Regular Expressions: From theory to practice

June 30, 2025

Regular expressions are a powerful tool for searching and manipulating text. They allow us to define patterns that can match specific sequences of characters. They find their use in most modern programming languages, text editors and command line tools. Let's explore them in a bit more detail.

What is a Regular Expression?

A regular expression is a special string of text which describes a search pattern. It can be used to check if a string matches the specific pattern defined by the regular expression.

An example of a regular expression is `a+b+`, which matches any string that starts with one or more `a`s followed by one or more `b`s. For example, it matches `ab`, `aaaabb`, but does not match `a`, `b`, `aa`, `bb`, or `abab`.

Historically, the study of regular expressions evolved as a theoretical subject, in the study of formal languages.

A formal language is a set of strings on an alphabet. There are many types of formal languages, each described by a different type of grammar, and a model of computation that can recognize the strings in the language. (See Chomsky hierarchy for more details.)

The simplest type of formal language is the **regular language**. As their name suggests, regular expressions are used to describe regular languages.

For example, the regular expression `a+b+` describes a regular language that consists of all strings that start with one or more `a`s followed by one or more `b`s.

The corresponding model of computation that recognises regular languages is the **finite automaton**.

Finite Automata

A finite automaton is a theoretical machine that can take a string as a input and either accept or reject it.

- A finite automaton contains a finite number of **states**.
- Between states, there exist predefined **transitions** that the automaton can take based on the input character it reads, to move to another state.

- One state is designated as the **start state** (usually denoted with an arrow pointing to it), this is where the automaton begins processing the input string.
- One or more states are designated as **accept states** (usually denoted with a double circle), if the automaton ends in one of these states after processing the input string, it accepts the string. Otherwise, it rejects the string.

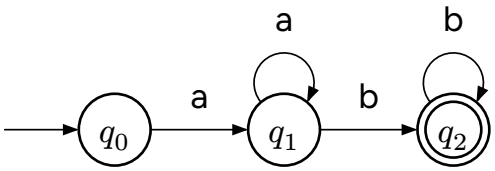


Figure 1: A finite automaton that accepts strings of the form `a+b+`.

The diagram above shows a finite automaton which accepts strings of the form `a+b+`. In fact, it is an example of a **deterministic finite automaton** (DFA), which means that for a given state and character, the next state is uniquely determined.

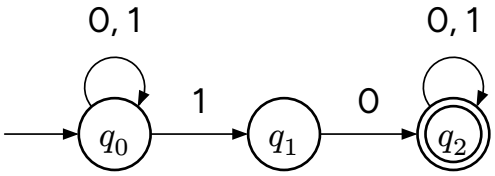


Figure 2: An NFA that accepts strings of the form `(0|1)*10(0|1)*`.

Figure 2 shows another finite automata, which accepts binary strings, which have the substring `10` at least once. However, this automaton is a **nondeterministic finite automaton** (NFA), which means that for a given state and character, there can be multiple possible next states. In this case, the automaton can stay in the same state or move to another state when it reads `1` from state q_0. In an NFA, the automaton is said to accept a string if there exists at least one valid path from the start state to an accept state that reads the entire string.

Imagine a computer program that implements an NFA matcher. When it reads the input string, if it encounters a character that has multiple possible transitions, it needs to try all possible paths to see if any of them lead to an accept state. This seems to be a more complex task than simulating a DFA, where the next state is uniquely determined by the current state and input character.

Now a surprising fact is that, both DFAs and NFAs can recognize the same class of languages, i.e., regular languages. In fact, for every NFA, there exists a DFA that recognizes the same language. A rough idea of the equivalence is

that it is possible to create a DFA that simulates the NFA by keeping track of all possible states it could currently be in. (See Powerset construction)

Here are a few more examples of finite automata:

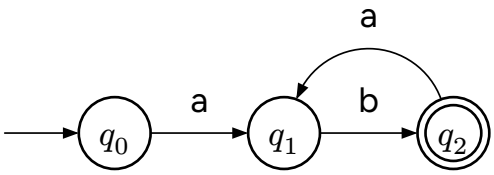


Figure 3: A DFA that accepts strings of the form `(ab)+`.

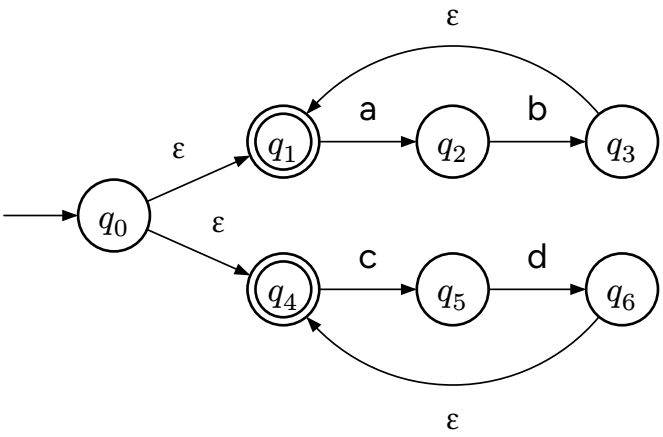


Figure 4: A NFA that accepts strings of the form `(ab)*|(cd)*`.

Figure 4 shows a NFA that accepts strings of the form `(ab)*|(cd)*`. It has **epsilon transitions** (transitions marked with ε), which are transitions that do not consume any input character. This means that the automaton can move from one state to another without reading any character from the input string. In the above example, the automaton can move from state q_0 to either q_1 or q_4 without consuming any input character, allowing it to accept strings that match either `ab` or `cd`. (Epsilon transitions are only allowed in NFAs, not in DFAs, since they add nondeterminism to the automaton.)

Regular Expressions and Finite Automata

To recap what has been said till now:

- Regular languages are a class of formal languages.
- Finite automata are a model of computation that can recognize regular languages. Both DFA and NFA are equivalent in power and can recognize the class of regular languages.
- Regular expressions are a way to describe regular languages.

This means for each regular language, there exists a finite automaton that recognizes it, and there exists a regular expression that describes it.

Formally, a regular expression is defined as follows:

1. The symbol ε is a regular expression that describes the language containing only the empty string.
2. For any character `c`, the string `c` is a regular expression that describes the language containing only the string `c`.
3. If `r1` and `r2` are regular expressions, then the **concatenation** `r1r2` is a regular expression that describes the language containing all strings formed by concatenating a string from the language described by `r1` with a string from the language described by `r2`.
4. If `r1` and `r2` are regular expressions, then the **union** `r1|r2` is a regular expression that describes the language containing all strings that are either in the language described by `r1` or in the language described by `r2`.
5. If `r1` is a regular expression, then the **Kleene star** `r1*` is a regular expression that describes the language containing all strings that can be formed by concatenating zero or more strings from the language described by `r1`.

(While denoting regular expressions, parentheses are used to group expressions, e.g., `(r1|r2)*`. Also though not in the formal definition, it is common to use `+` to denote one or more occurrences of a character or group.)

In the above formal definition of a regular expression, the basic operations are concatenation, union and Kleene star. By definition, regular languages are closed under these operations, thus they are called regular operations.

In order to implement a Regex engine, we can use the following approach:
1. Parse the regular expression to create a syntax tree that represents the structure of the expression.
2. Convert the syntax tree to a finite automaton (either DFA or NFA) that recognizes the language described by the regular expression.
3. Use the finite automaton to match the input string against the regular expression by simulating the transitions of the automaton based on the input characters.

Regular Expressions in the real world

The first uses of regular expressions in practice were in the 1960s, when Ken Thompson implemented regular expressions as a way to search for patterns in text files. He wrote a simple regular expression engine for the Unix text editor `ed`, which later evolved into `grep`, a popular command line tool for searching text files using regular expressions.

As they became more popular, different variants of regexes were developed, each with its own syntax and features. Some of the common features that regex engines support are:

- `.` : Matches any single character except newline. (Wildcard)
- `+` : Matches one or more occurrences of the preceding character or group.
- `?` : Matches zero or one occurrence of the preceding character or group.
- `*` : Matches zero or more occurrences of the preceding character or group.
- `\(` : Backslashes can be used to escape characters. For example, `\(` matches the literal character `(`, and `\)` matches the literal character `)`.
- Character classes: `[abc]` matches characters 'a', 'b' or 'c'; `[a-z]` matches any lowercase character, `[A-Za-z0-9]` matches any alphanumeric character. `[^pqr]` is a negated character class, which matches any character other than the ones in the class.
- Numeric Quantifiers: `{m}` matches exactly `m` occurences of the preceding element. `{m,n}` matches at least `m` and at most `n` occurences of the preceding element.
- Anchors: For search applications, `^` can be used to denote start of the file (or a line), and `$` for the end of the file (or a line).
- Shorthand notations like `\d` denotes `[0-9]`, `\w` denotes `[A-Za-z0-9_]`, and `\s` denotes whitespace characters (space, tab, newline, etc.).

For example,

- `.at` matches any three-character string, ending with "at", like `hat`, `cat`.
- `lo+l` matches any string starting and ending with `l`, and having a sequence of `o`s in between like `lol`, `loool`, `looooool`.
- `\(.*\)` matches any string starting with `(` and ending with `)`.
- `Reg(E|e)xp?` matches the words "RegExp", "RegEx", "Regexp" and "Regex".
- `[A-Za-z0-9]+` matches any alphanumeric string of one or more characters.
- `\d{2}-\d{2}-\d{4}` matches a date in the format `dd-mm-yyyy`, like `01-01-2020`.

All these new features do make regexes more concise, but not more powerful; there exist longer equivalents in the original regex syntax that can express the same patterns.

However, one common extension to regular expressions that does increase its power is the addition of **backreferences**. A backreference allows a regex to refer to a previously matched group. For example, the regex `(.*)\1` matches any string that contains a word followed by the same word again, like `hellohello`.

It is important to note that a regular expression with backreferences is not equivalent to the theoretical regular expressions. It does not recognise a regular language. So, correspondingly, there is no DFA or NFA that is equivalent to it. This added power comes with the cost of efficiency; the best known algorithm for matching strings is of exponential complexity, due to requirement of **backtracking**.

Most modern regex implementations support backreferences, and thus they are usually implemented using backtracking algorithms. This means that the engine tries one path at a time, and if it fails, it backtracks to try another path. This can lead to exponential time complexity in the worst case, but is necessary to support backreferences.

Creating a Regex Engine

Note: The full repository for the regex engine I built and that is outlined below is at https://github.com/dipamsen/regular.

Let us dive into how one may create their own regular expressions engine. As mentioned earlier, the entire workflow contains three steps:

1. Parse the regex into a parse tree.
2. Convert the parsed regex into an D/NFA
3. Simulate the D/NFA on the input string and check if it accepts or rejects it.

The syntax we will support will include `.` (wildcard), quantifiers `+`, `?`, `*` and `|` for union. This is a subset of the syntax in original regular expressions implementations in UNIX tools, but is equal in expressive power to the original regex syntax. We will not support backreferences, as they are not a part of regular expressions in the theoretical sense.

Parsing the Regex

This is the first step in our workflow. Now there are many ways to go about this. This goes into theory of compilers, which I won't delve into in this post (mainly because I don't know much about it), but I implemented a **recursive descent** parser based on a grammar:

```
Regex          ::= Alternation
Alternation    ::= Concatenation '|' ... '|' Concatenation
Concatenation  ::= QuantifiedAtom ... QuantifiedAtom
QuantifiedAtom ::= Atom Quantifier | Atom
Atom           ::= Character | Group | Dot
Dot            ::= '.'
Quantifier     ::= '*' | '+' | '?'
Group          ::= '(' Alternation ')'
```

(Note that here 'Alternation' represents a union operation. 'Atom' refers to a single unit on which quantifiers can be applied.)

The above block of text is a simplified version of what's called a 'Grammar', this one is more specifically is a **context-free grammar** (CFGs). This corresponds to another level in the Chomsky hierarchy: Above regular languages, we have **context free languages** (CFLs) which are recognised by CFGs. The language of regular expressions themselves is a CFL, so we need to write a CFL parser.

An interesting thing to note is that the grammar is based on the precedence order of operations. The normal precedence is quantifiers have the highest precendence, then concatenation, and finally union. The symbols in the grammar are structured in a similar hierarchy (union in the outermost level, next level being concatenation, and quantifiers being applied in the next level).

The complete parser data structures and code is in `parser.h` and `parser.c` - we have functions for each non-terminal symbol in the grammar, and they call each other according to the grammar rules, processing the input (i.e. the regex string) from left to right.

Let us assume that we are done with parsing, and we are left with a parse tree which represents the structure of the regular expression.

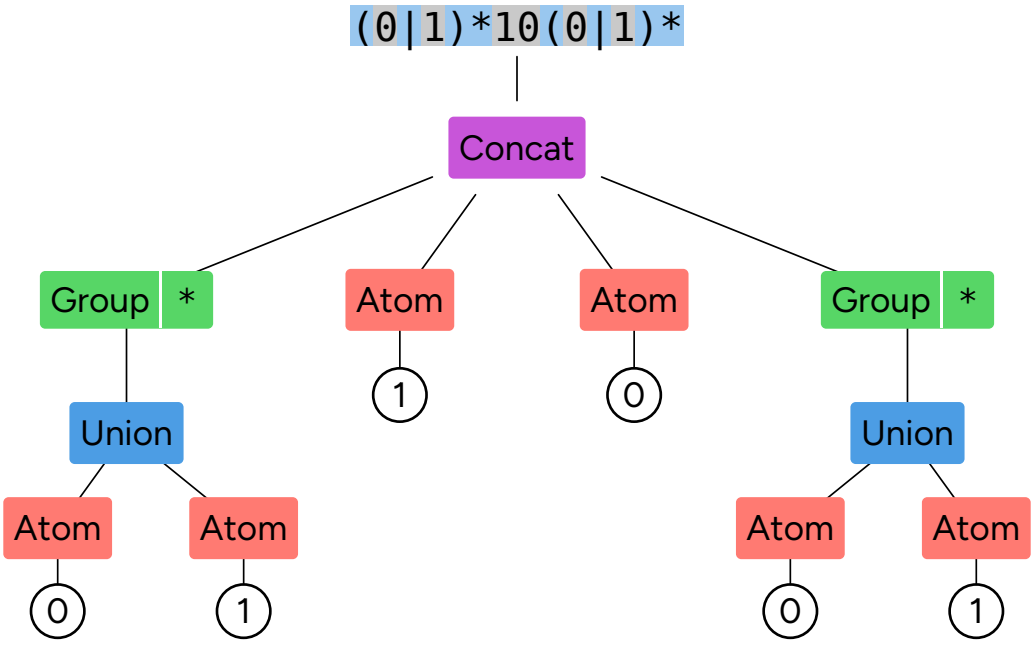


Figure 5: Simplified parse tree for a regular expression

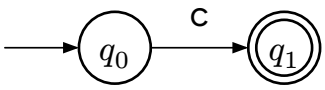
Creating a Finite Automaton

Next, we need a general way to convert any given (parsed) regular expression, into an NFA or a DFA. One way to do this is to use a **Thompson's construction**

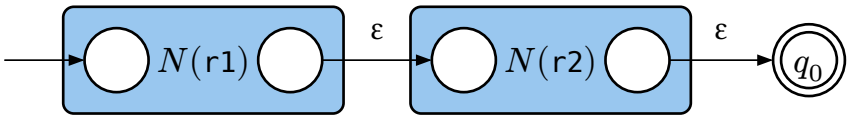
algorithm, which constructs an NFA from a regular expression. This algorithm is described below.

The idea of the algorithm is to recursively create the NFA for the subexpressions, and then combine them based on the operations in the regular expression. The algorithm defines how to construct NFAs for each atomic node in the parse tree, and how to combine them based on the operations. ($N(r)$ represents the NFA for the regular expression `r`.)

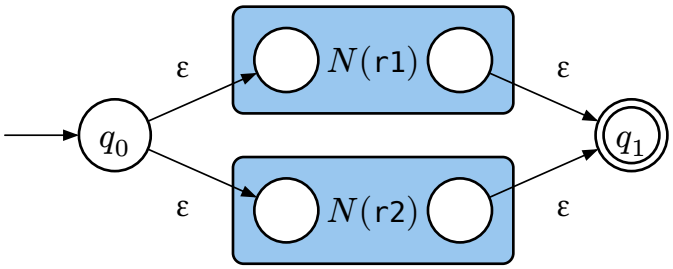
1. For a character `c`:



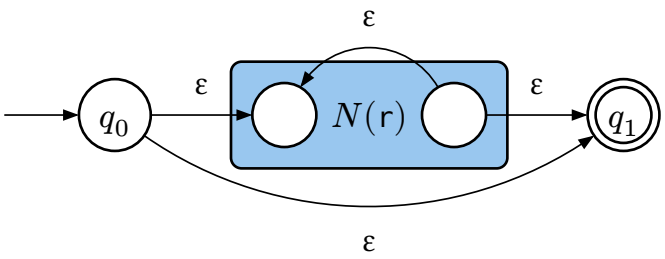
2. For a concatenation of two subexpressions `r1` and `r2` (`r1r2`):



3. For a union of two subexpressions `r1` and `r2` (`r1|r2`):



4. For a Kleene star of a subexpression `r` (`r*`):



By combining these constructions based on the structure of the regular expression, we can construct an NFA that recognizes the language described by the regular expression.

Here's an example of how the above constructions can be used to create an NFA from a regex.

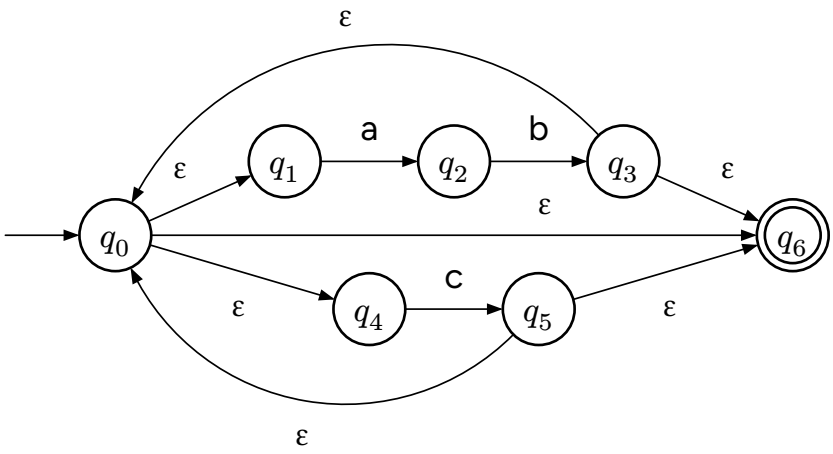


Figure 6: (Simplified) Thompson's NFA for the regex `(ab|c)*`.

Thompson's algorithm only provides these few constructions, since these are the regular operations that are used to define a pure regular expression. For our implementation, we need a few more operations (like +), which can be similarly constructed. We will discuss these later.

Simulating the NFA

Now that we have the NFA, our next question is how to simulate running the input on the NFA. We have a few options here:

1. Convert the NFA into a DFA (using powerset construction), and then simulate the DFA on the input string. Simulating a DFA is straightforward, since it has a unique next state for each input character.
2. Directly simulate the NFA on the input string.

Let us briefly discuss conversion to DFA. The powerset construction algorithm takes the NFA and creates a DFA that simulates it. The states of the DFA are sets of states of the NFA, and the transitions are defined based on the transitions of the NFA. Note that this can lead to an exponential blowup in the number of states, thus it is order $O(2^m)$ (for creating the DFA), where m is the number of states in the NFA. However, once the DFA is created, it can be simulated in linear time $O(n)$, where n is the length of the input string.

Due to the exponential blowup, this approach is not the best for large NFAs. Instead, we will focus on the second approach, which is to directly simulate the NFA.

There are two main ways to simulate the NFA:

1. **Backtracking**: At a given state, if there are multiple possible transitions for the next character, try each transition one by one, and backtrack if a

transition fails. If the NFA is imagined as a tree, this is similar to a depth-first search (DFS) traversal of the tree.

2. **Parallel simulation**: Keep track of all possible states the NFA could be in at any point in time, and simulate all of them in parallel. If we imagine the NFA as a tree, this is similar to a breadth-first search (BFS) traversal of the tree.

Under the tree analogy, it may seem as if both of these approaches are equivalent. However, as it turns out, they are not. The backtracking approach can lead to exponential time complexity in the worst case, while the parallel simulation approach gives a worst case linear time guarantee. Let us see some examples to understand why this might be the case.

For example, consider the regex, `(0|1)*000(0|1)*`. An NFA that recognises this language is shown below.

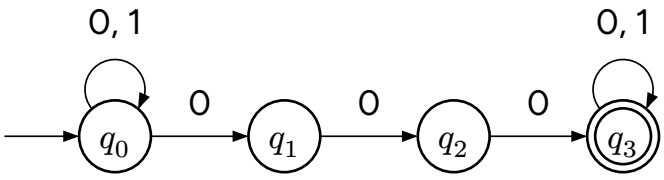


Figure 7: NFA for the language `(0|1)*000(0|1)*`

If we take the input `0001100`, we can visualise the running of a program that will try all paths by backtracking upon failure.

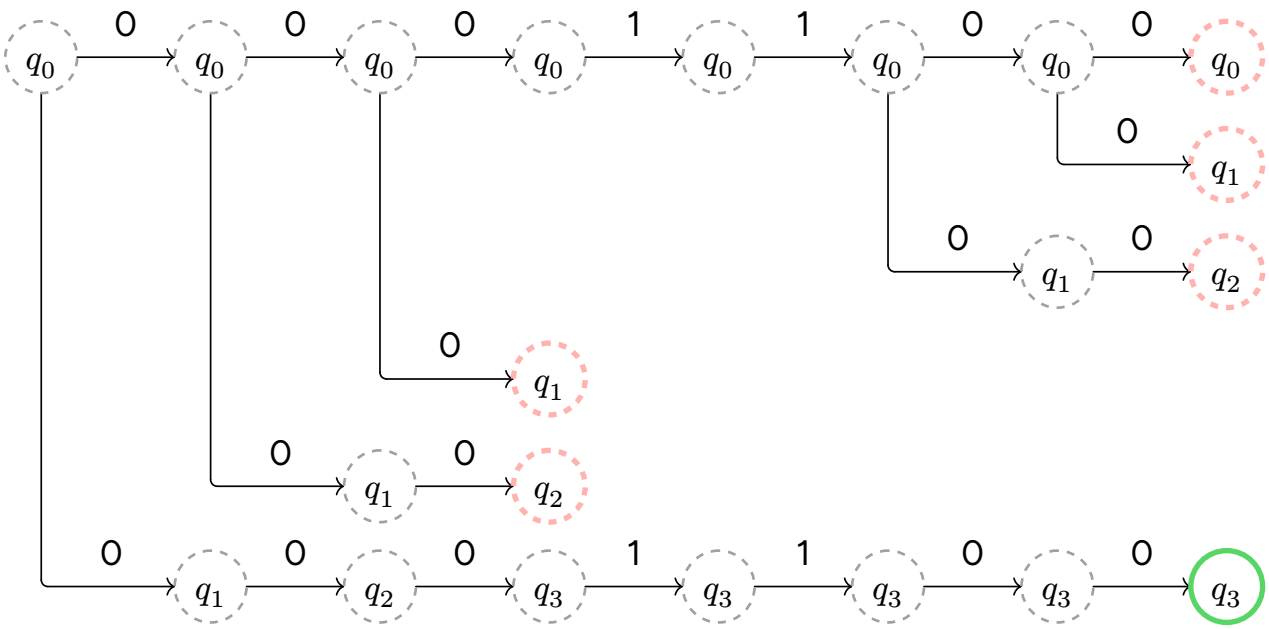


Figure 8: Backtracking scheme for input string `0001100`

In the above figure, all possible paths in the NFA are shown. A red outlined state represents a dead-end, i.e. either the entire string has been processed (and is on a non-accepting state), or the next character of the string has no

transition from the current state. In this situation, the engine should backtrack to the last position where there were multiple possibilities, and continue on another path.

On the other hand, how would we run the simulation in parallel? We need to keep track of a set of states (the active set). The active set starts with just the start state. Upon reading a character, we build a set of all possible states that one could visit from any one of the states currently in the set.

So in this example, initially the active set is $\{q_0\}$. Upon reading a 0, state q_0 can either transition to itself, or to q_1. Thus our new active set becomes $\{q_0, q_1\}$. Similarly, we simulate the state upto the end of the input, and since an accept state q_3 is finally present in the active set, this string is accepted by the NFA.

$$\{q_0\} \xrightarrow{0} \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} \xrightarrow{0} \begin{Bmatrix} q_0 \\ q_1 \\ q_2 \end{Bmatrix} \xrightarrow{0} \begin{Bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{Bmatrix} \xrightarrow{1} \begin{Bmatrix} q_0 \\ q_3 \end{Bmatrix} \xrightarrow{1} \begin{Bmatrix} q_0 \\ q_3 \end{Bmatrix} \xrightarrow{0} \begin{Bmatrix} q_0 \\ q_1 \\ q_3 \end{Bmatrix} \xrightarrow{0} \begin{Bmatrix} q_0 \\ q_1 \\ q_2 \\ \boldsymbol{q_3} \end{Bmatrix}$$

Figure 9: Parallel simulation of NFA on input string `0001100`

Now, if you carefully observe Figure 8 and Figure 9, it is clear that each 'column' in Figure 8 forms the corresponding active sets in Figure 9. This is expected, since both of them simulate running the NFA, but in different ways. If we take 'total number of states to be stored' as a crude measure of complexity, both approaches give equal complexity; at least for this example.

Furthermore, you may notice that the input string and the backtracking order is chosen such that the backtracker must explore all possibilities to find the matching path. For a different input string, it is possible that the backtracking algorithm may find the match without having to look at every possibility, and maybe even in the first try itself! However the parallel simulator cannot do this, since it parallelly simulates every path regardless of the type of input.

How then, can we justify backtracking being less efficient in the worst case?

Let's take a look at a different example. Look at the NFA given in Figure 10 (equivalent to `a*b?a*c`). Let us try simulating it on the input string `aaaa`.

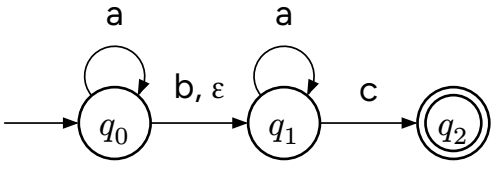


Figure 10: An example NFA

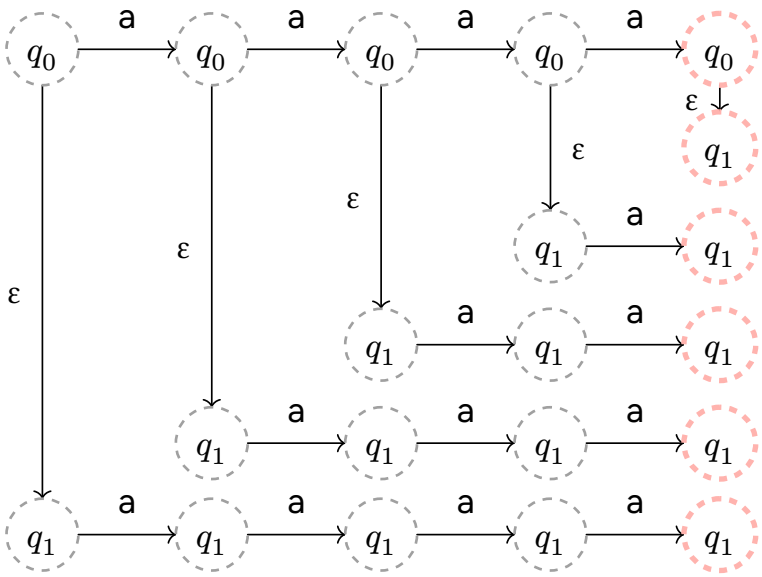


Figure 11: Backtracking scheme for input string aaaa

In this example, the string is not accepted by the NFA. Unlike the previous case, this NFA also has epsilon transitions, so the backtracker must also take them into account as possibilities. If we look into the specifics of this example, the different paths that the backtracker explores corresponds to the choice of when to take the ε path.

How would our parallel simulator handle epsilon transitions? Firstly, the set will start not just with q_0, but with ε-closure(q_0).

$$\varepsilon\text{-closure}(q) \stackrel{\text{def}}{=} \{\text{all states accessible from } q \text{ via } \varepsilon \text{ transitions}\}$$

In this case, the set will start out as $\{q_0, q_1\}$. Now, whenever we add some state to the set, we will also add the states in its ε-closure. This will ensure that at any step, the set holds every possible state the NFA could be in after reading the corresponding part of the input.

Here's the state sets for the parallel simulation:

$$\begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} \xrightarrow{a} \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} \xrightarrow{a} \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} \xrightarrow{a} \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} \xrightarrow{a} \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix}$$

Figure 12: Parallel simulation of NFA on input string aaaa

In this case, we have to store less number of states (in total) while doing a parallel simulation. From this, we can find a time complexity bound for the parallel simulator: If we take the regex/NFA to be constant, and the length of the input to be n, then the parallel matching algorithm is order $O(n)$, since at each step it does at most constant work. (If the NFA has m states, the active set will at most be holding m states at once, which is a constant.)

This is true, even if the number of valid paths through the NFA grows exponentially or polynomially (as is the case here). The key idea is this: **If the input string has some prefix which the NFA can match in more than one way, the backtracker explores each of these paths separately. On the other hand, the parallel simulator will merge them into a single path and only explore its subtree once.** This is inherent in its design; the active set cannot contain duplicate states.

This means that even if the number of valid paths grows nonlinearly, the parallel simulator avoids redundant work by collapsing equivalent states — ensuring linear time behavior.

Let's look at an extreme example. Consider the regex `(a|aa)*b`. If we take the input string to be something like `aaaaaaaaaaaaaaaaaaaaaaa`, it is clear that there is an explosion in the number of paths for the backtracker, since the string of a's can be matched in many different ways according to the regex, grouping 1 or 2 a's at a time. However, the parallel simulator will still run linear to the size of the input, as we have shown above. (You can check this for yourselves, by constructing the NFA for this regex and simulating it both ways by hand.)

In fact, most regex implementations in modern programming languages use backtracking, since they support more features than a normal regex allows, like backreferences. Therefore, they are vulnerable to this kind of explosion in the number of possibilities while backtracking on an input on a regex. This is called **catastrophic backtracking**.

You can try this on your own: Open any web browser, open its JavaScript console, and copy and paste the code below, and see the results for yourselves. (after running the code, wait a few seconds for the output to appear)

```js
console.time();
/(a|aa)*b/.test("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
console.timeEnd();
```

(Warning: If you increase the input string length more, it might hang your PC and even crash your browser!)

If you are thinking that this can be exploited as a security vulnerability; by making a program run a particular regex on some carefully chosen input, one could crash the program - you'd be right! This is a real security attack, known

as **ReDoS** - **Regular Expression Denial of Service** attack. (See here for a list of some vulnerabilities found in real world applications: ReDoS CVEs)

Fortunately, this is something we don't have to worry about. We will use parallel simulation for our regex engine, so it won't be vulnerable to catastrophic backtracking. Furthermore, it will be able to handle way longer strings in this example, than the one where JavaScript struggles to find the answer!

At this point, we have understood the theory behind constructing an NFA and simulating it. Let's look at how we can implement it now!

Implementation Details

This section discusses my implementation of the above ideas in C. This implementation is heavily inspired by Russ Cox's implementation, which in turn is based on Ken Thompson's original Regex engine which compiled the regular expression into machine code.

NFA Construction

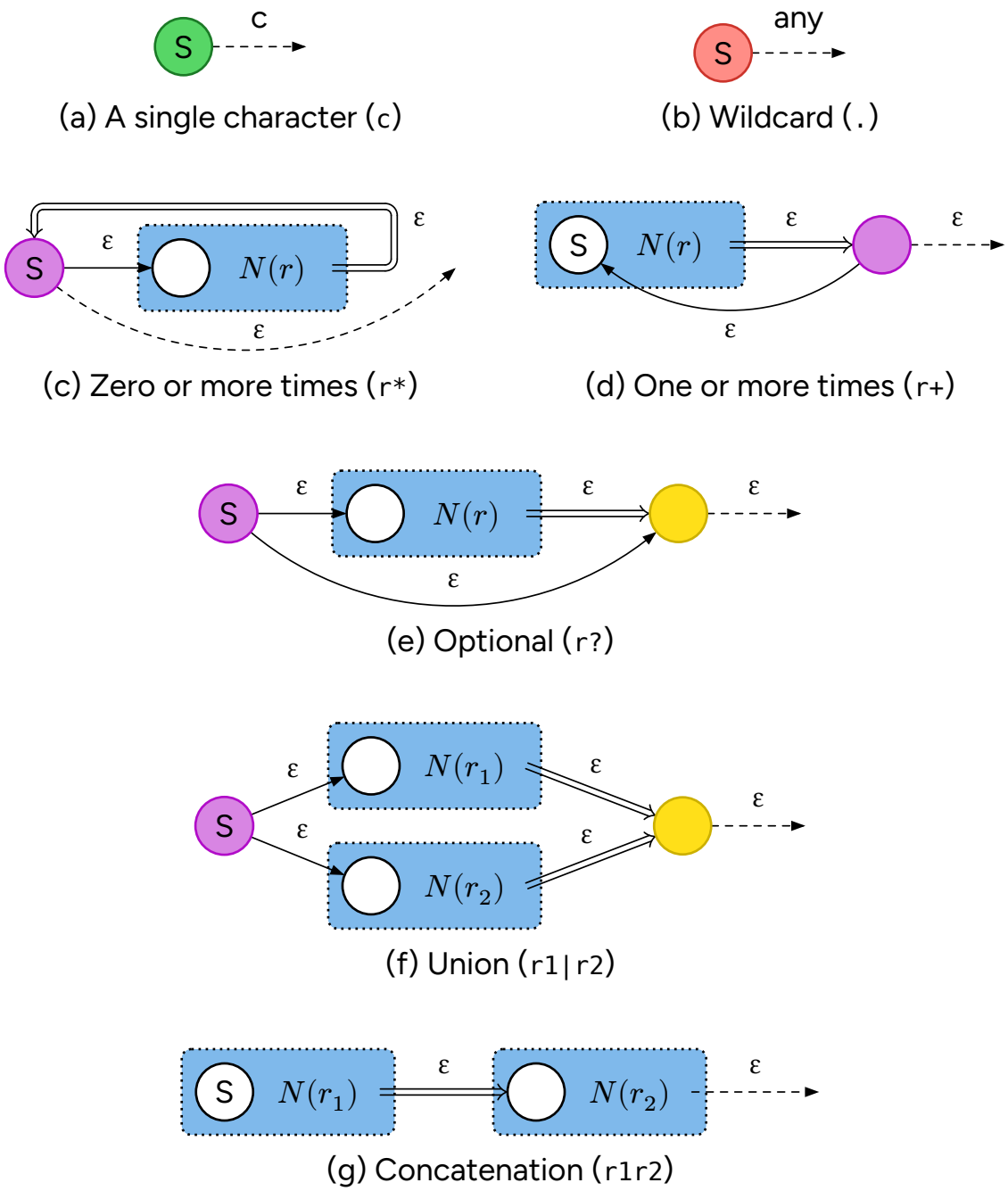
Instead of modelling a general NFA in C, we will observe some properties that we get from Thompson's construction, and model such a restrictive NFA.

Firstly, the partial NFAs we will create will not have end states, instead they will have dangling pointers for the last transition, this way they can be interconnected easily.

Figure 13 shows the partial NFAs for different operations that we may find in the regex string. Here, "S" denotes the start state of the partial NFA, and a dashed arrow represents the outgoing dangling pointer. Blocks labelled as $N(r)$ represent the partial NFAs corresponding to the subexprssions; they have their own start state, and an outgoing dangling pointer. A double arrow ($\Longrightarrow$) represents patching, i.e. connecting the dangling pointer to a state.

Now, if we follow the above construction, we can restrict our states to be of specific forms, rather than any general NFA state. Every state that the construction creates can be classified as one of the following types (correspondingly colored in the figure):

- **Character State** (green): A state with a single outgoing transition, which can be followed when the character is `c`.
- **Wildcard State** (red): A state with a single outgoing transition, which can be followed upon reading any character.
- **Split State** (purple): A state with two outgoing transitions, both of them being epsilon transitions.



(a) A single character (`c`)

(b) Wildcard (`.`)

(c) Zero or more times (`r*`)

(d) One or more times (`r+`)

(e) Optional (`r?`)

(f) Union (`r1|r2`)

(g) Concatenation (`r1r2`)

Figure 13: Different constructions of partial NFAs

- **Epsilon State** (yellow): A state with one outgoing epsilon transition.

What does this mean for our C implementation? This means that every state will have at most 2 outgoing transitions, and further, the state itself can hold its transitions - we won't need separate data structures for storing states and transitions. Here are the data structures that we will define for the construction (`construction.h`):

c

```c
#define EPSILON 256
#define SPLIT 257
```

```c
#define MATCH 258
#define DOT 259

typedef struct State {
    int ch;
    struct State *out;
    struct State *out1;
} State;

typedef struct Fragment {
    struct State *start;
    struct State **out;
} Fragment;
```

Here, the `State` struct corresponds to a state in the NFA, and `ch` stores a character (0-255) if it is a character state, or constants corresponding to the type of state. We have defined one more convinence state type `MATCH`, which we can patch onto the complete partial NFA, so that it can be used to check whether the NFA accepts or rejects the input string. The `State` also holds two State pointers, corresponding to its transitions.

We have one more struct called `Fragment`: This represents any general partial NFA. What it stores is its start state, as well the dangling out pointer of the partial NFA. Essentially, the way it works is that it is a double pointer which points to `State.out`, which will be `NULL`, and when we want to patch it, we will do

c

```c
void patch(State **out, State *target) {
    *out = target;
}

patch(frag.out, some_state)
```

and this will ensure to make the correct connections within states internally (It will connect the state having the dangling pointer to `some_state`).

Now all thats left to do is to implement different functions for each construction in Figure 13, and connect them together according to the structure of the input regex (that we have already parsed). The complete code is in `construction.c`, here are some of the constructions, modified for clarity.

c

```c
Fragment zero_or_more(Fragment frag) {
  State *loop = new_state(SPLIT, frag.start, NULL);
  patch(frag.out, loop);
  return (Fragment){loop, &loop->out1};
```

```
}

Fragment one_or_more(Fragment frag) {
  State *loop = new_state(SPLIT, frag.start, NULL);
  patch(frag.out, loop);
  return (Fragment){frag.start, &loop->out1};
}

Fragment optional(Fragment frag) {
  State *new_end = new_state(EPSILON, NULL, NULL);
  State *new_start = new_state(SPLIT, frag.start, new_end);
  patch(frag.out, new_end);
  return (Fragment){new_start, &new_end->out};
}
```

Matching on an input

Now, the time has come to simulate the NFA on a given input string. From the extensive discussion on this topic, we know that we will simulate the NFA in parallel, keeping track of all states that the NFA could be in at a certain step.

Data structuring is pretty easy, we just need a list of states:

c

```
typedef struct StateList {
    State *states[MAX_STATES];
    int count;
} StateList;
```

(Here, we are not using a Set data structure which would be more efficient, yet our algorithm is still linear time since our state list can only ever have at most a constant number of states.)

Before writing the actual match function, we define some helper functions:

c

```
void add_state(StateList *list, State *s) {
  if (s == NULL)
    return;
  for (int i = 0; i < list->count; i++) {
    // do not add duplicates
    if (list->states[i] == s)
      return;
  }
  if (s->ch == SPLIT || s->ch == EPSILON) {
    add_state(list, s->out);
    add_state(list, s->out1);
  } else {
```

```
13      list->states[list->count++] = s;
14    }
15  }
```

`add_state()` adds the given state to the state list, or if its a state with epsilon transitions, the states that can be reached from the given state via epsilon transitions. (Due to the restricted structure of our NFA, we don't need to add the state itself in this case).

c

```c
void step(StateList *current, int ch, StateList *next) {
  next->count = 0;
  for (int i = 0; i < current->count; i++) {
    State *s = current->states[i];
    // for every state s in the current state list
    if (s->ch == DOT || s->ch == ch) {
      // add the states which can be reached by reading ch
      add_state(next, s->out);
    }
  }
}
```

`step()` is the function that reads the next character from the input, and creates the new state list, based on the current state list.

c

```c
int is_match(StateList *current) {
  for (int i = 0; i < current->count; i++) {
    if (current->states[i]->ch == MATCH)
      return 1;
  }
  return 0;
}
```

This is very self-explanatory, it checks if the current state list has the `MATCH` state, which indicates success.

Finally, here's the `match` function in all its glory:

c

```c
int match(State *start, char *input) {
    StateList current;
    StateList next;
    current.count = 0;
    next.count = 0;

    add_state(&current, start);

```

```
    for (; *input; input++) {
        step(&current, *input, &next);
        // swap current and next
        StateList temp = current;
        current = next;
        next = temp;
    }

    return is_match(&current);
}
```

What this function does is maintains two state lists - `current` and `next`. It adds the start state into the `current` list, then construct the `next` list by reading the next character from the input. Then, it swaps the two lists and continues doing so until the end of input is reached. At this point, we return true if the `MATCH` state is present in the current state list.

Alright, we have now implemented the regex engine! You can visit the repo, and you can run it with the command `make all` and `./main`. One of the test cases it runs is

c

```
test("(a|aa)*b", "aaaaaaaaaaaaaaa...aaaaaaaaaa", false);
```

which is an input string 230 characters long (removed here for brevity) – way longer than the one JS was struggling with – and it runs in less than a second!

Conclusion

In this post, we have discussed some history, theory and implementation of a regex engine. We have seen how to construct an NFA from a regex, and how to simulate it in parallel on an input string. We also saw how this approach avoids catastrophic backtracking, and is thus more efficient than the backtracking approach.

As a final note, I would like to mention that though the backtracking approach is vulnerable to catastrophic backtracking, it is still used in most modern regex engines. This approach provides much greater flexibility and expressive power, allowing more complex regex extensions. On the other hand, using a parallel simulation avoids this entirely, there are no 'pathological' inputs that can cause the engine to hang or crash. But it can only express traditional regular languages.

If you are interested in learning more about regex engines, do check out the resources in the References below.

References

- Russ Cox - Implementing Regular Expressions: An amazing series of articles on history and implementation of regex engines.
  - Regular Expression Matching Can Be Simple and Fast
- Grammars, parsing and recursive descent by Kay Lack
- How regexes got catastrophic by Kay Lack
- Relevant Wikipedia articles:
  - Regular Expression
  - Thompson's Construction
  - Nondeterministic Finite Automaton
  - ReDoS
- Introduction to the Theory of Computation by Michael Sipser, Chapter 1: Regular Languages