An Experimental Study of SBH with Gapped Probes

Benoit Hudson

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-99-07
April 1999

An Experimental Study of SBH with Gapped Probes [*]

Benoît Hudson

April 20, 1999

Abstract

Sequencing by Hybridization (SBH) has been proposed as a means of automating the task of DNA sequencing. The original SBH proposal was able to sequence DNA of length about 2^k nucleotides with a cost $O(4^k)$ (in linear time). Preparata and Upfal have recently proposed a new method, which was predicted to be able to sequence DNA of length $O(4^k)$, still for a cost $O(4^k)$. This work describes a simulation of the method developed to test the theoretical results.

1 Introduction

Sequencing by Hybridization was proposed independently by several research teams (for an overview, see *Pevzner and Lipshutz* [5]). In these proposals, the target DNA is replicated and presented to a so-called chip which contains all strings of DNA of k nucleotides; each of these fragments is termed a *probe*. The probes will hybridize to the target if a substring of the target is the Watson-Crick complement to the probe; the probes which do hybridize are said to have fired. From the set of fired probes, we know the set of overlapping substrings of the target. The problem of reconstructing these into the target sequence has been well studied: the set of fired probes induces a graph, and reconstructing the sequence corresponds to traveling an Eulerian tour of said graph.

Using this method, sequences of length up to about 2^k can be sequenced. In practice, k is usually 8 or 10 because of the physical limitations of putting 4^k probes on a chip. The technique can therefore be applied to DNA sequences up to 1000 nucleotides long; [4] gives a "realistic" limit of 600 nucleotides.

Preparata, Frieze, and Upfal [6] describe a gapped chip which greatly outperforms the classical chip. In a gapped chip, the probes are made such that only one nucleotide can hybridize at some of the positions on the probe, while at others, any of the four can hybridize. Thus, we have a probe where some of the bases are known, and others are "don't cares", or gaps. The chip uses gapped probes which have first s contiguous bases which we term the *body*,

[*]This work was done under the supervision of Franco Preparata and Eli Upfal.

followed by r non-contiguous bases which are separated by $s-1$ bases from each other, collectively termed the *tail*. Within this family of chips, a chip is denoted as a (s, r) chip, and the cost parameter, k, is defined as $k \equiv s+r$. Each probe is $s+rs$ bases long, but only $s+r$ vary on the chip. Note that the $(k, 0)$ chip and the $(1, k-1)$ chip are identical to the classical chip $C(k)$.

```
acag...g...t...c
```

Figure 1: Example $(4, 3)$ probe. Dots are don't-cares.

The cost of this chip remains $O(4^k)$: each probe is longer than in the classical chip, but there are exactly as many as before. The advantage of this method becomes evident when we consider the reconstruction algorithm. At the highest level of abstraction, the algorithm starts with a certain substring of the sequence, and tries to extend it by one character at a time, until it has reached the end of the sequence. In doing this with a classical probing scheme, we span $k-1$ characters of the known substring to find the next character: we have no probes which span the new character and any part of the string before that. With an (s, r) probing scheme, we span $t \equiv s+rs-1$ nucleotides. Although no single probe covers that many bases (in fact, each covers at most $k-1$ known bases), we can use all the information by using several probes at once. This increased power allows us to apply the technique to sequences asymptotically $O(4^k)$ bases long.

1.1 Overview

This paper describes an implementation of the algorithm, a simulation package, and experimental results. First, in section 2, we outline the reconstruction algorithm and provide extensive pseudo-code. Sections 3 and 4 describe the implementation of the simulation. Section 5 documents some of the results gotten using the simulation package. Finally, section 6 offers user manuals for all programs in the simulation package.

2 Reconstruction Algorithm

The reconstruction algorithm for the Preparata-Upfal technique is quite simple in concept. Unlike in the case of classical chips, which cleverly translates the problem to the graph traversal domain, we remain in the text processing domain. The main loop of the algorithm simply takes a leading known substring of the target sequence, and extends it by one character. This is done until it becomes impossible to extend it anymore, at which point success is declared. It may also be the case that there is more than one character with which we can extend; if so, the algorithm fails and the sequence is said to be ambiguous (since it is not unambiguously reconstructible). Almost all of the complexity of the algorithm lies in trying to reduce the number of possible extensions.

To extend a given known substring, we consider the final t-gram of the string. A probe is said to extend the string if it matches the first s characters of the t-gram in all s characters of its body, and every sth character thereafter by the corresponding character of its tail. The last character of the tail is the extension character, since it is just beyond the known part of the string, and may indeed be the character which extends the known substring by one base. If there is only one such probe, then we know exactly which character is the only one which extends our substring. If there is no such probe, then we know we have reached the end. A difficulty arises if there are multiple such probes, each with a different possible extension character.

To handle this possibility, we "shift" the probes. A probe extends the string with shift (or offset) i if the first s characters of the body match the s characters starting at offset si in the t-gram, and an appropriate number of the tail characters match corresponding characters in the t-gram; and further, that the characters of the tail which do not correspond to any characters in the t-gram match a corresponding character in some probe which extends the string with offset $i - 1$. The base case is offset 0, which corresponds exactly to our previous definition of extension.

```
acggaactcggttacX
acgg...t...t...X
    aact...t...X...c
        cggt...X...c...t
            tacX...c...t...g
```

Figure 2: In a $(4, 3)$ probe, four probes can be used to check the extension character, denoted here by '`X`'; the four probes match appropriate characters in the t-gram at top, as well as appropriate characters in the other probes. Dots are don't-cares.

There is a boundary condition we have to contend with at the start of sequencing. The simulation results assume that we are given a primer—the first t-gram in the sequence. This can be achieved by prepending a known sequence to the target before replicating it. Another option is to pick an arbitrary probe, and extend using only the body, until we have t contiguous characters. This will often give several different possibilities; we choose the one which yields the longest sequence.

The simulation also appends randomly chosen characters at the end of the sequence. This was necessary in a previous version in order to handle a boundary condition at the end. The algorithm described here should not have that boundary condition; however, the simulation continues to attach an ending primer. Some testing should show whether it is still necessary; if so, there is probably a bug in the simulation code (but probably not in the algorithm code).

2.1 Pseudo-Code conventions

The pseudo-code presented below mostly adheres to the conventions in [2]. In particular, all calls are by value: modifying a parameter or any part of the parameter has no effect from the caller's point of view. All variables are local, except for the follwing global values: M, an array of sets of probes; s, r, and t, which retain their meanings in the discussion above; h, which will be defined in section 2.3; and P, the set of fired probes. We depart from the standard convention in having 0-based arrays. This is largely an artefact of the implementation being in C++.

M is an array $M[0..r]$ of sets of probes; each M_i contains the probes which extend the sequence with a shift i—they match with the t-gram in the body and the appropriate number of characters of the tail, and match with some probe in M_{i-1}.

2.2 Pseudo-Code

SEQUENCE is the top-level algorithm. It simply proceeds by building a dictionary from the set of fired probes P, then repeatedly extending the known substring, starting from the primer, until we can no longer extend.

```
SEQUENCE($P, primer$)
 1  $seq \leftarrow primer$
 2  BUILD-DICTIONARY($P$)
 3  while GETEXTENSION($seq$) succeeds
 4     do $e \leftarrow$ GETEXTENSION($seq$)
 5        $seq \leftarrow seq + e$
 6  switch cause of failure
 7     case no extensions found :
 8          return $seq$
 9     case multiple possible extensions found :
10          error "multiple extensions"
```

GETEXTENSION returns the single character which unambiguously extends the sequence, or an error if there was no such character.

```
GETEXTENSION($seq$)
 1  $i \leftarrow$ FILL-M($seq$)
 2  if $size[M_i] = 0$
 3     then return no extensions found
 4     else if UNIQUEEXTENSION($i$)
```

```
            then return  the extension character
            else  return  multiple extensions found
```

FILL-M is where we do most of the work. Its output is the index of the array M of sets of fired probes.

```
FILL-M($seq$)
tgram ← FINALTGRAM($seq$)
for $i \leftarrow 0$ to $r$
    do $M_i \leftarrow \emptyset$
        $A \leftarrow$ LOOKUP($tgram, i$)
        for  each $p \in A$
            do if CHECKINM($p, i$)
                   then $M_i \leftarrow M_i \cup p$
        if $size[M_i] \leq 1$ or UNIQUEEXTENSION($i$)
           then  ▷ The extension character is unambiguously in $M_i$
                 return $i$
 ▷ The extension character is ambiguous
return $r$
```

```
UNIQUEEXTENSION($i$)
   ▷ Return whether all probes in $M_i$ have the same extension character
```

CHECKINM returns whether the probe p matches at least one probe in M_i.

```
CHECKINM($p, i$)
if $i = 0$
   then return true
for  each $p' \in M_{i-1}$
     do if CHECKP'($p, p', i$)
              then return true
 ▷ If we are here, no pair $p$ and $p'$ matched
return false
```

CHECKP' returns whether the probe p (in M_{i-1}) and p' (in M_i) are compatible.

```
CHECKP'($p, p', i$)
if $i < r$
   then $start \leftarrow (r-1) - i$
   else  $start \leftarrow 0$
            if $body[p][s-1] \neq tail[p'][0]$
               then return false
for $j \leftarrow start$ to $r-2$
    do if $tail[p][j+1] \neq tail[p'][j]$
             then return false
 ▷ If we got here, no characters were different
return true
```

LOOKUP returns all probes which match the t-gram with shift i. This is an extremely naive implementation; the simulation cuts down the number of comparisons needed dramatically by using a hashtable keyed on $s-1$ characters, then checking linearly the remaining characters. Much better could be done with a trie.

```
LOOKUP($tgram, i$)
 1  $R \leftarrow \emptyset$
 2  for  each $p \in P$
 3      do  ▷ Check whether the body matches the $t$-gram
 4          for $j \leftarrow 0$ to $s-2$
 5              do if $body[p][j] \neq tgram[si+j]$
 6                    then next $p$
 7          if $i < r$
 8             then if $body[p][s-1] \neq tgram[si+s-1]$
 9                     then next $p$
10           ▷ Check whether the tail also matches
11          for $j = 0$ to $r-i-2$
12              do if $tail[p][j] \neq tgram[si+2s+js-1]$
13                    then next $p$
14           ▷ Once here, we know $p$ matches $tgram$ in all positions
15          $R \leftarrow R \cup p$
16  return $R$
```

2.3 Extended Algorithm

The original algorithm fails if, after r shifts, we still cannot disambiguate the extension character. When this situation occurs, we say that we have found a branching point; only one of the possible characters to append can be the correct one, and all other possibilities are spurious. Declaring failure in this case is correct: there are now at least two possible reconstructions of the sequence—one which is correct, and the other which uses the spurious branch. Without any other *a priori* knowledge of the system, we cannot distinguish between the two possibilities.

However, the probability that a spurious branch is longer than $h > t$ characters long falls exponentially with h. Therefore, one heuristic which improves the performance of the algorithm is to try to extend using all of the possible extensions, to a limit of h characters. We expect that only one of them (the correct branch) can be extended to that length, and that the spurious branches will die off quickly. Only if more than one branch can be extended h characters do we declare failure. If $h < t$, then the extra work expended is wasted: we have found extensions to exactly length t, so we know that all the possible extensions are warranted to this length.

The danger with this heuristic is that, in very rare cases, we return an incorrect sequence. In particular, if we are within h characters of the true end of the sequence when we find a branching point, and a spurious branch can be extended h characters, we will follow the spurious branch. When it ends, we will report the spurious path as the correct one. Another (even less likely) possibility is that we have a loop near the end. A loop occurs when we have a t-gram which can be extended by two possibilities: one of them leads to the end of the sequence, and the other leads to an extension which eventually yields the same t-gram again. If a loop occurs within the last h characters, the algorithm will dismiss the ending as a spurious path, and will instead follow the loop. This will happen infinitely (or rather, until the machine runs out of resources).

We take two measures to help reduce the probability of an error. If we are near the end of a sequence, and we have a branching point, it is possible that no extension can reach h characters. In this case, we return that the extension was ambiguous. In the case of an infinite loop, we declare failure if the length of the sequence exceeds the estimated length of the target. We can get, biochemically, an estimate within a certain error bound. This technique will work as long as the estimate is an overestimate. In the simulation, we use an estimate of twice the actual length.

2.4 Pseudo-Code

The extended algorithm only requires some minor changes to a few functions. The parameter h is assumed to have been set at the same time as the parameters s and r. The special case $h = 0$ is the original algorithm.

```
GetExtension($seq$)
 1  $i \leftarrow$ Fill-M($seq$)
 2  if $size[M[i]] = 0$
 3     then return  no extensions found
 4     else  if UniqueExtension($i$)
 5              then return  the extension character
 6              else  if $h = 0$
 7                       then return  multiple extensions
 8                       else  $best \leftarrow \epsilon$
 9                             for each  extension $c$
10                                do if PeekForward($h, seq + c$)
11                                     then if $best \neq \epsilon$
12                                             then $best \leftarrow c$
13                                             else  return  multiple extensions
14                             if $best = \epsilon$
15                                then return  multiple extensions
16                             return $best$
```

PeekForward takes the depth to which it is still allowed to extend, and a primer of the last $t - 1$ characters of the known substring, plus a possible extension character. If we have extended past $hleft$ characters, then this extension character is still a candidate; otherwise, it is to be eliminated from consideration.

```
PeekForward($hleft, primer$)
 1  $seq \leftarrow$ FinalTgram($primer$)
```

```
2  while GetExtension($seq$) succeeds and $hleft > 0$
3     do $e \leftarrow$ GetExtension($seq$)
4        $seq \leftarrow seq + e$
5        $hleft \leftarrow hleft - 1$
6  if $hleft = 0$
7    then return true
8    else return false
```

Sequence, finally, has to take into account the possibility of an infinite loop.

```
Sequence($P, primer$)
 1  $seq \leftarrow primer$
 2  $maxlen \leftarrow 2m$ ▷ Use some overestimate of the length
 3  Build-Dictionary($P$)
 4  while GetExtension($seq$) succeeds
 5     do $e \leftarrow$ GetExtension($seq$)
 6        $seq \leftarrow seq + e$
 7        if $length[seq] > maxlen$
 8           then error "infinite loop"
 9  switch cause of failure
10     case no extensions found :
11          return $seq$
12     case multiple possible extensions found :
13          error "multiple extensions"
```

3 Simulation Architecture

The simulation is a suite of related programs. One set of programs prepares a sequence or a number of sequences to use to test the algorithm; one other program—`gap`— implements both splitting the target into the set of fired probes and reconstructing the original sequence from the fired probes, and outputs some information regarding the success of the operation; a final set of programs interpret the data. See section 6 for specifics regarding the use of the programs. This section deals mainly with the implementation philosophy of the `gap` program and the related scripts.

The main program (`gap`) was written in C++, the other programs in either Perl or C.

3.1 Overview of the program suite

The programs `my_seqgen` and `proc_dna` both produce as output a list of sequences in the format used in the simulation. `my_seqgen` produces random sequences, while `proc_dna` splits a file describing a long sequence into a set of non-overlapping sequences.

The program `gap` implements the SBH scheme. It reads a file containing a set of sequences, and attempts to sequence each one. Section 4 discusses the implementation at greater length.

Programs `tester.pl` and `real_dna.pl` are scripts that generate sequences (randomly in the first case, from a file describing real DNA in the second case), have `gap` attempt to sequence them, and interpret the success rate and some other statistics.

Finally, a set of scripts in the `graph` directory interpret and plot the results from `tester.pl` and `real_dna.pl`.

3.2 Design philosophy

The design is intended to allow easily testing of very large numbers of sequencing, in various configurations. The package is intended to be general enough to support various analyses of the program and algorithm, and to allow adding extensions with relative ease.

Internally, there is a very clear separation of the setup code and the algorithm code. The setup code simulates the SBH chip: it is presented the actual sequence, and gives the algorithm just the set of fired probes. Great care has been taken to ensure the algorithm code does not "cheat," or use information that would not be available in a real implementation. In fact, the algorithm code is intended to be usable with only very minor changes in a real implementation of the method.

4 Implementation details

Most of the code for `gap` is well-documented. This section should serve as an introduction to the organization of the code. It is split into two functional objects: the setup code, invoked from the mainline and entirely in `main.C`; and the algorithm code, invoked from the setup code and entirely in the `Algo` class.

In addition, we have four utility classes. One, `Pattern`, represents an (s, r) probe as two strings (one s characters long, the other r characters long). Both setup and algorithm code use this class to represent a fired probe. The other two, `Hashtable` and `Array`, provide data structures used in the program. Finally, `SGramHash` is a specialization of `Hashtable`, its hash function defined by the first $s - 1$ characters of the key `Pattern`.

4.1 Setup Code

The setup code deals with the work of reading in an input file which contains a set of sequences. A sequence is represented as a string of characters in $\{a, c, g, t\}$. It immediately translates this to a string of bytes [0123], which simplifies the hash functions for hash tables keyed by a probe; then appends t random bases. The translated sequence is then used to create an array of `Pattern`s, taking care not to allow duplicates. This array is passed on to the algorithm, along with a copy of the first t characters of the sequence (it is copied to reassure ourselves that the algorithm only uses the first t characters). Finally, h—the number of characters by which to extend in the extended algorithm—is also passed along. By default, $h = 2t$, but the user can set it differently, notably to $h = 0$, which disables the extended algorithm. Having passed this information along, it lets the algorithm run. Once the algorithm has finished running, the setup code outputs some information which was collected during the run of the algorithm. Namely:

- The length of the input.
- The length of the output.
- Whether the algorithm claims to have succeeded.
- Whether its claim is correct—the sequence output by the algorithm is identical to the one input to the program.
- How many extensions took 0, 1, ..., r shifts before they were resolved.

4.2 Algorithm Code

The algorithm code is responsible for implementing the algorithm, as described in section 2. The code is in fact almost a transliteration, as the code and algorithm were developed in parallel. It is in an object of its own, `Algo`, in order to better encapsulate all of the algorithm, and disallow having the setup code and algorithm code depend on each other excessively. As we have mentioned, the intent is to have the algorithm code work equally well whether in the simulation or in a real application.

As input, `Algo` gets both the set of fired probes, and the initial t-gram. It also gets a buffer into which to write the sequence, merely for the sake of speeding up the simulation: it could just as easily allocate itself its own buffer, but dynamic memory allocation was a performance bottleneck in the simulations.

Note that the array gotten from the setup code is in fact in order of appearance of the probes in the sequence. However, the algorithm does not use this information. Similarly, the precise length is available, but the algorithm does not use it. To know whether we have fallen into an infinite loop, however, we use an estimate of m. As implemented, that estimate is $2m$, but any value greater than m could be used instead.

The set of fired probes is put into a dictionary to find appropriate probes in FillM. In this implementation, we use hashtables keyed by the first $s-1$ characters of the body. We only use $s-1$ characters because of the need to shift: after r shifts, the extension character is the last character of the body. This forces us, when we know more characters, to do a linear scan over the probes which match the first $s-1$ characters and eliminate those that do not match further characters. Ideally, we would use a dictionary which allows searching by a variable-length key such as a trie. The limitation of using only $s-1$ characters is most clear when $s=1$; here, the hashtable degenerates to a linked list.

We then proceed to the main loop of the algorithm, extending the initial given t-gram one character at a time. We also do the additional step of keeping track of how many shifts each extension required. The algorithm implemented is the extended one, although this is disabled if h (named `peekLimit_` in the code) is set to 0.

Having run the algorithm, we set a few variables relating which can be used to see if the algorithm failed, what the sequence it output was, and so on.

4.3 Other Programs

To prepare input for `gap`, either `my_seqgen` or `proc_dna` is used. The former (whose name is modified from the more flexible program `seqgen` by Belyi and Pevzner [1]) prepares a random sequence, while the latter takes a given, long sequence and cuts it into non-overlapping substrings of a given length; it is used to process real DNA.

The output is interpreted in a rudimentary fashion by `interp`. More useful are the scripts `tester.pl` and `real_dna.pl` which feed `gap` ever-larger sequences, and collect statistics for each sequence length they tried to have `gap` reconstruct. These can then be used by the `graph` suite of scripts to output the graphs shown in the following section.

5 Experimental Results

Several tests were run using the simulation package to gauge the value of the method and check that the theoretical results were borne out in practice. Some of the data were acquired using randomly generated DNA, others using real DNA. This section describes the results from these tests. Two main quantities were measured: the probability of successfully sequencing DNA of a given length (that is, the quality of the method), and the number of shifts done when sequencing that DNA (that is, the time cost of the method). The former quantity is the most important metric for the quality of this sequencing method, since the processing time is fairly small: from milliseconds to about half a minute on a desktop PC.

5.1 Data Acquisition

The data points for these graphs are generated by setting m to a given value, generating a number of sequences of length m, and calculating what percentage of the sequences were correctly reconstructed by the algorithm. The curves have points at exponentially (base 1.05) increasing m: they start at $m = 40$ and increase m by 5% per point. The exceptions are some of the classical chip examples, where they start at $m = 20$, and figure 8 b, where the step value is 1% per point.

The results obviously depend on several parameters in addition to the sequence length m: namely, s, r, and h. They also depend on the sequence itself: the analysis of the algorithm is a probabilistic one, and some sequences will be harder or impossible to sequence with a given chip.

The randomly generated DNA is generated using `my_seqgen`. Each data point is 1,000 sequences, except the $k = 10$ graphs, where each data point is only 250 sequences. The "uniform" data set has the distribution of each base be uniform; the "skewed" data set has the probability of a being 0.4, and that of the other three bases each 0.2. Both data sets were regenerated for each data point, separately for each graph.

The real DNA is taken from the National Center for Biotechnology Information genome data bank [3]. Sequences were gotten for the bacteria *Escherichia coli*, *Haemophilus influenzae*, and *Methanobacterium thermoautotrophicum.* These are respectively 4.5, 1.8, and 1.7 million bases long. These data were split into non-overlapping substrings of the given length; the number of sequences therefore varies according to the sequence length. Note that in the *H. influenzae* genome, some of the bases were not yet known, and denoted by letters other than $\{a, c, g, t\}$. In the processing which split the sequences, these unknown bases were ignored.

5.2 Probability of Success

The graphs in this section describe the probability that a given chip will succeed in sequencing the target DNA. The goal is to discuss the tradeoff of cost (chip size) to quality (success probability). The cost is assumed to be $O(4^k)$—that is, the chip area depends only on the number of probes.

Each graph is for a given value of k and h, and given data set (uniform, skewed, or real). Each curve on each graph is for a given (s, r). Note that the $(k, 0)$ and $(1, k-1)$ chips are identical (they both are the C_k classical chip), so only the $(k, 0)$ chip is displayed on these graphs. The top figure has $h = 0$; the bottom has $h = 2t$.

The graphs (figures 4 to 23) make it clear that the algorithm does in practice almost exactly as well as had been theoretically expected for uniformly distributed data; figure 3 is especially striking in that perspective. However, they also make it abundantly clear that real DNA is not uniformly distributed, as the curves for real DNA far below those for random DNA. Still, this technique does several orders of magnitude better than previous techniques. With $k = 8$, the classical chips claimed to be able to sequence length of up to about 600. The new method can get that far on real DNA with $k = 7$.

Note that in the $k = 7$ examples, some of the classical chip curves do not show up, because already at $m = 40$, the success rate is below 90%. This is the reason some of the classical chips were run starting at $m = 20$.

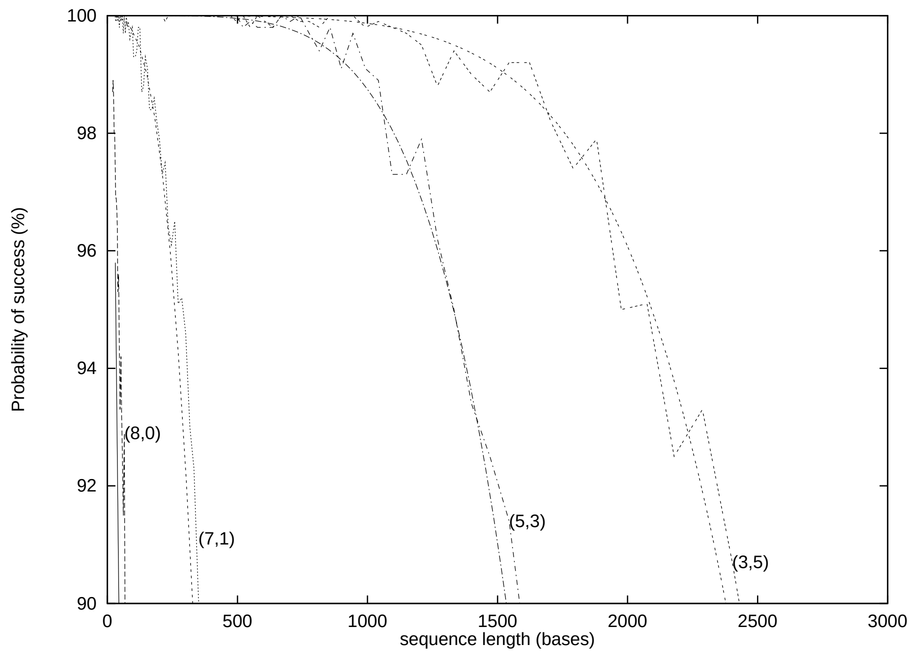
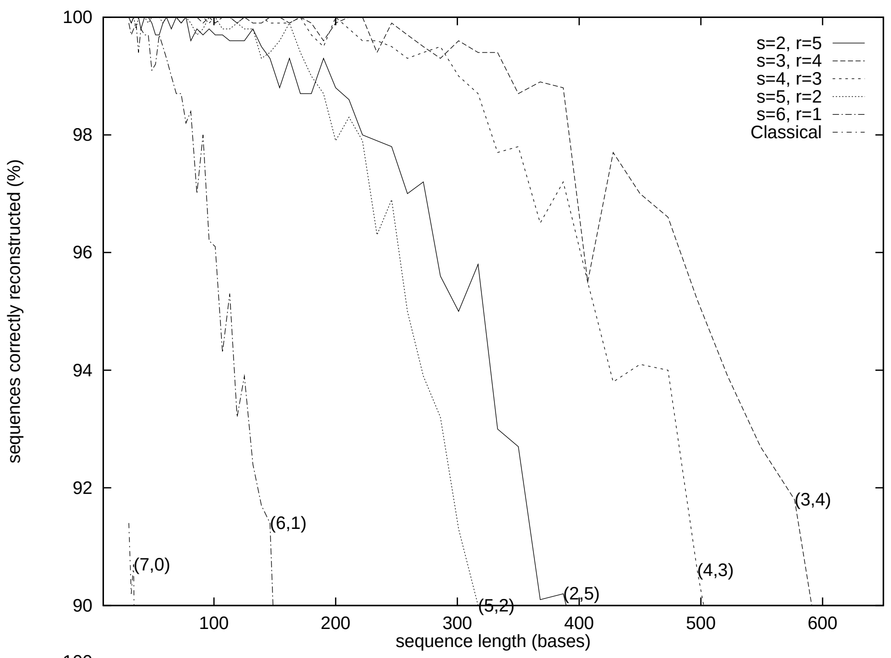


Figure 3: Theoretical curves for uniformly distributed data, overlaid on experimentally derived curves, $k = 8$, basic algorithm.



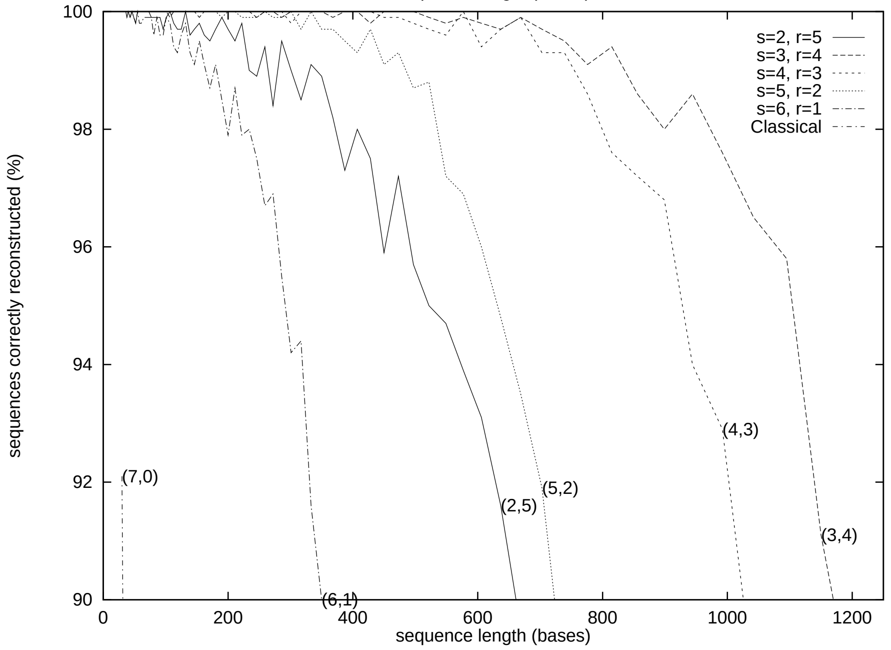
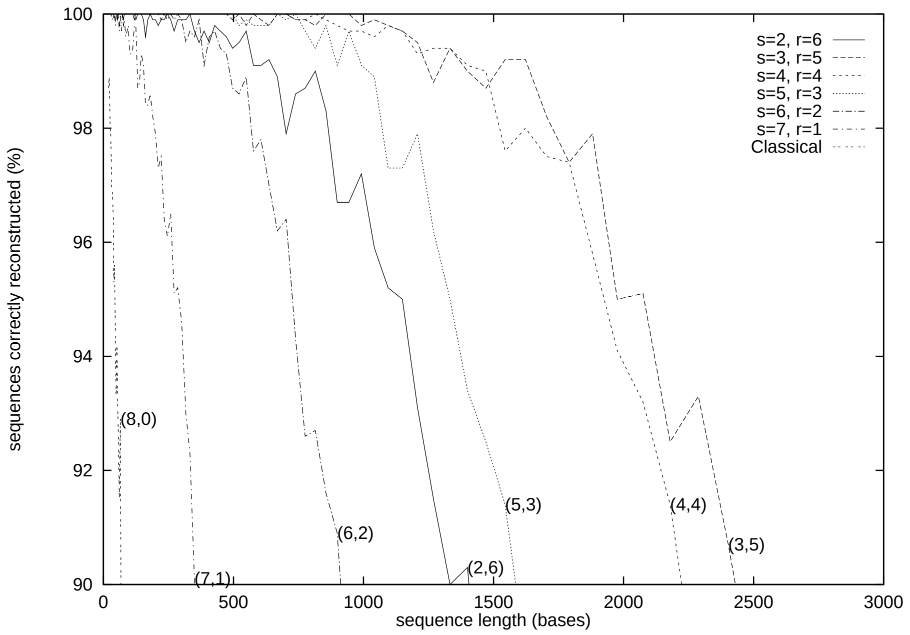


Figure 4: uniformly distributed data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



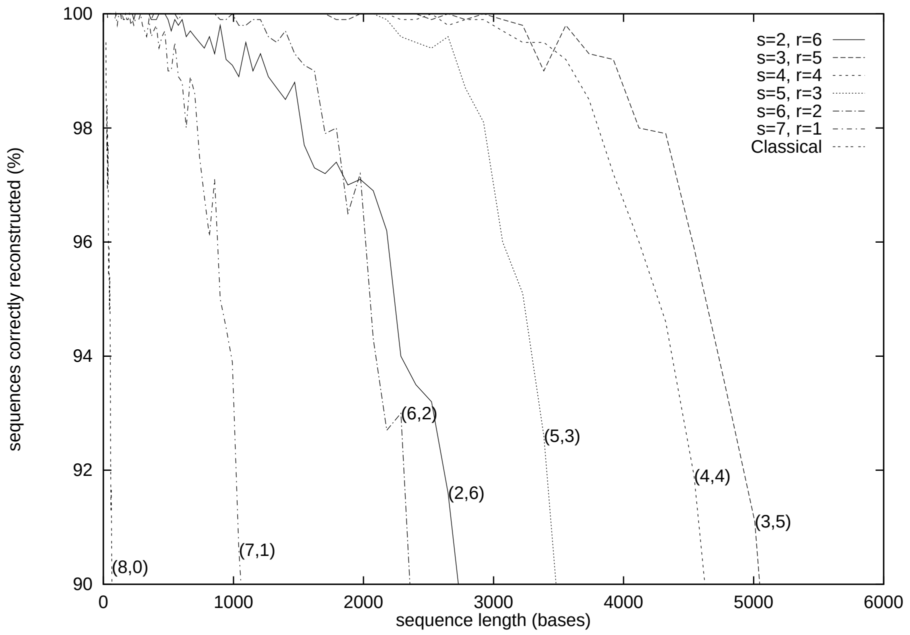
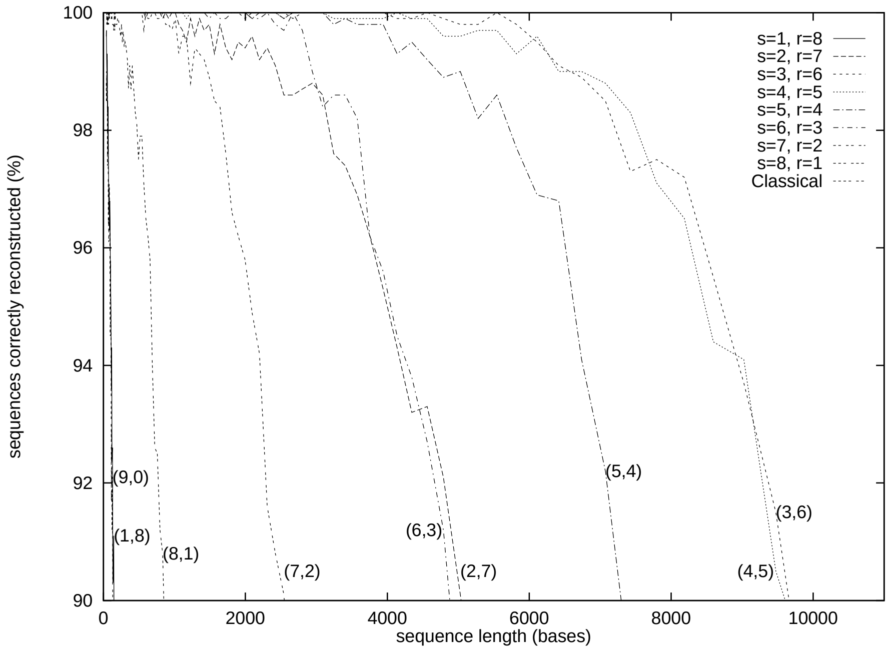


Figure 5: uniformly distributed data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



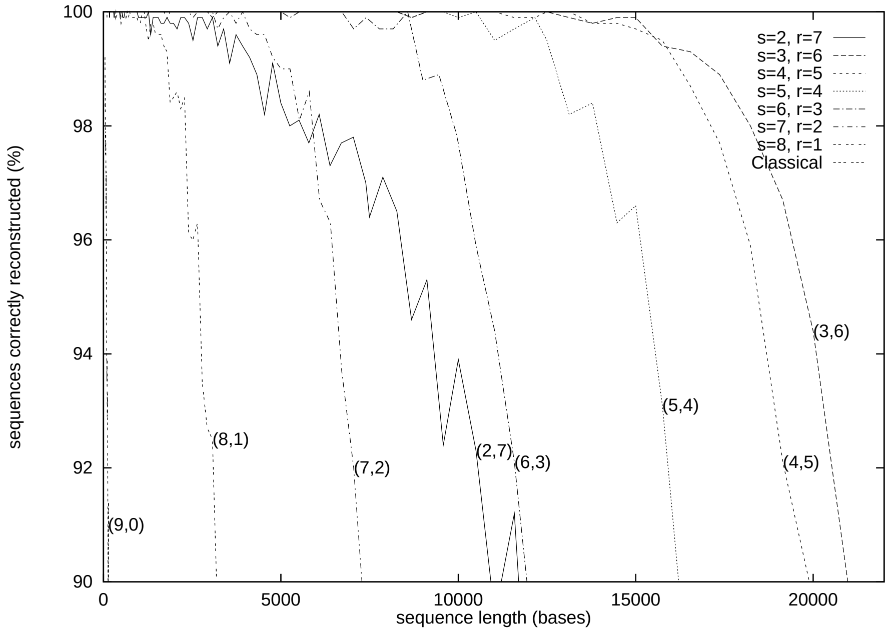
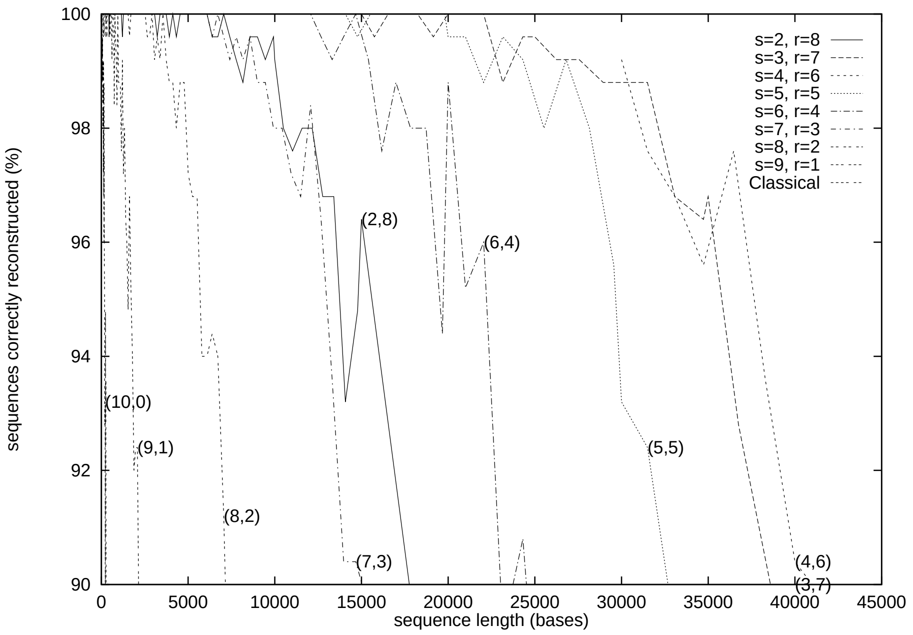


Figure 6: uniformly distributed data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



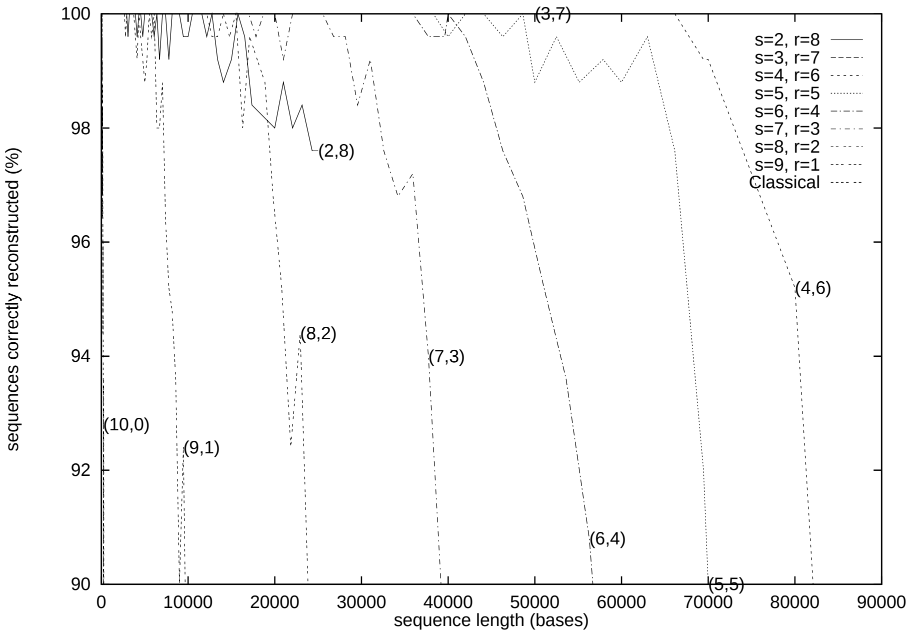
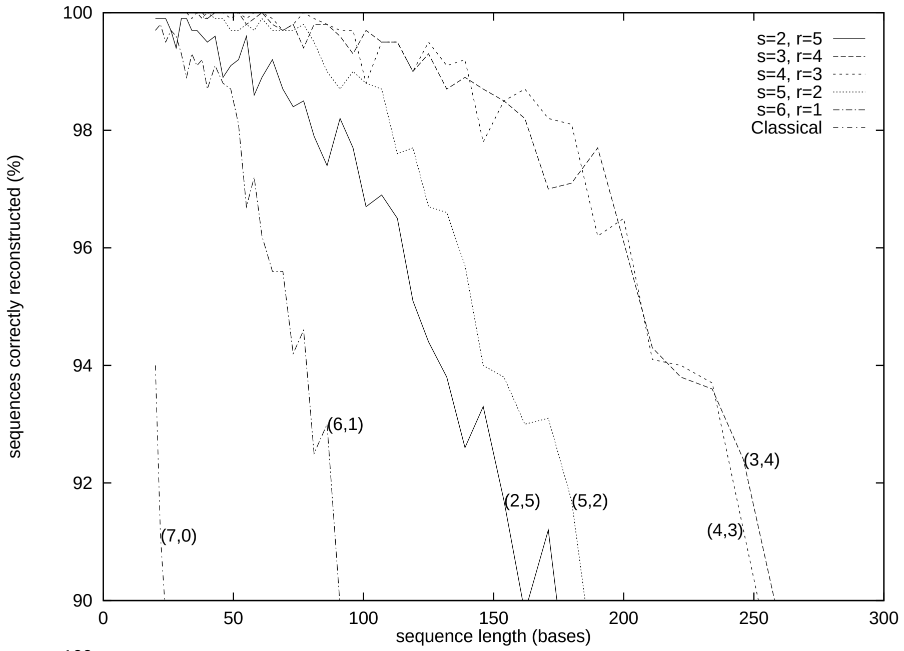


Figure 7: uniformly distributed data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



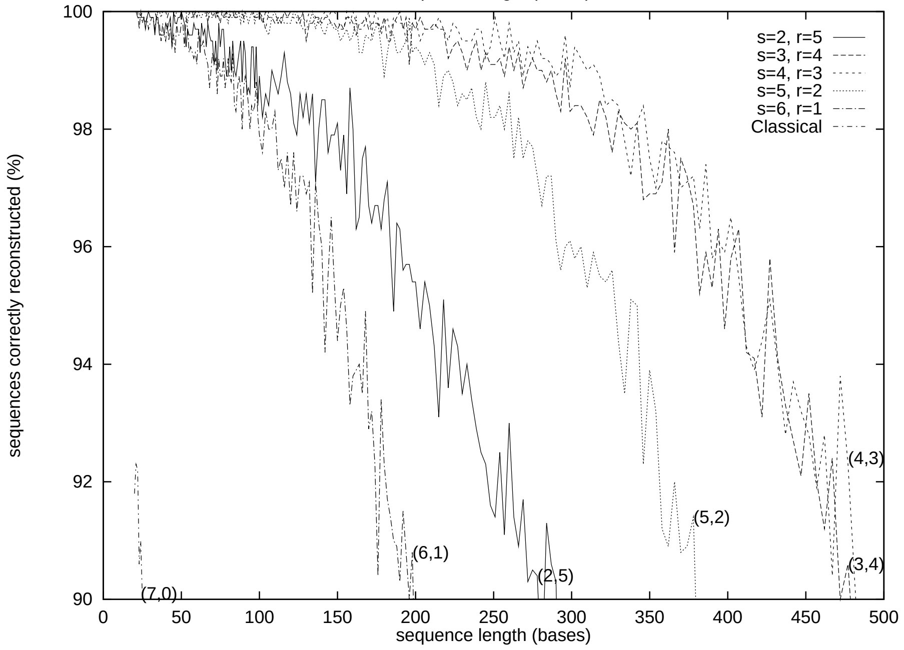
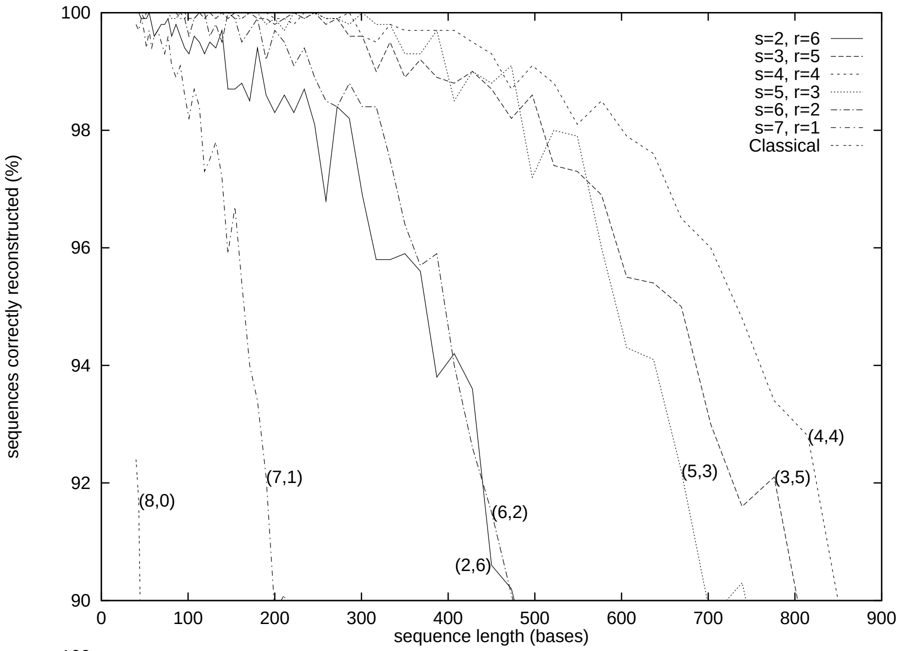


Figure 8: skewed data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



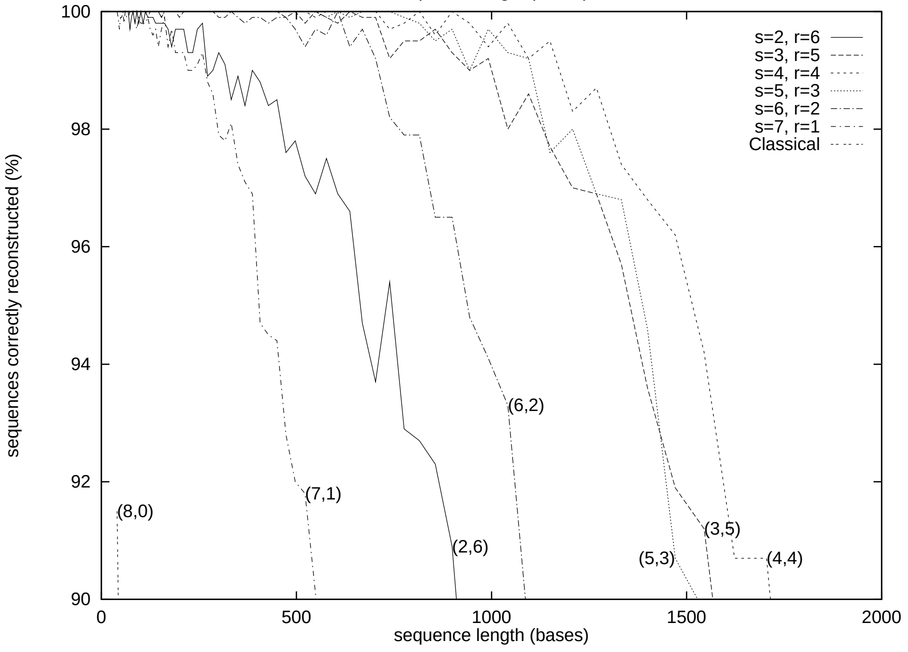
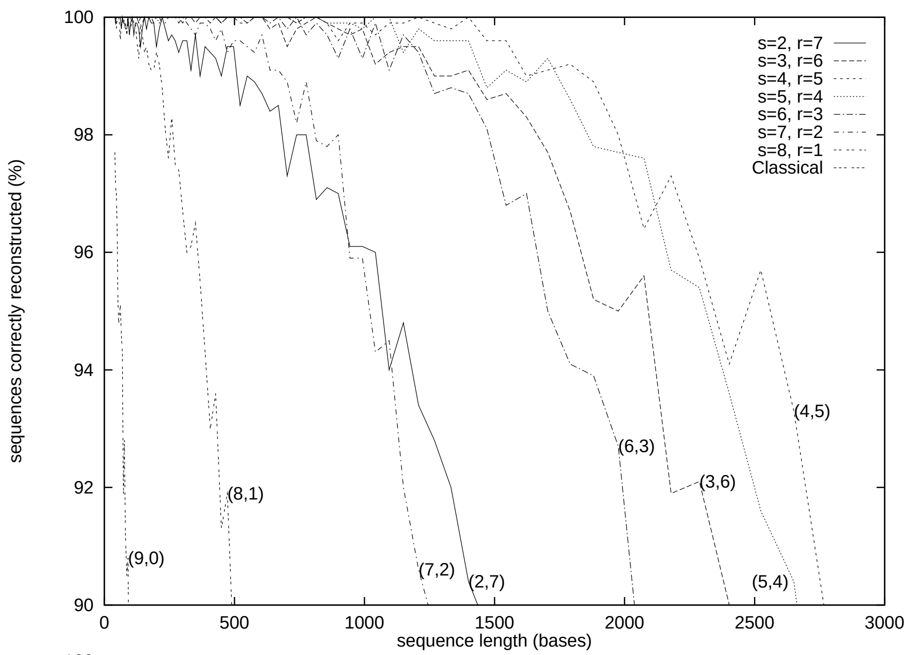


Figure 9: skewed data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



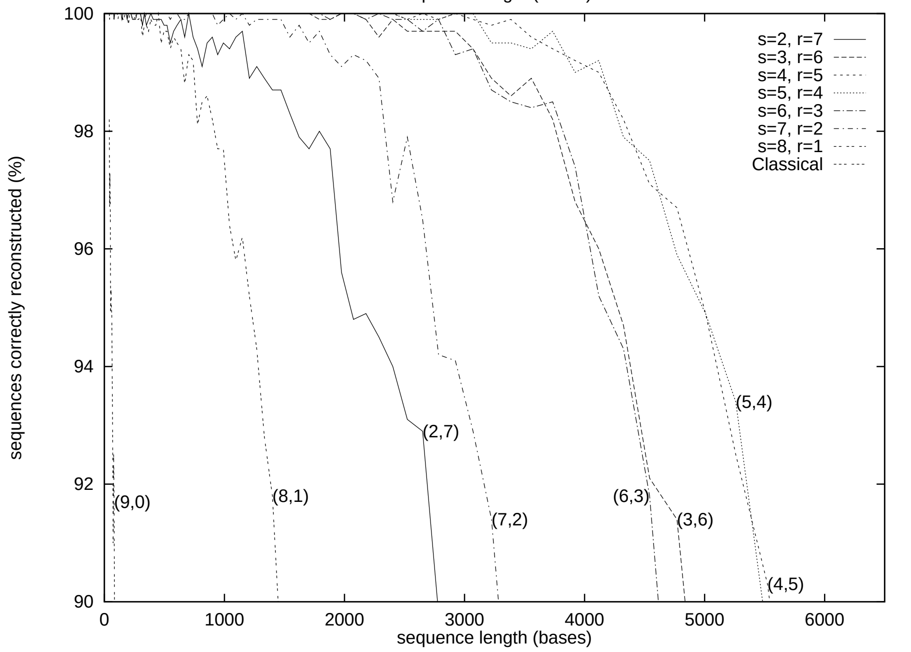


Figure 10: skewed data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.

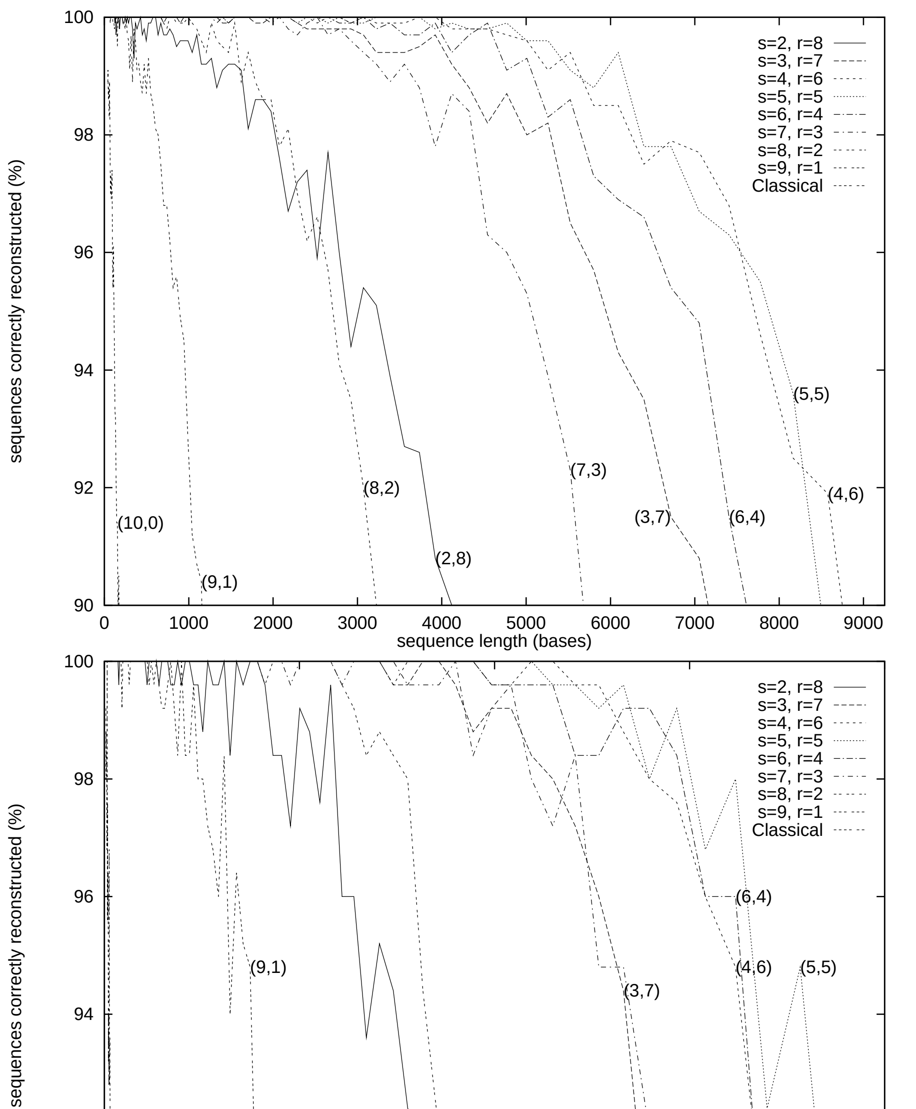
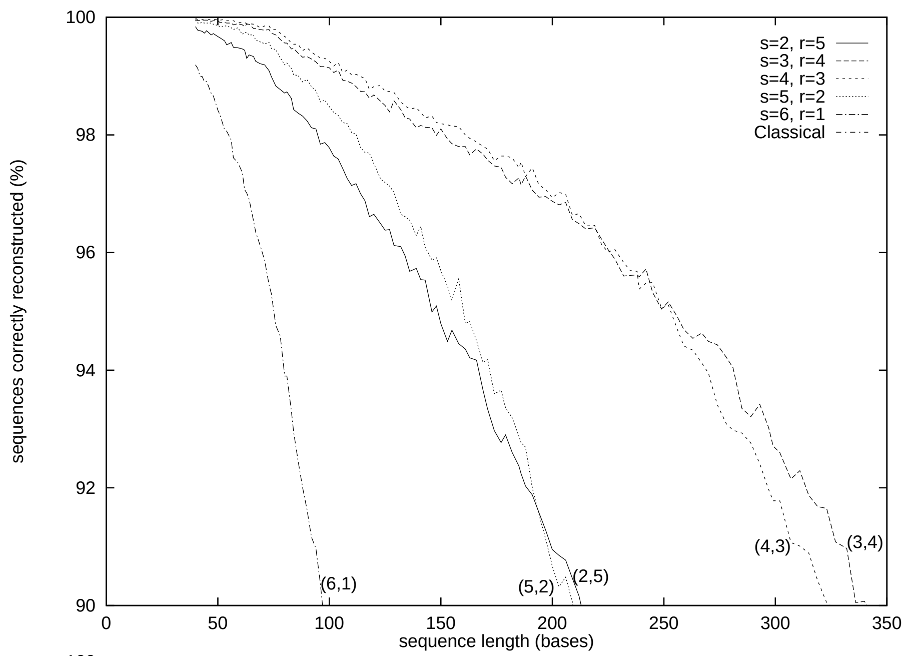


Figure 11: skewed data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



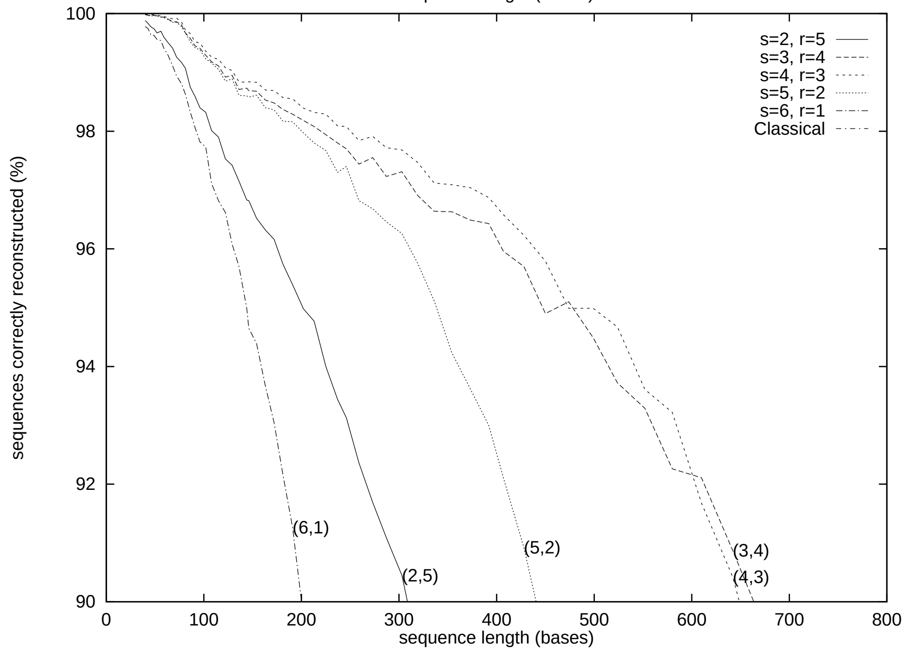
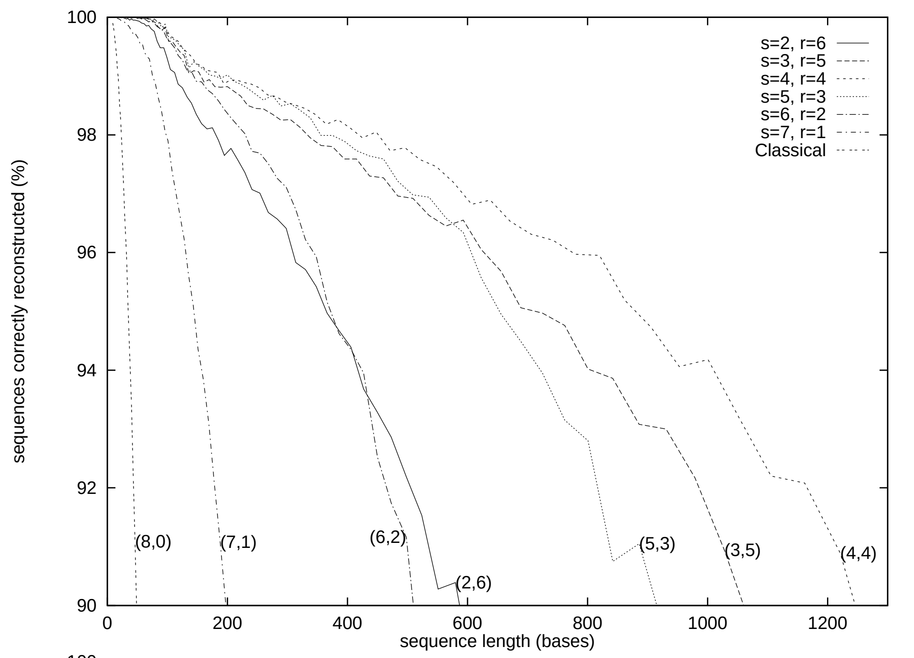


Figure 12: *M. thermoautotrophicum* data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



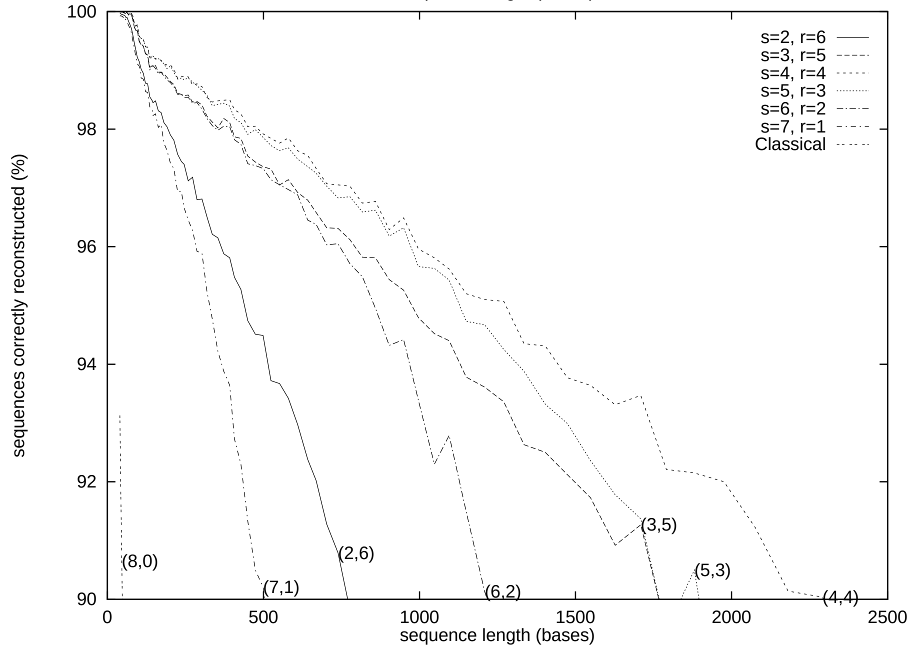
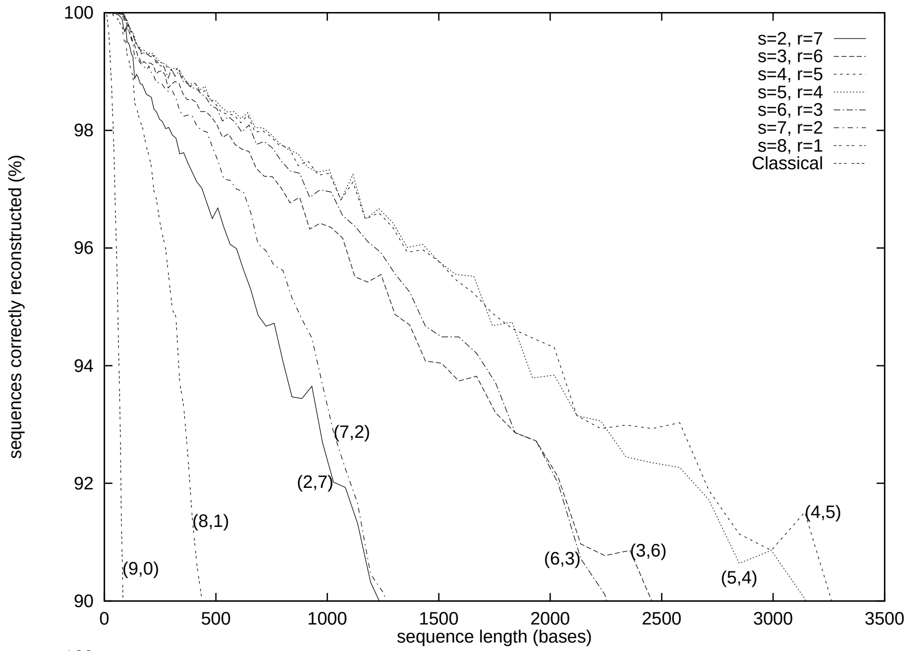


Figure 13: *M. thermoautotrophicum* data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



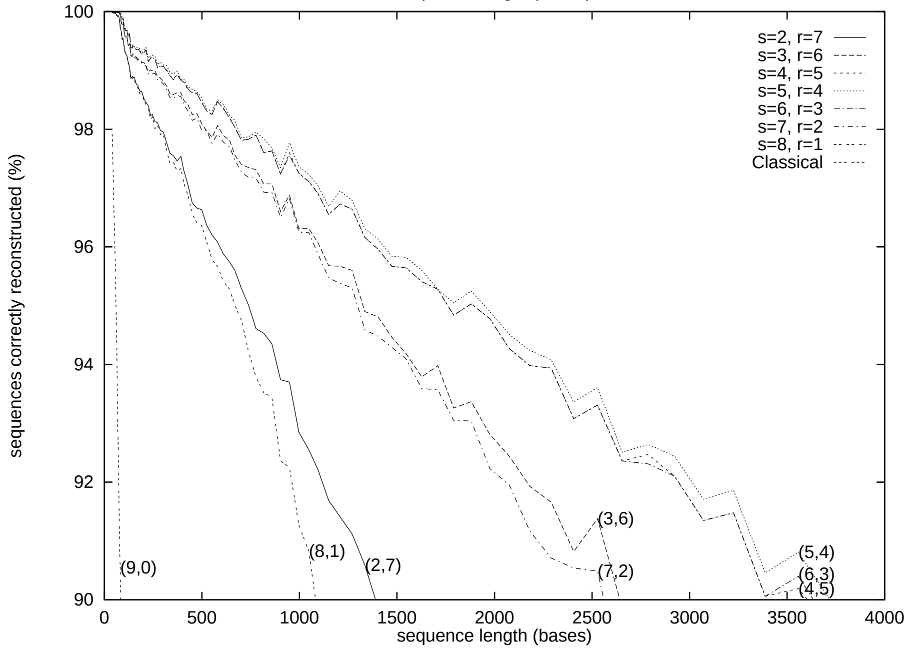
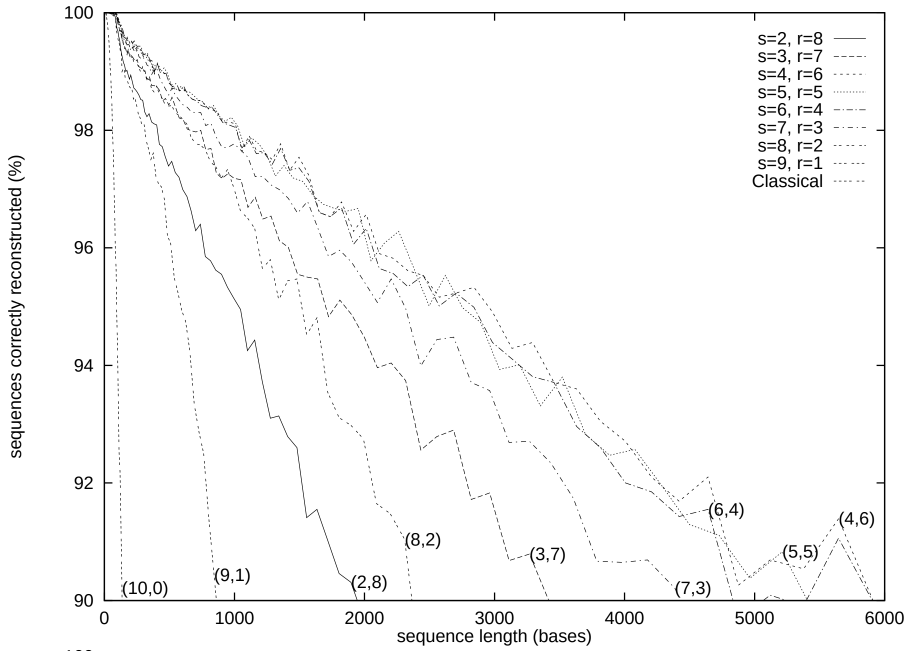


Figure 14: *M. thermoautotrophicum* data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



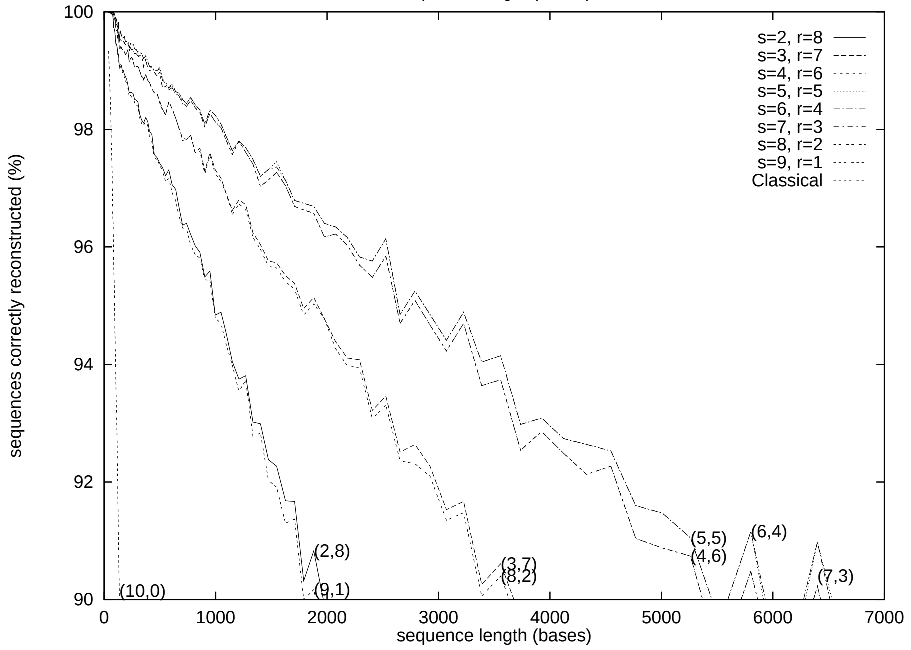
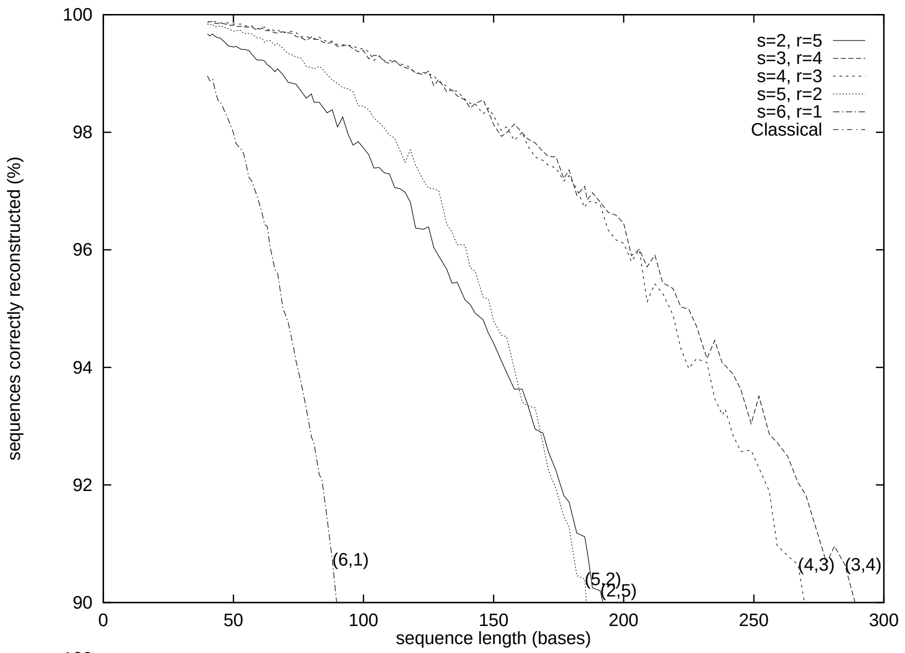


Figure 15: *M. thermoautotrophicum* data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



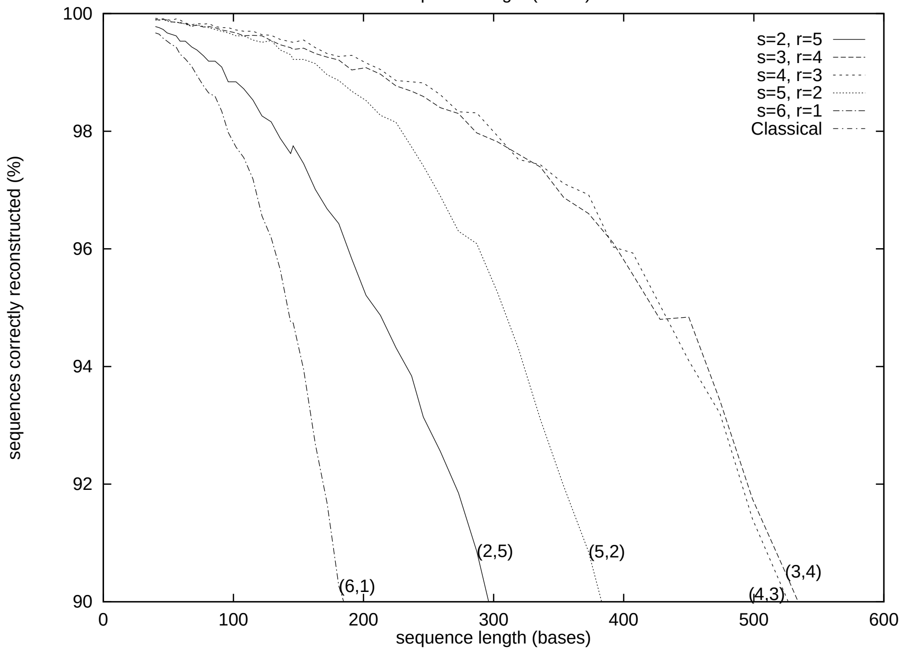
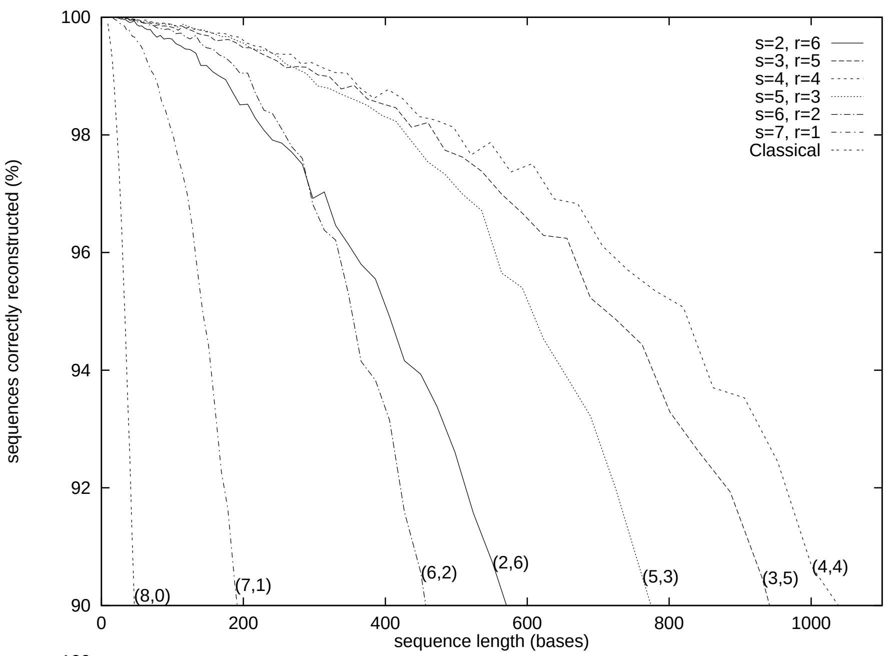


Figure 16: *H. influenzae* data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



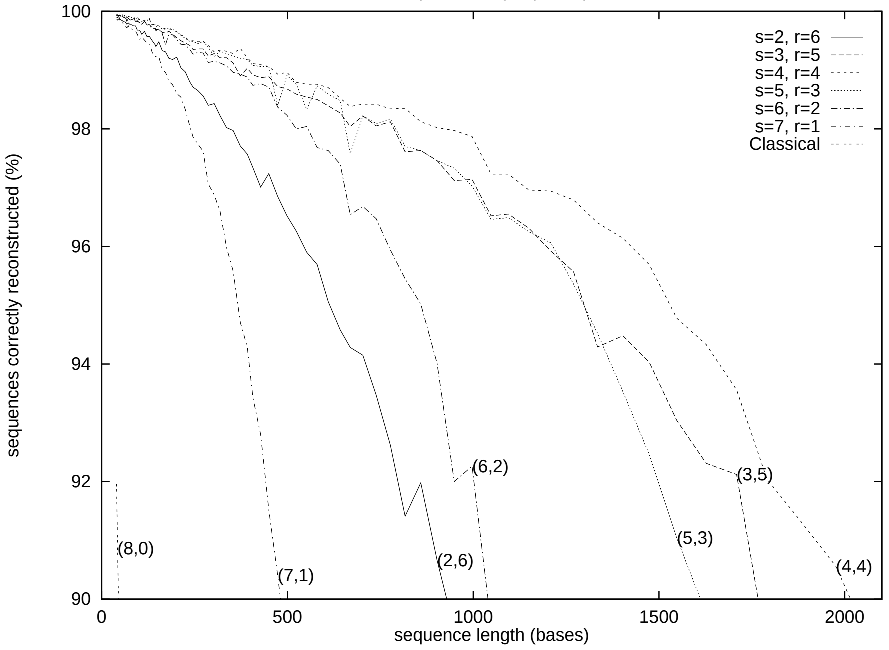


Figure 17: *H. influenzae* data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.

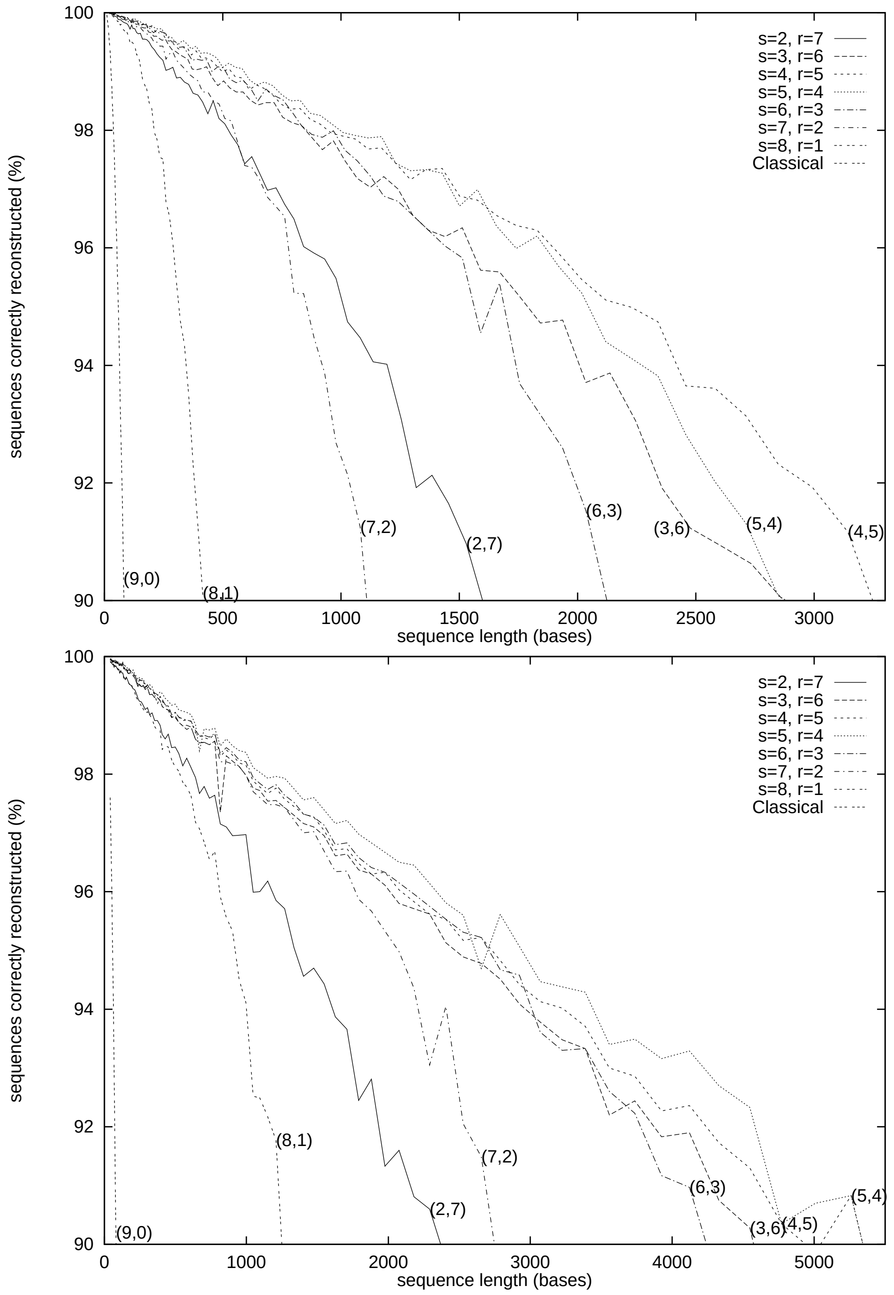
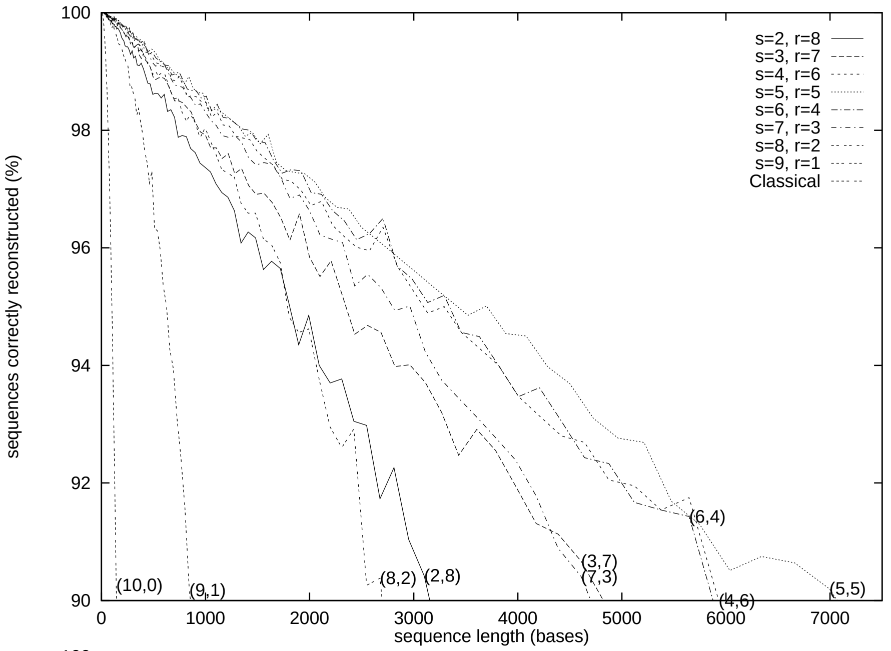


Figure 18: *H. influenzae* data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



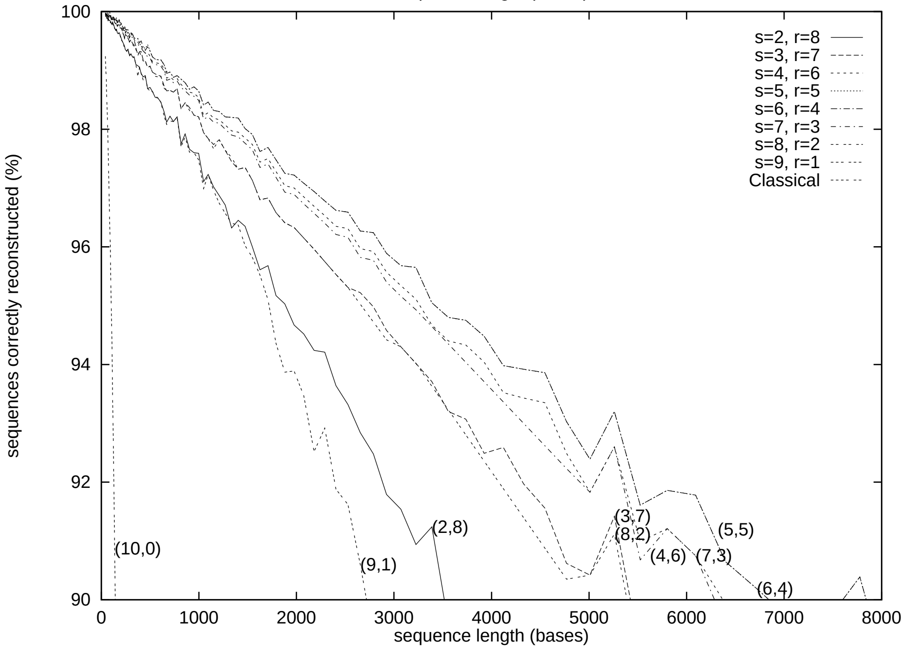
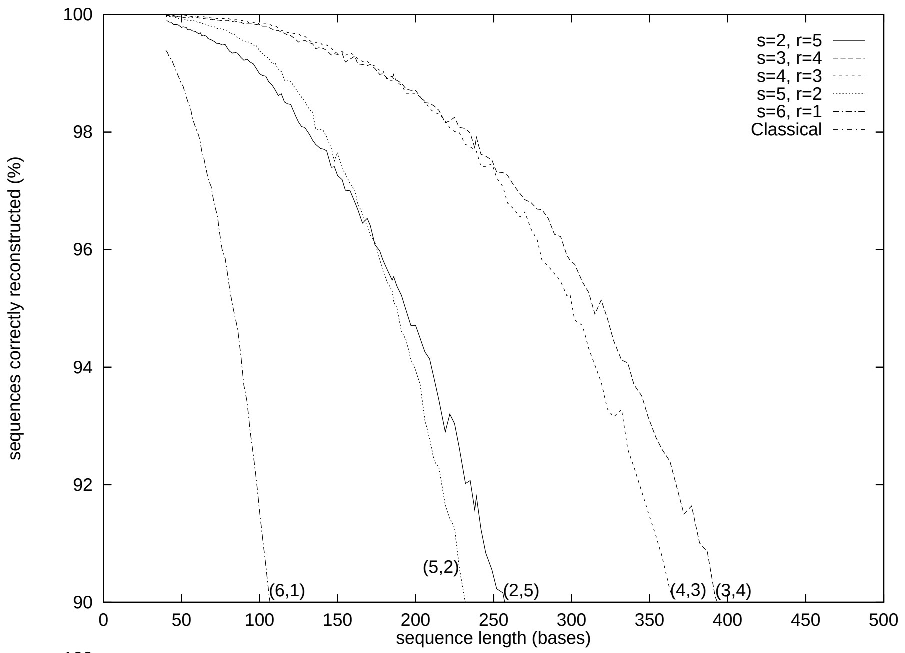


Figure 19: *H. influenzae* data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



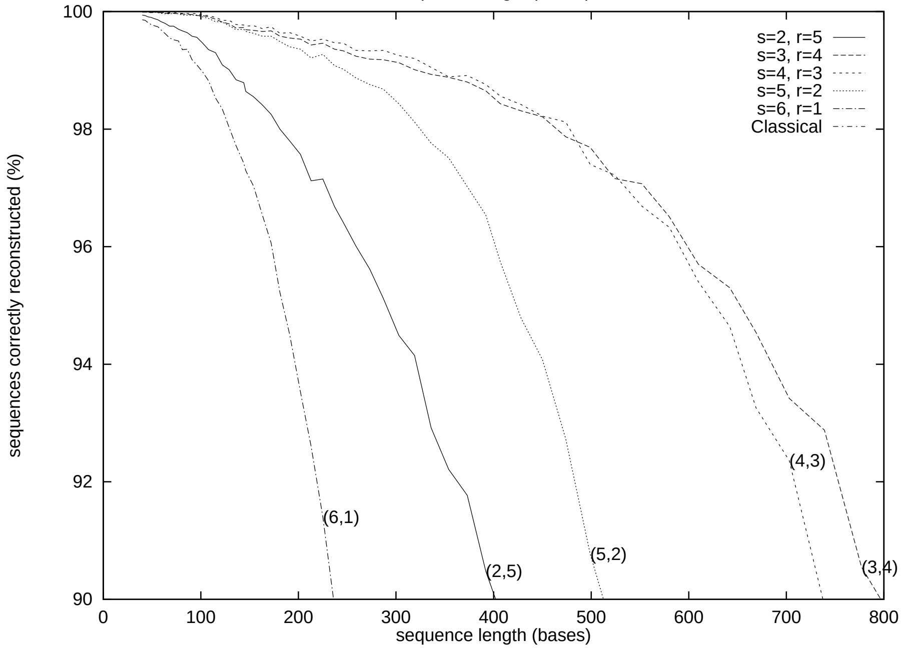
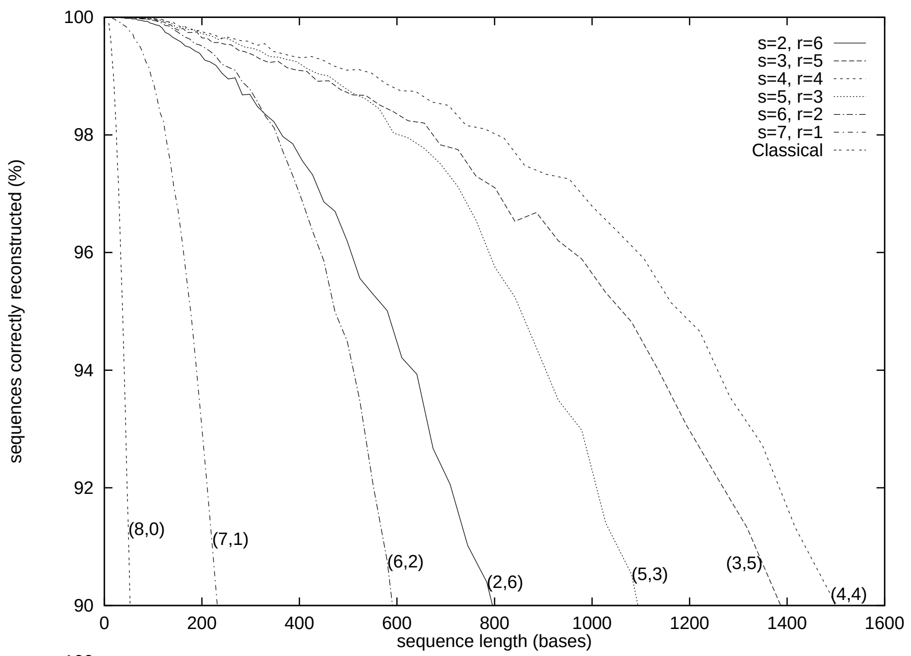


Figure 20: *E. coli* data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



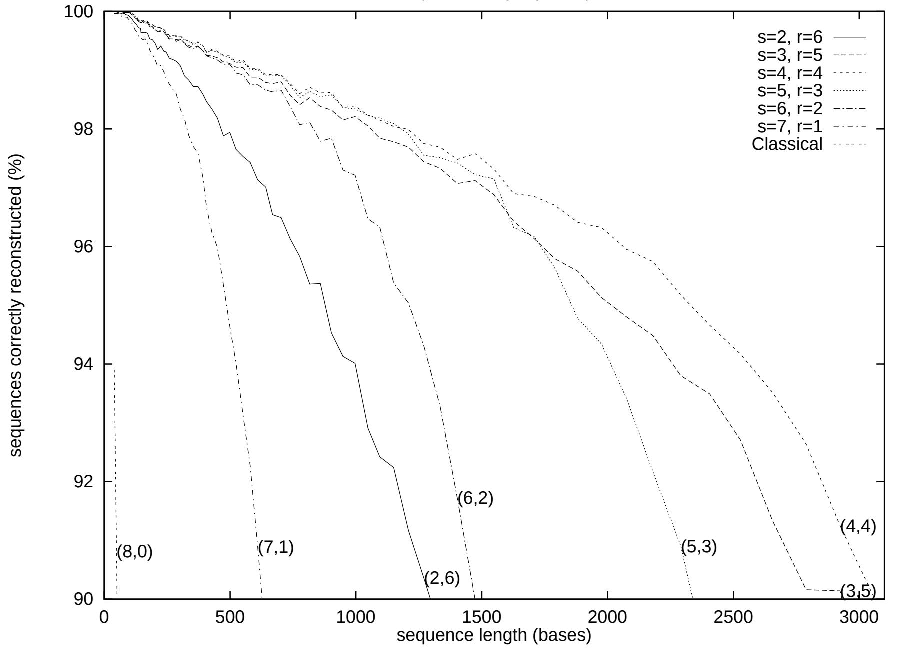
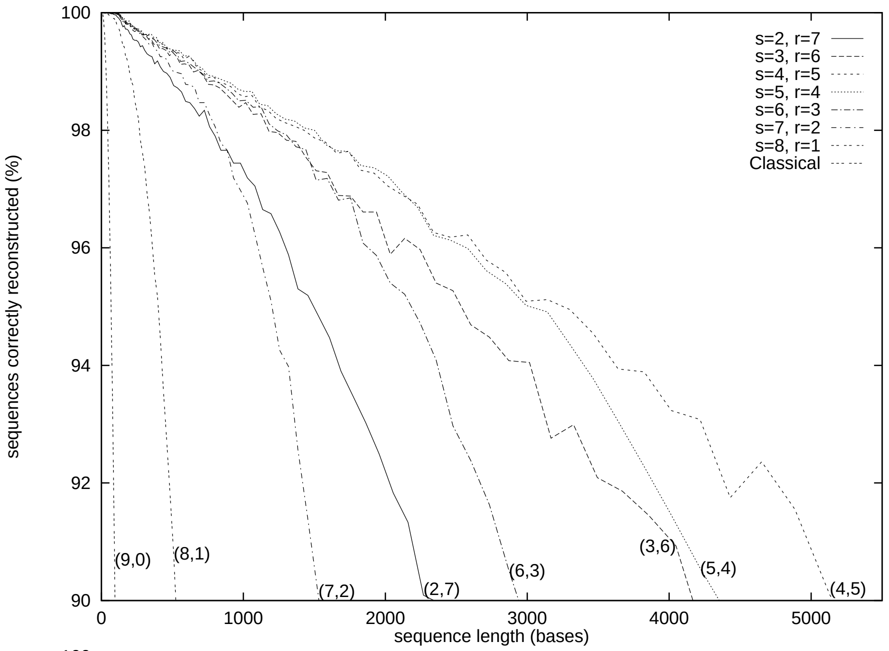


Figure 21: *E. coli* data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



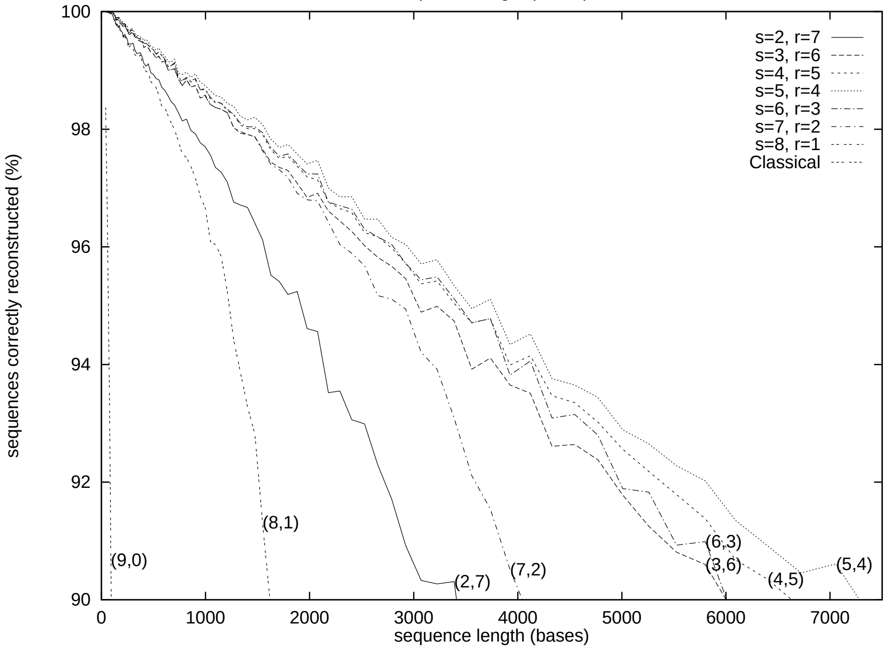
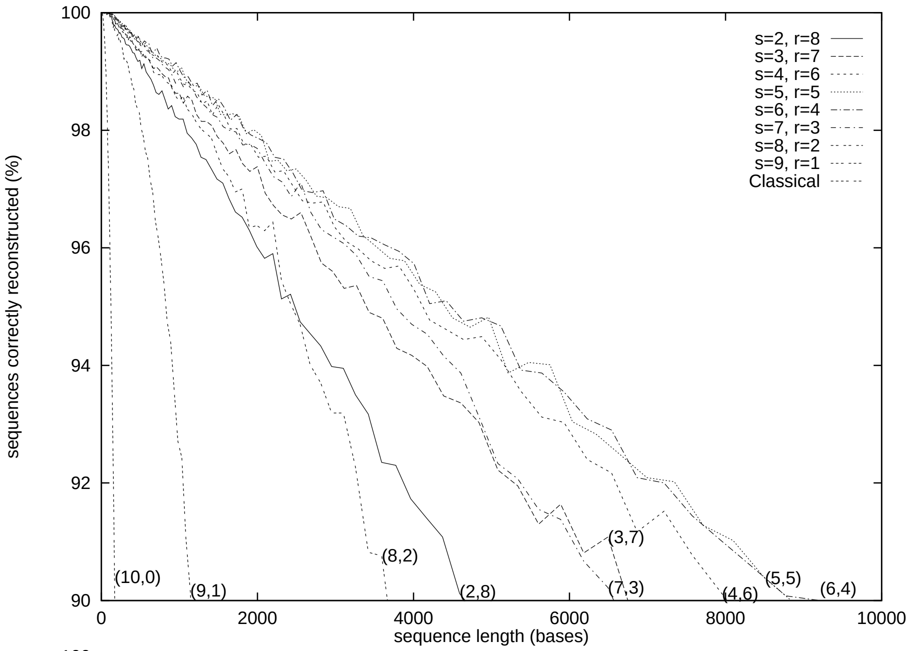


Figure 22: *E. coli* data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



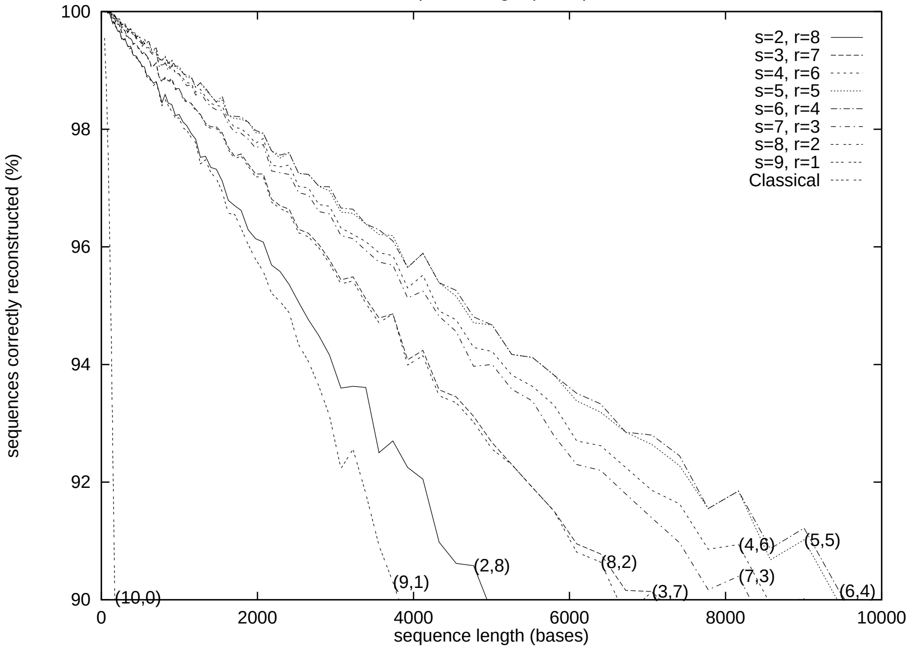


Figure 23: *E. coli* data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.

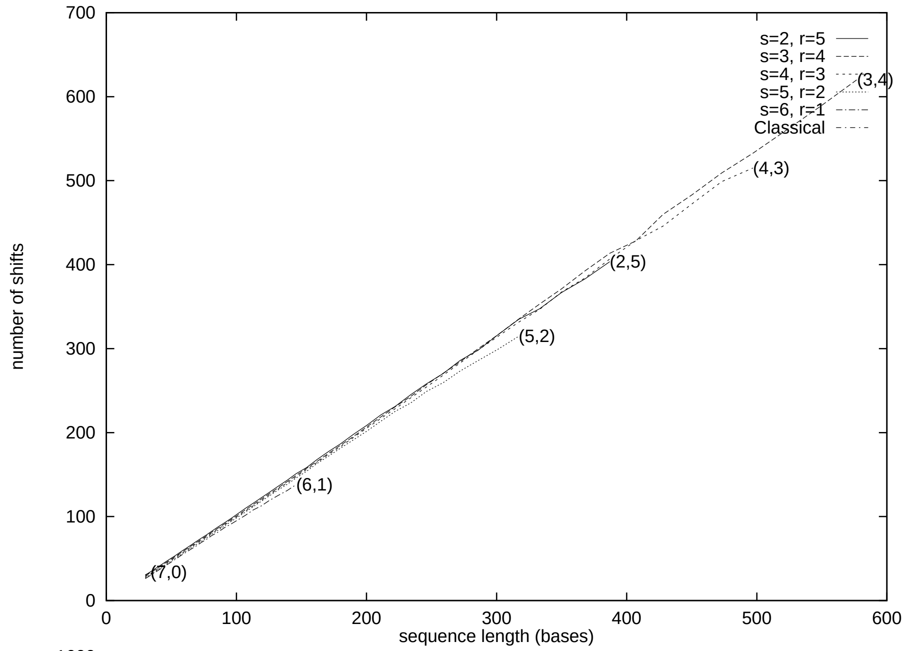
5.3 Number of extensions

Another metric was the number of shifts needed to disambiguate the extension character. This serves as a metric for the computation time cost of the method. Each shift requires a call to LOOKUP, and some calls to CHECKINM and so on. The cost model is that an extension that required i shifts costs $i+1$ time; in the extended algorithm, we count what would have been a failure in the basic algorithm as r shifts, and also add the number of shifts in each branch we followed thereafter.

The same sets of DNA were used as in the previous metric; indeed, the data were acquired at the same time as those used in the success rate metric were acquired. Each call to FILLM generated one count, of a shift of i (the return value). Since FILLM is called recursively in the extended algorithm, this correctly counts the total number of shifts. Unfortunately, only the randomly generated data is available, as `real_dna.pl` does not give trustworthy results for the number of shifts. The graphs which do not have this property were limited to success rates only as low as 90%.

This metric indicates that the number of extensions grows about linearly with sequence length, for reasonable values of m. The $h = 2t$ cases grow slightly faster than linearly, which would be accounted for by the cost of extending multiple possibilities. With the $O(k+n)$ implementation of LOOKUP, and with n (the number of probes found) typically small, this implies that the algorithm runs in linear time—at least in most useful cases. With the implementation in the simulation, LOOKUP is $O(k\frac{m}{4^{s-1}})$, so the algorithm runs in quadratic time.

The spikes apparent on the skewed data set are puzzling. The height is indicated on the graphs when they are clipped. Note they only occur on the $h = 2t$ cases, and when k is small (the most dramatic is $k = 7$, while $k = 10$ is smooth).



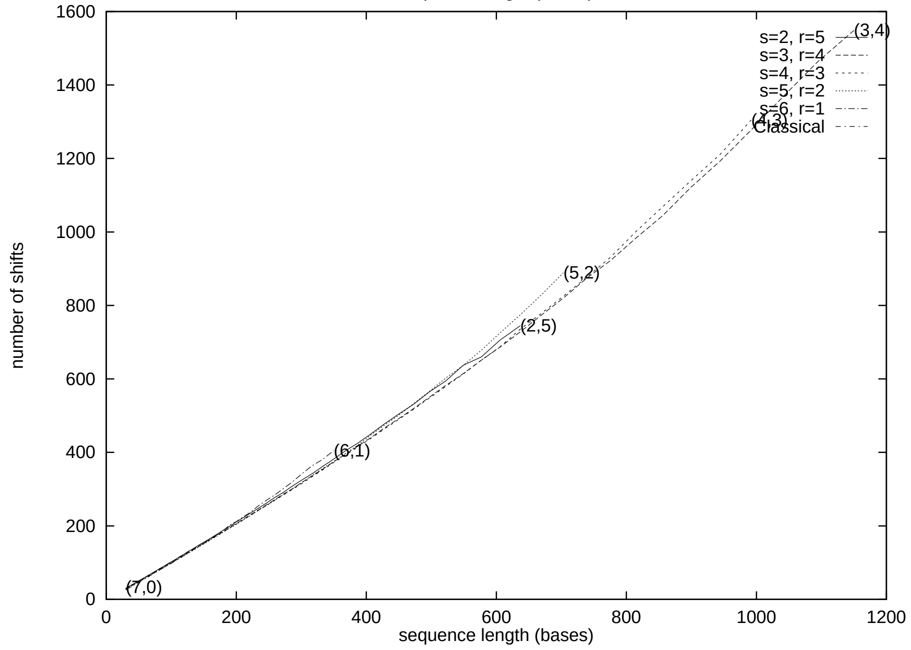
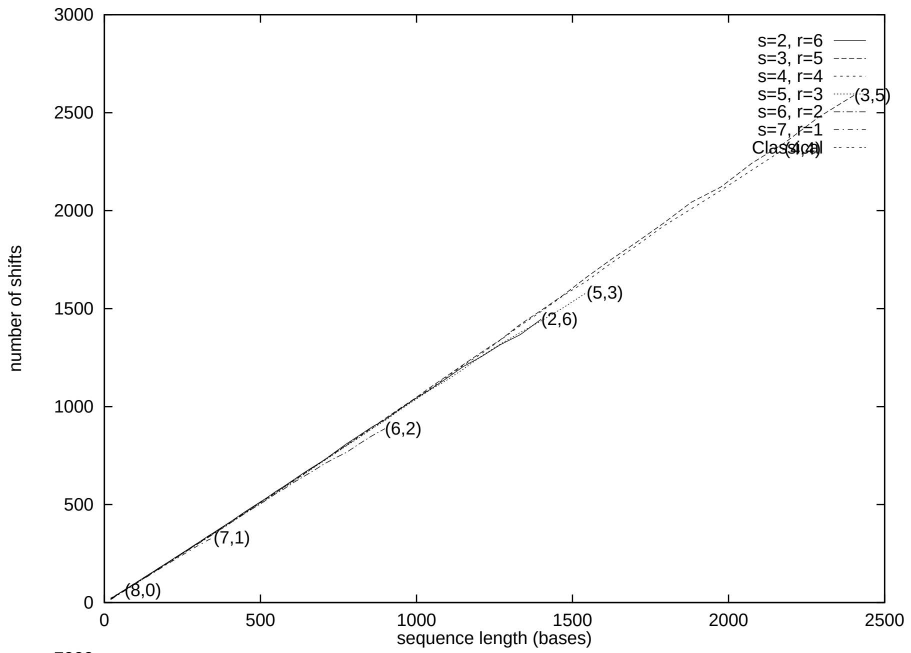


Figure 24: uniformly distributed data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



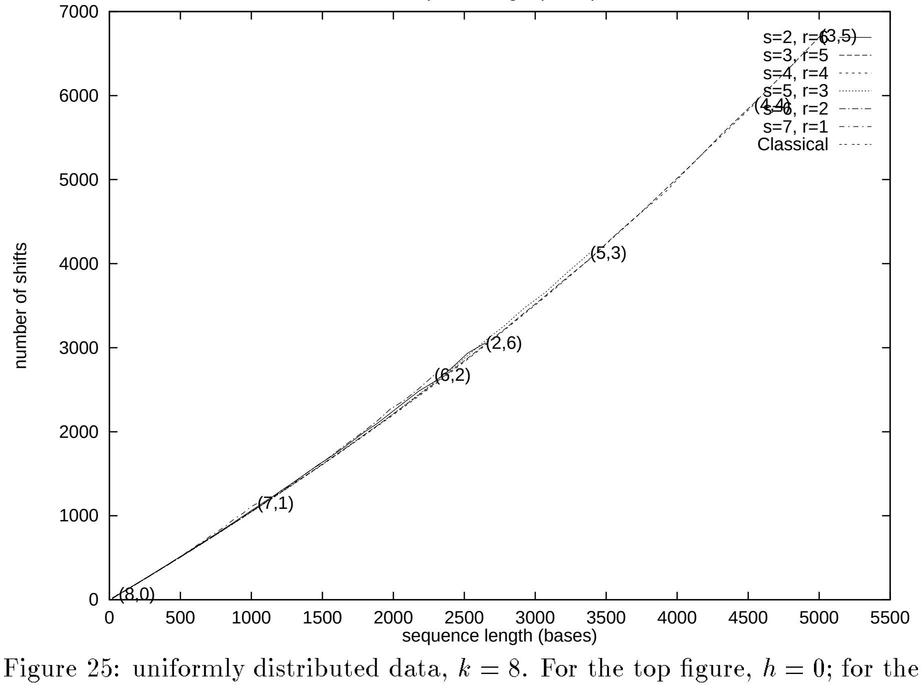
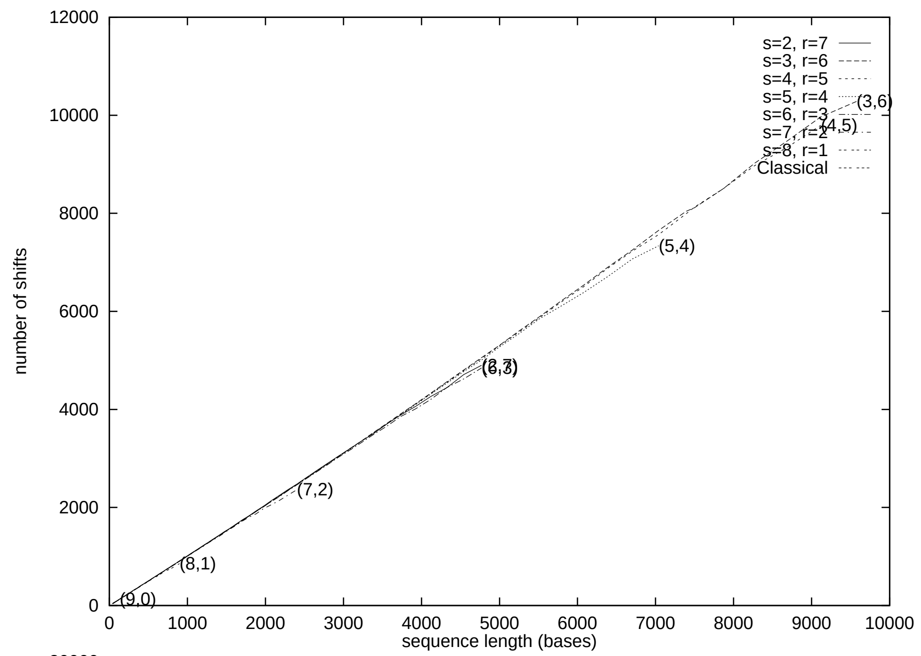


Figure 25: uniformly distributed data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



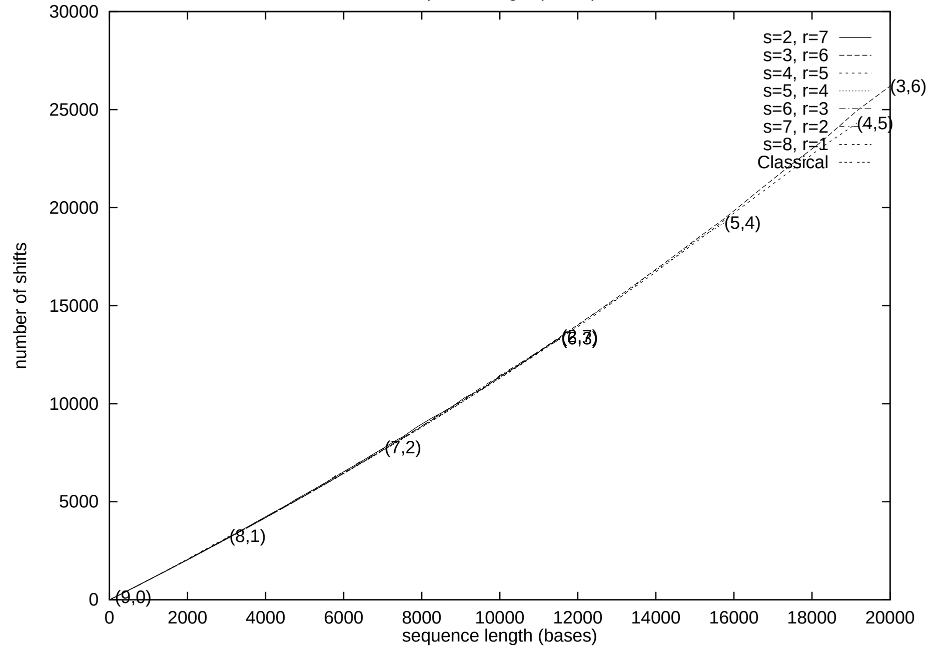
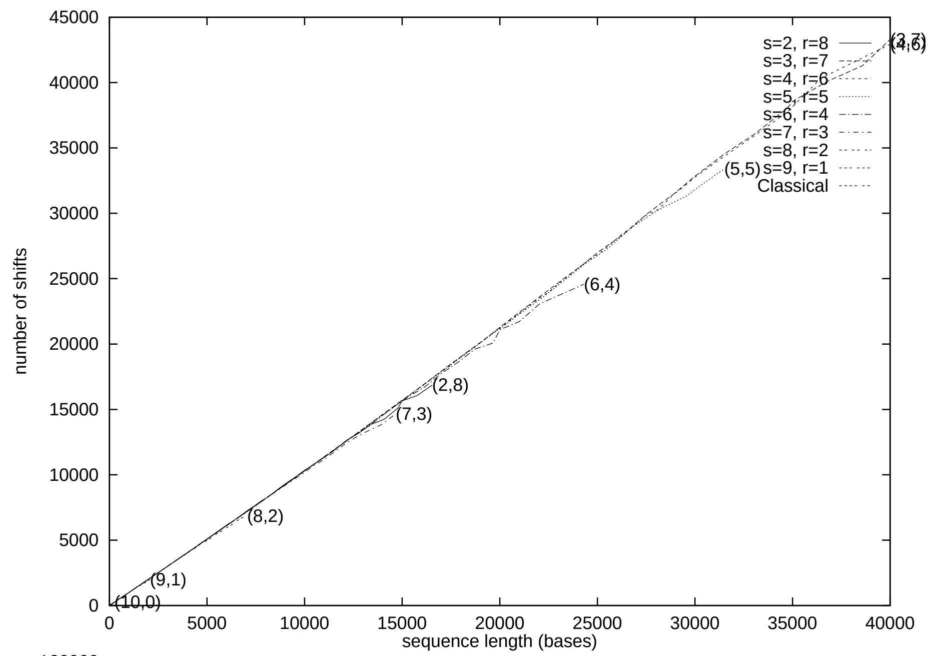


Figure 26: uniformly distributed data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



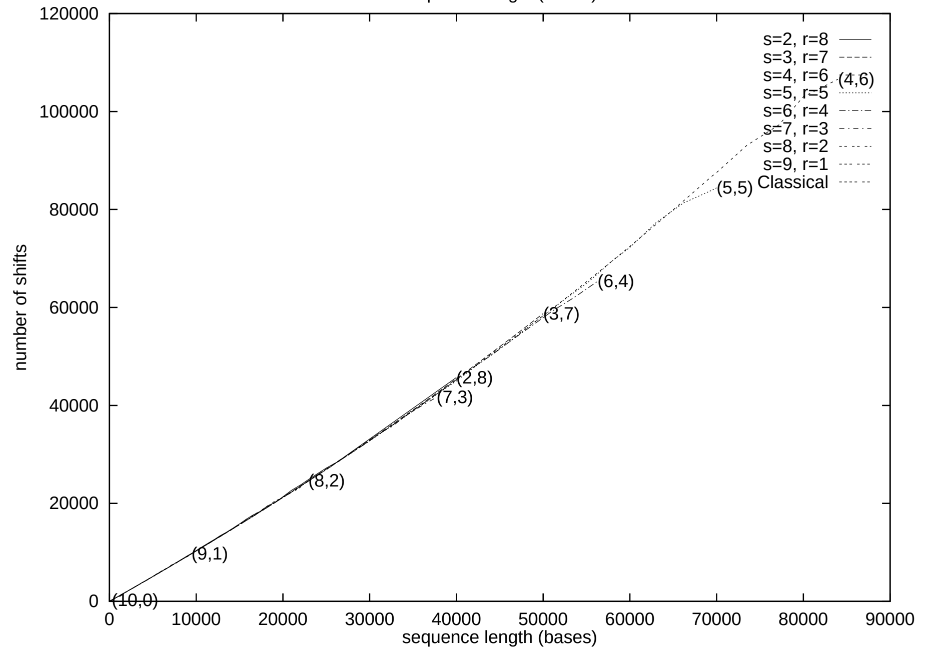
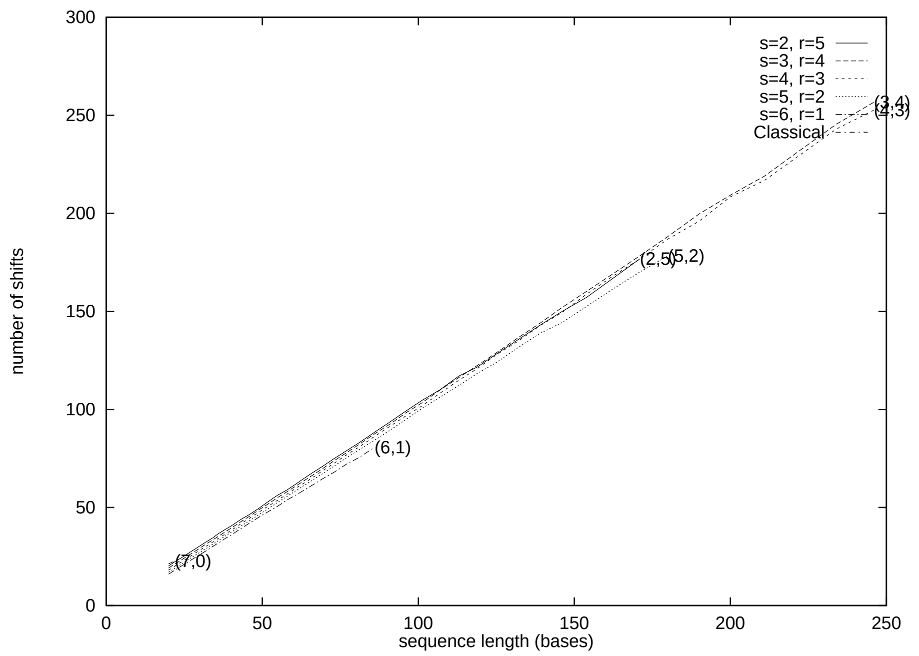


Figure 27: uniformly distributed data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



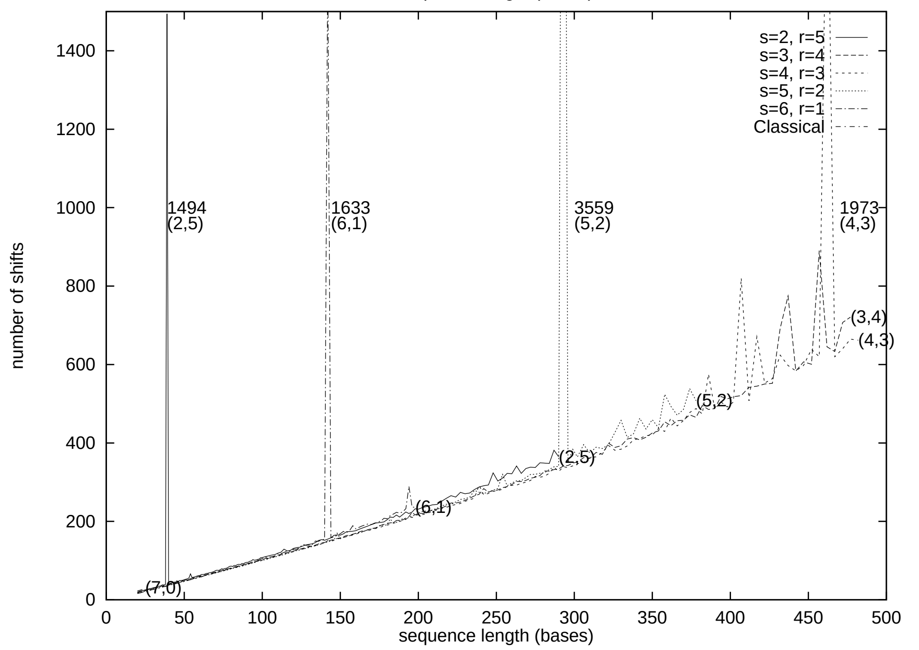
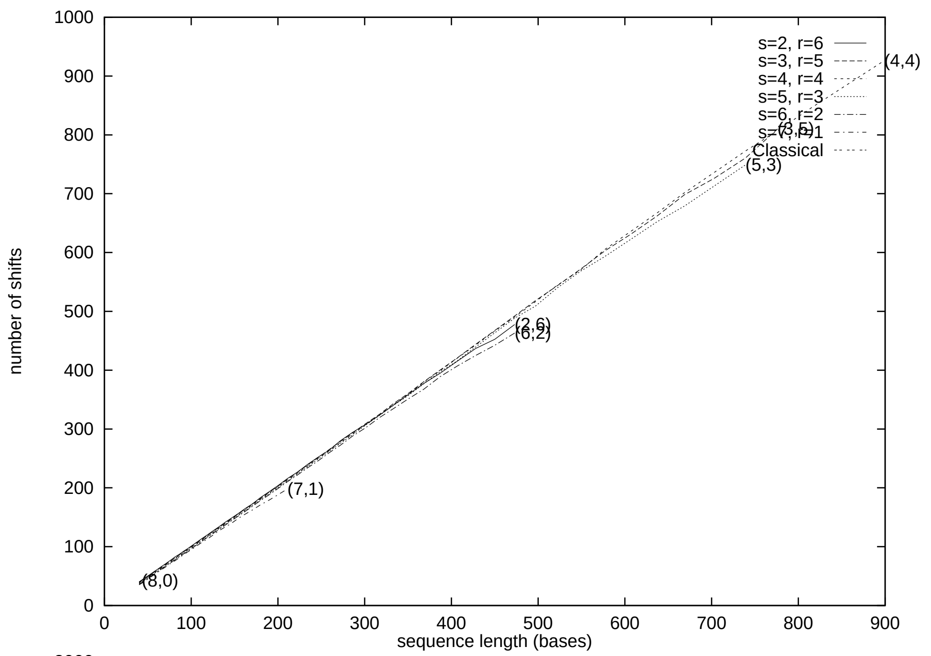


Figure 28: skewed data, $k = 7$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



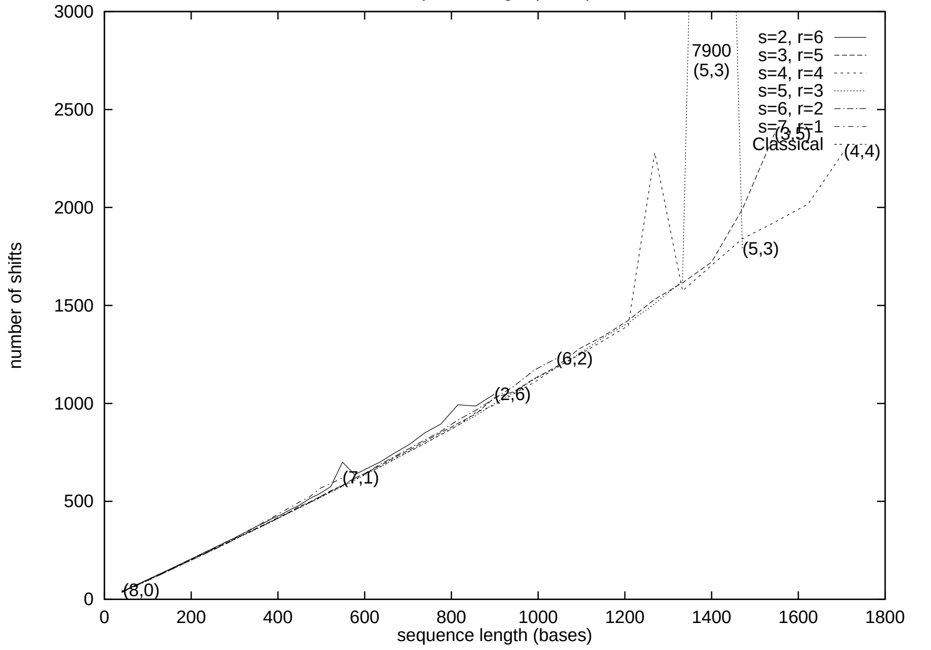
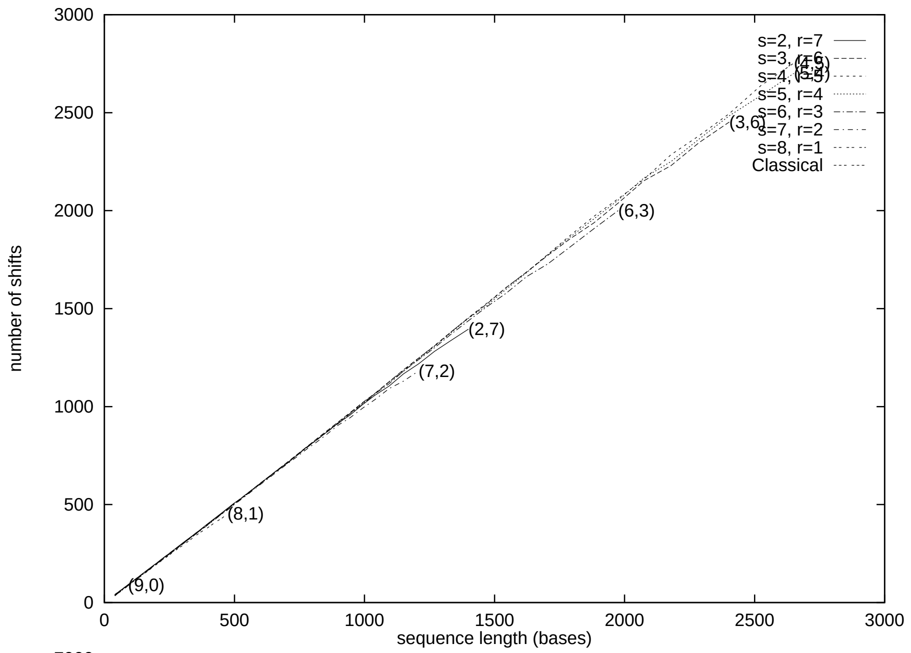


Figure 29: skewed data, $k = 8$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



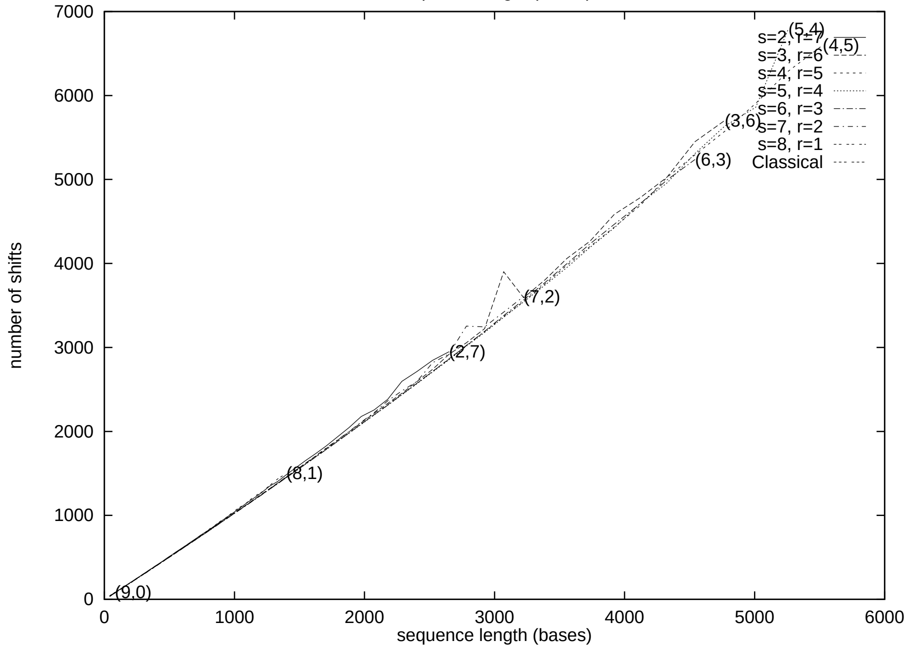
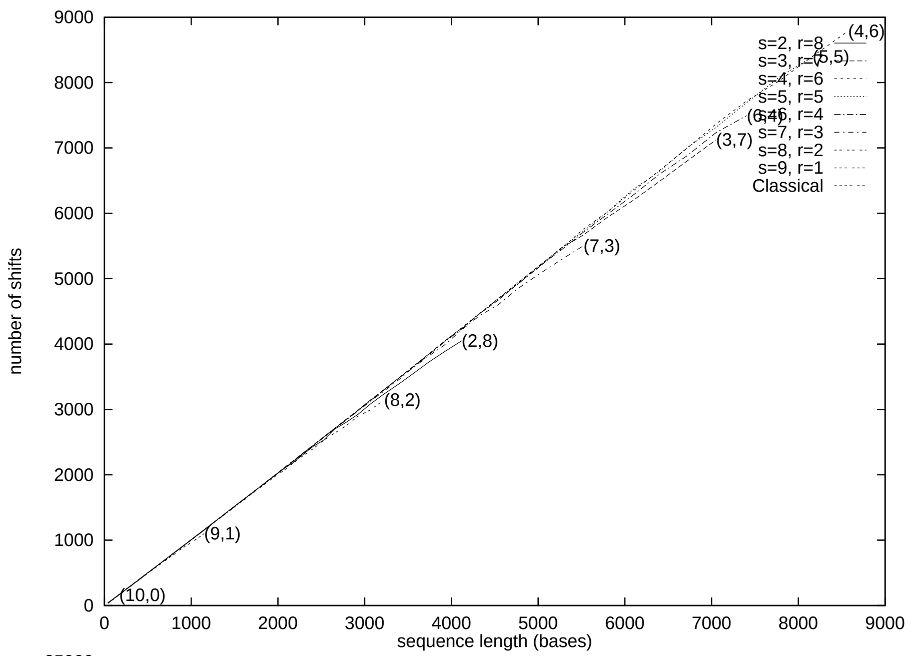


Figure 30: skewed data, $k = 9$. For the top figure, $h = 0$; for the bottom, $h = 2t$.



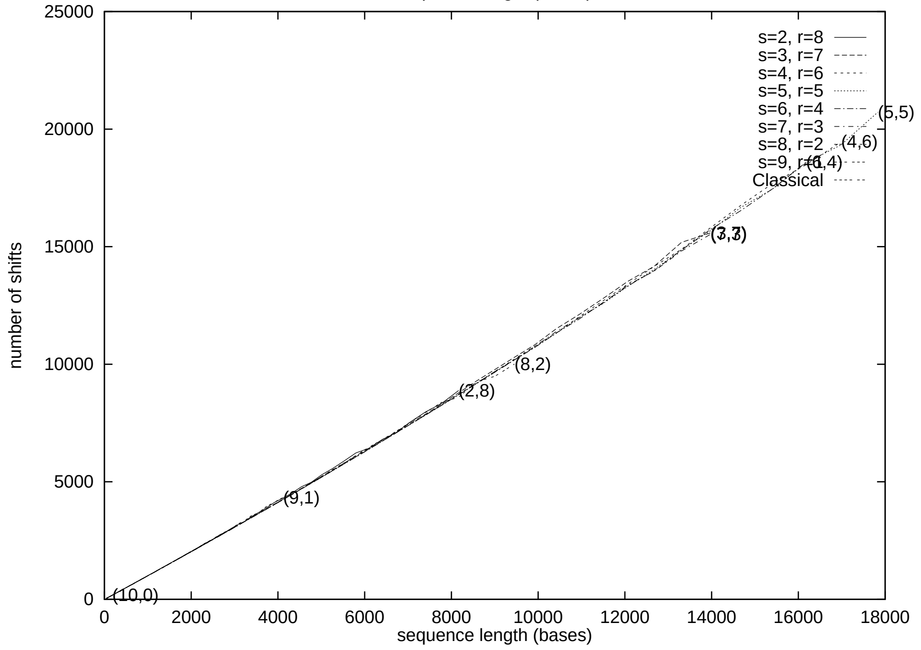


Figure 31: skewed data, $k = 10$. For the top figure, $h = 0$; for the bottom, $h = 2t$.

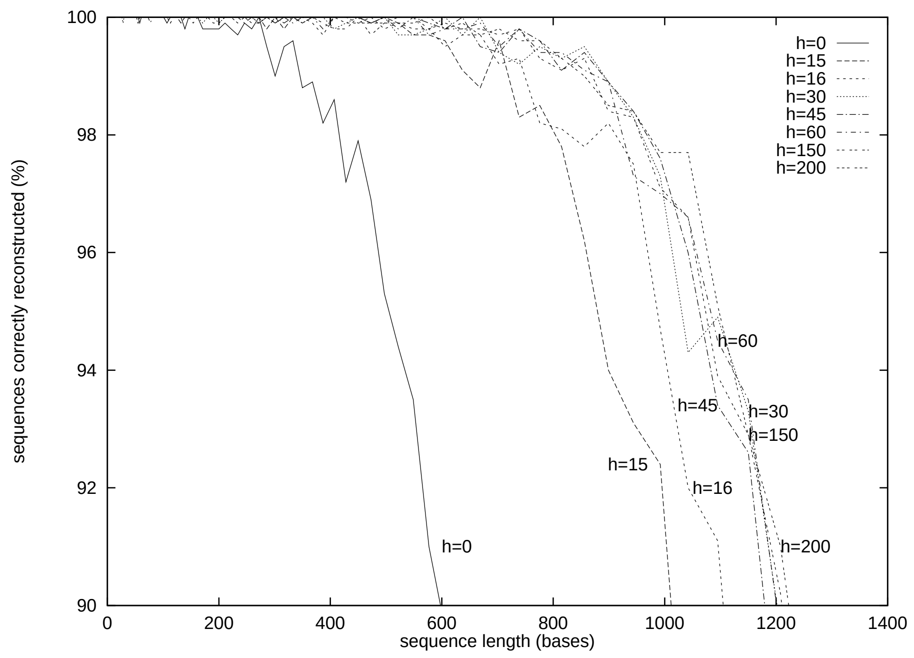


Figure 32: Effect of h on the performance of the method using a (3,4) chip.

5.4 Effect of h

All of the data above chose h as either 0 or $2t$. These were chosen respectively to show the power of the basic algorithm, and to show that of the extended algorithm. Below $h = t - 1$, the extended algorithm has no more power than the basic one, since any fooling probe must already have remained a fooling probe for that many characters. However, above that, the extended algorithm gives a great benefit. It is intuitive that the method should do increasingly better as h increases.

Figure 32 shows that, at least for a $(3, 4)$ chip and uniform random data, the success rate (and the maximum length) does increase with h. However, this increase is most marked as h ranges from 0 to $2t = 30$; after that point, we get almost no benefit by increasing h. More testing would have to be done to get a conclusive answer on this point, but it seems that $h = 2t$ was in fact well chosen.

6 User Manuals

This section offers user manuals for all programs in the software package. The programs are in C/C++, Perl, Awk, or sh. A makefile is provided which should allow compiling the C and C++ programs on any platform with GNU make and the GNU C/C++ compiler. It has been tested on Linux 2.0.31 and Solaris 2.6. The scripts may need to have an updated path for Perl or Awk, but otherwise should be platform-independent.

6.1 Compiling

The makefile provided with the package automates compilation of the programs written in C and C++. Currently, the compiler used is `gcc`. This can easily be changed by modifying the `CCDEBUG`, `CC`, `FASTOPTS`, and `PROFOPTS` lines; there are sample such lines for `egcc` and `CC` (the Sun Workshop compiler).

Most of the makefile is applicable to `gap`, as the other programs are all very simple (a single file).

The makefile has the following targets:

all: Default; compile a version with debugging information into `gap.new`. The defines `APPEND_CHARS`, `DEBUG`, and `DEALLOCATE_ALL` are turned on.

opt: Optimized build. Use the flags which yield the fastest executable. Note the Sun Workshop compiler yields much faster code than does `gcc`. The define `NDEBUG` is on.

prof: Profiled build. Include the compiler flags to collect profiling information. The optimization flags are also on, so that the profile is that of the optimized build.

clean: Remove temporary object files created during the compilation.

backup: Create a backup of the current source code. The backup file is in the `backups` directory, with a name corresponding to the current time.

depend: Discover the dependencies using `makedepend`(1).

6.2 gap

Usage: `gap [-x <h>] <s> <r> <file>`
Takes as input a sequence, and simulates trying to sequence it using an (s, r) gapped chip.

Input: The input format is a standard ASCII text file. Each sequence is bracketed by the lines "`ORIGIN`" and "`//`". There can be multiple sequences in a single file. A sequence consists of characters in [acgt] or [0123] (that is, ASCII 0x30 through 0x33). Whitespace (as denoted by `isspace(3)`) is ignored; characters outside of the admissible set of characters are translated to 'a.'

Output: The output format is one line per input sequence, where each line is a colon-separated list of fields. The fields are as follows:

1. length of the input (m)
2. 1 if the algorithm's output matches the program's input, 0 otherwise
3. 1 if the algorithm reported a failure, 0 otherwise.

This is followed by $r+1$ fields, where field i denotes the number of times an extension required exactly i shifts (r if we had to invoke the extended algorithm).

For example, a line `300:1:0:277:14:3:0:1` means we were trying to sequence a 300-base sequence, that we succeeded, and that we invoked the extended algorithm once. A line `2000:0:1:1110:369:154:83:86` means that we failed to sequence a 2000-base sequence, and that the algorithm noticed this.

Options: -x h: Specify the parameter h to the extended algorithm. The default is $2t$.

Examples: `gap 4 6 -`
Reads input from stdin, sequences it using a $(4,6)$ chip.

`gap -x0 3 4 test`
Reads input from test, sequences it using a $(3,4)$ chip using the basic algorithm.

Compiling: See section 6.1 for information about compiling the package using the makefile. In the makefile, except for `NO_BOOLEAN`, the following options should be included in `GLOBDEFS` to be used in all compilations, and in `DEFINES` if only for the debug compilation. `NO_BOOLEAN` should be set in `CC` and `CCDEBUG` as it depends on the compiler.

The following preprocessor defines apply:

APPEND_CHARS: Append sr random characters at the end of the sequence. Otherwise, no characters are added.

DEBUG: Include some debugging information and commands.

NDEBUG: Turn off `assert`(3) and some other debugging aids.

INCLUDE_STDOUT: Output very verbose information about the sequencing process.

DEALLOCATE_ALL: Deallocate all memory at the end of the program, instead of trusting the operating system to do this.

NO_BOOLEAN: Define a boolean data type, for compilers that do not have one (such as the Sun Workshop 4.2 compiler).

Bugs: The `lookup` function does not exist, and has instead been inlined by hand.

The implementation of LOOKUP takes time $O(k\frac{m}{4^s})$; this makes the algorithm be quadratic-time, when it should be linear-time. Using a trie would take $O(k)$ time.

6.3 tester.pl

Usage: `tester.pl [opts] <k> <s> <mlow>:<mhigh> <iter>`
Run `gap` on uniformly distributed random sequences of varying lengths, from $m = mlow$ to $m = mhigh$. *iter* sequences are generated for each length. Uses an (s, r) chip where $r = k - s$. Leaves output in file "`output.k s mlow:mhigh iter`." The run is stopped if the algorithm recognizes too few sequences, or when the sequence length exceeds *mhigh*.

Options:

-q:	Send no output to stdout. Otherwise, a new line of output is printed when m is increased.
-m:	Send mail when the job is finished.
-l:	Increase m linearly by the step value rather than exponentially.
-s *step*:	Specify the step value. Default is 1.1 for exponential growth, 10 for linear growth.
-d *pct*:	Die if the success rate falls below *pct*. Default is 50%.
-x *h*:	Specify the parameter to the extended algorithm, h. Default is $2t$; 0 disables the extended algorithm.
-o *file*:	Write output to *file* instead of to `output`.*k*.*s*.*h*.*mlow*.*mhigh*.*iter*

Examples: `tester.pl -qm -s1.05 -d88 8 3 50:50000 1000`
Uses a $(3, 5)$ chip on sequences of length 50 and higher, increasing in length by 5% until the success rate falls below 88% – that is, until fewer than 880 of the 1000 sequences of a given length cannot be reconstructed. The run will also stop if we exceed 50000 bases in length (but that will almost certainly not happen). h is the default of $2t$.

`tester.pl -x0 -d0 -l -s1 7 4 30:1000 100`
Uses a $(4, 3)$ chip on sequences of every length from 30 to 1000. The basic algorithm is used, since $h = 0$.

Output: The output is intended to be used as input to a later script, such as `splitfiles`, rather than to be read by a human operator. There is one line for each m run. The format is a colon-separated list, with the following fields:

1. s

2. r
3. m
4. Number of sequences of length m
5. Number correctly reconstructed
6. Percent correctly reconstructed (that is, 100*[4]/[5])
7. Number of failures Followed by $r + 1$ fields, where field i is the number of times we did exactly i shifts when extending.

6.4 real_dna.pl

Usage: `real_dna.pl [opts] <file> <min> <max> <s> <r>`

Run `gap` on real sequences of varying lengths, from $m = min$ to $m = max$. The real sequences are non-overlapping substrings of the string held in *file*. Uses an (s, r) chip. Leaves output in file "`file.min:max.s.r`." The run is stopped if the algorithm recognizes too few sequences, or when the sequence length exceeds max. The file is assumed to have been encoded using `comp_dna`.

Options:

-m:	Send mail when the job is finished.
-s *step*:	Specify the step value. Default is 1.1 for exponential growth, 10 for linear growth.
-d *pct*:	Die if the success rate falls below *pct*. Default is 50%.

Examples: `real_dna.pl -m -s1.05 -d88 sequences/ecoli.cmp 50 50000 3 5`

Uses a (3, 5) chip on sequences of length 50 and higher, increasing in length by 5% until the success rate falls below 88%. The run will also stop if we exceed 50000 bases in length (but that will almost certainly not happen).

Output: The output is similar but slightly different from that of `tester.pl`. Use `split_real` instead of `splitfiles`. There is one line for each m run. The format is a colon-separated list, with the following fields:

1. m
2. Number of sequences of length m
3. Number correctly reconstructed
4. Percent correctly reconstructed (that is, 100*[4]/[5])
5. Time spent (in seconds) Followed by $r + 1$ fields, where field i is the number of times we did exactly i shifts when extending.

Bugs: There is no error field in the output; this field should be added for completeness.

It seems the last line has its $r + 1$ shift-counting fields cut out.

There are other problems with the shift-counting fields.

6.5 my_seqgen

Usage: `my_seqgen [probabilities] <m> <iter> [seed]`

Randomly generates *iter* sequences each of length m, in the format used by `gap`. By default, the distribution is uniform, but each base can be given a different probability if needed. The random number generator is `random`(3); the seed is by default *time* **xor** *pid*, but can be specified by the user.

Options: -x *prob*: Set the probability (in percent) of getting the base 'x' to *prob*, where $x \in a, c, g, t$. The probability of multiple bases can be set; those of any unmentioned base are set to be equal, and such that the sum of the probabilities is 100%. *prob* can be neither negative nor greater than 100%.

Example: `my_seqgen 3241 1000`

Generates 1000 sequences each of length 3241. The data is uniformly distributed.

`my_seqgen -a40 3241 1000`

Generates 1000 sequences each of length 3241. The probability of generating 'a' is 40%, while those of 'c', 'g', and 't' are 20% each.

Notes: Pevznar and Belyi [1] included with their `sbhpack` package a similar program called `seqgen`. It provides more functionality, but it has a race condition which shows up when running in a distributed environment: occasionally, it falls into an infinite loop.

6.6 basecount

Usage: `basecount <n> <file>`

Reads a file and outputs the n-symbol entropy. The file is assumed to have been compressed by `comp_dna`.

Example: `basecount 7 ecoli.cmp`

Outputs the 7-symbol entropy of the *E. coli* genome.

Bugs: The dictionary used is direct-mapped. Therefore, the amount of memory used by the program is $O(4^n)$. In addition, the n-grams must fit within an int; each nucleotide is 2 bits, so with a 32-bit int, $n \leq 16$. Finally, the program assumes we are on a big-endian machine.

6.7 comp_dna, dcomp_dna

Usage: `comp_dna <fna-file>`

Compresses the FNA-format file by representing each nucleotide by only 2 bits rather than 8. Any characters not in [acgt] are ignored. The compressed file is output to stdout; this should be redirected to a file.

`dcomp_dna <cmp-file>`
Decompressed the file to stdout. The output is simply a long string of characters in [ACGT]; there is no whitespace, or any other characters. This is suitable *e.g.* as input to `proc_dna`.

Example: `comp_dna haemo.fna < haemo.cmp`
Compresses `haemo.fna` into `haemo.cmp`. The file size should fall to about a quarter. However, we will get warnings that `haemo.fna` contains characters not in [acgt]; these are simply ignored.

`dcomp_dna haemo.cmp | proc_dna - 1000` Uncompresses `haemo.cmp` and splits it into fragments of length 1000 (using `proc_dna`).

File format: The input format is:

1. 1-line comment (this is a bug: we should allow any number of comment-lines, each starting with a greater-than character).
2. list of nucleotides, in ASCII. Whitespace is allowed. Characters not in $\{A, C, G, T\}$ are allowed, to denote as-yet undetermined nucleotides.

The file format is:

1. 4-byte magic cookie (big-endian): `{ 0xac, 'g', 't', '\n' }`
2. 1-line (until new-line, or up to 256 bytes) comment
3. list of 2-bit nucleotides, $\{a, c, g, t\} \mapsto \{0, 1, 2, 3\}$

Note the file is a whole number of bytes long; if we do not have a multiple of four nucleotides in the file, then the file will end with several 'a' characters.

Installation: `comp_dna` and `dcomp_dna` are actually one and the same. If the program is invoked as `dcomp_dna`, the file argument is decompressed; otherwise, it is compressed. To do this, simply create a link from `dcomp_dna` to `comp_dna`:

```
ln -s comp_dna dcomp_dna
```

6.8 dna_loop.pl

Usage: `dna_loop.pl [opts] <file> <k1> [<k2> ...]`
Wraps around `real_dna.pl`. Sets up to run all combinations of (s, r) chips (except $s = 1$) for each value of k. m starts at $(r + 1)(s + 1)$ and ends at 50000. File is assumed to be compressed using `comp_dna`. No action is actually taken, except to print out the list of commands which would actually run all the tests. This can be redirected to a file to be run later.

Options: Options are exactly those of `real_dna` and are passed on without being interpreted.

Example: `dna_loop.pl -s1.05 ecoli.cmp 9 | sh`

Runs a test of sequencing *E. coli* using chips with $k = 9$, and increasing the fragment length by 5% per point.

`dna_loop.pl -s1.05 -d90 -m ecoli.cmp 8 9 10 | prun -`

Uses prun to schedule a test of *E. coli* using chips with $k = 8$, 9, and 10, increasing the fragment length by 5%, and stopping when the recognition rate falls below 90%. Mail is sent when a job (all tests for a particular (s, r) chip) is done.

6.9 interp, interp.words

Usage: `interp` or `interp.words`

Interprets the output of `gap`. `interp` prints out a line of output in colon-separated format: total number of sequences, number correct, percentage correct, and number of failures. `interp.words` prints out about the same information, in more easily readable format.

Example: `my_seqgen 500 500 | gap 3 4 - | interp.words`

Run `gap` over 500 random sequences of 500 nucleotides each, using a $(3, 4)$ chip; use `interp` to tell us how many of them it correctly sequenced.

6.10 printtable.pl

Usage: `printtable.pl [opts] <file1> [<file2> ...]`

Prints one row of a table, suitable for inclusion in a LaTeX document. Each file contributes one column. The table contains the value of m in each file which was the first time at which the success rate fell below 95%. The files are assumed to be in the format of the output of `real_dna.pl`, although other colon-separated formats can be accommodated.

Options:

- d *pct*: Use *pct* rather than 95% as the threshold.
- m *index*: Use *index* for the index of the field whose value is m. The fields must be colon-separated. The index number is 0-based.
- s *index*: Use *index* for the index of the field whose value is the success rate. The fields must be colon-separated. The index number is 0-based.

Example: `printtable.pl -m2 -s5 output.7*`

Print a table with one column per value of s and r, for $k = 7$ using the output format of `tester.pl`.

`printtable.pl -d90 haemo.cmp.*`

Print a table with one column per value of s and r, for all chips tested using `real_dna.pl` on *H. influenzae*. This row corresponds to the value of m at which less than 90% of the sequences were successfully reconstructed.

6.11 proc_dna

Usage: `proc_dna <file> <m>`
Splits an input file into multiple sequences each of length m. The output is in the format `gap` expects.

Example: `dcomp_dna ecoli.cmp | proc_dna - 1000`
Splits the *E. coli* genome into about 4500 sequences, each of length 1000.

6.12 display_graph

Usage: `display_graph <file>`
Displays the file using `gv -landscape`, which it forks into the background.

Example: `display_graph output.8.ps`

6.13 listall

Usage: `listall <k> [<k2> ...]`
List all files `output.s.r` where $s + r = k$.

Options: i *name*: Specify a name other than "`output`".

Example: `tar cf all9.tar 'listall 9'`
Creates a tar archive of all files created by splitfiles, with $k = 9$.

6.14 plotall

Usage: `plotall [options] <k> [<k2> ...]`
Generates plots for the data in postscript format.

Options:

i *name*:	specify name to include (default 'output')
n *file*:	don't run gnuplot; store the commands in *file*
o *name*:	specify output name
t *title*:	specify the plot title
x *range*:	specify a range (don't include brackets)
y *range*:	same as above, for y coordinate
b:	label the graph
c:	output to color postscript (rather than B/W)
e:	output to eps; uses .eps extension rather than .ps

Examples: `plotall -cb -y90:100 -ihaemo 8`

Generates a color plot for the $k = 8$ run of *H. influenzae*. The plot will only include data with a success rate of 90% or above; there will be labels on each curve at the last point before the curve falls outside of the graph.

`plotall -b -o"|lpr" 9`

Generates a plot for the $k = 9$ run of uniformly distributed data, including labels on the curves. The plot is immediately printed.

`plotall -be -t" " -y90:100 -n uniform.h0 7 8 9 10`

Write a file `uniform.h0` containing the gnuplot commands to generate plots "output.7.eps" and so on in eps format. These will have labels on the curves; the title is blank; the success rate is at least 90%. We can then modify `9.gnu` to change, for example, the precise locations of the labels. The easiest way to generate the graphs is to run `gnuplot < 9.gnu`; we can also, within gnuplot, do `load 'uniform.h0'`.

Bugs: There is no way to specify we don't want a title.

The -o option only applies either if it's a pipe to some other program, or if we're plotting only a single value of k.

After specifying the y axis, the x axis should be made to be only large enough to fit the data.

Labels are put at the last point at which the curve has not yet dropped below the y axis. However, they do not take account of the x axis.

6.15 rmall

Usage: `listall <k> [<k2> ...]`

Remove all files `output.s.r` where $s + r = k$. Such files are created by `splitfiles`.

Options: i *name*: Specify a name other than `output`.

Example: `rmall -iecoli 8 9 10`

Remove all files related to *E. coli*, with $k = 8$, 9, or 10.

6.16 sortall

Usage: `listall <k> [<k2> ...]`

Sort all files `output.s.r` where $s + r = k$. Such files are created by `splitfiles`. Useful as the last step before using `plotall`.

Options: i *name*: Specify a name other than `output`.

Example: `sortall -ihaemo 9`

Sort all files related to *H. influenzae*, $k = 9$.

6.17 split_extend

Usage: `split_extend <files>`
Splits the shift counts of the given files into distinct files. A directory should exist for each shift value. Files are deposited in those directories with name `o.`s`.`r, where s and r are taken from the input files. The output is assumed to be that of `tester.pl`.

Example: `split_extend ../output.*`
Splits all output files of `tester.pl`.

6.18 split_real

Usage: `split_real <file1> [<file2> ...]`
Splits files which are the output of `real_dna.pl` into input for plotall *et al.* The filenames are assumed to be in the form `<name>.*.s.r`

Example: `split_real haemo.cmp.*`
Splits all of the files output from running `real_dna.pl` on the *H. influenzae* sequence into files `haemo.8.0` and so on.

6.19 splitfiles

Usage: `splitfiles <field> <list of files>`
Splits the files listed into a format appropriate for plotall (and gnuplot). Will produce files called "output.s.r" (varying s and r), with lines of form `m value` where `value` is taken from the specified field.
The input format is colon-separated, starting with s:r:m
The field number is 0-based (s is field 0)

Example: `splitfiles 5 ../output.8*`
Splits all the files output by `tester.pl`, with $k = 8$. Field 5 in those files is the percent success rate, so `plotall 8` after this will plot the success rate of the last run of $k = 8$. Note the output may not be sorted; use `sortall`.

7 Further Work

Some of the work that remains to be done has been alluded to already. These are some possible directions for further work.

- The algorithm code should be made to comply to the division of labour as outlined in the algorithm section. Currently, LOOKUP is only implicitly implemented. This would make it easier to reimplement LOOKUP as a lookup in a trie rather than in a hash table.
- Throughout this work, we have assumed perfect hybridization. In practice, some errors occur; in fact, this has been a problem with previous SBH schemes. The behavior of the algorithm with real sets of fired probes should be studied.
- Some of the graphs have unexplained characteristics. For instance, the graphs of success rates for some of the real data look rather linear, unlike those for random data. Similarly with the spikes in the graphs of calls to LOOKUP in skewed data.

References

[1] I. Belyi and P. A. Pevzner. DNA_SPECTRUM software package, 1996.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[3] National Center for Biotechnology Information.
http://www.ncbi.nlm.nih.gov.

[4] P. Pevzner, Y. P. Lysov, K. R. Khrapko, A. V. Belyavsky, V. L. Florentiev, and A. D. Mirzabekov. Optimal chips for megabase dna sequencing. *J. Biomol. Struct. and Dyn.*, 9:399–410, 1991.

[5] P. A. Pevzner and R. J. Lipshutz. Towards DNA sequencing by hybridization. In *19th Symposium on Mathematical Foundations of Computer Science*, pages 143–258, 1994.

[6] F. P. Preparata, A. M. Frieze, and E. Upfal. On the power of universal bases in sequencing by hybridization. In *RECOMB99*, 1999.