Графы

Неориентированный граф

Определение. *Граф*, или *неориентированный граф* G — это упорядоченная пара $G = (V, E)$, где V — это непустое множество вершин или узлов, а E — множество пар (в случае неориентированного графа — неупорядоченных пар, т.е. двухэлементных подмножеств V, таким образом, петли в неориентированном графе обычно не рассматриваются) вершин, называемых рёбрами. И в неориентированных, и в ориентированных граф рёбра обычно обозначаются как (u, v).

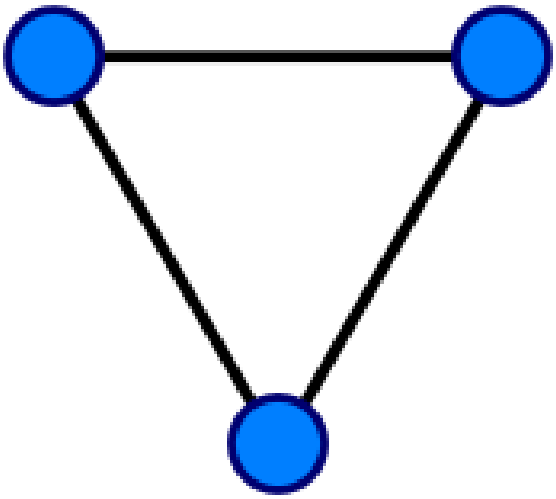

Рис. 1: Неориентированный граф

Маршрутом в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей вершиной ребром, т.е. маршрут от v_0 до v_k — это конечная последовательность (упорядоченный набор) $(v_0, v_1, \ldots, v_k)$, $(v_i, v_{i+1}) \in E$, для всех $i < k$. Длиной маршрута называется количество составляющих его рёбер, так для последовательности $(v_0, v_1, \ldots, v_k)$ она равна k. Всегда имеется маршрут (v_0, v_0) нулевой длины из вершины в неё саму. Часто вместо термина маршрут используют термин *путь*. Путь называется *простым*, если все его вершины различны (иногда называют *простой цепью*).

Иногда рассматривают более тонкое различие между *простой цепью* и *цепью*. В таком случае *цепью* называется маршрут без повторяющихся рёбер, откуда следует, что в простой цепи нет повторяющихся рёбер. Примером цепи, но не простой цепи, является цикл на картинке выше: рёбра не повторяются, но начальная и конечная вершина такого пути совпадают.

В неориентированном графе путь $(v_0, \ldots, v_k)$ образует *цикл*, если $k \geqslant 3$, $v_0 = v_k$ и все вершины $v_1, v_2, \ldots, v_k$ различны.

В неориентированном графе, таким образом, минимальный цикл состоит из 3 рёбер (петель нет, и одно ребро «туда-обратно» не засчитывается); цикл и простой цикл не различаются.

Неориентированный граф называется:

- *связным*, если для любых вершин u, v есть путь из u в v;
- *ациклическим*, если он не содержит циклов;
- *деревом*, если он связный и ациклический;
- *лесом*, если он содержит несколько непересекающихся деревьев.

Ориентированный граф

Определение. *Ориентированный граф* (сокращённо орграф) G — это упорядоченная пара $G = (V, E)$, где V — непустое множество вершин или узлов, и E — множество (упорядоченных) пар различных вершин, называемых ориентированными рёбрами (дугами).

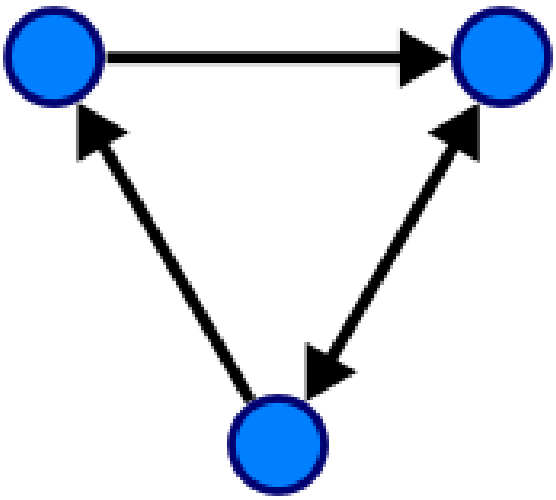

Рис. 2: Ориентированный граф

Дуга — это упорядоченная пара вершин (v, w), где вершину v называют началом, а w — концом дуги. Можно сказать, что дуга $v \to w$ ведёт от вершины v к вершине w. Дуги в ориентированном графе могут образовывать петли (v, v).

Понятия пути (маршрута) и простого пути в орграфе совпадают с таковыми в неориентированном графе. Но

В ориентированном графе путь $(v_0, v_1, \ldots, v_k)$ образует *цикл*, если $v_0 = v_k$ и путь содержит по крайней мере одно ребро, т.е. $k \geqslant 1$. Цикл называется *простым*, если кроме этого все вершины $v_1, v_2, \ldots, v_k$ различны.

Пример. Рассмотрим следующие графы:

Здесь изображены ориентированный граф G_1 с множеством вершин $\{1, 2, 3, 4, 5, 6\}$ и неориентированный граф G_2 с таким же множеством вершин.

В орграфе G_1 путь $(1, 2, 5, 4)$ является простым путём длины 3, а путь $(2, 5, 4, 5)$ простым не является, но является цепью.

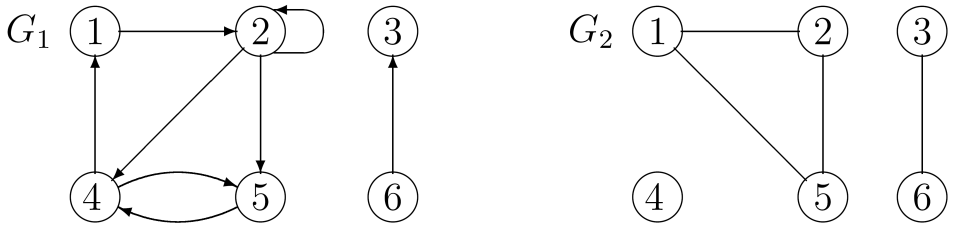

Рис. 3: Примеры на графах

В графе G_1 путь $(1, 2, 4, 1)$ образует тот же цикл, что и пути $(2, 4, 1, 2)$ и $(4, 1, 2, 4)$. Этот цикл простой. Цикл $(1, 2, 4, 5, 4, 1)$ не является простым. Петля $(2, 2)$ является циклом длины 1. В графе G_2 единственным циклом является путь $(1, 2, 5, 1)$, он же является цепью.

Иногда рассматривают более ещё сложные версии графов: мультиграфы (графы с «параллельными» рёбрами), псевдографы (графы с петлями), графы с помеченными рёбрами, смешанные графы и т.п.

Деревья в теории чаще всего рассматриваются как неориентированные графы, но тоже бывают подходы, когда рассматривают и ориентированные (направленные) деревья (Polytree).

Графы в целом широко применяются в программировании как способ описания систем со сложными связями.

Более востребованы ориентированные графы.

Бинарное отношение над конечным носителем может быть представлено в виде орграфа. Простым орграфом представимы антирефлексивные отношения, в общем случае требуется орграф с петлями. Если отношение симметрично, то его можно представить неориентированным графом, а если антисимметрично, то направленным графом.

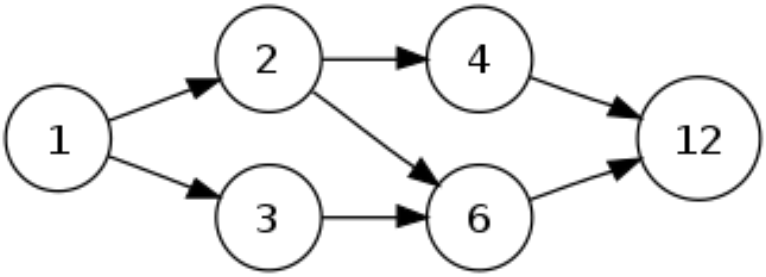

Рис. 4: Орграф отношения делимости

Еще более востребованы будут деревья, о которых мы поговорим позже. Известным примером будет дерево синтаксического разбора, структура `XML`, вложенных директорий в файловой системе и т.п.

Граф (математика)

Ориентированный граф

Дерево (теория графов)

Глоссарий теории графов

Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ.

Для графов в целом определено много полезных алгоритмов. Например, обходы графов в глубину и в ширину. Их реализация на Haskell получается не чисто функциональная, так как нужно сохранение внешнего состояния (например, для обхода графа в ширину требуется ведение глобальной очереди обойдённых вершин, а нерекурсивный вариант обхода в глубину требует глобального стека, также возникает необходимость раскраски посещённых и открытых вершин).

Если рассмотреть более простую задачу обхода дерева, то здесь ситуация за счет особой «индуктивной» структуры не требует запоминания пройденных вершин и хорошо реализуется в функциональном стиле.

Графы, деревья в Haskell

Орграфы

Ориентированные конечные графы представлены в Haskell пакетом Data.Graph.

Вершины — это просто целые числа:

```
type Vertex = Int
```

Рёбра (дуги) — это упорядоченные пары вершин:

```
type Edge = (Vertex, Vertex)
```

Собственно графы представлены в виде массивов «смежности», когда каждой вершине ставится в соответствие список смежных вершин:

```
type Graph = Array Vertex [Vertex]
type Bounds = (Vertex, Vertex)
```

`Bounds` — границы массива.

Функция-построитель (конструктор)

```
buildG :: Bounds -> [Edge] -> Graph
```

позволяет задавать орграф указанием границ (для целых чисел, задаваемых интервалом внутри границ) и списка дуг:

```
buildG (0,2) [(0,1), (1,2)] ==
  array (0,2) [(0,[1]),(1,[2]),(2,[])]
buildG (0,2) [(0,1), (0,2), (1,2)] ==
  array (0,2) [(0,[2,1]),(1,[2]),(2,[])]
```

Можно, как и в массивах, указывать отрицательные числа:

```
ghci> :m Data.Graph
ghci> buildG ((-1),1) [((-1),1), (0,1)]
array (-1,1) [(-1,[1]),(0,[1]),(1,[])]
```

В модуле Data.Graph уже реализован алгоритм обхода графа в глубину функцией `dfs`, и мы не будем рассматривать детали ее реализации и работы.

Деревья

О представлении деревьев в Haskell мы уже неоднократно говорили на лекциях, а на лекции-8 даже рассмотрели модуль Data.Tree и общий тип:

```
data Tree a = Node {rootLabel :: a, subForest :: Forest a}
   deriving ( Eq, Ord, Show, ...)

type Forest a = [Tree a]
```

где `Tree a` задаёт общее дерево («розовый куст», а `Forest a` — список таких деревьев, или лес.

Вот более ясное определение:

```
data RoseTree a = Node a [RoseTree a]
```

В модуле Data.Tree тоже есть множество готовых и полезных функций, и мы многие из них уже рассматривали. Например,

```
flatten :: Tree a -> [a]
```

возвращает список вершин в прямом обходе дерева (в глубину).

Нам, однако, было бы интересно задать самим функции обхода деревьев, тем более, что это уж и не так сложно сделать. Отметим при этом, что когда говорят об обходах графов, то в качестве результата получают так называемое *дерево поиска*. Когда мы обходим дерево, то в качестве результата в алгоритмах ниже будем указывать список посещённых вершин.

Рассмотрим некоторые подходы по мотивам статей

Functional programming with graphs

Arian Stolwijk. Tree Traversal, Depth first and Breadth first in Haskell

Depth First

Сначала сделаем для бинарного дерева:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
  deriving Show
```

Код будет очень простой и мы его уже встречали:

```
traverseDF :: Tree a -> [a]
traverseDF Empty        = []
traverseDF (Node a l r) =
  a : ((traverseDF l) ++ (traverseDF r))
```

Arian Stolwijk. Tree Traversal, Depth first and Breadth first in Haskell

Для обхода в глубину есть, кстати, целых три варианта:

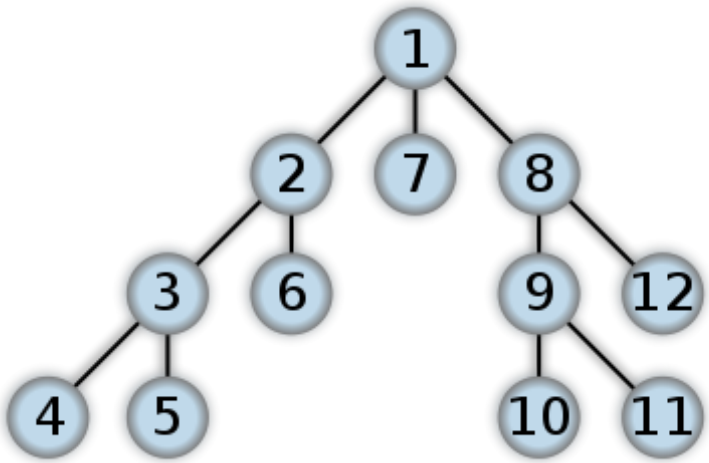

Рис. 5: Обход дерева в глубину

- прямой (pre-order);
- центрированный (in-order);
- обратный (post-order).

Кроме этих трёх основных схем, возможны более сложные гибридные схемы, такие как алгоритмы поиска с ограниченной глубиной, подобные поиску в глубину с итеративным углублением.

```
preorder :: Tree a -> [a]
preorder Empty = []
preorder (Node a l r) =
  a : ((traverseDF l) ++ (traverseDF r))

inorder :: Tree a -> [a]
inorder Empty = []
inorder (Node a l r) =
  (inorder l)++[x]++(inorder r)

postorder :: Tree a -> [a]
postorder Empty = []
postorder (Node a l r) =
  (postorder l)++(postorder l)++[x]
```

Ну и с помощью функции **concatMap** можем написать алгоритм обхода «розового куста»:

```
data RoseTree a = Node a [RoseTree a]

traverseDRT :: RoseTree a -> [a]
traverseDRT (Node x []) = [x]
traverseDRT (Node a list) =
  a : (concatMap traverseDRT list)

test = traverseDRT (Node 1 [Node 2 [], Node 3 []])
test' =
  traverseDRT (Node 1 [Node 2 [Node 5 [], Node 6 []], Node 3 [], Node 4 []])
```

Где в последнем случае закодировано дерево:

```
    1
  / | \
2   3  4
|\
5 6
```

Тестируем:

```
>test
[1,2,3]
> test'
[1,2,5,6,3,4]
```

Где полезная нам функция **concatMap** является гибридом-композицей **map** (при использовании обхода с функцией, возвращающей списки) и последующего склеивания их с помощью функции **concat**.

На самом деле есть разница, см.

Difference between concatMap f xs and concat $ map f xs?

Breadth First

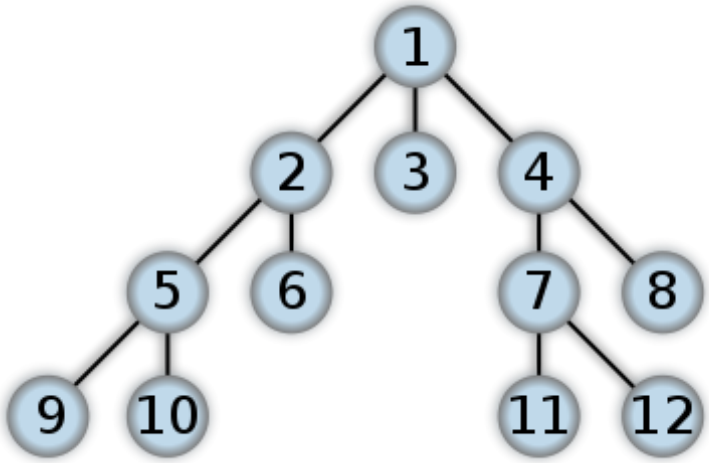

Рис. 6: Обход дерева в ширину

Здесь код немного сложнее. Рассмотрим сначала для случая бинарного дерева:

```
traverseBF :: Tree a -> [a]
traverseBF tree = tbf [tree]
    where
        tbf [] = []
        tbf xs = map nodeValue xs ++ tbf (concat (map leftAndRightNodes xs))
        nodeValue (Node a _ _) = a
        leftAndRightNodes (Node _ Empty Empty) = []
        leftAndRightNodes (Node _ Empty b)     = [b]
        leftAndRightNodes (Node _ a Empty)     = [a]
        leftAndRightNodes (Node _ a b)         = [a,b]
```

А теперь для обхода «розового куста»:

```
traverseBRT :: RoseTree a -> [a]
traverseBRT tree = tbf [tree]
    where
        tbf [] = []
        tbf xs = map nodeValue xs ++
           tbf (concatMap listNodes xs)
        nodeValue (Node a _) = a
        listNodes (Node _ []) = []
        listNodes (Node _ list) = list

test2 = traverseBRT (Node 1 [Node 2 [], Node 3 []])
test2' = traverseBRT (Node 1 [Node 2 [Node 5 [],
 Node 6 []], Node 3 [], Node 4 []])
```

Тестируем:

```
> test2
[1,2,3]
> test2'
[1,2,3,4,5,6]
```

Конечные автоматы

Конечные автоматы относятся к классу словарных алгоритмов. С помощью конечных автоматов можно решать задачи из достаточно широкого семейства алгоритмических проблем. Например, технология проектирования микросхем основана на результатах теории автоматов. Другой пример — это компиляторы, перерабатывающие текст программы, написанной на языке программирования высокого уровня, в программу на машинном языке. Работа любого такого компилятора состоит из двух стадий. Первая стадия — лексический анализ текста, который реализуется с помощью определённого конечного автомата. Вторая стадия — синтаксический анализ, который осуществляется с помощью автомата с магазинной памятью. Однако, как будет видно позднее, не любая алгоритмическая задача поддаётся решению с помощью конечного автомата.

Отличительной чертой конечных автоматов является отсутствие памяти вне процессора, т.е. вся память автомата занята программой. Эта особенность позволяет пролить свет на причины неспособности конечных автоматов решить любую алгоритмически разрешимую задачу (для алгоритмов определённого вида необходима дополнительная память).

ДКА

Определение. *Детерминированным конечным автоматом* (сокращённо ДКА, eng: DFA) называется упорядоченная пятёрка $\mathfrak{A} = (Q, A, \delta, q_0, F)$, состоящая из следующих объектов:

а) $Q = \{q_0, \ldots, q_m\}$ — конечный алфавит *внутренних состояний автомата*;

б) $A = \{a_0, \ldots, a_n\}$ — конечный *входной алфавит* автомата;

в) $\delta : Q \times A \to Q$ — *функция перехода*;

г) $q_0 \in Q$ — *начальное состояние*;

д) $F \subseteq Q$ — множество *выделенных (конечных) состояний*.

Графическое изображение детерминированных автоматов

Автоматы удобно изображать графически, используя следующие геометрические фигуры:

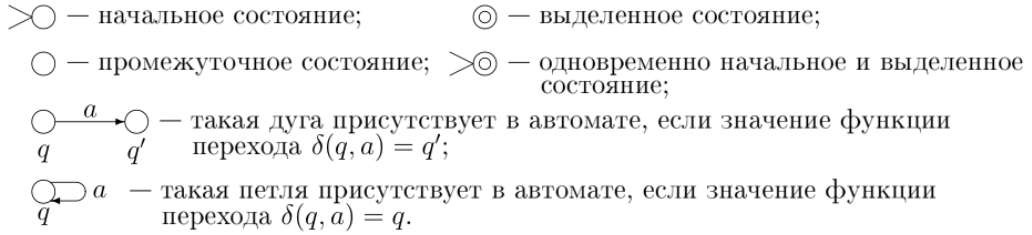

Рис. 7: Граф ДКА

Пример. Например, автомат $\mathfrak{A} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$, где $\mathcal{A} = \{a, b\}$, $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, а функция перехода задается соотношениями

$$\begin{array}{lll} \delta(q_0, a) = q_0, & \delta(q_1, a) = q_0, & \delta(q_2, a) = q_2, \\ \delta(q_0, b) = q_1, & \delta(q_1, b) = q_2, & \delta(q_2, b) = q_2, \end{array}$$

имеет следующее графическое изображение:

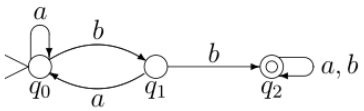

Рис. 8: Пример графа ДКА

Определение. Путём в детерминированном конечном автомате $\mathfrak{A} = (Q, A, \delta, q_0, F)$ назовём любую конечную последовательность $(r_0, s_1, r_1, \ldots, s_k, r_k)$, где $r_0, \ldots, r_k \in Q$, $s_1, \ldots, s_k \in A$ и $\delta(r_i, s_{i+1}) = r_{i+1}$ для всех $i < k$. Если $(r_0, s_1, r_1, \ldots, s_k, r_k)$ — путь в автомате, то будем обозначать его через

$$r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} \ldots \xrightarrow{s_k} r_k$$

и говорить, что слово $w = s_1 s_2 \ldots s_k$ читается вдоль дуг данного пути. В частности, пустое слово Λ читается вдоль пути (r_0), состоящего из одного состояния и не содержащего ни одной дуги.

Определение. Пусть $\mathfrak{A} = (Q, A, \delta, q_0, F)$ — ДКА. Расширим функцию δ до функции $\delta^* : Q \times A^* \to Q$ следующим образом индукцией по длине слова:

$$\delta^*(q, \Lambda) = q, \quad \delta^*(q, wa) = \delta(\delta^*(q, w), a),$$

где $q \in Q$, $w \in A^*$, $a \in A$.

Определение. Говорят, что ДКА $\mathfrak{A} = (Q, A, \delta, q_0, F)$ распознаёт слово $w \in A^*$, если $\delta^*(q_0, w) \in F$.

Другими словами, слово $w = s_1 s_2 \dots s_k$ распознаётся автоматом, если в нём существует путь

$$q_0 = r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} \dots \xrightarrow{s_k} r_k \in F$$

такой, что он начинается в начальном состоянии q_0, вдоль его дуг читается слово $s_1 s_2 \dots s_k$ и заканчивается в некотором выделенном состоянии.

НДКА

Определение. *Недетерминированным конечным автоматом* (сокращённо НДКА, eng: NDFA) называется упорядоченная пятёрка $\mathfrak{A} = (Q, A, \Delta, q_0, F)$, в которой Q, A, q_0, F определяются и называются так же, как в детерминированном случае, а *функция переходов* Δ является функцией вида $\Delta : Q \times A \to P(Q)$, где $P(Q)$ — множество всех подмножеств Q.

Определение. Путь в НДКА $\mathfrak{A} = (Q, A, \Delta, q_0, F)$, в которой Q, A, q_0, F определяется и обозначается так же, как в детерминированном случае, нужно лишь условие $\delta(r_i, s_{i+1}) = r_{i+1}$ заменить на условие $r_{i+1} \in \Delta(r_i, s_{i+1})$.

Определение. Говорят, что НДКА $\mathfrak{A} = (Q, A, \Delta, q_0, F)$ распознаёт слово $s_1 s_2 \dots s_k \in A^*$, если существует последовательность состояний $q_0 = r_0, r_1, \dots, r_k$ такая, что

$$\begin{aligned} r_1 &\in \Delta(r_0, s_1), \\ r_2 &\in \Delta(r_1, s_2), \\ &\vdots \quad \vdots \\ r_k &\in \Delta(r_{k-1}, s_k), \end{aligned}$$

и при этом $r_k \in F$.

Пример. Например, автомат $\mathfrak{A} = \langle Q, \mathcal{A}, \Delta, q_0, F \rangle$, где $\mathcal{A} = \{a, b\}$, $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, а функция переходов задаётся соотношениями

$$\begin{array}{lll} \Delta(q_0, a) = \{q_0\}, & \Delta(q_1, a) = \varnothing, & \Delta(q_2, a) = \varnothing, \\ \Delta(q_0, b) = \{q_0, q_1\}, & \Delta(q_1, b) = \{q_2\}, & \Delta(q_2, b) = \varnothing, \end{array}$$

имеет следующее графическое изображение:

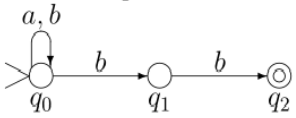

Рис. 9: Пример-1. НДКА

Построен с помощью визуализатора

Регулярные выражения и конечные автоматы

… алгоритм Томпсона.

Russ Cox. Regular Expression Matching Can Be Simple And Fast

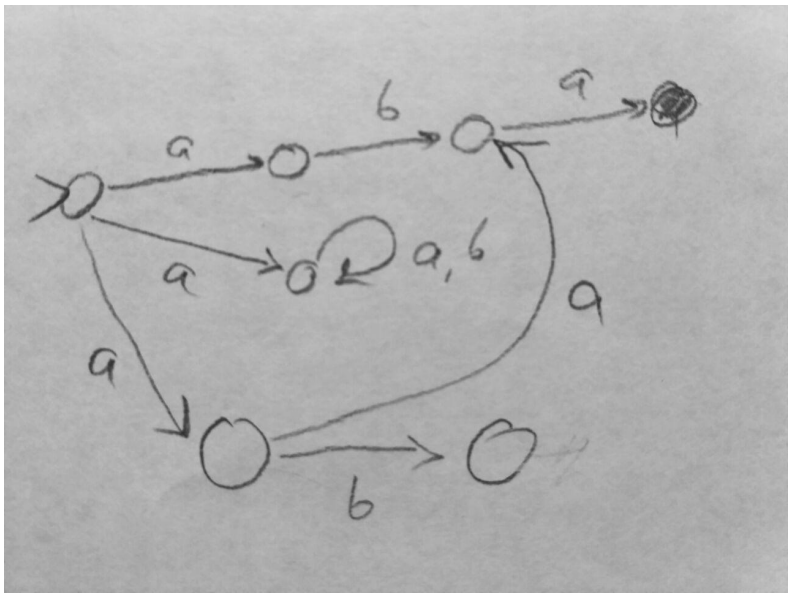

Рис. 10: Пример-2. НДКА

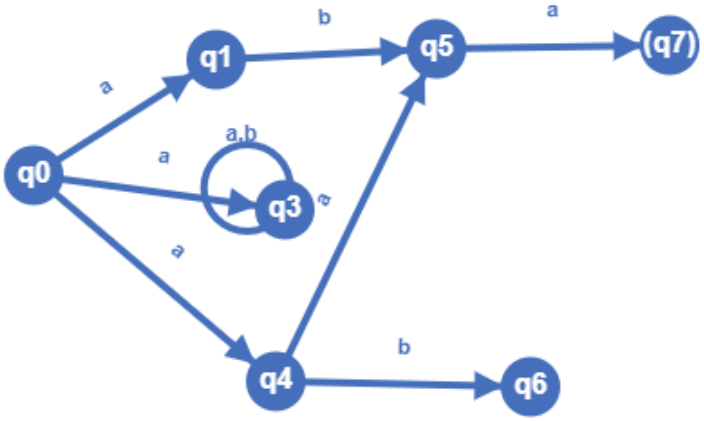

Рис. 11: Этот же пример

Определение. Пусть $\mathcal{A}$ — конечный алфавит, не содержащий символов (,), $\cup$, $*$. Определим по индукции множество *регулярных выражений* над алфавитом $\mathcal{A}$:

1^0. Множества $\varnothing$ и a, где $a \in \mathcal{A}$, являются *регулярными выражениями*.

2^0. Если α и β — регулярные выражения, то $(\alpha\beta)$, $(\alpha \cup \beta)$ и (α^*) тоже являются *регулярными выражениями*.

Рис. 12: Определение регулярных выражений

Определение. Определим отображение L из множества всех регулярных выражений над алфавитом $\mathcal{A}$ в множество всех языков над $\mathcal{A}$ следующим образом:

$$L(\varnothing) = \varnothing,$$
$$L(a) = \{a\}, \text{ для любого } a \in \mathcal{A},$$
$$L(\alpha\beta) = L(\alpha)L(\beta),$$
$$L(\alpha \cup \beta) = L(\alpha) \cup L(\beta),$$
$$L(\alpha^*) = L(\alpha)^*.$$

Определение. Язык L над алфавитом $\mathcal{A}$ называется *регулярным*, если существует регулярное выражение α над алфавитом $\mathcal{A}$ такое, что $L(\alpha) = L$. При этом будем говорить, что выражение α *задаёт* язык L.

Замечание. Из тождества $\varnothing^* = \{\Lambda\}$ следует, что язык $\{\Lambda\}$, состоящий из одного пустого слова, регулярен. Если $a_1, \ldots, a_s$ — произвольные символы из алфавита, то из тождества $\{a_1, \ldots, a_s\} = \{a_1\} \cup \ldots \cup \{a_s\}$ следует регулярность языка $\{a_1, \ldots, a_s\}$.

Рис. 13: Определение регулярных языков

версия, сохранённая в вебархиве: Russ Cox. Regular Expression Matching Can Be Simple And Fast

Конечные автоматы в Haskell

ДКА

Необходимая библиотека функций:

```
module DFAFold where

type State   = Int
type Symb    = Char

data DFinStAutomata = DFA {first,end :: State, fin :: [State],
d:: State -> Symb -> State }

d'  :: (State -> Symb -> State) -> State -> [Symb] -> State
d' = foldl

check :: DFinStAutomata -> [Symb] -> Bool
check dfa word = (d' (d dfa) (first dfa) word) `elem` (fin dfa)
```

Зададим примеры:

```
import DFAFold

dfa1 = DFA {
  first=1, end=4, fin=[4],
  d  = \q n -> case (q,n)  of
    (1, 'a') -> 2
    (1, 'b') -> 1
    (2, 'a') -> 2
    (2, 'b') -> 3
    (3, 'a') -> 4
```

```
    (3, 'b') -> 1
    (4, 'a') -> 4
    (4, 'b') -> 4}

dfa2 = DFA {
 first=1, end=4, fin=[4], d  = t}
   where
    t 1 'a' = 2
    t 1 'b' = 1
    t 2 'a' = 2
    t 2 'b' = 3
    t 3 'a' = 4
    t 3 'b' = 1
    t 4 'a' = 4
    t 4 'b' = 4

c1 = check dfa1 ”aba”
c2 = check dfa2 ”aba”
c3 = check dfa2 ”ab”
```

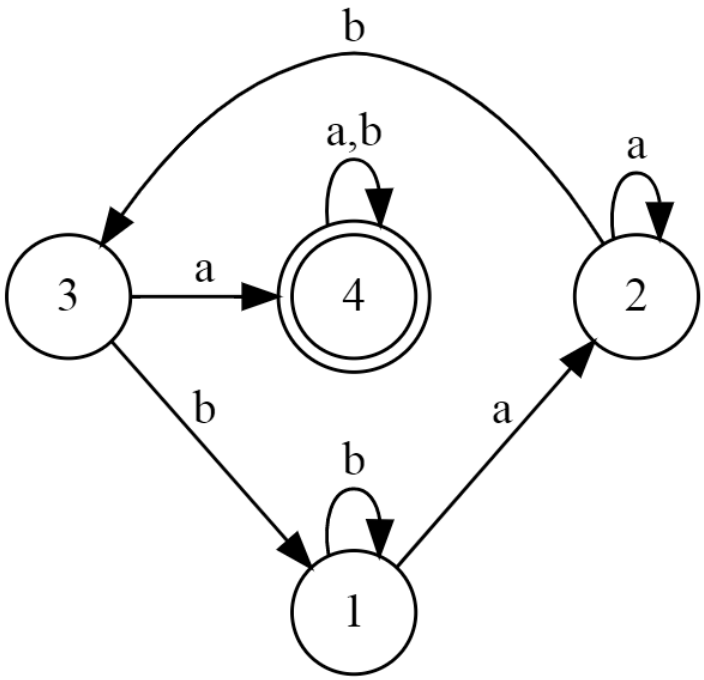

Рис. 14: Пример выше

Задан в визуализаторе Graphiz на языке описания графов DOT

```
digraph finite_state_machine {
    rankdir=LR;
    size=”8,5”
    node [shape = doublecircle]; 4;
    node [shape = circle];
    1 -> 2 [ label = ”a” ];
    1 -> 1 [ label = ”b” ];
```

```
    2 -> 2 [ label = ”a” ];
    2 -> 3 [ label = ”b” ];
    3 -> 4 [ label = ”a” ];
    3 -> 1 [ label = ”b” ];
    4 -> 4 [ label = ”a,b” ];
}
```

The DOT Language

Drawing graphs with dot

И вот проверка: распознавание слов:

```
*Main> c1
True
*Main> c2
True
*Main> c3
False
```

НДКА

Теперь то же самое для НДКА

```
module NDFAFull where

import Control.Monad
import Data.List

type State   = Int
type Symb    = Char

data NDFinStAutomata = NDFA {nfirst,nend :: State, nfin :: [State],
   dn:: State -> Symb -> [State], z:: State -> [State]}

ndrec1 ndfa word = foldM (dn ndfa) (nfirst ndfa) word

ndrec ndfa word = foldM dn_ (nfirst ndfa) word where
   zs q = do
      q1 <- (z ndfa) q
      if (q1 == q)
         then return q
         else (zs q1)
   dn_ q s = do
      q1 <- zs q  -- первое применение zs нужно только для входного
состояния
      q2 <- (dn ndfa) q1 s
      zs q2

-- отдельно для тестирования
zset ndfa q = do
      q1 <- (z ndfa) q
      if (q1 == q)
```

```
          then return q
          else (zset ndfa q1)

nz = NDFA {
  nfirst=1, nend=5, nfin=[3], dn  = \q a -> case (q,a)  of
    (1, 'a') -> [2]
    (1, 'b') -> [4]
    (_,_)    -> [],
  z = \q -> case q of
    2 -> [2,3]
    3 -> [3,5]
    x -> [x]}

ncheck ndfa word = not $ null (
  (ndrec ndfa word) `intersect`
  (nfin ndfa) )
```

Зададим пример:

```
import NDFAFull

ndfa1 = NDFA {
  nfirst=1, nend=4, nfin=[4], dn  = \q a -> case (q,a)  of
    (1, 'a') -> [2]
    (2, 'b') -> [3]
    (3, 'b') -> [4]
    (4, 'a') -> [4]
    (4, 'b') -> [4]
    (_,_)    -> [],
    z = \q -> [q]}

c  = ncheck ndfa1 ”aba”
c2 = ncheck ndfa1 ”abb”
```

И протестируем:

```
*Main> c
False
*Main> c2
True
```

Конструирование НДКА

```
move :: Int -> [State] -> [State]
move n list = map (n+) list

moveZ n z q    = move n (z q)
moveD n dn q s = move n (dn q s)

moveZZ n z q  = do
   q1 <- z q
   return (q1 + n)
```

```
conc :: NDFinStAutomata -> NDFinStAutomata -> NDFinStAutomata
conc ndfa1 ndfa2 = NDFA { nfirst = (nfirst ndfa1),
                          nend = (nend ndfa1) + (nend ndfa2),
                          nfin = move n (nfin ndfa2),
                          z = zz,
                          dn = dnn
                        }
  where
    n = nend ndfa1
    zz q | q <= n = if (q `elem` (nfin ndfa1))
                     then (z ndfa1 $ q) ++ [n+1]
                     else (z ndfa1) q
         | q >  n = moveZZ n (z ndfa2) (q-n)
    dnn q s | q <= n = (dn ndfa1) q s
            | q >  n = moveD n (dn ndfa2) (q-n) s


lit :: Symb -> NDFinStAutomata
lit s = NDFA { nfirst = 1,
               nend   = 2,
               nfin   = [2],
               z = \q -> [q],
               dn = dnn
             }
   where
     dnn n c | (n == 1) && (c == s) = [2]
             | otherwise = []

uni ndfa1 ndfa2 = NDFA { nfirst = 1,
                         nend = 1 + (nend ndfa1) + (nend ndfa2),
                         nfin = (move 1 (nfin ndfa1)) ++
                                   (move (n+1) (nfin ndfa2)),
                         z = zz,
                         dn = dnn
                        }
  where
    n = nend ndfa1
    zz q | q == 1     = [((nfirst ndfa1)+1), ((nfirst ndfa2)+n+1)]
         | q <= (n+1) = moveZZ 1 (z ndfa1) (q-1)
         | q >  (n+1) = moveZZ (n+1) (z ndfa2) (q-(n+1))
    dnn q s | q == 1 = []
            | q <= (n+1) = moveD 1     (dn ndfa1) (q-1) s
            | q >  (n+1) = moveD (n+1) (dn ndfa2) (q-(n+1)) s

kstar ndfa = NDFA { nfirst = 1,
                    nend = 1 + (nend ndfa),
                    nfin = 1:(move 1 (nfin ndfa)),
                    z = zz,
                    dn = dnn
```

```
                  } where
  n = nend ndfa
  zz q | q == 1 = [2] -- [(nfirst ndfa)+1]
       | q > 1  = if ((q-1) `elem` (nfin ndfa))
                     then (moveZZ 1 (z ndfa) (q-1)) ++ [2] -- [(nfirst
ndfa)+1]
                     else moveZZ 1 (z ndfa) (q-1)
  dnn q s | q == 1 = []
          | q > 1  = moveD 1 (dn ndfa) (q-1) s

import NDFAFull

t1 = lit 'a'
c1a = ncheck t1 ”a”
c1b = ncheck t1 ”b”

*Main> c1a
True
*Main> c1b
False

t2 = (lit 'a') `conc` (lit 'b')
c2aa = ncheck t2 ”aa”
c2ab = ncheck t2 ”ab”
c2a  = ncheck t2 ”a”
c2aba  = ncheck t2 ”aba”

*Main> c2aa
False
*Main> c2ab
True
*Main> c2a
False
*Main> c2aba

t3 = (lit 'a') `uni` (lit 'b')
c3aa = ncheck t3 ”aa”
c3ab = ncheck t3 ”ab”
c3a  = ncheck t3 ”a”
c3b  = ncheck t3 ”b”
c3aba  = ncheck t3 ”aba”

*Main> c3aa
False
*Main> c3ab
False
*Main> c3a
True
*Main> c3aba
False
*Main> c3b
True
```

```
t4 = kstar (lit 'a')
c4a  = ncheck t4 ”a”
c4aa  = ncheck t4 ”aa”
c4aaa  = ncheck t4 ”aaa”
c4  = ncheck t4 ””
c4b  = ncheck t4 ”b”
c4ab  = ncheck t4 ”ab”

*Main> c4a
True
*Main> c4aa
True
*Main> c4aaa
True
*Main> c4
True
*Main> c4b
False
*Main> c4ab
False
```