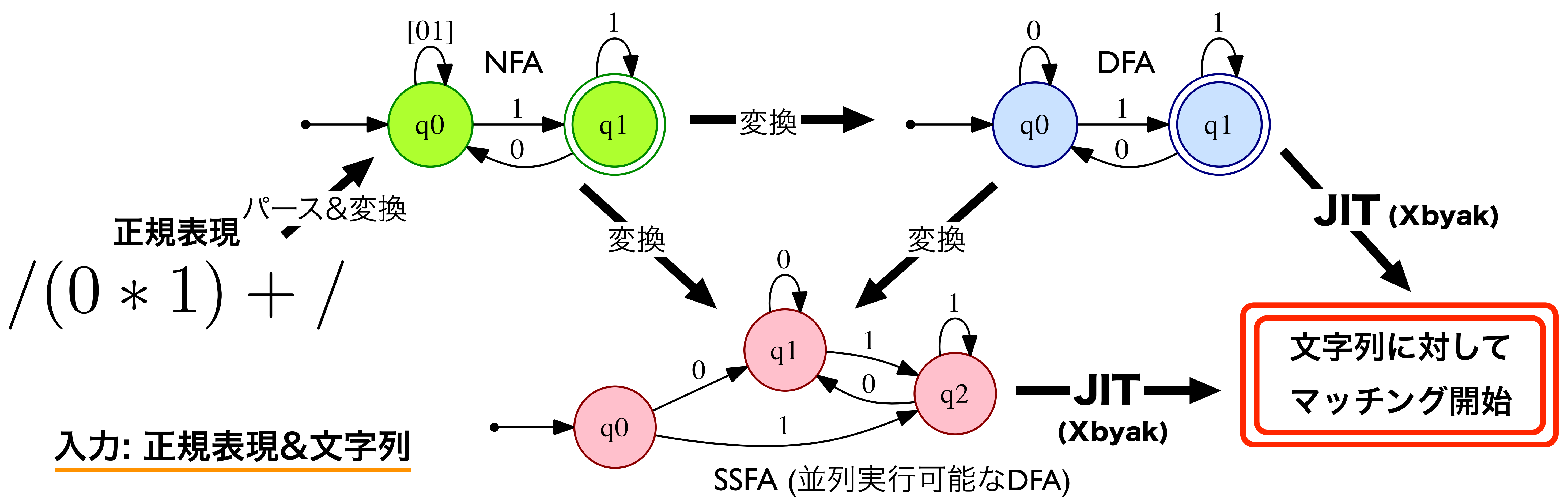


並列化と実行時コード生成を用いた正規表現マッチングの高速化

新屋 良磨[†] 光成 滋生^{††} 佐々 政孝[†]
[†] 東京工業大学 ^{††} サイボウズ・ラボ株式会社

我々は並列化と実行時コード生成を用いた高速な正規表現エンジン Regen を開発した。Regen における設計方針や特長を紹介し、既存実装との違いを説明する。

Regenのマッチング内部動作



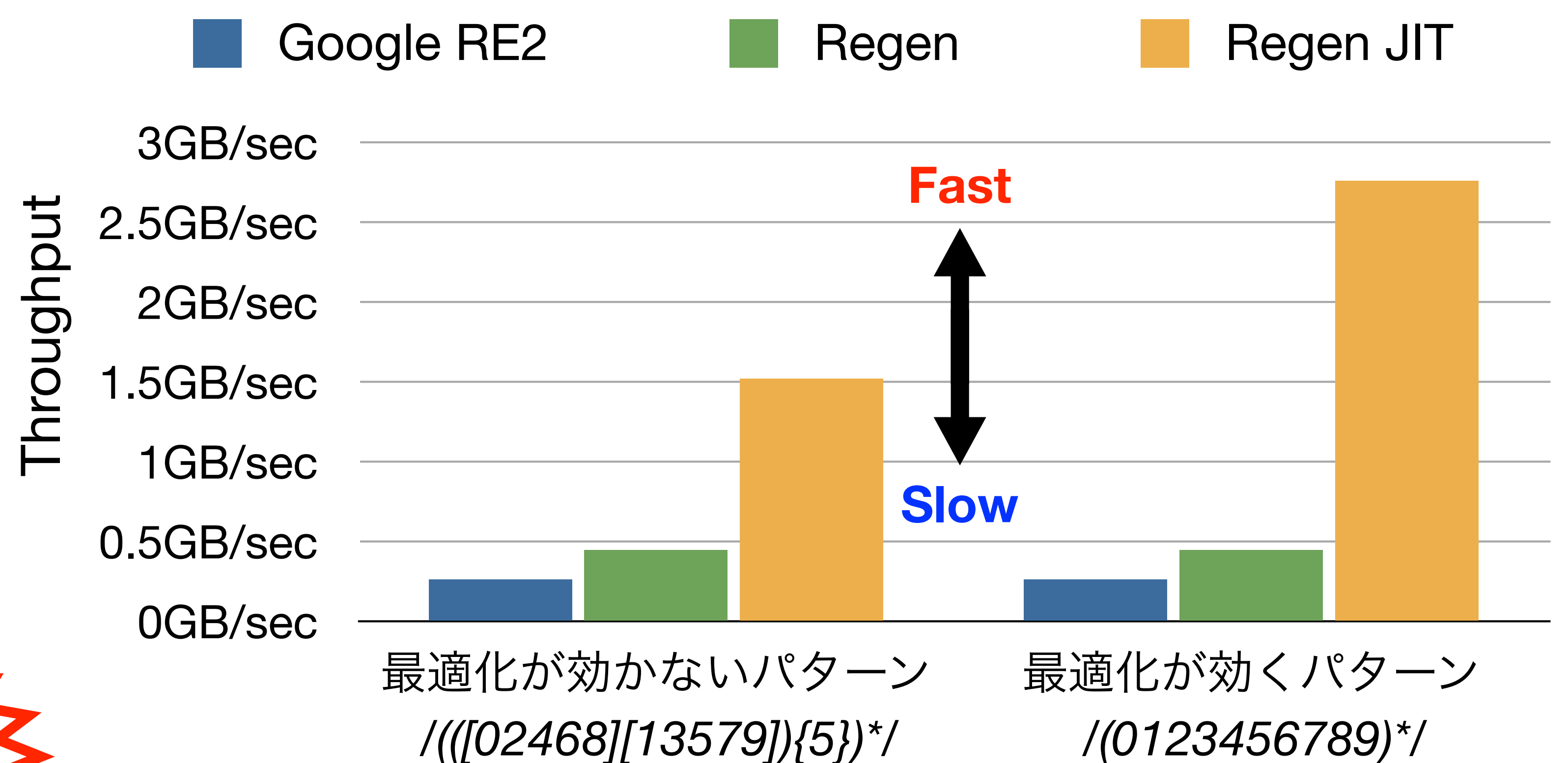
Regen内部のコード生成(JIT)部抜粋

```
if (dfa.IsAcceptState(i) &&
    !dfa.flag().suffix_match()) {
    inLocalLabel();
    cmp(arg3, 0); je(".ret");
    mov(tmp2, arg1);
    if (!dfa.flag().shortest_match()){
        jmp("@f"); }
    L(".ret");
    mov(reg_a, i); jmp("return");
    L("@@");
    outLocalLabel(); }

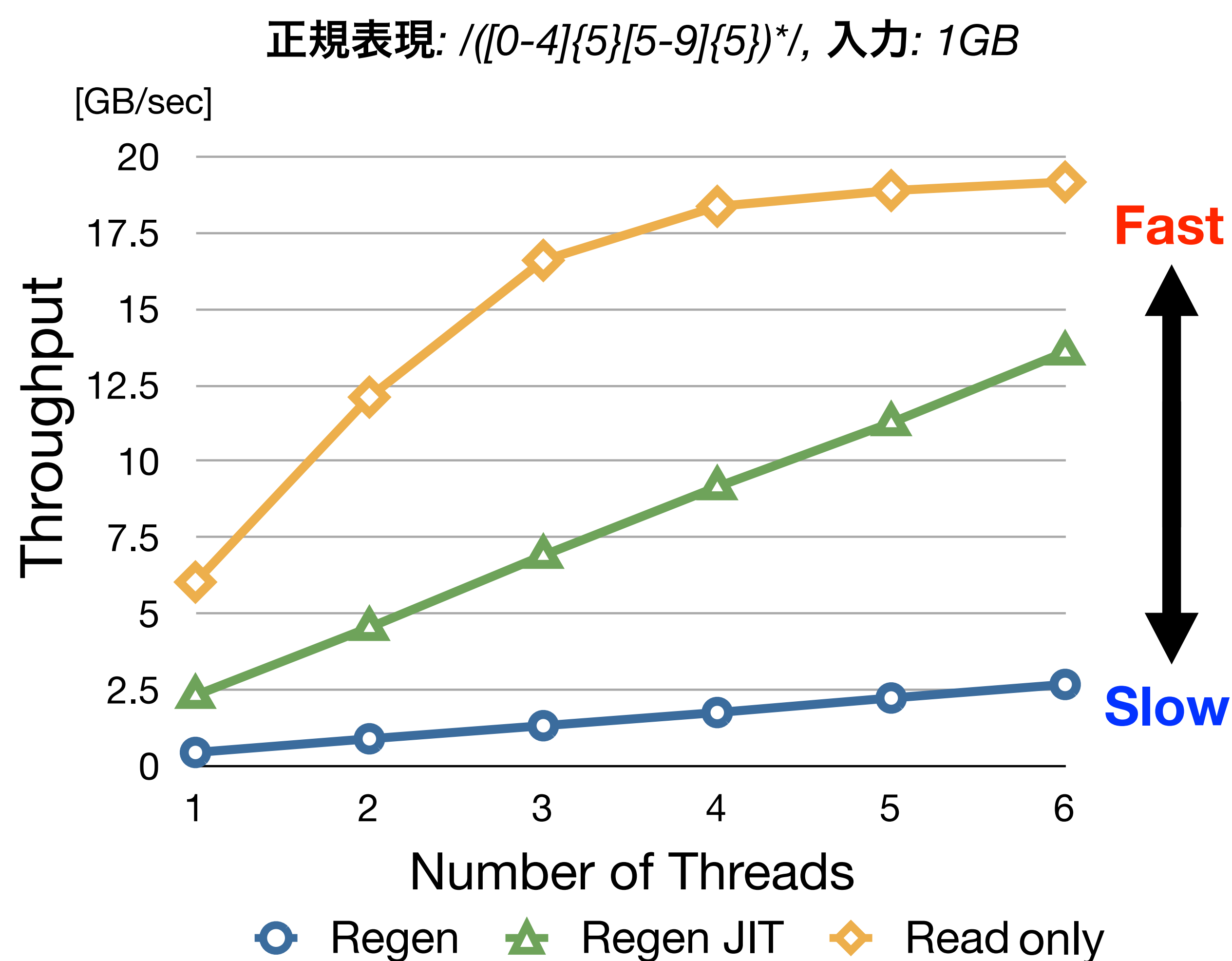
```

C++ JIT Library
Xbyak

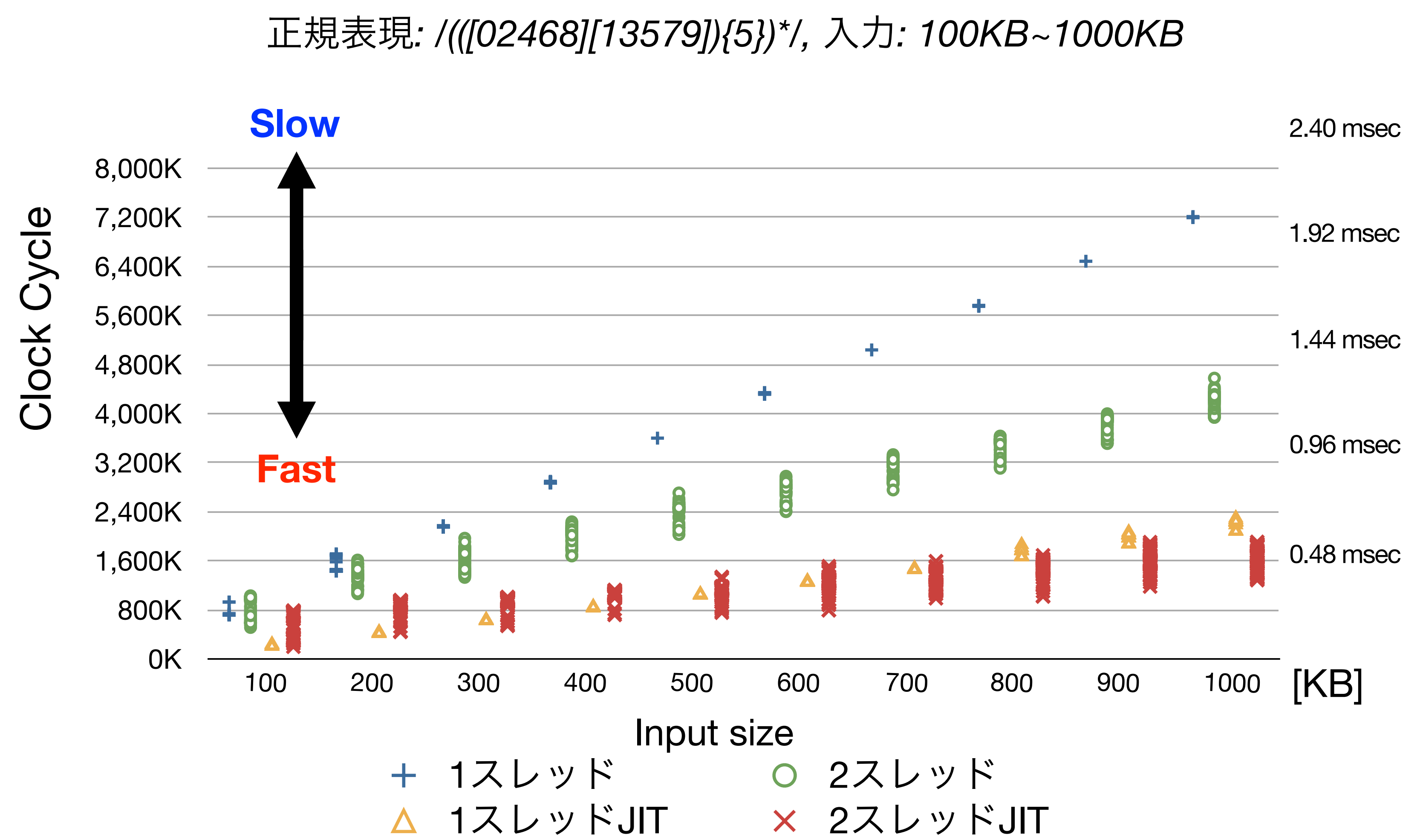
コード生成(JIT)による性能向上



並列マッチングによる性能向上



並列化のオーバーヘッド



正規表現エンジン「Regen」 & そのツール群の紹介

Regen でサポートしてる正規表現の拡張演算子, 及びツール群を紹介する。
課題や機能についてなど, 多くの意見が聞きたい。

拡張演算子

◆ 積集合

- $/R_1 \& R_2/ \rightarrow R_1$ と R_2 両方にマッチ

◆ 対称差

- $/R_1 \& \& R_2/ \rightarrow R_1$ か R_2 一方のみにマッチ

◆ 補集合

- $/!(R_1)/ \rightarrow R_1$ にマッチしない列にマッチ

◆ 逆順

- $/\sim(R_1)/ \rightarrow R_1$ のマッチ文字列の逆文字列

◆ 置換, シャッフル, 非強欲な繰り返し...

- 紹介してる演算全て DFA で実現

◆ 弱後方参照

- PCRE 等で使える後方参照の制限版

- 有限パターンの参照なら区別できる。

- $/(1|2|30?) = (\backslash1)/ \rightarrow$ 恒等式

- 無限パターン(繰り返し)は区別しない。

- $/(1|2+) = (\backslash1)/ \rightarrow /1 = 1|2+ = 2+ /$

◆ 上限付き再帰

- PCRE 等で使える再帰(?R)の制限版

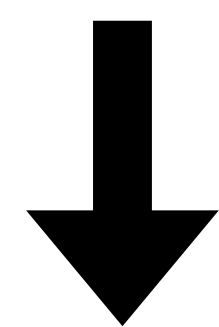
- $/a@b@c/ \rightarrow$ “*aabcbabcc*”

- $/\{(@\{0,2\}\})/ \rightarrow$ “()”, “(())”, “((()))”

正規表現から受理文字列出力

Example: 論理式を充足する解を全列挙

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

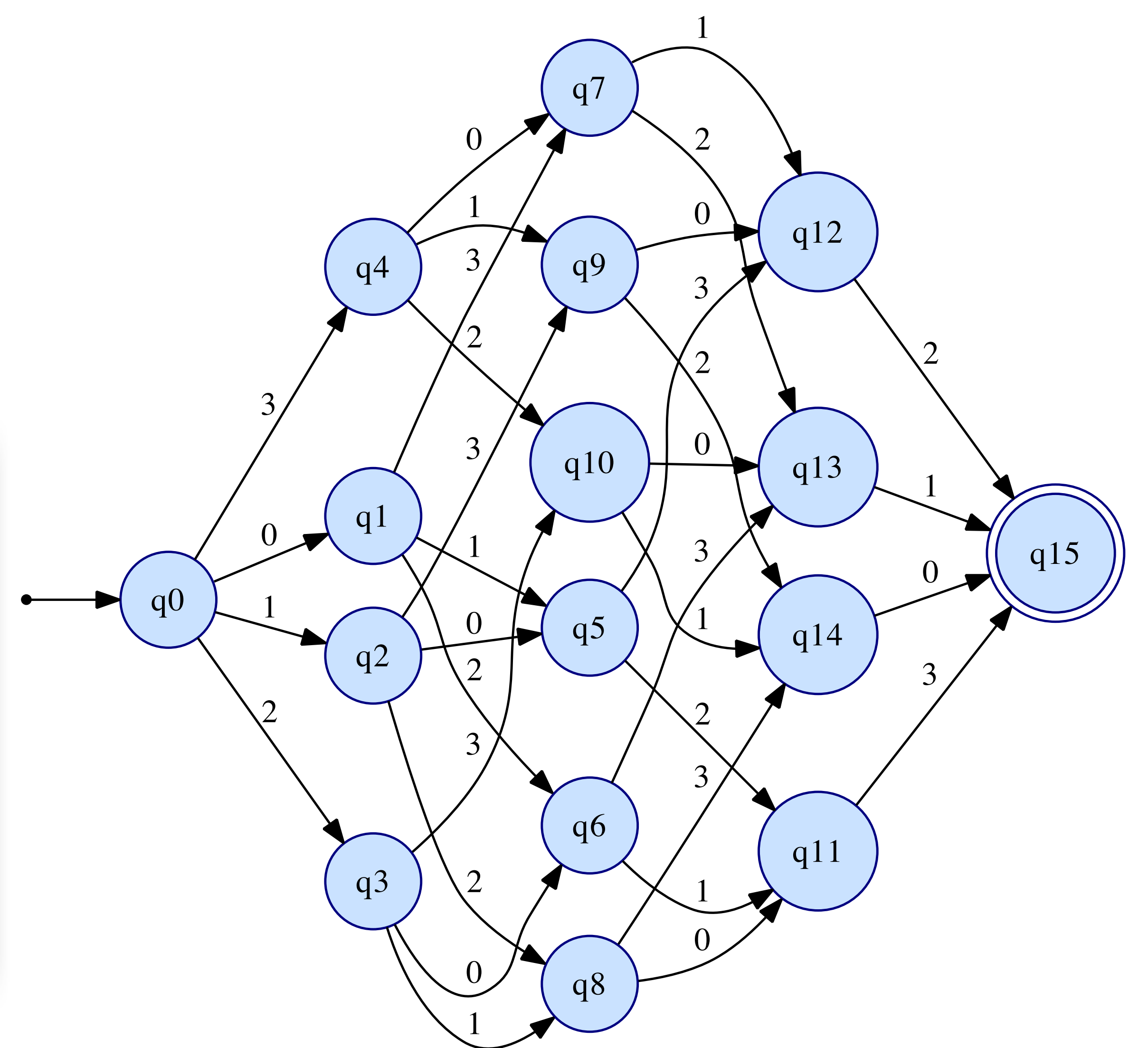


```
% ./recon -E -t '(1[01][01] | [01]1[01] | [01][01]0),  
& (0[01][01] | [01]0[01] | [01][01]1)'
```

```
000  
010  
011  
100  
101  
111
```

正規表現で
SATを解く!

正規表現からDFA(図)出力



実装済みツール群

◆ 正規表現Lib: Regen

- Google RE2 を参考
- Submatch 未対応

◆ 正規表現変換系: recon

- 正規表現 to 正規表現
- 正規表現 to DFA(図)
- 正規表現 to 受理文字列

◆ grep: regengrep

- JIT版は高速。
- まだ開発中(速度,機能)

今後の課題

◆ 他アーキテクチャ対応

- 今はX86-64のみ(Xbyak)
- LLVM? (抽象化で性能落ちない?)

◆ Submatch(キャプチャ)の対応

- 現在でもマッチした文字列全体の取得は可能。
- マッチした文字列の部分的な取得は未対応

◆ 性能を生かしたツール/システムの考案