при измерении слишком быстрых алгоритмов (время работы которых имеет порядок разрешающей способности) рекомендуется замерять не каждый вызов, а множество вызовов и вычитать время холостого прогона цикла;

измерения должны повторяться множество раз для одних и тех же условий (результатом эксперимента в таком случае является выборка, а не одно значение измерения);

при использовании JIT-компиляции результаты измерения первого вызова не учитываются, поскольку они будут завышены.

На этапе анализа результатов применяются различные статистические методы и вычисляются числовые характеристики выборки полученной по итогам проведения эксперимента. При анализе результатов, особое внимание следует уделять оптимизациям, которые могут существенно влиять на результаты измерений и зависят от версии JIT и аппаратного обеспечения.

Как можно заметить, на этапе проведения эксперимента существует много однотипных действий, которые необходимо выполнить. В рамках данной работы разработана архитектура приложения, направленного на автоматизацию значительной части процесса эксперимента.

Заключение. Временные затраты – важная характеристика алгоритмов. В современных условиях, при наличии дополнительных слоев абстракций между исходным кодом на языке высокого уровня и машинным кодом, реальные временные затраты могут значительно отличаться от расчетных. В работе выявлены основные этапы процесса измерения временных затрат и их особенности, а также способ автоматизации данного процесса.

1. Cormen, T. Introduction to Algorithms // T.H. Cormen,ʃCh.E. Leiserson,ʃR.L. Rivest,ʃC. Stein – 3rd Edition. – MIT Press, 2009. – 1292 p.
2. Acquiring high-resolution time stamps [Электронный ресурс]: Microsoft development network – Режим доступа: https://msdn.microsoft.com/ library/ windows/ desktop/ dn553408.aspx – Дата доступа: 19.12.2017.

КОДИРОВАНИЕ ТЕКУЩЕГО СОСТОЯНИЯ ДЕТЕРМИНИРОВАННОГО КОНЕЧНОГО АВТОМАТА ЗНАЧЕНИЕМ СЧЕТЧИКА КОМАНД

С.В. Сергеенко
Витебск, ВГУ имени П.М. Машерова

При анализе текста широкое распространение получили регулярные языки и описывающие их регулярные выражения [1]. Актуальным способом эффективного сопоставления текста с заданным регулярным выражением является использование алгоритма, построенный на основе детерминированного конечного автомата, который представляет собой математическую модель, обладающую относительно простой операционной семантикой. [2]

Цель исследования – выяснить возможность реализации детерминированного конечного автомата, в которой текущее состояние кодируется значением счетчика команд. То есть предоставить компилятору как можно больше информации о поведении конечного автомата.

Материал и методы. Материалом исследования служит детерминированный конечный автомат, его программная реализация на языке C++, в которой текущее состояние автомата задается значением счетчика команд. Поставленная цель достигается средствами обобщенного программирования посредством шаблонов в языке программирования C++. Кроме того, были использованы методы математического моделирования и общенаучные методы.

Результаты и их обсуждение. Алфавит и набор состояний конечного автомата, а также типы входного потока и получаемого результата задаются как псевдонимы типов, объявленные в рамках класса, указываемого как параметр шаблона класса `Common`, инкапсулирующего реализацию детерминированного конечного автомата. Для удобства дальнейшего использования, общий для различных конечных автоматов код вынесен в пространство имен `DFA`. Кроме того, введен вспомогательный шаблон класса `DfaTraits`, отвечающий за определение базовых типов, на которых основано определение класса `Common`.

```
namespace DFA {
 template<class States, class CharT, class Result = bool,
 class CharTraits=std::char_traits<CharT>>
```

```
 struct DfaTraits {
 typedef States state_type;
 typedef CharT alphabet_type;
 typedef Result result_type;
 typedef CharTraits alphabet_traits;
 static const result_type
  declined_result_value = result_type();
 };
 template<class Traits>
 struct Common {
 typedef Traits dfa_traits;
 typedef typename Traits::result_type result_type;
 typedef typename Traits::alphabet_type alphabet_type;
 typedef typename Traits::alphabet_traits
  alphabet_traits;
 typedef typename Traits::state_type state_type;
 typedef std::basic_istream<alphabet_type,
  alphabet_traits> stream_type;
 private:
 stream_type &s;
 public:
 Common(stream_type &is) : s(is) {}
 template<state_type st>
 result_type read() {
  typename alphabet_traits::int_type
  ch = s.get();
  if (ch == alphabet_traits::eof()) return check<st>();
  return next<st>(ch);
 }
 protected:
 template<state_type st>
 result_type check() {
  return dfa_traits::declined_result_value;
 }
 template<state_type st>
 result_type next(alphabet_type ch) {
  return read<dfa_traits::dead_state>();
 }
 };
}
```

Класс `Common`, инкапсулирующий реализацию конечного автомата, содержит ссылку на входной поток символов, а также шаблоны методов read, next и check, отвечающие, соответственно, за чтение очередного символа, определение очередного состояния и определение результата считывания строки при указанном в качестве параметра шаблона состоянии.

Выше описанный код применим при реализации любого детерминированного конечного автомата. Рассмотрим далее, как дополнить этот код для реализации некоторого конкретного конечного автомата. Нам необходимо определить тип, значения которого будут соответствовать состояниям конечного автомата, и класс, предназначенный для указания в качестве шаблонного параметра класса `Common`. Затем необходимо выполнить специализацию шаблонных функций `next` и `check`, чтобы задать функцию переходов и множество заключительных состояний конечного автомата.

```
enum states { q0, q1, q2 };
typedef DFA::DfaTraits<states, char> Dfa0Traits;
namespace DFA {
```

```
template<> template<>
bool Common<Dfa0Traits>::next<q0>(char ch) {
 switch(ch) {
 case 'e': return read<q0>();
 case 'a': case 'c': case 'f': return read<q1>();
 default: return read<q2>();
 }
}
template<> template<>
bool Common<Dfa0Traits>::next<q1>(char ch)
{
 switch(ch) {
 default: return read<q0>();
 case 'a': return read<q1>();
 case 'e': case 'f': return read<q2>();
 }
}
template<> template<>
bool Common<Dfa0Traits>::next<q2>(char ch)
{
 switch(ch) {
 case 'b': case 'c': case 'd': return read<q0>();
 default: return read<q1>();
 case 'f': return read<q2>();
 }
}
template<> template<>
bool Common<Dfa0Traits>::check<q2>() { return true; }
}
```

Для использования полученной реализации конечного автомата необходимо сначала создать объект класса `Common<Dfa0Traits>`, передав параметром конструктору поток входных символов, а затем вызвать метод `read<q0>` этого объекта, где `q0` – значение соответствующее стартовому состоянию конечного автомата.

Заключение. Для каждого используемого набора аргументов шаблона функции компилятором будет создаваться фрагмент кода, размещаемого по отдельному диапазону адресов. Следовательно, значение счетчика команд при выполнении результирующего кода будет определять значение шаблонного параметра, обозначающего текущее состояние конечного автомата.

1. Фридл, Дж. Регулярные выражения / Дж. Фридл. – СПб : «Питер», 2001. – 352 с.
2. Компиляторы: принципы, технологии и инструментарий / А. В. Ахо, М. С. Лам, Р. Сети, Дж. Д. Ульман. – 2-е изд. : Пер. с англ. – М. : ООО «И. Д. Вильямс», 2008. – 1184 с.

ПРИМЕНЕНИЕ РАСХОДЯЩИХСЯ СТЕПЕННЫХ РЯДОВ ДЛЯ ПОЛУЧЕНИЯ ФОРМУЛ ПРИБЛИЖЕННОГО НАХОЖДЕНИЯ РЕШЕНИЙ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

Ю.В. Трубников, М.М. Чернявский, А.М. Воронов
Витебск, ВГУ имени П.М. Машерова

Как известно, получение точного аналитического выражения корней алгебраического уравнения пятой и более высоких степеней через его коэффициенты является невозможным (теорема Абеля) [1, с. 103]. Численные алгоритмы решения алгебраических уравнений в своем большинстве являются итерационными, поэтому они не всегда удобны для применения на практике, поэтому актуальным является получение прямых алгоритмов нахождения