Applying, Improving and Analyzing Some Heuristics for Binary Codes in Combinatorial DNA Design

Lyndon Hiew
University of British Columbia
Dept. of Computer Science
lyndonh@cs.ubc.ca

Siamak Tazari
University of British Columbia
Dept. of Computer Science
siamakt@cs.ubc.ca

January 19, 2005

Abstract

Combinatorial algorithms have long been used to design error correcting codes with various constraints such as Hamming distance or weight. Such algorithms have been used to design DNA codes, which parallel error correcting codes in many ways. We implemented two algorithms from coding theory and adapted them for DNA. We tried several improvements and combinations and we will discuss the empirical analysis and the strengths and weaknesses of each variant.

1 Introduction

Short strands of DNA can now be synthesized quickly and economically. This has enabled its use in applications like DNA computing and nanostructures [12]. Given the ability to synthesize strands, the problem is designing a set of DNA strands that satisfy certain requirements. The combinatorial aspects of these requirements can be studied in the domain of coding theory, specifically the theory of error-correcting codes. Thus, various methods from coding theory have been applied, but the analysis of such algorithms is rarely mentioned in literature.

DNA consists of 4 complementary base pairs, A, C, G and T, where A pairs with T, and C with G. One strand hybridizes with another perfectly, if its reverse is complementary to the other (imperfect hybridization is also possible if the two strands do not match exactly). DNA design can be simplified to the problem of designing codewords of length n, using a 4-letter alphabet. The *Hamming distance* $H(x, y)$ of two codewords x and y is defined to be the number of positions in which they differ. One would like to have a set of codewords with minimum Hamming distance d (for some sufficiently large d) to make undesired hybridizations unlikely, i.e. to achieve that every codeword anneals to a distinct word w [10]. Furthermore, in order to prevent codewords from hybridizing between themselves, it is desirable to have codes, where the Hamming distance

between every word x and the reverse complement of all the codewords - including x - is at least d (again for some large enough d) [10]. We will not consider this constraint in this work. Though, it is possible to derive bounds for the case considering this constraint from the simpler case without this constraint [11].

There are also some thermodynamic constraints in DNA design. The strands should ideally have a similar melting point, allowing the hybridization of multiple strands to occur simultaneously [6]. One way to achieve this goal (i.e. to increase its probability) is to require that the strands have a similar number of G and C base pairs. In terms of coding theory, this is similar to the requirement of having codes of constant weight. The *Hamming Weight* of a codeword is the number of its non-zero positions. Codes with constant Hamming weight, constant-weight-codes, have been extensively studied in the coding literature. But instead of considering the number of non-zero (say, non-A) positions, we require the number of positions containing G or C to be constant. We call this number the GC-content of a strand and denote it by w - the letter usually used to denote the weight of codewords. In this terminology, we are now looking for sets of DNA words with constant GC-content w [11].

Finally, let us introduce some notation: we denote the maximum number of codes of length n over a q-ary alphabet with minimum Hamming distance d by $A_q(n, d)$. If we additionally require to have a constant Hamming weight w, we use the notation $A_q(n, d, w)$. When considering codes with minimum Hamming distance d to the reverse or reverse complement of all other words, we use the notations $A_q^R(n, d)$, $A_q^R(n, d, w)$, $A_q^{RC}(n, d)$ and $A_q^{RC}(n, d, w)$, respectively. For DNA codes with constant GC-content w we use $A_4^{GC}(n, d, w)$ and $A_4^{GC,RC}(n, d, w)$. In this work, we focus on combinatorial algorithms that attempt to find good lower bounds on $A_4^{GC}(n, d, w)$.

2 Related Work

Condon et al. [10] investigated various lower and upper bounds on $A_q^R(n, d)$ and $A_q^{RC}(n, d)$, especially for the cases $q = 2, 4$. King [11] provided some bounds on $A_4^{GC}(n, d, w)$ and constructed new DNA codes using lexicographic codes. In both of these works, one can realize the importance of general coding theory and their use of its vast literature, especially on binary codes.

Several combinatorial optimization algorithms have been applied to the constant weight code problem for binary codes, but few discuss their algorithm's running time and behavior. We would like to discuss three of these approaches here:

Dueck and Scheuer [2] introduced Threshold Accepting as a new general purpose stochastic local search method and applied it to binary constant weight codes. There was substantial emphasis on the results of Threshold Accepting compared to Simulated Annealing. The improved lower bounds for constant weight codes when compared to Simulated Annealing are presented. They did mention a CPU time of "very few seconds" for their algorithm, but only on the special case to find a 21-word code as a lower bound for $A_2(22, 10, 11)$. They also write that in order to find good codes in hard cases, more computational effort and larger experiments are necessary.

Koschnick [3] proposed an algorithm based on stochastic forward search and backward search.

He briefly mentions a range of values to use for the number of iterations that generally seemed to be working well in their experiments; but no running time estimates are given.

We looked at several related algorithms to find features and techniques that could be applied to our algorithms. A stochastic local search algorithm for finding DNA codewords was published by Tulpan et al. [9]. They successfully improved their initial, rather simple algorithm by randomly replacing a fraction of codewords whenever the algorithm had stagnated. Schrimpf et al. [8] proposed a similar idea, called "Ruin & Recreate" as a general-purpose optimization technique: "ruining" a portion of the solution and "recreating" it at each iteration. They applied it successfully to some vehicle routing problems but did not try it for codeword generation.

We also looked at Brouwer et al.'s work [1], which provides a comprehensive list of all then-known best lower and upper bounds on binary constant-weight codes. Other algorithms we considered were Dueck's "Great Deluge" algorithm [5], which he proposes as another general purpose optimization technique similar to Threshold Accepting, and Comellas et al.'s [4] genetic algorithm for finding binary codewords. We did not further investigate either of these methods and decided to focus on Threshold Accepting and Koschnick's algorithm instead.

3 Two Algorithms for Binary Codes

To start with, we implemented the algorithms of Koschnick and Dueck et al. [2, 3]. Both algorithms were originally designed for finding binary code words, but we adapted them to work on DNA code words. We then tried several modifications and combinations. We will discuss these in the next section. Here, we would like to describe the basic ideas behind these algorithms. Further details can be found in the given references [2, 3].

Dueck et al. proposed a general-purpose optimization method, which they called *Threshold Accepting (TA)*. It is a stochastic local search method with a parameter T, the current threshold. It starts with some candidate solution and iterates on these steps: it looks at the current candidate solution and creates another one in its local neighborhood by applying some small random perturbation to it. If this new candidate solution is at most T units worse in value than the current one, it accepts it as the new candidate solution and iterates. After some time, it decreases the value of T and continues.

In order to apply this to error-correcting codes, we have to define our problem as an optimization-problem: Given m, n, d, and w, we want to find a set of codewords W, with $|W| = m$, length n, weight w and minimum pairwise Hamming distance d. We can try to minimize the following objective function[1]:

$$c(W) = \sum_{H(x,y)<d} (d - H(x, y))$$

We can start with a random set W of size m obeying the weight constraint and apply TA, trying to minimize $c(W)$ until it becomes 0. When considering binary codes, we can choose a random transposition of a 0 and a 1 in a random word as our perturbation step. If our goal is to find lower bounds on $A_2(n, d, w)$, we can apply this method to increasing values of m.

[1] Dueck et al. used the square of the distances of violating pairs instead.

Koschnick's algorithm also tries to minimize $c(W)$ but uses a different approach[2]. They have two main functions, *forward_search* and *backstep*. In the forward search, they first create a list of all words that would contribute anything to $c(W)$. Then they process these words one by one in a random order as follows: look at all possible transpositions of a 0 and a 1 in the given word and choose the one that results in the largest decrease of the objective function. If no such transposition improves $c(X)$, process the next word of the list. Otherwise accept this change and *restart* the forward search. Their stopping criteria is based on a parameter called "limit_1". When this limit is reached, they do a backstep.

In the backstep, they choose a random error-value δ using a geometric distribution. Then they iteratively inspect the local neighbourhood of the current candidate solution until they find one with the precise value $c(X) + \delta$. If they see none, they choose one with the smallest value more than $c(X) + \delta$ among the ones they have seen. In our implementation, we used a value "limit_3" to decide how many neighbours to look at in this step. The algorithm stops, when backstep is performed more than "limit_2" times - another parameter.

4 Adaption and Improvements

In this section, we first describe how we adapted Threshold Accepting and Koschnick's algorithm to work with DNA strands instead of binary codewords. Then we will describe various improvements and combinations that we tried out and that eventually led to an algorithm with a satisfactory performance. A detailed empirical analysis follows in the next section.

4.1 From Binary Codewords to DNA Strands

For both of the mentioned algorithms, we first needed to generate a set W of m words of length n and GC-content w at random. We generated each word as follows: randomly choose the positions for the GC bases and assign them a G or a C with equal probability. Fill out the rest of the word with A and T bases, also uniformly at random.

The perturbation step for the binary codes consisted of randomly choosing a codeword, created by transposing a 0 and 1 in an existing binary codeword. If we applied the same method to DNA codewords, and transposed two positions at random, then the number of A, T, C and G letters would remain constant. Another problem could be caused by the initialization step randomly choosing a configuration with a disproportional number of G and A bases, compared to C and T bases. Only transposing positions would not ensure codewords to escape from being poorly configured. To probabilistically achieve all possible codeword configurations, the perturbation step in our adaption consists of the transposition of two positions or the random flipping of some position. Obviously the random change must be done so as to preserve the GC-content, so a G or C letter will be toggled to its other complementary base pair, and similarly an A or T will be replaced with its complementary base pair. The objective function could be used as given and did not need to be changed.

[2]This is only a rough description; for details see [3].

Implementing Koschnick's algorithm with this adaption, resulted in a decent performance, as can be seen in Section 5. Though, improvements were still desirable. In contrast, Threshold Accepting turned out to perform pretty poorly. So, variations and improvements were necessary.

4.2 Ruin & Recreate

We first considered Ruin and Recreate (R&R) [8], an optimization scheme that works by removing (a somewhat larger) part of the current candidate solution and recreating it, hopefully improving the value of the objective function. Our first attempt was to change the perturbation step to randomly replace a fraction of codewords. This lead to even worse results than our original threshold accepting algorithm. The comparison is discussed in Section 5.

Due to these poor results, we revised the algorithm so a fraction of codewords is replaced only when the algorithm stagnates. This method was used by Tulpan et al. [9] to contract "fat" right tails seen in the run-length distributions, which are caused by the stochastic search not finding any improvements. The perturbation step was the same as in the original, interchanging or replacing single bases in a codeword. This revision gave us slightly improved runs than the original threshold accepting algorithm when comparing their RLDs.

4.3 Storing the Best Candidate Solution

A common technique in stochastic local search is to store the best candidate solution found so far. This is a good idea, since it can happen that at one point we find a good candidate solution, "perturb away" from it and never find back there or to any better solution. In our case, this simply translates into stopping the TA algorithm when a solution with cost 0 is found, regardless of what the current threshold is. Any other "best" candidate solution with a strictly positive value would be of no use to us whatsoever.

Even though for the basic TA this is a very simple obvious thing to do, we actually made very good use of this idea in a different manner in other variations of the algorithm that we tried and will explain next.

4.4 Increasing the Threshold[3]

Let us state an interesting observation made oftentimes in local search: when we reach a local minimum and we are at the final steps of our algorithm (be it Threshold Accepting, Simulated Annealing or something else), where we do not accept (much) worse candidate solutions anymore, we will keep coming back to that local minimum and never escape that area. What is often done is to restart the search. But we decided to make use of another idea: our current candidate solution is probably not so bad, so we can slightly increase our threshold, thus moving away from this point - but not get too far away (as we would do by restarting). If our threshold gets zero and we're stuck

[3]Thanks to Frank Hutter for an interesting discussion about this idea used in Simulated Annealing.

again, then we increase again and iterate this process many times. This method turned out to be often successful in escaping local minima!

In detail, we did the following: We used a counter to keep track of the number of iterations we are spending at the current threshold value. We introduced a parameter called "stagnation", which specified how long to remain with a threshold value before decreasing it. We multiplied the value of this parameter with a constant, every time we decreased the threshold (we used the value $1.73 \approx \sqrt{3}$ for this constant, after trying out a few values between 1.4 and 2). The idea was that when the threshold is large, the search is rather random-like and not so significant but when it becomes small, we should spend more time in every stage. We also reset our counter whenever an improvement was found. When the threshold reached 0 and the current stagnation value was reached, we reset the values of the threshold and stagnation. We found out that resetting the threshold to 4 was a good choice. We iterated this process until a solution was found.

We did not use R&R but we modified our perturbation steps: at each iteration we chose with equal probability to do a transposition or to flip a base. Then we selected a word at random and looked at all possible transpositions resp. flips in that word and chose the one which would give the best benefit.

Also, we always carried the best candidate solution found so far. Whenever we wanted to reset the threshold value, we checked if the current candidate solution is more than 3 times as bad as the best one and if so, we set the current candidate solution to be that best one with probability one third and continued the search from there. This modification helped to get back to a "good" area when we drifted too far away.

4.5 Combining TA with Forward Search

Even though our last version performed reasonably well, it was still too slow in making the last moves - getting rid of that last 1 to 2 penalty points and reaching a solution. This was of course because of its random nature. A more structured search could possibly help out of this. We decided to make use of Koschnick's [3] forward search that we described in Section 3. This search looks more systematically into violating pairs and looks for a possible improvement. The key is that this function continues its search after finding an improvement and iterates until it finds a valid solution (as long as its iteration-limit is not reached). We inserted a call to this function just before resetting the threshold value - that is, at the point where we would normally give up. This turned out to be a huge improvement leading to finding solutions much faster!

Some other small changes that we made compared to the previous version were these: we refrained from multiplying the stagnation value every time, since there was no need to do it anymore; the forward search in the end would accomplish the thorough searching for us. Instead, we used slightly larger stagnation values to begin with. Also we decided to get back to the best-so-far candidate solution with a small probability (about 2%) even if the current candidate solution was not at least 3 times worse; this is because in harder cases, the best candidate solution found so far is often still large and even getting there is already hard enough - the algorithm would get away from it and not find its way back easily. We did this probabilistic resetting step whenever we reset the threshold, i.e. not after every small iteration but only when we decided to restart.

This version was indeed powerful and fast, as we will discuss in section 5 - especially for cases that were not very hard. But in very hard cases it still had trouble getting away from a local minimum if it got stuck in a large basin of attraction. This lead us to our next (final) version.

4.6 Combining Them All

We decided to incorporate R&R to attack these harder cases. The idea was that if we keep coming back to that same candidate solution value many times, we should perform a somewhat larger perturbation. We decided to destroy 2% of the current candidate solution and recreate those codewords from scratch. The recreation step was not completely random: we randomly decided about the GC-positions but when we had the choice between a G or C, resp. between a A or T, we chose the one that occured less often in that column in other words (if this really helped, we do not know). But we do know that using R&R occasionally did help to escape local minima and made it possible for us to attack harder problem instances.

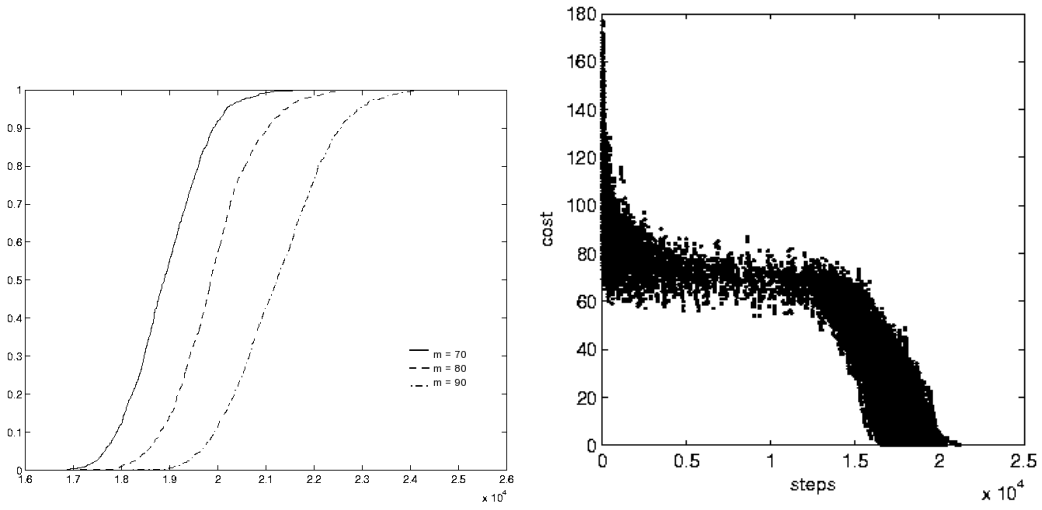
The scheme that we used was as follows: Whenever we came to the point where we wanted to increase the threshold again and start a new round, we compared the current best candidate solution value with the best candidate solution value of the last round. If it was the same, we would increment a counter, otherwise reset that counter. When this counter hit 10, we would do an R&R step (and reset the counter). The reason we did it this way, i.e. not very frequently, is that we saw that R&R disturbs the current candidate solution rather heavily, moving it often much away from its previous value - and that, in spite of the fact that we are ruining only 2 percent. So, it was only useful for doing what we intended it to do: effectively getting away from an area of the solution space, without loosing too much of the good structure found so far.

Also, we again slightly changed the rule for getting back to the best-so-far candidate solution: in the case where the current candidate solution was not at least 3 times as bad as the best one, we would get back to the best one with a 6% chance but only if its value was more than some constant (we used 6). We decided to do this because when the best-so-far candidate solution value is small, then the first condition (being at least 3 times as bad in addition to a one-third chance) is sufficient but when it is large, we would like to get back to it more frequently.

Note that this version does not differ with the previous version if the problem instance is not hard. This is because the condition for doing a R&R step is never met in these cases. But in hard cases, this variant is clearly more powerful as we will see in the analysis.

5 Empirical Analysis Results

To analyze the algorithms, we followed the methodology of Hoos et al. [7] for obtaining run-length distributions (RLDs). We obtained the RLDs of several problem instances for comparison. Each distribution consists of 1000 successful runs on a particular instance of a problem. The run-length was measured as the number of iterations of the algorithms. But we have to elaborate on that: first, note that an iteration of the basic TA is much more light-weight than an iteration in the later versions where we looked at the best possible flip/swap in a word. Also, when evaluating Koschnick's



(a) Run-length distribution (RLD) (b) cost improvement distributions

Figure 1: **(a)** Run-length distributions for the threshold accepting algorithm for $A_4^{GC}(8,4,4)$ and codeword set sizes m of 70, 80 and 90. **(b)** A distribution of cost improvements versus algorithm steps for 1000 runs of threshold accepting on $A_4^{GC}(8,4,4)$, $threshold = 30$, and set size of $m = 70$. Each data point represents the cost and iteration number when the algorithm found a better solution.

algorithm and his forward search in the combined methods, we counted every investigation of a word as one "iteration". The amount of work that the forward search does when investigating one word is very similar to the amount of work done in an iteration of the more sophisticated TAs, so we think this is a fair way of comparing them. Notice that running forward search only once can thus cause many such iterations.

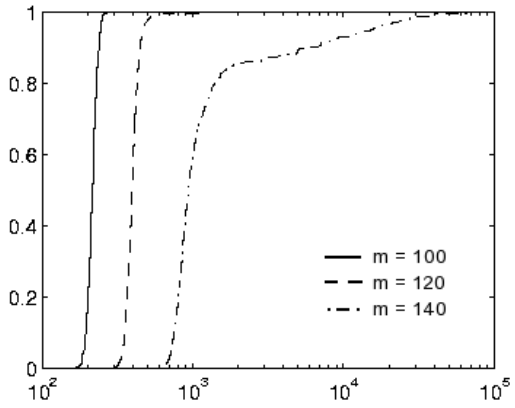
For our evaluation, we investigated the problem instance $A_4^{GC}(8,4,4)$. The best currently known lower bound given in [11] is $m = 224$. This bound is achieved by constructing a code of size 112 for $A_4^{GC,RC}(8,4,4)$ and then using a theoretical result (known as the halving bound) to get a solution for $A_4^{GC}(8,4,4)$ with twice as many codewords. It turns out that finding a code of this size using only local search is very hard. In fact, even much smaller values of m seem to be hard enough to make this instance interesting for investigation.

For RLDs of the basic threshold accepting, in order to gain some preliminary understanding of it, we decided to try much smaller values first. The sizes m of the set of DNA codewords we used were 70, 80, and 90. We avoided the use of larger values, due to the large amount of time required to obtain 1000 successful runs - we decided to work on algorithmic improvements instead. The initial threshold was set to 30, a value sufficiently large to allow the algorithm to find a solution. Values determining the number of iterations before deciding the algorithm stagnation or termination were also set to above optimal values, to ensure a solution is found. The RLDs and the distribution of cost improvements can be seen in Fig. 1.

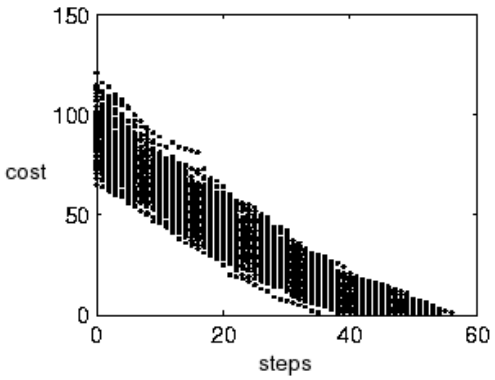
Threshold	Median	Mean
10	8637	8628
20	13778	13799
30	18886	18915

Table 1: Median and mean number of iterations for $A_4^{GC}(8,4,4)$ and $m = 70$, using different threshold values.

The choice of the initial threshold value does have an effect on the average number of iterations needed to find a solution. For $m = 70$, we obtained 1000 successful runs for threshold values 10, 20 and 30. In Table 1, we see that increasing the threshold value increases the number of iterations the algorithm must make to find a solution. The distribution of cost improvements in Figure 1(b) for a threshold value of 30, shows the cost stays fairly constant for many steps, until the threshold decreases to some optimal value, causing the cost to steadily decrease to 0. By using an optimal threshold value, one can avoid the cost improvement plateau and immediately obtain a steadily improving cost. The authors of threshold accepting mention that one of the advantages of it over simulated annealing is its relative indifference to threshold values. Having larger than optimal values will not significantly change the likelihood of success.



(a) Run-length distributions for Koschnick's algorithm



(b) Cost improvement distribution for Koschnick's algorithm

Figure 2: **(a)** Run-length distributions for Koschnick's algorithm for $A_4^{GC}(8,4,4)$ and codeword set sizes m of 100, 120 and 140. **(b)** A distribution of cost improvements versus algorithm steps for 100 runs of Koschnick's algorithm on $A_4^{GC}(8,4,4)$, and set size of $m = 70$.

We also did some brief testing of Koschnick's [3] method. It seems much faster and much more successful than our basic threshold accepting! For many different values, it is able to come close to and sometimes reach the best known lower bounds presented in [11]. A sample run-length distribution analysis for finding $A_4^{GC}(8,4,4)$ is shown in Figure 2(a). Note that here we were able to use much larger values for m and the program is considerably faster than our threshold accepting. But we also see that for $m = 140$, in about 20% of cases it gets stuck and suddenly needs considerably more time. This shows that its backstep is not particularly good in escaping local minima. The cost improvement distribution in Figure 2(b) shows a rapid cost decrease when

compared to the distribution in Figure 1(b).

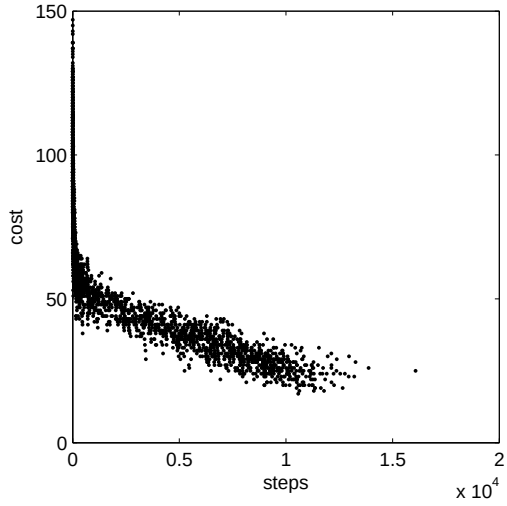


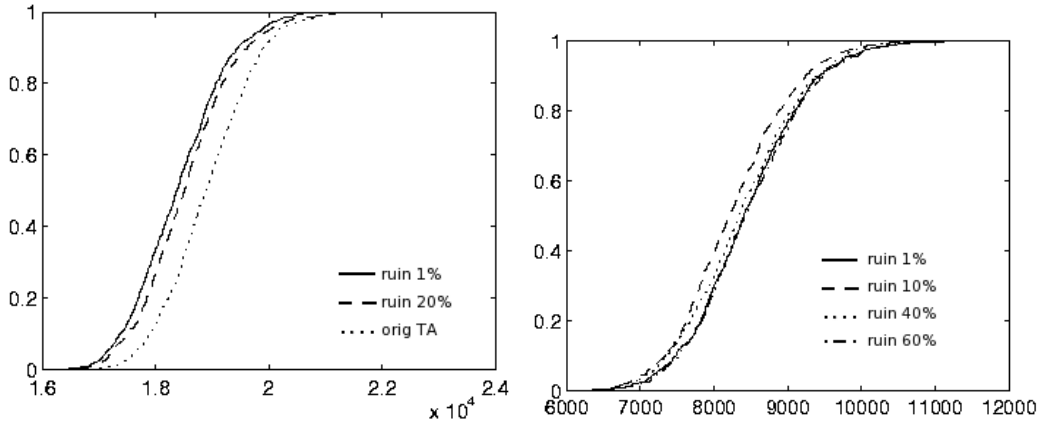
Figure 3: A distribution of cost improvements versus algorithm steps for 100 runs of our first attempt at Ruin & Recreate on $A_4^{GC}(8, 4, 4)$ and set size of $m = 70$.

In our first attempt at R&R, we found the performance to be far worse than our original algorithm. It required nearly two orders of magnitude more computation time to obtain the same number of solutions as the original. The cost improvement distribution is displayed in Figure 3. For 100 runs, the cost is never able to approach zero. Doubling the cutoff parameters for the algorithm's maximum number of iterations made no noticeable difference in the distribution. It was very evident this modification was not productive.

In Fig. 4(a), we see an RLD-comparison between the original TA vs. the version which uses R&R when it stagnates. We see that this version gives slightly improved results. We also tried ruining different fractions and see which value is better. The results are shown in Fig. 4(b). Using a fraction of $r = 20\%$, we were able to obtain 1000 successful runs in the cases $m = 100$ and 110, which we were not able to obtain using our original threshold accepting algorithm.

In Fig. 5(a), we see the RLDs for the variant presented in section 4.4, where we increased the threshold whenever it hit 0 along with some other modifications. We see that this version is powerful enough to be tested against the values $m = 100, 120$ and 140 and is faster on these instances than the original TA was on the smaller instances! We used an initial threshold value of 8 in all cases and set the stagnation parameter to $20, 30$ and 60, respectively. An interesting observation is the step-like structure of the RLDs. We think that this is due to the number of times where the threshold is reset: if an instance goes through without resetting, it will land below the first "step", if it resets once, it will be between the first and second "step" and so on. We can see that for smaller instances, we have fewer steps.

Fig. 5(b) shows the RLDs of the combination of our improved TA with Koschnick's forward search, as was described in section 4.5. We see that the RLDs of this case are much smoother and this algorithm is much faster than any other we have seen so far. So, this seems to be indeed a good version! We used the same values for the threshold and stagnation as in the previous case, except for $m = 140$, where we chose a threshold of 10 instead. Still, this variant is not very good



(a) RLDs for the original TA vs. the version using R&R on stagnation

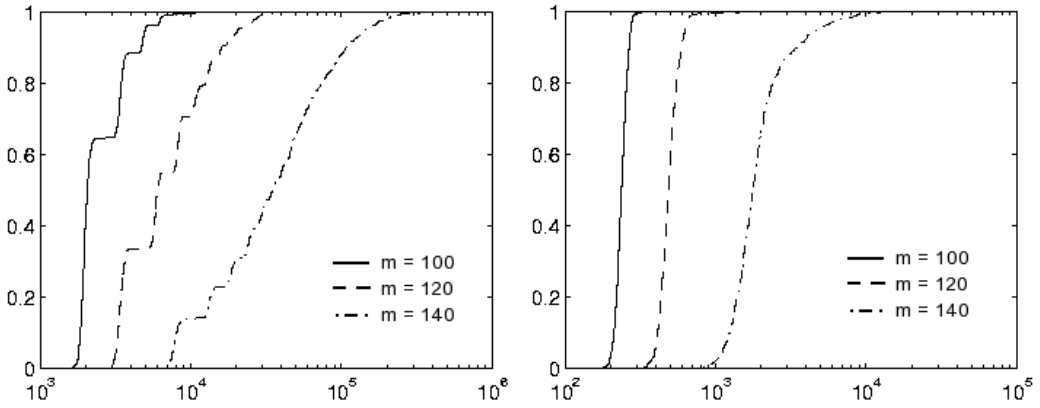
(b) RLDs for different percentages of ruining

Figure 4: **(a)** Run-length distributions for the original threshold accepting algorithm and the adapted R&R version with ruin-fractions 1% and 20%. All were run on the problem instance $A_4^{GC}(8,4,4)$ with $m = 70$ and threshold 30. **(b)** Run-length distributions of 1000 successful runs on $A_4^{GC}(8,4,4)$, threshold $= 10$ and set size of $m = 70$. The ruin-fraction had values of 1%, 10%, 40% and 100%.

for really hard instances; already in the upper 15% instances of the case $m = 140$, we can observe some difficulties; and obtaining successful runs for $m = 160$ was so slow that we decided to leave this test-case for our final version.

Finally in Fig. 6, we see the RLDs of our most sophisticated version, presented in section 4.6. Notice that here we used values $m = 120, 140$ and 160! The thresholds were set to $8, 10, 10$ and the stagnation parameters to $20, 60$, and 200, respectively. The algorithm is generally more successful in escaping local minima and thus is better suited for these harder cases. For $m = 120$, no R&R step happened, i.e. the performance was about the same as the last version - the condition of being stuck a long time never occurred. For $m = 140$, it occurred only 25 times in total during the whole 1000 runs that were tested. But for $m = 160$ it was used more frequently, 1127 times for 1000 runs. It is interesting to note that the distribution was not about one R&R step per run; many instances didn't need it at all, whereas others used it several times. We were also able to find a solution for $m = 180$, using about 7 million iterations and 192 R&R steps. When trying this value with the previous versions, the algorithms stopped without a solution. These results suggest that this version is especially well suited for hard problem instances.

We also shortly tried some other values for m, n, d and w and compared them against the bounds given in [11]. In some cases, such as $A_4^{GC}(6,3,3)$, we were able to hit the lower bounds in relatively short tests. It seems that in order to achieve some of these bounds (or maybe break them), a lot of computational effort is required when using stochastic local search methods. Since our aim was not to break these bounds (but to gain some understanding on the empirical performance of these algorithms), we did not invest the time to perform these tests.



(a) Increasing the Threshold (b) Combining TA with Forward Search

Figure 5: **(a)** RLDs for the variant presented in section 4.4, where the threshold is increased when it hits 0. Problem instances are $A_4^{GC}(8,4,4)$ with $m = 100, 120$ and 140. **(b)** RLDs for the variant combining the improved TA with Koschnick's forward search as presented in section 4.5. Problem instance same as in (a).

On CPU Times. We measured the CPU time of our implementations on an Intel Pentium 4 processor, 2GHz, 512KB L2-cache and 256MB RAM running Windows XP. Our programs were compiled and run using the Cygwin dll version 1.5.10. We used the test cases $n = 8, w = 4, d = 4$ and $m = 100, 120, 140$. The time needed for one iteration of the basic TA was on average $3.5 \cdot 10^{-5}s$. This is very low since one iteration of this basic version only performs one random transposition or flip and updates the cost-matrix. In the more sophisticated versions, where we scanned a word for its best improvement, the time per iteration increased to an average of $2 \cdot 10^{-4}s$. This value stayed the same for all subsequent versions, since the operations done in each iteration did not change significantly - it was the search-strategy of the algorithm that changed. This is also true for our final version that included ruin & recreate steps. Since an R & R operation was performed very rarely (as discussed above), its impact on the average per-iteration runtime turned out to be negligible - even on very hard cases: we ran our final version for the test case $m = 180$. The total runtime was 11 minutes and 42.279 seconds, the number of iterations 3462784 and the number of R & R steps 51; this results in $2.02 \cdot 10^{-4}s$ per iteration on average.

6 Conclusions and Future Work

Threshold accepting was presented by Dueck et al. [2] as a simpler algorithm with better performance than simulated annealing, but their results on error-correcting codes were an initial trial, still pending a more comprehensive evaluation. Based on the empirical results, this basic technique does not appear to perform well for DNA design.

We attributed this weak result to the effect of randomly replacing any codeword with a new one. Koschnick's and Tulpan et al.'s algorithms do not randomly replace codewords but search for violating codewords to replace. The likelihood of randomly finding a violating codeword decreases

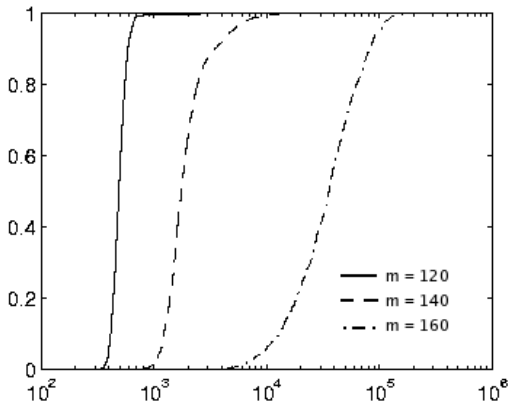


Figure 6: RLDs for the final variant given in section 4.6. Instances are $A_4^{GC}(8, 4, 4)$ with $m = 120, 140$ and 160.

as the solution gets better, so it is more advantageous to search for codewords that violate the distance constraint. Thus, searching for a good candidate codeword to replace ensures a higher probability of minimizing the objective function than replacing one at random.

Koschnick's algorithm [3] performs much better already in its basic version. It's main weakness seems to be its backstep, which is not good in escaping local minima.

We tried several modifications and combinations of these algorithms including Ruin & Recreate [8], storing and restoring the best known value of the search, and resetting the threshold, iterating many times. Our final result seems to perform reasonably well as can be seen in our thorough empirical analysis.

We conclude that combining Koschnick's forward search with the improved version of TA utilizes the strengths of both algorithms and discards most of their weaknesses. Also, R&R seems not to be a good idea on its own in this problem domain, but it appears to be a powerful tool to get out of local minima and can be used to attack harder problem instances.

Although not extensively discussed in this work, we also noticed that tweaking the various parameters of these algorithms does have some impact on their average runtime. In some cases, this impact is very small, in some other cases rather noticeable. For example, the initial threshold value did not influence the runtime significantly; but the value of the stagnation parameter did. Specifically, in hard cases it was necessary to set its value to a large number (compared to the values used for the easier cases) in order for the algorithm to succeed. See Section 5 for some values that we used in our tests. We believe that if one is interested in solving a particular problem instance, one should try to find suitable values for these parameters. What we did for this work was to experiment with some values and use the best ones for our empirical analysis. It is possible to do further, more thorough analysis about determining their optimal values as a future work step.

One more general observation that we made is that it seems like these stochastic local search methods that we considered perform very well when the number of solutions is relatively high. But as the optimal configurations become sparser and sparser in the solution space, these methods become less and less likely to succeed. For example, the theoretically achieved lower bound on

$A_4^{GC}(8,4,4)$ is 224 and it seems almost impossible to be achieved by our methods; also Tulpan et al. [9] considered much smaller values for m in this case. We believe that this solution must have a very specific structured configuration which is unlikely to be achieved by random steps.

Still, a possible direction for future research is to use our results to further improve the algorithms for DNA code generation. As we can see in this work, combining the strengths of different methods seems to be promising and one could focus on designing stochastic local search methods based on this idea with the aim to break the current known bounds on DNA code sets. A first step could be to test our algorithm more exhaustively on difficult test cases.

We did not consider other common constraints in DNA design, such as the RC constraint. It should not be very difficult to adapt our algorithm to obey some additional constraints, especially the RC constraint, since it basically comes down to changing the objective function that we are trying to minimize. We think that applying our methods to those cases could also yield interesting results.

References

[1] A.E. Brouwer, J.B. Shearer, N.J.A. Sloane, W.D. Smith: A New Table of Constant Weight Codes. IEEE Transactions on Information Theory, 36, 1334-1380, 1990.

[2] G. Dueck and T. Scheuer: Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing. Journal of Computational Physics, 90, 161-175, 1990.

[3] K.U. Koschnick: Some New Constant Weight Codes. IEEE Transactions on Information Theory, 37, 370-371, 1991.

[4] F. Comellas and R. Roca: Using Genetic Algorithms to Design Constant Weight Codes. Proceedings of the International Workshop on Applications of Neural Networks to Telecommunication, Lawrence Erlbaum, Hillsdale, NJ, 119-124, 1993.

[5] G. Dueck: New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel. Journal of Computational Physics, 104, 86-92, 1993.

[6] A.G. Frutos, Q. Liu, A.J. Thiel, A.M.W. Sanner, A.E. Condon, L.M. Smith, R.M. Corn: Demonstration of a Word Design Strategy for DNA Computing on Surfaces. Nucleic Acids Research, vol. 25, 4748-4757, 1997.

[7] H.H. Hoos and T. Stuetzle: Evaluating Las Vegas Algorithms - Pitfalls and Remedies. Proceedings of UAI-98, 1998.

[8] G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, G. Dueck: Record Breaking Optimization Results Using the Ruin and Recreate Principle. Journal of Computational Physics, Vol. 159 (2), 139-171, 2000.

[9] D. Tulpan, H.H. Hoos, A. Condon: Stochastic Local Search Algorithms for DNA Word Design. DNA Computing, 8th International Workshop on DNA-Based Computers, Springer LNCS vol. 2568, 229-241, 2002.

[10] A. Condon, R.M. Corn, A. Marathe: On Combinatorial DNA Word Design. Journal of Computational Biology, 8:3, 201-220, 2001.

[11] O.D. King: Bounds for DNA Codes with Constant GC-Content. The Electronic Journal of Combinatorics, vol. 10, R33, 1-13, 2003.

[12] T.H. LaBean: Introduction to Self-Assembling DNA Nanostructures for Computation and Nanofabrication in Computational Biology and Genome Informatics. (eds. J.T.L. Wang, C.H. Wu, P.P. Wang), ISBN 981-238-257-7, World Scientific Publishing, Singapore, 2003.