Algorithms in Genome Analysis, Spring 2023

Veli Mäkinen

Week 5

Genome analysis with suffix trees: maximal repeats, maximal unique matches, maximal overlaps

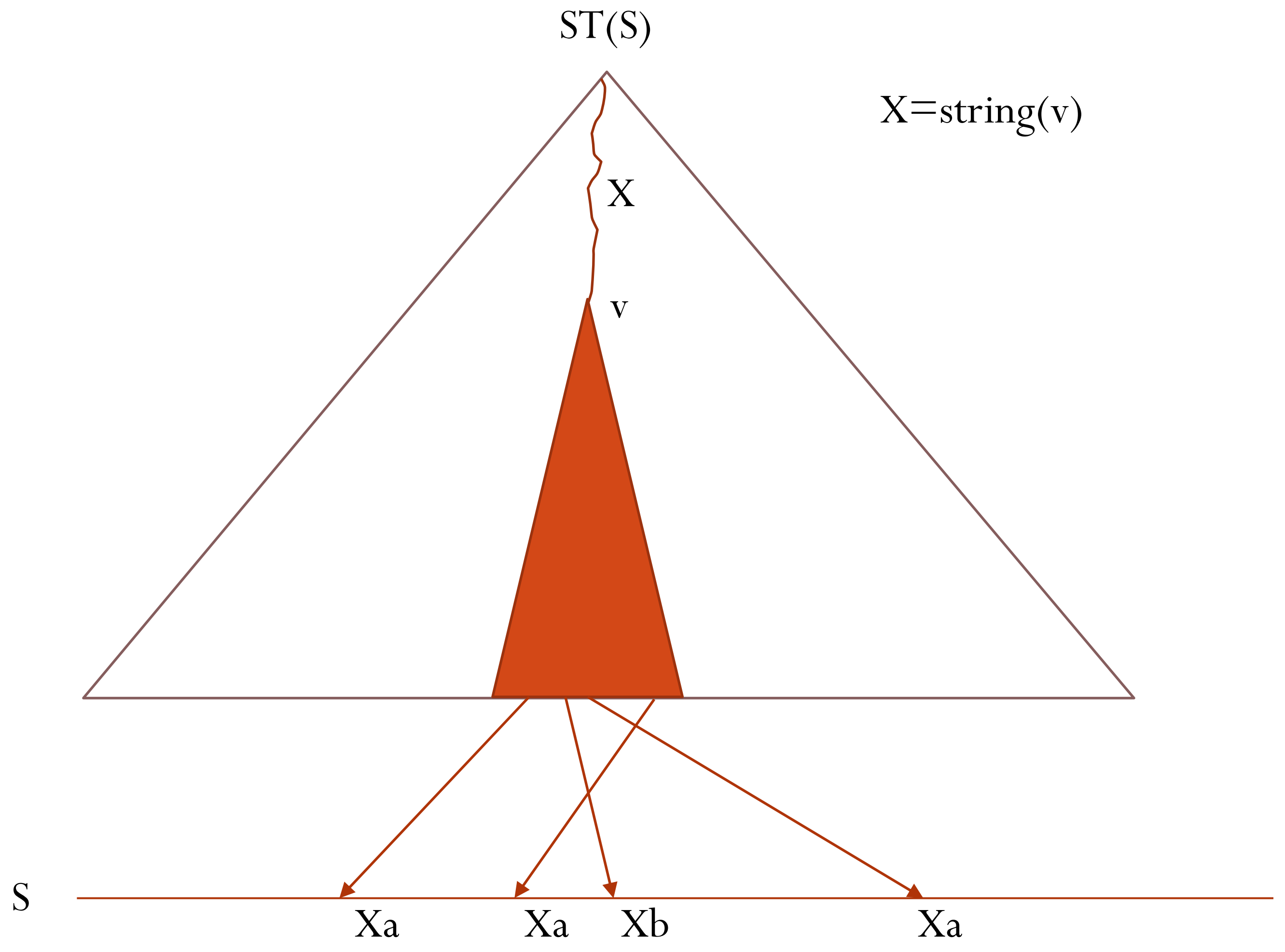
Maximal repeats

Compact encoding of the repeat structure of a genome

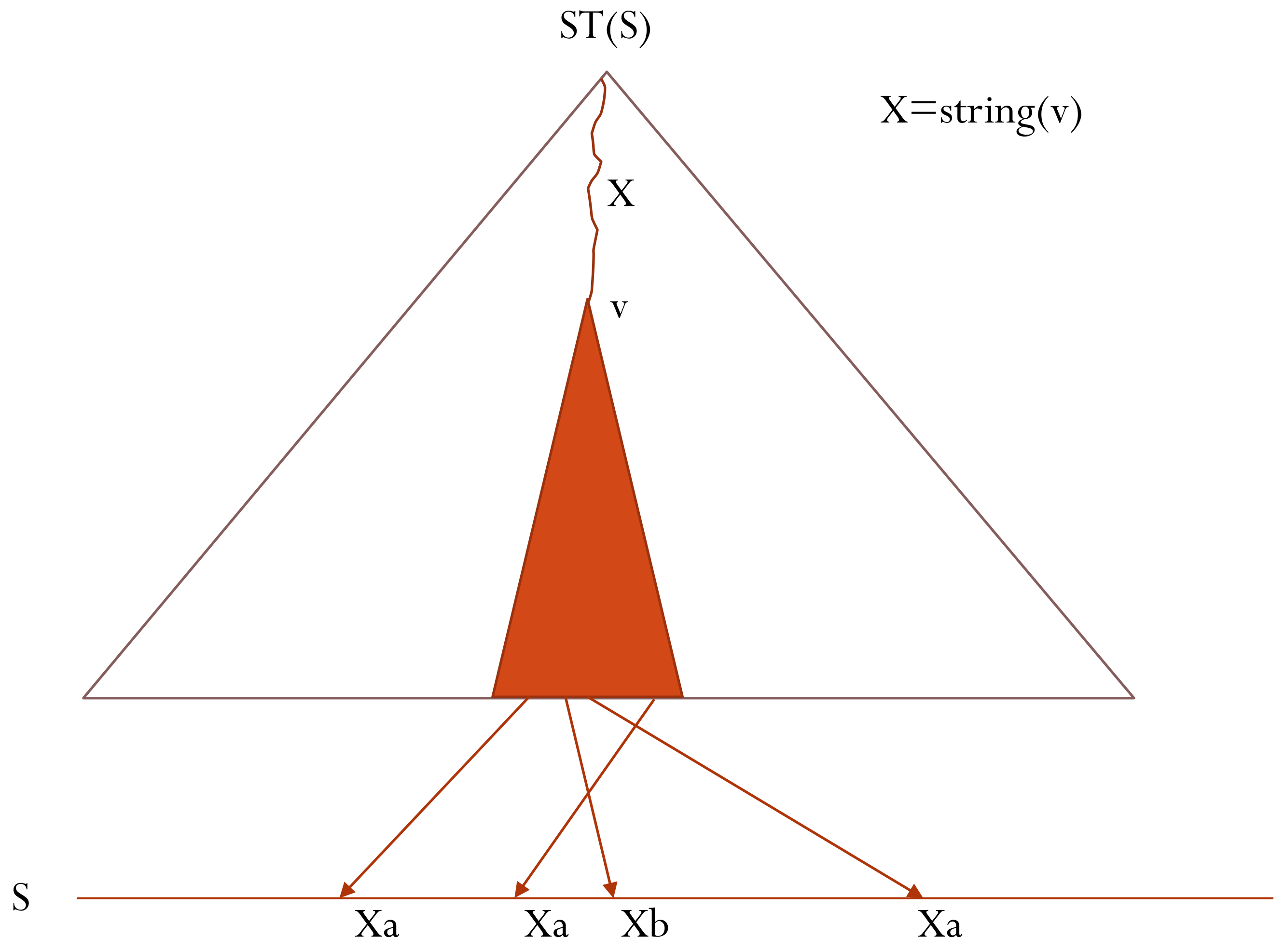
Maximal repeats

- A repeat is a substring of sequence S that has at least two occurrences: e.g. ACAGCAT
- Left-maximal (right-maximal) repeat is one that cannot be extended to the left (to the right, respectively) without loosing one of its occurrences
- If repeat is left- and right-maximal, it is called maximal repeat: e.g. C is left-maximal repeat but not right-maximal in ACAGCAT

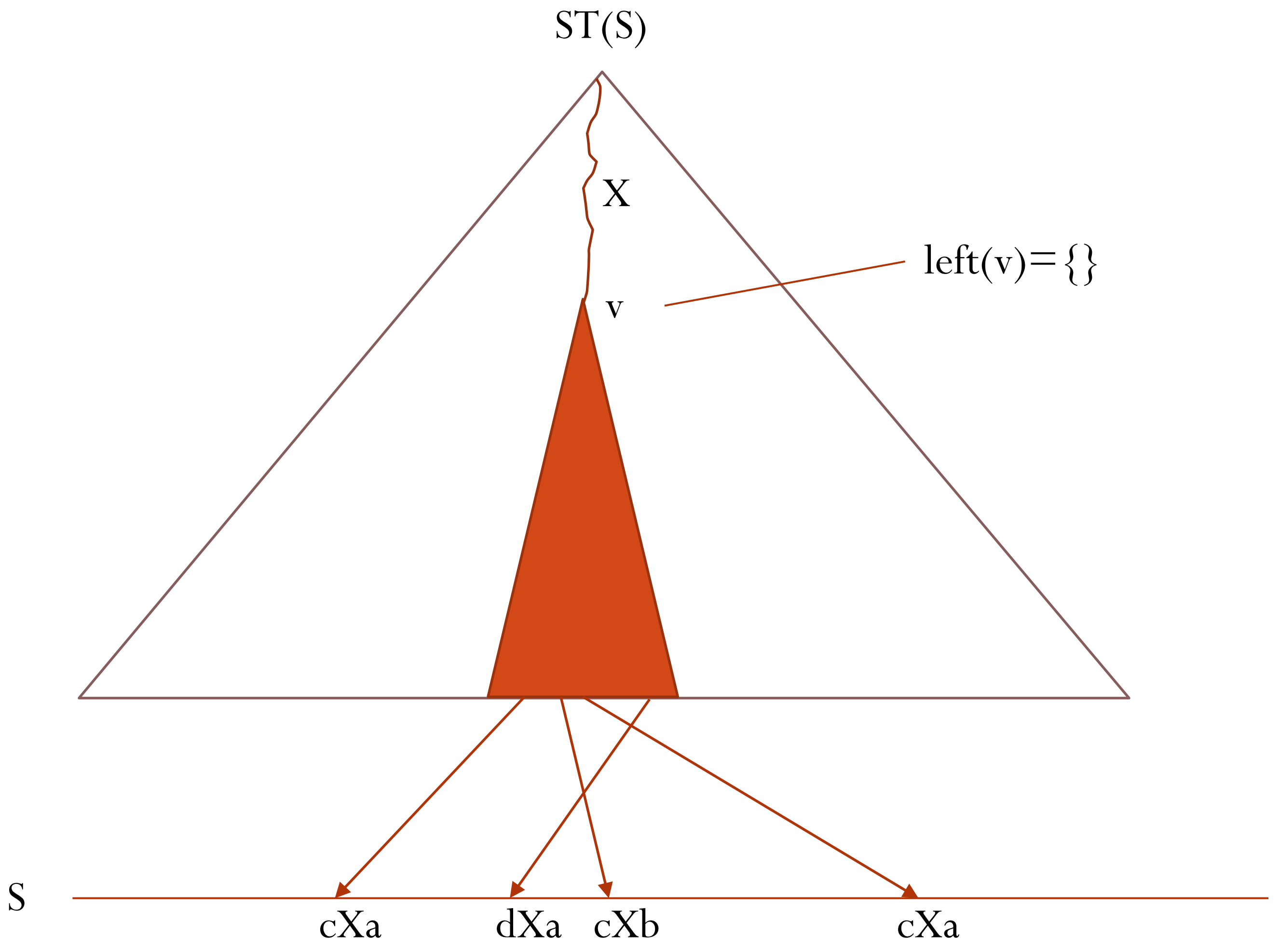
Right-maximal repeats = labels of paths from the suffix tree root to its internal nodes



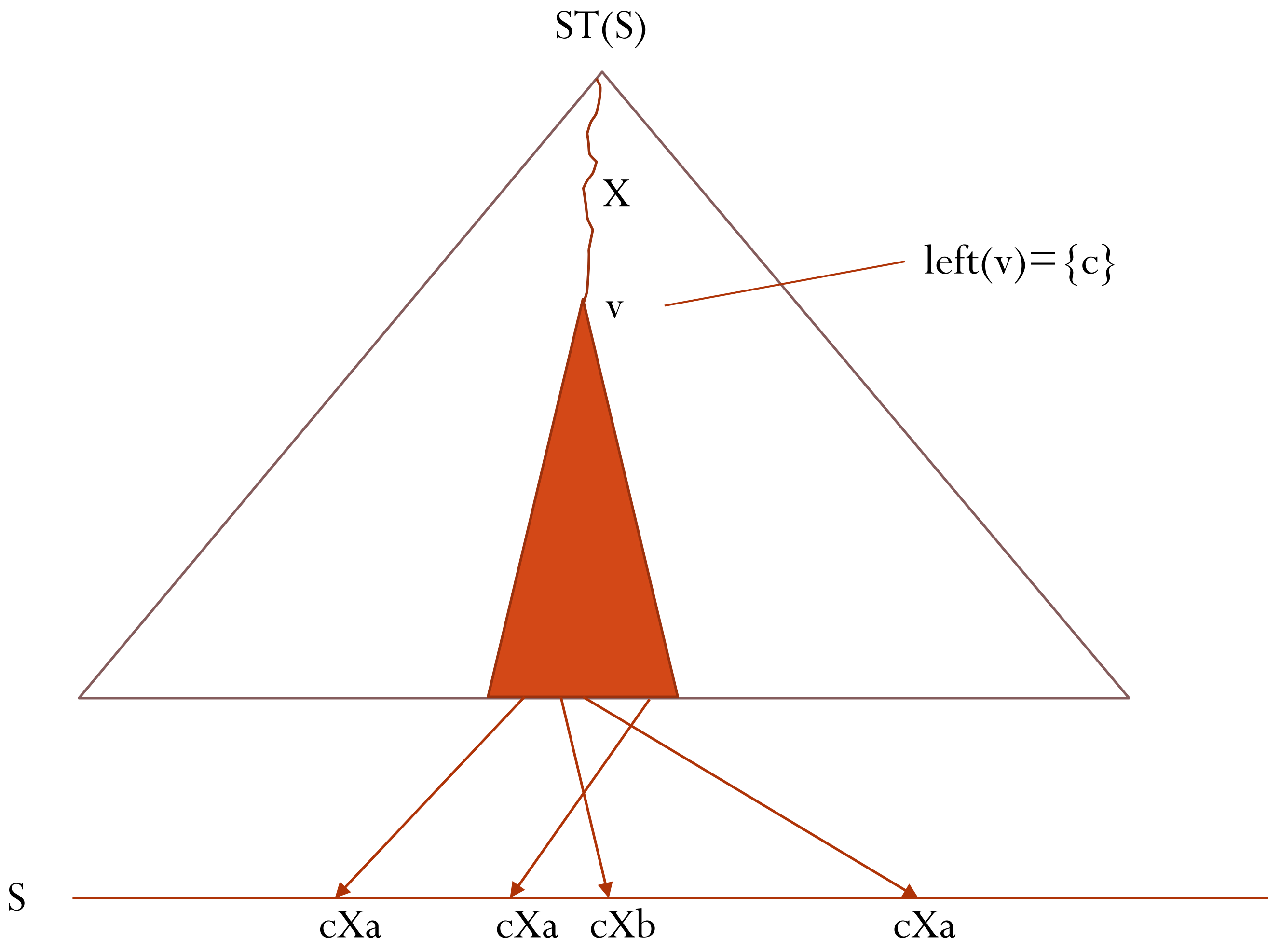
Right-maximal repeats = labels of paths from the suffix tree root to its internal nodes



Substring X is maximal repeat of sequence S iff X=string(v) for some node v of suffix tree of S and left(v)={}, where left(v)={c} if all occurrences of X in S are preceded by c, otherwise left(v)={}



left(v) can be computed bottom up for all nodes v in linear time
→ maximal repeats can be found in linear time



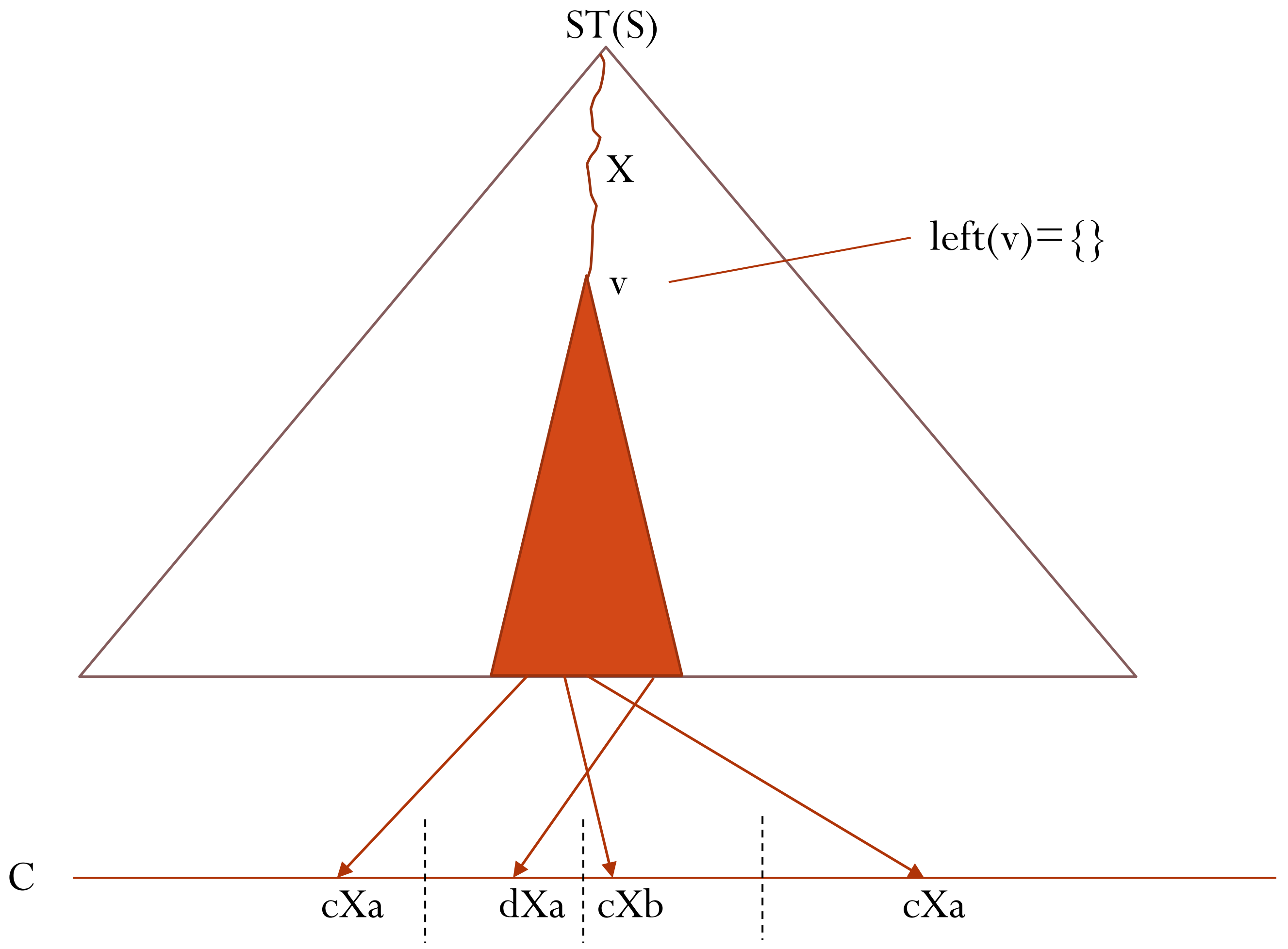
Maximal unique matches

- Repeat extension to multiple sequences
- Can be used to identify conserved regions for a divide-and-conquer type multiple sequence alignment heuristic

Maximal unique matches

- Consider a collection of d sequences $\{S^1, S^2, \ldots, S^d\}$
- Substring X is *maximal unique match (MUM)* if it occurs exactly once in each sequence S^i and its left- and right-extensions do not have this property
- Consider a generalized suffix tree: Suffix tree of the concatenation $C = S^1\$_1 S^2\$_2 \cdots S^d\$_d$, where we have added unique separator symbols in between
- Property 1: Substring X can be a MUM only if it is a maximal repeat in C
- Property 2: Substring X can be a MUM only if node v has exactly d leaves in its subtree, where v is s.t. string(v)=X
- What other properties must hold for substring X to be a MUM?

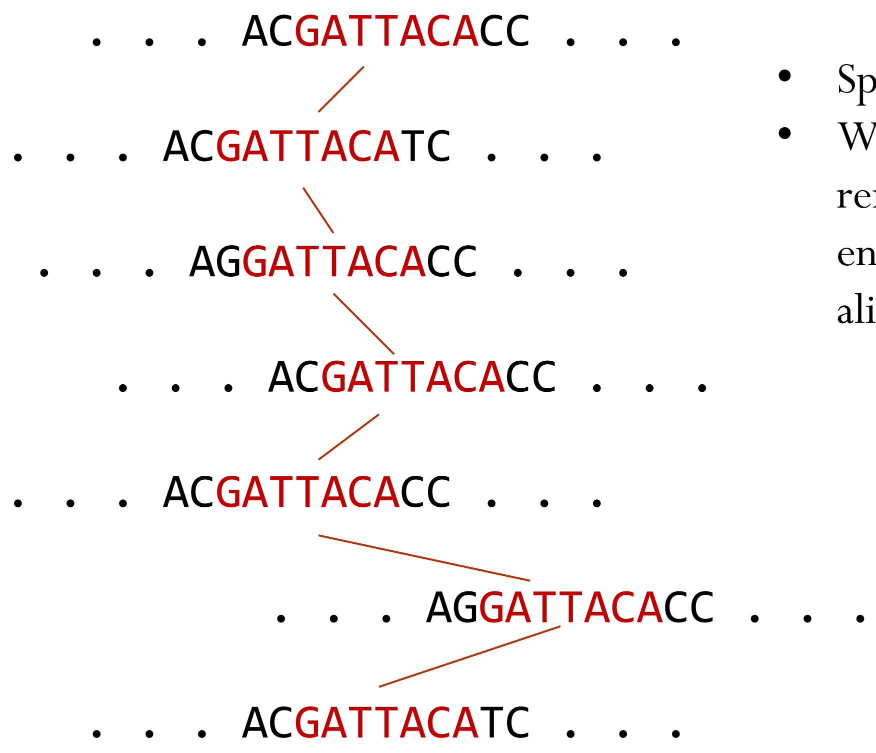
Let X be a MUM candidate (properties 1 and 2 holding).
Consider bit-vector $B^v[1..d]$ initialized to zeros.
Traverse the subtree of v and mark $B^v[i] = 1$ if a leaf points to a suffix starting at S^i.
Candidate X is a MUM iff $B^v[i]$ contains no zero.



MUMs in linear time

- MUM candidates can be found in linear time
- The subtrees corresponding to MUM candidates are disjoint
- Filling the bit-vectors take overall linear time
- Hence, MUMs can be found in linear time

MUMs as anchors for multiple alignment

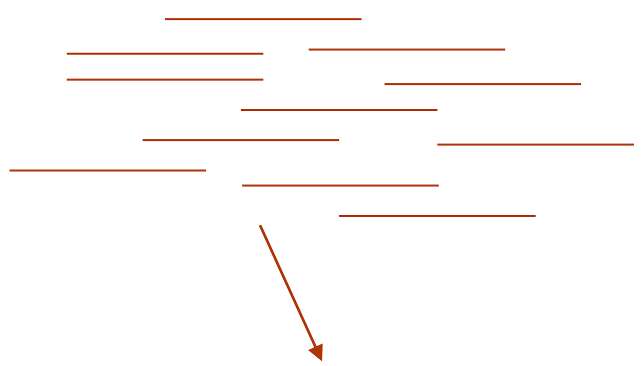


- Split at the MUM and recurse
- When no MUMs left, remaining MSA might be small enough to admit optimal alignment computation

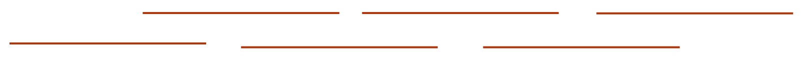
Maximal overlaps

How to build an assembly graphs efficiently

Maximal overlaps

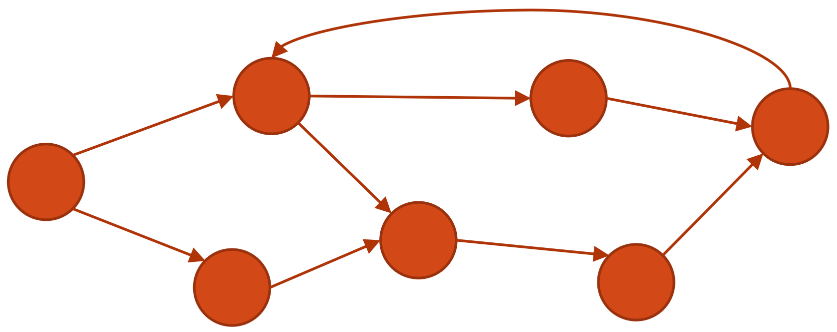


Set of sequencing reads



Subset sorted by suffix-prefix overlaps

Longer contiguous fragment of DNA

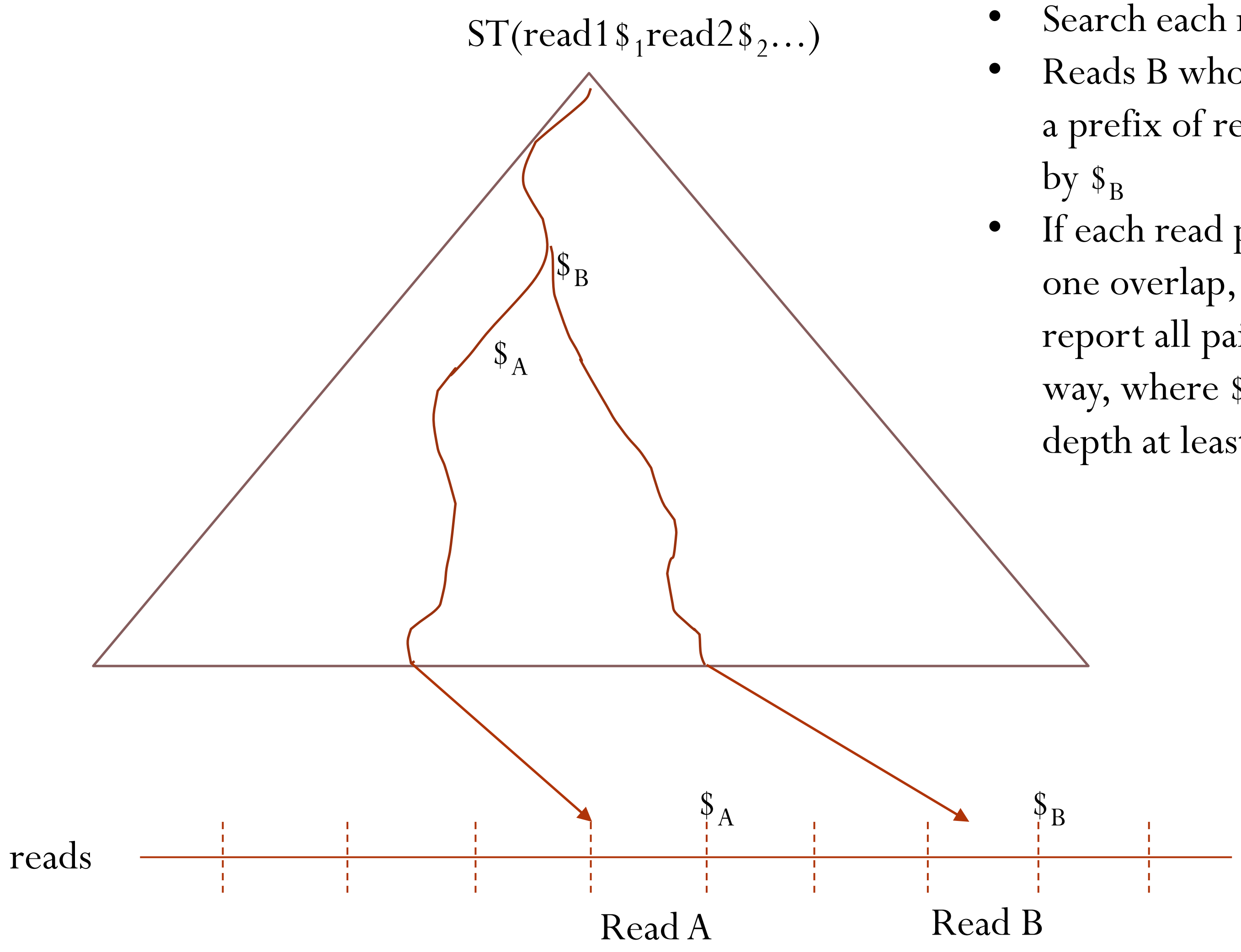


Overlap graph: nodes are reads, arcs present suffix-prefix overlaps

Maximal overlaps

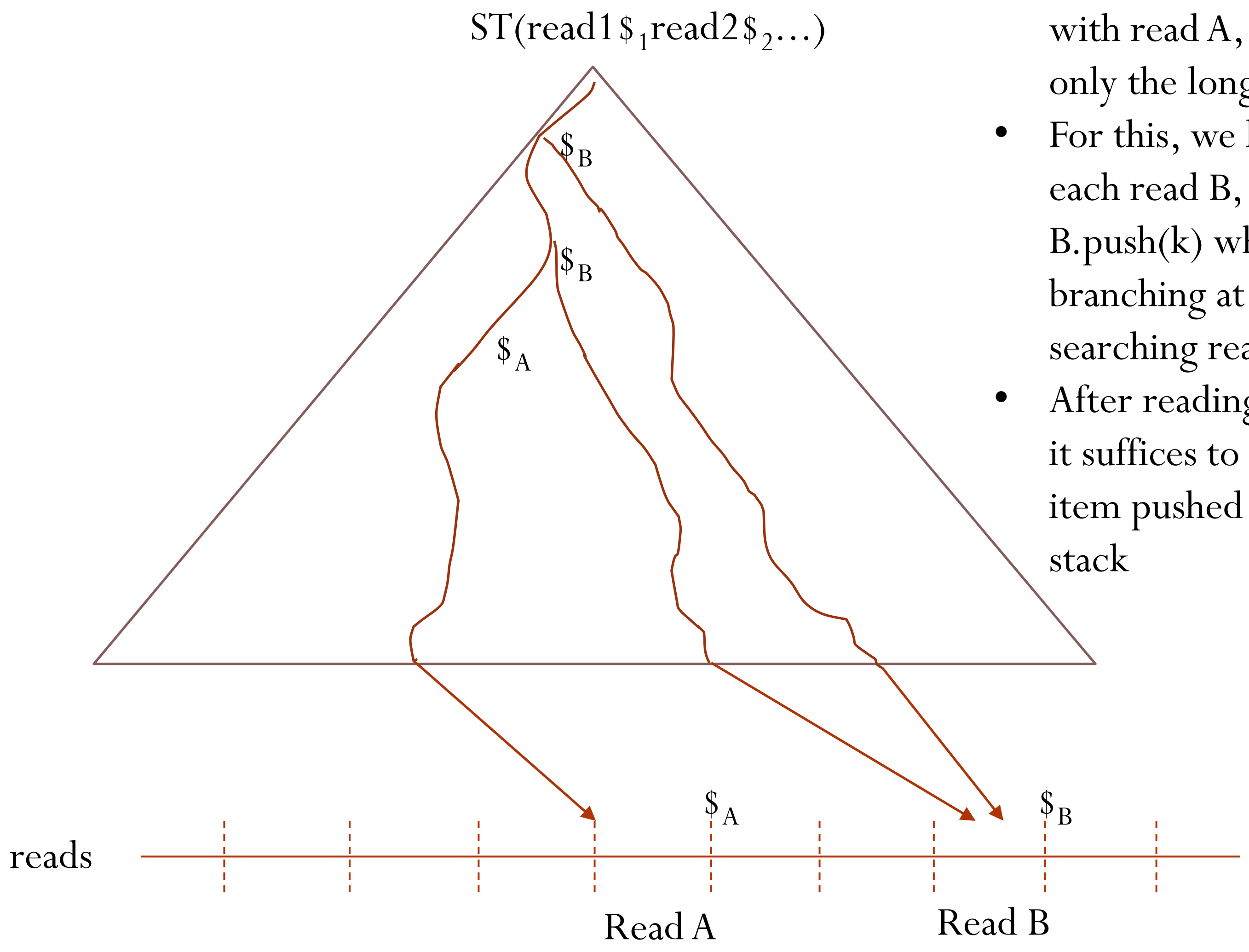
- Consider a set of reads
- We say that a pair of reads (A,B) has a significant suffix-prefix overlap if A[|A|-k+1..|A|]=B[1..k], where $k \geq \kappa$ and κ is a predefined threshold.
- For each pair of reads (A,B) with a significant overlap, we wish to report the length k of the largest overlap
- Can we find all such overlaps in linear time in the size of the input and output?

Maximal overlaps in suffix tree



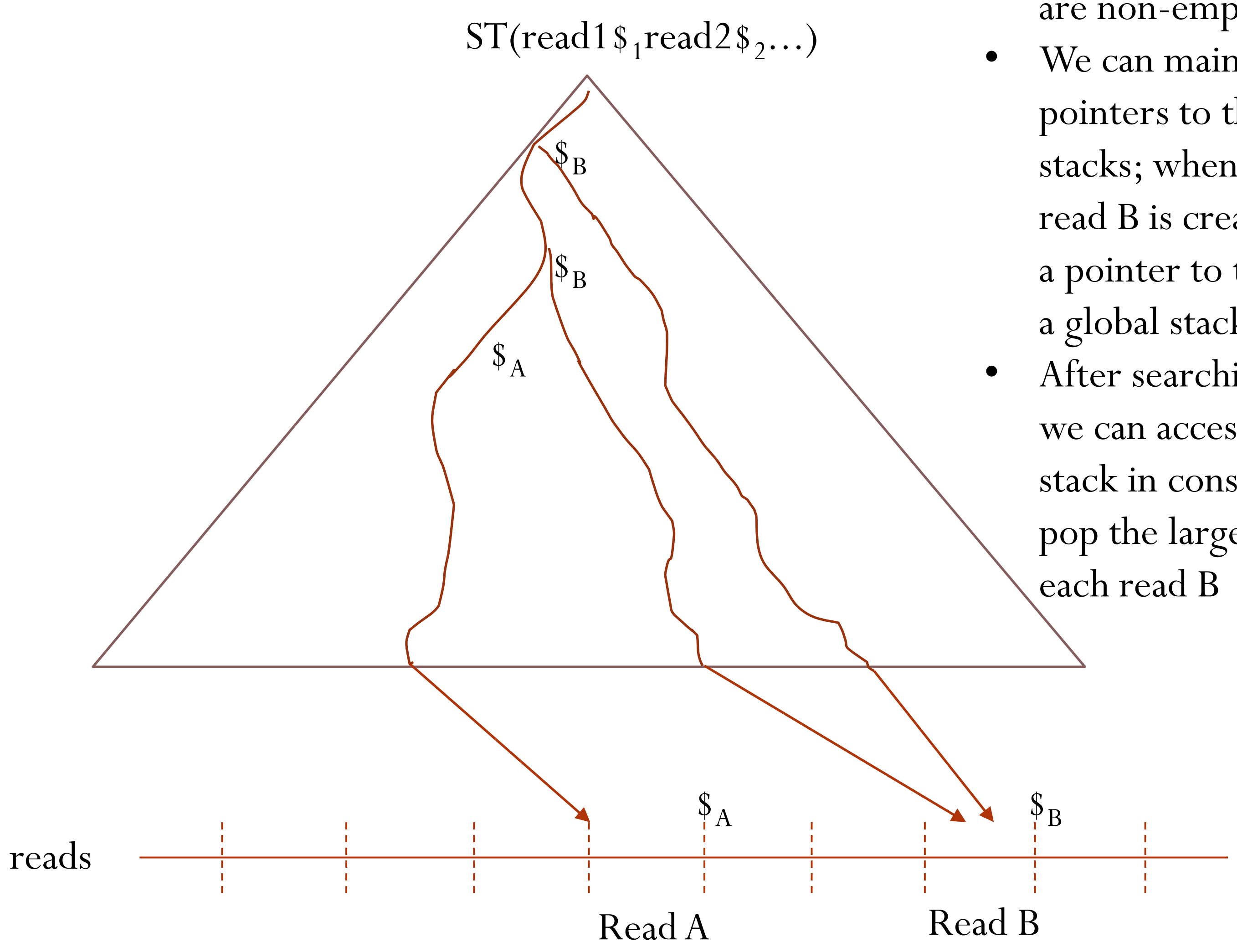
- Search each read A until its $\$_A$
- Reads B whose suffix overlaps a prefix of read A will branch by $\$_B$
- If each read pair has at most one overlap, one could just report all pairs (A,B) found this way, where $\$_B$ branches at string depth at least κ

Maximal overlaps in suffix tree



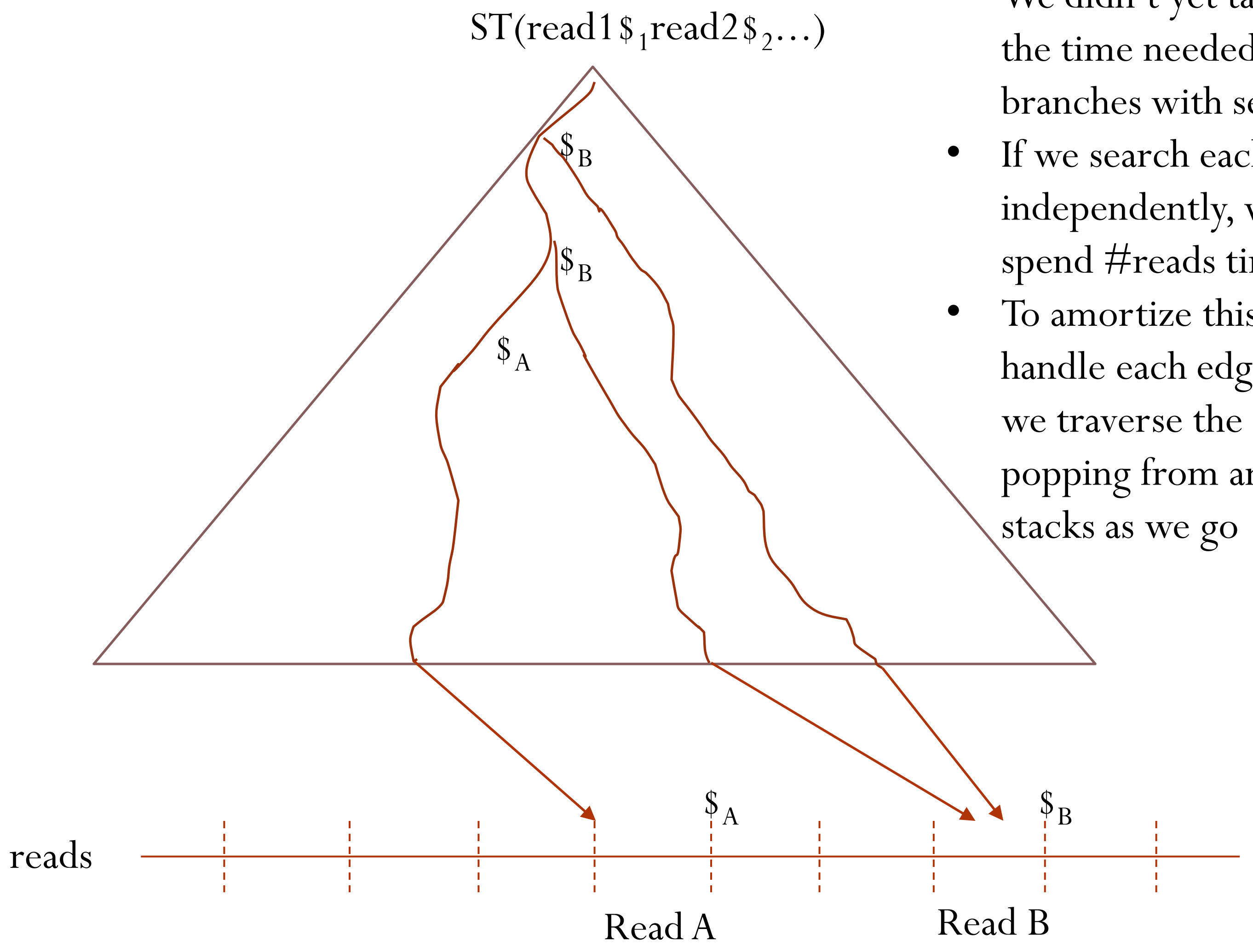
- If read B has multiple overlaps with read A, we wish to report only the longest
- For this, we keep a stack B for each read B, and apply B.push(k) when we see $\$_B$ branching at string depth k when searching read A
- After reading the whole read A, it suffices to output the last item pushed to each non-empty stack

Maximal overlaps in suffix tree



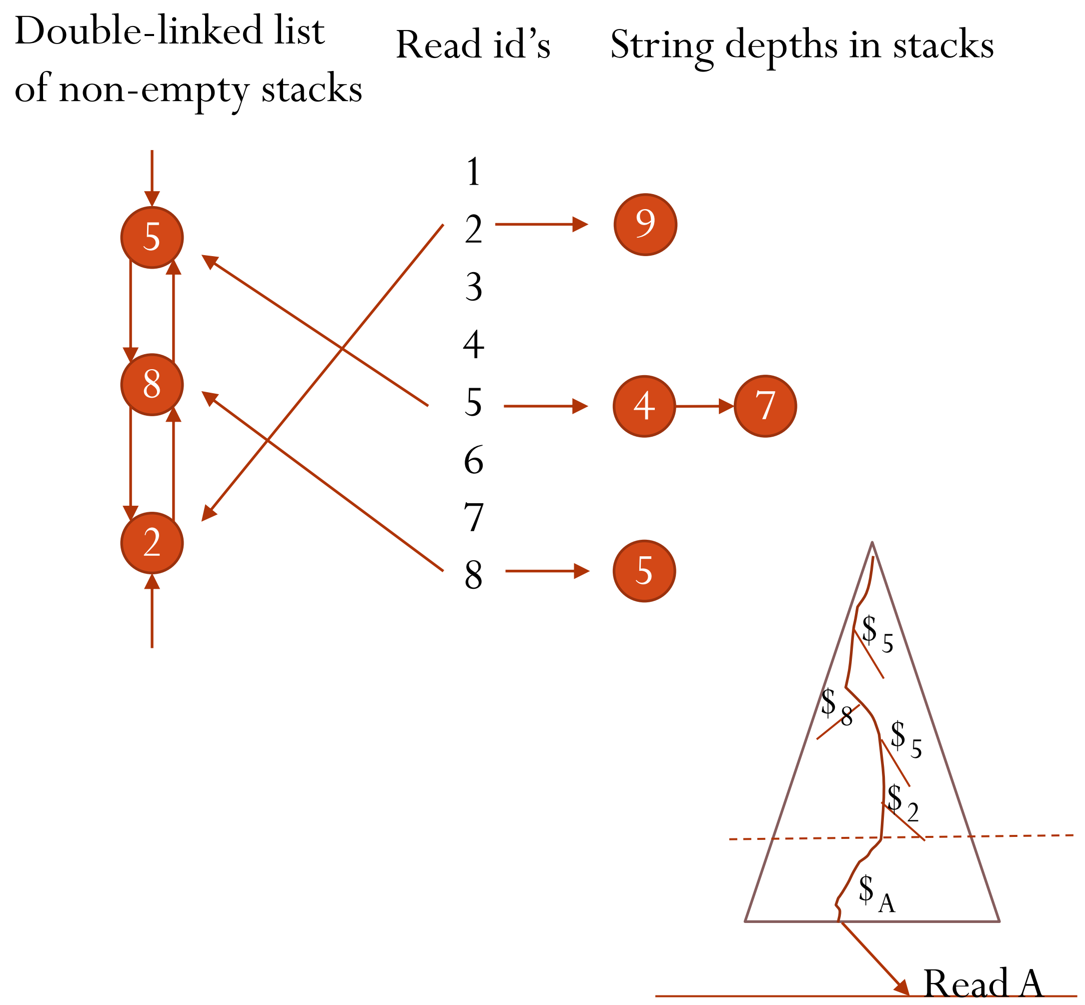
- How to know which stacks are non-empty?
- We can maintain a stack of pointers to the non-empty stacks; whenever stack for read B is created, we add a pointer to this stack in a global stack
- After searching read A, we can access each non-empty stack in constant time and pop the largest overlap between each read B

Maximal overlaps in suffix tree



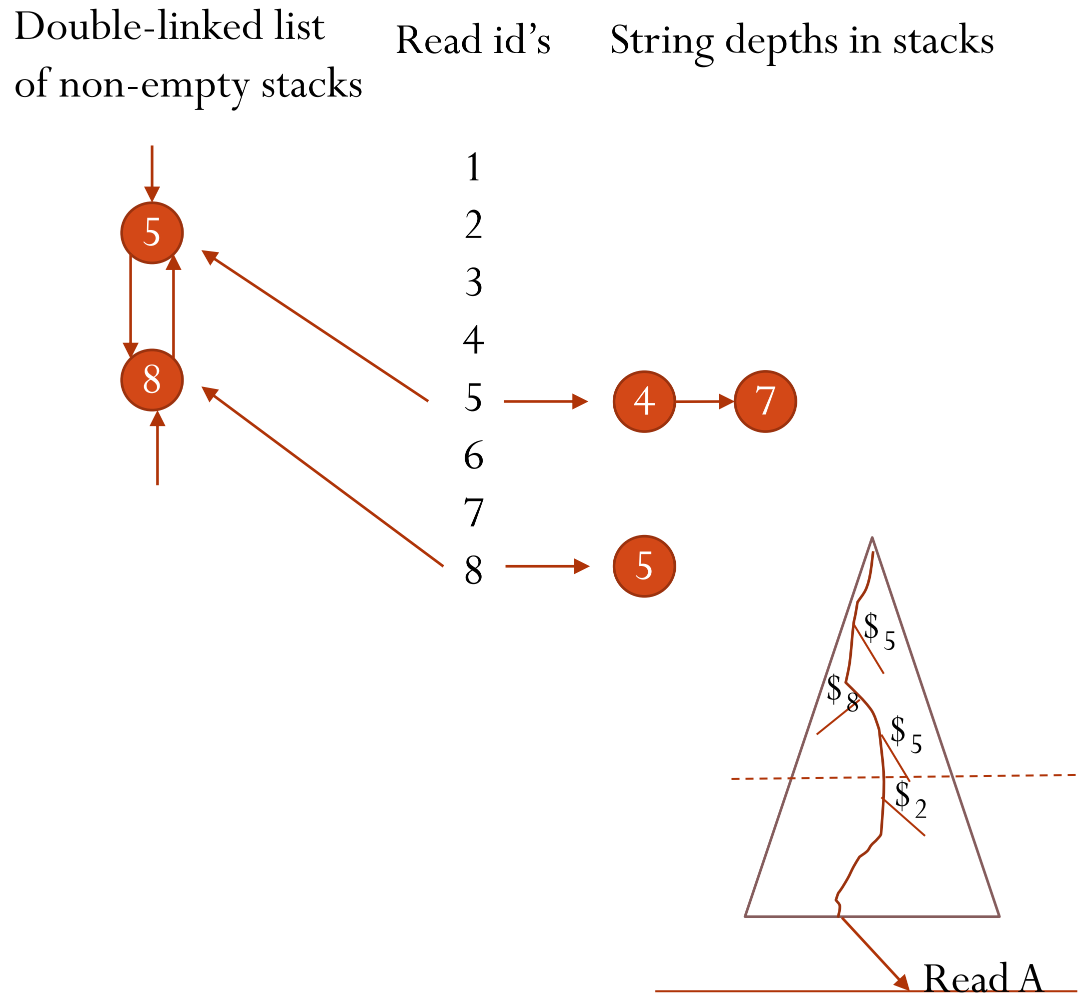
- Is this approach now optimal?
- We didn't yet take into account the time needed to detect the branches with separator symbols
- If we search each read A independently, we may need to spend #reads time for each overlap
- To amortize this cost (in order to handle each edge constant time), we traverse the whole tree once, popping from and pushing to the stacks as we go

Maximal overlaps in suffix tree



- When reaching locus corresponding to read A, one still gets the maximal overlaps from the non-empty stacks
- When a stack gets empty, pointer to it needs to be removed from the global stack; with some care (handling children in reverse order when coming back to a node), popping from the global stack suffices to remove the correct pointer
- Alternatively, one can replace the global stack with a double-linked list, and keep back-pointers to this list

Maximal overlaps in suffix tree



- When reaching locus corresponding to read A, one still gets the maximal overlaps from the non-empty stacks
- When a stack gets empty, pointer to it needs to be removed from the global stack; with some care (handling children in reverse order when coming back to a node, popping from the global stack suffices to remove the correct pointer
- Alternatively, one can replace the global stack with a double-linked list, and keep back-pointers to this list

Pseudo-code

```
DFS ST($R_1\#_1 R_2\#_2 \cdots R_d\#_d$)

↓ At v with child whose edge label starts with $\#_j \cdots$ :
    if s[j].empty() then
        p[j] = d-l-list.append(j)   // ...
    s[j].push(|l(v)|)   // string depth added to j-th stack

↓ At leaf v with $l(v) = R_i\#_i \cdots$ :
        for j ∈ d-l-list do:
            report (i, j, s[j].top())

↑ At v with child whose edge label starts with $\#_j \cdots$ :
        s[j].pop()
        if s[j].empty() then
            d-l-list.remove(p[j])   // ...
```