

Swap Distance

Manuel Eberl

February 10, 2026

Given two lists that are permutations of one another, the *swap distance* (also known as the *Kendall tau distance*) is the minimum number of swap operations of adjacent elements required to make the two lists the same.

Equivalently, the swap distance of two finite linear orders \preceq and \trianglelefteq is the number of disagreements of the two orders, i.e. of pairs (x, y) such that $x \prec y$ and $y \triangleleft x$.

This article defines these two notions of swap distance as well as their equivalence under the obvious isomorphism between lists and linear orders given by interpreting a list as a *ranking* of elements in descending order.

An efficient $O(n \log n)$ algorithm to compute the swap distance is also provided via the connection to the number of inversions of a list, for which an efficient algorithm is already available in the AFP.

Contents

1	The swap distance	3
1.1	Preliminaries	3
1.2	The swap distance of two linear orders	4
1.3	The swap distance of two lists	7
1.4	The relationship between swap distance and inversions	8
1.5	Swapping adjacent list elements	9
1.6	Swapping non-adjacent list elements	11
1.7	Swap distance as minimal number of adjacent swaps to make two lists equal	11

1 The swap distance

```
theory Swap-Distance
  imports Rankings.Rankings List-Inversions.List-Inversions
begin
```

The swap distance (also known as the Kendall tau distance) of two finite linear orders R, S is the number of pairs (x, y) such that $(x, y) \in R$ and $(y, x) \in S$.

By using the obvious correspondence between finite linear orders and lists of fixed length, the notion is transferred to lists. In this case, an alternative interpretation of the swap distance is as the smallest number of swaps of adjacent elements one can perform in order to make one list match the other one.

The swap distance is strongly related to the number of inversions of a list of linearly-ordered elements: if we rename the elements from 1 to n such that the first list becomes $[1, \dots, n]$, the swap distance is exactly the number of inversions in the second list.

This correspondence can be used to compute the swap distance in $O(n \log n)$ time using the merge sort inversion count algorithm (which is available in the AFP).

1.1 Preliminaries

```
primrec find-index-aux :: nat ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ nat where
  find-index-aux acc P [] = acc
  | find-index-aux acc P (x # xs) = (if P x then acc else find-index-aux (acc+1) P xs)
```

```
lemma find-index-aux-correct: find-index-aux acc P xs = find-index P xs + acc
  ⟨proof⟩
```

```
lemma find-index-aux-code [code]: find-index P xs = find-index-aux 0 P xs
  ⟨proof⟩
```

```
lemma inversions-map:
  fixes xs :: 'a :: linorder list
  assumes strict-mono-on (set xs) f
  shows inversions (map f xs) = inversions xs
  ⟨proof⟩
```

```
lemma inversion-number-map:
  fixes xs :: 'a :: linorder list
  assumes strict-mono-on (set xs) f
  shows inversion-number (map f xs) = inversion-number xs
  ⟨proof⟩
```

```
lemma inversion-number-Cons:
  inversion-number (x # xs) = length (filter (λy. y < x) xs) + inversion-number xs
  ⟨proof⟩
```

```

fun (in preorder) inversion-number-between-sorted-aux :: nat  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  nat where
  inversion-number-between-sorted-aux acc [] ys = acc
  | inversion-number-between-sorted-aux acc xs [] = acc
  | inversion-number-between-sorted-aux acc (x # xs) (y # ys) =
    (if  $\neg$ less y x then
      inversion-number-between-sorted-aux acc xs (y # ys)
    else
      inversion-number-between-sorted-aux (acc + length (x # xs)) (x # xs) ys)

lemma inversion-number-between-sorted-aux-correct:
  inversion-number-between-sorted-aux acc xs ys = acc + inversion-number-between-sorted xs ys
   $\langle$ proof $\rangle$ 

lemma inversion-number-between-sorted-code [code]:
  inversion-number-between-sorted xs ys = inversion-number-between-sorted-aux 0 xs ys
   $\langle$ proof $\rangle$ 

```

1.2 The swap distance of two linear orders

We first define the set of “discrepancies” between the two orders.

```

definition swap-dist-relation-aux :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\times$  'a) set
where
  swap-dist-relation-aux R1 R2 = {(x,y). R1 x y  $\wedge$   $\neg$ R1 y x  $\wedge$  R2 y x  $\wedge$   $\neg$ R2 x y}

```

On a linear order, the following simpler definition holds.

```

lemma swap-dist-relation-aux-def-linorder:
  assumes linorder-on A R1 linorder-on A R2
  shows swap-dist-relation-aux R1 R2 = {(x,y). R1 x y  $\wedge$   $\neg$ R2 x y}
   $\langle$ proof $\rangle$ 

```

```

lemma swap-dist-relation-aux-same [simp]: swap-dist-relation-aux R R = {}
   $\langle$ proof $\rangle$ 

```

```

lemma swap-dist-relation-aux-commute: swap-dist-relation-aux R1 R2 = prod.swap ` swap-dist-relation-aux
R2 R1
   $\langle$ proof $\rangle$ 

```

```

lemma swap-dist-relation-aux-commute': bij-betw prod.swap (swap-dist-relation-aux R1 R2) (swap-dist-relation-aux
R2 R1)
   $\langle$ proof $\rangle$ 

```

```

lemma swap-dist-relation-aux-dual:
  swap-dist-relation-aux R1 R2 = prod.swap ` swap-dist-relation-aux ( $\lambda$ x y. R1 y x) ( $\lambda$ x y. R2 y
x)
   $\langle$ proof $\rangle$ 

```

```

lemma swap-dist-relation-aux-triangle:

```

```

assumes linorder-on A R1 linorder-on A R2 linorder-on A R3
shows swap-dist-relation-aux R1 R3 ⊆ swap-dist-relation-aux R1 R2 ∪ swap-dist-relation-aux
R2 R3
⟨proof⟩

```

```

lemma finite-swap-dist-relation-aux:
assumes linorder-on A R1 finite A linorder-on B R2 finite B
shows finite (swap-dist-relation-aux R1 R2)
⟨proof⟩

```

```

lemma split-Bex-pair-iff: (Ǝ z ∈ A. P z) ←→ (Ǝ x y. (x, y) ∈ A ∧ P (x, y))
⟨proof⟩

```

```

lemma swap-dist-relation-aux-comap-relation:
assumes inj-on f A linorder-on A R linorder-on A S
shows swap-dist-relation-aux (comap-relation f R) (comap-relation f S) = map-prod f f ` swap-dist-relation-aux R S
  (is ?lhs = ?rhs)
⟨proof⟩

```

```

lemma swap-dist-relation-aux-restrict-subset:
swap-dist-relation-aux (restrict-relation A R) (restrict-relation A S) ⊆
swap-dist-relation-aux R S
⟨proof⟩

```

The swap distance is then simply the number of such discrepancies.

```

definition swap-dist-relation :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ nat where
  swap-dist-relation R1 R2 = card (swap-dist-relation-aux R1 R2)

```

```

lemma swap-dist-relation-same [simp]: swap-dist-relation R R = 0
⟨proof⟩

```

```

lemma swap-dist-relation-commute: swap-dist-relation R1 R2 = swap-dist-relation R2 R1
⟨proof⟩

```

```

lemma swap-dist-relation-dual:
  swap-dist-relation R1 R2 = swap-dist-relation (λx y. R1 y x) (λx y. R2 y x)
⟨proof⟩

```

```

lemma swap-dist-relation-triangle:
assumes linorder-on A R1 linorder-on A R2 linorder-on A R3 finite A
shows swap-dist-relation R1 R3 ≤ swap-dist-relation R1 R2 + swap-dist-relation R2 R3
⟨proof⟩

```

```

lemma swap-dist-relation-aux-empty-iff:
assumes linorder-on A R linorder-on A S
shows swap-dist-relation-aux R S = {} ←→ R = S
⟨proof⟩

```

```

lemma swap-dist-relation-eq-0-iff:
  assumes linorder-on A R linorder-on A S finite A
  shows swap-dist-relation R S = 0  $\longleftrightarrow$  R = S
  (proof)

lemma swap-dist-relation-comap-relation:
  assumes inj-on f A linorder-on A R linorder-on A S
  shows swap-dist-relation (comap-relation f R) (comap-relation f S) = swap-dist-relation R S
  (proof)

```

```

lemma swap-dist-relation-le:
  assumes preorder-on A R1 preorder-on A R2 finite A
  shows swap-dist-relation R1 R2  $\leq$  (card A) choose 2
  (proof)

```

The swap distance reaches its maximum of $n(n - 1)/2$ if and only if the two orders are inverse to each other.

```

lemma swap-dist-relation-inverse:
  assumes linorder-on A R finite A
  shows swap-dist-relation R ( $\lambda x y. R y x$ ) = (card A) choose 2
  (proof)

```

```

lemma swap-dist-relation-maximal-imp-inverse:
  assumes preorder-on A R1 preorder-on A R2 finite A
  assumes swap-dist-relation R1 R2  $\geq$  (card A) choose 2
  shows R2 = ( $\lambda y x. R1 x y$ )
  (proof)

```

```

lemma swap-dist-relation-maximal-iff-inverse:
  assumes linorder-on A R1 linorder-on A R2 finite A
  shows swap-dist-relation R1 R2 = (card A) choose 2  $\longleftrightarrow$  R2 = ( $\lambda y x. R1 x y$ )
  (proof)

```

```

lemma swap-dist-relation-restrict:
  assumes linorder-on B R linorder-on B S finite B
  shows swap-dist-relation (restrict-relation A R) (restrict-relation A S)  $\leq$ 
    swap-dist-relation R S
  (proof)

```

If the restriction of two relations to some set A has the same swap distance as the full relations, the two relations must agree everywhere except inside A .

```

lemma swap-dist-relation-restrict-eq-imp-eq:
  fixes R S A B
  assumes linorder-on A R linorder-on A S finite A
  defines R'  $\equiv$  restrict-relation B R
  defines S'  $\equiv$  restrict-relation B S
  assumes swap-dist-relation R' S'  $\geq$  swap-dist-relation R S

```

```

assumes  $xy: x \notin B \vee y \notin B$ 
shows  $R x y \longleftrightarrow S x y$ 
⟨proof⟩

```

1.3 The swap distance of two lists

The swap distance of two lists is defined as the swap distance of the relations they correspond to when interpreting them as rankings of “biggest” to “smallest”.

```

definition swap-dist :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  nat where
  swap-dist xs ys =
    (if distinct xs  $\wedge$  distinct ys  $\wedge$  set xs = set ys
     then swap-dist-relation (of-ranking xs) (of-ranking ys) else 0)

lemma swap-dist-le: swap-dist xs ys  $\leq$  (length xs) choose 2
⟨proof⟩

lemma swap-dist-same [simp]: swap-dist xs xs = 0
⟨proof⟩

lemma swap-dist-commute: swap-dist xs ys = swap-dist ys xs
⟨proof⟩

lemma swap-dist-rev [simp]: swap-dist (rev xs) (rev ys) = swap-dist xs ys
⟨proof⟩

lemma swap-dist-rev-left: swap-dist (rev xs) ys = swap-dist xs (rev ys)
⟨proof⟩

lemma swap-dist-triangle:
  assumes set xs = set ys distinct ys
  shows swap-dist xs zs  $\leq$  swap-dist xs ys + swap-dist ys zs
⟨proof⟩

lemma swap-dist-eq-0-iff:
  assumes distinct xs distinct ys set xs = set ys
  shows swap-dist xs ys = 0  $\longleftrightarrow$  xs = ys
⟨proof⟩

lemma swap-dist-pos-iff:
  assumes distinct xs distinct ys set xs = set ys
  shows swap-dist xs ys  $> 0 \longleftrightarrow$  xs  $\neq$  ys
⟨proof⟩

lemma swap-dist-map:
  assumes inj-on f (set xs  $\cup$  set ys)
  shows swap-dist (map f xs) (map f ys) = swap-dist xs ys
⟨proof⟩

```

The swap distance reaches its maximum of $n(n - 1)/2$ iff the two lists are reverses of

one another.

```

lemma swap-dist-rev-same:
  assumes distinct xs
  shows swap-dist xs (rev xs) = (length xs) choose 2
  {proof}

lemma swap-dist-maximalD:
  assumes set xs = set ys distinct xs distinct ys
  assumes swap-dist xs ys ≥ (length xs) choose 2
  shows ys = rev xs
  {proof}

lemma swap-dist-maximal-iff:
  assumes set xs = set ys distinct xs distinct ys
  shows swap-dist xs ys = (length xs) choose 2 ↔ ys = rev xs
  {proof}

lemma swap-dist-append-left:
  assumes distinct zs
  assumes set zs ∩ set xs = {} set zs ∩ set ys = {}
  shows swap-dist (zs @ xs) (zs @ ys) = swap-dist xs ys
  {proof}

lemma swap-dist-append-right:
  assumes distinct zs
  assumes set zs ∩ set xs = {} set zs ∩ set ys = {}
  shows swap-dist (xs @ zs) (ys @ zs) = swap-dist xs ys
  {proof}

lemma swap-dist-Cons-same:
  assumes z ∉ set xs ∪ set ys
  shows swap-dist (z # xs) (z # ys) = swap-dist xs ys
  {proof}

lemma swap-dist-swap-first:
  assumes distinct (x # y # xs)
  shows swap-dist (x # y # xs) (y # x # xs) = 1
  {proof}

```

1.4 The relationship between swap distance and inversions

The swap distance between a list xs containing the numbers $0, \dots, n-1$ and the list $[0, \dots, n-1]$ is exactly the number of inversions of xs .

```

lemma swap-dist-zero-upt-n:
  assumes mset xs = mset-set {0..<n}
  shows swap-dist [0..<n] xs = inversion-number xs
  {proof}

```

Hence, computing the swap distance of two arbitrary lists can be reduced to computing the number of inversions of a list by renaming all the elements such that the first list becomes $[0, \dots, n - 1]$.

```

lemma swap-dist-conv-inversion-number:
  assumes distinct: distinct xs distinct ys and set-eq: set xs = set ys
  shows swap-dist xs ys = inversion-number (map (index xs) ys)
  ⟨proof⟩

lemma swap-dist-code' [code]:
  swap-dist xs ys =
    (if distinct xs  $\wedge$  distinct ys  $\wedge$  set xs = set ys then
      inversion-number (map (index xs) ys)  $\text{else}$  0)
  ⟨proof⟩

```

1.5 Swapping adjacent list elements

```

definition swap-adj-list :: nat  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  swap-adj-list i xs = (if Suc i < length xs then xs[i := xs ! Suc i, Suc i := xs ! i]  $\text{else}$  xs)

```

```

lemma length-swap-adj-list [simp]: length (swap-adj-list i xs) = length xs
  ⟨proof⟩

```

```

lemma distinct-swap-adj-list-iff [simp]:
  distinct (swap-adj-list i xs)  $\longleftrightarrow$  distinct xs
  ⟨proof⟩

```

```

lemma mset-swap-adj-list [simp]:
  mset (swap-adj-list i xs) = mset xs
  ⟨proof⟩

```

```

lemma set-swap-adj-list [simp]:
  set (swap-adj-list i xs) = set xs
  ⟨proof⟩

```

```

lemma swap-adj-list-append-left:
  assumes i  $\geq$  length xs
  shows swap-adj-list i (xs @ ys) = xs @ swap-adj-list (i - length xs) ys
  ⟨proof⟩

```

```

lemma swap-adj-list-Cons:
  assumes i > 0
  shows swap-adj-list i (x # xs) = x # swap-adj-list (i - 1) xs
  ⟨proof⟩

```

```

lemma swap-adj-list-append-right:
  assumes Suc i < length xs
  shows swap-adj-list i (xs @ ys) = swap-adj-list i xs @ ys
  ⟨proof⟩

```

```

lemma swap-dist-swap-adj-list:
  assumes Suc i < length xs distinct xs
  shows swap-dist xs (swap-adj-list i xs) = 1
  (proof)

fun swap-adjs-list :: nat list  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  swap-adjs-list [] xs = xs
  | swap-adjs-list (i # is) xs = swap-adjs-list is (swap-adj-list i xs)

lemma length-swap-adjs-list [simp]: length (swap-adjs-list is xs) = length xs
  (proof)

lemma distinct-swap-adjs-list-iff [simp]:
  distinct (swap-adjs-list is xs)  $\longleftrightarrow$  distinct xs
  (proof)

lemma mset-swap-adjs-list [simp]:
  mset (swap-adjs-list is xs) = mset xs
  (proof)

lemma set-swap-adjs-list-list [simp]:
  set (swap-adjs-list is xs) = set xs
  (proof)

lemma swap-adjs-list-append:
  swap-adjs-list (is @ js) xs = swap-adjs-list js (swap-adjs-list is xs)
  (proof)

lemma swap-adjs-list-append-left:
  assumes  $\forall i \in \text{set is}. i \geq \text{length xs}$ 
  shows swap-adjs-list is (xs @ ys) = xs @ swap-adjs-list (map ( $\lambda i. i - \text{length xs}$ ) is) ys
  (proof)

lemma swap-adjs-list-Cons:
  assumes 0  $\notin$  set is
  shows swap-adjs-list is (x # xs) = x # swap-adjs-list (map ( $\lambda i. i - 1$ ) is) xs
  (proof)

lemma swap-adjs-list-append-right:
  assumes  $\forall i \in \text{set is}. \text{Suc } i < \text{length xs}$ 
  shows swap-adjs-list is (xs @ ys) = swap-adjs-list is xs @ ys
  (proof)

```

Swapping two adjacent elements either increases or decreases the swap distance by 1, depending on the orientation of the swapped pair in the other relation.

```

lemma swap-dist-relation-of-ranking-swap:
  assumes distinct (xs @ x # y # ys)
  shows swap-dist-relation R (of-ranking (xs @ x # y # ys)) + (if y  $\prec[R]$  x then 1 else 0) =
    swap-dist-relation R (of-ranking (xs @ y # x # ys)) + (if x  $\prec[R]$  y then 1 else 0)

```

$\langle proof \rangle$

1.6 Swapping non-adjacent list elements

If x and y are two not necessarily adjacent elements that are “in the wrong order”, swapping them always strictly decreases the swap distance.

```
lemma swap-dist-relation-swap-less:
  assumes linorder-on A R finite A
  assumes xy: R x y
  assumes distinct: distinct (xs @ x # ys @ y # zs)
  assumes subset: set (xs @ x # ys @ y # zs) = A
  shows swap-dist-relation R (of-ranking (xs @ x # ys @ y # zs)) >
    swap-dist-relation R (of-ranking (xs @ y # ys @ x # zs))
⟨proof⟩
```

```
lemma swap-dist-relation-swap-less':
  assumes xy: R (ys ! i) (ys ! j)  $\longleftrightarrow$  i < j
  assumes R: finite-linorder-on A R
  assumes distinct: distinct ys set ys = A
  assumes ij: i < length ys j < length ys i  $\neq$  j
  shows swap-dist-relation R (of-ranking ys) >
    swap-dist-relation R (of-ranking (ys[i := ys ! j, j := ys ! i]))
⟨proof⟩
```

The following formulation for lists is probably the nicest one.

```
lemma swap-dist-swap-less:
  assumes xy: of-ranking xs (ys ! i) (ys ! j)  $\longleftrightarrow$  i < j
  assumes distinct: distinct xs distinct ys set xs = set ys
  assumes ij: i < length ys j < length ys i  $\neq$  j
  shows swap-dist xs ys > swap-dist xs (ys[i := ys ! j, j := ys ! i])
⟨proof⟩
```

1.7 Swap distance as minimal number of adjacent swaps to make two lists equal

The swap distance between the original list and the list obtained after swapping adjacent elements n times is at most n .

```
lemma swap-dist-swap-adjs-list:
  assumes distinct xs
  shows swap-dist xs (swap-adjs-list is xs)  $\leq$  length is
⟨proof⟩
```

Phrased in another way, any sequence of adjacent swaps that makes two lists the same must have a length at least as big as the swap distance of the two lists.

```
theorem swap-dist-minimal:
  assumes distinct xs
  assumes  $\forall i \in \text{set is}. \text{Suc } i < \text{length } xs$ 
```

```

assumes swap-adjs-list is  $xs = ys$ 
shows length is  $\geq$  swap-dist  $xs$   $ys$ 
⟨proof⟩

```

Next, we will show that this lower bound is sharp, i.e. there exists a sequence of swaps that makes the two lists the same whose length is exactly the swap distance.

To this end, we derive an algorithm to compute a sequence of swaps whose effect is equivalent to the permutation $[0, 1, \dots, n - 1] \mapsto [i_0, i_1, \dots, i_{n-1}]$.

We first define the following function, which returns a list of swaps that pulls the i -th element of a list to the front, i.e. it corresponds to the permutation $[0, 1, \dots, n - 1] \mapsto [i, 0, 1, \dots, i - 1, i + 1, \dots, n - 1]$.

```

definition pull-to-front-swaps :: nat  $\Rightarrow$  nat list where
  pull-to-front-swaps  $i = rev [0..<i]$ 

```

```

lemma length-pull-to-front-swaps [simp]: length (pull-to-front-swaps  $i$ ) =  $i$ 
  ⟨proof⟩

```

```

lemma set-pull-to-front-swaps [simp]: set (pull-to-front-swaps  $i$ ) =  $\{0..<i\}$ 
  ⟨proof⟩

```

```

lemma pull-to-front-swaps-0 [simp]: pull-to-front-swaps 0 = []
  and pull-to-front-swaps-Suc: pull-to-front-swaps (Suc  $i$ ) =  $i \#$  pull-to-front-swaps  $i$ 
  ⟨proof⟩

```

```

lemma swap-adjs-list-pull-to-front:
  assumes  $i < length xs$ 
  shows swap-adjs-list (pull-to-front-swaps  $i$ )  $xs = (xs ! i) \# take i xs @ drop (Suc i) xs$ 
  ⟨proof⟩

```

We now simply perform the “pull to front” operation so that the first element is the desired one. We then do the same thing again for the remaining $n - 1$ indices (shifted accordingly) etc. until we reach the end of the index list.

This corresponds to a variant of selection sort that only uses adjacent swaps, or it can also be seen as a kind of reversal of insertion sort.

```

fun swaps-of-perm :: nat list  $\Rightarrow$  nat list where
  swaps-of-perm [] = []
  | swaps-of-perm ( $i \# is$ ) =
    pull-to-front-swaps  $i$  @ map Suc (swaps-of-perm (map (λj. if  $j \geq i$  then  $j - 1$  else  $j$ ) is))

```

```

lemma set-swaps-of-perm-subset: set (swaps-of-perm  $is$ )  $\subseteq (\bigcup_{i \in set is} \{0..<i\})$ 
  ⟨proof⟩

```

```

lemma swap-adjs-list-swaps-of-perm-aux:
  fixes  $i :: nat$ 
  assumes mset ( $i \# is$ ) = mset-set  $\{0..<n\}$ 
  shows mset (map (λj. if  $i \leq j$  then  $j - 1$  else  $j$ )  $is$ ) = mset-set  $\{0..<n - 1\}$ 
  ⟨proof⟩

```

The following result shows that the list of swaps returned by *swaps-of-perm* indeed have the desired effect.

```
lemma swap-adjs-list-swaps-of-perm:
  assumes mset is = mset-set {0.. xs}
  shows swap-adjs-list (swaps-of-perm is) xs = map (λi. xs ! i) is
  ⟨proof⟩
```

The number of swaps returned by *swaps-of-perm* is exactly the number of inversions in the input list (i.e. of the index permutation described by it).

```
lemma length-swaps-of-perm:
  assumes mset is = mset-set {0.. is}
  shows length (swaps-of-perm is) = inversion-number is
  ⟨proof⟩
```

Finally, we use the above to give a list of swap operations that map one list to another. The number of swap operations produced by this is exactly the swap distance of the two lists.

```
definition swaps-of-perm' :: 'a list ⇒ 'a list ⇒ nat list where
  swaps-of-perm' xs ys = swaps-of-perm (map (index xs) ys)
```

```
theorem swaps-of-perm':
  assumes distinct xs distinct ys set xs = set ys
  shows ∀ i ∈ set (swaps-of-perm' xs ys). Suc i < length xs
    swap-adjs-list (swaps-of-perm' xs ys) xs = ys
    length (swaps-of-perm' xs ys) = swap-dist xs ys
  ⟨proof⟩
```

Finally, we can derive the alternative characterisation of the swap distance.

```
lemma swap-dist-altdef:
  assumes distinct xs distinct ys set xs = set ys
  shows swap-dist xs ys = (INF is ∈ {is. swap-adjs-list is xs = ys}. length is)
  ⟨proof⟩
```

end

References

- [1] A. Belov and J. Marques-Silva. Muser2: An efficient MUS extractor. *J. Satisf. Boolean Model. Comput.*, 8(3/4):123–128, 2012.
- [2] A. Biere, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Report Series B*, pages 10–13. University of Helsinki, 2021.

- [3] P. Lammich. The GRAT tool chain – efficient (UN)SAT certificate checking with formal correctness guarantees. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 457–463. Springer, 2017.
- [4] N. Wetzler, M. Heule, and W. A. H. Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014, Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.