String Matching a Zillion Times

We propose to solve the following problem: Given a set of S1, S2, .. Sn words from a language L and matching acceptance functions F1, F2, .. Fn decide whether an bespoke input string S in belongs to the above set and return the associated acceptance function. Return NULL If S does not belong to the language L. The size of the problem is relatively modest, n in 100..1000, the length of the input words are all <32 symbols. The alphabet contains <50 symbols. This functionality is needed to pick a batch scheduler and will be run millions of times.

# A naive approach

A first naive approach would be to hardcode a linear search and compare strategy scanning over all elements of the language. In C++ this would look like:

extern char\* S1, S2, ..Sn;

extern FuncPtr P1, P2, ..Pn;

FuncPtr accept(char\* s)

{

if (strcmpi(s, S1) == 0) return P1;

if (strcmpi(s, S2) == 0) return P2;

..

if (strcmpi(s, Sn) == 0) return Pn;

return NULL; // „s” is not part of the language

}

If we know the relative frequency of each word, the “if” statements can be ordered such as the most frequently occurring inputs are tested first.

What happens if we save on the code and store the strings and function pointers in arrays as in the next exhibit? The answer is platform specific, and would depend on how your compiler and runtime allocates the strings in memory. Suffice to say, there is an additional indirection for each comparison through the indexed double pointer that is expected to be somewhat slower. Our intuition is confirmed by the execution times in table XXX.

extern char\*\* S;

extern FuncPtr \*P;

extern int n;

FuncPtr accept(char\* s)

{

for (int k = 1; k <= n; ++k) if (strcmpi(s, S[k]) == 0) return P[k];

return NULL; // „s” is not part of the language

}

# Hashing

Running a profiler over our solution can easily identify the performance bottleneck of our solution: a huge amount of time is spent comparing strings. Since string comparisons are slower than integer comparisons, next we propose to rewrite our algorithm making use of a hash function and integer comparisons. Assume we have an ideal hash function that takes an input string and returns an integer value corresponding to the string (see <https://en.wikipedia.org/wiki/Hash_function>).

* Next we use a little bespoke string hashing function I have found on the web a while ago. I’d gladly give credit to its author but cannot find it anymore. The source code is in the next exposit. The reader may find more useful information about hash functions in my article [1].
* We could have used the more powerful CRC hash function from the [CRC calculator](https://sourceforge.net/projects/crccalculator/) project (<https://sourceforge.net/projects/crccalculator/files/CRC/>), but our little project does not require that much effort.

unsigned int hash(const char\* s, const int len)

{// computes hash value for <s>. len = strlen(s) must be provided externally.

unsigned int hashVal = 0;

for (int x = 0; x < len; ++x)

{

hashVal ^= (hashVal << 5) + (hashVal >> 2) + s[x];

}

return hashVal;

}

We suggest testing the hash function on the language to make sure it does not produce collisions for the input language in a pre-processing step. In the unlikely event of collision the hash function or the input data needs to be modified (e.g. applying a deterministic bijection on the kth symbol of all words and the input string would lead o a completely different set of hash values of the without collisions).

extern int hash(char\*);

int H[n + 1]; // hash codes for the words of the language S1..Sn

void init()

{

for (int k = 1; k <= n; ++k) H[k] = hash(S[k]);

}

FuncPtr accept(char\* s)

{

int h = hash(s);

for (int k = 1; k <= n; ++k) if (h == hash[k]) return P[k];

return NULL; // „s” is not part of the language

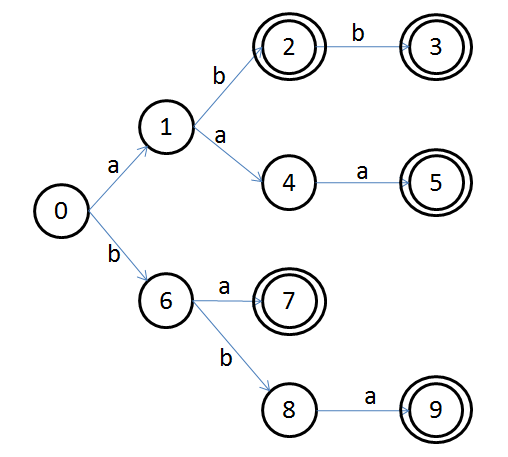
}

The values of the hash function are arbitrary integer values with huge gaps. This makes it very difficult to call directly the acceptance function corresponding to a particular value. Imagine if we could build a hash function that returns consecutive integers between 1 and n for S1 to Sn, we could essentially replace the linear search with *return P[hash(s)]*. This is called increasing minimal perfect hash function. It is very difficult to construct and may not be computationally very efficient for our problem.

# Automation

To further optimize our code without resorting to assembly language we now turn our attention to Finite State Automata (FSA), as in <https://en.wikipedia.org/wiki/Automata-based_programming>. This is a very powerful concept that usually provides optimal execution times.

We intend to construct a deterministic FSA with a distinct terminal state for each word of the language. For example, the following automate represents the strings: *abb, aaa, ba, bba, ab with acceptance functions P1 ..P5*.



The nodes are called states, and states 2, 3, 5, 7 and 9 are accepting the five words of our simple language. FSA are uniquely identified via their transition matrix. For small automata this is a most commonly represented as a {nbStates x nbSymbols} transition matrix, where T[state][symbol] contains the state the automata would move from state “state” if it is presented the input symbol ”symbol”. The transition matrix corresponding to our 5 word FSA is given below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **a** | **1** | **4** | **-1** | **-1** | **5** | **-1** | **7** | **-1** | **9** | **-1** |
| **b** | **6** | **2** | **3** | **-1** | **-1** | **-1** | **8** | **-1** | **-1** | **-1** |

Once the FSA is in state 2, 3, 5, 7 or 9 and there are no more input symbols the input string is accepted as part of the language and the corresponding pointer P1, P2, P3, P4 or P5 is returned. Please refer to the next exhibit for implementation details. As the transition matrix alone cannot tell whether a state is accepting or not, we need to maintain a state vector “states” indicating whether this state is a terminal state accepting a word of the language and holding the corresponding pointer.

Using an FSA to match the input signal is a major performance enhancement that should net us at least a one factor of magnitude speedup. In order to illustrate that tweaking the data representation is important we have encoded the transition matrix as 1) vector<vector> and 2) a row major user managed memory block, 3) a column major user managed memory block. The next exhibit shows the vector<vector> implementation.

The automaton for our language has more than three thousand states and almost forty symbols. It turns out the choice of the data structure is important. The fastest execution times were for the column major implementation, indicating that the locality of memory access is an important factor of the execution.

struct stateT

{

bool terminalState;

FuncPtr funcPtr;

} states[nbStates]; // representation of the automaton states

FuncPtr accept (char\* str)

{

int pos = 0;

int state = 0;

char c;

while (str[pos])

{

c = str[pos];

if (transitions[state][c] == 0) return NULL; // not matched

// follow transition for character <c>

state = transitions[state][c];

++pos;

}

//FSA has recognized “str” if it ended in a terminal state

if (states[state].terminalState)

{

return states[state].funcPtr;

}

return NULL;

}

If the automaton is bigger or the target architecture has a very small L1 cache the execution may benefit from using a transition list representation for the transition matrix. This is more advanced and requires rigorous planning.

* The automata is represented fully as a one dimensional array fsa[].
* All states (with transitions) are stored sequentially, starting with state 0 and followed by the rest of the states.
* For each state starting at fsa[k] the following data is stored
  + Number of outgoing transitions (zero for terminal node. Negative number if it is a non terminal accepting node.) : nbTrans
  + nbTrans numbers representing the transition symbols
  + pointer to acceptance function if node is terminal or accepting. This data is absent from non accepting nodes.
  + List of positions in fsa where the representation of the data for the next state is stored. This follows the same order as the transition symbols.

For optimal locality and saving one indirection we have renounced using the states vector and encoded the whole automaton with transitions, terminal states and function pointer in a single vector. The automaton from the previous example will be represented by the following table.

|  |  |  |
| --- | --- | --- |
| **Position "k"** | **fsa[k]** | **Explanation** |
| [0] | 2 | State 0: first element "2" indicates that there are 2 transitions out of this state. Accepted transitions are "a" and "b", with respective destinations in positions 5 (state 1) and 23 (state 6). |
|  | a |
|  | b |
|  | 5 |
|  | 21 |
| [5] | 2 | State 1: This is not an accepting state. There are two transitions out from this state, "a" leading to position 16 (state 4) and "b" to position 10 (state 2). |
|  | a |
|  | b |
|  | 16 |
|  | 10 |
| [10] | -1 | State 2: A negative number of outgoing transitions indicates that this state would accept pattern "ab" and return pointer P5. There is one outgoing transition for "b" towards position 10 (state 3) to match a longer string. |
|  | b |
|  | P5 |
|  | 14 |
| [14] | 0 | State 3: Is a terminal page with no outgoing transitions. It matches "abb" and contains the pointer P1. |
|  | P1 |
| [16] | 1 | State 4: One outgoing transition towards position 19(state 5) with symbol "a". |
|  | a |
|  | 19 |
| [19] | 0 | State 5: Terminal state for "aaa". |
|  | P2 |
| [21] | 2 | State 6: See description for state 1. |
|  | a |
|  | b |
|  | 26 |
|  | 28 |
| [26] | 0 | State 7: Terminal state for "ba". |
|  | P3 |
| [28] | 1 | State 8: See description for state 4. |
|  | a |
|  | 31 |
| [31] | 0 | State 9: Terminal state for "bba". |
|  | P4 |

## Path compression

If there is a guarantee that the input string S is part of the language we do not need to examine all its symbols. Once the FSA has found a suitable partial match, i.e. with S *L*, once the FSA has seen *a* and *b* it can return the acceptance function P1 for *abb* without looking at the last letter in *S*. If we would like the automata to accept partial matches, e.g. *aa* or *bb* as *abb, aaa* and *bba* we need to store more information in states 4 and 8 such that they should return the pointers for P1, P2 and P4. It is expected that the support for partial matches would lead to faster program execution.

The compressed automaton is shown in the next table.

|  |  |  |  |
| --- | --- | --- | --- |
| position | fsa[k] | .. |  |
| 0 | 2 | [16] | **-1** |
|  | a |  | a |
|  | b |  | P2 |
|  | 5 |  | 20 |
|  | 22 | [20] | 0 |
| [5] | 2 |  | P2 |
|  | a | [22] | 2 |
|  | b |  | a |
|  | 16 |  | b |
|  | 10 |  | 27 |
| [10] | -1 |  | 29 |
|  | b | [27] | 0 |
|  | P5 |  | P3 |
|  | 14 | [29] | **-1** |
| [14] | 0 |  | a |
|  | P1 |  | P4 |
| .. |  |  | 33 |
|  |  | [33] | 0 |
|  |  |  | P4 |

By now the reader should be able to understand the meaning of the encoding without additional commentary. Note the addition of an accepting function pointer to states 4 and 8 and negating the number of outgoing transitions to properly record the pointer.

So, was it worth? There is a 10% performance gain after adding this feature for all representations of the FSA (vector< vector >, row major, column major and transition lists). There is significant overlap between the words otherwise the gains would be more spectacular.

The building of the FSA transition matrix by incrementally adding the words of the language is trivial. Translating the transition matrix into the proposed transition list is relatively straightforward; the only complex bit is the path compression. There is no need to implement a slow post-order traversal of the transition matrix to find the states that lead to only one accepting terminal node. Realizing that the successor state is always bigger than its predecessor the reader can use a dynamic programming idea to scan all states starting from the last (using “nbstates” additional temporary memory) to complete this task in linear time. This is a significant saving over a recursive post-order traversal.

The source code used for building the FSA and the performance measurements is available from the author.

On my computer linear search and binsearch take about the same time for 10 elements, while for 9 elements linear search is about 15% fatser.

# Performance measures

Our intuition tells that the automata should be quickest on the data set, but it would be useful to see if we have gained at least a factor of magnitude in execution time for the dimension of our problem? The naive solution has excellent data locality, but its execution time is linear with the size of the language. The automaton is proportional with average length of the words (the big-oh coefficient is slightly worst for the transition list implementation which has the best data locality property).

We look at the amount of trashing for all six variants. Remember trashing, or page faults occurs when the code is requesting a memory location that is not yet in the cache.

Our efforts yield a factor of 10 speedup between the naive and the FSA implementations. This is broadly the ratio of n/m (approx. 300 words, 15-25 symbol long words).

Modern performance analysis cannot ignore data locality. How do we count page faults? need to exclude the part of the program constructing the

Test results

The program was run on a i7-3960K processor, cache sizes: L1 6x32k, L2 6x256k, L3 12M, as per

<http://www.cpu-world.com/CPUs/Core_i7/Intel-Core%20i7-3930K.html>

Visual Studio 2015 runnig on a sturdy old Windows 7.

# References

1. H. Albert-Lörincz, *Understanding hash functions*, forthcoming

Understanding hash functions

The calculation speed of the hash functions is compared in below table.

Don’t Point at me!

In this short article we take a look at pointers to class members in C++. The language does it’s best to discourage programmers from wanting to have a void pointer to a member function. Calling function pointers from outside a class instance is dangerous as it may violate encapsulation rules.

There is a not so widely known syntax for defining and calling function pointers to a member function. A pointer to a member function is different from an ordinary function pointer in more than one way:.

* The type declaration to a member function needs to be preceded by a scope
  + typedef bool (ordinaryPtr)(); // pointer to bool func() outside a class
  + typedef bool (tradeChecker::\*memberPtr)(); // pointer to bool func()member function
* Pointer to the member function can be created by adding the class visibility scope in the & operator
  + memberPtr p = &classNane::memberFunctioName;
  + ordinaryPtr p = &ordinaryFunctioName;
* The member function pointer can only be called within a given instance in contrast with function pointer to ordinary non-member functions that have fewer constraints.
  + (this->\*memberPtr)();
  + ordinaryPtr();

This is all great, but what happens if we have an outside container that stores void pointers, or if we do not want to (or cannot) extend the container with typed objects? Great all around reinterpret\_cast<void\*> does not work on member function pointers, so we need to be looking a bit harder to break the strong C++ type protection logic by venturing outside the scope of C++ language. A few lines of not-so portable assembly extension would do the job nicely:

typedef bool (tradeChecker::\*memberPtr)();

void\* memberPointertoVoid(memberPtr p)

{

void\* tmp;

\_asm

{

mov eax, dword ptr[p]

mov tmp, eax

}

return tmp;

}

memberPtr voidtoMemberPointer(void\* voidptr)

{

memberPtr p;

\_asm

{

mov eax, voidptr

mov dword ptr[p], eax

}

return p;

}

// Usage:

void\* ptr = memberPointertoVoid(&className::member);

p = voidtoMemberPointer(ptr);

(this->\*p)(); // Calls className::member

The English Language

How to get a collection of English books?

The Gutenberg project hosts over 50k full copyright free books (copyright free in the US). Their full catalogue and archive can be downloaded as explained under:

<https://www.gutenberg.org/wiki/Gutenberg:Feeds>

<https://www.gutenberg.org/wiki/Gutenberg:Information_About_Robot_Access_to_our_Pages#How_to_Get_Certain_Ebook_files>

What can you do with the text?

* Gather all English words and their frequencies.
* Information about the context
* Train hash functions
* Etc.