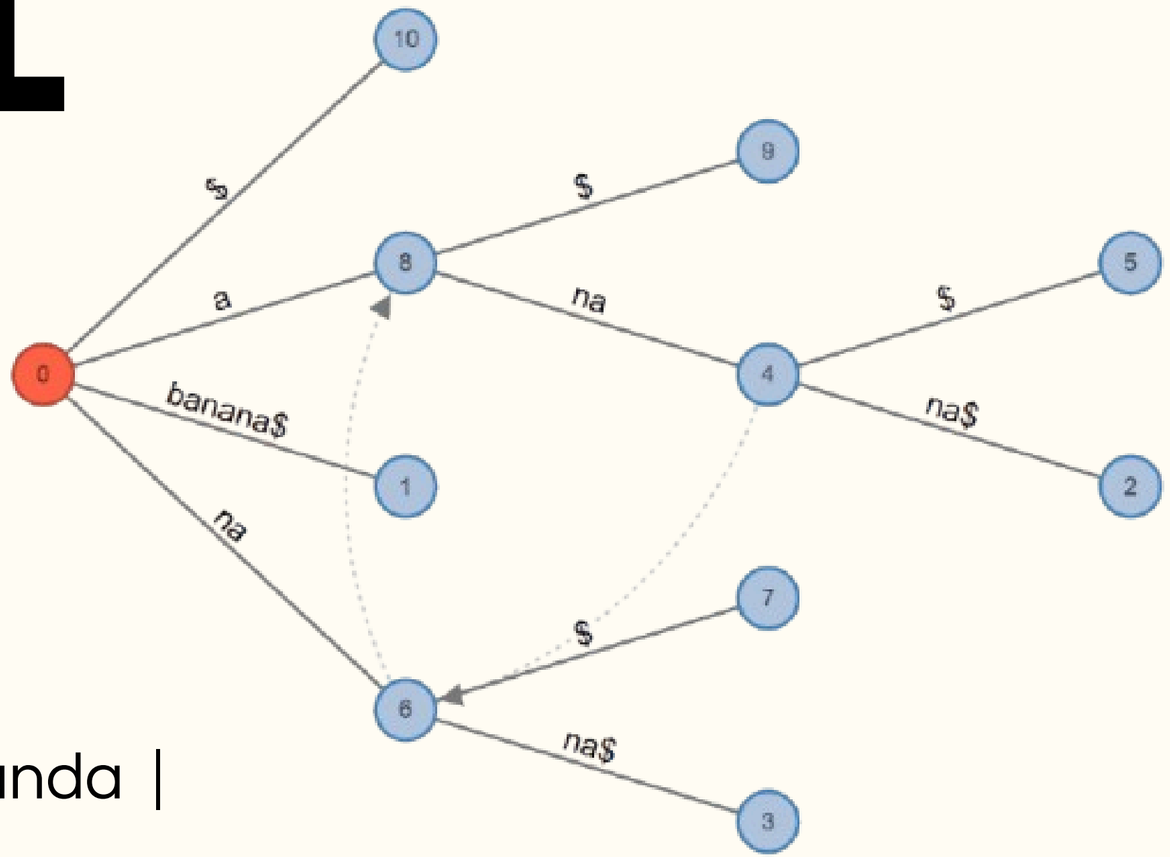
Algoritmo y Estructuras de Datos

SUFFIX TREE

& Ukkonen

& GST

& Genomas



INTEGRANTES:

| Elmer Villegas | Jose Huaman | Paulo Miranda |

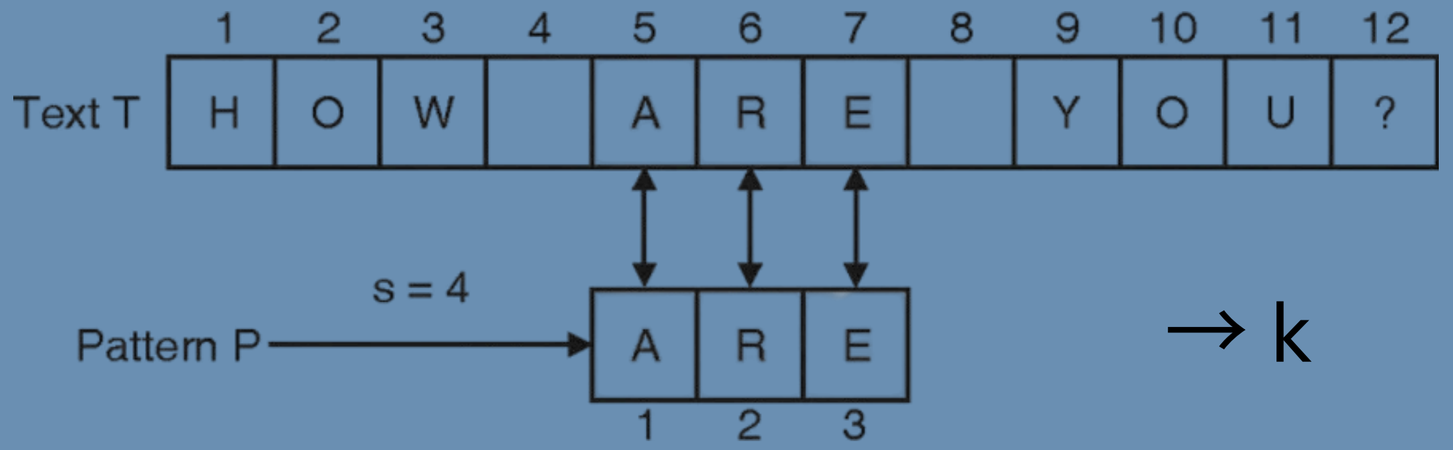
[your name here]

https://github.com/PauloMiraBarr/AED_SuffixTree

https://brenden.github.io/ukkonen-animation/

Introducción a los Suffix Trees

Problemas de optimización en cadenas...



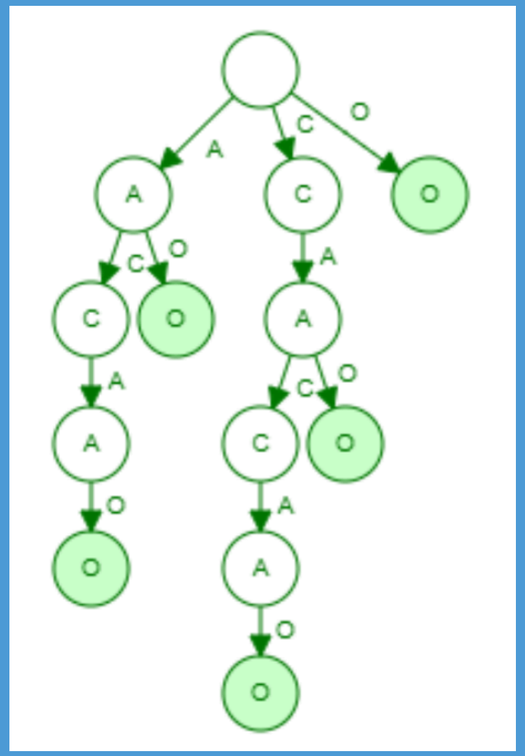
→ n

→ k

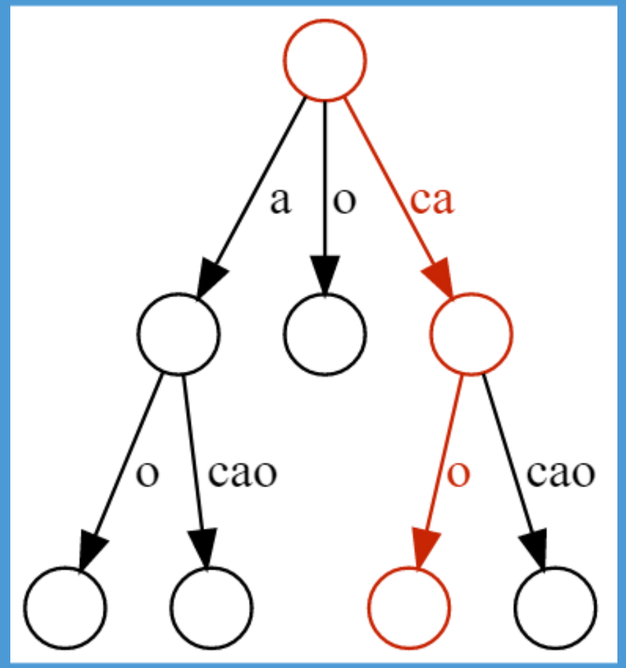
→ **Fuerza bruta**: O(nk)

No es malo, pero, ¿se puede mejorar?

De Suffix Trie a Suffix Tree



Compresión



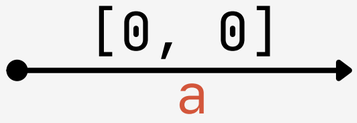
Muy lindo y todo pero,

¿y lo bueno?

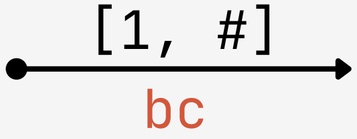
La construcción sigue siendo de tiempo cuadrático...

Trucos de Optimización

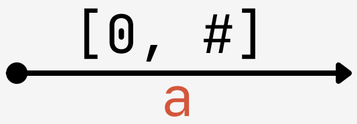
Truco 1: ¿Y si las aristas guardan índices en vez de caracteres?

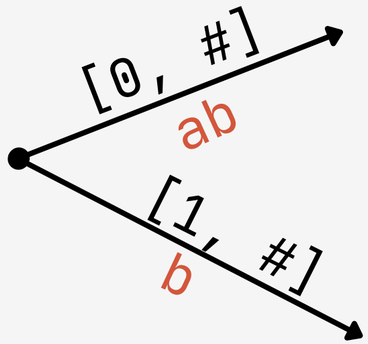


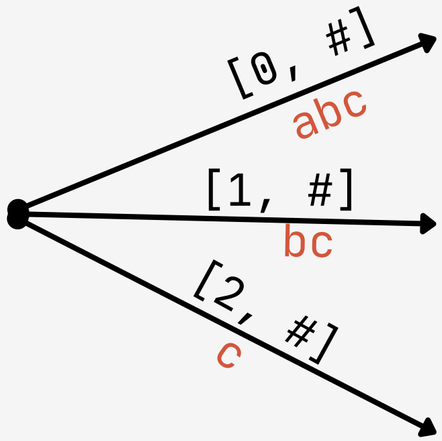
Truco 2: ¿Y si todos los que son aristas a hojas tienen el mismo puntero final? (***Regla 1***)




```
          0  1  2
s  =  a  b  c
```







Ukkonen formula 3 reglas, una ya la vimos

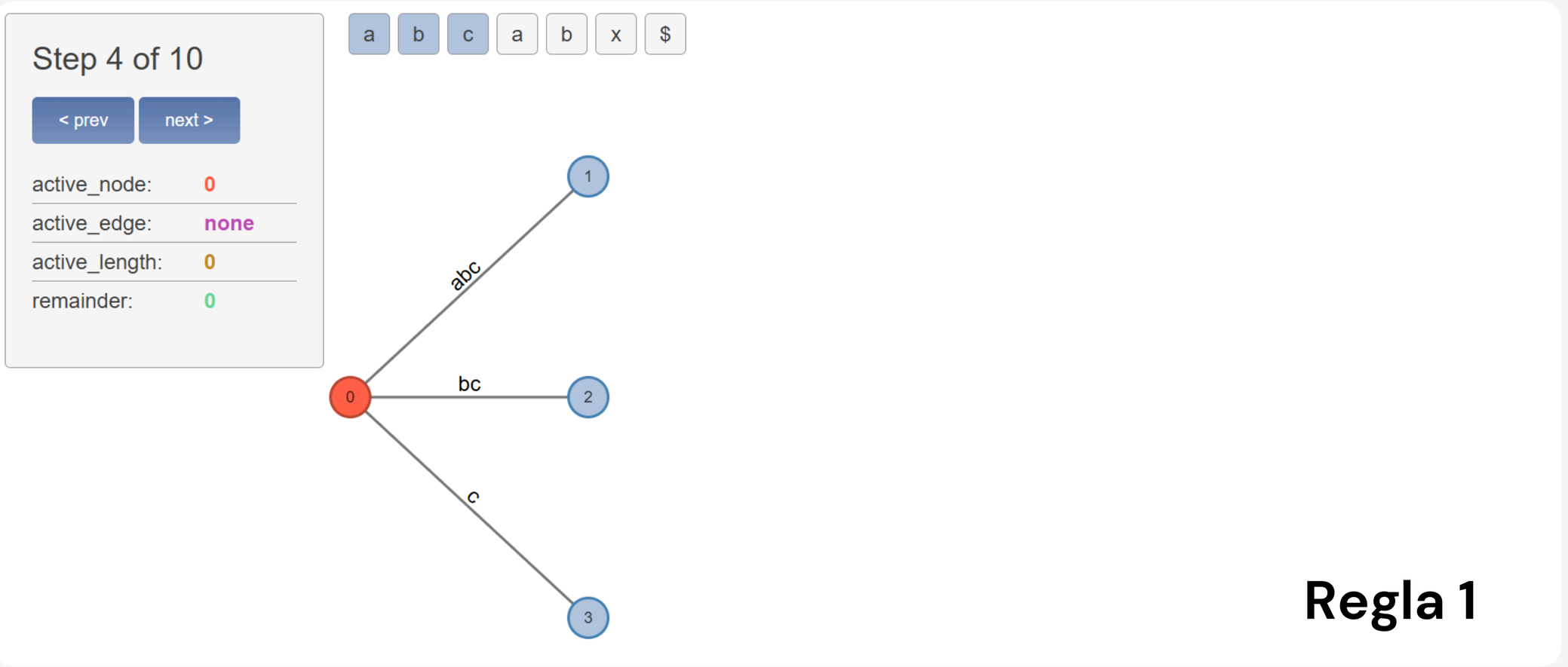
Regla 2 (split)

Si el caracter no coincide con el actual de la arista, se parte creando un nodo hoja que representa el sufijo en inserción. Esta es la regla que nos introduce los suffix links.

Regla 3 (stop rule)

Si el carácter que vamos a insertar ya está representado en el árbol en la posición donde debería ir, implica extensión *implícita*, por lo que se detienen las extensiones y creaciones.

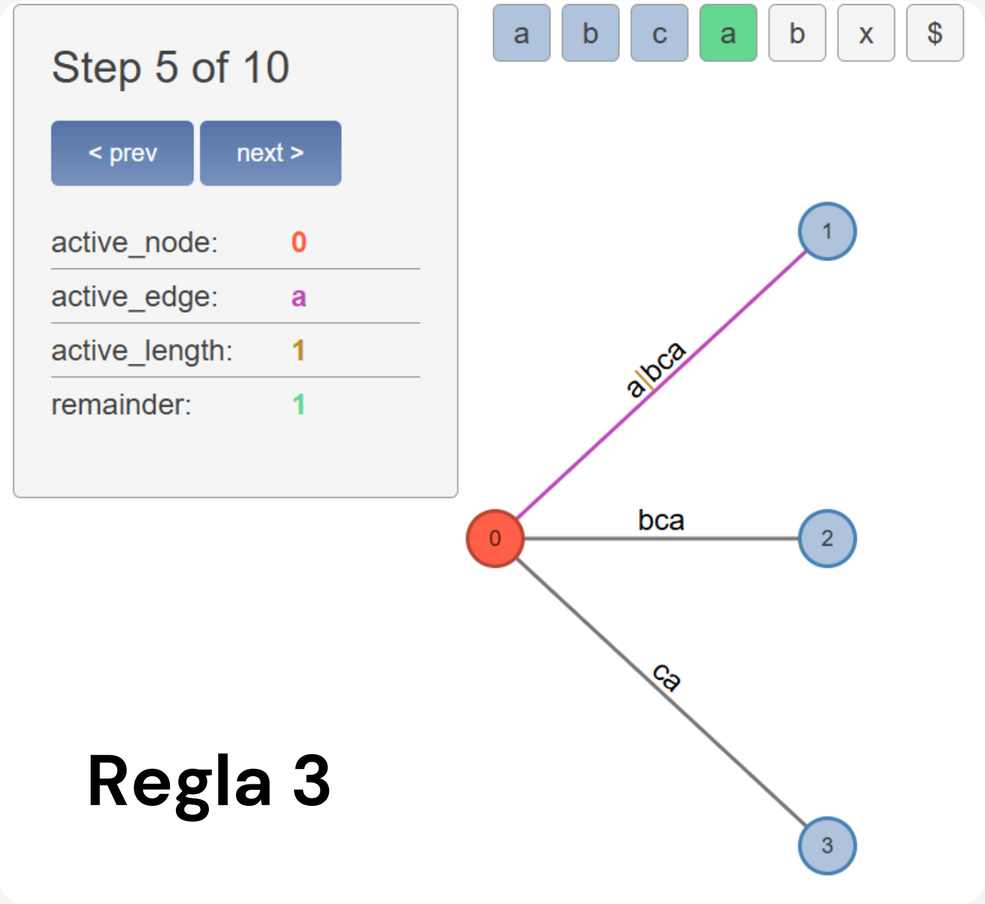
Ejemplo con la cadena *abcabx*



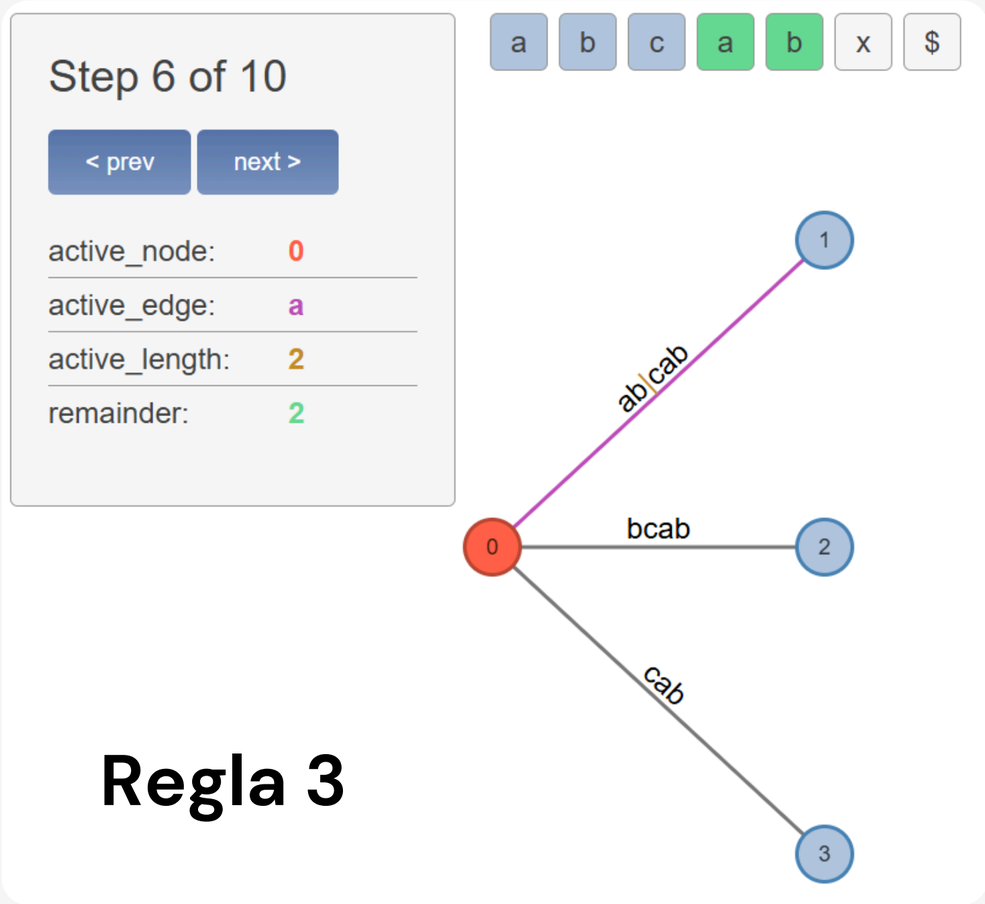
Regla 1

https://brenden.github.io/ukkonen-animation/

Ejemplo con la cadena *abcabx*



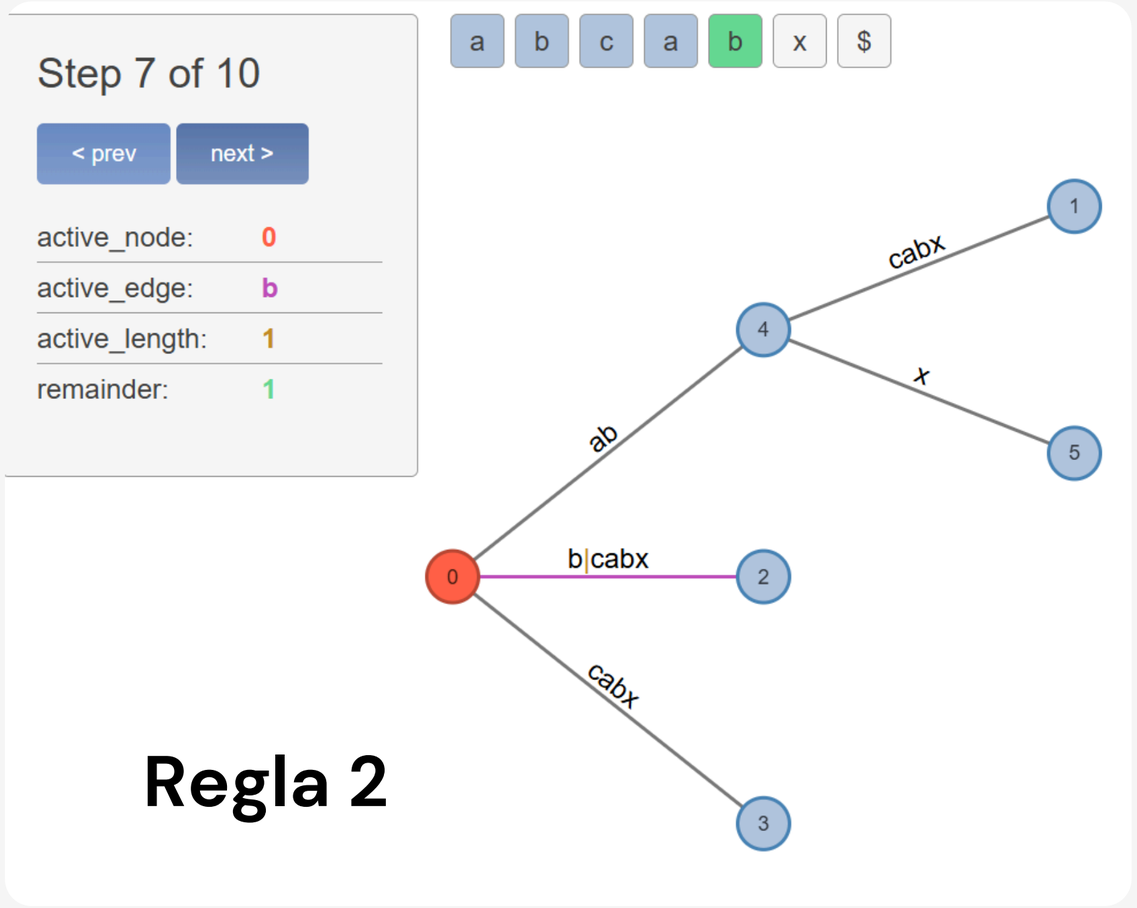
Regla 3



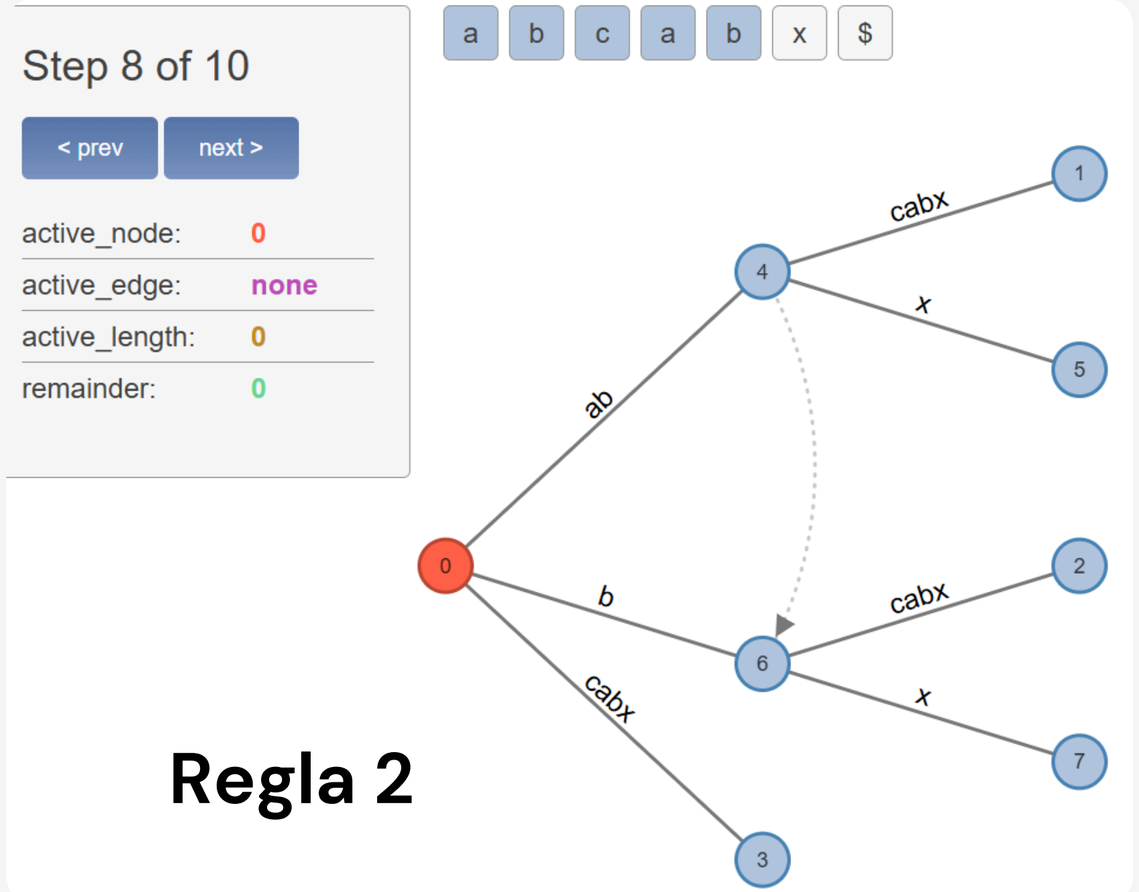
Regla 3

https://brenden.github.io/ukkonen-animation/

Ejemplo con la cadena *abcabx*



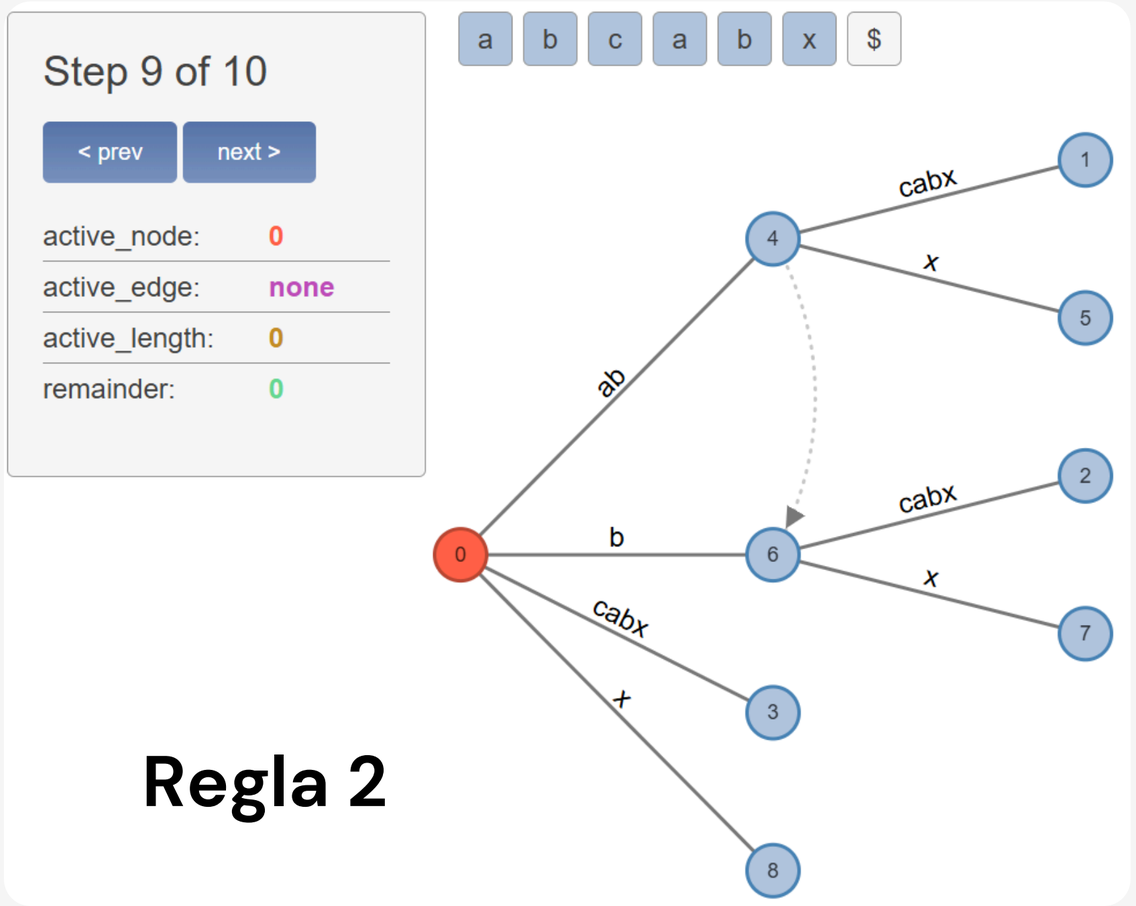
Regla 2



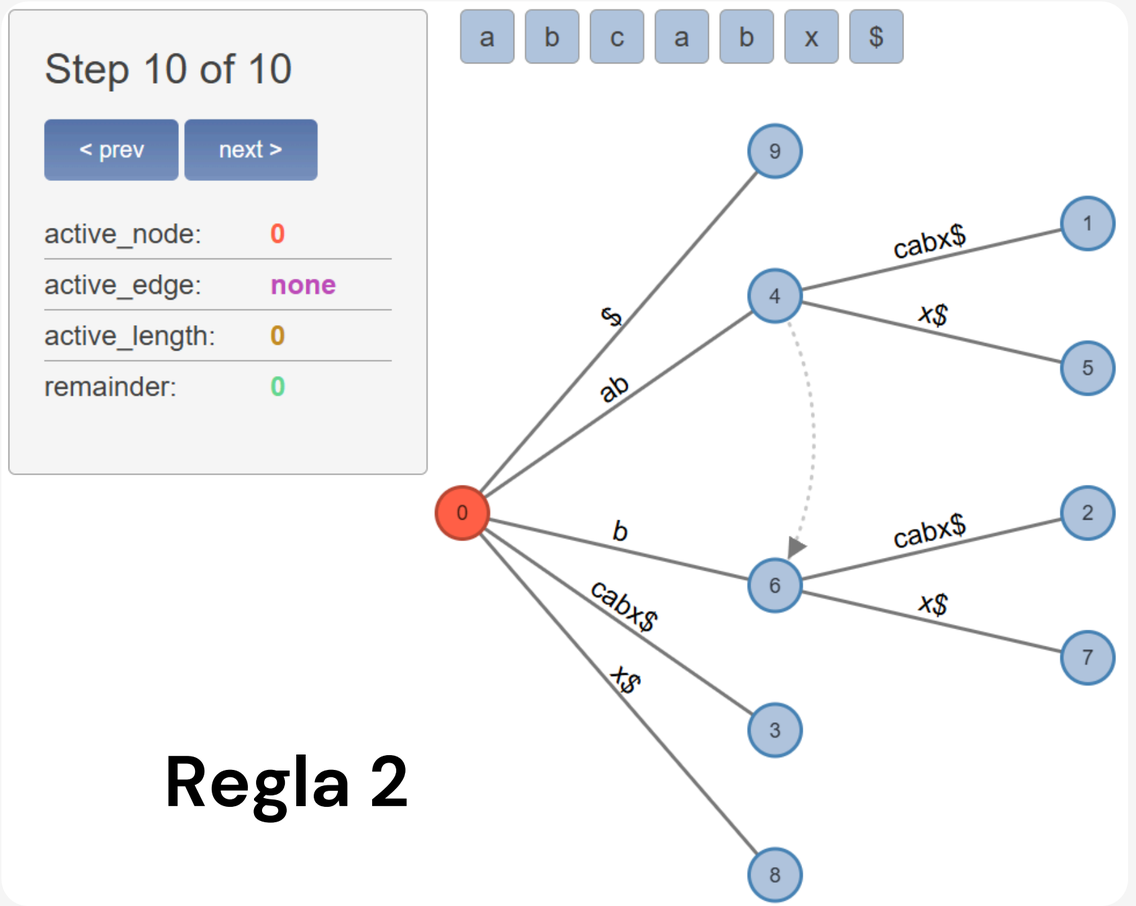
Regla 2

https://brenden.github.io/ukkonen-animation/

Ejemplo con la cadena *abcabx*



Regla 2



Regla 2

https://brenden.github.io/ukkonen-animation/

Resumen de complejidad

Operación	Objetivo	Complejidad
test and split	Dividir una arista (regla 2)	O(1)
reference	Realizar funciones equivalentes active edge	Mem: O(1)
find alpha	Se encarga de hallar la arista por la cual propagar	O(1)
add substring	Consume las demás funciones para general el suffix	O(n)

```
struct Node {
    std::unordered_map<char, Transition> g;
    Node* suffix_link;
    ColorSet colors;
    Node();
    virtual ~Node();
    virtual Transition find_alpha_transition(char alpha);
    void mark_string(int string_id);
    bool has_single_string() const;
    int get_single_string_id() const;
    void merge_colors(const ColorSet& other);
};
```

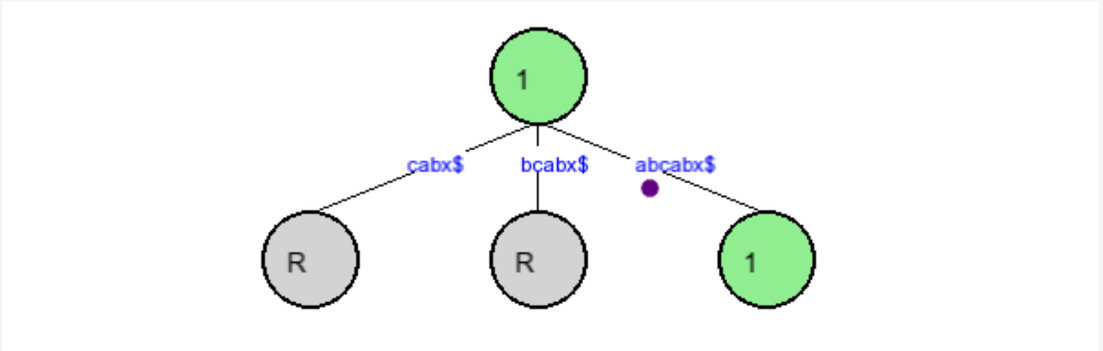
```
struct MappedSubstring {
    int ref_str;
    Index l;
    Index r;
    MappedSubstring();
    MappedSubstring(int ref, Index left, Index right);
    bool empty() const;
    int lenght() const;
};

struct Transition {
    MappedSubstring sub;
    Node* tgt;
    Transition();
    Transition(MappedSubstring s, Node* t);
    bool is_valid() const;
};
```

```
struct ReferencePoint {
    Node* node;
    int ref_str;
    Index pos;
    ReferencePoint(Node* n, int ref, Index p);
};
```

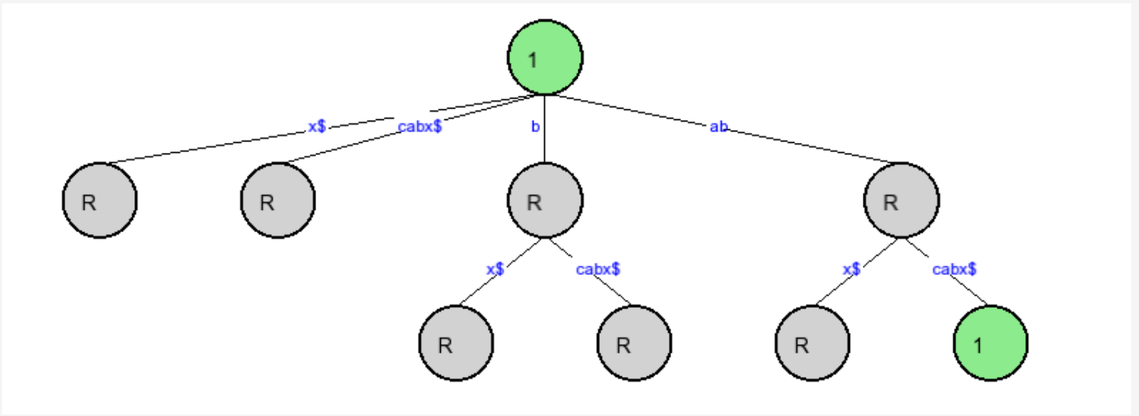
```
// METODOS AUXILIARES
std::string substring_to_string(const MappedSubstring& substr) const;
bool test_and_split(Node* n, MappedSubstring kp, char t, const std::string& w, Node** r);
ReferencePoint update(Node* n, MappedSubstring ki);
ReferencePoint canonize(Node* n, MappedSubstring kp);
Index get_starting_node(const std::string &s, ReferencePoint* r);
int deploy_suffixes(const std::string& s, int sindex);
bool contains_end_token(const std::string& str) const;
```

```
SuffixTree();
int add_string(const std::string &str);
bool is_suffix(const std::string& str);                                    ';
bool is_substring(const std::string& str);
int get_string_count() const;
std::string get_string(int id) const;
void compute_colors();
```

```cpp
bool SuffixTree::test_and_split(Node* n, MappedSubstring kp,
    char t, const std::string& w, Node** r)
{
    Index delta = kp.r - kp.l;
    if(delta < 0) {
        *r = n;
        Transition t_trans = n->find_alpha_transition(t);
        if (t_trans.is_valid() and t_trans.tgt != nullptr and kp.ref_str > 0) {
            t_trans.tgt->mark_string(kp.ref_str);
        }
        return t_trans.is_valid();
    }
    char tk = w[kp.l];
    Transition tk_trans = n->find_alpha_transition(tk);
    MappedSubstring kp_prime = tk_trans.sub;
    auto str_it = haystack.find(kp_prime.ref_str);
    const std::string& str_prime = str_it->second;
    if (str_prime[kp_prime.l + delta + 1] == t) {
        *r = n;
        return true; // es endpoint
    }
```




```cpp
    // Crear nuevo nodo intermedio
    *r = new Node();
    Transition new_trans = tk_trans;
    new_trans.sub.l += delta + 1;
    (*r)->g.insert({str_prime[new_trans.sub.l], new_trans});
    tk_trans.sub.r = tk_trans.sub.l + delta;
    tk_trans.tgt = *r;
    n->g[tk] = tk_trans;
    colors_computed = false;
    return false;
}
```

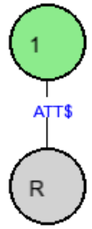


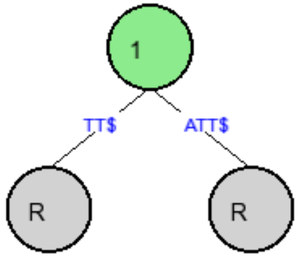
```
ReferencePoint SuffixTree::update(Node* n, MappedSubstring ki) {
    Node* oldr = &tree.root;
    Node* r = nullptr;
    bool is_endpoint = false;
    const std::string& w = haystack[ki.ref_str];
    MappedSubstring ki1 = ki;
    ki1.r = ki.r - 1; // Excluir el último carácter
    ReferencePoint sk(n, ki.ref_str, ki.l);
    // Probar y dividir en el punto actual
    is_endpoint = test_and_split(n, ki1, w[ki.r], w, &r);
    while (!is_endpoint) {
        // Crear nueva hoja
        Leaf* r_prime = new Leaf();
        // Agregar transición al nodo actual
        // El substring va desde ki.r hasta el infinito (representado con max)
        r->g.insert({w[ki.r],
                     Transition(MappedSubstring(ki.ref_str, ki.r,
                                std::numeric_limits<Index>::max()),
                                r_prime)});
        // Actualizar suffix links
        if (oldr != &tree.root) {
            oldr->suffix_link = r;
        }
        oldr = r;

        // Seguir el suffix link
        sk = canonize(sk.node->suffix_link, ki1);
        ki1.l = ki.l = sk.pos;

        // Probar y dividir en el nuevo punto
        is_endpoint = test_and_split(sk.node, ki1, w[ki.r], w, &r);
    }
    // Actualizar último suffix link
    if (oldr != &tree.root) {
        oldr->suffix_link = sk.node;
    }
return sk;
```

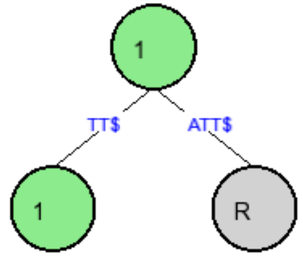
```
ReferencePoint SuffixTree::canonize(Node* n, MappedSubstring kp) {
    if (kp.r < kp.l) {
        // Substring vacío, ya es canónico
        return ReferencePoint(n, kp.ref_str, kp.l);
    }

    const std::string& str = haystack[kp.ref_str];

    Transition tk_trans = n->find_alpha_transition(str[kp.l]);
    Index delta;

    // Mientras el substring sea más largo que la arista actual
    while ((delta = tk_trans.sub.r - tk_trans.sub.l) <= kp.r - kp.l) {
        kp.l += 1 + delta;
        n = tk_trans.tgt;

        if (kp.l <= kp.r) {
            tk_trans = n->find_alpha_transition(str[kp.l]);
        }
    }

    return ReferencePoint(n, kp.ref_str, kp.l);
}
```

```
int SuffixTree::deploy_suffixes(const std::string& s, int sindex) {
    ReferencePoint active_point(&tree.root, sindex, 0);

    Index i = get_starting_node(s, &active_point);

    if (i == std::numeric_limits<Index>::max()) {
        return -1;
    }

    for (; i < s.size(); ++i) {
        MappedSubstring ki(sindex, active_point.pos, i);
        active_point = update(active_point.node, ki);
        ki.l = active_point.pos;
        active_point = canonize(active_point.node, ki);
    }
```

Ejemplo con la cadena *ATT* y *ATC*

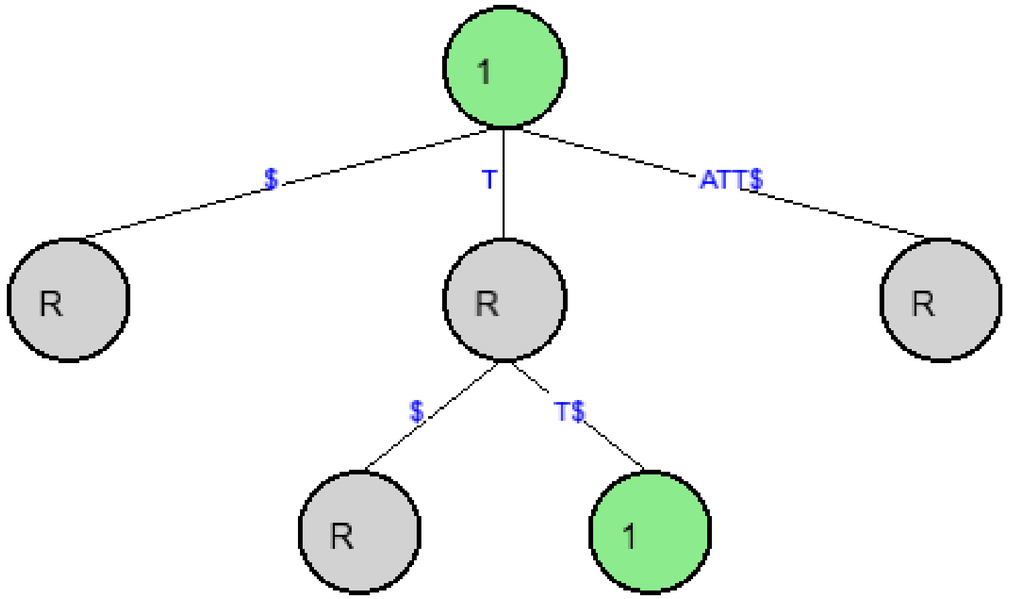






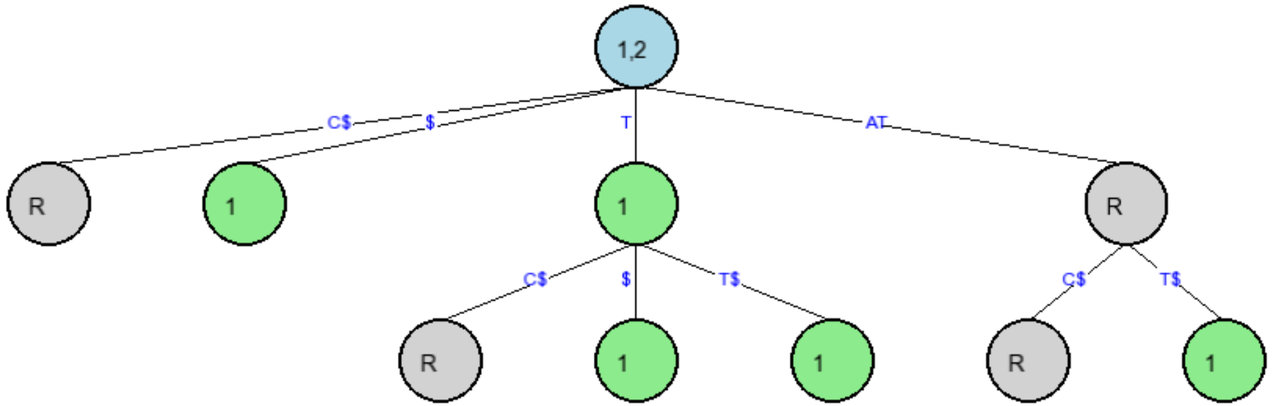
Elaboración propia

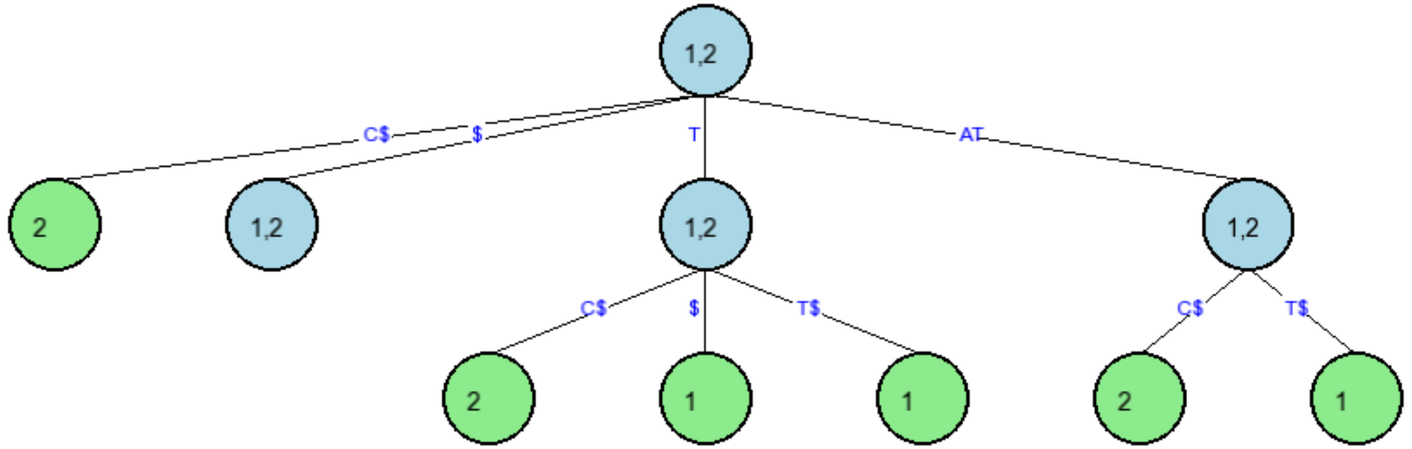
Ejemplo con la cadena *ATT* y *ATC*



Elaboración propia

Ejemplo con la cadena *ATT* y ***ATC***





Elaboración propia

Ejemplo de uso en Genomática (por refencias)