

Pattern Finder 2.0

ALGORITHM TO FIND SCRAMBLED PATTERN

AAKASH SAHA

Table of Contents

Abstract	2
Introduction	2
Description of the construction and operation of the algorithm	2
Key Terminology	2
Brief Description	3
Algorithm	6
Working Setup and Output	16
Use Case	16
Introduction	16
Requirement	16
Output	17

Abstract

Pattern recognition is the automated recognition of patterns and regularities in data. Pattern recognition is closely related to artificial intelligence and machine learning, and they are used extensively in applications which involves data mining and knowledge discovery in databases (KDD) etc.

Introduction

The need of a new pattern recognition algorithm came up while handling some use cases where identification of ***mutually exclusive pattern*** was required from a large data set ***where it might happen that the characters of the patterns are jumbled in between at every repetition.***

Example –

Let the data set be
679626749672467274698823862764897

The largest repetitive pattern (with jumbled characters) in the data set cannot be identified with any known existing pattern searching algorithm, even with naked eye.

The requirement was to identify that pattern which met the above stated requirement.

6796267496724**672746988**23862764897

The largest repeating pattern in the data set would be 672746988

Description of the construction and operation of the algorithm

The algorithm works by taking into consideration a probable pattern and then trying to find out if the pattern exists elsewhere in the data set.

Key Terminology

Left Set – The pattern taken into consideration

Right Set – The pattern which is checked if it is equals to the left set.

DeltaLeft – Extra set of characters in the Left Set as compared to the Right Set.

DeltaRight - Extra set of characters in the Right Set as compared to the Left Set.

MostPromisingPatternCandidate – The Left Set when it is equals to the Right set.

MostPromisingPattern – The MostPromisingPatternCandidate which satisfies the condition/clause to be the most suitable pattern.

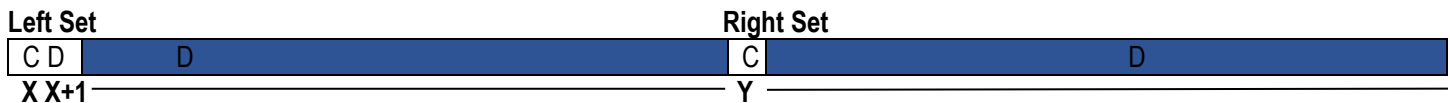
Brief Description

For every character (C) in the data set (location = X), the algorithm starts considering a Left Set (initially containing just the character), and then considers a subsequent occurrence of the same character (C) (location = Y) as the beginning of Right Set (also initially containing just the character).

If Left Set = Right Set, we set MostPromisingPatternCandidate as either of the sets and add the next character (D) which is present at location X+1 in the data set to the Left Set. 'D' is added in DeltaLeft and we try to balance the same in the Right Set.

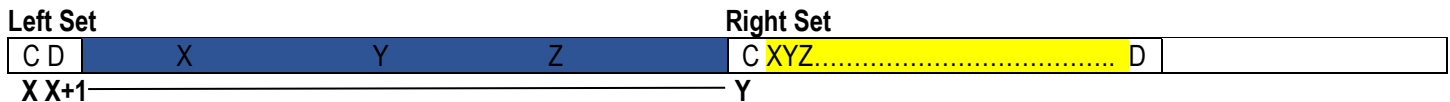
Balancing is done by finding the closest (D) from position Y (NOTE: the searching domain should be from X+2 to Y-1 on left of Right Set and Y+1 to the *end of the set* at right of Right Set) and add it to the Right Set.

NOTE: If D is present on both the sides of Right Set. We proceed with both the possibilities separately to identify the most promising pattern candidates in each case, which can come up during the balancing process(es).



Searching domain while balancing Right Set

While balancing some extra characters might get added in the Right Set which are between D and the position Y. We add the characters in DeltaRight and try to balance the Left Set. (NOTE: the searching domain should be from X+2 to Y-1 on left of Right Set at right of Left Set to prevent overlapping of the sets).



Searching domain while balancing Left Set

The process continues, until no more balancing is possible or when DeltaLeft and DeltaRight is empty i.e Left Set = Right Set. We set any one of them as a MostPromisingPatternCandidate.

Some, final condition/clause (example: length) should be applied on the selected MostPromisingPatternCandidate(s) to determine the MostPromisingPattern.

Example –

Consider the data set to be 1263782236181698942

Explanation of the balancing process

Start with C = 1

Left Set = [1], Left Current Location = 1

Next occurrence of 1 is 10,

Right Set = [1], Right Current Location = 10

Since, Left Set = Right Set, a most promising pattern candidate = [1]

Add next character ('2') to Left Set, so Left Set is now = [1,2], DeltaLeft is [2], Left Current Location is 1.

1263782236181698942

^

Now, when we try to add '2' to the right set. We find –

- | | |
|--|---|
| 1. 2 is present left to Right Current Location at position 7 | 1263782236181698942
^right current location (10) |
| 2. 2 is present right to Right Current Location at position 18 | 1263782236181698942
^right current location (10) |

We will consider both the cases -

case 1:

So, right Set is now [2,3,6,1], with DeltaRight as [3,6] and Right Current Location as 7.

We balance the DeltaRight characters in Left set. '3' is located at position 3. We add '3' to left Set. However, a '6' is present at location 2 which lies in between the left Current location (1) and the location of '3'. So, we add it too. Thus '6' of DeltaRight gets automatically balanced.

So, left Set is now [1,2,6,3] and right Set is [2,3,6,1]; since both the sets are same, a **most Promising Pattern Candidate** will be **[1,2,6,3]**

We save all details as a candidate for future validation by applying the condition/clause for selection of the most promising pattern.

case 2:

So, right Set is now [1,8,1,6,9,8,9,2], with DeltaRight as [8,1,6,9,8,9] and Right Current Location as 18.

We try to balance the left set with the DeltaRight. However, we can see that we cannot balance 1, as it is not present in the searching domain for left set.

1263782236181698942

So, Case 2 becomes invalid.

We now consider the result of Case 1,

We add '7' (next character) to Left Set, so left set becomes [1,2,6,3,7], DeltaLeft is '7'. This '7' cannot be balanced at the right Set as there is no extra '7' present near the Right Current Index. (both left & right)

We try the same for the next occurrence of '1',

Location = 12,

Again, left set = [1], Left Current Location = 1

Right Set = [1], Right Current Location = 12

Left Set = Right Set, most promising pattern candidate for this combination = [1]

Add next character (2) to Left Set, so Left Set is now = [1,2], DeltaLeft is [2], Left Current Location is 1.

We need to balance the delta Left in the right Set. Now, '2' is present at the left of the Right Current Location (in the interval range (Left Current Location + 1, to the starting of the right set)) and present at the right of the of the Right Current Location (in the interval of range (end of right set to the end of the data set). So, we again have 2 cases-

- | | |
|--|---|
| 1. 2 is present left to Right Current Location at position 7 | 1263782236181698942
^right current location (12) |
| 2. 2 is present right to Right Current Location at position 18 | 1263782236181698942
^right current location (12) |

Case 1: 1263782236181698942

leftSet = [1,2]

rightSet = [2,3,6,1,8,1] DeltaRight = [3,6,1,8]

however, all characters in DeltaRight cannot be balanced in the leftSet. (Note: DeltaRight has '1' which cannot be balanced at leftSet)

Case 1: 1263782236181698942

leftSet = [1,2]

rightSet = [1,6,9,8,9,4,2] DeltaRight = [6,9,8,9,4]

however, all characters in DeltaRight cannot be balanced in the leftSet. (Note: DeltaRight has '9' which cannot be balanced at leftSet)

We break from the loop and try with the next character.

The process should be executed with Character 'C' taken as '2', then '6' and so on.

After collating all the possible candidates for most promising pattern, the largest pattern can be considered as the most promising pattern .

Example - Let the data set be, 153355114224176671107155311534214211677166153553511124241617671. The largest pattern should be –

153355114224176671107155311534214211677166153553511124241617671

Algorithm

The algorithm has 6 parts –

1. Main
2. FixLeft
3. FixRight
4. manageBalancedDetails
5. controller
6. findMostPromisingDataSet

=====

#Name - main

Params – dataset > dataset provided as input from pattern must be found

Description – The algorithm initiates the balancing operation, obtains the balancedObject. It filters and groups the patterns occurred in dataset into unique sets and applies the required conditions/clauses to determine the most promising pattern.

Returns – the most promising pattern and all it's information.

main (dataString)

1. Initialize testDataSet = []
2. for data in dataString repeat till Step 3
3. Add data in testDataSet
4. Initialize balancedObject = []
5. Initialize setsConsidered = []
6. Initialize patternDictList = []
7. Call controller(balancedObject, testDataSet)
8. Set index = -1
9. For ele in balancedObject, repeat till Step 20
10. Set index = index + 1
11. If ele['leftSet'] is in setsConsidered GOTO Step 7
12. Initialize patternDict = {}
13. Else, store ele['leftSet'], ele['rightSet'], occurrences of the leftSet and rightSet(~[[leftStartPos1, leftEndPos1],[rightStartPos1, rightEndPos1]]), overall pattern set (~[leftSet, rightSet]) in patternDict.
14. Store ele['leftSet'] and ele['rightSet'] in setsConsidered.
15. For ele2 in balancedObject from index+1 to last, repeat till Step 20
16. If ele2['leftSet'] is not equals to ele['leftSet'], GOTO step 20
17. If ele['rightSet'] is present in patternDict['rightSet'], GOTO Step 20
18. Add ele['rightSet'] in patternDict['rightSet'] and patternDict['patterns'].
19. Add occurrence of ele['rightSet'] in patternDict['occurrences'].
20. Store the patternDict in patternDictList.
21. Call findMostPromisingDataSet(patternDictList) and store the return in result.
22. return result

Working Process (Dry Run)

Sample Parameters =

dataString = '9123762136'

//dataString is a string comprising of VLAN priorities of input packet, pattern searching algorithm will operate on this dataSet

Step 1: Initialize testDataSet = []

Step 2: for data in dataString

Add data in testDataSet

So, testDataSet = [9,1,2,3,7,6,2,1,3,6]

Step 4: balancedObject = []

Step 5: setsConsidered = []

Step 6: patternDictList = []

Step 7: Call controller(balancedObject, testDataSet)

The controller finds all the probable patterns from the data string and mines all required information of the patterns and their respective repetition. The same gets stored in the balancedObject.

The balanced object now is –

```
[
{'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['1'], 'leftEndPos': 1, 'leftSet': ['1'], 'rightStartPos': 7},
{'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['2', '1'], 'leftEndPos': 2, 'leftSet': ['1', '2'], 'rightStartPos': 6},
{'leftStartPos': 1, 'rightEndPos': 8, 'rightSet': ['2', '1', '3'], 'leftEndPos': 3, 'leftSet': ['1', '2', '3'], 'rightStartPos': 6},
{'leftStartPos': 2, 'rightEndPos': 6, 'rightSet': ['2'], 'leftEndPos': 2, 'leftSet': ['2'], 'rightStartPos': 6},
{'leftStartPos': 3, 'rightEndPos': 8, 'rightSet': ['3'], 'leftEndPos': 3, 'leftSet': ['3'], 'rightStartPos': 8},
{'leftStartPos': 5, 'rightEndPos': 9, 'rightSet': ['6'], 'leftEndPos': 5, 'leftSet': ['6'], 'rightStartPos': 9}
]
```

Step 8: Set index = -1

Step 9: For ele in balancedObject

ele = {'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['1'], 'leftEndPos': 1, 'leftSet': ['1'], 'rightStartPos': 7}

Step 10: index = index+1

Step 11: Is ele['leftSet'] in setsConsidered? No. Continue

Step 12: Set patternDict = {}

Step 13: Store data in patternDict

patternDict['leftSet'] = leftSet1

patternDict['rightSets'] = [rightSet]

patternDict['patterns'] = [leftSet, rightSet]

patternDict['occurrences'] = [[leftStartPos, leftEndPos], [rightStartPos, rightEndPos]]

So, patternDict is now,

{'rightSets': [['1']], 'patterns': [['1'], ['1']], 'leftSet': ['1'], 'occurrences': [[1, 1], [7, 7]]}

Step 15: Store ele['leftSet'] and ele['rightSet'] in setsConsidered. So, setsConsidered = [['1'], ['1']]

Step 16: For ele2 in balancedObject from index+1 to last.

Ele2 = {'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['2', '1'], 'leftEndPos': 2, 'leftSet': ['1', '2'], 'rightStartPos': 6},

Step 17: Is ele['leftSet'] = ele2['leftSet']? No. (ele['leftSet'] = ['1'] and ele2['leftSet'] = ['1', '2']). GOTO Step 15

We find none of the ele2['leftSet'] matches with ele['leftSet'].

Step 20: Store patternDict in patternDictList. Goto Step 9

The loop iterates for all the elements (ele) of the balanced objects from Step 9.
patternDictList after complete iteration becomes –

```
[
{'rightSets': [['1']], 'patterns': [['1'], ['1']], 'leftSet': ['1'], 'occurrences': [[1, 1], [7, 7]]},
{'rightSets': [['2', '1']], 'patterns': [['1', '2'], ['2', '1']], 'leftSet': ['1', '2'], 'occurrences': [[1, 2], [6, 7]]},
{'rightSets': [['2', '1', '3']], 'patterns': [['1', '2', '3'], ['2', '1', '3']], 'leftSet': ['1', '2', '3'], 'occurrences': [[1, 3], [6, 8]]},
{'rightSets': [['2']], 'patterns': [['2'], ['2']], 'leftSet': ['2'], 'occurrences': [[2, 2], [6, 6]]},
{'rightSets': [['3']], 'patterns': [['3'], ['3']], 'leftSet': ['3'], 'occurrences': [[3, 3], [8, 8]]},
{'rightSets': [['6']], 'patterns': [['6'], ['6']], 'leftSet': ['6'], 'occurrences': [[5, 5], [9, 9]]}
]
```


]

Step 21: result = findMostPromisingDataSet(patternDictList)

We get result as –

```
{'rightSets': [['2', '1', '3']], 'patterns': [['1', '2', '3'], ['2', '1', '3']], 'leftSet': ['1', '2', '3'], 'occurrences': [[1, 3], [6, 8]]}
```

Step 22: Return the result

```
=====
=====
```

#Name - controller

Params – balancedObject > A dictionary/associative containing all the possible most promising candidates.

dataset > dataset provided as input from pattern must be found

Description –The object triggers the balancing process and identification of most promising pattern candidate.

Returns – Returns balanced object comprising of all most promising pattern candidates which can be possible in the dataset.

controller(balancedObject, dataSet)

1. Set lenOrig = length(dataSet)
2. for i in range(0 to lenOrig) , repeat till Step 14
3. Set testDataSet = dataSet[i:]
4. Set startCharacter = testDataSet[0]
5. Initialize testSetLocation = []
6. for j in range(0 to length(testDataSet)) repeat till Step 7
7. If (testDataSet[j] = startCharacter) then add j to testSetLocation
8. for index in range(1 to length(testSetLocation)) repeat till Step 14
9. Initialize leftSet = [startCharacter]
10. Initialize rightSet = [startCharacter]
11. Initialize leftStartPos = leftEndPos = 0
12. Initialize rightStartPos = rightEndPos = testSetLocation[index]
13. Initialize deltaLeft = []
14. Call fixRight
15. Return balancedObject

```
=====
```

Working Process (Dry Run)

Input Parameters

balancedObject = {}

dataset = [1,9,1,2,3,7,6,2,1,3,6]

Step 1: Set lenOrig = length(dataSet)

Step 2: for i in range(0 to lenOrig) , repeat till Step 14

i = 0

Step 3: Set testDataSet = dataSet[0:]

testDataSet = [1,9,1,2,3,7,6,2,1,3,6]

Step 4: Set startCharacter = testDataSet[0]
startCharacter = [1]
Step 5: Set testSetLocation = []
Step 6: for j in range(0 to length(testDataSet)) repeat till Step 7
Step 7: If (testDataSet[j] = startCharacter) then add j to testSetLocation
testLocation = [0,2,8]
Step 8: for index in range(1 to 3) repeat till Step
Step 9: Initialize leftSet = [1]
Step 10: Initialize rightSet = [1]
Step 11: Initialize leftStartPos = leftEndPos = 0
Step 12: Initialize rightStartPos = rightEndPos = testSetLocation[1] = 2
Step 13: Initialize deltaLeft = []
Step 14: Call fixRight with all required arguments.
Step 15: return balancedObject

#Name - manageBalancedDetails

Params – balancedObject > A dictionary/associative containing all the possible most promising candidates.

dataset > dataset provided as input from pattern has to be found

leftSet > first set of data

rightSet > set of predictive data

leftStartPos > position of first character of leftSet in the dataset

leftEndPos > position of last character of leftSet in the dataset

rightStartPos > position of first character of right set in the dataset

rightEndPos > current position of the right set.

startCharacter > The character from which the balancing was started at the beginning

rightSetCombination > Sets containing states of rightFix while balancing deltaLeft at RightSet. (default, empty set)

Description – The algorithm manages a balancedObject which contains a list of all the states where leftSet was equals to rightSet while analyzing the data set taken as input.

manageBalancedDetails(balancedObject, dataSet, testDataSet, leftSet, rightSet, leftStartPos, leftEndPos, rightStartPos, rightEndPos, rightSetCombination)

1. Initialize dictDetails = {}
2. Set dictDetails['leftSet'] = leftSet
3. Set dictDetails['rightSet'] = rightSet
4. Set dictDetails['leftStartPos'] = leftStartPos + length(dataSet) - length(testDataSet)
5. Set dictDetails['leftEndPos'] = leftEndPos + length(dataSet) - length(testDataSet)
6. Set dictDetails['rightStartPos'] = rightStartPos + length(dataSet) - length(testDataSet)
7. Set dictDetails['rightEndPos'] = rightEndPos + length(dataSet) - length(testDataSet)
8. Add dictDetails in balancedObject
9. If leftEndPos is not < rightStartPos – 1, then GOTO step 14
10. Set leftEndPos = leftEndPos + 1
11. Set leftSet = leftSet + Character at leftEndPos in testDataSet
12. extraLeftCharAdded = Character at leftEndPos in testDataSet
13. Call fixRight
14. Exit

Working Process (Dry Run)

Sample Parameters =

```

balancedObject  [{'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['1'], 'leftEndPos': 1, 'leftSet': ['1'], 'rightStartPos': 7}]
dataset         ['9', '1', '2', '3', '7', '6', '2', '1', '3', '6']
testDataSet     ['1', '2', '3', '7', '6', '2', '1', '3', '6']
leftSet         ['1', '2']
rightSet        ['2', '1']
leftStartPos     0
leftEndPos       1
rightStartPos    5
rightEndPos      6
rightSetCombination []

```

Step 1: dictDetails = {}**Step 2:** dictDetails['leftSet'] = ['1', '2'] (~leftSet)**Step 3:** dictDetails['rightSet'] = ['2', '1'] (~rightSet)**Step 4:** dictDetails['leftStartPos'] = leftStartPos + length(dataset) - length(testDataSet)

So, dictDetails['leftStartPos'] = 0 + 10 - 9 = 1

Step 5: dictDetails['leftEndPos'] = leftEndPos + length(dataset) - length(testDataSet)

So, dictDetails['leftEndPos'] = 1 + 10 - 9 = 2

Step 6: dictDetails['rightStartPos'] = rightStartPos + length(dataset) - length(testDataSet)

So, dictDetails['rightStartPos'] = 5 + 10 - 9 = 6

Step 7: dictDetails['rightEndPos'] = rightEndPos + length(dataset) - length(testDataSet)

So, dictDetails['rightEndPos'] = 6 + 10 - 9 = 7

Step 8: Add dictDetails in balancedObject

balancedObject =

```

[{'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['1'], 'leftEndPos': 1, 'leftSet': ['1'], 'rightStartPos': 7}, {'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['2', '1'], 'leftEndPos': 2, 'leftSet': ['1', '2'], 'rightStartPos': 6}]

```

Step 9: Is leftEndPos is not < rightStartPos - 1? No. Then proceed to step 10.**Step 10:** Set leftEndPos = leftEndPos + 1

So, leftEndPos = 2

Step 11: Set leftSet = leftSet + Character at leftEndPos in testDataSet

So, leftSet = ['1', '2', '3']

Step 12: extraLeftCharAdded = Character at leftEndPos in testDataSet

So, extraLeftCharAdded = ['3']

Step 13: Call fixRight**Step 14:** Exit

```

=====
=====

```

#Name - fixLeft

Params – balancedObject > A dictionary/associative containing all the possible most promising candidates.

dataset > dataset provided as input from pattern has to be found

leftSet > first set of data

rightSet > set of predictive data

leftStartPos > position of first character of leftSet in the dataset

leftEndPos > position of last character of leftSet in the dataset

rightStartPos > position of first character of right set in the dataset

rightEndPos > current position of the right set.

deltaRight > extraSet of characters in right Set which needs to be balanced at left set
 startCharacter > The character from which the balancing was started at the beginning
 rightSetCombination > Sets containing states of rightFix while balancing deltaLeft at RightSet. (default, empty set)

Description – The algorithm balances the left Set with the set of characters present in deltaRight to ensure that all characters present in Right Set are present in the Left Set.

**fixLeft(balancedObject, dataSet,
 testDataSet, leftSet, rightSet, leftStartPos, leftEndPos, rightStartPos, rightEndPos, deltaRight, rightSetCombination)**

1. Initialize extraCharacterAddedAtLeft as empty set.
2. For character (C) in deltaRight, repeat till Step 8
3. For all data from leftEndPos + 1 to rightStartPos -1, repeat till Step 7
4. If C = data
5. Remove C from deltaRight, update leftNewPos, GOTO Step 2
6. Else, if data in deltaRight, remove data from deltaRight and update leftNewPos, GOTO Step 8
7. Else add data in extraCharacterAddedAtLeft.
8. If C is not removed, return 0 (indicating all characters of deltaRight cannot be balanced at leftSet)
9. If extraLeftCharAdded = [], i.e no extra data added in LeftSet while balancing call manageBalancedDetails.
10. Else call FixRight.

#Example of fixLeft

153355114224176671107**15531153**4214211677166153553511124241617671

Let set = [1,5,3,3,5]

Right set = [1,5,5,3,1,1,5,3]

leftEndPos = 4

rightStartPos = 21

deltaRight = [1,1,5] (marked in blue)

Working process (Dry Run)

Step 1> Set extraCharacterAddedAtLeft = []

Step 2> Character C = 1,

Step 3> For characters from position 5 to 20, we find C,

Step 4> Data at position (5) = '5'. [15335**5**114224176671107155311534214211677166153553511124241617671]

'5' is not equals to '1' (C), Skip 5

Step 5> Skipped as Step 4 is False

Step 6> '5' is in deltaRight ([1,1,5]). **Remove 5 from the deltaRight.** deltaRight is now [1,1], Update leftNewPos to 5.

Step 7> Skipping Step 7 as Step 6 is True.

Step 4> Data at position (6) = '1'. [153355**1**14224176671107155311534214211677166153553511124241617671]

'1' is equals to '1' (C),

Step 5> Remove '1' from deltaRight. deltaRight is now [1]. Update leftNewPos to 6

Break from inner loop

Step 2> Character C = 1,

Step 3> For characters from position 5 to 20, we find C,

Step 4> Data at position (7) = '1'. [1533551**1**4224176671107155311534214211677166153553511124241617671]

'1' is equals to '1' (C),

Step 5> Remove '1' from deltaRight. deltaRight is now []. Update leftNewPos to 7

Break from inner loop

deltaRight = []. We exit from the outer loop

Now,

Let set = [1,5,3,3,5,5,1,1]

Right set = [1,5,5,3,1,1,5,3]

153355114224176671107**15531153**4214211677166153553511124241617671

Left set and Right Set has exact same set of characters.

Since, no extra characters were added at the Left Set during balancing, i.e *extraCharacterAddedAtLeft* is [], we call *manageBalancedDetails* with all the managed details, left set, right set, index position of the sets etc.

NOTE - If some extra characters are added in the left set (in Step 7 of *fixLeft*) during the balancing process, the extra characters needs to be balanced in *fixRight* (below). We call, *fixRight* with all the required parameters. The process terminates only when further balancing is not possible. A best promising pattern is then selected from the balanced sets gathered by the *manageBalancedDetails* by applying some condition/clause.

=====

#Name - *fixRight*

Params – *balancedObject* > A dictionary/associative containing all the possible most promising candidates.

dataset > dataset provided as input from pattern has to be found

leftSet > first set of data

rightSet > set of predictive data

leftStartPos > position of first character of *leftSet* in the dataset

leftEndPos > position of last character of *leftSet* in the dataset

rightStartPos > position of first character of right set in the dataset

rightEndPos > current position of the right set.

deltaLeft > extraSet of characters in left Set which needs to be balanced at right set

startCharacter > The character from which the balancing was started at the beginning

rightSetCombination > Sets containing states of *rightFix* while balancing *deltaLeft* at *RightSet*. (default, empty set)

Description – The algorithm balances the right Set with the set of characters present in *deltaLeft* to ensure that all characters present in Left Set are present in the Right Set.

fixRight(balancedObject, dataSet, testDataSet, leftSet, rightSet, leftStartPos, leftEndPos, rightStartPos, rightEndPos, deltaLeft, rightSetCombination=[])

1. Initialize *extraCharacterAddedAtRight* as empty set.
2. While True, repeat till Step 29
3. If (*deltaLeft* = []) break from loop, Goto step 30
4. Set character (C) = character at 0th index of *deltaLeft*,
5. Initialize *leftLocation* as [False, infinity], *rightLocation* as [False, infinity]
6. Initialize *deltaLeftTestLeft* = *deltaLeftTestRight* = *deltaLeft*
7. Initialize *extraCharacterAddedAtRightTestLeft* = *extraCharacterAddedAtRightTestRight* = []
8. Initialize *rightSetLeftTest* = *rightSetRightTest* = *rightSet*
9. For all data in dataset from *rightStartPos* – 1 to *leftEndPos*+ 1, repeat till step 13
10. Set *rightSetLeftTest* = data + *rightSetLeftTest*
11. If data = C, set *leftLocation* as [True, index of data], remove C from *deltaLeftTestLeft*. GOTO step 14
12. Else if data in *deltaLeftTestLeft*, remove C from *deltaLeftTestLeft*. GOTO step 9

13. Else add data in extraCharacterAddedAtRightTestLeft
14. For all data in dataset from rightStartPos + length of rightSet to last index of dataSet, repeat till step 18
15. Set rightSetRightTest = rightSetRightTest + data
16. If data = C, set rightLocation as [True, index of data], remove C from deltaLeftTestRight. GOTO step 19
17. Else if data in deltaLeftTestRight, remove C from deltaLeftTestRight. GOTO step 14
18. Else add data in extraCharacterAddedAtRightTestRight.
19. If leftLocation[0] is False, i.e C is absent at left of RightSet, GOTO Step 21
20. Store all details of occurrence including rightStartPos, rightEndPos, leftStartPos, leftEndPos, extraRightTemp (~extraRightAddedAtLeft + extraRightTemp), leftSet, rightSet (~rightSetTestLeft), deltaLeft (~deltaLeftTestLeft) in rightSetCombination.
21. If rightLocation[0] is False, i.e C is absent at right of RightSet, GOTO Step 23
22. Store all details of occurrence including rightStartPos, rightEndPos, leftStartPos, leftEndPos, extraRightTemp (~extraRightTemp + extraRightAddedAtRight), leftSet, rightSet (~rightSetTestRight), deltaLeft (~deltaLeftTestRight) in rightSetCombination.
23. Initialize anyRightFixRemaining as False
24. For any storedRightSetDetails in rightSetCombination, Repeat till Step 28
25. If deltaLeft is [], GOTO step 24
26. Restore the variable information.
27. Remove the storedRightSetDetails from rightSetCombination
28. Set anyRightFixRemaining as True and break from the loop and GOTO step 2
29. If anyRightFixRemaining = False, break from loop, GOTO step 30
30. If deltaLeft = [], call manageBalancedDetails and return
31. If rightSetCombination = [], return
32. Else, for each storedRightSetDetails in rightSetCombination, Repeat till step 34
33. If extraRightTemp of storedRightSetDetails = [], call manageBalancedDetails and Remove storedRightSetDetails
34. Else call fixLeft and remove storedRightSetDetails.
35. Exit

Working Progress (Dry Run)

```

balancedObject  []
dataset         ['9', '1', '2', '3', '7', '6', '2', '1', '3', '6']
testDataSet     ['1', '2', '3', '7', '6', '2', '1', '3', '6']
leftSet         ['1']
rightSet        ['1']
leftStartPos    0
leftEndPos      0
rightStartPos   6
rightEndPos     6
deltaLeft       []
rightSetCombination []

```

Step 1: extraCharacterAddedAtRight = []

Step 2: Enter while loop.

Step 3: Is deltaLeft = [] ?, yes. break. GOTO Step 30

Step 30: Call manageBalancedDetails.

ManageBalancedDetails stores leftSet, rightSet and the parameters as a possible most promising pattern candidate in the **balancedObject**. (NOTE – leftSet = rightSet). *It calls rightSet internally, with leftSet containing the next character.*

```

balancedObject  [{'leftStartPos': 1, 'rightEndPos': 7, 'rightSet': ['1'], 'leftEndPos': 1, 'leftSet': ['1'], 'rightStartPos': 7}]
dataset         ['9', '1', '2', '3', '7', '6', '2', '1', '3', '6']
testDataSet     ['1', '2', '3', '7', '6', '2', '1', '3', '6']

```

```

leftSet          ['1', '2']
rightSet          ['1']
leftStartPos      0
leftEndPos        1
rightStartPos      6
rightEndPos        6
deltaLeft         ['2']
rightSetCombination []
    
```

Step 1: extraCharacterAddedAtRight = []

Step 2: Enter while loop.

Step 3: Is deltaLeft = [] ?, No. Continue

Step 4: C = 1

Step 5: leftLocation = [False,infinity], rightLocation = [False,infinity]

Step 6: deltaLeftTestLeft = deltaLeftTestRight = deltaLeft = [1]

Step 7: extraCharacterAddedAtRightTestLeft = extraCharacterAddedAtRightTestRight = []

Step 8: rightSetLeftTest = rightSetRightTest = rightSet = [1]

Step 9: For all data in dataset from rightStartPos – 1 to leftEndPos+ 1, (5 to 2) repeat till step 13

Data = 2

Step 10: Set rightSetLeftTest = data + rightSetLeftTest

rightSetLeftTest = [2,1]

Step 11: Is data = C?, Yes. set leftLocation as [True, 5], remove C from deltaLeftTestLeft. GOTO step 14

deltaLeftTestLeft = []

Step 14: For all data in dataset from rightStartPos + length of rightSet to last index of dataset (7 to 8), repeat till step 18

Data = 3

Step 15: Set rightSetRightTest = rightSetRightTest + data

rightSetRightTest = [1,3]

Step 16: If data = C, ? No

Step 17: Else if data in deltaLeftTestRight, remove C from deltaLeftTestRight. GOTO step 14

Step 18: Else add data in extraCharacterAddedAtRightTestRight.

extraCharacterAddedAtRightTestRight = [3]

Data = 6

Step 15: Set rightSetRightTest = rightSetRightTest + data

rightSetRightTest = [1,3,6]

Step 16: If data = C, ? No

Step 17: Else if data in deltaLeftTestRight, remove C from deltaLeftTestRight. GOTO step 14

Step 18: Else add data in extraCharacterAddedAtRightTestRight.

extraCharacterAddedAtRightTestRight = [3,6]

Step 19: If leftLocation[0] is False, No. Continue.

Step 20: Store all details of occurrence including rightStartPos, rightEndPos, leftStartPos, leftEndPos, extraRightTemp (~extraRightAddedAtLeft + extraRightTemp), leftSet, rightSet (~rightSetTestLeft), deltaLeft (~deltaLeftTestLeft) in rightSetCombination.

[[{'deltaLeft': [], 'rightSet': ['2', '1'], 'leftStartPos': 0, 'leftEndPos': 1, 'extraRightTemp': [], 'rightEndPos': 6, 'leftSet': ['1', '2'], 'rightStartPos': 5}]]

Step 21: If rightLocation[0] is False, Yes GOTO Step 23

Step 23: Initialize anyRightFixRemaining as False

Step 24: For any storedRightSetDetails in rightSetCombination, Repeat till Step 28

Step 25: Is deltaLeft is []? ,Yes GOTO step 24

Step 29: If anyRightFixRemaining = False, Yes

Step 30: If deltaLeft = [], No.

Step 31: Is rightSetCombination = []? No

Step 32: For each storedRightSetDetails in rightSetCombination, Repeat till step 34

Step 34: Is extraRightTemp of storedRightSetDetails = []?, call manageBalancedDetails and Remove storedRightSetDetails

Step 35: Exit

The process continues with successive trial and balance, until all the most promising pattern candidates are saved in the balancedObject

#Name - findMostPromisingDataSet

Params – patternDictList > a list comprising of all the patterns which have repeated once or more in the data set. And all the corresponding informations, example – occurrence positions, pattern set, etc.

Description – The algorithm applies the largest most promising pattern from all the candidates.

findMostPromisingDataSet(patternDictList)

1. Initialize largestPtrn = [] and obj = None
2. For ele in patternDictList repeat till Step 3
3. If(length(ele['leftSet']) > length(largestPtrn)) then set largestPtrn = ele['leftSet'] and obj = ele
4. return obj

Working Progress (Dry Run)

Sample Parameter -

patternDictList =

```
[
{'rightSets': [['1']], 'patterns': [['1'], ['1']], 'leftSet': ['1'], 'occurrences': [[1, 1], [7, 7]]},
{'rightSets': [['2', '1']], 'patterns': [['1', '2'], ['2', '1']], 'leftSet': ['1', '2'], 'occurrences': [[1, 2], [6, 7]]},
{'rightSets': [['2', '1', '3']], 'patterns': [['1', '2', '3'], ['2', '1', '3']], 'leftSet': ['1', '2', '3'], 'occurrences': [[1, 3], [6, 8]]},
{'rightSets': [['2']], 'patterns': [['2'], ['2']], 'leftSet': ['2'], 'occurrences': [[2, 2], [6, 6]]},
{'rightSets': [['3']], 'patterns': [['3'], ['3']], 'leftSet': ['3'], 'occurrences': [[3, 3], [8, 8]]},
{'rightSets': [['6']], 'patterns': [['6'], ['6']], 'leftSet': ['6'], 'occurrences': [[5, 5], [9, 9]]}
]
```

Step 1: largestPtrn = [], obj = None

Step 2: For ele in patternDictList repeat till Step 3

Ele = {'rightSets': [['1']], 'patterns': [['1'], ['1']], 'leftSet': ['1'], 'occurrences': [[1, 1], [7, 7]]},

Step 3: Is (len(ele['leftSet']) > len(largestPtrn))? Yes. set largestPtrn = ele['leftSet'] and obj = ele
 largestPtrn = ['1'],
 obj = {'rightSets': [['1']], 'patterns': [['1'], ['1']], 'leftSet': ['1'], 'occurrences': [[1, 1], [7, 7]]},

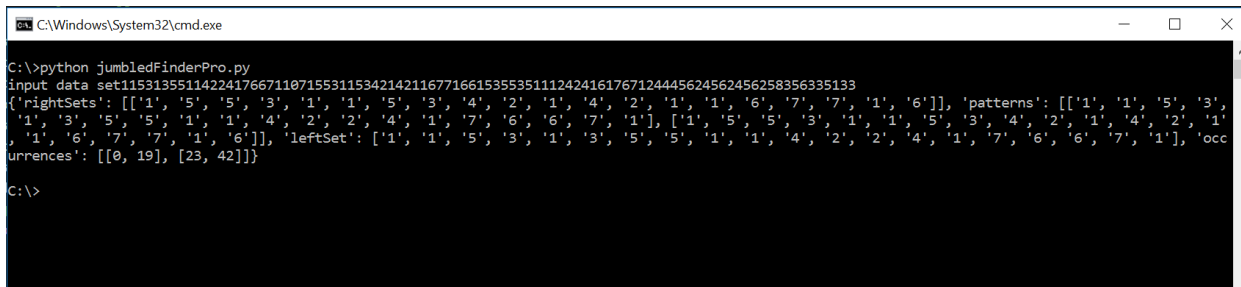
Repeat Step 3 for all the elements of the patternDictList

Step 4: Return obj

The final obj should be -

Obj = {'rightSets': [['2', '1', '3']], 'patterns': [['1', '2', '3'], ['2', '1', '3']], 'leftSet': ['1', '2', '3'], 'occurrences': [[1, 3], [6, 8]]}

Working Setup and Output



```

C:\Windows\System32\cmd.exe
C:\>python jumbledFinderPro.py
input data set1153135511422417667110715531153421421167716615355351112424161767124445624562456258356335133
{'rightSets': [['1', '5', '5', '3', '1', '1', '5', '3', '4', '2', '1', '4', '2', '1', '1', '6', '7', '7', '1', '6']], 'patterns': [['1', '1', '5', '3',
'1', '3', '5', '5', '1', '1', '4', '2', '2', '4', '1', '7', '6', '6', '7', '1'], ['1', '5', '5', '3', '1', '1', '5', '3', '4', '2', '1', '4', '2', '1',
'1', '6', '7', '7', '1', '6']], 'leftSet': ['1', '1', '5', '3', '1', '3', '5', '5', '1', '1', '4', '2', '2', '4', '1', '7', '6', '6', '7', '1'], 'occ
urrences': [[0, 19], [23, 42]]
C:\>

```

The algorithm takes in a dataset as input and finds out all the patterns which repeat inside the dataset having their characters jumbled at every repetition. The algorithm also identifies the location of the pattern and its repetitions inside the dataset

Use Case

Introduction

IEEE 802.1 Qbv Conformance

A DUT (AVB switch) supporting Qbv has gates, where each gate is dedicated towards forwarding packets with a specific VLAN priority for a duration during which the gate is open.

Example – a switch supporting Qbv has 8 gates (1,2,3,4...8) where Gate -1 forwards packets with VLAN priority 1, Gate -2 forwards packet with VLAN priority 2 and so on. Now, if for a specific duration multiple gates are open, the gate can forward any combination of packets with valid VLAN priority.

Example – For time 250 us, Gate 2, Gate 3 and Gate 4 are open. So, during that duration, at Rx, we can receive packets with VLAN priority 2,3,4 respectively. In next cycle, during the same duration we can get packets with VLAN priority 2,4,3 respectively.

So, the VLAN priorities at Rx side forms a pattern like 1,2,3,4,5,6,9,1,1,2,4,3,5,1,4,3,2,5 and so on.

Requirement

Find the gate schedule from the packets found at the Rx side. Or in other words, we need to find the Qbv schedule pattern which are repeating at every clock cycle. The characters of the pattern can be jumbled in between.

How the jumbled pattern finder algorithm fits in

A flavor of the algorithm can be used to find the Qbv schedule with a minor modification –

Instead of finding the largest promising pattern as the most promising pattern, the clause of most promising pattern should be –

The time difference between the arrival time of last packet of a pattern and the arrival time of first packet of the next repetition should be minimum and constant.

The pattern will be used to predict the Qbv schedule information.

Output

Capture File as input -

Apply a display filter ... <Ctrl-/>							Expression...		
No.	Time	Source	Destination	Protocol	Length	Info			
165	8.548376608	22:22:22:22:00:04	44:44:44:44:00:04	0xffff	100	PRI: 4 DEI: 0 ID: 3			
166	8.548426608	22:22:22:22:00:05	44:44:44:44:00:05	0xffff	200	PRI: 5 DEI: 0 ID: 3			
167	8.548436608	22:22:22:22:00:06	44:44:44:44:00:06	0xffff	300	PRI: 6 DEI: 0 ID: 3			
168	8.548501618	22:22:22:22:00:02	44:44:44:44:00:02	0xffff	400	PRI: 2 DEI: 0 ID: 3			
169	8.548515185	00:00:00_00:00:00	00:00:00_00:00:00	0xffff	600	PRI: 3 DEI: 0 ID: 3			
170	8.548576625	00:00:00_00:00:00	00:00:00_00:00:00	0xffff	120	PRI: 7 DEI: 0 ID: 3			
171	8.548626593	00:00:00_00:00:00	00:00:00_00:00:00	0xffff	260	PRI: 1 DEI: 0 ID: 3			
172	8.548726593	00:00:00_00:00:00	00:00:00_00:00:00	0xffff	550	PRI: 2 DEI: 0 ID: 3			
173	8.548826593	00:00:00_00:00:00	00:00:00_00:00:00	0xffff	905	PRI: 6 DEI: 0 ID: 3			

Result

```
{
  "Cycle Time": 1000000,
  "Flow Details": {
    "0": {
      "Delay": 0,
      "Priority": 4,
      "Frame Size": 100,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "1": {
      "Delay": 50000,
      "Priority": 5,
      "Frame Size": 200,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "2": {
      "Delay": 60000,
      "Priority": 6,
      "Frame Size": 300,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "3": {
      "Delay": 126000,
      "Priority": 2,
      "Frame Size": 400,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "4": {
      "Delay": 175000,
      "Priority": 3,
      "Frame Size": 600,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "5": {
      "Delay": 201000,
      "Priority": 7,
      "Frame Size": 120,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "6": {
      "Delay": 250000,
      "Priority": 1,
      "Frame Size": 260,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "7": {
      "Delay": 350000,
      "Priority": 2,
      "Frame Size": 550,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "8": {
      "Delay": 450000,
      "Priority": 6,
      "Frame Size": 905,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "9": {
      "Delay": 500000,
      "Priority": 3,
      "Frame Size": 722,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "10": {
      "Delay": 601000,
      "Priority": 6,
      "Frame Size": 477,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "11": {
      "Delay": 675000,
      "Priority": 5,
      "Frame Size": 345,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "12": {
      "Delay": 750000,
      "Priority": 1,
      "Frame Size": 810,
      "VlanId": 3,
      "Frame Rate": 1000
    },
    "13": {
      "Delay": 851000,
      "Priority": 4,
      "Frame Size": 738,
      "VlanId": 3,
      "Frame Rate": 1000
    }
  },
  "Flow Count": 14
}
```

The pattern is the set of "Priority" present in JSON file in the output. All relevant data are mined from the packets having the priorities.

