Regular Expressions in C++, Present and Future

Tim Shen <timshen@google.com>

tuple: (T1, T2, T3)

list: [T]

set: { T }

map: K → V

Regex

Stack Overflow outage July 20, 2016

^[\s\u200c]+|[\s\u200c]+$

Regex

Stack Overflow outage July 20, 2016

^[\s\u200c]+|[\s\u200c]+$ + backtracking regex engine

Regex

Stack Overflow outage July 20, 2016

^[\s\u200c]+|[\s\u200c]+$ + backtracking regex engine = $O(n^2)$

Overview

- Regex → NFA
- Algorithms
- "Pessimizations"
- Future

Nondeterministic Finite Automaton (NFA)

```
struct State {
  StateType type;
  union {
    char ch;
    // ...
  };
  vector<State*> Successors();
};
```

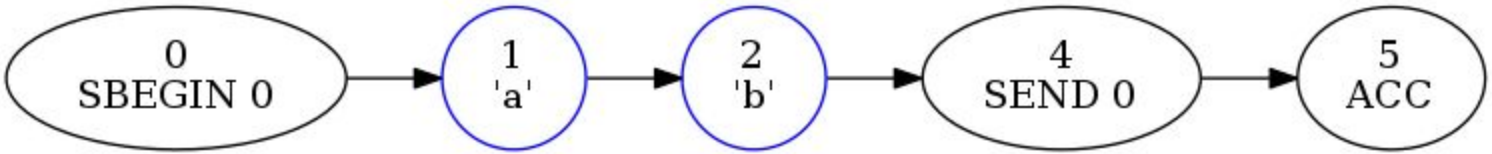
Nondeterministic Finite Automaton (NFA)

```
struct State {
  StateType type;
  union {
    char ch;
    // ...
  };
  vector<State*> Successors();
};
```

```
enum StateType {
  ACC,
  MATCH,
  ALT,
  REPEAT,
  // ...
};
```

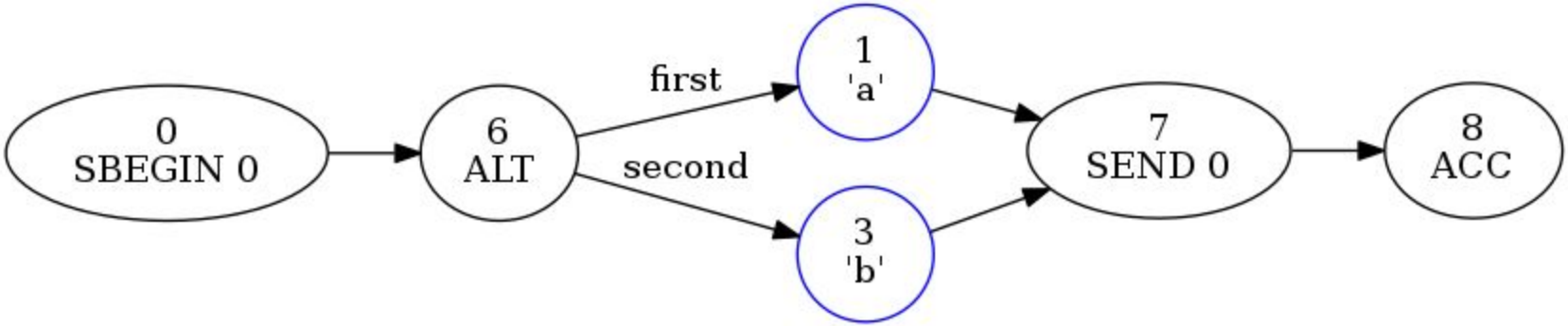
Regex → NFA

ab



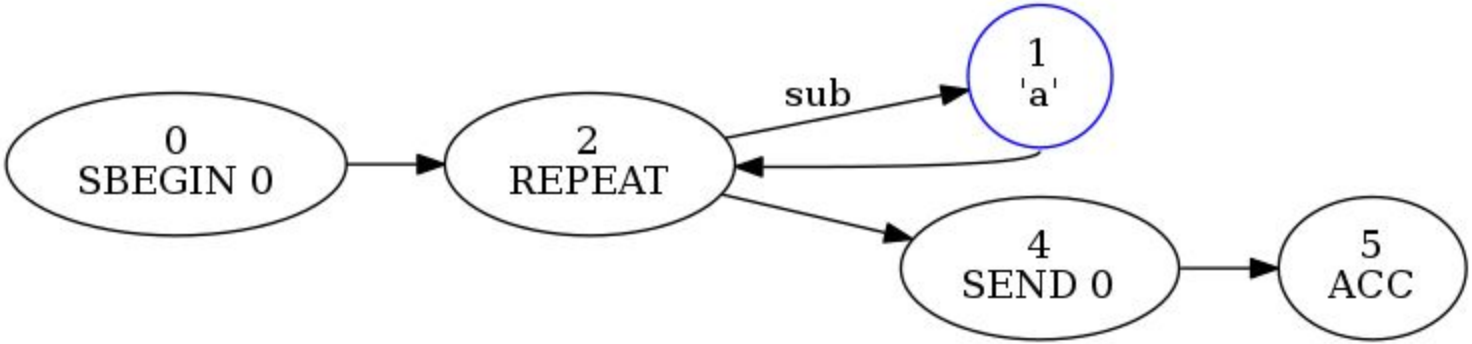
Regex → NFA

a|b



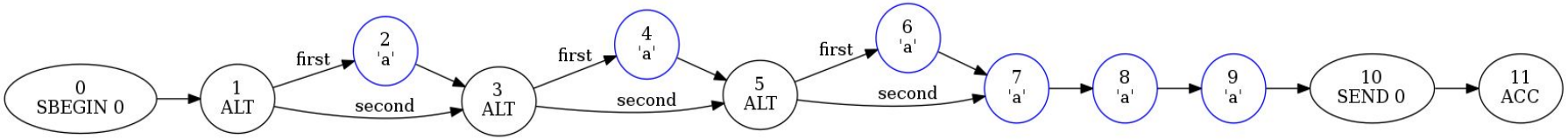
Regex → NFA

a*



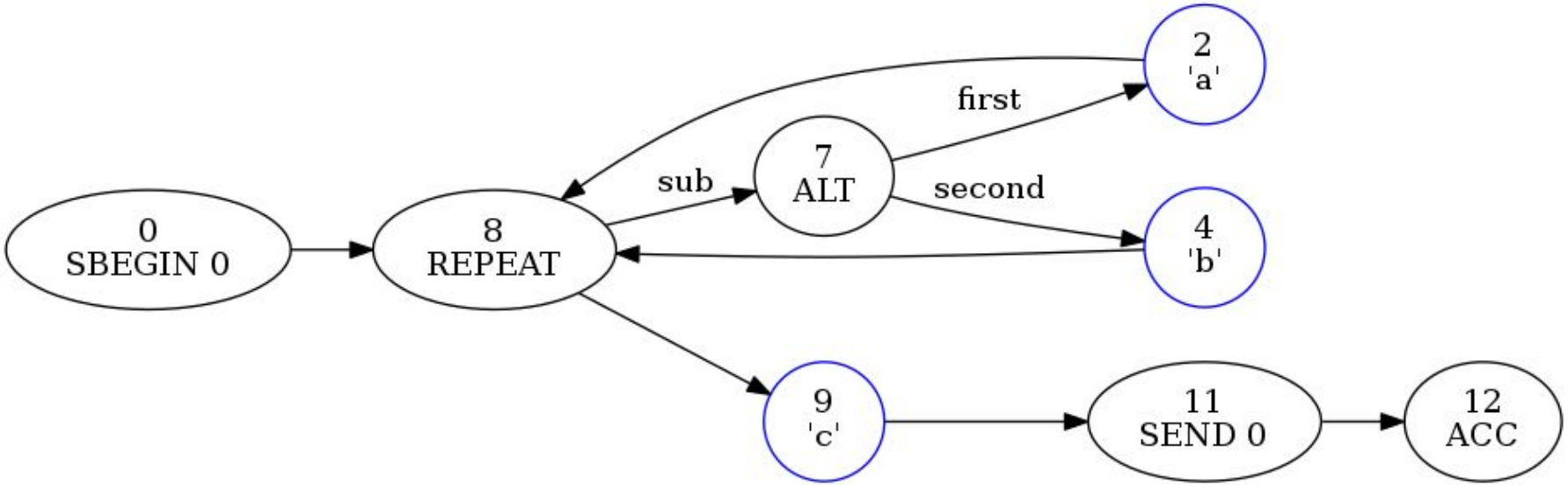
Regex → NFA

a?a?a?aaa



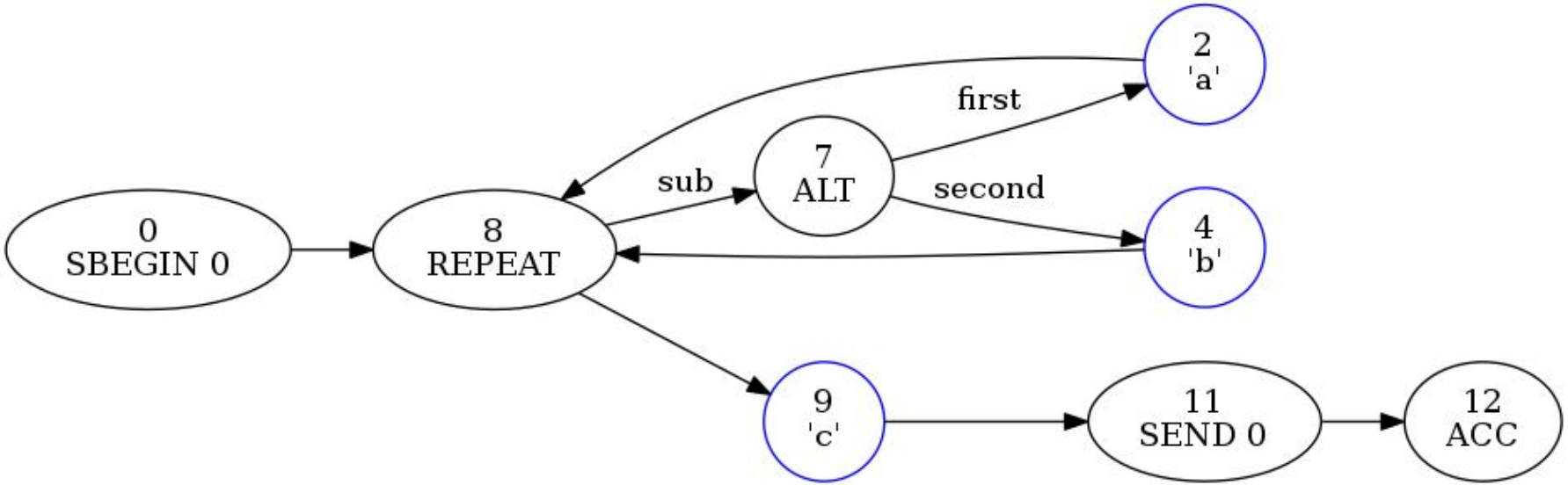
Regex → NFA

(a|b)*c



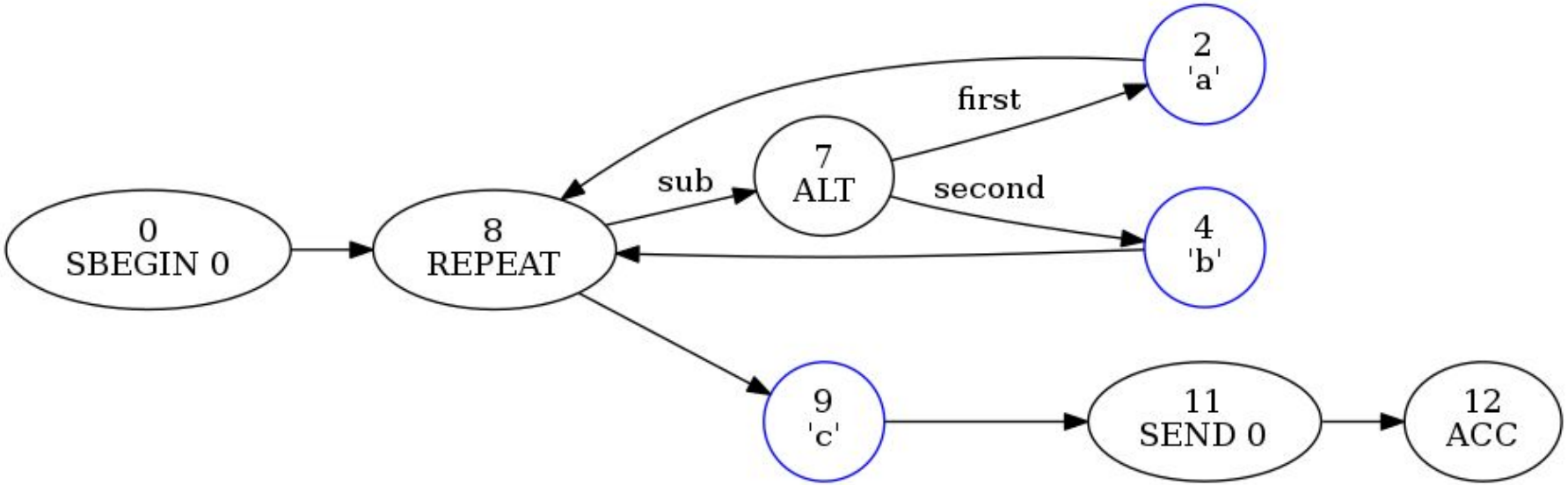
Regex → NFA

(a|b)*c ← abac



Regex → NFA

(a|b)*c ← abac

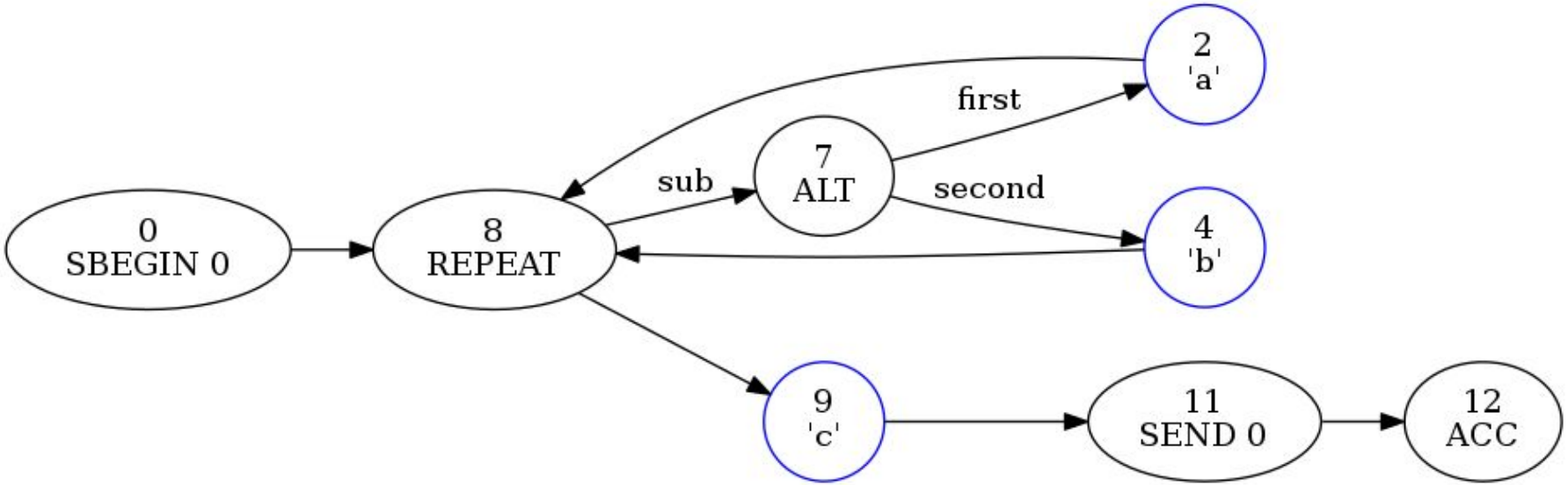


Path: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 9, 11, 12]

Depth-First Search (DFS)

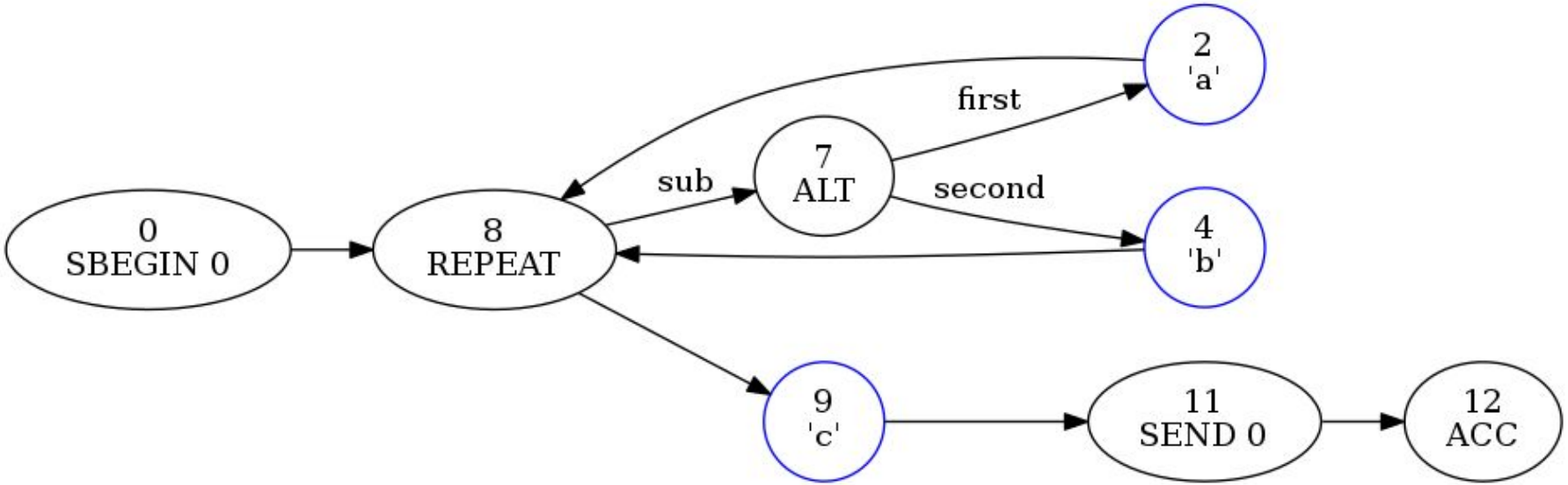
Depth-First Search (DFS)

(a|b)*c ← abac



Depth-First Search (DFS)

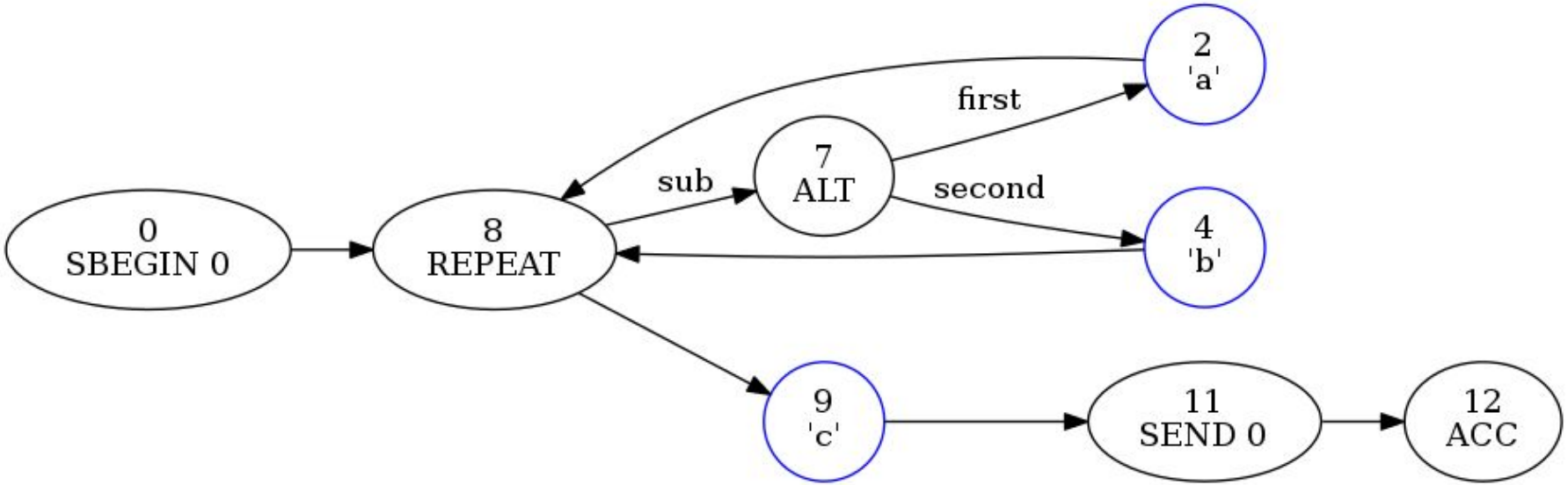
(a|b)*c ← abac



Stack: [0]

Depth-First Search (DFS)

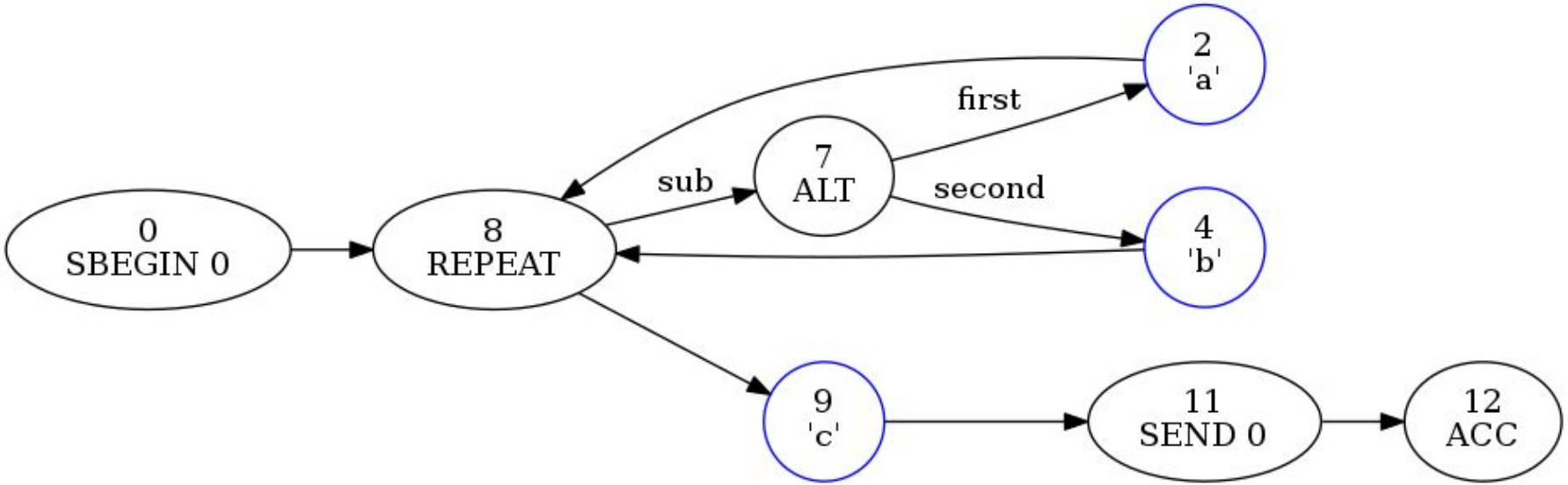
(a|b)*c ← abac



Stack: [0, 8]

Depth-First Search (DFS)

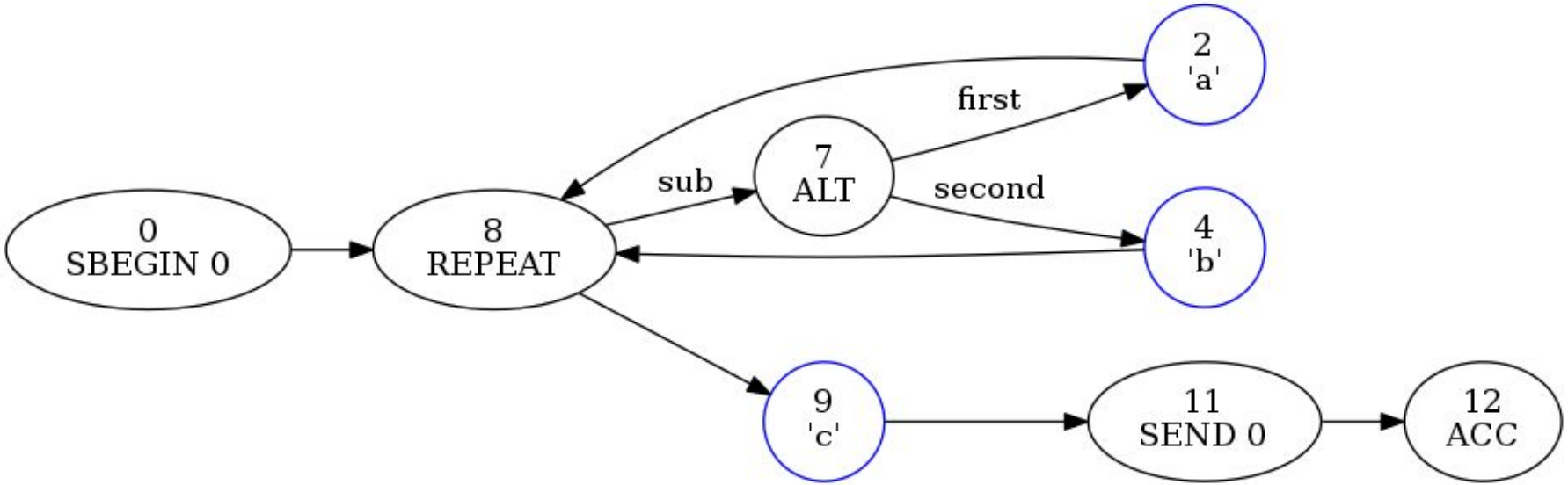
(a|b)*c ← abac



Stack: [0, 8, 7]

Depth-First Search (DFS)

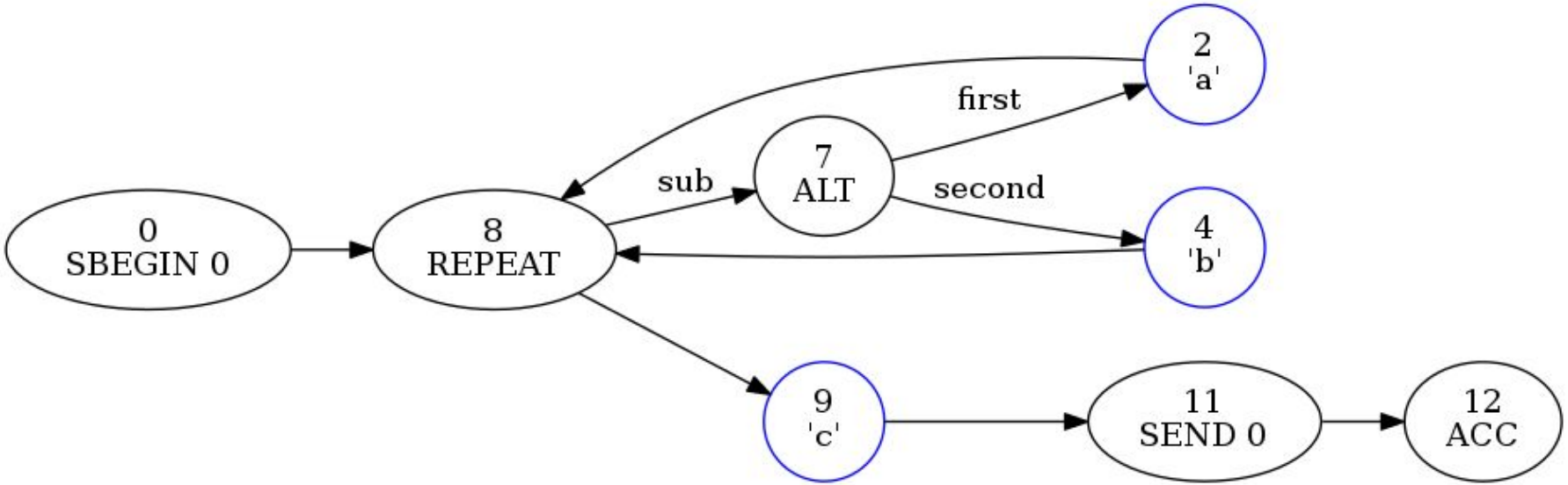
(a|b)*c ← abac



Stack: [0, 8, 7, 2]

Depth-First Search (DFS)

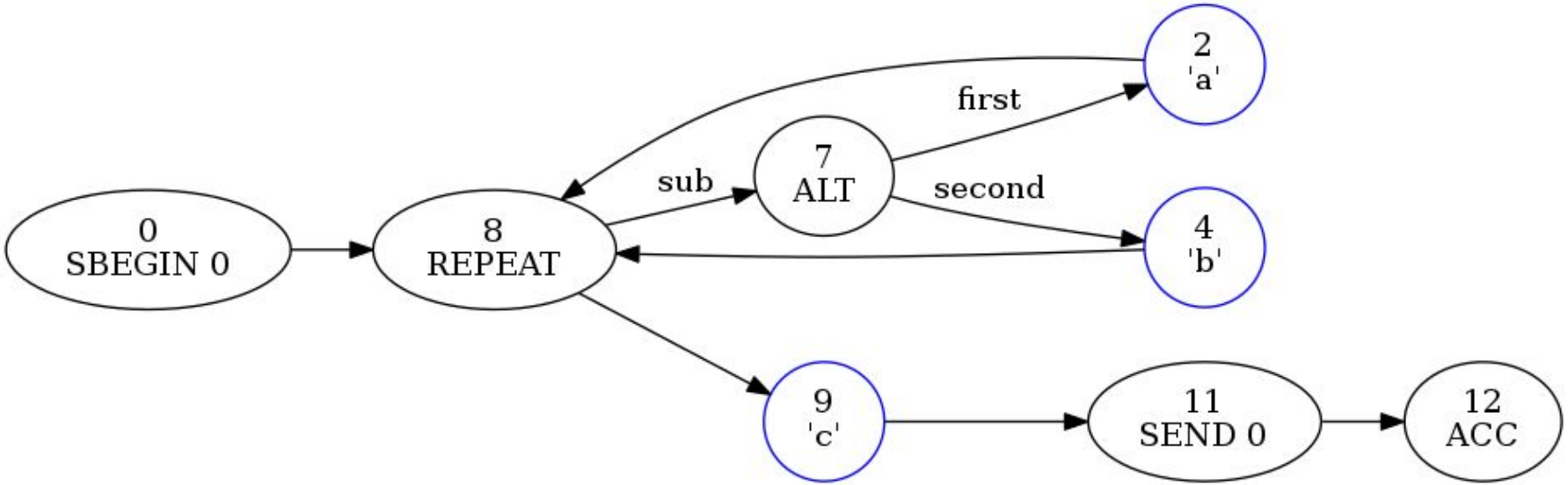
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8]

Depth-First Search (DFS)

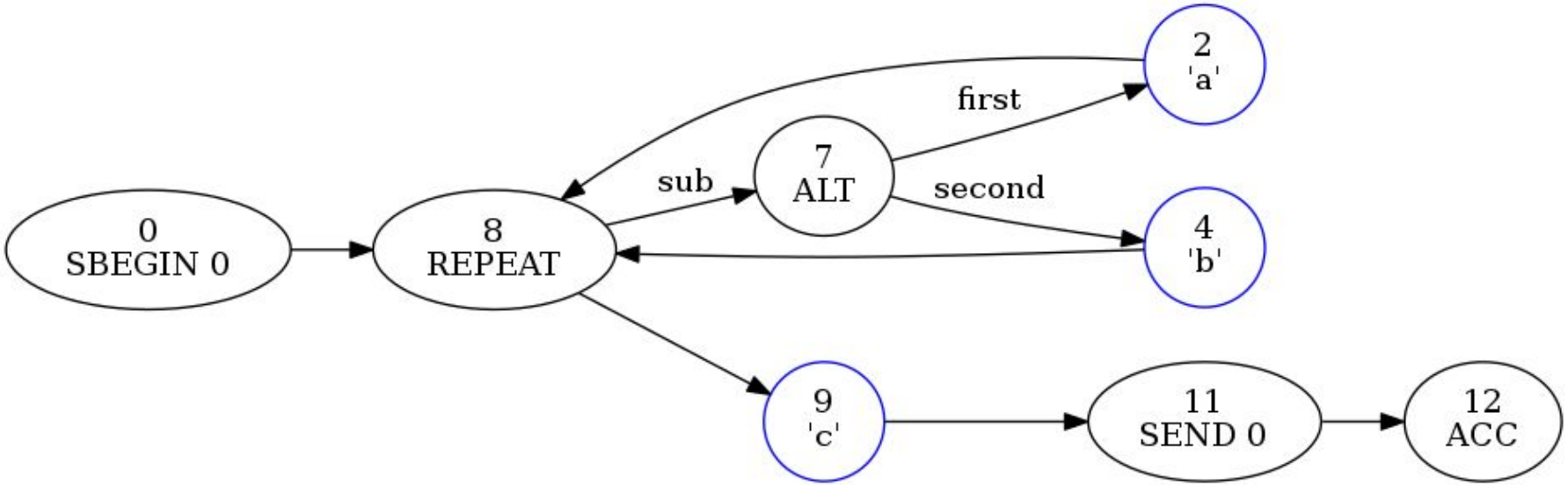
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7]

Depth-First Search (DFS)

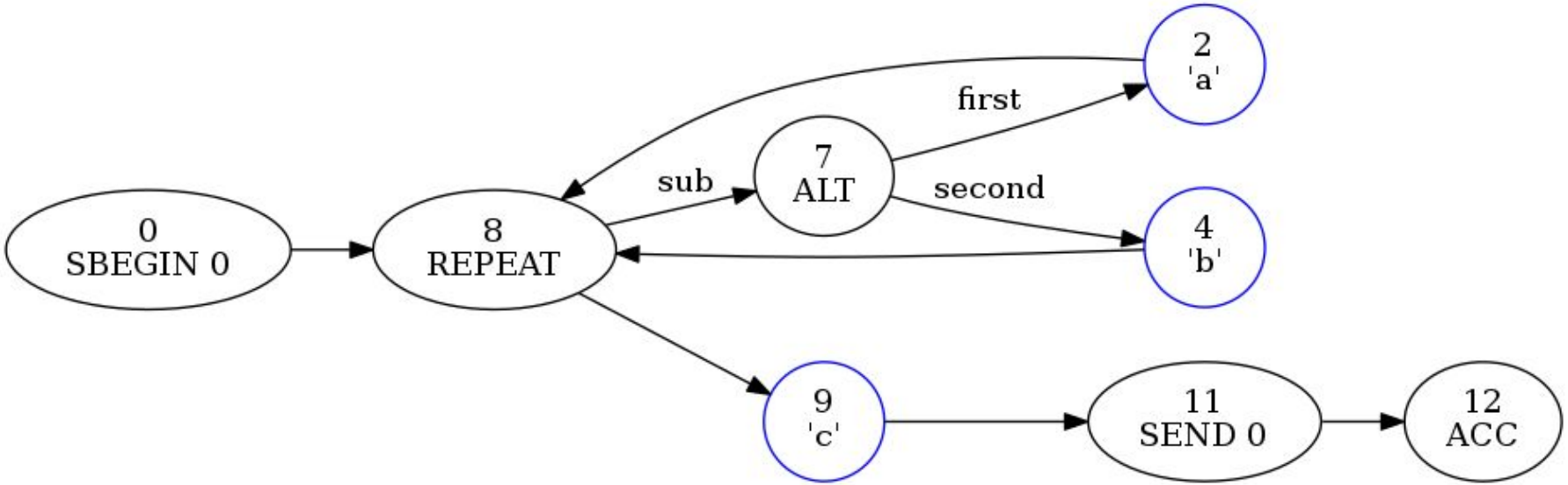
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 2]

Depth-First Search (DFS)

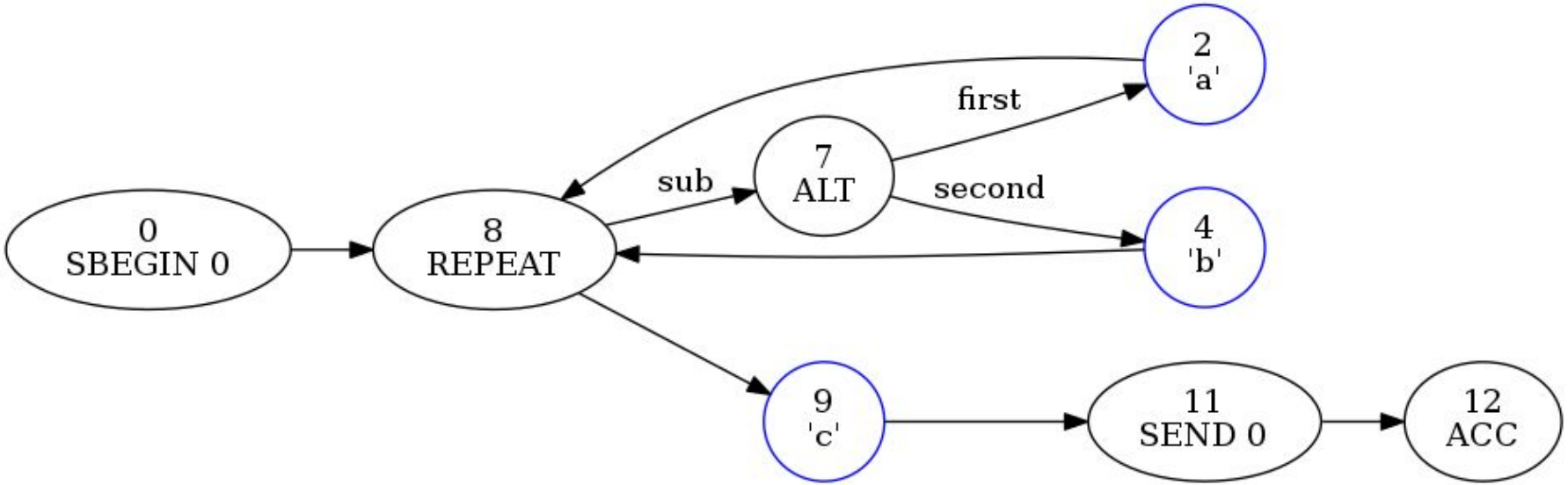
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7]

Depth-First Search (DFS)

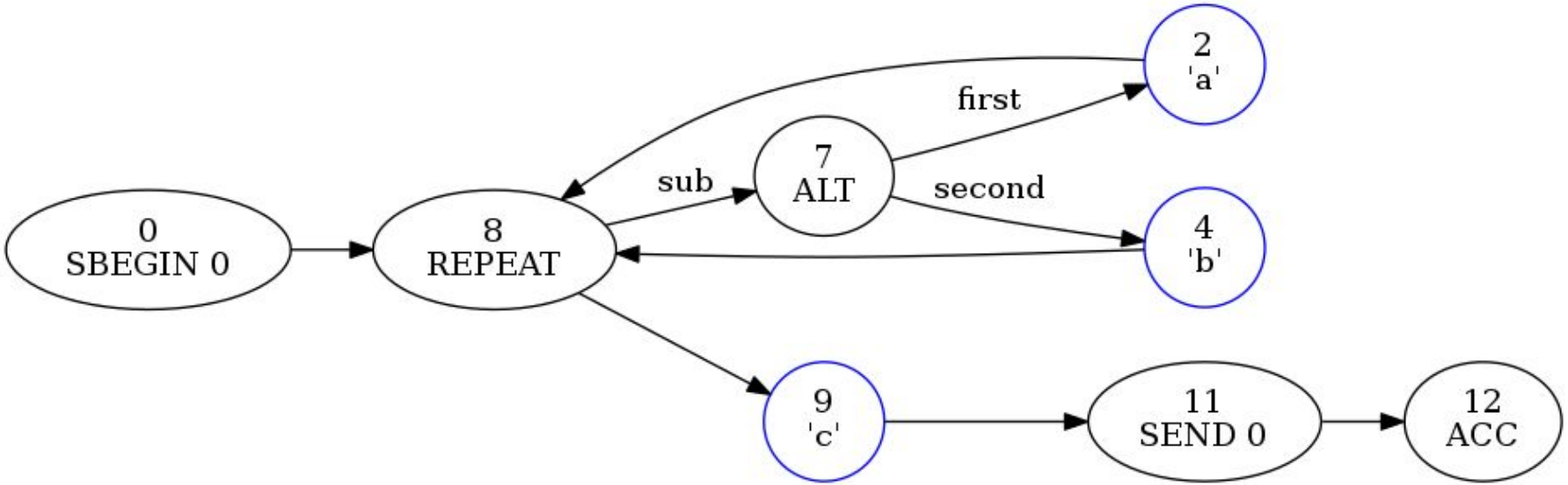
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4]

Depth-First Search (DFS)

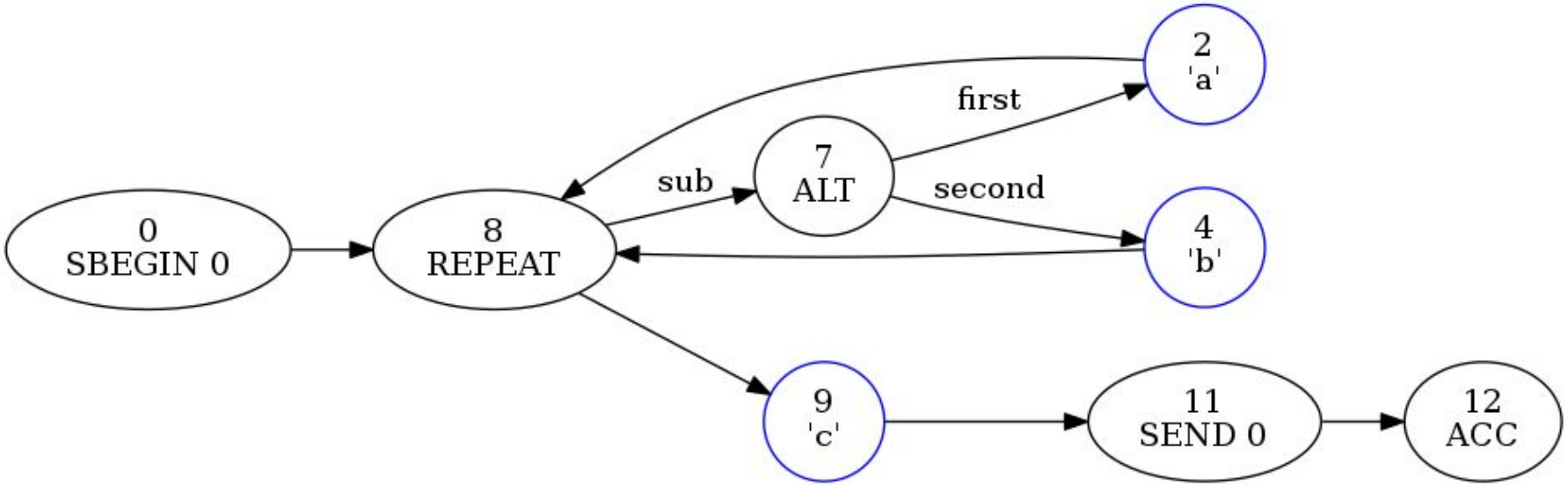
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8]

Depth-First Search (DFS)

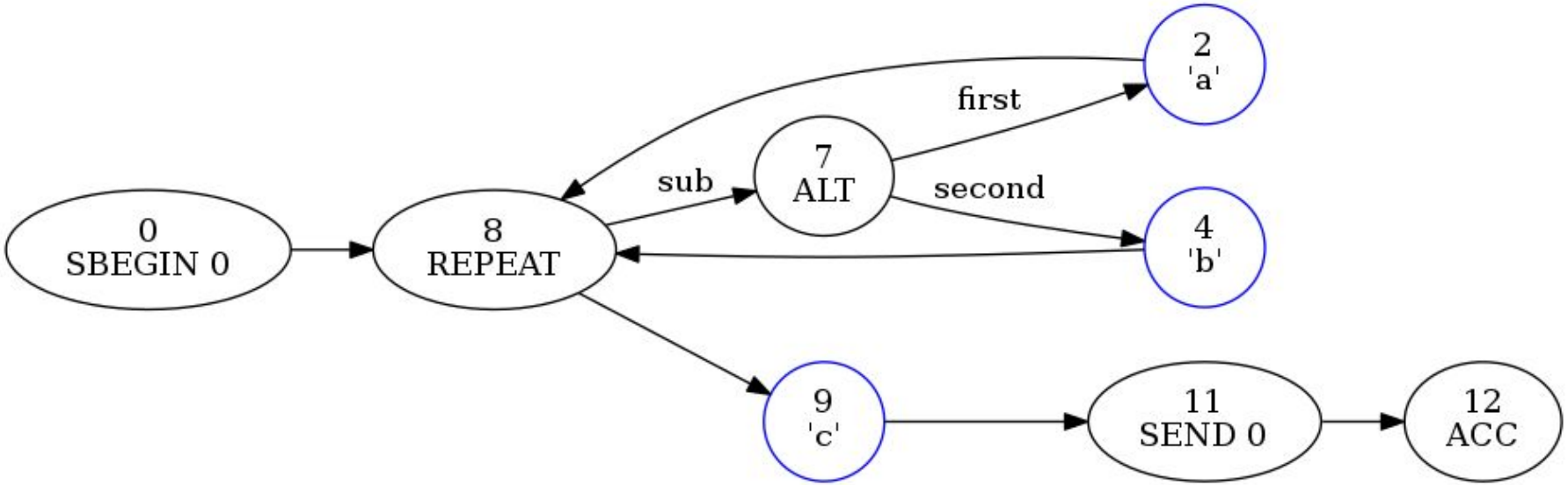
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7]

Depth-First Search (DFS)

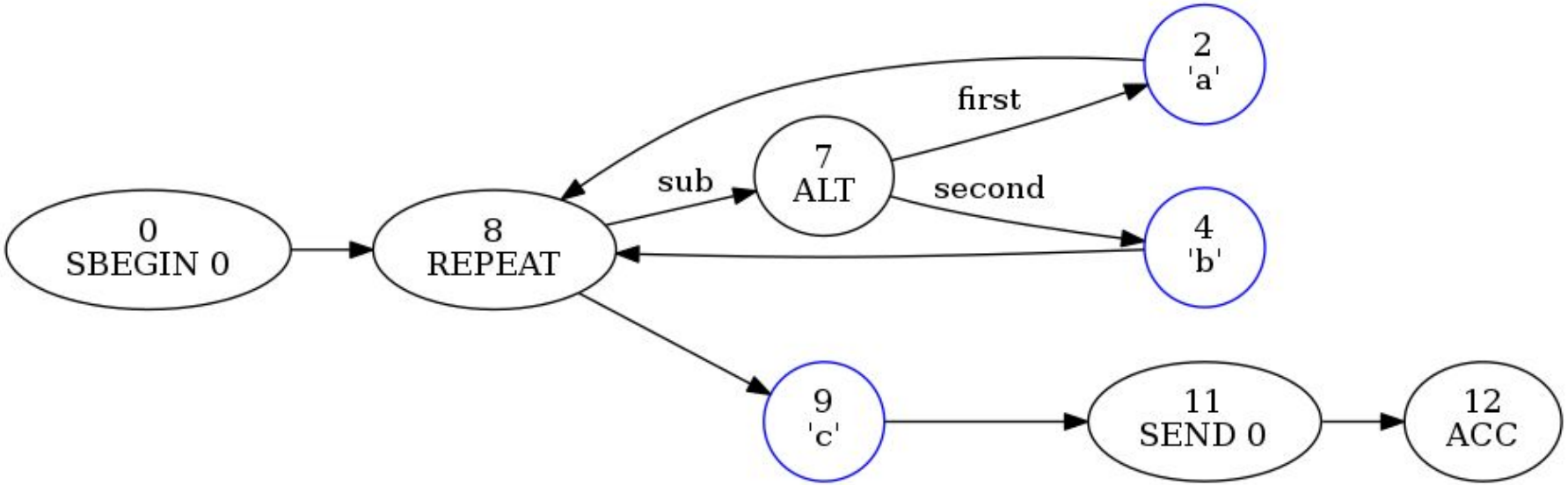
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2]

Depth-First Search (DFS)

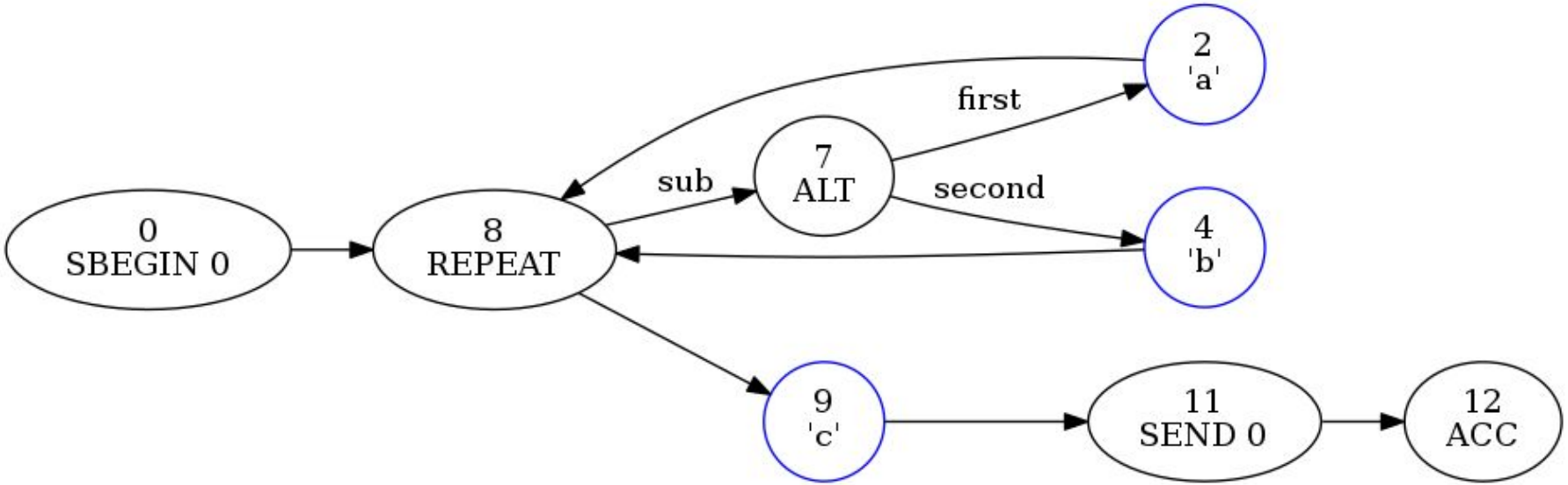
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8]

Depth-First Search (DFS)

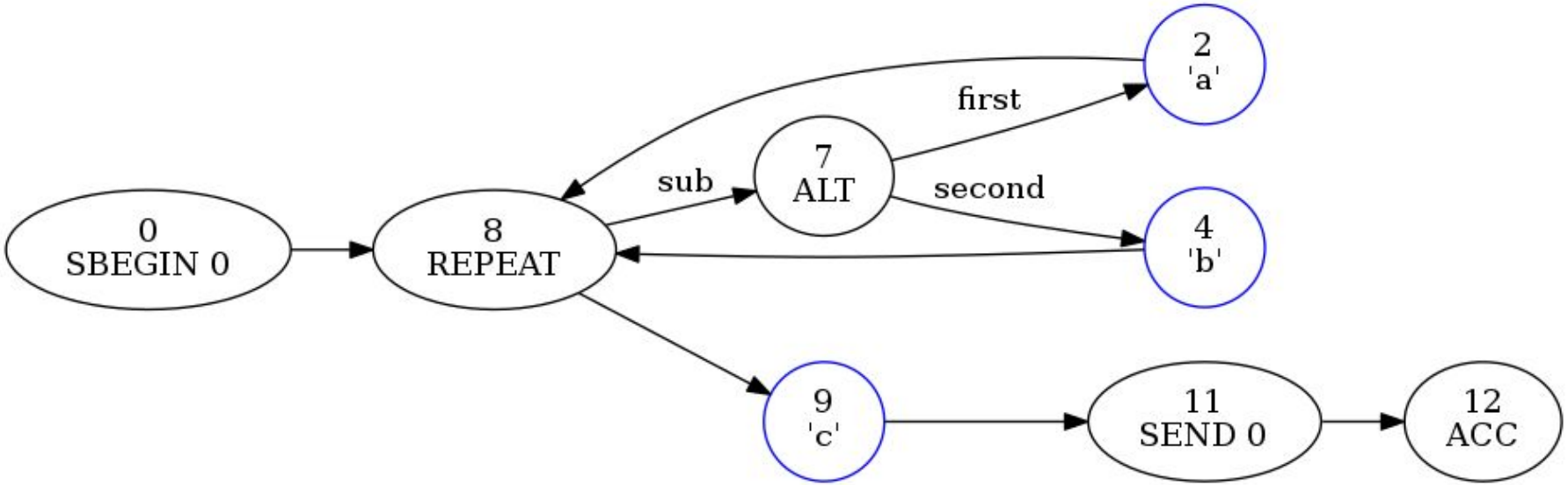
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 7]

Depth-First Search (DFS)

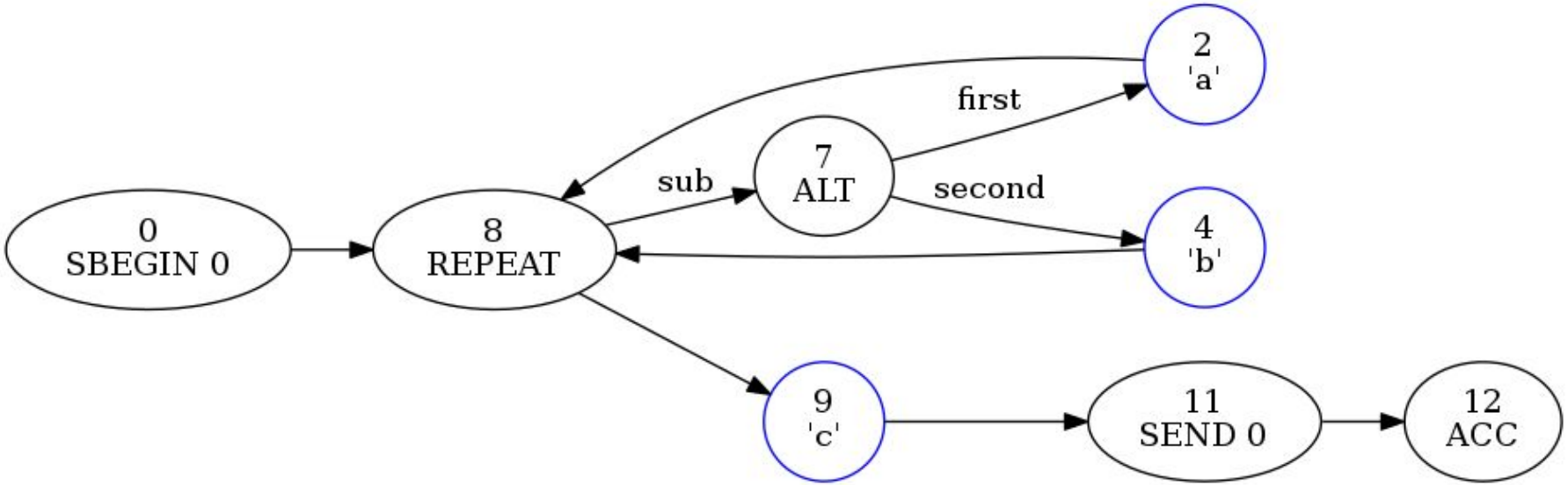
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 7, 2]

Depth-First Search (DFS)

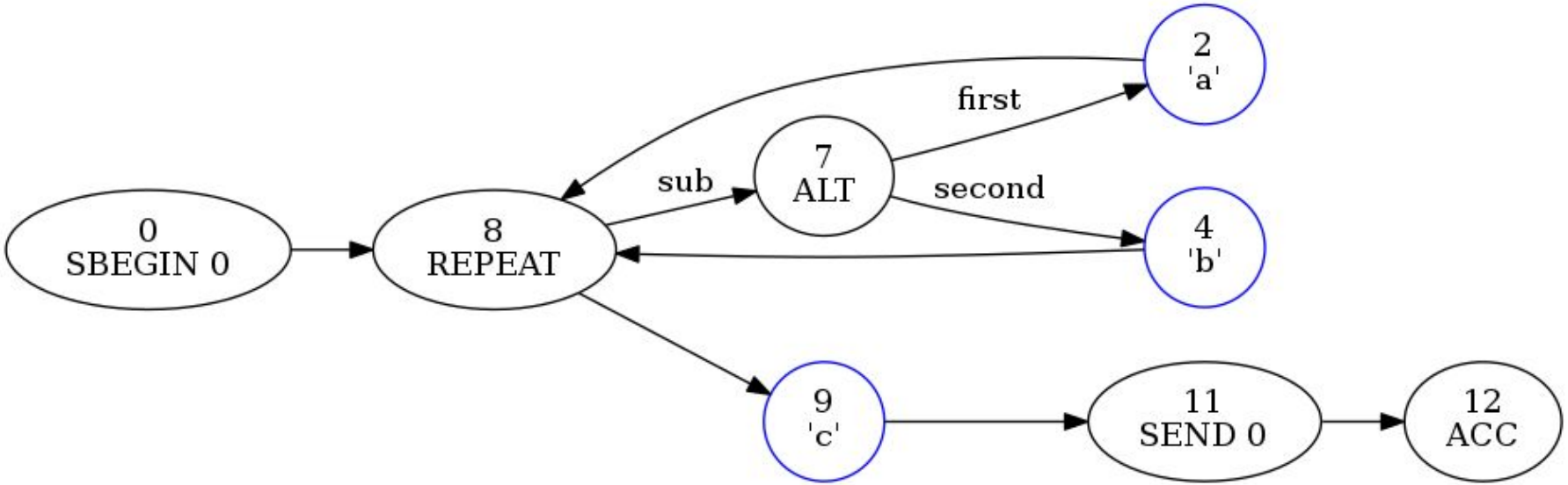
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 7]

Depth-First Search (DFS)

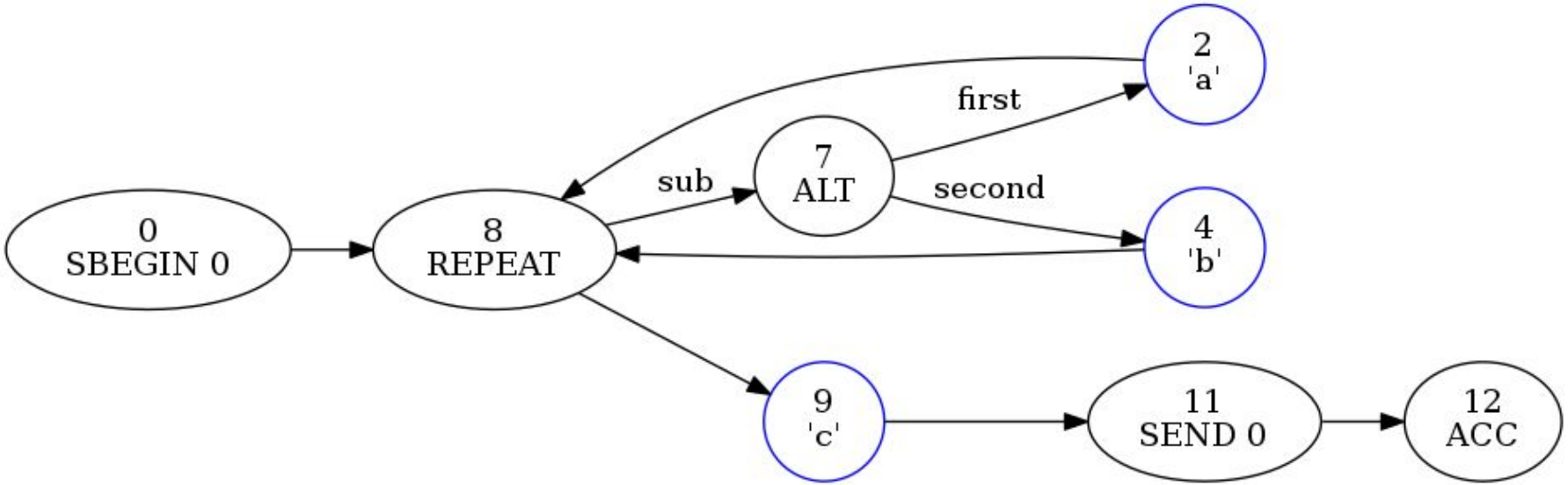
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 7, 4]

Depth-First Search (DFS)

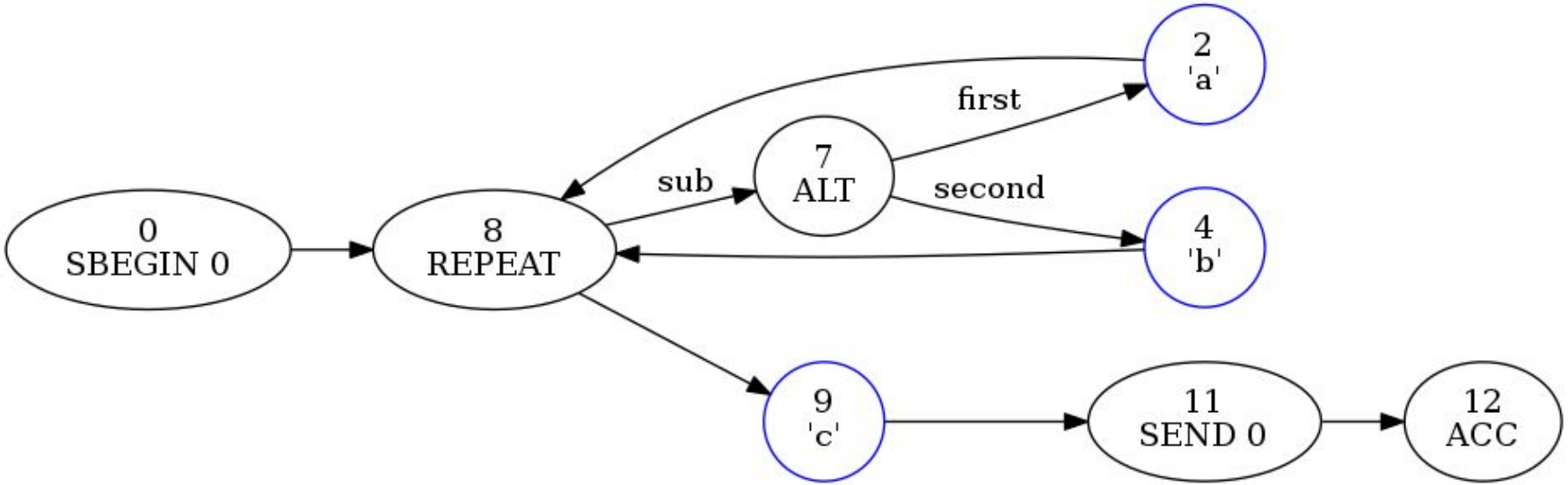
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 7]

Depth-First Search (DFS)

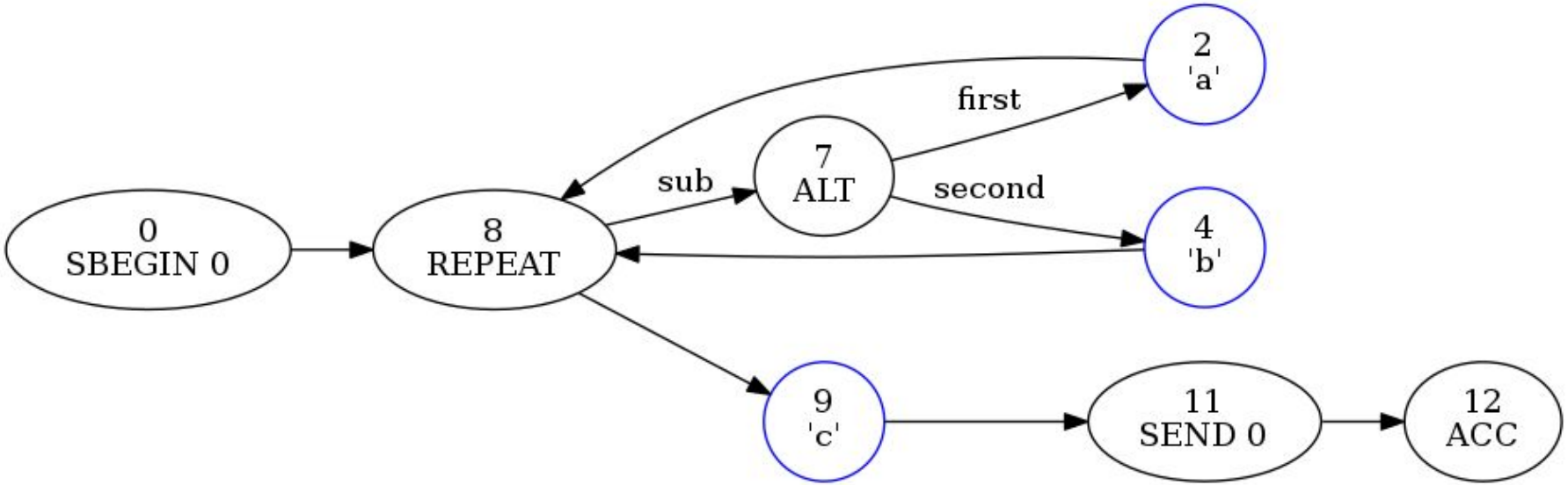
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8]

Depth-First Search (DFS)

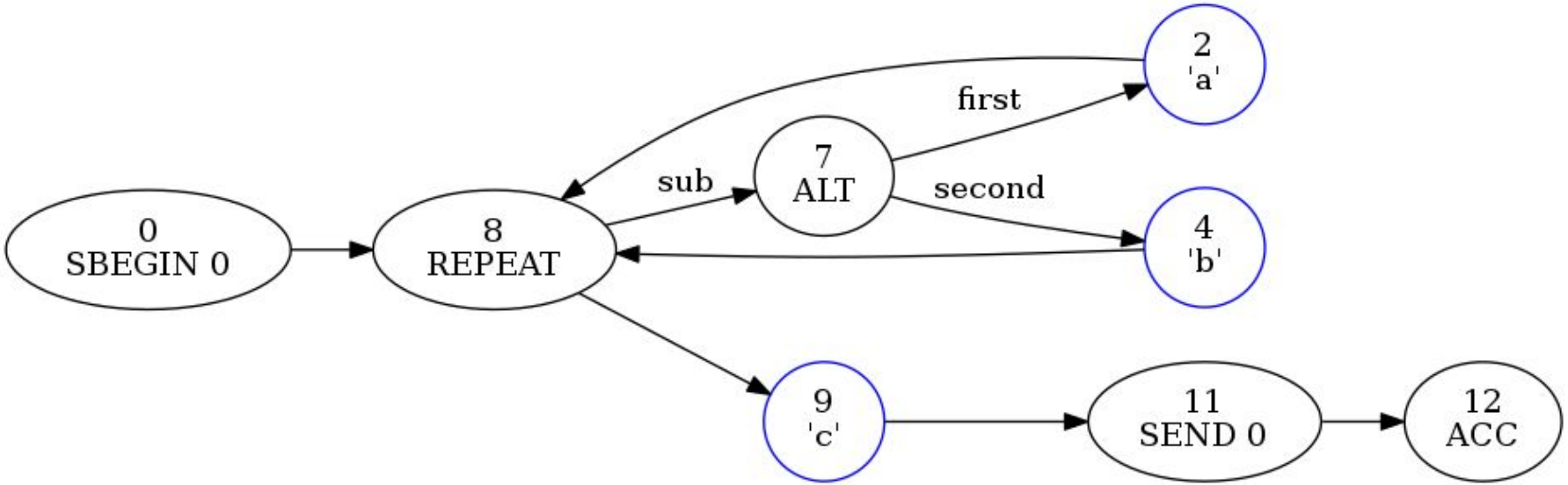
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 9]

Depth-First Search (DFS)

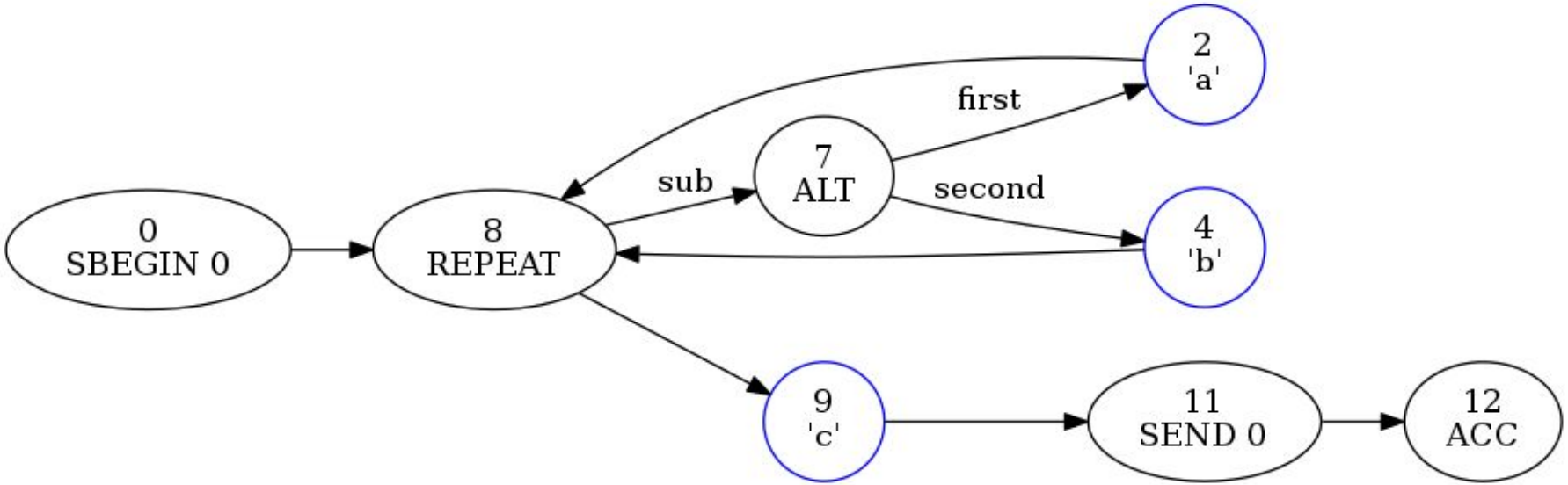
(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 9, 11]

Depth-First Search (DFS)

(a|b)*c ← abac



Stack: [0, 8, 7, 2, 8, 7, 4, 8, 7, 2, 8, 9, 11, 12]

Depth-First Search (DFS)

```
bool Dfs(const State& state, const char* target) {









}
```

Depth-First Search (DFS)

```
bool Dfs(const State& state, const char* target) {

  if (state.type == ACC) return true;









}
```

Depth-First Search (DFS)

```
bool Dfs(const State& state, const char* target) {

  if (state.type == ACC) return true;

  if (state.type == MATCH && *target++ != state.ch) return false;









}
```

Depth-First Search (DFS)

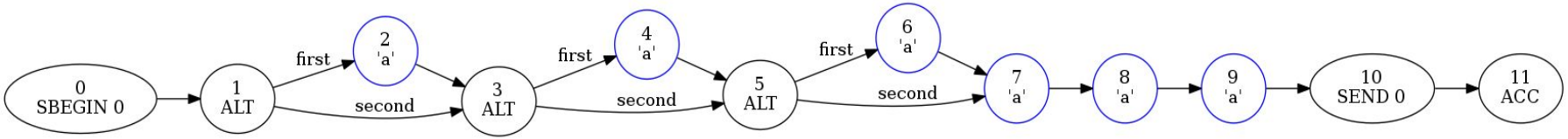
```
bool Dfs(const State& state, const char* target) {
  if (state.type == ACC) return true;
  if (state.type == MATCH && *target++ != state.ch) return false;
  for (const State* next : state.Successors())
    if (Dfs(*next, target)) return true;

}
```

Depth-First Search (DFS)

```
bool Dfs(const State& state, const char* target) {
  if (state.type == ACC) return true;
  if (state.type == MATCH && *target++ != state.ch) return false;
  for (const State* next : state.Successors())
    if (Dfs(*next, target)) return true;
  return false;
}
```

Depth-First Search (DFS)

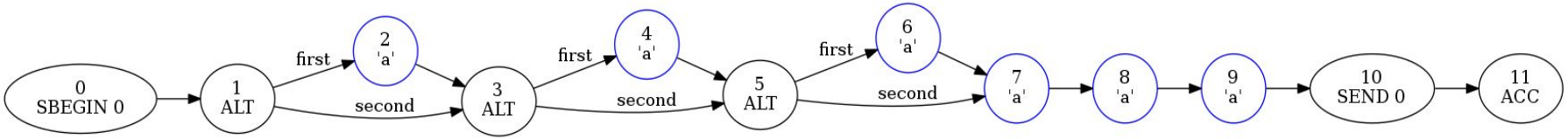
a?a?a?aaa ← aaa



Number of paths = 2^3

Depth-First Search (DFS)

a?{n}a{n} ← a...a for n times



Number of paths = 2^n

Depth-First Search (DFS)

Time Complexity:

Exponential

Space Complexity:

O(|target| + |states|)

DFS & Memoization

DFS & Memoization

```
bool Dfs(const State& state, const char* target);
```

DFS & Memoization

```
bool Dfs(const State& state, const char* target);
```

(State, String) → bool

DFS & Memoization

```
bool Dfs(const State& state, const char* target);
```

(State, String) → bool

in C++:

DFS & Memoization

```
bool Dfs(const State& state, const char* target);
```

(State, String) → bool

in C++:

- `map<pair<const State*, const char*>, bool>`

DFS & Memoization

```
bool Dfs(const State& state, const char* target);
```

(State, String) → bool

in C++:

- `map<pair<const State*, const char*>, bool>`
- `bool memoized[1024][65536]`

DFS & Memoization

```
bool Dfs(const State& state, const char* target);
```

(State, String) → bool

in C++:

- `map<pair<const State*, const char*>, bool>`
- `bool memoized[1024][65536]`
- …

DFS & Memoization

```
map<pair<const State*, const char*>, bool> memoized;
```

DFS & Memoization

```
map<pair<const State*, const char*>, bool> memoized;

bool Dfs(const State& state, const char* target) {




}
```

DFS & Memoization

```
map<pair<const State*, const char*>, bool> memoized;

bool Dfs(const State& state, const char* target) {

  auto key = make_pair(&state, target);




}
```

DFS & Memoization

```
map<pair<const State*, const char*>, bool> memoized;

bool Dfs(const State& state, const char* target) {
  auto key = make_pair(&state, target);
  if (memoized.count(key)) return memoized[key];


}
```

DFS & Memoization

```
map<pair<const State*, const char*>, bool> memoized;

bool Dfs(const State& state, const char* target) {
  auto key = make_pair(&state, target);
  if (memoized.count(key)) return memoized[key];
  return memoized[key] = DfsImpl(state, target);
}
```

DFS & Memoization

Time Complexity:

O(|target| * |states|)

Space Complexity:

O(|target| * |states|)

DFS & Memoization

Time Complexity:

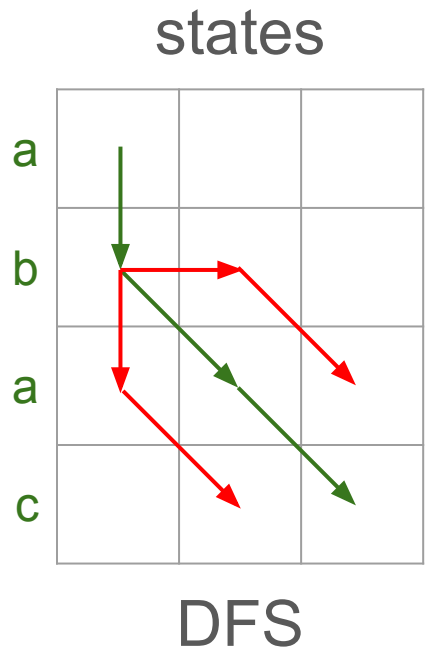
O(|target| * |states|)

Space Complexity:

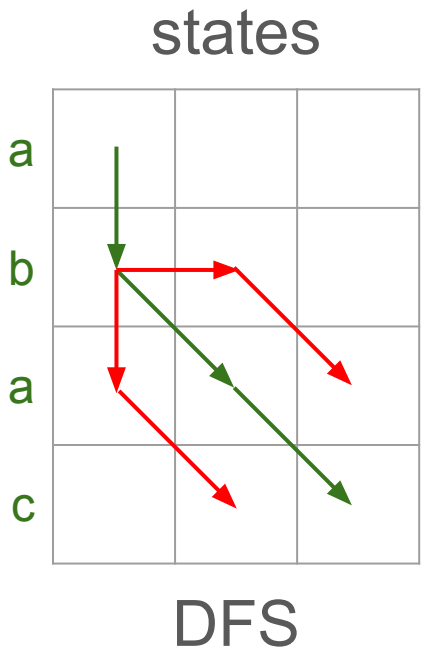
O(|target| * |states|)

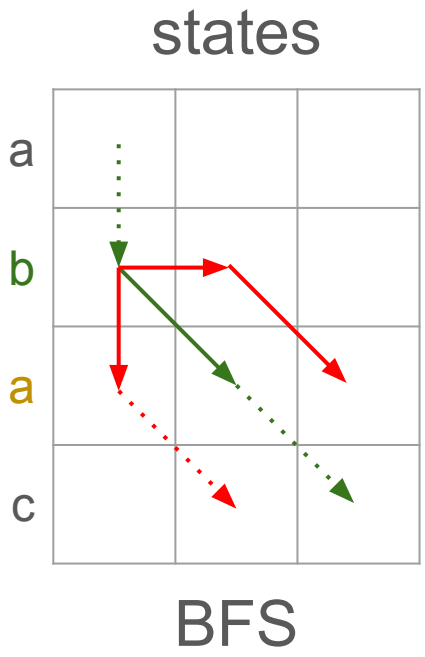
Breadth-First Search (BFS)

Breadth-First Search (BFS)



Breadth-First Search (BFS)





Breadth-First Search (BFS)

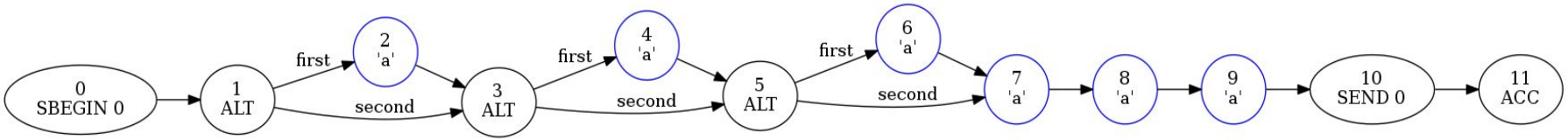
EpsilonClosure(*s*) : { State } → { State }

returns states that are

- reachable from any input states
- by at least one epsilon moves

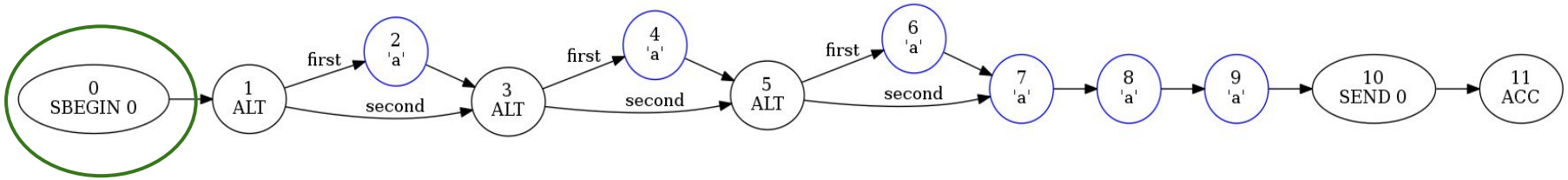
Breadth-First Search (BFS)

a?a?a?aaa



Breadth-First Search (BFS)

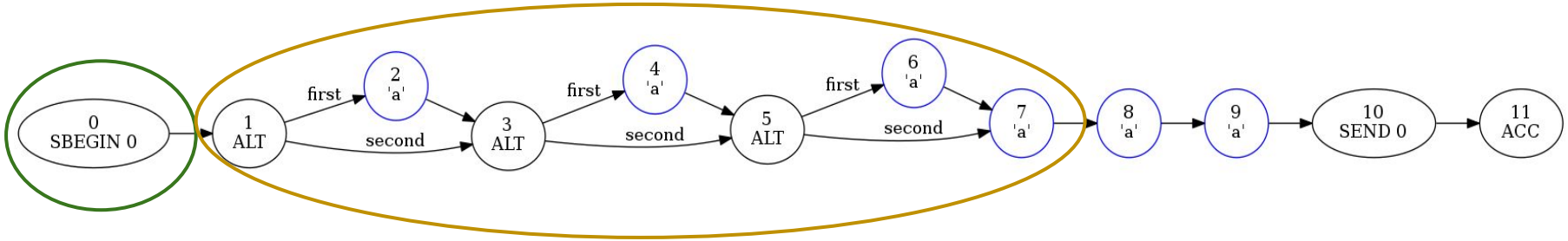
a?a?a?aaa ← aaa



Active = { 0 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

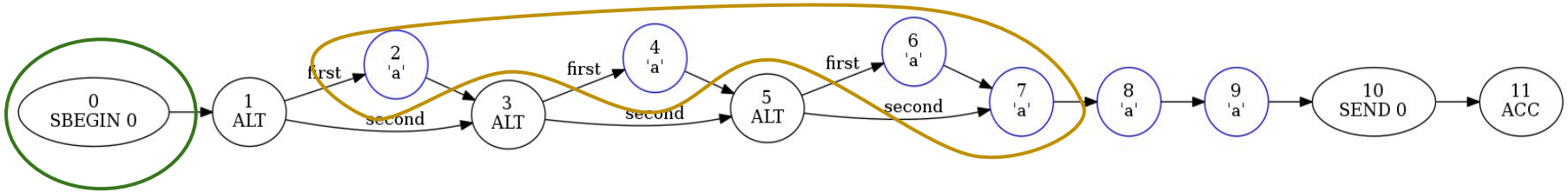


Active = { }

EpsilonClosure({ 0 }) = { 1, 2, 3, 4, 5, 6, 7 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

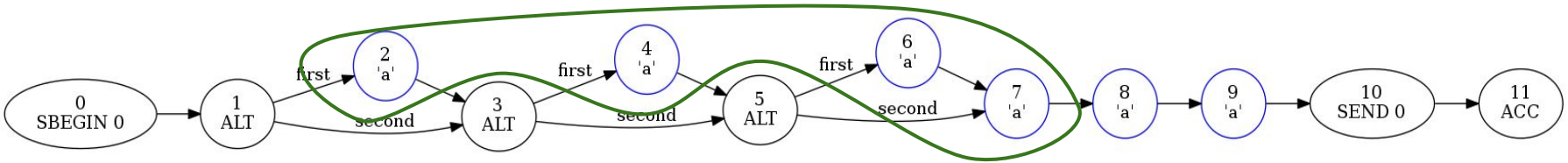


Active = { }

EpsilonClosure({ 0 }).FilterBy('a') = { 2, 4, 6, 7 }

Breadth-First Search (BFS)

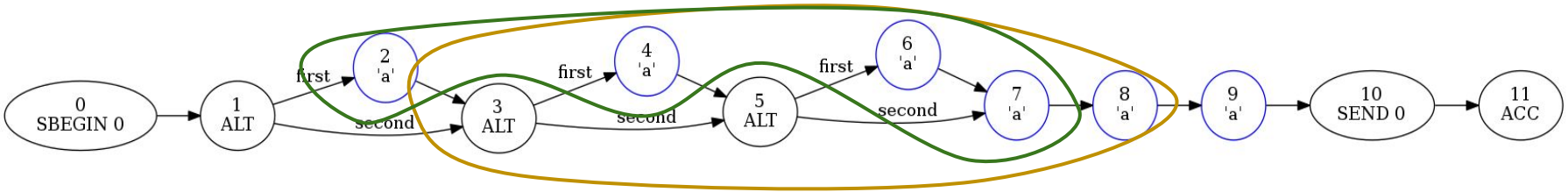
a?a?a?aaa ← aaa



Active = { 2, 4, 6, 7 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

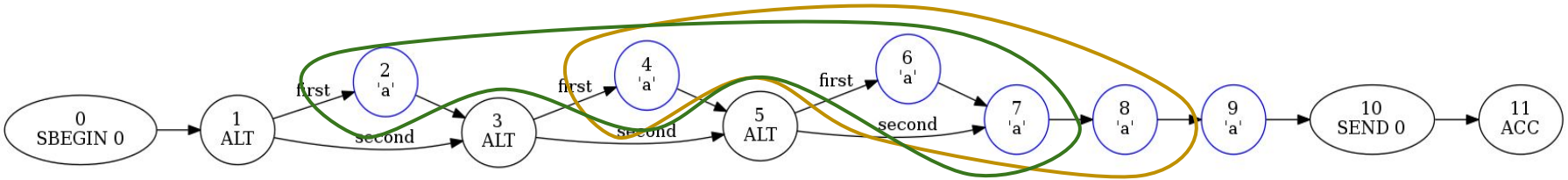


Active = { }

EpsilonClosure({ 2, 4, 6, 7 }) = { 3, 4, 5, 6, 7, 8 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

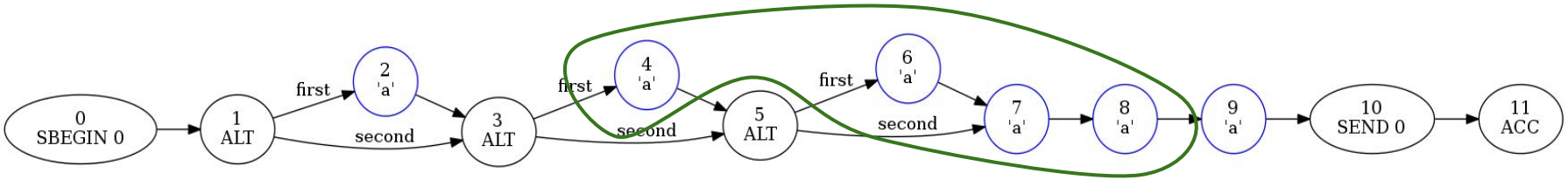


Active = { }

EpsilonClosure({ 2, 4, 6, 7 }).FilterBy('a') = { 4, 6, 7, 8 }

Breadth-First Search (BFS)

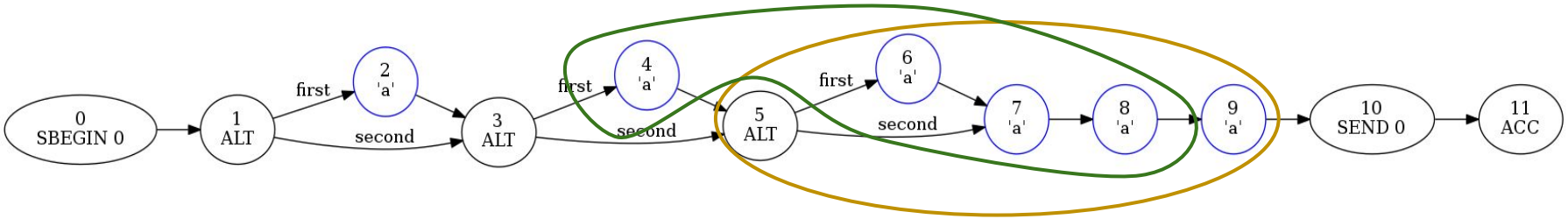
a?a?a?aaa ← aaa



Active = { 4, 6, 7, 8 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

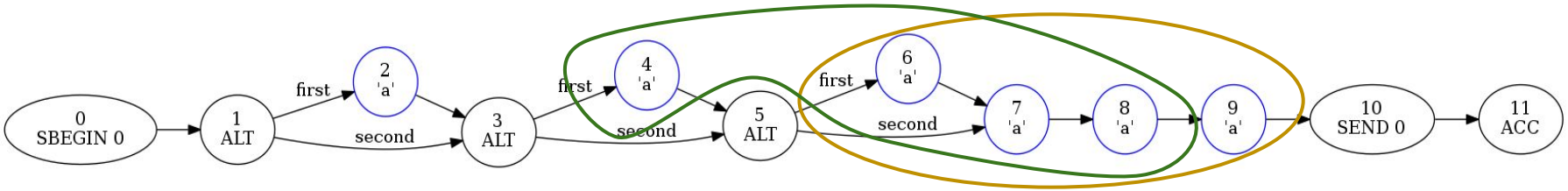


Active = { }

EpsilonClosure({ 4, 6, 7, 8 }) = { 5, 6, 7, 8, 9 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

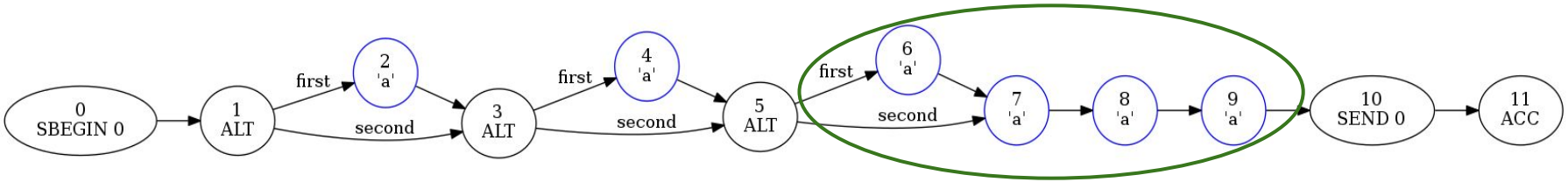


Active = { }

EpsilonClosure({ 4, 6, 7, 8 }).FilterBy('a') = { 6, 7, 8, 9 }

Breadth-First Search (BFS)

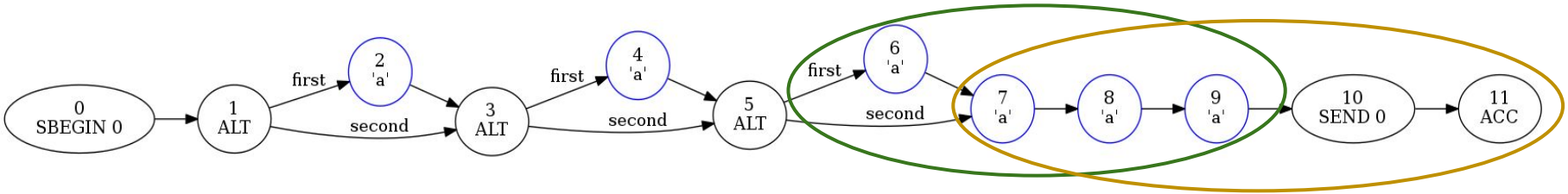
a?a?a?aaa ← aaa



Active = { 6, 7, 8, 9 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa

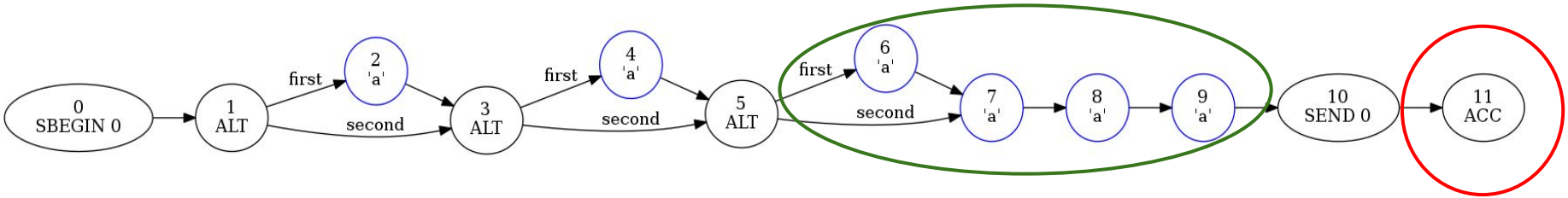


Active = { }

EpsilonClosure({ 6, 7, 8, 9 }) = { 7, 8, 9, 10, 11 }

Breadth-First Search (BFS)

a?a?a?aaa ← aaa



Active = { }

EpsilonClosure({ 6, 7, 8, 9 }).FilterBy(ACC) = { 11 }

Breadth-First Search (BFS)

```
bool Bfs(const State& state, const char* target) {









}
```

Breadth-First Search (BFS)

```
bool Bfs(const State& state, const char* target) {

  set<const State*> active = {&state};









}
```

Breadth-First Search (BFS)

```
bool Bfs(const State& state, const char* target) {
  set<const State*> active = {&state};
  for (; *target; target++)
    active = FilterByMatch(EpsilonClosure(active), *target);


}
```

Breadth-First Search (BFS)

```
bool Bfs(const State& state, const char* target) {
  set<const State*> active = {&state};
  for (; *target; target++)
    active = FilterByMatch(EpsilonClosure(active), *target);
  return !FilterByAcc(EpsilonClosure(active)).empty();
}
```

Breadth-First Search (BFS)

Time Complexity:

O(|target| * |states|)

Space Complexity:

O(|states|)

Breadth-First Search (BFS)

Time Complexity:

O(|target| * |states|) ← Slower than DFS "on average"

Space Complexity:

O(|states|)

Deterministic Finite Automaton (DFA)

Deterministic Finite Automaton (DFA)

Precompute `FilterByMatch(EpsilonClosure(s), c)`

Deterministic Finite Automaton (DFA)

Precompute `FilterByMatch(EpsilonClosure(s), c)`

({ State }, char) → { State }

Deterministic Finite Automaton (DFA)

Precompute `FilterByMatch(EpsilonClosure(s), c)`

({ State }, char) → { State }

↓

(int, char) → int

Deterministic Finite Automaton (DFA)

Precompute `FilterByMatch(EpsilonClosure(s), c)`

({ State }, char) → { State }

↓

(int, char) → int

↓

vector<array<int, 128>>

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {



}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {

  int id = GetOrAllocId(s);









}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {

  int id = GetOrAllocId(s);

  if (id < dfa->size()) return;









}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {
  int id = GetOrAllocId(s);
  if (id < dfa->size()) return; dfa->emplace_back();




}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {
  int id = GetOrAllocId(s);
  if (id < dfa->size()) return; dfa->emplace_back();
  for (unsigned char c = 0; c < 128; c++) {




  }
}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {
  int id = GetOrAllocId(s);
  if (id < dfa->size()) return; dfa->emplace_back();
  for (unsigned char c = 0; c < 128; c++) {
    auto next = FilterByMatch(EpsilonClosure(s), c);


  }
}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {
  int id = GetOrAllocId(s);
  if (id < dfa->size()) return; dfa->emplace_back();
  for (unsigned char c = 0; c < 128; c++) {
    auto next = FilterByMatch(EpsilonClosure(s), c);
    (*dfa)[id][c] = GetOrAllocId(next);
  }
}
```

Deterministic Finite Automaton (DFA)

```
void GenDfa(set<const State*> s, vector<array<int, 128>>* dfa) {
  int id = GetOrAllocId(s);
  if (id < dfa->size()) return; dfa->emplace_back();
  for (unsigned char c = 0; c < 128; c++) {
    auto next = FilterByMatch(EpsilonClosure(s), c);
    (*dfa)[id][c] = GetOrAllocId(next); GenDfa(next, dfa);
  }
}
```

Deterministic Finite Automaton (DFA)

Matching Time Complexity:

$O(|target|)$

Space Complexity:

$O(2^{|states|})$

Deterministic Finite Automaton (DFA)

Matching Time Complexity:

$O(|target|)$

Space Complexity:

$O(2^{|states|})$ ← .*a.{n}

Algorithms × Efficiency

Optimized Slow Vulnerable

	"Average" Time	Worst Time	"Average" Space	Worst Space	Killer input
DFS	|target|	exponential	|target|+|states|	|target|+|states|	a?{32}a{32}
DFS & Memoization	|target|	|target|*|states|	|target|+|states|	|target|*|states|	long texts
BFS	C*|target|	|target|*|states|	|states|	|states|	large regex
DFA	|target|	|target|	|states|	exponential	.*a.{32}

Algorithms × Efficiency

Stack Overflow outage July 20, 2016

^[\s\u200c]+|[\s\u200c]+$ + backtracking regex engine = $O(n^2)$

Algorithms × Efficiency

Stack Overflow outage July 20, 2016

^[\s\u200c]+|[\s\u200c]+$ + backtracking regex engine = $O(n^2)$

↑

DFS

Implementations × Algorithms

	DFS	DFS & Memoization	BFS	DFA
Boost.Regex	✔			
Boost.Xpressive	✔			
libstdc++	✔		✔	
libc++	✔			
MSVC <regex>	✔			
PCRE	✔		✔	
RE2		✔	✔	✔

"Pessimizations"

Capturing Groups

Example: (a*)(b*)(c*) ← aabccc

group[0] = aabccc

group[1] = aa

group[2] = b

group[3] = ccc

Capturing Groups

DFS & Memoization:

(State, String) → bool

BFS:

State → bool

DFA:

({ State }, char) → { State }

Capturing Groups

DFS & Memoization:

(State, String) → bool

[String]

BFS:

State → (bool, [String])

DFA:

N/A

Capturing Groups

DFS & Memoization:

(State, String) → bool

[String]

BFS:

State → (bool, [String])

DFA:

N/A

Solution: limit the number of capturing groups (std::regex::mark_count)

Backreferences

Example: ((a|b)*)\1

Matches: aaabaaab

Does not match: aaababab

Backreferences

DFS & Memoization:

(State, State) → bool

BFS:

State → bool

DFA:

({ State }, char) → { State }

Backreferences

DFS & Memoization:

(State, State, Path) → bool

BFS:

(State, Path) → bool

DFA:

({ (State, Path) }, char) → { (State, Path) }

Backreferences

DFS & Memoization:

(State, State, Path) → bool

BFS:

(State, Path) → bool

DFA:

({ (State, Path) }, char) → { (State, Path) }

Solution: forbid it (libstdc++ std::regex_constants::__polynomial)

Brace Quantifiers

a{3}

↓

aaa

Brace Quantifiers

a{3}{3}

↓

aaa...a for 9 times

Brace Quantifiers

a{1000}{1000}

↓

aaa...a for 1,000,000 times

Brace Quantifiers

a{1000}{1000}

↓

aaa...a for 1,000,000 times

Solution: set state limit (std::regex_constants::error_type::error_space)

Algorithms × Features

	Capturing Groups	Backreferences	Brace Quantifiers
DFS	✔	✔	✔
DFS & Memoization	✔	✗	—
BFS	—	✗	—
DFA	?	✗	—

Future

Compile-time regex

Future:

- `template<const char s[]> static_regex {};`
- `regex_match("aaab", static_regex<"a*b">());`

Compile-time regex

Future:

- `template<const char s[]> static_regex {};` ← language change
- `regex_match("aaab", static_regex<"a*b">());` ← library change

Compile-time regex

Future:

- `template<const char s[]> static_regex {};` ← language change
- `regex_match("aaab", static_regex<"a*b">());` ← library change

Today:

- `template <char...> static_regex {};`
- `my_regex_match("aaa", static_regex<'a', '*', 'b'>());`

Compile-time regex

81-line proof of concept:

For `regex_match(string(1000, 'a'), regex("a*"));`

11x faster than libstdc++ with g++ -O2;

40x faster than libc++ with clang++ -O2;

10x faster than Boost.Regex 1.54

1x compared to Boost.Xpressive 1.54

Compile-time regex

```
template <typename Left, typename Right>
struct MatchImpl<ConcatExpr<Left, Right>> {
  template <typename Continuation>
  static bool Apply(const char* target, Continuation cont) {
    return MatchImpl<Left>::Apply(
      target, [cont](const char* rest) -> bool {
                return MatchImpl<Right>::Apply(rest, cont);
              });
 }
}
```

Algorithms on Non-characters

```
enum Event {

  CASE_OPEN = 256, CASE_CLOSE, SWAP_PART, REBOOT,

};
```

Algorithms on Non-characters

```
enum Event {
  CASE_OPEN = 256, CASE_CLOSE, SWAP_PART, REBOOT,
};
basic_regex<Event> re {
    CASE_OPEN, ".*", REBOOT, "{3}", ".*", CASE_CLOSE };
```

Recap

- 4 algorithms
  - DFS
  - DFS & Memoization
  - BFS
  - DFA
- 3 "pessimizations"
  - capturing groups
  - backreferences
  - brace quantifiers
- 2 changes to the standard
  - `template <const char s[]>`
  - `basic_regex<NonCharacter>`

Thank you :)