The Fisher–Yates shuffle

Manuel Eberl

February 23, 2021

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 2
  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 . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1 Fisher–Yates shuffle

theory Fisher-Yates
  imports HOL Probability.Probability
begin

lemma integral-pmf-of-multiset:
  \( A \neq \{\#\} \implies \left( \int x. (f x :: \text{real}) \ \partial \text{measure-pmf} \ (\text{pmf-of-multiset} \ A) \right) = \right( \sum_{x \in \text{set-mset} \ A} \text{of-nat} (\text{count} A x) \times f x \right) / \text{of-nat} (\text{size} A) \)
  by (subst integral-measure-pmf [where \( A = \text{set-mset} A \)])
  (simp-all add: sum-divide-distrib mult-ac)

lemma pmf-bind-pmf-of-multiset:
  \( A \neq \{\#\} \implies \text{pmf} (\text{pmf-of-multiset} A > \text{map-pmf} f) y = \right( \sum_{x \in \text{set-mset} A} \text{of-nat} \ (\text{count} A x) \times \text{pmf} (f x) y \right) / \text{of-nat} (\text{size} A) \)
  by (simp add: pmf-bind integral-pmf-of-multiset)

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

1.1 Swapping elements in a list

definition swap where \( \text{swap} \ x s i j = x s[i := x s j, j := x s ! i] \)

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 = [] \iff xs = []
  by (simp add: swap-def)

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

lemma map-swap: i < length xs ⇒ j < length xs ⇒ map f (swap xs i j) = swap (map f xs) i j
by (simp add: swap-def map-update map-nth)

lemma swap-swap: i < length xs ⇒ j < length xs ⇒ swap (swap xs i j) j i = xs
by (intro nth-equalityI) (auto simp: nth-swap nth-list-update)

lemma mset-swap: i < length xs ⇒ j < length xs ⇒ mset (swap xs i j) = mset xs
by (simp add: mset-update swap-def nth-list-update)

lemma hd-swap-0: i < length xs ⇒ hd (swap xs 0 i) = xs ! i
unfolding swap-def by (subst hd-conv-nth) (subst nth-list-update | force)+

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.

lemma pmf-of-set-permutations-of-multiset-nonempty:
assumes (A :: 'a multiset) ≠ {#}
shows pmf-of-set (permutations-of-multiset A) =
do { x ← pmf-of-multiset A;
   xs ← pmf-of-set (permutations-of-multiset (A − {#x#}));
   return-pmf (x#xs)
} (is ?lhs = ?rhs)
proof (rule pmf-eqI)
fix xs :: 'a list
show pmf ?lhs xs = pmf ?rhs xs
proof (cases xs ∈ permutations-of-multiset A)
  case False
  with assms have xs ∉ set-pmf ?lhs by simp
  moreover from assms False have xs ∉ set-pmf ?rhs
    by (auto simp: permutations-of-multiset-Cons-iff)
  ultimately show ?thesis by (simp add: set-pmf-eq)
next
  case True
  with assms have nonempty: xs ≠ [] by (auto dest: permutations-of-multisetD)
  hence range-Cons: xs ∈ range ((#) x) ⇐⇒ hd xs = x for x
    by (cases xs) auto
  from True nonempty
  have hd-tl: (hd xs ∈ # A ∧ tl xs ∈ permutations-of-multiset (A − {#hd xs#}))
    by (cases xs) (auto simp: permutations-of-multiset-Cons-iff)
  from assms have pmf ?rhs xs =
    (∑ x∈set-mset A. real (count A x) * pmf (map-pmf ((#) x)
(pmf-of-set (permutations-of-multiset (A - \{#x#\)))) xs / real (size A) (is - = ?S / -)

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

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

lemma shuffle-singleton [simp]: shuffle [x] = return-pmf [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 xs-nonempty [simp]: xs \neq []
shows shuffle xs = 
do { i \leftarrow pmf-of-set \{..<length xs\}; 
let ys = swap xs 0 i; 
zs \leftarrow shuffle (tl ys); 
return-pmf (hd ys # zs)
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)
} unfolding shuffle-def
  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
doi {j ← pmf-of-set {..<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 (\ys. \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 \geq \text{length} \ xs)
    case True
    show \ ?thesis
    proof (cases \ i \geq \text{length} \ xs)
      case False
      with True have \ \text{length} \ xs = Suc \ i \ and \ i = \text{length} \ xs - 1 \ by \ simp-all
      hence \ xs \neq [] \ by \ auto
      hence \ xs = \text{butlast} \ xs \@ [\text{last} \ xs] \ by \ (rule append-butlast-last-id [\text{symmetric}])
      also have \ \text{butlast} \ xs = \take i xs \ by \ (simp add: \text{butlast-conv-take} i)
      finally have \ eq: \take i xs \@ [\text{last} \ xs] = \xs \ ..
      moreover have \ \text{xs} = \take i xs \@ \text{drop} i xs \ by \ simp
      ultimately have \ \take i xs \@ [\text{last} \ xs] = \take i xs \@ \text{drop} i xs \ by \ (rule \ trans)
      hence \ \text{drop} i xs = [\text{last} \ xs] \ by \ (subst (asm) \text{same-append-eq}) \ simp-all
      with True show \ ?thesis \ by \ (simp add: eq fisher-yates-aux.simps)
    qed \ (simp-all \ add: fisher-yates-aux.simps)
  next
    case True
    from False have \ \text{xs-nonempty} [\text{simp}]; \ xs \neq [] \ by \ auto
    have \ fisher-yates-aux \ i \ xs =
      \map-pmf-of-set \ \{\i. < \text{length} \ xs\} \ \\Rightarrow (\lambda j. \text{fisher-yates-aux} \ (i+1) \ (\swap \ i \ j))
    \ using False \ by \ (subt fisher-yates-aux.simps) \ simp
    also have \ \{\i. < \text{length} \ xs\} = ((\lambda j. \ j + i) \cdot \{\i. < \text{length} \ xs - i\})
    \ using False \ by \ (simp add: \text{lessThan-atLeast0})
    also from False have \ \text{pmf-of-set} \ .. = \map-pmf (\lambda j. \ j + i) \ (\text{pmf-of-set} \ \{\i. < \text{length} \ xs - i\})
    \ by \ (simp-all \ add: \text{lessThan-empty-iff})
    also from False \ have \ \text{length} \ xs - i = \text{length} \ drop \ i \ xs \ by \ simp
    also have \ \text{pmf-of-set} \ (\lambda j. \ j + i) \ (\text{pmf-of-set} \ \{\i. < \text{length} \ drop \ i \ xs\}) \ \Rightarrow
      (\lambda j. \text{fisher-yates-aux} \ (i + 1) \ (\swap \ i \ j)) =
      \map-pmf-of-set \ (\i. < \text{length} \ drop \ i \ xs) \ \Rightarrow (\lambda j. \text{fisher-yates-aux} \ (i + 1) \ (\swap \ i \ j) \ (\\text{drop} \ i \ xs \@ \text{hd} \ ys \# \ zs) \ (\text{is} = \text{bind-pmf} - \ ?T)
    proof \ (intro \text{bind-pmf-cong} refl)
      fix \ j \ assume \ j \in \text{set-pmf} \ (\text{pmf-of-set} \ \{\i. < \text{length} \ drop \ i \ xs\})
      with False have \ j < \text{length} \ drop \ i \ xs \ by \ (simp-all \ add: \text{lessThan-empty-iff})
      define \ ys \ where \ ys = \swap \ drop \ i \ xs \ i \ j \;
      z\leftarrow \shuffle (\text{tl} \ ys); \;
      return-pmf \ (\take i \ xs \@ \text{hd} \ ys \# \ zs) \ (\text{is} = \text{bind-pmf} - \ ?T)
    proof \ (intro \text{bind-pmf-cong} refl)
      fix \ j \ assume \ j \in \text{set-pmf} \ (\text{pmf-of-set} \ \{\i. < \text{length} \ drop \ i \ xs\})
      with False have \ j < \text{length} \ drop \ i \ xs \ by \ (simp-all \ add: \text{lessThan-empty-iff})
      define \ ys \ where \ ys = \swap \ drop \ i \ xs \ i \ j \;
      \text{fisher-yates-aux} \ (i + 1) \ ys = \map-pmf \ ((\@) \ (\take \ i \ys)) \ (\shuf\left\{(\i + 1) \ ys\}\right\}
      \ using False \ \text{unfolding} \ \text{ys-def} \ by \ (intro \ 1.IH) \ simp-all
      also from False have \ \text{take} \ (i + 1) \ ys \ = \ \text{take} \ i \ ys \@ \{\text{hd} \ (\text{drop} \ i \ ys)\}
      \ by \ (simp \ add: \text{ys-def} \ \text{take-hd-drop)
also have drop \((i+1)\) \(ys = tl\ (drop\ i\ ys)\) by (simp add: \(ys\text{-def} tl\text{-drop}\ Suc\))
also from \(False\ j\) have drop \(i\) \(ys = swap\ (drop\ i\ xs)\ 0\ j\)
   by (simp add: \(ys\text{-def} swap\text{-def} drop\text{-update-swap}\ add\text{-ac}\))
also from \(False\ j\) have \(take\ i\ ys = take\ i\ xs\)
   by (simp add: \(ys\text{-def} swap\text{-def} add\text{-ac}\))
finally show fisher-yates-aux \((i + 1)\) \(ys = \_T\ j\)
   by (simp add: \(ys\text{-def} map\text{-pmf-def}\ Let\text{-def}\ bind\text{-assoc-pmf}\ bind\text{-return-pmf}\))
qed
also from \(False\ j\) have \(\ldots = map\text{-pmf} (\lambda zs. take\ i\ xs @ zs) \ (shuffle\ (drop\ i\ xs))\)
   by (subst shuffle-fisher-yates-step \[of\ drop\ i\ xs\])
   (simp-all add: \(map\text{-pmf-def} Let\text{-def}\ bind\text{-return-pmf}\ bind\text{-assoc-pmf}\))
finally show \(?thesis\).
qed

**definition** fisher-yates where
fisher-yates = fisher-yates-aux 0

**lemma** fisher-yates-correct: fisher-yates \(xs = shuffle\ xs\)
**unfolding** fisher-yates-def
by (subst fisher-yates-aux-correct) (simp-all add: \(map\text{-pmf-def} bind\text{-return-pmf}\'))

### 1.5 Backwards Fisher-Yates Shuffle

We can now easily derive the classical Fisher–Yates shuffle, which goes through the list from back to front and show its equivalence to the forward Fisher–Yates shuffle.

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

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

**lemma** fisher-yates-alt-aux-altdef:
\(i < length\ xs \implies fisher-yates-alt-aux\ i\ xs =\)
   \(map\text{-pmf\ rev} (fisher-yates-alt-aux\ (length\ xs - i - 1)\ (rev\ xs))\)
**proof** (induction \(i\) \(xs\) rule: fisher-yates-alt-aux.induct)
   case \(1\ i\ xs\)
   show \(?case\)
   **proof** (cases \(i = 0\))
   case False
   with \(1\).prems have \(map\text{-pmf\ rev} (fisher-yates-alt-aux\ (length\ xs - i - 1)\ (rev\ xs))\)
   = pmf-of-set \{length\ \(xs\) - Suc\ \(i\).<length\ \(xs\)\} \\(\Rightarrow\)
      \((\lambda z. fisher-yates-alt-aux\ (Suc\ (length\ xs - Suc\ i)))\)
      \((swap\ (rev\ xs))\ (length\ xs - Suc\ i)\)
      \(\Rightarrow\)
      \((\lambda x. return-pmf\ (rev\ x))\))
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 (λj. length xs - Suc j) {..i} {length xs - Suc i..<length xs}
  by (intro bij-betwI [where g = λj. length xs - Suc j]) auto
from bij have {length xs - Suc i..<length xs} = (λj. length xs - Suc j) ' {..i}
  by (simp add: bij-betw-def)
also from bij have pmf-of-set ... = map-pmf (λj. length xs - Suc j) (pmf-of-set {..i})
  by (subst map-pmf-of-set-inj) (auto simp: bij-betw-def)
also have map-pmf (λj. length xs - Suc j) (pmf-of-set {..i})
  = pmf-of-set {..i} (λx. fisher-yates-aux (Suc (length xs - Suc i)) (swap (rev xs) (length xs - Suc i) 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 ... = pmf-of-set {..i} (λj. 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 ... = fisher-yates-alt-aux i xs
  by (subst fisher-yates-alt-aux.simps[of i]) simp-all
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)
    (simp-all add: fisher-yates-alt-aux-correct shuffle-def map-pmf-of-set-inj)
qd

1.6 Code generation test

Isabelle’s code generator allows us to produce executable code both for 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.

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

end

References

