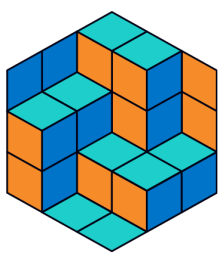
DFA minimization

Hopcroft's algorithm

Andrew Ratkov, Alexander Okhotin

Saint Petersburg State University, MCS



2024-10-15

1. Basic definitions

Definition 1.1.: *DFA (deterministic finite automaton)* is a tuple $(\Sigma, Q, q_0, \delta, F)$ where:

- Σ is an alphabet
- Q is a set of states
- q_0 is a starting state
- $\delta : Q \times \Sigma \to Q$ are the rules of transitions
- F is a set of accepting states

Definition 1.2.: String $w = a_1 a_2 ... a_n \in \Sigma^*$ is *accepting from state q* if $\delta(q, w) := \delta(...\delta(q, a_1), ..., a_n) \in F$.

Definition 1.3.: Let A be a DFA. *L(q, A)* is the set of all accepting from state q words.

Definition 1.4.: Word $w \in \Sigma^*$ is *accepting* if it is accepting from q_0. *L(A) (language set by DFA)* is the set of all accepting words.

Definition 1.5.: Let A and B be two DFAs. If $L(A) = L(B)$, we will say that A and B are *equivalent*.

Definition 1.6.: Let A be a DFA. DFA A' is *a minimized version of A* if A and A' are equivalent and A' has the least possible number of states.

Moore's algorithm of minimization works with complexity $O(n^2)$ (more accurately, $O(|\Sigma|n^2)$), where n is a number of states ($n = |Q|$).

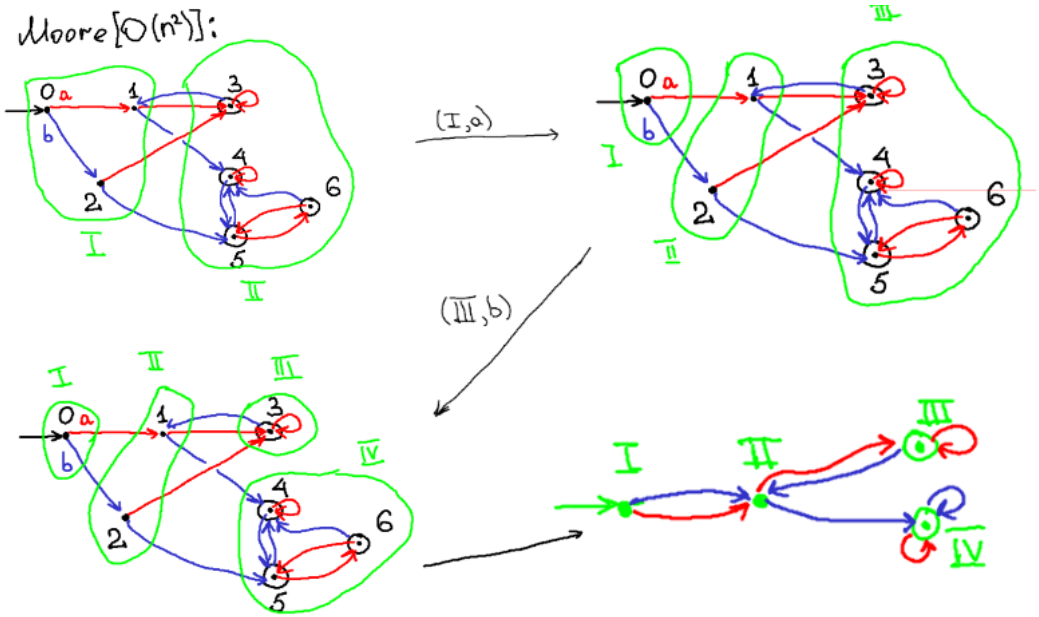


Figure 1: example of Moore alg. running

Figure 2: Seems to be too slow, isn't it?

2. Plans and goals

1. Parse original Hopcroft's article.
2. Learn how to proof correctness and complexity of the algorithm (partly they are not mentioned in the paper).
3. Write implementation of the algorithm in C++, test it and make sure it works in expected $O(|\Sigma| n \log(n))$ time.
4. Write documentation for 3.
5. Extend it to a small C++ library.

3. New definitions

Definition 3.1.: States q_1 and q_2 from DFA A are said to be *equivalent* ($q_1 \sim q_2$) if $L_{q_1}(A) = L_{q_2}(A)$.

Definition 3.2.: If states q_1 and q_2 from DFA A are not equivalent then exists a *separating string*: a string that belongs only to one of the languages $L_{q_1}(A)$, $L_{q_2}(A)$. $(w \in \Sigma^* : w \in L_{q_1}(A) \setminus L_{q_2}(A) \vee w \in L_{q_2}(A) \setminus L_{q_1}(A)\ )$

The main point of the algorithm is to divide the states into blocks: at the end each block becomes an equivalence class. At the beginning all the states will be split into 2 blocks (F and $Q \setminus F$). Each iteration, it will try to split some of the blocks while it does not realize that in every block all the states are equivalent. Algorithm will keep the following thesis: if states q_1 and q_2 are in different blocks, then they have a separating string.

For convenience, I will mark all blocks like $B(0), B(1), ..., B(k-1)$ at the moment when all states are split in k blocks.

Statement 3.1: Any block consists of only accepting states or of only rejecting states.

Lemma 3.2: Let's fix a block $B(i)$ and a symbol $a \in \Sigma$. Lets consider an arbitrary block $B(j)$ and define $B'(j) = \{t \in B(j) \mid \delta(t, a) \in B(i)\}$ and $B''(j) = \{t \in B(j) \mid \delta(t, a) \notin B(i)\}$.
Then $\forall q_1 \in B'(j), \forall q_2 \in B''(j) : q_1 \not\sim q_2$.

Proof: Let $p_1 = \delta(q_1, a)$ and $p_2 = \delta(q_2, a)$. Obviously, $p_1 \in B(i)$ and $p_2 \notin B(i)$, so exists a separating string w for p_1 and p_2. Then, aw is a separating for q_1 and q_2. □

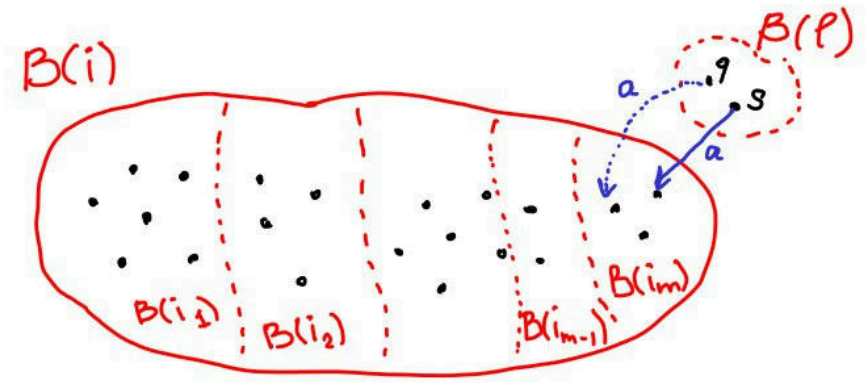
Definition 3.3.: Suppose that all the states are separated in k blocks: $B(0), ..., B(k-1)$. Select a random block $B(i)$ and a symbol $a \in \Sigma$. For each $j < k$ define $B'(j)$ and $B''(j)$ as in Lemma 3.2. If both $B'(j)$ and $B''(j)$ are not empty, replace $B(j)$ with them.
We will call such operation *grinding with block B(i) and letter a*.
If number of blocks increases, the grinding is *useful*, else it is *useless*.

Statement 3.3: Equivalent states continue to stay in the same block during grinding.

Statement 3.4: If any possible grinding (with any block and any letter) is useless, then all blocks are the equivalence classes.

Lemma 3.5: Assume that grinding with block $B(i)$ and letter $a \in \Sigma$ happens during the algorithm. After a while block $B(i)$ was replaced with blocks $B(i_1), ..., B(i_m)$. Then grindings with blocks $B(i_1), ..., B(i_{m-1})$ and letter a happen.
After that, grinding with block $B(i_m)$ and letter a is useless.

Proof: Select a random state s such as $\delta(s, a) \in B(i_m)$. Define the block where this state is as $B(l)$. All we need to proof is that for any state q from $B(l)$, $\delta(q, a) \in B(i_m)$.
Let's assume the opposite. If $\delta(q, a) \in B(i_1)$, states s and q should have been split in different blocks when grinding with block $B(i_1)$ and letter a happened (contradiction).
...
If $\delta(q, a) \in B(i_{m-1})$, states s and q should have been split in different blocks when grinding with block $B(i_{m-1})$ and letter a happened (contradiction).
If $\delta(q, a)$ is even not in old block $B(i)$, states s and q should have been split in different blocks when grinding with block $B(i)$ and letter a happened (contradiction).
Then, $\delta(q, a) \in B(i_m)$.

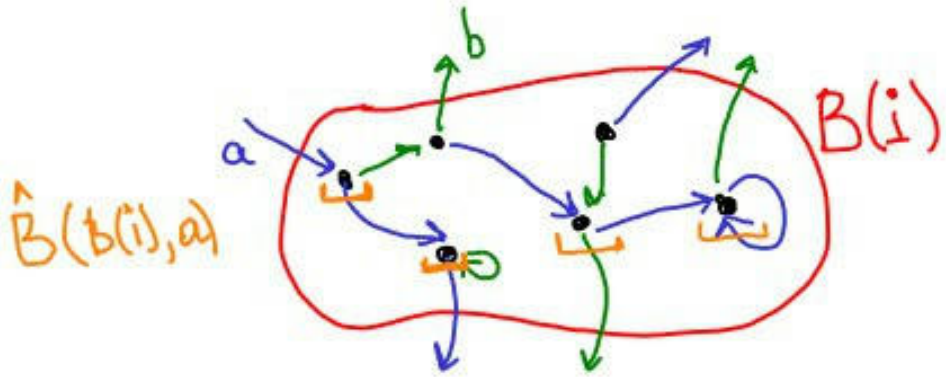


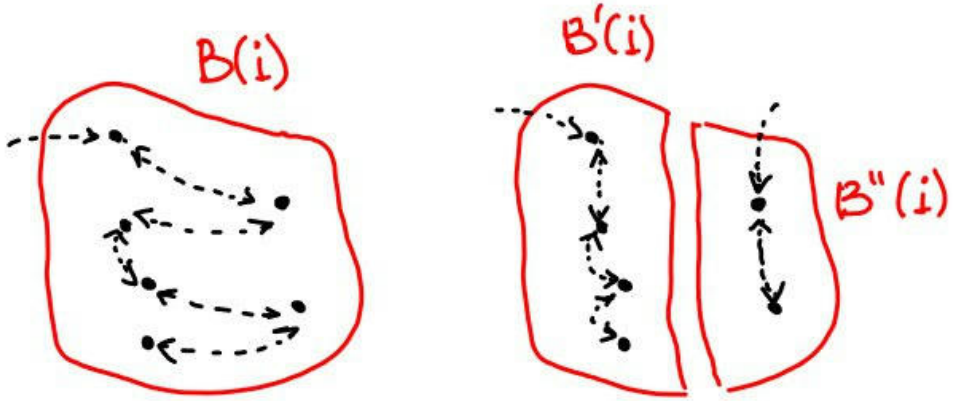
□

4. Pre-algorithm moves and aspects

- Delete states which are unreachable from q_0. Can be done in $O(|\Sigma|n)$ time using BFS (breadth-first search).
- Previously count δ^{-1} (*reversed delta*): $Q \times \Sigma \to 2^Q$, $\delta^{-1}(q,a) = \{t \in Q \mid \delta(t,a) = q\}$. Can be counted in $O(|\Sigma|n)$ time using $O(|\Sigma|n)$ memory.
- States in each block will be stored in double-linked lists, so actions like insertion or deletion of a state can be performed in $O(1)$ time. This means that splitting one block $B(i)$ into two new blocks $B'(i)$ and $B''(i)$ can be performed in $O(\min(|B'(i)|, |B''(i)|))$ time.

- States of block with a reachable by any letter $a \in \Sigma$ will be marked as:
  $\hat{B}(B(i), a) := \{t \in B(i) \mid \exists q \in Q : \delta(q,a) = t\}$
  Obviously, $\hat{B}(B(i), a) \subset B(i)$.
  Sets $\hat{B}(B(i), a)$ also will be saved as double-linked lists (every state, which is reachable by a symbol has links on the next and on the previous states of same block which are also reachable by letter a).

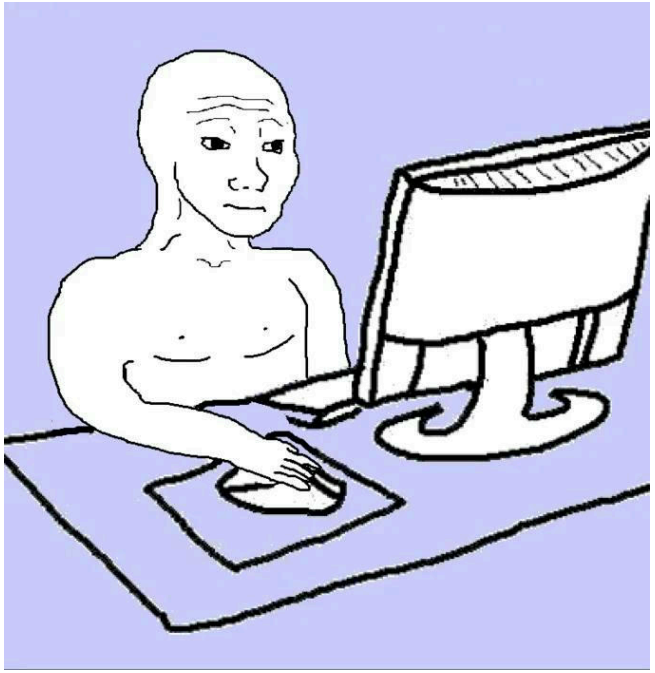




- The algorithm will perform grindings by some blocks and symbols until it realizes that any possible grinding is *useless*. For each symbol $a \in \Sigma$ we will define $L(a)$ as the set of blocks, for which grinding with letter a is not performed yet. I will call such sets as *grinding sets*.

5. Algorithm

```
1  for a ∈ Σ:
2  |  L(a) := ∅
3  colors = 2
4  B(0) := F, B(1) := Q \ F
5  for a ∈ Σ:
6  |  if |B̂(B(0), a)| ≤ |B̂(B(1), a)|:
7  |  |  L(a).add(B(0))
8  |  else:
9  |  |  L(a).add(B(1))
10 while exists a: L(a) ≠ ∅:
11 |  get such a and remove random block B(i) from L(a)
12 |  do grinding with block B(i) and symbol a
13 |  upgrade L
```

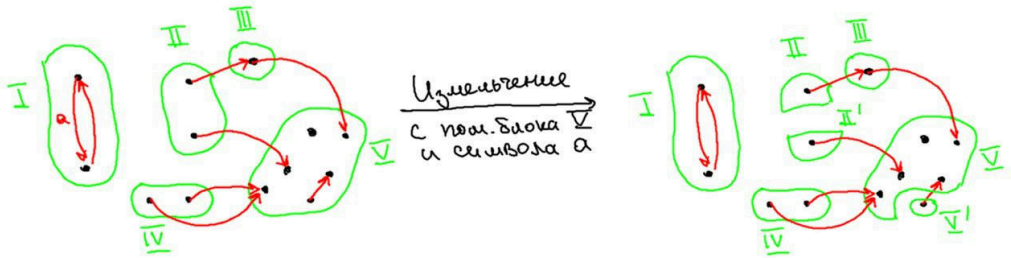


But how to do grinding with block $B(i)$ and symbol a?

- According to the definition: $S := \delta^{-1}(B(i), a)$. And each block $B(k)$ will be separated into 2 new blocks:
  $B'(k) := B(k) \setminus S$,
  $B''(k) := B(k) \cap S$.
  If both are not empty, $B(k)$ will be replaced by them.

```
1 for r ∈ B̂(B(i), a):
2 |  for s ∈ δ⁻¹(s, a):
3 |  |  remove state s to a new block
```

this ↑ can be implemented in $O(|S|)$



- Upgrading L:

```
1 for each split block B(k): (B(k) → B(k), B(l))
2 |  for a ∈ Σ:
3 |  |  if B(k) ∈ L(a):
4 |  |  |  L(a).add(B(l))
5 |  |  else: ← using Lemma 3.2
6 |  |  |  if |B̂(B(k), a)| ≤ |B̂(B(l), a)|:
7 |  |  |  |  L(a).add(B(k))
8 |  |  |  else:
9 |  |  |  |  L(a).add(B(l))
```

6. Asymptotic

Statement 6.1: For each $a \in \Sigma$ each block B was extracted from $L(a)$ not more than $\log(n)$ times.

Statement 6.2: Each state $q \in Q$ changed it's block not more than $\log(n)$ times.

Lemma 6.3: Algorithm works in $O(|\Sigma| n \log(n))$ time complexity.

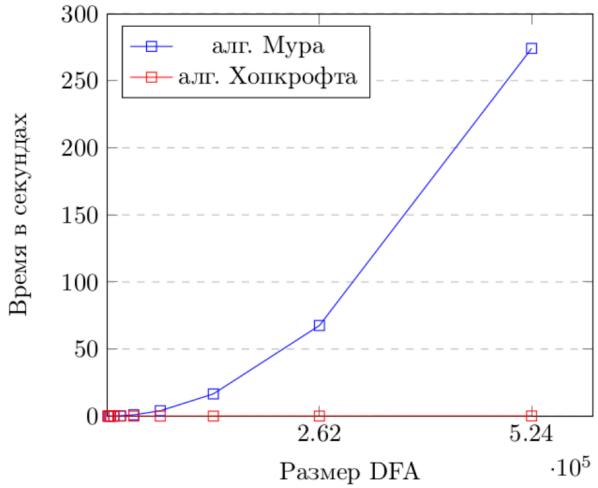
Proof:

- Deleting unreachable states and other pre-algorithm performances: $O(|\Sigma|n)$.
- State moves (from older blocks to newer): $O(|\Sigma| n \log(n))$ (because of Statement 6.2, each move takes $O(|\Sigma|)$ time, amount of states is n).
- Insertions and extractions from grinding sets L: $O(|\Sigma| n \log(n))$ (actually $O(|\Sigma|n)$, but it's harder to proof).

To sum up, everything works in $O(|\Sigma| n \log(n))$ time! □

7. Results

https://github.com/AndrewRatkov/Hopcroft-algorithm-realization



- Memory usage: $\left(40 + \frac{1}{8} + |\Sigma|\left(36 + \frac{1}{8}\right)\right) \cdot n$ bytes.

n	Алгоритм Хопкрофта (сек.)	Алгоритм Мура (сек.)
12	0.000747	0.0010522
13	0.001364	0.0017582
14	0.002397	0.0029991
15	0.00477	0.0049912
16	0.010971	0.0135385
17	0.023378	0.0270009
18	0.047164	0.0590049
19	0.115261	0.127855
20	0.302292	0.296526
21	0.672999	0.65721
22	1.43655	1.5208
23	3.25043	3.84943
24	6.82125	8.92102
25	13.5632	18.8834
26	27.5825	40.6004
27	59.4406	–

Сравнение работы алгоритмов минимизации автоматов, имеющих 2^{n-1} состояний.

Алгоритм Хопкрофта

Алгоритм Мура

Thank you for your attention!

The project was supported by Yandex and not supported by Tinkoff and VK.