Additional Searching Algorithms

N3411=12-0101

2012-09-23

Marshall Clow (mclow.lists@gmail.com)

Overview

`std::search` is a powerful tool for searching sequences, but there are lots of other search algorithms in the literature. For specialized tasks, some of them perform significantly better than `std::search`. In general, they do this by precomputing statistics about the pattern to be searched for, betting that this time can be made up during the search.

I am proposing adding two new search algorithms to the standard library:

```
* Boyer-Moore
* Boyer-Moore-Horspool
```

Interface

Both algorithms have the same interface

In this section, I will refer to `xxsearch`, a wild-card for both algorithms.

Procedural interface

The basic interface is the same as std::search, with the exception that the predicate parameter is not supported.

```
template <typename CorpusIter, typename PatternIter>
CorpusIter xxsearch ( CorpusIter corpus_first, CorpusIter corpus_last,
        PatternIter pattern_first, PatternIter pattern_last );
```

Effects: Finds a subsequence of equal values in a sequence.

Returns: The first iterator i in the range `[corpus_first,corpus_last - (pattern_last-pattern_first))` such that for any non-negative integer `n` less than `pattern_last - pattern_first` the following corresponding condition holds: `*(i + n) == *(pattern_first + n)`. Returns `corpus_first` if `[pattern_first,pattern_last)` is empty, otherwise returns `corpus_last` if no such iterator is found.

`CorpusIter::value_type` and `PatternIter::value_type` need to be equality comparable (and possibly more)

Object Interface

Since these algorithms compute information about the pattern that they are searching, it makes sense to save this information, in case the user wants to search for the same pattern in several different data sets. We define an object with the following interface: *(again, with a generic name)*

```
template <typename PatternIter>
class xxsearch_obj {
public:
    xxsearch_obj ( PatternIter first, PatternIter last );
//  Copy and move constructors and assignment operators here
```

```
    template <typename CorpusIter>
    CorpusIter operator ( CorpusIter corpus_first, CorpusIter corpus_last ) const;
};
```

This adds an additional constraint upon the user; the pattern must remain unchanged between the time that the constructor is called, and the time that the final call to `operator()` returns.

These search objects can be copied and moved - though that is not shown in this section.

The procedural interface can be trivially implemented in terms of the object interface:

```
template <typename CorpusIter, typename PatternIter>
CorpusIter xxsearch ( CorpusIter corpus_first, CorpusIter corpus_last,
        PatternIter pattern_first, PatternIter pattern_last ) {
    xxsearch_obj<PatternIter> ( pattern_first, pattern_last );
    return obj ( corpus_first, corpus_last );
    }
```

Note: Implementing such an object interface for std::search is also trivial, since `std::search` does no pre-computation (again, leaving out copy/move operations):

```
template <typename PatternIter>
class std_search_obj {
public:
    std_search_obj (PatternIter first, PatternIter last) : first_ (first), last_(last) {}

    template <typename CorpusIter>
    CorpusIter operator ( CorpusIter corpus_first, CorpusIter corpus_last ) const {
        return std::search ( corpus_first, corpus_last, first_, last_ );
        }

private:
    PatternIter first_;
    PatternIter last_;
};
```

Individual Algorithm Details

Boyer-Moore

The Boyer–Moore string search algorithm is a particularly efficient string searching algorithm, and it has been the standard benchmark for the practical string search literature. The Boyer-Moore algorithm was invented by Bob Boyer and J. Strother Moore, and published in the October 1977 issue of the Communications of the ACM, and a copy of that article is available at http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf; another description is available on Wikipedia

The Boyer-Moore algorithm uses two precomputed tables to give better performance than a naive search. These tables depend on the pattern being searched for, and give the Boyer-Moore algorithm larger a memory footprint and startup costs than a simpler algorithm, but these costs are recovered quickly during the searching process, especially if the pattern is longer than a few elements. Both tables contain only non-negative integers.

In the following discussion, I will refer to `pattern_length`, the length of the pattern being searched for; in other words, `std::distance ( pattern_last, pattern_first )`.

The first table contains one entry for each element in the "alphabet" being searched (i.e, the corpus). For searching (narrow) character sequences, a 256-element array can be used for quick lookup. For searching other types of data, a hash table can be used to save space. The contents of the first table are: For each element in the "alphabet" being processed (i.e, the set of all values contained in the corpus) If the element does not appear in the pattern, then `pattern_length`, otherwise `pattern_length - j`, where `j` is the maximum value for which `*(pattern_first +`

`j) == element.`

Note: Even though the table contains one entry for each element that occurs in the corpus, the contents of the table only depend on the pattern.

The second table contains one entry for each element in the pattern; a `std::array<pattern_length>` works well. Each entry in the table is basically the amount that the matching window can be moved when a mismatch is found.

The Boyer-Moore algorithm works by at each position, comparing an element in the pattern to one in the corpus. If it matches, it advances to the next element in both the pattern and the corpus. If the end of the pattern is reached, then a match has been found, and can be returned. If the elements being compared do not match, then the precomputed tables are consulted to determine where to position the pattern in the corpus, and what position in the pattern to resume the matching.

The Boyer-Moore algorithm requires that both `CorpusIter` and `PatternIter` be random-access iterators.

Boyer-Moore-Horspool

The Boyer-Moore-Horspool search algorithm was published by Nigel Horspool in 1980. It is a refinement of the Boyer-Moore algorithm that trades space for time. It uses less space for internal tables than Boyer-Moore, and has poorer worst-case performance.

Like the Boyer-Moore algorithm, it has a table that (logically) contains one entry for each element in the pattern "alphabet". When a mismatch is found in the comparison between the pattern and the corpus, this table and the mismatched character in the corpus are used to decide how far to advance the pattern to start the new comparison.

A reasonable description (with sample code) is available on Wikipedia

The Boyer-Moore algorithm requires that both `CorpusIter` and `PatternIter` be random-access iterators.

Performance

These tests were run with a corpus approximately 2 million entries long, with patterns of length in the range 150 .. 200 elements. The data was base64-encoded text. Each test was run 100 times, and the results normalized to `std::search`. The "object" tests used the same object for each text, without rebuilding the internal data tables.

The titles of the test indicate where the pattern is located in the corpus being searched; "At Start", etc. "Not found" is the case where the pattern does not exist in the corpus, i.e, the search will fail.

Algorithm	At Start	Middle	At End	Not found
std::search	100.0	100.0	100.0	100.0
Boyer-Moore (procedural)	486.1	10.07	15.91	9.236
Boyer-Moore (object)	39.36	9.859	15.09	8.647
Boyer-Moore-Horspool (procedural)	154.2	9.111	12.12	8.113
Boyer-Moore-Horspool (object)	36.96	8.896	12.88	8.476

An implementation of these algorithms (and the test program) is available in the Boost.Algorithm library.

Source code is available for download as part of the Boost 1.51.0 release at
http://sourceforge.net/projects/boost/files/boost/1.51.0/