ABSTRACT
Summary: Tandem Repeat Occurrence Locator (TROLL), is a light-weight Simple Sequence Repeat (SSR) finder based on a slight modification of the Aho–Corasick algorithm. It is fast and only requires a standard Personal Computer (PC) to operate. We report running times of 127 s to find all SSRs of length 20 bp or more on the complete Arabidopsis genome—approx. 130 Mbases divided in five chromosomes—using a PC Athlon 650 MHz with 256 MB of RAM.
Availability: TROLL is an open source project and is available at http://finder.sourceforge.net.
Contact: castelo@capsl.udel.edu

Genomes, particularly of eukaryots, contain a large number of repeated sequences. Simple motifs, known as Simple Sequence Repeats (SSRs) or microsatellites, are fairly abundant. In plant genomes, for example, they are reported to occur once in approximately every 6 Kb (thousands of base pairs) (Cardle et al., 2000). Single nucleotide repeats have been used in the population genetic analysis of chloroplast genomes (Powell et al., 1995). Easily assayed with PCR and highly polymorphic, these structures can be used as genetic mapping markers (Morgante and Olivieri, 1993) and in population genetics (Powell et al., 1995).

The algorithmic approaches to find repeats are basically two. A model can be defined for a repeat and later used to locate regions satisfying the definition. This approach provides a comprehensive list of repeats, exact or degenerate, and has been implemented on some tools such as the Tandem Repeats Finder (Benson, 1999) and Sputnik (Abajian, 1994). However, when the repeat sequences (motifs) are known a priori, the so-called dictionary approach provides a faster and more scalable solution to the problem. This approach uses a dictionary of known motifs and scans the input sequence searching for dictionary entries.

In this work we present a program, Tandem Repeat Occurrence Locator (TROLL), that uses the dictionary approach to find tandem repeats of pre-selected motifs.

ALGORITHM AND IMPLEMENTATION
TROLL is based on the Aho Corasick Algorithm (ACA) (Aho and Corasick, 1975) which was conceived to solve the dictionary problem, i.e. to find all occurrences from a list of patterns in a text string. The ACA uses a keyword tree to find all occurrences of any pattern from a set in a text string. The keyword tree also stores information about pattern similarity in the form of failure links. When the algorithm encounters a partial match in the text, at the mismatch character, it uses the failure links to continue the search without having to re-sample characters in the text. Use of this mechanism allows the ACA to achieve linear time complexity (Aho and Corasick, 1975; Gusfield, 1997).

TROLL uses the ACA to find pre-selected patterns in a text string, but also keeps track of tandem repetitions so that microsatellites (SSRs) can be located. In this context, patterns become motifs and the text string becomes a DNA sequence. Figure 1(a) shows an overview of TROLL’s operation structure. In stage 1, the motif file provided by the user is used to build the keyword tree—this is the pre-processing stage. In stage 2, the input sequences are scanned one by one. With the use of repeat buffers, TROLL keeps track of which motif occurrences constitute a valid tandem repeat—according to the user defined minimum length—and which ones are just isolated motif appearances. Buffer operations, reads and writes, are done in constant time so that the overall time complexity of the algorithm is kept linear.

The Repeat Buffers
TROLL uses repeat buffers to avoid missing repeats or counting the same occurrence multiple times. Each time it encounters a match, i.e. a motif from the motif list that is present in the DNA sequence, it refers to the repeat buffers to check whether this match continues an SSR or is a potential seed for one—we will refer to this process as the SSR test.

For each possible motif size given by the user, a set of buffers is created—see box Repeat Buffers in Figure 1(a). Together, all the buffers form a triangular $n \times n$ matrix
from the text. Figure 1(b) shows the procedure where $k$ algorithmic notation. 

whenever any cell be reset to zero. This test is performed in linear time. 

$(SSR (cell pi is given by a buffer pointer formula, 

pared to $R_i$ to account for the appearance of these redundant repeats a 

viously, for each motif length

$k$ at position

$at a motif for a SSR (cell 

A motif shift-

ing

is the position of the current character being read

Each SSR test involves one buffer position only, which is given by a buffer pointer formula, $p_i(k) = \text{mod}(k/i)$, where $k$ is the position of the current character being read from the text. Figure 1(b) shows the procedure SSRTest, in algorithmic notation. $m$ is a match for a motif of length $i$ at position $k$ in the DNA sequence.

At any given time, up to $i$ repeats may occur simultaneously, for each motif length $i$. This is due to motif shifting, i.e. when a repeat of a motif longer than 1bp also includes a repeat of its shifts. For example, ATCATCAT will be a repeat of (ATC)$_2$, (TCA)$_2$ or (CAT)$_2$. In order to account for the appearance of these redundant repeats a master record $R_i$ is used. $R_i$ stores the length of the longest occurrence in a line $i$, at any given moment, and is updated whenever any cell $p_i(k)$ is updated. When a buffer position $cell[p_i(k)]$ is to be replaced, its cell $p_i(k)]$.length is compared to $R_i$; if they have the same value, then the repeat to be replaced is guaranteed to be the current longest for this motif length, and will be compared to the minimum repeat length set by the user before being discarded. $R_i$ will then be reset to zero. This test is performed in linear time.

Complexity Analysis

TROLL can be divided in three parts. The pre-processing module (PP), the Aho–Corasick module (AC) and the buffer control module (BC). The PP module is called first during execution, to build the keyword tree from a given set of input patterns. This task is done only once per execution of the program. The AC module is in charge of finding the patterns from the keyword tree in the text. At every match, it calls the BC module. The BC module is the SSRTest procedure, that detects and keeps track of repeats. From Aho and Corasick (1975); Gusfield (1997), we have that PP and AC have time complexity of $O(n + m + k)$, respectively, where $n$ is the total length of all patterns being searched, $k$ is the number of occurrences found and $m$ is the length of the text string. The time complexity for the BC module is $O(k)$ since all its operations take constant time, as can be seen from Figure 1(b). Therefore, the time complexity for TROLL is $O(n + m + k)$.

INTERFACE

TROLL is designed to have a powerful yet simple interface. The operation is performed through command line interaction. The input arguments let the user indicate the SSRs’ minimum length desired, the maximum motif length and the files containing the motif list and the DNA sequence. The program comes with a list of all motifs up to 5 bp. The output lists starting position of each repeat, the motif, and the repeat length (in bp).

For example, the command sequence troll -M5 -m30 motifs.dat chromosomes1.dat > results.dat will cause TROLL to search the file chromosomes1.dat for SSRs of length at least 30 bp (-m30), composed of any motif listed in the file motifs.dat, and store the output in the file results.dat. The -M5 indicates the length of the longest motif in motifs.dat, i.e. 5 bp.

Flexibility was the main guideline in our design: the output file can be easily imported into other applications, e.g. Matlab, or spread sheets, and Unix shell pipelines can easily send TROLL’s output through user-designed filter scripts.
Table 1. Execution times for TROLL and Sputnik. The test load is the Arabidopsis thaliana first chromosome (approx. 33 Mb long). Platform: AMD Athlon 650 MHz, 256 MB RAM, Slackware Linux 8.0. Average time for 10 executions, Negligible standard deviation

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TROLL</td>
<td>41.33 s</td>
</tr>
<tr>
<td>Sputnik</td>
<td>47.24 s</td>
</tr>
</tbody>
</table>

**PERFORMANCE**

TROLL’s sole purpose is to find exact tandem repeats. This specialized implementation gives TROLL the advantage over more generalist tools.

The most well known programs that list tandem repeats from a DNA sequence are Tandem Repeat Finder (TRF) (Benson, 1999) and Sputnik (Abajian, 1994). Both also find *degenerate tandem repeats*, i.e. sequences where the motif is repeated with a certain degree of variation, defined by a score system similar to sequence comparison scoring—usually a positive score for matches, and negative values for mismatches and gaps.

Table 1 shows an execution time comparison between Sputnik and TROLL. A small modification was necessary on the Sputnik source code, where we increased the size of the input buffer to allow for the file sizes in this test. Both programs were compiled using the same level of optimization. The input DNA sequence was 33 Mb (millions of base pairs) long (the first chromosome of Arabidopsis thaliana). The search was for exact tandem repeats of length greater than 19 bp, and all motifs 1 to 5 bp long, except motifs that are tandem repeats of others—e.g. ATAT is a tandem repeat of AT. The total number of motifs searched was 1336. TRF was also tested, but it was left out of Table 1 because it only allows for input sequences of at most 5 Mb. Since its source code was not available, a modification was not possible. With a 5 Mb input, TRF’s average execution time was 41.6 s. The tests were performed on an AMD Athlon 650 MHz, with 256 MB of RAM, running Slackware Linux 8.0.

TROLL shows better performance than the available tools, achieved through specialization. Besides speed, TROLL’s advantages include an uncluttered output format that allows for easy integration with other tools. The command line interface may seem unfriendly to some users, but it can be mastered easily.

**ACKNOWLEDGMENTS**

The authors thank Dr. J.-Antoni Rafalski and Dr. Michele Morgante for discussions and feedback on this work. This work was supported by the Delaware Biotechnology Institute (DBI), the Computer Architecture and Parallel Systems Laboratory (CAPSL) and the UNIDEL-99N grant, from the University of Delaware.

**REFERENCES**