Gap Analysis Using Algebra

R Glen Cooper
29 Jan 20

Introduction

This article shows how to solve a gaps and islands problem using simple algebra.

No SQL Server loops or cursors are required.

Problem

Scheduling constraints occasionally require the presence of some "feature" over all time intervals of fixed width. A typical example might be the following rule for rostering flight attendants:

> *Rosters of one month's duration must have at least 24 consecutive hours free of duty among any 7 consecutive days*

Dividing the month into 720 hours, a roster may be represented by a sequence of 720 binary digits (0s and 1s), where 0 represents rest and 1 represents work.

Then this constraint may be expressed as:

> *Sequences of 720 binary digits must have at least 24 consecutive 0s among any consecutive 168 binary digits*

More generally, for any $1 \leq L \leq M \leq N$:

> *Sequences of N binary digits must have at least L consecutive 0s among any consecutive M binary digits*

A SQL Server function for checking this constraint will be presented, using a select query and some high school algebra.

But first we need a few definitions.

A *work period* W is any sequence w1, w2, …, wN of binary digits.

Any subsequence of a work period containing just 0s is called a *rest period*.

A rest period is *maximal* if it is not contained in a larger rest period.

In the following work period of length 22, some of the rest periods are shown, one per line. Those that are maximal are shaded.

0	0	0	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	1	0	0
0																					
0	0																				
0	0	0																			
	0	0																			
				0																	
					0																
				0	0																
									0												
											0	0									

The following function defines the constraint for any W, N, M, L:

Sliding(W,N,M,L)

Return True if work period W of length N has every subsequence of length M containing at least L consecutive 0s. Otherwise, return False.

It assumes that $1 \leq L \leq M \leq N$.

For example, Sliding(W,720,168,24) tests the constraint in the above example for any W.

A brute-force method for Sliding(W,N,M,L) would be the following:

Start with the work period's left-most subsequence w1, w2, …, wM of length M and see if it contains at least L consecutive 0s. If successful, "slide" to the right by one position w2, w3, …, wM+1 and repeat. The constraint fails when fewer than L consecutive 0s can be found.

However, the actual computation takes an algebraic approach, which is much simpler.

First, we need another definition.

For a given W, N, M, L a maximal rest period is called *basic* if its length is at least L.

Clearly any pair of distinct basic periods are separated by at least one 1 (otherwise they wouldn't be maximal).

In the above example, if L = 2, the basic periods are shown below:

0	0	0	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	1	0	0
0	0	0																			
				0	0																
											0	0									
															0	0					
																				0	0

On the other hand, if L = 3, only the first maximal rest period is basic.

But if L = 1, then they are all basic.

The Sliding function computes its result by using a simple select query on the positions and lengths of the basic periods.

It's able to do this because of the following theorem, which does all the heavy lifting:

Sliding Theorem

The constraint fails if and only if at least one of the following conditions holds:

(1) There are no basic periods

(2) The starting position s of the first basic period satisfies:

$$s \geq M - L + 2$$

(3) The ending position e of the last basic period satisfies:

$$e \leq N - M + L - 1$$

(4) The ending position e of some basic period and the starting position s of the next basic period satisfy:

$$s - e \geq M - 2L + 3$$

Roughly speaking, the constraint fails if and only if a sufficiently large gap exists between a pair of adjacent basic periods, or the first one starts too late, or the last one ends too soon, or there aren't any basic periods.

The proof of this theorem may be found in the Resource section, along with the functions described here.

Therefore, we just need to set up a table of positions and lengths of the basic periods before checking the above inequalities with simple SQL.

To do that, we must first remove the non-basic periods by changing their 0s to 1, which may be done because of the following:

Key Observation

All non-basic rest periods may be removed by converting their 0s to 1 without affecting the constraint. That's because they'll never be used to verify the existence of L consecutive 0s since they have fewer than L 0s.

This reset is critical, since the proof of the Sliding theorem assumes that adjacent rest periods are always basic.

Remove Non-Basic Periods

A function called fnBasic is used to remove all non-basic rest periods. It accepts a string @String with width @Width before changing all maximal substrings of 0s to 1s whenever their length is less than Width.

For example, consider the string '0001001110'.

Let's see how fnBasic changes 0s to 1 for all maximal substring of 0s whose length < 2:

'0001001111' = dbo.fnBasic('0001001110',2)

First, it creates two identical tables @T and @T0 containing the positions Pos and values V for each character:

Pos	Val
1	0
2	0
3	0
4	1
5	0
6	0
7	1
8	1
9	1
10	0

Then it removes from @T0 those rows where Val = 1:

Pos	Val
1	0
2	0
3	0
5	0
6	0
10	0

Then it uses DENSE_RANK() in a common table expression for @T0:

```
;WITH cte0 AS
(
      SELECT
            Pos,
            DR = Pos - DENSE_RANK() OVER(ORDER BY Pos)
```

```
      FROM @T0
)
INSERT INTO @S(Start, Finish, Width)
SELECT
      Start = MIN(Pos),
      Finish = MAX(Pos),
      Width = MAX(Pos) - MIN(Pos) + 1
FROM cte0
GROUP BY DR
```

to create a new table @S:

Start	Finish	Width
1	3	3
5	6	2
10	10	1

listing the Start, Finish, and Width of each maximal rest period in the original string.

Note that DR = Pos - DENSE_RANK() is constant over rows where Pos is sequential (eg. 1,2,3), so we can group by DR to obtain minimum and maximum values of Pos on any maximal subsequence of 0s. That way, we can obtain a list of start and end positions for all maximal subsequences of 0s. These positions are inserted into the table @S.

Then it joins with @T to set its 0s to 1 for maximal rest periods whose Width < @Width:

```
UPDATE @T
SET Val = 1
WHERE Pos IN
(
SELECT Pos FROM @T d1
INNER JOIN @S d2
ON
d1.Pos >= d2.Start
AND
d1.Pos <= d2.Finish
AND
d2.Width < @Width
)
```

Pos	Val
1	0
2	0
3	0
4	1
5	0
6	0
7	1
8	1
9	1
10	1

Constraint Checker

At this point we can implement the constraint checker fnConstraint(W,M,L), where the length N of W is computed at run time:

```
ALTER FUNCTION [dbo].[fnConstraint]
(
@W    VARCHAR(MAX),
@M    int,
@L    int
)
RETURNS int
AS

BEGIN

-- Declarations
DECLARE @N            int
DECLARE @Result       VARCHAR(MAX)
DECLARE @T            TABLE (Pos INT IDENTITY, Val INT)
DECLARE @T0           TABLE (Pos INT IDENTITY, Val INT)
DECLARE @T1           TABLE (Pos INT IDENTITY, Val INT)
DECLARE @S            TABLE (Val INT, Start INT, Finish INT, Width INT)
DECLARE @I            int
DECLARE @Start        int
DECLARE @Finish       int
DECLARE @Width        int

-- Get length of work period
SET @N = len(@W)

-- Return NULL if parameters incorrect
IF @N IS NULL RETURN NULL
IF @N = 0 RETURN NULL
IF @L > @M RETURN NULL
IF @M > @N RETURN NULL
IF @L < 1 RETURN NULL

-- Change all maximal substrings of 0s to 1s if length < @L
-- This ensures that all maximal substrings of 0s are basic
-- The integrity checks below assume that this is the case
SET @W = dbo.fnBasic(@W,@L)

-- Insert characters of @W as integer-valued rows in three
-- identical tables
SET @I = 0
WHILE @I < @N
      BEGIN
      SET @I = @I +1
      INSERT INTO @T(Val)  VALUES (substring(@W,@I,1))
      INSERT INTO @T0(Val) VALUES (substring(@W,@I,1))
      INSERT INTO @T1(Val) VALUES (substring(@W,@I,1))
      END

-- Delete 0-valued rows in @T0
```

```
DELETE @T0 WHERE Val = 0

-- Delete 1-valued rows in @T1
DELETE @T1 WHERE Val = 1

-- Set up common table expressions for @T0 and @T1.
-- Note that DR = Pos - DENSE_RANK() is constant over
-- rows where Pos is sequential (eg. 5,6,7), so we
-- can group over DR to obtain minimum and maximum
-- values of Pos on any maximal subsequence of 0s or 1s.
-- That way, we can obtain a list of start and end
-- positions for all maximal subsequences of 0s or 1s.
-- Insert this list into a new table @S.
;WITH cte0 AS
(
      SELECT
            Pos,
            DR = Pos - DENSE_RANK() OVER(ORDER BY Pos)
      FROM @T0
),
cte1 AS
(
      SELECT
            Pos,
            DR = Pos - DENSE_RANK() OVER(ORDER BY Pos)
      FROM @T1
)
INSERT INTO @S(Val, Start, Finish, Width)
SELECT
      Val = 1,
      Start = MIN(Pos),
      Finish = MAX(Pos),
      Width = MAX(Pos) - MIN(Pos) + 1
FROM cte0
GROUP BY DR

UNION

SELECT
      Val = 0,
      Start = MIN(Pos),
      Finish = MAX(Pos),
      Width = MAX(Pos) - MIN(Pos) + 1
FROM cte1
GROUP BY DR

-- Set @Result to 0
SET @Result = 0

-- If no basic periods, Return = -1
IF NOT EXISTS (SELECT Val FROM @S WHERE Val = 0)
      BEGIN
      SET @Result = -1
      RETURN @Result
      END

-- If first basic period starts too late, Return = start position
```

```
SET @Start = (SELECT TOP 1 Start FROM @S WHERE Val = 0 ORDER BY Start)
IF @Start >= @M - @L + 2
      BEGIN
      SET @Result = @Start
      RETURN @Result
      END

-- If last basic period ends too early, Return = finish position
SET @Finish = (SELECT TOP 1 Finish FROM @S WHERE Val = 0 ORDER BY Start Desc )
IF @Finish <= @N - @M + @L - 1
      BEGIN
      SET @Result = @Finish
      RETURN @Result
      END

-- If adjacent basic periods are too far apart, Return = start of 1s
-- that are separating adjacent basic periods.
-- Note that the second basic period may not exist if the gap being examined
-- occurs at the end of @W.
SET @Width = (
SELECT TOP 1 Width FROM @S
WHERE Val = 1 AND Width + 1 >= @M - 2*@L + 3
ORDER BY Start
)
IF @Width IS NOT NULL
      BEGIN
      SET @Start = (
      SELECT TOP 1 Start FROM @S WHERE Val = 1 AND Width + 1 >= @M - 2*@L + 3
      ORDER BY Start
      )
      SET @Result = @Start
      RETURN @Result
      END

-- Return
RETURN @Result

END
```

The constraint checker fnConstraint returns the following values (rather than T or F):

If there are no basic sets, return -1
If there are no constraint violations, return 0
Otherwise, return the location of the first constraint violation found

Let see how this works for a specific example:

SELECT dbo.fnConstraint('0001001110',4,2)

First, the function fnBasic sets all non-basic 0s to 1:

```
SET @W = dbo.fnBasic(@W,@L)
```

so @W = ‘0001001111’.

Then it creates three identical tables @T, @T0, @T1 containing the positions Pos and values V for each character:

@T

Pos	Val
1	0
2	0
3	0
4	1
5	0
6	0
7	1
8	1
9	1
10	1

Then it removes from @T0 those rows where Val = 0, and removes from @T1 those rows where Val = 1:

@T0

Pos	Val
4	1
7	1
8	1
9	1
10	1

@T1

Pos	Val
1	0
2	0
3	0
5	0
6	0

Then, using DENSE_RANK, we can obtain a list of start and end positions of all maximal subsequences of 0s or 1s:

Val	Start	Finish	Width
0	1	3	3
1	4	4	1
0	5	6	2
1	7	10	4

Note that the Width of each row where Val = 1 represents the gap between adjacent basic periods. Also, the Start and Finish values where Val = 0 represents the starting and ending position of each basic period.

From this the Sliding Theorem determines if the constraint fails (and if it does, shows where the failure occurred).

In this case

6 = dbo.fnConstraint('0001001111', 4, 2)

which is returned by condition (3) in the Sliding Theorem.

General Gap Analysis

In the Resource section a stored procedure called spString2Table is included that enables simple gap analysis, such as computing the average gap between consecutive 0s or the largest subsequence of consecutive 0s.