DFA Operations

Complement, Product, Union, Intersection, Difference, Equivalence and Minimization of DFAs

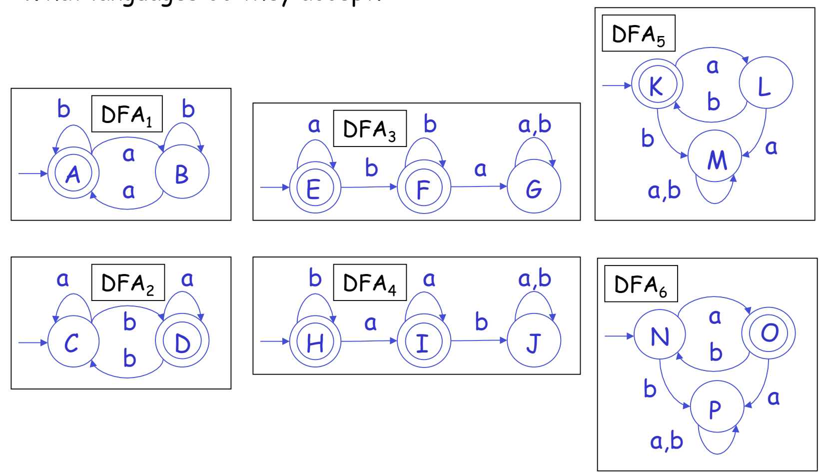
Wednesday, September 28, 2011
Reading: Sipser pp. 45-46, Stoughton 3.11 – 3.12

CS235 Languages and Automata

Department of Computer Science
Wellesley College

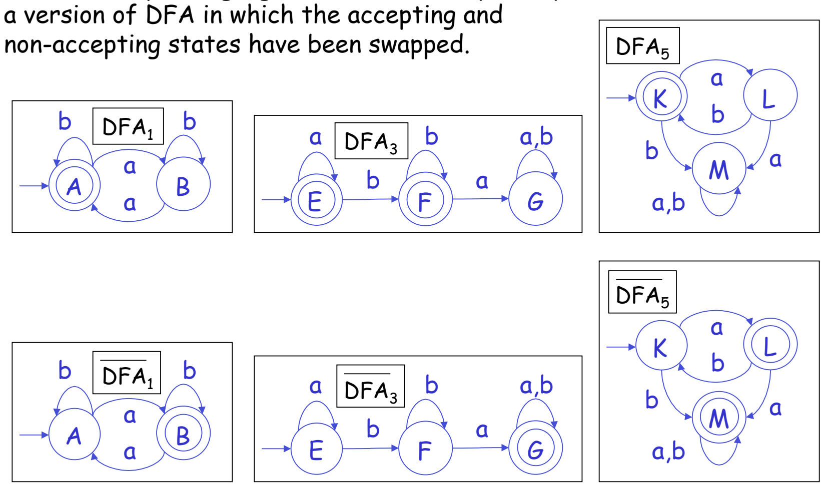
Some DFAs

Here are some simple DFAs we will use as examples in today's lecture. What languages do they accept?



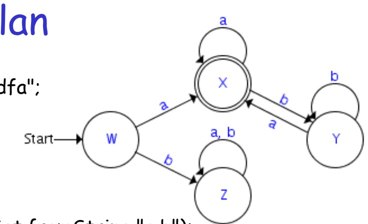
Complement of DFAs

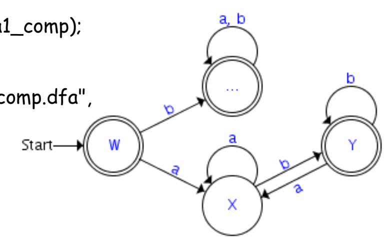
If DFA accepts language L, then $\overline{L}$ is accepted by $\overline{\text{DFA}}$, a version of DFA in which the accepting and non-accepting states have been swapped.



DFA Complement in Forlan

```
- val dfa1 = DFA.input "begin_and_end_with_a.dfa";
val dfa1 = - : dfa

- DFA.complement;
val it = fn : dfa * sym set -> dfa

- val dfa1_comp = DFA.complement (dfa1, SymSet.fromString "a,b");
val dfa1_comp = - : dfa

- SymSet.toString (DFA.states dfa1_comp);
val it = "W, X, Y, <dead>" : string

- SymSet.toString (DFA.acceptingStates dfa1_comp);
val it = "W, Y, <dead>" : string

- DFA.output ("dfa_begin_and_end_with_a_comp.dfa",
              dfa1_comp);
val it = () : unit
```





Product of DFAs

We can run two DFAs in parallel on the same input via the **product construction**, as long as they share the same alphabet.

Suppose $DFA_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $DFA_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$

We define $DFA_1 \times DFA_2$ as follows:

States: $Q_{1x2} = Q_1 \times Q_2$

Alphabet: Σ

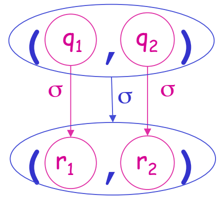
Transitions:

$\delta_{1x2} \in Q_{1x2} \times \Sigma \rightarrow Q_{1x2}$

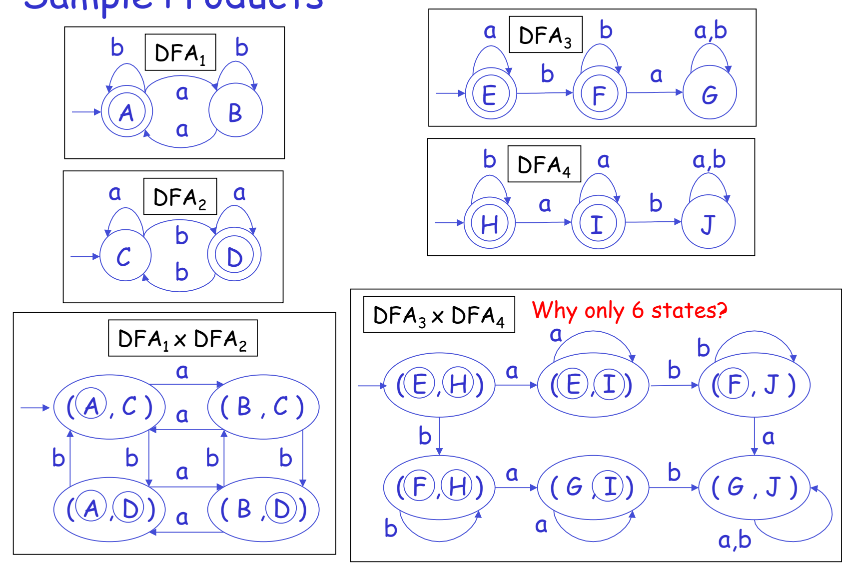
$\delta_{1x2}\,(\,((q_1,q_2), \sigma)\,)$
$= (\,\delta_1(\,(q_1,\sigma)\,), \delta_2(\,(q_2,\sigma)\,)\,)$

Start State: $s_{1x2} = (s_1, s_2)$

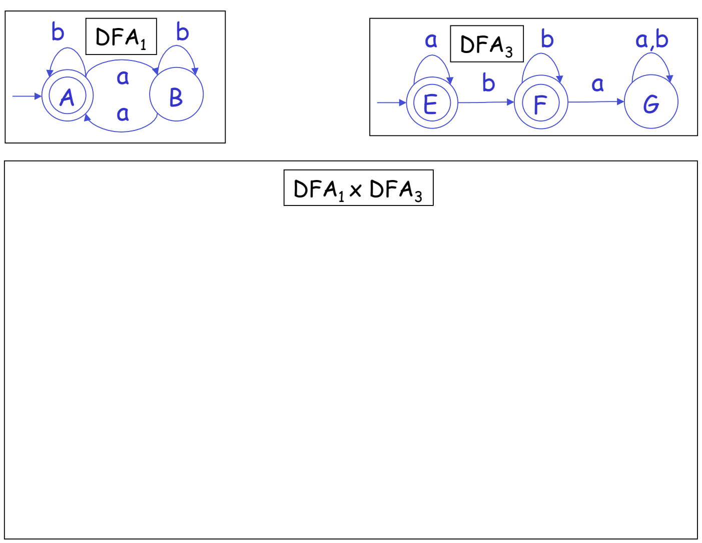
Final States: Definition depends on how we use product



Sample Products

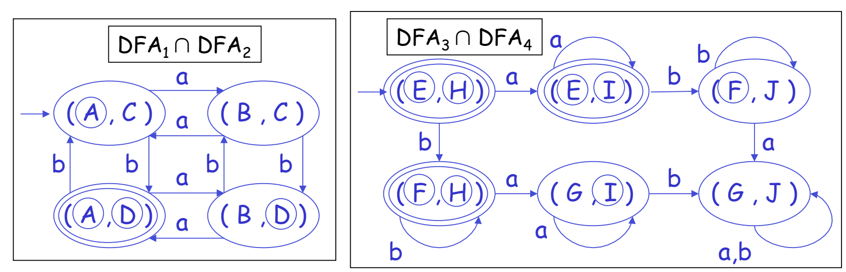


Practice



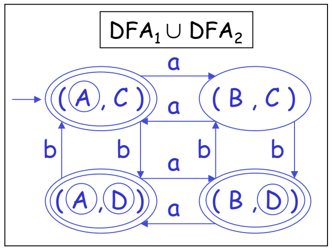
Intersection of DFAs

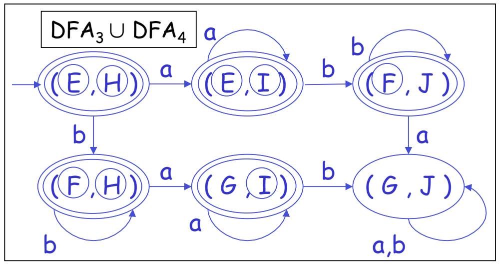
We can intersect DFA_1 and DFA_2 (written $DFA_1 \cap DFA_2$) by defining the accepting states of $DFA_1 \times DFA_2$ as those state pairs in which **both** states are final states of their DFAs.



Union of DFAs

We can union DFA_1 and DFA_2 (written $DFA_1 \cup DFA_2$) by defining the accepting states of $DFA_1 \times DFA_2$ as those state pairs in which **either** state is a final state of its DFA.



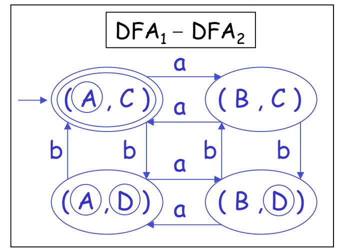


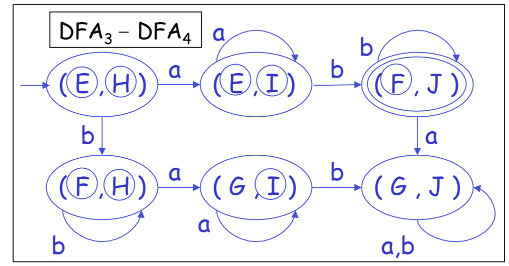
Difference of DFAs

The difference of two DFAs (written $DFA_1 - DFA_2$) can be defined in terms of complement and intersection:

$$DFA_1 - DFA_2 = DFA_1 \cap \overline{DFA_2}$$

So we can take the difference of DFA_1 and by defining the final states of $DFA_1 - DFA_2$ as those state pairs in which the first state is final in DFA_1 and the is second state is not final in DFA_2.





What is a Closure Property?

A set S is closed under an n-ary operation f iff $x_1, \ldots, x_n \in S$ implies $f(x_1, \ldots, x_n) \in S$

Examples:

- Bool is closed under negation, conjunction, disjunction.
- Nat is closed under + and * but not – and /.
- Int is closed under +, *, and -, but not /.
- Rat is closed under +, *, -, and / (except division by 0).

Some Closure Properties of Regular Languages

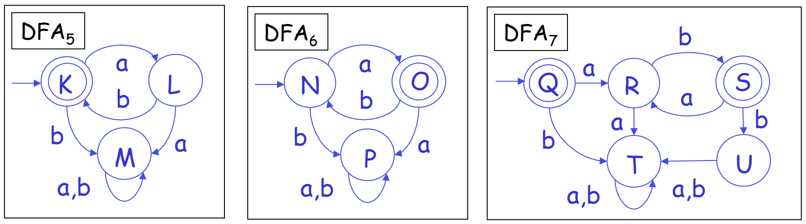
Recall that a language is regular iff there is a DFA that accepts it.

Based on the previous DFA constructions, we know the following **closure properties** of regular languages.

Suppose L_1 and L_2 are regular languages. Then:

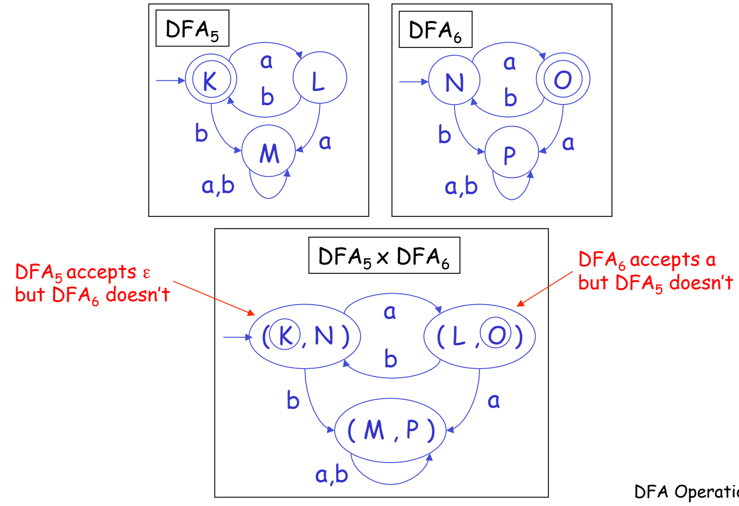
- $\overline{L_1}$ and $\overline{L_2}$ are regular;
- $L_1 \cup L_2$ is regular;
- $L_1 \cap L_2$ is regular;
- $L_1 - L_2$ and $L_2 - L_1$ are regular.

Are Any of the Following DFAs Equivalent?



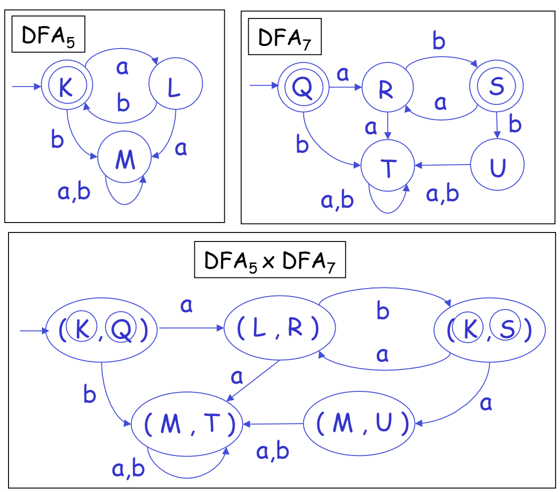
DFA_5 and DFA_6 are Not Equivalent

Look at their product!



DFA_5 and DFA_7 Are Equivalent

Look at their product!



DFA Equivalence Algorithm

To determine if DFA_1 and DFA_2 are equivalent, construct $DFA_1 \times DFA_2$ and examine all state pairs containing at least one accepting state from DFA_1 or DFA_2:

- If in all such pairs, both components are accepting, DFA_1 and DFA_2 are equivalent --- i.e., they accept the same language.
- If in all such pairs, the first component is accepting but in some the second is not, the language of DFA_1 is a **superset** of the language of DFA_2 and it is easy to find a string accepted by DFA_1 and not by DFA_2
- If in all such pairs, the second component is accepting but in some the first is not, the language of DFA_1 is a **subset** of the language of DFA_2, and it is easy to find a string accepted by DFA_2 and not by DFA_1
- If none of the above cases holds, the languages of DFA_1 and DFA_2 are unrelated, and it is easy to find a string accepted by one and not the other.

Products in Forlan

val inter : dfa * dfa -> dfa

val minus : dfa * dfa -> dfa

datatype relationship

= Equal | Incomp of str * str | ProperSub of str | ProperSup of str

val relation : dfa * dfa -> relationship

val relationship : dfa * dfa -> unit

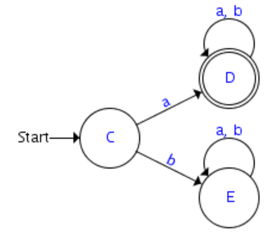
val subset : dfa * dfa -> bool

val equivalent : dfa * dfa -> bool

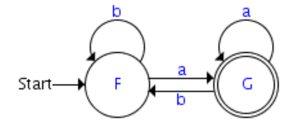
Note that a union operator is missing. It really should be there!
We'll see later how it can be defined.

Forlan Products: Example 1

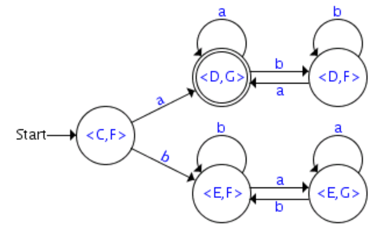
```
- val bwa = DFA.input "begin_with_a.dfa";
val bwa = - : dfa
```




```
- val ewa = DFA.input "end_with_a.dfa";
val ewa = - : dfa
```

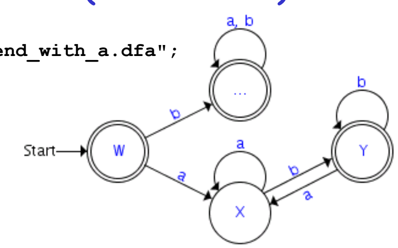



```
- val baewa = DFA.inter(bwa,ewa);
val baewa = - : dfa

- DFA.output("baewa.dfa", baewa);
val it = () : unit
```



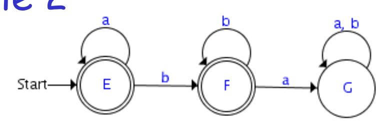
Forlan Products: Example 1 (Continued)

```
- val dfa1 = DFA.input "begin_and_end_with_a.dfa";
val dfa1 = - : dfa

- DFA.relationship(baewa, dfa1);
languages are equal
val it = () : unit

- DFA.relation(baewa, dfa1);
val it = Equal : DFA.relationship

- DFA.relation(bwa, baewa);
val it = ProperSup [-,-] : DFA.relationship

- let val DFA.ProperSup s = it in Str.toString s end;
stdIn:19.9-19.29 Warning: binding not exhaustive
          ProperSup s = ...
val it = "ab" : string

- DFA.subset(baewa, bwa);
val it = true : bool
```

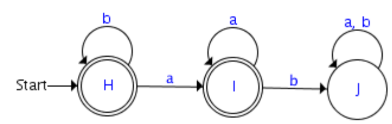


Forlan Products: Example 2

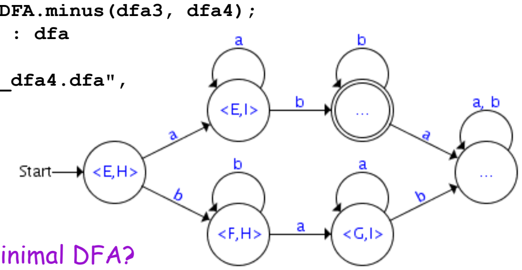
```
- val dfa3 = DFA.input "dfa3.dfa";
-val dfa3 = - : dfa
```




```
- val dfa4 = DFA.input "dfa4.dfa";
-val dfa4 = - : dfa
```

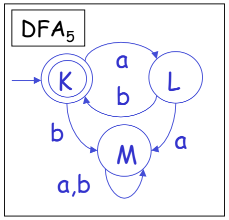


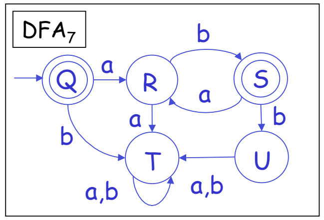

```
- val dfa3_minus_dfa4 = DFA.minus(dfa3, dfa4);
-val dfa3_minus_dfa4 = - : dfa

- DFA.output("dfa3_minus_dfa4.dfa",
dfa3_minus_dfa4);
- val it = () : unit
```



Is this a minimal DFA?

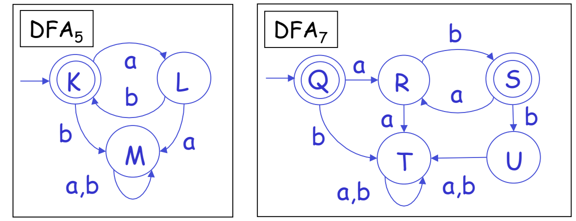
Minimal DFAs





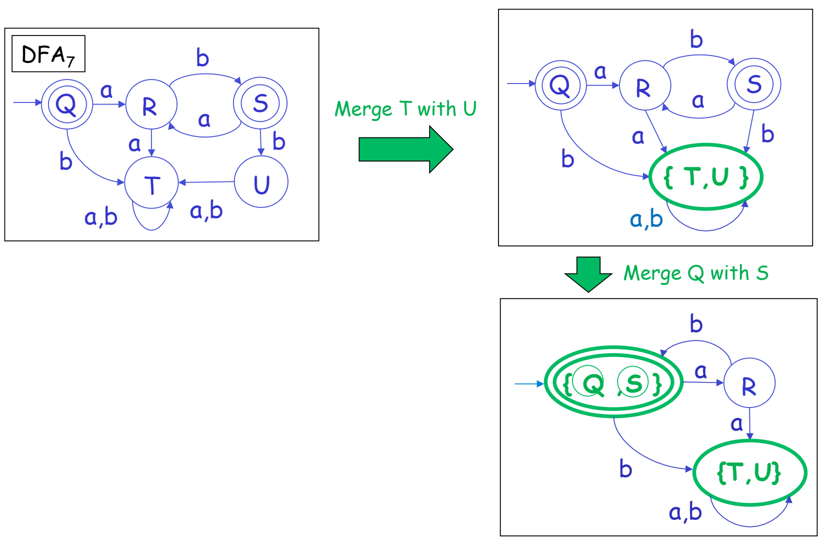
- A DFA is minimal if it has the smallest number of states of any DFA accepting its language.
- Is DFA_5 minimal?
- Is DFA_7 minimal?

State Merging

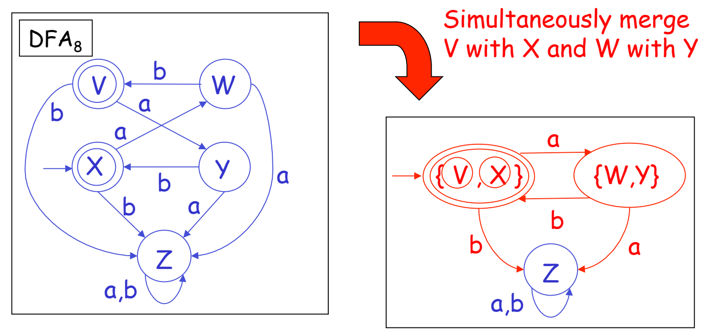


- A DFA is not minimal iff two states can be merged to form a single state without changing the meaning of the DFA.
- Final states and non-final states can never be merged.
- Can merge two states iff for each symbol they transition to mergeable states.
- Which states in DFA_7 can be merged?

State Merging in DFA_7



Problem: States Can't Always be Merged Iteratively



Key to solution: rather than iterating to find *mergeable* state pairs, iterate to find all state pairs that are provably *unmergeable*. Then any remaining state pair is mergeable.

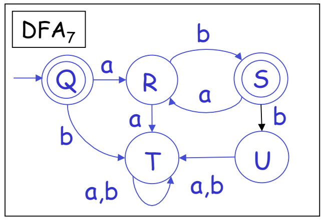
This is an example of a **greatest fixed point iteration**, in which items are assumed related unless proven otherwise.

DFA Minimization Algorithm: Step 1

List all pairs of states than **must not** be merged = pairs of one final and one non-final state.

Other pairs **might** be mergeable; they are considered mergeable until proven otherwise.

It's a good idea to keep track of state pairs in half of a table*:



	U	T	S	R
Q	U_ε	U_ε	?	U_ε
R	?	?	U_ε	
S	U_ε	U_ε		
T	?			

Table$_1$

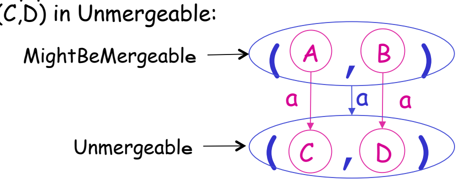
U_s Unmergeable by string s

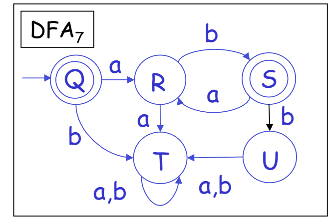
? MightBeMergeable

* **Lyn adapted this table representation from Katie Sullivan (Olin) and the subscripted Unmergeability from Anna Loparev (Wellesley)**

DFA Minimization Algorithm: Step 2

Change from MightBeMergeable to Unmergeable any pair (A,B) such that there is a transition to a (C,D) in Unmergeable:





Repeat this step until no more state pairs can be changed.

	U	T	S	R
Q	U_ε	U_ε	?	U_ε
R	?	?	U_ε	
S	U_ε	U_ε		
T	?			

Table$_1$

In Table$_1$, pairs (R,T) and (R,U) be changed:

$(R,T) \xrightarrow{b} (S,T)$

$(R,U) \xrightarrow{b} (S,T)$

	U	T	S	R
Q	U_ε	U_ε	?	U_ε
R	U_b	U_b	U_ε	
S	U_ε	U_ε		
T	?			

Table$_2$

In Table$_2$, no pairs can be changed

DFA Minimization Algorithm: Step 3

When no more pairs can be changed from MightBeMergeable to Unmergeable, merge the pairs remaining in MightBeMergeable.

	U	T	S	R
Q	U_ε	U_ε	?	U_ε
R	U_b	U_b	U_ε	
S	U_ε	U_ε		
T	?			

Table$_2$

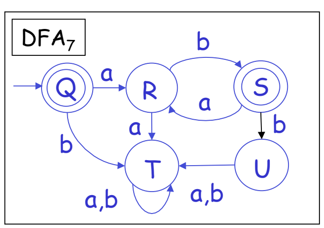
	U	T	S	R
Q	U_ε	U_ε	M	U_ε
R	U_b	U_b	U_ε	
S	U_ε	U_ε		
T	M			

Table$_2$

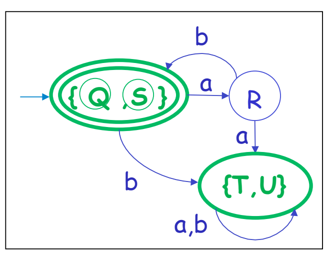
U_s Unmergeable by s

? MightBeMergeable

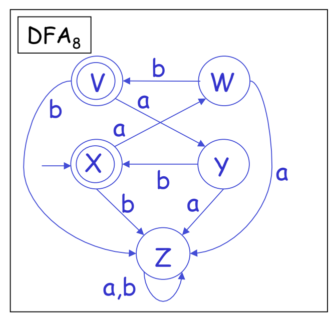
M Mergeable



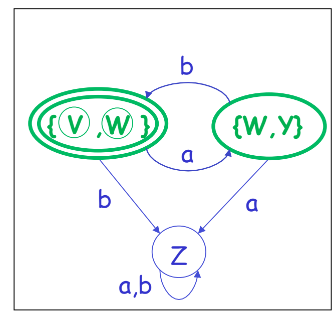
Merge Q with S and T with U



DFA Minimization: More Practice



Merge V with X and W with Y



	Z	Y	X	W
V	U_ε	U_ε	?	U_ε
W	?	?	U_ε	
X	U_ε	U_ε		
Y	?			

Table$_1$

	Z	Y	X	W
V	U_ε	U_ε	?	U_ε
W	U_b	?	U_ε	
X	U_ε	U_ε		
Y	U_b			

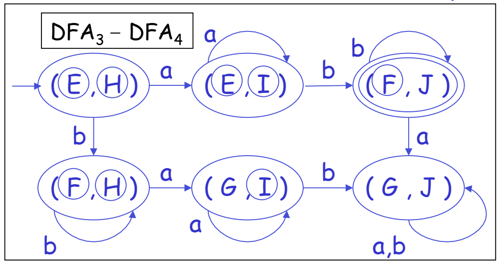
Table$_2$

	Z	Y	X	W
V	U_ε	U_ε	M	U_ε
W	U_b	M	U_ε	
X	U_ε	U_ε		
Y	U_b			

Table$_3$

Both examples happen to converge after 1 iteration of step 2, but in general can take 0 to (|Q|-1) iterations.

DFA Minimization: One more example

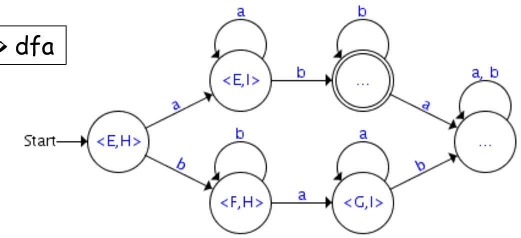


	FJ	GJ	GI	FH	EI
EH	U_ε				
EI	U_ε				
FH	U_ε				
GI	U_ε				
GJ					

Minimization in Forlan

val minimize : dfa -> dfa

Example:




```
-val dfa3_minus_dfa4_min = DFA.minimize(dfa3_minus_dfa4);
val dfa3_minus_dfa4_min = - : dfa

- DFA.output("dfa3_minus_dfa4_min.dfa", dfa3_minus_dfa4_min);
val it = () : unit
```