More minimal change orderings

Contents

1 More on revolving door order

1.1 Recall

The revolving door order on k-subsets of $\{1, 2, \ldots, n\}$:

$$R_0(n) = [\underbrace{0\,0\,\ldots 0}_{k}].$$

Likewise,

$$R_n(n) = [\underbrace{1\,1\,\ldots 1}_{k}]$$

Now, assume we have a minimal change sequence $R_k(n-1)$ for all k. Then

$$R_k(n) = 0 \cdot \{R_k(n-1)\}, 1 \cdot \{R_{k-1}(n-1)^R\}$$

For example,

$$R_1(2) = [0\, R_1(1), 1\, R_0(1)^R] = [0\,1, 1\,0].$$

1.2 Successor algorithm for the revolving door order

The algorithm is a bit intricate. Let's begin by stating it and then seeing why it works.

Algorithm: SuccessorRevolvingDoor

```
input: S, k, n.   S is a k-subset of n
j=1
while j in S
    j = j+1
if k-j odd
    if j=1
        decrease the smallest element of S  (STEP A)
    else if j=2
        increase the smallest element of S  (STEP B)
    else
        replace j-2 by j in S  (STEP C)
else
    e = jth smallest element of S
    if e+1 not in S
        if j=1
            increase the smallest element of S  (STEP D)
        else if k=k and e=n
            return no sucessor
        else
```

```
            replace j-1 by e+1 in S  (STEP E)
    else
        replace e+1 by j in S  (STEP F)
return S
```

The various steps have been labelled A through F to make them easier to refer to.

First notice that in step E, since we were not in the previous `else if`, then we must have e¡n, and so adding e+1 to S is valid. Next consider how each step affects j. Given S let T be the permutation returned by the algorithm. Let j be the value of j calculated by the algorithm on S and let j' be the value of j on T. Then

- If S is in A and the smallest element was 2 (and so is decreased to 1) then $j' = j + 1$. Otherwise $j' = j$.
- If S is in B then $j' = j$.
- If S is in C then $j' = j - 2$.
- If S is in D then $j' = j$.
- If S is in E then $j' = j - 1$.
- If S is in F then if $e = j$ we are filling in all the way up to e so $j' = j + 2$. Otherwise we are guaranteed to have no j in S and so $j' = j + 1$.

With this notation, lets now prove the algorithm is correct, that is that T really is the successor to S.

The proof is by induction on n. For the base case just check on $n = 1$. Now assume the algorithm works for $n - 1$ and consider n.

First suppose $n \notin S$. The only way the algorithm is in any way different from if it were called with $n - 1$ in place of n is in the case $j = k$ and $e = n - 1$. In that case the algorithm with $n - 1$ returns no successor and so by the induction hypothesis along with the characterization of the first as last elements from last time $S = \{1, 2, \ldots, k - 1, n - 1\}$ and so the algorithm returns $\{1, 2, \ldots, k - 2, n - 1, n\}$ which is correct. In all other cases with $n \notin S$ the algorithm is the same as with $n - 1$ and so by induction hypothesis is correct.

Now suppose $n \in S$. First note that the only way to obtain $j = k$ is when $S = \{1, 2, \ldots, k - 1, n\}$ in which case the algorithm correctly gives no successor. In all other cases $n \in S$ is not touched, so let T be S after the algorithm has run, let $\widetilde{S}$ be S with n removed and let $\widetilde{T}$ be T with n removed. $\widetilde{S}$ and $\widetilde{T}$ are in $R_{k-1}(n - 1)$ and so the parity of k has changed for $\widetilde{S}$ and $\widetilde{T}$.

Now work case by case using the observations above.

- If S is in A with first element greater than 2 then $\widetilde{T}$ has the same j but the parity of k has changed, so $\widetilde{T}$ is in D. D reverses what A did and so by induction $\widetilde{T}$ has $\widetilde{S}$ as its successor. Thus by definition T is the correct successor to S.
- If S is in A with first element 2 then in $\widetilde{T}$ the parity of j and k have both changed. This gives that $\widetilde{T}$ is in B. B also reverses A and so we can conclude as above that T is the correct successor to S
- If S is in B then in $\widetilde{T}$ the parity of j and k have bother changed so $\widetilde{T}$ is in A which reverses B and so we conclude as above.
- If S is in C then the parity of j in $\widetilde{T}$ is the same as in S while the parity of k has changed. Thus $\widetilde{T}$ is in F with $e = k$ which reverse C so as above we're done.
- If S is in D then j is unchanged in $\widetilde{T}$ but the parity of k has changed so $\widetilde{T}$ is in A which reverse D so done.
- If S is in E then j and k both change parity in $\widetilde{T}$ so $\widetilde{T}$ is in F which reverse E so done.

- If S is in F with $e \neq j$ then j and k both change parity in $\widetilde{T}$ so $\widetilde{T}$ is in E so done.
- If S is in F with $e = j$ then the parity of j stays the same while the parity of k changes in $\widetilde{T}$. This puts $\widetilde{T}$ in C and so again we're done.

This covers all cases, so by induction the algorithm returns the revolving door successor.

2 Minimal change order for permutations

2.1 Recall

Recall that a permuation of $\{1, 2, \dots, n\}$ is a bijection of $\{1, 2, \dots, n\}$ with itself. We can represent a permutation by the list of its values. For example given the permutation $\sigma : \{1, 2, 3\} \to \{1, 2, 3\}$ defined by $\sigma(1) = 3$, $\sigma(2) = 2$ and $\sigma(3) = 1$ we can represent σ by the list $[3, 2, 1]$.

2.2 Trotter-Johnson order

Let us consider the minimal distance between two permutations to be those permutations that differ by a transposition or swap. Thus, the permutations at distance 1 from the identity permutation are all precisely the set of transpositions, that is, permutations consisting of exactly one cycle of length 2. Let us add the additional constraint that the swap must occur for adjacent entries. This means for two permutations σ and τ of minimal distance that there exists some k such that

$$\sigma(i) = \tau(i) for i \in [1..n] \setminus \{k, k+1\}; \quad \sigma(k) = \tau(k+1) \text{ and} \sigma(k+1) = \tau(k).$$

For example,$\sigma = [3\,7\,1\,2\,5\,6\,4]$ and $\tau = [3\,7\,1\,5\,2\,6\,4]$. We now give a recursive scheme to generate permutations under this minimal distance. Let $\sigma = [\sigma_1\,\sigma_2 \dots \sigma_n]$ be a permutation of n. We associate the

$$\sigma^{\leftarrow} = \begin{bmatrix} \sigma_1 & \sigma_2 & \dots & \sigma_n & n+1 \\ \sigma_1 & \sigma_2 & \dots & n+1 & \sigma_n \\ & & \vdots & & \\ \sigma_1 & n+1 & \dots & \sigma_{n-1} & \sigma_n \\ n+1 & \sigma_1 & \dots & \sigma_{n-1} & \sigma_n \end{bmatrix} \qquad \sigma^{\rightarrow} = \begin{bmatrix} n+1 & \sigma_1 & \sigma_2 & \dots & \sigma_n \\ \sigma_1 & n+1 & \dots & \sigma_{n-1} & \sigma_n \\ & & \vdots & & \\ \sigma_1 & \sigma_2 & \dots & n+1 & \sigma_n \\ \sigma_1 & \sigma_2 & \dots & \sigma_n & n+1 \end{bmatrix}$$

For example

$$[1\,3\,2]^{\rightarrow} = \begin{bmatrix} 4 & 1 & 3 & 2 \\ 1 & 4 & 3 & 2 \\ 1 & 3 & 4 & 2 \\ 1 & 3 & 2 & 4 \end{bmatrix}$$

Now we describe the code. Let $R(n)$ be a listing of the permutations of length n, and suppose that $r_n(k)$ be the k-th element. We describe $R(n+1)$:

$$R(n+1) = \begin{bmatrix} r_n(1)^{\leftarrow} \\ r_n(2)^{\rightarrow} \\ \vdots \\ r_n(n!-1)^{\leftarrow} \\ r_n(n!)^{\rightarrow} \end{bmatrix}$$

This ordering is called the Trotter-Johnson ordering of permutations.

2.3 Trotter-Johnson rank and unrank

Its not too hard to rank recursively, we just need to know if we are zigzagging forwards or backwards, which we can determine from the rank of the permutation of the recursive call.

```
Algorithm: RecursiveRankTrotterJohnson
input L, n.  L a permutation of n written as a list of values
if n=1 then return 0
k = 1
while L(k) != n
    k = k+1
let S = L with n removed
r = RecursiveRankTrotterJohnson(S,n-1)
if r even
    return nr + n - k
return nr + k - 1
```

How do we write this nonrecursively? We just need to do the analogous calculation for every value, not just n.

```
Algorithm: RankTrotterJohnson
input L, n.  L a permutation of n written as a list of values
r = 0
for j from 2 to n
    k = 1
    i = 1
    while L(i) != j
        if L(i) < j
            k = k+1
        i = i+1
    if r even
        r = jr + j - k
    else
        r = jr + k - 1
return r
```

For example if we apply this algorithm to $L = (3, 4, 2, 1)$, $n = 4$ we begin with $r = 0$. When $j = 2$ we calculate $k = 1$; r is even so we have $r = 2 \cdot 0 + 2 - 1 = 1$. When $j = 3$ we calculate $k = 1$; r is odd so we have $r = 3 \cdot 1 + 1 - 1 = 3$. When $j = 4$ we calculate $k = 2$; r is odd so $r = 4 \cdot 3 + 2 - 1 = 13$. So that algorithm gives the rank of 13 which is correct.

The unrank algorithm is build similarly

```
Algorithm: UnrankTrotterJohnson
input n, r.
L(1) = 1
r2 = 0
for j from 2 to n
    r1 = floor((r*j!)/n!)
    k = r1 - j*r2
    if r2 is even
        for j from j-1 down to j-k
            L(i+1) = L(i)
        L(j-k) = j
    else
        for j from j-1 down to k+1
            L(i+1) = L(i)
        L(k+1) = j
    r2 = r1
return L
```

For the successor function see your homework. A reference for this material is Combinatorial Algorithms by Kreher and Stinson, chapter 2.