The Inversions of a List

Manuel Eberl
June 11, 2019

Abstract

This entry defines the set of \textit{inversions} of a list, i.e. the pairs of indices that violate sortedness. It also proves the correctness of the well-known $O(n \log n)$ divide-and-conquer algorithm to compute the number of inversions.

Contents

1 The Inversions of a List 1

1.1 Definition of inversions .............................. 1
1.2 Counting inversions ................................ 3
1.3 Stability of inversions between lists under permutations 4
1.4 Inversions between sorted lists ....................... 4
1.5 Merge sort ........................................... 5
1.6 Merge sort with inversion counting .................. 7

1 The Inversions of a List

theory List-Inversions
  imports Main HOL-Library.Permutations
begin

1.1 Definition of inversions

context preorder
begin

We define inversions as pair of indices w.r.t. a preorder.

inductive-set inversions :: 'a list ⇒ (nat × nat) set for xs :: 'a list where
  $i < j \implies j < \text{length } xs \implies \text{less } (xs ! j) (xs ! i) \implies (i, j) \in \text{inversions } xs$

lemma inversions-subset: inversions xs ⊆ Sigma {..<\text{length } xs} (\lambda i. \{i<..<\text{length } xs\})
  \langle proof \rangle
**Lemma** finite-inversions [intro]: finite (inversions xs)

(\textit{proof})

**Lemma** inversions-altdef: inversions \(xs = \{(i, j). i < j \land j < \text{length } xs \land \text{less } (xs \! j) (xs \! i)\}\)

(\textit{proof})

**Lemma** inversions-code:

\[
\text{inversions } xs = \\
\text{Sigma } \{..<\text{length } xs\} (\lambda i. \text{Set.filter } (\lambda j. \text{less } (xs \! j) (xs \! i)) \{i..<\text{length } xs\})
\]

(\textit{proof})

**Lemmas** (in −) [code] = inversions-code

**Lemma** inversions-trivial [simp]: length \(xs \leq \text{Suc } 0 \implies \text{inversions } xs = \{\}

(\textit{proof})

**Lemma** inversions-imp-less:

\(z \in \text{inversions } xs \implies \text{fst } z < \text{snd } z\)

\(z \in \text{inversions } xs \implies \text{snd } z < \text{length } xs\)

(\textit{proof})

**Lemma** inversions-Nil [simp]: \(\text{inversions } [] = \{\}\)

(\textit{proof})

**Lemma** inversions-Cons:

\[
\text{inversions } (x \# \text{xs}) = \\
(\lambda j. (0, j + 1)) \cdot \{j \in \{..<\text{length } xs\}. \text{less } (xs \! j) x\} \cup \\
\text{map-prod } \text{Suc } \text{Suc } \cdot \text{inversions } xs (\textit{is - = ?rhs})
\]

(\textit{proof})

The following function returns the inversions between two lists, i.e. all pairs of an element in the first list with an element in the second list such that the former is greater than the latter.

**Definition** inversions-between :: \(\text{'a list } \Rightarrow \text{'a list } \Rightarrow (\text{nat } \times \text{ nat}) \text{ set}\)

\(\text{where} \)

\(\text{inversions-between } xs \ ys = \)

\(\{(i, j) \in \{..<\text{length } xs\} \times \{..<\text{length } ys\}. \text{less } (ys \! j) (xs \! i)\}\)

**Lemma** finite-inversions-between [intro]: finite (inversions-between \(xs\ \ ys\))

(\textit{proof})

**Lemma** inversions-between-nil [simp]:

\(\text{inversions-between } [] \ ys = \{\}\)

\(\text{inversions-between } xs \ [] = \{\}\)

(\textit{proof})

We can now show the following equality for the inversions of the concatenation of two lists:

2
**proposition** inversions-append:

fixes \(xs \ ys\)
defines \(m \equiv \text{length } xs\) and \(n \equiv \text{length } ys\)
shows inversions \((xs @ ys)\) =
    inversions \(xs \cup \text{map-prod } ((+) \ m) \ ((+) \ m) \ \text{'} \text{ inversions } ys \cup \
    \text{map-prod id } ((+) \ m) \ \text{'} \text{ inversions-between } xs \ ys
\hspace{1cm} \text{(is - \equiv \ yrhs)}
\langle \text{proof} \rangle

### 1.2 Counting inversions

We now define versions of `inversions` and `inversions-between` that only return the number of inversions.

**definition** inversion-number :: 'a list ⇒ nat where
    inversion-number \(xs\) = card (inversions \(xs\))

definition inversion-number-between 
    where
    inversion-number-between \(xs \ ys\) = card (inversions-between \(xs \ ys\))

**lemma** inversions-between-code:
    inversions-between \(xs \ ys\) =
    \(\text{Set.filter } (\lambda (i,j). \text{ less } (ys ! j) (xs ! i)) (\{..<\text{length } xs\} \times \{..<\text{length } ys\})\)
\langle \text{proof} \rangle

lemmas (in −) [code] = inversions-between-code

**lemma** inversion-number-nil [simp]: inversion-number [] = 0
\langle \text{proof} \rangle

**lemma** inversion-number-trivial [simp]: length \(xs\) ≤ Suc 0 ⇒ inversion-number \(xs\) = 0
\langle \text{proof} \rangle

**lemma** inversion-number-between-nil [simp]:
    inversion-number-between [] \(ys\) = 0
    inversion-number-between \(xs\) [] = 0
\langle \text{proof} \rangle

We again get the following nice equation for the number of inversions of a concatenation:

**proposition** inversion-number-append:
    inversion-number \((xs @ ys)\) =
    inversion-number \(xs\) + inversion-number \(ys\) + inversion-number-between \(xs \ ys\)
\langle \text{proof} \rangle
1.3 Stability of inversions between lists under permutations

A crucial fact for counting list inversions with merge sort is that the number of inversions between two lists does not change when the lists are permuted. This is true because the set of inversions commutes with the act of permuting the list:

**lemma inversions-between-permute1:**
\[\text{assumes } \pi \text{ permutes } \{..<\text{length } xs\}\]
\[\text{shows } \text{inversions-between } (\text{permute-list } \pi ) \text{ } xs = \text{map-prod } (\text{inv } \pi ) \text{ } \text{id} \text{ } \text{'} \text{ } \text{inversions-between } xxs \text{ } ys\]
\[\text{⟨ proof ⟩}\]

**lemma inversions-between-permute2:**
\[\text{assumes } \pi \text{ permutes } \{..<\text{length } ys\}\]
\[\text{shows } \text{inversions-between } xxs \text{ } (\text{permute-list } \pi ) \text{ } ys = \text{map-prod } \text{id } \text{'} \text{ } \text{inv } \pi \text{ } \text{'} \text{ } \text{inversions-between } xxs \text{ } ys\]
\[\text{⟨ proof ⟩}\]

**proposition inversions-between-permute:**
\[\text{assumes } \pi 1 \text{ permutes } \{..<\text{length } zs\} \text{ and } \pi 2 \text{ permutes } \{..<\text{length } ys\}\]
\[\text{shows } \text{inversions-between } (\text{permute-list } \pi 1 ) \text{ } xxs \text{ } (\text{permute-list } \pi 2 ) \text{ } ys = \text{map-prod } (\text{inv } \pi 1 ) \text{ } (\text{inv } \pi 2 ) \text{ } \text{'} \text{ } \text{inversions-between } xxs \text{ } ys\]
\[\text{⟨ proof ⟩}\]

**corollary inversion-number-between-permute:**
\[\text{assumes } \pi 1 \text{ permutes } \{..<\text{length } zs\} \text{ and } \pi 2 \text{ permutes } \{..<\text{length } ys\}\]
\[\text{shows } \text{inversion-number-between } (\text{permute-list } \pi 1 ) \text{ } xxs \text{ } (\text{permute-list } \pi 2 ) \text{ } ys = \text{inversion-number-between } xxs \text{ } ys\]
\[\text{⟨ proof ⟩}\]

The following form of the above theorem is nicer to apply since it has the form of a congruence rule.

**corollary inversion-number-between-cong-mset:**
\[\text{assumes } mset xxs = mset xxs' \text{ and } mset yys = mset yys'}\]
\[\text{shows } \text{inversion-number-between } xxs \text{ } yys = \text{inversion-number-between } xxs' \text{ } yys'}\]
\[\text{⟨ proof ⟩}\]

1.4 Inversions between sorted lists

Another fact that is crucial to the efficient computation of the inversion number is this: If we have two sorted lists, we can reduce computing the inversions by inspecting the first elements and deleting one of them.

**lemma inversions-between-Cons-Cons:**
\[\text{assumes sorted-wrt less-eq } (x \neq xxs) \text{ and sorted-wrt less-eq } (y \neq yys)\]
\[\text{shows } \text{inversions-between } (x \neq xxs) \text{ } (y \neq yys) = \text{if } \neg \text{less } y \text{ } x \text{ then} \text{map-prod } \text{Suc } \text{id } \text{'} \text{ } \text{inversions-between } xxs \text{ } yys\]
\[\text{⟨ proof ⟩}\]

4
else
{..<\text{length} (x\#xs)} \times \{O\} \cup
\text{map-prod id Suc \ ' inversions-between} (x \# xs) y)

(proof)

This leads to the following analogous equation for counting the inversions
between two sorted lists. Note that a single step of this only takes constant
time (assuming we pre-computed the lengths of the lists) so that the entire
function runs in linear time.

lemma inversion-number-between-Cons-Cons:
assumes sorted-wrt less-eq (x \# xs) and sorted-wrt less-eq (y \# ys)
shows inversion-number-between (x \# xs) (y \# ys) =
\begin{align*}
\text{if} \ \neg\ \text{less y x then} \\
\text{inversion-number-between} xs (y \# ys) \\
\text{else} \\
\text{inversion-number-between} (x \# xs) ys + \text{length} (x \# xs)
\end{align*}

(proof)

We now define a function to compute the inversion number between two
lists that are assumed to be sorted using the equalities we just derived.

fun inversion-number-between-sorted :: \textquoteleft a list \Rightarrow \textquoteleft a list \Rightarrow\textquoteleft nat where
inversion-number-between-sorted [] ys = 0
| inversion-number-between-sorted xs [] = 0
| inversion-number-between-sorted (x \# xs) (y \# ys) =
\begin{align*}
\text{if} \ \neg\ \text{less y x then} \\
\text{inversion-number-between-sorted} xs (y \# ys) \\
\text{else} \\
\text{inversion-number-between-sorted} (x \# xs) ys + \text{length} (x \# xs)
\end{align*}

theorem inversion-number-between-sorted-correct:
sorted-wrt less-eq xs \Rightarrow sorted-wrt less-eq ys \Rightarrow
inversion-number-between-sorted xs ys = inversion-number-between xs ys
(proof)

end

1.5 Merge sort

For convenience, we first define a simple merge sort that does not compute
the inversions. At this point, we need to start assuming a linear ordering
since the merging function does not work otherwise.

context linorder
begin

definition split-list
where split-list xs = (let n = \text{length} xs div 2 in (\text{take} n xs, \text{drop} n xs))
fun merge-lists :: 'a list ⇒ 'a list ⇒ 'a list where
merge-lists [] ys = ys
| merge-lists xs [] = xs
| merge-lists (x # xs) (y # ys) =
  (if less-eq x y then x # merge-lists xs (y # ys) else y # merge-lists (x # xs) ys)

lemma set-merge-lists [simp]: set (merge-lists xs ys) = set xs ∪ set ys
 ⟨proof⟩
lemma mset-merge-lists [simp]: mset (merge-lists xs ys) = mset xs + mset ys
 ⟨proof⟩
lemma sorted-merge-lists [simp, intro]:
  sorted xs ⇒ sorted ys ⇒ sorted (merge-lists xs ys)
 ⟨proof⟩

fun merge-sort :: 'a list ⇒ 'a list where
merge-sort xs =
  (if length xs ≤ 1 then
   xs
  else
   merge-lists (merge-sort (take (length xs div 2) xs))
   (merge-sort (drop (length xs div 2) xs)))
lemmas [simp del] = merge-sort.simps
lemma merge-sort-trivial [simp]: length xs ≤ Suc 0 ⇒ merge-sort xs = xs
 ⟨proof⟩
theorem mset-merge-sort [simp]: mset (merge-sort xs) = mset xs
 ⟨proof⟩
corollary set-merge-sort [simp]: set (merge-sort xs) = set xs
 ⟨proof⟩
theorem sorted-merge-sort [simp, intro]: sorted (merge-sort xs)
 ⟨proof⟩
lemma inversion-number-between-code:
inversion-number-between xs ys = inversion-number-between-sorted (sort xs) (sort ys)
 ⟨proof⟩
lemmas (in -) [code-unfold] = inversion-number-between-code
1.6 Merge sort with inversion counting

Finally, we can put together all the components and define a variant of merge sort that counts the number of inversions in the original list:

```plaintext
function sort-and-count-inversions :: 'a list ⇒ 'a list × nat where
sort-and-count-inversions xs =
  (if length xs ≤ 1 then
   (xs, 0)
    else
    let (xs1, xs2) = split-list xs;
     (xs1', m) = sort-and-count-inversions xs1;
     (xs2', n) = sort-and-count-inversions xs2
    in
     (merge-lists xs1' xs2', m + n + inversion-number-between-sorted xs1' xs2'))
```

The projection of this function to the first component is simply the standard merge sort algorithm that we defined and proved correct before.

```plaintext
theorem fst-sort-and-count-inversions [simp]:
  fst (sort-and-count-inversions xs) = merge-sort xs
⟨proof⟩
```

The projection to the second component is the inversion number.

```plaintext
theorem snd-sort-and-count-inversions [simp]:
  snd (sort-and-count-inversions xs) = inversion-number xs
⟨proof⟩
```

```plaintext
lemmas (in −) [code-unfold] = snd-sort-and-count-inversions [symmetric]
```

References