Genome Alignment

From Sung §4.1-4.4,§4.6

Genome Alignment

Let's assume (though we have not gotten to the details of how to do it yet) that we have both my and Dr. Salamah's genome and we want to find the differences. What do we do?

- We know that we can use Smith-Waterman
- Lets assume we have it as two 3×10^9 base sequences:
  - The computation time and memory are on the order of 9×10^{18}
  - Computing each cell of the table would take 52 days*
  - The table would need 250 XB to store**
  - Then there is traceback....

* assuming 2GHz, 1 core, and that each cell takes only one cycle
** as integers with no overhead

Genome Alignment

One other caveat:

- That would only account for SNPs and indels
- As we discussed theres also structural changes that wouldn't be found

Genome Alignment

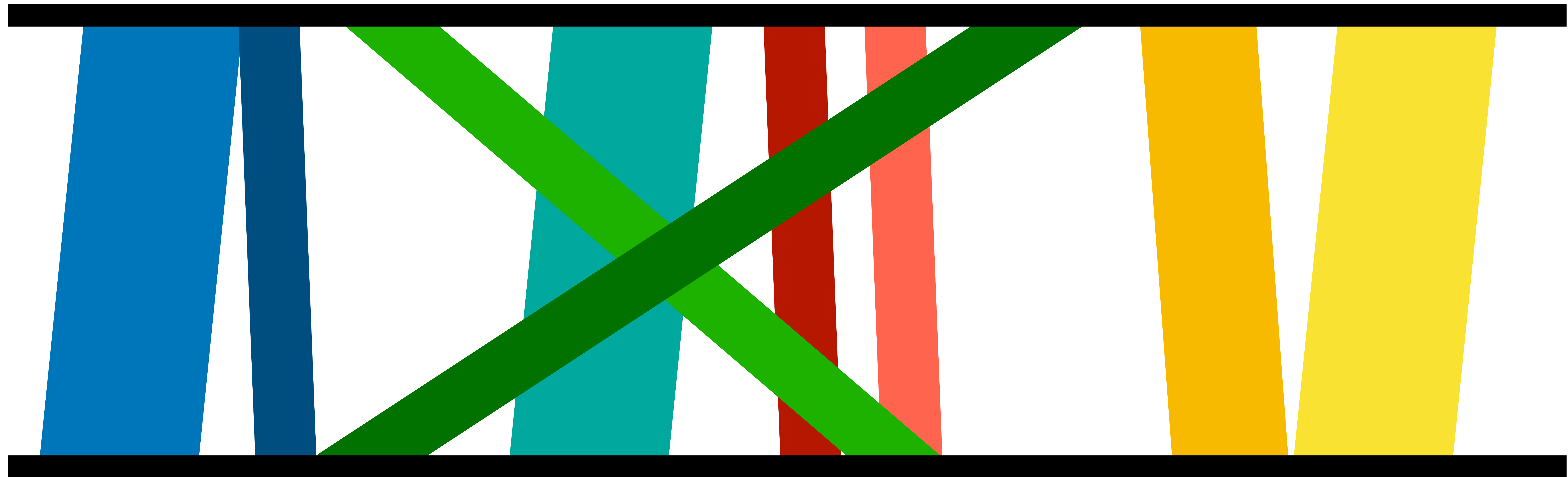
So what do we do?

Well, we know genomes are **very** similar, so they will share regions of highly similar sequence.

Most specialized tools for this problem follow 3 basic steps:

1. **Identify potential anchor points** -- ideally these points can be found quickly and will provide a limited number of locations to investigate further
2. **Identify groups of anchors that are *co-linear* and *non-overlapping*** -- these will provide further evidence that there is a region that should be aligned.
3. **Close any gaps between anchors** -- this will complete the entire alignment if the anchors cover the whole sequence.

Genome Alignment

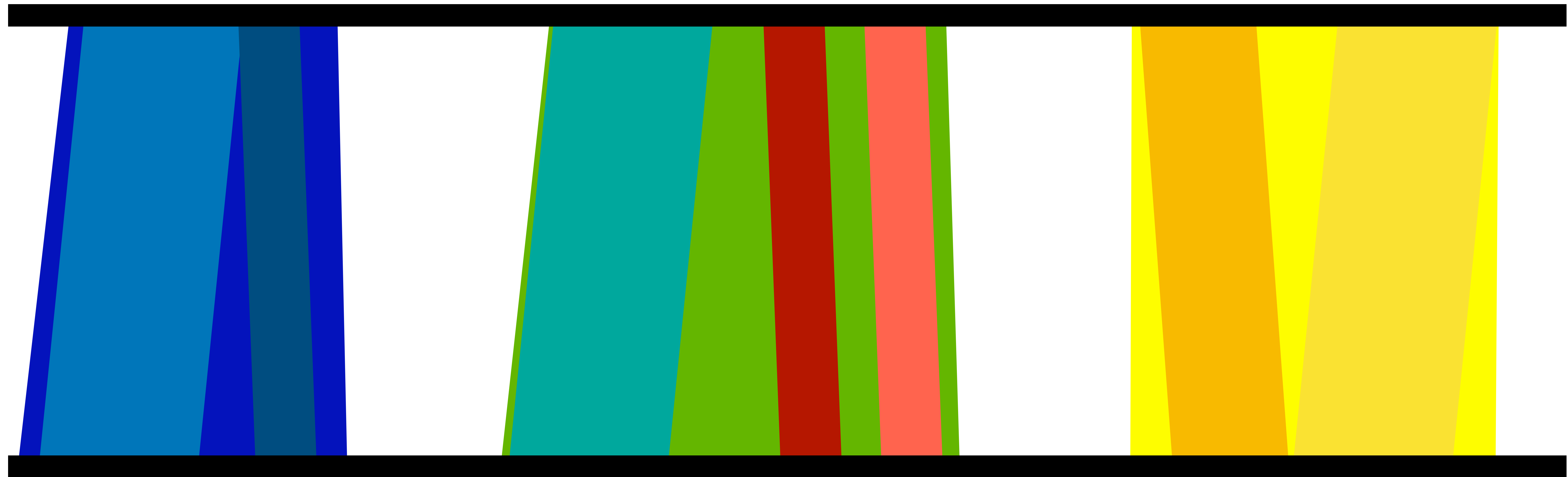


1. **Identify potential anchor points**

Genome Alignment

2.**Identify groups of anchors that are *co-linear* and *non-overlapping***

Genome Alignment



3. **Close any gaps between anchors**

Genome Alignment

3.**Close any gaps between anchors**

Genome Alignment

Lets start with what an anchor is!

- The simplest version is the *k-mer*
- Basically you chop both sequences into all of its k length substrings (overlapping)
- You use all locations with the same k-mer as a potential anchor
- Its fast normally since you can use hashing to do this lookup (we will talk about this later)
- But has a tradeoff: small k gives lots of matches but its faster, large k is slow but more precise to a point.

Maximal Unique Matches (MUMs)

Lets break that phrase down:

- They are matches, meaning the same substring in both occur in both sequences
- They are unique, meaning the substring only occurs once in each sequence
- They are maximal, meaning the characters to the left and right don't match

Maximal Unique Matches (MUMs)

More formally:

Definition 4.1 Given two genomes *A* and *B*, a Maximal Unique Match (MUM) substring is a common substring of both *A* and *B* of length longer than a specified minimum length *d* (commonly *d=20* is used) such that

- it is maximal, that is, it cannot be extended in either direction without incurring a mismatch; and
- it is unique in both sequences.

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say $d=3$:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Maximal Unique Matches (MUMs)

An example, lets say *d=3*:

```
S = ACGACTCAGCTACTGGTCAGCTATTACTTACCGC#
T = ACTTCTCTGCTACGGTCAGCTATTCACTTACCGC$
```

Whats the algorithm we just used for that?

Maximal Unique Matches (MUMs)

Brute Force:

- for every position *i* in *A* and *j* in *B*:
  - find the **longest common prefix** (call it *P*) of *A[i....n]* and *B[j....m]*
  - check to ensure $|P| \geq d$
  - check if *P* is unique in both genomes

Maximal Unique Matches (MUMs)

Brute Force:

- for every position *i* in *A* and *j* in *B*:
    - find the **longest common prefix** (call it *P*) of *A[i....n]* and *B[j....m]*
    - check to ensure $|P| \geq d$
    - check if *P* is unique in both genomes

Maximal Unique Matches (MUMs)

Brute Force:

- for every position i in A and j in B:
  - find the **longest common prefix** (call it P) of $A[i....n]$ and $B[j....m]$
  - check to ensure $|P| \geq d$
  - check if P is unique in both genomes

What tools do we have that we could use?

Maximal Unique Matches (MUMs)

Using suffix trees:

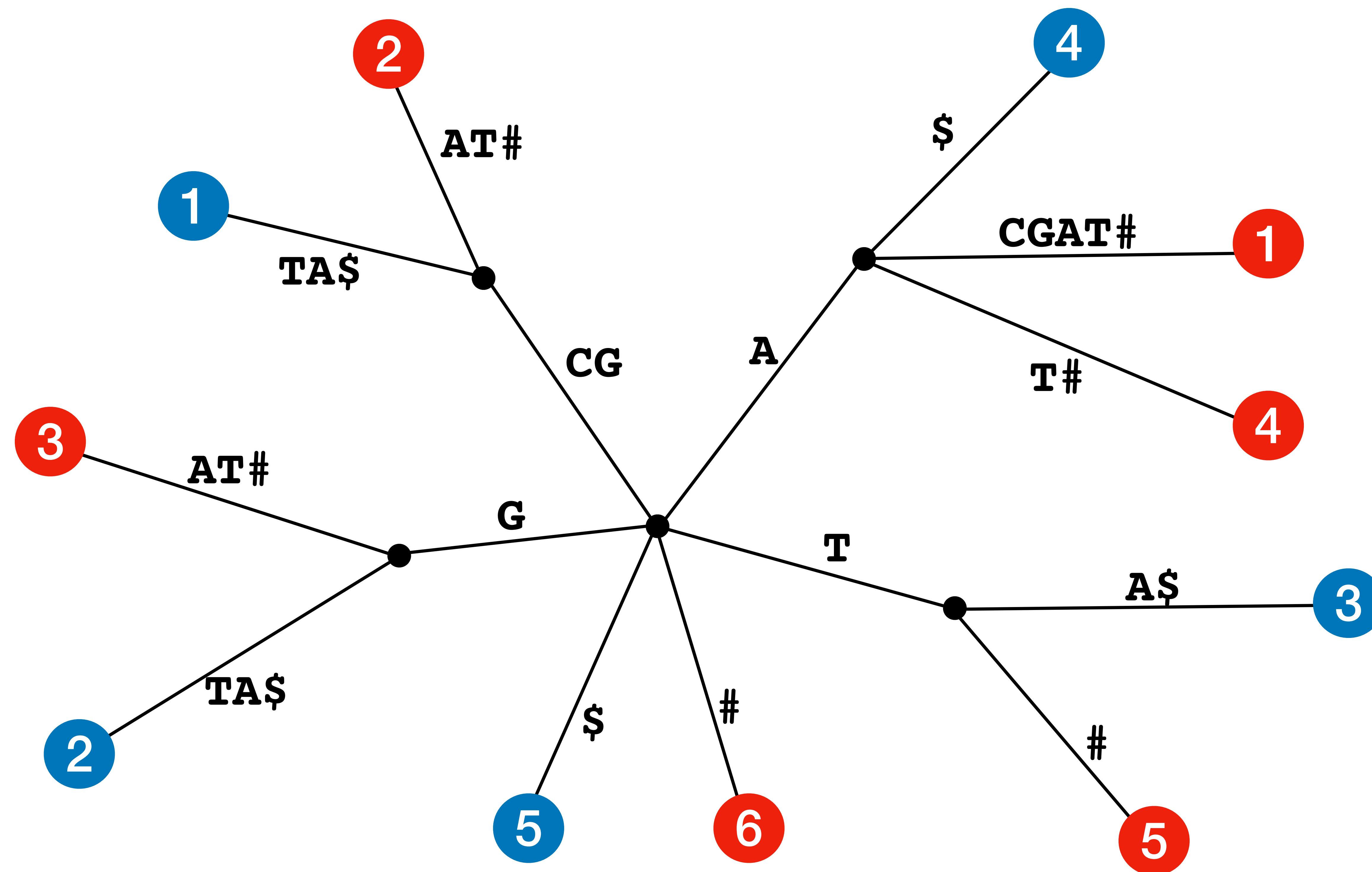
- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if *A[i-1] ≠ B[j-1]*, if so report as a MUM

d = 1



S = ACGAT#

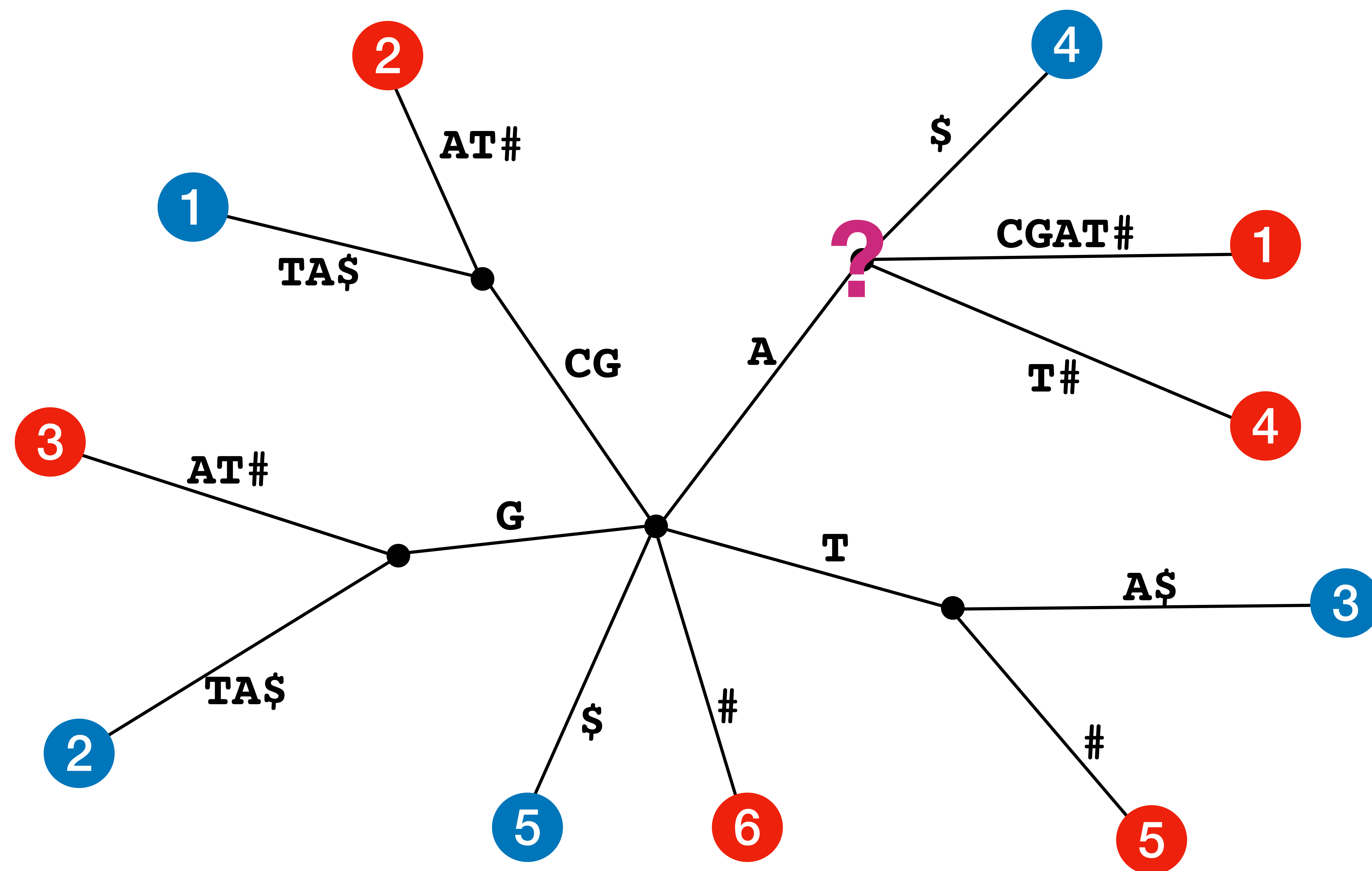
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

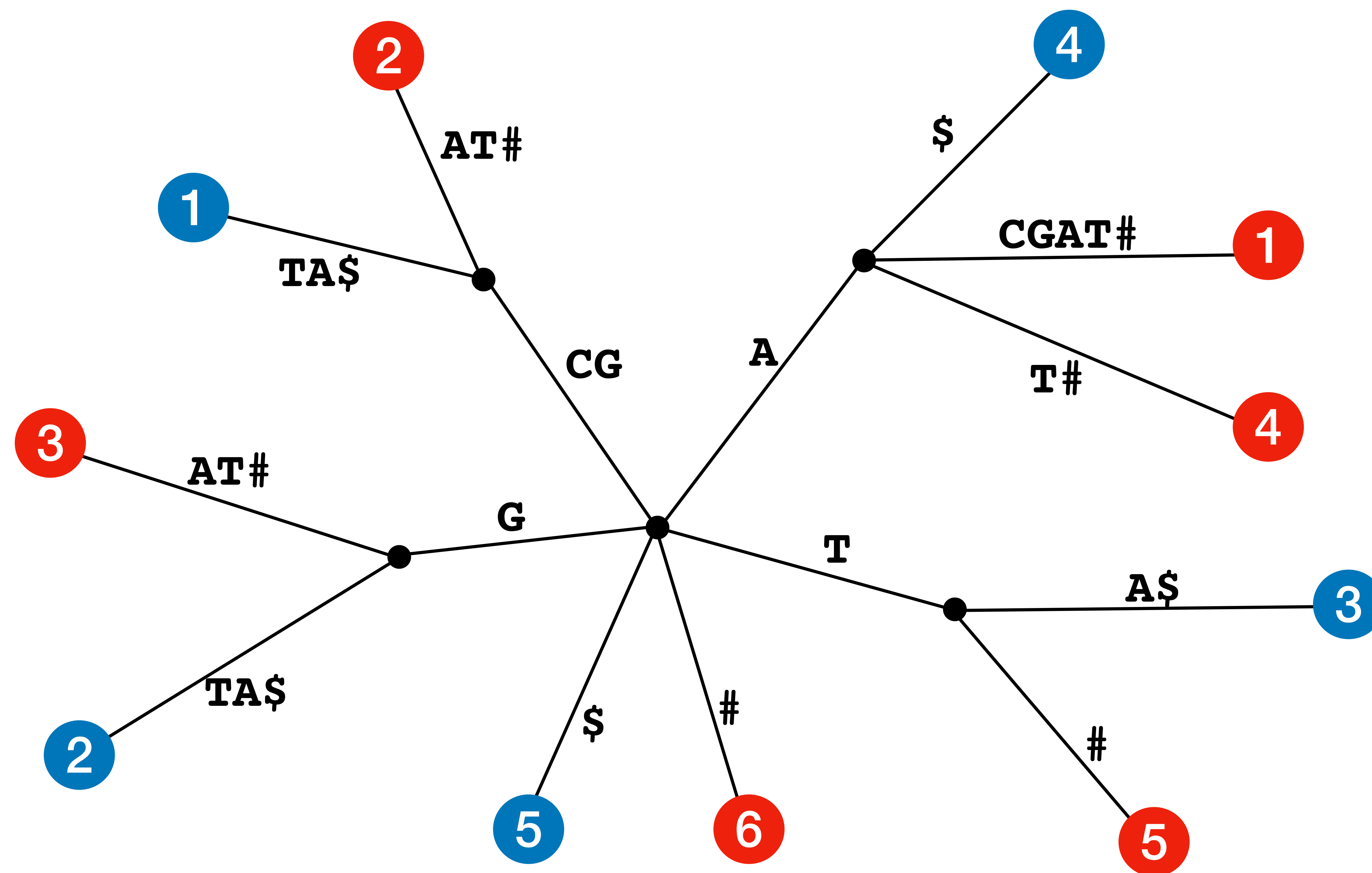
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if *A[i-1] ≠ B[j-1],* if so report as a MUM

d = 1



S = ACGAT#

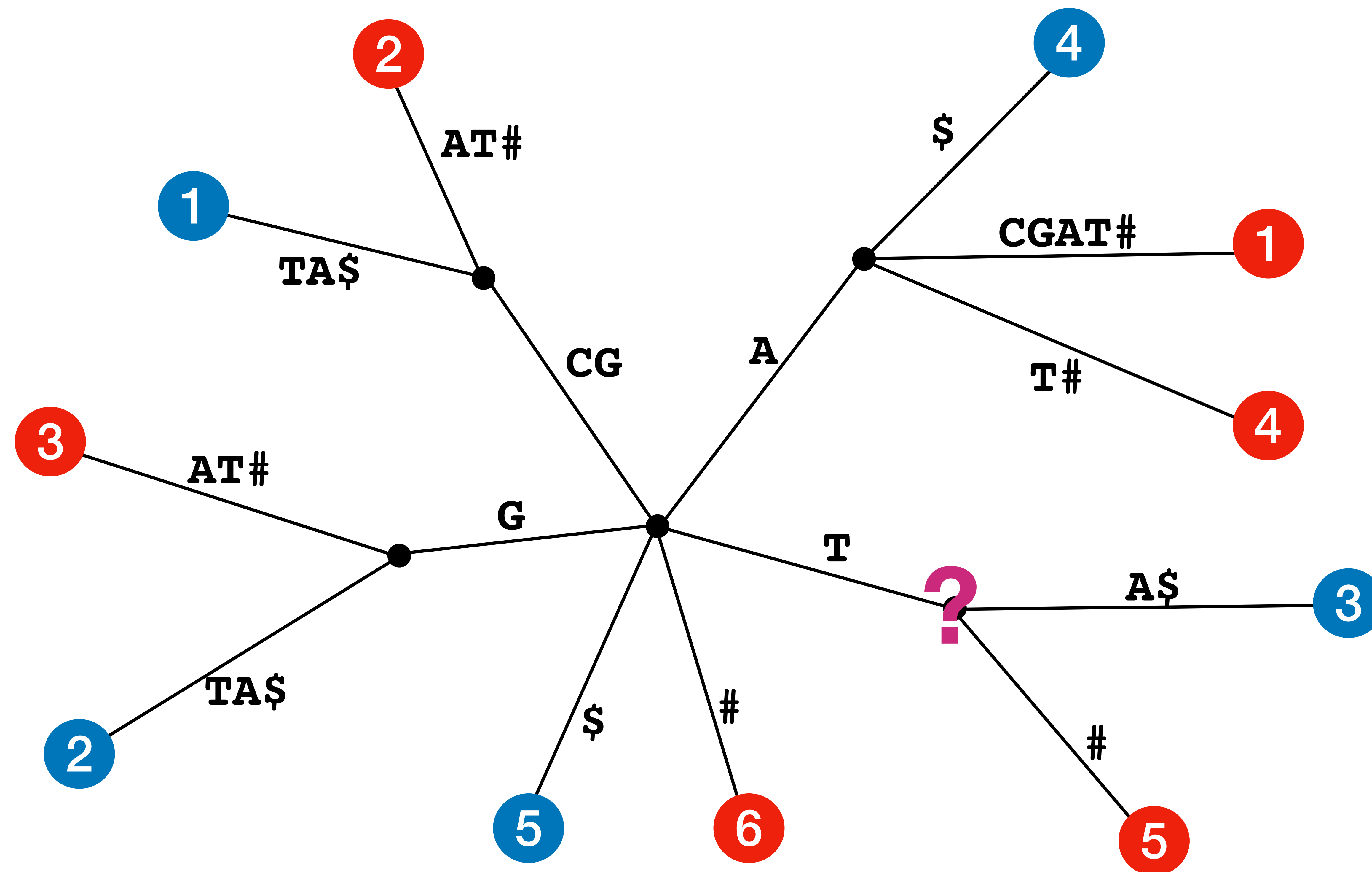
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

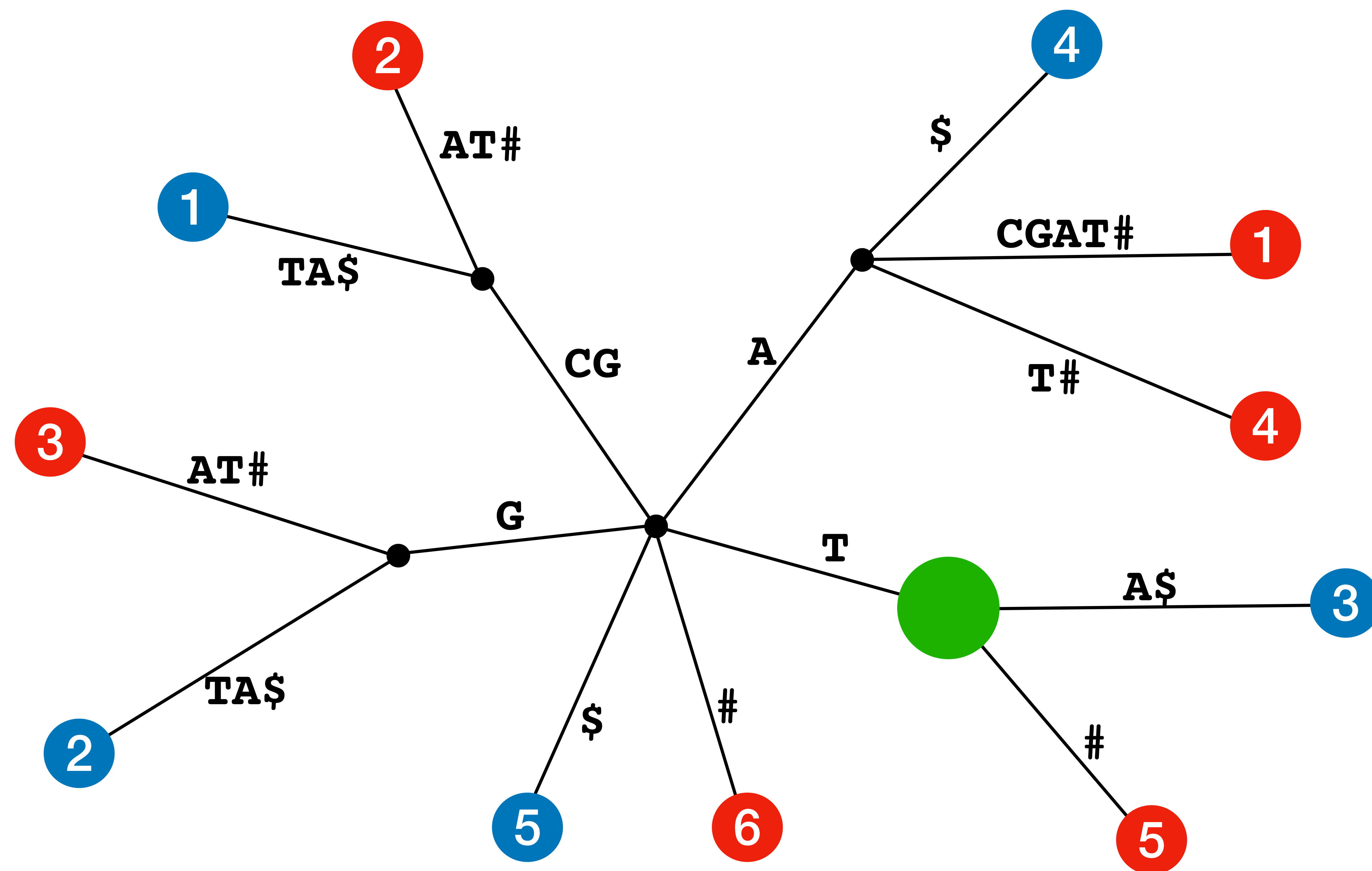
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i\text{-}1] \neq B[j\text{-}1]$, if so report as a MUM

d = 1



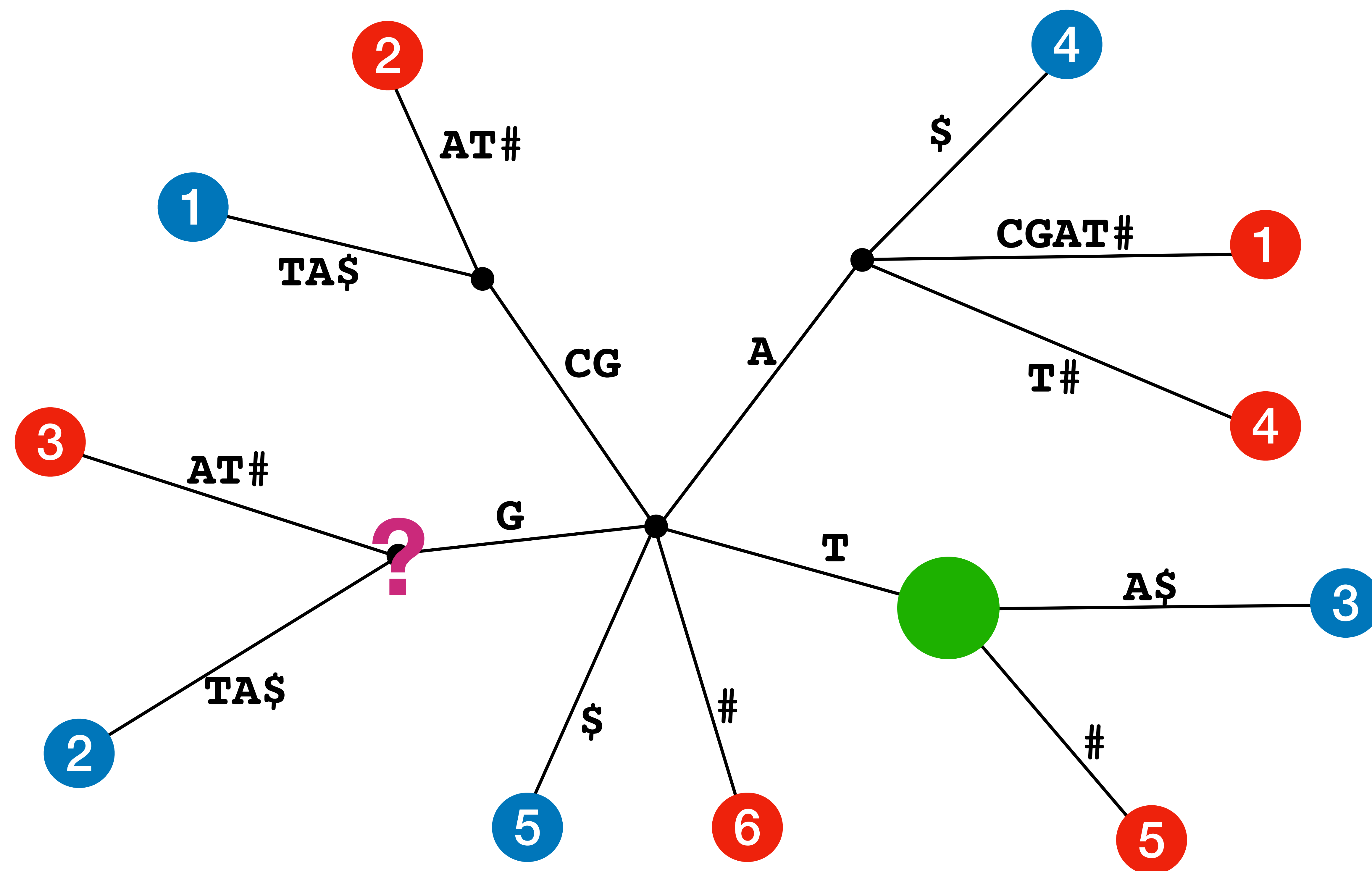
S = ACGAT#
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if *A[i-1] ≠ B[j-1]*, if so report as a MUM

d = 1



S = ACGAT#

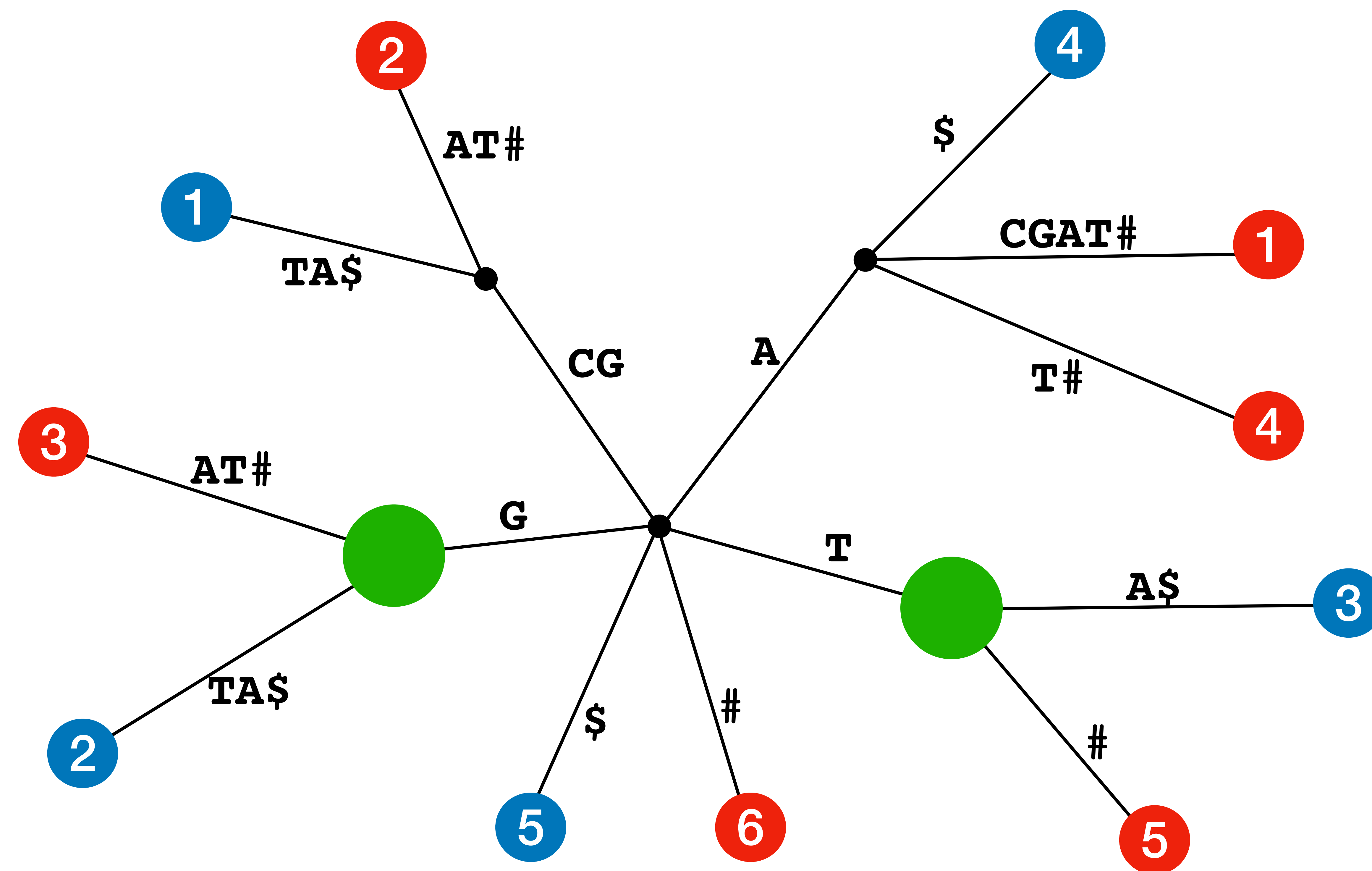
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

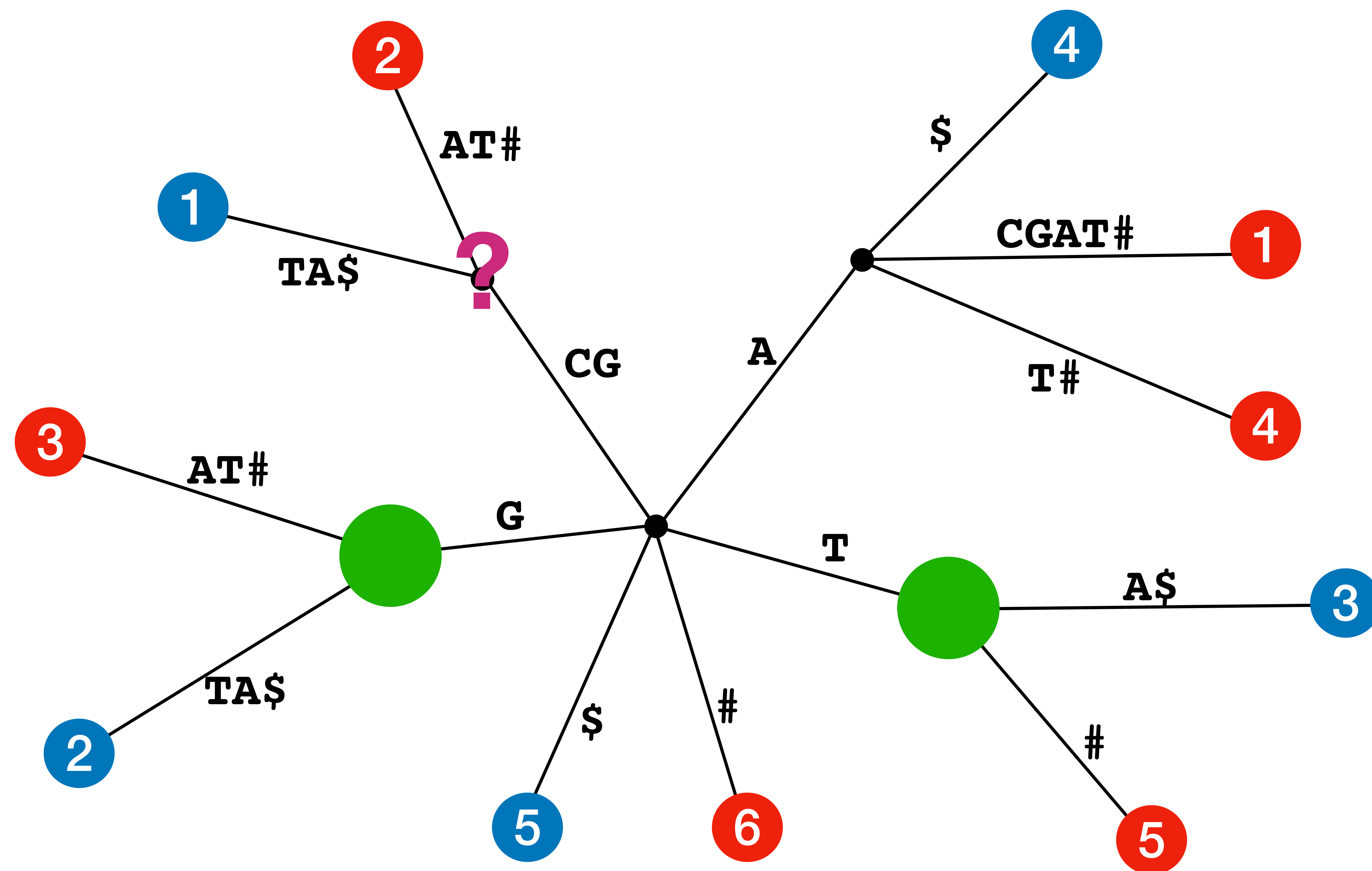
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i\text{-}1] \neq B[j\text{-}1]$, if so report as a MUM

d = 1



S = ACGAT#

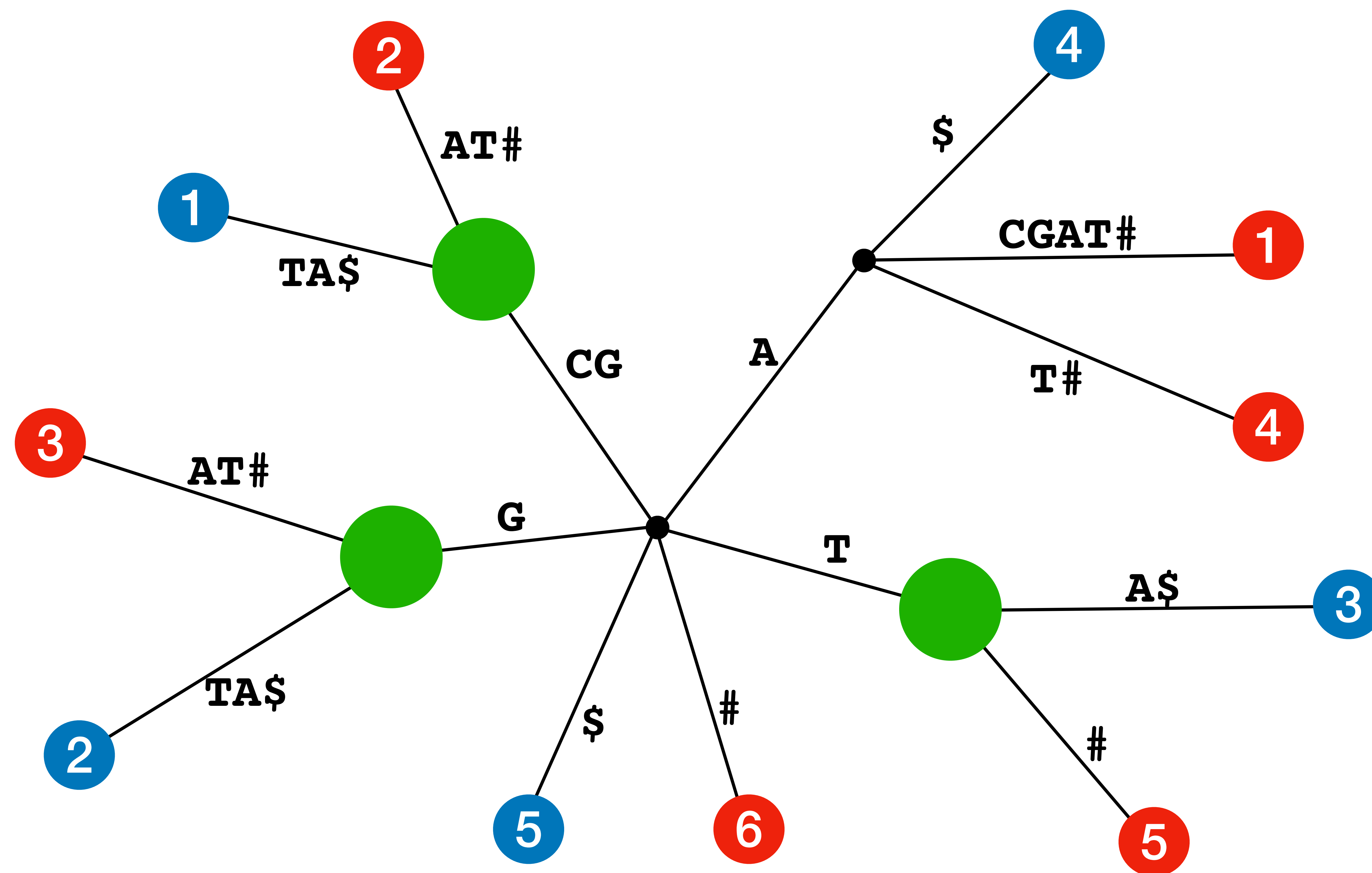
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



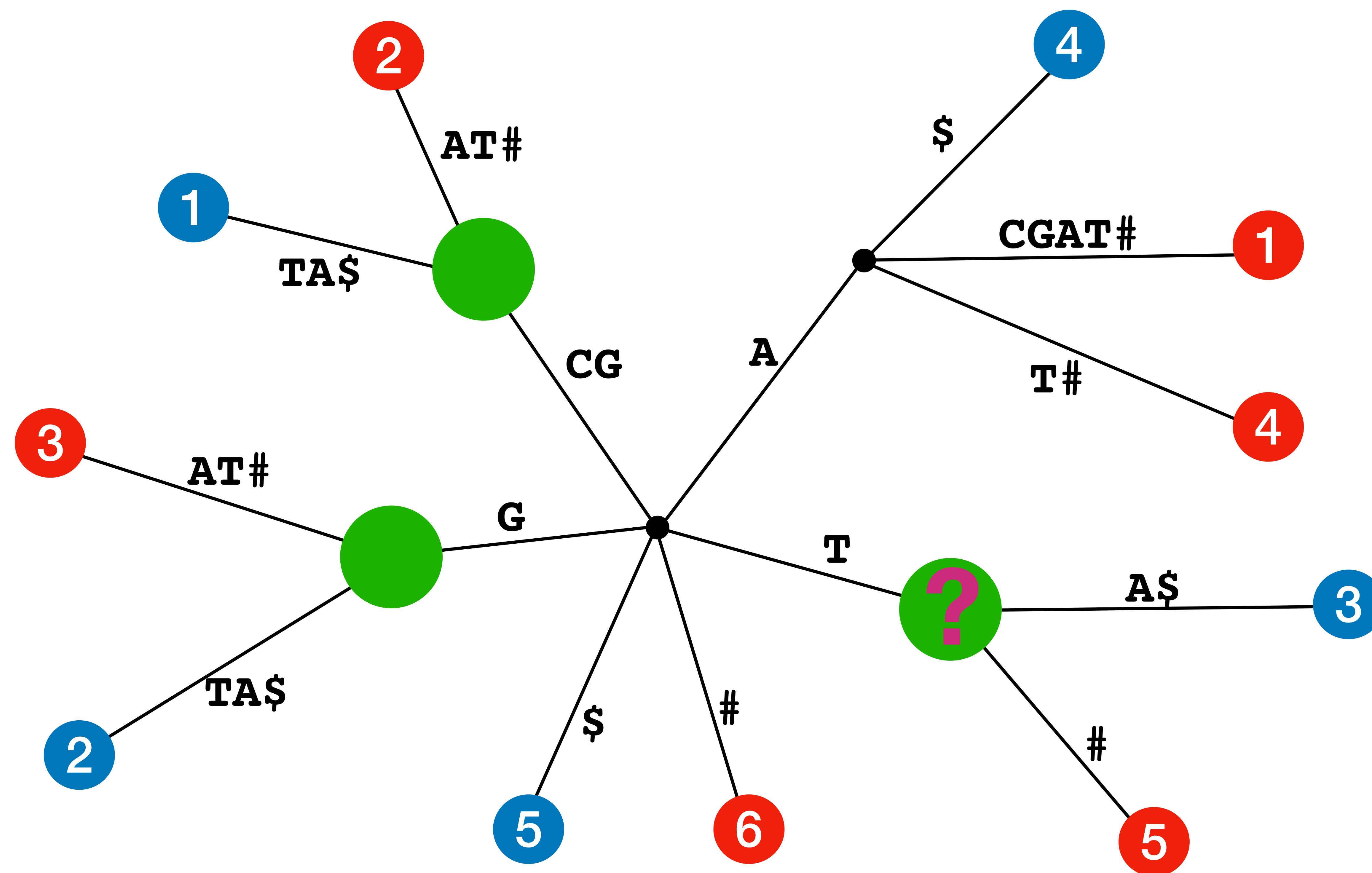
S = ACGAT#
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

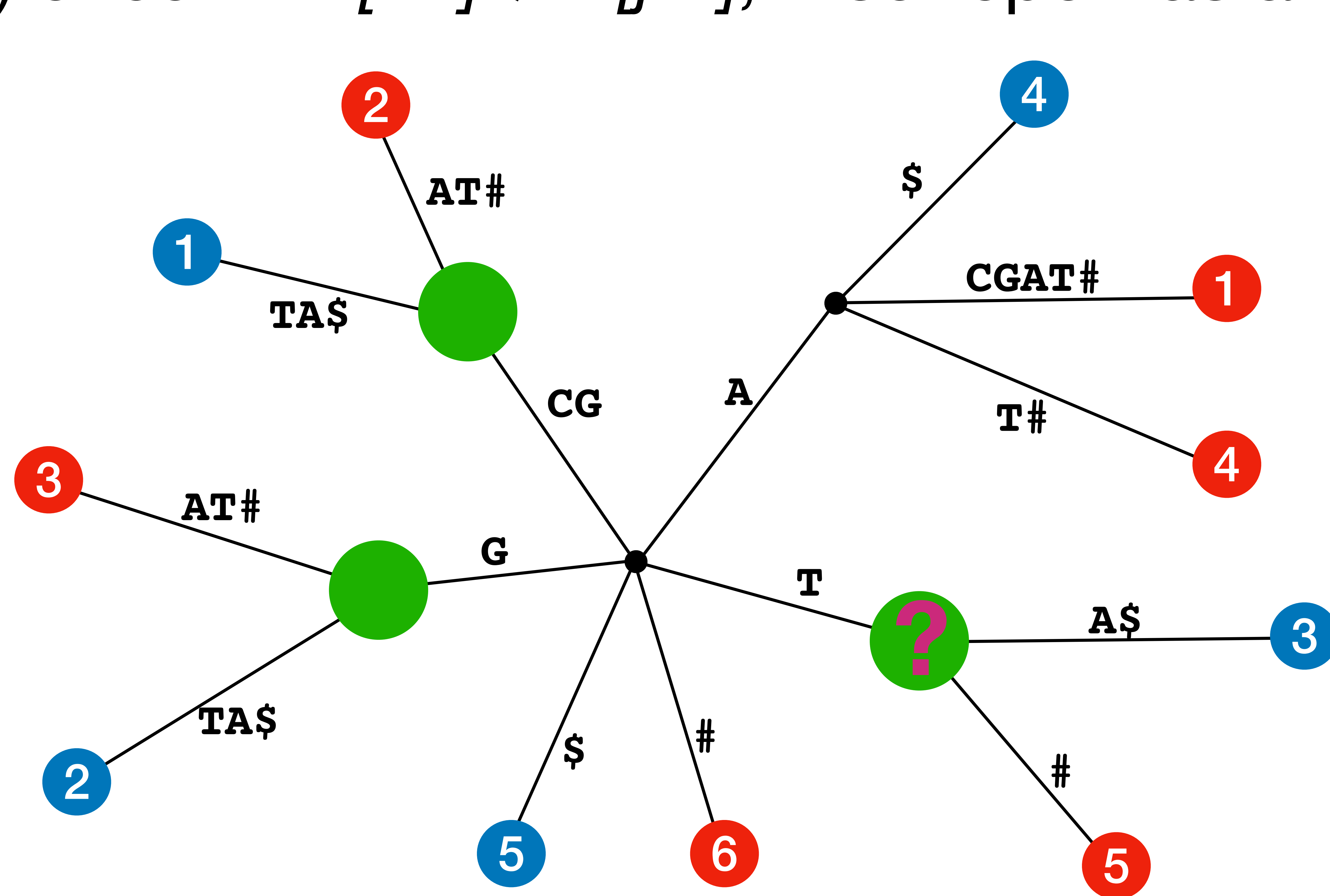
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

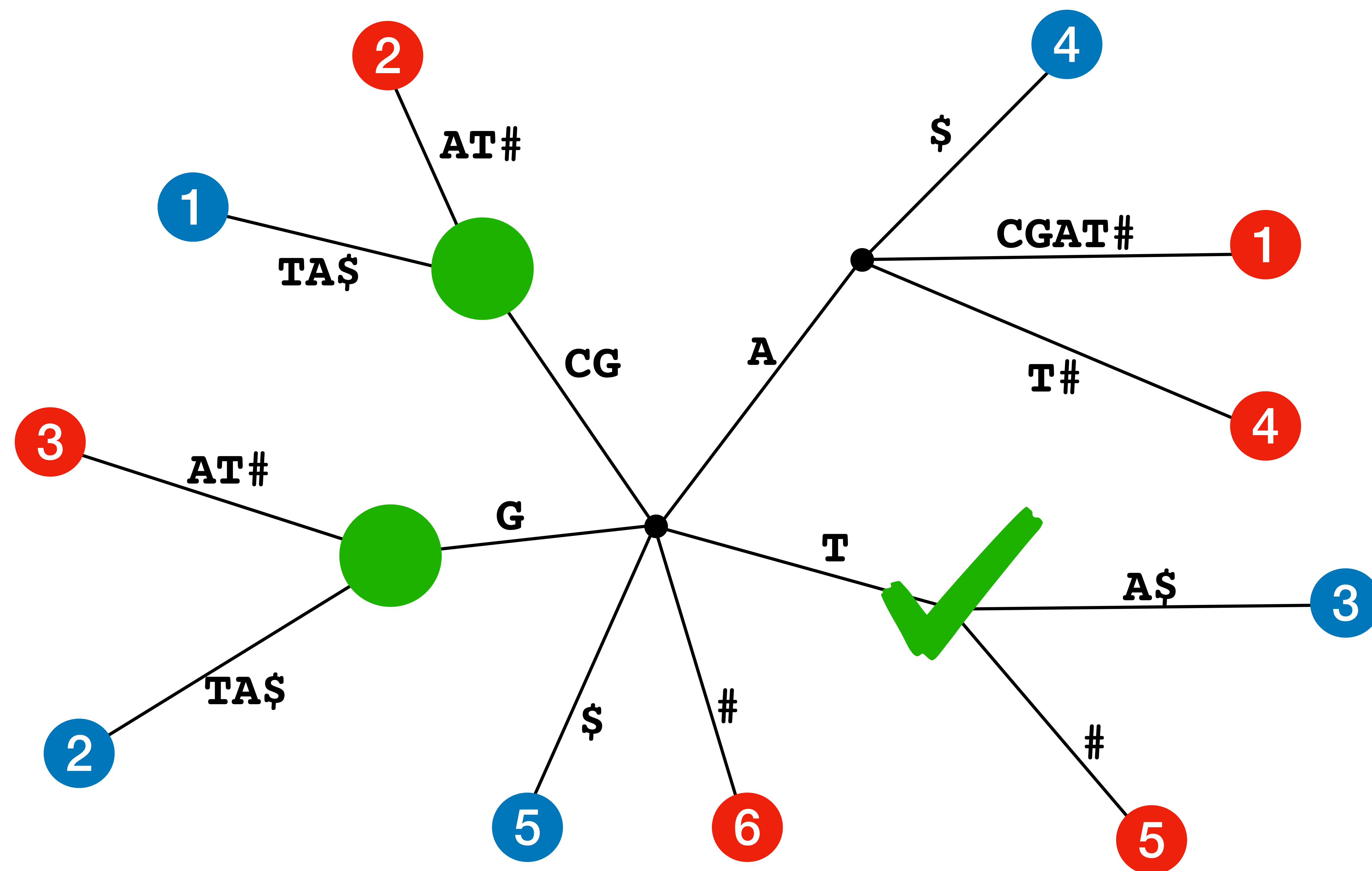
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if *A[i-1] ≠ B[j-1]*, if so report as a MUM

d = 1



S = ACGAT#

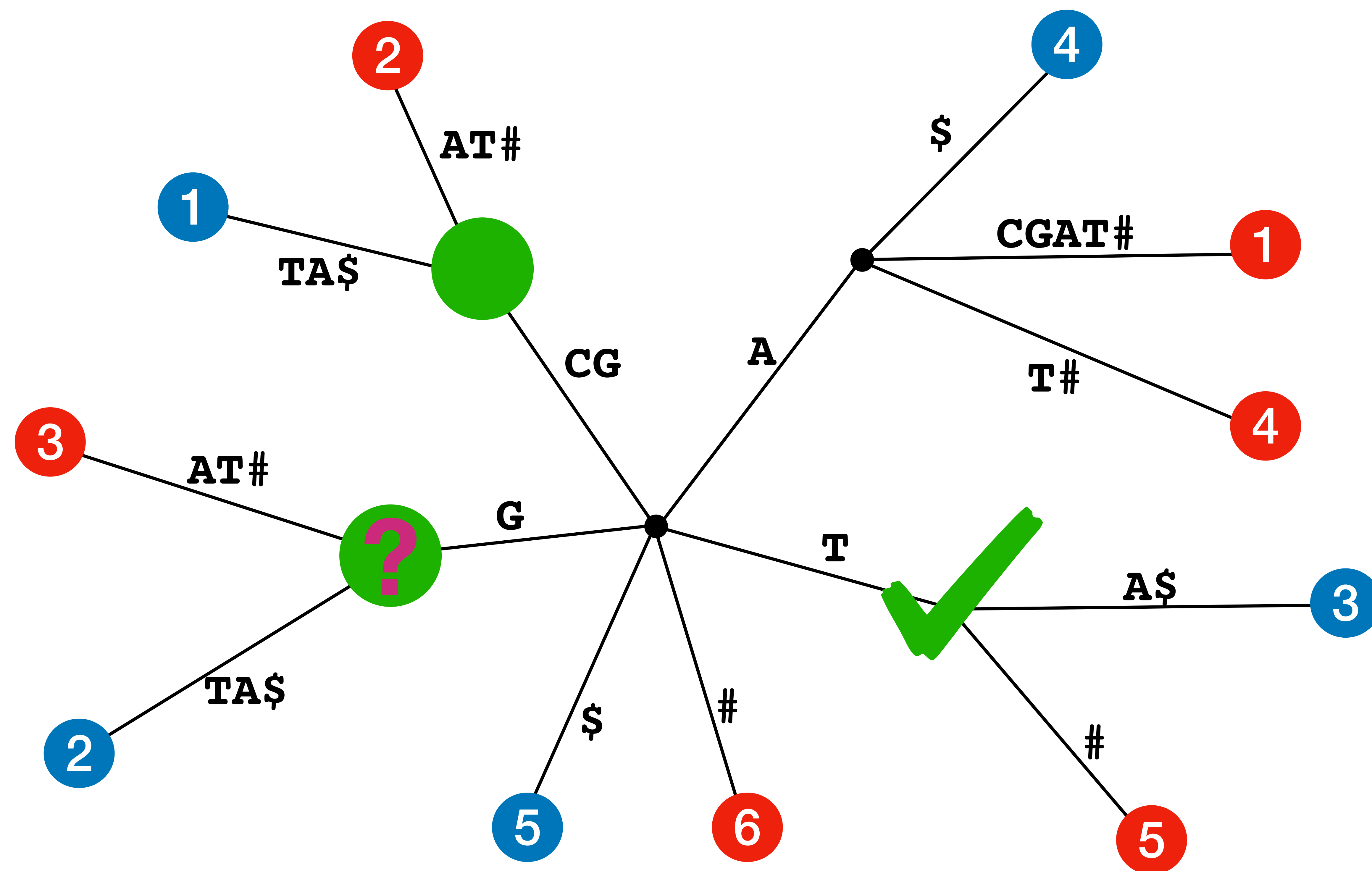
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

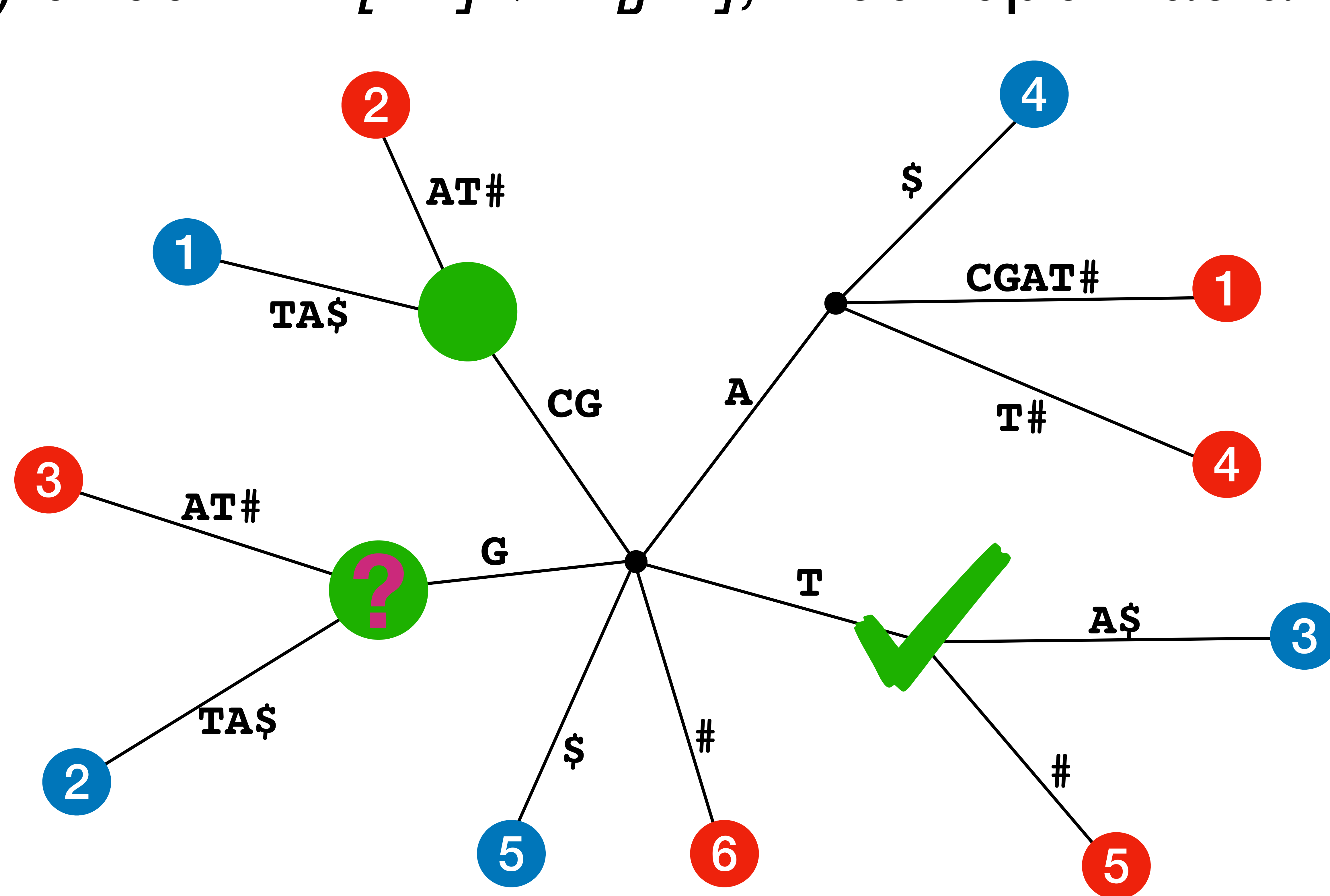
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

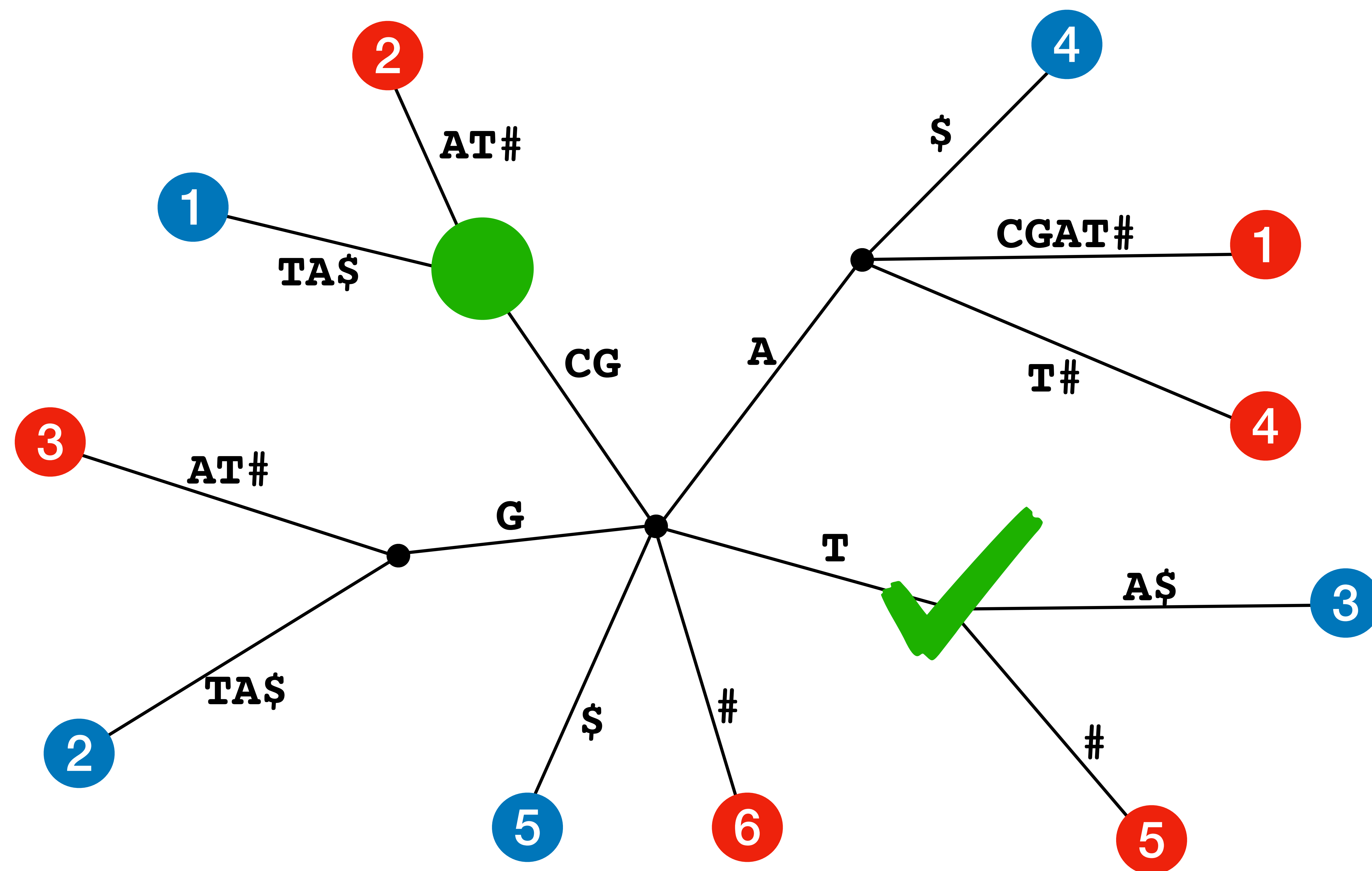
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



S = ACGAT#

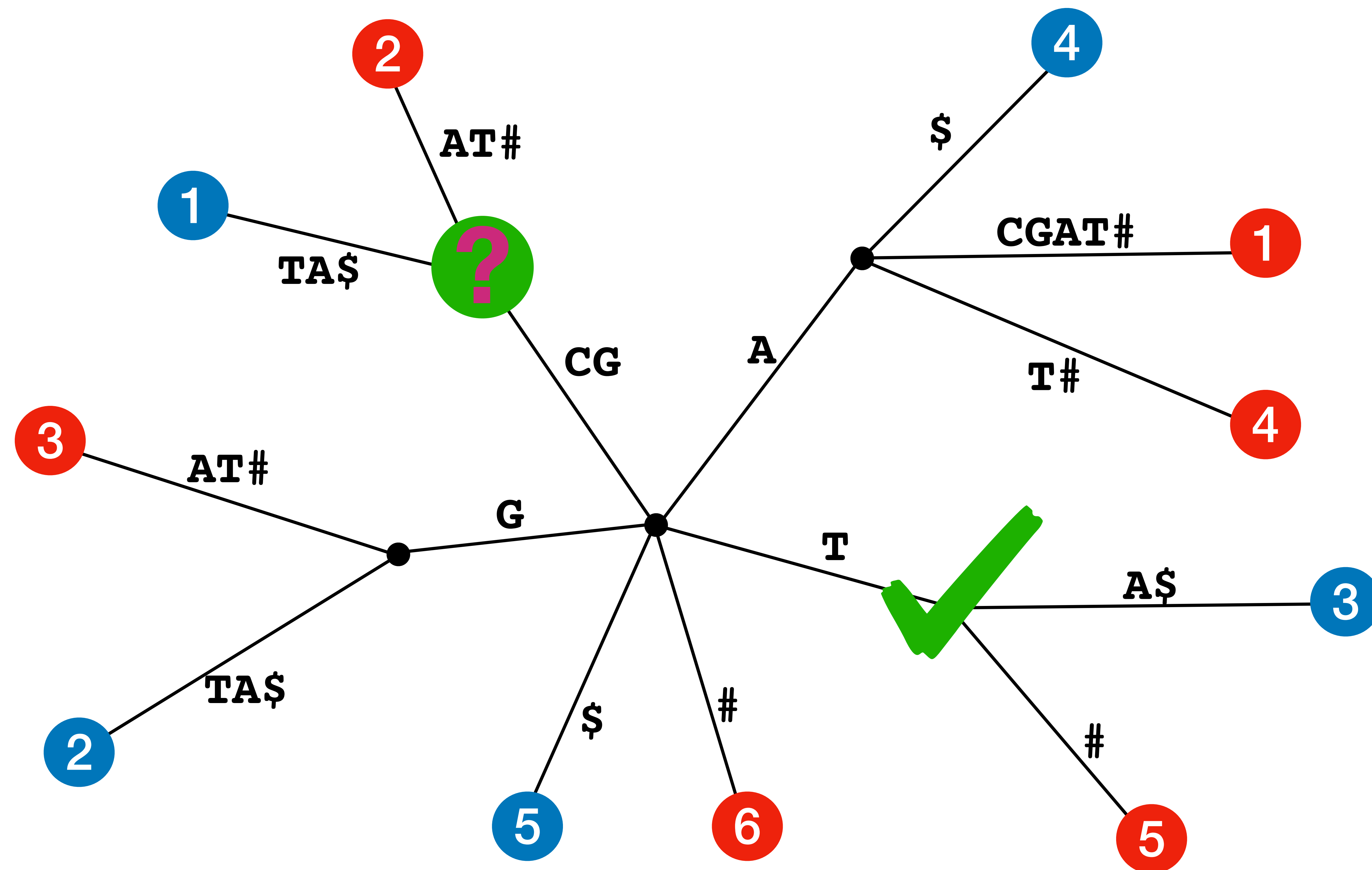
T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

d = 1



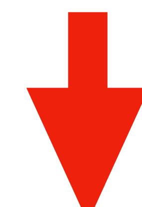
S = ACGAT#

T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

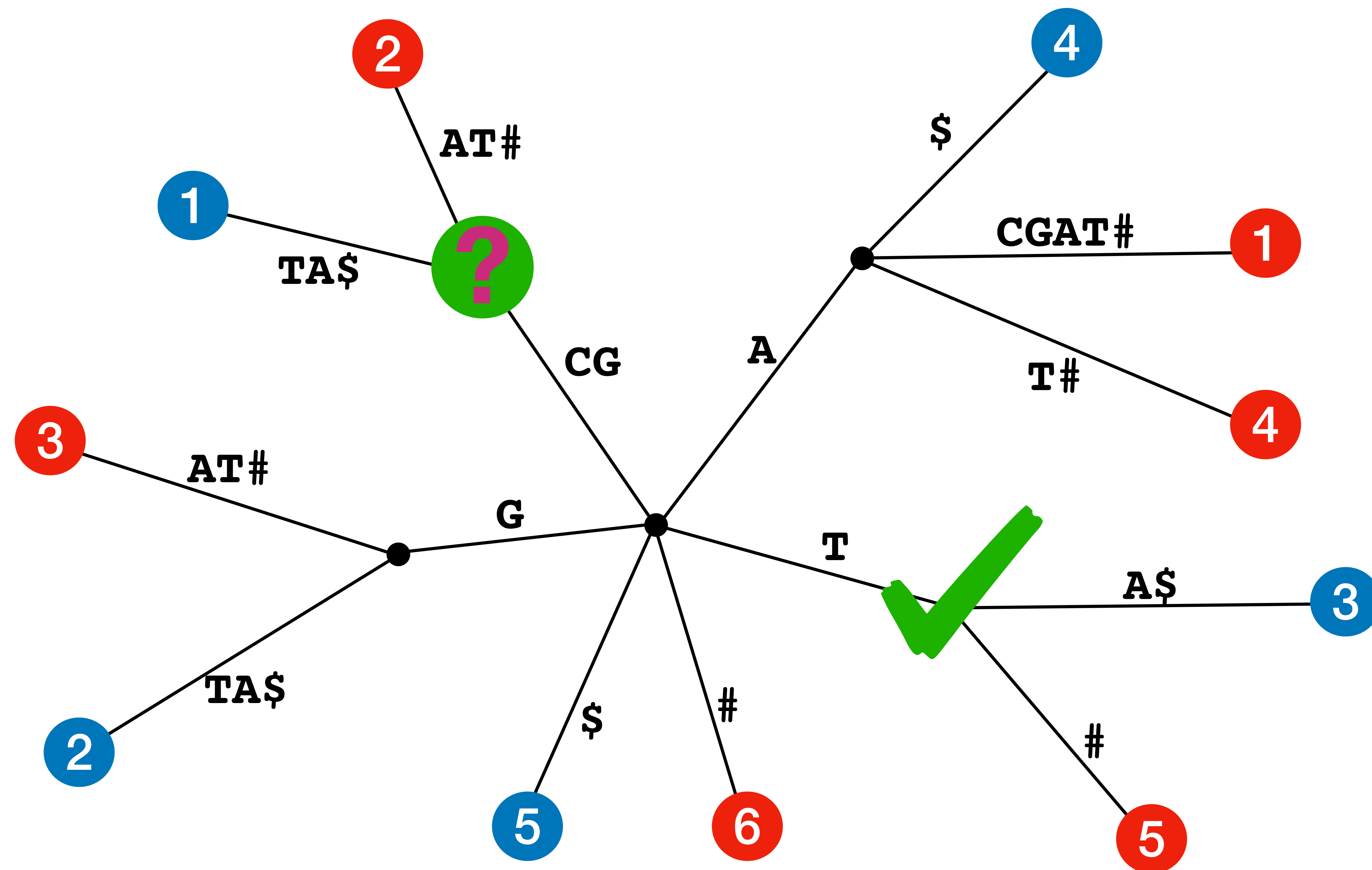
- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if *A[i-1] ≠ B[j-1]*, if so report as a MUM



S = ACGAT#

T = CGTA$

d = 1

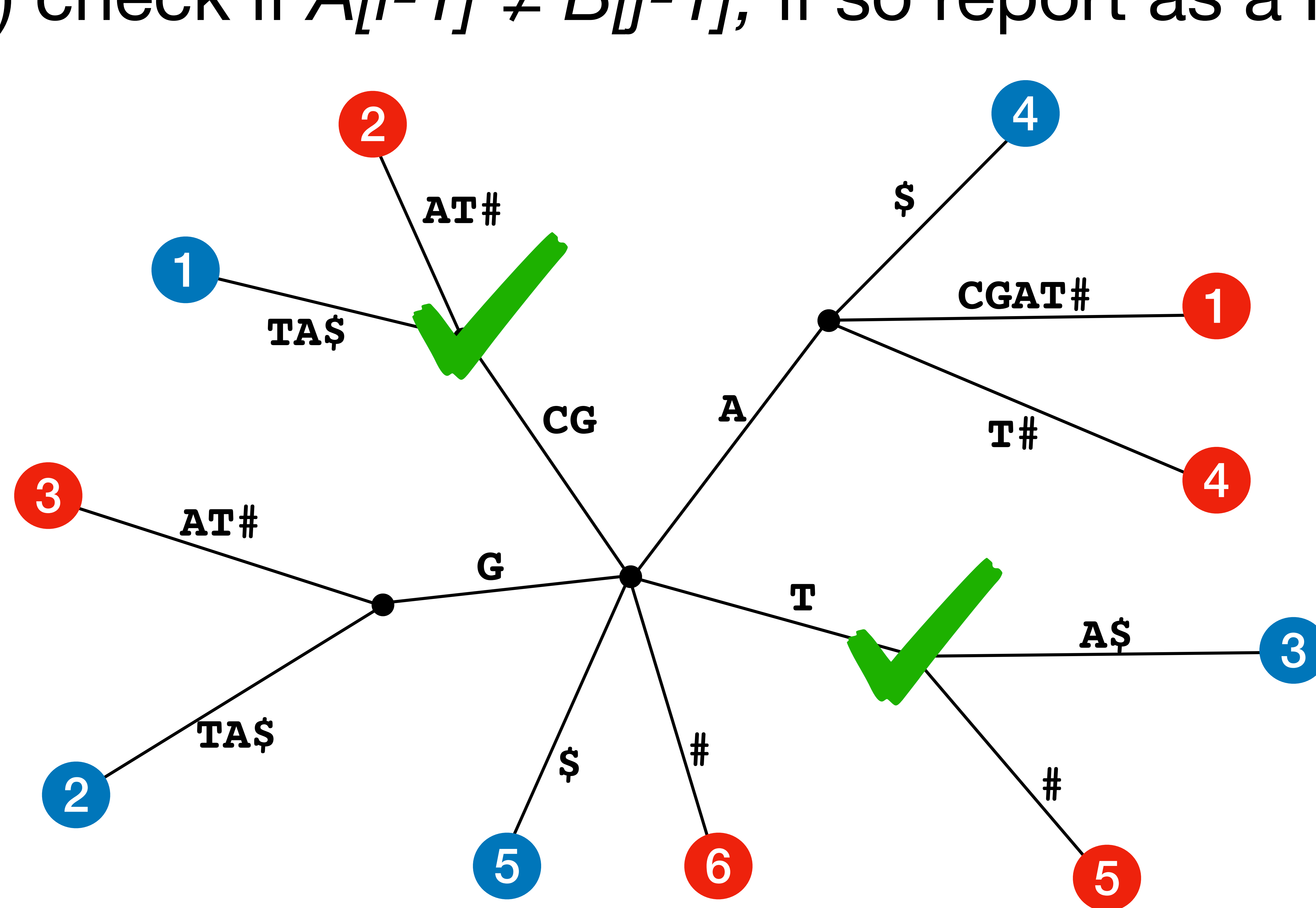


Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if *A[i-1] ≠ B[j-1],* if so report as a MUM

d = 1



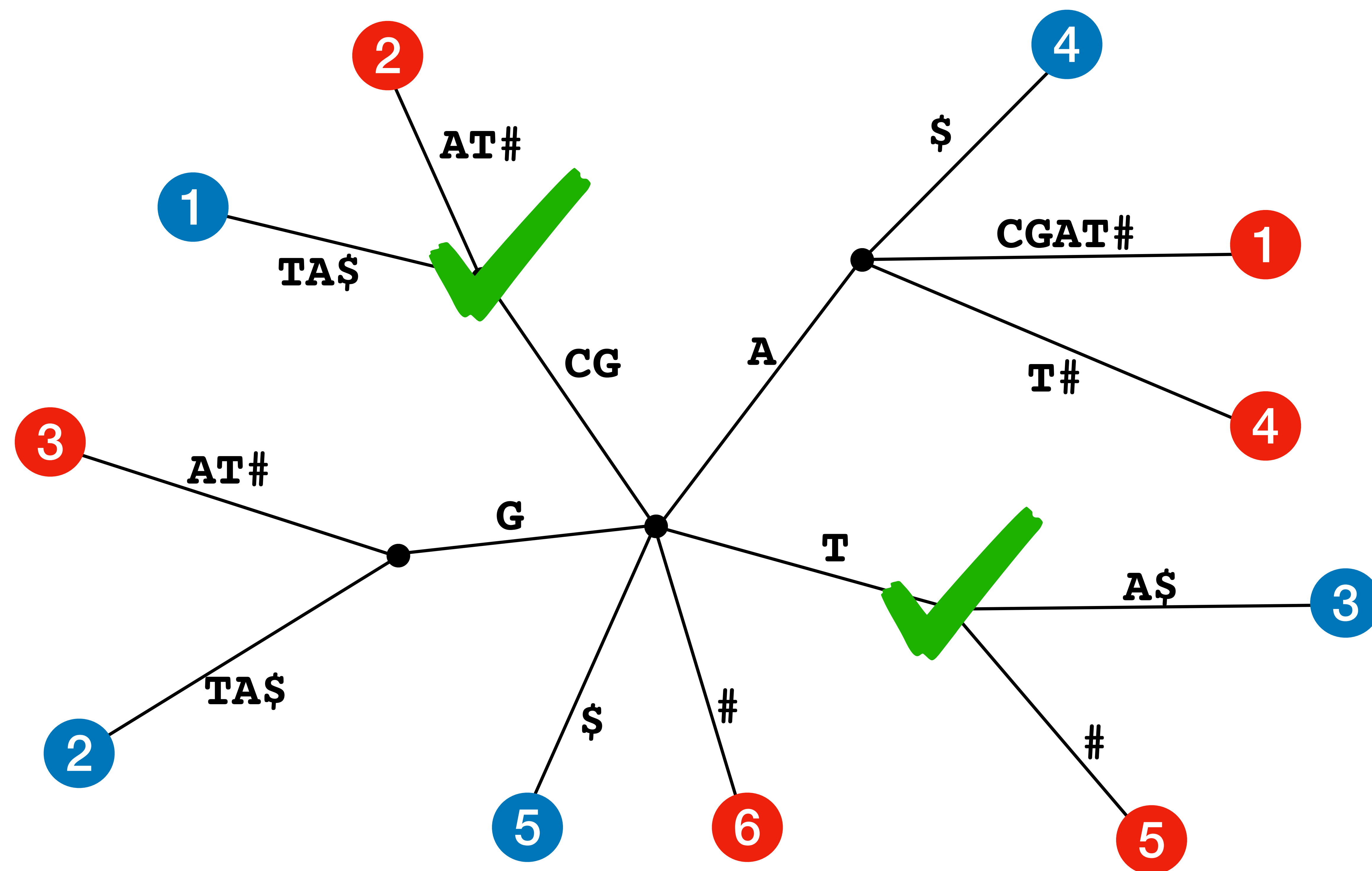
S = ACGAT#

T = CGTA$

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM



d = *1*

S = ACGAT#
T = CGTA$

Algorithm says the MUMs are:
T
CG

Maximal Unique Matches (MUMs)

Using suffix trees:

- Build a generalize suffix tree for *A* and *B*
- Mark (list) all internal nodes with exactly 1 child from each sequence
- For each marked node (lets say the children are labeled *i* from *A* and *j* from *B*) check if $A[i-1] \neq B[j-1]$, if so report as a MUM

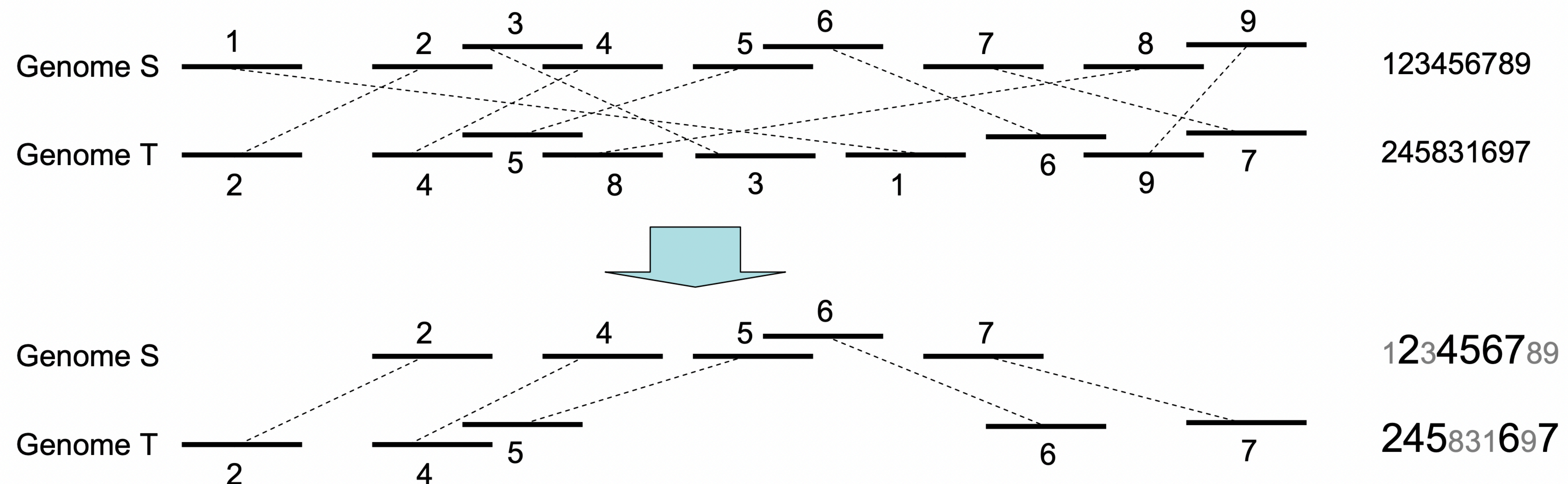
What about running time?

- building the tree: $O(m+n)$
- marking nodes: bounded by number of internal nodes, so $O(m+n)$
- checking prefixes: again bounded by internal nodes, $O(m+n)$

MUMmer

Similar genomes will not only share unique sequences, but also preserve order of these sequences, therefore we can identify the similarity (not necessarily the alignment) by finding the **longest common sequence** of MUMs from the two sequences.

This is the basis of the original MUMmer program (v1).



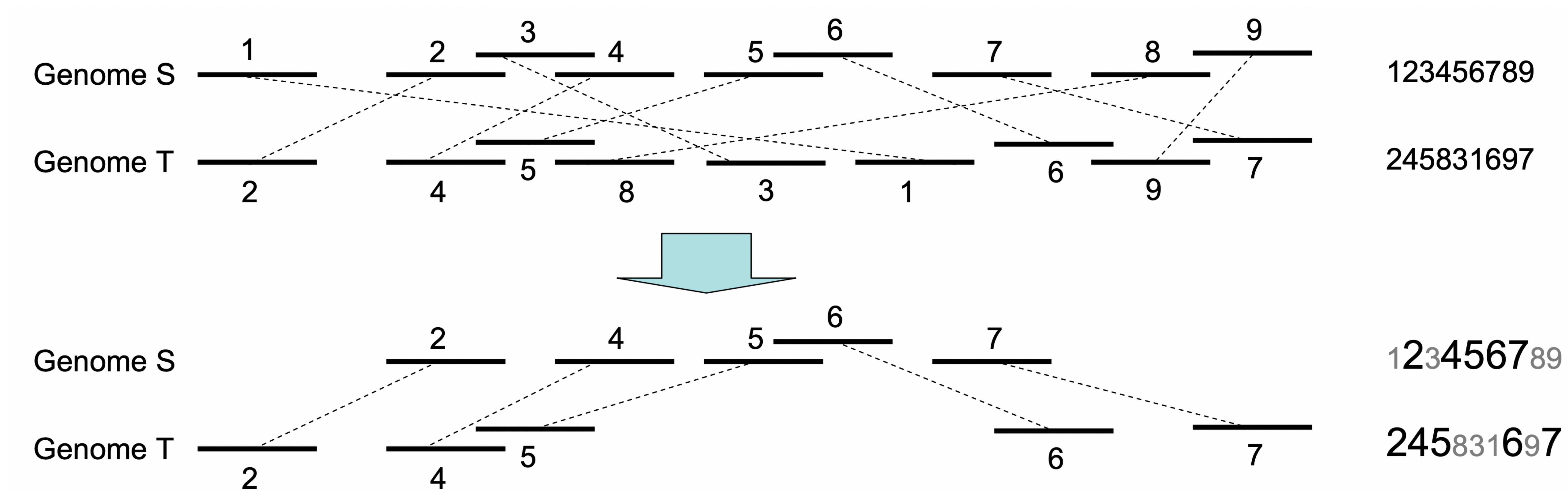
LCS: Dynamic Programming

lets let *V[i,j]* be the length of the longest common subsequence between *P[1....i]* and *Q[1....j]*

And define a function δ such that $P[i] = Q[\delta(i)]$

$$V[i,j] = \max \begin{cases} V[i-1,j] & \text{// P[i] is not involved in the LCS} \\ 1 + V[i-1,\delta(i)-1] \quad j \geq \delta(i) & \text{// P[i] is involved in the LCS} \end{cases}$$

LCS: Dynamic Programming



$$V[i,j] = \max \begin{cases} V[i-1,j] \\ 1 + V[i-1,\delta(i)-1] & j \geq \delta(i) \end{cases}$$

V	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1
2	0	1	1	1	1	1	1	1	1	1
3	0	1	1	1	1	2	2	2	2	2
4	0	1	2	2	2	2	2	2	2	2
5	0	1	2	3	3	3	3	3	3	3
6	0	1	2	3	3	3	3	4	4	4
7	0	1	2	3	3	3	3	4	4	5
8	0	1	2	3	4	4	4	4	4	5
9	0	1	2	3	4	4	4	4	5	5

LCS in $O(n \log n)$ time

The idea is to sparsity the DP using a two key observations:

- going across the rows, only increment by 1
- we only need to compute new values in a narrow region

V	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1
2	0	1	1	1	1	1	1	1	1	1
3	0	1	1	1	1	2	2	2	2	2
4	0	1	2	2	2	2	2	2	2	2
5	0	1	2	3	3	3	3	3	3	3
6	0	1	2	3	3	3	3	4	4	4
7	0	1	2	3	3	3	3	4	4	5
8	0	1	2	3	4	4	4	4	4	5
9	0	1	2	3	4	4	4	4	5	5

→ (1,1), (2,2), (3,3), (7,4), (9,5)

LCS in $O(n \log n)$ time

V	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1
2	0	1	1	1	1	1	1	1	1	1
3	0	1	1	1	1	2	2	2	2	2
4	0	1	2	2	2	2	2	2	2	2
5	0	1	2	3	3	3	3	3	3	3
6	0	1	2	3	3	3	3	4	4	4
V[i-1,*]	0	1	2	3	3	3	3	4	4	5
V[i,*]	0	1	2	3	4	4	4	4	4	5
9	0	1	2	3	4	4	4	4	5	5

$\delta(i) = 4$ ↓   **j'** ↓

Let j' be the smallest integer greater than $\delta(i)$ such that $V[i-1,\delta(i)-1]+1 < V[i-1,j']$.

In that case we can see that

$$V[i,j] = \begin{cases} 1 + V[i-1,\delta(i)-1] & \delta(i) \leq j \leq j'-1 \\ V[i-1,j] & \text{otherwise} \end{cases}$$

LCS in $O(n \log n)$ time

Therefore, we construct a new row (i) by

1. copying the tuples from i-1
2. delete all tuples $(j, V[i-1,j])$ where $j \geq \delta(i)$ and $V[i-1,j] \leq V[i-1,\delta(i)-1]+1$
3. insert $(\delta(i), V[i-1,\delta(i)-1]+1)$.

2	0	1	1	1	1	1	1	1	1	1
3	0	1	1	1	1	2	2	2	2	2
4	0	1	2	2	2	2	2	2	2	2
5	0	1	2	3	3	3	3	3	3	3
6	0	1	2	3	3	3	3	4	4	4
V[i-1,*]	0	1	2	3	3	3	3	4	4	5
V[i,*]	0	1	2	3	4	4	4	4	4	5
9	0	1	2	3	4	4	4	4	5	5

$\delta(i) = 4$

j'

(1,1), (2,2), (3,3), (7,4), (9,5)

(1,1), (2,2), (3,3), **(7,4)**, (9,5)

(1,1), (2,2), (3,3), (9,5)

(1,1), (2,2), (3,3), **(4, 3+1)**, (9,5)

(1,1), (2,2), (3,3), (4,4), (9,5)

Step 2

Step 3

LCS in $O(n \log n)$ time

If we store the tuples as a binary search tree: search, insert, & delete are $O(\log n)$ time each.

Since we can insert at most n tuples (one per row), and each tuple can only be deleted once. Therefore those operations are at most $O(n \log n)$ time total.

Original MUMmer

Step 1: identify all MUMs. This can be done in $O(n)$ time assuming the sequences are of length n.

Step 2: employ LCS on the MUMs. Lets call the number of MUMs m, here $m << n$. This takes $O(m \log m)$ time.

Step 3: employ another alignment algorithm to fill small gaps (various tools used here).

MUMmer2 & MUMmer3

These improvements changed the following:

- Improved memory consumption using better implementations of STs
- Using a simple ST, rather than a generalized tree: build the tree on one them *stream* the other over it to find MUMs
- Implementing clustering, since there may be major structural changes, don't just find one LCS, split the problem down and find groups of LCSs
- Relaxing uniqueness

Dot Plots

A concept used often in (computational) biology is visualization to get a general idea of what the data means.

In genome alignment (and many other large alignment problems as we will see) we use a dot plot.

- Each position in the graph represents a location in the two genomes.
- A dot represents a match of a certain length surrounding those locations

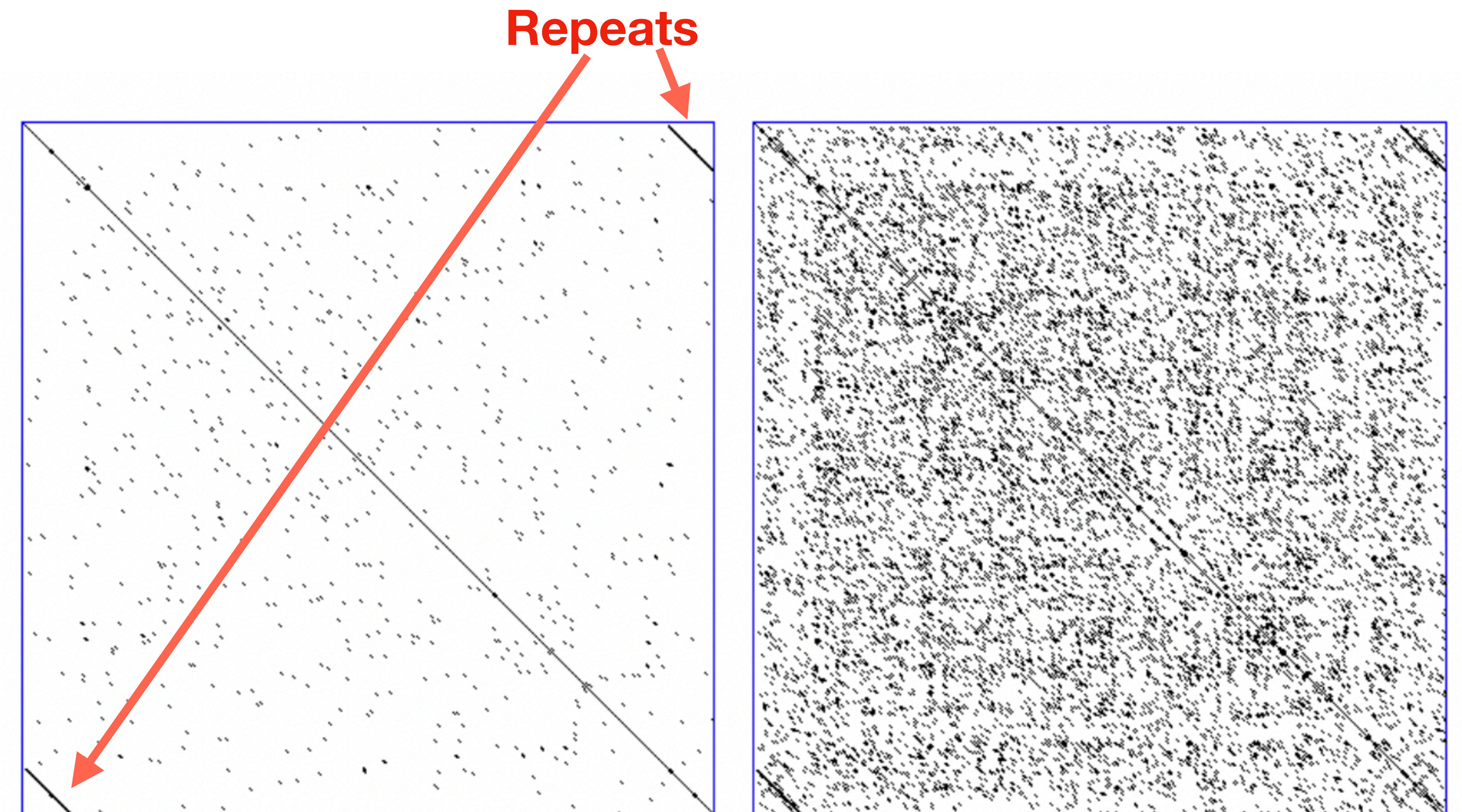
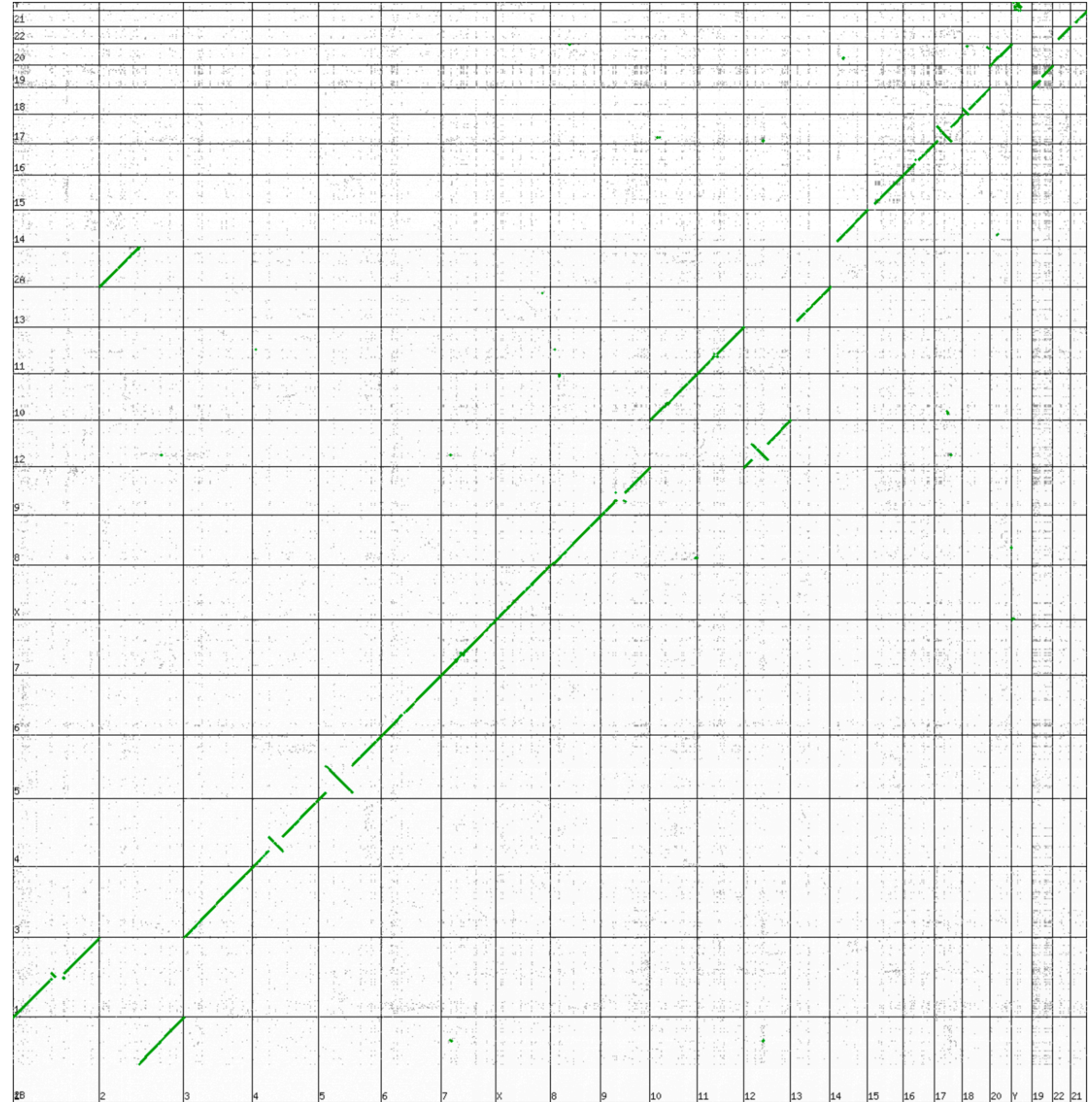


FIGURE 4.19: The dot plot between a HIV type 1 subtype C (AB023804) and itself. The dot plot on the left was generated using window size 9 and allowing no mismatches. The dot plot on the right was generated using window size 9 and allowing at most 1 mismatch.

Dot Plot

Human vs. Chimpanzee



https://genomevolution.org/wiki/index.php/File:Master_7029_7071.CDS-CDS.blastn.dag.go_c4_D20_g10_A5.aligncoords.gcoords_ct0.w1200.png