String matching using automata

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 10

Traditional string matching

- Text `t`, pattern `p` of of lengths n, m
- For each starting position `i` in `t`, compare `t[i:i+m]` with `p`
  - Scan `t[i:i+m]` right to left

Traditional string matching

- Text `t`, pattern `p` of of lengths n, m
- For each starting position `i` in `t`, compare `t[i:i+m]` with `p`
  - Scan `t[i:i+m]` right to left
- Can skip some positions `i` on mismatch [Boyer,Mooore]
  - Case 1: `t[i+j]` does not appear in `p`
    - Update `i` to `i+j+1`
  - Case 2: `t[i+j]` appears in `p` before `p[j]`
    - Precompute `last[c]` for each `c` in `p`
    - Align `p[last[c]]` with `t[i+j]`
    - Update `i` to `j - last[c]`

Traditional string matching

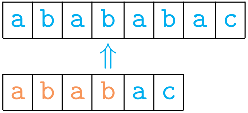
- Text `t`, pattern `p` of of lengths n, m
- For each starting position `i` in `t`, compare `t[i:i+m]` with `p`
  - Scan `t[i:i+m]` right to left
- Can skip some positions `i` on mismatch [Boyer,Mooore]
  - Case 1: `t[i+j]` does not appear in `p`
    - Update `i` to `i+j+1`
  - Case 2: `t[i+j]` appears in `p` before `p[j]`
    - Precompute `last[c]` for each `c` in `p`
    - Align `p[last[c]]` with `t[i+j]`
    - Update `i` to `j - last[c]`
- Worst case remains $O(nm)$: `t = aaa...a`, `p = baaaa`

Remembering previous matches

- Can we intelligently re-use partial matches

Remembering previous matches

- Can we intelligently re-use partial matches



- Keep track of longest prefix of `p[:k]` that we have matched

Remembering previous matches

- Can we intelligently re-use partial matches

a	b	a	b	a	b	a	c

$\Uparrow$

a	b	a	b	a	c

- Keep track of longest prefix of `p[:k]` that we have matched
- This prefix is a suffix of `t[:i+j]`

Remembering previous matches

- Can we intelligently re-use partial matches

a	b	a	b	a	b	a	c

a	b	a	b	a	c

- Keep track of longest prefix of `p[:k]` that we have matched
- This prefix is a suffix of `t[:i+j]`
- If `t[i+j] == p[k]`, extend the match

Remembering previous matches

- Can we intelligently re-use partial matches

a	b	a	b	a	b	a	c

$$\Uparrow$$

a	b	a	b	a	c

- Keep track of longest prefix of `p[:k]` that we have matched
- This prefix is a suffix of `t[:i+j]`
- If `t[i+j] == p[k]`, extend the match
- If `t[i+j] != p[k]`, reset longest match for `t[:i+j+1]`

Remembering previous matches

- Can we intelligently re-use partial matches

a	b	a	b	a	b	a	c

$\Uparrow$

a	b	a	b	a	c

- Keep track of longest prefix of `p[:k]` that we have matched
- This prefix is a suffix of `t[:i+j]`
- If `t[i+j] == p[k]`, extend the match
- If `t[i+j] != p[k]`, reset longest match for `t[:i+j+1]`
- To reset, use precomputed values

Precomputing longest match as a graph

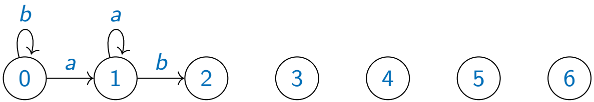
- Pattern `p` of length `m`

Precomputing longest match as a graph

- Pattern `p` of length `m`
- Graph with `m+1` nodes, $\{0, 1, \ldots, m\}$
  - Node i denotes match upto `p[:i]`

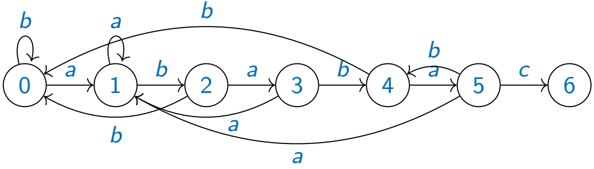
Precomputing longest match as a graph

- Pattern `p` of length `m`
- Graph with `m+1` nodes, $\{0, 1, \ldots, m\}$
  - Node i denotes match upto `p[:i]`
- Edges describe how to extend the match
- `t[j] = a`, `t[:j]` matches `p[:i]`
  - Edge $i \xrightarrow{a} k$, `t[:j+1]` matches `p[:k]`
    - If `t[j] = p[i]`, $k = i + 1$
    - Else find longest prefix of `p` that matches suffix of `t[:j+1]`



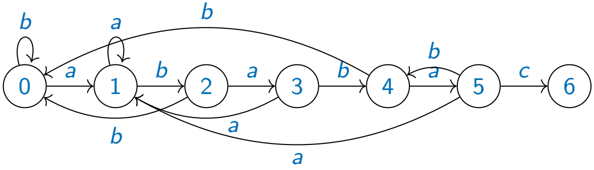
Precomputing longest match as a graph

- Pattern `p` of length `m`
- Graph with `m+1` nodes, $\{0, 1, \ldots, m\}$
  - Node i denotes match upto `p[:i]`
- Edges describe how to extend the match
- `t[j] = a`, `t[:j]` matches `p[:i]`
  - Edge $i \xrightarrow{a} k$, `t[:j+1]` matches `p[:k]`
    - If `t[j] = p[i]`, $k = i + 1$
    - Else find longest prefix of `p` that matches suffix of `t[:j+1]`



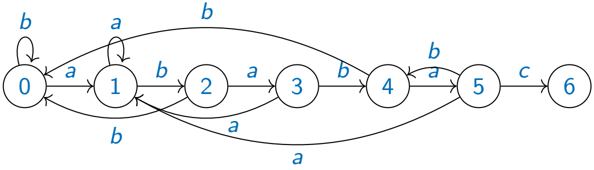
Precomputing longest match as a graph

- Pattern p of length m
- Graph with m+1 nodes, $\{0, 1, \dots, m\}$
  - Node i denotes match upto p[:i]
- Edges describe how to extend the match
- t[j] = a, t[:j] matches p[:i]
  - Edge $i \xrightarrow{a} k$, t[:j+1] matches p[:k]
    - If t[j] = p[i], $k = i + 1$
    - Else find longest prefix of p that matches suffix of t[:j+1]
- Brute force, $O(m^2)$ per edge



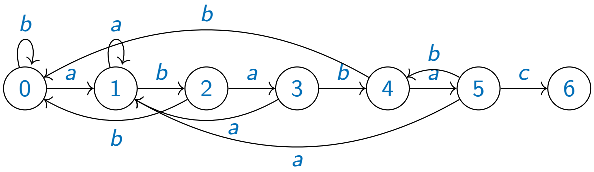
Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions



Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0

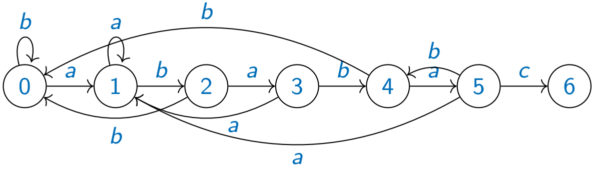


Processing *abababac*

0

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

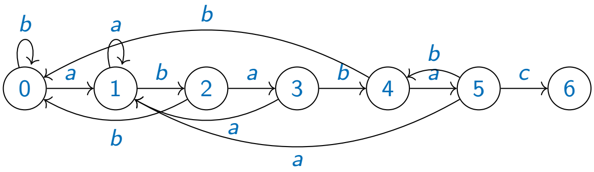


Processing *abababac*

$0 \xrightarrow{a} 1$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

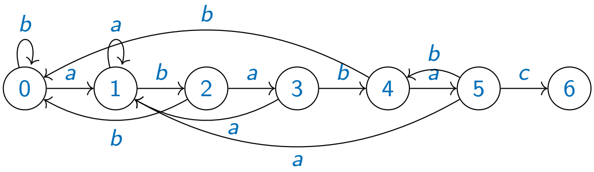


Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

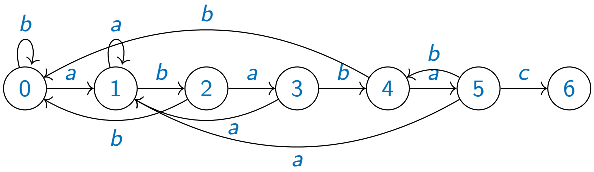


Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

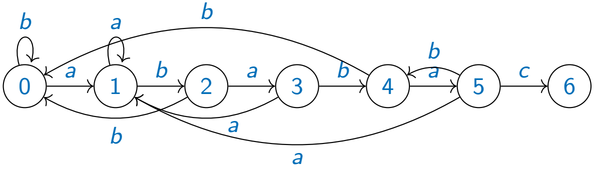


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

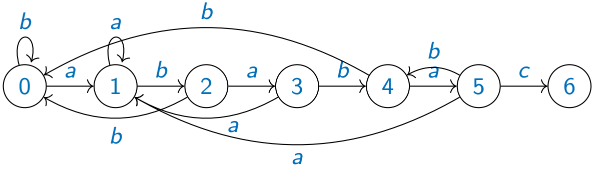


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

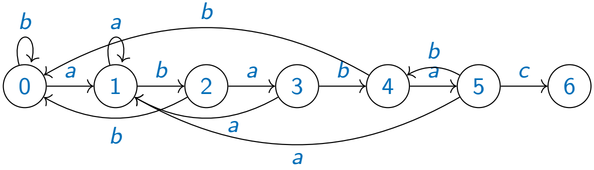


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix

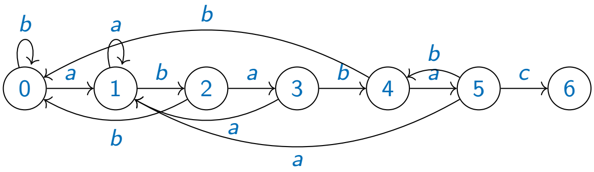


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix
- If we reach the final state m, we have found a full match for `p`

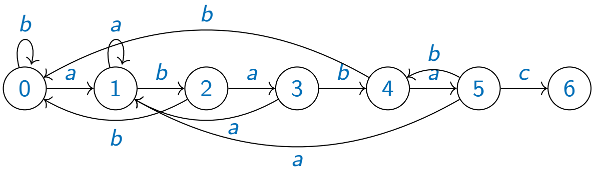


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
  - Nodes are states
  - Edges are tranitions
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- Updates the longest matching prefix
- If we reach the final state m, we have found a full match for `p`
- Single scan of `t` suffices



Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$$

Summary

- Build an automaton to keep track of longest matching prefix
- Using this automaton, we can do string matching in $O(n)$
  - The algorithm we described finds only the first match
  - Restart at the next position to find subsequent matches

Summary

- Build an automaton to keep track of longest matching prefix
- Using this automaton, we can do string matching in $O(n)$
  - The algorithm we described finds only the first match
  - Restart at the next position to find subsequent matches
- Bottleneck is precomputing the automaton
  - Computing each edge $i \xrightarrow{a} j$ took $O(m^2)$
  - Do this for each $i \in \{0, 1, \ldots, m\}$ and each $a \in \Sigma$
  - Overall $O(m^3 \cdot |\Sigma|)$

Summary

- Build an automaton to keep track of longest matching prefix
- Using this automaton, we can do string matching in $O(n)$
  - The algorithm we described finds only the first match
  - Restart at the next position to find subsequent matches
- Bottleneck is precomputing the automaton
  - Computing each edge $i \xrightarrow{a} j$ took $O(m^2)$
  - Do this for each $i \in \{0, 1, \ldots, m\}$ and each $a \in \Sigma$
  - Overall $O(m^3 \cdot |\Sigma|)$
- Do this in time $O(m)$ — [Knuth, Morris, Pratt]