\_bundles \cup \{edge\_bundles\}$
            end for
            $common\_bundles \leftarrow \bigcap node\_bundles$
            if $|common\_bundles| == 0$ then
                remove $c$ from $candidates$
                break
            end if
        end if
    end for
end for
// Calculate the stability of each candidate
$stabilities \leftarrow$ empty list
for each $c$ in $candidates$ do
    $stability = \sum_{e \in c} W[e]$ // Sum the edge weights
    append $stability$ to $stabilities$
end for
// Select the candidate with the highest stability
$indices \leftarrow$ argsort $stabilities$ (by descending value)
$L \leftarrow$ length of $indices$
$index\_1 \leftarrow indices[0]$ //index of candidate with the highest stability
$max\_stability\_structure \leftarrow candidates[index\_1]$
// Determine whether the highest stability structure is unique
// This is necessary for mechanism extraction (discussed in Section 5)
$bool\_degenerate \leftarrow 0$
// If there was more than one candidate
if $L > 1$ then
    $index\_2 \leftarrow indices[1]$ //Index of the candidate with the second highest stability
    if $stabilities[index\_1] == stabilities[index\_2]$ then
        $bool\_degenerate \leftarrow 1$ // The highest stability structure is not unique
    end if
end if
return ($max\_stability\_structure$, $stabilities[index\_1]$, $bool\_degenerate$ )
```

5 Mechanism extraction

Mechanism extraction is an algorithm that takes as input a list of *target* segment structures and it outputs a mechanism that can *reproduce* these structures *without error* when it is simulated. Mechanism extraction consists of two steps. First, the target segment structures are *merged* into an unweighted graph. Then, the edges of this graph are *assigned weights* using an iterative procedure similar to simulated annealing. A full example of mechanism extraction is shown in Figure 2.

5.1 Merging

The *unweighted* mechanism graph represents TO segments that interact with each other in at least one state. The unweighted mechanism graph is equal to the union of the target TO structures. In addition, certain edges of the unweighted mechanism may be grouped together into sets called bundles. Edges are part of the same bundle if they are incident on the *same node* in the *same segment structure*. Merging is detailed in Algorithm 2. An example unweighted mechanism is shown in Figure 2 a).

5.2 Weight assignment

Once the unweighted mechanism graph has been obtained, weights are assigned to it. In general, mechanism graphs having identical edges, nodes, and bundles, but different weights, will produce different segment structures when simulated. The *weight assignment* algorithm searches for weights such that the mechanism reproduces the target segment structures. Each edge of the unweighted mechanism is initially assigned a *random* weight, and these weights are *mutated* through an iterative process until a solution is found. The weights at iteration i are denoted by $w(i)$. Each iteration, the current mechanism graph is simulated to generate a list of *predicted* segment structures. The *similarity* between the predicted and target segment structures is assessed, and an error is calculated. The higher the similarity, the lower the error. If the error of $w(i)$ is smaller than or equal to the error of $w(i-1)$ (i.e. the current iteration is an improvement over the previous one), then $w(i)$ is *always accepted.* Otherwise, $w(i)$ is only accepted with a certain *probability.* As explained in the main text, the segment structure of each state is the candidate structure with the highest stability. Note that for certain weight assignments, multiple candidate structures of a state may have maximum stability. In this case, the candidate mechanism graph is considered *invalid* and it is assigned a very high error as a penalty. The reason for this is that we want the segment structures generated by simulation to be non-ambiguous. Weight assignment is detailed in Algorithms 3 and 4, and is illustrated in Figure 2 b). Note that the graphs generated by mechanism extraction are not unique. Two mechanism graphs with

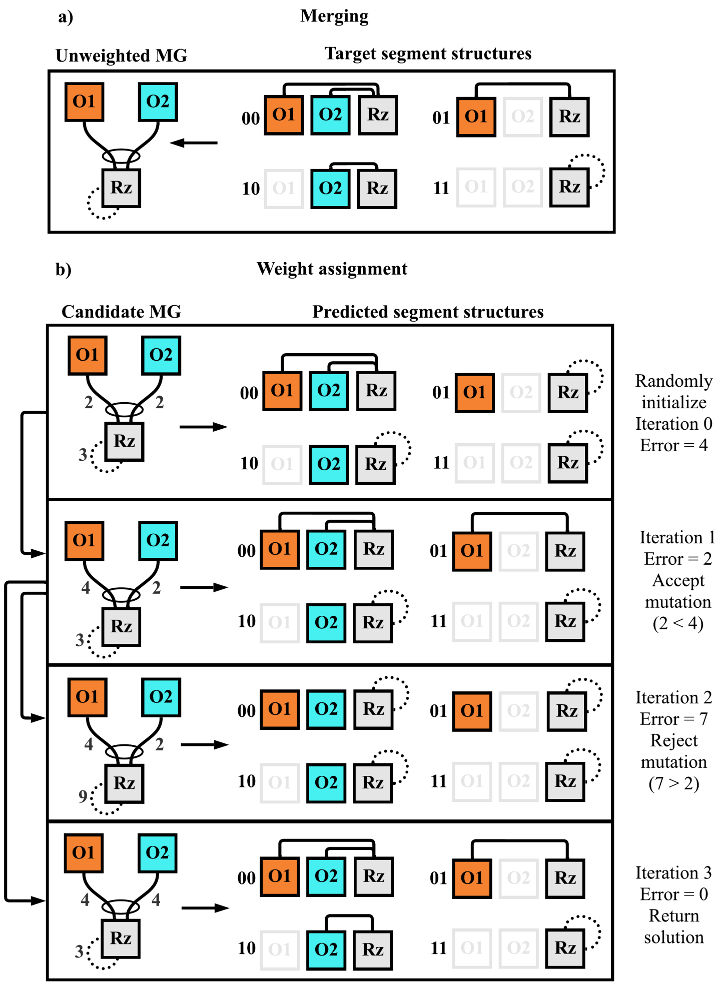

Fig. 2 a) The target segment structures are merged into an unweighted mechanism graph (UMG). The UMG has the same nodes as the target segment structures. Its edges are equal to the union of the edge sets of the target segment structures. Since O1 and O2 are both incident on Rz in the same state (00), they are grouped into a bundle. b) The UMG is assigned a set of random weights. Each iteration, the mechanism graph is simulated with its current weights. An error is calculated based on how dissimilar the target and predicted segment structures are. Then, a random weight of the mechanism graph is mutated. If this mutation results an error reduction, it is always accepted. Otherwise, it is only accepted with a certain probability. If the mutation is rejected, a new one is applied to the parent graph. If the target and predicted structures match, the current candidate mechanism graph is returned as the solution. List of abbreviations: O_n (OBS n), Rz (ribozyme).

Algorithm 2: Merge target segment structures

```
Inputs: $segment\_structures$ // There is one segment structure per state
$E \leftarrow \emptyset$
$V \leftarrow \emptyset$
$provisional\_bundles \leftarrow \emptyset$
for each $structure$ in $segment\_structures$ do
    $(e, v) \leftarrow$ (edges, nodes) of $structure$
    //Take the union of the segment structures
    $E \leftarrow E \cup e$
    $V \leftarrow V \cup v$
    for each $node$ in $v$ do
        $node\_edges \leftarrow$ the set of all edges in $e$ that are incident on $node$
        if $|node\_edges| > 1$ then
            $bundle \leftarrow node\_edges$
            $provisional\_bundles \leftarrow provisional\_bundles \cup bundle$
        end if
    end for
end for
// Discard any bundles that are strict subsets of other bundles
$bundles \leftarrow$ copy $provisional\_bundles$
for each $bundle\_i$ in $provisional\_bundles$ do
    for each $bundle\_j$ in $provisional\_bundles$ do
        if $bundle\_i \subset bundle\_j$ then
            remove $bundle\_i$ from $bundles$
            break
        end if
    end for
end for
return $(E, V, bundles)$ // Unweighted mechanism graph
```

different weights and/or bundles can generate the same segment structures when simulated.

5.3 Experimental setup

We performed mechanism extraction for each 3-input NPN function. For each function, the input to the mechanism extraction algorithm was the segment structures of that function's canonical ribogate. Before running the algorihtm, we made slight changes to the second segment structures of two functions. For f-135-3I, we removed the OBS2-OBS2 self-loop in states 000 and 101. For f-27-3I, we added an edge an OBS3-Rz edge in state 000. None of these modifications affect the output state of the ribogate and they result in cleaner mechanism graphs.

Algorithm 3: Assign mechanism weights

```
Inputs: ($E$, $V$, $B$) // The edges, nodes, and bundles of the mechanism graph
        $target\_structures$ // The segment structures we wish for the mechanism
                                graph to reproduce when simulated
// Score a random initial weight assignment
$min\_weight \leftarrow 1$
$max\_weight \leftarrow 8$
$prev\_weights \leftarrow$ empty dictionary
for each $edge$ in $E$ do
    $w \leftarrow$ random integer between $min\_weight$ and $max\_weight$
    $prev\_weights[edge] \leftarrow w$
end for
$prev\_error \leftarrow score\_mechanism\_graph(E, V, B, prev\_weights, target\_structures)$
// Iterate until a solution is found or the max # of iterations is exceeded
$num\_iterations \leftarrow 2000$
for $i$ from 0 to $num\_iterations - 1$ do
    $weights \leftarrow prev\_weights$
    // Mutate weight
    $edge \leftarrow$ random edge from $E$
    $weights[edge] \leftarrow$ random integer between $min\_weight$ and $max\_weight$
    // Score mechanism graph
    $error \leftarrow score\_mechanism\_graph(E, V, B, weights, target\_structures)$
    // Accept or reject mutation
    if $error == 0$ then
        return $(E, V, B, weights)$ // Return the complete mechanism graph
    else if $error <= prev\_error$ then
        $prev\_error \leftarrow error$
        $prev\_weights \leftarrow weights$
    else
        $jump\_probability \leftarrow exp(prev\_error - error)$
        $jump\_sample \leftarrow$ random floating point number between 0 and 1
        if $jump\_probability > jump\_sample$ then
            $prev\_error \leftarrow error$
            $prev\_weights \leftarrow weights$
        end if
end for
return NULL // No solution
```

5.4 Manual post-processing

The mechanism extraction algorithm does not natively handle partial bundles. Instead, some graphs will have partially overlapping bundles. In this step, we manually remove these bundles and replace them with a partial bundle that allows a maximum of two concurrent edges. Specifically, for f-127 and f-135, we remove the {OBS1-RZ, OBS2-Rz}, {OBS1-Rz, OBS3-Rz}, and {OBS2=Rz, OBS3-Rz} bundles and replace them with a {OBS1-Rz, OBS2-Rz, OBS3-Rz} (max 2) partial bundle.

Algorithm 4: Score mechanism graph

```
Inputs: ($E$, $V$, $B$) // The edges, nodes, and bundles of the mechanism graph
        $W$ // A dictionary storing the weight assigned to each edge
        $target\_structures$ // The segment structures we wish for the mechanism
                                graph to reproduce when simulated
$num\_states \leftarrow$ length of $target\_structures$
$occupied\_nodes\_all \leftarrow$ the set of occupied nodes for each state
$error \leftarrow 0$
for $i$ from 0 to $num\_states$ - 1 do
    $occupied\_nodes \leftarrow occupied\_nodes\_all[i]$
    $predicted\_structure, stability, bool\_degenerate \leftarrow$
      $simulate\_mechanism\_graph(E, V, B, W, occupied\_nodes)$
    if $bool\_degenerate == 1$ then
        // Severely penalize mechanism graphs that generate non-unique structures
        $error \leftarrow 1000$
        break
    else
        $E_p \leftarrow$ edges of predicted_structure
        $E_t \leftarrow$ edges of target_structures[i]
        // The error is equal to the # of edges that are in the predicted,
        // but not the target structure (and vice versa)
        $state\_error \leftarrow |(E_p \cup E_t)\backslash(E_p \cap E_t)|$
        $error \leftarrow error + state\_error$
    end if
end for
return $error$
```

6 Linear inseparability and OBS-OBS interactions

In this analysis, we consider *base* structures that are in competition with each other. In each state, the candidate segment structures are substructures of these base structures. The specific structure of a candidate (and by extension its stability) depends on which edges are available, which in turn depends on which OBSs are unoccupied by inputs. This allows us to express the stability of each candidate in terms of the input state.

In the case of family 1, two base structures compete with each other: 1) the three OBS-Rz edges and 2) the Rz-Rz edge. These are illustrated in Figure 3 b). The stabilities of these two base structures are expressed by Equations 4 and 5, respectively:

$$i'_1 w_{O_1 R} + i'_2 w_{O_2 R} + i'_3 w_{O_3 R} \tag{1}$$

$$w_{RR} \tag{2}$$

The i'_n term is equal to 0 when the n^{th} input is present and 1 when it is absent. Therefore, the $i'_n w_{O_n R}$ term indicates than an OBS-Rz edge only contributes to the stability when its corresponding input is absent. The ribogate is active if the candidate derived from the second base structure is more stable than the candidate derived from the first one. This occurs if the following inequality holds true:

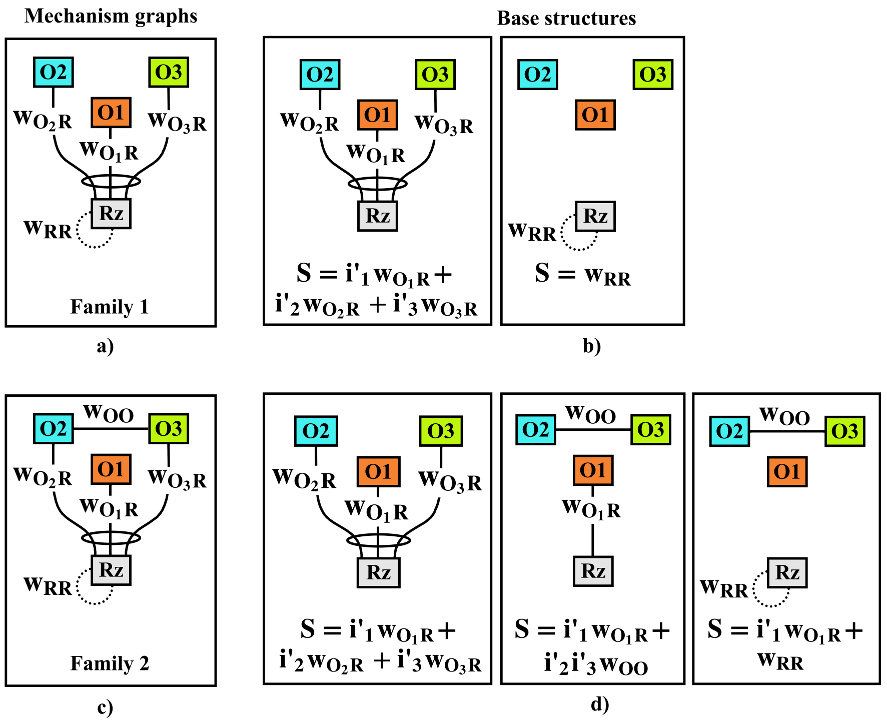
Mechanism graphs
Base structures
O2
O1
O3
w_{O_2R}
w_{O_1R}
w_{O_3R}
Rz
w_{RR}
Family 1
a)
$S = i'_1 w_{O_1R} + i'_2 w_{O_2R} + i'_3 w_{O_3R}$
$S = w_{RR}$
b)
w_{OO}
Family 2
c)
$S = i'_1 w_{O_1R} + i'_2 w_{O_2R} + i'_3 w_{O_3R}$
$S = i'_1 w_{O_1R} + i'_2 i'_3 w_{OO}$
$S = i'_1 w_{O_1R} + w_{RR}$
d)

Fig. 3 a) Mechanism graph of a family 1 ribogate. There are three OBS-Rz edges with weights w_{O_nR}. There is also an Rz-Rz edge with weight w_{RR}. c) Mechanism graph of a family 2 ribogate. It is the same as the family 1 graph, but it contains an addition OBS2-OBS3 segment with weight w_{OO}. b, d) In each state, the ribogate adopts a structure that is a subset of one of these base structures. The stability (S) of that structure is represented by the equation at the bottom of the box. Its value depends on which OBSs are unoccupied by inputs (i_n). Refer to main text for a detailed discussion on the role of stability values in determining linear separability.

$$i'_1 w_{O_1R} + i'_2 w_{O_2R} + i'_3 w_{O_3R} < w_{RR} \tag{3}$$

In the case of family 2, *three* base structures now compete with each other: 1) the three OBS-Rz edges, 2) the Rz-Rz edge and the OBS2-OBS3 edge, and 3) the OBS1-Rz edge and the OBS2-OBS3 edge. These are illustrated in Figure 3 d). Their stabilities are expressed by Equations 4, 5, and 6, respectively:

$$i'_1 w_{O_1R} + i'_2 w_{O_2R} + i'_3 w_{O_3R} \tag{4}$$
$$w_{RR} + i'_2 i'_3 w_{OO} \tag{5}$$
$$i'_1 w_{O_1R} + i'_2 i'_3 w_{OO} \tag{6}$$

The non-linear $i'_2 i'_3 w_{OO}$ term indicates that the OBS2-OBS3 edge only contributes to the stability when inputs 2 and 3 are both absent. The ribogate is active if the candidate derived from the second base structure is more stable than both the one derived from the first and third ones. This occurs if the following *two* inequalities holds true:

$$\begin{aligned} i'_1 w_{O_1R} + i'_2 w_{O_2R} + i'_3 w_{O_3R} &< w_{RR} + i'_2 i'_3 w_{OO} \\ i'_1 w_{O_1R} &< w_{RR} \end{aligned} \tag{7}$$

The above analysis shows that ribogates governed by the additive segment model *intrinsically* implement linear (Inequality 3) and non-linear (Inequalities 7) decision boundaries as they change shape in response to various inputs. Crucially, the factor that distinguishes these two types of decision boundaries is not the *number* of segments, but rather the *complexity* of the interactions between them. This enables a single ribogate to implement functions that would require multiple standard logic gates or artificial neurons.

7 Additive segment competition vs secondary structure prediction

In this work, we have used RNA secondary structure prediction to *design* ribogates and additive segment competition (ASC) to *analyze* them. We now take a moment to compare the two processes. ASC is an abstract version of RNA secondary structure folding and the two processes have many similarities. Both treat structure prediction as an optimization problem: folding uses dynamic programming [1] to find the secondary structure with the lowest free energy whereas ASC uses exhaustive search to find the segment structure with the highest stability. Both impose limits on the number of partners that their nodes may have. Finally, both apply constraints to certain nodes (OBS nucleotides in folding and OBS segments in ACS) to prevent them partnering with other nodes, thereby changing the optimal structure.

Despite these many similarities, there are some key differences. Segment structures generated by ACS may have nodes with self-loops and multiple partners. They are also much more concise: they have a maximum of 5 nodes whereas their corresponding secondary structures have more than 100. Critically, ACS models a structure's stability as a sum of *independent* edge weights. This allows us to easily reason about the effect of adding or removing certain edges. This is not the case in folding: the free energy of an RNA secondary structure is not simply the sum of the independent contributions of its base-pairs. Rather, it is the result of many *non-additive* effects such as base-pair stacking and loop entropy [2].

Despite its simplicity, ACS appears to be a plausible model of ribogate behavior, being able to reproduce the observed segment structures of each canonical ribogate with virtually no error. *Because* of its simplicity, we have seen that ribogates can be grouped into families, and that OBS-OBS or OBS-Negator interactions are required for linear inseparability. We have also seen that like artificial neurons, ribogates can implement different functions by changing their weights, but that unlike neurons, they can solve entirely new classes of problems by changing their interactions. These insights suggest that ACS is not only a useful model of rigobate behavior, but a potential new form of unconventional computing that requires further investigation.

References

1. J. S. McCaskill, Biopolymers 29, 1105 (1990).
2. D. H. Turner and D. H. Mathews, Nucleic Acids Res 38, D280 (2010).