Language Understanding Systems

(Weighted) Finite State Transducers

Evgeny A. Stepanov

SISL, DISI, UniTN & VUI, Inc.
evgeny.stepanov@unitn.it

Outline

1. Finite State Acceptors & Transducers
2. FSA/FST Operations
3. FST Operations
4. Text to FSM
5. FSA/FST Exercises

Section 1

Finite State Acceptors & Transducers

Finite State Transducers

Acceptor

- FSM with 1 tape: input
- accepts/recognizes strings

Transducer

- FSM with 2 tapes: input & output
- translates input to output string

FSA File Format

FSM: A.txt

from_state	*to_state*	*input_symbol*	(*weight*)
0	0	red	0.5
0	1	green	0.3
1	2	blue	0.0
1	2	yellow	0.6
2	0.8		

Lexicon/Symbol File: A.lex

<eps>	0
red	1
green	2
blue	3
yellow	4

FST File Format

FSM: A.txt

from_state to_state input_symbol output_symbol (weight)

```
0   0     red      yellow   0.5
0   1     green    blue     0.3
1   2     blue     green    0.0
1   2     yellow   red      0.6
2   0.8
```

Lexicon/Symbol File: A.lex

```
<eps>     0
red       1
green     2
blue      3
yellow    4
```

Basic Commands: FSA

Compilation

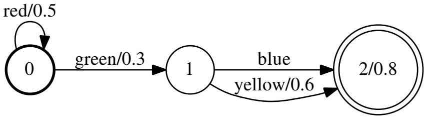
```
fstcompile --acceptor --isymbols=A.lex A.txt > A.fsa
```

Printing

```
fstprint --acceptor --isymbols=A.lex A.fsa
```

Drawing

```
fstdraw --acceptor --isymbols=A.lex A.fsa |
dot -Tpng > A.png
```



Basic Commands: FST

Compilation

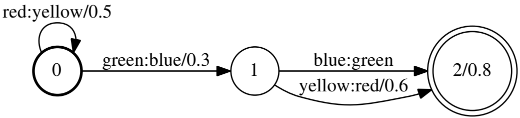
```
fstcompile --isymbols=A.lex --osymbols=A.lex A.txt >
A.fst
```

Printing

```
fstprint --isymbols=A.lex --osymbols=A.lex A.fst
```

Drawing

```
fstdraw --isymbols=A.lex --osymbols=A.lex A.fst |
dot -Tpng > A.png
```



Section 2

FSA/FST Operations

Regular Expressions

- $\emptyset$ is a Regular Expression
- Each symbol from Σ is a Regular Expression
- If α and β are Regular Expressions, then so is $(\alpha \circ \beta)$
- If α and β are Regular Expressions, then so is $(\alpha \cup \beta)$
- If α is a Regular Expression, then so is (α^*)

Languages expressed using Regular Expressions are called Regular Languages

Closure Properties of Regular Languages

RL are closed under following operations:

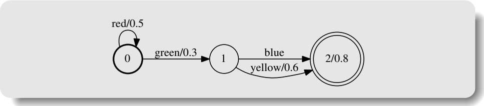
- **Intersection**: if L_1 and L_2 are RL, then so is $L_1 \cap L_2$, the language consisting of the set of strings in both L_1 and L_2
- **Difference**: if L_1 and L_2 are RL, then so is $L_1 - L_2$, the language consisting of the set of strings in both L_1 and not in L_2
- **Complementation**: if L_1 is a RL, then so is its complement $\bar{L_1}$
- **Reversal**: if L_1 is a RL, then so is L_1^R, the set of reversals of all strings in L_1

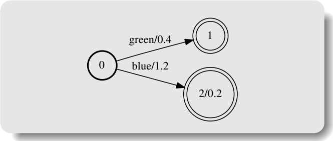
FSA Operations

Operation	Implementation
Concatenation (Product)	`fstconcat`
Union (Sum)	`fstunion`
Kleene*	`fstclosure`
Intersection	`fstintersect`
Difference	`fstdifference`
Reversal	`fstreverse`
Complement	N/A

Concatenation (Product)

- Equation: $C = AB$
- Command: `fstconcat A.fsa B.fsa > C.fsa`

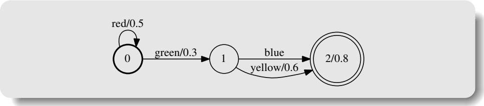


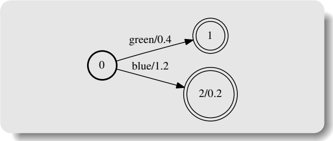


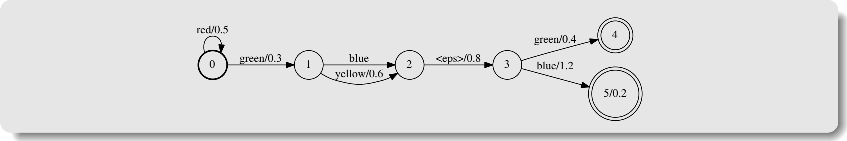
Signals & Interactive Systems

Concatenation (Product)

- Equation: $C = AB$
- Command: `fstconcat A.fsa B.fsa > C.fsa`

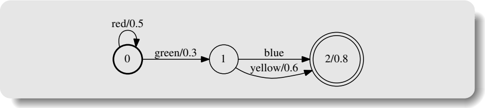


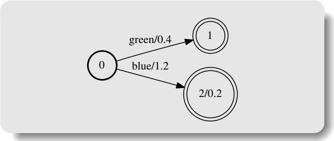




Union (Sum)

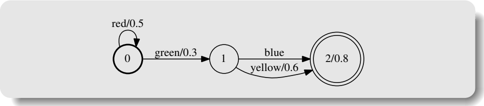
- Equation: $C = A \cup B$ $(C = A + B)$
- Command: `fstunion A.fsa B.fsa > C.fsa`

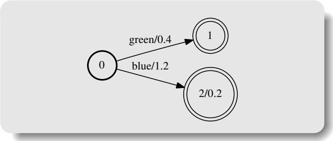


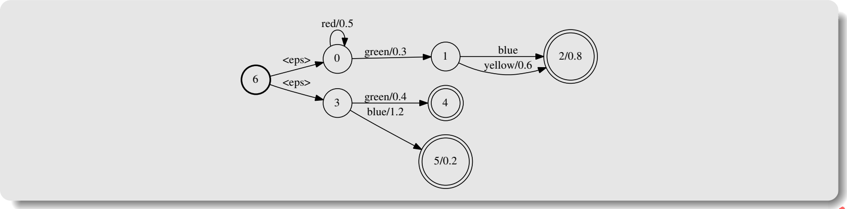


Union (Sum)

- Equation: $C = A \cup B$ $(C = A + B)$
- Command: `fstunion A.fsa B.fsa > C.fsa`



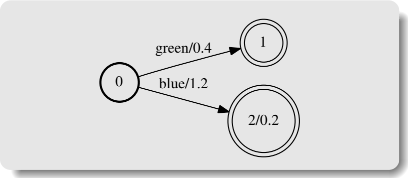




Signals & Interactive Systems

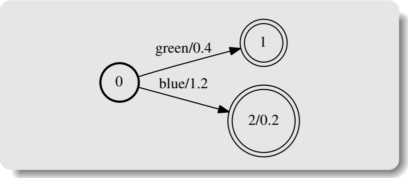
(Concatenative) Closure

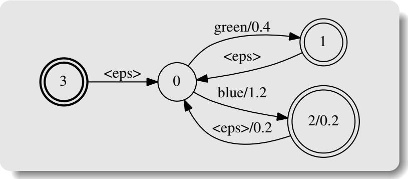
- Equation: $C = B^{*} = B^{0} + B^{1} + B^{2} + \ldots$
- Command: `fstclosure B.fsa > C.fsa`



(Concatenative) Closure

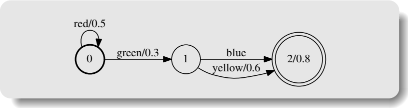
- Equation: $C = B^* = B^0 + B^1 + B^2 + \dots$
- Command: `fstclosure B.fsa > C.fsa`

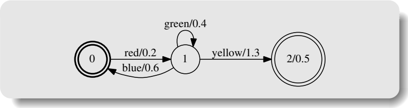




Intersection

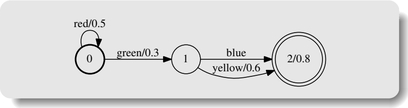
- Equation: $C = A \cap B$
- Command: `fstintersect A.fsa B.fsa > C.fsa`

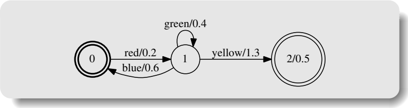


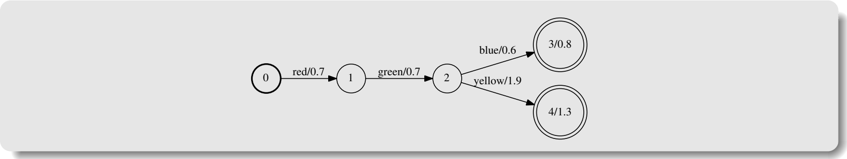


Intersection

- Equation: $C = A \cap B$
- Command: `fstintersect A.fsa B.fsa > C.fsa`

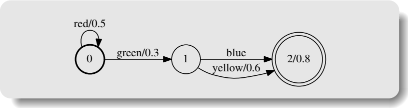


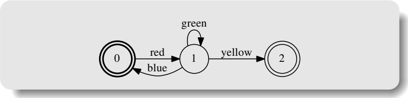




Difference

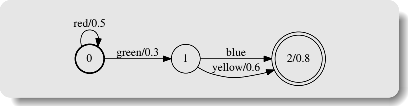
- Equation: $C = A - B$: B – Unweighted & Deterministic
- Command: **`fstdifference A.fsa B.fsa > C.fsa`**

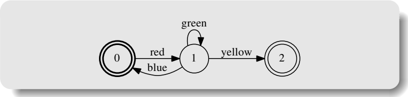


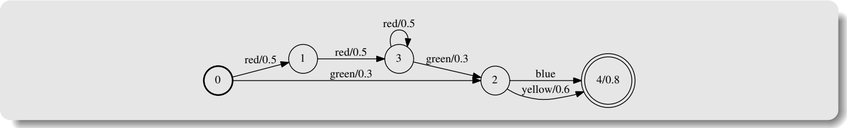


Difference

- Equation: $C = A - B$: B – Unweighted & Deterministic
- Command: `fstdifference A.fsa B.fsa > C.fsa`

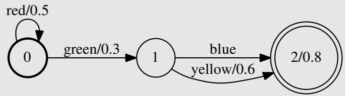






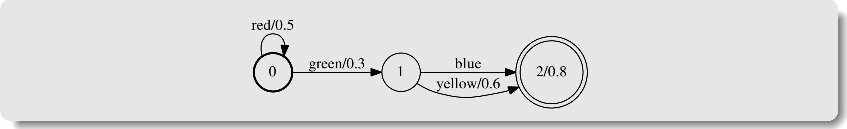
Reversal

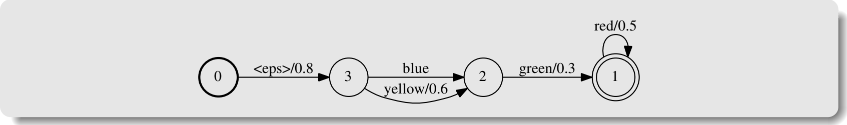
- Equation: $C = A^R$
- Command: `fstreverse A.fsa > C.fsa`



Reversal

- Equation: $C = A^R$
- Command: `fstreverse A.fsa > C.fsa`





Complement

- Equation: $C = A^{-1}$
- Command: **N/A**

Complement

- Equation: $C = A^{-1}$
- Command: **N/A**

WHY?

Complement

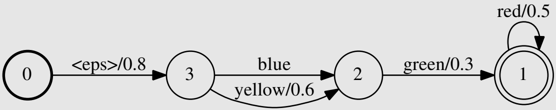
- Equation: $C = A^{-1}$
- Command: **N/A**

WHY?

- Alphabet issue
- Non-determinism issue

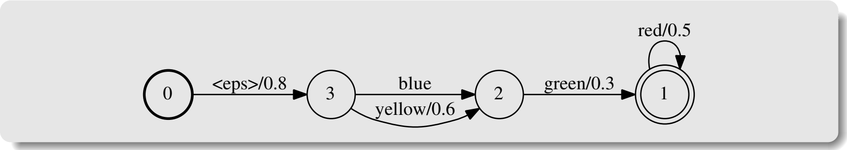
Epsilon Removal

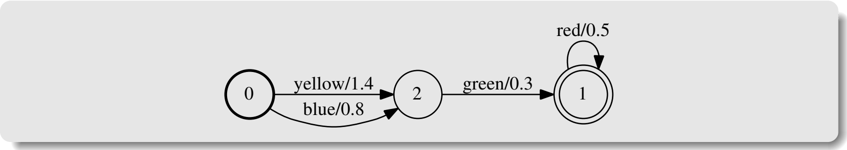
- Command: `fstrmepsilon A.fsa > C.fsa`



Epsilon Removal

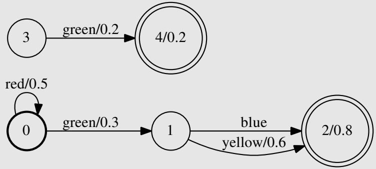
- Command: `fstrmepsilon A.fsa > C.fsa`





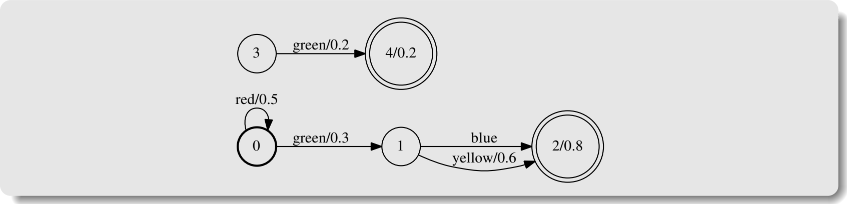
Trimming: Remove ‘unreachable’ states

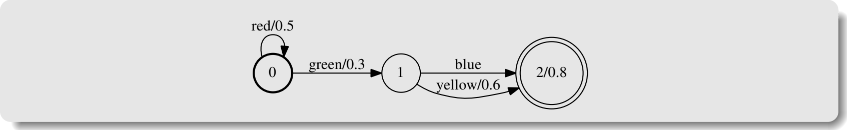
- Command: `fstconnect A.fsa > C.fsa`



Trimming: Remove 'unreachable' states

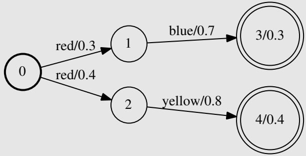
- Command: `fstconnect A.fsa > C.fsa`





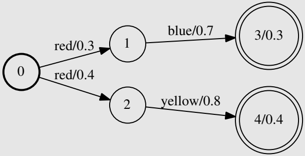
Determinization

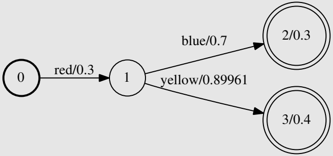
- Command: `fstdeterminize A.fsa > C.fsa`



Determinization

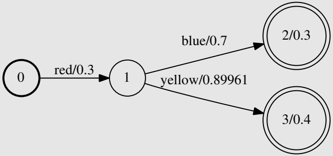
- Command: `fstdeterminize A.fsa > C.fsa`





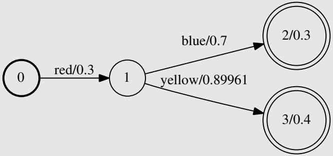
Minimization

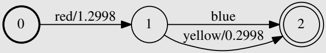
- Command: `fstminimize A.fsa > C.fsa`
- returns the minimal deterministic FSM equivalent to the input FSM, which must be a deterministic acceptor. Epsilon arcs are treated the same as other symbols.



Minimization

- Command: `fstminimize A.fsa > C.fsa`
- returns the minimal deterministic FSM equivalent to the input FSM, which must be a deterministic acceptor. Epsilon arcs are treated the same as other symbols.





FSM Utilities

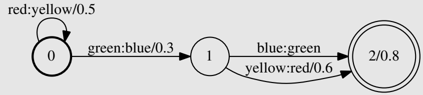
- **fstarcsort** sorts the arcs in an FSM per state. Some operations depend on the FSM arcs being sorted. It is a good idea to always sort compiled FSMs prior to working on them.
- **fsttopsort** sorts an FSM so that all transitions are from lower to higher state IDs. It is useful to apply it before printing.
- **fstinfo** prints out information about an FSM.

Section 3

FST Operations

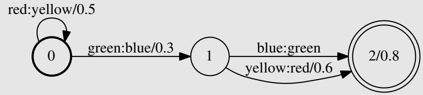
Projection

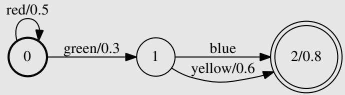
- Equation: $A = \pi_1(T)$
- Command: `fstproject A.fst > A.fsa`
- converts a *transducer* into an *acceptor* by retaining only the input or output (with `--project_output`) label on each transition



Projection

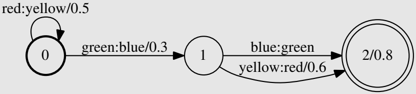
- Equation: $A = \pi_1(T)$
- Command: **`fstproject A.fst > A.fsa`**
- converts a *transducer* into an *acceptor* by retaining only the input or output (with `--project_output`) label on each transition





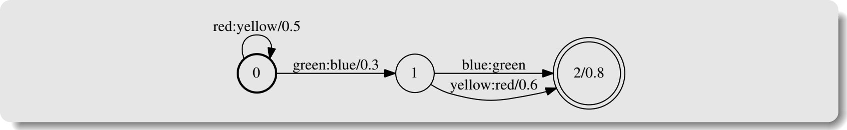
Inverse

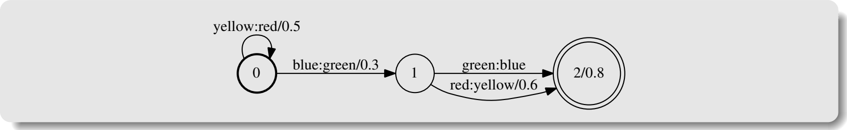
- Command: `fstinvert A.fst > C.fst`
- inverts a transducer; transposes the input and output symbols on each transition



Inverse

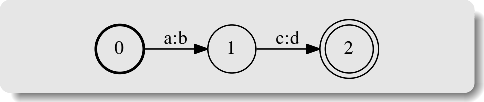
- Command: `fstinvert A.fst > C.fst`
- inverts a transducer; transposes the input and output symbols on each transition

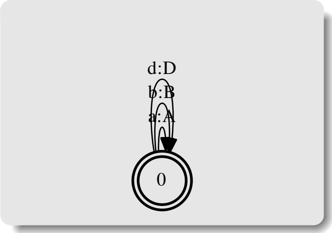




Composition

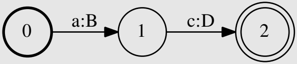
- Command: `fstcompose A.fst B.fst > C.fst`
- ‘composes’ FSMs:
  given 2 FSMs: fsm_1 that transduces from s_1 to s_2 and fsm_2 that transduces from s_2 to s_3, returns fsm_3 that transduces from s_1 to s_3 with the 2 costs combined
- *acceptor* is treated as a transducer to itself





Composition

- Command: `fstcompose A.fst B.fst > C.fst`
- 'composes' FSMs:
  given 2 FSMs: fsm_1 that transduces from s_1 to s_2 and fsm_2 that transduces from s_2 to s_3, returns fsm_3 that transduces from s_1 to s_3 with the 2 costs combined
- *acceptor* is treated as a transducer to itself



Section 4

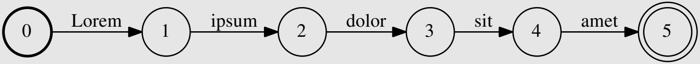
Text to FSM

Text as FSM

- How do we represent text as FSM?
- e.g.: *Lorem ipsum dolor sit amet*

Text as FSM

- How do we represent text as FSM?
- e.g.: *Lorem ipsum dolor sit amet*



Solution: text2fsa script (bash)

```
#!/bin/bash
# read input string from STDIN
str=$1
# parse it into array using space as separator
arr=($(echo $str | tr '␣' '\n'))
# set initial state
state=0
# iterate through array
# printing current and next states & token
for token in ${arr[@]}
do
        echo -e "$state\t$((state+1))\t$token"
        # increment state
        ((state++))
done
# print final state
echo $state
```

How do we test whether FSM accepts a string?

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

1. Create an FSM of the string & compile it

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

1. Create an FSM of the string & compile it
   - using script
   - `echo 'star of thor' | farcompilestrings --symbols=lex.txt --unknown_symbol='<unk>' --generate_keys=1 --keep_symbols | farextract --filename_suffix='.fst'`

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

1. Create an FSM of the string & compile it
    - using script

    -
      ```
      echo ‘star of thor’ |
      farcompilestrings --symbols=lex.txt
      --unknown_symbol=’<unk>’ --generate_keys=1
      --keep_symbols |
      farextract --filename_suffix=’.fst’
      ```
2. Intersect the string FSM with the FSM you are testing (e.g. LM)

Signals & Interactive Systems

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

1. Create an FSM of the string & compile it
   - using script
   -
     ```
     echo ‘star of thor’ |
     farcompilestrings --symbols=lex.txt
     --unknown_symbol=’<unk>’ --generate_keys=1
     --keep_symbols |
     farextract --filename_suffix=’.fst’
     ```
2. Intersect the string FSM with the FSM you are testing (e.g. LM)
   - `fstintersect string.fsa test.fsa > out.fsa`

Signals & Interactive Systems

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

1. Create an FSM of the string & compile it
   - using script
   - `echo ‘star of thor’ | farcompilestrings --symbols=lex.txt --unknown_symbol=’<unk>’ --generate_keys=1 --keep_symbols | farextract --filename_suffix=’.fst’`
2. Intersect the string FSM with the FSM you are testing (e.g. LM)
   - `fstintersect string.fsa test.fsa > out.fsa`
3. Print the output FSM

How do we test whether FSM accepts a string?

Intersect the string with the FSM.

1. Create an FSM of the string & compile it
   - using script

   -
     ```
     echo ‘star of thor’ |
     farcompilestrings --symbols=lex.txt
     --unknown_symbol=’<unk>’ --generate_keys=1
     --keep_symbols |
     farextract --filename_suffix=’.fst’
     ```
2. Intersect the string FSM with the FSM you are testing (e.g. LM)

   -
     ```
     fstintersect string.fsa test.fsa > out.fsa
     ```
3. Print the output FSM

   -
     ```
     fstprint --isymbols=lex.txt --osymbols=lex.txt
     out.fsa
     ```

Signals & Interactive Systems

Clean Output

```
fstintersect string.fsa test.fsa |
fstshortestpath |
fstrmepsilon |
fsttopsort |
fstprint --isymbols=lex.txt --osymbols=lex.txt
```

Section 5

FSA/FST Exercises

FST/FSA Exercises

- Mohri et al. (1996) FSM Toolkit Exercises

Exercise 1

Given the alphabet $L = \{a, b, ..., z, A, B, ..., Z, < space >\}$, create an automaton that:

a) accepts a letter in L (including space).

b) accepts a single space.

c) accepts a capitalized word (where a word is a string of letters in L excluding space and a capitalized word has its initial letter uppercase and remaining letters lowercase).

d) accepts a word containing the letter a.

Exercise 2

Using the automata in Exercise 1 as the building blocks, use appropriate FSM operations on them to create an automaton that:

a) accepts zero or more capitalized words followed by spaces.

b) accepts a word that is capitalized and contains the letter a.

c) accepts a word that is capitalized or does not contain an a.

Exercise 3

Epsilon-remove, determinize, and minimize each of the automata in Exercise 2. Give the number of states and arcs before and after these operations.

Exercise 4

Consider the automaton:

```
0 1 1
0 2 2
1 1 1
2
3 4 4
4 3 3
4
```

a) How many states can be reached from the initial state?

b) How many states can reach a final state?

c) Compile this automaton and then remove all useless states.

Exercise 5

Given the alphabet $\{a, b, ..., z, < space >\}$,

a) create a transducer that implements *rot13* cipher: $a \rightarrow n$, $b \rightarrow o$, ..., $m \rightarrow z$, $n \rightarrow a$, $o \rightarrow b$, ..., $z \rightarrow m$.

b) encode and decode the message “`my secret message`” (assume $< space > \rightarrow < space >$).

Exercise 9

Given the alphabet $L = \{A, G, T, C\}$,

a) create transducer T that implements edit distance
$d(x, x) = 0, x \in L$
$d(x, y) = d(x, \epsilon) = d(\epsilon, y) = 1, x \neq y \in L$

b) using T find the best alignment between the strings 'AGTCC' and 'GGTACC'