The Fisher–Yates shuffle

Manuel Eberl

August 16, 2018

Abstract

This work defines and proves the correctness of the Fisher–Yates shuffle [1, 2, 3] for shuffling – i.e. producing a random permutation – of a list. The algorithm proceeds by traversing the list and in each step swapping the current element with a random element from the remaining list.

Contents

1 Fisher–Yates shuffle
   1.1 Swapping elements in a list .................................. 2
   1.2 Random Permutations ............................................ 3
   1.3 Shuffling Lists ...................................................... 4
   1.4 Forward Fisher-Yates Shuffle ................................... 5
   1.5 Backwards Fisher-Yates Shuffle ................................. 7
   1.6 Code generation test .............................................. 9
1 Fisher–Yates shuffle

theory Fisher-Yates
imports HOL-Probability.Probability
begin

lemma integral-pmf-of-multiset:
A ≠ (#) ⇒ (∫ x. (f x :: real) ∂ measure-pmf (pmf-of-multiset A)) =
(∑ x∈set-mset A. of-nat (count A x) * f x) / of-nat (size A)
by (subst integral-measure-pmf[where A = set-mset A])
(simp-all add: sum-divide-distrib mult_ac)

lemma pmf-bind-pmf-of-multiset:
A ≠ (#) ⇒ pmf (pmf-of-multiset A >>= f) y =
(∑ x∈set-mset A. real (count A x) * pmf (f x) y) / real (size A)
by (simp add: pmf-bind integral-pmf-of-multiset)

lemma pmf-map-inj-inv:
assumes inj-on f (set-pmf p)
assumes ∀ x. f' (f x) = x
shows pmf (map-pmf f p) x = (if x ∈ range f then pmf p (f' x) else 0)
proof (cases x ∈ f' set-pmf p)
  case True
  from this obtain y where y: y ∈ set-pmf p x = f y by blast
  with assms(1) have pmf (map-pmf f p) x = pmf p y
    by (simp add: pmf-map-inj)
  also from y assms(2) have y = f' x by simp
  finally show ?thesis using y by simp
next
  case False
  hence x ∉ set-pmf (map-pmf f p) by simp
  hence pmf (map-pmf f p) x = 0 by (simp add: set-pmf-eq)
  also from False have 0 = (if x ∈ range f then pmf p (f' x) else 0)
    by (auto simp: assms(2) set-pmf-eq)
  finally show ?thesis .
qed

1.1 Swapping elements in a list

definition swap where swap xs i j = xs[i := xs!j, j := xs ! i]

lemma length-swap [simp]: length (swap xs i j) = length xs
by (simp add: swap-def)

lemma swap-eq-Nil-iff [simp]: swap xs i j = []⇔xs = []
by (simp add: swap-def)

lemma nth-swap: i < length xs ⇒ j < length xs ⇒
\begin{align*}
\text{swap } xs \ i \ j \ k &= (if \ k = i \ then \ xs \ ! \ j \ else \ if \ k = j \ then \ xs \ ! \ i \ else \ xs \ ! \ k) \\
&\text{by (auto simp: swap-def nth-list-update)}
\end{align*}

\textbf{lemma map-swap:} \(i < \text{length } xs \implies j < \text{length } xs \implies \text{map } f (\text{swap } xs \ i \ j) = \text{swap} (\text{map } f \ xs) \ i \ j\)
\begin{itemize}
\item \text{by (simp add: swap-def map-update map-nth)}
\end{itemize}

\textbf{lemma swap-swap:} \(i < \text{length } xs \implies j < \text{length } xs \implies \text{swap} (\text{swap } xs \ i \ j) \ j \ i = xs\)
\begin{itemize}
\item \text{by (intro nth-equality1) (auto simp: nth-swap nth-list-update)}
\end{itemize}

\textbf{lemma mset-swap:} \(i < \text{length } xs \implies j < \text{length } xs \implies \text{mset} (\text{swap } xs \ i \ j) = \text{mset} \ xs\)
\begin{itemize}
\item \text{by (simp add: mset-update swap-def nth-list-update)}
\end{itemize}

\textbf{lemma hd-swap-0:} \(i < \text{length } xs \implies \text{hd} (\text{swap } xs \ 0 \ i) = \text{xs} ! \ i\)
\begin{itemize}
\item \text{unfolding swap-def by (subst hd-conv-nth) (subst nth-list-update | force)}+
\end{itemize}

\subsection{1.2 Random Permutations}

First, we prove the intuitively obvious fact that choosing a random permutation of a multiset can be done by first randomly choosing the first element and then randomly choosing the rest of the list.

\textbf{lemma pmf-of-set-permutations-of-multiset-nonempty:}
\begin{itemize}
\item \text{assumes} \((A :: 'a multiset) \neq \{\#\})\)
\item \text{shows} \(\text{pmf} (\text{permutations-of-multiset } A) = \)
\item \text{do} \(\{x \leftarrow \text{pmf-of-multiset } A; \)
\item \(\xs \leftarrow \text{pmf-of-set} (\text{permutations-of-multiset} (A - \{\# x \#\})); \)
\item \text{return-pmf} \(\langle x \# \xs \rangle \}\)
\item \(\text{is } ?\text{lhs} = ?\text{rhs}\)
\end{itemize}
\textbf{proof (rule pmf-eqI)}
\begin{itemize}
\item \text{fix } \xs :: 'a list
\item \text{show } \text{pmf } ?\text{lhs } \xs = \text{pmf } ?\text{rhs } \xs
\item \text{proof (cases } \xs \notin \text{permutations-of-multiset } A) \)
\item \text{case } \text{False}
\item \text{with } \text{assms have } \xs \notin \text{set-pmf } ?\text{lhs} \text{ by simp}
\item \text{moreover from } \text{assms False have } \xs \notin \text{set-pmf } ?\text{rhs}
\item \text{by (auto simp: permutations-of-multiset-Cons-iff)}
\item \text{ultimately show } ?\text{thesis} \text{ by (simp add: set-pmf-eq)}
\item \text{next}
\item \text{case } \text{True}
\item \text{with } \text{assms have } \text{nonempty; } \xs \neq \emptyset \text{ by (auto dest: permutations-of-multisetD)}
\item \text{hence } \text{range-Cons: } \xs \in \text{range } (\{\#\} \ x) \iff \text{hd } \xs = x \text{ for } x
\item \text{by (cases } \xs \text{) auto}
\item \text{from } \text{True } \text{nonempty}
\item \text{have } \text{hd-tl: } \text{hd } \xs \in \# A \land \text{tl } \xs \in \text{permutations-of-multiset} (A - \{\# \text{hd } \xs \#\})
\item \text{by (cases } \xs \text{) (auto simp: permutations-of-multiset-Cons-iff)}
\item \text{from } \text{assms have } ?\text{rhs } \xs =
\end{itemize}
\[
\sum_{x \in \text{set-mset } A} \text{real} \left( \text{count } A \cdot x \right) \cdot \text{pmf} \left( \text{map-pmf} \left( \# x \right) \right) \cdot \text{xs} / \text{real} \left( \text{size } A \right)
\]

unfolding map-pmf-def [symmetric] by (simp add: pmf-bind-pmf-of-multiset)
also have \( \text{S} = \)
\[
\sum_{x \in \text{set-mset } A} \text{if } x = \text{hd } \text{xs} \text{ then } \text{real} \left( \text{count } A \cdot (\text{hd } \text{xs}) \right) / \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A - \{\# (\text{hd } \text{xs})\} \right) \right) \text{ else } 0 \right)
\]
using range-Cons hd-tl
by (intro sum.cong refl, subst pmf-map-inj-inv[where \( f' = \text{tl} \)])
also have \( \ldots = \text{real} \left( \text{size } A \right) / \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A - \{\# (\text{hd } \text{xs})\} \right) \right) \right) \)
using hd-tl by (simp add: sum.delta)
also from hd-tl have \( \ldots \) \( = \text{real} \left( \text{size } A \right) / \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A \right) \right) \right) \)
xs
using assms True by simp
finally show \( \text{?thesis} \) ..
qed

1.3 Shuffling Lists

We define shuffling of a list as choosing from the set of all lists that correspond to the same multiset uniformly at random.

definition shuffle :: 'a list \Rightarrow 'a list pmf
where
shuffle \( \text{xs} = \text{pmf-of-set} \left( \text{permutations-of-multiset} \left( \text{mset } \text{xs} \right) \right) \)

lemma shuffle-empty [simp]: shuffle \( \text{[]} = \text{return-pmf} \text{[]} \)
by (simp add: shuffle-def pmf-of-set-singleton)

lemma shuffle-singleton [simp]: shuffle \([x] = \text{return-pmf} \text{[x]} \)
by (simp add: shuffle-def pmf-of-set-singleton)

The crucial ingredient of the Fisher–Yates shuffle is the following lemma, which decomposes a shuffle into swapping the first element of the list with a random element of the remaining list and shuffling the new remaining list.

With a random-access implementation of a list – such as an array – all of the required operations are cheap and the resulting algorithm runs in linear time.

lemma shuffle-fisher-gates-step:
assumes \( \text{xs-nonempty} \) [simp]: \( \text{xs} \neq \text{[]} \)
shows \( \text{shuffle } \text{xs} = \)
do \{ \( i \leftarrow \text{pmf-of-set} \text{[..<length } \text{xs}] \);
let \( \text{ys} = \text{swap } \text{xs} 0 i \);
\( \text{zs} \leftarrow \text{shuffle } (\text{tl } \text{ys}) \);
proof –

have shuffle xs = do {x ← pmf-of-multiset (mset xs);
                   xs ← pmf-of-set (permutations-of-multiset (mset xs - {#x#}));
                   return-pmf (x#xs)}

by (simp add: pmf-of-set-permutations-of-multiset-nonempty)
also have pmf-of-multiset (mset xs) =
      pmf-of-multiset (image-mset (!) xs) (mset (upt 0 (length xs))))
by (subst mset-map [symmetric]) (simp add: map-nth)
also have ... = map-pmf (!) xs (pmf-of-set {..<length xs})
by (subst map-pmf-of-set) (auto simp add: map-pmf-of-set atLeast0LessThan
lessThan-empty-iff)
also have do {x ← map-pmf (!) xs (pmf-of-set {..<length xs});
             ys ← pmf-of-set (permutations-of-multiset (mset xs - {#x#}));
             return-pmf (x # ys)} =
      do {i ← pmf-of-set {..<length xs};
           ys ← pmf-of-set (permutations-of-multiset (mset xs - {#xs !i#}));
           return-pmf (xs !i # ys)}

by (simp add: map-pmf-def bind-assoc-pmf bind-return-pmf)
also have ... = do {i ← pmf-of-set {..<length xs};
                   let ys = swap xs 0 i;
                   zs ← shuffle (tl (swap xs 0 i));
                   return-pmf (hd ys # zs)} unfolding Let-def shuffle-def
by (intro bind-pmf-cong refl, subst (asm) set-pmf-of-set)
(auto simp: lessThan-empty-iff mset-tl mset-swap hd-swap-0)
finally show ?thesis by (simp add: Let-def)
qed

1.4 Forward Fisher-Yates Shuffle

The actual Fisher–Yates shuffle is now merely a kind of tail-recursive version
of decomposition described above. Note that unlike the traditional Fisher–
Yates shuffle, we shuffle the list from front to back, which is the more natural
way to do it when working with linked lists.

function fisher-yates-aux where

fisher-yates-aux i xs = (if i + 1 ≥ length xs then return-pmf xs else
do { j ← pmf-of-set {i..<length xs};
     fisher-yates-aux (i + 1) (swap xs i j)})
by auto
termination by (relation Wellfounded.measure (λ(i,xs). length xs − i)) simp-all

declare fisher-yates-aux.simps [simp del]
lemma fisher-yates-aux-correct:

fisher-yates-aux i xs = map-pmf (λys. take i xs @ ys) (shuffle (drop i xs))

proof (induction i xs rule: fisher-yates-aux.induct)

case (1 i xs)

show ?case

proof (cases i + 1 ≥ length xs)

case True

show ?thesis

qed (simp-all add: fisher-yates-aux.simps)

next

case False

from False have xs-nonempty [simp]: xs ≠ [] by auto

have fisher-yates-aux i xs =

  pmf-of-set {..<length xs} ⊢= (λj. fisher-yates-aux (i+1) (swap xs i j))

  using False by (subst fisher-yates-aux.simps) simp

  also have {..<length xs} = (λj. j + i) ▷ {..<length xs - i})

  using False by (simp add: lessThan-atLeast0)

  also from False have pmf-of-set ... = map-pmf (λj. j + i) (pmf-of-set {..<length xs - i})

  by (subst map-pmf-of-set-inj) (simp-all add: lessThan-empty-iff)

  also from False have length xs - i = length (drop i xs) by simp

  also have map-pmf (λj. j + i) (pmf-of-set {..<length (drop i xs)}) \[\frac{}{}\]

  (λj. fisher-yates-aux (i + 1) (swap xs i j)) = pmf-of-set {..<length (drop i xs)} \[\frac{}{}\]

  (λj. fisher-yates-aux (i + 1) (swap xs i (j+i)))

  by (simp add: map-pmf-def bind-return-pmf bind-assoc-pmf)

  also have ... = do {j ← pmf-of-set {..<length (drop i xs)}; let ys = swap (drop i xs) 0 j; zs ← shuffle (tl ys); return-pmf (take i xs @ hd ys # zs}) (is - = bind-pmf - ?T)

proof (intro bind-pmf-cong refl)

  fix j assume j ∈ set-pmf (pmf-of-set {..<length (drop i xs)})

  with False have j: j < length (drop i xs) by (simp-all add: lessThan-empty-iff)

  define ys where ys = swap xs i (j + i)

  have fisher-yates-aux (i + 1) ys = map-pmf ((@() (take (i+1) ys)) (shuffle (drop (i+1) ys))

  using False j unfolding gs-def by (intro 1.IH) simp-all

  also from False have take (i+1) ys = take i ys @ [hd (drop i ys)]
by \((simp\ add:\ ys-def\ take-hd-drop)\)

also have \(drop\ (i+1)\ \text{ys} = tl\ (drop\ i\ \text{ys})\) by \((simp\ add:\ ys-def\ tl-drop\ drop-Suc)\)

also from False \(j\) have \(drop\ i\ \text{ys} = \text{swap}\ (drop\ i\ \text{xs})\ 0\ j\)
by \((simp\ add:\ ys-def\ swap-def\ drop-update-swap\ add-ac)\)

also from False \(j\) have \(take\ i\ \text{ys} = \text{take}\ i\ \text{xs}\)
by \((simp\ add:\ ys-def\ swap-def)\)

finally show fisher-yates-aux \((i+1)\ \text{ys} = ?T\ j\)
by \((simp\ add:\ ys-def\ map-pmf-def\ Let-def\ bind-assoc-pmf\ bind-return-pmf)\)

qed

also from False have ...
= map-pmf \((\lambda zs.\ \text{take}\ i\ \text{xs} \odot zs)\) (shuffle (drop i \text{xs}))
by \((subst\ \text{shuffle-fisher-yates-step}[of\ drop\ i\ \text{xs}])\)
(simp-all add: map-pmf-def Let-def bind-return-pmf bind-assoc-pmf)

finally show ?thesis .

qed

定义 fisher-yates  where
fisher-yates = fisher-yates-aux 0

引理 fisher-yates-correct: fisher-yates \(\text{xs}\) = shuffle \(\text{xs}\)

从 fisher-yates-def

by \((subst\ fisher-yates-aux-correct)\) (simp-all add: map-pmf-def bind-return-pmf ')

1.5 向后 Fisher-Yates Shuffle

我们可以很容易地推导出经典的 Fisher–Yates 洗牌，它从后向前遍历列表，并证明它与向前 Fisher–Yates 洗牌等价。

fun fisher-yates-alt-aux  where
fisher-yates-alt-aux \(i\ \text{xs}\) = (if \(i = 0\) then return-pmf \(\text{xs}\) else
do {j <- pmf-of-set \{..\}\;
fisher-yates-alt-aux \((i-1)\) (swap \(\text{xs} i\ \text{j}\)))

declare fisher-yates-alt-aux.simps [simp del]

引理 fisher-yates-alt-aux-altdef:
\(i < \text{length}\ \text{xs}\ \Rightarrow\ \text{fisher-yates-alt-aux}\ \text{i}\ \text{xs} =\)
map-pmf rev (fisher-yates-alt-aux (length \(\text{xs}\) - \(i + 1\)) (rev \(\text{xs}\))

proof (induction \(i\ \text{xs}\) rule: fisher-yates-alt-aux.induct)

case \((1\ i\ \text{xs})\)

show ?case

proof (cases \(i = 0\))
  case False
  with \(1.\text{prems}\) have map-pmf rev (fisher-yates-alt-aux (length \(\text{xs}\) - \(i + 1\)) (rev \(\text{xs}\))) =

  pmf-of-set \(\text{length}\ \text{xs}\) - \(\text{Suc}\ \text{i}...\text{length}\ \text{xs}\) \(\geq\)
  \((\lambda x.\ \text{fisher-yates-alt-aux}\ \text{(Suc}\ (\text{length}\ \text{xs}\ - \(\text{Suc}\ \text{i})\))\)
  (swap (rev \(\text{xs}\)) (length \(\text{xs}\) - \(\text{Suc}\ \text{i}\) \(\text{x}\)) \(\geq\)
  \((\lambda x.\ \text{return-pmf}\ (\text{rev}\ \text{x}))\))

7
by (subst fisher-yates-aux.simps) (auto simp: map-pmf-def bind-return-pmf bind-assoc-pmf)
also from 1.prems False
have bij: bij_betw (\lambda j. length xs - Suc j) {..i} {length xs - Suc i..<length xs}
  by (intro bij_betwI [where g = \lambda j. length xs - Suc j]) auto
from bij have \(length xs - Suc i..<length xs\) = (\lambda j. length xs - Suc j) 
  {..i}
  by (simp add: bij_betw_def)
also from bij have pmf-of-set {..i} = map-pmf (\lambda j. length xs - Suc j) (pmf-of-set {..i})
  by (subst map-pmf-of-set-inj) (auto simp: bij_betw_def)
also have \(map-pmf (\lambda j. length xs - Suc j) (pmf-of-set {..i}) \cong\)
  (\lambda x. fisher-yates-aux (Suc (length xs - Suc i))
  (swap (rev xs) (length xs - Suc i) x) \cong\ (\lambda x. return-pmf (rev x)))
using 1.prems False
by (auto simp add: map-pmf_def bind-assoc-pmf bind-return-pmf Suc_diff_Suc
  swap_def rev_update rev_nth intro! bind-pmf-cong)
also have \{..i\} \cong (\lambda j. fisher-yates-alt-aux (i - 1) (swap xs i j))
using 1.prems False 1.IH [symmetric] by (auto intro! bind-pmf-cong)
also from 1.prems False have \{..i\} \cong fisher-yates-alt-aux i xs
by (subst fisher-yates-alt-aux-altdef) simp
finally show thesis ..
qed (insert 1.prems, simp-all add: fisher-yates-aux.simps fisher-yates-alt-aux.simps)
qed

definition fisher-yates-alt where
fisher-yates-alt xs = fisher-yates-alt-aux (length xs - 1) xs

lemma fisher-yates-alt-aux-correct:
fisher-yates-alt xs = shuffle xs
proof (cases xs = [])
case True
thus thesis
  by (simp add: fisher-yates-alt-def fisher-yates-alt-aux.simps)
next
case False
thus thesis unfolding fisher-yates-alt-def
  by (subst fisher-yates-alt-aux-altdef)
(qsimp-all add: fisher-yates-alt-aux-correct shuffle-def map-pmf-of-set-inj)
qed
1.6 Code generation test

Isabelle’s code generator allows us to produce executable code both for Fisher-Yates.shuffle and for fisher-yates and fisher-yates-alt. However, this code does not produce a random sample (i.e. a single randomly permuted list) – which is, in fact, the only purpose of the Fisher–Yates algorithm – but the entire probability distribution consisting of $n!$ lists, each with probability $1/n!$.

In the future, it would be nice if Isabelle also had some code generation facility that supports generating sampling code.

```plaintext
value [code] shuffle "abcd"
value [code] fisher-yates "abcd"
value [code] fisher-yates-alt "abcd"
end
```

References

