Computing Virtual Nesting Controls for Network Revenue Management under Customer Choice Behavior

Online Appendix

Garrett J. van Ryzin
Graduate School of Business, Columbia University, New York, New York 10027,
e-mail: gjv1@columbia.edu.

Gustavo J. Vulcano
Stern School of Business, New York University, New York, New York 10012,
e-mail: gvulcano@stern.nyu.edu.

A Complexity analysis

The procedure for calculating a sample based gradient accounting for choice behavior consists of two passes: In the *forward* pass, we keep track of the state of the network observed by the arrival stream of customers (i.e., available capacity met by customer t, and the quantity allocated to the different products in the preference order). The *backward* pass rebuilds the capacity found by each customer, identifies all the binding protection levels, and calculates the gradients with respect to protection levels and leg capacities accordingly. One can show that the overall complexity of this algorithm is determined by the backward pass. In online Appendix C, we present a detailed pseudocode of the procedure for the parallel flight case (e.g. see Examples 2, 3, 4 and 6 in Section 4 of the main paper).

Taking equations (A3) and (A4) in Appendix A, the complexity is defined by the computation of the derivatives of the acceptance function u_j. For a given customer t, following equations (C2) and (C3) in Appendix C, the following is a sketch of the calculation procedure of the partial derivatives with respect to protection levels and capacity for the general network case:

For every customer t:
  For every product $[k]$ in the preference list ($k = 1, 2, \ldots, p_t$) do:
    If y_{ic} is the unique binding protection level of product $[k]$, then
      the partial derivatives with respect to y_{ic} and x_i are -1 and 1, respectively.
    If there is a unique binding protection level of product $[k]$ other than y_{ic},
    or if product $[k]$ exhausts the quantity q_t, then
      Go through the list of products $[1], \ldots, [k-1]$, and for each such product
      compute the partial derivatives with respect to y_{ic} and x_i.

Let $\bar{p}$ be an upper bound for the number of purchasing choices among customers (i.e., $\bar{p} \leq p_t, \forall t$). Typically, we could consider a value $\bar{p}$ smaller than 10, meaning that there are less than ten possible itineraries that a customer considers. For every value t, the number of operations is bounded by a constant that depends on $\bar{p}$. Therefore, the overall complexity is given by $O(T)$, which is the same

order of magnitude as the simpler version of the problem with no choice behavior studied in van Ryzin and Vulcano [28, Section 2.4].

B Properties of the revenue function

This model inherits the theoretical properties described for the revenue function in van Ryzin and Vulcano [28, Section 2.5], and we refer the technical reader to the proofs therein.

Here, we just summarize the main findings. The first result is a negative one: The sample path revenue function is not quasiconcave in the protection levels vector, implying that we cannot preclude the possibility that there maybe local optima in the expected revenue of our continuous problem.

Theorem B1 *There exist sample paths ω on which the sample path revenue function $R(y, \omega)$ for the continuous problem is not quasiconcave.*

Next lemma justifies the interchange of the expectation and differentiation operators on a sample path ω:

Lemma B1 *For the randomly perturbed model (9)-(10), the gradient $\nabla_y E[R(y, \omega)]$ exists for all $y \in \Theta$, and $\nabla_y E[R(y, \omega)] = E[\nabla_y R(y, \omega)]$.*

The following result is critical for solving the smoothed version of (7):

Theorem B2 *The objective function $g(y)$ is continuously differentiable on Θ.*

Finally, we need the boundedness of the variance of the stochastic partial derivative of the revenue function to prove the local convergence of the stochastic algorithm.

Lemma B2 *There exists a finite constant C such that $Var\left(\frac{\partial}{\partial y_{ic}} R(y, \omega)\right) < C$.*

C Pseudo-code for sample path gradient calculation

We present here a detailed pseudocode of the sample path gradient calculation for the parallel flight case (e.g., see Examples 2, 3, 4, and 6 in the main body of the paper).

```
procedure Compute_Sample_Path_Gradient(x,r,j,q,p,Nres,res,T,y)
{
/* input definitions
x[i] = remaining capacity on resource i (modified by program)
r[j] = revenue obtained from selling a unit of product j
j[t,n] = n-th product preferred by customer t on sample path
q[t] = quantity requested by customer t on sample path
p[t] = number of products with positive utility preferred by customer t
Nres[j] = number of resources used by product j
res[j,n] = n-th resource used by product j
T = total number of customer requests
y[i,c] =  protection level for class c on resource i. We assume y[i,0]=0
/*

/* local variables
min_space[t,n] = minimum remaining capacity for n-th product in the preference order for customer t
u[t,n] = quantity accepted for n-th product in the preference order for customer t
dar[j] = displacement adjusted revenue for product j
dar_spilled = displacement adjusted revenue "spilled from above" for the current product
              in customer t preference
gradX[i] = sample path gradient with respect to x
gradY[i,c] = sample path gradient with respect to y
/*

/* functions used /*
InitializeVector(v,k); /* initialize vector v with value k
Min(a,b);         /* returns minimum of a and b /*

/* initialize gradients to zero /*
InitializeVector(gradX,0);
InitializeVector(gradY,0);

/* forward pass /*
for (t:=1 to T)
  {q:=q[t];
   for (n:=1 to p[t])
     {j:=j[t,n];  /* until exhausting quantity q, loop over products following the preference
                     of customer t /*
```

```
      MIN:=INFTY;
      for (k:=1 to Nres[j]) /* compute remaining capacity for j /*
        {i:=res[j,k];
         if (x[i]-y[i,c[i,j]-1] < MIN)
           {if (x[i]-y[i,c[i,j]-1] >= 0)
              MIN:=x[i]-y[i,c[i,j]-1];  /* save the new minimum (remaining capacity) for j /*
            else
              MIN:=0;  /* protection level for resource i is beyond the remaining capacity /*
           }
        }
      min_space[t,n]:=MIN; /* save remaining capacity for the n-th product in the order /*
      u[t,n]:=Min(q,MIN);  /* accepted amount of the n-th product is minimum between q and MIN /*
      q:=q-u[t,n];
      for (k:=1 to Nres[j])  /* update remaining capacity /*
        {i:=res[j,k];
         x[i]:=x[i]-u[t,n];
        }
     }
  }

/* backward pass /*
for (t:=T down to 1)
  {q:=q[t];
   dar_spilled:=0;
   InitializeVector(dar,0);
   for (n:=p[t] down to 1)
     {j:=j[t,n];
      MIN:=min_space[t,n];
      SUM:=0;
      for (k:=1 to Nres[j])
        {i:=res[j,k];
         x[i]:=x[i]+u[t,n];
         SUM:=SUM+gradX[i];
        }
      dar[j]:=r[j]-SUM;
      if (0 < u[t,n] .and. u[t,n] == MIN)
        {for (l:=1 to Nres[j])
           {i:=res[j,l];
            if (x[i]-y[i,c[i,j]-1] == MIN)
              {gradX[i]:=gradX[i] + dar[j] - dar_spilled;
               for (c:=1 to c[i,j]-1) /* search for protection levels for classes higher than c[i,j] /*
                 {if (x[i]-y[i,c] == MIN)  /* update gradY if y[i,c] is binding /*
```

```
                gradY[i,c]:=gradY[i,c] - dar[j] + dar_spilled;
              }
            }
          }
        }
      else
        if ( (0 < u[t,n] .and. u[t,n] < MIN) .or. (u[t,n] == 0 .and. MIN > 0) )
          dar_spilled:=r[j]-SUM;
    }
  }
}
```