NAME

dfa, dfa_new, dfa_ignore, dfa_halton, dfa_restore, dfa_begin, dfa_add, dfa_close, dfa_perror, dfa_errstr, dfa_nextstate, dfa_contains, dfa_accepts, dfa_extent, dfa_dump, dfa_scanf, dfa_sscanf, dfa_fscanf, dfa_pscanf, dfa_malloc, dfa_memalloc, dfa_memincr, dfa_memfree, dfa_destruct – deterministic finite-state automaton subroutines

SYNOPSIS

```
#include <dfa.h>

#define DFA_MUNGEBIT  1
#define DFA_ISACCEPTING(stateptr)  DFA_ISMUNGED(stateptr)
#define DFA_ISMUNGED(P) ((int)(P) & DFA_MUNGEBIT)
#define DFA_MUNGED(P)   ((unsigned long)(P) | DFA_MUNGEBIT)
#define DFA_UNMUNGED(P) ((unsigned int)(P) & ~DFA_MUNGEBIT)

typedef char DFA, PNTR DFAPtr;

typedef struct DFA_State {
     struct DFA_State PNTR next[1];
  } DFA_State, PNTR DFA_StatePtr;

typedef union { /* Patterns may need to be identified by a variety of means */
     void *p;
     long i;
     unsigned long u;
  } DFA_PatID, *DFA_PatIDPtr;

typedef struct DFA_Patlist { /* linked list of patIDs */
     /* the "next" field in last item is NULL */
     struct DFA_Patlist PNTR next;
     DFA_PatID patID;
  } DFA_Patlist, *DFA_PatlistPtr;

typedef struct DFA_Accept {
     DFA_StatePtr retState; /* retState is Munged iff one patID is in the list */
     DFA_PatlistPtr listptr; /* pointer to linked list of patIDs */
  } DFA_Accept, PNTR DFA_AcceptPtr;


DFAPtr dfa_new(amin, amax)
   int amin, amax;

DFA_Error dfa_ignore(dp, a)
   DFAPtr dp;
   int a;

DFA_Error dfa_halton(dp, a)
   DFAPtr dp;
   int a;

DFA_Error dfa_restore(dp, a)
   DFAPtr dp;
   int a;

DFA_Error dfa_begin(dp)
   DFAPtr dp;

DFA_PatlistPtr dfa_add(dp, pattern, patlen, patID)
   DFAPtr dp;
   unsigned char *pattern;
   size_t patlen;
   DFA_PatID patID;

DFA_Error dfa_close(dp)
   DFAPtr dp;

DFA_StatePtr dfa_nextstate(S, ch, pos, report)
   DFA_StatePtr S;
   int ch;
   size_t pos;
   int (*report)(size_t pos, void *patID);

DFA_Error dfa_destruct(dp)
   DFAPtr dp;

DFA_Error dfa_check(dp)
   DFAPtr dp;

DFA_PatlistPtr dfa_contains(dp, pattern, patlen, patID)
   DFAPtr dp;
   const unsigned char *pattern;
   size_t patlen;
   void *patID;

DFA_PatlistPtr dfa_accepts(dp, pattern, patlen)
   DFAPtr dp;
   const unsigned char *pattern;
   size_t patlen;

unsigned long dfa_extent(dp)
   DFAPtr dp;

unsigned long dfa_size(dp)
   DFAPtr dp;


DFA_Error dfa_perror(s)
   char *s;

char *dfa_errstr(errno)
   DFA_Error errno;

DFA_Error dfa_dump(dp, fp)
   DFAPtr dp;
   FILE *fp;


void *dfa_malloc(dp, nbytes)
   DFAPtr dp;
   size_t nbytes;

void * (*dfa_memalloc(falloc)) (size_t)
   void *(*falloc)(size_t);

void (*dfa_memfree(ffree)) (void *)
   void (*ffree)(void *);

size_t dfa_memincr(newincr)
   size_t newincr;


DFA_Error dfa_scanf(dp, report)
   DFAPtr dp;
   int (*report)(size_t pos, DFA_AcceptPtr ap);
```

DFA_Error dfa_sscanf(dp, query, querylen, report)
DFAPtr dp;
const DFA_Letter ∗query;
size_t querylen;
int (∗report)(size_t pos, DFA_AcceptPtr ap);

DFA_Error dfa_fscanf(dp, fp, report)
DFAPtr dp;
int (∗report)(size_t pos, DFA_AcceptPtr ap);
FILE ∗fp;

DFA_Error dfa_pscanf(dp, gettoken, stream, report)
DFAPtr dp;
int (∗gettoken)(void ∗stream), (∗report)(size_t pos, DFA_AcceptPtr ap);
void ∗stream;

DESCRIPTION

These functions create a deterministic finite-state automaton (DFA) that accepts or recognizes variable length, metacharacter-free expressions in an input stream. A Mealy machine implementation is employed, wherein output activity is linked to state transitions (*accepting* transitions), rather than in association with the states themselves (*accepting* states) as in a Moore machine. This can reduce enormously the number of states required in a multi-pattern automaton and marginally increases its search speed over using a lookup table. Compared to a sparsely populated lookup table, an equivalent Mealy machine is compact and may easily reside in some central processor data caches. In cases where an equivalent lookup table would be fully populated, the DFA implementation produces an automaton that is only marginally larger than the lookup table, by the storage equivalent of a single state. As pattern lengths increase, lookup tables quickly (exponentially) become too large to be practical, whereas a DFA as implemented here will handle such situations with ease as long as the pattern space is not significantly explored. A hash table method could be used instead to conserve storage, but at the expense of slower search speeds.

An initial DFA structure is created by calling **dfa_new**. The input alphabet is defined at this point as the set of all integers from *amin* through *amax.* Prior to calling **dfa_begin** to create the initial state of the automaton, optional calls to **dfa_ignore** and **dfa_halton** may be made, to specify individual letter values that the automaton is to either ignore or halt on, respectively, when encountered in the input stream. **dfa_restore** may be called to restore default behavior for any letter specified in a previous call to **dfa_ignore** or **dfa_halton**.

A single call to **dfa_begin** creates the initial state of the automaton. Only after this has been done can consecutive calls be made to **dfa_add** to augment the skeletal automaton to recognize each *pattern* of length *patlen* bytes. On success, **dfa_add** returns a pointer to the DFA_Patlist structure for each newly added pattern/patID combination; on failure, NULL is returned and a description of the error is saved in **dfaerrno.** A visible string description of the error can be displayed to stderr by calling **dfa_perror.**

The structure of a DFA is completed by calling **dfa_close**. This function does not create any additional states, but it does allocate a temporary wrap-around buffer of DFA_State pointers and demands permanent heap storage for additional accepting transitions that may be required.

A DFA is built within longword-aligned storage obtained either by calling the default memory allocator malloc() or by calling the memory allocation function *falloc* specified to **dfa_memalloc**. *falloc* should return a NULL pointer if no further storage is available or if the DFA is otherwise not permitted to grow any further. *falloc* will be called with a request for a longword-aligned block of size *memincr* bytes, which may be altered from its default value by calling **dfa_memincr**. Typically, *memincr* has a value of several thousand bytes, or enough storage for at least a few states and their associated accepting transitions. Each block of this storage is then doled out in smaller chunks as necessary.

All heap storage acquired for a DFA through calls to the memory allocator can be released by a single call to **dfa_destruct**. For each successful call to the memory allocator, **dfa_destruct** calls either the default memory free function **free(3)**, or the function specified as the *ffree* argument to **dfa_memfree**, with the address of a block of storage that had been returned by the memory allocator.

After a DFA has been completed by calling **dfa_close**, use of the DFA to search an input stream requires persistent testing of a ‘‘munge’’ flag encoded in the lowest-order bit of **DFA_State** pointers. A munge flag that is set reveals the encounter of an accepting transition. The **DFA_ISACCEPTING**, **DFA_ISMUNGED**, **DFA_MUNGED**, and **DFA_UNMUNGED** macros are used to test, set, and clear the **munge** flag. When processing the input stream, a munged **DFA_State** pointer indicates that the *unmunged* pointer actually references an accepting transition structure of type **DFA_Accept,** not the next state; the appropriate next-state pointer must be retrieved from the **retState** field of the **DFA_Accept** structure.

The following sample code searches standard input using a previously created DFA, *dp*.

```
/* Sample code to search stdin using a DFA */

int report();
DFA_StatePtr S; /* Current state pointer */
DFA_AcceptPtr A; /* for interpreting accepting transitions */
DFA_PatlistPtr P; /* for processing lists of patIDs */
int c; /* holds each input character */


/* Initialize the automaton by placing it in State 0 */
S = dp->state0;

for (;;) {
    do {  /* PRIMARY INPUT PROCESSING LOOP */
        /* Get next token from the input stream */
        c = getchar();
     /* Transit to the next state, checking for acceptance */
    } while ( !DFA_ISACCEPTING(S = S->next[c]) );

    /* Check for end of file condition */
    if (c == EOF) return;

    /* Process the accepting transition */
    A = (DFA_AcceptPtr)DFA_UNMUNGED(S);

    /* Obtain the return state */
    S = A->retState;

    /* Get the head of the linked list of PatIDs */
    P = &A->pl;

    /* Report each pattern accepted in this transition */
    do
        if (report(P->patID) != 0) break;
    while (P = P->chain); /* Last item in list has NULL chain pointer */
}
```

Further examples of how to use DFAs may be found in the source code for **dfa_fscanf**.

Each DFA structure begins with a header that contains information such as the alphabet in use, the initial state, memory management details, etc. Some of the useful elements within the DFA structure are:

opstate	the operational state of the automaton.
state0	the address of the initial state.

statesize	the size (in bytes) of each state in the automaton.
nstates	the number of states in the automaton.
asize	the number of letters in the alphabet.
amin	the smallest letter value present in the alphabet.
amax	the largest letter value present in the alphabet.

Other characteristics of a DFA are not readily available as specific elements in the structure. For a few of these cases, special purpose functions are provided. **dfa_extent** returns the total amount of storage (in bytes) allocated to the DFA, while **dfa_size** returns the amount of that storage (in bytes) which is actually in use.

dfa_dump produces a brief feature summary of the DFA and a full, printable dump of the binary contents of the states, accepting transitions, and *patID* lists on the output stream *fp*.

The enumerated operational states maintained in the **opstate** field of a DFA structure are:

dfaOpstateInit	basic initialization completed.
dfaOpstateMemInit	memory management established.
dfaOpstateAlphaInit	alphabet has been validated and processed.
dfaOpstateVirgin	the DFA has not been fully initialized by **dfa_begin** and contains no states.
dfaOpstateOpen	the DFA is ready for adding a pattern.
dfaOpstateBuilding	a pattern is being added to the DFA.
dfaOpstateClosing	the DFA is in the process of being closed.
dfaOpstateClosed	the DFA is closed.
dfaOpstateZombie	the DFA is garbage, but somehow still lives.
dfaOpstateDestroyed	the DFA has been destroyed/deallocated and should not be referenced.

dfa_pscanf is a sample input parser using a fully constructed DFA pointed to by *dp*. *gettoken* is called repeatedly by **dfa_pscanf** with the specified *stream* argument (which is not checked) to obtain each character from the input stream. When an accepting transition is encountered, the user-supplied function *report* is called with the current location in the input stream (*pos*) and the *patID* of a recognized pattern. *report* should return 0 on success, or return non-zero on failure upon which **dfa_pscanf** will abort its search of the input stream. Upon obtaining the return value EOF (*i.e.*, signed integer value -1) from a call to *gettoken*, **dfa_pscanf** returns normally to its caller.

RETURN VALUES

Functions that return a size indicate an error condition with zero return values; a non-zero return value indicates no error. Functions that return an int Functions that return a pointer indicate errors by returning NULL **(0)**. Upon receipt of an error return value, **dfa_error** may be called to determine the precise error encountered. **dfa_clearerr** may be called to reset an error condition.

ERRORS

The functions will fail under the following enumerated error conditions:

dfaErrBadPtr	*dp* is not a valid DFA pointer.
dfaErrBadParm	a bad value was specified for one or more parameters.
dfaErroMemincrTooSml	*memincr* is too small; see **dfa_memincr.**
dfaErrNoMem	insufficient memory available to complete request.
dfaErrPatlen	*patlen* is zero.
dfaErrOpstate	*dp* is in the wrong operational state for the called function.

dfaErrNonAlpha	*pattern* contains a letter not present in the *alphabet.*
dfaErrNullParm	a required parameter is NULL.
dfaErrAlphaSize	invalid alphabet size.
dfaErrFileIO	stream I/O error encountered by **dfa_dump.**
dfaErrReport	the report function called by **dfa_scanf, dfa_sscanf, dfa_fscanf,** or **dfa_pscanf** returned non-zero.
dfaErrUnknown	non-specific error.

RESOURCE UTILIZATION

Efficiency of DFA construction can be broken down into the separate requirements for states, accepting transitions, linked lists of *patID*s associated with the accepting transitions, and working storage. Time and storage requirements for the states are no worse than linear functions of the total length of the patterns. Each state contains one next-state pointer for every letter in the alphabet, so the size of a DFA is linear in the alphabet size. The Mealy machine implementation imposes an upper bound on the number of states that is the summation of $S**m$, for all $m = 0, 1, 2,... n-1$, where S is the number of letters in the alphabet and n is the length of the longest pattern accepted by the DFA. This worst-case situation is encountered when all possible patterns of length n are to be recognized by the DFA, corresponding to situations where the equivalent lookup table has all slots filled. When S is a power of two, the number of states is bounded by $S**n - 1$.

Since each state contains S next-state pointers, the maximum number of next-state pointers in a DFA is the product of S and the upper bound on the number of states, or the summation of $S**m$, for all $m = 1, 2, 3,... n$. When S is a power of two, the total number of next-state pointers is limited to $S**(n+1) - S$. Compared to a lookup table for any size alphabet (fully populated or not), the lookup table requires storage for $S**n$ entries. Essentially the leaf states of a worst-case DFA are equivalent in total size to the lookup table; the root state plus any internal states comprise the additional storage required by the DFA in worst case situations. For arbitrarily sized alphabets, the number of pointers required by a DFA over that required for a lookup table is the summation of $S**m$, for all $m = 1, 2, 3,... n-1$. If the alphabet is a power of two in size, this simplifies to $S**n - 2$. Thus, in worst case situations, the DFA is not quite twice the size of an equivalent, fully populated lookup table. For short pattern lengths, this makes little practical difference. For long pattern lengths, many applications exist which do not require that the entire pattern space be explored. Furthermore, for speed in searching (using logical shift operations rather than multiplies), lookup tables are often chosen to be the largest power of two in size necessary to support the desired alphabet. When such a rule is used to pick the size of a lookup table, depending on how much larger than a power of two the alphabet is, the equivalent worst-case DFA may approach half the size of the lookup table because there is no performance gained by using states that are larger than necessary.

Requirements for the accepting transitions are linear when all patterns are equal length, and more complicated when some patterns are internal substrings of others. Accepting transitions individually occupy little storage (4 bytes overhead on common platforms), but their cumulative storage can amount to far more than the storage required for the states themselves.

The output lists associated with accepting transitions also have small but additive time and storage dependencies (8 bytes overhead per element in a linked list). In many instances, resources are conserved by joining existing output lists to produce longer linked lists that have both initial and internal entry points; in other cases, lists of *patID*s may be re-used without duplication.

BUGS

This manual page has not been rigorously examined for accuracy and is expected to contain numerous errors, some substantive.

Hiding the **munge** flag in pointers is not the best machine-independent programming practice, but it does conserve significant time and storage when it works. On most platforms encountered, the least-significant bit (bit 0) is used for the **munge** bit. On the Cray YMP, however, a different bit must be used, for which bit 30 has been selected.

In making sweeping changes to the DFA library in February, 1992, **dfa_close** was left in an incomplete state--it allocates a temporary storage space (WABuf) that is usually larger than necessary but still small enough for most practical applications even on modest-sized microcomputers. When time permits, this function will be brought back into shape so that the temporary storage is smaller and wraps around on itself as it did in earlier versions.

The memory allocation and free functions employed, as well as the size of each request made for additional storage, apply globally to all DFA structures currently in use. If necessary, it would be trivial to have these characteristics set at the time a DFA is created by **dfa_new** and retained throughout the life of the DFA, independent of any subsequent calls to **dfa_memalloc**, **dfa_memfree** and **dfa_memincr**.

Note that the *pattern* argument to **dfa_add** is a pointer to an array of unsigned char, while the *amin* and *amax* arguments to **dfa_new** are signed integers. A slightly modified version of **dfa_add** needs to be written that accepts pointers to arrays of signed char or int.

Automata created by this library are not guaranteed to contain the smallest number of states needed to recognize a given list of patterns.

Fragmentation of the free pool by a temporary wrap-around buffer may occur if the need for more accepting transitions by **dfa_close** should result in one or more calls to *falloc.*

When insufficient storage is available to create a DFA, **dfa_begin**, **dfa_add** and **dfa_close** will fail. These functions do not back out any intermediate modifications made to the DFA. Even if sufficient storage becomes available at a later time, a DFA that was specified in a failed call to **dfa_begin**, **dfa_add** or **dfa_close** should never be used again in calls to these functions. **dfa_destruct** can and should be called to return any allocated storage to the free pool.

dfa_ignore, **dfa_halton** and **dfa_restore** may not be applied to a DFA once it has been the subject of a call to **dfa_begin**.

dfa_add may not be applied to a DFA once it has been the subject of a call to **dfa_close**.

No means are currently provided to save a DFA to disk in a re-usable format or to share a DFA image between unrelated processes.

The information produced by **dfa_dump** may be voluminous and could be easier to interpret.

No attempt is made by **dfa_add** to filter duplicate calls using the same combination of *patID* or *pattern* arguments. If recognition of identical patterns by an automaton is to be avoided, the functions **dfa_accepts** and **dfa_contains** may be useful for screening the calls made to **dfa_add**.

In linked lists of *patID*s, identical patterns are listed in the reverse order from which they were made the subject of calls to **dfa_add** or **dfa_addx**. Any pattern which is a suffix of a longer pattern is listed after the longer pattern.

SEE ALSO

Aho, Hopcroft and Ullman (1974). *The Design and Analysis of Computer Algorithms,* page 335. Addison-Wesley, Reading, MA.

Aho and Corasick (1975). *Efficient String Matching: An Aid to Bibliographic Search*, Communications of the Association for Computing Machinery **18**(6):333–340.

Hopcroft and Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*, pages 42-45. Addison-Wesley, Reading, MA.

EPILOGUE

The methods used in this function library were developed independently by the author during the early 1980s, while a student in the laboratory of Michael Botchan at the University of California, Berkeley. The C language implementation described here was initiated in 1989, while the author was a staff fellow at the NCBI. It was expanded somewhat at Washington University in 1995. Subsequently, a description of many of the ideas implemented here was found in Aho and Corasick (1975). In particular, Aho-Corasick's Algorithm 2 is analogous to **dfa_add**; and **dfa_close** is analogous to a consolidation of Aho-Corasick's Algorithms 3 and 4, just as the authors suggest would be done in practice. Notable differences include the use of a Mealy machine here and a Moore machine by Aho-Corasick; more efficient storage of output lists here when some patterns are suffixes of others; and a more efficient, hierarchical ordering here of tests for the end of the text string being searched and the acceptance of some pattern(s).

ACKNOWLEDGEMENTS

The author wishes to thank former upperclassman Michael J. Karels for the keen suggestion which led to this development: that automata be used to identify restriction enzyme recognition sites in a single pass through a sequence; Glen Herrmannsfeldt for pointing me to Aho and Corasick (1975); and former classmate J. Michael Cherry for his encouragement.

AUTHOR

Warren Gish, Washington University School of Medicine, Department of Genetics, 4444 Forest Park Avenue, Saint Louis, MO 63108. gish@watson.wustl.edu. Former address: National Center for Biotechnology Information, National Library of Medicine, Building 38A Room 8N-806, 8600 Rockville Pike, Bethesda, MD 20894. gish@ncbi.nlm.nih.gov.