Automata CS345
ReDoS Project Report

Timothy Worthington-Fitnum (20766920)

Table of Contents

1 Introduction

Investigation into the effectiveness of memoization to remove catastrophic backtracking from regular expression (**regex**) matchers. Making use of the regex engine built by Marcin Chwedzczuk, found here https://github.com/marcin-chwedczuk/reng.

This is a Spencer style (backtracking) engine which makes use of a recursive descent parser and a matcher. The recursive descent parser creates an abstract syntax tree (AST) out of the regex we have just created. The general idea of the backtracking engine is very simple, it first itemizes all possible solution candidates(in this cases checks for matches) and returns the first match, if not proceeds through all of the possibilities until it has exhausted the AST or has run out of memory.
Explored in this paper is why the engine could run out of memory, get caught in a possible infinite loop of recursion and how to fix and ensure that the engine avoids any catastrophic backtracking by implementing small fixes and memoization techniques.

2 Bug Fixes

The original engine suffered from catastrophic backtracking especially when fed problematic regexs such as regexs that lead to NFAs with epsilon loops when applying Thompson-like constructions. Why this is an issue and why it causes catastrophic backtracking is explained in more detail later.
The Fix:
The engine stores the current input position in a global variable, before the engine tries to match. The engine checks if it has not advanced in the current position of the input, the engine then forces passed the epsilon transition, as this is where the backtracking gets stuck, and continues to match from the next transition onwards.
This fixes the issue before any memoization has been implemented and used. The use of memoization would also resolve the issue of catastrophic backtracking and is explored in more detail later.
There is also an issue with huge input strings where the engine will run out of stack space. This is simply fixed by setting the Java flag -Xss10M at run time. The speed of the execution is still noteworthy, it is only the stack space required to execute that becomes costly.

3 Implemented

Positive and Negative lookahead has been added to the engine. This gives the regex the ability to ensure that specific strings do not follow on from specific characters. This allows for more advanced form validation amongst other things. Lookaheads are signaled by either (?=) (positive), or (?!) (lookahead) symbols. The engine, once noted, then returns the rules that are needed to be looked ahead. Code snippet below gives a better idea of how:

```
if (lookahead(0, RTokenType.QMARK)) {
  consume(RTokenType.QMARK);
  if (lookahead(0, RTokenType.EQUAl)) {
     consume(RTokenType.EQUAl);
     RAst check = lookAheadCheck();
     refGroup.put(refGroupNum, check);
     refGroupNum++;
     consume(RTokenType.RPAREN);
     return RAst.posLookAhead(check);
```

Figure 3.1;

Backreferences have also been added but are capped at a maximum of 9 referable groups. Backreferences are stored, when a group has been made, in a hashmap. Their group number is the specific key for each group referenced. When a backreference is called in the regex, the specific group that is referred to is therefore retrieved and its regex rules are placed back into the regex for matching.
Backreference form validation has also been implemented while parsing the regex. Simply keeping track of how many groups have been created and then ensuring the back-referenced group exists within the regex.

Encoding schemes determine how to store the result of the regex, and determine the type of data structure that needs to be used.

Memoization has been implemented with the following encoding schemes:

BIT_MAP - making use of a 2-d array like structure where each position contains two attributes. One to indicate if the position has been set already, and therefore we only

have to return the value, and the other one which stores the value that has been memoized which can be retrieved later one.
HASH_TABLE - an associative array abstract data type which maps keys to values. The hash table makes use of a hash function to ensure there are no clashes in entry keys. The engine makes use of a very simple “function”. The hash table stores values with the key entry being the current ID of the RAst node currently operating + “ ” + the current string position. This ensures that no key clashes can occur and makes for easy retrieval of values when they are later needed. The value being stored is the result of the regex matching for the specific node currently operating.
The policies determined which nodes would need to be memoized and allowed for different techniques to be used for each storage type scheme.

The following policies have been implemented:

All - all states are memoized. This results in significantly more overhead than other memoization policies but it also allows for less computations if there are a significant amount of repeats.

IN_DEGREE_GREATER_THAN_1 - all states with a degree greater than 1. Nodes with a degree greater than one are the only ones to be memoized in this policy. The degree of each node can be calculated during parsing but instead are calculated if this policy is used. Iterating through the whole regex and calculating each nodes degree before starting the matching process. Then when matching each nodes degree is checked to be greater than or less than and equal to one. If greater, the node is then memoized. If not, then the node matches without memoization and will need to recompute if it is ever called in the same position in the string again.

ANCESTOR_NODES - state connected to all lower-level states. The implementation in this engine makes all repeat nodes an ancestor node as well as if a Parent Node has a repeat as a child node, the Parent Node is also set as an ancestor node. This is set during parsing and is done by setting a variable to true or false. Then when matching, each node is checked for being an ancestor. If an ancestor, the node it then memoized. If not, then the node matches without memoization and will need to recompute if it is ever called in the same position in the string again.

4 Not-Implemented

The engine also does not support all of the required memoization, it does not utilise the Run Length Encoding memoization scheme.

Memoization of backreferences and lookaheads do not function as required as the check for whether the function call actually being run in the lookahead or backreference, in this implementation, will not have the same parameters. Therefore this will require the lookahead or backreference to be memoized into the current position given that it fits into the memoization policy.

5 Problematic Regular Expressions

Problematic regexs, among others, are those expressions that lead to non-deterministic finite automaton's (NFA) with epsilon loops when applying Thompson-like constructions. Thompson-like construction is the algorithm for transforming a regex into an equivalent NFA.
The issue arises as most engines are implemented in a way that transforms the regex given into a NFA, i.e resembling a NFA. When the engine has to traverse the NFA, any epsilon loops can result in catastrophic backtracking as seen in Figure 3.1 below.

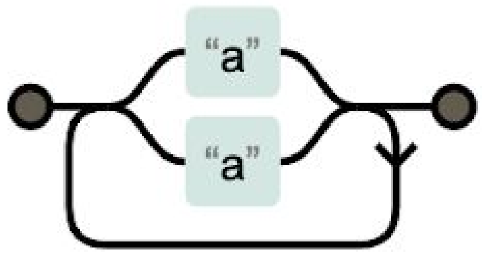


Figure 3.1: **Regex *(a|a)+$* with exponential worst-case behavior.**

The above NFA leads to catastrophic backtracking on the string aa…...ab. For a single input a, there are 2 paths to acceptance, when adding an additional a the paths grow by a factor of 2. So for 8 a's there are 2^8 paths. Now if the string ended in b, the matcher would match all the a's up to the b and fail. The matcher would then try the different path just before the b was found and fail again. The engine would recursively step backwards through each input trying the other path leading to catastrophic backtracking.

As mentioned above, one of the fixes for this issue was keeping a track of the current position in the input and everytime the engine runs the epsilon transition, check if the position in the string has been advanced. If it had not been advanced, which it would not have been in the problematic case, force the engine to move onto the next transition, as to escape it from the epsilon transition check it would continuously be backtracking on.

When working with regexs, to try and avoid problematic and evil regexs, it is best to try and create a longer regex and be more precise about structure and character classes.

This helps to avoid any exponential paths as it enables the engine to decide that it will not match the input at an early checking position than later on after much more computation has been done.This causes good regexs to run faster as they predict their input more accurately. The faster the engine can throw out non-matching input, the fewer cycles are wasted and the more precise the engine is.

6 Memoization

Memoization consists of storing the results of a function call in order to speed up the engine considering the engine will more than likely be using the same function call with the same parameters in the same execution. The initial call results in the computing of the function and storing its result in memory before the engine returns the result. Now any subsequent call to the function with the same parameters, in this instance the same key, the engine only has to fetch the result it had already computed and return that value. This results in maintaining states between function calls as the function call itself behaves on its specific state, such as nodes(computing positions) will have different keys for different positions on the input string.
This speeds the engine up as it no longer needs to compute the same match more than once. It can quickly retrieve the result previously computed, from the hash table or bitmap, without wasting cycles.

Due to the need for storing the function calls, memoization requires additional overhead. There is a tradeoff for matching time and space needed by the engine. This is discussed in more detail below.

7 Space and Time Complexity

The differing memoization policies utilised result in differing amounts of additional overhead required. For instance, when memoizing all nodes, the additional overhead will be the size of the hash table, which is the same as a bitmap, which will have N x M entries, where N is number of nodes and M is length of input string.
Now there is very little trade off in using memoization with respect to easy to compute regex, as the additional space used does not trump the minor speed up it produces.

Where memoization becomes truly effective is when working with complex, problematic and evil regexs with non-linear matching time. As explored in more detail in[2] and [9], memoization becomes highly effective in speeding up the matching time of non-linear regexs. There are many different techniques that can be used and each has its own benefits that are also shown slightly in [9].

When memoizing all nodes the space requirement is at its highest as every function called is memoized, this leads to a possibly N x M memoized function-calls.
In degree greater than 1: Only nodes with degree greater than one are memoized which results in less overhead being used but may not offer as much speed up as certain matching calls could still be required to be called more than one.
Ancestor Nodes: Only ancestor nodes will be memoized which results in less overhead being needed but, like In degree greater than 1, may require matching calls to be computed more than once.

The difference between using the hashtable and the bitmap is that the hashtable is more dynamic and uses the space only when it is required whereas the bitmap requires all the memory upfront.

8 Functionality

Bugs from the original implementation have been fixed and adjusted as mentioned above. The engine now runs more optimally on problematic regexs, yet due to the recursive nature of the engine and the regexs the execution time is still problematic.

The engine works by feeding in a regex and then reading in an input string to find a match. The regex can read positive and negative lookaheads as well as backreferences, as well as the basic capabilities of the original engine.

Backreferences are capped at a maximum of 9 referable groups.
Backreferences make use of a HashMap where the group number is stored as the Key and the Value is the RAst inside of that group. This makes for efficient retrieval when backreferencing onto that group. It also makes for simple putting of the RAst into the HashMap whenever we have finished parsing a group.

9 Testing

Initial engine suffered from matching performance issues on certain types of regexs, and also would not be able to handle problematic regexs such as (a*)* and would hang (compute for an enormous time period), as discussed above.

Below we will explore the effectiveness of the memoization schemes and policies as well as how they differ from each other with regards to speed-up and additional memory usage.
The different techniques and combinations will be run with a few complex and problematic regexs, to get a sense of the real world effectiveness, as well as some

simple regexs where the additional memory usage and speed-up may become more comparable.

The following semantics will be used when reading from the tables:
NOMEM - No memoization.
BMALL - BitMap encoding, memoizing all nodes.
BMDEG - BitMap encoding, memoizing only those nodes with degree greater than 1.
BMANC - BitMap encoding, memoizing only those nodes that are Ancestor nodes.
HTALL - HashTable encoding, memoizing all nodes
HTDEG - HashTable encoding, memoizing only those nodes with degree greater than 1.
HTANC - HashTable encoding, memoizing only those nodes that are nodes.

EM - Estimated additional memory used.
DNC - Did not compute, a cutoff time of 4 minutes (240 000 ms)
All tests are run 3 times, an average time is scored and represented.

System specifications that ran the test can be found here [1].

Experiment 1:

Regular Expression: ^(a|a)*(?!(a|a)*)*b$.
Input String: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Results: No Match found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC
Time (ms)	28.3	37	28	27	27	48.3	64.6

Table 9.1:

Experiment 2:

Regular Expression: ^(a|a)*(?!(a|a)*)*\1(b+)\2$
Input String: aaaaaaaaaaaaaaaaaaaabbbbbbbbbbbb
Results: No Match Found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC

Time (ms)	12 551.3	27	14278.3	27	26.3	14 361.6	26.3

Table 9.2:

Experiment 3:

Regular Expression: ^([a-z]{3,4})(@)\1(.com)$
Input String:abc@abc.com
Results: Match Found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC
Time (ms)	48	26.6	27.6	26.7	29	26.6	29

Table 9.3:

Experiment 4:

Regular Expression: ^.* (.*)\[(.*)\]:.*$
Input String: 50.0.134.125 - - [26/Aug/2014 00:27:41] \"GET / HTTP/1.1\" 200 14 0.0005
Results: No Match found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC
Time (ms)	33.6	28	34.6	29.3	28	34.3	27.6

Table 9.4:

Experiment 5:

Regular Expression: ^(a|a)+q$
Input String: aaaaaaaaaaaaaaaaaaaaaaaaaaaaad
Results: No Match found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC
Time (ms)	189 451	35	215 305.3	31	25.6	219 391.3	27.6

Table 9.5:

Experiment 6:

Regular Expression: ^(([a-z])+.)+[A-Z]([a-z])+$
Input String: aaaaaaaaaaaaaaaaaaaaaaaAaaaaaaaaaaaaaaaaa
Results: Match Found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC
Time (ms)	32	27	32.6	34	25	40.3	26.3

Table 9.6:

Experiment 7:

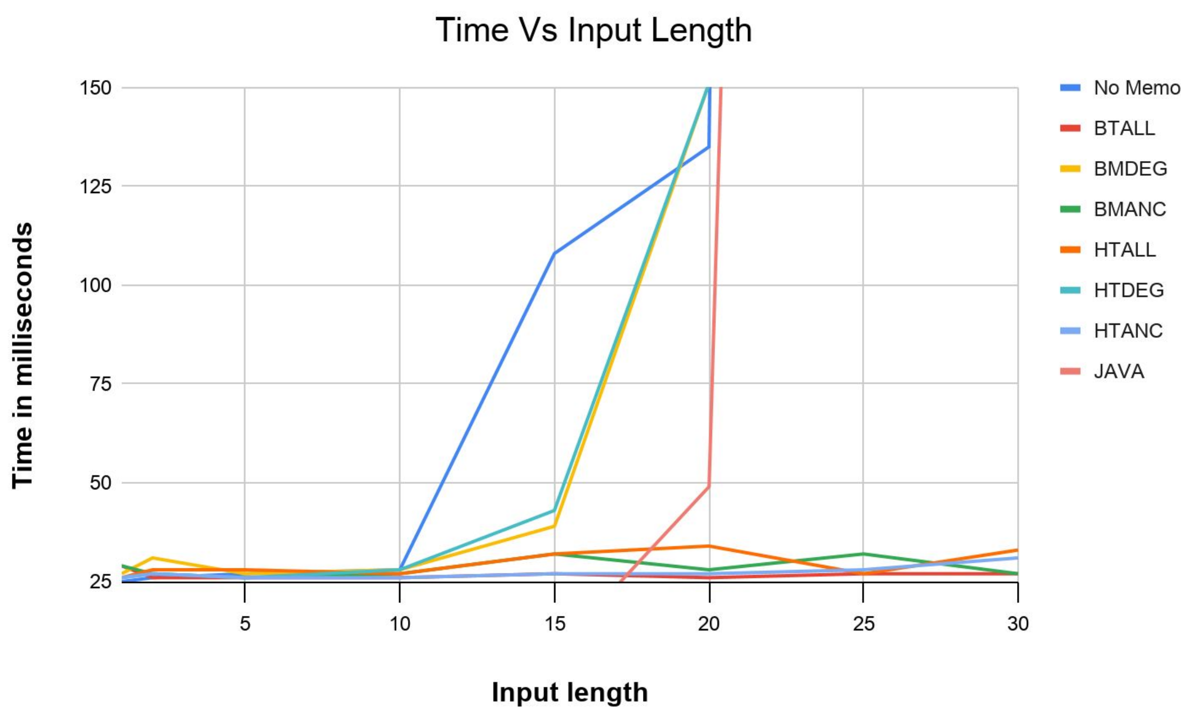
Regular Expression: ^(([a-z])+.)+[A-Z]([a-z])+$
Input String: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Results: No Match Found

Technique	NOMEM	BMALL	BMDEG	BMANC	HTALL	HTDEG	HTANC
Time (ms)	50 855.6	29.3	60 641	34.6	42.3	60 538	31

Table 9.7:

Conclusion:

The engine, using memoization, almost always outperforms no memoization. For certain regexs the engine seems to suffer with In Degree Greater Than 1 policy and as a result is sometimes worse than no memoization in the experiments above. The speed up for most experiments would indicate that the additional space required is worth it. Experiments 2 and 7 show exceptional speed up on all memoization techniques besides the In Degree Greater Than 1 policy. This indicates that the engine is faulty with regards to its calculations with respect to the degree of each node.
Run length encoding is still the most optimal technique and trumps the techniques that have been utilised above.



<u>Figure 9.8: Memoization Techniques vs Java Regex Matcher vs No Memoization</u>

<u>Figure 9.8</u>, was created using data from running the regex ^(a|a)+q$ for the strings d,ad,aaaad,aaaaaaaaad,..., aaaaaaaaaaaaaaaaaaaaaaaaaaaaad. Once can clearly see that the no memoization is exponential as soon as the input string length reaches 10. The Java Regex Matcher also reaches exponential time increase when input string length reaches 18.

10 JAVA REGEX MATCHER

As seen in ***<u>Figure 9.8</u>*,** the Java Regex Matcher is superior to the memoization techniques implemented against the regex **^(a|a)+q$** for string length less than 15. As soon as the string length exceeds 15 (the case aaaaaaaaaaaaaad), the Java Regex Engine matching time climbs exponentially and performs far worse than the memoization techniques.

11 NP-Complete

Using the proof presented by Alfred V. Aho in *<u>Algorithms for Finding Patters in Strings</u>*, matching with backreferences is NP-Complete.

Let’s consider, given a regex pattern consisting of a backreference, say r, and an input string s. We want to answer if the string s has a substring matched by r.

Now if we use a reduction of the vertex-cover problem (VCP), simplification found here vertex-cover problem.
Let E_1, E_2… E_m be subsets of cardinality 2 of some finite set of vertices V. The VCP is to decide, given a positive integer k, if there is a subset V’ of V with at most k elements, such that V`consists are at least one element in each $E_i$. The $E_i$ can be seen as edges of the a graph with V`as being the set of vertices so that each edge contains at least one vertex in V`. The VBP is NP-Complete. Extending this into the pattern-matching problem for backreferences as follows.

Let N be the parenthesized string (n_1 | n_2 |..|n_f) with V = { n_1 , … n_f}. Have # be a unique symbol.
For 1≤ i ≤ k, let

$$x_i = N^* N \% v_i N^* \#$$

Where the v_i’s are distinct names. Similarly for 1≤ i ≤ m, let

$$y_i = N^* N \% w_i N^* \#$$

Where the w_i’s are distinct variable names. For 1≤ i ≤ m, let

$$z_i = w_i^* v_1^* v_2^* \ldots v_k^* w_i^* \#$$

Let *t* be the new regex pattern x_i … $x_k y_1 ... y_m z_1 ... z_m$.
Making a new input string *s*, so that if the *t* matches *s*, then the VCP has a solution of size k.
Let *u* be the string $n_1 n_2 ... n_f$ # repeated k+m times. Let e_i be the string ab# where E_i = {*a*,*b*} for 1≤ i ≤ m. Finally, let *s* be the input string $u e_1 ... e_m$.
Now see that *t* matches *s* if and only if the set of vertices assigned to the $v_1, \ldots, v_k$ forms a vertex cover for the set of edges {$E_1, ..., E_m$}. Thus *t* matches *s* if and only the VCP has a solution of cardinality at most k.

Matching the regex with backreference in nondeterministic polynomial time is easy. Easiest approach to match deterministically is to use backtracking to keep track of the possible substrings of the input that can be matched to *t*.
Because there are $O(n^2)$ possible substrings for each variable in r, where n is length of input. Therefore for k variables in r there are at most $O(n^{2k})$ possible assignments. Therefore matching can be done in at worst $O(n^{2k})$ time.

12 NFA and Regular Expression Ambiguity

An NFA is said to be ambiguous if some string can be accepted by traversing more than one different paths of states and transitions. Ambiguity in regexs is when a regex can produce a string in more than one distinct manner. This becomes problematic when using regexs to match with strings.
When a string can be matched by a regex in more than one way, which way should the string be matched is the problematic question. The major issue does not arise when the regex can perfectly match the string, but more so when the regex can partially or almost fully match but the tail of the string does not match.
Due to the many different ways a regex could possibly match with a string, we can experience catastrophic backtracking when the string is long and is only unmatched towards the tail. Use the following as an example. The regex ^(a*)*$ and input aaaaaaaa….b. The regex will match all the a's up until the b, then due to the backtracking nature of the engine, it will step back to the previous a and try a different path. This will continue to happen until the engine has exhausted all possible paths or has run out of memory.
The issue arises when epsilon transitions are created when converting a regex to an NFA. These are also known as Thompson constructions and can wreak havoc on a system that is not prepared to handle them.

13 Theoretical Shortfalls

Similarly discussed above, shortfalls arise in backtracking engines as greedy quantifiers arise. They will try to match as many valid characters as they possibly can. This can cause major issues with backtracking matchers as they will match to the very end of a string and if a match is not found, backtrack and try to find another match and continue in this loop until they have exhausted the input, given there was no match to be made. This is greedy and trying to get the longest match.
Greedy quantifiers can also match much more than expected and can move to the end of an input steam very quickly. The regex, if no match is found, will then want to jump back to the position that made it progress so far in the first place, generally a kleene star or plus, and give up the last character. This process happens many times and eventually the regex has given up enough characters to find a match that was not intended.

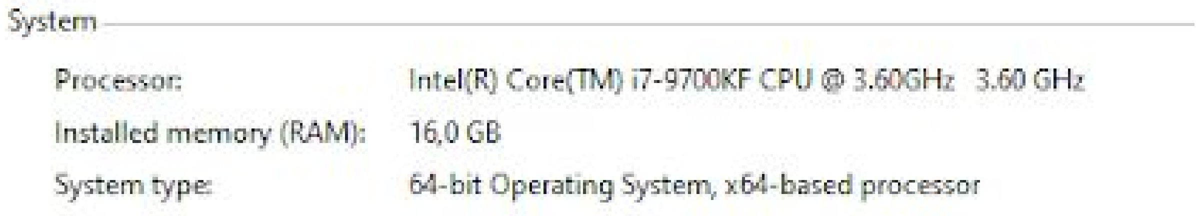
14 Edge Cases

The only edge cases that appeared were specific regexs and input of lengths greater than 250. This sometimes caused a StackOverflow error. The error was easily fixed by allocating more memory for the stack as discussed above in [2].

15 Bonus Additions

Backreferences have been implemented. The implementation is limited as it is capped to a maximum of 9 referenceable groups. With a little work on the parser this number could be extended to 99, but this was beyonds the scope of the implementation. Validation for backreferences has also been implemented and a group cannot be referenced if it does not exist.

16 Specifications for Testing



System

Processor:	Intel(R) Core(TM) i7-9700KF CPU @ 3.60GHz 3.60 GHz
Installed memory (RAM):	16,0 GB
System type:	64-bit Operating System, x64-based processor

17 References

1. Geoff Langdale. (2019). *Question: Is matching fixed regexes with Back-references in P?.* Available: https://branchfree.org/2019/04/04/question-is-matching-fixed-regexes-with-back-references-in-p/. Last accessed October 2020.
2. Jan van Leeuwen (Ed.). 1991. Handbook of theoretical computer science (vol. A): algorithms and complexity. MIT Press, Cambridge, MA, USA.
3. Mike Rosulek. (). *Reduction of VERTEX COVER to Perl Regular Expression Matching.* Available: http://www.mikero.com/misc/code/vertex-cover2.html. Last accessed October 2020.