A. Pattern Matching with Regular Expressions using Recursive / Backtracking Search

- Concentrate on basic regular expressions built using $\epsilon, \Sigma$, concatenation, $|, (...)$, and Kleene star.
- Matching problem: Does a given string of symbols $w$ from $\Sigma$ conform to a given regular expression?
  - Is $w \in L(R)$ where $R$ is a regular expression?
  - Easiest part of problem: Matching individual symbol, concatenated together
    - Does $w = a_1 a_2 a_3 \ldots$ match regular expression $b_1, b_2, b_3 \ldots$? Check $a_1 = b_1, a_2 = b_2, \ldots$.
    - Given an r.e. $R = a R_1$ (character $a$ followed by rest of expression) and string $w = b v$, verify that $a = b$ and continue search with $v$ and $R_1$. If $a \neq b$, the search fails.
    - To keep track of where we are in the search, let's maintain the string of characters we've approved so far and the string of remaining characters. (Their concatenation should always match the original starting string.)
    - So to match $a R_1$ with input $b v$ (and approved string $x$), check for $a = b$. If they match, then continue matching with $R_1$ and $v$ and approved $x a$. If they don't match, fail.
  - For a general concatenation of expressions $R_1 R_2$ and input $w$, the process is similar except that instead of matching the first symbol of $w$ against a symbol for $R_1$, we recursively look for a match with all of $R_1$ and, if that succeeds, then continue the match with $R_2$.
    - Match $R_1$ against $w$. If it fails, we fail.
      - If it succeeds with some suffix $w'$ remaining from $w$ and approved prefix $x'$ from $w$, then match $R_2$ against $v$. If that fails, then we fail.
        - If it succeeds with some tail $w''$ and approved $x''$, then we succeed with $w''$ leftover and approved string $x' x''$.
    - For an alternation $R_1 \mid R_2$, the process is easier than concatenation: Search with $R_1$ and $w$; if it succeeds, we succeed; if it fails, then search with $R_2$ with $w$ (the same $w$). We succeed iff that succeeds.
  - If $R$ is the empty string symbol $\epsilon$, we succeed with $w$ unchanged and the empty string as the approved prefix.
  - There's a subtlety with the Kleene star. $R^*$ is like an infinite alternation $\epsilon \mid R \mid RR \mid RR R \mid \ldots$. To "loop" through the sequence of $R$'s, we can just match against $R$ ($R^*$). So essentially we can treat $R^* = \epsilon \mid RR^*$.
    - The subtle problem is that $\epsilon$ matches any string, so with $\epsilon \mid RR^*$, we'll always succeed with just $\epsilon$ and skip $RR^*$.
    - If we look at a concrete example, like match $a^*$ against $aaaab$, what we probably want is to look for as many $a$'s as possible, so we should succeed, approving $aaa a$ and leaving $b$.
    - The general question is "When matching $R$ against $w$, if more than one prefix of $w$ matches, then which prefix should we approve?"
• For $R^*$, it seems like we want the longest prefix. It’s surprisingly easy to guarantee this: Instead of treating $R^*$ like $\varepsilon \mid R \ R^*$, treat it like $R \ R^* \mid \varepsilon$. Recursively, we’ll try to match $R \ R^*$ which will in effect start by trying to search $R \ R \ R^*$, which will try $R \ R \ R \ R^*$ and so on.
  • For our example, $a^*$ against $aaaab$, first we’ll match $a$ against $aaaab$, succeed with approved $a$ and leftover $aab$, then $a^*$ against $aab$ first matches $a$ against $aab$, which succeeds approving $a$ and leaving $ab$, and so on. The recursion has to stop because eventually we reach $b$.
• Activity question: The "Treat $R^*$ like $R \ R^* \mid \varepsilon$" backtracking algorithm relies on an assumption about what happens when we match the $R$ part of $R \ R^*$ against the string. What is the assumption, and what happens if the assumption is not met?
• Activity question: This backtracking algorithm does not always return the longest matching prefix. Give an example of when this happens (i.e., give an $R$ and $w$) and discuss how to modify the algorithm so that it always returns the longest matching prefix.

B. Displaying a Backtracking R.E. Match using a Graph
• It’s fairly easy to take the backtracking matching algorithm and encode it as a graph problem.
• We start with a distinguished start node, and if the overall search succeeds, we’ll end in a distinguished accepting node (or nodes). Ending in a non-accepting node indicates failure.
• So given that we’re at some node in the graph and want to diagram a search for regular expression $R$, we want to extend the graph to process all of $R$.
  • If $R = a \ R'$ where $a$ is a character, then our graph can go from the current node (looking for $a \ R'$) to the subgraph for $R'$ iff the leading character of $w$ is $a$. In the graph, we can draw a directed arrow between the two nodes and label it with $a$, to remind us when the transition is allowed. The target node is the one where we "accept" $a$.
  • If we have a sequence of characters $a_1 \ a_2 \ a_3 \ldots \ R'$, then we’ll have a path of nodes with arcs labeled $a_1, a_2, \ldots$ and concatenating those characters will give the prefix part of $w$ that we’ve approved so far.
  • More generally, for two concatenated expressions $R_1 \ R_2$, we should extend the graph to handle $R_1$ and then extend that to handle $R_2$. When describing the subgraph for each kind of subexpression, we must be careful to designate an accepting node. So for $a \ R'$, extending the graph to handle $a$ leaves us at a node that we can start $R'$ from.
  • If $R = \varepsilon$, we can introduce a new node and label the arc to it with $\varepsilon$. (So it’s similar to having an actual character.)
  • For an alternation $R_1 \mid R_2$, we can think in terms of drawing subgraphs for $R_1$ and $R_2$ and connecting them in to the overall graph. To connect the subgraph for $R_1$ to our overall graph, we can introduce an $\varepsilon$ arc from our current node to the starting node for $R_1$; similarly, we introduce an $\varepsilon$ arc from our current starting node to the starting node for $R_2$. We also need an overall accepting node: Introduce a new node and connect the accepting nodes for $R_1$ and $R_2$ to it using $\varepsilon$.
  • Similarly, for $R^*$, draw the subgraph for $R$. Add a new node for accepting $R^*$; certainly we want to connect the end node of $R$ to that node. To get the looping effect of $R^*$, we add an $\varepsilon$ arc from
the accepting node for $R$ to the starting node for $R$. Since $R^*$ might match $R$ zero times, we also introduce an $\varepsilon$ arc from the start node for $R$ to the accepting node for $R$.

- As an example, here is are two graphs for $a (b \mid c)^*$.
  - The first graph follows the algorithm in the book for generating one from a regular expression.
  - The second graph shows the result of eliminating some redundant $\varepsilon$-arrows.
C. Nondeterministic Finite Automata (NFA)

- The graph that corresponds to \( R \) is an example of a **nondeterministic finite automaton**.
  - *Automaton* because it encodes an algorithm.
  - *Finite* because there are a finite number of nodes in the graph.
    - Each node is called a "state" of the automaton.
  - *Nondeterministic* because we might have multiple "next" states to go to.
- Nondeterminism arises a couple of ways.
  - Two arcs labeled with the same character that leave the same state but go to different targets.
    - When we do the matching, we have to decide which way to go.
    - An arc labeled \( a \) and one labeled \( b \) (where \( a \neq b \)) does give us a fixed choice.
  - An NFA can also be "in" more than one state at a time.
    - If we have a state \( S_1 \) connected to \( S_2 \) by an \( \varepsilon \) arc, then we have the choice of staying in \( S_1 \) or going to \( S_2 \). Since the first character of \( w \) can lead us to different nodes, the choice may be important.
- A general NFA also allows a state to have no arc labeled with a particular character.
  - E.g., you might have a node with out-arrows labeled \( a \) and \( b \) but not \( c \).
  - If you're in that state and see character \( c \), the matching fails. (People sometimes say you get "stuck" and can't proceed.)
- Our matching algorithm for regular expressions correspond to backtracking search paths through the NFA they generate.
- For every regular expression, there exists an NFA that accepts exactly the same language.
- Not obvious, but for any NFA, there is a regular expression with the same language.
- So regular expressions and NFA's have the same **expressiveness**.

Deterministic Finite Automata (DFAs)

- A **deterministic finite state automaton** (DFA) is an NFA with no nondeterminism. At every node, your choice (given the next input character) is fixed. A DFA is easier to execute in that it doesn't involve backtracking.
- For every NFA there exists an equivalent DFA; in fact, we can calculate it.
  - Say NFA node \( S \) goes to two different states on character \( a \). When we use backtracking search, we follow one of those arcs and backtrack to try the other arc if necessary.
  - In the DFA, we'll instead have a single set of states that answers the question "What are all the NFA states I could have gotten to from \( S \) with an arc labeled \( a \)?"  
    - Whenever execution reaches this DFA state we'll be in one of the NFA states but not know which one.\(^1\)

\(^1\) Or if you prefer, we can think of being in all of those states simultaneously by following all possible search paths simultaneously. For example, using threads, whenever we encounter a state with multiple outgoing arcs with the same label, fork off a new thread of control and follow both paths simultaneously.
• Since we now have a set of NFA states, we’ll have to look at all outgoing edges from states in that set, which in turn can make us create other DFA states that are sets of NFA states.

D. **Converting from an NFA to an Equivalent DFA**

Here’s an algorithm for the NFA-to-DFA conversion.

- Initialize a set of unprocessed DFA states by adding all the NFA states.
- As long as there’s an unprocessed DFA state, call it set \( U \)
  - For every character \( a \) in the alphabet, go through the NFA states in \( U \) and collect all the outgoing NFA arcs labeled \( a \). Let \( T \) be the set of NFA states that are targets of these arcs.
    - This whole bundle of NFA arcs needs to be a single state in the final DFA, so make \( T \) a state in the DFA with a DFA arc labeled \( a \) from \( U \) to \( T \).
    - If \( T \) is not already in the set of processed DFA states, add \( T \) to the set of unprocessed DFA states.
      - Also, if there exist any NFA-accepting states in \( T \), then mark \( T \) as an accepting state in the DFA.
  - With \( U \), once we finish looking at all the characters in the alphabet, move \( U \) from the set of unprocessed DFA states to the set of processed states and go back to process a new \( U \).
- This algorithm can be modified to handle an NFA with \( \varepsilon \)-transitions and yielding an NFA without them.
  - When we first look at \( U \), before processing any of the alphabet characters, find all target NFA states from states in \( U \) that use paths of \( \varepsilon \)-transitions.
  - Add those states to \( U \), let’s call the result \( U' \).
  - Check every \( \varepsilon \)-transition arc leading to a state in \( U' \); if it’s part of a \( \varepsilon \)-transition path from \( U \) to \( U' \), remove the arc from the NFA.
    - (We can’t just delete all \( \varepsilon \)-arcs to states in \( U' \) because there can be \( \varepsilon \)-arcs into \( U' \) that don’t come from states in \( U \).)
  - If \( U' \) is not already in the set of processed states, use it to replace \( U \) in the set of unprocessed states and continue the algorithm.

E. **Calculating \( \varepsilon \)-Closures; Removing Stuck Configurations**

- If you look only at \( \varepsilon \)-transitions instead of actual alphabet characters, then the algorithm above calculates the \( \varepsilon \)-closure of the original NFA.
- If the original NFA had any "stuck" state/character combinations (i.e., a state \( S \) with no outgoing arc labeled \( a \)), then it’s possible for the final DFA to be have some state \( S' \) that has no outgoing arc labeled \( a \).
  - We can model this "stuck" behavior as going to an **error state** (a state that never leads to an accepting state).
  - Add a single new error state \( E \) and add a DFA arc labeled \( a \) from \( S' \) to \( E \).
  - We only need one \( E \), but we need an arc to \( E \) for every missing arc in the DFA.
• (I.e., one arc for every combination of $a$ and $S'$ where there's no outgoing arc from $S$ labeled $a$.)
• $E$ also needs circular arcs for every character in the alphabet: For each character $a$, add an $E$-to-$E$ link labeled $a$. This ensures that once we get to $E$, we never leave.
• Arcs from $\{T_1, T_2\}$ — every arc in the original NFA that goes from either $T_1$ or $T_2$ just goes from $\{T_1, T_2\}$ now.
  • E.g., on $a$, $S$ goes to $T_1$ and $T_2$, from $T_1$ we go to $T_3$ on $b$ or $c$, from $T_2$ we go to $T_4$ on $a$ or $c$.
  • $S$ via $a$ to $\{T_1, T_2\}$, $\{T_1, T_2\}$ via $b$ to $\{T_3\}$, $\{T_1, T_2\}$ via $a$ to $\{T_4\}$
  • $\{T_1, T_2\}$ has two arcs labeled $c$ to $\{T_3, T_4\}$ -- replace it by just one arc
  • Just generally if we have a DFA state $S$, where set $S \subseteq$ states of NFA, then look for all arcs labeled $a$ out of any state in $S$. We want to bundle all of those NFA arcs into one DFA

![Original NFA](image1)

NFA after $\epsilon$-closure

![NFA after $\epsilon$-closure](image2)

arc; to do that, we the set of target states has to be a state in the DFA. All those target states in the NFA have to be combined into one node in the DFA.
• We may have to recursively join subsets of the states of the NFA to get bigger ones.
Set-of-States DFA

DFA With Error State (need arcs from Err to itself)