The entire idea behind NFA $\rightarrow$ Regular Expression conversion is to label edges by regular expressions instead of just single symbols.

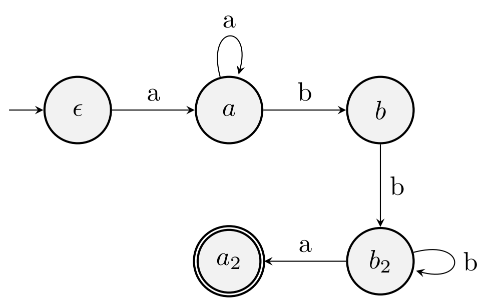
By the end of the conversion we will have just two nodes, I and F with one big long regexp connecting them.

But problem 1(a) is harder. Here is a problem similar to 1(a) that I can solve without earning an academic integrity violation:

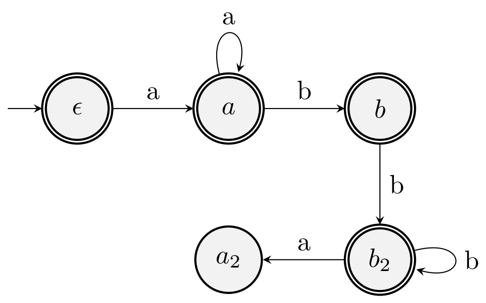
Find a regular expression for the complement of $\mathbf{L}\left((\mathbf{aa^*b})(\mathbf{bb^*a})\right)$

The first step is to convert that regular expression to an NFA, which we will change the accepting states of to make it accept the inverse.

That NFA for $(aa^*b)(bb^*a)$ looks like:

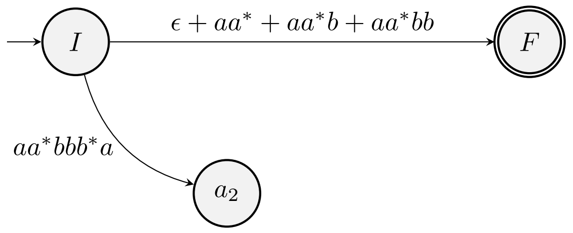


If we want the NFA for its complement, we just change which states are accepting (the accepting states are double-ringed. We notate it in class with a line away but I don't know how to do that in latex):



Now this accepts the complement of $(aa^*b)(bb^*a)$. It will accept anything so long as it doesn't reach a_2.

Now let's run the NFA $\rightarrow$ regex conversion. I'll save us both some time and just show the answer, which I'll get by removing nodes and re-labeling edges accordingly:

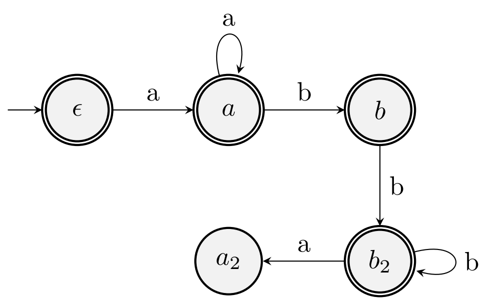


It seems like our answer is: $aa^*bb + aa^*b + aa^* + \epsilon$

But that doesn't look like the complement of $(aa^*b)(bb^*a)$ at all! We're looking for string that this regex doesn't match, and yet our answer doesn't even let us start a string with b, which would be an easy way to make a string that isn't matched by $(aa^*b)(bb^*a)$

That's a terrible, terrible problem. We followed the algorithm! How did we get here?

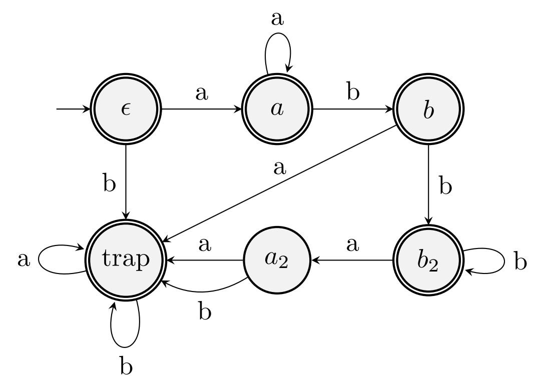
Let's look at the NFA we started the conversion with:



What if we plug in a string that starts with b? The machine outright rejects it, without putting it in a rejecting state. There's no b edge away from the ϵ node, so we just stop executing.

That's hugely, hugely bad. We took all the rejecting states from the very first machine and made them the accepting states for this machine. If we can reject strings without ending in a rejecting state, it's going to totally screw up our algorithm.

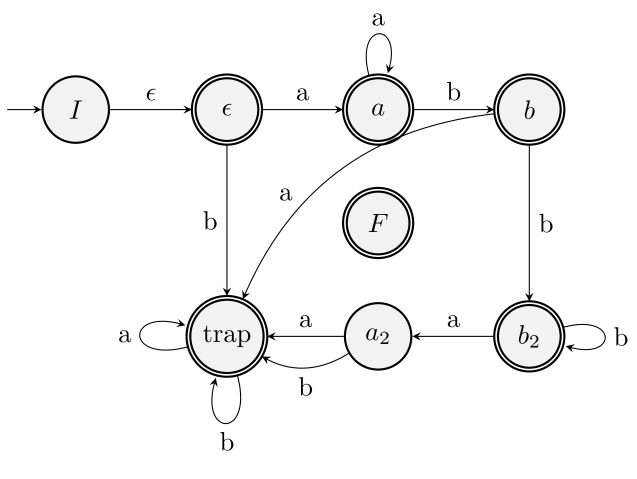
That is why problem 1(a) is hard. You need to fully define the NFA that you convert your regex into. When we do it correctly, it looks like:



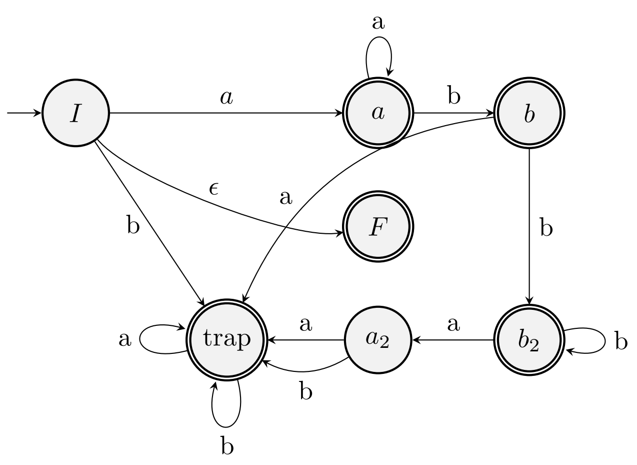
This totally changes the game. **Every node needs an edge for every symbol in the alphabet**. If you are going to reject a string, do it by forcing it into a rejecting state (like trap here). This is because you complement the NFA by marking all rejecting states accepting, allowing you to accept the strings you would normally reject. You just cannot do that if you don't have a rejecting state to end on.

Let's walk through the process with the correct NFA.

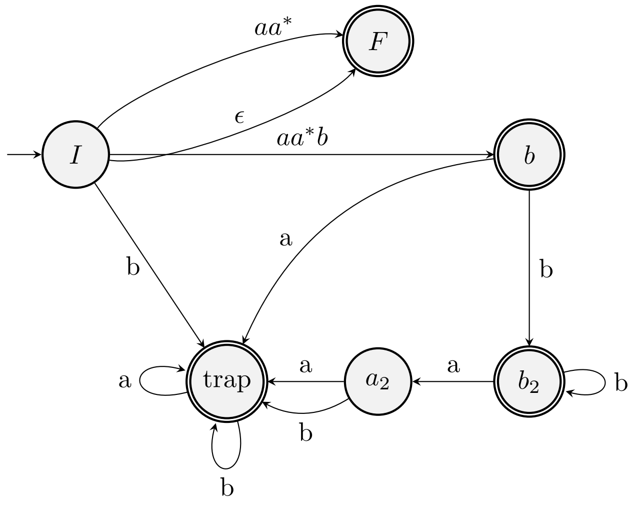
Add I and F (it's too crowded to add epsilon transitions from all the accepting states to F, pretend they're there)



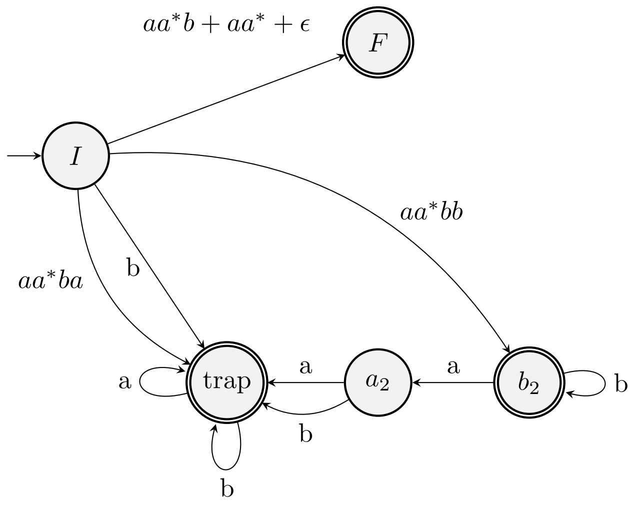
Remove ϵ



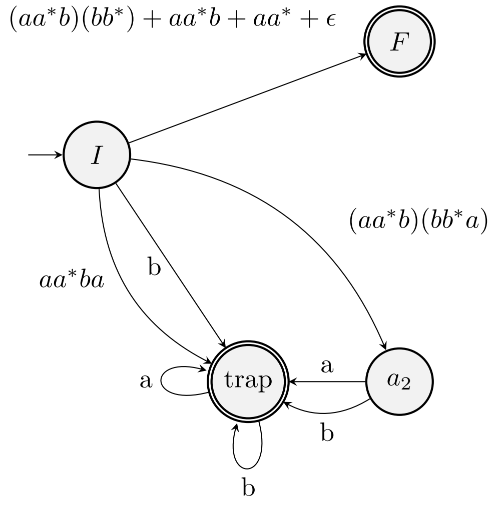
Remove a, move F because its getting crowded



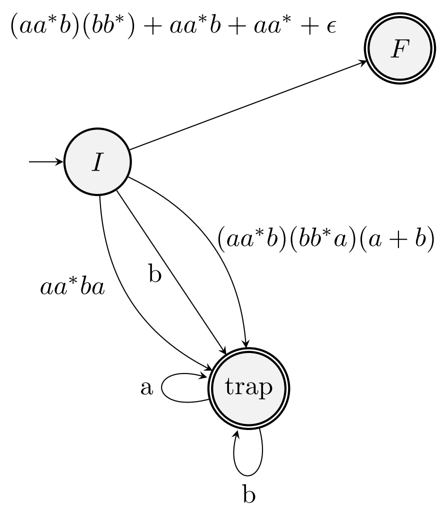
Remove b, collapse the edges of F down to a single, nice one because it's still too crowded



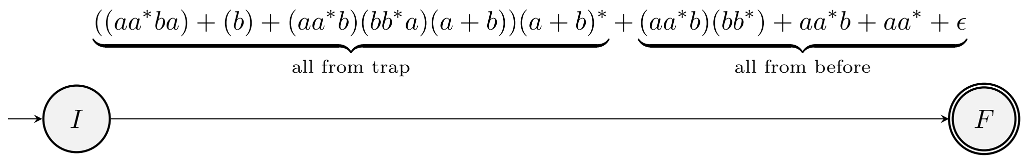
Remove b_2



Now remove a_2



Now the big boy, remove trap. Remember that all accepting states have implied epsilon transitions to F, so we move the rest of this monster expression to F


$\underbrace{((aa^*ba) + (b) + (aa^*b)(bb^*a)(a+b))(a+b)^*}_{\text{all from trap}} + \underbrace{(aa^*b)(bb^*) + aa^*b + aa^* + \epsilon}_{\text{all from before}}$
I
F

And that's our regex. It's a total beast, completely gnarly, and way, way bigger and more correct than what we started with. Who knows if it's correct. I sure don't. I must have made a mistake somewhere, but it shows how important it is that when you convert the regex to nfa, make sure EVERY SINGLE NODE has an edge for EVERY SINGLE SYMBOL in the alphabet.