CS 3313
Foundations of Computing:

Equivalence of NFA and DFA

http://gw-cs3313-2021.github.io

1

Regular Languages – Summary

- DFA model – deterministic
- Non-deterministic Finite Automata
- Regular expressions to formally define languages
- Equivalences:
  - Proof/Algorithm to convert Reg. Expression to NFA
  - Proof/Algorithm to convert DFA to Reg. Expression
  - Proof/Algorithm to convert NFA to DFA

These proofs show equivalence of Reg.Expr and Finite Automata

non-determinism does not add to compute power of DFAs

2

Today…..

1. Outline algorithm to generate Regular expression from a DFA
2. Proof/algorithm to convert NFA (without λ moves) to a DFA
3. Proof/algorithm to convert λ-NFA to NFA without λ

- Approach of converting NFA to DFA using (2) and (3) is slightly different from textbook

DFA/NFA to Regular Expression

- Given any DFA M, there is a regular expression r that defines exactly L(M)
- Find the labels of the paths from start state to each final state
  - Concatenate labels on the path
  - If we have two choices of paths with labels w_1 and w_2 then "or" the paths to get w_1+w_2
  - If there is a cycle, with path labelled w, then w^*
- We discussed a process of generating an expression by examining the DFA/NFA…what we want is a constructive proof that can lead to an algorithm

Algorithm to generate Regular Expression from Finite Automata

- Can we design an algorithm that generates a NFA for any input regular expression ? Why ?
- Prove: Given a DFA M, construct a RE to represent L(M)
  - Constructive proof that can be implemented as an algorithm
    - What we present here is different from the textbook
- **key idea**: formulate the problem as a graph theoretic problem and develop dynamic programming solution
  - *Dynamic programming is a very important and often used technique to solve problems*
    - *Break down a problem, recursively, into simpler subproblems & optimal solution constructed from optimal sol for subproblems*

5

DFA-to-RE Algorithm

- A strange sort of induction.
- States of the DFA are named $1, 2, \ldots, n$.
- Induction is on k, the maximum state number we are allowed to traverse along a path.
- Derive set of strings (reg. exp.) that go from state q_i *to* q_j without passing through any state numbered k or greater
- Similar to the Floyd Warshall algorithm to compute for all pairs of nodes, the shortest paths between them in the graph
  - Did you see this before ?.....VERY useful (and often used) algorithm!

6

6

Key Ideas for DFA-to-RE Algorithm

- DFA M= $(Q, \Sigma, \delta, q_1, F)$
- N states: $(q_1, q_2, \ldots q_n)$
- Start state: q_1
- Consider path from state q_i to q_j that pass through states numbered at most k -- call these *k-paths*
  - Denote the set of strings that take DFA from q_i *to* q_j going through states *at most k* as $R(i,j,k)$
  - Derive regular expression for this set of strings
  - When $i=1$ and q_j is a final state, this represents the set of strings accepted by the DFA

7

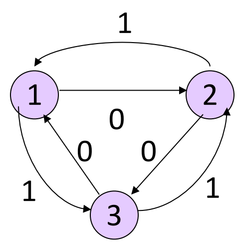
k-Paths

- A *k-path* is a path through the graph of the DFA that goes through no state numbered higher than k.
- Endpoints are not restricted; they can be any state.
- *n-paths* are unrestricted – can go through any state
- RE is the union of RE's for the *n-paths* from the start state to each final state.

8

8

Example: k-Paths



0-paths from 2 to 3:
RE for labels = **0**.

1-paths from 2 to 3:
RE for labels = **0**+**11**.

2-paths from 2 to 3:
RE for labels =
(**10**)***0**+**1**(**01**)***1**

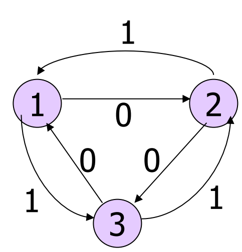
3-paths from 2 to 3:
RE for labels = ??

9

9

DFA to RE Constructive Proof: k-Path Induction

- Let R_{ij}^{k} be the regular expression for the set of labels of *k-paths* from state i to state j.
- Basis: $k=0$. only arcs or a node by itself
- R_{ij}^{0} = sum of labels of arc from i to j.
  - $\varnothing$ if no such arc.
  - But add λ if $i=j$.



- $R_{12}^{0} = \mathbf{0}$.
- $R_{11}^{0} = \varnothing + \lambda = \lambda$.

Notice algebraic law:
∅ plus/union anything =
that thing.

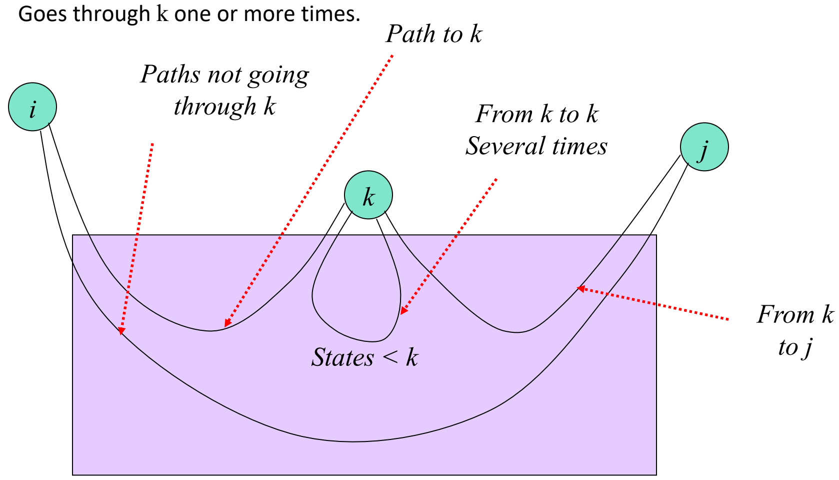
10

10

DFA to RE Constructive Proof: k-Path Induction

◆Let R_{ij}^{k} (r.e. for *k-paths* from state i to state j).

- Inductive case: A *k-path* from *i* to *j* either: (1) Never goes through state *k*, or (2) Goes through k one or more times.



11

11

k-Path Inductive Case

- A k-path from i to j either:
  1. Never goes through state k, or
  2. Goes through k one or more times.

$$R_{ij}^{k} = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^{*}\, R_{kj}^{k-1}.$$

Doesn't go through k

Goes from i to k through at most k-1

Zero or more times from k to k

Then, from k to j going through at most k-1

12

12

Algorithm:

- For each $1 <= i,j <= n$, compute compute the table for $R(i,j)$ for $k=0,1,2…n$ where $R(i,j)$ contains the regular expression for R_{ij}^{k} (or to visualize as a table, $R(i,j,k)$)

k=0

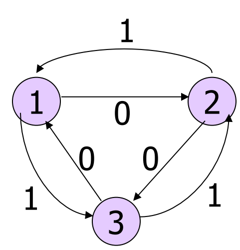
	1	2	3
1	λ	0	1
2	1	λ	0
3	0	1	λ

k=1

	1	2	3
1	λ	0	1
2	1	λ+10	0+11
3	0	1+00	λ+01

13

Example: k=2



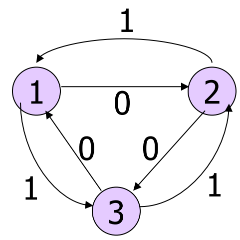
- $R_{12}^{2} = R_{12}^{1} + R_{12}^{1}(R_{22}^{1})R_{22}^{1}$
  - $0 + 0(\lambda + 10)^{*} (\lambda + 10) = 0 + 0 (10)^{*}$
- $R_{31}^{2} = R_{31}^{1} + R_{32}^{1}(R_{22}^{1})^{*}R_{21}^{1}$
- $R_{32}^{2} = R_{32}^{1} + R_{32}^{1}(R_{22}^{1})^{*} R_{22}^{*}$
- $R_{23}^{3} = R_{23}^{2} + R_{23}^{2}(R_{33}^{2})^{*}R_{33}^{2}$

	1	2	3
1	λ +(0(λ+10)*1	0+(λ+10)(10)*(λ+10) = 0+0(10)*= 0 (10)*	1+(0(λ+10)*(0+11)
2	1+(λ+10)(λ+10)*1= 1+(10)*1	(λ+10)+ (λ+10)(λ+10)*(λ+10)= (λ+10)*= (10)*	(0+11)+ (λ+10)(λ+10)*(0+11)= (0+11)+(10)*(0+11)
3	0 + (1+00)(λ+10)*(1)= 0 + (1+00)(10)*(1)	(1+00)+ ((1+00).(λ+10)*(λ+10)) = (1+00) (10)*	(λ+01) + ((1 +00) (λ+10)*(0+11))

14

14

Example: k=1



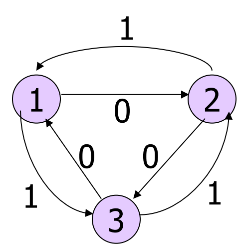
- $R_{12}^1 = R_{12}^0 + R_{11}^0 (R_{11}^0)^* R_{12}^0$

- $R_{22}^1 = R_{22}^0 + R_{21}^0 (R_{11}^0)^* R_{12}^0$

15

15

Example: k=1



- $R_{12}^1 = R_{12}^0 + R_{11}^0 (R_{11}^0)^* R_{12}^0$
  - $0 + \lambda\, (\lambda)^*\ 0 \ = 0$
- $R_{22}^1 = R_{22}^0 + R_{21}^0 (R_{11}^0)^* R_{12}^0$
  - $\lambda + 1 (\lambda)^* 0 = \lambda \ + 10$
- $R_{23}^1 = R_{23}^0 + R_{21}^0 (R_{11}^0)^* R_{13}^0$
  - $0 + 1\ (\lambda)^*\ 1 \ = (0+11)$

16

16

DFA to RE: Algorithm - Final Step

- The RE with the same language as the DFA is the sum (union) of R_{1j}^{n}, where:
  1. n is the number of states; i.e., paths are unconstrained.
  2. 1 (q_1) is the start state.
  3. *j* is one of the final states.

- In terms of an algorithm,

  R_{ij}^{k} *is* $R(i,j,k)$ with $1 <= i, j <= n$ and $0 <= k <= n$.

  - Implies $O(n^3)$ algorithm

17

17

Next: Equivalence to NFA and DFA

- Equivalence of automata models:

  two classes of automata are equivalent if they are equally 'powerful' – i.e., solve exactly the same set of problems

- In terms of languages accepted, model M1 and M2 are equivalent if any language accepted by M1 is accepted by M2 and vice versa

- We show DFAs and NFAs are equivalent
  - they accept exactly the same class of languages..Regular languages

18

Recall NFA Definition

- $M = (Q, \Sigma, \delta, q_0, F)$
- A finite set of states, typically Q.
- An input alphabet, typically Σ.
- A transition function, δ from $Q \times \Sigma$ to 2^Q
- A start state (q_0) in Q
- A set of final states $F \subseteq Q$.
- Difference with DFAs: transition function reads input a in state q and goes to a subset of states in Q
- NFA with λ moves: δ from $Q \times \{ \Sigma \cup \lambda \}$ to 2^Q
  - *Can make a move without reading input*

19

19

Language of an NFA

- A string w is accepted by an NFA if $\delta(q_0, w)$ contains at least one final state.

  $L(M) = \{ w \mid \delta(q_0, w) \cap F \neq \emptyset \}$

The language of the NFA is the set of strings it accepts.

- Extended Transition function extend to strings as follows:
- Basis: $\delta(q, \lambda) = \{q\}$
- Induction: $\delta(q, wa)$ = the union over all states p in $\delta(q, w)$ of $\delta(p, a)$

$$\delta(q, wa) = \bigcup_{p \in \delta(q, w)} \delta(p_i, a)$$

20

20

Equivalence of DFA's, NFA's

- A DFA can be turned into an NFA that accepts the same language.
  - If $\delta_D(q, a) = p$, let the NFA have $\delta_N(q, a) = \{p\}$.
  - Then the NFA is always in a set containing exactly one state – the state the DFA is in after reading the same input.
- Any NFA (with or without λ moves) can be transformed to an equivalent DFA accepting the same language
  - First show how λ NFA can be turned into a NFA that accepts the same language
  - Next, show how NFA without λ moves can be converted to a DFA that accepts the same language

21

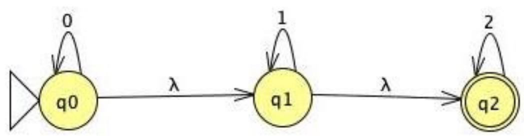
Paths in the λ-NFA and Concept of E-Closure

- A path from state p to state q is labelled with symbols from alphabet OR labeled with empty string
- To transform an NFA with λ-moves to an NFA without λ moves, of particular interest are paths labeled with empty string
  - An edge labeled with empty string implies from a state q, we can go to another state p without reading an input
- **Definition:** E-closure of a state = Path where all edges are labeled with empty string

22

Example – E-Closure

- NFA with λ moves
- E-Closure (q_0) = { q_0, q_1, q_2}
- E-Closure(q_1) ={ q_1, q_2}
- E-Closure(q_2) ={q_2}
- E-Closure({q_0, q_2}) = { q_0, q_1, q_2}



E-Closure: Definition & Extended δ

- E-Closure(q) = set of states p that you can reach from q following only edges labeled with empty string
- Can extend E-Closure to set of states:

For a set of states *P*: *E-Closure(P)* = $\bigcup_{q \epsilon P}$ *E-closure(q)*

- δ' Extended transition (over strings) for NFA
  - Basis: δ' (q, λ) = E-closure (q)
  - Ind.: δ' (q, xa) =
    - Start with (q, x) = S (set S of states)
    - Take the union of E-Closure(δ(p, a)) for all p in S.

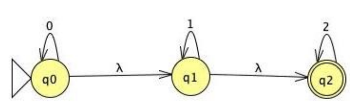
Equivalence of NFA and ϵ-NFA

- Every NFA is an λ-NFA (It just has no transitions on empty string ε)
- Every λ -NFA is an NFA: requires us to take an λ-NFA and construct an NFA that accepts the same language.
  - We do so by combining λ-transitions with the next transition on a real input.
- Start with an λ-NFA with states Q, inputs Σ, start state q_0, final states F, and transition function δ_E.
- Construct an "ordinary" NFA with states Q, inputs Σ, start state q_0, final states F', and transition function δ_N.
- Compute $\delta_N(q, a)$ as follows:
  1. Let S = E-Closure(q).
  2. $\delta_N(q, a)$ is the union over all p in S of $\delta_E(p, a)$.
- F' = the set of states q such that CL(q) contains a state of F.
- A straightforward proof of induction shows $\delta_E(q_0,w)$ is in F if and only if $\delta_N(q, w)$ is in F'

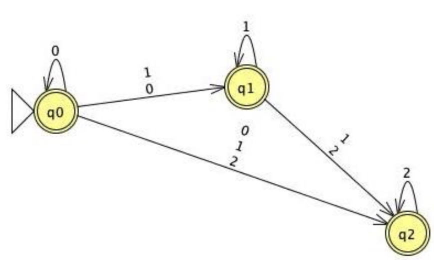
25

Example: Equivalence of NFAs

$$\delta'(q_0,0)= E\text{-}Cl\ (\delta(\delta'(q_0,\lambda),0)$$
$$= E\text{-}Cl\ (\delta\ \{q_0,q_1.q_2\},0)$$
$$= E\text{-}Cl\ (\{\delta(q_0,0)\} \cup \{\delta(q_1,0)\} \cup \{\delta(q_2,0)\}$$
$$= \{q_0,q_1,q_2\} \cup \emptyset \cup \emptyset$$
$$= \{q_0,q_1,q_2\}$$



$\delta'(q_0,1)= \{q_1,q_2\}$
$\delta'(q_0,2)= \{q_2\}$
$\delta'(q_1,0)= \emptyset$
$\delta'(q_1,1)= \{q_1,q_2\}$
$\delta'(q_1,2)= \{q_2\}$
$\delta'(q_2,0)= \emptyset$
$\delta'(q_2,0)= \emptyset$
$\delta'(q_2,2)= \{q_2\}$



Equivalence of NFAs and DFAs

- Surprisingly (?), for any NFA there is a DFA that accepts the same language.
- Proof is the *subset construction*.
  - Note: this means the number of states of the DFA can be exponential in the number of states of the NFA.
- Thus, NFA's accept exactly the regular languages.
- Importance of a constructive proof…..
  - The procedure to construct a DFA from the NFA provides us with an algorithm we can use to automate the process!

27

27

Transitions in a NFA

- Question: Given an NFA with n states in set Q, what is $\delta(q, w)$?
  - A subset S_i of Q
  - How many subsets can we have ?
- Example: $Q=\{q_0, q_1, q_2\}$ what can $\delta(q, w)$ be for any state $q \in \{q_0, q_1, q_2\}$ and any input w ?
- define a set of 2^n elements, $Q_D=\{p_1, p_2, \ldots, p_m\}$ where $m=2^n$

  and a one to one & onto mapping from 2^Q *to* Q_D
- *for each subset i of Q, we label it with an element* p_i
- Question: if $\delta(q, a)= S_i$ then using new labels…..?
  - $\delta(q, a)= p_i$ which is a single element…i.e., deterministic!

28

NFA to DFA Proof: Subset Construction

- Given an NFA with states Q, inputs Σ, transition function δ_N, state state q_0, and final states F, construct equivalent DFA D with:
  - States $Q_D = 2^Q$ (Set of subsets of Q).
  - Input alphabel Σ
  - Start state $\{q_0\}$.
  - Final states = all those with a member of F.
- The transition function δ_D is defined by:

$\delta_D(\{q_1,\ldots,q_k\}, a)$ is the union over

all $i = 1,\ldots,k$ of $\delta_N(q_i, a)$.

29

29

Critical Point

- The DFA states have *names* that are sets of NFA states.
- But as a DFA state, an expression like {p,q} must be understood to be a single symbol, not as a set.
- Analogy: a class of objects whose values are sets of objects of another class.
- Observe: after reading any input w, the NFA can be in a subset $\delta(q, w)$ – this subset is denoted by **one** element in Q_D
  - *To simulate the NFA for each input, the DFA keeps track of the subset of states that the NFA can be in after reading input*

30

30

Proof of Equivalence: Subset Construction

- Given NFA N= (Q, **Σ** , δ_N , q_0, F) define

  DFA M= (Q', **Σ** , δ_D, q_0', F')

  where: Q' = 2^Q – all subsets of Q

  – Label each element in *Q'* as $[q_{i1}, q_{i2},\ldots,q_{ik}]$ to denote the set $\{q_{i1}, q_{i2},\ldots,q_{ik}\}$
- $q_0' = [q_0]$ and *F'* = set of states in *Q'* that contain a state in *F*
- *Define* $\delta_D([q_1,q_2,\ldots,q_i],a) = [p_1, p_2,\ldots,p_j]$

  if and only if

  $$\delta_N(\{q_1,q_2,\ldots,q_i\},a) = \{p_1, p_2,\ldots,p_j\}$$

  (i.e., apply δ_N *to each element in* $(\{q_1,q_2,\ldots,q_i\})$)

31

31

Proof of Equivalence: 1

- The proof is almost a pun.
- Show by induction on $|w|$ that

  $$\delta_N(q_0, w) = \delta_D(\{q_0\}, w)$$
- Basis: $w = \lambda$: $\delta_N(q_0, \lambda) = \delta_D(\{q_0\}, \lambda) = \{q_0\}$.

32

32

Proof of Equivalence - 2

- Inductive Step: Assume IH for strings |w| =n.
- Let $w = xa$ with $|x|=n$
  - IH holds for x.
- From definition of extended $\delta_D([q_0], xa) = \delta_D(\delta_D([q_0], x), a)$
- from inductive hypothesis:

  $\delta_D([q_0], x) = [p_1,p_2...,p_j]$ *if and only if* $\delta_N(q_0, x)= \{p_1,p_2...,p_j\}$
- From definition of δ_D

  $\delta_D([p_1,p_2...,p_j],a) = [r_1,r_2,...r_k]$ *if and only if*
  $\delta_N(\{p_1,p_2...,p_j\}, a) = \{r_1,r_2,...r_k\}$
- Therefore $\delta_D([q_0], xa)= [r_1,r_2,...r_k]$ *iff* $\delta_N(q_0,x a) = \{r_1,r_2,...r_k\}$

33

Proof of Equivalence - 3

- $\delta_D([q_0], xa)= [r_1,r_2,...r_k]$ *iff* $\delta_N(q_0,x a) = \{r_1,r_2,...r_k\}$
- From definition of DFA final states F',

  $[r_1,r_2,...r_k]$ is in F' *iff* $\{r_1,r_2,...r_k\}$ *contains a state in F*
- *Therefore, w =xa is accepted by DFA) iff it is accepted by NFA*

34

Algorithm….slight modification

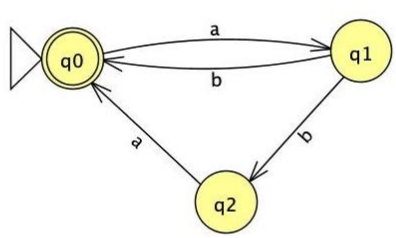
- Straightforward mapping of proof to algorithm – works but….?
- Number of states in NFA = n then number of states in DFA = ?

- A more practical algorithm: start with start state and define the transition function for all reachable states

•Turns out there can be a further optimization by finding equivalent states and eliminating them.

35

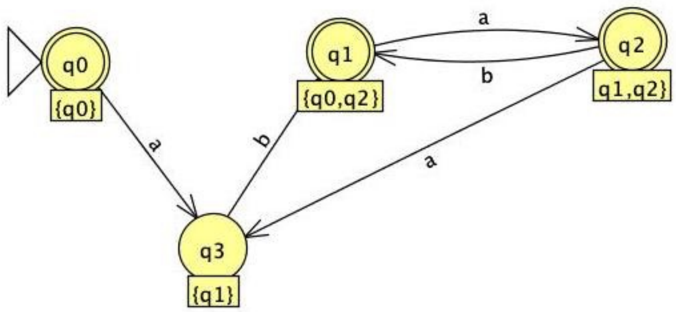
Example



36

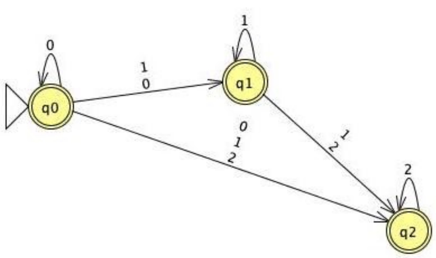
Example – equivalent DFA

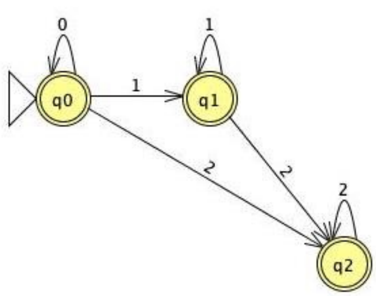
- From construction:



37

NFA and Equivalent DFA





38

Summary

- Reg. Expr, DFA's, NFA's, and λ–NFA's all accept exactly the same set of languages: the regular languages.
  - NFA = DFA *and* λ–NFA = NFA, therefore DFA= λ–NFA
- NFAs types are easier to design but only DFA can be implemented!
- Algorithms to convert from NFA to DFA…..
  - But could end up with a large number of states….
    - Can we minimize the number of states?
- Next…the BIG question = properties of regular languages
  - What types of languages are regular? What happens when we combine reg. lang. using set and algebraic operations? How do we know if the language is not regular ?
    - How can we **prove** that a language/problem is not regular ?

39

39