LARGE PARSIMONY

INF589 - Final project

Julien Gadonneix, Rémi Pommé, Inès Fonquernie

TABLE OF CONTENTS

How could we address the large parsimony problem starting from that of small parsimony ?

DEFINITIONS

Small parsimony

Goal : find the **most parsimonious labels for the tree's internal nodes**. We already have a rooted phylogenetic tree, and each leaf is labelled with a sequence.

Large parsimony

Goal : find an **unrooted tree** that **minimizes the parsimony score**, i.e. the minimum number of evolutionary changes needed to explain the differences between the observed sequences.

Same problem on a larger scale

Input : Collection of strings of equal length
Output : Unrooted binary tree that minimizes the parsimony score among all possible

GENERATION OF INPUTS - IMPLEMENTATION

```
def generate_seq_aligned_naive(n,k):
     alignment = []
     bases = ["A", "C", "T", "G"]
     epsilon_indel = 0.05
     epsilon_mut = 0.1
     seq = ""
     for _ in range(n):
          if np.random.rand()<epsilon_indel:
               seq+="-"
          else :
               i = np.random.randint(4)
               seq+=bases[i]
               alignment.append(seq)
     for _ in range(k-1):
          seq = ""
          for j in range(n):
               if np.random.rand()<epsilon_indel:
                    seq = seq + "-"
               elif np.random.rand()<epsilon_mut :
                    i = np.random.randint(4)
                    seq = seq+ bases[i]
               else :
                    seq= seq + alignment[0][j]
          alignment.append(seq)
     return alignment
```

Input n : the length of the desired alignment
k : the number of sequences in the alignment

Output alignment : an array containing all the sequences aligned

```
def generate_seq_unaligned_naive(n,k):
     alignment = []
     bases = ["A", "C", "T", "G"]
     epsilon_mut = 0.05
     seq = ""
     sigma = n//8
     length = int(np.random.normal(loc = n, scale
     = sigma))
     for _ in range(length):
          i = np.random.randint(4)
          seq+=bases[i]
          alignment.append(seq)
     for _ in range(k-1):
          seq = ""
          length = int(np.random.normal(loc = n,
          scale = sigma))
          for j in range(length):
               if np.random.rand()<epsilon_mut or
               j >= len(alignment[0]):
                    i = np.random.randint(4)
                    seq = seq+ bases[i]
               else :
                    seq= seq + alignment[0][j]
          alignment.append(seq)
     return alignment
```

Input n : the approximated length
k : the number of sequences

Output alignment : an array containing all the sequences unaligned

GENERATION OF INPUTS - IMPLEMENTATION

```
def generate_seq_aligned(n,k):
      alignment = []
      bases = ["A", "C", "T", "G"]
      epsilon_indel = 0.05
      epsilon_mut = 0.1
      seq = ""
      for _ in range(n):
            if np.random.rand()<epsilon_indel:
                  seq+="-"
                  if epsilon_indel == 0.05:
                        epsilon_indel = 0.4
                  else :
                        epsilon_indel = epsilon_indel * 0.9
            else :
                  if epsilon_indel != 0.05:
                        epsilon_indel = 0.05
                  i = np.random.randint(4)
                  seq+=bases[i]
                  alignment.append(seq)
      for _ in range(k-1):
            seq = ""
            for j in range(n):
                  if np.random.rand()<epsilon_indel:
                        seq = seq + "-"
                        if epsilon_indel == 0.05:
                              epsilon_indel = 0.4
                        else :
                              epsilon_indel = epsilon_indel * 0.9
                  elif np.random.rand()<epsilon_mut
            or(alignment[0][j]=="-" and np.random.rand()<0.7):
                        if epsilon_indel != 0.05:
                              epsilon_indel = 0.05
                        i = np.random.randint(4)
                        seq = seq+ bases[i]
                        else :
                              if epsilon_indel != 0.05:
                                    epsilon_indel = 0.05
                        seq= seq + alignment[0][j]
            alignment.append(seq)
      return alignment
```

generate_seq_aligned **dynamically adjusts the indel probability** based on recent events.

- This approach attempts to simulate some form of **dependency between indel events**, unlike *generate_seq_aligned_naive*, which uses a fixed probability for indels and mutations at each position.

Example of alignment obtained with *generate_seq_aligned*

'TTGAGAACACTAGT-'
'TTGAGAACGCTAGTA'
'TTTAGAA--CTAGT-'
'GTGAGAACACTAGTC'

Example of alignment obtained with *generate_seq_aligned_naive*

'-TGTAATCACTGTAC'
'-TGTAATTACTGTAC'
'-TGTAATCACTTTA-'
'-TGTAATCACT-TA-'

Example of alignment obtained with *generate_seq_unaligned_naive*

'GAGGACATAGCACATTCAAC'
'GAGGACATAGCACATTCAACTG'
'GAGAACATAGTACATCCAAC'
'GAGGACATAGCACATTCAACG'

GENERATION OF INPUTS - IMPLEMENTATION

```python
def extend_or_truncate_sequence(reference_seq, target_length):
    seq_len = len(reference_seq)
    if seq_len < target_length:
        factor = (target_length + seq_len - 1) // seq_len
        extended_seq = (reference_seq *
        factor)[:target_length]
    else:
        extended_seq = reference_seq[:target_length]
    return extended_seq


def generate_similar_sequences(reference_seq, num_sequences,
mutation_rate, target_length):
    extended_ref_seq =
    extend_or_truncate_sequence(reference_seq, target_length)
    sequences = [extended_ref_seq]
    for _ in range(num_sequences - 1):
        new_seq = list(extended_ref_seq)
        for i in range(target_length):
            if random.random() < mutation_rate:
                bases = ['A', 'C', 'G', 'T']
                bases.remove(extended_ref_seq[i])
                new_base = random.choice(bases)
                new_seq[i] = new_base
        sequences.append(Seq(''.join(new_seq)))

    return sequences
```

extend_or_truncate_sequence

Input

- reference_seq : sequence of reference
- target_length : target length of the sequence

Output

extended_seq : sequence truncated or extended to reach target length

generate_similar_sequences

Input

- reference_seq : sequence of reference
- num_sequences : number of sequences in the alignment
- mutation_rate : mutation rate between the sequences
- target_length : target length of the sequence

Output

sequences : similar sequences

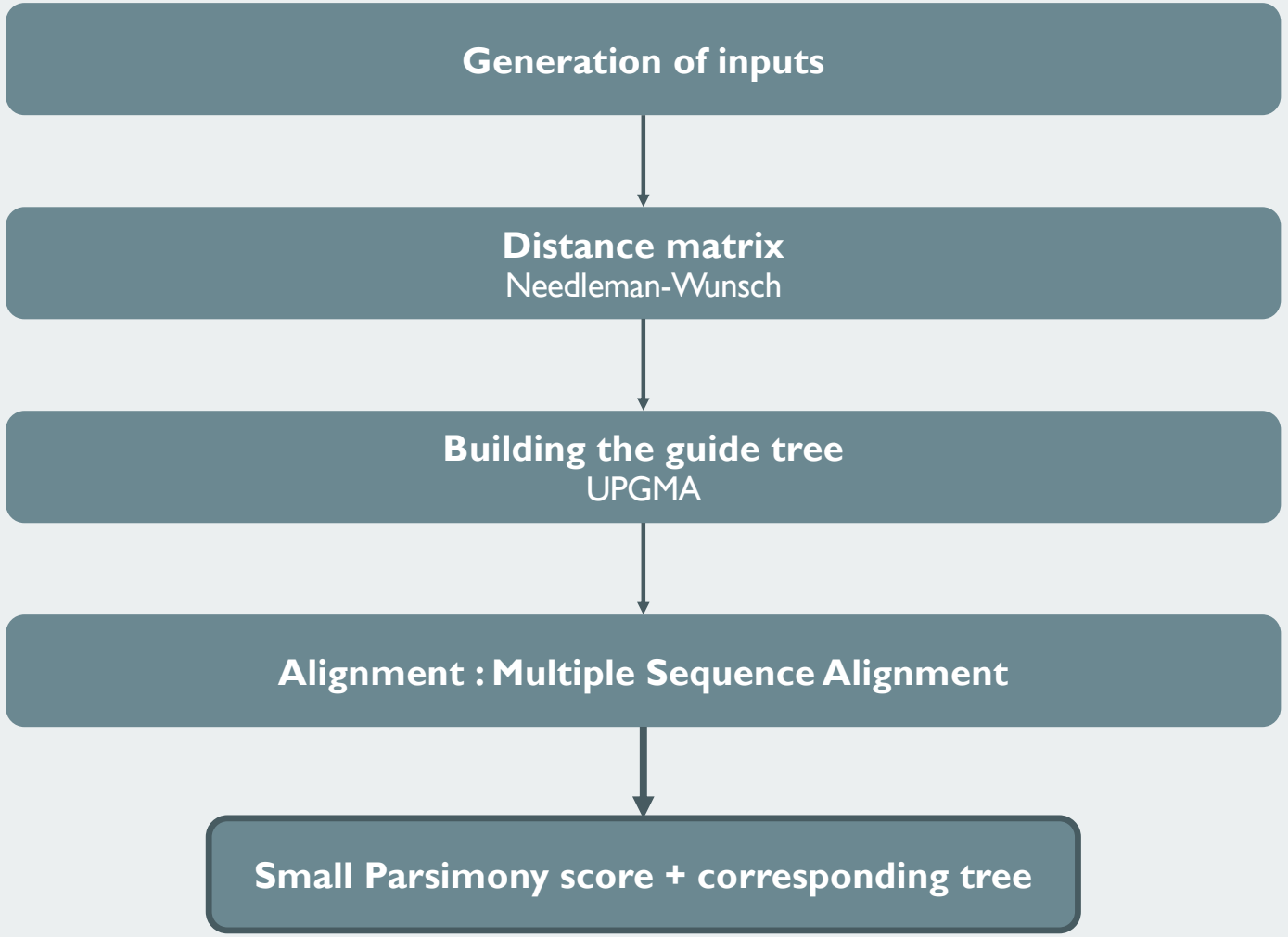
➢ simulate genetic sequences that share a **common origin** but have undergone random mutations over time, while controlling the total length of the sequences generated

```
reference_seq = Seq("ATCGATCGATCGATCG")
num_sequences = 5
mutation_rate = 0.3
target_length = 20
```

→

ATCGATCGATCGATCGATCG
ATCAATTCATCTATCGCACG
ATCGCGCGAACGATCGATCG
ATCGACCGTCAAATAGATCG
CTCAGTCGATTGTTCGCTCG

GENERATION OF INPUTS - IMPLEMENTATION



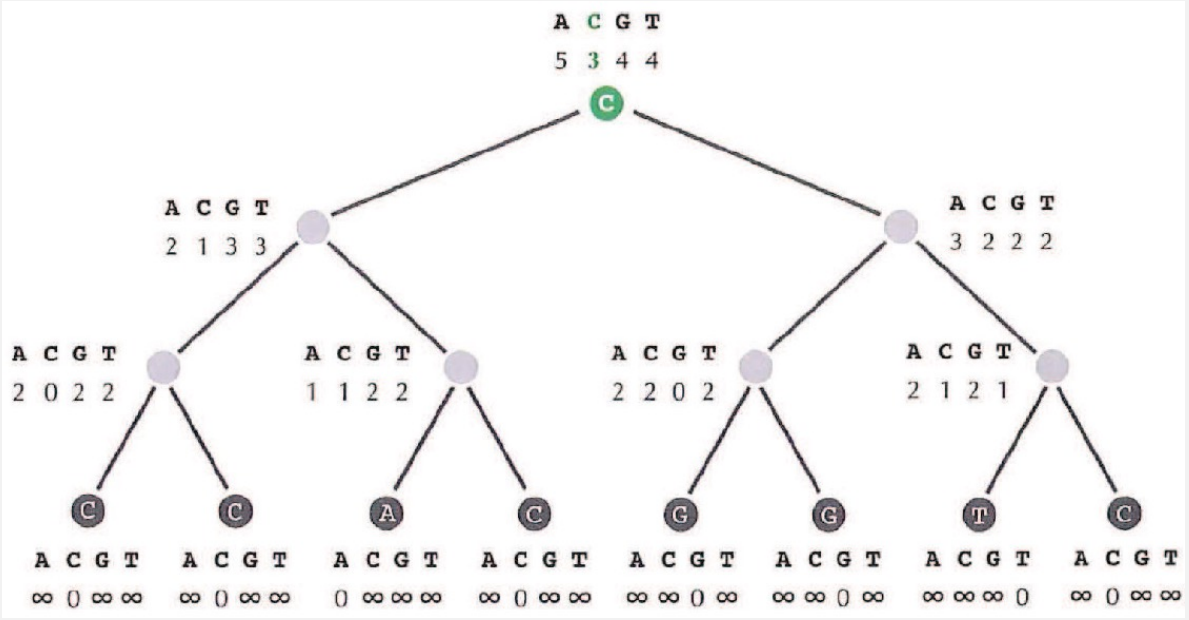
SMALL PARSIMONY STRATEGY

For a given position in the alignment, compute the scores of each base

Repeat this step for each column

Sum the scores of each column

Get the final minimum score (and an ancestor sequence)



The final score depends on the tree considered in the algorithm : it can be optimized

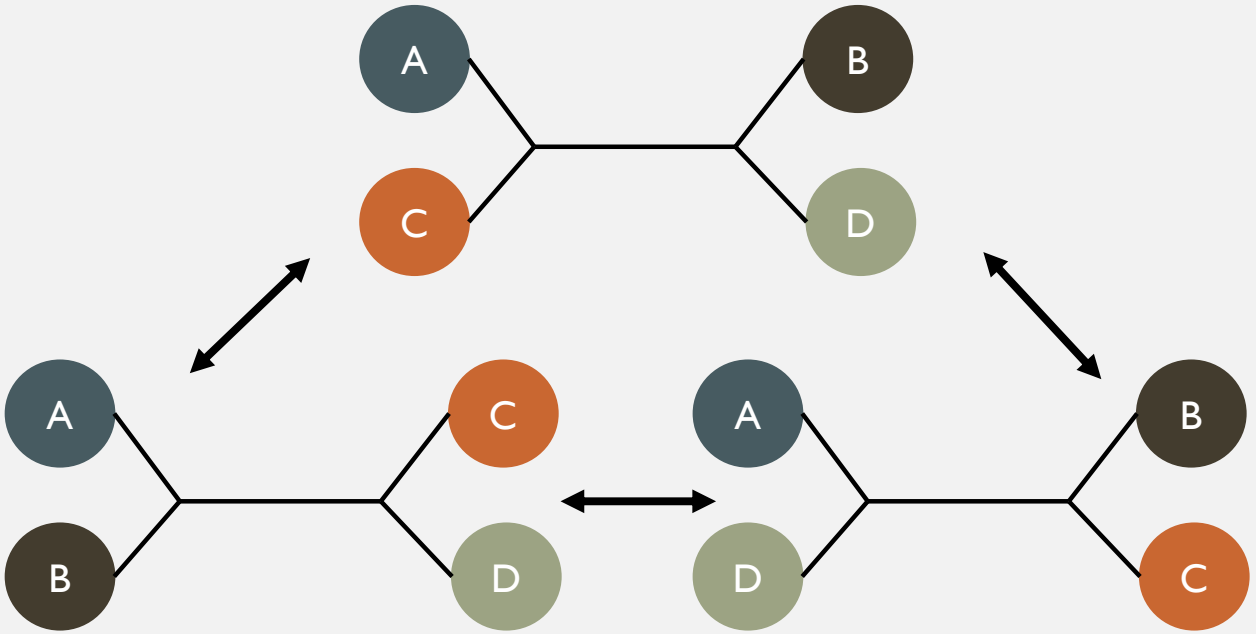
NEAREST NEIGHBOR INTERCHANGE STRATEGY

Random tree

Evaluate it
Small parsimony score

Generate a new one
Nearest neighbor interchange

If no more possible interchange
Return best score and best tree



NEAREST NEIGHBOR INTERCHANGE - IMPLEMENTATION

```
def nni_node_all(node):
     subtrees_ = [node]
     if len(node.children) == 2 and all(len(child.children) == 2 for child in
node.children):
          subtrees_ = []
          children0 = nni_node_all(node.children[0])
          children1 = nni_node_all(node.children[1])
          for child0 in children0:
               for child1 in children1:
                    new_node = copy.deepcopy(node)
                    new_node.children[0] = child0
                    new_node.children[1] = child1
                    subtrees_.append(new_node) #No change : 1st version
                    new_node2 = copy.deepcopy(new_node)
                    new_node2.children[0].children[0],
                    new_node2.children[1].children[0] = child1.children[0],
                    child0.children[0]
                    subtrees_.append(new_node2) #2nd version
                    new_node3 = copy.deepcopy(new_node)
                    new_node3.children[0].children[0],
                    new_node3.children[1].children[1] = child1.children[1],
                    child0.children[0]
                    subtrees_.append(new_node3) #3rd version
     elif len(node.children) ==2 :
          subtrees_ = []
          children0 = nni_node_all(node.children[0])
          children1 = nni_node_all(node.children[1])
          for child0 in children0:
               for child1 in children1:
                    new_node = copy.deepcopy(node)
                    new_node.children[0], new_node.children[1] = child0,
          child1
                    subtrees_.append(new_node)
     return subtrees_
```

Previous steps :

Step 1 : Generate sequences

Step 2 : Build the tree using *guide_tree*

Step 3 : Convert the tree from tuple to an object

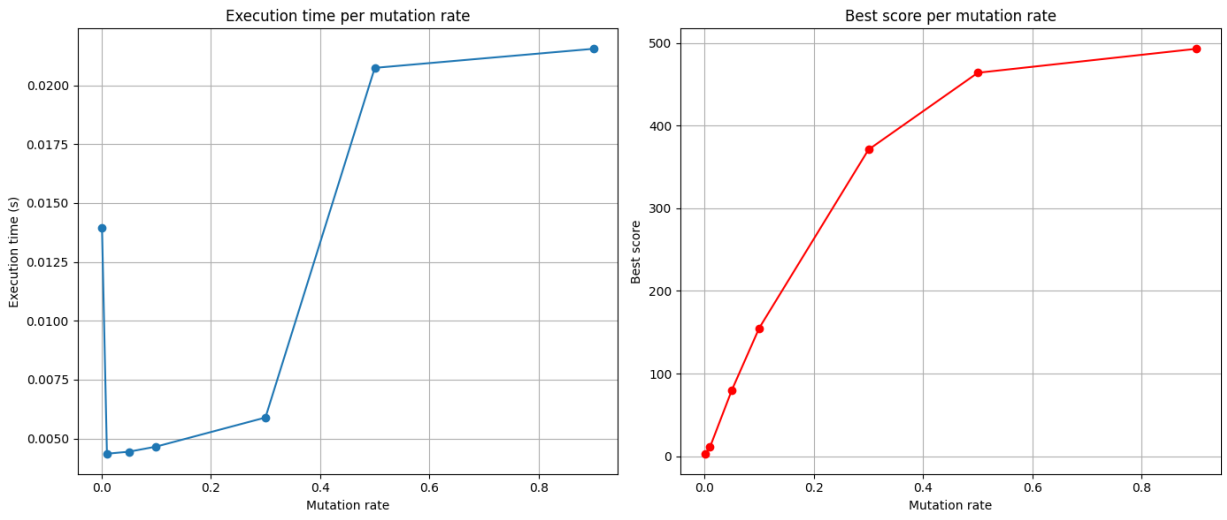
Output : list of all possible subtrees

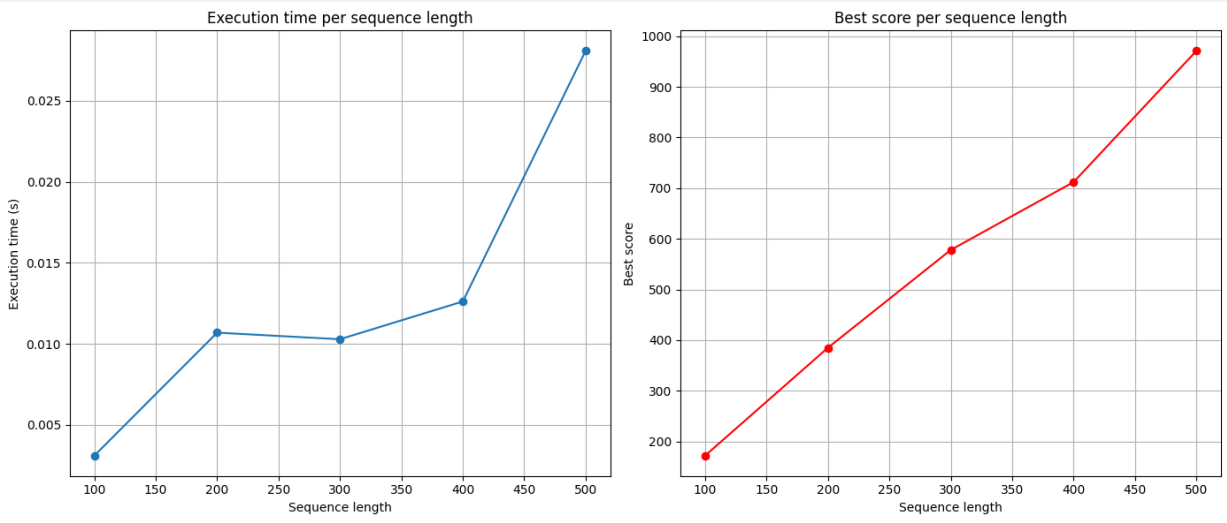
Next steps :

Step 1 : Align the sequences using *progressive_alignment*

Step 2 : Calculate the minimal score using small parsimony

NEAREST NEIGHBOR INTERCHANGE – PERFORMANCE TEST





- The variation of the execution time is difficult to interpret
- Best score increases linearly with sequence length
- Best score reaches a plateau at a certain mutation rate

NEAREST NEIGHBOR INTERCHANGE - IMPLEMENTATION

```
def nni_node_optimized(node, msa):
    if len(node.children) <= 1 or all(len(child.children) < 2 for child in node.children):
        score = 1
        return score, node
    _, best_score = small_parsimony(msa, node.to_tuple())
    best_structure = node
    if len(node.children) == 2 and all(len(child.children) == 2 for child in node.children):
        _, child0 = nni_node_optimized(node.children[0],msa)
        _, child1 = nni_node_optimized(node.children[1],msa)
        new_node = copy.deepcopy(node) #No change : 1st version
        new_node.children[0] = child0
        new_node.children[1] = child1
        _, temp_score1 = small_parsimony(msa, new_node.to_tuple())
        if temp_score1 < best_score:
            best_score = temp_score1
            best_structure = new_node
        new_node2 = copy.deepcopy(new_node) #2nd version
        new_node2.children[0].children[0], new_node2.children[1].children[0] =
        child1.children[0], child0.children[0]
        _, temp_score2 = small_parsimony(msa, new_node2.to_tuple())
        if temp_score2 < best_score:
            best_score = temp_score2
            best_structure = new_node2
        new_node3 = copy.deepcopy(new_node) #3rd version
        new_node3.children[0].children[0], new_node3.children[1].children[1] =
        child1.children[1], child0.children[0]
        _, temp_score3 = small_parsimony(msa, new_node3.to_tuple())
        if temp_score3 < best_score:
            best_score = temp_score3
            best_structure = new_node3
    else :
        _, child0 = nni_node_optimized(node.children[0], msa)
        _, child1 = nni_node_optimized(node.children[1], msa)
        new_node = copy.deepcopy(node) #3rd version
        new_node.children[0], new_node.children[1] = child0, child1
        _, temp_score = small_parsimony(msa, new_node.to_tuple())
        if temp_score < best_score:
            best_score = temp_score
            best_structure = new_node
    return best_score, best_structure
```

Previous steps :

Step 1 : Generate sequences

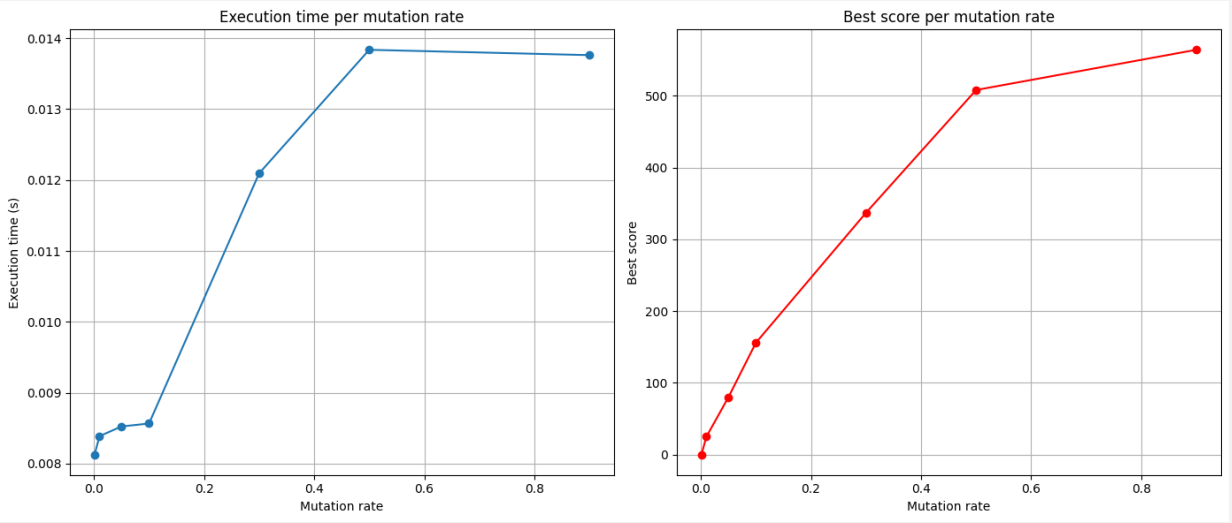
Step 2 : Build the tree using *guide_tree*

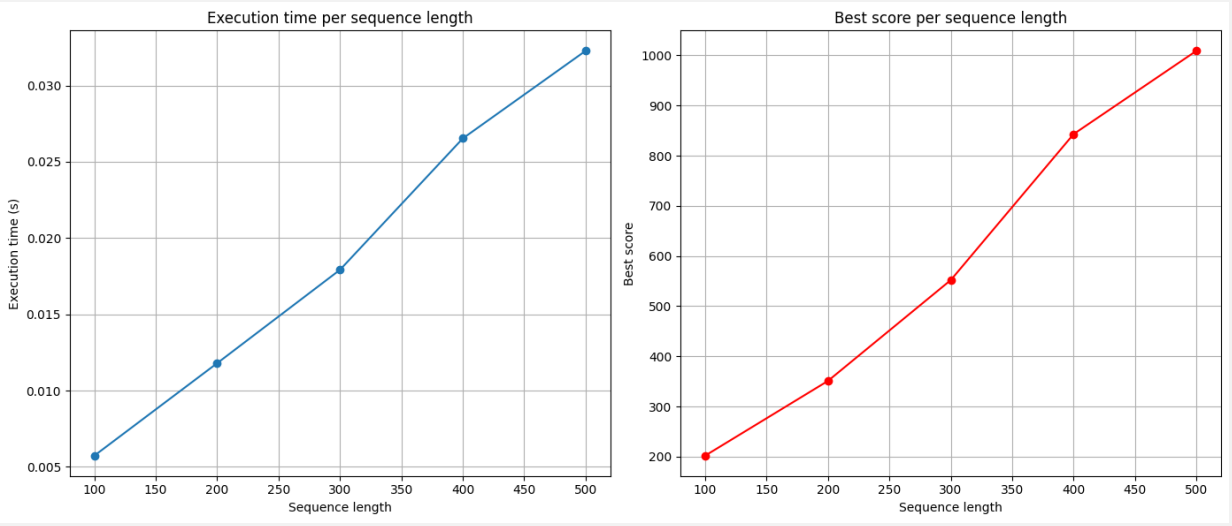
Step 3 : Convert the tree from tuple to an object

Step 4 : Align the sequences using *progressive_alignment*

Output : best score and tree

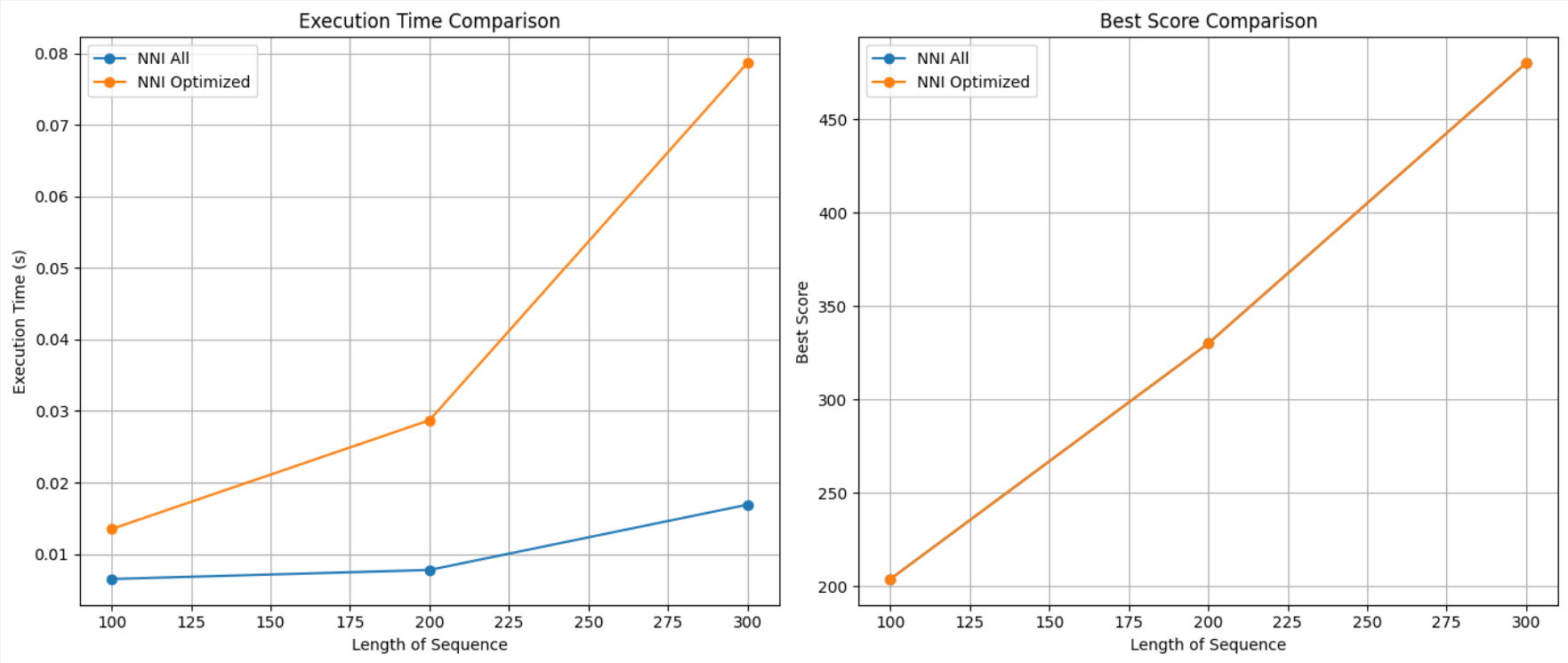
NEAREST NEIGHBOR INTERCHANGE – PERFORMANCE TEST





- The variations of best score and execution time are the same
- It increases linearly with sequence length
- It reaches a plateau at a certain mutation rate

NEAREST NEIGHBOR INTERCHANGE – COMPARISON



- The size of the memory required is optimized, but the execution time can sometimes be better.
- The best score obtained is the same.

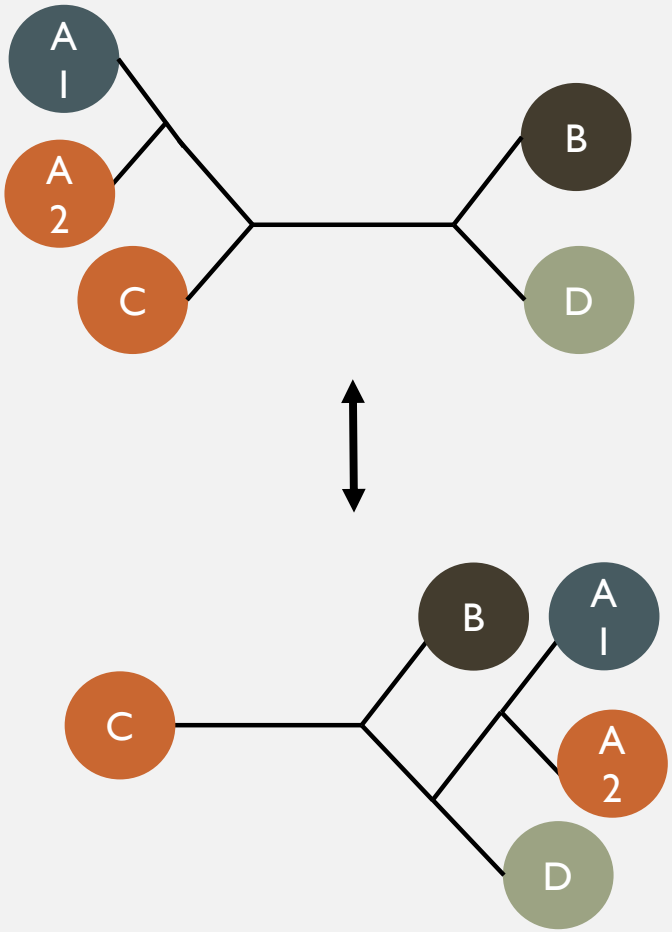
PRUNING AND REGRAPHTING STRATEGY

Random tree

Evaluate it
Small parsimony score

Generate a new one
Prune and regrapht

If no more prune and regrapht
Return best score and best tree



PRUNING AND REGRAPHTING STRATEGY– IMPLEMENTATION

```python
def collect_subtrees(node, subtrees):
  '''
  Returns all the possible subtrees
  Args:
    node : the node we want to obtain the subtrees from
    subtrees : the list we want to fill

  Returns:
    Nothing, it fills the subtrees list
  '''
  if len(node.children) == 2:
    collect_subtrees(node.children[0], subtrees)
    collect_subtrees(node.children[1], subtrees)
    subtrees.append(node)

def find_and_remove_subtree(node, subtree):
  '''Finds the subtree as input and remove it from the tree'''
  if len(node.children) == 2:
    if node.children[0] == subtree:
      node = node.children[1]
    elif node.children[1] == subtree:
      node = node.children[0]
    else:
      node.children[0] = find_and_remove_subtree(node.children[0], subtree)
      node.children[1] = find_and_remove_subtree(node.children[1], subtree)
  return node

def regrapht_subtree(node, subtree):
  '''Adds the subtree as input at a random leaf'''
  if len(node.children) == 0:
    node.add_child(copy.deepcopy(node))
    node.label = None
    node.add_child(subtree)
  else:
    coin = np.random.uniform()
    if coin > 0.5:
      node.children[0] = regrapht_subtree(node.children[0], subtree)
    else:
      node.children[1] = regrapht_subtree(node.children[1], subtree)
  return node
```

```python
, node, subtree, nodes):
t a random leaf'''
  if len(node.children) == 0:
    node.add_child(copy.deepcopy(node))
    node.label = None
    node.add_child(subtree)
    nodes.append(origin)
  else:
    origin2 = copy.deepcopy(origin)
    all_regrapht_subtree(origin2, node.children[0], subtree, nodes)
    origin3 = copy.deepcopy(origin)
    all_regrapht_subtree(origin3, node.children[1], subtree, nodes)


def prune_and_regrapht(node):
  '''Moves a random subtree at a random position'''
  if len(node.children) == 2:
    node2 = copy.deepcopy(node)
    subtrees = list()
    collect_subtrees(node2.children[0], subtrees)
    collect_subtrees(node2.children[1], subtrees)
    random_subtree = rd.choice(subtrees)
    node2 = find_and_remove_subtree(node2, random_subtree)
    node2 = regrapht_subtree(node2, random_subtree)
    return node2


def prune_and_regrapht_all(node):
  '''Moves a random subtree at all possible positions'''
  if len(node.children) == 2:
    node2 = copy.deepcopy(node)
    subtrees = list()
    collect_subtrees(node2.children[0], subtrees)
    collect_subtrees(node2.children[1], subtrees)
    random_subtree = rd.choice(subtrees)
    node2 = find_and_remove_subtree(node2, random_subtree)
    nodes = list()
    all_regrapht_subtree(node2, node2, random_subtree, nodes)
    return nodes
```

PRUNING AND REGRAPHTING STRATEGY– IMPLEMENTATION

```python
def prune_and_regrapht_greedy(node, n, msa):
  """
  A method to obtain a better structure by pruning and regraphting our current best structure
  """
  _, best_score = small_parsimony(msa, node.to_tuple())
  best_struc = node
  for i in range(n):
    new_node = prune_and_regrapht(best_struc)
    _, new_score = small_parsimony(msa, new_node.to_tuple())
    if new_score < best_score:
      best_score = new_score
      best_struc = new_node
  return best_score, best_struc


def prune_and_regrapht_naive(node, n, msa):
  """
  A method to obtain a better structure by pruning and regraphting our first structure (node)
  """
  _, best_score = small_parsimony(msa, node.to_tuple())
  best_struc = node
  for i in range(n):
    new_node = prune_and_regrapht(node)
    _, new_score = small_parsimony(msa, new_node.to_tuple())
    if new_score < best_score:
      best_score = new_score
      best_struc = new_node
  return best_score, best_struc
```

Previous steps :

Step 1 : Generate sequences

Step 2 : Build the tree using *guide_tree*

Step 3 : Convert the tree from tuple to an object

Step 4 : Align the sequences using *progressive_alignment*

↓

Output : rearranged tree

↓

Next step :

Step 1 : Repeat this action a huge number of times

PRUNING AND REGRAPHTING STRATEGY– IMPLEMENTATION

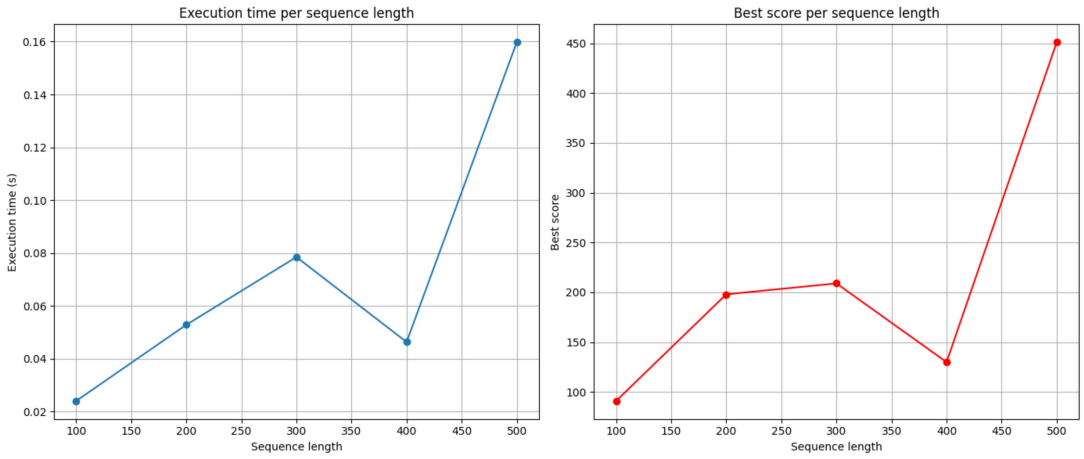
Previous steps :
Step 1 : Generate sequences
Step 2 : Build the tree using *guide_tree*
Step 3 : Convert the tree from tuple to an object
Step 4 : Align the sequences using *progressive_alignment*

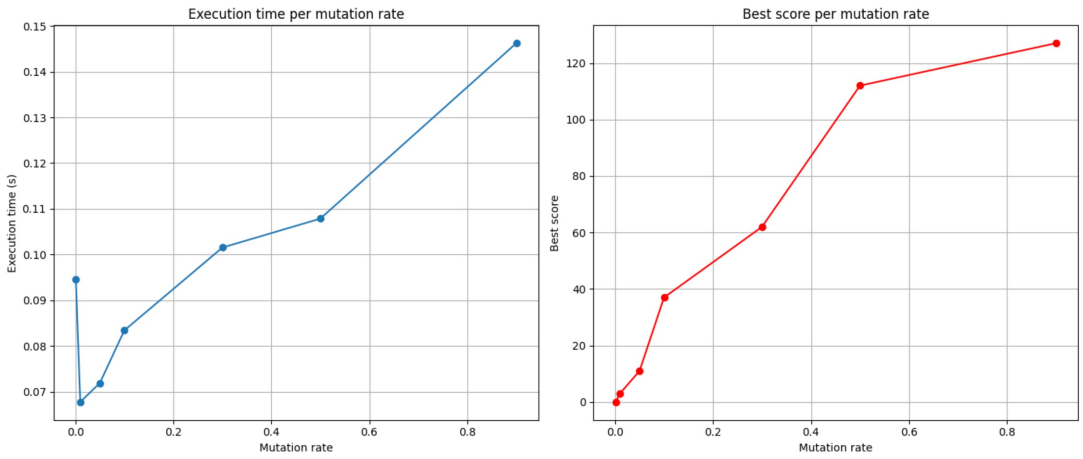
→ **Output** : list of all possible regraphted tree

→ **Next step** :
Step 1 : Repeat this action a huge number of times

```
def prune_and_regrapht_exhaustive(node, n, msa):
  """
  A method to obtain a better structure by pruning and regraphting our first structure (node)
  However, unlike the previous one, this method tries all the possible positions of the random subtree moved by the pruning and regraphting
  """
  _, best_score = small_parsimony(msa, node.to_tuple())
  best_struc = node
  for i in range(n):
    nodes = prune_and_regrapht_all(node)
    for new_node in nodes:
      _, new_score = small_parsimony(msa, new_node.to_tuple())
      if new_score < best_score:
        best_score = new_score
        best_struc = new_node
  return best_score, best_struc
```

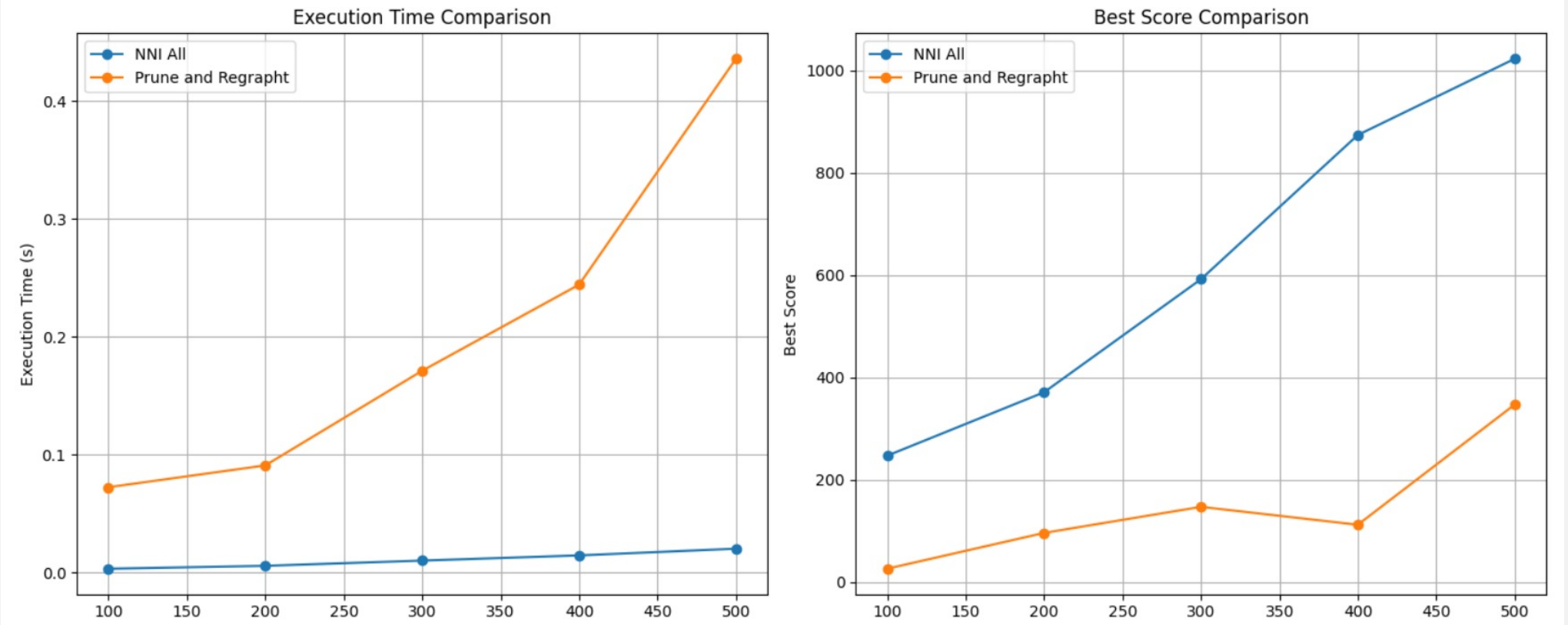
PRUNING AND REGRAPHTING STRATEGY– PERFORMANCE TEST





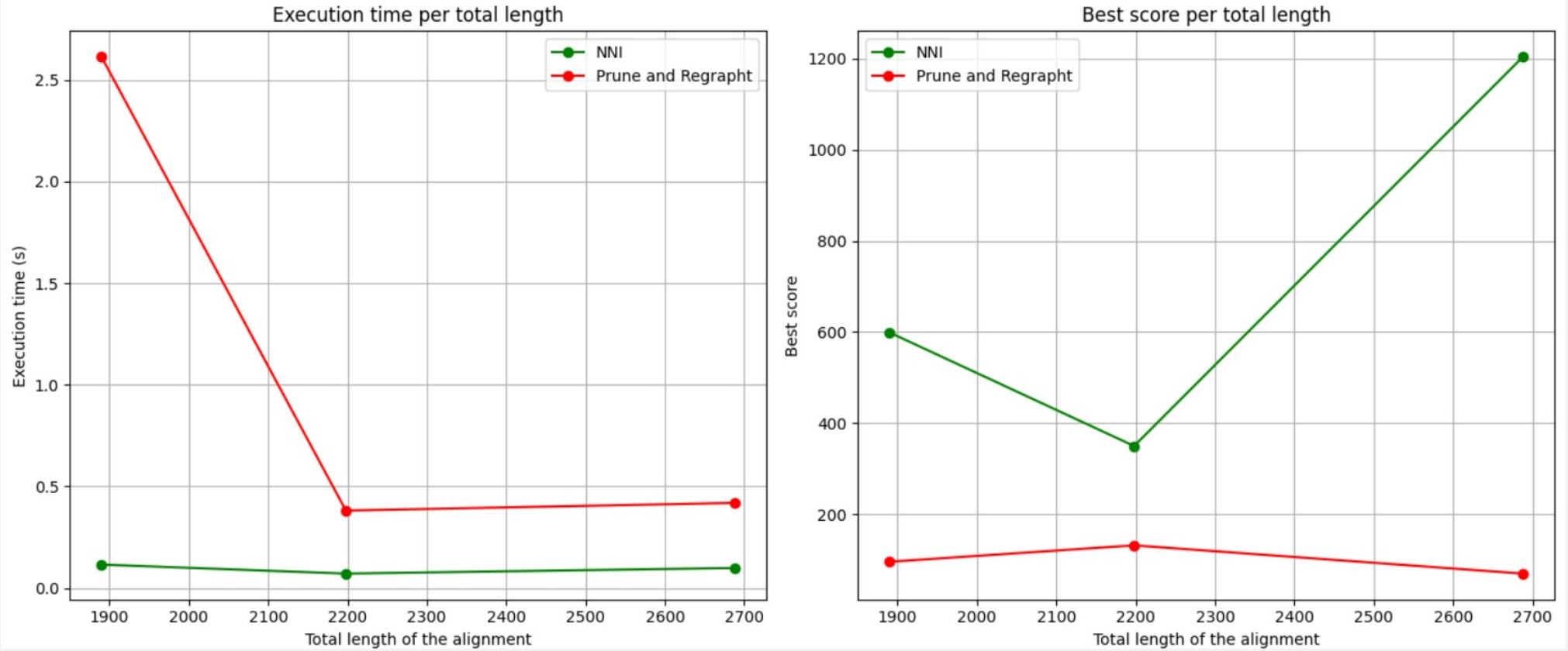
- The variations of best score and execution time are the same
- It increases linearly with sequence length
- It increases linearly with mutation rate

COMPARISON OF THE TWO STRATEGIES



- Prune and regrapht strategy has better results than NNI
- Prune and regrapht strategy has however a bigger execution time than NNI
- The difference of results may be explained by the diversity of tree structure brought by the new strategy

COMPARISON OF THE TWO STRATEGIES ON REAL ALIGNMENTS



- We observe the same difference of results, which depends on the alignments
- The 1st alignment is composed of a large number of small sequences
- The two other alignments are composed of a small number of large sequences

CONCLUSION

How could we address the large parsimony problem starting from that of small parsimony ?

A solid constructive heuristic thanks to:

- UPGMA
- Multiple sequence alignment
- Small parsimony

Efficient improving heuristics with the:

- nearest neighbor interchange
- pruning and regraphting