technische universität
dortmund

Search Strategies for DNA Sequence Design Software

Udo Feldkamp

Algorithm Engineering Report
TR09-4-014
Dec. 2009
ISSN 1864-4503

Faculty of Computer Science
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

Search Strategies for DNA Sequence Design Software

Udo Feldkamp
TU Dortmund, Chemistry, BCMT
Otto-Hahn-Str. 6
44227 Dortmund, Germany
udo.feldkamp@tu-dortmund.de

Abstract

Designing oligonucleotide sequences that are suitable for applications in nanobiotechnology or DNA Computing is not feasable without the help of computers. Such molecules have to hybridize specifically and with high and homogeneous efficiency to their respective complement. A computer program called dsc (DNA Sequence Compiler) translates specifications of physical, chemical and logical properties of DNA molecules into base sequences meeting these specifications. It maps the search for DNA sequences to the search for paths through a graph of base tuples with a fixed length.

Two extensions of the underlying algorithm are described here, grouping sequences for a parallel path search and tolerating violations against the sequence dissimilarity rule that enforces hybridization specificity. The effects of these extensions on success probabilities and sequence quality is examined, and some strategic hints on how to use dsc are given.

1 Introduction

The careful design of oligonucleotide sequences is an important step in rational engineering nanobiotechnology constructs, implementing DNA Computing algorithms and selecting probes for DNA microarrays. Such oligonucleotides are chains composed of the four nucleobases adenine, cytosine, guanine and thymine, connected through a sugar-phosphate backbone. The two distinct ends of the chain are labelled $5'$ and $3'$. In texts and databases they are usually represented as strings of the characters A, C, G and T, written (if not stated otherwise) in $5'$-to-$3'$ direction. A and T are called complementary bases because they may form base pairs by forming hydrogen bridges. The same applies to C and G. If several consecutive bases of a DNA molecule in $5' - 3'$-direction are complementary to a stretch of consecutive bases of another molecule in $3' - 5'$-direction, these stretches are also called complementary and they may form the famous double-helical duplex in a reaction called hybridization. Since this reaction is, under proper environmental conditions, occurring spontaneously, and is directed by the choice of base sequences, DNA hybridization is an excellent molecular implementation of programmable self-assembly, a bottom-up construction method in nanotechnology. Besides the rather simple hybridization of single DNA strands on microarrays or in many DNA Computing algorithms, larger sets of well-designed oligomers may also form complex two-

or three-dimensional structural motifs, which can be employed as building blocks in further self-assembly reactions [12].

Oligomers for nanotechnology or similar applications are supposed to hybridize specifically with their intended partner (usually the perfect Watson-Crick-complement) and avoid cross-hybridization with other DNA molecules that are partially complementary. Hybridization has to occur with high and homogeneous efficiency, in order to gain high yields of intermediate and final products and to avoid a bias in self-assembly. Depending on the application, additional requirements may emerge. For example, certain subsequences like restriction sites might be fixed, or such sequences may not occur in any oligomer. Since the number of different base sequence pools grows exponentially with the number of nucleotides in all oligomers of the set, the use of computers is essential for finding an oligomer pool meeting all requirements.

Seeman, the pioneer of DNA nanotechnology, also pioneered in the field of DNA sequence design software with his program SEQUIN [31]. It supports the user in designing DNA sequences for branched junction motifs. The user may manually add single bases or short series of bases, and the program indicates whether short base tuples of a preset length occur more than once (which should be avoided). Starting with the famous experiment of Adleman in 1994 [2], the field of DNA Computing emerged and grew, and so did DNA sequence design software.

While it was sufficient for Adleman's demonstration to choose the sequences completely by random [2], it is more advisable to at least filter the random sequences for having undesirable properties, like being too similar, having a wrong GC content, or melting at too diverse temperatures [3, 5, 9]. More elaborate search strategies comprise stochastic local search [36, 37], adaptive walk [1, 23], simulated annealing [35], or evolutionary algorithms [6, 7, 3, 27, 28, 17, 29, 32, 21, 22, 33]. The main objective, i.e. the likeliness of crosshybridization, has been modelled e.g. with Hamming-distance [3, 36, 6], H-measure, a variant of Hamming distance measured against the complement of the second sequence, and where the sequences are also shifted against each other [18, 35, 32, 21], computational incoherence, a thermodynamic-based measure for the rate of undesired hybridizations [26, 17, 27, 28], or the energy gap, a safety distance between the least stable desired and the most stable undesired hybridization [1, 23]. All these search methods have in common that they first generate sequence sets and then evaluate and select them. Other methods enforcing sequences dissimilarity already during the construction process are the template-map method, where a small set of bitstrings with a minimum Hamming distance and an error-correcting code are mapped onto a large set of DNA sequences with the same minimum Hamming distance [15], a similar shuffle operator mapping sets of short sequences on larger sets of sequences with a minimum H-measure [24], and the use of DeBruijn-sequences [8] which restrict the occurence of short tuples and have been extended to regard also the complements of tuples [34]. In recent years, some tools with graphical user interfaces supporting the construction of the structural motif have been published like Uniquimer [38], Tiamat [40] or GIDEON [4].

2 The DNA Sequence Compiler

The DNA Sequence Compiler (`dsc`) uses SEQUIN's approach to sequence dissimilarity, i.e. it avoids multiple occurrence of tuples of length n_b, but it automates the search process [10, 13]. It also enforces its dissimilarity restriction during the construction process. A set of sequences is said to be n_b-unique if every n_b-tuple that is part of a sequence in the set occurs only once in this set, and its complement does not occur at all. In dsc, all n_b-tuples for a chosen n_b are arranged in a directed graph (Figure 1), in which an edge between tuples t_1 and t_2 exists iff the last $n_b - 1$ bases of t_1 are identical with the first $n_b - 1$ bases of t_2 (i.e., t_2 may follow t_1 immediately in a sequence, with $n_b - 1$ bases overlapping).

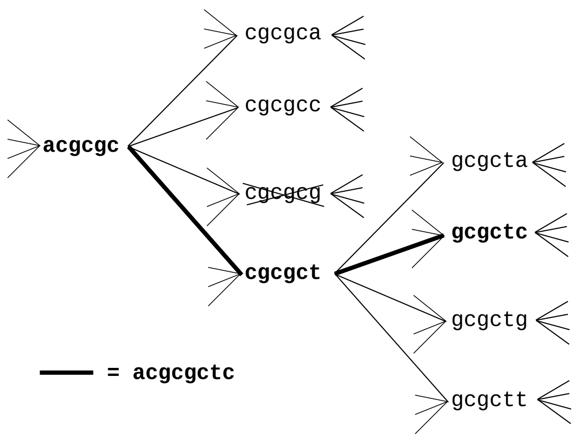


Figure 1: Graph of n_b-tuples. A sequence of length 8 is represented by a path of $8-6+1 = 3$ vertices containing 6-tuples. The vertex `cgcgcg` is self-complementary and therefore not used.

Using this graph, n_b-unique sequences can be mapped to vertex-disjoint paths, with the additional requirement that also complementary tuples of visited vertices do not appear in the solution path set. The DNA Sequence Compiler uses a simple algorithm starting at random vertices, extending paths by randomly choosing unused successor vertices and backtracking when a tuple has no more unused successors [13]. Additional requirements are also enforced, e.g. fixed subsequences determine the choice of successor vertices, violation of restrictions concerning melting temperature or GC-ratio of the molecule also trigger backtracking etc. A description language for nucleic acid molecules called DeLaNA has also been developed as an in- and output format for the compiler [11]. Since it allows the decomposition of strands into substrands which are intended to hybridize with distinct other (parts of) molecules or to stay single stranded in the nanotechnology application, a variety of target structures for diverse applications can be specified and instantiated with appropriate base sequences. A complete description of DeLaNA and all features of the software can be found in the manual, available at http://ls11-www.cs.uni-dortmund.de/molcomp/downloads/CANADA/Manual.pdf.

3 Grouping Sequences

3.1 Parallel path growth

The basic algorithm searches sequences one after another, i.e. the search for a path is only started after the path for the predecessing sequences is completely found. Backtracking tries only to find alternative paths for the currently generated sequence, all sequences that have been generated before are no longer subject to correction. The path search algorithm gains more flexibility when it can grow several paths in parallel, and particularly when it can still correct several paths by backtracking. Thus, the parallel growth of paths increases the chances for successfully finding sequences meeting all given requirements [13].

One would expect, following the arguments above, that most flexibility and highest chances for success can be gained when generating all sequences in parallel. Unfortunately, this is not feasable. It is possible to construct concatenations of sequences such that choosing the successor of a particular n_b-tuple retroactively restricts the already made choice of previously selected tuples. Programming and runtime effort for recognizing and resolving such conflicts would be quite high, therefore a compromise between one-sequence-at-a-time and all-sequences-at-once has been implemented: Assigning sequences to groups each of which is generated in parallel such that no pair of sequences within the same group produces such a conflict. This can be assured when no two sequences are generated in parallel that are specified as concatenated in the *in vitro* application. This also includes sequences that are, within the concatenation, seperated by another substring that is shorter than the tuple length n_b. For the sake of brevity more detailled explanations are omitted here, because these would have to be long and would not add important insight.

3.2 Grouping and Coloring

The problem of finding groups of "independend" sequences can be mapped onto the graph coloring problem. All sequences are arranged as vertices in a graph. Two vertices are connected by an edge if the according sequences are specified as concatenated in the application. This holds also if the two sequences are seperated by another sequence shorter than n_b or by a concatenation of sequences that has a total length below n_b. A pair of sequences will not produce a conflict as described above if they are not adjacent (connected by an edge). Thus, solving the coloring problem on this graph, i.e. labelling each vertex with a color such that no two adjacent vertices have the same color, is the same as dividing all sequences into conflict-free groups.

For quickly finding a good coloring, dsc uses the algorithm of Welsh and Powell [39]. This heuristic greedy algorithm first sorts all vertices in decreasing order of their degree. Then it proceeds through the vertices in this order and allocates each vertex the first color that is not yet allocated to an adjacent vertex.

3.3 Effect of grouping on run time and success rate

3.3.1 Materials and methods

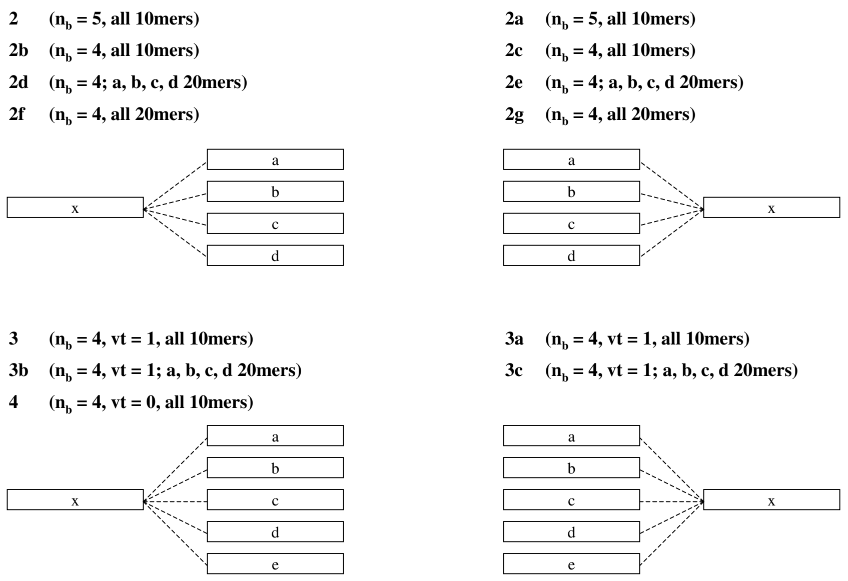
In order to verify the hypothesis whether grouping sequences for a parallel search really benefits the search, i.e. increases the probability for a successful search and maybe even lets dsc make fewer steps through the graph, 14 different test cases were examined. For each such input scenario, dsc tried to find according sequences, 100 times with sequential search (called g1 strategy) and 100 times with grouping and parallel search (gc strategy). In some cases (9 through 14), a third path search strategy (gb) was applied (see below). Success rate (number of successfull runs / 100), the number of steps the algorithm makes through the graph, and the run-time in ms were measured.

Table 1 gives a short summary of the examined scenarios. There are three groups of test cases. The "artificial" cases (1 – 4, see Figure 2) represent several scenarios

Case	Description
1	100 sequences of length 20, no concatenations, $n_b = 6$ (positive control)
2	10-mer with 3'-end concatenation to four different 10-mers, $n_b = 5$
3	10-mer with 3'-end concatenation to five different 10-mers, $n_b = 4$, violation tolerance
4	10-mer with 3'-end concatenation to five different 10-mers, $n_b = 4$, no violation tolerance (negative control)
5	3-armed junction, arm length = 10, sticky end length = 6, $n_b = 4$
6	DAE-DX-tile comprising four 11-mers and two 10-mers, $n_b = 5$
7	4×4-tile comprising eight 10-mers and eight 11-mers, $n_b = 5$, violation tolerance
8	binary random number generator comprising four 20-mers, one 10-mer and two preset 6-mers, $n_b = 4$, violation tolerance
9	two groups of four 10-mers, four concatenation pairs, $n_b = 4$
10	like case 9, with four additional concatenations, $n_b = 4$
11	like case 9, with six additional concatenations, $n_b = 4$
12	identical to case 11, but with violation tolerance
13	like case 9, with all concatenations between groups, $n_b = 4$
14	identical to case 13, but with violation tolerance

Table 1: Overview of input scenarios. Test cases 1 – 4 are "artificial", 5 – 8 are "real-world" examples and 9 – 14 are "bipartite" sequence sets. Variants of cases 2 and 3 (see text) are not listed. See Figures 2 – 4 for sketches of the scenarios.

that motivated the hypothesis that parallel path search will improve the chance of success. Case 1 is a simple pool of 100 sequences without any concatenation. This case should not pose any problem to dsc and serves as a positive control. Cases 2 and 3 are typical "branching" scenarios, where one sequence is concatenated with several different other sequences, so that the path representing the first sequence has to branch to several different paths. In case 3, the first path has to branch to five different paths, but its last vertex has only four successors. Thus, in order to make these five concatenations possible, it is necessary to tolerate violation of the n_b-uniqueness, i.e. dsc is allowed to use n_b-tuples more than once here. This violation tolerance is restricted to the first successor of the last vertex of the first sequence. In case 4, such violations are not tolerated, so dsg must not find sequences for this case, which serves as a negative control. Several different variants of cases 2 and 3 were examined (2a – 2g and 3a – 3c, Figure 2) in order to identify the scenario properties that have the strongest influence on success chances and runtime. Test cases 5 through 8 represent "real-world" scenarios (Figure 3). They describe structural motifs designed for nanotechnology applications. This group comprises a three-armed branched junction [30], a double-crossover tile of the type DAE-DX [16], a cross-shaped motif called 4×4 motif [41], and duplexes with sticky ends for the linear assembly of long helices representing binary random numbers [25, 10]. The last group of test cases (9 – 14, Figure 4) describes sequence sets that can be devided into two independend groups, i.e. there are no concatenations of two sequences of the same group. For these "bipartite" cases, an additional grouping strategy was applied. The identifiers for sequences of the first group all started with the letter 'x', sequences of the second group carried identifiers starting with 'y'. Thereby, dsc could explicitely devide the sequences into two user-defined groups for parallel path search, without using the graph coloring algorithm. In all bipartite cases, each sequence group comprises four sequences, with different concatenations (Figure 4). Case 9 simply connects these sequences to four pairs. Four additional concatenations are added in case 10, such that each sequence is concatenated with two partners. Some more concatenations make case 11 more difficult, particularly because one sequence (x4) now has four concatenation partners. Case 12 reduces the difficulty for this scenario by tolerating violations of n_b-uniqueness. In cases 13 and 14, each sequence has four partners, with violations forbidden in 13 and tolerated in 14.



2 (n_b = 5, all 10mers)
2b (n_b = 4, all 10mers)
2d (n_b = 4; a, b, c, d 20mers)
2f (n_b = 4, all 20mers)
2a (n_b = 5, all 10mers)
2c (n_b = 4, all 10mers)
2e (n_b = 4; a, b, c, d 20mers)
2g (n_b = 4, all 20mers)
x
a
b
c
d
3 (n_b = 4, vt = 1, all 10mers)
3b (n_b = 4, vt = 1; a, b, c, d 20mers)
4 (n_b = 4, vt = 0, all 10mers)
3a (n_b = 4, vt = 1, all 10mers)
3c (n_b = 4, vt = 1; a, b, c, d 20mers)
e

Figure 2: Concatenation diagrams of artificial test cases. Boxes represent sequences with the 5'-end at the left and the 3'-end at the right end. Dashed lines show which sequences are concatenated, and at which ends. In cases 2, 2b, 2d and 2f, the path for sequence x must diverge into four paths for the sequences a through d. In the other variants of case 2, four paths converge to one. In case 4 and the variants of case 3, x has five concatenation partners. Length of unique tuples n_b is varied as well as sequence length. vt=0 means no violation of n_b-uniqueness is allowed, whereas with vt=1 uniqueness violation is tolerated in the first step of path branching.

Experiments were done using the -t option of dsc, version 3.04, on a Centrino Duo processor with 2 GHz, 1 GB RAM, under Windows XP with Service Pack 2. All DeLaNA files describing the input scenarios can be found at http://ls11-www.cs.uni-dortmund.de/molcomp/downloads/example_inputs.jsp. Because the pseudo random number generator implemented in dsc uses a different random seed every second, a pause of one second was enforced between successive runs.

3.3.2 Results and discussion

In all examined cases, the Welsh-Powell algorithm was able to find the optimal number of colors (Table 2). This is not very surprising because the graphs are rather small. For the bipartite cases 9 – 14, it is noteworthy that the gc strategy

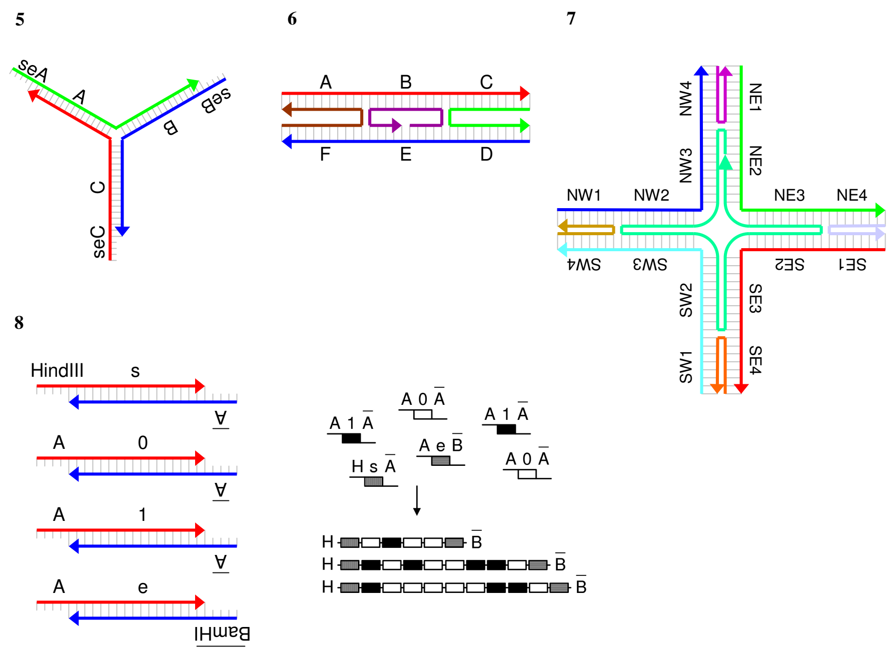


Figure 3: Structural sketches of real world cases. Colored arrows represent oligonucleotide backbones, pointing towards the 3'-end. Light gray lines show base pairs (long) or unpaired bases (short). Please note that in these sketches the number of lines does not have to match the number of bases in the actual scenarios. Labels identify subsequences treated as sequence objects by dsc. A bar above a label indicates the Watson-Crick complement. For example, the green strand in structure 5 is the concatenation of seA, A and the complement of B. The cases comprise a three-armed branched junction with sticky ends (5), a double-crossover (DX-) tile (6), a 4x4-tile (7), and a set of duplexes with sticky ends that can self-assemble to long double helices representing random binary numbers. HindIII and BamHI are recognition sites for restriction enzymes.

also seperates all sequences into two groups, and thus should be comparable to the gb method.

case	1	2	3	4	5	6	7	8	9	10	11	12	13	14
colors	1	2	2	2	3	2	3	2	2	2	2	2	2	2

Table 2: Number of colors found by the Welsh-Powell algo. For each test case, the optimal number of colors was found.

The first test case not only served as a positive control that can always be successfully generated by dsc, as described in Table 1. In an additional experiment, the number of sequences in the pool was increased in order to find the limit of dsc's capabilities. As can be seen in Figure 5, the g1 strategy starts failing to find all sequences when the pool size exceeds 111 and cannot generate a pool of 117 sequences or more. In contrast, the success rate for the gc strategy stays at 100 % up to a pool size of 119 before it decreases a bit more slowly than for the g1 strategy, finally reaching 0 % at 127 sequences. Furthermore, both strategies need equal numbers of steps through the graph for finding sequences for case 1, but the gc method is much faster considering real run time (Figure 6). The g1-strategy needs the additional time for calling the actual generation routine more often (once

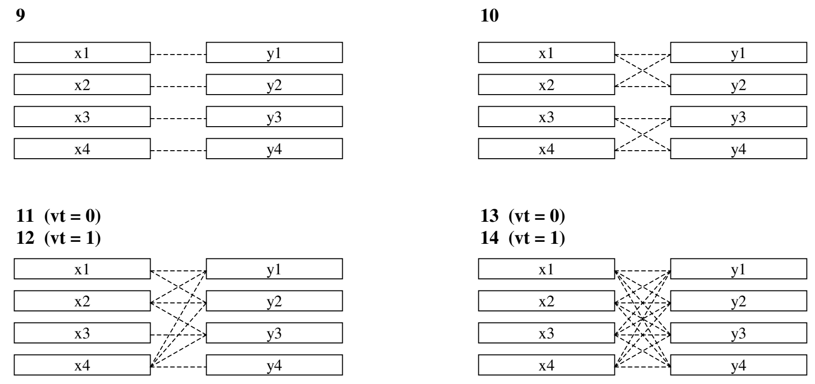


Figure 4: Concatenation diagrams of bipartite cases. See caption of Figure 2 for explanation of the representation and text for description of the cases.

per sequences instead of once per group), and particularly, for preparing several data structures before calling the routine. Thus, if there are no concatenations, it is advisable to use the gc strategy.

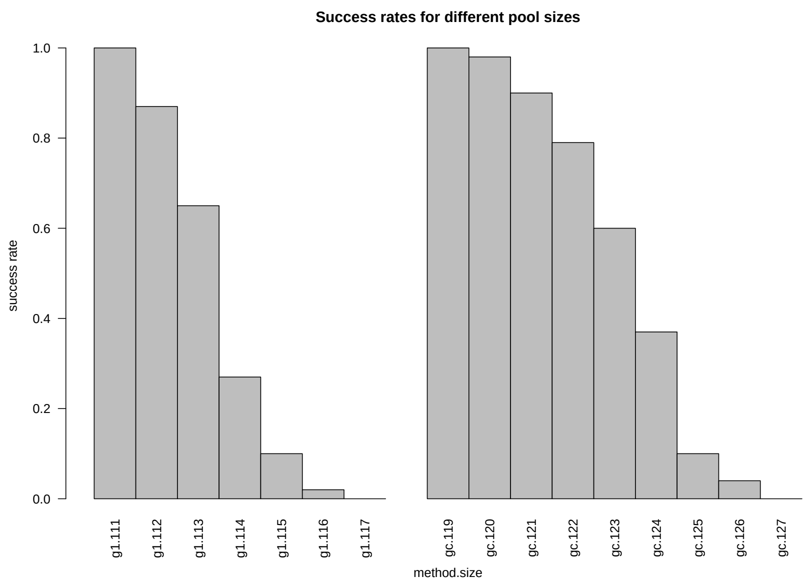


Figure 5: Success rates for sequence pools without concatenations. With the gc method dsc is able to generate bigger pools and success rates declines more slowly.

Cases 2e and 2g are not considered in the further discussion, because some runs with these scenarios ran too long and had to be stopped manually, so that it was not possible to measure 100 runs. As expected, the negative control case 4 can never be successfully processed. The other artificial input scenarios 2 and 3, representing the motivation for parallel sequence search, in fact show a higher success rate for gc than for g1. But since the cases are quite similar regarding the branching pattern (one path diverging to or paths, see Figure 2), and the success rates are in both cases and with both methods rather high, some variants were examined (2a – 2g and 3a – 3c). These comprised mirrored branchings (four or five paths converging to one) and were more difficult to process, since dsc had less tuples (with smaller length n_b) available, or more tuples were needed because of longer sequences. Figure 2 gives an overview over these variant cases and the differences between them. As

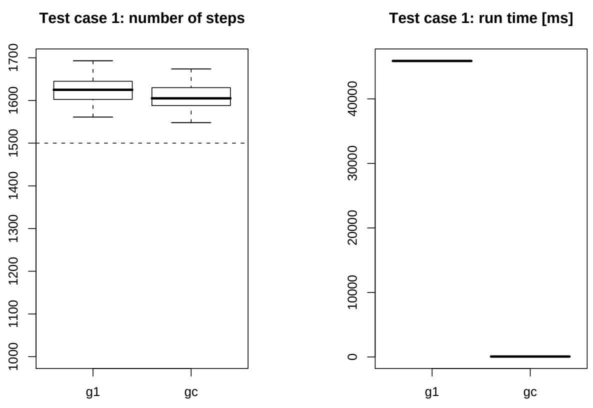


Figure 6: Time for test case 1. The boxplots show the quartiles and the medians of all 100 runs. The left diagram shows the distribution of the number of steps for both methods, g1 and gc. The dashed horizontal line indicates the minimum number of steps needed for pool generation. The right diagram shows the run times for both methods, measured in milliseconds. The horizontal lines are actually very flat boxes.

shown in Figure 8, these differences lead to notable differences in success rates. In most variant cases, the advantage of gc over g1 became clearer. In particular, the cases 2d, 2f, 3b and 3c, which were very difficult for g1 (success rates $<$ 20 %), could still be successfully processed using gc (success rates $>$ 50 %). The cases 2a and 3a were still too easy for both strategies, so that there is nearly no difference in success rates between g1 and gc. In particular, the differences are smaller than for their mirror cases 2 and 3. A rather surprising result is the drastic decrease in success between mirrored cases 3b and 3c for the gc method.

While the artificial cases confirm the hypothesis, the real-world cases 5 – 8 show a drastically different picture. Here, g1 is consistently more successful than gc. Two of the bipartite cases (10 and 12) show similar results. Case 9 is too easy to show any significant difference, and 11, 13 and 14 seem to be too hard for any strategy. No bipartite case supports the hypothesis. Interestingly, the gb strategy is only in one case (10) superior to the other two methods. In this case, it is in particularly much more successful than gc, despite the fact that both methods seperate the sequences into two groups, and only one such seperation is possible (all x-sequences into one group, all y-sequences into the other, compare Figure 4). Running dsc again for these cases, setting the protocol level to 2 (using the -p 2 option), revealed that both strategies generate the sequence groups in different orders. While gb is fixed to generate the x-sequences before the y-sequences, gc processes the y-sequences first for cases 9, 10, 13 and 14. Obviously, these different search orders lead to drastically different chances of success.

This inconsistency is also visible when regarding the time needed for sequence generation.[1] The artificial cases 2, 3 and their variants reveal an interesting difference in runtime between successful and failed runs (Figure 9, only some representative cases are shown, the boxplot for the other cases look similar). In the diverging

[1]Here, only the number of steps are shown and discussed. Boxplots for real runtime in milliseconds looked identical to those for the number of steps in all cases but 1, where the difference is shown in Figure 6.

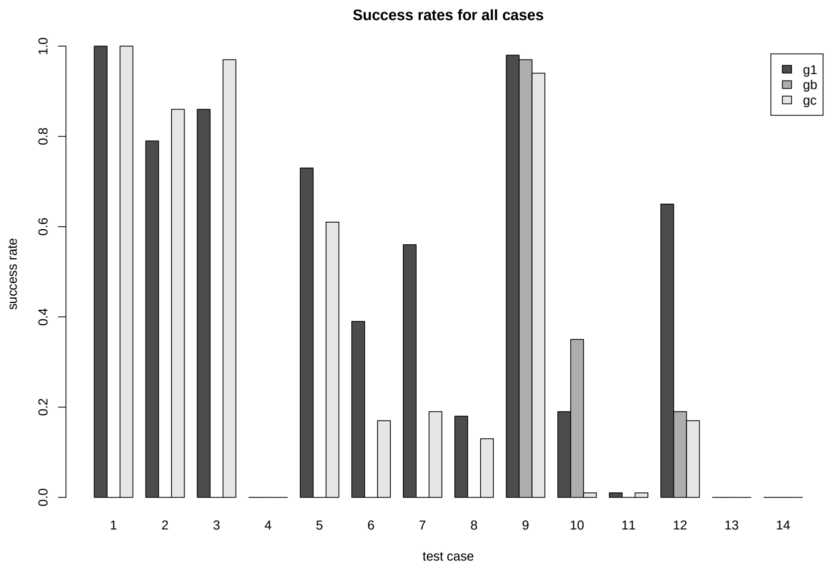


Figure 7: Success rates for the 14 test cases. Each column shows the successful fraction of all 100 runs. Each group of columns belongs to one test case. Within one group, the left column shows the success rate of method `g1`, the right one of `gc`. For the bipartite cases 9 – 14, the success rate for `gb` is shown as a middle column. Case 1 is the positive control (allways successful), case 4 serves as the negative control (never successful).

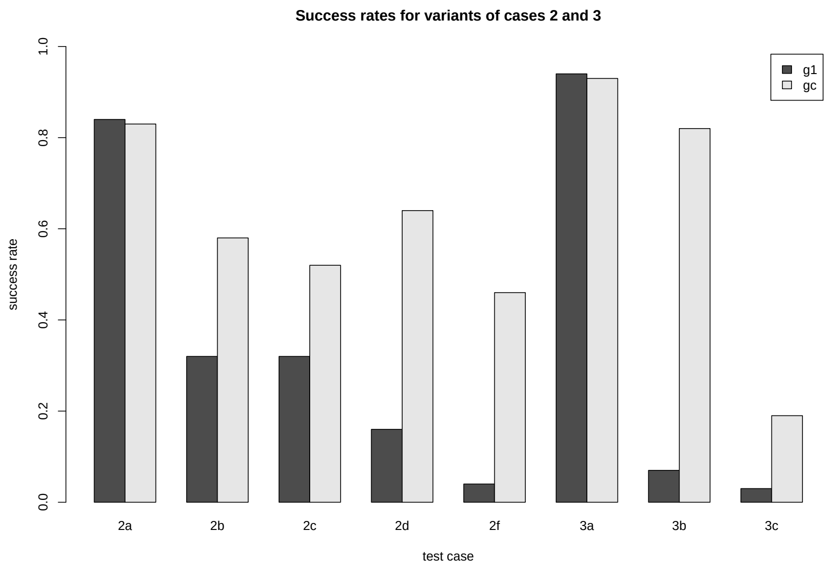


Figure 8: Success rates for the variants of cases 2 and 3. Cases 2e and 2g are not shown because the full set of 100 runs could not be completed (see text).

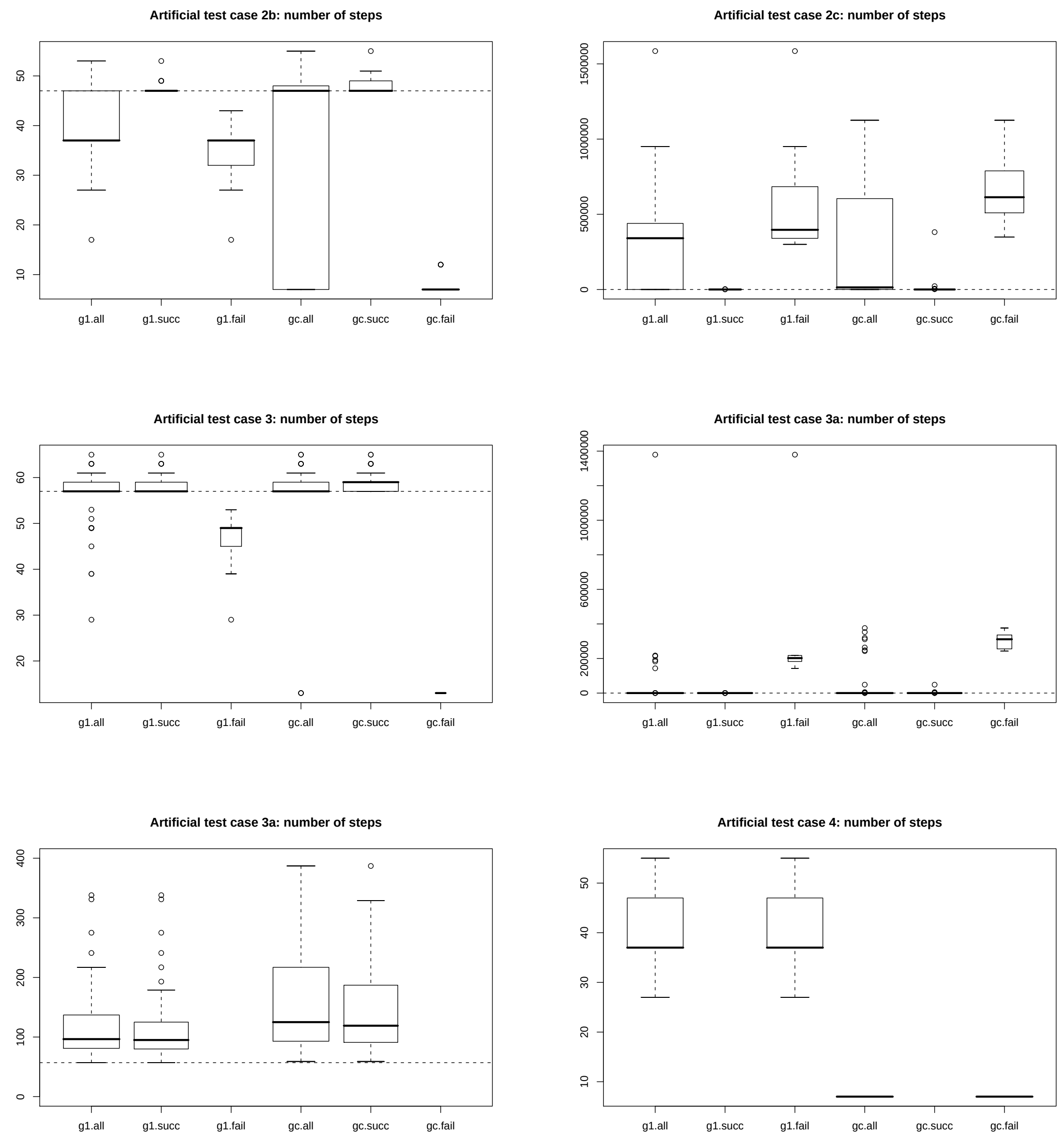


Figure 9: Number of steps through the graph for artificial test cases. The six boxes of each plot show the distribution of needed steps for all, successful and failed run for both strategies, `g1` and `gc`, respectively. The dashed horizontal lines indicate the theoretical minimum number of steps for successful generation. Please note that a zoomed-in plot for case 3a is shown (bottom left), where some successful outliers and all failed runs are not visible.

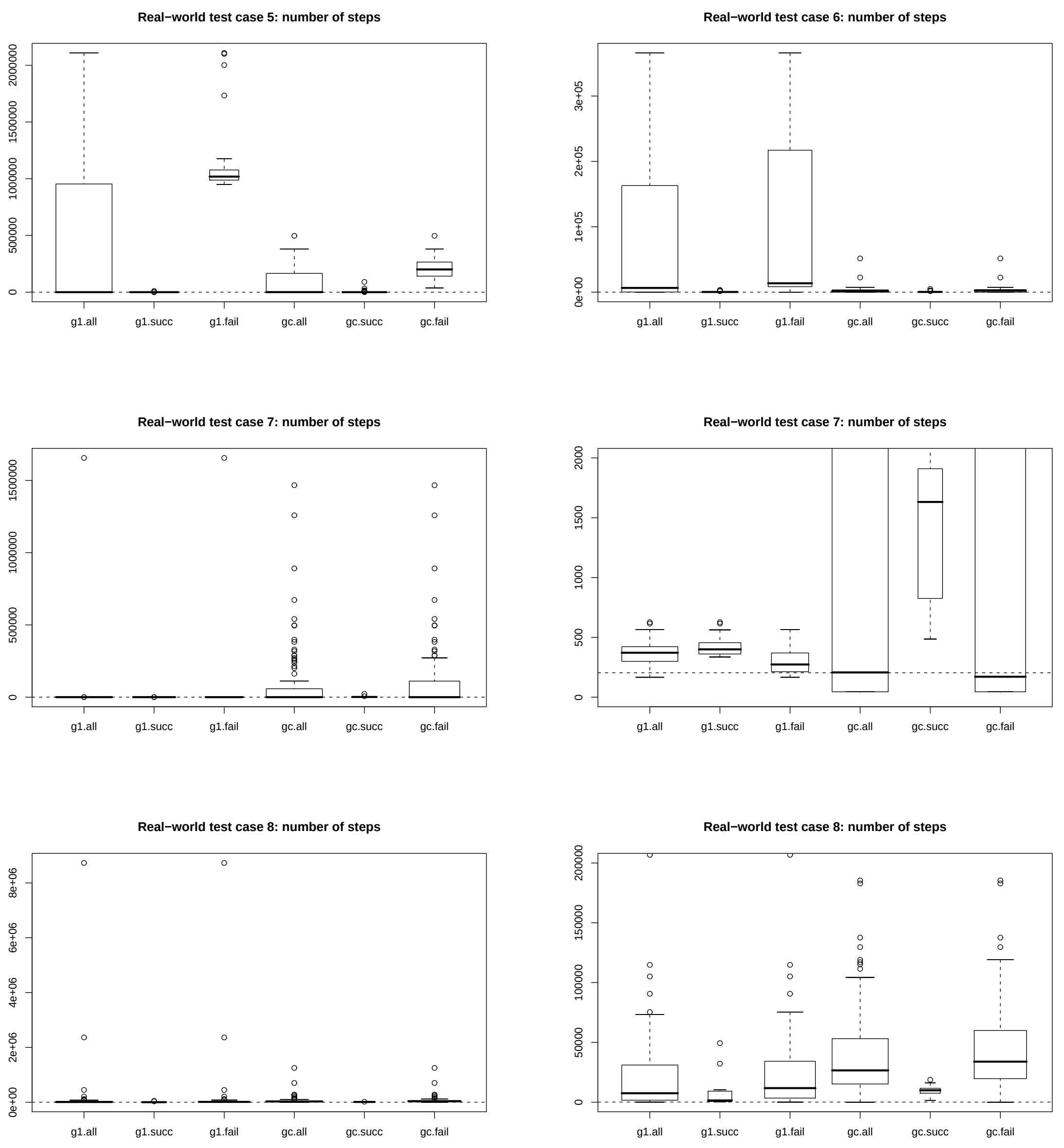


Figure 10: Number of steps through the graph for real-world test cases. The layout for each plot is the same as in Figure 9. Please note that zoomed-in plots for cases 7 and 8 are shown, where some runs are no longer visible.

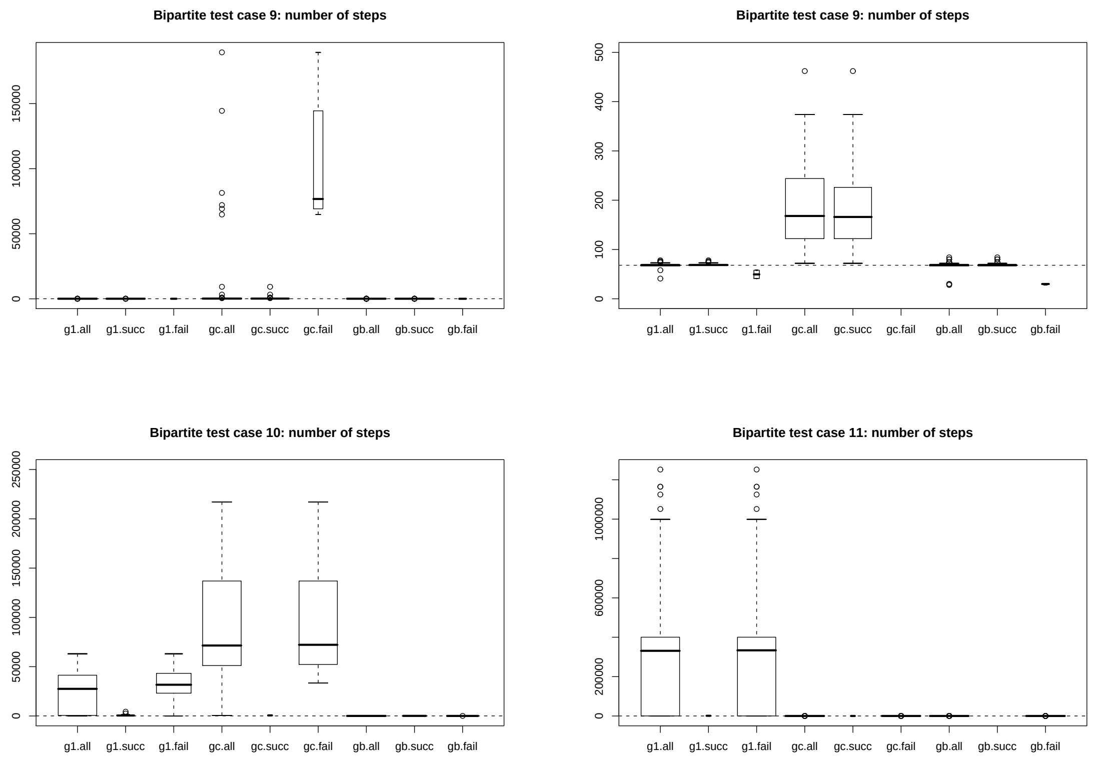


Figure 11: Number of steps through the graph for bipartite test cases. The layout for each plot is the same as in Figure 9, but three additional boxes show the distributions for the gb-strategy. Please note that several zoomed-in plots are shown, where some runs are no longer visible.

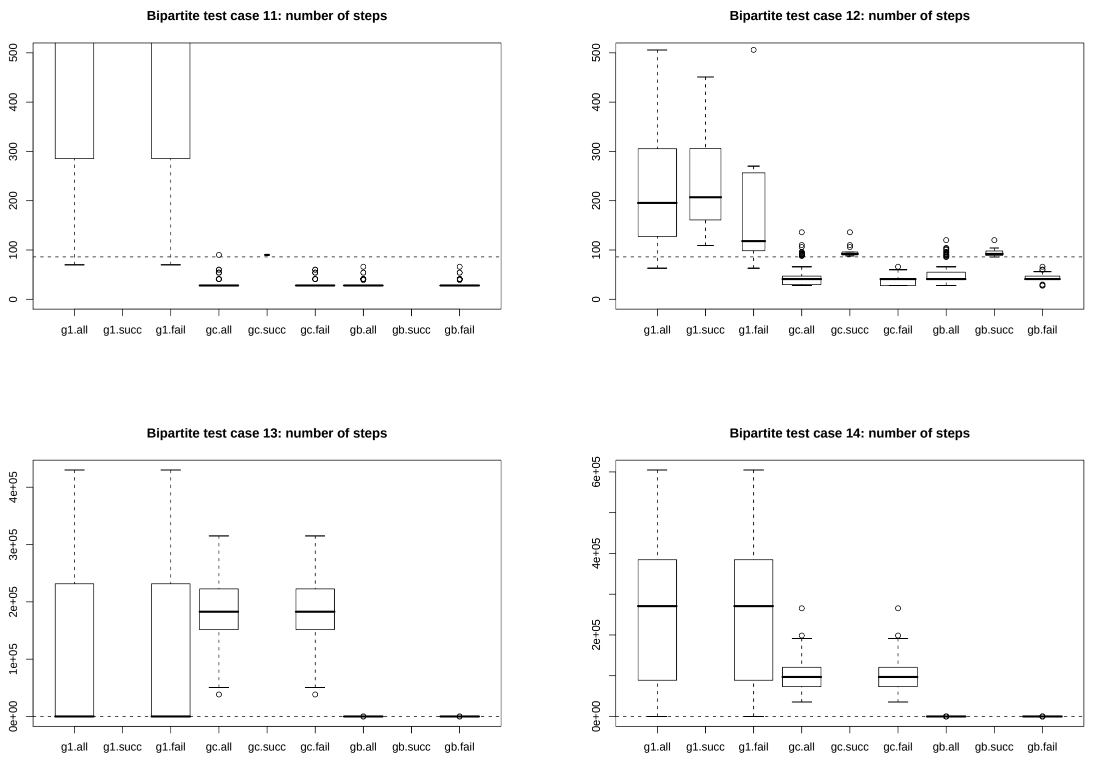


Figure 12: Continuation of Figure 11.

cases (left column in Figure 2), the failed runs needed less steps than the successful ones. In the converging cases (right column in Figure 2), the opposite is the case. This effect does not depend on whether g1 or gc is used. The critical region, where dsc is most likely to fail, is around the concatenation point. This point lies at the begin (5'-end) of sequences a – e in the diverging cases, but at their end (3'-end) in the converging cases. In all artificial cases, sequence x is generated before the others. Thus, when generating sequences a – e, some bases in the overlapping region are fixed, restricting tuple choice. In the diverging cases this critical region is processed when dsc just starts to generate sequences a – e (all with gc, or one of them with g1), while in the converging cases, if the algorithm fails to converge the paths, it will try to fix this with excessive backtracking through a – e, which uses a lot of steps. The gc strategy even enhances this effect, drastically in the diverging cases, very slightly in the converging cases. This is visible as greater differences between the boxes for successful and failed runs in Figure 9. In the divergent cases, gc tends to give up quicker, because it reaches the critical region immediately after having generated x, whereas g1 first manages to generate some of the sequences a – e before it fails. In the converging cases, the additional flexibility of gc unfortunately causes dsc to try longer to find suiting sequences through excessive backtracking, before it admits failure. This effect could also explain the drastic decrease in success for gc from case 3b to 3c observed above.

In the real-world cases, again, life is not that simple. In cases 5, 6 and 8, failed runs needed more time than successful ones, in case 7 the opposite is true. In case 8, gc slightly enhances this difference, in cases 5 and 6 it drastically weakens this effect. In case 7, gc needed notably more steps for successful runs than g1. This latter phenomenon was not experienced in any of the artificial cases.

Also in the bipartite cases, the gc strategy sometimes speed up failed runs, sometimes it slows them down (compare for example the structurally similar cases 13 and 14 in Figure 12). Interestingly, in all cases where gc and gb generate the sequence groups in different orders (9, 10, 13 and 14), gb is consistently faster. This demonstrates that the order in which the sequence groups are processed strongly affects not only the chances of success, but also run time. Furthermore, it supports the consideration above, that it is beneficial for the algorithm to hit the difficult region (i.e. the branching region) early (i.e. at the 5'-end) when generating the second group of sequences, with the bases of the neighboring sequences already fixed.

In conclusion the experiments confirm the train of thoughts that lead to the hypothesis, but they also showed that it cannot be generalized to all input scenarios. Thus, grouping sequences and generating them in parallel can improve chances of success as well as run time, but it does not have to. Therefore, when using dsc for actual applications, it is advisable to try both strategies, g1 as well as gc. Only when there are no concatenations, gc clearly outperforms g1. It could also be seen that the order in which the sequence groups are generated can have a drastic effect on success and run time. The version of dsc used here determines the order with Breadth First Search for g1 and with the degree order of the Welsh-Powell-algorithm for gc. As a consequence of the experimental results, a new version of the design software now includes an option to change the group order by shuffling them randomly.

4 Error Tolerance

4.1 Being less strict

So far, n_b-uniqueness was strictly enforced by dsc. The only exception was the controlled tolerance of uniqueness violations around multiple concatenations of sequences that correspond to branching of paths through the tuple graph. But one can suspect that there are input scenarios for which dsc does not manage to generate the desired sequences, just because there are only a few free n_b-tuples missing. Increasing n_b would result in an overall quality loss for all generated sequences. In this situation, it seems preferable to allow a few violations, i.e. the multiple use of some tuples, so that only a local quality loss here and there has to be accepted in order to increase the chance of success.

Therefore, two (rather simple) strategies for allowing errors everywhere, randomly, and with an adjustable error rate have been added to dsc, and the effects these strategies and different selections of their parameters have on success rates, run time and real error rates were examined.

4.2 Two methods

Both strategies affect the successor choice when elongating paths through the tuple graph. The first one allows errors with a fixed error probability (ep-strategy). In the strict case, dsc collects all vertices succeeding the current one that are not yet used (nor are their complements), and then randomly and uniformly selects one of these candidates as the actual successor vertex. Using ep, the path search algorithm ignores with a user-defined probability whether a vertex is already used or not when collecting successor candidates. This means that p is not the probability for actually making an error, but only for taking an "illegal" tuple into consideration as a candidate. Whether it is tolerated or not is individually determined for each illegal successor vertex.

The second strategy always considers already used tuples as appropriate successors, but prefers legal ones with a fixed bias (fb-strategy). This means that all four successor vertices are considered as candidates, but the probability for each illegal candidate is p times the probability for any legal one, where p is again a user-defined parameter. For $p = 1.0$, each candidate is drawn with 25% probability, regardless of whether it has been already used or not; for $p = 0.5$ each legal vertex is drawn with 1/3 and each illegal one with 1/6 probability. As a consequence of this strategy, dsc always succeeds in generating sequences for any input scenario.[2] Even when there are no more unused vertices left, all four successors are taken as candidates, nevertheless, and one of them is randomly drawn.

4.3 Effect on success rate, run time, error rate and error position

Allowing dsc to (sometimes) use n_b-tuples multiple times should definitely raise chances for successful sequence generation. Furthermore, run time for successful runs should decrease, because less backtracking will be necessary. On the other hand, runs that fail to find all desired sequences may run longer, since the error tolerance introduces more flexibility that dsc might exploit before admitting failure. The rate of actually made errors should, of course, increase with parameter p, but their location may be interesting. For the ep-strategy, errors should be uniformly distributed over all positions within all sequences. But when regarding only successful runs, i.e. runs where dsc has managed to find tuples around critical points like multiple concatenations, errors in these regions might be necessary for success, and therefore, errors might concentrate around critical points. For the fb-strategy, errors should also occur more frequently later during the generation process, because then there are fewer legal tuples left. In order to test these hypotheses, experiments were made with both strategies.

4.3.1 Materials and methods

As input scenarios, a selection of the cases described in Table 1 was examined. Pools of unconcatenated 6-unique 20mers are comprised under case 1. For evalutation of the effects on success rates, varying pool sizes were examined. For other measurements, pool size was fixed to 118 (run time and error rate) and 150 (error position). Two pairs of mirrored artificial scenarios were selected (2b/2c and 2f/2g), as well as a scenario where dsc has to make at least one error in order to be successful. This case was examined in two variants, one where this error is allowed by violation tolerance (3c), and a new case where this is not allowed, and the search algorithm must fail with strict successor choice (3i). Three real-world cases 6, 7 and 8 were also examined.

For these experiments, version 3.05 of dsc was used[3]. Successor-choice strategy and the according parameter was varied by altering dsc's configuration file. Sequence

[2]This holds only if no restrictions of other sequence properties like melting temperature are applied.
[3]The difference to version 3.04 used in the grouping experiment is the facility to allow errors.

grouping using the coloring algorithm was applied in all experiments. Hardware is the same as specified in section 3.3.1.

For the ep-strategy, success rates, run time, error rate and error position were measured over 100 runs per test case. Since the fb-strategy always succeeds without any backtracking, only error rates and error position were measured for this variant. Run time is measured in steps through the tuple graph, error rate as $1 - \frac{\text{number of different tuples used}}{\text{number of tuples needed in strict case}}$. For error position analysis, sequence number and position in the sequence for n_b-tuples occurring more than once were extracted from the ReappearingBaseStrands.txt output files, and histograms were produced. For case 1, "typical" output files were selected, for other cases, the results of 10 different runs were accumulated to get meaningful numbers. For the ep-strategy, parameter p was chosen from $p \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$, except for case 1 where $p = 0.4$ and $p = 0.5$ were omitted. For fb, $p \in \{0.0, 0.1, 0.2, 0.5, 1.0, 2.0\}$.

4.3.2 Results and discussion

For the sake of brevity, not all results for each test case are presented, only some representative examples are shown and discussed here. Missing results are similar to those shown here or did not contribute anything helpful to the discussion.

Error probability

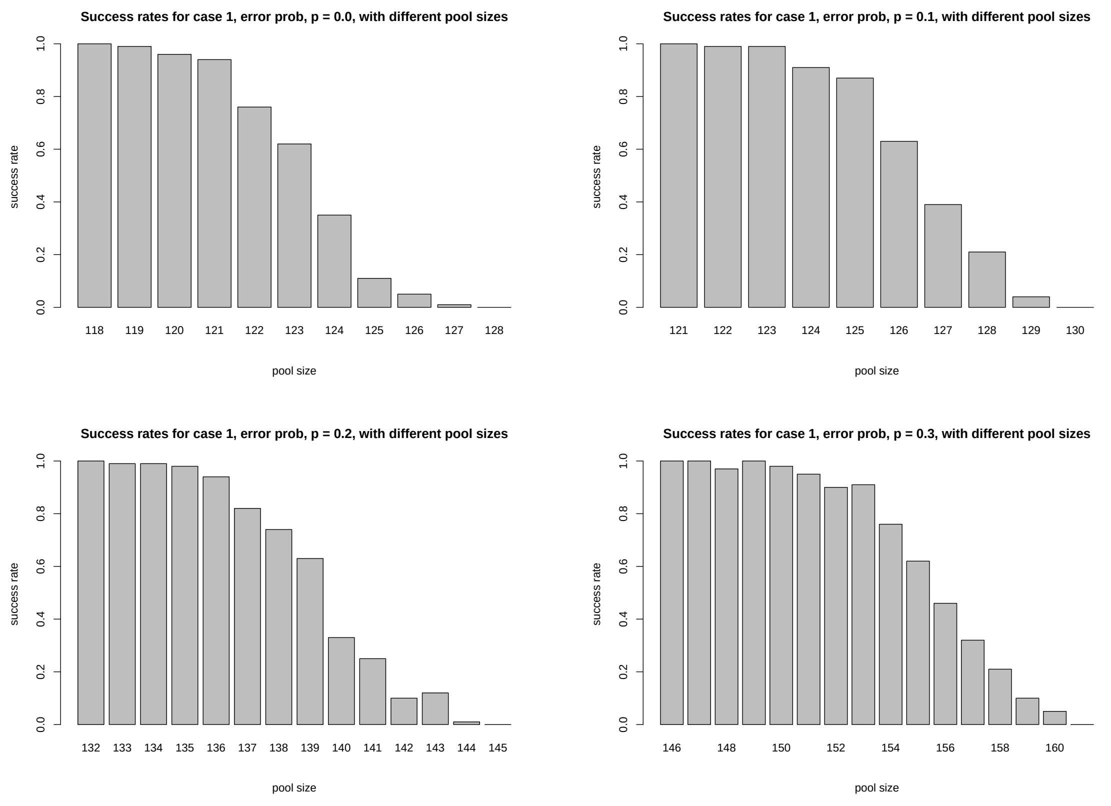


Figure 13: Success rates for case 1, ep-strategy. With growing p, bigger sequence pools can be generated, and the decline of success rate becomes slower.

As shown in Figure 13, bigger pools can be generated with increasing p, which is no surprise. More interesting, the distance between pool sizes that can reliably be generated from one p-value to the next increases with p, too. The maximum pool sizes with success rates of 1.0 are 118, 121, 132, and 147 for $p = 0.0, 0.1, 0.2, 0.3$, respectively, with distances of 3, 11 and 15. Furthermore, the distance between the greatest pool size with success rate 1.0 and the smallest pool size with success rate 0.0 grows with p. Both effects are probably caused by a kind of positive feedback loop, where higher p allows dsc to make more errors, so that it can generate more sequences; and in these additional sequences, it can make even more errors, etc. If one sets p to the maximum value of 1.0, errors are always allowed, and an infinite set of pools can be generated.

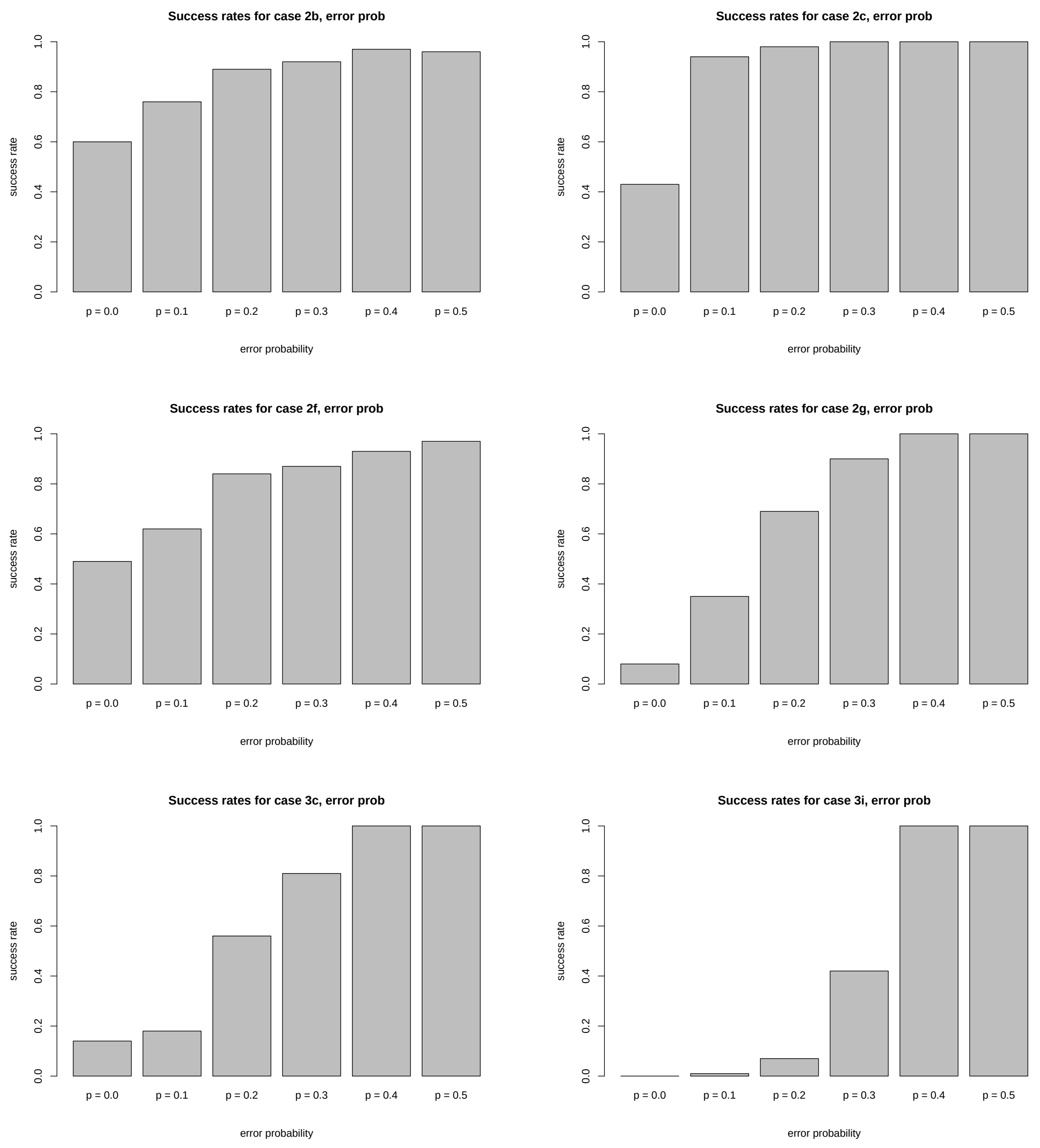


Figure 14: Success rates for artificial test cases, ep-strategy. All scenarios benefit from error tolerance, but in notably different ways.

For $p = 0.0$, the convergent artificial test cases 2c and 2g start with lower success rates than their divergent mirrored counterparts 2b and 2f, but with growing p, their success rates become quickly higher than those of their mirror cases (Figure 14). Obviously, the convergent cases benefit more from the ep-strategy. In these cases, the critical regions (path branching points) where uniqueness violations are most helpful are processed rather late, when there are not much unused tuples left and errors are more often necessary for success. In contrast, in the divergent cases these critical regions are processed earlier, when there are still a lot of unused, legal tuples available, and allowing used, illegal tuples has a weaker effect. The steep rise of the columns for case 2g clearly suggests the existence of a "bottleneck". In order to cope with a particularly difficult point in the path search, a certain p-level is necessary. When this level is reached, the rest of the sequences obviuosly pose no problem for the search algorithm. This can also be seen in case 3i, where at least one error is necessary for success, since the first tuple of x must have five predecessors, but there are only four tuples that would fit there (see Figures 1 and 2). When p is high enough to reliably make this error, the rest of sequence x is no problem. Here, a really high error probability ($p = 0.4$) is necessary, which leads to drastically more errors than the one that is really needed (error rate of 25 %, compare Figure 18). In order to keep the error rate small, the controlled tolerance of uniqueness violations around path branching points still is a sensible strategy.

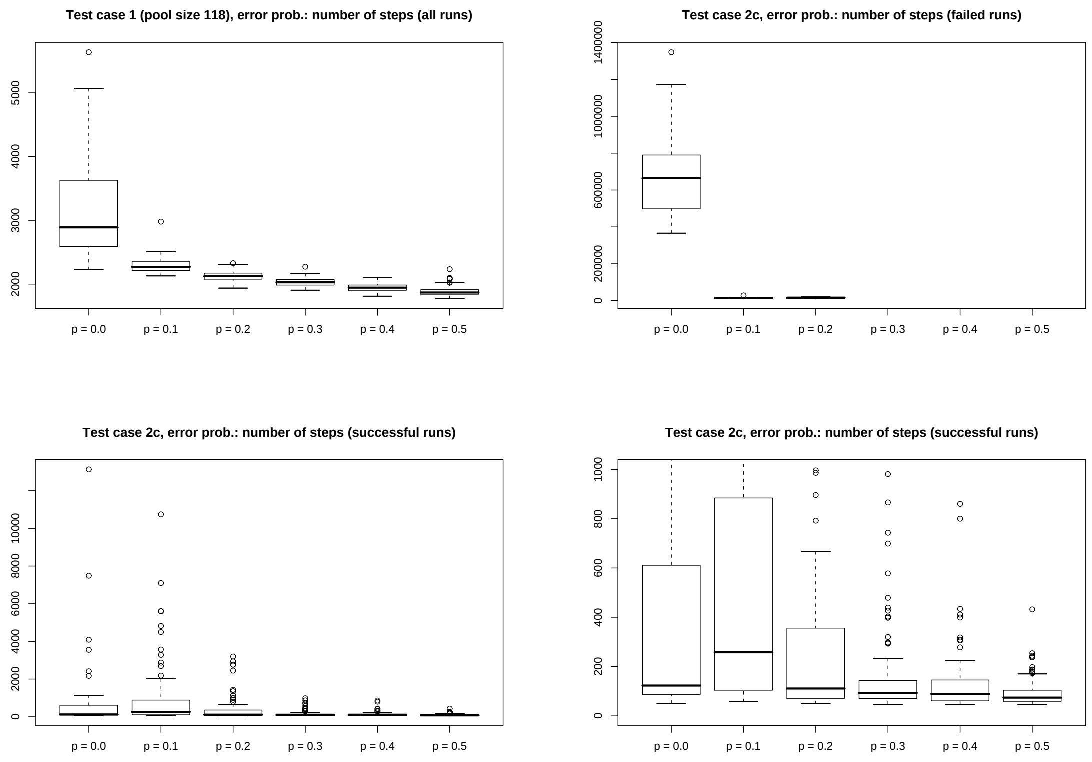


Figure 15: Number of steps through the graph for artificial test cases, ep-strategy. The boxes within one plot show the distributions for different values of p for successful or failed runs only. Please note that zoomed-in plots for successful runs of cases 2c and 2g are added, where some runs are no longer visible.

For pools of unconcatenated sequences, the run time measured in steps through the tuple graph steadily decreases with p, as expected, because allowing more errors

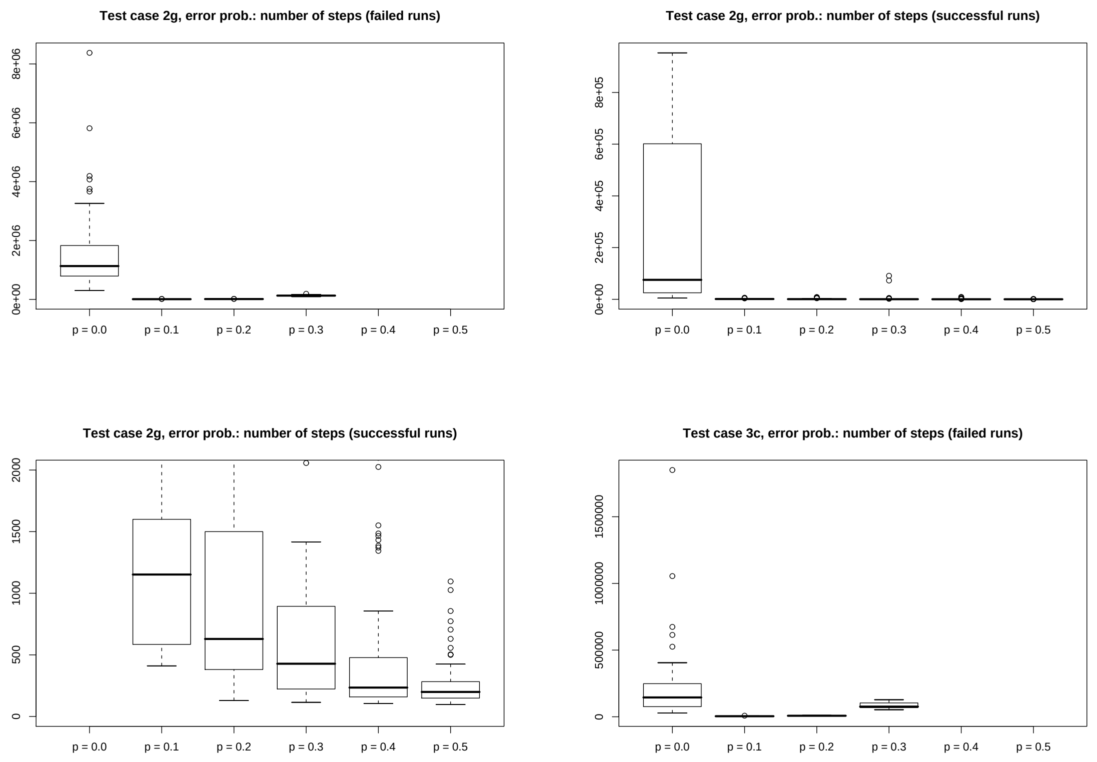


Figure 16: Continuation of Figure 15.

means less backtracking that has to be done (Figure 15). The same picture can be seen for the successful runs of cases 2f, 2g, 3c and 3i, as well as for failed runs of 2c (Figures 15 and 16.) But in several cases (2g, 3c, 3i, 8), the failed runs show a suprisingly different behaviour. There, run time first decreases with growing p, before starting to increase again for higher values of p. This effect can also be seen for the successful runs of case 8 (Figure 17). The successful runs of case 2c show a reverse phenomenon, with run time first increasing and showing the expected decrease only for $p \geq 0.2$. In cases 6 and 7, run time grows continually over the whole range of examined values of p. The runs and cases not yet mentioned did not show much change over the examined range of p, mainly because they had not much potential for reducing their run time. While the hypothesis that successful runs need less and failed runs need more time with higher p is overall supported by many testcase, it has to be refined in order to be generally valid. Failed runs can benefit from small values of p, too, needing less time. The reason for this phenomenon is not quite clear. Maybe dsc can here reach a critical point more quickly that causes it to fail. On the other hand, an increase in run time can be observed for successful runs if p is high enough or the input scenario is difficult enough. Increasing p also means enlarging the search space, because the search is no longer limited to the space of n_b-unique sequence sets. Obviously, tolerating errors has always a time-increasing as well as a time-decreasing effect, and it depends on the actual value of p and the input scenario which of the two influences dominates the actual net effect on run time.

The actual error rate grows with increasing p, which is as expected. but only in some cases (like 2f, see Figure 18) this growth can be called linear. For the

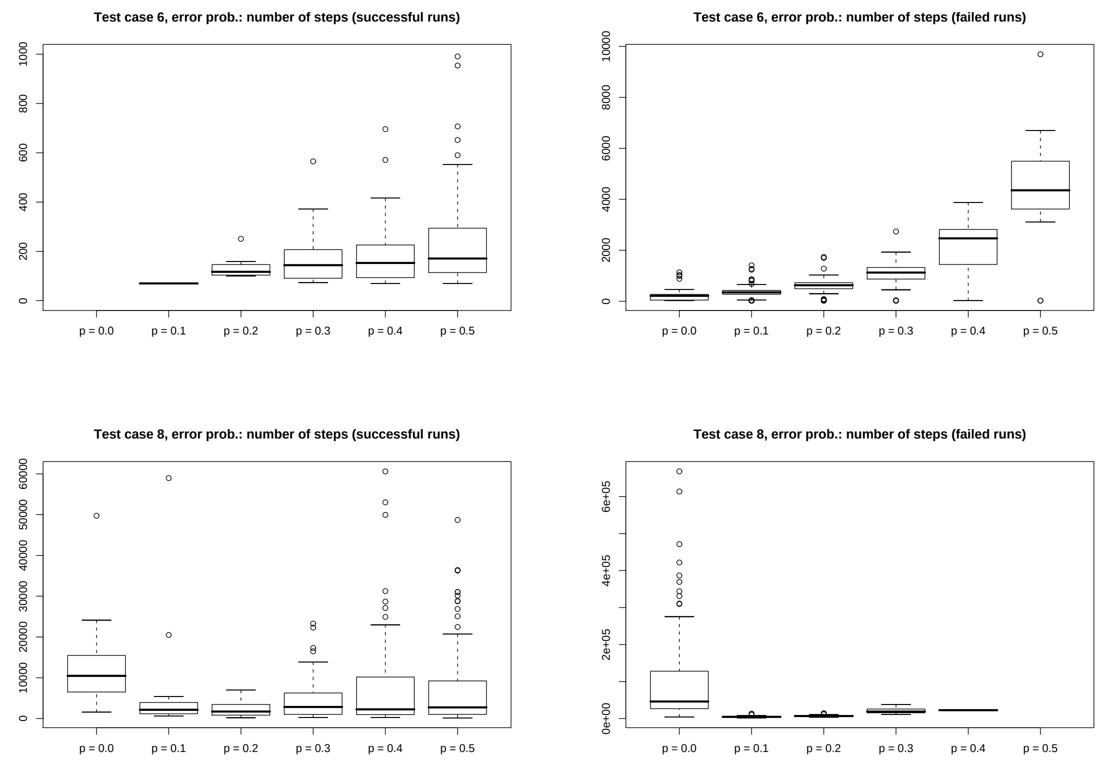


Figure 17: Number of steps through the graph for real-world test cases, `ep`-strategy. Plot layout is the same is in Figure 15. Please note that the upper left plot (successful runs of case 6) is zoomed in, so that some high outliers are not visible.

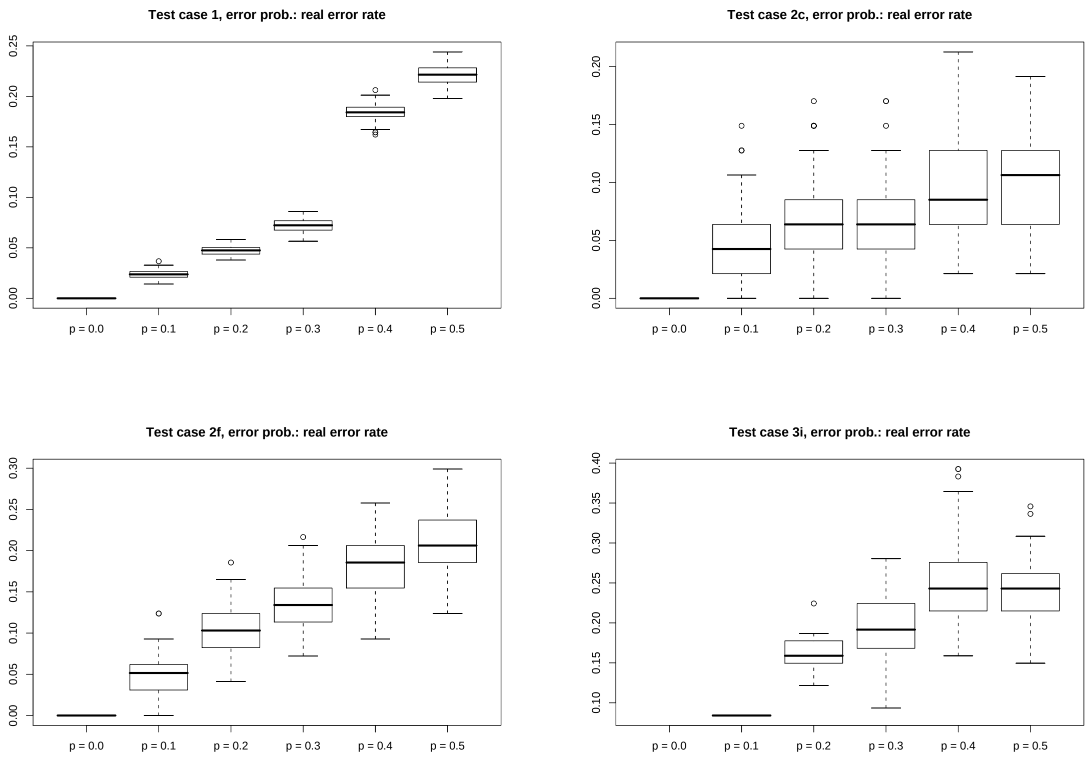


Figure 18: Real error rates, ep-strategy. See text for details on how the error rate is calculated. The real error rates grow with p, but only seldomly with a constant slope.

unconcatenated case 1, the rate first grows nicely linear, but then makes a jump at $p = 0.4$. The reason for this strange behaviour is yet unknown. An easier to explain phenomenon is a kind of saturation effect that occurs in some cases (2c, 3i, 7). The error rate first grows rather steadily, but between the rates for $p = 0.4$ and $p = 0.5$ there is no longer a notable difference. In the beginning of the path search process, there are only legal, yet unused tuples. Therefore, a certain amount of legal tuple choices is inevitable.

For error position analysis, case 1 had a pool size of 150 sequences, so dsc had to make errors in order to be successful. The strategy parameter was set to $p = 0.3$. These errors seem arbitrarily spreaded over all sequences in the pool (Figure 19). Error frequencies for higher positions within the sequences (i.e. closer to the 3'-end, and later in the generation process that tries to find all paths in parallel) seem a little bit higher than for lower positions. Maybe the difference is so small because more errors are *made* later, when there are less legal tuples to choose from, but the other occurrence of any illegally chosen tuple is somewhere at an earlier position, leading to a measured error there, too.

Because in the artificial cases sequences have different length, and in particular the concatenation regions are shorter than the actual sequences, error frequencies have been normalized for these cases by dividing the number of errors in a sequence by the number of n_b-tuples needed for the complete sequence. The resulting histograms clearly support the hypothesis (Figure 19). For the variants of case 2, the first 5 columns show frequencies of illegal tuples that lie completely in the sequences defined in the input file. The remaining columns show the frequencies of illegal tuples that overlap two sequences at the concatenation site. For 3c and 3i, concatenations start with the seventh column. These results confirm our hypothesis that errors tend to accumulate around concatenation sites where paths through the tuple graph have to branch.

Fixed bias

For case 1, two pool sizes, 118 and 214, were examined. Even with strict search strategy allowing no errors, dsc should have no problem to generate 118 sequences. For 214 sequences, setting $p = 1.0$ (legal and illegal tuples are chosen with equal probability) resulted in a real error rate very close to 0.5, so this is a kind of equilibrium point. If dsc would manage to put all legal tuples into the pool, and then add the same number of illegal ones, one should get $2 \times 118 = 236$ sequences. Apparently, dsc does not exploit the full potential for choosing legal tuples so that the equilibrium between legal and illegal tuples is alread reached at 214 sequences. Regarding the real error rates for different settings of p for both pool sizes, a growth with a rather narrow distribution can be seen (Figure 20, left hand plots). The right hand plots in Figure 20 show the medians alone in a scatterplot with cartesian axes. Saturation effects, as discussed above, can clearly be seen.

Because the gc-strategy is applied, the errors are more or less uniformly distributed over the sequences (data not shown), but a clearly higher error concentration at later positions within the sequences (i.e. closer to the 3'-end) can be witnessed for $p \leq 0.2$ (Figure 21). Obviously, dsc had only few legal tuples left when reaching these late positions. For $p \geq 0.5$, errors are so abundant that no position-dependent differences in frequency are visible.

As representatives of the other test cases, only 3c and 3i are shown and discussed here, the other cases show a similar behaviour. The difference between these input scenarios is that violations around the path branching points are tolerated in 3c

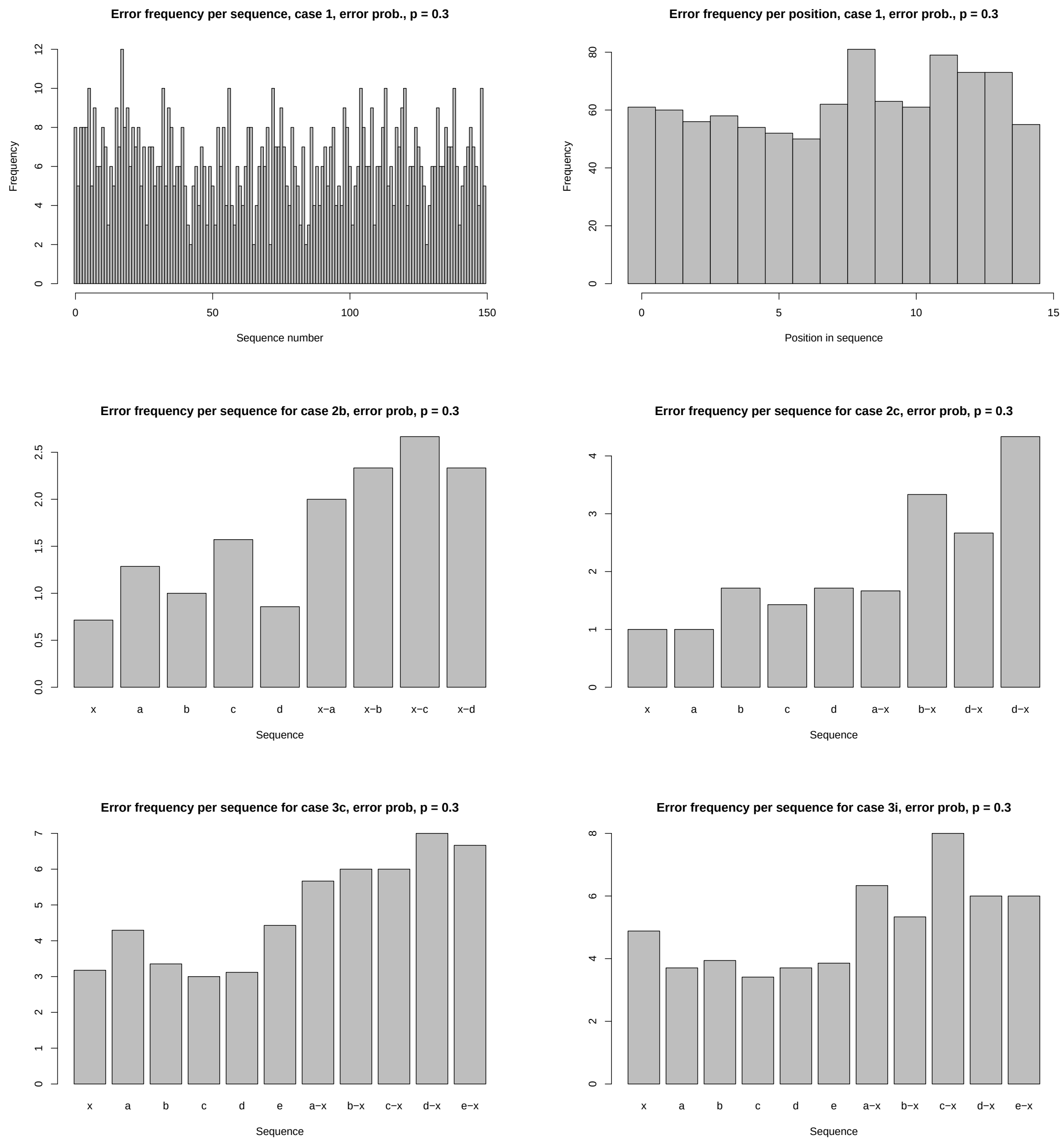


Figure 19: Error positions, ep-strategy. Each histogram shows distributions of positions where multiply occurring tuples are located. The upper right plot shows the error frequency for each position within the 20mers of case 1, all other diagrams show error distributions over the sequences. For the artificial cases, the last few columns correspond to regions comprising tuples that overlap two concatenated sequences. Please note that frequencies for these cases are sequence length normalized (see text).

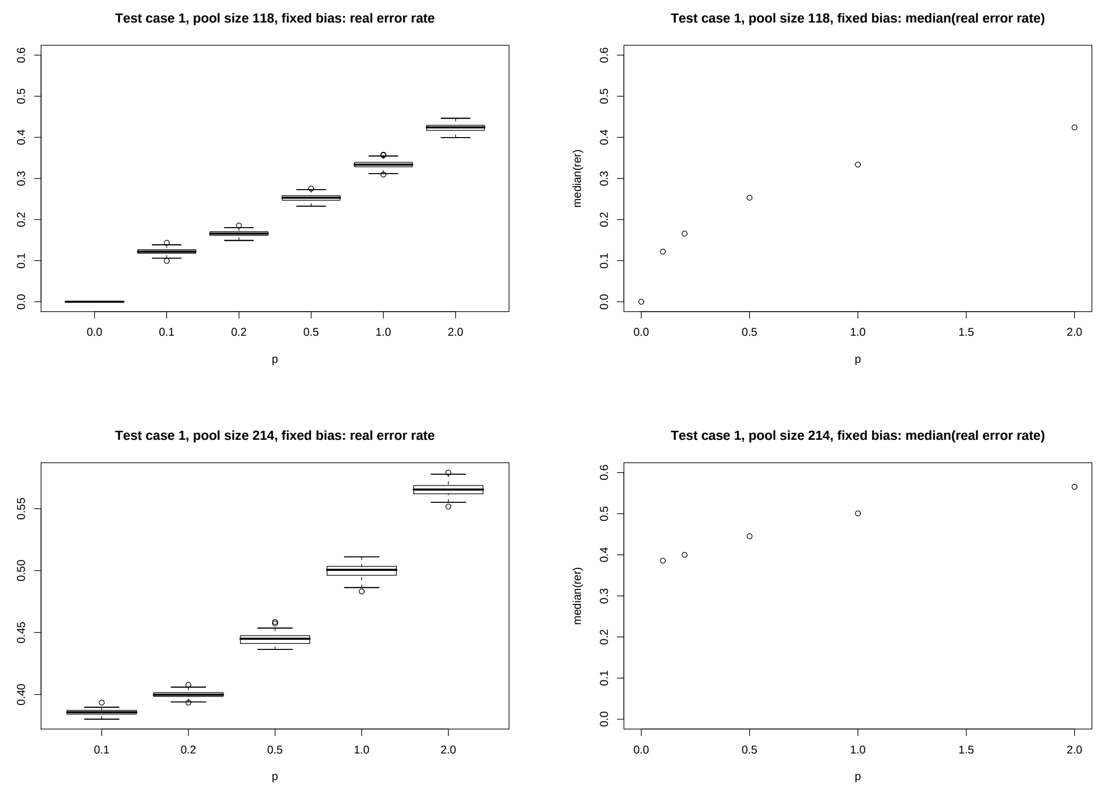


Figure 20: Real error rates, test case 1, pool sizes 118 and 214, fp-strategy. The boxes in the left hand plots show the distributions of all respective 100 runs, the right hand plots show only the medians with a properly scaled abscissa.

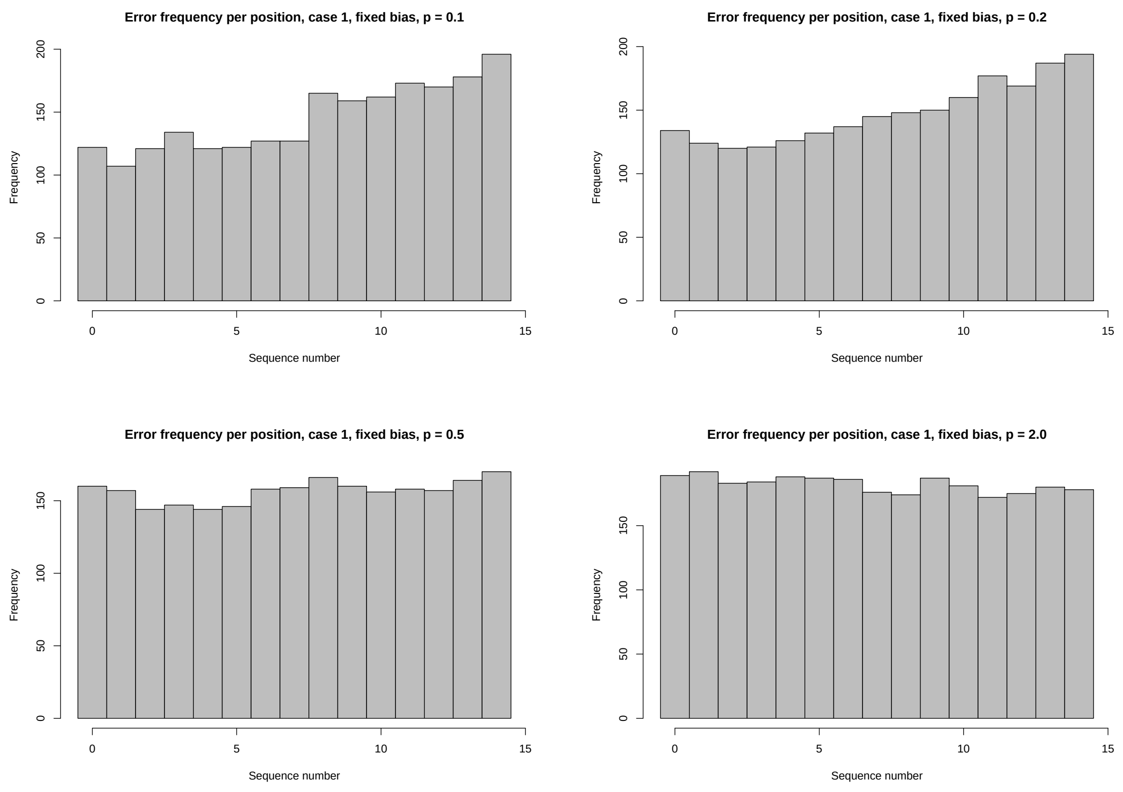


Figure 21: Error frequency per position in sequence, test case 1, pool size 214, fp-strategy. For small p, error accumulate at late positions (top row), for higher p, this difference disappears (bottom row).

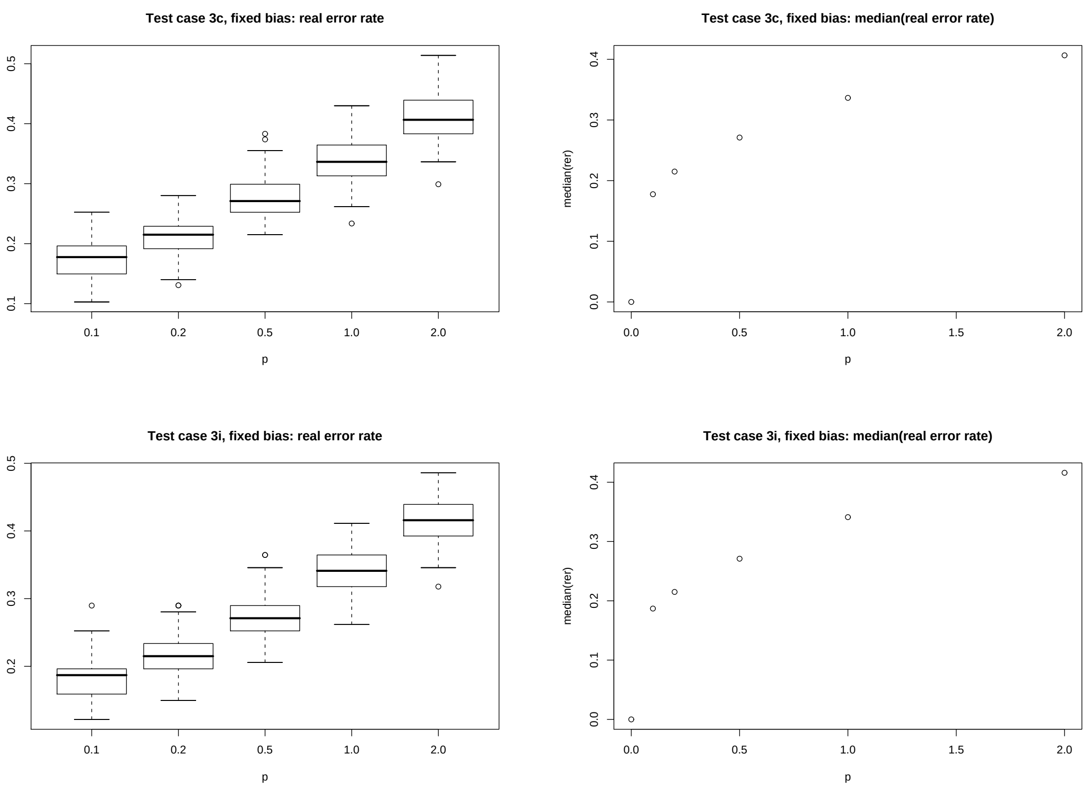


Figure 22: Real error rate, artificial cases, fp-strategy. The plot layout is the same as in Figure 20.

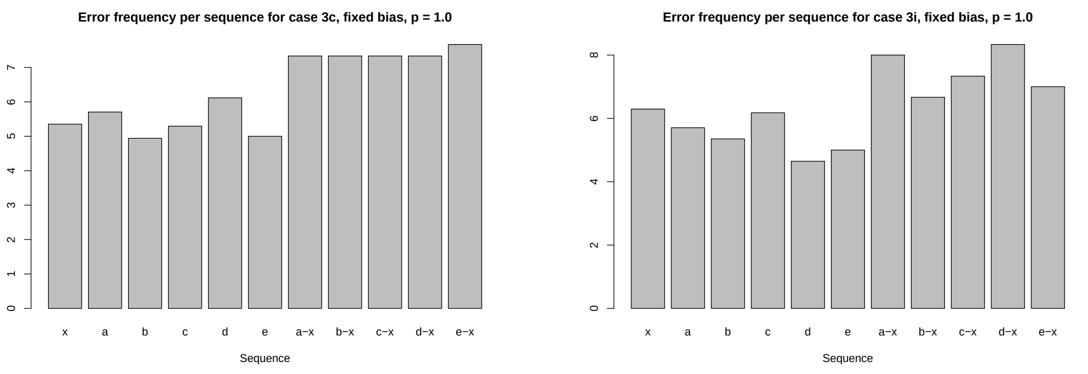


Figure 23: Error frequency per sequence, artificial cases, fp-strategy. Errors tend to accumulate in concatenation regions (left-hand columns). Please note that the error frequencies are sequence-length-normalized (see text).

but not in 3i. As the real error rates shown in Figure 22 clearly demonstrate, this difference does not play a role anymore as soon as the fb-strategy is applied. The left-hand plots also show saturation effects. The error frequency histograms for $p = 1.0$ in Figure 23 again show slightly higher error frequencies in the concatenation regions.

Discussion of ep and fp

Overall, the hypotheses are generally supported, but clearly have to be refined and applied with care. Raising p of course improves the chances for a successful sequence generation, but it is very difficult to predict the influence of a concrete value for p on the actual success probability for a given input scenario. Run time for successful as well for failed runs first decreases with growing p, as expected, but for higher p it increases again. A smaller pressure to use backtracking (decreasing run time) and a larger search space (increasing run time) for growing p have differently strong influence on run time. For small p, the first effect dominates net run time, for greater p, it becomes dominated by the second effect. While the real error rate certainly grows with p, a saturation effect can often be witnessed, where a basic amount of legal tuples seems to be inevitable. For small enough p, errors concentrate around difficult path branching points, but for higher p, errors become more uniformly distributed.

5 Conclusion and Outlook

In conclusion, both newly introduced strategies, sequence grouping for parallel path search as well as allowing some multiple occurrences of tuples, make DNA sequence design with dsc more flexible, since they allow to increase chances for successful sequence generation. But some surprising results contradicting the initial hypotheses, particularly for the real-world input scenarios, demonstrate that it is difficult to give general guidelines for strategy and parameter choice. But at least, some rules of thumb can be derived from the experiments reported here:

- When there are no concatenations, always use the gc-strategy.
- Otherwise, it strongly depends on the actual input case whether g1 or gc has higher chances of success. When in doubt, try both.
- The order in which the sequences or sequence groups are generated can have a striking effect on success probability. Shuffle this order randomly, which is now possible with a new version of dsc (version 3.09).
- If one decides to allow errors, always try the controlled violation tolerance around path branching points first, before changing to less specific strategies ep and fb. This loss of specificity becomes stronger with increasing p.
- The fb-strategy always succeeds in finding all desired sequences, but may amass more errors than necessary. Also, errors will probably accumulate in sequences or regions within sequences that are generated later. It is preferable to try ep with a small p.
- A run with fp and a very small p may at least give a good hint at the rate of errors that is necessary for success.

In all the experiments reported here, no other restrictions, concerning for example melting temperature, GC-ratio or predetermined subsequences were applied.

Adding such requirements may make the prediction of the influence of the presented strategies on success rates, run time or real error rates even more complicated. But a thorough investigation does not seem sensible, since the somewhat inconsistent results show that it is already difficult enough to predict anything without these restrictions. Judging from the author's everyday experience with the software, additional requirements tend to make dsc fail more often, and do so quicker.

Further developments of the design software may improve user-convenience by automizing repetition of path search algorithm runs, with different parameters, different strategies, different sequence orders, following the rules of thumb listed above. Another desirable feature would be a graphical user interface that allows the user to design the targeted spacial structure of the DNA molecules by drawing double- and single-stranded molecules on the screen, with an automized translation of such a drawing into a DeLaNA-description. A more functional development could improve the position specificity of tolerated errors. After one has mananged to successfully generate sequences with the ep-strategy using a certain p, most probably there are a lot of multiple occurrences of n_b-tuples that are not really necessary. In a postprocessing step, dsc could try to get rid of these errors. For example, a simple evolutionary algorithm could mutate the sequences, i.e. replace single nucleotides, in the illegal tuples, measuring whether this improves the error rate or not. Improved sequence sets are then further mutated and so on.

Finally, *in vitro* investigations in whether a certain amount of errors really has measurable effects on the yield and purity of the target structures assembled from the designed sequences would be extremely helpful in choosing sensible search strategies and parameters.

References

[1] Ackermann, J., and Gast, F.-U. Word design for biomolecular information processing. *Zeitschrift für Naturforschung 58a* (2003), 157–161.

[2] Adleman, L. M. Molecular computation of solutions to combinatorial problems. *Science 266* (Nov. 1994), 1021–1024.

[3] Arita, M., Nishikawa, A., Hagiya, M., Komiya, K., Gouzu, H., and Sakamoto, K. Improving sequence design for DNA computing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)* (2000), D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., Morgan Kaufmann, pp. 875–882.

[4] Birac, J. J., Sherman, W. B., Kopatsch, J., Constantinou, P. E., and Seeman, N. C. Architecture with GIDEON, a program for design in structural DNA nanotechnology. *Journal of Molecular Graphics and Modelling 25* (2006), 470–480.

[5] Deaton, R., Chen, J., Bi, H., and Rose, J. A. A software tool for generating non-crosshybridizing libraries of DNA oligonucleotides. In Hagiya and Ohuchi [19], pp. 252–261.

[6] Deaton, R., Murphy, R. C., Garzon, M., Franceschetti, D. R., and Stevens, Jr., S. E. Good encodings for DNA-based solutions to combinatorial problems. In *Proceedings of the Second Annual Meeting on DNA Based Computers, held at Princeton University, June 10-12, 1996* (1996), DIMACS:

Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 159–171.

[7] Deaton, R., Murphy, R. C., Rose, J. A., Garzon, M., Franceschetti, D. R., and Stevens, Jr., S. E. Genetic search for reliable encodings for DNA-based computation. In *First Conference on Genetic Programming (GP'96)* (Stanford University, 1996), MIT Press.

[8] deBruijn, N. G. A combinatorial problem. *Proceedings of the Koninklijke Nederlandsche Akademie van Wetenschappen 49*, 6–10 (1946), 758–764.

[9] Faulhammer, D., Cukras, A. R., Lipton, R. J., and Landweber, L. F. Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academy of Sciences 97*, 4 (Feb. 2000), 1385–1389.

[10] Feldkamp, U. Ein DNA-Sequenz-Compiler. Technical Report of the Systems Analysis Research Group SYS–2/00, University of Dortmund, Department of Computer Science, November 2000.

[11] Feldkamp, U. CANADA: Designing nucleic acid sequences for nanobiotechnology applications. submitted.

[12] Feldkamp, U., and Niemeyer, C. M. Rational design of DNA nanoarchitectures. *Angewandte Chemie International Edition 45* (2006), 1856–1876.

[13] Feldkamp, U., Rauhe, H., and Banzhaf, W. Software tools for DNA sequence design. *Genetic Programming and Evolvable Machines 4*, 2 (June 2003), 153–171.

[14] Ferretti, C., Mauri, G., and Zandron, C., Eds. *Preliminary Proceedings of the tenth International Meeting on DNA Computing, June 7-10, 2004 - University of Milano-Bicocca* (2004).

[15] Frutos, A. G., Liu, Q., Thiel, A. J., Sanner, A. M. W., Condon, A. E., Smith, L. M., and Corn, R. M. Demonstration of a word design strategy for DNA computing on surfaces. *Nucleic Acids Research 25*, 23 (1997), 4748–4757.

[16] Fu, T.-J., and Seeman, N. C. DNA double-crossover molecules. *Biochemistry 32* (1993), 3211–3220.

[17] Garzon, M., Deaton, R. J., Rose, J. A., and Franceschetti, D. R. Soft molecular computing. In *Proceedings of the 5th DIMACS Workshop on DNA Based Computers, held at the Massachussetts Institute of Technology, Cambridge, MA, USA June 14 - June 15, 1999* (1999), E. Winfree and D. K. Gifford, Eds., American Mathematical Society, pp. 91–100.

[18] Garzon, M. H., Deaton, R., Neathery, P., Franceschetti, D. R., and Murphy, R. A new metric for DNA computing. In *Conference on Genetic Programming, GP-97* (Stanford University, Stanford, California, July13–16, 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds. Special Track on DNA computing.

[19] Hagiya, M., and Ohuchi, A., Eds. *DNA Computing : 8th International Workshop on DNA-Based Computers, DNA8 Sapporo, Japan, June 10-13, 2002. Revised Papers* (2003), vol. 2568 of *LNCS*, Springer.

[20] Jonoska, N., and Seeman, N. C., Eds. *DNA Computing, 7th International Workshop on DNA-Based Computers, DNA 2001, Tampa, U.S.A., 10-13 June 2001* (2002), LNCS, University of South Florida, Tampa, FL, Springer.

[21] Kim, D., Shin, S.-Y., Lee, I.-H., and Zhang, B.-T. NACST/Seq: A sequence design system with multiobjective optimization. In Hagiya and Ohuchi [19], pp. 242–251.

[22] Lee, I.-H., Kim, S., and Zhang, B.-T. Multi-objective evolutionary probe design based on thermodynamic criteria for HPV detection. In *PRICAI 2004* (2004), C. Zhang, H. Guesgen, and W. Yeap, Eds., vol. 3157 of *LNAI*, Springer, pp. 742–750.

[23] Penchovsky, R., and Ackermann, J. DNA library design for molecular computation. *Journal of Computational Biology 10*, 2 (2003), 215–229.

[24] Phan, V., and Garzon, M. H. The capacity of DNA for information encoding. In Ferretti et al. [14], pp. 405–415.

[25] Rauhe, H., Vopper, G., Feldkamp, U., Banzhaf, W., and Howard, J. C. Digital DNA molecules. In *Preproceedings of the 6th International Workshop on DNA-Based Computers, DNA 2000, Leiden, The Netherlands, June 2000* (2000), A. E. Condon and G. Rozenberg, Eds., Leiden center for natural computing, p. 271. Poster, manuscript available under http://ls11-www.informatik.uni-dortmund.de/molcomp/Publications/publications.html.

[26] Rose, J. A., Deaton, R. J., Franceschetti, D. R., Garzon, M., and Stevens, Jr., S. E. A statistical mechanical treatment of error in the annealing biostep of DNA computation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)* (1999), W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 2, Morgan Kaufmann, pp. 1829–1834.

[27] Rose, J. A., Deaton, R. J., Hagiya, M., and Suyama, A. The fidelity of the tag-antitag system. In Jonoska and Seeman [20], pp. 138–149.

[28] Rose, J. A., Hagiya, M., and Suyama, A. The fidelity of the tag-antitag system II: Reconciliation with the stringency picture. In *Proceedings of the 2003 Congress on Evolutionary Computation (CEC'03)* (2003), IEEE press, pp. 2740–2747.

[29] Ruben, A. J., Freeland, S. J., and Landweber, L. F. PUNCH: An evolutionary algorithm for optimizing bit set selection. In Jonoska and Seeman [20], pp. 150–160.

[30] Seeman, N. C. Nucleic acid junctions and lattices. *Journal of Theoretical Biology 99* (1982), 237–247.

[31] Seeman, N. C., and Kallenbach, N. R. Design of immobile nucleic acid junctions. *Biophysical Journal 44* (Nov. 1983), 201–209.

[32] Shin, S.-Y., Kim, D.-M., Lee, I.-H., and Zhang, B.-T. Evolutionary sequence generation for reliable DNA computing. In *Proceedings of the 2002 Congress on Evolutionary Computing CEC'02* (2002), IEEE, pp. 79–84.

[33] Shin, S.-Y., Lee, I.-H., Kim, D., and Zhang, B.-T. Multiobjective evolutionary optimization of DNA sequences for reliable DNA computing. *IEEE Transactions on Evolutionary Computation 9*, 2 (2005), 143–158.

[34] Smith, W. D. DNA computers in vitro and in vivo. In *Proceedings of a DIMACS Workshop, held at Princeton University, 4 April 1995, Amer. Math. Soc., 1996* (1995), DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 121–186.

[35] Tanaka, F., Nakatsugawa, M., Yamamoto, M., Shiba, T., and Ohuchi, A. Towards a general-purpose sequence design system in DNA computing. In *Proceedings of the 2002 Congress on Evolutionary Computing CEC'02* (2002), IEEE, pp. 73–78.

[36] Tulpan, D. C., Hoos, H. H., and Condon, A. E. Stochastical local search algorithms for DNA word design. In Hagiya and Ohuchi [19], pp. 229–241.

[37] Tulpan, D. C., Hoos, H. H., Condon, A. E., Shortreed, M., Bong, S. C., and Smith, L. M. Thermodynamically based DNA code design. In Ferretti et al. [14].

[38] Wei, B., Wang, Z., and Mi, Y. Uniquimer: A de novo DNA sequence generation computer software for DNA self-assembly. In *DNA Computing: 12th International Meeting on DNA Computing, DNA12, Seoul, Korea, June 2006. Revised Selected Papers* (2006), C. Mao and T. Yokomori, Eds., vol. 4287 of *LNCS*, Springer, pp. 266–273.

[39] Welsh, D. J. A., and Powell, M. B. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal 10*, 1 (1967), 85–86.

[40] Williams, S., Lund, K., Lin, C., Wonka, P., Lindsay, S., and Yan, H. Tiamat: A three-dimensional editing tool for complex DNA structures. In *DNA 14, June 2-6, 2008, Prague, Czech Republic, Preliminary Proceedings* (2008), A. Goel, F. C. Simmel, and P. Sosik, Eds., Silesian University in Opava, pp. 112–121.

[41] Yan, H., Park, S. H., Finkelstein, G., Reif, J. H., and LaBean, T. H. DNA-templated self-assembly of protein arrays and highly conductive nanowires. *Science 301* (2003), 1882–1884.