Suffix Tree and Array

String Matching

- So far we learned how to find "approximate" matches – the alignments. And they are difficult. Finding exact matches are much easier.
- To search for a short string P of length m in a large text T of length n.
- Applications:
  - Keyword searching
  - DNA reads mapping
- Type I: Match only once.
  - E.g. KMP algorithm and Apostolico-Giancarlo algorithm.
  - O(m) to preprocess, and O(n) to match.
- Type II: Match multiple patterns multiple times.
  - Better index T first to speed up the matching time.

Things To Study

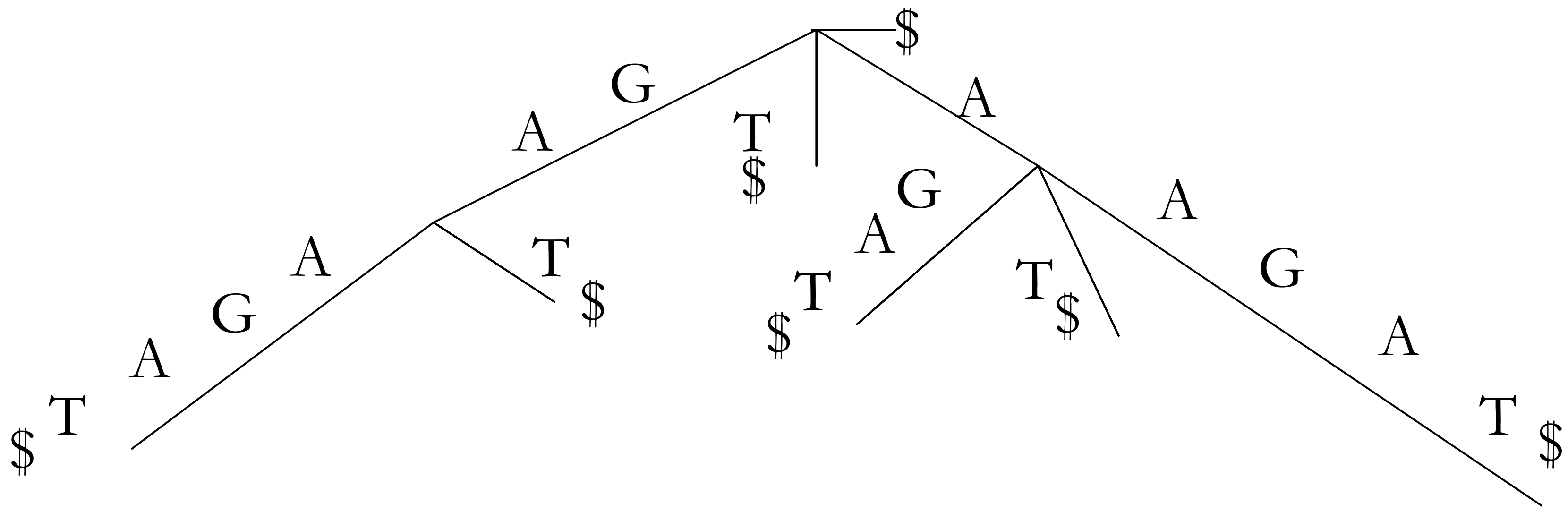
- Suffix tree and array are two data structures for this purpose.
- Suffix Tree
  - Data structure
  - A few examples of using suffix tree to solve practical problems.
- Suffix Array
  - Data structure
  - The skew algorithm for constructing suffix array.

A Little History

- 1973, Weiner introduced the concept of suffix tree (position tree), which Donald Knuth subsequently characterized as "Algorithm of the Year 1973".
- 1990, Gene Myers and Udi Manber proposed suffix array.
  - Gene Myers: former VP Informatics Research at Celera Genomics
  - Udi Manber: VP engineering, Google.
- 1992, Gonnet, Baeza-Yates & Snider independently discovered suffix array (called PAT array).
  - Gaston Gonnet: cofounders Maplesoft and OpenText.
  - Baeza-Yates: VP for Yahoo! Europe and Latin America.

As a picture

- Here is the suffix tree for GAAGAT$



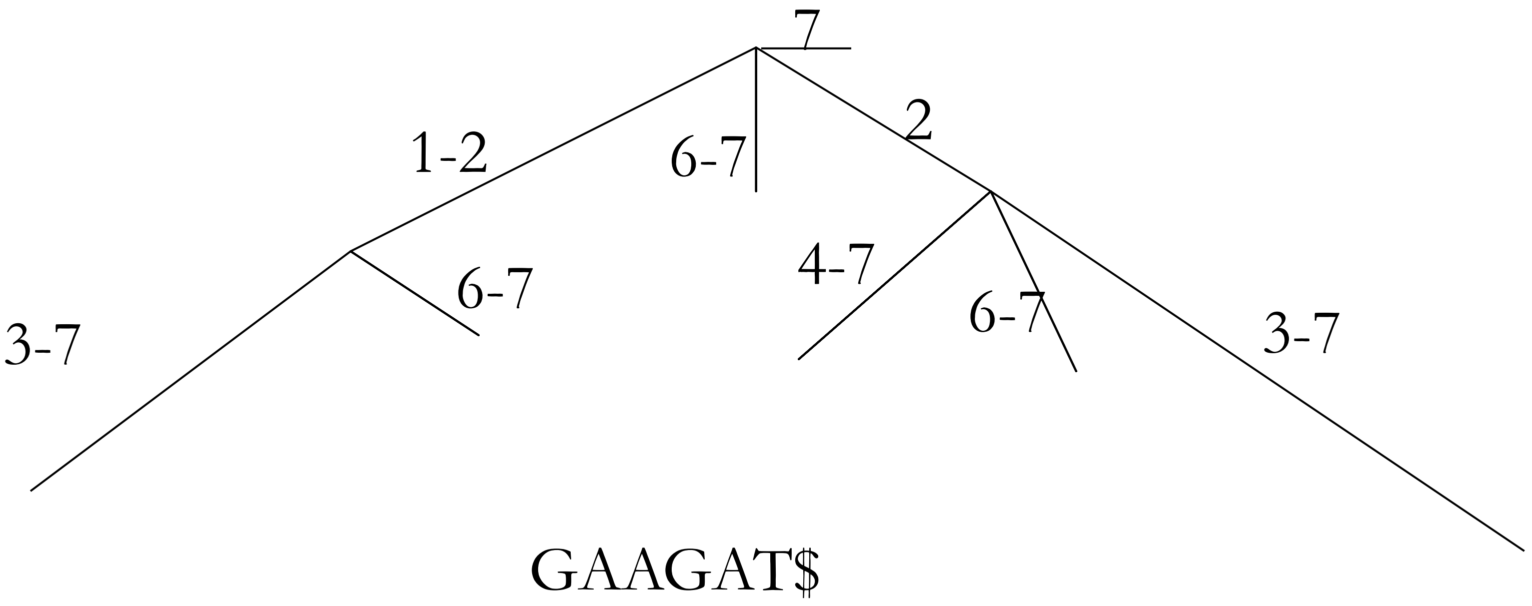
- An edge is labelled with a substring of the original string.
- A **node**'s label is the concatenation of all edge labels for the path leading to that node.
- The path from the root, *r*, to any leaf *x* is a suffix of the string S.
- Suppose there is a special "end-of-string" character, each suffix will end at the leaf.
- Each internal node has at least 2 children.
- Edge labels to the child nodes of an internal node start with different letters.

Application I. Search for a substring.

- Any substring of S is a **prefix** of a **suffix**.

- Example of using this: Is the string x a substring of S?
    - Start at the root, and follow paths labelled by the characters of x. If you can get to the end of x, then yes, it is.

Linear Space Structure



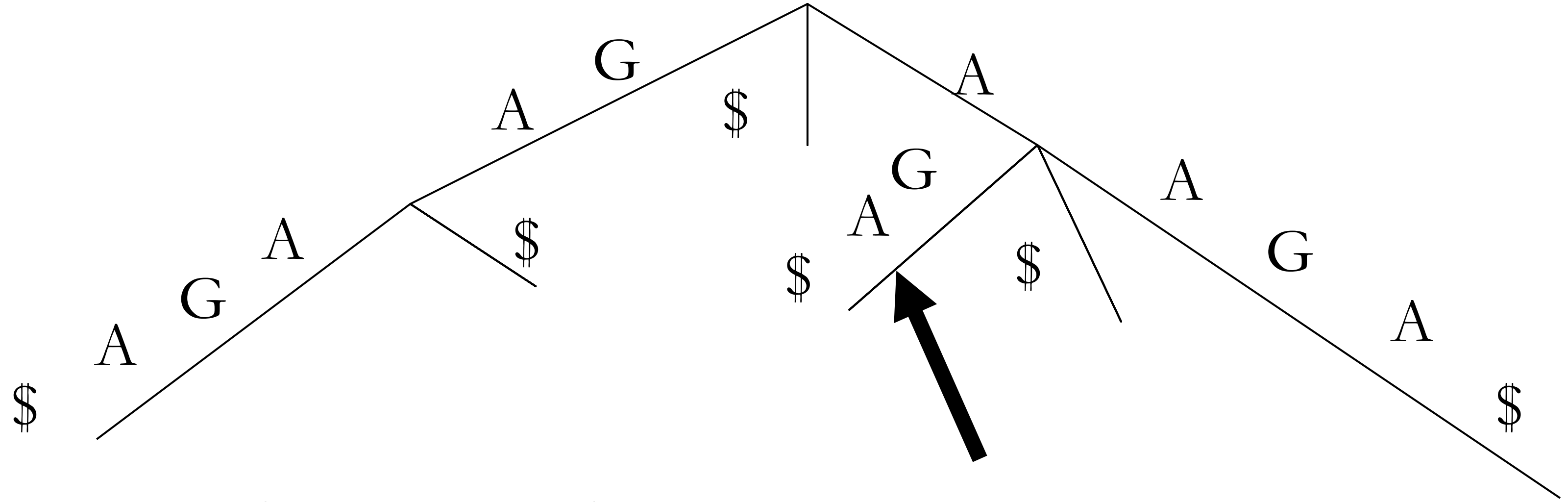
- Each edge doesn't need to be labelled with a string, but just with starting and ending in the sequence.
- This is the same suffix tree as before, but in **linear space**.

How to construct a suffix tree?

- There is a linear time algorithm to construct a suffix tree. (We will not study it.)
- We'll examine a quadratic-time algorithm (quite intuitive).
- The idea is to
  - Start with an empty tree.
  - Iteratively add more suffices into the tree (from shortest to longest).

One round

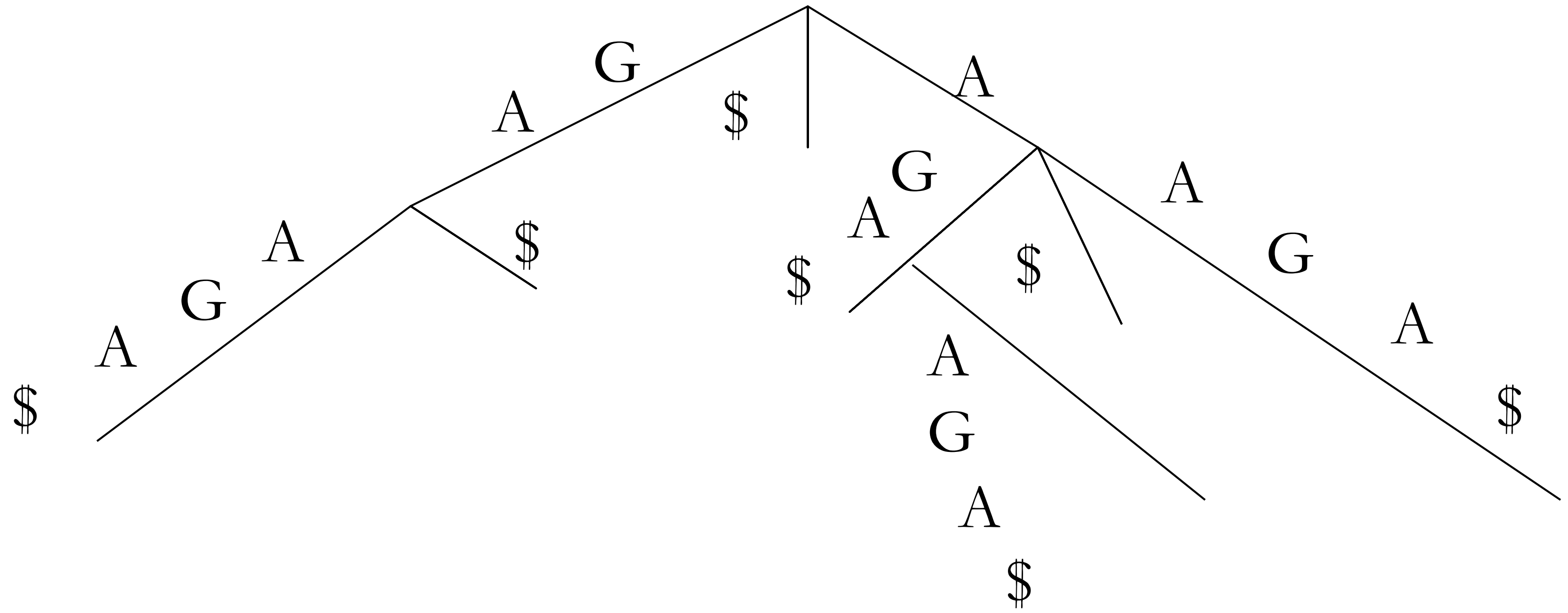
- Suppose the following is the suffix tree for GAAGA$, add another suffix AGAAGA$.



- First, follow the edges for A and for GA from the root.
- Then split after the A since the only path in the tree is for $, and we have an A, instead.
- Add a new edge for AGA$.

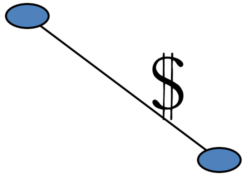
New tree

- This yields this new tree for AGAAGA$



Quadratic Time Construction

- Given: A string S of length *m* over a finite alphabet. The last character of S is a unique $ character.
- We'll build the suffix tree from right to left.
  - S[m..m], S[m-1..m], S[m-2..m], ……
- Begin with this tree:



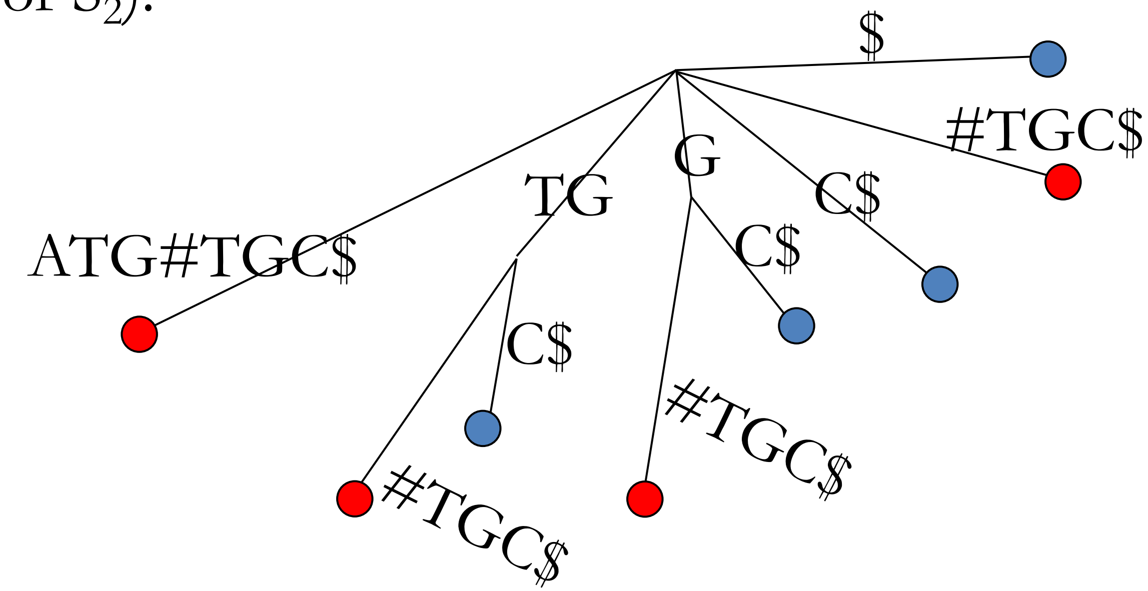
- Then, for $i = m$ downto 1:
- Follow the letters of S[*i*…*m*] along the edges of the tree T.
- When we reach a point where no path exists, break the current edge and add a new edge for what is left.
- Time complexity: $O(m^2)$. (Remember: The best algorithm has linear time.)

Application II: Longest Common Substring

- What's the longest substring common to both S_1 and S_2?
- Straightforward algorithm will try to compare all substrings of equal length. This takes cubic time.
- Can we do better?

Longest Common Substring with Suffix Tree

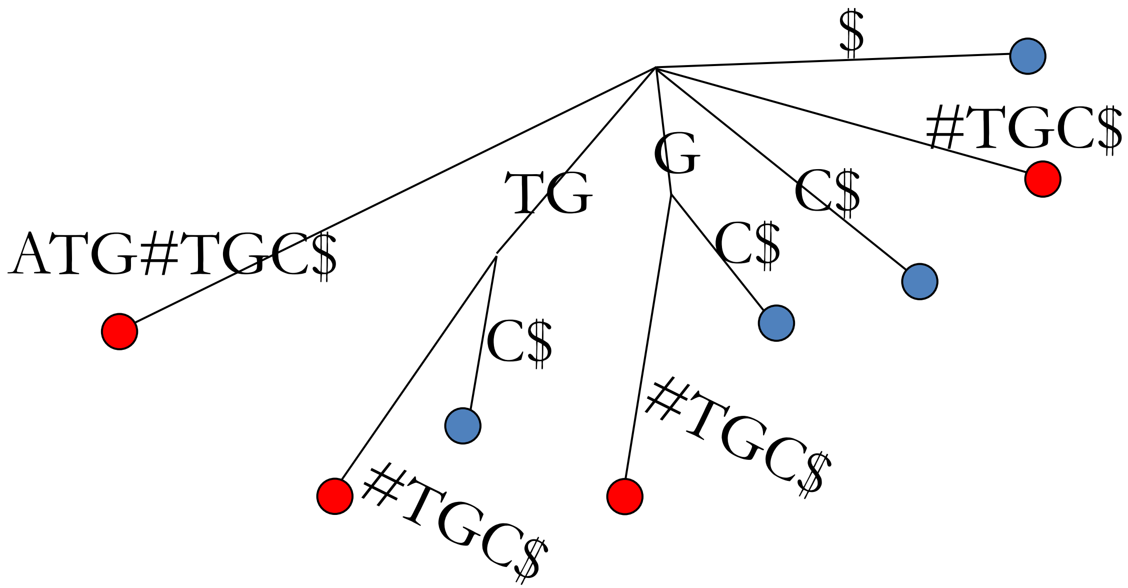
- Build a suffix tree for $S=S_1\#S_2\$$, where # and $ are unique characters.
- All suffixes of S_1 end with an edge including $\#S_2\$$. So we can label whether a leaf belongs to S_1 or S_2
- Substrings are prefixes of suffixes, i.e. internal and leaf nodes of the tree.
- Each common substring is the prefix of at least two suffixes, each from an input string (S_1 or S_2).
- Longest?



Example

ATG#TGC$

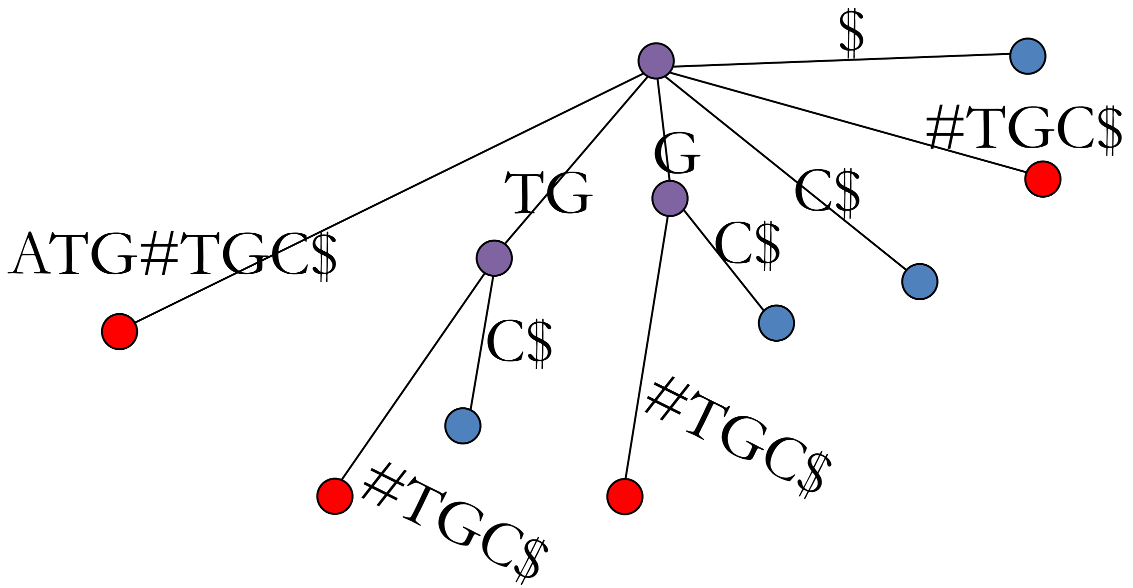
Step 1. Label leaves as red or blue, depending on whether it is a suffix starting in first or second string.



Example

ATG#TGC$

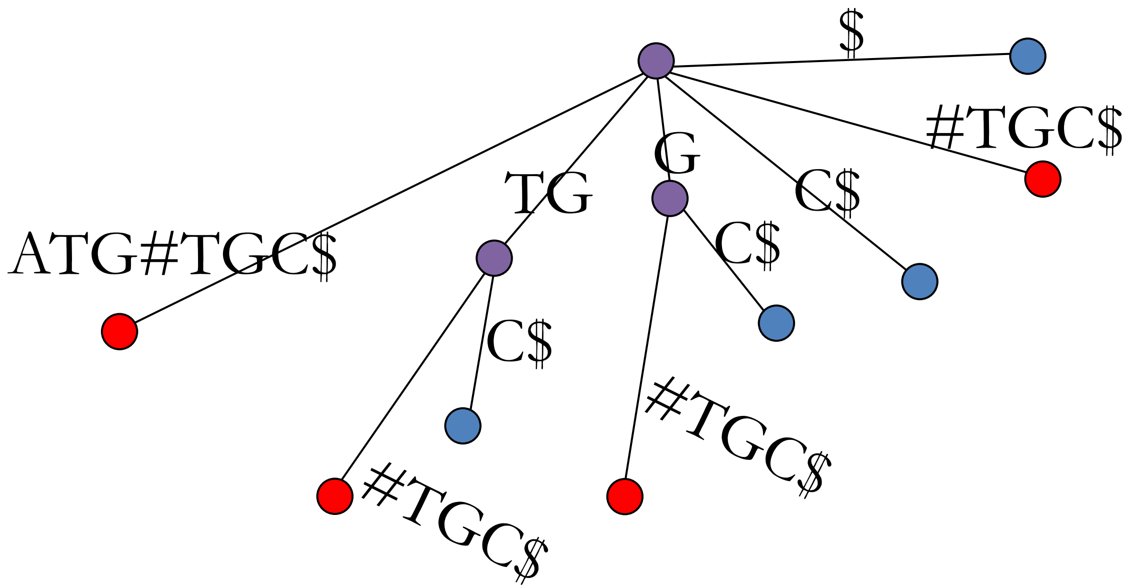
Step 2. In a bottom up order, label internal nodes. If all child nodes have the same color, label it with the same color; If not, label it with purple.



Example

ATG#TGC$

Step 3. Find the purple node with the longest path to the root.



Algorithm Summary

- 1. Build suffix tree of $S_1\#S_2\$$
- 2. Color all leaf nodes
  - red if *v*'s label is a substring of S_1
  - blue if it's a substring of S_2
- 3. Color all internal nodes from bottom up
  - red (or blue) if all child nodes are red (or blue)
  - purple if otherwise
- 4. Find the purple node with longest path label.
- Complexity: Linear time, linear space.
- Sketch proof of correctness:
  - Let t be the longest common substring. Follow the path label t starting from the root. The path can't stop in the middle of the edge – otherwise t is not the longest. Then the path has to stop at an internal node. And it has to be purple.

Application III: Maximal Unique Match

Maximal Unique Matches

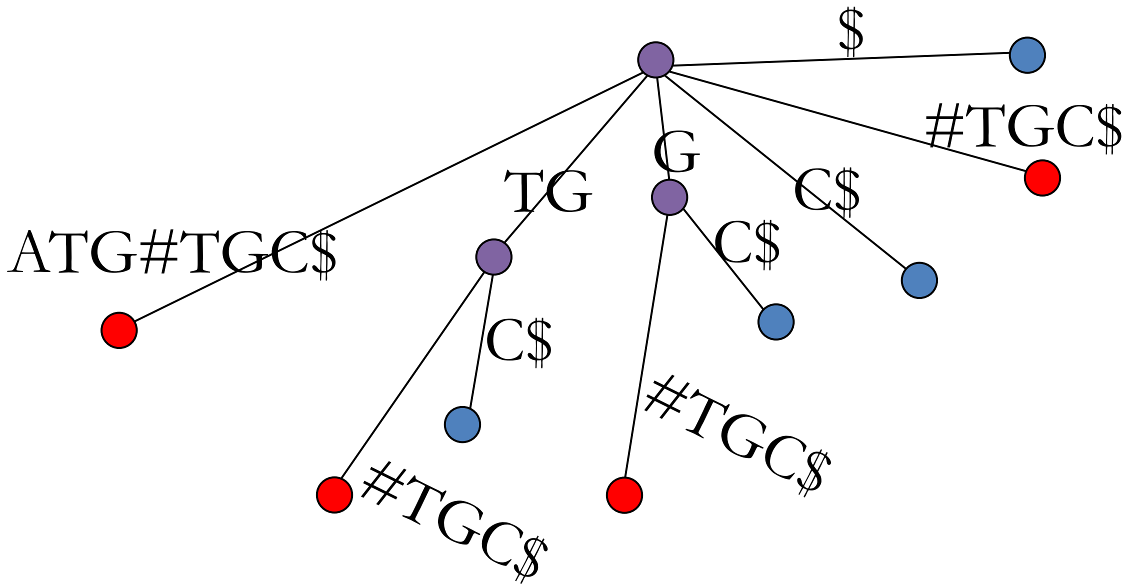
- Given two strings, a MUM (Maximal Unique Match) is a string that occurs exactly once in each string, and is maximal (can't be extended either way and still be a match).
- E.g. ATGAATC vs. AGATC
  - AT is not.
  - G is not.
  - GA is a MUM.
  - ATC is a mum.

How to find mums?

- Build a suffix tree for $S_1\#S_2\$$
- Color the nodes as in the longest common substring algorithm.
- Each MUM must be a purple internal node that has exactly two leaf children: one red and one blue.
  - It is shared by the two strings.
  - It can't extend to the right by an additional letter and still be shared.
  - It must be unique.

Example:

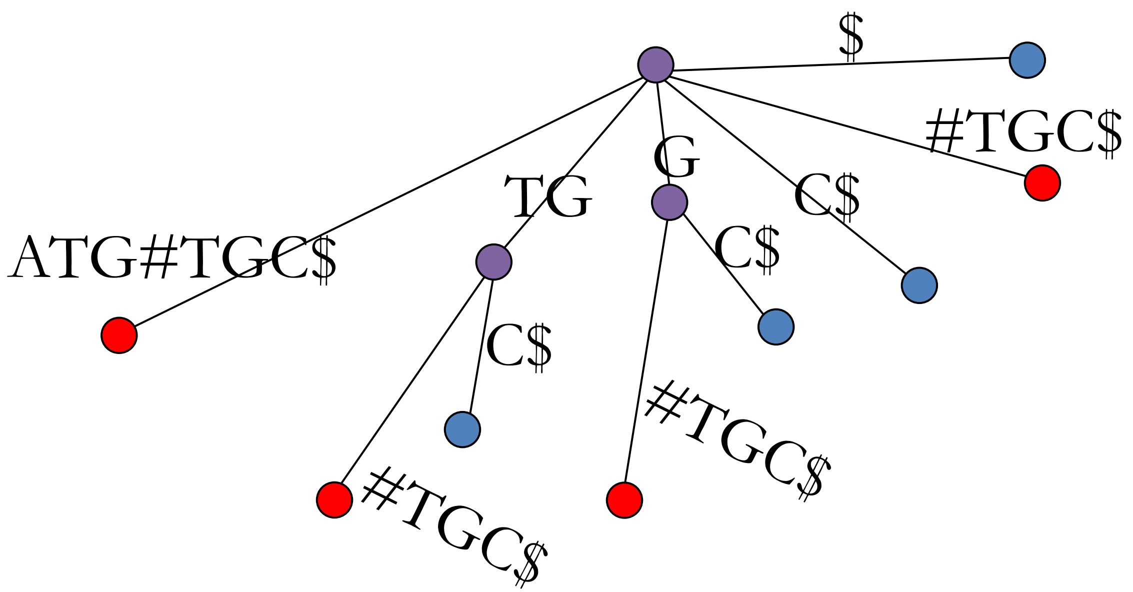
ATG#TGC$



How to find mums?

- But a purple internal node may not be a MUM: only because the two occurrences may still extend to the left.
  - Node G is not: For G's two occurrences, the left character are both T.
  - Node TG is: For TG's two occurrences, the left characters are A and #, respectively.
- But it is easy to compute the left character of each leaf
  - It is a suffix, and we know its path's starting position in the original string.

Example:
ATG#TGC$



Summary

- Build a suffix tree for $S_1\#S_2\$$.
- For each leaf v, define left(v) be the letter at left of suffix v.
- Find the internal nodes that
  - Have exactly two child leaves
  - The two child leaves are two suffixes from S1 and from S2, respectively.
  - The two child leaves must have two different left characters.
- Linear time.
- After find all MUMs, use them as anchor to speed up global alignment.

MUMMER: Large-scale Global Alignment

- Large-scale global alignment

- Idea:
- Pick some "anchors" through which the true alignment is very likely to fall.
- Align the regions between the anchors either recursively or just using classical global alignment tools.

- MUMs are good anchors: maximal, unique, match.
- First program that does so: MUMMER by Delcher et al.

Quick Note on Suffix Array

- Suffix tree is not a compact data structure.
  - A lot of pointers
- Gene Myers and Udi Manber (VP enginnering, Google) proposed suffix array.
- A suffix array stores the positions in a string. Each position is an integer so this is a length n integer array.
- Each position corresponds to a suffix starting at this position.
- The suffix array is sorted according to the string order of the corresponding suffixes.

Suffix Array

- AGAAGAT

1 = AGAAGAT
2 = GAAGAT
3 = AAGAT
4 = AGAT
5 = GAT
6 = AT
7 = T

3 = AAGAT
1 = AGAAGAT
4 = AGAT
6 = AT
2 = GAAGAT
5 = GAT
7 = T

3, 1, 4, 6, 2, 5, 7

String Matching

- Binary search to find substring of length m.
  - $O(m \log n)$ if implemented straightforwardly
  - $O(m + \log n)$ if with an auxiliary data structure called longest common prefix (LCP) array. We do not study this but you should be aware of this fact.

Suffix Array Construction

- The construction of suffix array is also referred to as suffix sorting, which can be done in linear time.
  - LCP array also takes linear time to construct
- We only learn one of the linear time suffix sorting algorithms.

Skew Algorithm For Suffix Sorting

- Let S_0, S_1, S_2, …, S_{n-1} be all the n suffixes. S_i starts at i-th position.
- Skew algorithm uses divide and conquer. But it divides the problem into unequally sized parts.
- Two sets $SA^0 = \{S_i : i = 0 \bmod 3\}$ and $SA^{12} = \{S_i : i = 1 \text{ or } 2 \bmod 3\}$.

Skew Algorithm Example

- Example: mississippi

SA^0

```
mississippi
  sissippi
     sippi
        pi
```

SA^{12}

```
ississippi     ssissippi
   issippi        ssippi
      ippi           ppi
         i
```

Skew Algorithm For Suffix Sorting

- Plan:
  - 1. Sort SA^{12} recursively.
  - 2. Sort SA^{0} with the help of the sorted SA^{12}.
  - 3. Merge sort SA^{0} and SA^{12}.
- Our goal is to do step 2 and 3 in linear time. If this can be achieved, then the time complexity is
  - $T(n) = O(n) + T(2n/3)$.
  - This leads to $T(n)=O(n)$.
  - Compare with merge sort.

Skew Algorithm For Suffix Sorting

- 1. Sort SA^{12} recursively.
- 2. Sort SA^{0} in linear time.
- 3. Merge sort SA^{0} and SA^{12} in linear time.

How to sort SA^{12} recursively

mississippi

SA^{12}

mod 1

```
ississippi
   issippi
      ippi
         i
```

mod 2

```
ssissippi
   ssippi
      ppi
```

- We need to know the order of these suffixes.
- In order to solve it recursively, we need to reduce the problem to a smaller suffix sorting problem.

Reduction to a smaller suffix sorting problem

SA^{12}

mod 1

```
ississippi00
   issippi00
      ippi00
         i00
```

mod 2

```
ssissippi
   ssippi
      ppi
```

i s s	i s s	i p p	i 0 0

s s i	s s i	p p i

- Pad 0 to make their length multiple of 3. Then treat each string as a string of “triplets”. Each subset is the suffixes of the “triplet string”.
- We connect the two “triplet strings” together to make a longer string. We put the one with padding at the left.

i s s	i s s	i p p	i 0 0	s s i	s s i	p p i

Reduction

SA^{12}

mod 1

```
ississippi00
   issippi00
      ippi00
         i00
```

mod 2

```
ssissippi
   ssippi
      ppi
```

```
ississippi00ssissippi
   issippi00ssissippi
      ippi00ssissippi
         i00ssissippi
            ssissippi
               ssippi
                  ppi
```

- Now check all the suffixes of the concatenated triplet string. Their relative order can be used to build the relative order of SA^{12} easily.
- We are almost there, except that keeping tripling the size (number of bytes) of the "character" is a problem.

Renaming

- We solve the unlimited expansion problem by a trick called renaming. It maps each unique triplet to a single unique integer.
- To rename, we first sort the triplets, and then assign integer values sequentially to unique triplets. Sorting triplets can be done in linear time by radix sort.
- This ensures
  - The max value is always bounded by the length of array.
  - The suffix order is unchanged.

```
i00 -> 0
ipp -> 1
iss -> 2
ppi -> 3
ssi -> 4
```

```
ississippi00ssissippi

  2  2  1  0  4  4  3
```

Renaming Example

```
i00  ->  0
ipp  ->  1
iss  ->  2
ppi  ->  3
ssi  ->  4
```

```
ississippi00ssissippi
```

```
2  2  1  0  4  4  3
```

```
ississippi00ssissippi        2 2 1 0 4 4 3
   issippi00ssissippi          2 1 0 4 4 3
      ippi00ssissippi            1 0 4 4 3
         i00ssissippi              0 4 4 3
            ssissippi                4 4 3
               ssippi                  4 3
                  ppi                    3
```

Recursion

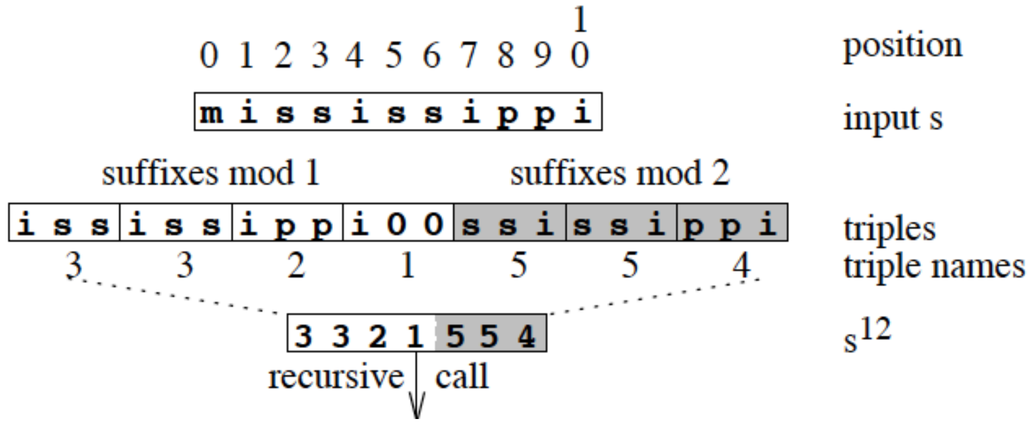
- After renaming, we just suffix sort the new integer string, which has length approximately 2n/3. This can be done by recursion.
- The time complexity of renaming is dominated by sorting the triplets. This can be solved in linear time with radix sort.

Radix Sort

- **Radix Sort**: Multiple passes. Each pass stable sorts according to one digit. From the least to the most significant digit.
- original: its, iss, ipp, abc, att
- pass1: abc, ipp, its, iss, att
- pass2: abc, ipp, iss, its, att
- pass3: abc, att, ipp, iss, its

- Radix sorting requires O(k) space, where k is the size of the alphabet.
- Each pass takes linear time. And only 3 passes needed in our case. So it is linear time.

Recap Sort S^{12} recursively



1. Padding and concatenation to get string of triplets.
2. Radix sort the triplets to get an ID (name) of each triple.
3. Recursion to get the suffix order on the string of IDs.

Skew Algorithm For Suffix Sorting

- We assume SA^{12} is sorted already, and learn the other two steps first.
- 1. Sort SA^{12} recursively.
- 2. Sort SA^{0} in linear time.
- 3. Merge sort SA^{0} and SA^{12} in linear time.

Sort S^0 in linear time

- $S_i = s[i]\ S_{i+1}$.
- For all S_i in SA^0, S_{i+1} has been sorted already. Use s[i] to do another pass of radix sorting will give us the right order of SA^0. This takes linear time.

Sorted SA12

```
10: i
 4: issippi
 1: ississippi
 7: ippi
 8: ppi
 5: ssippi
 2: ssissippi
```

0	1	2	3	4	5	6	7	8	9	10
m	i	s	s	i	s	s	i	p	p	i

To sort SA0

```
0: mississippi
3: sissippi
6: sippi
9: pi
```

Skew Algorithm For Suffix Sorting

- 1. Sort SA^{12} recursively.
- 2. Sort SA^{0} in linear time.
- 3. Merge sort SA^{0} and SA^{12} in linear time.

Merge

Sorted SA12

```
10: i
 4: issippi
 1: ississippi
 7: ippi
 8: ppi
 5: ssippi
 2: ssissippi
```

Sorted SA0

```
0: mississippi
9: pi
6: sippi
3: sissippi
```

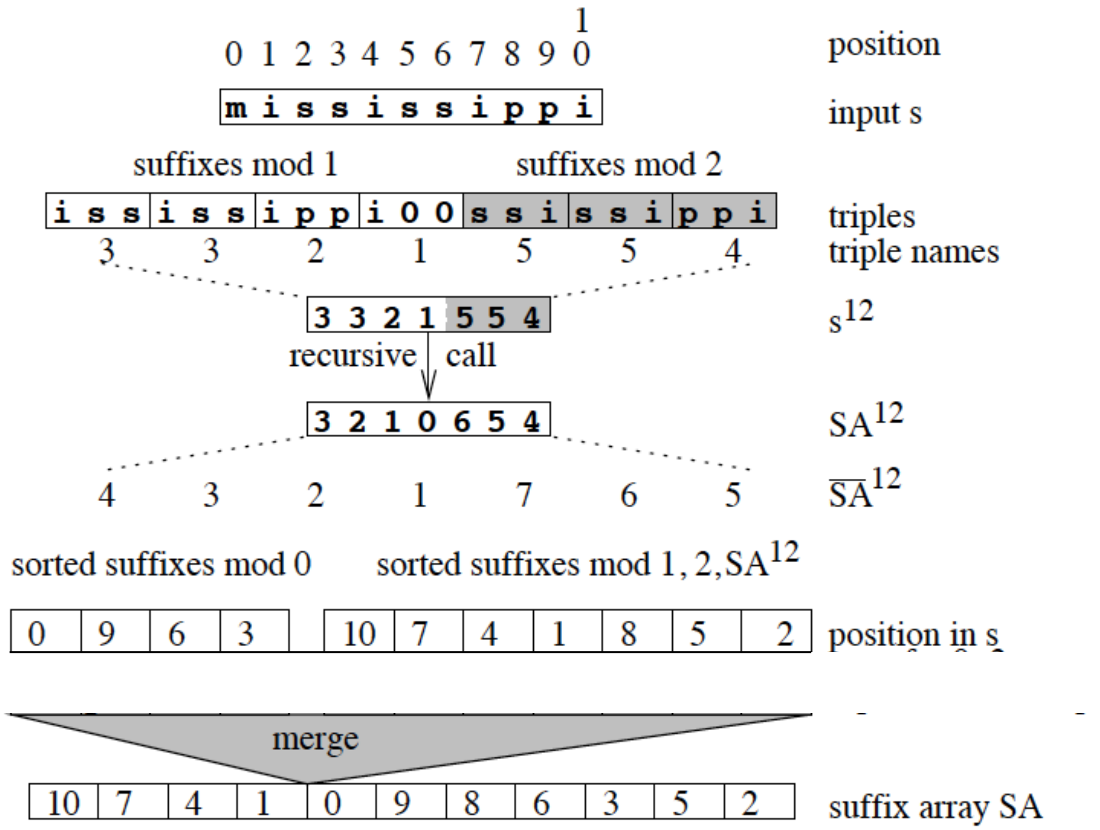
- Would be a simple merge if comparison of two takes constant time.
- Trouble is when two suffices share a long prefix, which takes more than constant time to compare.

E.g. what if S5 = aaaa… and S6=aaaa…

Merge S^0 and S^{12}

- Merging only requires to compare a suffix S_j with j mod 3 = 0 with a suffix S_i with i mod 3 != 0. :
- Case 1: If i mod 3 = 1, we write S_i as (s[i], S_{i+1}) and S_j as (s[j], S_{j+1}).
  - Since (i +1) mod 3 = 2 and (j + 1) mod 3 = 1, the relative order of S_{j+1} and S_{i+1} can be determinded from their position in SA^{12}.
- Case 2: If i mod 3 = 2, we compare the triples (s[i], s[i + 1], S_{i+2}) and (s[j], s[j + 1], S_{j+2}).

Recap


position
0 1 2 3 4 5 6 7 8 9 10
m i s s i s s i p p i
input s
suffixes mod 1
suffixes mod 2
i s s i s s i p p i 0 0 s s i s s i p p i
triples
3 3 2 1 5 5 4
triple names
3 3 2 1 5 5 4
s^{12}
recursive call
3 2 1 0 6 5 4
SA^{12}
4 3 2 1 7 6 5
$\overline{SA}^{12}$
sorted suffixes mod 0
sorted suffixes mod 1, 2, SA^{12}
0 9 6 3
10 7 4 1 8 5 2
position in s
merge
10 7 4 1 0 9 8 6 3 5 2
suffix array SA

C codes

- 50 lines of C++ codes were given in J.C.M. Baeten et al. (Eds.): ICALP 2003, LNCS 2719, pp. 943–955, 2003.

- http://www.mpi-inf.mpg.de/~sanders/programs/suffix/