Course Project 4
Backreferences and NP-Completeness

CSE 30151 Spring 2025

Version of 2024-12-31
Due date: Tuesday 2025-04-22

In this project, we return to `grep` and allow backreferences inside regular expressions themselves. For example, in thef regular expression

```
((a|b)*)\g<1>
```

the backreference (`\g<1>`) must match a substring equal to the contents of group 1. Thus this expression recognizes the language $\{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$. This new feature takes `grep` beyond regular languages and even beyond context-free languages. In fact, you'll show that this makes `grep` NP-complete by implementing a SAT solver with it.

You will need a correct solution for CP3 to complete this project. If your CP3 doesn't work correctly (or you just weren't happy with it), you may use the official solution or another team's solution, as long as you properly cite your source.

Getting started

The project repository should include the following files:

```
bin/
  parse_re
  bgrep
  solve_sat
examples/
  sipser-phi.cnf
  unsat-2.cnf
  random-1.cnf
  ...
tests/
  test-cp4.sh
cp4/
```

Please place the programs that you write into the `cp4/` subdirectory.

1 Backreferences: Syntax

Note: Part 1 should be done before Part 2, but Part 3 can be worked on independently of Parts 1–2.

Extend your regular expression parser to allow backreferences. The syntax of backreferences is the same as in the replacement strings of CP3. It's probably easiest to treat each backreference as a single terminal symbol, even though it contains multiple characters. That is, let Σ be as defined in CP2, and redefine

$$T = \Sigma \cup \{\,\texttt{|}, \texttt{*}, \texttt{(}, \texttt{)}, \dashv\} \cup \{\texttt{\textbackslash g<}k\texttt{>} \mid k \geq 1\}.$$

To convert a string into a sequence of terminal symbols, add a preprocessing step (called a *lexer*) that inputs a string and outputs a list of tokens, each of which is an element of T.

The grammar for regular expressions must now include backreferences, and treats them similarly to ordinary symbols:

$$\mathsf{P} \rightarrow \texttt{\textbackslash g<}k\texttt{>} \qquad\qquad \text{for } k \geq 1$$

Correspondingly, extend the parse table with these new rows:

input		parse stack		semantic stack	
read	peek	pop	push	pop	push
ε	$\{\texttt{\textbackslash g<}k\texttt{>}\}$	P	\g<k> backref[k]	ε	ε
\g<k>		\g<k>	ε	ε	ε
ε		backref[k]	ε	ε	backref[k]

Also note that several transitions have "peek" sets that include T, but the definition of T has changed.

Here are some example regular expressions with their abstract syntax trees:

expression	syntax tree
\g<99>	backref[99]
\g<99>*	star(backref[99])

To test your extension, update parse_re from CP2 to handle backreferences. For example:

```
$ cp4/parse_re '(a)\g<1>\g<1>'
concat(concat(group[1](symbol[a]),backref[1]),backref[1])
```

(If you want bin/parse_re to have this behavior, pass it the -g -b options.) Test by running tests/test-cp4.sh.

2 Backreferences: Semantics

To understand backreferences in more detail, consider the expression

(a|b)*\g<1>

Note that, unlike the example at the beginning of this document, the star is now outside the group. If group 1 matches multiple substrings, then, just as in CP3, it's only the last substring whose contents are used. Thus abb matches but aba does not.

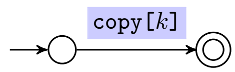
On the other hand, if group 1 does not match any substrings, then nothing (not even the empty string) can match \g<1>. So, if the input string is ε, group 1 does not match any substring. So \g<1> cannot match anything, so the input string does not match.

Here's another example:

`(aaa*)\g<1>\g<1>*`

This recognizes the language $\{\mathtt{a}^n \mid n \text{ is not prime}\}$. Do you see why?

There's no speed requirement for `bgrep`, so you have considerable latitude in how you implement backreferences. You're not even required to continue using NFAs, but if you do, you can convert each backreference \g<k> into the following NFA:



where copy[k] is a special transition, like open[k] and close[k] from CP3.

In CP1, you wrote a function to test whether a NFA M accepts a string w using a search through the graph of configurations (q, i), where $q \in Q$ and $0 \leq i \leq |w|$. Suppose that the current configuration is (q, i) and the contents of group k is g_k. If there is a transition $q \xrightarrow{\texttt{copy}[k]} r$, we can check whether $w_i \cdots w_{i+|g_k|-1} = g_k$. If so, we can create a new configuration $(r, i + |g_k|)$. But to do this, we need to be able to know what g_k is, and our configurations don't contain this information. So, we can redefine configurations to include information about where groups have been opened and closed so far.

For example, consider again the regular expression `(aaa*)\g<1>\g<1>*`. This is equivalent to the NFA shown in Figure 1. On string `aaaaaa`, the graph of configurations is shown in Figure 2. Each node is a configuration, which now includes not only a state and string position, but also information about the start and end of each group if it is known. Note that there are now two configurations for state q_6 and position 6, because there are two ways to get there that have two different contents for group 1.

Modify your NFA matching function to handle backreferences. You can use the method sketched above, or some other method. Depending on how much information you put into configurations, you may or may not need to watch out that regular expressions like `()*` don't cause an infinite loop. (The test script includes a test for this, so if the tests don't hang, then there's nothing to worry about.)

Finally, write a program called `bgrep` (backtracking `grep`) that has the same usage as `agrep` but allows backreferences inside regular expressions, as described above. Run `tests/test-cp4.sh` to test it.

3 SAT solver

Adding backreferences to regular expressions increases their power a lot; in fact, it makes matching NP-complete. Write a program that demonstrates this by reducing Boolean satisfiability to regular expression matching with backreferences.

`sat_to_re` *`cnf-file regexp-file string-file`*

- *`cnf-file`*: name of file containing formula ϕ in conjunctive normal form (see below)
- *`regexp-file`*: name of file to write regular expression α to (see below)
- *`string-file`*: name of file to write string w to (see below)
- Effect: Write α and w such that ϕ is satisfiable if and only if w matches α.

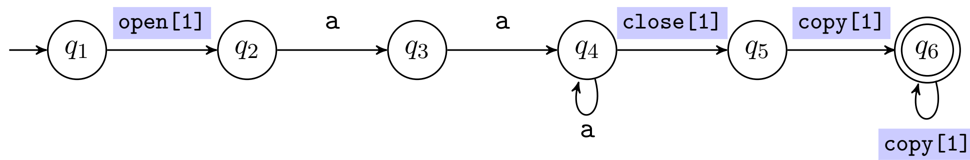


Figure 1: NFA equivalent to the regular expression `(aaa*)\g<1>\g<1>*`. Useless epsilon transitions are omitted for simplicity.

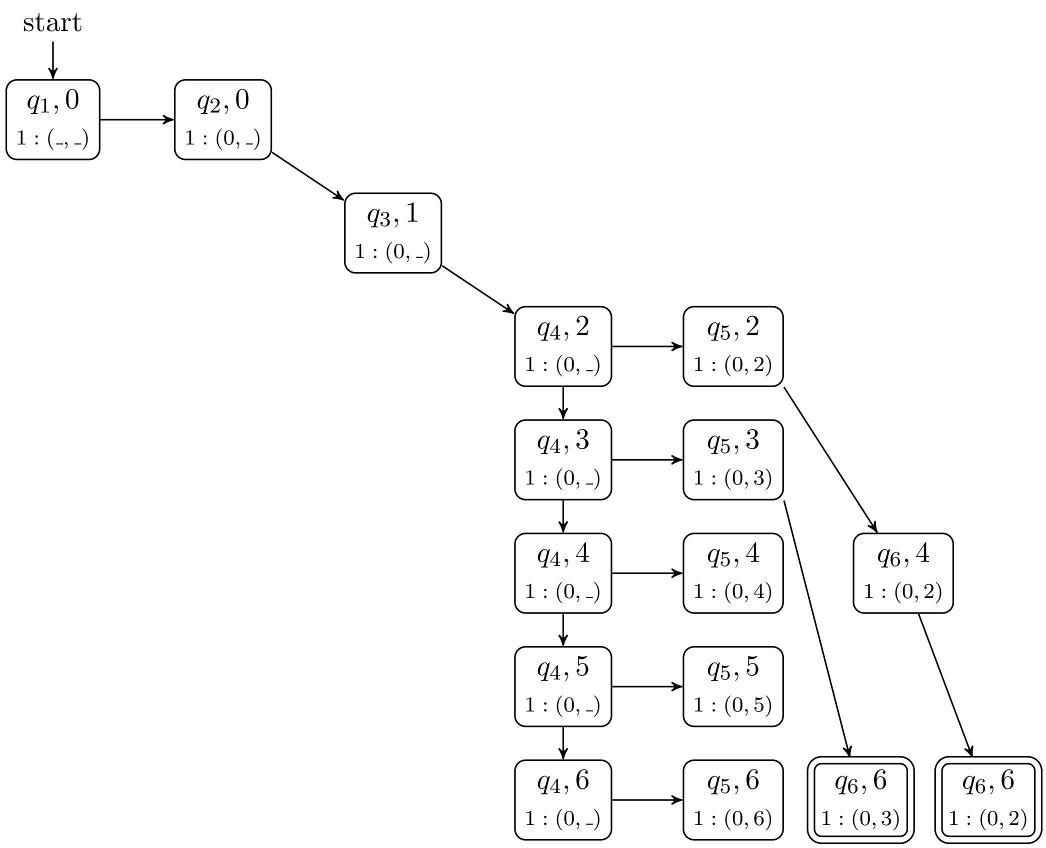


Figure 2: Graph of configuration for the NFA of Figure 1 and the input string `aaaaaa`.

The *`cnf-file`* has one line per clause.[1] Each line is a white-space separated list of integers. An integer $i > 0$ stands for the literal x_i, and an integer $i < 0$ stands for the literal $\overline{x_i}$. For example, the formula $(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$ is specified by the file (`examples/sipser-phi.cnf`):

```
1 1 2
-1 -2 -2
-1 2 2
```

If the formula is satisfiable, then `bgrep` run on the regular expression and string should print something:

```
$ bin/solve_sat examples/sipser-phi.cnf
satisfiable
$ cp4/sat_to_re examples/sipser-phi.cnf regexp string
$ bin/bgrep -f regexp string
something
```

(For this to work, you should use our `bgrep`. The `-f` option to `bgrep` tells it to read the regular expression from a file instead of the command line.) But if the formula is satisfiable, then `bgrep` run on the regular expression and string should print nothing:

```
$ solve_sat examples/unsat-2.cnf
unsatisfiable
$ cp4/sat_to_re examples/unsat-2.cnf regexp string
$ bin/bgrep -f regexp string
```

The test script `tests/test-cp4.sh` does this for several example formulas.

The test script also measures the running time of your program. Your program must run in polynomial time; if it does, then this curve should look roughly linear:

```
n=   4096 t=  0.08    *
n=   8192 t=  0.16     *
n=  16384 t=  0.34     *
n=  32768 t=  0.78      *
n=  65536 t=  1.82       *
n= 131072 t=  3.69        *
```

Submission instructions

Your code should build and run on `student`nn`.cse.nd.edu`. The automatic tester will clone your repository, change to its root directory, run `make -C cp4`, and then run `tests/test-cp4.sh`. You're advised to try all of the above steps and ensure that all tests pass.

To submit your work:

1. Push your repository to GitHub.

2. In GitHub, create a new release by clicking on "Releases," then "Draft a new release."

[1]This is a simplified version of the DIMACS CNF format, a standard format for SAT solvers.

3. Fill in "Release title" with `cp4` if you're submitting the whole assignment, `cp4-1` if you're submitting part 1, `cp4-2` if you're submitting part 2, etc.
4. Click on "Choose a tag," then type the same name you used for the release title, then "Create new tag: `cp4`... on publish."
5. Finally, click "Publish Release."

Rubric

Part 1 (`parse_re`)	3
Part 2	
matching backreferences	6
group matches multiple times	3
group matches no times	3
`bgrep`	3
Part 3	
reading CNF formulas	3
correctness	6
polynomial time	3
Total	30