More minimal change orderings

Contents

1 More on revolving door order

In Lecture 15 we defined the minimal change order, $R_k(n)$, for the k-subsets of $\{n, n-1, \ldots, 1\}$ as follows.

$$R_0(n) = \emptyset, \quad R_n(n) = \{n, n-1, \ldots, 1\}, \quad R_k(n) = R_k(n-1),\ \{n\} \cup R_{k-1}(n-1)^R \text{ (for } 0 < k < n\text{)}.$$

Example. For example, we have $R_1(3) = \{1\}, \{2\}, \{3\}$ and $R_2(3) = \{2,1\}, \{3,2\}, \{3,1\}$ so, writing as subsets and as bit strings, we have

$$\begin{aligned} R_2(4) = R_2(3),\ \{4\} \cup R_1(3)^R &= \{2,1\},\ \{3,2\},\ \{3,1\},\ \{4,3\},\ \{4,2\},\ \{4,1\} \\ &= [0\,0\,1\,1],\ [0\,1\,1\,0],\ [0\,1\,0\,1],\ [1\,1\,0\,0],\ [1\,0\,1\,0],\ [1\,0\,0\,1]. \end{aligned}$$

1.1 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
if k=0 or k=n
    return "no successor"
else
    s := 1
    while s is not in S            // Find the smallest element s in S
        s := s+1
    j := 1
    while j is in S                // Find the smallest number j missing from S
        j := j+1
    if k-j is odd
        if j=1
            replace s by s-1 in S              // (CASE A)
        else if j=2
            replace 1 by 2 in S                // (CASE B)
        else
            replace j-2 by j in S              // (CASE C)
    else
        e := j+1
        while e is not in S        // Find the jth smallest element e of S
            e := e+1
        if e=n
            return "no successor"
        else if e+1 not in S
             if j=1
                replace s by s+1 in S          // (CASE D)
             else
                replace j-1 by e+1 in S        // (CASE E)
        else
            replace e+1 by j in S              // (CASE F)
    return S
```

Various cases have been labelled A through F to make them easier to refer to.

First we check that the numbers j and s are correctly found because at that point we have $0 < k < n$. Also e is correctly found since at that point $k - j$ is even, so $S \neq \{k, k-1, \ldots, 2, 1\}$. We consider how each case affects j. Let T be the permutation returned by the algorithm when S is input. Let j' be the value of j if the algorithm is run with input T instead of S. Then

- If S is in case A and $s = 2$, then 2 is replaced by 1 so $j' = j + 1$.
- If S is in case A and $s \neq 2$, then s is replaced by $s - 1 > 1$ so $j' = j$.
- If S is in case B, then $j' = j - 1$.
- If S is in case C, then $j' = j - 2$.
- If S is in case D, then $j' = j$.
- If S is in case E, then $j' = j - 1$.
- If S is in case F and $e = j + 1$, then $\{1, 2, \ldots, j, j+1\} \subseteq T$ so $j' = j + 2$.
- If S is in case F and $e > j + 1$, then $j + 1 \notin T$ so $j' = j + 1$.

We now prove that the algorithm is correct, that T really is the successor to S. Since $R_0(n)$ and $R_n(n)$ each list just one subset, the algorithm correctly returns `no successor` when $k = 0$ or $k = n$. Thus we may assume that $0 < k < n$.

The proof is by induction on n. For the base case $n = 1$, there is nothing to check since no integer k has $0 < k < n$. 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 were called with $n - 1$ in place of n is when $j = k$ and $e = n-1$, so $S = \{1, 2, \ldots, k-1, n-1\}$. Here the algorithm with $n-1$ returns `no successor` and the algorithm with n falls into case D or case E. If $j = 1$ (case D), then $S = \{n - 1\}$ and it returns the correct successor $T = \{n\}$. Otherwise (case E) it returns $T = \{1, 2, \ldots, k-2, n-1, n\}$ which is the correct successor of S. In all other cases with $n \notin S$ the algorithm is the same as with n as with $n - 1$ and returns the correct successor by the induction hypothesis.

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 returns `no successor`. In every other case n is not removed from S, so $n \in T$. Let

$$\widetilde{S} = S - \{n\} \qquad \text{and} \qquad \widetilde{T} = T - \{n\}.$$

By the definition of $\{n\} \cup R_{k-1}(n-1)^R$, T is the successor of S if and only if $\widetilde{S}$ is the successor of $\widetilde{T}$. By the induction hypothesis, the successor of $\widetilde{T}$ in $R_{k-1}(n-1)$ is correctly found by the algorithm. To finish this proof we need to verify, in each of the cases above, that if the algorithm changes S into T, then it changes $\widetilde{T}$ into $\widetilde{S}$. Since $\widetilde{T}$ is in $R_{k-1}(n-1)$, the parity of k has changed for $\widetilde{T}$ compared to S.

- If S is in case A with $s = 2$, then for $\widetilde{T}$ the parities of j and k have both changed, compared to S. Thus $\widetilde{T}$ is in case B, which replaces 1 with 2, reverses the action of case A and outputs $\widetilde{S}$.
- If S is in case A with $s > 2$, then $\widetilde{T}$ has the same j but the parity of $k - j$ has changed, so $\widetilde{T}$ is in case D. With input $\widetilde{T}$, case D reverses what case A did to S (decrementing/incrementing s), so it outputs $\widetilde{S}$.
- If S is in case B, then in $\widetilde{T}$ the parity of j and k have both changed so $\widetilde{T}$ is in case A, which reverses the effect of case B and outputs $\widetilde{S}$.
- If S is in case C, then the parity of j in $\widetilde{T}$ is the same as in S so the parity of $k - j$ has changed. Thus $\widetilde{T}$ is in case F with $e = j' + 1 = j - 1$, which reverses the effect of case C and outputs $\widetilde{S}$.
- If S is in case D, then j is unchanged in $\widetilde{T}$ so the parity of $k - j$ has changed. Therefore $\widetilde{T}$ is in case A which reverses the action of case D and outputs $\widetilde{S}$.
- If S is in case E, then j and k both change parity in $\widetilde{T}$. Therefore $\widetilde{T}$ is in case F which reverses the action of case E and outputs $\widetilde{S}$.

- If S is in case F with $e = j+1$, then the parity of j stays the same, so parity of $k-j$ changes in $\widetilde{T}$. This puts $\widetilde{T}$ in case C, which reverses the action of case F and outputs $\widetilde{S}$.
- If S is in case F with $e > j+1$, then j and k both change parity. Thus $\widetilde{T}$ is in case E, which reverses the action of case F and outputs $\widetilde{S}$.

This covers all cases, completing the proof. □

2 Minimal change order for permutations

2.1 Recall

Recall that a permuation of $\{1,2,\ldots,n\}$ is a bijection of $\{1,2,\ldots,n\}$ 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 Johnson-Trotter 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

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

For example, $\sigma = [3\,7\,1\,2\,5\,6\,4]$ and $\tau = [3\,7\,1\,5\,2\,6\,4]$ differ by such a minimal change. We now give a recursive scheme to generate permutations under this minimal distance. Let $\sigma = [\sigma_1\,\sigma_2\,\ldots\sigma_n]$ be a permutation of n. We associate σ with two lists of permutations of $1,2,\ldots,n+1$, which we may record as rows of a matrix.

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

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 Johnson-Trotter ordering of permutations.

2.3 Johnson-Trotter 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. For the recursive call, we input the permutation S of $\{1, 2, \ldots, n-1\}$ obtained from L by deleting the entry n

Algorithm: RecursiveRankJohnsonTrotter

```
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                // Find the position k of n in L
    k := k+1
let S := L with n deleted
r := RecursiveRankJohnsonTrotter(S,n-1)
if r is even
    return nr + n - k
else
    return nr + k - 1
```

How do we write this nonrecursively? We just need to do the analogous calculation for every value $j = 2, 3, \ldots, n$, not just n. For each j, we need to find the position k of the entry j in the permutation of $\{1, 2, \ldots j\}$ obtained from L by deleting all the entries larger than j. Let us denote this deleted permutation by L_j. In the above algorithm, we used $S = L\_(n-1)$. Using k we can find the compute the rank r of L_j in terms of the rank of $L\_(j-1)$.

Algorithm: RankJohnsonTrotter

```
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        // Find the position k of j in the deleted permutation L_j
        if L(i) < j
            k := k+1
        i := i+1
    if r is 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 $L\_2 = [2\ 1]$ and $k = 1$; since r is even, we update $r := 2 \cdot 0 + 2 - 1 = 1$. When $j = 3$ we calculate $L\_3 = [3\ 2\ 1]$ and $k = 1$; r is odd so we update $r := 3 \cdot 1 + 1 - 1 = 3$. When $j = 4$ we calculate $L\_4 = L = [3\ 4\ 2\ 1]$ and $k = 2$; r is odd so $r := 4 \cdot 3 + 2 - 1 = 13$. So $\textsc{Rank}([3\ 4\ 2\ 1]) = 13$.

The unrank algorithm is built similarly

Algorithm: UnrankJohnsonTrotter

```
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.