修士 (工学) 学位論文

Master's Thesis of Engineering

Cerium による文字列の並列処理

Parallel processing of strings using Cerium

平成 27 年度 3 月

琉球大学大学院　理工学研究科
情報工学専攻

琉球大学

大学院理工学研究科

情報工学専攻

Universty of the Ryukyus

Graduate School of Engineering and Science

Infomation Engineering Course

要 旨

目 次

図 目 次

表 目 次

第1章　introduction

正規表現はオートマトンに変換することができ、そしてオートマトンの受理の問題は Class NC と呼ばれる問題でもある。この問題は計算機の台数が多ければ多いほど高速化できるという特徴を持ち、並列化に向いている問題といえる。コンピュータの動作やゲームの動作などの多くの問題はオートマトンの受理問題に落としこむことができるので、この問題を解決すれば様々な問題に対応できるようになる。本研究では Cerium 上に正規表現を実装することにより。

word count などを早く処理するため I/O の並列化膨大なファイル

第2章　Cerium

Cerium は、Cell 向けに開発された並列プログラミングフレームワークである。Cell は Sony Computer Entertainment 社が販売した PlayStation3 に搭載されているヘテロジニアスマルチコア・プロセッサである。本章では Cerium の実装について説明する。

2.1　Cerium の概要

Cerium は当初 Cell 向けに開発され、C/C++ で実装されている。現在では Linux、MacOS X 上で動作する並列プログラミングフレームワークである。

Cerium は TaskManager、SceneGraph、Rendering Engine の 3 要素から構成されている。本研究では汎用計算フレームワークである TaskManager を利用して文字列の並列計算を行なった。

図 2.1 は Cerium が Task の生成/実行する場合のクラス構成図である。TaskManager で依存関係が解消され、実行可能になった Task は ActiveTaskList に格納される。ActiveTaskList に格納された Task は、依存関係が解消されているのでどのような順番で実行されても問題はない。Task は転送を行いやすい TaskList に変換され、CpuType に対応した Scheduler に転送される。なお、転送は Synchronozed Queue である mail を通して行われる。

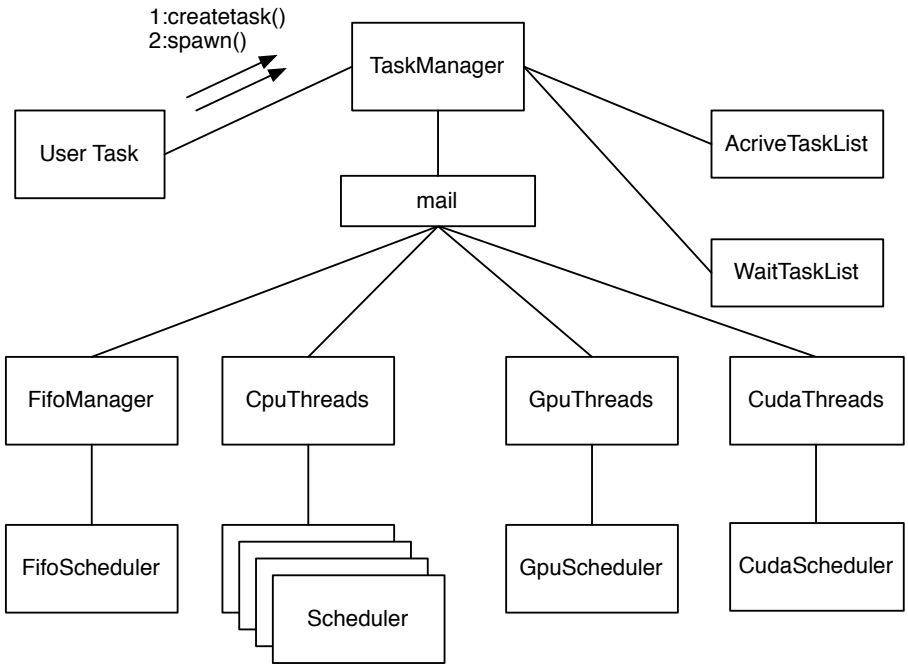

図 2.1: Task Manager

2.2 Cerium TaskManager

Cerium TaskManager では、処理の単位を Task として記述していく。関数やサブルーチンを Task として取り扱い、その Task にて Input Data/Output Data 及び Task の依存関係を設定する。そして Task は設定された依存関係を考慮しながら実行される。

Input Data で格納した 2 つの数を乗算し、Output Data に演算結果を格納する multiply という例題のソースコード 2.1 を以下に示す。

また、Task の生成時に用いる API 一覧を表 2.2 に示す。

ソースコード 2.1: Task の生成

```
1  multi_init(TaskManager *manager)
2  {
3      float *A, *B, *C;
4
5      // create Task
6      HTaskPtr multiply = manager->create_task(MULTIPLY_TASK);
7
8      // set device
9      multiply->set_cpu(SPE_ANY);
10
11     // set inData
12     multiply->set_inData(0, (memaddr)A, sizeof(float)*length);
13     multiply->set_inData(1, (memaddr)B, sizeof(float)*length);
14
15     // set outData
16     multiply->set_outData(0, (memaddr)C, sizeof(float)*length);
```

```
17
18      // set parameter
19      multiply->set_param(0,(long)length);
20
21      // spawn task
22      multiply->spawn();
23  }
```

create_task	Task を生成する
set_inData	Task への入力データのアドレスを追加
set_outData	Task への出力データのアドレスを追加
set_param	Task へ値を一つ渡す。ここでは length
set_cpu	Task を実行するデバイスの設定
spawn	生成した Task を TaskList に set

表 2.1: Task 生成における API

次に、デバイス側で実行される Task のソースコードを 2.2 に示す。

ソースコード 2.2: Task

```
1  static int
2  run(SchedTask *s) {
3       // get input
4       float *i_data1 = (float*)s->get_input(0);
5       float *i_data2 = (float*)s->get_input(1);
6
7       // get output
8       float *o_data = (float*)s->get_output(0);
9
10      // get parameter
11      long length = (long)s->get_param(0);
12
13      // calculate
14      for (int i=0; i<length; i++) {
15           o_data[i] = i_data1[i] * i_data2[i];
16      }
17      return 0;
18  }
```

また表 2.2 は Task 側で利用する API である。Task 生成時に設定した Input Data や parameter を取得することができる。

表 2.2: Task 側で使用する API

get_input	Scheduler から input data を取得
get_output	Scheduler から output data を取得
get_param	set_param した値を取得

Task 生成時に設定できる要素を以下に列挙する。

- Input Data
- Output Data
- Parameter
- CpuType
- Dependency

Input/Output Data、Parameter は関数の引数に相当する。Cpu Type は Task を動作させるデバイスを設定することができ、Dependency は他の Task との依存関係を設定することができる。

第3章　並列処理向け I/O

ファイル読み込みなどの I/O を含むプログラムは、読み込み時間が Task の処理時間と比較して大きくなることが多い。計算処理の並列化を図ったとしても I/O がボトルネックになってしまい処理全体が高速にならない。従来の例題のファイル読み込み部分では mmap を利用していたが、読み込みと Task が並列に動くような実装を行ない、プログラム全体の高速化を図った。

本項では mmap による読み込みと、今回実装した並列処理向け I/O について述べる。

3.1　mmap

Cerium の例題ではファイル読み込みを mmap にて実装していた。

mmap は function call 後にすぐにファイルを読みに行くのではなく、仮想メモリ領域にファイルの中身を対応させ、 その後メモリ空間にアクセスされたときに、OS が対応したファイルを読み込む。

そのため、mmap によるファイルを読み込みは読み込み後に Task を実行するので、その間は他の CPU が動作せず並列度が落ちる。

また、読み込む方法が OS 依存となってしまうため環境に左右されやすく、プログラムの書き手が読み込みの制御をすることが難しい。

図 3.1 は mmap で読み込んだファイルに対して Task1 、 Task2 がアクセスしてそれぞれの処理を行うときのモデルである。

Task1 が実行されると仮想メモリ上に対応したファイルが読み込まれ、読み込み後 Task1 の処理が行われる。その後 Task2 も Task1 と同様の処理が行われるが、これら 2 つの Task の間に待ちが入る。

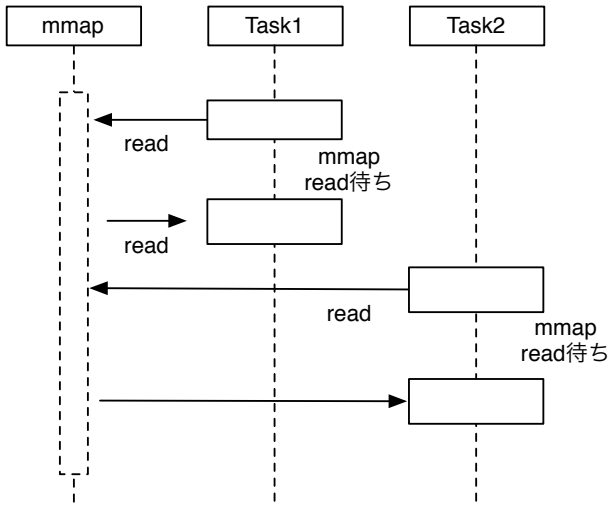

図 3.1: mmap Model

3.2 Blocked Read

読み込みを独立した Thread で行ない、ファイルを一度に全て読み込むのではなくある程度の大きさ (Block) 分読み込み、読み込まれた部分に対して並列に Task を起動する。これを Blocked Read と呼び、I/O の読み込みと Task の並列化を図った。

ファイルを読み込む Task (以下、Blocked Read) と、読み込んだファイルに対して計算を行う Task を別々に生成する。Blocked Read は一度にファイル全体を読み込むのではなく、ある程度の大きさで分割してから読み込みを行う。分割して読み込んだ範囲に対して Task を実行する。

ファイル読み込みを含むプログラムを Blocked Read で読み込み処理をしたとき以下の図 3.2 の様になる。

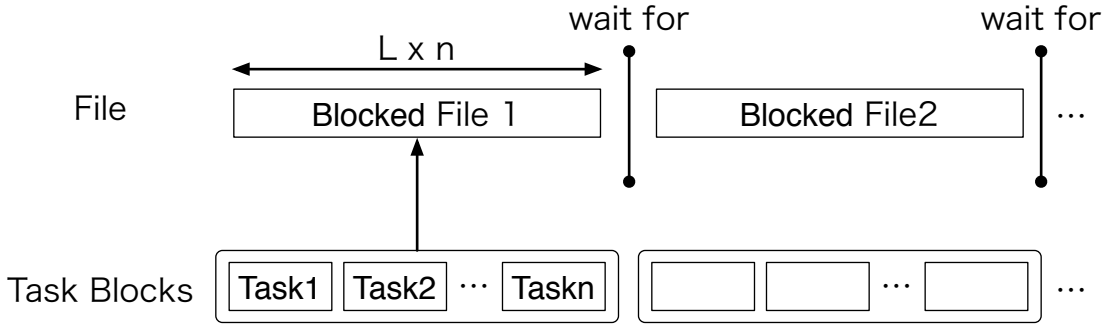

図 3.2: BlockedRead による WordCount

Task を一定の単位でまとめた Task Block ごとに生成して Task を行なっている。Task Block で計算される領域が Blocked Read で読み込む領域を追い越して実行してしまう

と、まだ読み込まれていない領域に対して計算されてしまう。その問題を解決するために依存関係を適切に設定する必要がある。Blocked Read による読み込みが終わってから TaskBlock が起動されるようにするため、Cerium の API である wait_for にて依存関係を設定する。

(図 3.3)

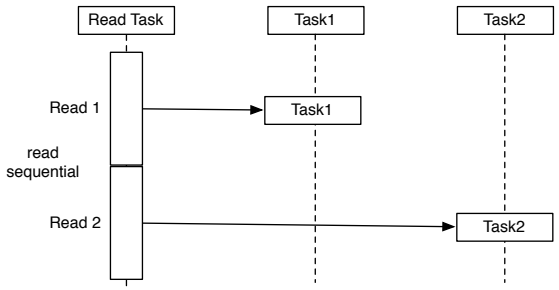

図 3.3: BlockedRead Model

3.3 I/O 専用 thread の追加

Blocked Read は読み込みを含む処理なので、Task 1 つあたりの処理時間が大きくなる。Blocked Read がファイルを読み込む前提で Task がその領域に対して計算を行うので、ReadTask の処理によってプログラム全体の処理速度が左右されてしまう。

Cerium Task Manager では、それぞれの Task に対してデバイスを設定することができる。SPE_ANY 設定をすると、Task Manager が CPU の割り振りを自動的に行う。しかし、自動的に割り振りを行なってしまうと、Blocked Read Task 間に Task が割り込まれてしまい、読み込みが遅延してしまう可能性がある。(図 3.4)

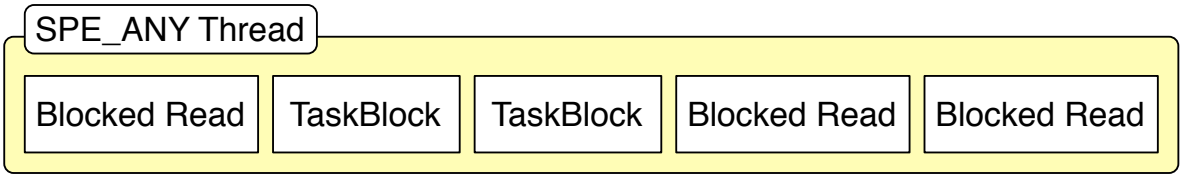

図 3.4: BlockedRead と Task を同じ thread で動かした場合

そこで、Task が Blocked Read Task 間に割り込まれないようにするため、I/O 専用 thread である iO_0 の設定を追加した。

IO_0 は SPE_ANY とは別 thread の scheduler で動作するので、SPE_ANY で動作している Task に割り込むことはない。しかし、読み込みの終了を通知し、次の read を行う時に

他の Task がスレッドレベルで割り込んでしまう事があるため、pthread_getschedparam() で IO_0 の priority の設定を行う必要がある (図:3.5)。

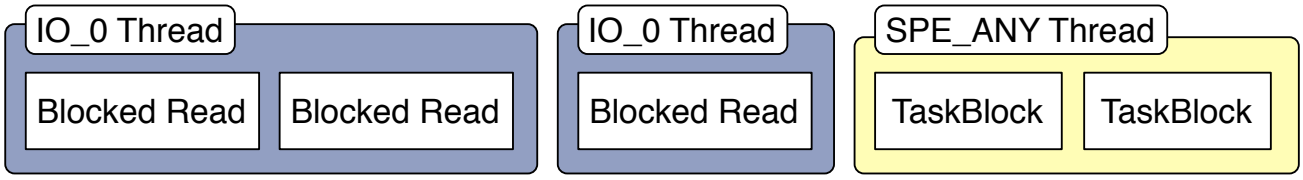

図 3.5: IO Thread による BlockedRead

第4章　Cerium による文字列処理の例題

本項ではファイルを読み込んで処理する流れとそれの例題を記述する。例題として、単語数を数える Word Count、文字列探索を行う Boyer Moore Search、正規表現を挙げる。

4.1　文字列処理の並列処理

文字列処理を並列で処理する場合を考える。まずファイルを読み込み、ファイルをある一定の大きさで分割する (divide a file)。そして、分割されたファイル (Input Data) に対して文字列処理 (Task) をおこない、それぞれの分割単位で結果を出力する (Output Data)。それらの Output Data の結果が出力されたあとに、結果をまとめる処理を行う (Print Task)。(図 4.1)

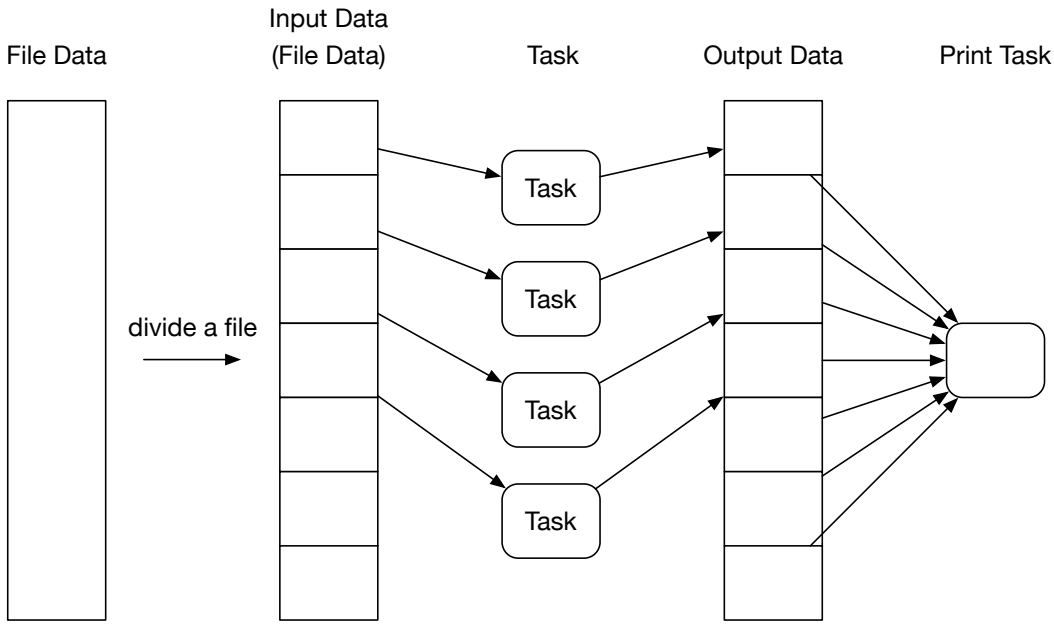

図 4.1: File 読み込みから処理までの流れ

File 分割時に分割された部分の整合性についてはそれぞれの例題にて述べる。

4.2 Word Count

Word Count は読み込んだテキストに対して単語数を数える処理である。Input Data には分割されたテキストが対応しており、Output Data には単語数と行数を出力する。

読み込んだテキストを先頭から見ていき、単語の末端に空白文字か改行文字があれば単語数、改行文字があれば行数を数えることができる。

分割された部分に単語が含まれた場合、単語数や行数について整合性を取る必要がある。図 4.2 ではファイル分割無しの Word Count である。

分割しない状態では単語数 (Word Num) 3、行数 (Line Num) 2 となる。

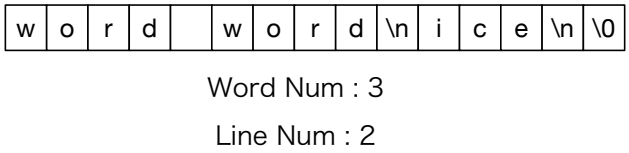

図 4.2: ファイル分割無しの Word Count

図 4.3 では単語で分割された場合である。分割されたファイルそれぞれの結果を合計すると単語数 4、行数 2 となり、分割されていない時と結果が変わってしまう。

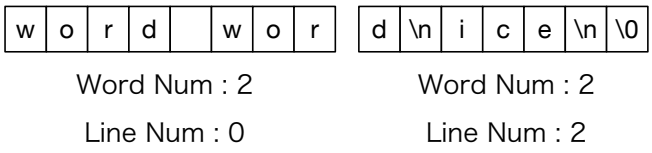

図 4.3: ファイル分割有りの Word Count

この問題の解決方法として、分割されたファイルの一つ目が文字列で終わり、二つ目のファイルの先頭が文字列で始まった場合はそれぞれの単語数の合計数から 1 引くことにより整合性を取ることができる。

4.3 Boyer-Moore String Search

読み込んだテキストファイルに対してある特定の文字列検索を行う例題として、Boyer-Moore String Search が挙げられる。Boyer-Moore String Search は 1977 年に Robert S. Boyer と J Strother Moore が開発した効率的なアルゴリズムである。[1]

以下、テキストファイルに含まれている文字列を text、検索する文字列を pattern と定義する。

原始的な検索アルゴリズムとして力任せ法が挙げられる。力任せ法は text と pattern を先頭から比較していき、pattern と一致しなければ pattern を 1 文字分だけ後ろにずらして再度比較をしていくアルゴリズムである。text の先頭から pattern の先頭を比較していき、文字の不一致が起きた場合は pattern を後ろに 1 つだけずらして再比較を行う。(図 4.4)

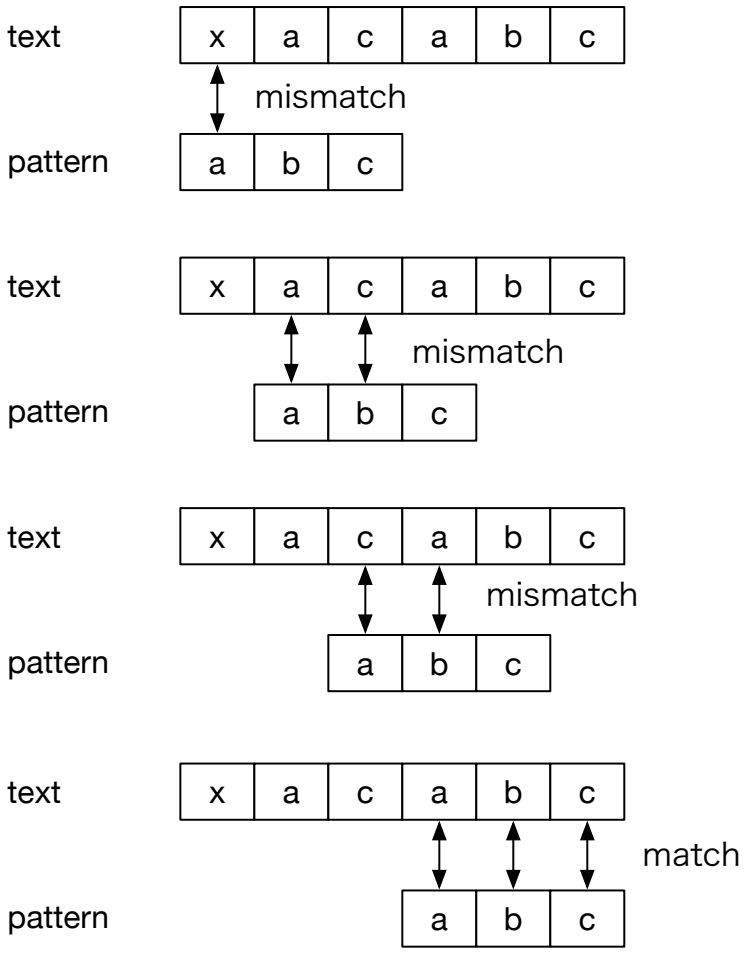

図 4.4: 力まかせ法

このアルゴリズムは実装が容易であるが、 text と pattern の文字数が大きくなるにつれて、比較回数も膨大になる恐れがある。text の長さを n、pattern の長さを m とすると、力任せ法の最悪計算時間は $O(nm)$ となる。

力任せ法の比較回数を改善したアルゴリズムが Boyer-Moore String Search である。力任せ法との大きな違いとして、text と pattern を先頭から比較するのではなく、 pattern の末尾から比較していくことである。さらに不一致が起こった場合は、その不一致が起こった text の文字で再度比較する場所が決まる。

図 4.5 は、text と pattern の末尾が不一致を起こして、そのときの text が pattern に含まれていない場合である。不一致した text の文字が pattern に含まれていない場合は、pattern を比較する場所に match することはないので、pattern の長さ分だけ後ろにずらすことができる。

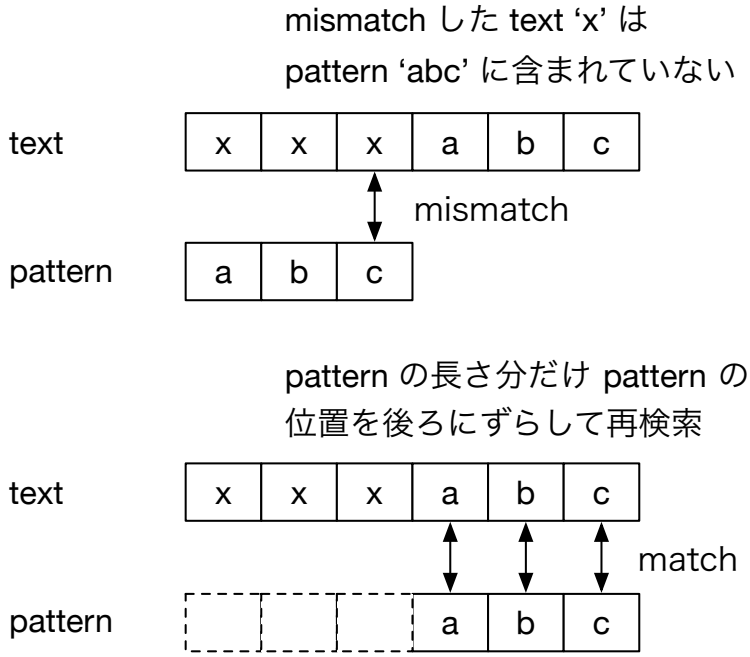

図 4.5: pattern に含まれていない文字で不一致になった場合

図 4.6 は不一致が起こったときの text の文字が pattern に含まれている場合である。この場合は pattern を後ろに 2 つずらすと text と pattern が一致する。

不一致したときの text の文字が pattern に含まれていた場合の後ろにずらす量は、pattern の長さから含まれていた文字が pattern の何文字目に含まれているかを引いた値となる。この場合、pattern の文字列の長さは 3 で text で不一致を起こした文字 'a' が pattern の 1 文字目に含まれているので、2 文字分だけ後ろにずらすことができる。

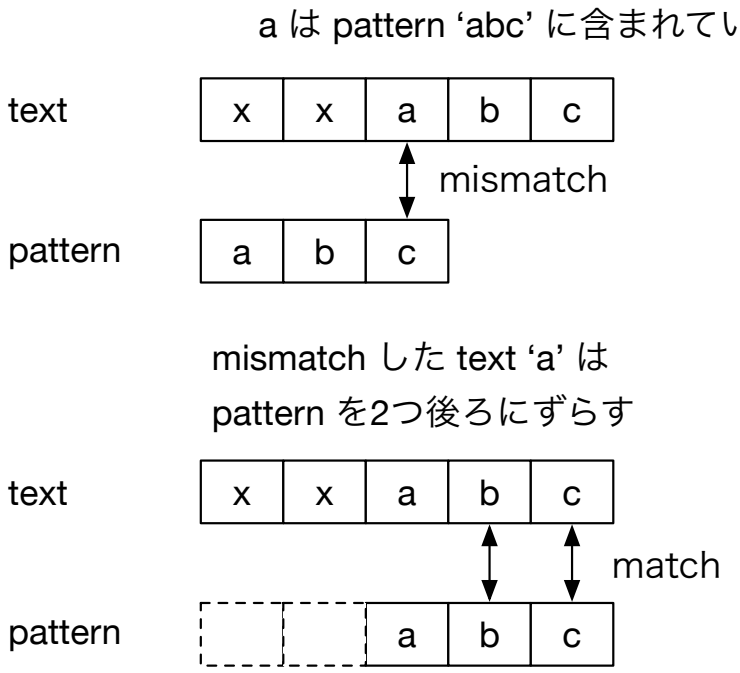

図 4.6: pattern に含まれている文字で不一致になった場合

図 4.7 は不一致が起こったときの text の文字が pattern に含まれ、その不一致文字が pattern に複数含まれている場合である。

pattern の長さは 4 で、不一致を起こした時の text の文字 'a' は pattern の 1 番目と 3 番目に含まれている。pattern を後ろにずらす量は 1 か 3 となる。ずらす量を 3 にすると、pattern が含まれている text を見逃す可能性があるので、この場合 'a' で不一致したときは最小の値 1 をとる。

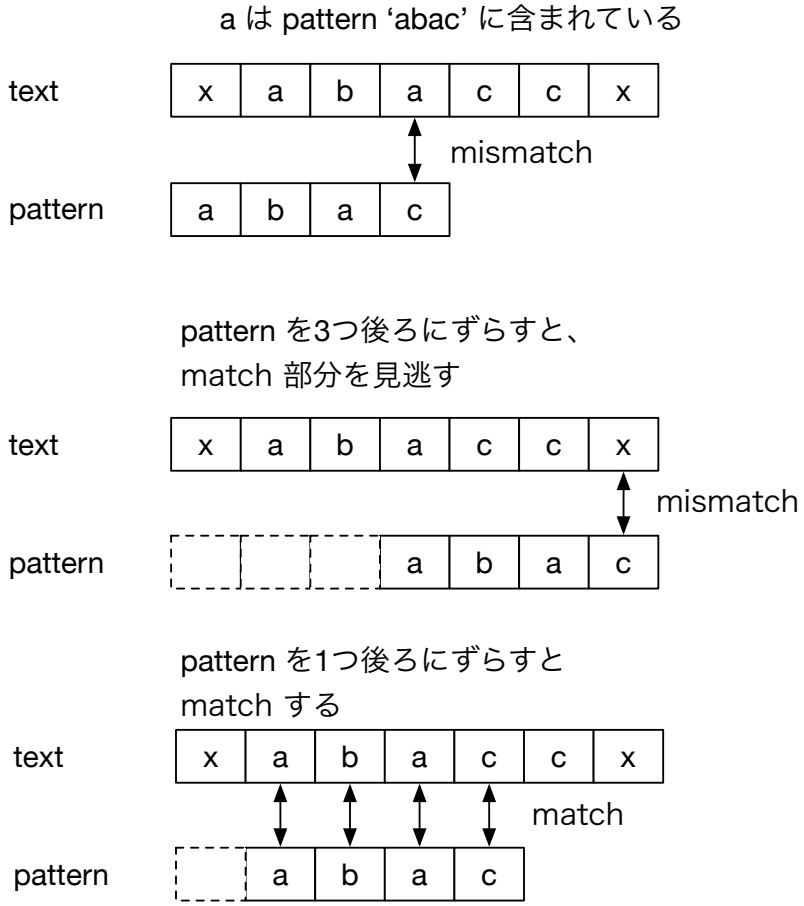

図 4.7: pattern に同じ文字が複数入り、その文字で不一致になった場合

pattern と text と不一致時の処理をまとめると、

- pattern に含まれていない文字で不一致した場合は、 pattern の長さだけ後ろにずらす。
- pattern に含まれている文字の場合は、pattern の長さから pattern に含まれている文字の位置を引いた数だけ後ろにずらす。
- pattern に含まれている文字でその文字が pattern に複数含まれている場合は後ろにずらす量も複数現れる。その中の最小の値だけ後ろにずらす。

text 分割時に、分割部分で pattern が含まれる場合が存在する。その場合は、本来の読み込み部分の text の長さ L に加えて、pattern の長さ s から 1 引いた数だけ多く読みこむように設計することで、正しく結果を算出することができる。(図 4.8)

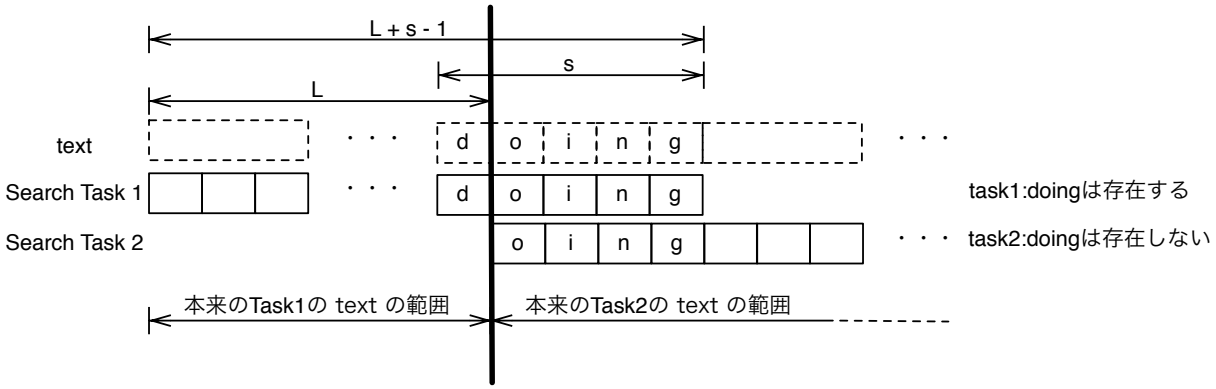

図 4.8: 分割周りの処理

4.4　正規表現

(正規表現の簡単な概要をここに)
実装した正規表現マッチャのアルゴリズムは、

1. 与えられた正規表現を構文解析し、正規表現木に変換する。
2. 正規表現木から非決定性オートマトン (以下、NFA) か決定性オートマトン (以下、DFA) に変換する。
3. NFA に変換された場合、Subset Construction による NFA から DFA への変換をおこなう。
4. DFA を元に文字列検索を行ない結果を返す。

となる。本項はそれぞれのアルゴリズムについて述べていく。

4.4.1　正規表現木の生成

まずはじめに、図 4.9 のように与えられた正規表現から正規表現木に変換する。与えられた正規表現を頭から一文字ずつ読み込み、読み込んだ文字やメタ文字と呼ばれる正規表現での特殊記号を元に木を構成していく。

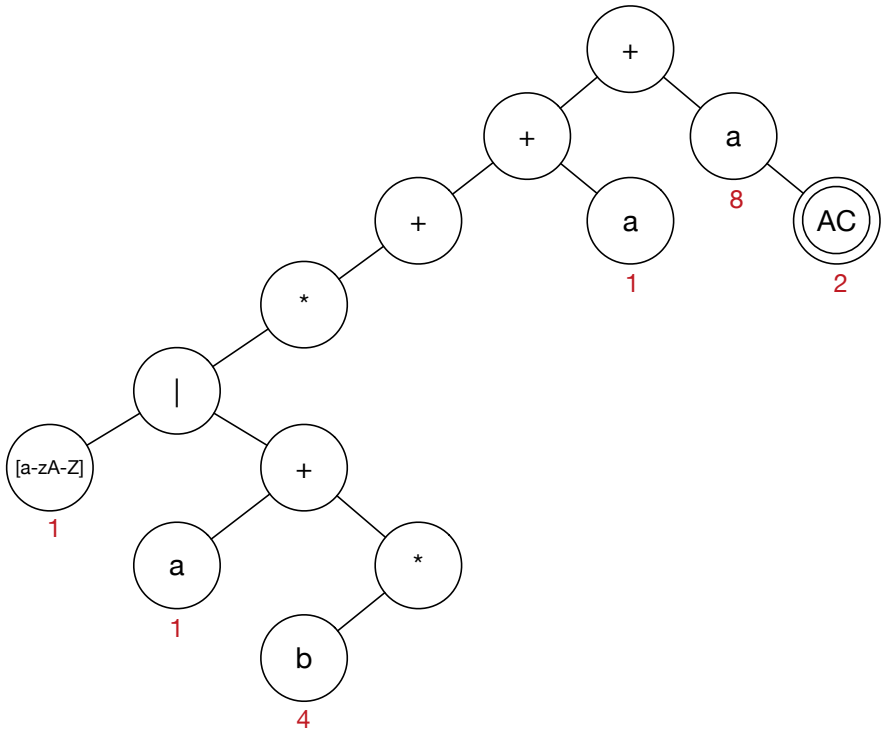

図 4.9: 正規表現から正規表現木への変換の例

本実装でサポートするメタ文字は、正規表現の基本三演算子 [2] に文字クラスとグループを加えている。(表 4.1)

AB	連続した文字 (連接)
A*	直前の文字の 0 回以上の繰返し
A|B	A または B(選択)
[A-Z]	A-Z の範囲の任意の一文字 (文字クラス)
( )	演算の優先度の明示 (グループ)

表 4.1: サポートしているメタ文字一覧

また、これらのメタ文字は数式の四則演算のように結合順位を持っている。それぞれのメタ文字の結合順位は表 4.2 のようになる。

これらの条件踏まえた上で正規表現木を生成していく。

結合順位	メタ文字
高	() (グループ化)
	[ ] (文字クラス)
	* 繰返し
	連接
低	| 選択

表 4.2: メタ文字の結合順位

また、以下よりメタ文字を含まない文字や文字クラスのことを文字、文字が連接されている場合を文字列、全ての文字が含まれている場合は正規表現と表現する。

正規表現木は与えられた正規表現を先頭から一文字ずつ読み込み、読み込んだ文字やメタ文字を一定のルールに従って生成していく。文字やメタ文字、文字クラスは正規表現木のノードとして表現され、メタ文字が現れた時に親子関係が決定される。

文字が読み込まれた場合はノードを生成し、それらが連接された文字は '+' ノードを親ノードとして、左に前の文字、右に後ろの文字が接続される。(図 4.10)

また、文字列のように連接が連続した場合、連接済みの '+' ノードを左の子ノードとしてさらに '+' ノードで結合していく。(図 4.11)

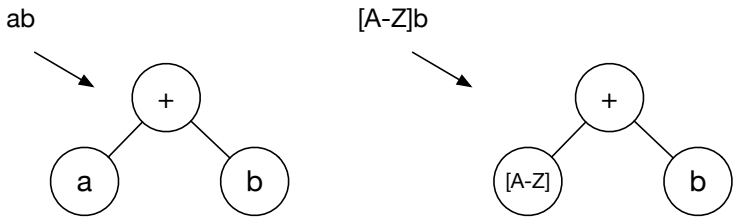

図 4.10: 文字の連接

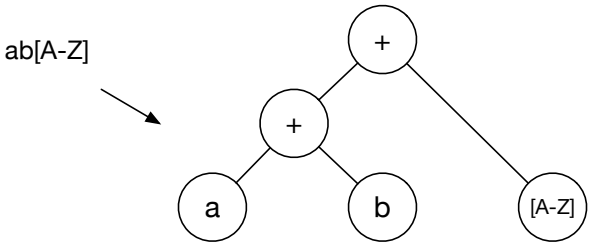

図 4.11: 文字列の連接

選択 '|' が読み込まれた場合、親ノードを '|' として、 '|' の直前の正規表現は左ノード、

直後の正規表現は右ノードとした木が構成される。‘|’ は直前と直後の正規表現の関係を表しているので、左右のノードに正規表現の要素を持ったノードとなる。

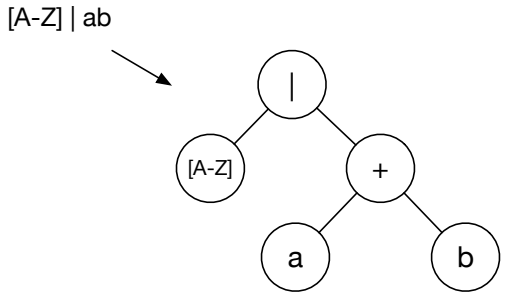

図 4.12: 選択

繰返し ‘*’ が読み込まれた場合、‘*’ の直前の正規表現を左の子ノードとした木が生成される。また ‘*’ は、‘*’ の直前の正規表現だけに結合するので、右の子ノードに何かしらのノードが生成されることはない。

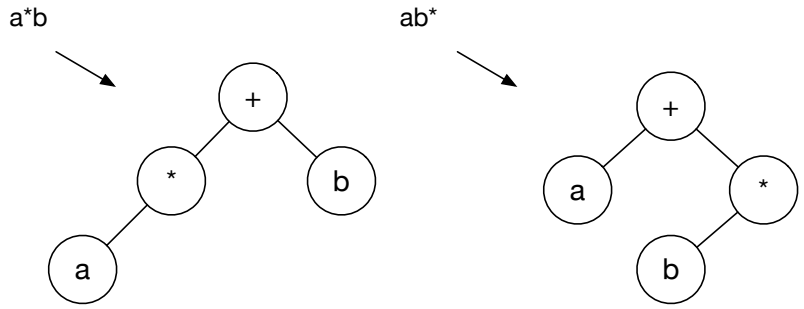

図 4.13: 繰返し

グループ化 ‘(’ ‘)’ が読み込まれた場合、‘(’ ‘)’ 内をひとかたまりの正規表現として木を構成する。構成後さらに文字列が読み込まれれば、上記のルールにしたがって木が構成される。

正規表現が連接した場合も文字の連接と同様に ‘+’ を親ノードとして接続していく。

これらのルールに則って正規表現木を構成し、それを元に DFA・NFA を生成していく。

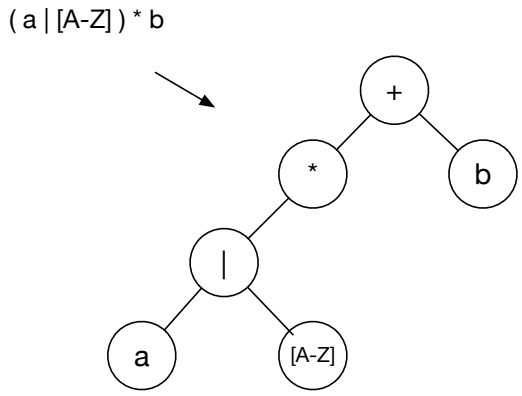

図 4.14: グループ

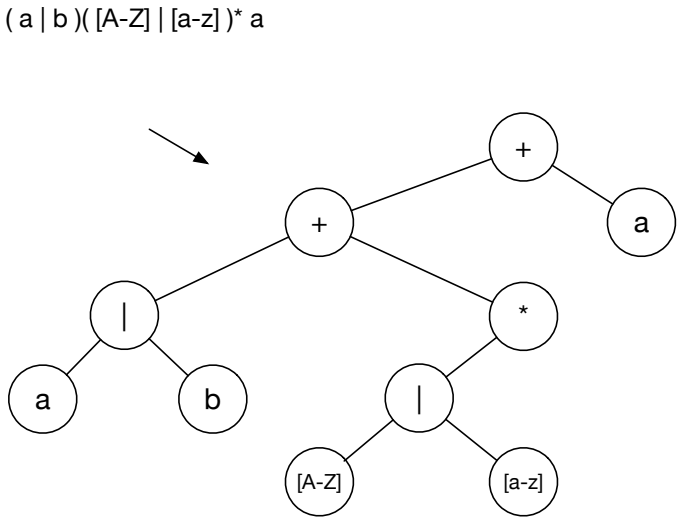

図 4.15: 正規表現の連接

4.4.2　正規表現木から DFA・NFA の生成

次に正規表現木から非決定性有限オートマトン (NFA)、決定性有限オートマトン (DFA) を生成する。

オートマトンは、入力に対して状態に対応した処理を行ない結果を出力する仮想的な自動機械である。正規表現はオートマトンで表現することができるので、状態と入力 (ここでは正規表現) が判れば次はどのような状態になるのか示すことができる。

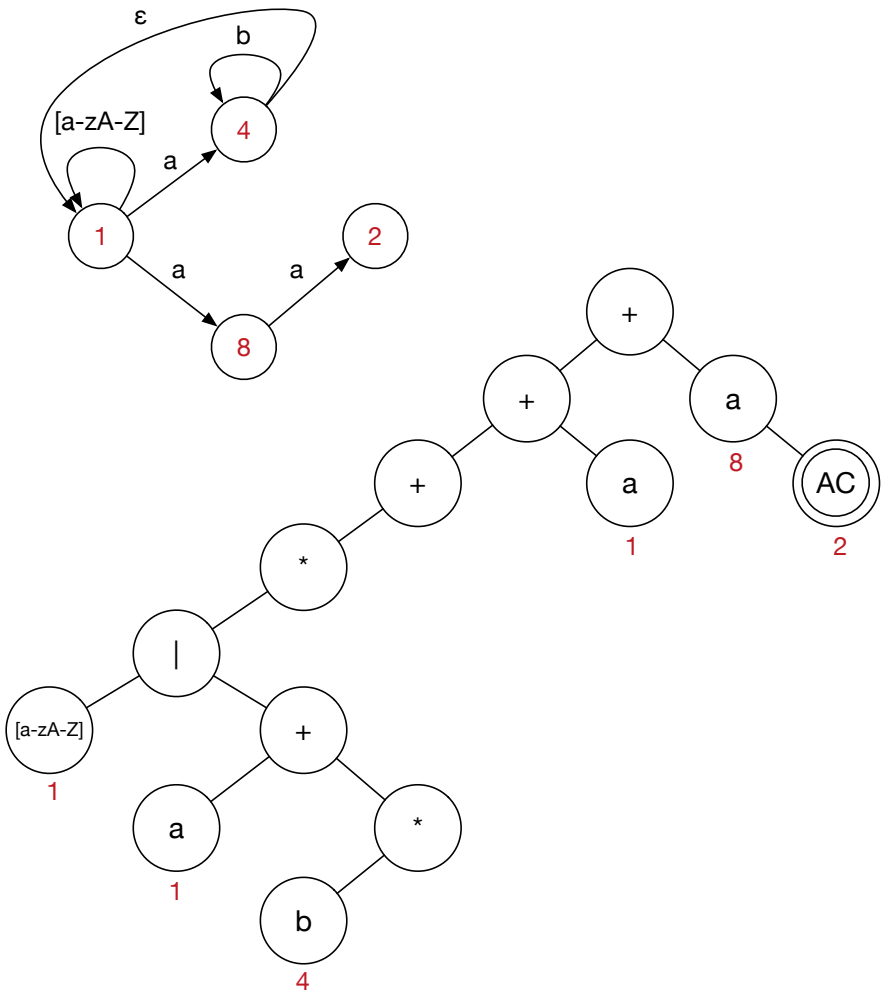

図 4.16: 与えられた正規表現のオートマトンと正規表現木の例

正規表現木を深さ優先探索にて左から辿っていき、文字のノードにそれぞれ状態を番号で割り振りを行う。また、状態の振り方は探索した際のメタ文字のノードに沿って割り振りを行う。

それぞれのメタ文字がどのような状態を割り振るか紹介する。また、番号 1 は初期状態、番号 2 は受理状態を表している。

図 4.17 は連接 ‘+’ で接続されている場合の正規表現である。受理される文字列の集合は { ab } である。a が入力されれば別の状態になり、その状態で b が入力されれば受理状態に遷移する。これより ‘+’ で接続された木の状態割当は、‘+’ の左ノードの状態とは別の新しい状態を生成して割り当てる。

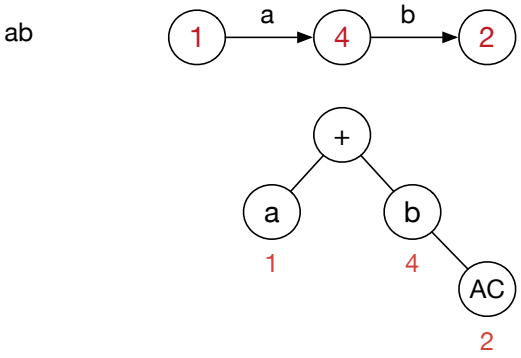

図 4.17: 連接の状態割当

図 4.18 は選択 ‘|’ で接続されている場合の正規表現である。受理される文字列の集合は { a, b } である。この場合は a か b が入力されれば受理状態に遷移する。これより ‘|’ で接続された木の状態割当は、‘|’ の左ノードと右ノードが同じ状態となり、新しい状態は生成されない。

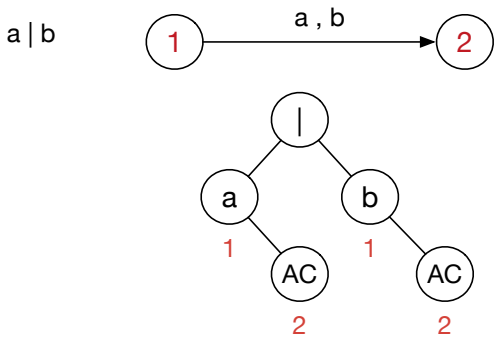

図 4.18: 選択 ‘|’ で接続されているときの状態割当

図 4.19 は連接 ‘+’ と選択 ‘|’ の組み合わせで接続されている場合の正規表現である。受理される文字列の集合は {ac,bc} である。この場合、初期状態に a か b が入力されると次の状態に遷移し、遷移した状態に c が入力されると受理状態に遷移する。連接 ‘+’ と選択 ‘|’ の状態割当方法の組み合わせにて状態を決定することができる。

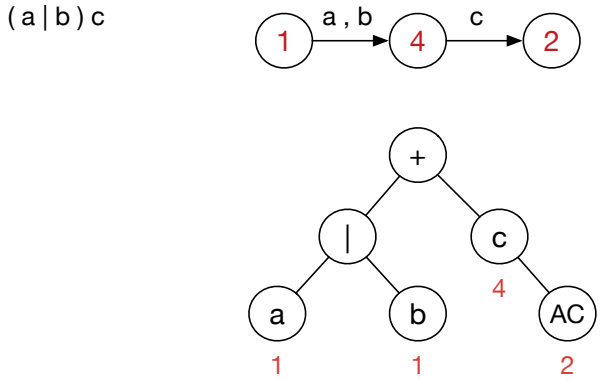

図 4.19: 選択 ‘|’ と連接の組み合わせの状態割当

図 4.20 は連接 ‘+’ の前の文字に繰返し ‘*’ が接続されている場合の正規表現である。受理される文字列の集合は {b,ab,aab,aaab,aa...ab} である。この場合、初期状態に a が入力されると自分自身の状態に遷移する。遷移先を自分自身にすることによって、繰返しを表現することができる。その次に b が入力されると受理状態に遷移する。これより、‘+’ の左ノードに ‘*’ が接続されていたら、‘*’ に接続されている木の一番左と ‘+’ の右ノードに同じ状態が割り当てられる。

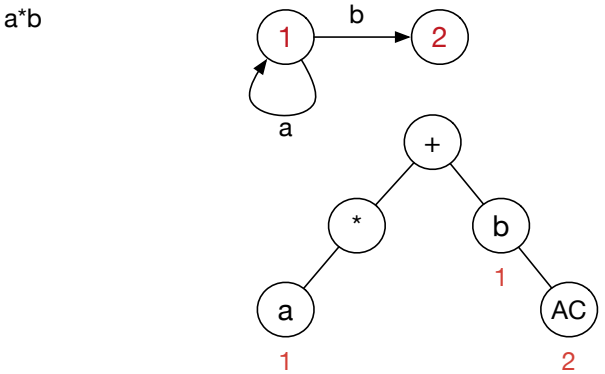

図 4.20: 連接の前の文字に ‘*’ が接続されているときの状態割当

図 4.21 は連接 ‘+’ の後の文字に繰返し ‘*’ が接続されている場合の正規表現である。受理される文字列の集合は {a,ab,abb,abb,abb...bb} である。この場合、初期状態に a が入力されると受理状態に遷移する。しかし、受理状態でも b がそれ以降に入力されれば、自分自身に状態遷移する。これより、‘+’ の右ノードに ‘*’ が接続されていたら、‘+’ の左ノードに接続されている木の最後の状態に受理状態を付け加える。また、‘*’ に接続されている木の最後の状態にも受理状態を付け加える。

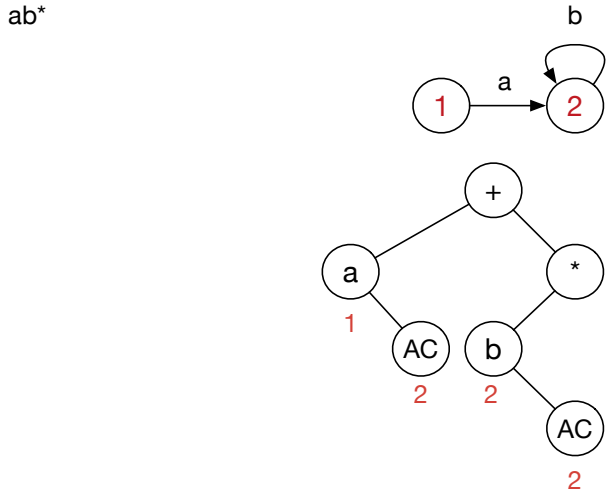

図 4.21: 連接の後ろの文字に ‘*’ が接続されているときの状態割当

図 4.22 は連接 ‘+’ が連続しており、連接の途中で繰返し ‘*’ が接続されている場合の正規表現である。受理される文字列の集合は {ac,abc,abbc,abbbc,abb...bbc} である。この場合、初期状態に a が入力されると次の状態に遷移する。その状態で b が入力されると自分自身に遷移し、c が入力されると受理状態に遷移する。

これより、連接中に ‘*’ があれば新しい状態を生成し、その状態を ‘*’ の親ノードのさらに親ノードの右ノードに同じ状態にする。

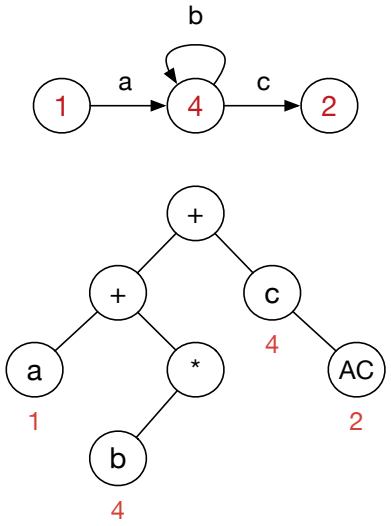

図 4.22: 連接中に ‘*’ が接続されているときの状態割当

図 4.23 は選択 ‘|’ がグループ化によって一つの正規表現となり、それ自身が繰り返されている場合の正規表現である。受理される文字列の集合は {c,ac,bc,aabc,abbc,a..ab..bc} である。この場合、初期状態に a か b が入力されると自分自身の状態に遷移する。その状態で c が入力されると受理状態に遷移する。これは、選択 ‘|’ と繰返し ‘*’ の状態割当方法の組み合わせにて状態を決定することができる。まず ‘a |b’ は同じ状態を割り当て、その親ノードが ‘*’ なので ‘*’ の親の右ノードに同じ状態を割り当てる。

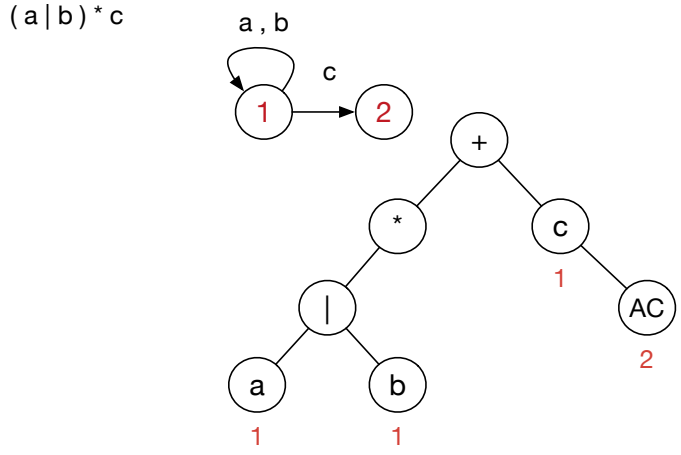

図 4.23: 選択 ‘|’ と繰返し ‘*’ の組み合わせの状態割当

以上の規則で正規表現木を辿った時にノードに対して状態を割り振る。まとめると、

- 左子ノードが '*' でない '+' は新しい状態を作る
- '|' が親ノードの場合、子ノードの最初の状態は同じ状態。
- '*' があれば、次の状態は '*' に接続されている木の先頭の状態と同じ。次の状態が受理状態なら先頭の状態と受理状態の組み合わせになる。

これにより、正規表現木に状態の割り振りを行ない、入力を行なったら状態が遷移するようにできた。現在の状態 (current state) と入力 (input) によって次の状態 (next state) が一意に決まっており、それをテーブル化して正規表現をファイルにかける。(図 4.24)

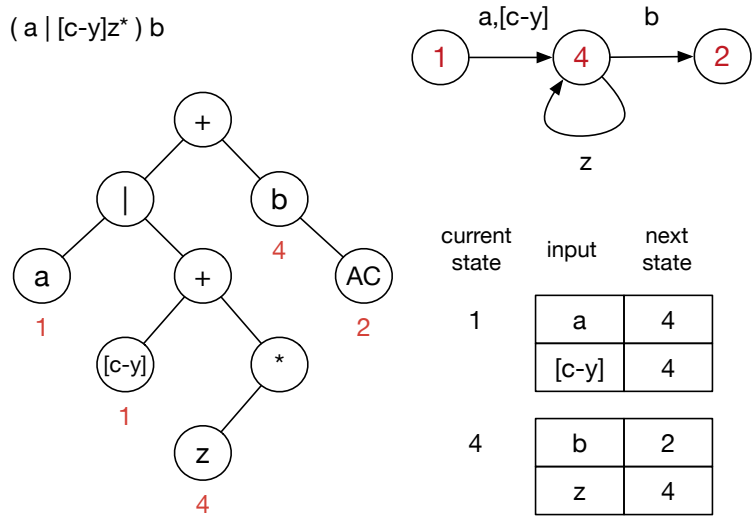

図 4.24: dfa

しかし、生成された正規表現木によっては、現在の状態と入力による次の状態が一意に決まらない場合もある。図??

4.4.3　Subset Construction による NFA から DFA の変換

4.4.4　DFA を元にパターンマッチを行う

現在のステートに含まれる組み合わせ状態をとってくる
組み合わされた個々の charclass を merge して新しい charclass をつくり、組み合わせ状態に登録する
生成した状態は stateArray に格納する
新しい状態ができなくなったら終了

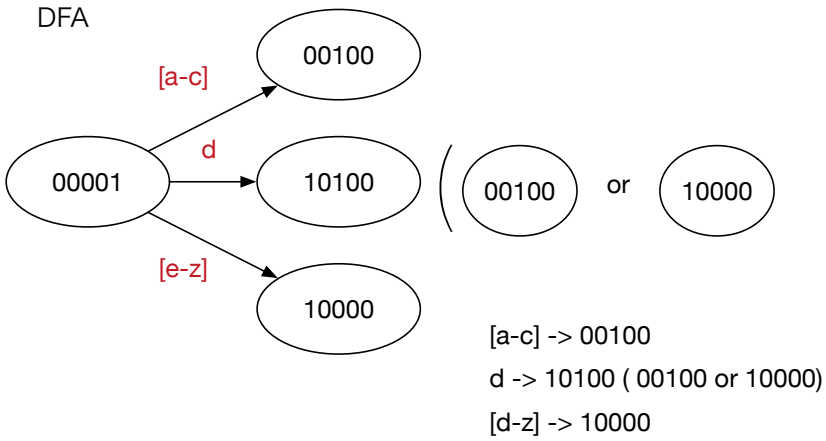

図 4.25: dfa

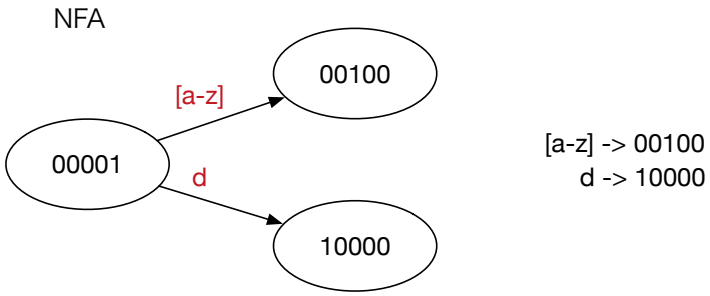

図 4.26: nfa

current state	condition	next state
1	[A-Z]	1
	[b-z]	1
	a	13
4	b	4
8	a	2
13	[A-Z]	1
	[c-z]	1
	a	15
	b	5

current state	condition	next state
15	[A-Z]	1
	[c-z]	1
	a	15
	b	5
5	[A-Z]	1
	[c-z]	1
	a	13
	b	5

図 4.27: Transition Table

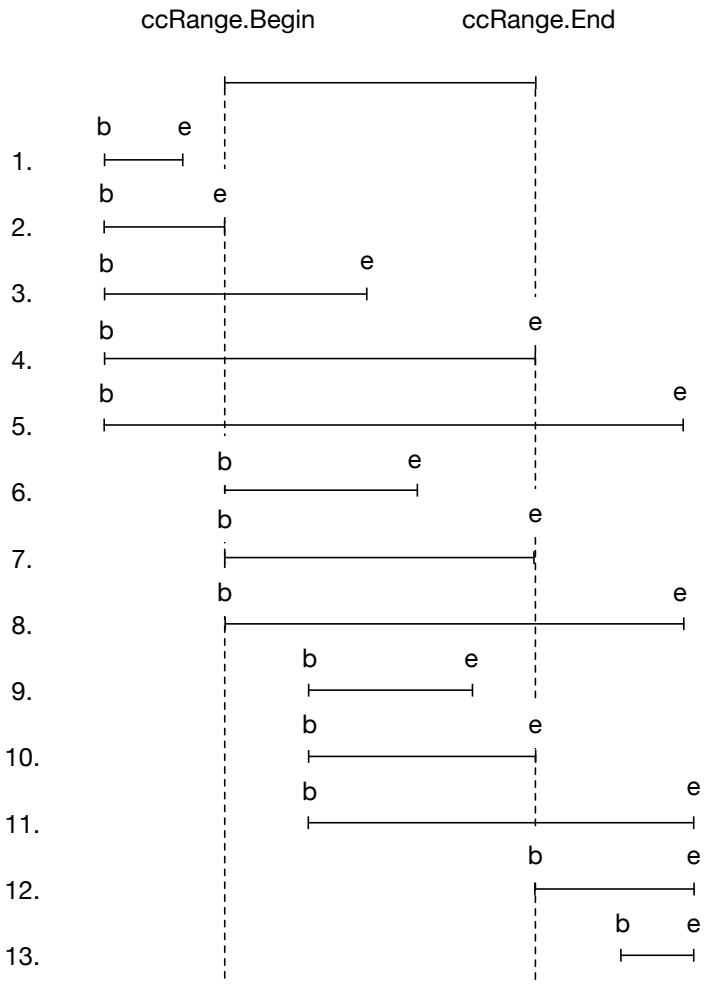

図 4.28: 2 つの Character Class を merge するときの全パターン

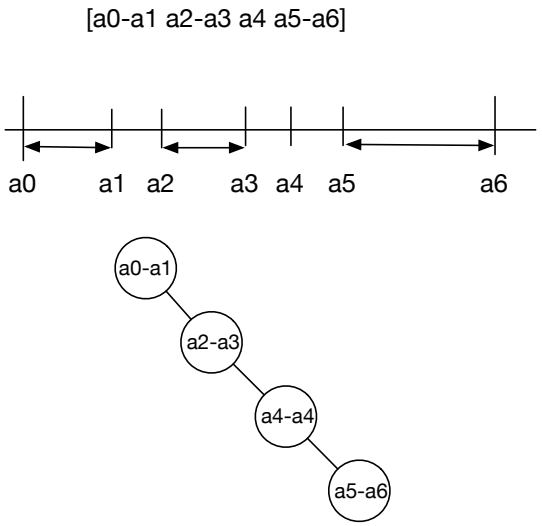

図 4.29: Character Class を二分木で表示

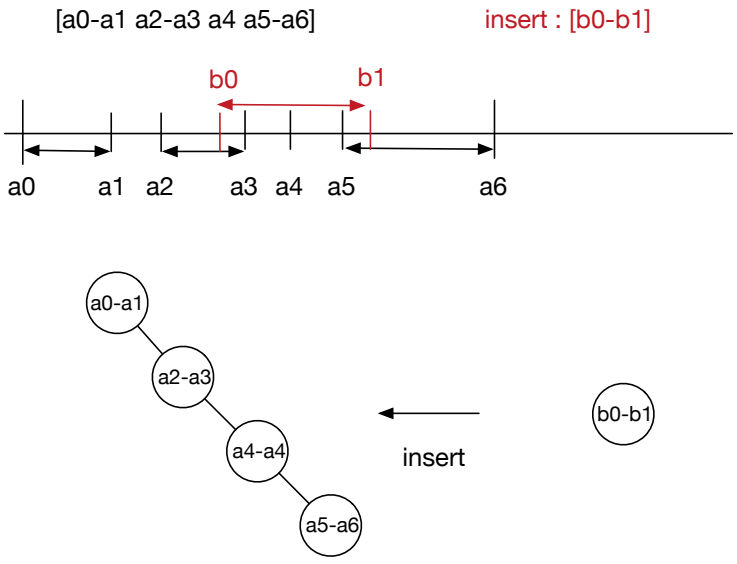

図 4.30: ある Character Class の二分木に対して、新しい Character Class を insert

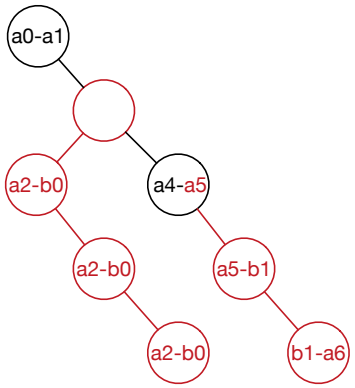

図 4.31: insert 後の Character Class の二分木

第5章　評価・考察

5.1　I/O の測定

5.2　Word Count

5.3　Boyer Moore Search

5.4　正規表現

第6章　結論

参考文献

[1] R.S. Boyer. J.S.Moore. *A Fast String Searching Algorithm*, 1977.

[2] 新屋 良磨, 鈴木 勇介, 高田 謙. 正規表現技術入門 (技術論評社), 2015.

[3] 金城裕. 並列プログラミングフレームワーク cerium の改良. 琉球大学工学部情報工学科平成 24 年度学位論文 (修士), March 2012.

[4] 渡真利勇飛. マルチプラットフォーム対応並列プログラミングフレームワーク. 琉球大学大学院理工学研究科情報工学専攻平成 26 年度学位論文 (修士), 2013.

[5] 河野 真治新屋 良磨. 動的なコード生成を用いた正規表現マッチャの実装. 第 52 回プログラミング・シンポジウム, January 2011.