P0638R0: Crochemore-Perrin search algorithm for `std::search`

Date: 2017-05-03
Reply-to: Ed Schouten <ed@nuxi.nl>
Audience: Library

Overview

N3905 extended the `std::search` function to support external search algorithms. Implementations of the Boyer-Moore and Boyer-Moore-Horspool algorithms are provided by default. This proposal attempts to add a third algorithm, namely the Two-Way String Matching algorithm by Maxime Crochemore and Dominique Perrin, which has a couple of very interesting properties:

- It has a linear worst-case running time, both during the preprocessing and matching phases.
- It's in-place: it requires a constant amount of memory, both during computation and storage of the preprocessed state. Memory usage is also independent with respect to the size of the alphabet.

Though the Crochemore-Perrin search algorithm isn't very well-known by name, it is being used in practice. It's a perfect fit for implementing `strstr()`, `wcsstr()` and `memmem()`, which is why it is used by many modern C libraries, like glibc, musl, FreeBSD's libc, etc.

The algorithm works by cutting (*factorizing*) the pattern in two pieces. While matching, the algorithm scans through the input, searching for the pattern's suffix. Only when a full match of the suffix has been found, it attempts to match the prefix. By choosing the position at which the pattern is factorized carefully (yielding a *critical factorization*), the algorithm is capable of skipping larger amounts of input upon mismatches (based on the pattern's *period*). This makes the algorithm run in linear time.

Ordered and unordered alphabets

What's truly novel about the Crochemore-Perrin algorithm is the way it computes the critical factorization and period of the pattern, which again is done in-place and in linear time. To realize this, the algorithm requires that the alphabet has a total order. In 2015, Dmitry Kosolobov published an algorithm for computing the critical factorization only assuming an equivalence relation. Unfortunately, this algorithm is not in-place, which defeats the purpose.

Introducing a searcher that uses `std::less` is actually fairly consistent. Just like for our sets and maps, we will now have two different types of searchers. Ones that depend on hashing and equivalence and another one that depends on a total order.

Wording

Add to [functional.syn], header <functional> synopsis, under ‘searchers’:

```
template<class RandomAccessIterator,
         class BinaryPredicate = less<>>
  class crochemore_perrin_searcher;
```

Add a new paragraph to the end of [func.search], searchers:

> The Crochemore-Perrin searcher implements the Crochemore-Perrin (“Two-Way”) search algorithm. Preprocessing and matching both use only a constant amount of memory, while still providing a linear worst-case running time. This algorithm requires that a total order on the alphabet can be defined.

Add a new section after [func.search.bmh], class template boyer_moore_horspool_searcher:

```
  template <class RandomAccessIterator1,
            class BinaryPredicate = less<>>
    class crochemore_perrin_searcher {
    public:
      constexpr crochemore_perrin_searcher(RandomAccessIterator1 pat_first,
                                           RandomAccessIterator1 pat_last,
                                           BinaryPredicate pred = BinaryPredicate());
      template <class RandomAccessIterator2>
        constexpr pair<RandomAccessIterator2, RandomAccessIterator2>
          operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
    private:
      RandomAccessIterator1 pat_first_;      // exposition only
      RandomAccessIterator1 pat_last_;       // exposition only
      BinaryPredicate pred_;                 // exposition only
    };
```

```
constexpr crochemore_perrin_searcher(RandomAccessIterator1 pat_first,
                                     RandomAccessIterator1 pat_last,
                                     BinaryPredicate pred = BinaryPredicate());
```

Requires: The value type of `RandomAccessIterator1` shall meet the `DefaultConstructible`, `CopyConstructible`, and `CopyAssignable` requirements.

Effects: Constructs a `crochemore_perrin_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, and `pred_` with `pred`.

Throws: Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1` or the copy constructor or `operator()` of `BinaryPredicate`.

Complexity: At most O(`pat_last` - `pat_first`) applications of the predicate.

```
template <class RandomAccessIterator2>
  constexpr pair<RandomAccessIterator2, RandomAccessIterator2>
operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
```

Requires: Copy existing phrasing from [func.search.bmh]'s `operator()`.

Effects: Copy existing phrasing from [func.search.bmh]'s `operator()`.

Returns: Copy existing phrasing from [func.search.bmh]'s `operator()`.

Complexity: At most O(`last` - `first`) applications of the predicate.

In [algorithm.syn], header <algorithm> synopsis, under 'search', add `constexpr` to this prototype:

```
template <class ForwardIterator, class Searcher>
  ForwardIterator search(ForwardIterator first, ForwardIterator last,
                         const Searcher& searcher);
```

In [alg.search], search, add `constexpr` to this prototype:

```
template<class ForwardIterator, class Searcher>
  ForwardIterator search(ForwardIterator first, ForwardIterator last,
                    const Searcher& searcher);
```

Example implementation

The pseudocode given in figures 17 and 21 of the original paper does a pretty good job of describing how the algorithm can be implemented. When making use of this pseudocode, keep in mind that it uses 1-indexed arrays.

An example implementation based on the pseudocode can be found on GitHub.