# Symmetrical TSP segment operations documentation

Filip Hustic

November 2020

## 1 Concepts

TSP problem is represented with a matrix of distances between pairs of cities. In symmetric TSP the matrix of distances is symmetric(D = $D^T$). A solution of the problem $\pi$ is a permutation of city indexes which forms a tour $\pi = (c_1, c_2, ..., c_{n-1}, c_n)$, where $c_i$ is index of the $i$th city on the tour. Each solution has its own objective value equal to the length of the tour.
Here we are dealing with circular permutations, so:
$(c_1, c_2, ..., c_{n-1}, c_n) = (c_n, c_1, ..., c_{n-2}, c_{n-1}) = (c_n, c_{n-1}, ..., c_2, c_1)$

### 1.1 Reversal

A reversal $\varphi(i, j)$ is operation on permutation which reverses order of permutation elements from position $i$ to position $j$. Where the element at position $i$ is included in reversing process while the element at position $j$ is not included.
e.g. for permutation $\pi$ = (0 1 2 3 4 5)
$\varphi(0, 3)\pi$ = ( 2 1 0 3 4 5)
$\varphi(0, 4)\pi$ = ( 3 2 1 0 4 5)
Since $\pi$ is circular permutation following moves are possible:
$\varphi(4, 0)\pi$ = ( 0 1 2 3 5 4)
$\varphi(4, 2)\pi$ = ( 5 4 2 3 1 0)

### 1.2 Identity permutation

An Identity permutation $I$ is permutation in which for each adjacent pair of elements $\pi_i$ and $\pi_{i+1}$ holds:

$$|\pi_i - \pi_{i+1}| \in \{1, n-1\} \ \ , i = 0, ..., n-2$$

$$|\pi_0 - \pi_{n-1}| \in \{1, n-1\}$$

Where $n$ is the number of elements in permutation. A difference between the first and the last element is also checked because we are dealing with circular

Identity permutation.
e.g. $I = (0\ 1\ 2\ 3\ 4\ 5) = (5\ 0\ 1\ 2\ 3\ 4) = (5\ 4\ 3\ 2\ 1\ 0) = (4\ 3\ 2\ 1\ 0\ 5)$

## 1.3 Breakpoints

A break point of permutation $\pi$ is a point between adjacent pair of elements which are not adjacent in the Identity permutation.
e.g.
($\|$ 0 1 2 $\|$ 5 4 3), where pairs of elements (2,5) and (3,0) are not adjacent in the identity permutation
(1 $\|$ 3 2 $\|$ 4 5 0), where pairs of elements (1,3) and (2,4) are not adjacent in the identity permutation

In the proposed algorithm breakpoint $b_i$ is found at position $i$ if:

$$|\pi_i - \pi_{i-1}| \notin \{1, n-1\} \ \ , i = 1, ..., n-1$$

Or at position 0 if:

$$|\pi_0 - \pi_{n-1}| \notin \{1, n-1\}$$

## 1.4 Removing the breakpoints

A breakspoint can be removed by applying reversal $\varphi(i, j)$ on the permutation $\pi$, where breakpoints are detected at position $i$ and $j$ and elements $\pi_i$ and $\pi_j$ or $\pi_{i-1}$ and $\pi_{j-1}$ are adjacent in the Identity permutation.
e.g.
For $\pi = (\|\ 0\ 1\ 2\ \|\ 5\ 4\ 3)$ breakpoints are detected at positions $i = 0$ and $j = 3$. Since $\pi_0 = 0$ and $\pi_3 = 5$ are adjacent in $I$, by applying $\varphi(0, 3)\pi$ we get:
$\varphi(0, 3)\pi = (2\ 1\ 0\ 5\ 4\ 3)$, by which two breakpoints where eliminated.

e.g
For $\pi = (\|\ 0\ 1\ \|\ 3\ 2\ \|\ 5\ 4)$ and $i = 2$ and $j = 4$, elements left to the breakpoints, $\pi_1 = 1$ and $\pi_3 = 2$, are adjacent in $I$, by applying $\varphi(2, 4)\pi$ we get:
$\varphi(2, 4)\pi = (\|\ 0\ 1\ 2\ 3\ \|\ 5\ 4)$, where one breakpoint was eliminated.

e.g.
$\pi = (\ \|\ 3\ 2\ \|\ 5\ 4\ \|\ 1\ 0)$ This is a case when no breakpoints can be removed by applying just one reversal. In this case we can remove at least one breakpoint in two moves. First move would be reversal between any pair of breakpoints such that there is no other breakpoint between them.

Lets say bp1 is left breakpoint and bp2 is right breakpoint and there is no other breakpoint between bp1 and bp2. Now by applying $\varphi(bp1, bp2)$ element el that was on the right of bp1 will come on the left of bp2 and there will exist another element, bigger or smaller in $I$ to el that is also on the left of some other breakpoint. So now there exist a move that will remove at least one breakpoint. For example:
$\pi = (\ \|\ 3\ 2\ \|\ 5\ 4\ \|\ 1\ 0)$ $\quad$ $\varphi(2, 4)\pi = (\ \|\ 3\ 2\ \|\ 4\ 5\ \|\ 1\ 0)$

Before applying the move, element 4 was on the left to the breakpoint and elements 3 and 5 were both to the right of breakpoint. After applying $\varphi(2,4)$ to $\pi$ element 4 ended up on the right of the breakpoint, so applying reversal between positions of elements 3 and 4 will result in removal of breakpoint:
$\varphi(0,2)\pi = (\ |\ 2\ 3\ 4\ 5\ |\ 1\ 0)$

Same happened with element 5, which is now left to the breakpoint and so is element 0. Since 5 and 0 are adjacent in $I$ reversal between breakpoints on the right to 5 and 0 would also end up with removal of the breakpoint:
$\varphi(4,0)\pi = (\ |\ 3\ 2\ |\ 4\ 5\ 0\ 1)$

# 2 Implementation

In an API, operations on segments are implemented so they support geometric recombination which is used in a Sga solver. Geometric recombination is done by picking two solutions $s_1$ and $s_2$ and randomly sampling new solution which is on the path between two solutions. In the case of STSP implementation this sampling is supported using adaptation of sorting by reversals algorithm. Sorting by reversals objective is to remove all breakpoints in permutation by applying reversals to it. Once all breakpoints are removed Identity permutation is left and sorting is complete.

## 2.1 Data structures

Algorithm uses three data structures: solution, move and segment.

A solution is consist of int array data containing ordering of cities in the solution, n which is number of cities in a problem and objValue which is a length of traversal of cities in the solution. The solution contains more data structures which are used in other parts of API but not in this algorithm so they are not mentioned.

```
struct solution{
  int *data;
  int n;
  double objValue;
};
```

A move is consist of 2 dimensional int array. Elements of the array are two indexes between which reversal will be applied in the solution or in a segment. e.g. data[0,4] refers to reversal $\varphi(0,4)$

```
struct move{
  int data[2];
};
```

The segment is the main data structure with which algorithm works. Each of its members will be described in detail in the next section.

```
struct segment {
    int n;
    int *data;
    int *datai;
    int *breakPoints;
    int *breakPointsi;
    int numBp;
    int *s2_datai;
};
```

## 2.2 Initialization of the segment

A segment represents mapping of the solution s1 elements to positions those elements have in the solution s2. This mapping is represented by data array. Beside the mapping, segment consists of breakpoints information stored in arrays datai, breakPoints and breakPointsi. The algorithm objective is to construct the moves which will remove the breakpoints from the segment, leading to transformation of segment data to the identity matrix $I$. Applying the same moves to the solution s1 will lead to transformation of s1 data to s2 data.

Initialization of the segment is done in two steps. First the segment data is constructed and then breakpoint information is obtained from the segment data.

1) Constructing segment data
Like said, here we map solution s1 elements to positions those elements have in solution s2. This information is saved in the segment data. The segment data is constructed using formula:

$$seg \rightarrow data[i] = s2 \rightarrow data\_inv[s1 \rightarrow data[i]] \quad i = 0, .., n-1$$

Where $s2 \rightarrow data\_inv$ is inverse of $s2 \rightarrow data$.

e.g.
$s_1 \rightarrow$ data = ( 1 0 3 4 5 2 ) $s_2 \rightarrow$ data = ( 3 0 4 5 2 1 )
Where inverse of s2 is: $s_2 \rightarrow$ data_inv = (1 5 4 0 2 3)
Segment data is: $seg \rightarrow$ data = (5 1 0 2 3 4)
And segment data inverse: $seg \rightarrow$ data_inv = (2 1 3 4 5 0)

The first element of $s_1[0] = 1$ is on index 5 in $s_2$, so $seg[0] = 5$.
The second element of $s_1[1] = 0$ is on index 1 in $s_2$, so $seg[1] = 1$.
The third element of $s_1[2] = 3$ is on index 0 in $s_2$, so $seg[2] = 0$.
And so on...

2) Breakpoint information
Second step of the segment initialization is finding breakpoints in the segment. Position of each active breakpoint is stored in $seg \rightarrow breakpoints$. Additional information is stored in $seg \rightarrow breakpointsi$ which is inverse of $seg \rightarrow breakpoints$ with addition that if there is no active breakpoint on position $i$, so there is no element $i$ in $seg \rightarrow breakpoints$, $seg \rightarrow breakpointsi[i]$ = -1. This additional information about existing/nonexistent breakpoint at position $i$ will latter be

used in other functions. The segment stores one more additional information which is the number of the active breakpoints $seg \rightarrow numBp$.

e.g.
$seg \rightarrow$data = ( | 3 2 | 5 4 | 1 0)
$seg \rightarrow$breakPoints = (0 2 4)
$seg \rightarrow$breakPointsi = (0 -1 1 -1 2 -1)
$seg \rightarrow$numBp = 3

Code used to initialize segment:

```
1  struct segment *initSegment(struct segment *seg, const struct
       solution *s1, const struct solution *s2){
2    int i,n = s1->n, diff;
3
4    for(i = 0; i < n; ++i)
5      seg->s2_datai[s2->data[i]] = i;
6
7    for(i = 0; i < n; ++i){
8      seg->data[i] = seg->s2_datai[s1->data[i]];
9      seg->datai[seg->data[i]] = i;
10   }
11
12   seg->numBp = 0;
13   // Check if elements at first and last position are adjacent
14   diff =  abs( seg->data[0] - seg->data[n - 1] );
15   if(diff != 1 && diff != (n - 1) ){
16     seg->breakPoints[seg->numBp++] = 0;
17     seg->breakPointsi[0] = 0;
18   }else{
19     seg->breakPointsi[0] = -1;
20   }
21
22   for(i = 1; i < n; ++i){
23     diff = abs( seg->data[i] - seg->data[i - 1] );
24     if( diff != 1 && diff != (n - 1) ){
25       seg->breakPoints[seg->numBp] = i;
26       seg->breakPointsi[i] = seg->numBp;
27       seg->numBp ++;
28     }
29     else{
30       seg->breakPointsi[i] = -1;
31     }
32   }
33   return seg;
34 }
```

The segment data is constructed in the first part of the code, lines 4 - 10. After that breakpoint information is obtained. If two adjacent elements are not adjacent in the Identity permutation, there exists a breakpoint between them and the position of the breakpoint is stored in seg$\rightarrow$ *breakpoints*. Along with that the position of breakpoint in seg$\rightarrow$ *breakpoints* is stored in seg$\rightarrow$ *breakpointsi*. We are dealing with circular permutation, so code first checks for the breakpoint between elements at position 0 and *n-1*, lines 14-20, and then it

does checks between all other adjacent pairs of elements, lines 22-31.

## 2.3 Random move towards

Random move towards constructs a move such that applying it(applying reversal) to the segment results in removal of at least one breakpoint. If there doesn't exist such a move, a adjustment move is constructed. By applying the adjustment move to the segment, there will exist move for the adjusted segment that will remove at least one breakpoint.

Code for constructing random move towards:

```
1  struct move *randomMoveTowards(struct move *v, struct segment *seg)
       {
2    int r,bp1,bp2, numBpCopy = seg->numBp;
3
4    if(seg->numBp == 0)
5      return NULL;
6
7    //While there are unexamined breakpoints loop through them
8    while(numBpCopy){
9      r = randint(numBpCopy - 1);
10     bp1 = seg->breakPoints[r];
11     bp2 = findSuitableBreakPoint(seg, bp1);
12
13     if(bp2 != -1){
14       v->data[0] = bp1;
15       v->data[1] = bp2;
16       return v;
17     }
18     numBpCopy--;
19     swap(seg->breakPoints, r, numBpCopy);
20     swap(seg->breakPointsi, bp1, seg->breakPoints[r]);
21   }
22
23   // If non was found pick randomly two neighbouring bp
24   r = randint(seg->numBp - 1);
25   bp1 = seg->breakPoints[r];
26   for( bp2 = bp1 + 2;; ++bp2)
27     if( seg->breakPointsi[bp2 % n] != -1 )
28       break;
29   v->data[0] = bp1;
30   v->data[1] = bp2 % n;
31   return v;
32 }
```

In one iteration of the while loop algorithm randomly picks one breakpoint from the active breakpoints, lines 9-10. In line 11 auxiliary function *findSuitableBreakPoint* is called which returns position of the second breakpoint bp2 such that $\varphi(bp1, bp2)seg \rightarrow data$ results in removal of at least one breakpoint. If there does exist such bp2, bp1 and bp2 are returned as the move, line 13-16. If there doesn't exist such bp2, *findSuitableBreakPoint* returned -1, bp1 is swapped in $seg \rightarrow breakPoints$ with a breakpoint at a position numBpCopy,

line 19. By doing this bp1 is disabled from being picked again in the next iteration. Since $seg \rightarrow breakPoints$ was changed, $seg \rightarrow breakPointsi$ also needs to be changed and swap between element bp1 and element which was at position numBpCopy in $seg \rightarrow breakPoints$ is done, line 20.

e.g. seg->data = [| 0 | 4 5 6| 2 1| 3]
breakPoints = [0 1 4 6]
breakPointsi = [0 1 -1 -1 2 -1 3]
numBpCopy = 4

First Iteration:
r = 0 //randomly picked
bp1 = breakPoints[r] = 0
Possible moves: $\varphi(0,1)$, $\varphi(0,4)$, $\varphi(0,6)$
bp2 = -1 // none of the moves result in the removal of breakpoint
swap(breakPoints, r, numBpCopy)
→ breakpoints = [6 1 4 0]
swap(breakPointsi, bp1, breakPoints[r]), breakPoints[r] = 6
→ breakpointsi = [3 1 -1 -1 2 -1 0]

Second Iteration:
r = 2 //randomly picked
bp1 = 4
Possible moves: $\varphi(4,6)$ $\varphi(4,0)$, $\varphi(4,1)$
bp2 = 6 // $\varphi(4,6)$ results in removal of breakpoint

Since $\varphi(4,6)$ results in removal of break point: $\varphi(4,6)$data = [| 0 | 4 5 6| 1 2 3]
[4,6] is returned as the move.

If the while loop iterated through all active breakpoints and there wasn't suitable pair of breakpoints found, algorithm picks one breakpoint bp1 randomly, lines 24-25, and finds first breakpoint bp2 that is on the right from bp1. To do so for loop is used, lines 26-27. Loop increments bp2 until it finds first position bp2 % n with active breakpoint on it. Applying move $\varphi(bp1, bp2 \ \% \ n)$ to the segment will set segment for removal of at least one breakpoint in the next algorithm iteration. The example of this is given in subsection 1.4. It is important to notice that for loop starts from the position bp1 + 2 because reversal $\varphi(bp1, bp1+1)$ doesn't result with any change.

## 2.4 Find suitable second breakpoint

A breakspoint can be removed by applying reversal $\varphi(bp1, bp2)$ on the permutation $\pi$, where breakpoints are detected at positions bp1 and bp2 and elements $\pi_{bp1}$ and $\pi_{bp2}$ or $\pi_{bp1-1}$ and $\pi_{bp2-1}$ are adjacent in the Identity permutation $I$.

Code for finding bp2:

```
1  int findSuitableBreakPoint(struct segment *seg, int bp1){
2    int elemAtBp, biggerAdjPos, smallerAdjPos , n = seg->n;
3
4    // Check element right to the breakpoint
5    elemAtBp = seg->data[bp1];
6    biggerAdjPos = seg->datai[(elemAtBp + 1) % n];
7    smallerAdjPos = seg->datai[(elemAtBp - 1 + n) % n];
8
9    if(seg->breakPointsi[biggerAdjPos] != -1)
10     return biggerAdjPos;
11   else if(seg->breakPointsi[smallerAdjPos] != -1)
12     return smallerAdjPos;
13
14   // Check element left to the breakpoint
15   elemAtBp = seg->data[(bp1 - 1 + n) % n];
16   biggerAdjPos = seg->datai[(elemAtBp + 1) % n];
17   smallerAdjPos = seg->datai[(elemAtBp - 1 + n) % n];
18
19
20   if(seg->breakPointsi[(biggerAdjPos + 1) % n] != -1)
21     return (biggerAdjPos + 1) % n;
22   if(seg->breakPointsi[(smallerAdjPos + 1) % n] != -1)
23     return (smallerAdjPos + 1) % n;
24
25   return -1;
26 }
```

The algorithm first looks at a element right to the breakpoint bp1 and finds positions of bigger and smaller element to it (elements adjacent to elemAtBp in the identity permutation $I$), lines 5-7. If there is active breakpoint on the right to the bigger or smaller element, position of that breakpoint is returned, lines 9-12. Reminder, $seg \rightarrow breakPointsi[i]$ has value -1 if there is no active breakpoint at position $i$ and some value $\geq 0$ if there is active breakpoint at position $i$.

If nothing was returned element left to the bp1 is obtained and bigger and smaller element positions are found again, lines 15-17. The algorithm now checks if one of those positions are also to the left of some breakpoint and if so position of that breakpoint is returned, lines 20-23.

If again nothing is returned there doesn't exist bp2 such that reversal between bp1 and bp2 removes breakpoint and -1 is returned for indication.

## 2.5 Applying the move to segment

Applying the move to segment consists of applying reversal to seg→data. Reversal is determined by the move which is returned in randomMoveTowards. After the reversal is applied new breakpoints can be removed while new ones can appear.

Code for applying the move to segment:

```
1  struct segment *applyMoveToSegment(struct segment *seg, const
       struct move *v){
2    int i, j, diff, n = seg->n;
3    // Applying the move to data
4    reversal(seg->data, v, seg->n);
5
6    // Apply change that occurred to datai, breakPoints and
       breakPointsi
7    for(i = (v->data[0] - 1 + n) % n; i != v->data[1]; i = j ){
8      j = (i + 1) % n;
9      seg->datai[seg->data[j]] = j;
10
11     diff = abs(seg->data[i] - seg->data[j]);
12     if( diff == 1 || diff == (n - 1) ){ //If elements are adjacent
13       if( seg->breakPointsi[j] != -1 ){ //And there is bp in
       between them
14         seg->numBp --;
15         seg->breakPointsi[seg->breakPoints[seg->numBp]] = seg->
       breakPointsi[j];
16         swap(seg->breakPoints, seg->breakPointsi[j], seg->numBp);
17         seg->breakPointsi[j] = -1;
18       }
19     }else if( seg->breakPointsi[j] == -1 ){
20         seg->breakPoints[seg->numBp] = j;
21         seg->breakPointsi[j] = seg->numBp;
22         seg->numBp ++;
23     }
24
25   }
26   return seg;
27 }
```

The algorithm first applies reversal $\varphi(bp1, bp2)$ to $seg \rightarrow data$, line 4. Because of the reversal there may have been removal of some breakpoints while some new breakpoints could have appeared between positions bp1 and bp2. Because of that algorithm loops from starting position $i$ = bp1-1 to the position $i$ = bp2. In each iteration $j$ is calculated to be adjacent to the $i$. In calculations modulo n is used because $seg \rightarrow data$ is circular permutation.

Now at each iteration algorithm calculates value difference between elements at position $i$ and $j$, line 11. If those elements are adjacent in $I$, difference is equal to 1 or n-1. If that is a case there shouldn't be a breakpoint between them and that means there shouldn't be a breakpoint at position $j$. If it turns out that there is a breakpoint at position $j$, line 13, that breakpoint should be removed from $seg \rightarrow breakPoints$ and change should also be applied to $seg \rightarrow breakPointsi$ and $seg \rightarrow numNp$ should be reduced, lines 14-17.

On the other hand if difference wasn't equal to 1 or n-1, there should be breakpoint at position $j$. If it turns out that there isn't a breakpoint at postion $j$, line 19, breakpoint will be added to $seg \rightarrow breakPoints$, position of that breakpoint in $seg \rightarrow breakPoints$ will be updated in $seg \rightarrow breakPointsi$ and $seg \rightarrow num$ will be increased, lines 19-22.

Also since reversal has changed positions of elements in $seg \rightarrow data$, in each iteration $seg \rightarrow datai$ is updated with new position of elements, line 9.

e.g. $seg \rightarrow$data = [5 | 1 | 4 3 | 0 | 2]
datai = [4 1 5 3 2 0]
breakPoints = [1 2 4 5]
breakPointsi = [-1 0 1 -1 2 3]

After $\varphi(2,5)data$ = [5 | 1 0 | 3 4 | 2], we can see that breakpoints on positions 2 and 4 disappeared while new one on position 3 appeared. New state of the segment is:

seg->data = [5 | 1 0 | 3 4 | 2]
datai = [2 1 5 3 4 0]
breakPoints = [1 3 5]
breakPointsi = [-1 0 -1 1 -1 2]

## 2.6 Final example

Like said before if we apply same moves to the segment and to solution s1, segment data will tend to transform itself into identity matrix $I$ while solution s1 will tend to transform itself into s2.

e.g. $s1$ = [1 5 3 0 2 4] $s2$ = [3 4 0 2 5 1]

$seg$ = [ | 5 4 | 0 | 2 3 | 1]

$\varphi(3,5)seg$ = [ | 5 4 | 0 | 3 2 1]
$\varphi(3,5)s1$ = [1 5 3 2 0 4]

$\varphi(0,2)seg$ = [ | 4 5 0 | 3 2 1]
$\varphi(0,2)s1$ = [5 1 3 2 0 4]

$\varphi(0,3)seg$ = [0 5 4 3 2 1] = $I$
$\varphi(0,3)s1$ = [3 1 5 2 0 4] = s2

Here we are dealing with circular permutation so s1 really transformed itself into s2.