DFA Minimization Algorithms

Stefano Genetti

Contents

1 Introduction

In this document I present an implementation of the Watson-Daciuk's Deterministic Finite Automata (DFA) minimization algorithm. The algorithm functionality is based on the contents illustrated in the paper *"An efficient incremental DFA minimization algorithm"*[1] (hereafter called *the paper*) written by Bruce W. Watson and Jan Daciuk.
The code has been fully written in C programming language.
I assume that the DFA minimization problem is well known by the reader.

At the beginning of this document the attention is on the key aspects of the Watson-Daciuk's algorithm. The purpose of the following sections is to reason about data structures used in the actual implementation, efficiency and optimization strategies that has been adopted. Then there is described how to compile the code, the structure of the input and how to read the output.
In order to check the correctness of the output given by the algorithm I have also implemented the DFA minimization algorithm that we have studied during the course, hereafter called *Partition Refinement Algorithm*. At the end of the document there are some indications about this last.

1.1 Notes on notation

In order to achieve a better explanation, I have decided to follow the notation used in the paper. The DFA is represented in this context with the 5-tuple $(Q, \Gamma, \delta, q_0, F)$ where Q is the finite set of states, Γ is the input alphabet, $\delta \in Q \times \Gamma \longrightarrow Q \cup \{\perp\}$ is the transition function ($\perp$ is used to designate the invalid state), $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of final states.

[1]Bruce W. Watson, Jan Daciuk *An efficient omcremental DFA minimization algorithm*
`https://www.cambridge.org/core/journals/natural-language-engineering/article/abs/an-efficient-incremental-dfa-minimization-algorithm/06AB7BA9976CF464149602462F5D780D`

2 Watson-Daciuk algorithm

[Folder: PaperImplementation]
According to the paper the Watson-Daciuk algorithm is an incremental DFA minimization algorithm, meaning that it could be run on a DFA at the same time as the automaton is being used to process a string for acceptance. Furthermore, the minimization algorithm may be halted at any time, with the intermediate result being usable to partially minimize the DFA. All of the other known minimization algorithms have intermediate results which are not usable for partial minimization.

The correctness of the algorithm is explained in the paper extending results of previous works and it is not further commented in my document.

The purpose of the algorithm is to divide the DFA states in classes of equivalence. The code has been structured to strongly map the pseudocode presented in the paper, so the same pseudocode is used here to explain the functionality of the algorithm.

Algorithm 1 Watson-Daciuk

```
1: $S \leftarrow \emptyset$
2: $Def\_Ineq \leftarrow ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$
3: $Def\_Equiv \leftarrow \{(q,q) | q \in Q\}$
4: {**invariant**: $Def\_Ineq \subseteq \neg Equiv \wedge Def\_Equiv \subseteq Equiv$}
5: **while** ( $(Def\_Ineq \cup Def\_Equiv) \neq Q \times Q$ ) **do** {
6:     **let** $p, q : (p,q) \in ((Q \times Q) \setminus (Def\_Ineq \cup Def\_Equiv))$
7:     **if** $equiv(p, q, \max(|Q| - 2, 0)\{$
8:         $Def\_Equiv \leftarrow Def\_Equiv \cup \{(p,q),(q,p)\}$
9:     **else** {
10:        $Def\_Ineq \leftarrow Def\_Ineq \cup \{(p,q),(q,p)\}$
11:    }
12: }
13: merge states according to $Def\_Equiv$ (equivalence is transitive)
```

The algorithm partitions states into two sets: *Def_Equiv*, *Def_Ineq*. The former is the collection of equivalent states, the latter is the set of states which are not equivalent. At the beginning these two sets are initialized noting that final states are never equivalent to nonfinal ones, and that a state is always equivalent to itself. Each couple of states is tested and the two sets (*Def_Equiv*, *Def_Ineq*) are incrementally populated according to function *equiv*. This last is the heart of the whole computational process. *equiv* is a recursive function, it takes two states (p,q) as input and returns true iff p and q are equivalent, false otherwise. At the end of the day states are merged according to *Def_Equiv*.
According to the paper, the repetition in this algorithm can be interrupted and the partially computed *Def_Equiv* can be safely used to merge states.
The pseudocode of function *equiv* is presented here but it will be more clear later in the text when data structures and efficiency are studied. In essence it returns the value of a local variable *eq* whose value changes according to the equivalence or nonequivalence of the input parameters p and q.

Algorithm 2 Equiv

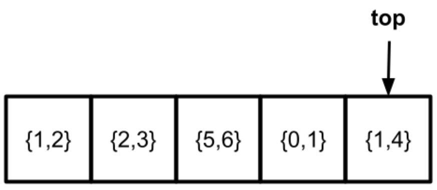
```
func equiv($p, q, k$) {
    if $p = q$ then $eq = TRUE$
    else if $class[p] \neq class[q]$ then $eq = FALSE$
    else if $k = 0$ then $eq = TRUE; rl = 0$
    else if $\{p, q\} \in S$ then $eq = TRUE; rl = index(\{p, q\}, S)$
    else if $\{p, q\} \in Def\_Ineq$ then $eq = FALSE$
    else if $\{p, q\} \in P$ then $eq = TRUE; rl = index(\{p, q\}, P)$
    else {
        if $level = 0 \vee in(p) > 1 \vee in(q) > 1$ {
            $S = S \cup \{\{p, q\}\}$
            $level = level + 1$
            $pushed = true$
        }
        $rl' = |Q|$
        $eq = TRUE$
        for $a : a \in \Gamma_p \cap \Gamma_q$ do {
            $eq = eq \wedge equiv(\delta(p, a), \delta(q, a), k - 1)$
            $rl' = min(rl', rl)$
        }
        $rl = rl'$
        if $pushed$ {
            $S = S \setminus \{\{p, q\}\}$
            $level = level - 1$
        }
        if $eq$ {
            if $rl > level$ {
                $merge(\{p, q\})$
            }
            else {
                $P[rl] = P[rl] \cup \{\{p, q\}\}$
            }
        }
        else {
            $Def\_Ineq = Def\_Ineq \cup \{\{p, q\}\}$
        }
        if $rl = level$ {
            $rl = |Q|$
        }
        if $eq$ {
            if $rl = |Q|$ {
                $\forall_{\{r,s\} \in P[level]} merge(\{r, s\})$
            }
            else {
                $P[rl] = P[rl] \cup P[level]$
            }
        }
        else {
            $\forall_{\{r,s\} \in P[level]} Def\_Ineq = Def\_Ineq \cup \{\{r, s\}\}$
        }
        $P[level] = \emptyset$
    }
    return $eq$
}
```

3 Data structures

The aim of this chapter is to discuss about data structures which have been used in order to implement the algorithm in a proper and efficient way.
C programming language offers huge flexibility in terms of memory handling. Although static allocation may lead to a more efficient execution, dynamic allocation has been preferred in the implementation and in the usage of data structures. Indeed, the allocation of memory of predefined size has not been considered a good choice in this context and could cause waste of memory. On the other hand, dynamic allocation ensures that only the required memory is allocated.
Each data structure used has its own header file and a corresponding .cpp file.

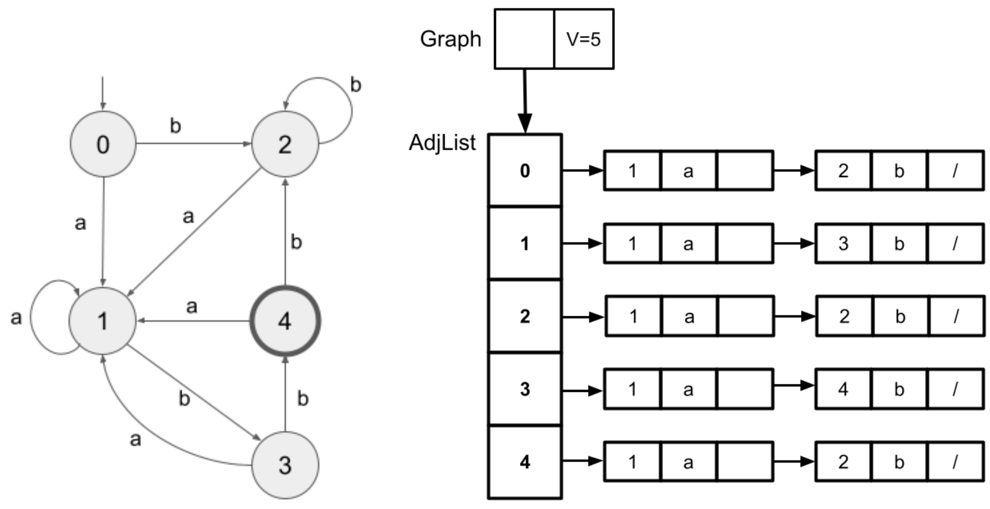
3.1 S - stack of pairs of states [*stack.h*]

As explained in the paper the purpose of variable S is to detect cycles.
The Stack has been implemented as an array. Each element is of type *StatePair* which simply describes a pair of states. Data is pushed and popped accordingly to the value of index *top* which points to the last inserted value in the data structure.



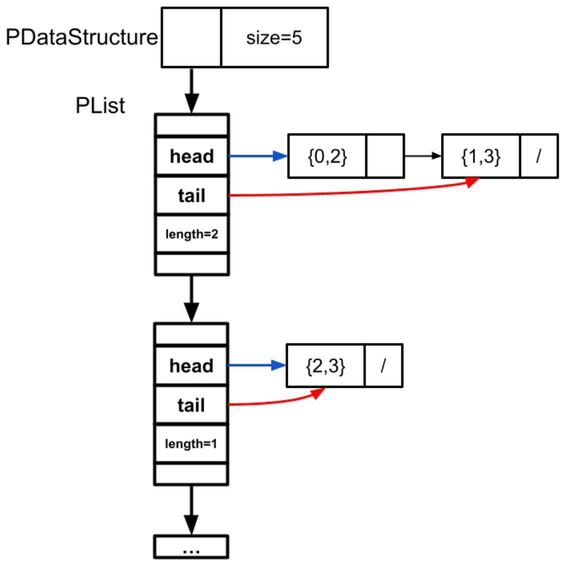
3.2 DFA [*graph.h*]

The Deterministic Finite State automata (*struct Graph*) has been implemented with a sequence of adjacency lists (*struct AdjList*).
Each state (represented by an integer progressive number) has its own list of adjacent nodes (*struct Node*).
Given a node n, an adjacent node s is characterized by a value (*int value*) which denotes state's number, a character c (*char label*) such that $\delta(n, c) = s$, a pointer to the next node in the adjacency list (*Node* next*).
For each adjacent list there is a pointer (*head*) which points the first element of the list. Furthermore at the expense of a heavier memory usage, each adjacency list of a state n has an array attribute *edgeLables* such that $\forall_{c \in \{a,b,c,...z\}} edgeLables[c] = s \iff \delta(n, c) = s$. Here the assumption that possible terminals are only small letters from a to z, is convenient since we can index the array directly with the integer value associated to each character according to language's letter codification.
Remember: I assume that the DFA is characterized by a total transition function; as a consequence there are no ϵ-transitions and each state has one and only one c-transition for every terminal c.



3.3 P - Dependency list [*list.h*]

As explained in the paper, during the recursive process, there can be states whose equivalence can depend upon the current pair of states $\{p, q\}$. They are stored in $P[level]$, where *level* is the recursion level (or the number of items in S) for the current pair. Each level of P has been implemented with a singly linked list of pair of states. For efficiency reasons every list of pair of states maintains a pointer to the head and to the tail of the list.



3.4 Merge–Find Set [*mfset.h*]

At the end of the computational process, states are merged according to the content of *Def_Equiv*. Partitions of equivalent states have been implemented as disjoint sets of a Merge-Find set data structure. The literature suggests different solutions with different computational complexities to implement this data structure. I have decided to use a linked list representation.
Each disjoint set is represented by a linked list. The first object of a list is the master of the corresponding set. Each element of the list contains: a value, a pointer to the following element, a pointer to the list's master.
The data structure interface is defined as follows:

Algorithm 3 Merge-Find Set

```
#Creates $n$ components $\{1\}, ..., \{n\}$
Mfset Mfset(int n)

#Return the master of the partition $x$ belongs to
int find(int x)

#Merge the two components which have nodes $x$ and $y$ as master elements
merge(int x, int y)
```

Typically with this kind of implementations, the operations have the following computational complexity:

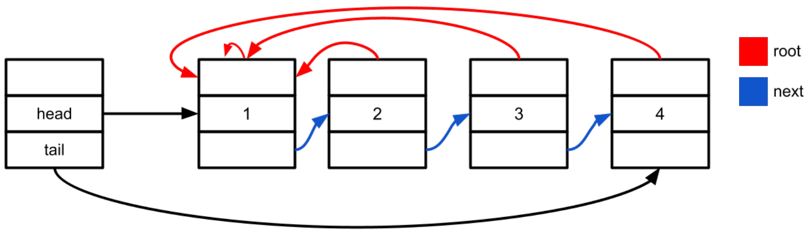
- $find(x)$ - simply read the pointed master node, $O(1)$
- $merge(x, y)$ - the two lists are linked together and then all the pointers to the master node of one of the two lists has to be modified, $O(n^2)$ in the worst case

In order to improve the performances of the operations on the data structure it has been used an heuristic technique. I have decided to adopt the so called weight heuristic:

- Each list keeps track of its own length. The length of the list can be maintained in constant time

- When *merge* procedure is called the shortest list is attached to the longest. In this way there will be less pointers to be updated

It can be shown that with this improvement the computational amortized cost of procedure *merge* becomes $O(logn)$.



4 Reasoning about efficiency and optimizations

The first sketch of the algorithm presented in the paper has a worst-case running time of $O(\Gamma^{|Q|})$, exponential in the number of states. With the following improvements it can be demonstrated that the time complexity of the algorithm is $O(|Q|^2 G(|Q|^2))$, where $G(n)$ is the inverse of Ackermann's function ($G(n) \leq 5$ for all "practical" values of n, i.e. for all $n \leq 2^{2^{16}}$).

4.1 A limit on the recursion depth

From previous papers, it is known that the depth of the recursion can be bounded by $max((|Q| - 2), 0)$ without affecting the result. That is the reason why function *equiv* has a third parameter k: it is used to track the recursion depth.

4.2 Pre-sorting

The first sketch of the algorithm divides states into two classes: final and non-final states. Only pairs of states each belonging to the same class are tested for equivalence. The paper proposes to divide the set of states into more classes. The suggested criterion includes not only the finality of states, but also the number of out-transition, and their labels. However, it has been assumed that the input DFA is characterized by a total transition function and as a consequence all the states in the automaton have the same number of transitions with the same labels, so there is nothing to gain with pre-sorting.

4.3 Rememebering only re-entrant states on stack

It has already been explained in the section about data structures the rule of variable S. Actually, there is no need to add to the stack pairs of states that cannot start a cycle. There are two ways in which those cycles can be started:

- a single chain of transitions may lead from the initial pair back to it – the cycle is started at the initial pair;
- the cycle starts later on, but it can only start from a state that has more than one in-transition; one in-transition leads from a path from a state in the initial pair (outside the cycle), so there must be at least another in-transition from within the cycle.

This control can be seen at line 9 of the pseudocode of *equiv* procedure.

4.4 Memoization

Using memoization as an optimization technique improves the code efficiency significantly.
Each element $a_{i,j}$ of a matrix *memoizationTable* of size $Q \times Q$ is initialized with a special value (-1) to indicate that the corresponding subproblem (*equiv(i,j)*) has not been solved yet. When it is necessary to solve the equivalence of a couple of states p, q, first the algorithm checks the value of $memoizationTable[p][q]$:

- $memoizationTable[p][q] = -1$: the subproblem has not been solved yet, solve the problem and store the result in the table

- $memoizationTable[p][q] = 1$: the equivalence of p, q has already been calculated, they are equivalent
- $memoizationTable[p][q] = 0$: the equivalence of p, q has already been calculated, they are not equivalent

Having choose a two-dimensional array indexed with states numbers allows the storing and the retrieving of this information in constant time.

It is possible to note that final states are never equivalent to nonfinal ones, and that a state is always equivalent to itself. As a result the memoization table is initialized as follows:

$$\forall p, q \mid (p \in Q \setminus F \land q \in F)\ memoizationTable[p][q] = 0$$

$$\forall p, q \mid (p = q)\ memoizationTable[p][q] = 1$$

5 Code usage instructions

5.1 Input

The input must be stored in a file named *input.txt* structured as follows:

- in the first line there are two numbers N, M separated by a white space. These are respectively the number of states ($|Q|$) and the number of edges of the DFA
- in the second row there is a number $s0$ which represents the start state
- in the third row, $cardF$ is the cardinality of the set of final states ($|F|$)
- the next $cardF$ lines list all the final states
- the following $N - cardF$ (i.e. $|Q| - |F|$) lines list all the non-final states
- in the successive line there is $cardA$ which denotes the number of terminals
- the next $cardA$ lines list all the possible labels for the graph's edges
- in the last M lines there are triplets of values $from, to, label$ which indicates that there is a directed edge from node $from$ to node to labeled with terminal $label$, i.e. $\delta(from, label) = to$

input.txt

```
5 10
0
1
4
0
1
2
3
2
a
b
0 1 a
0 2 b
1 1 a
1 3 b
2 1 a
2 2 b
3 1 a
3 4 b
4 1 a
4 2 b
```

Important assumptions:

- Input DFA is characterized by a total transition function so each node has an outgoing edge for each $a \in \Gamma$
- Only lower case letters from a to z are valid candidates for edge labels
- States must be represented as a sequence of consecutive natural numbers starting from 0 (which of course is not necessarily the start state)
- Execution time of the algorithm is reasonable for DFA with $\sim 10^3$ nodes. When the order of magnitude is $\sim 10^4$ the execution is dramatically slower. This is mainly due to the data structure initialization process, the recursive mechanism, the fact that the input DFA has total transition function which leads to a very dense graph

5.2 Output

The output of the computational process is stored in *output.txt*.

output.txt

```
PARTITION[0] - SIZE=1
{
1
}
PARTITION[1] - SIZE=2
{
2
0
}
PARTITION[2] - SIZE=1
{
3
}
PARTITION[3] - SIZE=1
{
4
}
```

The file lists the sets of equivalent states of the final minimized DFA.
In the example states number 0 and 2 are equivalent.

5.3 Compilation

For the sake of simplicity in the project folder (*PaperImplementation*) there is a *Makefile*. As a result to compile the program it is sufficient to type:

```
$ make
```

Eventually it is possible to remove unnecessary *.o* files produced during the compilation with:

```
$ make clean
```

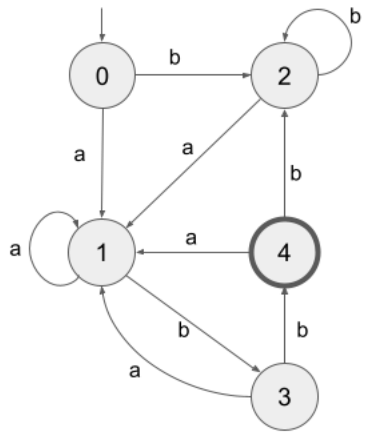
The code is compiled using the GNU Compiler Collection (gcc) compiler.

After that, the code can be executed with:

```
$ ./main.out
```

5.4 Testing

5.4.1 Example 1



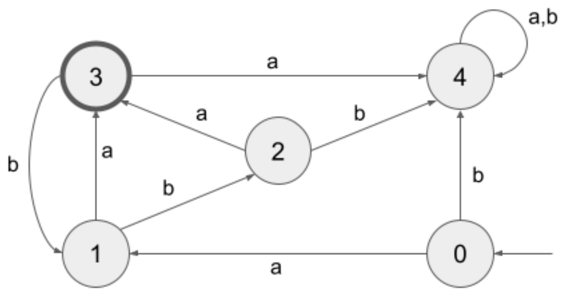
input.txt

```
5 10
0
1
4
0
1
2
3
2
a
b
0 1 a
0 2 b
1 1 a
1 3 b
2 1 a
2 2 b
3 1 a
3 4 b
4 1 a
4 2 b
```

output.txt

```
PARTITION[0] - SIZE=1
{
1
}
PARTITION[1] - SIZE=2
{
2
0
}
PARTITION[2] - SIZE=1
{
3
}
PARTITION[3] - SIZE=1
{
4
}
```

5.4.2 Example 2



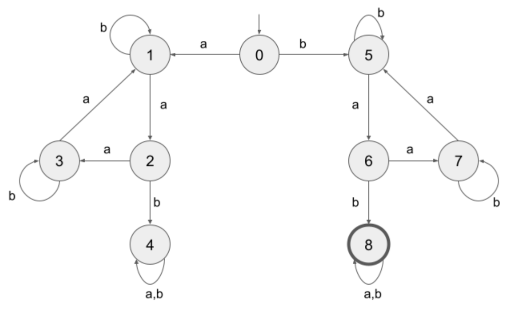
input.txt

```
5 10
0
1
3
0
1
2
4
2
a
b
0 1 a
0 4 b
1 3 a
1 2 b
2 3 a
2 4 b
3 4 a
3 1 b
4 4 a
4 4 b
```

output.txt

```
PARTITION[0] - SIZE=1
{
0
}
PARTITION[1] - SIZE=1
{
1
}
PARTITION[2] - SIZE=1
{
2
}
PARTITION[3] - SIZE=1
{
3
}
PARTITION[4] - SIZE=1
{
4
}
```

5.4.3 Example 3



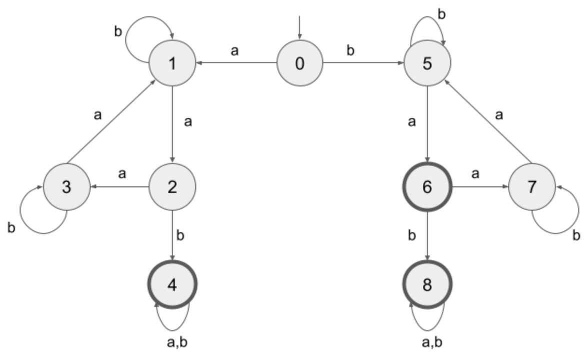
input.txt

```
9 18
0
1
8
0
1
2
3
4
5
6
7
2
a
b
0 1 a
0 5 b
1 2 a
1 1 b
2 3 a
2 4 b
3 1 a
3 3 b
4 4 a
4 4 b
5 6 a
5 5 b
6 7 a
6 8 b
7 5 a
7 7 b
8 8 a
8 8 b
```

output.txt

```
PARTITION[0] - SIZE=1
{
0
}
PARTITION[1] - SIZE=4
{
2
1
3
4
}
PARTITION[2] - SIZE=1
{
5
}
PARTITION[3] - SIZE=1
{
6
}
PARTITION[4] - SIZE=1
{
7
}
PARTITION[5] - SIZE=1
{
8
}
```

5.4.4 Example 4



input.txt

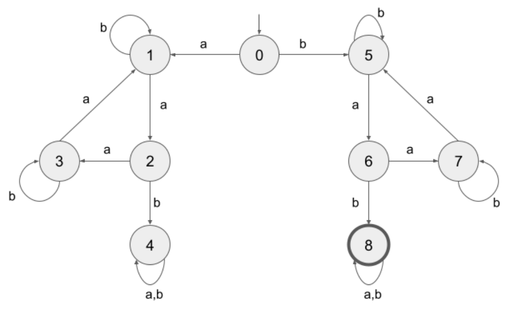
```
9 18
0
3
4
6
8
0
1
2
3
5
7
2
a
b
0 1 a
0 5 b
1 2 a
1 1 b
2 3 a
2 4 b
3 1 a
3 3 b
4 4 a
4 4 b
5 6 a
5 5 b
6 7 a
6 8 b
7 5 a
7 7 b
8 8 a
8 8 b
```

output.txt

```
PARTITION[0] - SIZE=1
{
0
}
PARTITION[1] - SIZE=1
{
1
}
PARTITION[2] - SIZE=1
{
2
}
PARTITION[3] - SIZE=1
{
3
}
PARTITION[4] - SIZE=1
{
5
}
PARTITION[5] - SIZE=1
{
6
}
PARTITION[6] - SIZE=1
{
7
}
PARTITION[7] - SIZE=2
{
8
4
}
```

6 Partition Refinement Algorithm

[Folder: slideImplementation]
When the automaton has got a lot of nodes, it is difficult to check the output correctness by hand. Therefore, I have implemented the *"Partition Refinement"* DFA minimization algorithm. The algorithm has been implemented to reproduce the one explained during the course, so its general characteristics won't be discussed in this document. With the purpose of exposing my implementation in an efficient and short way I am going to propose an execution example of the code:



Partitions are stored in a list of linked lists named *struct Partition** partitionList*. Given a state it is often needed to know the partition it belongs to. This operation is really computational demanding if it is implemented as a search through the elements of *partitionList*. Hence, as a solution to this problem, an array named *class* is adopted, such that: $\forall s \in Q \; class[s] = c$ means that state s belongs to partition number c.

At the beginning there are two disjoint partitions separating final and non-final states:

Partitions:
0: $\underline{8}$
1: $\underline{7}$ 6 5 4 3 2 1 0

class[0]=1
class[1]=1
class[2]=1
class[3]=1
class[4]=1
class[5]=1
class[6]=1
class[7]=1
class[8]=0

Each partition has its own master state (underlined) which dictates the "behaviour" of the partition. In essence in the partition will remain only the states that are equivalent to the master.
The algorithm compares the master transition function with the one of all the states in its partition. In this case it is possible to observe that $\delta(7, b) \in 1$ whereas $\delta(6, b) \in 0$. Hence state 6 and 7 are not equivalent. As a result, 6 is moved in a new partition along with all the other states s such that $\delta(s, b) \in 0$.

Partitions:
0: $\underline{8}$
1: $\underline{7}$ 5 4 3 2 1 0
2: $\underline{6}$

class[0]=1
class[1]=1
class[2]=1
class[3]=1
class[4]=1
class[5]=1
class[6]=2
class[7]=1
class[8]=0

The algorithm keeps to analize iteratively all the partitions in this way until it is possible to split some of them (canSplit==true in the code). Obviously, for partitions with a single element no operation is needed.

$\delta(7, a) \in 1$ whereas $\delta(5, a) \in 2$

Partitions:
0: $\underline{8}$
1: $\underline{7}$ 4 3 2 1 0
2: $\underline{6}$
3: $\underline{5}$

class[0]=1
class[1]=1
class[2]=1
class[3]=1
class[4]=1
class[5]=3
class[6]=2
class[7]=1
class[8]=0

$\delta(7, a) \in 3$ whereas $\delta(4, a) \in 1$

Partitions:
0: $\underline{8}$
1: $\underline{7}$
2: $\underline{6}$
3: $\underline{5}$
4: $\underline{0}$ 1 2 3 4

class[0]=4
class[1]=4
class[2]=4
class[3]=4
class[4]=4
class[5]=3
class[6]=2
class[7]=1
class[8]=0

$\delta(0, b) \in 3$ whereas $\delta(1, b) \in 4$

Partitions:
0: $\underline{8}$
1: $\underline{7}$
2: $\underline{6}$
3: $\underline{5}$
4: $\underline{0}$
5: $\underline{4}$ 1 2 3

class[0]=4
class[1]=5
class[2]=5
class[3]=5
class[4]=5
class[5]=3
class[6]=2
class[7]=1
class[8]=0

END

6.1 Code usage instructions

For the sake of simplicity the input structure, the output structure, the compilation procedure, are exactly the same which have been described for the Watson-Daciuk algorithm implementation. This time the code is located in "*slideImplementation*" folder.