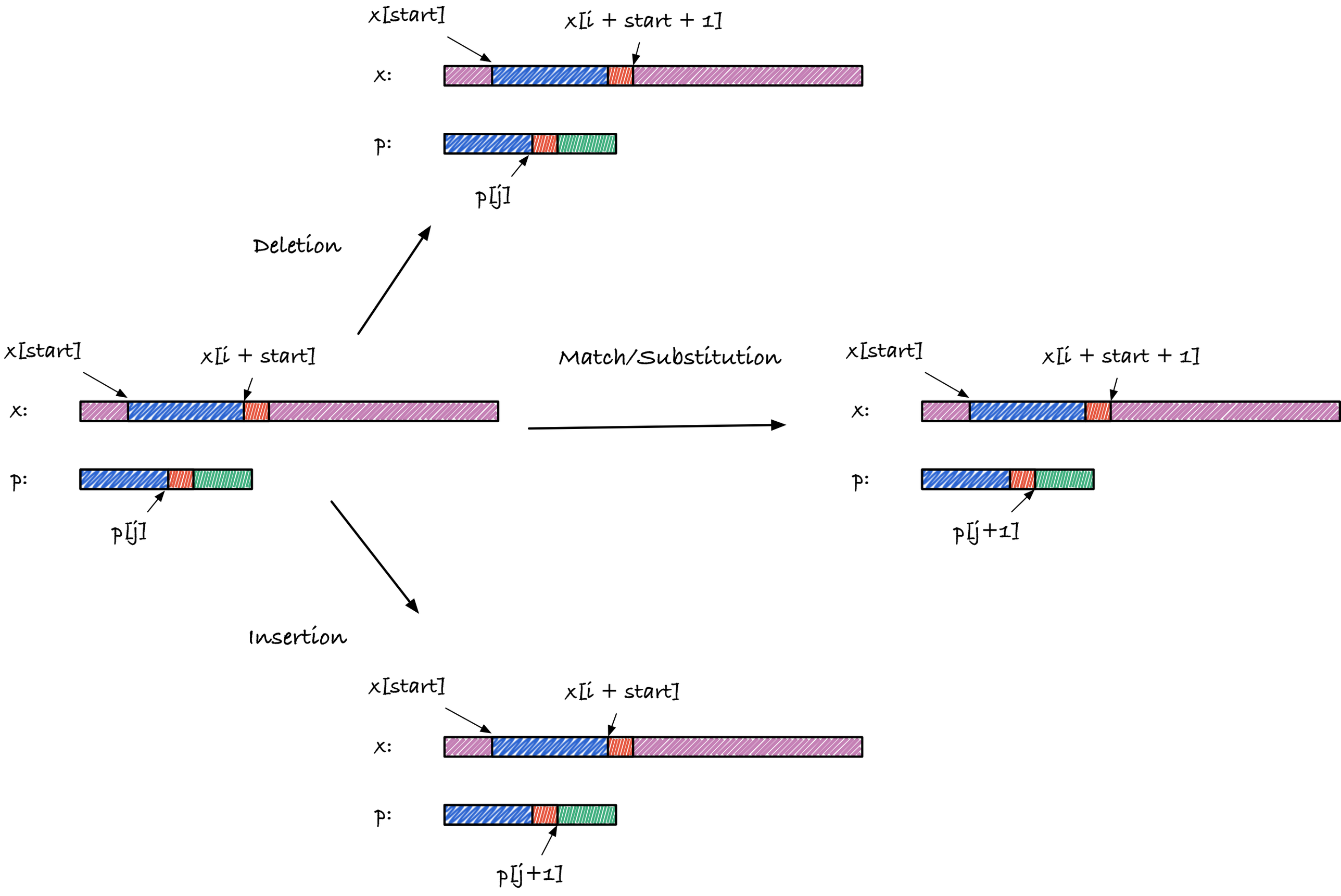
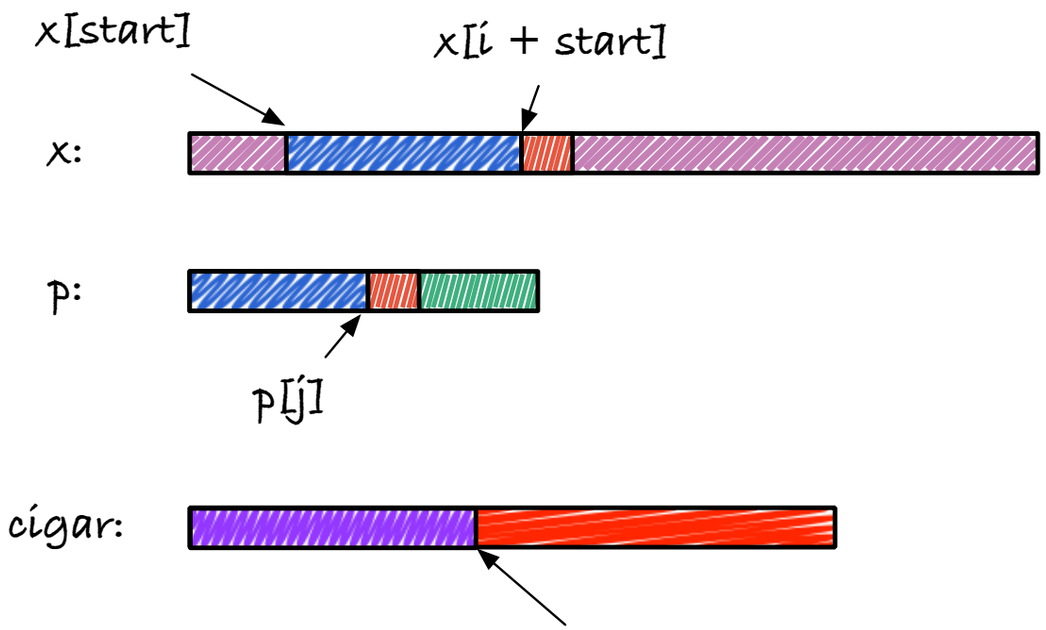
Read mapping

Approximative mapping using suffix trees


x[start]
x[i + start + 1]
x:
p:
p[j]
Deletion
x[start]
x[i + start]
x:
Match/Substitution
x[start]
x[i + start + 1]
x:
p:
p[j]
p:
p[j+1]
Insertion
x[start]
x[i + start]
x:
p:
p[j+1]



It is easier to build a "pseudo cigar" with individual operations and not length encoded.

It is easy to translate such a cigar string into the right format at the end of the computations.

match [x[start + i] =p[j]]:
cigar[k] = M
i = i + 1; j = j + 1; k = k + 1

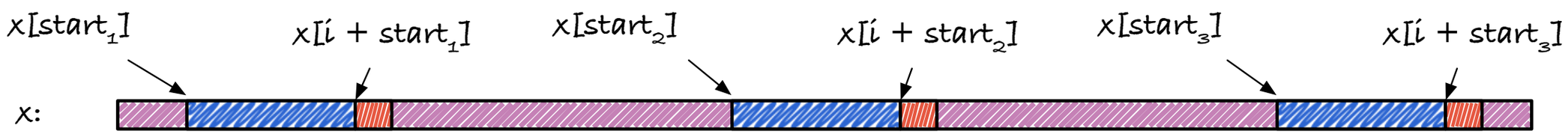
substitution
[x[start + i]!=p[j] and edits > 0]:
cigar[k] = M
i = i + 1; j = j + 1; k = k + 1
edits = edits - 1

Deletion [edits > 0]:
cigar[k] = D
i = i + 1; k = k + 1
edits = edits - 1

Insertion [edits > 0]:
cigar[k] = I
j = j + 1; k = k + 1
edits = edits - 1

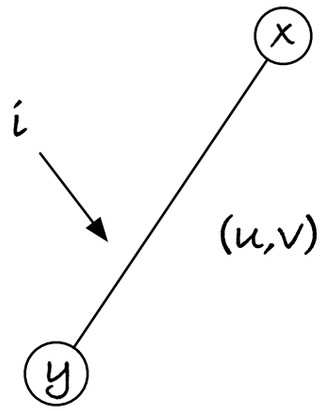
If we search in a suffix tree, we search from multiple different start points.

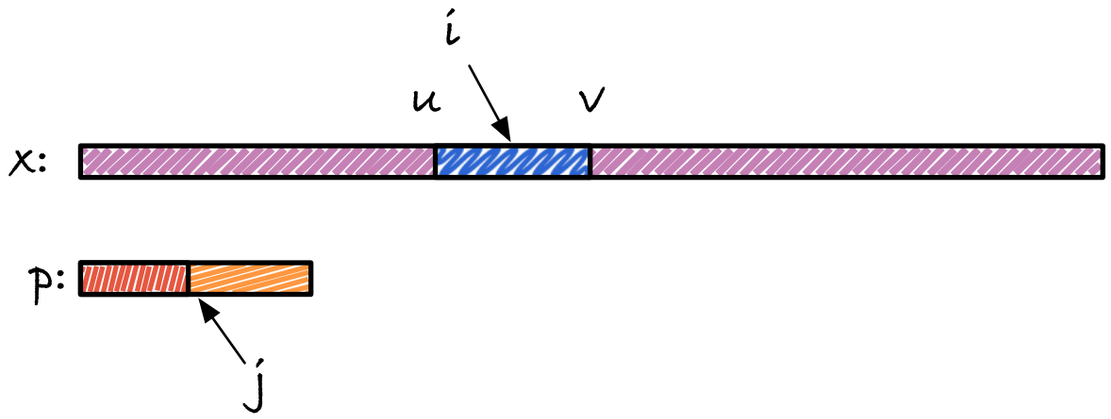
The cost of matches/substitutions will differ from location to location.



Recursive function search(i,j,k,edits,u,v)

For edges:



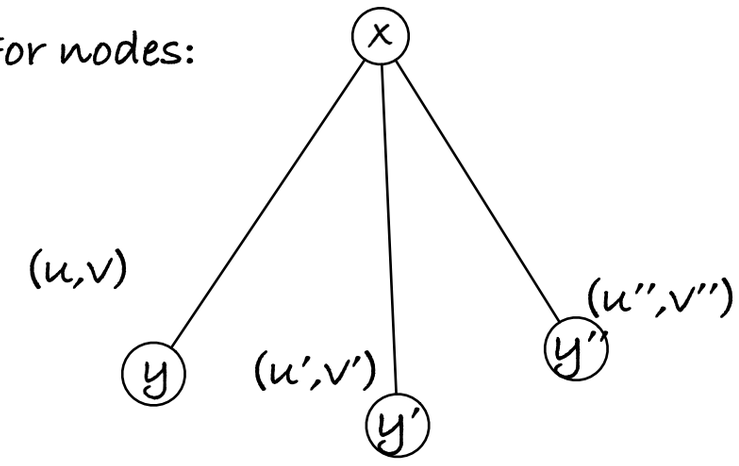


search(j, i, k, edits, u, v)

```
if edits < 0 return
if j = m: report leaves in subtree and return
if i = v: handle_node(j, y, edits) and return

if x[i] = y[j]:
   cigar[k] = 'M'
   search(i+1, j+1, k + 1, edits, u, v)
else:
   cigar[k] = 'M'
   search(i+1, j+1, k + 1, edits - 1, u, v)
cigar[k] = 'I'
search(i + 1, j, k + 1, edits - 1, u, v)
cigar[k] = 'D'
search(i, j, k + 1, edits - 1, u, v)
```

For nodes:



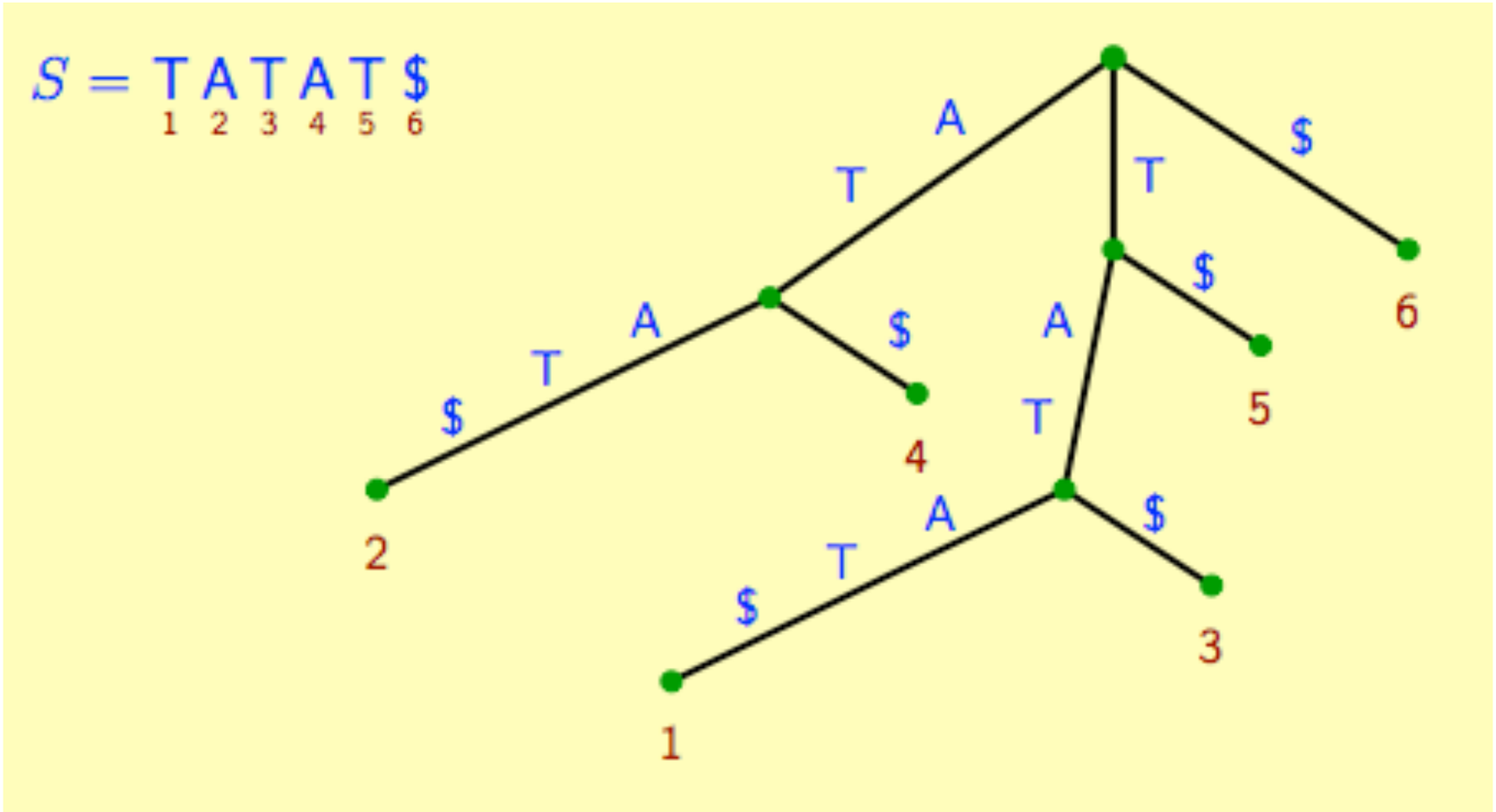
handle_node(j, x, edits):

For each child y(u,v)
  search(j, u, k, edits, u, v)

Suffix-tree approaches

- If we use a suffix tree we can explore all potential "start" pointers at the same time

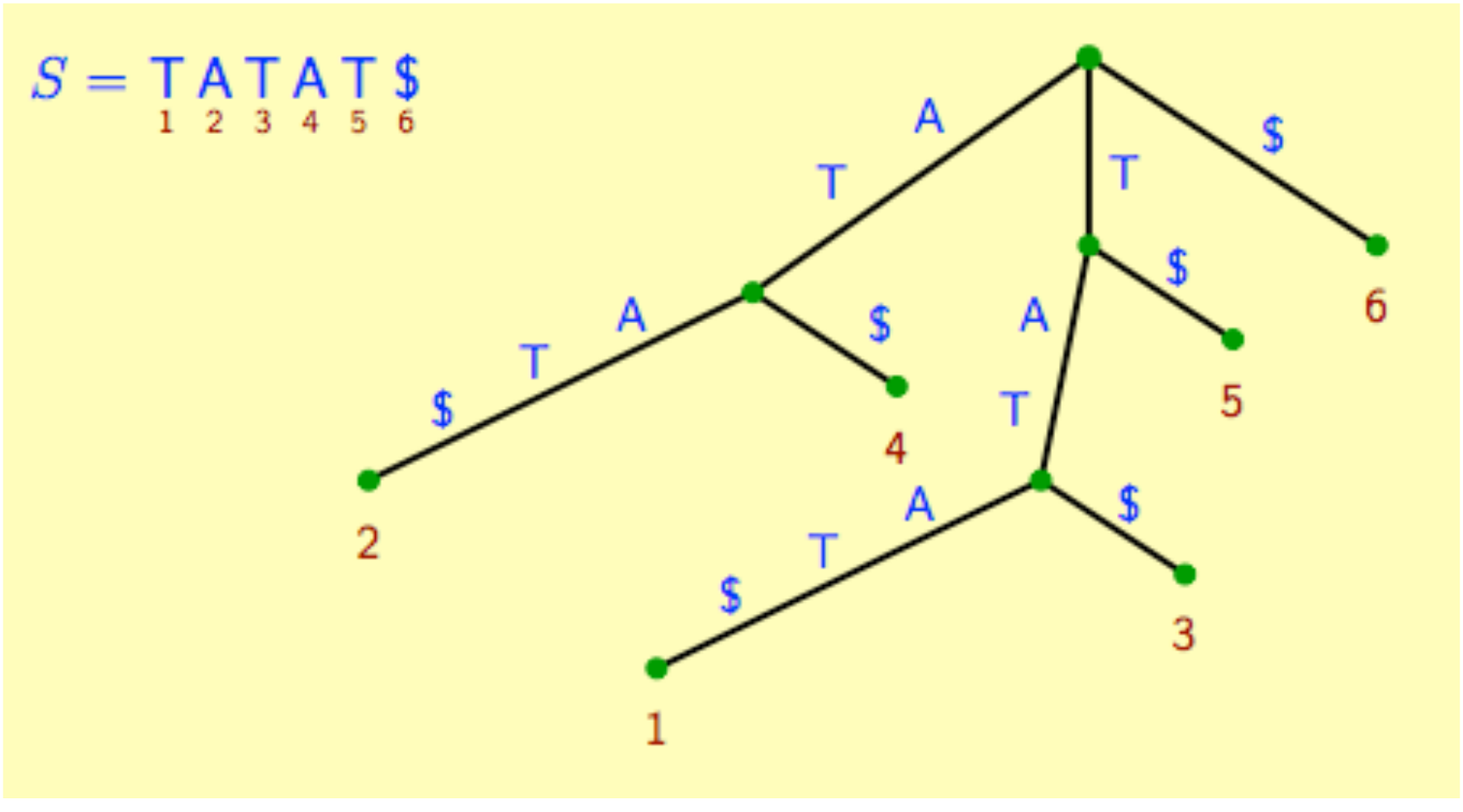
Branch and bound approach to approximative matching



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

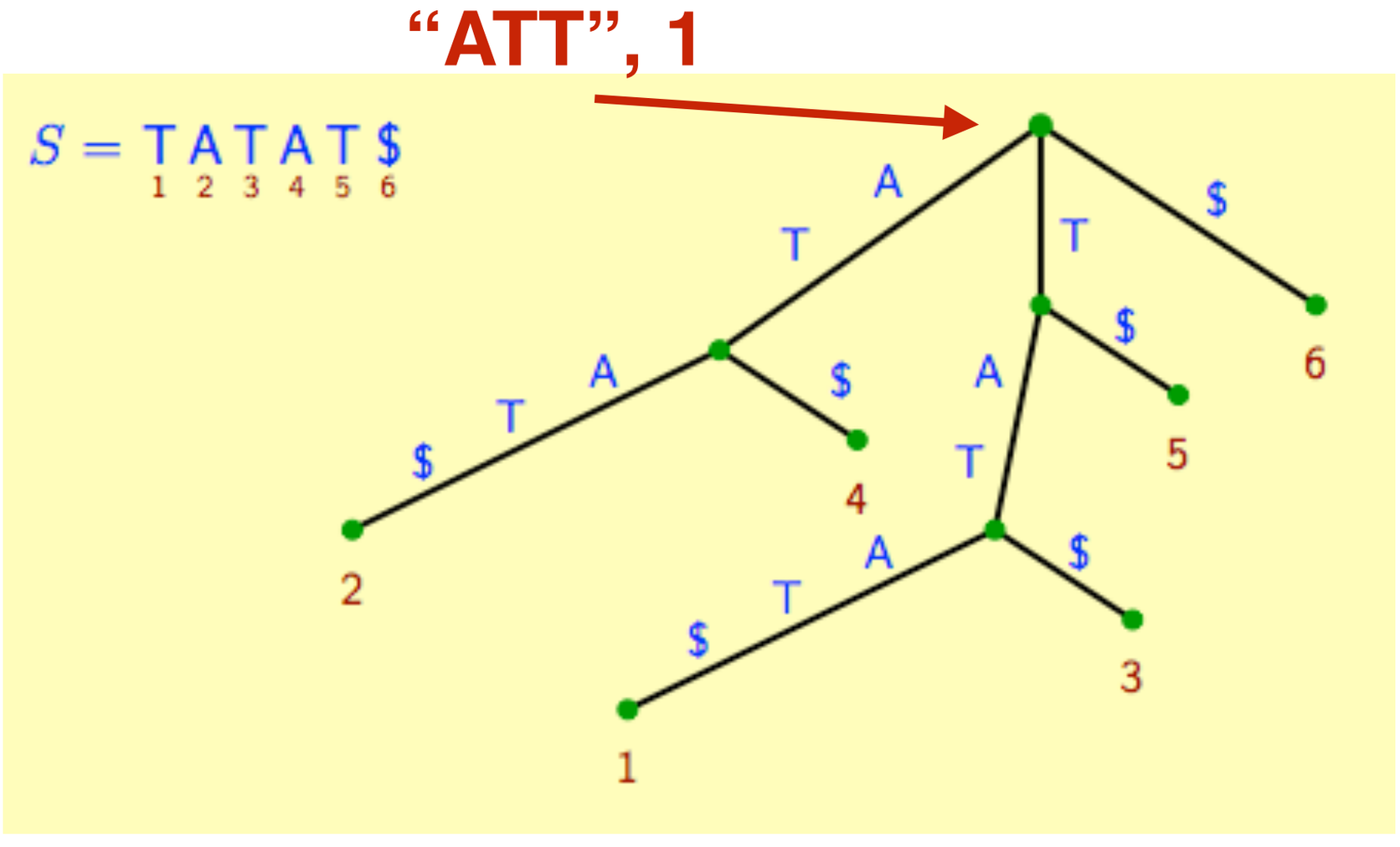
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

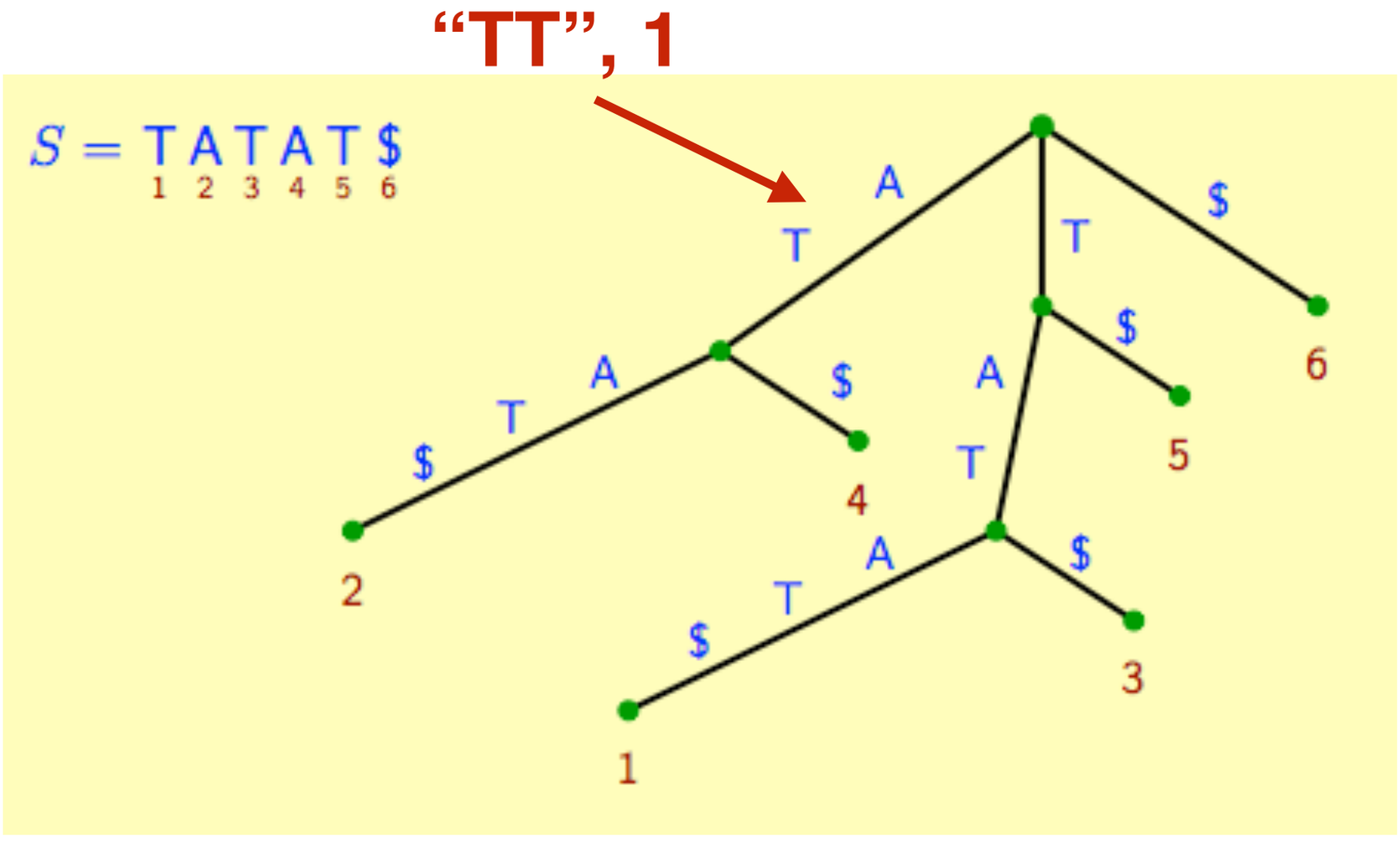
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

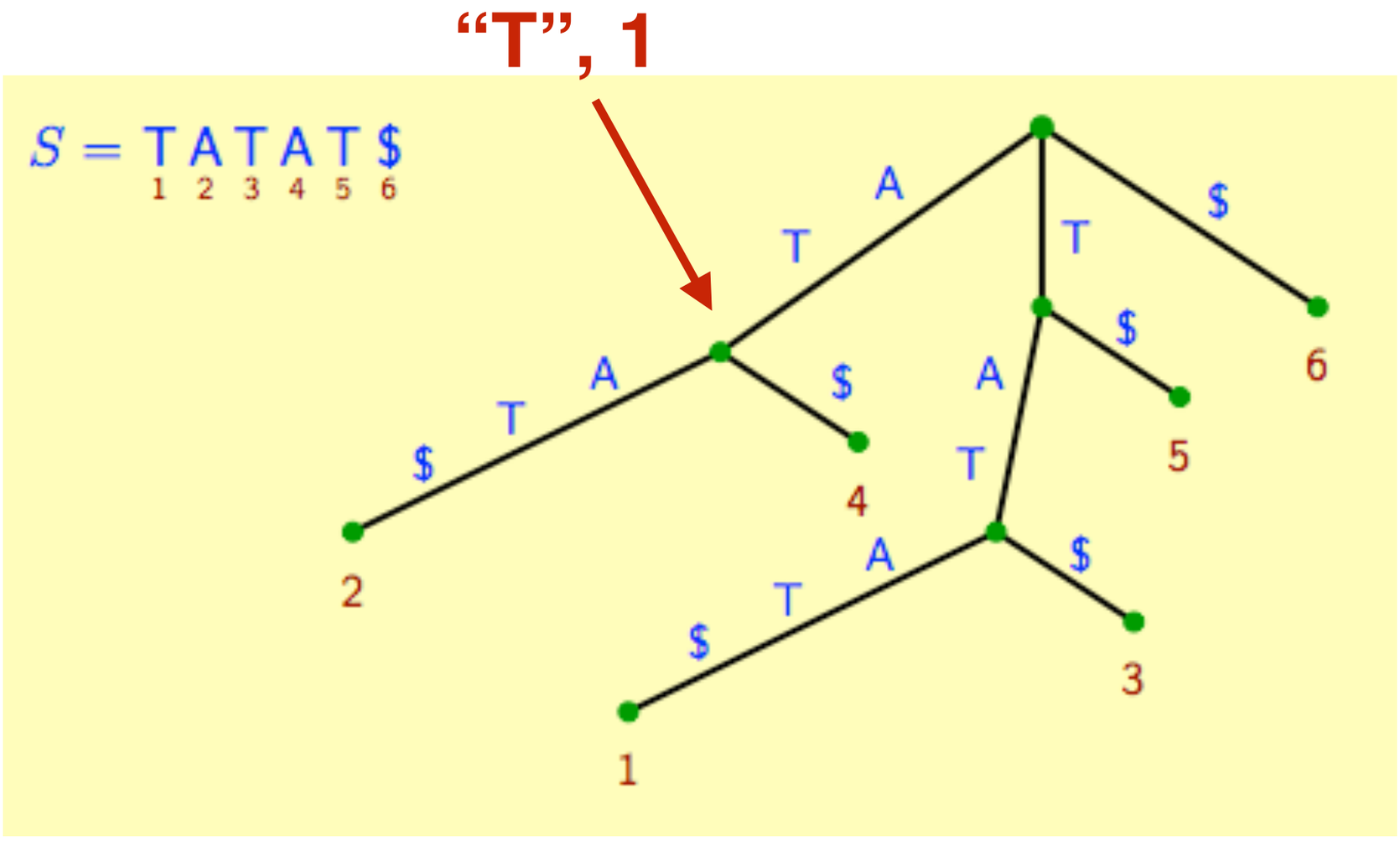
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

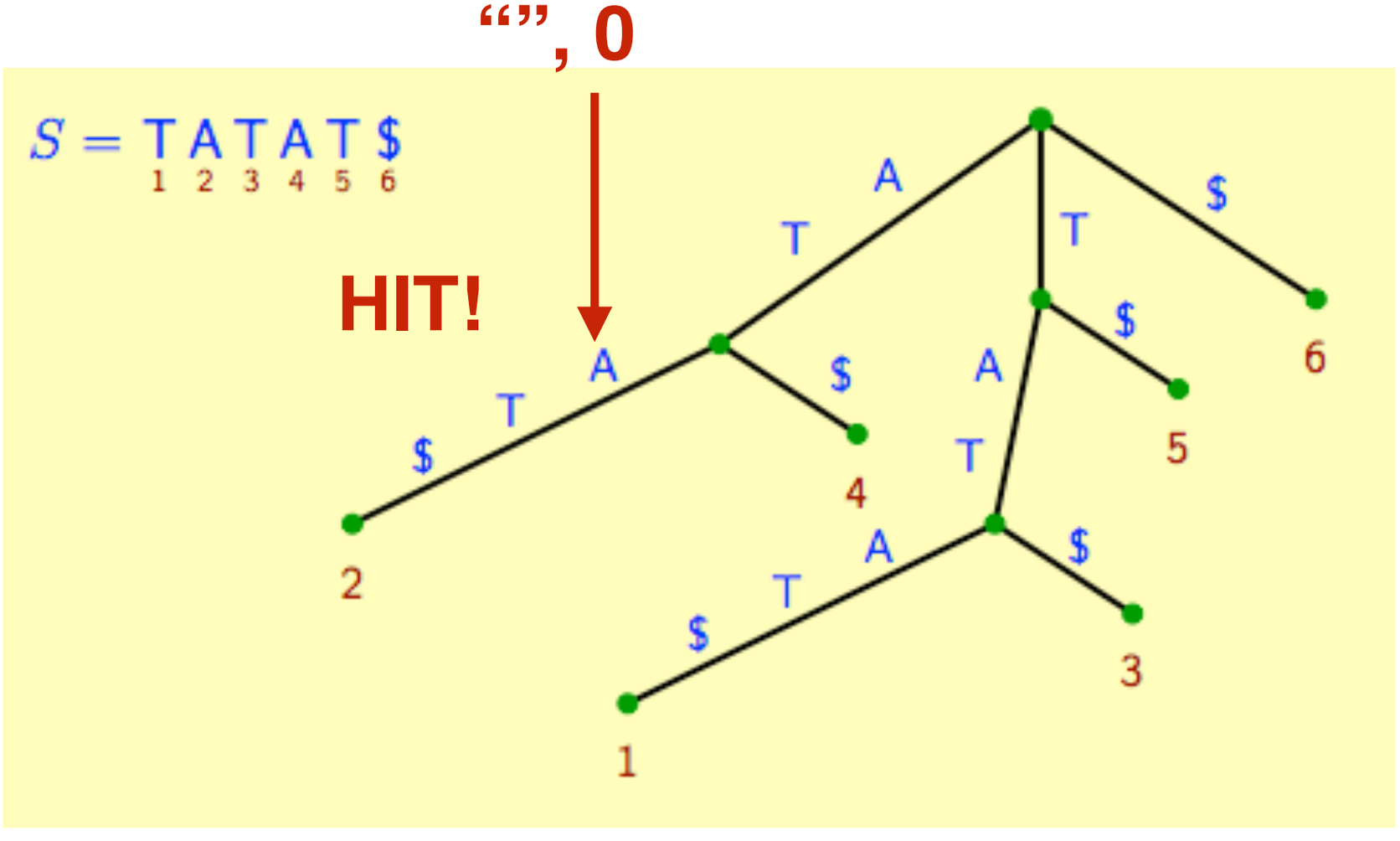
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

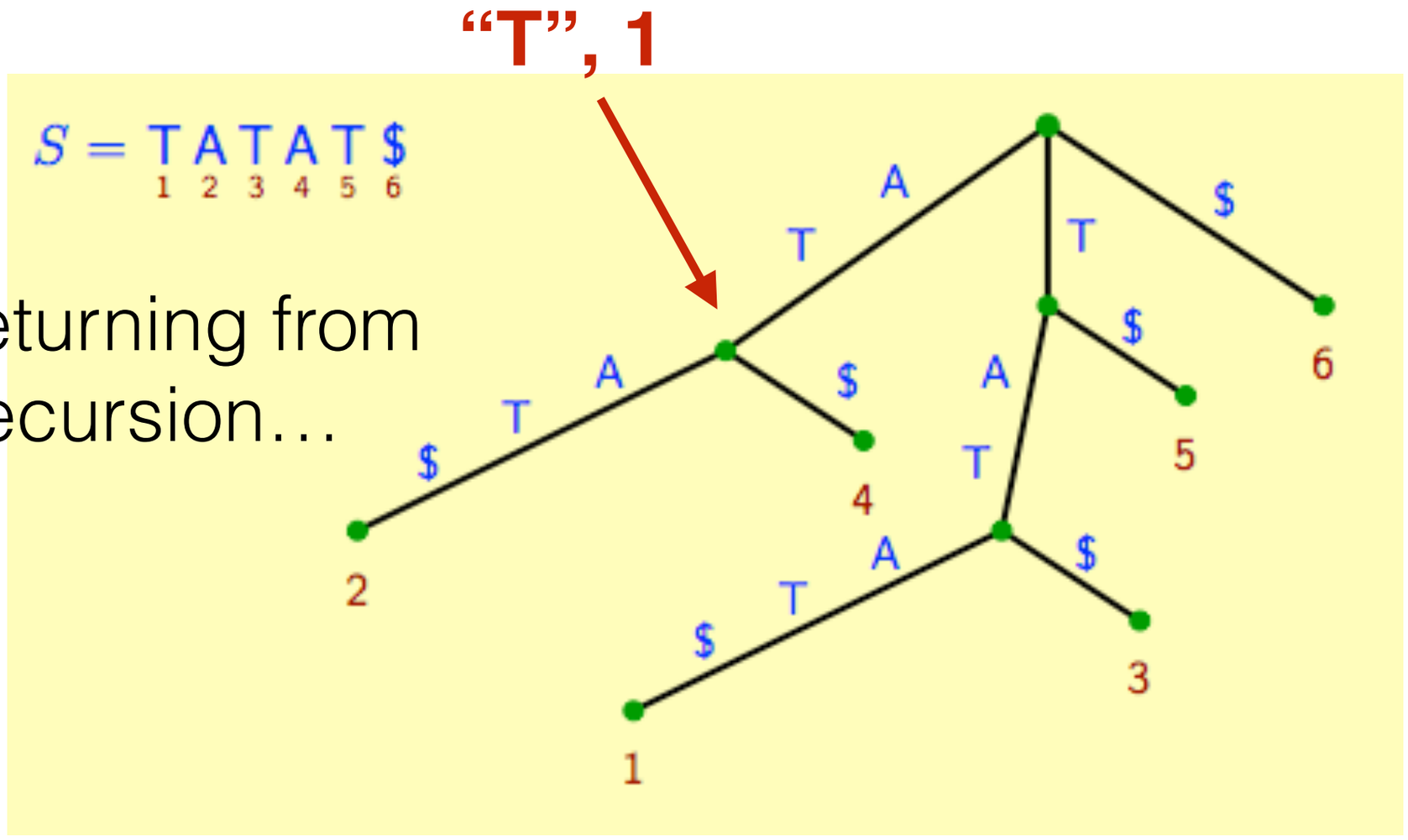
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

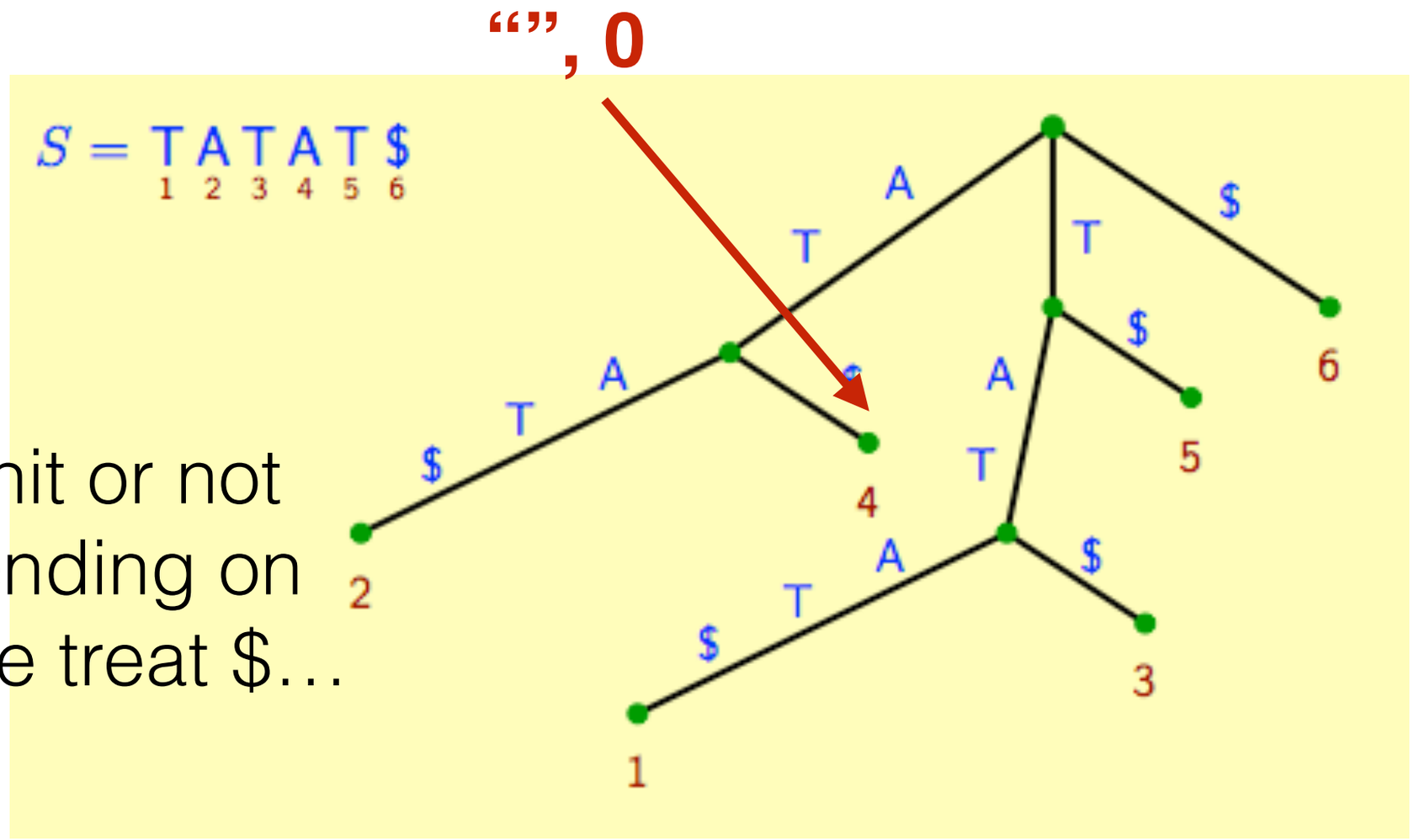
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left

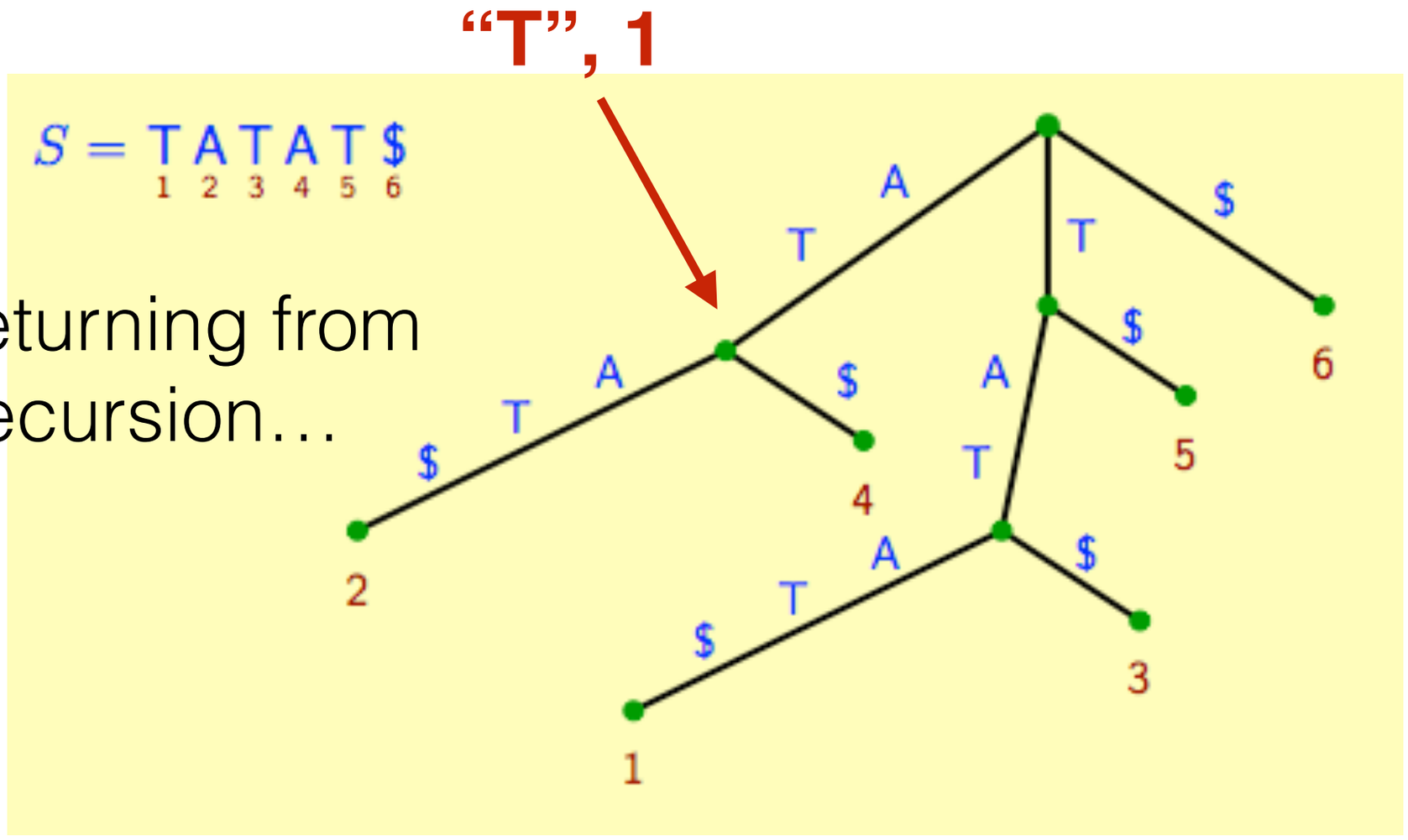


…a hit or not depending on how we treat $…

Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

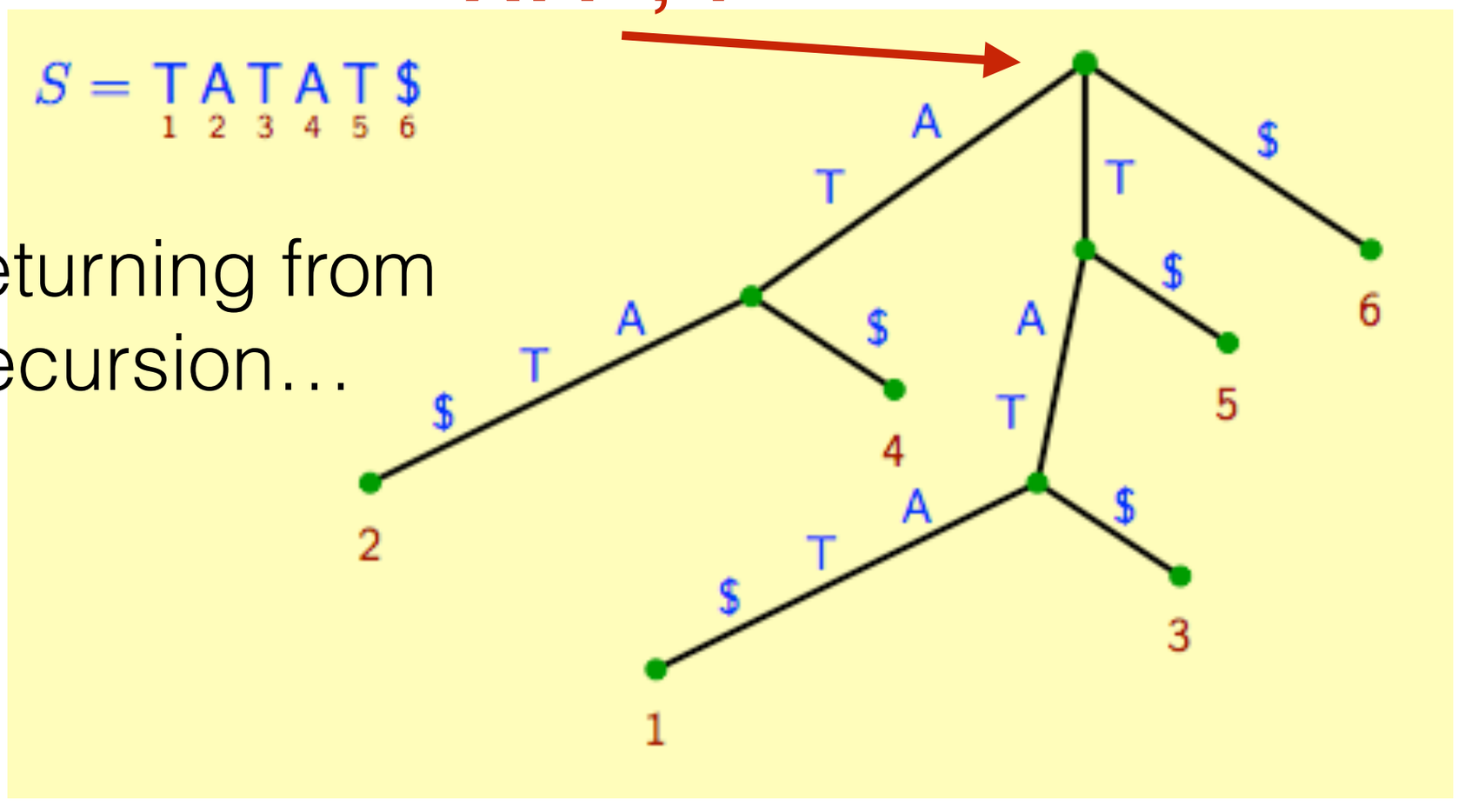
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

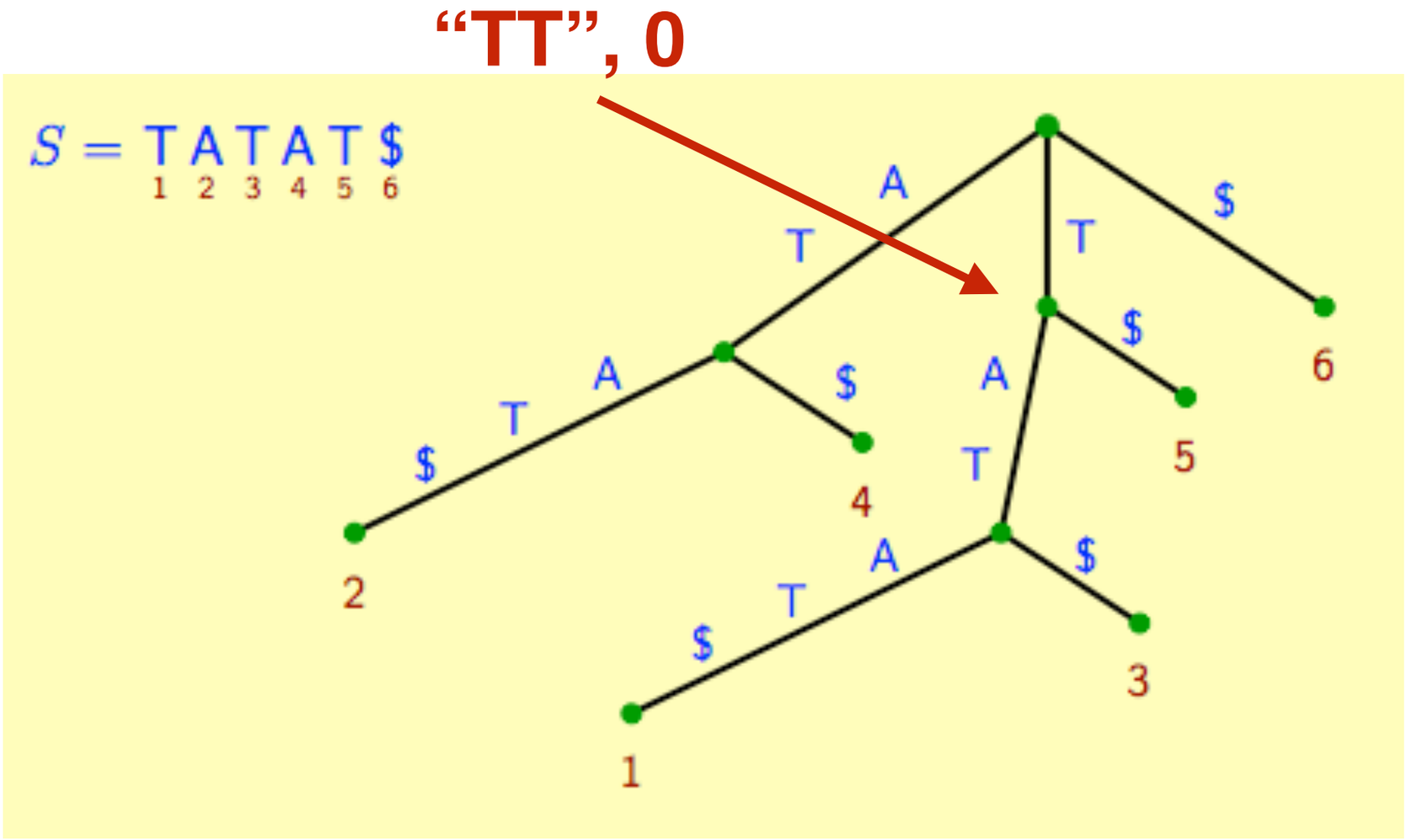
- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left



Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left

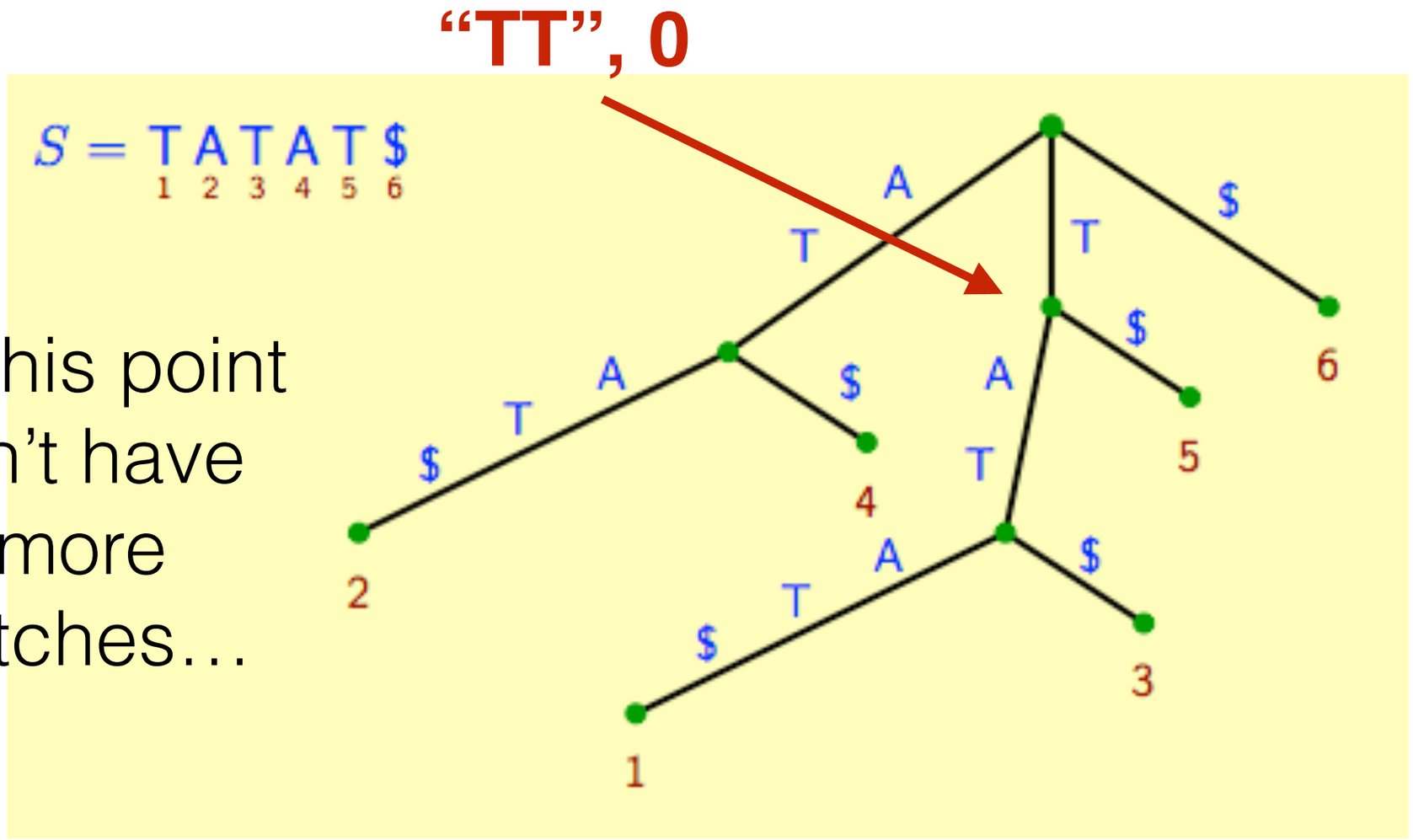


Search for ATT allowing at most one mismatch

Depth first search for pattern p with k mismatches remaining…

- we either match and move down the tree with k left
- or we mismatch and move down the tree with k-1 left

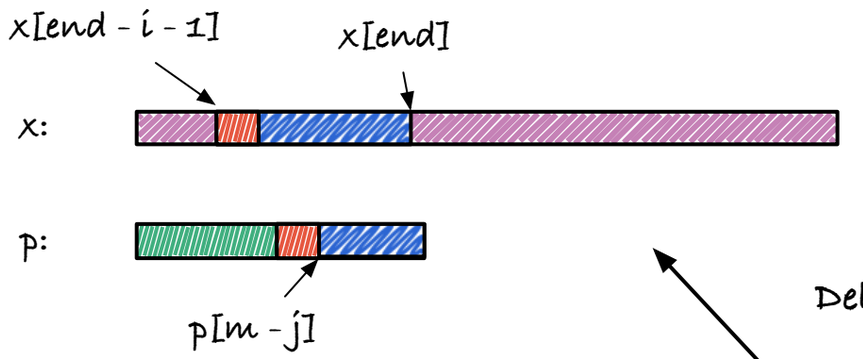
…from this point we don't have any more mismatches…



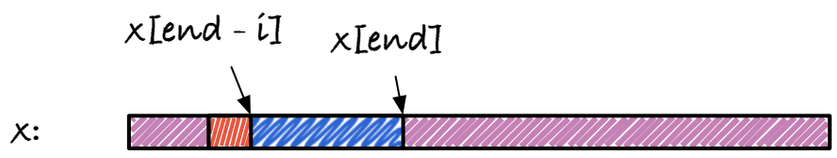
Search for ATT allowing at most one mismatch

Read mapping

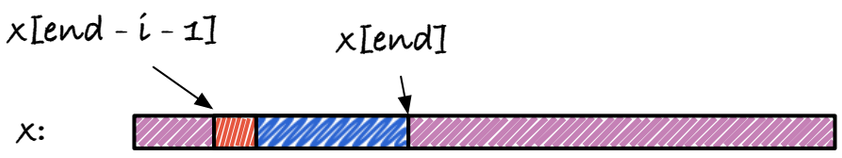
Approximative matching using BWT

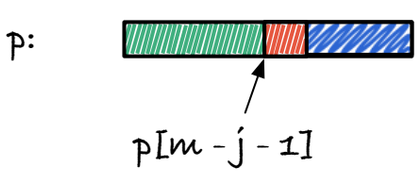


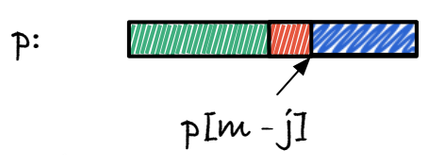
Deletion



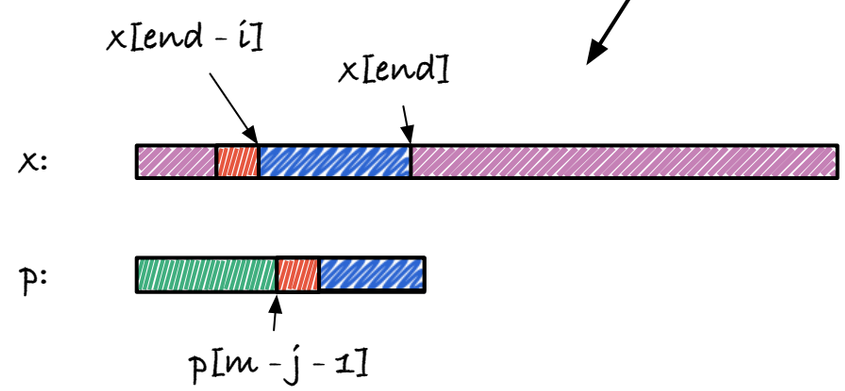
Match/Substitution







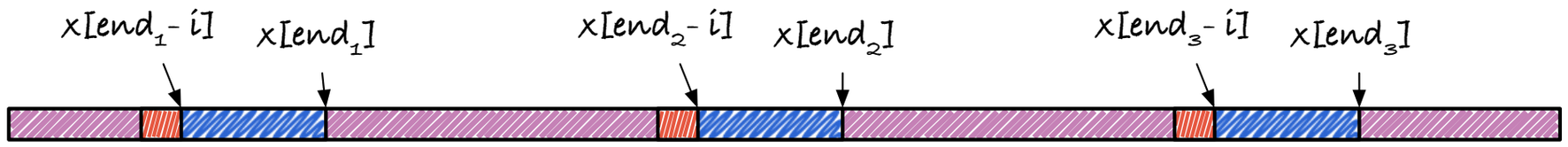
Insertion

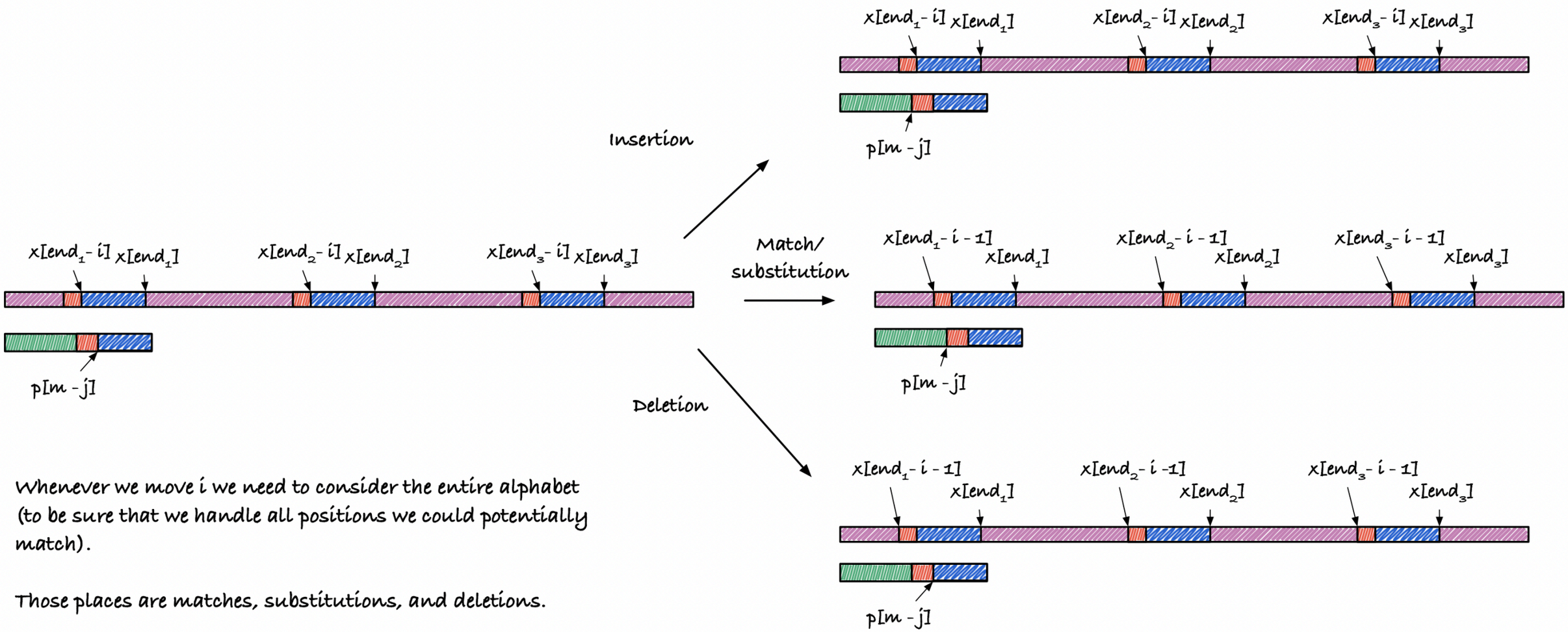


If we search with BWT, we search from multiple different end points.

The cost of matches/substitutions will differ from location to location.

The way we search will, based on characters that jump around the tables,
means that we also need to consider multiple characters for insertion.





Whenever we move i we need to consider the entire alphabet (to be sure that we handle all positions we could potentially match).

Those places are matches, substitutions, and deletions.

```
uint32_t new_L;
uint32_t new_R;

// M-operations
unsigned char match_a = pattern[i];
// Iterating alphabet from 1 so I don't include the sentinel.
for (unsigned char a = 1; a < alphabet_size; ++a) {

    new_L = C(a) + O(a, L);
    new_R = C(a) + O(a, R);


    int edit_cost = (a == match_a) ? 0 : 1;
    if (edits - edit_cost < 0) continue;
    if (new_L >= new_R) continue;

    *cigar = 'M';
    rec_approx_matching(iter, new_L, new_R, i - 1,
                        edits - edit_cost,
                        cigar + 1);
}

// I-operation
*cigar = 'I';
rec_approx_matching(iter, L, R, i - 1, edits - 1, cigar + 1);

// D-operation
*cigar = 'D';
for (unsigned char a = 1; a < alphabet_size; ++a) {
    new_L = C(a) + O(a, L);
    new_R = C(a) + O(a, R);
    if (new_L >= new_R) continue;

    rec_approx_matching(iter, new_L, new_R, i,
                        edits - 1, cigar + 1);
}
```

The D table

- We have a branch and bound search algorithm but we only stop the recursion when we get to the point where we have used too many edits.

- If we have a lower bound on the edits we must use to continue at every step we can quick early.

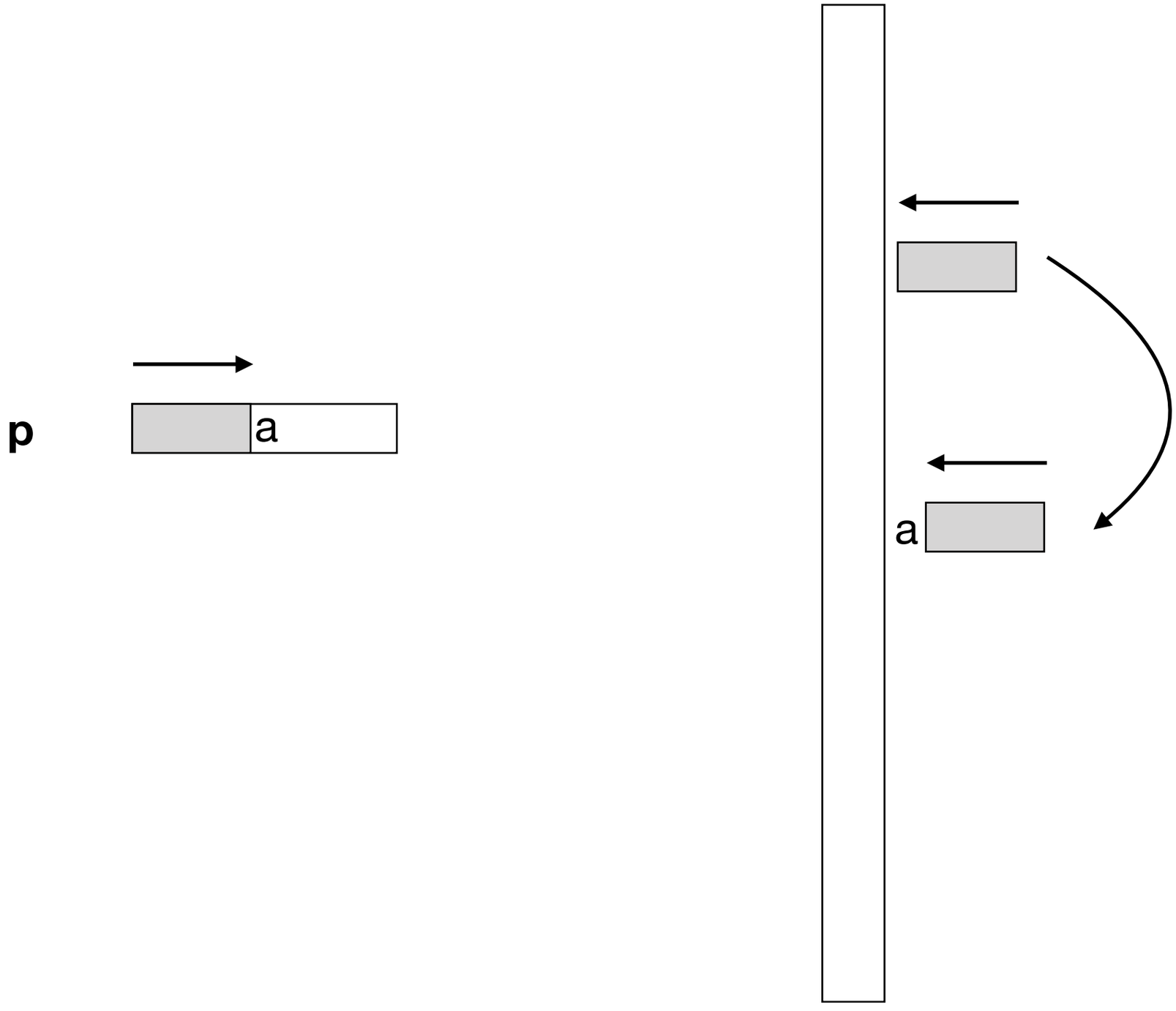
- The D table gives us such a lower bound.

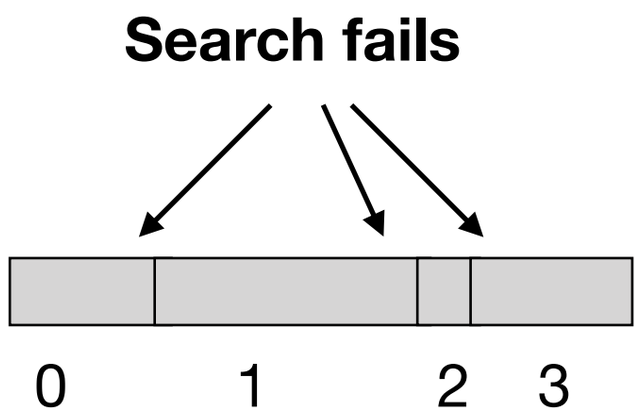
The D table

- We work out how many edits are needed in a prefix of the pattern. We do this by exact matching the reversed pattern against the suffix array of the reversed string.

- When we cannot continue a match we know that at least one edit is needed, so we record that.

- Then we continue with a complete attempt at matching the remaining pattern.


p
a
a


Search fails
0
1
2
3

```
int min_edits = 0;
uint32_t L = 0, R = sa->length;
for (uint32_t i = 0; i < m; ++i) {
    uint8_t a = pattern[i];
    L = C(a) + RO(a, L);
    R = C(a) + RO(a, R);
    if (L >= R) {
        min_edits++;
        L = 0;
        R = sa->length;
    }
    iter->D_table[i] = min_edits;
}
```

RO is the O table
for the reversed suffix array

In the search we break before we reach the edit distance zero if we can see that it is impossible to get to the beginning of the pattern with enough edits left.

```
int lower_limit = iter->D_table[i];
if (edits  < lower_limit) {
      return; // we can never get a match from here
}
```

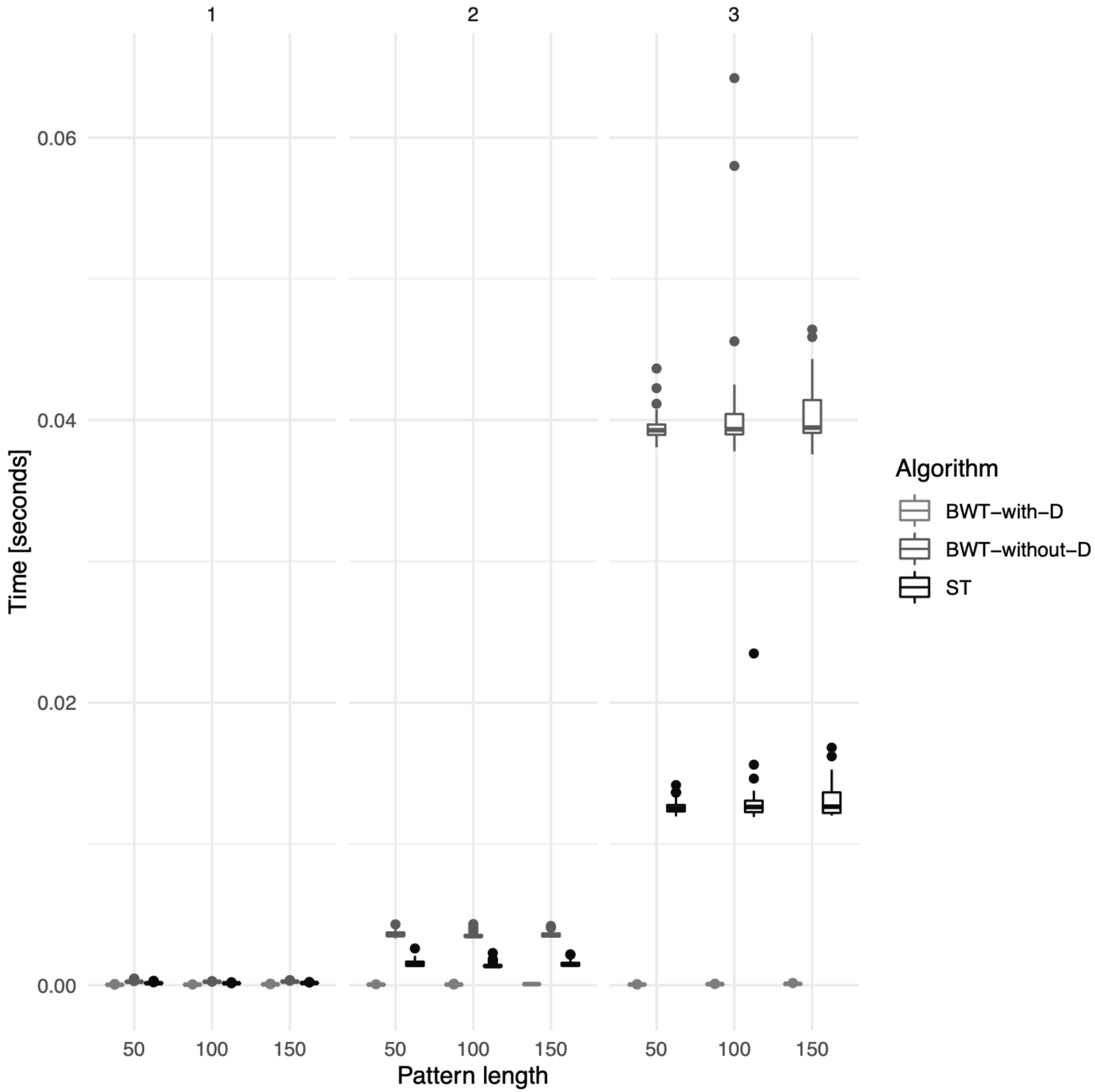


Figure 82: Comparison of approximative search algorithms

Conclusions

- Approximative matching is an important problem in bioinformatics
- One example — with a lot of biological applications — is short read mapping
    - We have seen two efficient approaches to do approximative matching by exhaustively searching for possible matches

That's all Folks!