The SPARE Parts: A C++ toolkit for String PAttern REcognition

Bruce W. Watson
Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada
E-mail: watson@RibbitSoft.com
Web: www.RibbitSoft.com
Phone: +1-250-762-6591
Fax: +1-250-762-6511

1 Introduction and related work

In this paper, we outline the design, design rationale, and use of the SPARE Parts, a C++ toolkit for string pattern matching. The algorithms implemented in the toolkit include:

- The Knuth-Morris-Pratt [KMP77] single keyword pattern matching algorithm.
- Several variants of the Boyer-Moore [BM77] single keyword pattern matching algorithms.
- Several variants of the Aho-Corasick [AC75] multiple keyword pattern matching algorithms.
- Several variants of the Commentz-Walter [Com79a, Com79b] multiple keyword pattern matching algorithm.

In addition to the original papers, all of these algorithms are extensively treated (with full correctness arguments) in a taxonomy presented in [Wats95, Chapter 4].

The SPARE Parts is the second generation string pattern matching toolkit from the Eindhoven University of Technology. The toolkit is publicly available (though not in the public domain) for noncommercial use. A re-implemented (and greatly extended) toolkit, known as the SPARE Parts II, is available for commercial licensing from Ribbit Software Systems Inc. In addition to the functionality of the SPARE Parts, it contains approximate pattern matcher, regular pattern matchers, multi-dimensional pattern matchers, and a highly tuned set of foundation classes. The first generation toolkit (called the Eindhoven Pattern Kit, written in C, and described in [Wats94, Appendix A]) is a procedural library based upon the original taxonomy of pattern matching algorithms [WZ92]. Experience with the toolkit revealed a couple of deficiencies, detailed as follows. The rudimentary and explicit memory management facilities in C caused numerous errors in the code, and made it difficult to perform pattern matching over more than one string simultaneously (in separate threads of the program) without completely duplicating the code. While the performance of the toolkit was excellent, some of the speed was due to sacrifices made in the understandability of the client interface.

There are other existing pattern matching toolkits, notably the toolkit of Hume and Sunday [HS91]. Their toolkit consists of a number of implementations of Boyer-Moore type algorithms — organized so as to form a taxonomy of the Boyer-Moore family of algorithms. Their toolkit was primarily designed to collect performance data on the algorithms. As a result, the algorithms are implemented (in C) for speed and they sacrifice some of the safety (in terms of error checking) that would normally be expected of a general toolkit. Furthermore, the toolkit does not include any of the non-Boyer-Moore pattern matching algorithms (other than a brute-force pattern matcher) — most noticeably, there are no multiple keyword pattern matchers.

The SPARE Parts is a completely object-oriented implementation of the algorithms appearing in [Wats95, Chapter 4]. The SPARE Parts is designed to address the shortcomings of both of the toolkits described above. The following are the primary features of the class library:

- The design of the SPARE Parts follows the structure of the taxonomy in [Wats95, Chapter 4] very closely. As a result, the code is easier to understand and debug. In addition, the SPARE Parts includes implementations of almost all of the algorithms described in [Wats95, Chapter 4].
- The use of C++ (instead of C) for the implementation has helped to avoid many of the memory management-related bugs that were present in the original toolkit.
- The client interface to the toolkit is particularly easy to understand and use. The flexibility introduced into the interface does not reduce the performance of the code in any significant way.
- The toolkit supports multi-threaded use of a single pattern matching object.

This paper is structured as follows:

- Section 2 considers issues in the design and implementation of class libraries.
- Section 3 gives an introduction to the client interface of the toolkit. It includes some examples of programs which use the SPARE Parts.
- Section 4 presents some experiences with the toolkit and the conclusions of this chapter.
- Section 5 gives some information on how to obtain and compile the toolkit.
- The appendices provide more extensive details about the C++ classes.

2 Designing and implementing class libraries

In this section, we briefly discuss some of the issues involved in designing, implementing, and presenting class libraries (or *toolkits*). The following description of a toolkit is taken from [GHJV95, p. 26]:

> A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They are the object-oriented equivalent of subroutine libraries.

We will use the terms *class library, library,* and *toolkit* interchangeably. We will also use the term *client* to refer to a program that makes use of classes in the toolkit, or the author of such a program. The important aspects and design goals of a toolkit are:

- Toolkits do not provide a user interface. (Toolkits that do provide user interfaces should be placed in the category of 'application program'.)
- The classes in the toolkit must have a coherent design, meaning that they are designed and coded in the same style. They have a clear relationship and a logical class hierarchy.
- The client interface to the library must be easily understood, permitting clients to make use of the library with a minimum of reading.

- The efficiency of using the classes in the toolkit must be comparable to hand-coded special-purpose routines — the toolkit must be applicable to production quality software.
- To provide an educational use for the toolkits, and to allow clients to easily modify classes and member functions, the method of implementation must be clear and understandable.

The toolkit is implemented in the C++ programming language, which was chosen because of its object-oriented features and its widespread availability. Efforts were made to refrain from using obscure features of C++ (such as RTTI or name spaces), or language features not easily found in other object-oriented programming languages (such as multiple-inheritance).

Throughout this paper, we assume that the reader is familiar with the C++ language and object-oriented terminology (especially the C++ variety).

The general process of library design will not be described here, as there is a large body of literature discussing this issue. The following books are of particular relevance:

- [GHJV95, Souk94] discuss 'design patterns' (not related to our pattern matching problem) which are used heavily in library design.
- [CE95], [Stro91, Chapter 13] and [Stro94, Chapter 8] provide a general discussion of C++ library design.
- [MeyB94] is an excellent treatment of the design of a number of loosely coupled libraries in the Eiffel programming language. Many of the concepts and techniques discussed in the book are broadly applicable to C++ as well.
- [Plau95, Teal93] discuss the design and implementation of specific C++ libraries — the standard C++ library[1] and the IOStreams (input and output) class libraries respectively.

The toolkit related terms that we will use are defined as follows. We define the following types of object-oriented classes:

User: A class intended for use by a client program.

Client: A class defined in the client program.

Implementation: A class defined in the toolkit for exclusive use by the toolkit. The class is used to support the implementation of the client classes.

Foundation: Those implementation classes which are simple enough to be reused in other (perhaps unrelated) class libraries.

Interface: An abstract (pure virtual) class which is declared to force a particular public interface upon its inheritance descendants.

Base: An inheritance ancestor of a particular class.

Derived: An inheritance descendant of a particular class.

2.1 Motivations for writing class libraries

There are a number of motivations for creating the class libraries:

- Until now, few general purpose toolkits of pattern matchers existed. The ones that do exist are not intended for general use in production quality software.
- The level of coherence normally required to implement a toolkit was not previously possible. The literature on pattern matching algorithms was scattered and in some places incomplete. With the construction of the taxonomy, [Wats95, Chapter 4], all of the algorithms are described in a coherent fashion, allowing us to base the class library structures on the taxonomy structure.

[1] Plauger's book considers the implementation of an early, and now defunct, draft of the standard library

- The uniformity of implementation that was possible (given the taxonomy) had two important effects:
  - Clients need not examine the source code in order to make a decision on which class to use; the quality of the implementations of each of the pattern matchers is roughly the same.
  - Uniformity gives greater confidence in the accuracy of relative performance comparing different algorithms.
- The toolkit and the taxonomy can serve as examples of implementation techniques for class libraries; in particular methods for organizing template classes[2] and class hierarchies.
- Implementing the abstract algorithm can be painless and fun, given the taxonomy presentation of the algorithms and their correctness arguments.

2.2 Code sharing

One of the main aims of object-oriented programming is that it permits, and even encourages, code sharing (or code reuse). The code reuse in object-oriented programming corresponds neatly with the factoring of common parts of algorithms in the taxonomy.

Although code sharing can be achieved in a number of ways, in this section we discuss four techniques which could have been used in the design of the toolkits. The first discussion centres around the use of base classes (with virtual member functions) versus templates. The second discussion concerns the use of composition versus protected inheritance.

2.2.1 Base classes versus templates

A number of the pattern matching objects have common functionality, and it seems wasteful to duplicate the code in each of the specific types of pattern matchers.

The obvious design involves creating a new base class and factoring the common code into the base class. Each of the pattern objects would then inherit from this base, and provide specific virtual member functions to obtain the desired functionality. For example, the Commentz-Walter algorithms all share a common algorithm skeleton; they each have specific shift functions. We could create a CW base class with the functionality of the skeleton, and provide a virtual 'shift distance' member function to obtain the Commentz-Walter variants.

The advantage of this approach is its elegance. It provides a relatively easy to understand class hierarchy, which reflects the structure of the taxonomy. Furthermore, a member function which takes (as a parameter) a pointer to a CW object need not know which particular variant (of a CW object) the pointer points to, only that the CW object satisfies the general CW functionality. This solution provides code reuse at both the source language and executable image levels. The disadvantage is that it would require a virtual function call for every shift. Indeed, if the same technique was used to factor the common code from the Aho-Corasick variants, it would require a virtual function call for every character of the input string.

The other approach is to create a template class CW, which takes a 'shifter class' as its template (type) parameter. We would then provide a number of such shifter classes, for use as template parameters — each giving rise to one of the Commentz-Walter variants. The primary advantage of this approach is that it is efficient: when used to implement the Aho-Corasick algorithms, each character in the input string will require a non-virtual function call (which may be inlined, unlike virtual function calls). The disadvantages are twofold: pointers to the variants of the CW algorithms are not interchangeable, and code will be generated for each of the CW variants. The code reuse is at the source level, and not at the executable image level.

It is expected that few clients of the toolkits will instantiate objects of different CW classes, for example. A programmer writing an application using pattern matching is more likely to choose a

[2] We use the term *template class*, as opposed to *class template* suggested by Carroll and Ellis in [CE95]. Our choice was made to correspond to the term *generic class* used in some other object-oriented programming languages.

particular type of pattern matcher, as opposed to creating objects of various different types. For this reason, the advantages of the template approach are deemed to outweigh its disadvantages, and we prefer to use it over base classes in the toolkits.

2.2.2 Composition versus protected inheritance

Composition (sometimes called the *has-a* relationship) and protected inheritance (sometimes called the *is-a* relationship) are two additional solutions to code sharing. We illustrate the differences between these two solutions using an example. When implementing a *Set* class, we may wish to make use of an already-existing *Array* class. There are two ways to do this: composition and protected inheritance.

With protected inheritance, class *Set* inherits from *Array* in a protected way. Class *Set* still gets the required functionality from *Array*, but the protected inheritance prevents the *is-a* relation between *Set* and *Array* (that is, we cannot treat a *Set* as an *Array*). The advantage of this approach is that it is elegant, and it is usually the approach taken in languages such as Smalltalk and Objective-C [Budd91]. The disadvantage is that the syntax of C++ places the inheritance clause at the beginning of the class declaration of *Set*, making it plain to all clients of *Set* that it is implemented in terms of *Array*. Furthermore, protected inheritance (and indeed private inheritance) is one of the rarely-used corners of C++, and it is unlikely to be familiar to the average programmer [MeyS92, Murr93].

In a composition approach, an object of class *Set* has (in its private section) an object of class *Array*. The *Set* member functions invoke the appropriate member functions of *Array* to provide the desired functionality. The advantage of this approach is that it places all implementation details in the private section of the class definition. The disadvantage is that it deviates from the accepted practice (in some other languages) of inheriting for implementation. It is, however, the standard approach in C++. At first glance, it would appear that composition can lead to some inefficiency: in our example, an invocation of a *Set* member function would, in turn, call an *Array* member function. These extra function calls, usually called *pass-throughs*, are frequently eliminated through inlining.

There are no efficiency-based reasons to choose one approach over the other. For this reason, we arbitrarily choose composition because of the potential readability and understandability problems with protected inheritance.

2.3 Coding conventions and performance issues

At this time, coding in C++ presents at least two problems: the language is not yet stable (it is still being standardized) and, correspondingly, the "standard" class libraries are not yet stable.

In designing the libraries, every effort was made to use only those language features which are well-understood, implemented by most compilers and almost certain to remain in the final language. Likewise, the use of classes from the proposed standard library, or from the Standard Template Library [SL94], was greatly restricted. A number of relatively simple classes (such as those supporting strings, arrays, and sets) were defined from scratch, in order to be free of library changes made by the standardizing committee. A future version of the toolkits will make use of the standard libraries once the International Standards Organization has approved the C++ standard.

In the object-oriented design process, it is possible to go overboard in defining classes for even the smallest of objects — such as alphabet symbols. In the interests of efficiency, we draw the line at this level and make use of integers for such basic objects.

Almost all of the classes in the toolkits have a corresponding class invariant member function, which returns *TRUE* if the class is structurally correct, and *FALSE* otherwise. Structural invariants have proven to be particularly useful in debugging and in understanding the code (the structural invariant is frequently a good first place to look when trying to understand the code of a class). For this reason, they have been left in the released code (they can be disabled as described in the next section).

We use a slightly non-traditional way of splitting the source code into files. The public portion of a class declaration is given in a `.hpp` file, while the private parts are included from a `.ppp` file. There is a corresponding `.cpp` file containing all of the out-of-line member function definitions. A `.ipp` file contains member functions which can be inlined for performance reasons. By default the member functions in the `.ipp` file are out-of-line. The inlining can be enabled by defining the macro `INLINING`. To implement such conditional inlining, the `.ipp` file is conditionally included into the `.hpp` or the `.cpp` file. The inlining should be disabled during debugging or for smaller executable images.

3 Using the toolkit

In this section, we describe the client interface of the toolkit and present some examples of programs using the toolkit.

The client interface defines two types of abstract pattern matchers: one for single keyword pattern matching, and one for multiple keyword pattern matching. (A future version of SPARE Parts can be expected to include classes for regular expression pattern matching — for example, an implementation of the algorithm described in [Wats95, Chapter 5].) All of the single keyword pattern matching classes have constructors which take a keyword. Likewise, the multiple keyword pattern matchers have constructors which take a set of keywords. Both types of pattern matchers make use of *call-backs* (to be explained shortly) to register matched patterns. In order to match patterns using the call-back mechanism, the client takes the following steps (using single keyword pattern matching as an example):

1. A pattern matching object is constructed (using the pattern as the argument to the constructor).
2. The client calls the pattern matching member function *PMSingle::match*, passing the input string and a pointer *f* to a client defined function which takes an **int** and returns an **int**[3]. (This function is called the *call-back function*.)
3. As each match is discovered by the member function, the call-back function is called; the argument to the call is the index (into the input string) of the symbol immediately to the right of the match. (If there is no symbol immediately to the right, the length of the input string is used.)
4. If the client wishes to continue pattern matching, the call-back function returns the constant *TRUE*, otherwise *FALSE*.
5. When no more matches are found, or the call-back function returns *FALSE*, the member function *PMSingle::match* returns the index of the symbol immediately to the right of the last symbol processed.

We now consider an example of single keyword pattern matching.

The following program searches an input string for the keyword `hisher`, printing the locations of all matches along with the set of matched keywords:

```
#include "com-misc.hpp"
#include "pm-kmp.hpp"
#include <iostream.h>

static int report( int index ) {
    cout << index << '\n';
    return( TRUE );
```

[3]The integer return value is a Boolean value; recall that *TRUE* and *FALSE* have type **int** in C and C++. A recent draft of the C++ standard indicates that **bool** will be a new type (and a new keyword); the compilers used in the development of the SPARE Parts do not support this yet.

```
}

int main( void ) {
    auto PMKMP Machine( "hisher" );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

The header file `com-misc.hpp` provides a definition of constants *TRUE* and *FALSE*. Header file `pm-kmp.hpp` defines the Knuth-Morris-Pratt pattern matching class, while header file `iostream.h` defines the input and output streams, including the standard output *cout*. Function *report* is our call-back function, simply printing the index of the match (to the standard output), and returning *TRUE* to continue matching. The *main* function (the program mainline) creates a local KMP machine, with keyword `hisher`. The machine is then used to find all matches in string `hishershey`. (Recall that, in C and C++, a pointer to the beginning of the string is passed to member *match*, as opposed to the entire string.)

In addition to the KMP algorithm defined in `pm-kmp.hpp`, other single keyword pattern matchers are defined in header file `bms.hpp`, which contains suggestions for instantiating some of the Boyer-Moore variants. Additionally, a brute-force single keyword pattern matcher is defined in `pm-bfsin.hpp`.

Multiple keyword pattern matching is performed in a similar manner, as the following example shows. The following program searches an input string for the keywords `his`, `her`, and `she`, printing the locations of all matches:

```
#include "com-misc.hpp"
#include "string.hpp"
#include "set.hpp"
#include "acs.hpp"
#include <iostream.h>

static int report( int index, const Set<String>& M ) {
    cout << index << M << '\n';
    return( TRUE );
}

int main( void ) {
    auto Set<String> P( "his" );
    P.add( "her" ); P.add( "she" );
    auto PMACOpt Machine( P );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

Header file `string.hpp` defines a string class, while `set.hpp` defines a template class for sets of objects. Header file `acs.hpp` defines the Aho-Corasick pattern matching classes. Function *report* is our call-back function, simply printing the index of the match (to the standard output) and the set of keywords matching, and returning *TRUE* to continue matching. Note that the call-back function has a different signature for multiple keyword pattern matching: it takes the index of the symbol to the right of the match, and the set of keywords matching with *index* as their right end-point.

The *main* function (the program mainline) creates a local AC machine from the keyword set. The machine is then used to find all matches in string `hishershey`. In the following two sections, we consider ways to use the SPARE Parts more efficiently in certain application domains.

3.1 Multi-threaded pattern matching

One important design feature (as a result of the call-back client interface) of the SPARE Parts is that it supports multi-threading. This can lead to high performance in applications hosted on multi-threading operating systems. For example, consider an implementation of a keyword `grep` application, in which 1000 files are to be searched for occurrences of a given keyword. The following are three potential solutions:

- In a sequential solution, a single pattern matching object is constructed and each of the 1000 files are scanned (in turn) for matches.
- In a naïve multi-threaded solution, 1000 threads are created (each corresponding to one of the input files). Each of the threads construct a pattern matching object, which is then used to search the file.
- An efficient solution is to create a single matching object, with 1000 threads sharing the single object. Each of the threads proceeds to search its file, using its own invocation of member function *PMSingle::match*.

The last (most efficient) solution would not have been possible without the call-back client interface.

3.2 Alternative alphabets

The default structure in the SPARE Parts is to make use of the entire ASCII character set as the alphabet. This can be particularly inefficient and wasteful in cases where only a subset of these letters are used. For example, in genetic sequence searching, only the letters a, c, g, and t are used. The SPARE Parts facilitates the use of smaller alphabets through the use of *normalization*. Header file `alphabet.hpp` defines a constant *ALPHABETSIZE* (which, by default is *CHAR_MAX*). The alphabet which SPARE Parts uses is the range $[0, ALPHABETSIZE)$. An alternative alphabet can be used by redefining *ALPHABETSIZE*, and mapping the alternative alphabet in the required range. The mapping is performed by functions *alphabetNormalize* and *alphabetDenormalize*, both declared in `alphabet.hpp` (by default, these functions are the identity functions). The only requirement is that the functions map 0 to 0 (this is used to identify the end of strings). In the genetic sequence example, we would make use of the following version of header `alphabet.hpp`:

```
#include <assert.h>
#define ALPHABETSIZE 5

inline char alphabetNormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 'a':   return( 1 );
    case 'c':   return( 2 );
    case 'g':   return( 3 );
    case 't':   return( 4 );
    default:    assert( !"Non-genetic character" );
    }
}

inline char alphabetDenormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 1:     return( 'a' );
    case 2:     return( 'c' );
    case 3:     return( 'g' );
    case 4:     return( 't' );
```

```
    default:    assert( !"Non-genetic image" );
    }
}
```

4 Experiences and conclusions

Designing and coding the SPARE Parts lead to a number of interesting experiences in class library design. In particular:

- The SPARE Parts comprises 5787 lines of code in 59 `.hpp`, 32 `.cpp`, 43 `.ppp`, and 49 `.ipp` files.
- Compiling the files, with the WATCOM C++32 Version 9.5b compiler, shows that the size of the object code varies very little for the various types of pattern matchers.
- The taxonomy presented in [Wats95, Chapter 4] was critical to correctly implementing the many complex precomputation algorithms.
- Designing and structuring generic software (reusable software such as class libraries) is much more difficult than designing software for a single application. The general structure of the taxonomy proved to be helpful in guiding the structure of the SPARE Parts.
- In [Wats95, Chapter 4], we consider the relative performance of the algorithms implemented in the SPARE Parts. It is also helpful to consider how the implementations in the SPARE Parts fare against commercially available tools such as the `fgrep` program. Four `fgrep`-type programs were implemented (using the SPARE Parts), corresponding to the Knuth-Morris-Pratt, Aho-Corasick, Boyer-Moore, and Commentz-Walter algorithms. The four tools were benchmarked informally against the `fgrep` implementation which is sold as part of the MKS toolkit for MS-DOS. The resulting times (to process a 984149 byte text file, searching for a single keyword) are:

`fgrep` *variant*	MKS	KMP	BM	AC	CW
Time (sec)	3.9	5.1	4.2	4.7	4.0

  These results indicate that using a general toolkit such as the SPARE Parts will result in performance which is similar to carefully tuned C code (such as MKS `fgrep`). Much more extensive test results are reported in [Wats95, Wats96].

Detailed records were kept on the time required for designing, typing, compiling (and fixing syntax errors), and debugging the toolkit. The time required to implement the toolkit is broken down as follows (an explanation of each of the tasks is given below):

Task	Design	Typing	Compile/Syntax	Debug	Total
Time (hrs:min)	6:00	13:40	10:05	5:15	35:00

Most of these times are quite short compared to what a software engineer could expect to spend on a project of comparable size. The following paragraphs explain exactly what each of the tasks entailed:

- The *design* phase involved the creation of the inheritance hierarchy and the declaration (on paper) of all of the classes in the toolkit. (A C++ declaration provides names and signatures of functions, types, and variables, whereas a definition provides the implementation of these items.) The design phase proceeded exceptionally smoothly, thanks to a number of things:
  - The inheritance hierarchy followed directly from the structure of the taxonomy.

- The decisions on the use of templates (instead of virtual functions) and call-backs were made on the basis of experience gained with the FIRE Engine. These decisions were also somewhat forced by the efficiency requirements on the toolkit, as well as the need for multi-threading.
- Representation issues, such as the selection of data structures, were resolved using experience gained with the earlier Eindhoven Pattern Kit.

- Once the foundation classes were declared and defined, typing the code amounted to a simple translation of guarded commands (appearing in [Wats95]) to C++.
- The times required for compilation and syntax checking were minimized by using a very fast integrated environment (BORLAND C++) for initial development. Only the final few compilations were done using the (slower, but more thoroughly optimizing) WATCOM C++ compiler. The advantages of using a fast development environment on a single user personal computer should not be underestimated.
- Since the C++ code in the toolkit was implemented directly from the abstract algorithms (for which correctness arguments are given), the only (detected) bugs were those involving typing errors (such as the use of the wrong variable, etc.). Correspondingly, little time needed to be spent on debugging the toolkit.

New research in pattern matching requires that tools such as the SPARE Parts evolve. The following are some of the upcoming changes:

- The toolkit will use a version of the C++ Standard Template Library.
- In light of the success and widespread applicability of the commercialized version, SPARE Parts II, the SPARE Parts will be template parameterized to support input strings and patterns over arbitrary alphabets (as opposed to the rudimentary alphabet support now provided).
- Efforts will begin to integrate approximate pattern matching algorithms into the toolkit.

5 Obtaining and compiling the toolkit

The SPARE Parts is available for anonymous ftp from `ftp.win.tue.nl` in directory:

`/pub/techreports/pi/watson.phd/spare/`

The toolkit, and some associated documentation, are combined into a `tar` file. A number of different versions of this file are stored — each having been compressed with a different compression utility.

The SPARE Parts has been successfully compiled with BORLAND C++ Versions 3.1 and 4.0, and WATCOM C++32 Version 9.5b on MS-DOS and MICROSOFT WINDOWS 3.1 platforms. Since the WATCOM compiler is also a cross-compiler, there is every reason to believe that the code will compile for WINDOWS NT or for IBM OS/2. The implementation of the toolkit makes use of only the most basic features of C++, and it should be compilable using any of the template-supporting UNIX based C++ compilers.

A version of the SPARE Parts will remain freely available (though not in the public domain). Contributions to the toolkit, in the form of new algorithms or alternative implementations, are welcome.

References

[AC75] AHO, A.V. and M.J. CORASICK. Efficient string matching: an aid to bibliographic search, *Comm. ACM,* **18**(6) (1975) 333–340.

[BM77] BOYER, R.S. and J.S. MOORE. A fast string searching algorithm, *Comm. ACM,* **20**(10) (1977) 62–72.

[Budd91] BUDD, T.A. *An introduction to object-oriented programming.* (Addison-Wesley, Reading, MA, 1991).

[CE95] CARROLL, M.D. and M.A. ELLIS. *Designing and coding reusable C++.* (Addison-Wesley, Reading, MA, 1995).

[Com79a] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, in: H.A. Maurer, ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer-Verlag, Berlin, 1979) 118–132.

[Com79b] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[GHJV95] GAMMA, E., R. HELM, R. JOHNSON, and J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, Reading, MA, 1995).

[HS91] HUME, S.C. and D. SUNDAY. Fast string searching, *Software—Practice and Experience* **21**(11) (1991) 1221–1248.

[KMP77] KNUTH, D.E., J.H. MORRIS and V.R. PRATT. Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.

[MeyB94] MEYER, B. *Reusable Software: The Base Object-Oriented Component Libraries.* (Prentice Hall, Englewood Cliffs, NJ, 1994).

[MeyS92] MEYERS, S. *Effective C++: 50 specific ways to improve your programs.* (Addison-Wesley, Reading, MA, 1992).

[Murr93] MURRAY, R.B. *C++ strategies and tactics.* (Addison-Wesley, Reading, MA, 1993).

[Plau95] PLAUGER, P.J. *The Draft Standard C++ Library.* (Prentice Hall, New Jersey, 1995).

[SL94] STEPANOV, A. and M. LEE. Standard Template Library, Computer Science Report, Hewlett-Packard Laboratories, 1994.

[Souk94] SOUKUP, J. *Taming C++: Pattern Classes and Persistence for Large Projects.* (Addison-Wesley, Reading, MA, 1994).

[Stro91] STROUSTRUP, B. *The C++ programming language.* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[Stro94] STROUSTRUP, B. *The Design and Evolution of C++.* (Addison-Wesley, Reading, MA, 1994).

[Teal93] TEALE, S. *C++ IOStreams Handbook.* (Addison-Wesley, Reading, MA, 1993).

[Wats94] WATSON, B.W. The performance of single-keyword and multiple-keyword pattern matching algorithms, Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/pattm/`.

[Wats95] WATSON, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995).

[Wats96] WATSON, B.W. The performance of single and multiple keyword pattern matching algorithms, *Workshop on String Processing* (Recife, Brazil, 1996). Available from the author.

[WZ92] WATSON, B.W. and G. ZWAAN. A taxonomy of keyword pattern matching algorithms, Computing Science Report 92/27, Eindhoven University of Technology, The Netherlands, 1992. Available from the author.

[WZ95] WATSON, B.W. and G. ZWAAN. A taxonomy of sublinear keyword pattern matching algorithms, Computing Science Report 95/13, Eindhoven University of Technology, The Netherlands, 1995. Available from the author.

A Abstract pattern matchers

In this section, we briefly consider the two abstract pattern matching classes which are used to define the client interfaces of single and multiple keyword pattern matchers.

User class A.1 (*PMSingle*)

Files: `pm-singl`

Description: Class *PMSingle* defines the call-back client interface outlined in the examples presented earlier. The brute-force, Knuth-Morris-Pratt, and Boyer-Moore classes implement the defined interface.

Implementation: As an abstract class, there is no implementation.

□

User class A.2 (*PMMultiple*)

Files: `pm-multi`

Description: This class defines the call-back client interface outlined in the examples presented earlier. The brute-force, Aho-Corasick, and Commentz-Walter classes implement the defined interface.

Implementation: As an abstract class, there is no implementation.

□

B Concrete pattern matchers

In this section, we describe the classes which implement the interface defined by classes *PMSingle* and *PMMultiple*. The treatment of each of the families of classes includes any auxiliary (non-foundation) classes used. A summary of the classes and their template parameters (if any) is given in Section B.6. We first consider the brute-force pattern matchers, followed by the KMP, AC, CW, and BM pattern matchers.

B.1 The brute-force pattern matchers

The brute-force pattern matchers are the most basic of the classes. While they are easy to understand, they are not intended for use in production quality software; they are used for benchmarking the other pattern matchers.

User class B.1 (*PMBFSingle*, *PMBFMulti*)

Files: `pm-bfsin`, `pm-bfmul`

Description: The brute-force pattern matchers are naïve implementations of pattern matchers. They are only intended to form a baseline, against which the other (more efficient) classes can be measured. As a result, they are not intended for serious use.

Performance: Instead of improving the performance of these classes, the client should make use of one of the other pattern matcher classes.

□

B.2 The KMP pattern matcher

User class B.2 (*PMKMP*)

Files: `pm-kmp`

Description: This pattern matcher implements the Knuth-Morris-Pratt single keyword pattern matching algorithm. It inherits from *PMSingle* and implements the interface defined there.

Implementation: This class maintains the pattern keyword and a *FailIdx* representing the indexing failure function. The implementation of member function *match* corresponds to the body of the Knuth-Morris-Pratt algorithm.

□

Implementation class B.3 (*FailIdx*)

Files: `failidx`

Description: Class *FailIdx* is the indexing failure function for use in *PMKMP*. The constructor takes a keyword. The only interesting member function is one to apply the failure function.

Implementation: The class contains an array (of size $|p| + 1$ for keyword p) of integers, representing the function. The constructor implements the classic KMP precomputation — see [Wats94, Appendix A].

□

B.3 The AC pattern matchers

We now consider the Aho-Corasick family of pattern matchers. As the derivation in [Wats95] shows, all of the Aho-Corasick variants share the same algorithm skeleton. The primary difference is in the mechanism used to compute the transition function on an input symbol. For speed, the AC algorithm skeleton is implemented via a template class as opposed to a base class. A number of variants (instantiations) of the Aho-Corasick objects are declared in the header `acs.hpp`, which is intended for client use.

Implementation class B.4 (*PMAC*)

Files: `pm-ac`

Description: Template class *PMAC* implements the skeleton of the AC algorithms. The template argument must be one of the *ACMachine...* classes (called a *transition machine*) which is used to compute the next transition. The class inherits from *PMMultiple* and implements the corresponding interface. The header file is not intended to be used directly by clients; use `acs.hpp` instead.

Implementation: The implementation contains an *ACMachine...* object. The *PMAC* constructor passes the keyword set to the transition machine constructor.

Performance: The class is already highly tuned. The member functions of the transition machine (the template argument) should be inline for high performance.

□

B.3.1 AC transition machines and auxiliary classes

The transition machines are used in the AC skeleton (template class *PMAC*). The variety of transition machines corresponds to the different methods of computing the next transition discussed in [Wats95].

Implementation class B.5 (*ACMachineOpt*)

Files: `acmopt`

Description: This class provides an implementation of the optimized Aho-Corasick transition function, as described in [Wats95]. Member functions are provided to compute the next state (make a transition) and to compute the output (matched keywords) of a particular state (these member functions are the minimum interface required by template *PMAC*). It implements function the optimized Aho-Corasick transition and output functions.

Implementation: The class contains a *Gamma* and an *ACOutput*. The member functions are pass-throughs to these classes.

□

Implementation class B.6 (*ACMachineFail*)

Files: `acmfail`

Description: Class *ACMachineFail* implements the Aho-Corasick failure function method of computing the next transition, as described in [Wats95].

Implementation: The class contains an *EFTrie* and an *ACOutput*. The computation of the transition function is done with a linear search. The output member function is simply a pass-through to the *ACOutput*.

□

Implementation class B.7 (*ACMachineKMPFail*)

Files: `acmkmpfl`

Description: Class *ACMachineKMPFail* is an implementation of the (multiple keyword) Knuth-Morris-Pratt method of computing the next transition.

Implementation: The class contains an *FTrie* and an *ACOutput*. The transition function is computed by linear search.

Performance: Most inefficiencies are due to the difference between this class and *ACMachineFail*.

□

Implementation class B.8 (*Gamma*)

Files: `acgamma`

Description: Class *Gamma* implements the 'optimized' Aho-Corasick transition function. The constructor takes an *FTrie* and an *FFail*. The main member function computes the image of the function, given a *State* and a character.

Implementation: Class *Gamma* is implemented via a *StateTo*< *SymbolTo*<*State*> >. The constructor performs a breadth-first traversal of the trie, using the failure function to compute function the Aho-Corasick transition function.

Performance: There are methods of computing the transition function directly from the keyword set without the *FTrie*. Computing the trie and the failure function independently is more costly (in both space and time), but provides a modular separation of the functions and keeps the constructor for *Gamma* manageable.

□

Implementation class B.9 (*EFTrie*)

Files: `aceftrie`

Description: Class *EFTrie* implements the extended forward trie. The constructor takes an *FTrie*.

Implementation: This class is implemented with a *StateTo< SymbolTo<State> >*. The constructor simply copies the *FTrie* and extends it.

Performance: The performance could be significantly increased by not copying the trie from scratch. Unfortunately, this is not possible because a single *FTrie* is used to construct the *EFTrie* and the *ACOutput* objects in *ACMachineFail* so the *State*s of the *EFTrie* and the *ACOutput* objects correspond.

□

Implementation class B.10 (*ACOutput*)

Files: `acout`

Description: *ACOutput* implements the Aho-Corasick output function. The constructor takes the set of keywords, the corresponding forward trie (*FTrie*), and the corresponding forward failure function (*FFail*).

Implementation: This class contains a *StateTo< Set<String> >*. The constructor performs a breadth-first traversal of the trie, using the failure function to compute the output function — implementing the algorithm given in [WZ92].

Performance: The high performance of this class depends quite heavily upon the use-counting of class *String* (see User class C.2).

□

B.4 The CW pattern matchers

As outlined in [Wats95], all of the Commentz-Walter variants share a common algorithm skeleton. The difference lies in how the safe shift distance is computed. For this reason, the skeleton is defined as a template class as follows. The variants of the Commentz-Walter algorithm are defined (via **typedef**) in header `cws.hpp`.

User class B.11 (*PMCW*)

Files: `pm-cw`

Description: Class *PMCW* implements the Commentz-Walter skeleton. It inherits from *PMMultiple* and implements the public interface defined there. The template argument must be one of the *CWShift...* classes. The argument provides the safe shift distance during the scanning of the string.

Implementation: A *PMCW* contains a shifter object, an *RTrie* (a reverse trie for scanning the string), and a *CWOutput* (output function for detecting a match). The constructor passes the set of keywords through to the sub-objects.

Performance: Improvements in performance can most easily be gained by improving the shifter objects or the implementation of the reverse trie and output function. The member functions of the safe shift objects should be inline, since they are simple and they are called repeatedly in the inner repetition of *match*.

□

B.4.1 Safe shifters and auxiliary functions

The safe shifter classes form the basis of the Commentz-Walter algorithms. The choice of which one to use in a given application is dominated by a tradeoff between precomputation time and greater shift distances. For applications in which the input string is relatively short, class *CWShiftNLA* has the fastest precomputation but provides the smallest shift distances. (Actually, *CWShift-Naive* provides a shift distance of 1; it is intended only for use in benchmarking the algorithms.) For an application in which the time to scan the string outweighs the time for precomputation, *CWShiftRLA* is the best choice.

Implementation class B.12 (*CWShiftNaive*)

Files: `cwshnaiv`

Description: *CWShiftNaive* implements a naïve safe shift distance of 1 in the Commentz-Walter algorithm. This class is intended for benchmarking use, as opposed to serious applications.

Implementation: The implementation is trivial since no data is stored. The shift distance member function simply returns 1.

□

Implementation class B.13 (*CWShiftNLA*)

Files: `cwshnla`

Description: This class implements the 'no-lookahead' shift distance of.

Implementation: The class contains a single shift function. In the constructor, local *D1* and *D2* shift functions are constructed. These shift functions are combined into a single shift. In the description in [Wats95] the combining of the two would be done as the input string is scanned. They are combined at precomputation time for performance and space reasons.

Performance: The performance would be difficult to improve as the *D1* and *D2* shift functions are already combined at precomputation time.

□

Implementation class B.14 (*CWShiftWBM*)

Files: `cwshwbm`

Description: Class *CWShiftWBM* implements the 'weak Boyer-Moore' shift distance.

Implementation: The implementation is through *D1*, *D2*, and *CharBM*. The amount of shift distance contributed by each of the three functions are combined as the input string is scanned.

Performance: Since the shift distance is combined at string-scanning time, it is important that auxiliary functions such as *min* and *max* are inline functions.

□

Implementation class B.15 (*CWShiftNorm*)

Files: `cwshnorm`

Description: This class implements the 'normal' Commentz-Walter shift function.

Implementation: The implementation is similar to that of *CWShiftWBM*, with the exception that the *CharBM* is replaced by a *CharCW*.

Performance: See the performance clause for *CWShiftWBM*.

□

Implementation class B.16 (*CWShiftOpt*)

Files: `cwshopt`

Description: Class *CWShiftOpt* is an implementation of the 'optimized' Commentz-Walter shift distance.

Implementation: The implementation is similar to that of *CWShiftWBM*, except that the *D1* and *CharBM* are replaced by a *DOpt*.

Performance: See the performance clause for *CWShiftWBM*.

□

Implementation class B.17 (*CWShiftRLA*)

Files: `cwshrla`

Description: This class implements the 'right lookahead' shift function.

Implementation: The implementation is similar to *CWShiftOpt*, with the addition of a *CharRLA*.

Performance: See the performance clause for *CWShiftWBM*.

□

Implementation class B.18 (*CharCW*, *CharBM*, *CharRLA*)

Files: `cwchar`, `cwcharbm`, `cwcharrl`

Description: These three classes are the shift functions which are based upon a single character (the first mismatching character). The definitions of the functions can be found in [Wats95].

Implementation: Classes *CharBM* and *CharRLA* are implemented through a single array, while *CharCW* is implemented through two nested arrays. Their constructors all make breadth-first traversals of the reverse trie. The shift member functions are trivial lookups.

□

Implementation class B.19 (*D1*, *D2*, *DOpt*)

Files: `cwd1`, `cwd2`, `cwdopt`

Description: These three classes implement the Commentz-Walter shift component functions d_1, d_2, and d_{opt}. Classes *D1* and *D2* map a *State* to an integer, while *DOpt* maps a *State* and a character to an integer.

Implementation: The implementations of *D1* and *D2* are in terms of *StateTo*< **int** >, with *DOpt* as *StateTo*< *SymbolTo*< **int** > >. The constructors take an *RTrie* which they traverse, implementing the precomputation algorithm given in [WZ95].

Performance: The derivations of the precomputation algorithms in [WZ95] indicate that the currently implemented algorithms are not likely to be improved.

□

Implementation class B.20 (*CWOutput*)

Files: `cwout`

Description: *CWOutput* implements the Commentz-Walter output function (which determines if a match has been found). The class maps a *State* (from the reverse trie) to a string if the *State* corresponds to a keyword. There is a member function which reports if a given *State* corresponds to a keyword.

Implementation: The implementation uses a *StateTo*<*String**>. If the entry is 0, then the corresponding *State* does not correspond to a keyword. If the entry is not 0, the entry points to the corresponding keyword. The constructor takes an *EFTrie* and the set of keywords.

□

B.5 The BM pattern matchers

Like the Aho-Corasick and the Commentz-Walter algorithms, all variants of the Boyer-Moore algorithm share a common skeleton. Again, the skeleton is implemented as a template class, with the template parameters being used to instantiate the different possible variants. Examples of how to instantiate some of the variants are given in header file `bms.hpp`. The structure of these variants of the BM algorithms are derived fully in [Wats95].

User class B.21 (*PMBM*)

Files: `pm-bm`

Description: This template function implements the Boyer-Moore variants. It inherits from *PMSingle* and implements the public interface defined there. The class takes three template parameters:

- A 'match order' which is used to compare the keyword to the input string. It must be one of the *STrav*... classes.
- A 'skip loop' which is used to skip portions of the input text that cannot possibly contain a match. The argument must be one of the *SL*... classes.
- A 'match information' shift distance class which is used to make larger shifts through the input string, after a match attempt. The argument must be one of the *BMShift*... classes.

For more on each of these three components, see [Wats95].

Implementation: The implementation contains a copy of the keyword and an object of each of the template arguments.

Performance: A fast implementation relies on the skip loop, match order, and shifter member functions all being inline.

□

The Boyer-Moore class *PMBM* takes three template arguments, making it one of the more complex template classes. We now consider an example of an instantiation of the class.

Example B.22 (Instantiating template class *PMBM*): The first of the template arguments is the match order. For this, we select the 'reverse' match order class *STravREV*. As our 'skip loops', we select class *SLFast1*. For our shifter, we select *BMShift11* shifter class. We can now declare a pattern matching object for string `hehshe` as follows:

```
#include "bms.hpp"

static PMBM< STravREV, SLFast1, BMShift11<STravREV> > M( "hehshe" );
```

Note that the same string traverser class must appear as the first template argument to *PMBM* and as the template argument to the shifter class. □

B.5.1 Safe shifters and auxiliary functions

The match orders, the skip loops, and the shifters form the core of the implementation of the BM algorithm variants. The match order classes will be described in Implementation classes C.7. The skip loop classes (class names starting with *SL*...) and the shifters (class names starting with *BMShift*...) are described below, along with some of the shift components.

Implementation class B.23 (*SLNone*, *SLSFC*, *SLFast1*, *SLFast2*)

Files: `bmslnone`, `bmslsfc`, `bmslfst1`, `bmslfst2`

Description: The skip loop classes are used to skip portions of the input string in which no matches are possible. *SLNone* makes no shift through the input string (it is only included for completeness, since it is derived in [Wats95] and in [HS91]).

Implementation: The constructors simply take the pattern keyword. Some of the skip loops store lookup tables to compute the shift distance. In the case of *SLSFC*, the shift distance is 1.

Performance: The current implementations are for maximum performance in time. Space is sacrificed in favour of speed. All of the member functions should be inlined.

□

Implementation class B.24 (*BMShiftNaive*)

Files: `bmshnaiv`

Description: The naïve shift distance class, *BMShiftNaive*, provides a safe shift distance of 1. It is intended only for benchmarking purposes.

□

Implementation class B.25 (*BMShift11*, *BMShift12*)

Files: `bmsh1-1`, `bmsh1-2`

Description: These two classes implement two of the possible shift distances considered in [Wats95]. Both classes are template classes, which expect a string traverser (*STrav*...) class as their template argument. The particular traverser used in the instantiation must be the same traverser class used as the match order in *PMBM*.

Implementation: Class *BMShift11* makes use of an *S1* and a *Char1*, while *BMShift12* makes use of an *S1* and a *Char2*. In both cases, the two shift components are combined during the scanning of the string. The template argument is used to instantiate the correct versions of the component shift functions.

Performance: Since the shift components are combined during the scanning of the input string, the performance could be improved by combining them in the constructor.

□

Implementation class B.26 (*Char1*, *Char2*)

Files: `bmchar1`, `bmchar2`

Description: These two classes are shift components, implementing functions $char_1$ and $char_2$ given in [Wats95]. Since these two functions depend upon the particular match order in use, these two classes are defined as template classes. The template argument must be one of the *STrav...* classes.

Implementation: The definitions of functions $char_1$ and $char_2$ contain **MIN** quantifications. As a result, the constructors perform a linear search to compute the functions. The linear search is general, since it makes use of the template argument (the string traverser).

Performance: When *STravFWD* or *STravREV* are used as the template argument, the linear search can be performed more efficiently (as in the classical Boyer-Moore precomputation). Such traverser specific precomputation can be written as template instantiation overriding functions. (This has not yet been done — it should be done by clients who require higher performance from the SPARE Parts.)

□

Implementation class B.27 (*S1*)

Files: `bms1`

Description: This class implements the s_1 shift component given in [Wats95]. As with shift components $char_1$ and $char_2$, this class depends upon the particular string traverser in use. *S1* is a template class which expects a string traverser as its template argument.

Implementation: See the implementation of classes *Char1* and *Char2*.

Performance: See *Char1* and *Char2*.

□

B.6 Summary of user classes

In this section, we present two tables which summarize the various user classes for pattern matching. The first table summarizes the descendents of abstract single keyword pattern matching class *PMSingle*. All three of the concrete classes are described along with their possible template arguments (if any):

Class	*Description*	
PMBFSingle	Brute-force pattern matcher	
PMKMP	Knuth-Morris-Pratt pattern matcher	
PMBM	Boyer-Moore pattern matcher template Three template arguments required, as follows:	
	Match orders	
	STravFWD	Forward (left to right)
	STravREV	Reverse (right to left)
	STravOM	Optimal mismatch (increasing frequency)
	STravRAN	Random
	Skip loops	
	SLNone	No skip
	SLSFC	Leftmost keyword symbol compared (always 1 symbol shift)
	SLFast1	Rightmost symbol compared
	SLFast2	Rightmost symbol compared (greater shift)
	Shifters	
	BMShiftNaive	Shift of one symbol
	BMShift11	Shift without mismatching symbol information
	BMShift12	Shift with mismatching symbol information

The following table summarizes the concrete class descendents of the abstract multiple keyword pattern matcher class *PMMultiple*. Two of them are template classes and their possible template arguments are summarized as well:

Class	*Description*	
PMBFMulti	Brute-force pattern matcher	
PMAC	Aho-Corasick pattern matcher template Single template argument required:	
	Transition machines	
	ACMachineOpt	Optimal transition function
	ACMachineFail	Failure function with extended trie
	ACMachineKMPFail	Knuth-Morris-Pratt failure function with forward trie
PMCW	Commentz-Walter pattern matcher template Single template argument required:	
	Shifters	
	CWShiftNaive	Shift of one symbol
	CWShiftNLA	No lookahead symbol used
	CWShiftWBM	Weak Boyer-Moore
	CWShiftNorm	Normal Commentz-Walter
	CWShiftOpt	Optimized Commentz-Walter
	CWShiftRLA	Lookahead right one symbol

C Foundation classes

In this section, we consider the design and implementation of the foundation classes and functions. These classes and functions are not of primary concern to the client, but are used to construct classes which form the client interface. Some of these classes are reused in the FIRE Lite — a toolkit of finite automata algorithms also by the author.

A number of these classes will be replaceable by standard library classes once the draft C++ standard becomes stable and implementations of the draft standard start to appear.

C.1 Miscellaneous

A number of header files and their corresponding definitions do not fall into a particular category. Header `com-misc.hpp` contains definitions of constants *TRUE* and *FALSE* and integer maximum and minimum functions *max* and *min*.

Implementation class C.1 (*State*)

Files: `state`

Description: Tries and finite automata require the definition of states. This header contains a definition of states and some constants, in particular an *INVALIDSTATE* and a *FIRSTSTATE*. The *FIRSTSTATE* is used (by convention) as the start state in tries and finite automata.

Implementation: *State* is not defined as a class. Instead, it is **typedef**'d to be an integer for efficiency reasons.

□

User class C.2 (*String*)

Files: `string`

Description: The raw string conventions in C and C++ are too rudimentary to be used effectively and safely. This class provides a higher level (and safer) mechanism for using strings. The interfaces provides members for indexing the individual characters in the string, copying strings, assignment, length of the string, and an output operator (stream insertion).

Implementation: The class is implemented through use-counting with a private class. This makes assignment and copying of strings particularly efficient, but it adds an additional level of indirection to many of the operations. The cost of the extra indirection was found to be negligible compared to the cost of creating complete copies of strings.

Performance: The length of the string is kept in the private class. It requires a complete traversal of the string, using standard function *strlen*. Since this can be particularly inefficient for very large strings, it could be replaced with more efficient methods of determining the length when the string is constructed from a file. The inefficiency of the extra indirection will very likely be removed (when examining individual characters of the string) by a good optimizing compiler.

□

C.2 Arrays, sets, and maps

In this section, we describe the basic template classes used to construct more complex objects.

Implementation class C.3 (*Array*)

Files: `array`

Description: As with strings, the raw C and C++ facilities for arrays are not safe and flexible enough for our purposes. An *Array* template class constructs arrays of objects of class *T*. The operators available in raw arrays are provided. Notable additions are: bounds-checked indexing into the array, a stream insertion operator (assuming that class *T* has an insertion operator), and the ability to resize the array dynamically.

Implementation: An *Array* is implemented by an array of objects, an apparent size (to the client), and real size (for dynamic resizing). The class constant *growthSize* is used during resize operations to allocate some extra elements; these extra elements can be used later to avoid another call to the memory allocator.

Performance: The copy constructor and the assignment operator both make copies of the underlying array. This costly operation could be avoided through the use of use-counting. The class constant *growthSize* is a tuning constant which is 5 by default. Other values may provide higher performance in certain circumstances.

□

Implementation class C.4 (*Set*)

Files: `set`

Description: Sets of objects are used in a variety of places. The template class *Set* implements a set of objects of class *T*. Common set operations (such as element add, remove, union, membership tests, size) are available, as well as a rudimentary iterator facility, and an insertion operator. The class is replaceable by a standard one, once the draft C++ standard is stable. The current draft of the standard proposes to use the Standard Template Library by Stepanov and Lee [SL94]. The Standard Template Library definition puts forth a more complex set of iterators than needed in the SPARE Parts.

Implementation: The implementation is via an *Array*. This makes management of the size of the set particularly simple. Most of the member functions of *Set* are simple pass-throughs to the corresponding *Array* members.

Performance: The performance is most easily improved through modifications in class *Array*.

□

Implementation class C.5 (*StateTo*)

Files: `stateto`

Description: Template class *StateTo* implements a function mapping a *State* to an object of class *T*. Member functions are provided for setting up the function, applying the function, and for adjusting the range of states in the domain.

Implementation: *StateTo* is implemented in terms of an *Array*. The member functions are mostly pass-throughs to the *Array* members.

Performance: Since the implementation is through class *Array*, the performance of *StateTo* is most easily improved by improving *Array*.

□

Implementation class C.6 (*SymbolTo*)

Files: `symbolto`

Description: Template class *SymbolTo* implements a function from characters to an object of class *T*. The characters in the domain are assumed to be in the range $[0, ALPHABETSIZE)$. Member functions are provided for setting up the function and for applying it.

Implementation: Class *SymbolTo* is implemented in terms of an *Array*. The member functions are mostly pass-throughs to the *Array* members.

Performance: The performance is most easily improved through improvements to class *Array*.

□

C.3 Tries and failure functions

Tries and failure functions form the basis for the multiple keyword pattern matching algorithms. Tries and match orders (for use in the Boyer-Moore algorithms) are both implemented in terms of string traversers. String traversers, tries, and failure functions are described in this section.

Implementation class C.7 (*STravFWD*, *STravREV*, *STravOM*, *STravRAN*)

Files: `stravfwd`, `stravrev`, `stravom`, `stravran`

Description: *String traversers* are synonymous with match orders. For a given string of length n, a string traverser is a bijection on $[0, n)$. It can be used to traverse the characters of the string in a particular order. Class *STravFWD* corresponds to the identity function, which traverses the string in the left to right direction. Class *STravREV* allows one to traverse the string in the right to left direction. Classes *STravOM* and *STravRAN* correspond (respectively) to an optimal mismatch order and a random order. The latter two classes are used primarily in the Boyer-Moore algorithms, while the first two find use in class *Trie*. All of the traversers have constructors which take a keyword.

Implementation: Since class *STravFWD* implements the identity function, it does not contain any private data. Class *STravREV* only contains the length of the keyword in order to implement the bijection. The other traversers have not yet been implemented.

Performance: Since these classes are used to consider the characters in strings, it is important that they are inlined.

□

Implementation class C.8 (*Trie*)

Files: `trie`, `tries`

Description: Template class *Trie* implements tries. The template parameter must be one of the string traverser classes. *Trie* has a constructor which takes a set of strings, yielding the corresponding trie. The string traverser determines in which order the strings are traversed in the constructor. Using *STravFWD* as the template parameter gives a forward trie, while using *STravREV* gives a reverse trie. Forward and reverse tries are **typedef**'d in header `tries.hpp`; this header is intended for use by clients.

There are a special set of member functions to perform breadth-first traversals of the trie, and to determine the depth (when the trie is considered as a tree) of a particular *State* (since each *State* corresponds to a string, the depth of the *State* is equal to the length of the string).

Implementation: *Trie* is implemented using a *StateTo*< *SymbolTo*<*State*> >. An additional *StateTo* maps *State*s to integers, keeping track of the depth of the *State*s for the breadth-first traversals. Almost all of the member functions are simple pass-throughs. The constructor uses the depth-first method of constructing the trie from the set of strings.

Performance: The use of nested mappings (*StateTo*s and *SymbolTo*s) is highly efficient in time, but it is relatively space inefficient. An implementation which uses trees would save considerable space at the expense of access time. The disadvantage to such an implementation is that each transition requires more time.

□

Implementation class C.9 (*Fail*)

Files: `fail`, `fails`

Description: Failure functions are implemented using template class *Fail*. *Fail* has a constructor which takes a *Trie* as parameter. The type of trie (forward or reverse) determines the type of failure function constructed (forward or reverse). Consequently, the template argument to *Fail* must be either *STravFWD* or *STravREV*. The template argument is only used to determine the type of the *Trie* taken as parameter by the constructor. Header `fails.hpp` contains **typedef**s of the forward and reverse tries; this header file is intended for use by clients.

Implementation: A failure function is implemented as a *StateTo*<*State*>. The constructor uses a standard breadth-first traversal of the *Trie* (see [WZ92] for algorithms constructing failure functions from tries).

Performance: The performance can only be improved through improvements to class *StateTo*.

□