EuroGP-2018. M. Castelli, L. Sekanina, M. Zhang *Eds.*, LNCS 10781, p220–236, Parma. 4-6 April 2018 Springer. doi:10.1007/978-3-319-77553-1_14 *Preprint*

Evolving Better RNAfold Structure Prediction

William B. Langdon and Justyna Petke and Ronny Lorenz

CREST, Computer Science, UCL, London, WC1E 6BT, UK
Institute for Theoretical Chemistry, University of Vienna, 1090 Vienna, Austria

Abstract. Grow and graft genetic programming (GGGP) evolves more than 50000 parameters in a state-of-the-art C program to make functional source code changes which give more accurate predictions of how RNA molecules fold up. Genetic improvement updates 29% of the dynamic programming free energy model parameters. In most cases (50.3%) GI gives better results on 4655 known secondary structures from RNA_STRAND (29.0% are worse and 20.7% are unchanged). Indeed it also does better than parameters recommended by Andronescu, M., et al.: Bioinformatics **23**(13) (2007) i19–i28.

Keywords genetic improvement, genetic algorithms, genetic programming, software engineering, SBSE, software maintenance of empirical constants, Bioinformatics, local search, genomic and phenotypic Tabu restrictions, genetic repair

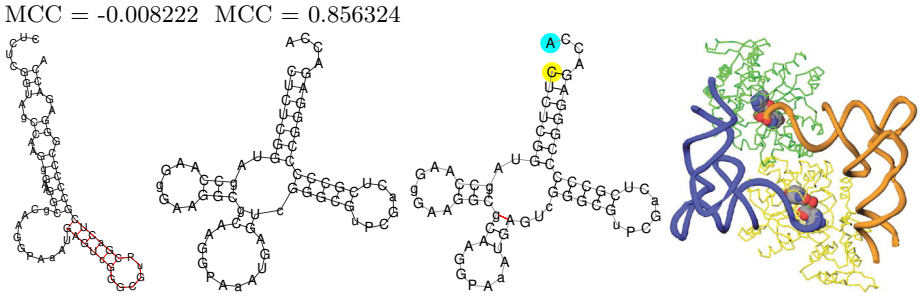

Fig. 1. 1), 2) and 3) are secondary structures (i.e. folding patterns) for RNA molecule PDB_01001. 1) Prediction made original RNAfold does not match well true structure 3. For example the highlighted hairpin loop (red) is not in 3. 2) Prediction made with parameter changes given in Section 3.3. 3) True structure. 4) Three dimensional structure. Two (blue, orange) RNA molecules in a Yeast protein complex [1, Fig 2. A]. (MCC explained on page 7.)

1 Background: RNA, Genetic Improvement, RNAfold

The central dogma of biology [2] is essentially about the flow of information in all forms of life. In its simple form it says that this fundamental information is transcribed from DNA into messenger RNA, which in turn is translated into protein. Like DNA, RNA is a long chain biomolecule composed of 4 bases (A, C, G and U). An RNA molecule's sequence of bases is known as its primary structure. Much of the interesting biology occurs when RNA is a single strand (unlike the more stable double stranded DNA helix). Like DNA the four bases can form relatively weak temporary bonds with their complementary base. (E.g., C pairs with G, and A with U.) How an RNA chain folds up on itself to form these complementary pairing is known as its secondary structure (e.g. diagrams 1), 2) and 3) in Figure 1). The tertiary, three dimensional structure, in turn relies on the secondary structure. (See solid colour in diagram 4) in Figure 1 for two examples.) In people about $\frac{3}{4}$ of the DNA is transcribed into RNA but less than 3% is translated into protein. Other than conveying information for protein manufacture, there are some well known biological uses of RNA. E.g., enzymes which catalyse reactions between biomolecules. Also some transcribed RNA regulates gene expression. Much of the chemistry of biomolecules is governed by their three dimensional shape. These areas are relatively new, and this, and other uses of RNA, have sparked renewed interest in RNA and its structure.

While tertiary structure prediction for RNA is still in its infancy and is limited to very small molecules, the hierarchical nature of RNA folding allows one to infer most of an RNA molecule's function from its secondary structure. Computer programs have had some success at predicting RNA secondary structure, i.e., the folding patterns of real RNA molecules (see Figure 1). Mostly these are based on estimating the free energy associated with each possible secondary structure using dynamic programming and assuming the molecule will adapt the structure with the lowest energy. In principle, considering all possible RNA folding patterns is not feasible, but many patterns can be discarded as not being biologically plausible. For example, the structure of many RNA molecules is known and very few known structures have knots. Indeed, in RNA molecules of known structure, on average 95% of the structure is also free of pseudoknots [3, Table 1]. It is common for structure prediction software to assume that RNA contains no knots [4]. Such dynamic programming based approaches scale approximately as $O(n^3)$, where n is the number of bases in the RNA molecule. (In [5] we showed great savings can be made by running such algorithms on low cost parallel GPUs.) RNAfold [6] is the widely used de facto state-of-the-art in RNA secondary structure prediction. It is a key component of the popular internet based medical research game EteRNA [7]. However RNAfold is only as good as its underlying model allows. For example it assumes only standard RNA base-to-base binding are possible. (In panel 3 of Figure 1 the red line (g↔A) indicates a non-standard RNA base-to-base binding.)

Grow and graft genetic programming (GGGP) [8,9,10,11,5,12,13] builds on genetic improvement (GI) [14,15]. GI has been used to improve the performance of existing software, e.g. by reducing runtime [16], energy [17] and memory foot-

print [18], but (excluding software transplanting [19] and automatic bug repair [20]) typically it tries not to change programs' outputs.

We applied GGGP to RNAfold's C code [21]. Using traditional methods to identify performance critical components, recoding them using Intel's SSE vector instructions and then using GP [22,23,24] to further improve the new code. However, evolutionary search found only small increments on the human written parallel code. Nevertheless, the manually written code has been included into the standard ViennaRNA package since version 2.3.5 (14 April 2017)[1]. It is also being used by the EteRNA development team internally [25].

After speeding up RNAfold by 30% [21], the next stage was to apply GGGP to improve the accuracy of RNAfold's predictions. In technical report [26] we applied GP to the C source code and obtained a small improvement, whereas here we apply it directly to the (internal) parameters of RNAfold's dynamic programming RNA energy model. Notice here and (in the tech report) we allow (nay encourage, require) evolution to change the output of the program. I.e. to make functional changes.

See the references for introductions to GP and GGGP in particular. The next section describes our variable length linear GP system. We train it on a subset of known RNA structures from RNA_STRAND [3]. Whilst Section 3 describes the results of applying GP to RNAfold and show the improvements generalise to unseen RNA molecules. We conclude (Section 4) that evolution can improve prediction of RNA secondary structure and potentially Genetic Improvement could be widely applied to legacy chemical, physical and Bioinformatics [27] software containing empirically generated constants since maintaining such constants often lags behind knowledge in the model's target domain.

2 Genetic Improvement System

In earlier analysis [21,26] we had established that RNAfold uses dynamic programming to both calculate the minimum free energy of each RNA molecule's secondary structure and the structure itself. To do this it uses numeric constants which specify different aspects of the energy calculation. E.g. the binding energy between C and G bases and how tightly RNA can fold up on itself to form hairpin loops (an example hairpin loop is shown in red in the first RNA structure in Figure 1). These parameters are held in C strings (4), `float` (1) and `int` (51 521) variables. For simplicity we only allow evolution to change the `int` values. (Our approach is summarised in Table 1.) The `int` values are stored in 31 named variables and arrays, see Table 2. Also profiling with GNU gcov [26] we had showed that all 31 variables were read at some point when RNAfold is run on the training data. Although these variables are derived from others, e.g. to compensate for changes in temperature, they are the ones directly used by dynamic programming to predict secondary structures.

[1] The ViennaRNA package must first be configured with `./configure --enable-sse`. `https://www.tbi.univie.ac.at/RNA/documentation.html`

Table 1. GGGP to improve RNAfold's secondary structure predictions by mutating its 51 521 `int` dynamic programming parameters.

Representation:	Variable length list of 3 types (`>`, `<`, `+=`) of mutations (Section 2.3)
Fitness:	Apply mutations in order to the parameters (Table 2) before running RNAfold on training data from RNA_STRAND with less than 155 bases (681 molecules). Compare its answers with the real structure and with the default parameters' answers. Calculate the MCC between the mutated parameters' predictions and the real answers. See Section 2.4.
Population:	Panmictic, non-elitist, generational. 2000 members.
Parameters:	Initial population of random single mutants. 50% truncation selection. 50% two point crossover, 50% mutation. In generations 1–100 half mutations simply append an additional `>`, `<` or `+=` gene whilst the others apply creep mutation (± 1 to ± 5, or ± 10 to ± 50) to on average at least 20% of replacement values. No size limit.

Table 2. 31 (10 scalars + 21 arrays) RNAfold parameters which can be optimised. Data structures marked$^{\mathrm{E}}$ hold energy values which are always multiples of 10. (Mutation ensures they remain multiples of ten.) The original values of Tetraloop_E$^{\mathrm{E}}$ and Triloop_E$^{\mathrm{E}}$ are mostly zero† and so mutation of Tetraloop_E$^{\mathrm{E}}$ is limited to the first 15 elements and in Triloop_E$^{\mathrm{E}}$ to just the first element. NBPAIRS=7 and MAXLOOP=30.

noLP		mismatchM$^{\mathrm{E}}$	[NBPAIRS+1][5][5]
uniq_ML		mismatchExt$^{\mathrm{E}}$	[NBPAIRS+1][5][5]
dangles		dangle5$^{\mathrm{E}}$	[NBPAIRS+1][5]
min_loop_size		dangle3$^{\mathrm{E}}$	[NBPAIRS+1][5]
rtype	[8]	mismatchH$^{\mathrm{E}}$	[NBPAIRS+1][5][5]
gquad		stack$^{\mathrm{E}}$	[NBPAIRS+1][NBPAIRS+1]
special_hp		bulge$^{\mathrm{E}}$	[MAXLOOP+1]
pair	[21][21]	int11$^{\mathrm{E}}$	[NBPAIRS+1][NBPAIRS+1][5][5]
noGUclosure		int21$^{\mathrm{E}}$	[NBPAIRS+1][NBPAIRS+1][5][5][5]
TerminalAU$^{\mathrm{E}}$		internal_loop$^{\mathrm{E}}$	[MAXLOOP+1]
MLintern$^{\mathrm{E}}$	[NBPAIRS+1]	ninio[2]$^{\mathrm{E}}$	
MLclosing$^{\mathrm{E}}$		mismatch1nI$^{\mathrm{E}}$	[NBPAIRS+1][5][5]
MLbase		int22$^{\mathrm{E}}$	[NBPAIRS+1][NBPAIRS+1][5][5][5][5]
hairpin$^{\mathrm{E}}$	[31]	mismatch23I$^{\mathrm{E}}$	[NBPAIRS+1][5][5]
Tetraloop_E$^{\mathrm{E}}$	[200] (15)	mismatchI$^{\mathrm{E}}$	[NBPAIRS+1][5][5]
Triloop_E$^{\mathrm{E}}$	[40] (1)		

total 51521 `int`

† The energy contributions for Tetraloop and Triloop are only used under special circumstances. They represent tabulated exceptions of small hairpin loops that do not follow the values provided in hairpin. They are only used when the sequences in question match the corresponding patterns stored in the character arrays Tetraloop and Triloop.

2.1 Representation

Each member of the population is a variable length list of mutations (Section 2.3). These are applied one at a time in left to right order. Each mutation applies to one of the 31 variables and arrays in Table 2 but can potentially change many values in it. Once the whole individual has been processed, the final parameter values are loaded into RNAfold, which is then run on a training set of 681 RNA molecules (< 155 base pairs long) and its predictions of their structure is compared with their known structure to give the individual's fitness.

2.2 Initial Population

2000 individuals each containing one randomly chosen mutant were created. In later generations, mutations can be changed, one more mutation can be added, and individuals can be recombined using linear two point crossover. The mutations are split approximately equally between the three primary mutation operators. Our new mutation operators are designed to respect the existing characteristics of the energy model's parameters (see following sections and Table 2).

Several of the arrays store parameters that are dependent on each other due to symmetry [28, page 6170]. As far as the code is concerned this is behind the scenes but it reduces the number of independent variables. In particular, interior loop contributions should be symmetric since evaluation should yield the same result no matter from which side you are looking at it. Currently mutation does not enforce this. Effectively we rely on the fitness function. In future perhaps each mutation could enforce symmetry. Alternatively we can envision additional mutation operators which do respect symmetry or indeed mutation operators which remove asymmetry. E.g., by replacing asymmetric pairs by their mean value. Adding more mutation operators, rather than more careful design of the existing ones, might perhaps be beneficial [29]. Another alternative, which would be more like traditional optimisation, would be to adjust the independent variables directly outside of RNAfold.

2.3 Genetic Search Operators: Mutation and Crossover

To create a new mutation, one of the 31 data structures (Table 2) is chosen uniformly at random. If one of the ten scalars is chosen, it is assigned a new value. Scalars with value 0 or 1 are inverted, those with values of 2 or 3 are give a new value chosen uniformly between 0 and 1 or between 0 and 2 and otherwise it is incremented by a multiple of 10 between -50 and +50 (not zero).

If one of the 21 arrays is chosen, one of the three array mutations (`>`, `<`, `+=`) is chosen uniformly at random.

Replace Values Mutation > The *array name* value1>value2 mutation operator is interpreted to mean every element of *array name* whose value is currently value1 is overwritten by value2.

Notice we can build individuals composed of multiple mutations. These are applied strictly in left-right order.

An array element is chosen uniformly at random and its default value is noted. Then another element is similarly chosen. If the second value is small (i.e. 0, 1, ..., or 8) then the second value is used. If it is not small or it is negative, then 50% of the time it is used and 50% of the time a random energy value which is a multiple of 10 between -50 and +50 (not zero) is added to it before it is used. In all cases the second value must be different from the first.

Overwrite Mutation < The *array name* index<value2 mutation operator is interpreted to mean every element of *array name* which matches index is overwritten by value2.

Having chosen an array, `<` next chooses one or more elements in the array. When the array has multiple indexes (Table 2) each is processed independently. Half the time every element in that particular index is selected (denoted by `*`) and the other half one of the legal indexes is chosen uniformly at random. (It appears some of the arrays are coded with index 0, i.e. the standard C convention, but element 0 is never used. Our mutation operator does not take notice of this and so mutating [0] may be a silent mutation.)

value2 is chosen as in > mutation (see previous section). However, if multiple parts of the array are to be updated (i.e. there is one or more `*` in the array index) then there is no check that value2 if different from the existing value.

Increment Mutation += The *array name* index+=value2 mutation operator is interpreted to mean every element of *array name* which matches index is to be replaced by its default value incremented by value2.

The array index is chosen in the same way as with < mutation. As before value2 is given by the default of a uniformly random chosen element of the array. If is small, a value between -5 and +5 (not zero) is chosen, otherwise a multiple of ten between -50 and +50 (not zero) is chosen as value2.

Creep Mutation Creep mutation changes the value2 in existing mutations. Therefore it is not used in the initial generation. In subsequent generations half the children are created by mutation and half by crossover. Half the mutants are created by appending an additional mutation of one of the three primary types (`>, <, or +=`) to the parent whilst creep mutation is applied to the existing genes in the parent in the other 50% of the time. Creep mutation is applied uniformly at random to the existing mutations. As an anti-bloat mechanism, it is applied at least once and then on average to 20% of existing genes. Note, as individuals increase in size, they will tend to be modified to a greater extent. (However, this proved to be insufficient to prevent bloat, see Figure 3 page 9.)

If the existing value2 is INF (i.e. 10000000) then no change is made. If creep mutation is applied to a scalar with a current value2 which is small (i.e. ≤ 3) then, if its value is 0 it is changed to 1. Otherwise the value is either increased by 1 or by -1.

If value2 is not small or we are dealing with an array then the size of the change to value2 is given by a tangent distribution [30], here $\pm \lfloor \tan(\frac{\pi}{4}(1 + \frac{3}{4}r) \rfloor$ (where r is chosen uniformly at random between 0.0 and 1.0.) This gives a non-uniform chance of ± 1 (54.6%), ± 2 (24.1%), ± 3 (13.0%) ± 4 (8.1%) and very little chance of ± 5 (0.178%).

Tabu: Preventing Genotypic Convergence As we did in [11,31], we insist that each chromosome in the whole run must be unique. I.e., we impose a genotypic Tabu restriction that the same individual is never created twice. In an effort to prevent bloated individuals side stepping this by adding genes which simply redo previous changes, each individual is reduced to a canonical form. For example, if a scalar is mutated more than once, the newest value2 is used and the earlier genes are removed from the individual. However, (as noted above) this failed to prevent bloat and it turns out that in our implementation, as programs get bigger, reducing in particular crossover to canonical form, gets increasingly time consuming.

2.4 Fitness Function

Each member of the population is interpreted as a series of mutations (previous sections) to give the final values for the parameters to be modified. As mentioned above it is impossible to use `+=` to change INF and so such mutations are ignored. If all mutations are ignored, then the individual is invalid and its fitness is not evaluated and it cannot be a parent of the next generation. Value2 can be increased only up to INF. During evolution, the INF restriction effected 0.2% of individuals.

The released code `RNAfold.c` was tweaked so that before running the dynamic programming code the original parameters of the energy model are overwritten with the mutated values.

The tweaked exe is run on all the training data. I.e., one $1/3^{\text{rd}}$ of RNA_STRAND which are less than 155 bases long. This means running the mutant's exe up to 681 times. That is, once for each of the short training RNA molecules. These are the same sequences as we used in [26]. In retrospect this is perhaps too many. For example, in [16] we used just five but these were randomly changed every generation.

RNAfold was run with option `--noPS` to suppress the production of nice pictures of the predicted structure. (The defaults were used for all other options.)

RNAfold produces its prediction as a text string made of nested brackets (to indicate pairs of bases which bind together) and "." (for unbound bases). As we did in [26] this is piped into the standard ViennaRNA (2.3.0) utility `b2ct` which converts the bracket string into .ct file format. The output from `b2ct` is piped into a comparison gawk script which calculates the Matthew's correlation coefficient $\text{MCC} = \frac{(TP \times TN - FP \times FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$. Where:

- TP = true positives, number of predicted pairs which are in RNA_STRAND's .ct file.
- TN = true negatives, total number of possible pairings not in TP, FP or FN.
  I.e. $TN = n(n-1)/2 - TP - FP - FN$ (where n is the length of the RNA molecule).
- FP = false positives, number of predicted pairs which are not in RNA_STRAND's .ct file.
- FN = false negatives, number of pairs in RNA_STRAND's .ct file but not in the mutant's prediction.

Naturally, TN tends to be large, hence we follow Lorenz et al. [6] and use Matthew's correlation coefficient as it deals well with large class imbalances [6]. The gawk script also counts the number of cases where the predicted base pair binding is different between the mutated parameters and the default (unmutated) parameters. A mutant must make at least one change to stand a chance of being selected to be a parent.

The average MCC is computed. If it is more than 0.1 worse than the mean MCC calculated for the unmutated parameters, the individual cannot be a parent. The eligible individuals in the current generation are sorted by their average MCC. And the top half are selected to be parents of the next generation.

Tabu: Preventing over searching the same fitness In order to try and encourage diversity in the evolutionary search, we apply a phenotypic Tabu limit: Each fitness value, i.e. average MCC value, can only be used as a parent $0.01\times$the population size ($0.01 \times 2000 = 20$) in the whole run. Once this limit has been reached, individuals of exactly this MCC are passed over and individuals with a lower fitness are selected to be parents.

No Sandbox Protection Against Rogue Mutants Since evolution is not permitted to change any of the code, no particular precautions were taken against badly behaved mutants.

About 2.2% of mutants caused RNAfold to fail, 90% of them with a segmentation error. For example, in the initial generation, all six mutations which change rtype (excluding rtype[0]) to a value outside the range 0..10 cause a segmentation error. Mutants which fail at runtime are not permitted to be parents of the next generation.

3 Results

The variable length representation evolutionary computation GI system was run with a population of 2000 for 100 generations (see Table 1). The training improvement in average MCC is shown in Figure 2 and together with the evolution of size (bloat) in Figure 3. The best individual from the last generation had an average MCC on the training set of 0.737044 (RNAfold release 2.3.0 scores 0.663946) and had bloated to size 2849.

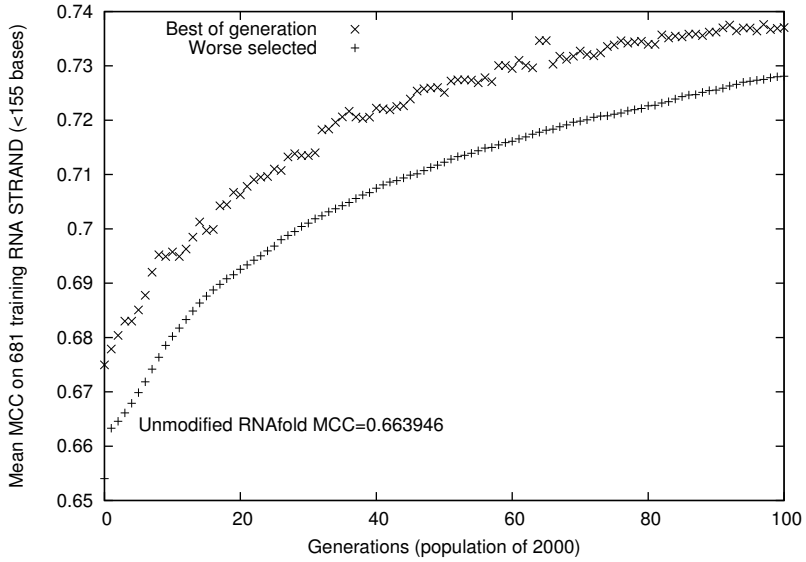

Fig. 2. Evolution of fitness (mean MCC). 1000 children whose fitness lies between best and worst are chosen to be parents of the next generation. (See also Figure 3.)

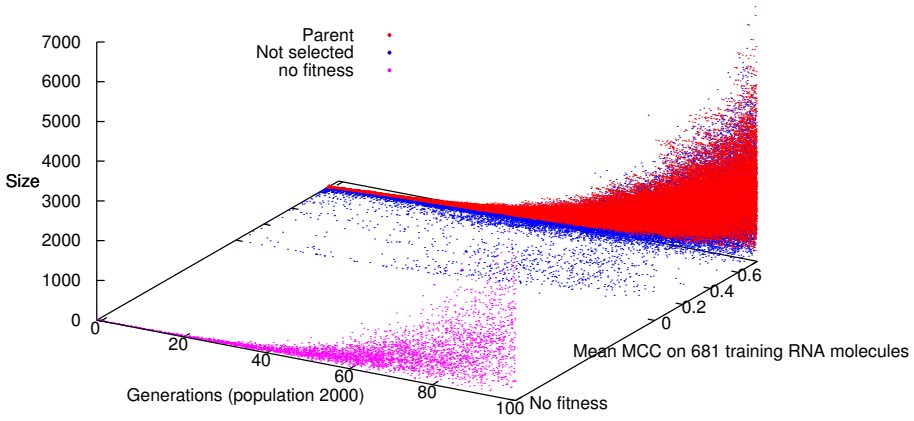

Fig. 3. Evolution of fitness and size (vertical). Children selected to be parents of the next generation shown in red (1000 per gen). Purple 4566 children which failed during fitness evaluation. Blue others also not selected.

3.1 Post Evolution Tidy

As bloat is common, there is often a post evolution phase where each part of the best individual is tested one at a time to see if that component can be removed without loosing the overall benefit [16]. Weimer et al. [32] use delta debugging to trim their bug fixing patches but we use a simple hill climber. Starting at the front of the best evolved individual, we progressively remove each mutation

and test the new individual on the whole training set. If it fails or it performs worse than the evolved individual, the deleted mutation is restored, otherwise it is deleted permanently. Then we test the next gene and so on, until we reach the end of the evolved individual. By which point each part of it has been checked. This reduced the evolved individual from 2849 mutations to 49 and the mean MCC had increased very marginally (by 0.000533) to 0.737577.

Here we ran the hill climbing a second time which further reduced it from 49 mutations to 42, with a final fitness of 0.737752. Again a very slight increase (0.000175) in performance on the training molecules. (No more changes were made when a third pass was tried.)

3.2 Generalisation Performance

The cleaned up individual (i.e. with 42 mutations) retains its performance when tried on similar length RNA molecules not used during training (average MCC 0.737752 training, 0.730137 on 682 holdout examples containing less than 155 bases). Indeed it extrapolates well to the whole of the holdout set (1553 RNA molecules from RNA_STRAND of any length). When including the larger RNA molecules, RNAfold's performance falls (release 2.3.0's mean MCC is 0.541106) but our mutant is still better, mean MCC = 0.568323. Figure 4 (page 12) compares the performance of the new RNAfold against the released code across all 1553 RNA molecules of the holdout set.

RNAfold has the ability (via its -P option) of loading other parameter settings. Andronescu et al. [33] optimised the setting and their "better optimized" values have been included in the ViennaRNA package (v2.3.0) in the file `misc/rna_andronescu2007.par`. In Figure 4 we show our 42 mutant GGGP parameters also do better than Andronescu et al. [33] (solid v. dotted line).

3.3 Changes to the Energy Model Parameters

The 42 changes cover 19 of the 31 data structures. All but 2 (ninio[2] and TerminalAU) are arrays. Together they change 14 732 `int` parameters (29% of them all). Table 3 summarises these by data structure. The data structures are sorted by their individual impact in Table 3, but, of course, the changes are interlinked and cannot readily be treated in isolation. Next, we describe a few of the changes which seem to have most impact and try and explain how they work.

mismatchH Array mismatchH has three indexes (see Table 2). The first, type, is calculated via a look up from the other two. The second, si1, is given by the base after the current active i position along the RNA molecule, the third, sj1, is similarly given by the base before the current active j position. Thus mismatchH *,*,*+=-90 mismatchH *,*,3<-130 mismatchH *,1,2<-80 corresponds to mismatchH[*,A,C] set to -80, mismatchH[*,*,G] set to -130, and all others being reduced by -90.

Table 3. Impact of the 42 components of the cleaned up evolved patches to 51 521 `int` paramters of RNAfold's dynamic programming model of RNA secondary structure. First column: components grouped by data structure (order in group is still significant). 2nd number of `int` changed. 3rd responsibility for fitness change (mutations build on each other, so isolated changes only give an indiaction of their importance). 4th again impact, this time on number of bonds changes across the whole training set. Last column describes changes with impact > 2%. See also Section 3.3.

internal_loop *+=-40	29	-6.91%	667	Add 40 to internal_loop[2..30] ([0] and [1] are INF and so cannot be incremented).
MLintern *+=10	8	-3.25%	437	MLintern[0..7] were all -90, now -80 except [3] is -150.
MLintern 3<-150				
ninio[2] 80		-2.50%	501	Was 60 now 80.
mismatch23I 70>10000000	108	-1.40%	131	
dangle5 *,*+=60	40	-1.27%	101	
int22 260>80 int22 180>280 int22 *,*,2,*,*,*+=10 int22 280>200 int22 200>10000000	10454	0.05%	37	
mismatchI *,*,0<100 mismatchI *,*,1+=-10 mismatchI 2,3,1+=-100 *,4,*+=-40	96	0.05%	617	
int11 *,*,*,*<200	1600	1.22%	1306	
int11 6,*,*,2+=-70				
dangle3 5,*+=-80	5	1.28%	13	
mismatch1nI 70>110	125	1.89%	173	
TerminalAU 80		3.04%	759	Was 50 now 80.
rtype 6<6 rtype 2+=1	2	3.05%	1257	[2] 1←2 and [6] was 5 becomes 6, page 14
mismatchExt *,*,*+=80	200	3.90%	320	+80 is added to all elements, except 1 in
mismatchExt *,*,1<-40				5 is set to -40.
stack -100>60 stack -140>0 stack 2,2+=-20 stack *,4<-50	14	6.08%	2135	[0,4] 10000000←−50 [1,4] −140←−50 [1,7] −140←0 [2,2] −340←−360 [2,4] −150←−50 [3,5] −140←0 [4,1] −140←0 [4,4] 30←−50 [4,6] −100←60 [5,3] −140←0 [5,4] −60←−50 [6,4] −100←−50 [7,1] −140←0 [7,4] 30←−50
int21 230>260	1669	6.51%	287	283 values that were 230 replaced by 260.
int21 *,*,*,*,3+=-70				161 values of 220 replaced by INF. And
int21 220>10000000				1225 cases (of a possible 1600) where int21[*,*,*,*,3] is reduced by 70.
bulge *+=40	30	7.53%	635	All bulge[1..30] increased by 40. ([0]is INF and so cannot be incremented).
mismatchM -70>-130	142	10.70%	1227	15 cases where -70 is replaced by -130.
mismatchM *,3,*+=20				2 cases where -110 is replaced by -130.
mismatchM *,1,*+=-40				20 cases where -60 is replaced by -40.
mismatchM -110>-130				40 cases where [*,0,*] is reduced by -170,
mismatchM *,0,*+=-170				35 [*,1,*] by -40, and 30 [*,3,*] by -40.
mismatchM -60>-40				
hairpin *<560	30	14.75%	1217	All hairpin[*] are set to 560 (Figure 5).
mismatchH *,*,*+=-90	180	16.30%	1610	39 cases where mismatchH [*,*,3] is set
mismatchH *,*,3<-130				to -130. 8 cases mismatchH [*,1,2] be-
mismatchH *,1,2<-80				comes -80 and 133 where other values in mismatchH are reduced by -90.
Total:	14732			

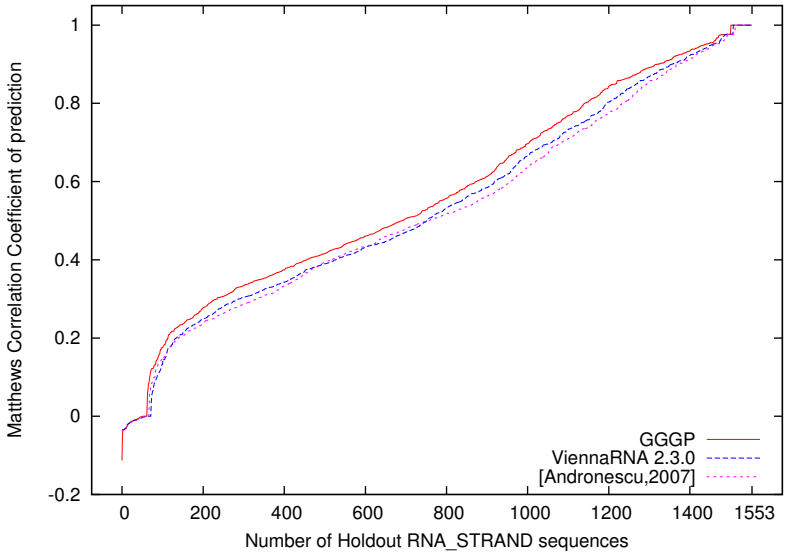

Fig. 4. Performance of best GGGP run (after two hill climbing passes) on 1553 RNA_STRAND molecules not used in training (all lengths). Dashed line performance of unmodified RNAfold 2.3.0 on same molecules. GGGP gives better predictions on 769 RNA molecules, worse 471 and same 313, $p < 10^{-16}$. GGGP also does better than parameters from the RNA_STRAND team [33], `RNAfold -P ViennaRNA-2.3.0/misc/rna_andronescu2007.par`, dotted line, $p < 10^{-15}$.

mismatchH[*,A,C]← −80 means: where the base after i is A and the base before j is a C, the energy predicted for a hairpin loop is mutated to -80. In both the cases which matter, [6,A,C] and [7,A,C], the hairpin energy was -30. I.e., the hairpin energy has been reduced by -50. Thus GI has made pairs 6,A,C and 7,A,C appear more beneficial by -50.

mismatchH[*,*,G]← −130: in the fifteen cases which matter, mismatchH [(1 4 and 7),*,G], by default holds values from -240 to -10. All 15 are over written with -130.

mismatchH *,*,*+=-90 reduces by -90 elements of mismatchH which were -250 to +20. That is, these 152 elements of mismatchH now have values of -340 to -70.

Since the dynamic programming calculation works on relative changes, it is the differences in changes between all the components of the energy calculation which determine which of the possible RNA folds are taken to be RNAfold's final prediction. In summary, GI changed 180 values in the mismatchH array, which has changed the attractiveness of 169 types of i,j pairings (those adjacent to 6,A,C (1), 7,A,C (1), (1 4 and 7),*,G (15) and 152 others, i.e. *,*,*).

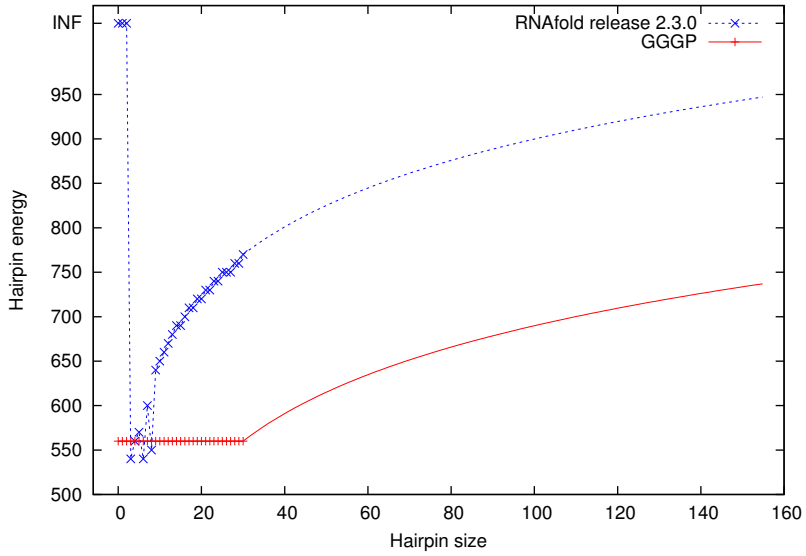

Fig. 5. Reduced energy penalty in RNAfold ($\times$ original, + evolved) for forming hairpin loops of varying sizes (x-axis). The tightest loop allowed in RNA limits x to be more than 3. (Remember training data is < 155 bases long.)

hairpin hairpin (see Table 2) holds the penalty (i.e. a positive value) for forming a loop of a given size. Loops longer than 30 lie outside hairpin's valid index range and are given by a log term. The GI change hairpin *<560 sets all values of hairpin[0..30] to 560. Figure 5 shows, in all but 3 cases, the evolved version has a lower penalty, thus encouraging the formation of hairpin folds.

To try and asses the importance of the hairpin mutations by themselves, we tried restoring the 31 default values of hairpin. As expected this performs less well on the 681 training RNA molecules (now average MCC is 0.730457 v. 0.737752 for 42 mutations). On the 1553 molecules of the holdout set it also does less well: better 344, worse 455, same 754, $p < 10^{-4}$. In summary, evolution has found a way of simplifying the contents of the hairpin array (i.e. setting the whole array to one value, 560) which is significantly better when included but in only 7% of cases is the change in MCC more than 0.1.

mismatchM The C `int` array mismatchM has the same three index as mismatchH (above). It stores energy values associated with the stabilising effect of a base pairing being adjacent to a free end or a multiloop (called a dangling end). Like mismatchH it supplies an energy value according to the bases adjacent to the two active positions i and j and their types. By default (like mismatchH) all values are negative (actually between -160 and -30), except in mismatchH the first 25 values (i.e. mismatchH[0,*,*]) are INF, whereas with mismatchM they are zero. Again mismatchM does not use array elements with index 0 also type 7 also does not seem to be used, meaning the mutations (given in Table 3) affect

59 index positions. Five increase (i.e. penalise) bond pairs by +40 (all -80 to -40, and G↔G or G↔A). Twelve increase (i.e. penalise) bond pairs by +20 (-140 to -120, -120 to -100 and -60 to -40, C↔C, C↔U, G↔*, U↔U).

The other 22 cases reduce the penalty by -40. Changes to C↔C or U↔C encourage bonds by reducing the energy by -70 to -130. Whilst changes for A↔* dangling end bonds are also treated more favourably by -40 but cover a large range of initial values (-160 to -30, including -70).

Manual Removal of rtype Array type does not hold energy values but (together with array pair) correspond to the internal coding of base pairs. For example, a C-G pair encoding is 1, and its reverse type (rtype array at position 1) is 2, which is the same encoding as that of a G-C pair. We were therefore surprised that evolution had changed rtype.

Of the 42 mutations, two affect rtype: The first, rtype[2]← 2, (value 2 refers to G-C pairings) so rtype[G-C] now contains the code for itself (rather than for C-G). The second, rtype[6]← 6, (value 6 refers to U-C pairings) so rtype[U-C] now also contains the code for itself (rather than for C-U). Notice rtype is no longer a permutation.

Table 3 shows the two rtype mutations by themselves gave a small improvement (MCC 0.666190 v. 0.663946) averaged over the 681 training molecules compared to no mutations. Suggesting during the run mutating rtype had an evolutionary advantage. However, this does not sustain to the end of the GGGP process.

As a post-hoc experiment, we manually removed both changes to rtype. On the training set of 681 short molecules it has an average MCC of 0.724700, (i.e. slightly worse than the end of the run best mutant, 0.737044, and worse than the cleaned up 42 mutant). However, on the 1553 RNA molecules in the holdout set the average MCC is now 0.569085 (remember the 42 mutant's mean MCC is slightly lower at 0.568323) but a non-parametric two sided sign test does not show a significant difference. We should perhaps remove the 2 evolved rtype changes, since removing them does not make the prediction worse and it certainly makes the mutants simpler, however, the statistics do not allow us to claim it is better.

4 Conclusions

Our previous work [26] suggested that the parameters of the dynamic programming model of the energy changes used by folding RNA were a suitable route for making non-function preserving changes to RNAfold. These parameters are derived from detailed scientific measurement of RNA. However, they are not set in stone and have been manually updated in the past to incorporate new scientific knowledge of how RNA behaves. Andronescu et al. [33] fitted the RNAfold free energy parameters by formulating a constraint optimization problem, which is quite complicated, time consuming and tedious and our GI does better (see Section 3.2 and Figure 4).

It is typical for RNA molecules with more challenging non-standard bindings to be excluded when testing RNA prediction software [6]. However we have attempted to evolve the state-of-the-art program to match all of the known RNA structures. The new version does better overall, in some cases its predictions are much better, but there are some (albeit a smaller number) where it does worse.

Genetic programming is routinely used to generate from scratch small models of physical systems (e.g. Eureqa [34]) but here we have shown it can potentially be widely used to automatically update constants within sizeable programs which have taken years to develop and are in daily use but where the task of keeping up with the latest empirical data is highly skilled, labour intensive and liable to drag months behind current scientific knowledge.

Tuned RNAfold parameters

The complete changes to RNAfold's default parameters are given in Table 3. However, for ease of use we manually converted them into a free energy parameter file `rna_gi.par` compatible with all programs of the ViennaRNA Package. Thus, they can be optionally loaded at runtime, e.g. `RNAfold -P rna_gi.par`. This required removing rtype (page 14) and ensuring matrices stack, int11 and int22, are symmetric. In the case of stack, Table 3, it can be made symmetric by adding another mutation: stack 4,*<-50. Similarly int11 is made symmetric by adding int11 *,6,2,*+=-70 and int22 by adding int22 *,*,*,*,2,*+=10 immediately after int22 *,*,2,*,*,*+=10. The parameters evolved will be shipped with the next ViennaRNA package.

Acknowledgements

I am grateful for the assistance of Rhiju Das and Fernando Portela, and our anonymous reviewers.

References

1. Tsunoda, M., et al.: Structural basis for recognition of cognate tRNA by tyrosyl-tRNA synthetase from 3 kingdoms. Nucleic Acids Res. **35**(13) (2007) 4289–4300
2. Crick, F.: Central dogma of molecular biology. Nature **227** (1970) 561–563
3. Andronescu, M., et al.: RNA STRAND: The RNA secondary structure and statistical analysis database. BMC Bioinformatics **9**(1) (2008) 340
4. Reeder, J., et al.: pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. Nucleic Acids Research **35**(suppl 2) W320–W324
5. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In: GI 2015 Workshop. 805–810
6. Lorenz, R., et al.: ViennaRNA package 2.0. Algorithms Mol Biol. **6**(1) (2011)
7. Lee, J., et al.: RNA design rules from a massive open laboratory. PNAS **111**(6)
8. Harman, M., Jia, Y., Langdon, W.B.: Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system. In: SSBSE. (2014) 247–252.
9. Jia, Y., Harman, M., Langdon, W.B., Marginean, A.: Grow and serve: Growing Django citation services using SBSE. In: SSBSE Challenge Track. (2015) 269–275.

10. Kocsis, Z.A., Swan, J.: Genetic programming + proof search = automatic improvement. (Journal of Automated Reasoning) to appear.
11. Langdon, W.B., Lam, B.Y.H., Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In: GECCO. (2015) 1063–1070
12. Langdon, W.B.: Genetic improvement of software for multiple objectives. In SSBSE. LNCS 9275, Springer (2015) 12–28 Invited keynote.
13. Langdon, W.B., Lam, B.Y.H., Modat, M., Petke, J., Harman, M.: Genetic improvement of GPU software. GP & EM **18**(1) (2017) 5–44
14. Langdon, W.B.: Genetically improved software. In Gandomi, A.H., et al., eds.: Handbook of Genetic Programming Applications. Springer (2015) 181–220
15. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. (IEEE Transactions on Evolutionary Computation) In press.
16. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1) (2015) 118–135
17. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: GECCO, ACM (2015) 1327–1334
18. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In Silva, S., et al., eds.: GECCO, Madrid, ACM (2015) 1375–1382
19. Marginean, A., Barr, E.T., Harman, M., Jia, Y.: Automated transplantation of call graph and layout features into Kate. In SSBSE. LNCS 9275 (2015) 262–268
20. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1) (2012) 54–72
21. Langdon, W.B., Lorenz, R.: Improving SSE parallel code with grow and graft genetic programming. In Petke, J., et al., eds.: GI-2017, ACM (2017) 1537–1538
22. Koza, J.R.: Genetic Programming. MIT press (1992)
23. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction. Morgan Kaufmann (1998)
24. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. freely available at `http://www.gp-field-guide.org.uk` (2008)
25. Das, R. Personal Communication (2017)
26. Langdon, W.B.: Evolving better RNAfold C source code. Technical Report RN/17/08, University College, London, London, UK (2017)
27. MacKerell Jr., A.D., Banavali, N., Foloppe, N.: Development and current status of the CHARMM force field for nucleic acids. Biopolymers **56**(4) (2000) 257–265
28. Zuber, J., et al.: A sensitivity analysis of RNA folding nearest neighbor parameters identifies a subset of free energy parameters with the greatest impact on RNA secondary structure prediction. Nucleic Acids Research **45**(10) (2017) 6168–6176
29. Angeline, P.J.: Multiple interacting programs: A representation for evolving complex behaviors. Cybernetics and Systems **29**(8) (1998) 779–803
30. Langdon, W.B.: Genetic Programming and Data Structures. Kluwer (1998)
31. Langdon, W.B., Lam, B.Y.H.: Genetically improved BarraCUDA. BioData Mining **20**(28) (2017)
32. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE. (2009) 364–374
33. Andronescu, M., et al.: Efficient parameter estimation for RNA secondary structure prediction. Bioinformatics **23**(13) (2007) i19–i28
34. Schmidt, M., Lipson, H.: Distilling free-form natural laws from experimental data. Science **324**(5923) (2009) 81–85