Searching for Multiple Patterns

02-714

Slides by Carl Kingsford

Exact Set Matching Problem

Problem. *Given a set of patterns $P = \{P_1,\ldots,P_z\}$, and a text T, find all exact occurrences of every P_i in T.*

- Easy to solve in $\sum_i(|P_i| + |T|) = O(n + zm)$ where $n = \sum_i|P_i|$ and $m = |T|$.

- Can be solved in time $O(n + m + k)$ in several different ways. E.g.:

  Aho-Corasick: based on keyword trees

  Using suffix trees directly

- Can be solved quickly in practice using Wu-Mandber (a hash-based method).

Aho-Corasick

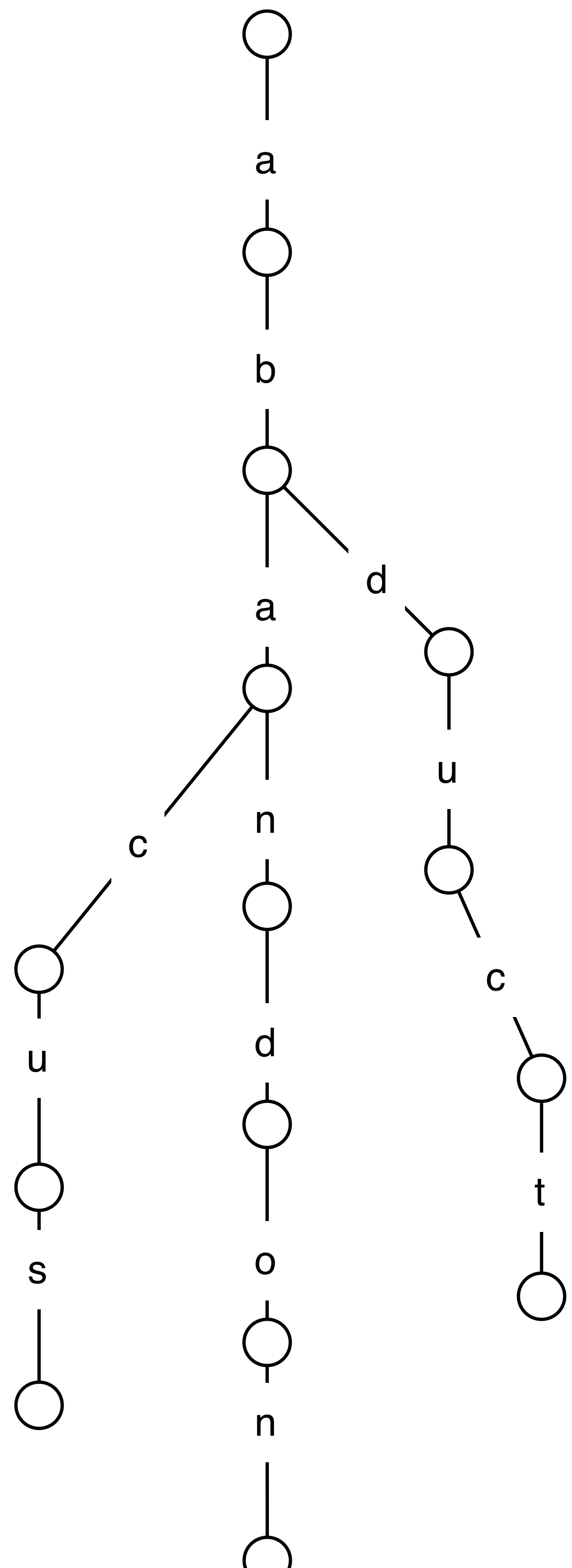
A prefix approach
(following Gusfield)

Keyword Tree

Def. *A keyword tree K(P) of a set of patterns P is a tree where:*

1. *each edge is labeled with a letter*
2. *edges leading from u to its children all have different labels*
3. *there is a function n(i) that gives the node such that pattern i is spelled out on the unique path from root to n(i).*

P = {abandon, abduct, abacus}



Aho-Corasick Failure Function

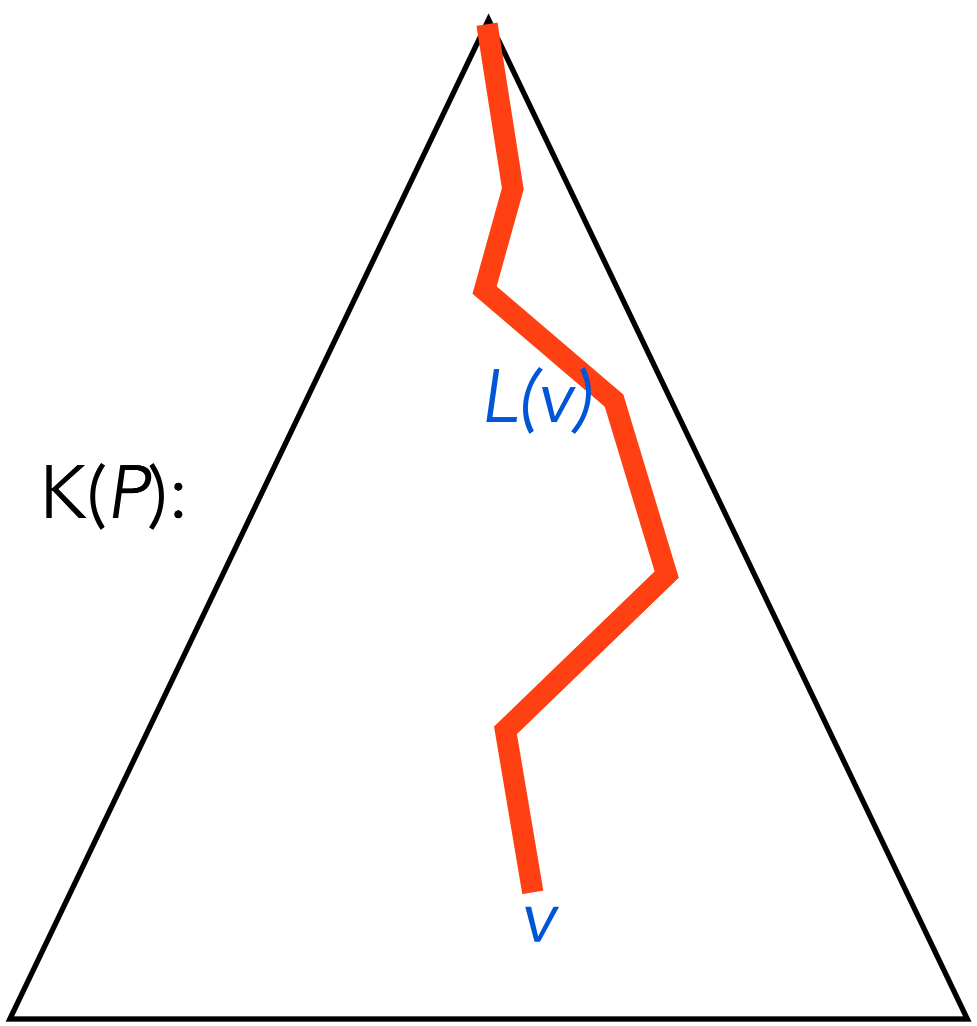
Notation.

L(v) := the string spelled out by the path from the root to node v.

lp(v) := the longest proper suffix of L(v) that is also a prefix of some pattern in P.

f(v) := the node representing string lp(v) in K(P).

K(*P*):



Thm. f(v) always exists and is unique for any node v in K(P).

Proof: lp(v) is a prefix of a pattern, and every pattern is represented by a unique path in K(P) on which every prefix is spelled out.

Aho-Corasick Failure Function

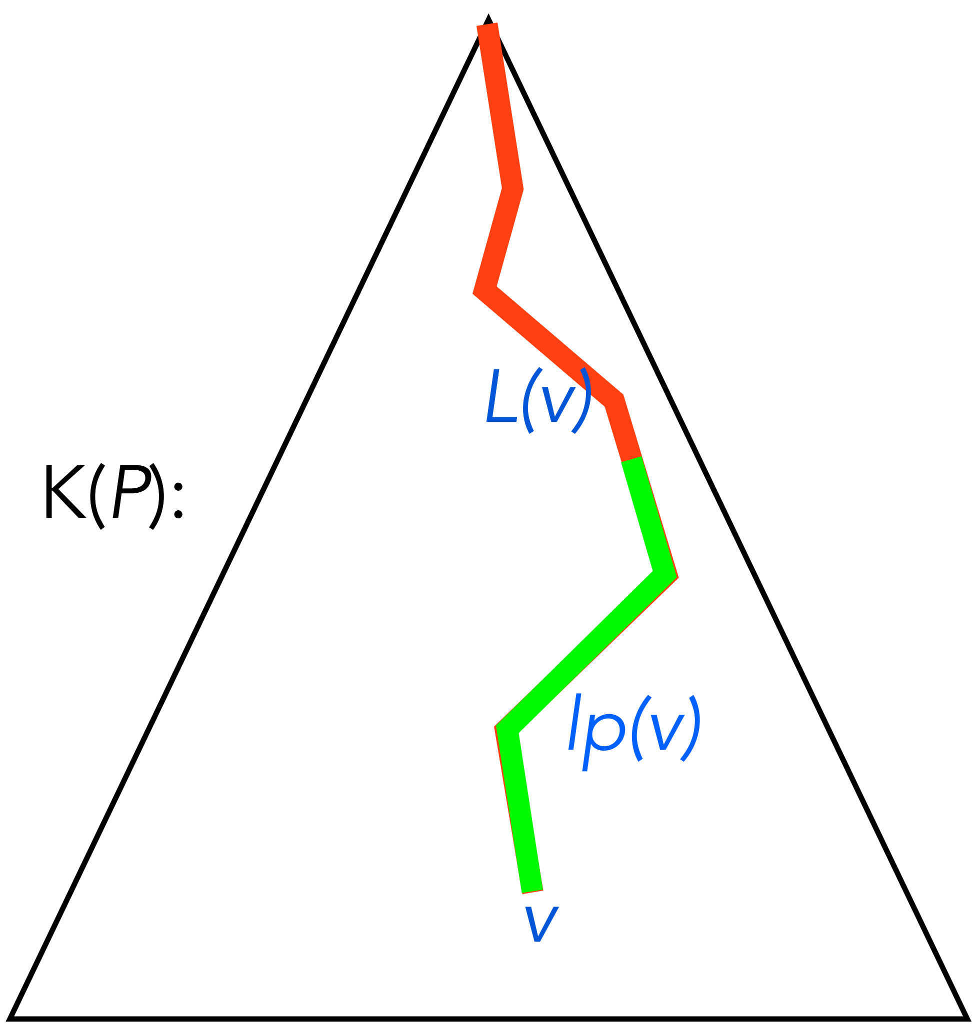
Notation.

L(v) := the string spelled out by the path from the root to node v.

lp(v) := the longest proper suffix of L(v) that is also a prefix of some pattern in P.

f(v) := the node representing string lp(v) in K(P).

K(*P*):



Thm. f(v) always exists and is unique for any node v in K(P).

Proof: lp(v) is a prefix of a pattern, and every pattern is represented by a unique path in K(P) on which every prefix is spelled out.

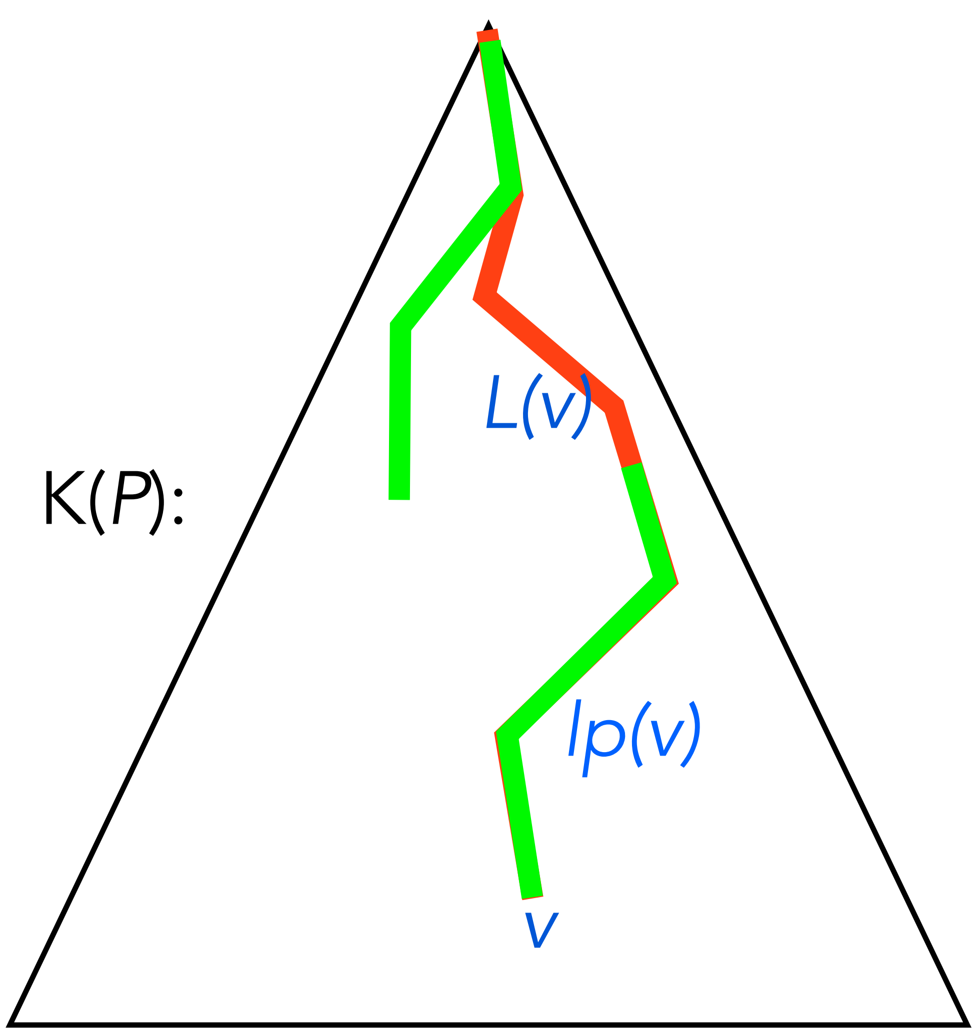
Aho-Corasick Failure Function

Notation.

L(v) := the string spelled out by the path from the root to node v.

lp(v) := the longest proper suffix of L(v) that is also a prefix of some pattern in P.

f(v) := the node representing string lp(v) in K(P).



Thm. f(v) always exists and is unique for any node v in K(P).

Proof: lp(v) is a prefix of a pattern, and every pattern is represented by a unique path in K(P) on which every prefix is spelled out.

Aho-Corasick Failure Function

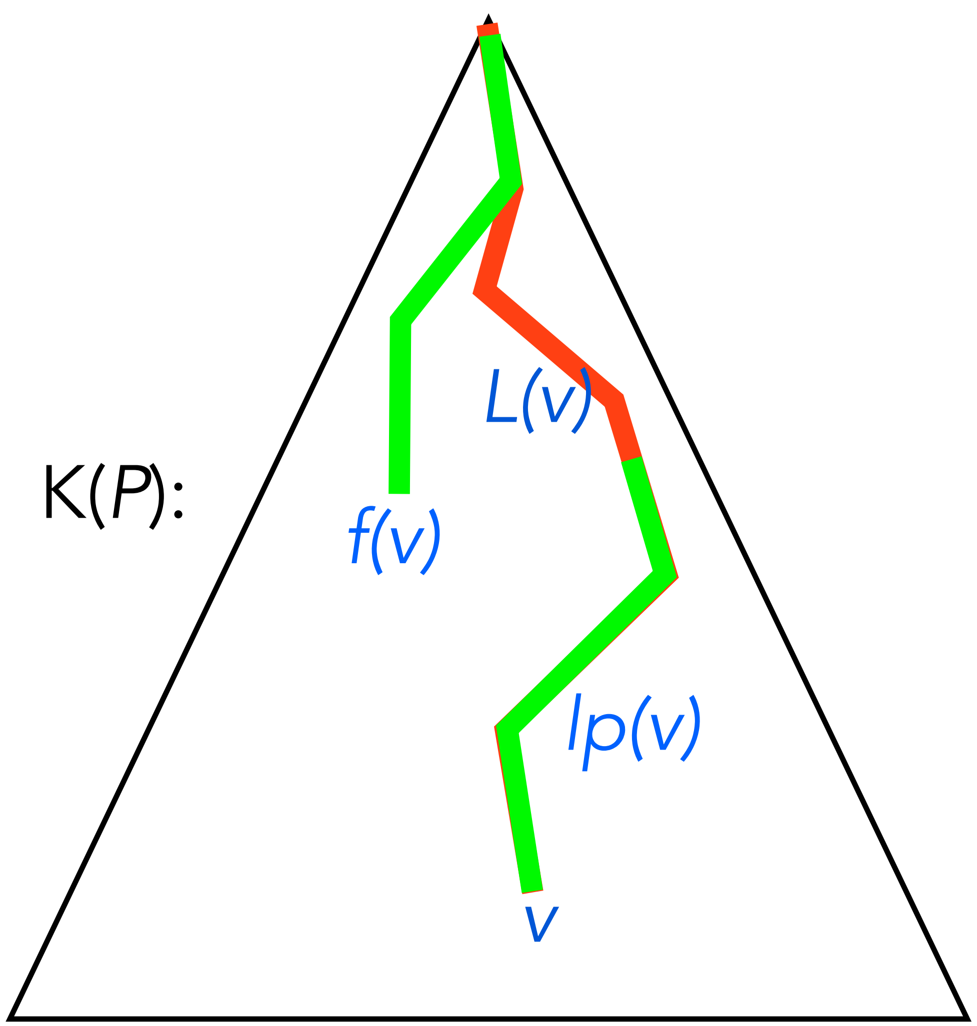
Notation.

L(v) := the string spelled out by the path from the root to node v.

lp(v) := the longest proper suffix of L(v) that is also a prefix of some pattern in P.

f(v) := the node representing string lp(v) in K(P).

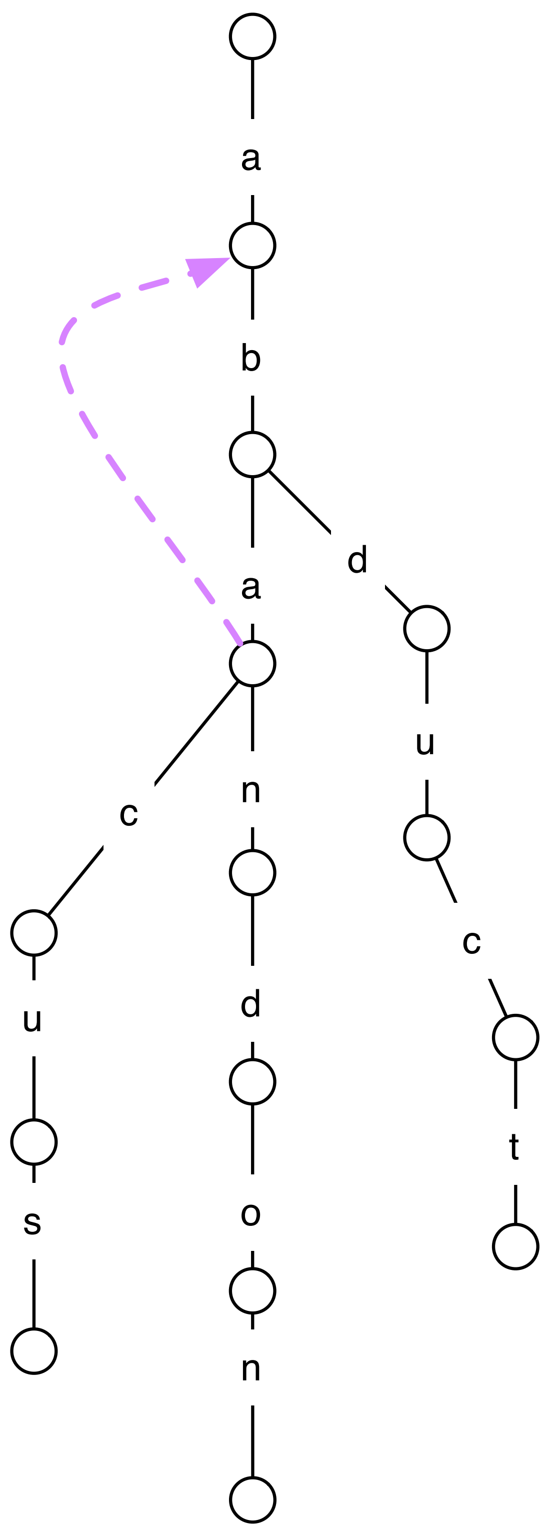
K(*P*):

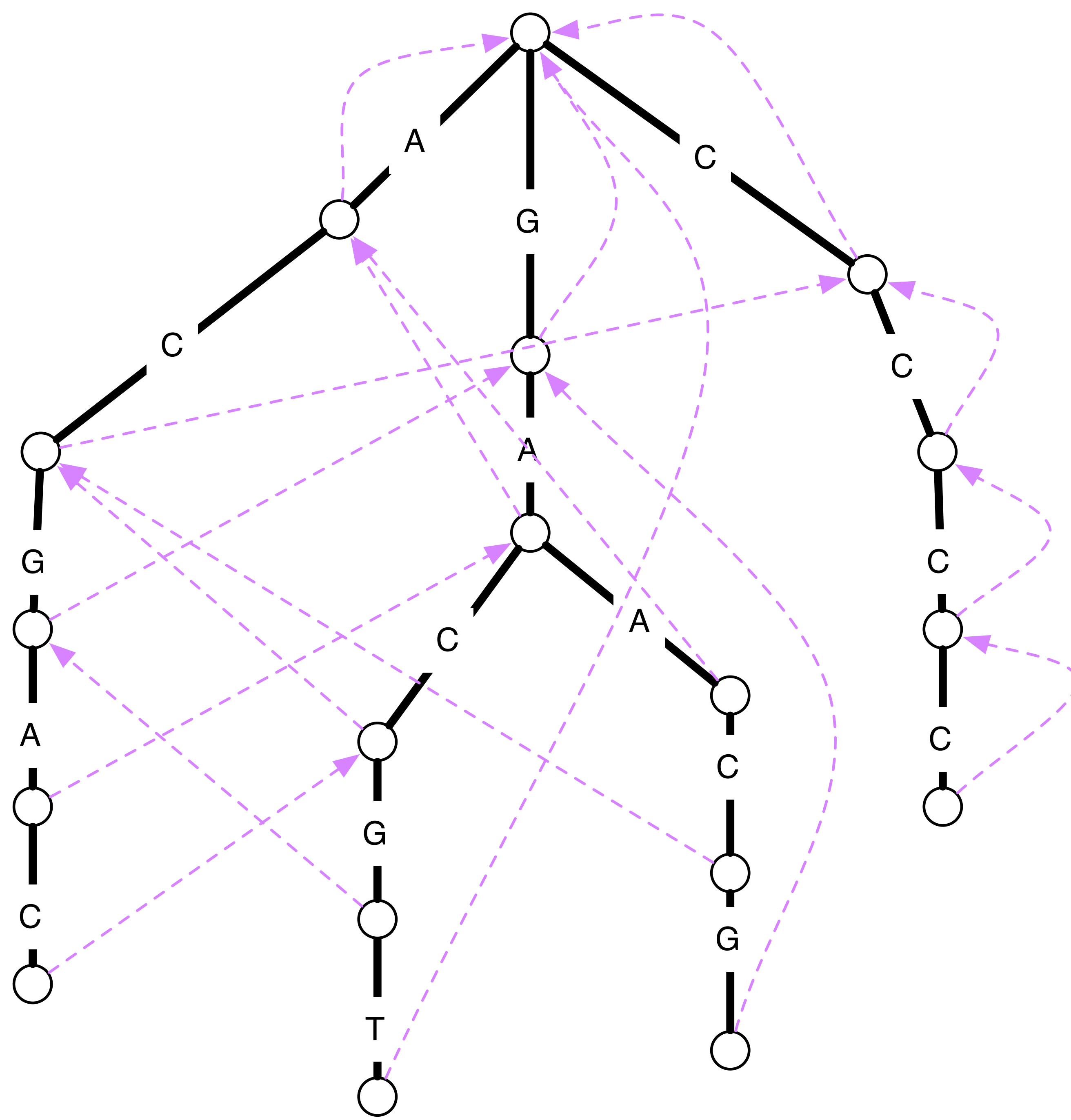


Thm. f(v) always exists and is unique for any node v in K(P).

Proof: lp(v) is a prefix of a pattern, and every pattern is represented by a unique path in K(P) on which every prefix is spelled out.

Example K(P) with Failure Functions

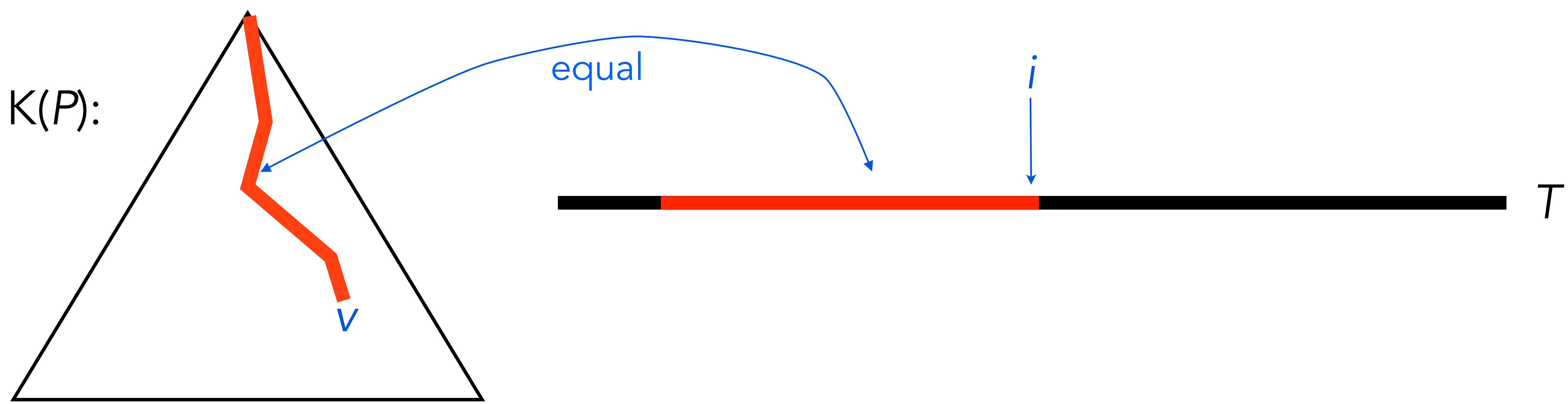




P = {ACGAC, GACGT, GAACG, CCCC}

Aho-Corasick Search

Walk down string and tree at same time, matching characters:

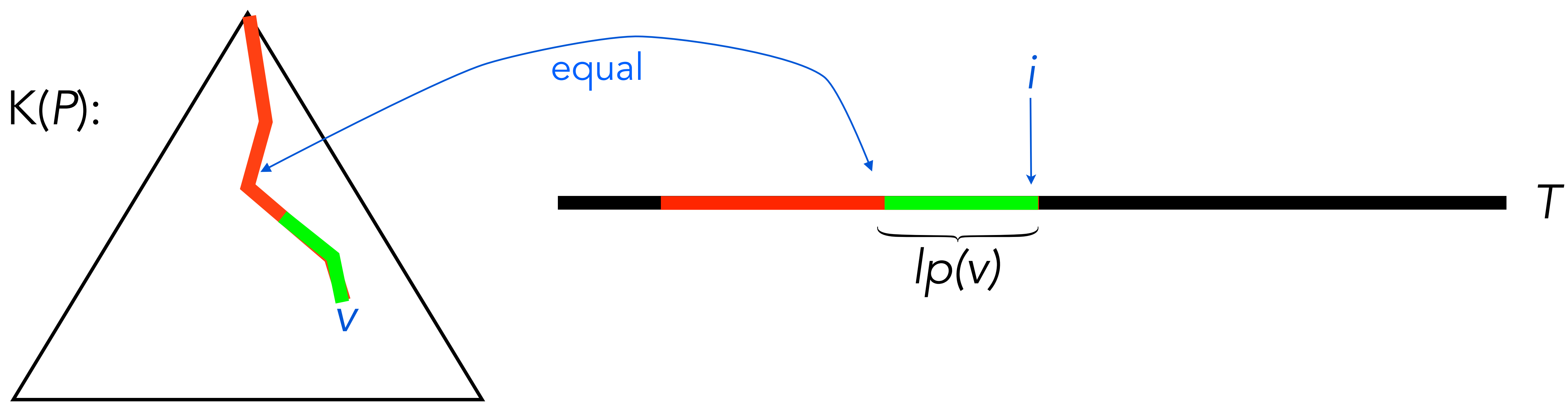


If you get to a node that represents a full pattern, report an occurrence.

If you get stuck at node v, jump to node $f(v)$

Aho-Corasick Search

Walk down string and tree at same time, matching characters:

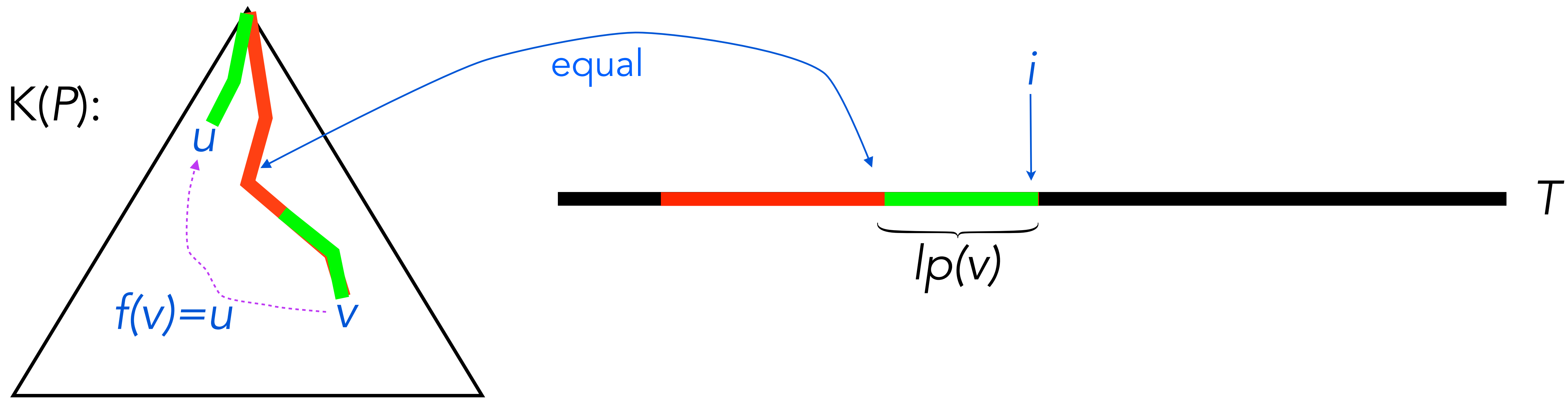


If you get to a node that represents a full pattern, report an occurrence.

If you get stuck at node *v*, jump to node *f*(*v*)

Aho-Corasick Search

Walk down string and tree at same time, matching characters:

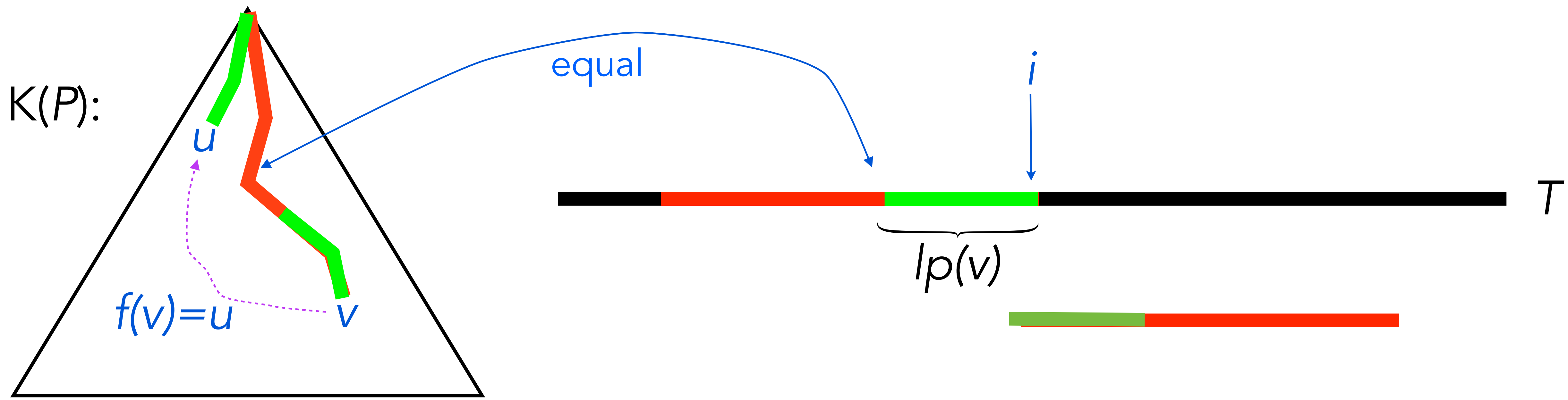


If you get to a node that represents a full pattern, report an occurrence.

If you get stuck at node *v*, jump to node *f*(*v*)

Aho-Corasick Search

Walk down string and tree at same time, matching characters:



If you get to a node that <u>represents a full pattern</u>, report an occurrence.

If you get stuck at node v, jump to node $f(v)$

Running Time

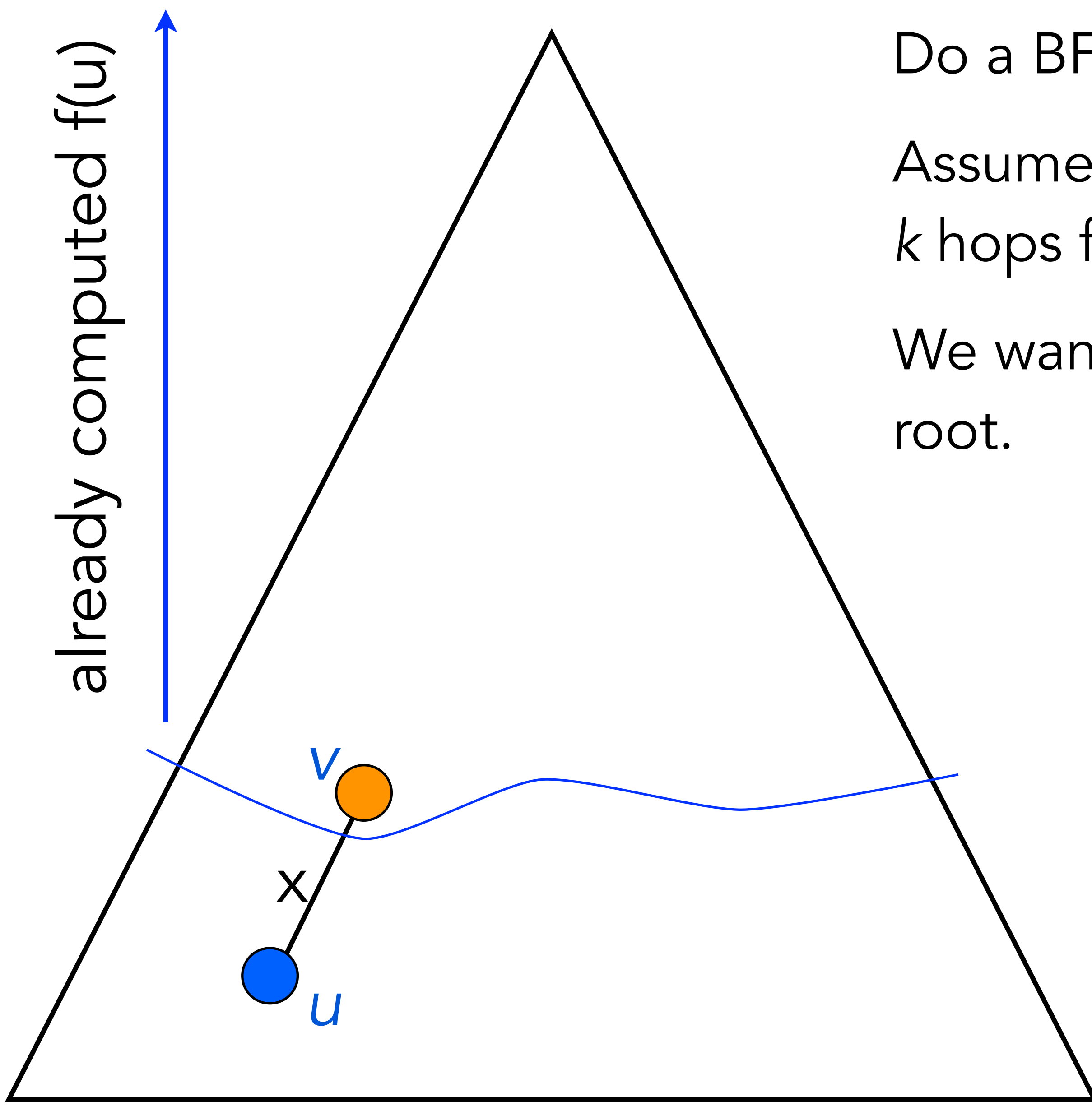
Nearly identical analysis to KMP:

Index i into T is never decremented. Every character can be matched at most once.

Every mismatch results in a "shift" of the pattern of size at ≤ the number of current matched characters: can have at most O(|T|) total mismatches.

$$\Rightarrow O(\underbrace{\text{total length of patterns}}_{\text{build the keyword tree}} + \overset{\text{search T}}{\overset{\downarrow}{|T|}} + \underbrace{\#\text{ of positions output}}_{\text{output the positions}})$$

Computing f(u)



Do a BFS of K(P).

Assume we've computed *f*(*v*) for all u at fewer than *k* hops from the root.

We want to compute *f*(*u*) for *u* at *k*+1 hops from the root.

Let v be the parent of u and x be the character on the (v,u) edge.

We know f(v).

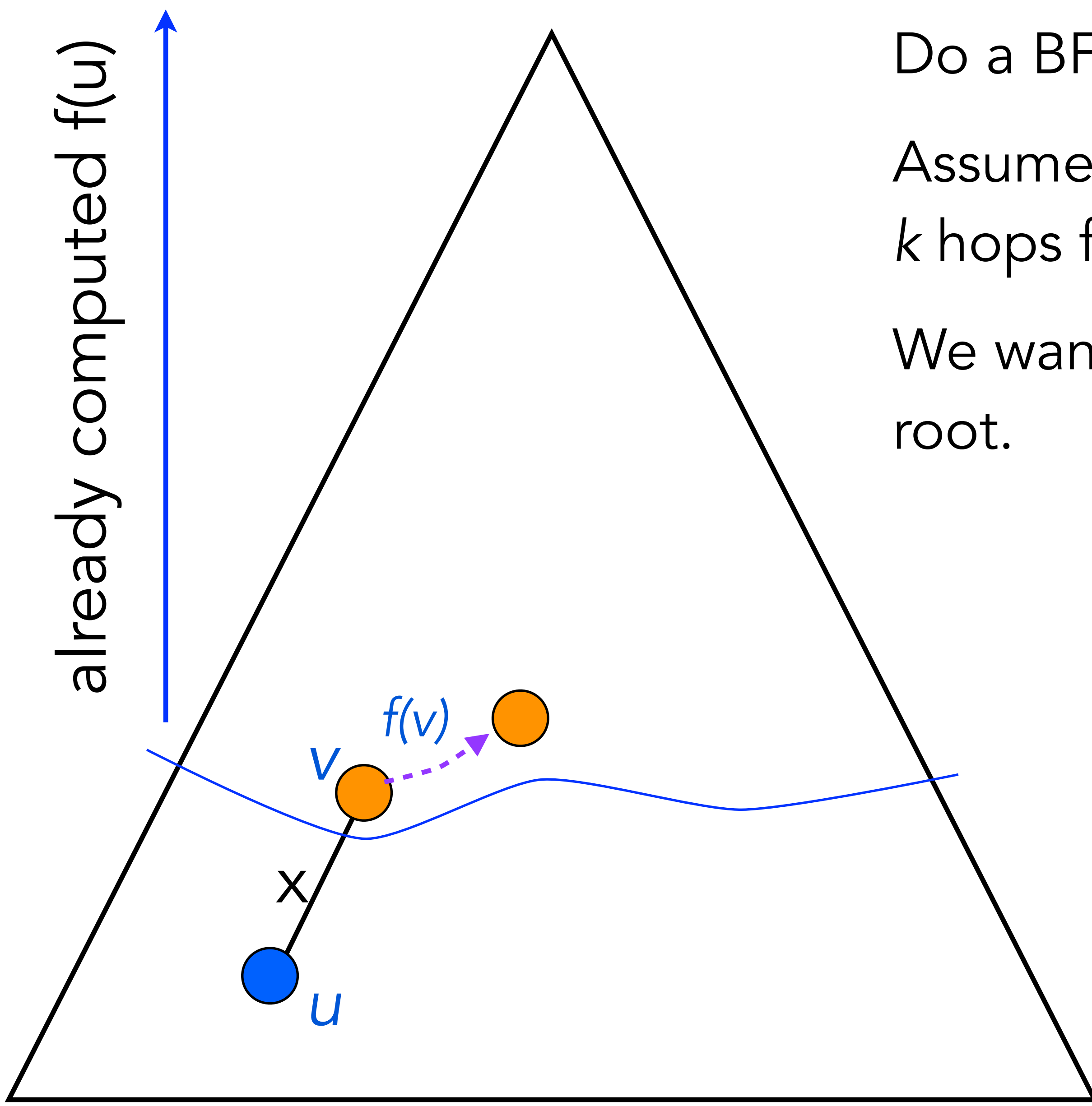
Traverse the chain of f(v), f(f(v)), f(f(f(v)), etc. until you find a node with a child edge labeled x.

Set f(u) equal to that node.

Idea: f(v) is the longest suffix of L(v) that matches a prefix of a pattern, f(f(v)) is the longest suffix of L(f(v)) that matches a prefix, and so on.

We want the longest (first encountered) one of those suffixes that can be extended with x.

Computing f(u)



Do a BFS of K(P).

Assume we've computed *f*(*v*) for all u at fewer than *k* hops from the root.

We want to compute *f*(*u*) for *u* at *k*+1 hops from the root.

Let v be the parent of u and x be the character on the (v,u) edge.

We know f(v).

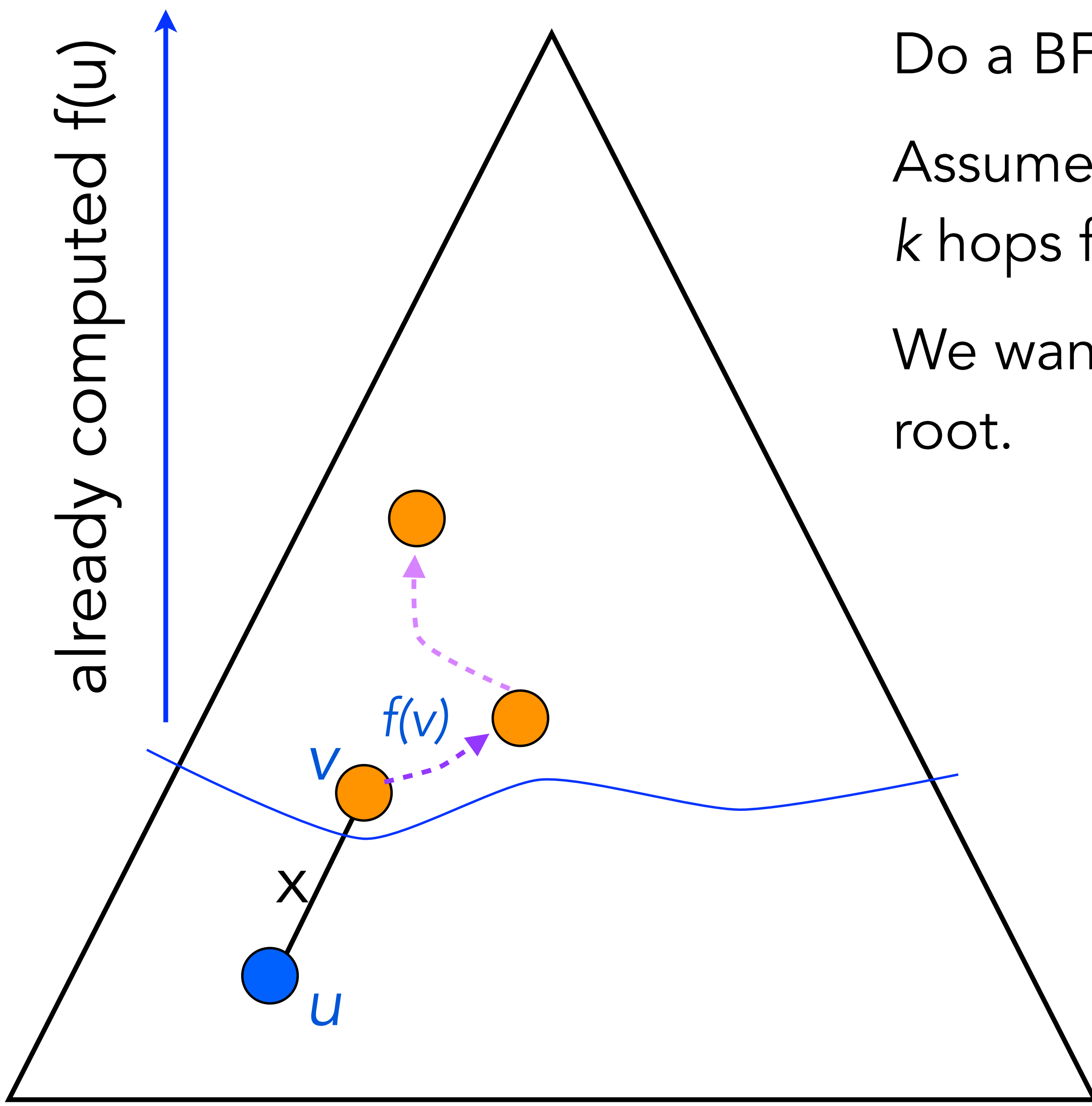
Traverse the chain of f(v), f(f(v)), f(f(f(v)), etc. until you find a node with a child edge labeled x.

Set f(u) equal to that node.

Idea: f(v) is the longest suffix of L(v) that matches a prefix of a pattern, f(f(v)) is the longest suffix of L(f(v)) that matches a prefix, and so on.

We want the longest (first encountered) one of those suffixes that can be extended with x.

Computing f(u)



Do a BFS of K(P).

Assume we've computed *f*(*v*) for all u at fewer than *k* hops from the root.

We want to compute *f*(*u*) for *u* at *k*+1 hops from the root.

Let v be the parent of u and x be the character on the (v,u) edge.

We know f(v).

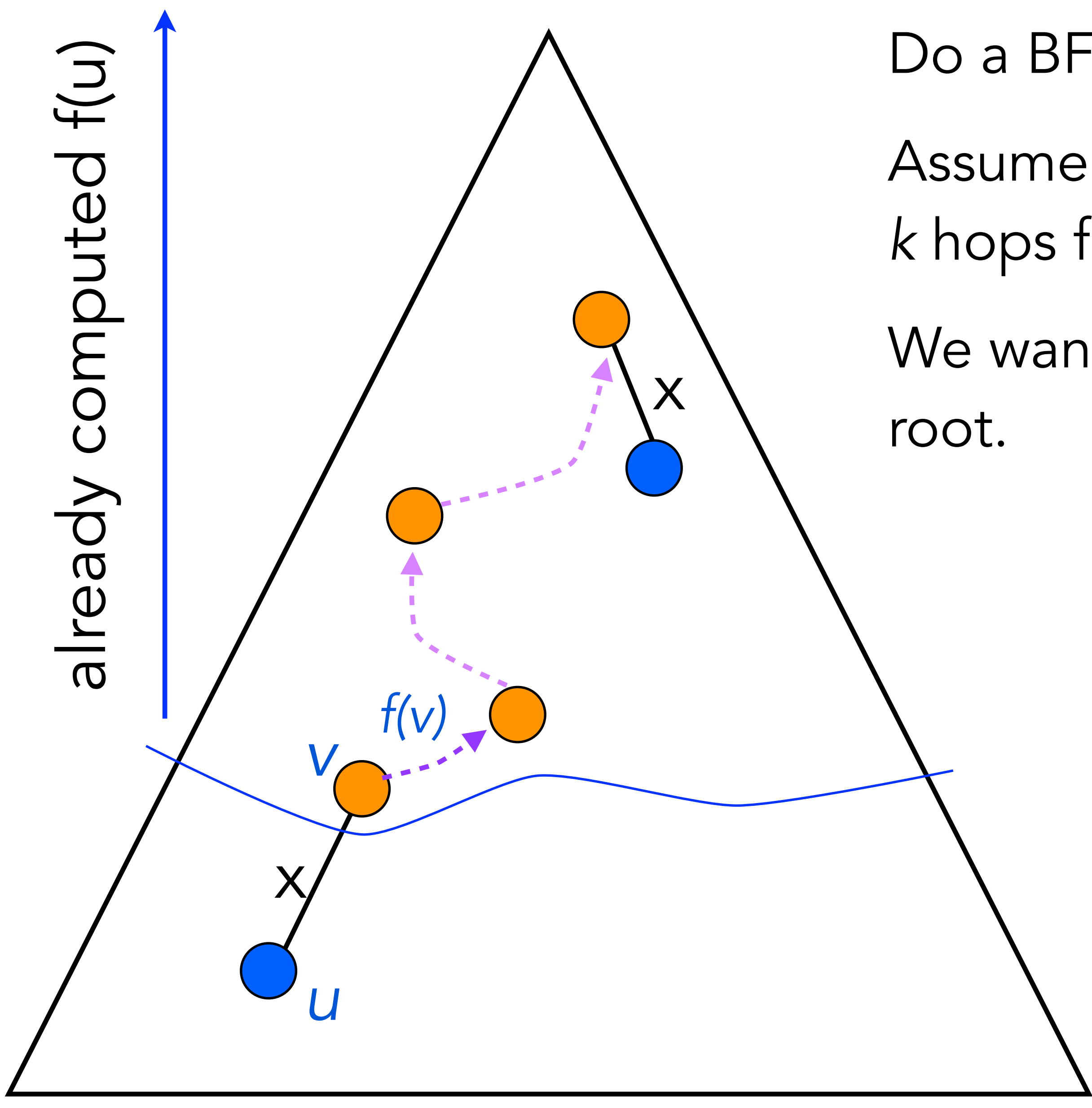
Traverse the chain of f(v), f(f(v)), f(f(f(v)), etc. until you find a node with a child edge labeled x.

Set f(u) equal to that node.

Idea: f(v) is the longest suffix of L(v) that matches a prefix of a pattern, f(f(v)) is the longest suffix of L(f(v)) that matches a prefix, and so on.

We want the longest (first encountered) one of those suffixes that can be extended with x.

Computing f(u)



Do a BFS of K(P).

Assume we've computed *f*(*v*) for all u at fewer than *k* hops from the root.

We want to compute *f*(*u*) for *u* at *k*+1 hops from the root.

Let v be the parent of u and x be the character on the (v,u) edge.

We know f(v).

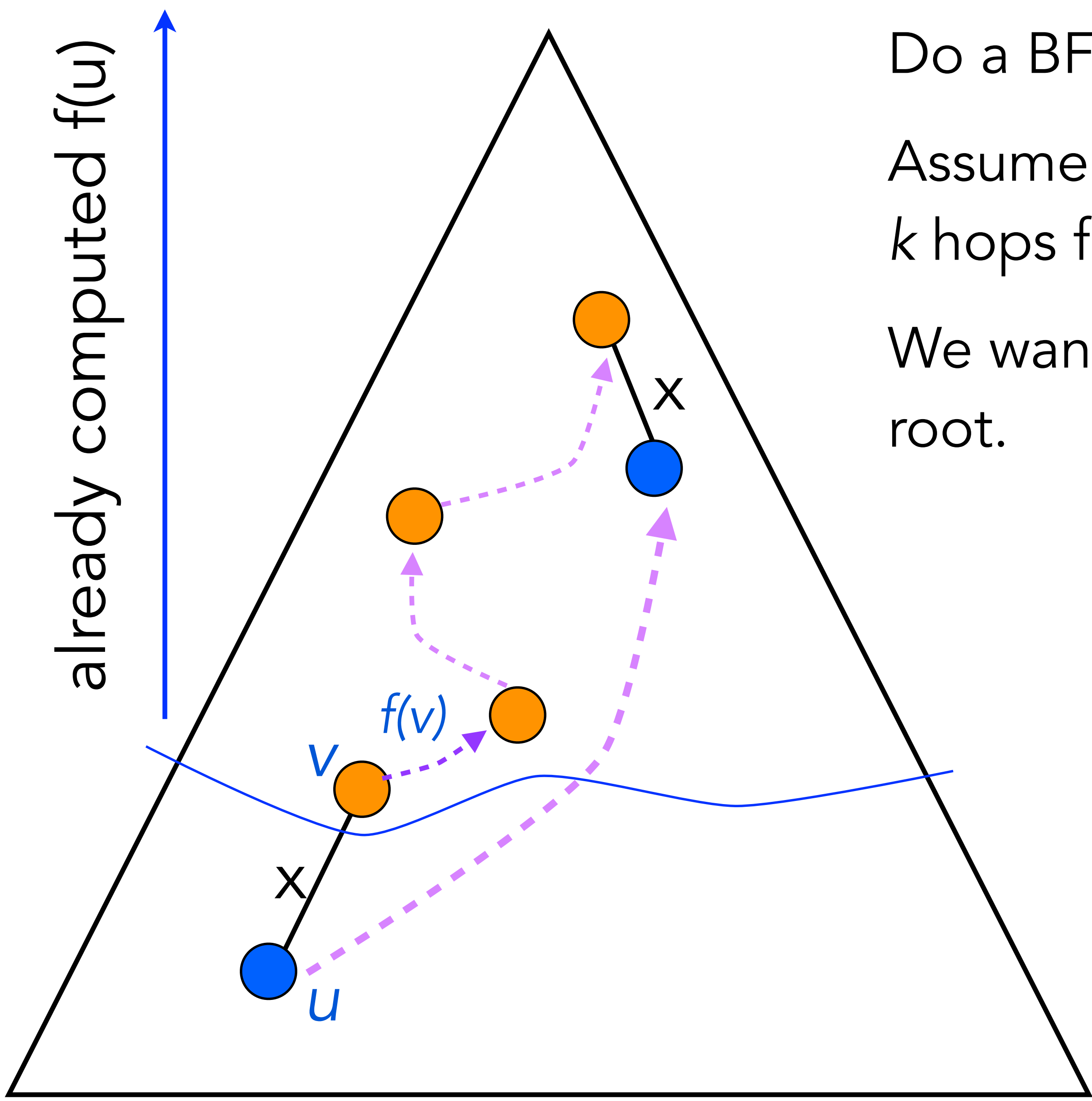
Traverse the chain of f(v), f(f(v)), f(f(f(v)), etc. until you find a node with a child edge labeled x.

Set f(u) equal to that node.

Idea: f(v) is the longest suffix of L(v) that matches a prefix of a pattern, f(f(v)) is the longest suffix of L(f(v)) that matches a prefix, and so on.

We want the longest (first encountered) one of those suffixes that can be extended with x.

Computing f(u)



Do a BFS of K(P).

Assume we've computed *f*(*v*) for all u at fewer than *k* hops from the root.

We want to compute *f*(*u*) for *u* at *k*+1 hops from the root.

Let v be the parent of u and x be the character on the (v,u) edge.

We know f(v).

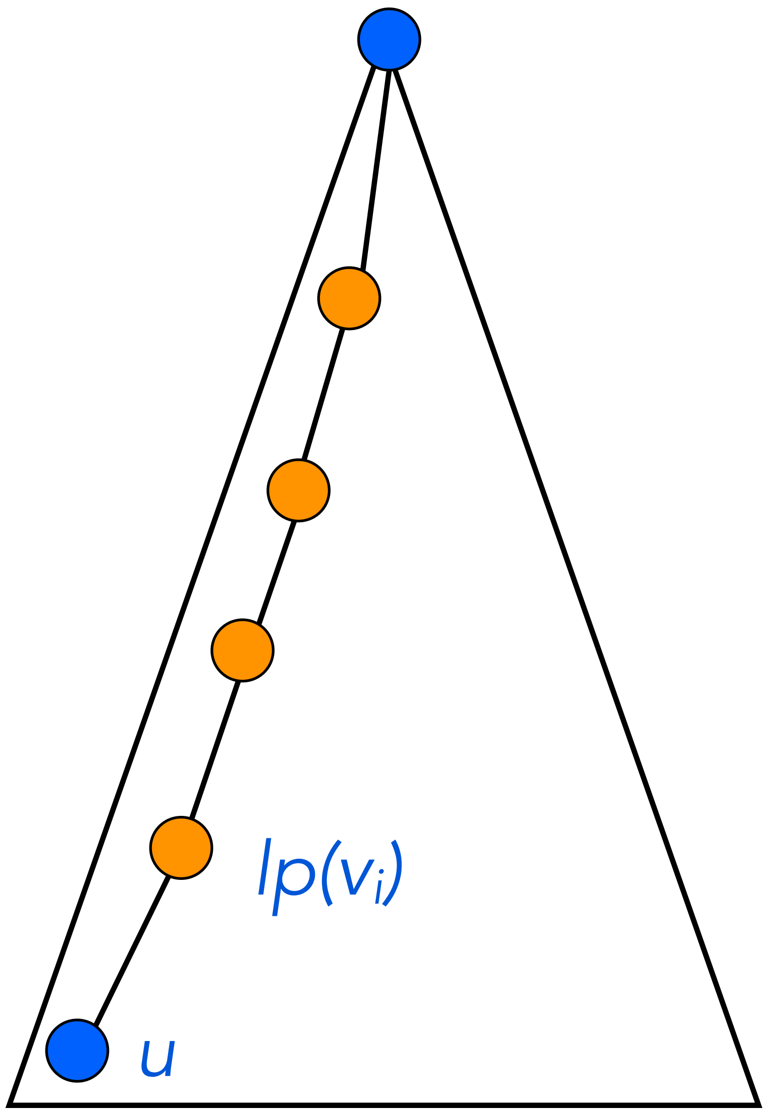
Traverse the chain of f(v), f(f(v)), f(f(f(v)), etc. until you find a node with a child edge labeled x.

Set f(u) equal to that node.

Idea: f(v) is the longest suffix of L(v) that matches a prefix of a pattern, f(f(v)) is the longest suffix of L(f(v)) that matches a prefix, and so on.

We want the longest (first encountered) one of those suffixes that can be extended with x.

Running time of computing the f(u)



Consider path $v_1,\ldots,v_k$ from root to *u*.

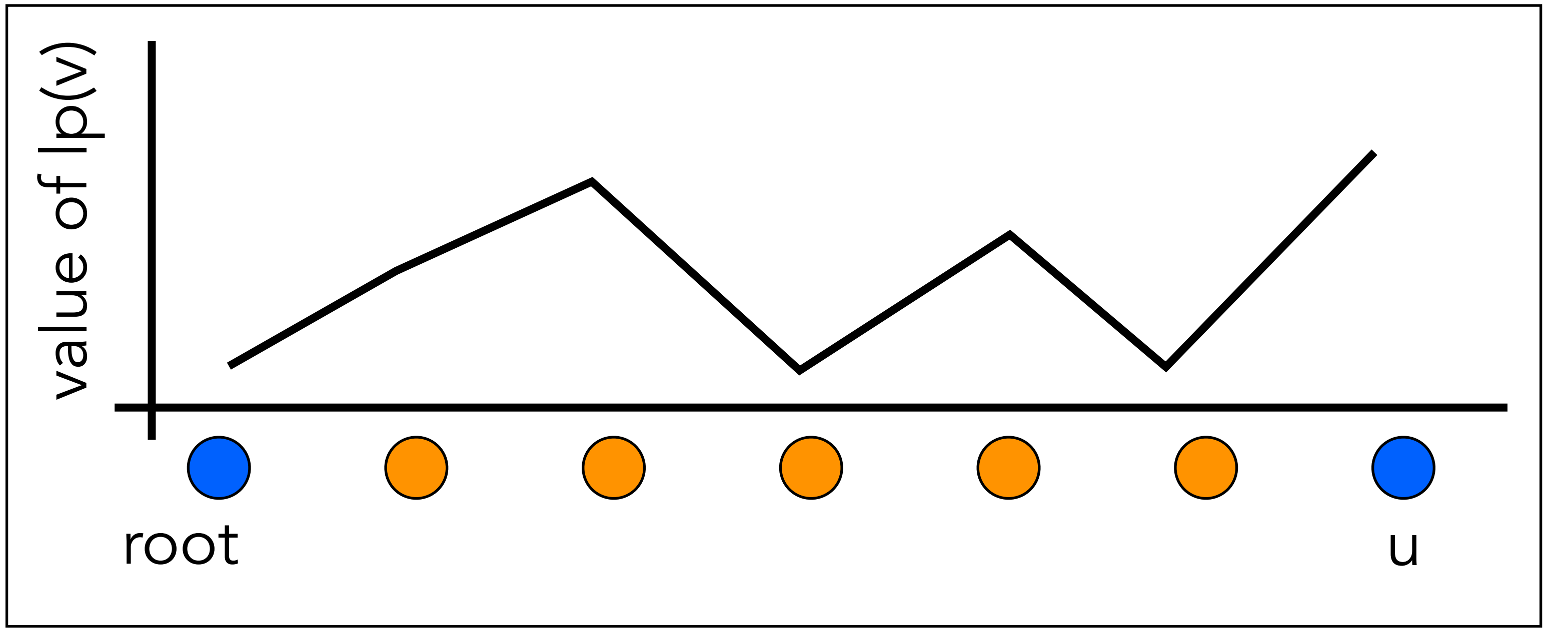
lp increases by at most 1 when we go from v_i to v_{i+1}

lp decreases by at least 1 when we follow an f(v) link.

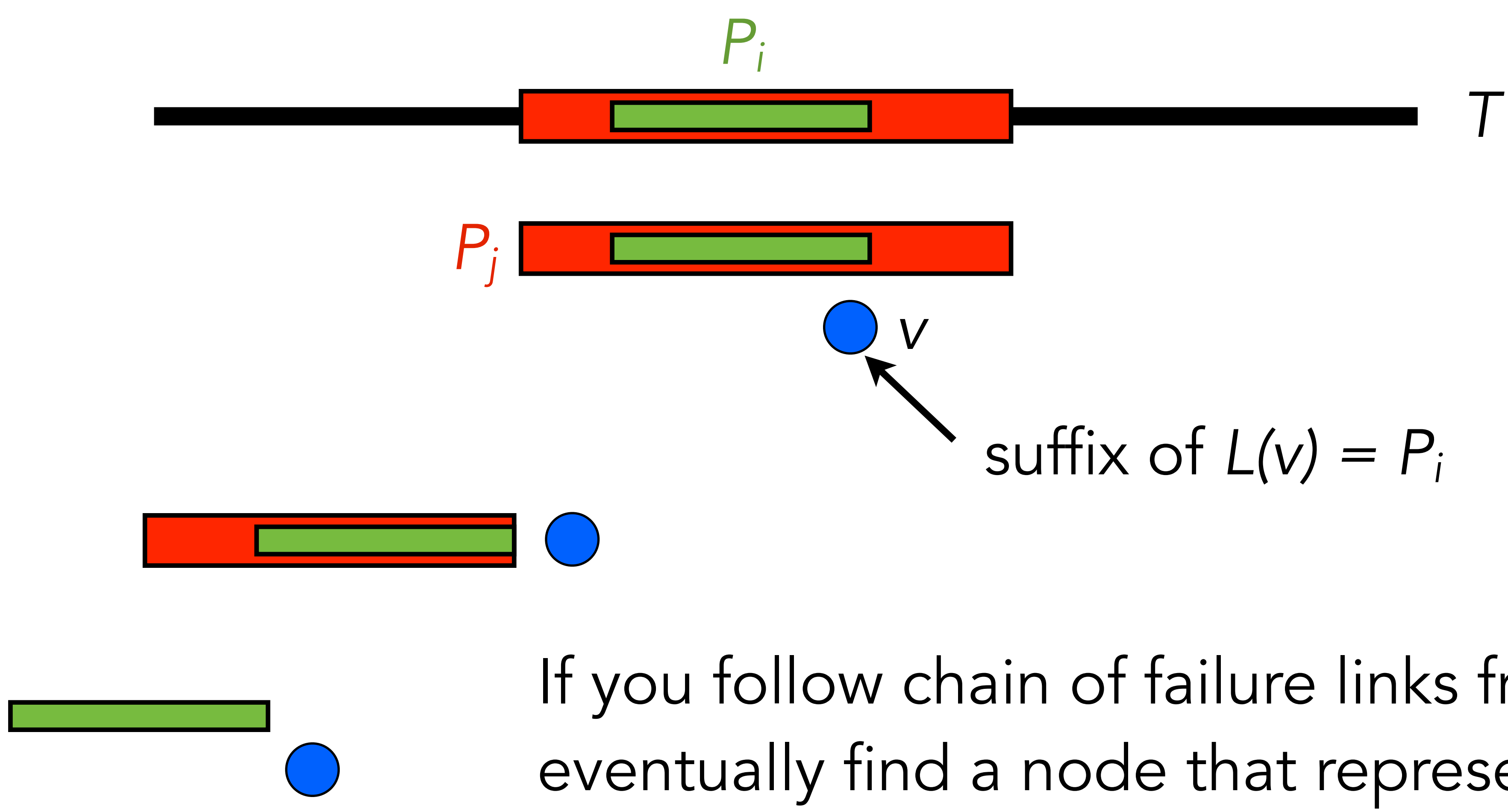
lp is never negative.

So we can "charge" the cost of following the link to the cost of just walking down the path.

Therefore running time =
O(total size of keyword tree) =
O(size of pattern set)



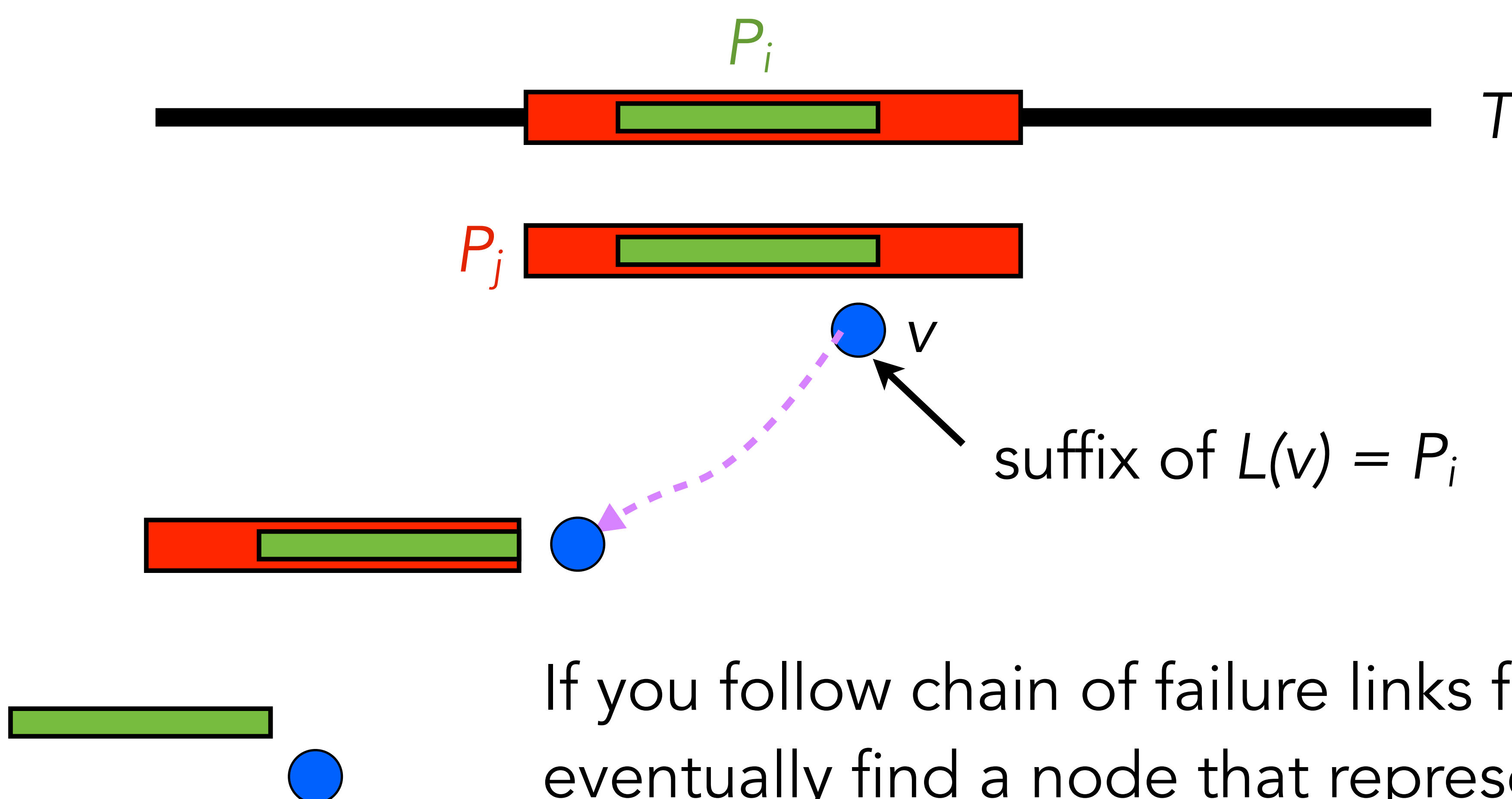
One Bug: If P_i is a substring of P_j



If you follow chain of failure links from v, you eventually find a node that represents P_i.

v <u>represents a full pattern</u> := v is labeled as a full pattern, or there is some node labeled as a full pattern reachable following failure links from v.

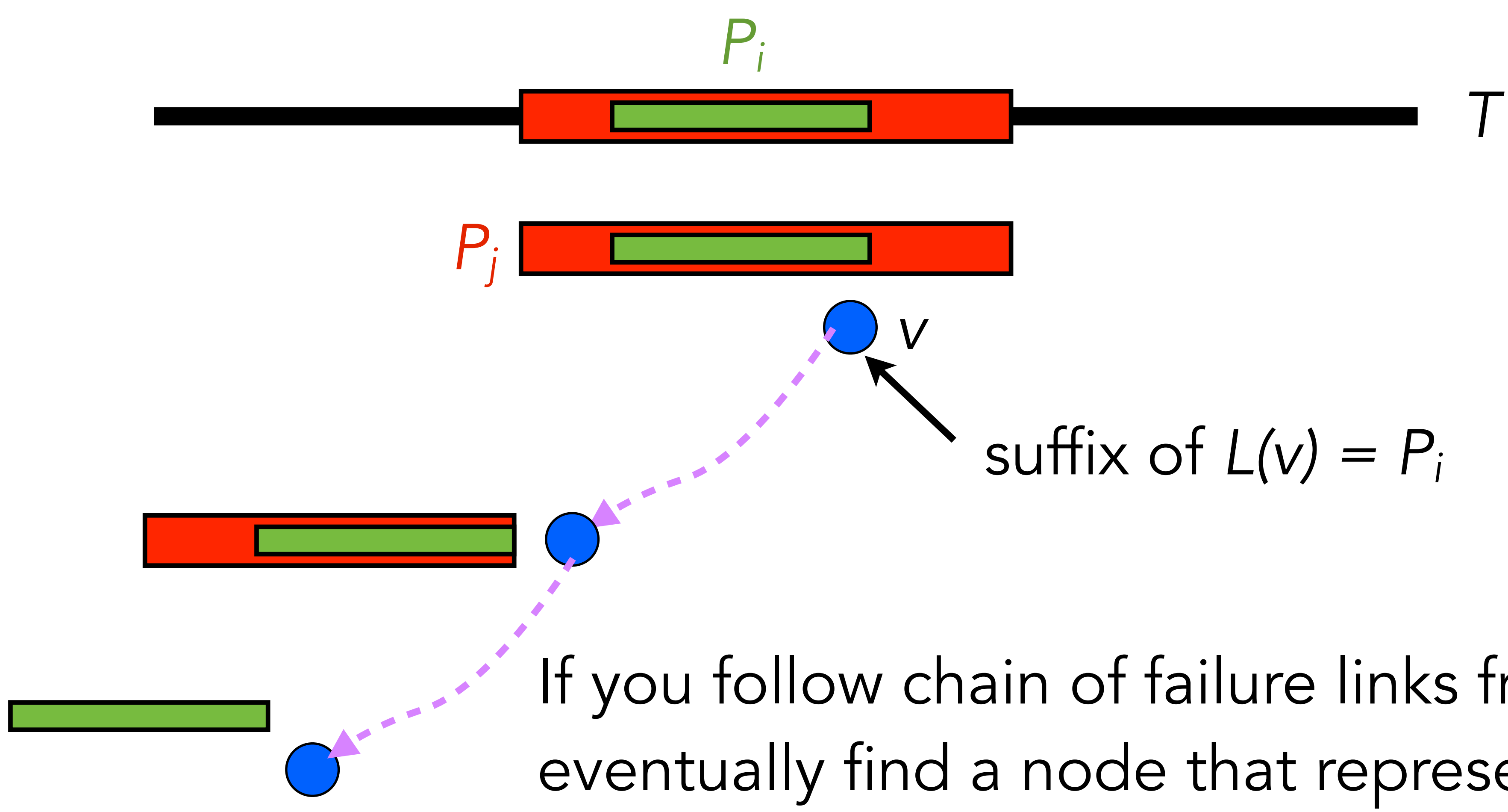
One Bug: If P_i is a substring of P_j



If you follow chain of failure links from v, you eventually find a node that represents P_i.

v represents a full pattern := v is labeled as a full pattern, or there is some node labeled as a full pattern reachable following failure links from v.

One Bug: If P_i is a substring of P_j



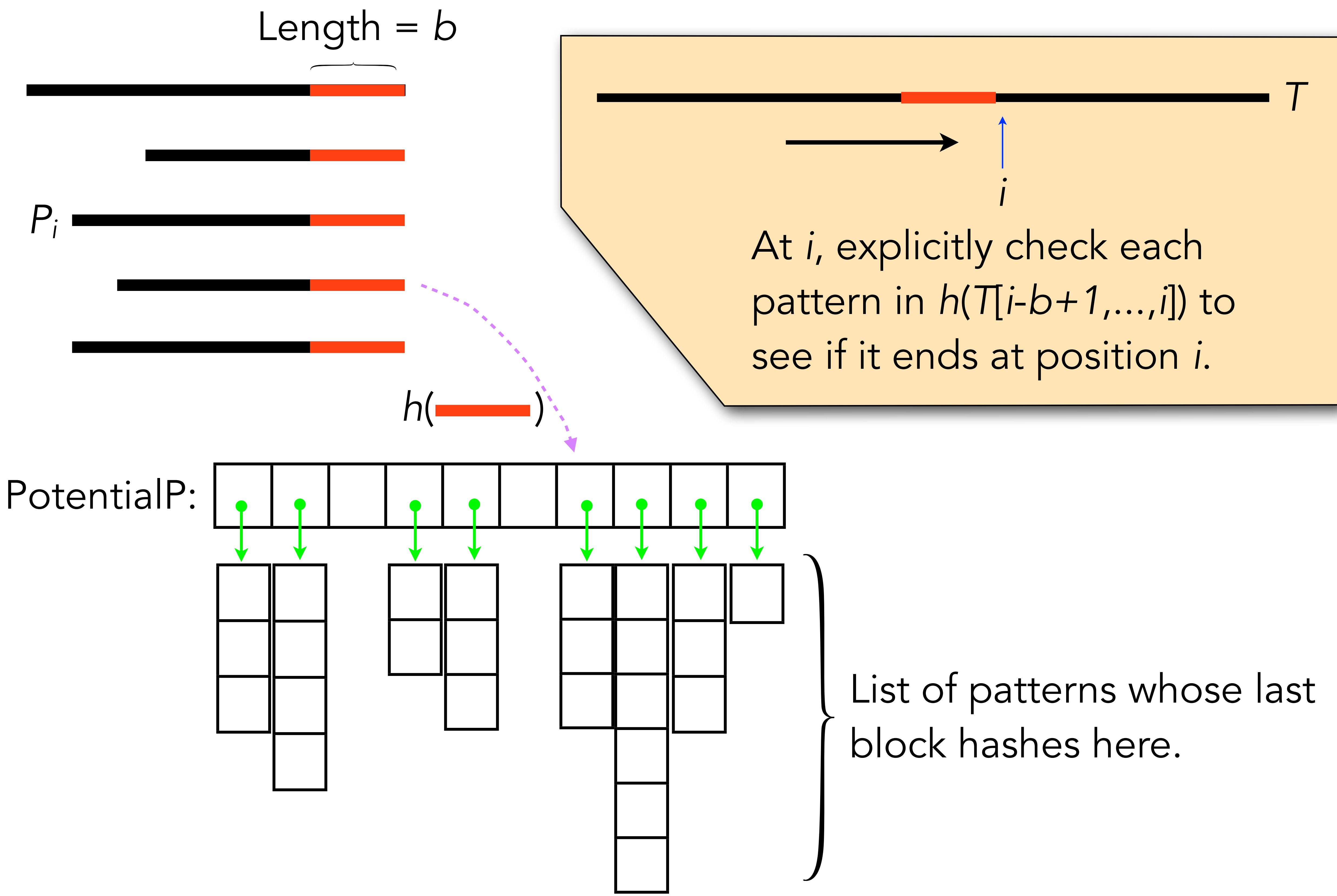
If you follow chain of failure links from v, you eventually find a node that represents P_i.

v <u>represents a full pattern</u> := v is labeled as a full pattern, or there is some node labeled as a full pattern reachable following failure links from v.

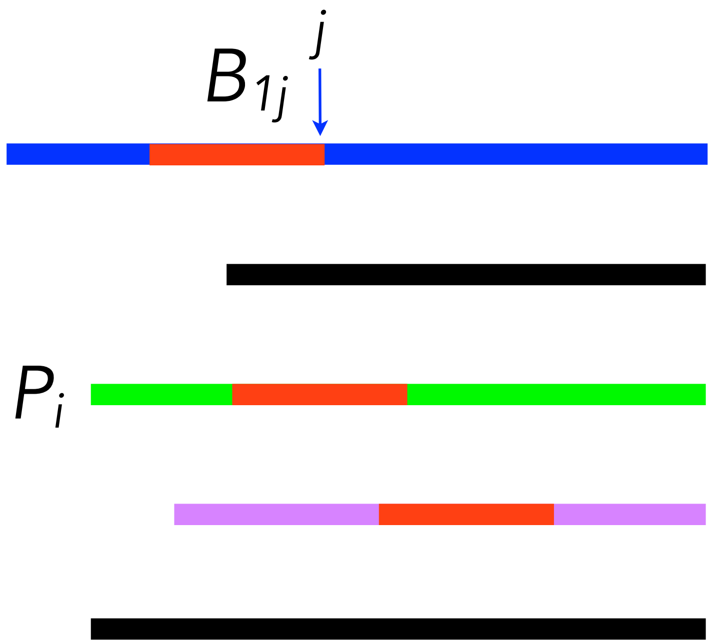
Wu-Mandber

A suffix approach

Wu-Mandber: Check



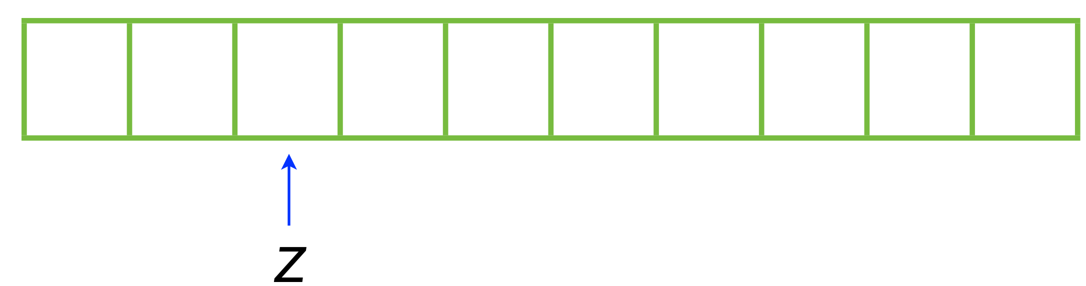
Wu-Mandber: Shift



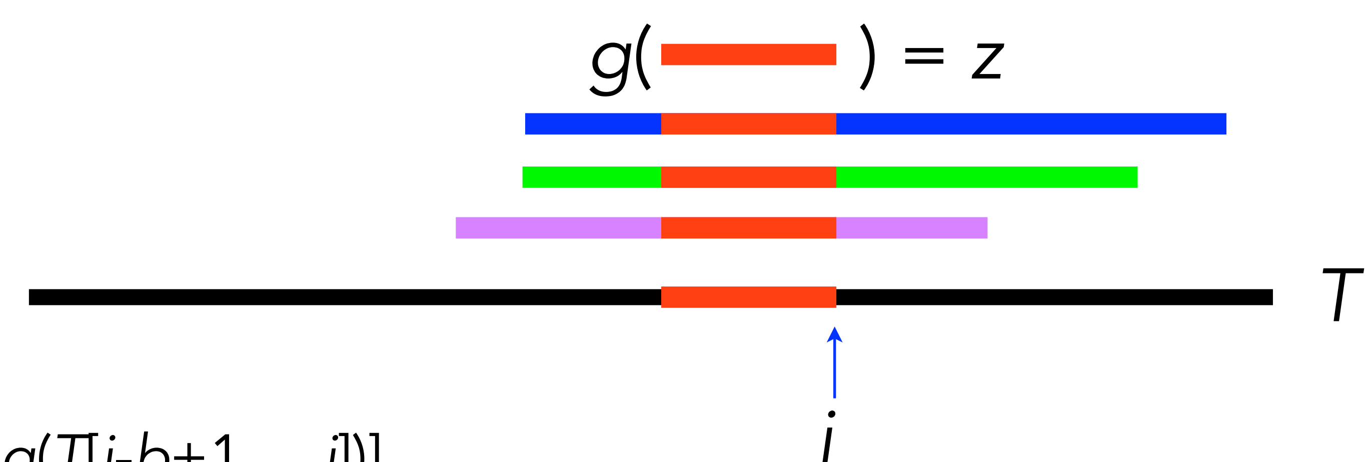
B_{ij} := block of length b ending at position j in pattern P_i.

$= \min \{ |P_i| - j : g(B_{ij}) = z \}$

GoodShift:



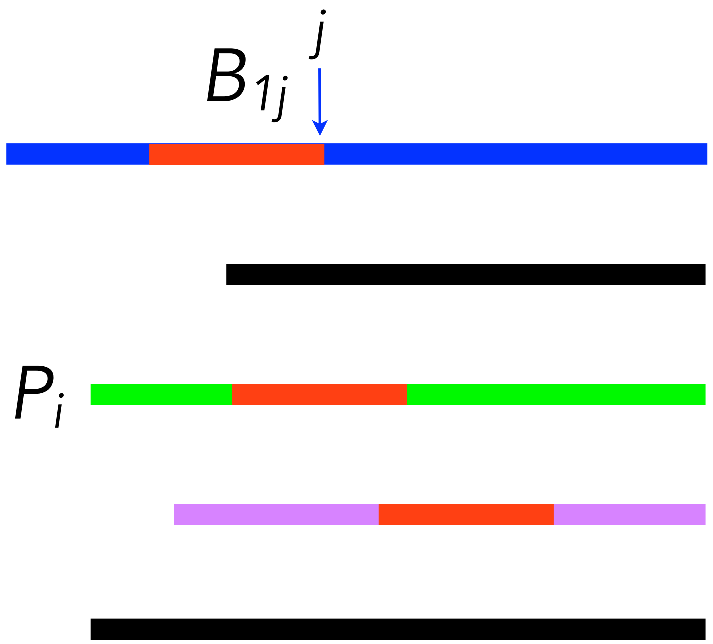
GoodShift[z] contains the amount that it is safe to shift by if we know T ending at i hashes to z with hash function g.



Shift i by GoodShift[$g(T[i\text{-}b+1,\ldots,i])$]

If Shift = 0: perform the Check on previous slide, and shift by 1.

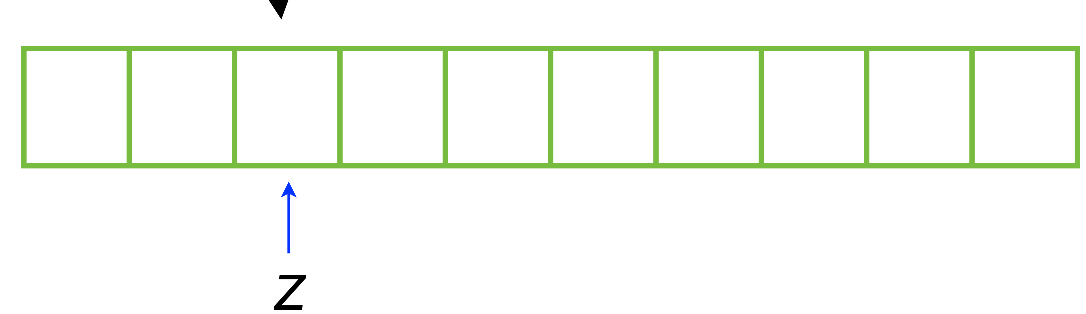
Wu-Manber: Shift



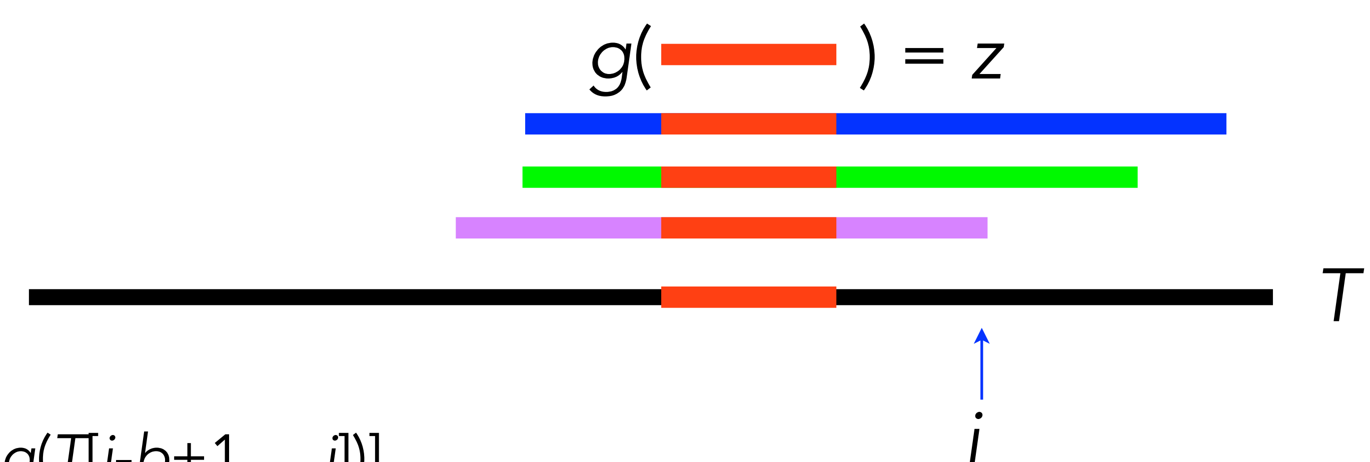
B_{ij} := block of length b ending at position j in pattern P_i.

$= \min \{ |P_i| - j : g(B_{ij}) = z \}$

GoodShift:



GoodShift[z] contains the amount that it is safe to shift by if we know T ending at i hashes to z with hash function g.



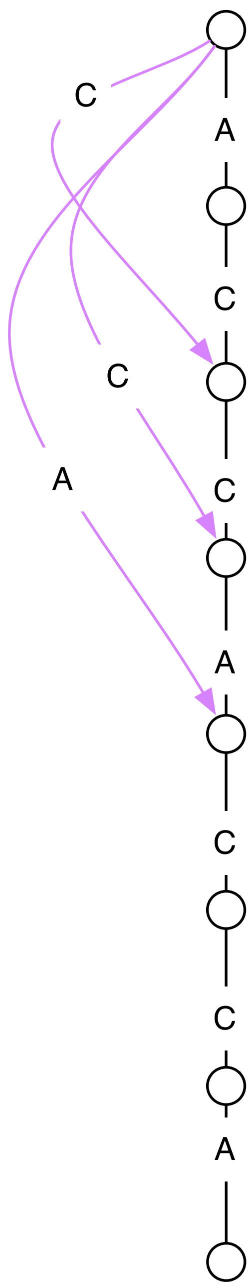
Shift i by GoodShift[$g(T[i-b+1,\ldots,i])$]

If Shift = 0: perform the Check on previous slide, and shift by 1.

Oracle Machine-based Approaches

(following Navarro & Raffinot)

Oracle-based Approach for 1 String

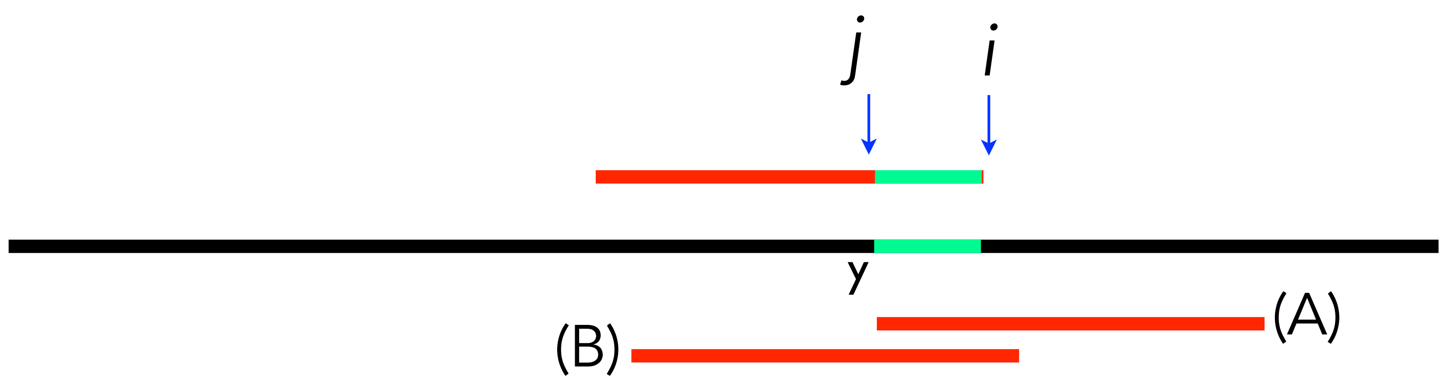


Factor Oracle: An FSA where every substring of P is spelled out by some path to the root.

Factor oracle search:

Build a factor oracle F on reverse(P)

At position i in T: walk backwards, simultaneously walking in F



(A) If we get stuck in *F* at position *j*, shift *P* to start just after *j*.

Works because: y must not be a substring of *P*.

(B) If we match |P| characters, we report a match and shift by 1.

Using Multi-string Matching For Filtering

(following Navarro & Raffinot)

Filtering for Approximate Matches

Let k be the maximum number of mismatches we will allow.

Thm. Let $P = p_1 \ldots p_j$ (where p_i are substrings), and let $a_1 \ldots a_j$ be non-negative integers with $\sum_i a_i = A$.
If Q and P match with ≤ k errors, then for some $1 \leq i \leq j$, Q contains a substring that matches p_i with $\leq \lfloor a_i k / A \rfloor$ errors.

Proof. If every sub-pattern p_i matched with $\geq 1 + \lfloor a_i k / A \rfloor$ errors, then there would be $\geq \sum_i (1 + \lfloor a_i k / A \rfloor) = k + 1$ total errors, a contradiction.

Idea: throw out parts of T to speed up approximate matching.

PEX

If $a_i = 1$ for all i and $A = k + 1$:

$\Longrightarrow$ some subpattern matches with < $\lfloor k / (k+1) \rfloor$ errors

$\Longrightarrow$ some subpattern matches exactly.

1. Divide P into k+1 equal-size chunks $p_1 \ldots p_{k+1}$
2. Use a multipattern search algorithm to find occurrences of $p_1 \ldots p_{k+1}$
3. Search region around each p_i match to see if it can be extended to a full P match.

PEX

If $a_i = 1$ for all i and $A = k + 1$:

$\Longrightarrow$ some subpattern matches with $< \lfloor k / (k+1) \rfloor$ errors

$\Longrightarrow$ some subpattern matches exactly.

1. Divide P into k+1 equal-size chunks $p_1 \ldots p_{k+1}$
2. Use a multipattern search algorithm to find occurrences of $p_1 \ldots p_{k+1}$
3. Search <u>region around</u> each p_i match to see if it can be extended to a full P match.