CSE491/596 Lecture 9/11/20: NFA-to-DFA Example and Basic GNFAs

We make a slight change to the heart of the proof where we left off. The change saves some time in executing the NFA-to-DFA construction when ϵ-arcs are present and reduces errors. First define

$$\underline{\delta}(p,c) \;=\; E(\{q : (p,c,q) \;\in\; \delta\})$$

for any state $p \in Q$ and char c. Recall $E(\cdot)$ is ϵ-closure. So what this means in simple terms is:

1. **First** take arc(s) on c out of the state p.
   - If there are none, **stop** and put $\underline{\delta}(p,c) = \emptyset$.
   - Else collect all states q reached on those arc(s).
2. **Then,** for each state q reached by processing c, add states reached on any series of ϵ-arcs out of q, if there are any.

Now we can give a new definition of the DFA's transition function Δ: for any $P \subseteq Q$ and $c \in \Sigma$,

$$\Delta(P,c) \;=\; \bigcup_{p \in P} \underline{\delta}(p,c)\;.$$

The difference is that we avoid worrying about initial ϵ-arcs that could come before processingf c. We only have to track *trailing* ones in a machine diagram. The reason is that the trailing arcs at the previous step already took care of any initial ones now. Initializing the start state S of the DFA M to have all states reached by ϵ-arcs out of s in N sets this in motion. We need to prove for all i:

$$G(i) : \Delta^*(S, x_1 \cdots x_i) \;=\; \{r : N\,can\,process\,x_1 \cdots x_i\,from\,s\,to\,r\}.$$

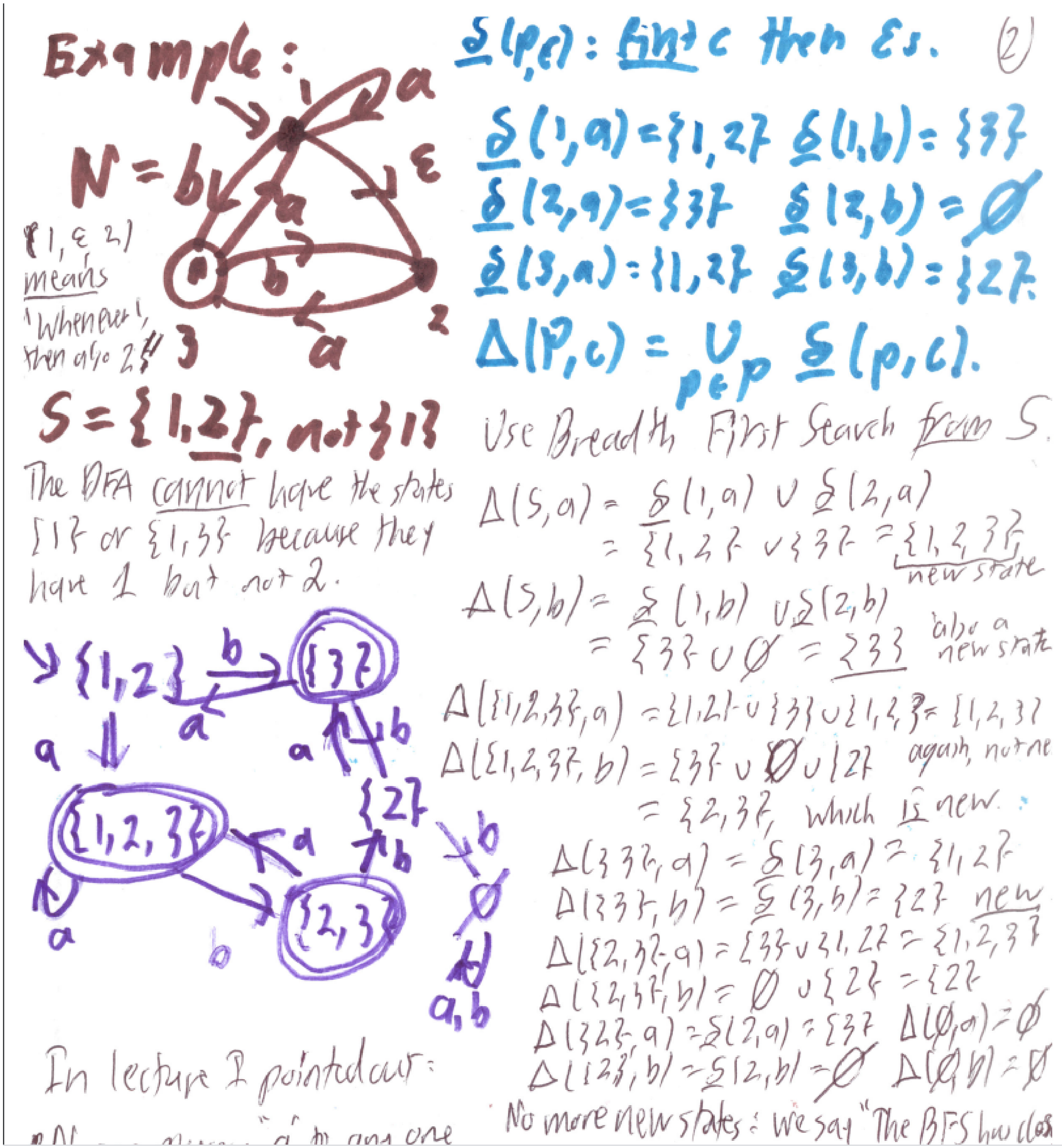
Here we have *extended* Δ, a function of a state and a char, to Δ^* which is a function of a state and a *string*, by the basis $\Delta^*(R,\epsilon) = R$ for all $R \in \mathbb{Q}$ and for $i \geq 1$,

$$\Delta^*(R, x_1 \cdots\; x_{i-1}x_i) \;=\; \Delta\Big(\Delta^*(R, x_1 \cdots\; x_{i-1}),\; x_i\Big)\,.$$

So let R_{i-1} stand for $\Delta^*(S, x_1 \cdots\; x_{i-1})$. Then by the inductive hypothesis $G(i-1)$, R_{i-1} equals the set of states q such that N can process $x_1 \cdots\; x_{i-1}$ from s to q. Now put $R_i = \Delta(R_{i-1}, x_i)$.

- Let $r \in R_i$. Then $r \in \underline{\delta}(q, x_i)$ for some $q \in R_{i-1}$. By IH $G(i-1)$, N can process $x_1 \cdots\; x_{i-1}$ from s to q. And N can process x_i from q to r by definition of $r \in \underline{\delta}(q,x_i)$. So N can process $x_1 \cdots x_i$ from s to r.
- Suppose N can process $x_1 \cdots x_i$ from s to r. Then---and this is the key point---the processing goes to some state q just before the char x_i is processed. By IH $G(i-1)$, q belongs to R_{i-1}. Moreover, $r \in \underline{\delta}(q, x_i)$ because we first do the step that processed the char x_i at q, then any trailing ϵ-arcs. Thus $r \in \Delta(R_{i-1}, x_i)$, which means $r \in R_i$.

Thus we have established that R_i equals the set of states r such that N can process $x_1 \cdots x_i$ from s to r. This is the statement $G(i)$, which is what we had to prove to make the induction go through. This finally proves the NFA-to-DFA part of Kleene's Theorem. ☒



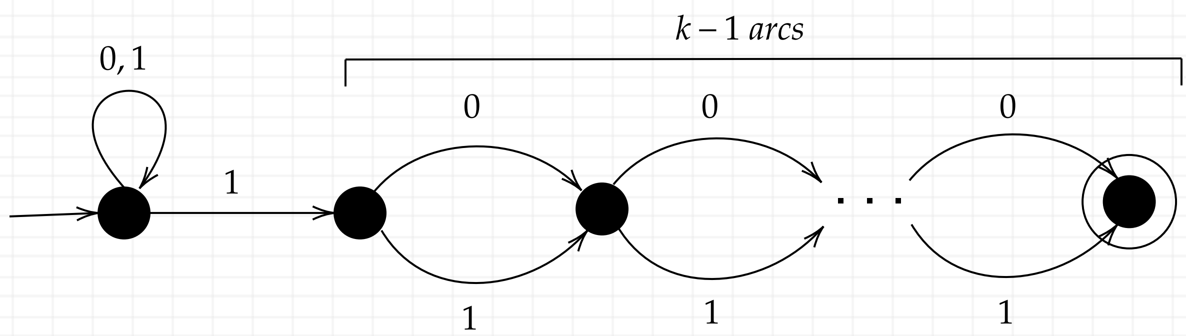
The extra things pointed out have to do with how the states of the DFA tell what the NFA can and cannot process:

- The NFA cannot process the string bbb from its start state at all. However you try, you come to the NFA state 2 being unable to process a b. Nor can it process bbb from any other state.

- However, N can process a from start to any one of its three states:
  - $(1, a, 1)$
  - $(1, a, 1)(1, \epsilon, 2)$
  - $(1, \epsilon, 2)(2, a, 3)$.

    This is shown in the DFA by the single arc $(S, a, \{1, 2, 3\})$.
- But in the string $x = abbb$, even though the initial a "turns on all three lightbulbs of N", the final bbb still cannot be processed by N. The DFA M does process it via the computation $(S, a, \{1, 2, 3\})(\{1, 2, 3\}, b, \{2, 3\})(\{2, 3\}, b, \{2\})(\{2\}, b, \emptyset)$, but that computation ends at $\emptyset$, which---when present at all---is always a dead state.

Another example: The "Leap of Faith" NFAs N_k for any $k > 1$:



$$L(N_k) = (0+1)^* 1 (0+1)^{k-1}$$
$$= \left\{ x \in \{0,1\}^* : \textit{the kth bit of x from the end is a } 1 \right\}.$$

Fact (will be proved next week): Whereas the NFA N_k has only $k+1$ states, the smallest DFA M_k such that $L(M_k) = L(N_k)$ requires 2^k states.
This is a case of **exponential blowup** in the NFA-to-DFA algorithm.

Now here is a simple algorithm for telling whether a given string x *matches* a given regexp α:

1. Convert α into an equivalent NFA N_α.
2. Convert N_α into an equivalent DFA M_α.
3. Run M_α on x. If it accepts, say "*yes*, it matches", else say "no match".

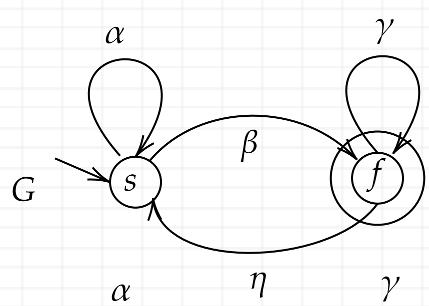
This algorithm is *correct*, but it is *not efficient*. The reason is that step 2 can blow up. An intuitive reason for the gross inefficieincy is that step 2 makes you create in advance all the "set states" that would ever be used on all possible strings x, but most of them are unnecessary for the particular x that was given.

There is, however, a better way that builds just the set-states $R_1, \dots, R_i, \dots, R_n$ that are actually encountered in the particular computation on the particular x. We have $R_0 = S = E(s)$ to begin with. To build each R_i from the previous R_{i-1}, iterate through every $q \in R_{i-1}$ and union together all the sets $\underline{\delta}(q, x_i)$. If N_α has k states---which roughly equals the number of operations in α---then that takes order $n \cdot k \cdot k$ steps. This is at worst cubic in the length $\widetilde{O}(n+k)$ of x and α together, so this counts as a **polynomial-time algorithm**. It is in fact the algorithm actually used by the `grep` command in Linux/UNIX.

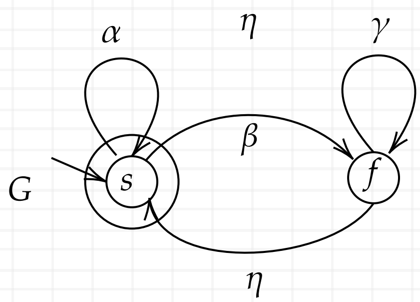
Generalized NFAs (GNFAs) --- having only 2 states.

A *generalized NFA* G can have any regular expression on any arc. A string x is "accepted" by G if it can be broken into m substrings such that each substring matches the respective regexp in a path of m arcs of G that begins at s and ends in a final state f. A regular NFA in in fact a GNFA in which every arc has a "basic" regular expression---that is, just a char **c** in Σ, or ϵ.

I do not regard GNFAs as "machines" that can be "executed"---even in the sense where we could say that the `grep` algorithm executed the NFA N_α on x. I regard them as helpful shorthand for diagramming languages. The most illuminating case IMHO of this is for two-state GNFAs:



$$L(G) = L_{sf} = \left(\alpha + \beta\gamma^*\eta\right)^*\beta\gamma^*$$
$$= \alpha^*\beta\left(\gamma + \eta\alpha^*\beta\right)^*$$



$$L(G) = L_{ss} = \left(\alpha + \beta\gamma^*\eta\right)^*.$$

η is pronounced "ate-a" in the US, "eat-a" in the UK.