ISYE 6669 Final Project Report

Luffina Huang
Ali Mujtaba Lakdawala
Tess Leggio

Nov. 30, 2018

Problem 1: Column Generation for the Diet Problem

1. **Complete the construction of the column generation code and submit the mos file online and also submit a physical copy with the final report.**

See file: diet.colgen.V1.mos

2. **Print out the final solution from the code and submit it in the final report**

See file: OUTPUT-diet.colgen.V1.docx

23 columns generated
Total Cholesterol = 0
Total Calories = 1570.33

Foods and quantities chosen:

Spices, basil, dried	0.043662
Salt, table	0.032192
Oil, vegetable, industrial, palm kernel (hydrogenated), confect	0.169762
Cereals ready-to-eat, GENERAL MILLS, Whole Grain TOTAL	0.163674
Cereals ready-to-eat, KASHI Heart to Heart by KELLOGG	0.137218
Mushrooms, shiitake, dried	0.22785
Nuts, brazilnuts, dried, unblanched	0.189391
Water, bottled, non-carbonated, CALISTOGA	49.9141

Soy protein isolate, potassium type, crude protein basis	0.500377
Leavening agents, baking powder, double-acting, straight phosph	0.010382
Leavening agents, cream of tartar	0.084067
Rice bran, crude	0.219409
Whale, beluga, eyes, raw (Alaska Native)	0.007405
Sweeteners, tabletop, fructose, dry, powder	2.27781
Fruit-flavored drink, powder, with high vitamin C, low calorie	0.106948

3. **Find an optimal diet with the total calorie intake between 1800 cal and 2000 cal. and resolve the above problem. Report in the final report the composition of optimal diet and the total amount of each nutrient provided by the diet.**

See files: diet.colgen.V2.mos, OUTPUT-diet.colgen.V2.docx

23 columns generated
Total Cholesterol = 0
Total Calories = 2000

Foods and quantities chosen:

Spices, basil, dried	0.036657
Salt, table	0.03165
Oil, vegetable, industrial, palm kernel (hydrogenated), confect	0.655666
Cereals ready-to-eat, GENERAL MILLS, Whole Grain TOTAL	0.162583
Cereals ready-to-eat, KASHI Heart to Heart by KELLOGG	0.138502

Mushrooms, shiitake, dried	0.227938
Nuts, brazilnuts, dried, unblanched	0.18937
Water, bottled, non-carbonated, CALISTOGA	49.9144
Soy protein isolate, potassium type, crude protein basis	0.501168
Leavening agents, baking powder, double-acting, straight phosph	0.012777
Leavening agents, cream of tartar	0.085064
Rice bran, crude	0.222594
Whale, beluga, eyes, raw (Alaska Native)	0.006433
Sweeteners, tabletop, fructose, dry, powder	2.27871
Fruit-flavored drink, powder, with high vitamin C, low calorie	0.107344

Total Nutrients of Chosen Food Quantities:

Protein	56
Carbohydrate, by difference	300
Calorie	2000
Water	5000
Energy	8369.56
Calcium, Ca	1000
Iron, Fe	27.604

Magnesium, Mg	400
Phosphorus, P	1226.05
Potassium, K	3500
Sodium, Na	1500
Zinc, Zn	16.2845
Copper, Cu	3.15585
Manganese, Mn	5.86523
Selenium, Se	400
Vitamin A, RAE	900
Vitamin E (alpha-tocopherol)	30
Vitamin D	400
Vitamin C, total ascorbic acid	305.918
Thiamin	2.10524
Riboflavin	2.4888
Niacin	35
Pantothenic acid	12.5639
Vitamin B-6	4.51372
Folate, total	580.5
Vitamin B-12	6

Vitamin K (phylloquinone)	80
Cholesterol	0
Fatty acids, total trans	0
Fatty acids, total saturated	65.5416

Problem 2: Solution Strategies for the Cutting Stock Problem

Problem 2.1: The Kantorovich Formulation: A Modeling Exercise

1. **How many variables and constraints are there in this formulation?**

a. There are 300 total variables.
   i. y variables: $K = 60$
   ii. x variables: $K * m = 60 * 4 = 240$

b. There are 724 total constraints, 64constraints excluding variable bounds and integrality constraints.
   i. Constraint 1: $m = 4$
   Demand must be satisfied with equality for each item

$$\sum_{k=1}^{K} x_i^k = b_i \quad \forall\, i \in \{1,\dots,m\}$$

   ii. Constraint 2: $K = 60$
   The width of a large roll can not be exceeded for each large roll

$$\sum_{i=1}^{m} w_i x_i^k \le y \;\; {}^{k}W \quad \forall\, k \in \{1,\dots,K\}$$

   iii. Constraint 3: $2(K * m) + 3(K) = 2(60 * 4) + 3(60) = 600$
   Variable bounds and integrality constraints.

$$x_i^k \ge 0 \;\; \forall\, k \in \{1,\dots,K\}, i \in \{1,\dots,m\}$$
$$y^k \ge 0 \;\; \forall\, k \in \{1,\dots,K\}$$
$$y^k \le 0 \;\; \forall\, k \in \{1,\dots,K\}$$
$$y^k \; integer \;\; \forall\, k \in \{1,\dots,K\}$$
$$x_i^k \; integer \;\; \forall\, k \in \{1,\dots,K\}, i \in \{1,\dots,m\}$$

2. **Use the files cs.Kantorovich.partial.mos as a starting point. Finish the Xpress code and solve the problem using the data file kant1.dat. Submit the mos file online and also submit a physical copy with the final report. Print out the optimal objective value and the optimal solution in the report.**

See files cs.Kantorovich.V1.mos, OUTPUT-cs.Kantorovich.V1.docx

Optimal objective value: 18
Optimal solution:
y(1)=1, x(1,2)=2, x(1,4)=3
y(4)=1, x(4,1)=15
y(6)=1, x(6,1)=1, x(6,2)=5, x(6,4)=1
y(7)=1, x(7,2)=2, x(7,4)=3

y(10)=1, x(10,1)=1, x(10,2)=5, x(10,4)=1
y(12)=1, x(12,2)=4, x(12,3)=1, x(12,4)=1
y(13)=1, x(13,1)=7, x(13,4)=2
y(17)=1, x(17,2)=1, x(17,3)=3, x(17,4)=1
y(18)=1, x(18,1)=1, x(18,2)=5, x(18,4)=1
y(30)=1, x(30,2)=2, x(30,4)=3
y(31)=1, x(31,2)=2, x(31,4)=3
y(35)=1, x(35,2)=7
y(36)=1, x(36,4)=4
y(37)=1, x(37,2)=4, x(37,3)=1, x(37,4)=1
y(45)=1, x(45,2)=2, x(45,4)=3
y(51)=1, x(51,2)=4, x(51,3)=1, x(51,4)=1
y(52)=1, x(52,2)=1, x(52,3)=3, x(52,4)=1
y(54)=1, x(54,2)=4, x(54,3)=1, x(54,4)=1

3. **Next change K to 600 and change m to 10 and use the data file cs1.dat. What do you observe? Is it easy or hard to solve the problem using the Kantorovich formulation?**
   a. **How many branch-and-bound (BB) nodes are searched by the BB algorithm?**
      13987 nodes
   b. **What is the objective value of the best lower bound and the objective value of the best integer solution found?**
      Best lower bound = 339.78
      Best integer solution = 341
   c. **What is the optimality gap given by the above two numbers?**
      0.357771%
   d. **How many integer solutions are found by the BB algorithm?**
      12 solutions
   e. **How many seconds did the algorithm take?**
      The algorithm took 41.8s before we manually stopped it
   f. **Lastly, relax the integrality constraints**
      Upon relaxing the integrality constraints, the Simplex dual algorithm ran in 1688 iterations to find an optimal objective of 339.78 in 0.0s

      Solving the large integer problem was hard to solve using the Kantorovich formulation of the cutting stock problem, as demonstrated by the large number of branch-and-bound nodes searched by the algorithm and the relatively long algorithm run time. Xpress was able to solve the LP relaxation of the same problem very quickly, in only 1688 iterations and in less than 0.1s.

Problem 2.2: The Gilmore-Gomory Formulation: Use Column Generation

1. **Complete the column generation code. Submit the completed mos file online and also submit a physical copy with the final report.**

See file: cs.colgen.V2.2.mos

2. **Print out solution summary for each test instance (cs1.dat, cs2.dat, cs3.dat), following the instructions in the cs.colgen.partial.mos file.**

See file: OUTPUT-cs.colgen.V2.2.docx

3. **In your report, discuss the differences between the LP solutions and the integer solutions obtained in the three test instances, also the differences between the integer solutions of the two approaches (rounding and resolving). Which approach produces better solution?**

Equality Demand Constraint:

File Name	*LP Solution*	*Rounding up*	*Constraining to Integer*
cs1.dat	339.78 rolls	345 rolls	364 rolls
cs2.dat	47.925 rolls	52 rolls	99 rolls
cs3.dat	14.7 rolls	17 rolls	19 rolls
kant1.dat	17.8736 rolls	19 rolls	22 rolls

Total Waste Equality Demand Constraint

File Name	*LP Solution*	*Rounding up*	*Constraining to Integer*
cs1.dat	0 waste	0 waste	3633 waste
cs2.dat	0 waste	0 waste	4086 waste
cs3.dat	0 waste	0 waste	1075 waste
kant1.dat	7.24138 waste	8 waste	750 waste

The linear program solutions for all data files are strictly less than both integer solutions of the same problems. This makes sense given that the feasible region of the integer program is contained within the feasible region of the linear program. Therefore, the optimal cost of the linear relaxation will always be less than or equal to

the optimal cost of the integer program, i.e., $Z_{LP} \leq Z_{IP}$.

Between the two integer solutions, rounding up the linear program solution resulted in a lower cost than constraining the variables to be integer. This is because the rounding up method does not actually satisfy the equality demand constraint, while the other integer formulation satisfies the equality demand constraint. Due to this additional constraint, the feasible region of the program which constrains all variables to be integer is smaller than that of the rounding up method. Therefore, the optimal cost of the rounding up method will always be less than or equal to that of the strict integer program, i.e., $Z_{rounding\ up} \leq Z_{IP}$.

Thus, for the problem with the equality demand constraint: $Z_{LP} \leq Z_{rounding\ up} \leq Z_{IP}$. The linear program will produce the best solution in terms of optimal cost. The strict integer program will produce the best solution if the paper company requires that rolls must be used in full. The rounding up method is not a good method to solve this problem as it does not satisfy the demand constraint exactly.

4. Now, change the demand constraint from = to ≥, that is, $\sum_{j=1}^{N} \boldsymbol{a_{ij}x_j} \geq \boldsymbol{b_i}$.

Solve the LP relaxation of the resulting cutting stock problem and obtain the integer solutions using the same procedure as in the above question for kant1.dat, cs1.dat, cs2.dat, cs3.dat. Compare the results to the ones of the equality demand constraint. If the two formulations give different solutions, elaborate how the solutions are different for each test case.

See files: cs.colgen.V2.2_greaterequal.mos, OUTPUT-cs.colgen.V2.2_greaterequal.docx

Greater than or Equal to Demand Constraint:

File Name	*LP Solution*	*Rounding up*	*Constraining to Integer*
cs1.dat	339.78 rolls	345 rolls	342 rolls
cs2.dat	47.925 rolls	52 rolls	51rolls
cs3.dat	14.7 rolls	17 rolls	15 rolls
kant1.dat	17.8736 rolls	19 rolls	19 rolls

Total Waste Greater than or Equal to Demand Constraint:

File Name	*LP Solution*	*Rounding up*	*Constraining to Integer*

cs1.dat	0 waste	0 waste	10 waste
cs2.dat	0 waste	0 waste	73 waste
cs3.dat	0 waste	0 waste	30 waste
kant1.dat	7.24138 waste	8 waste	8 waste

The linear program solutions for all data files are strictly less than both integer solutions of the same problems. This makes sense given that the feasible region of the integer program is contained within the feasible region of the linear program. Therefore, the optimal cost of the linear relaxation will always be less than or equal to the optimal cost of the integer program, i.e., $Z_{LP} \leq Z_{IP}$.

Between the two integer solutions, the strict integer program that constrains all variables to be integer resulted in a lower optimal cost than that of the rounding up solution. Both of the integer approaches are subject to the same constraints, so it makes sense that the strict integer program would result in lower optimal costs as the IP is solving for the optimal IP solution directly, while the rounding up method solves for the LP solution and find the nearest feasible integer solution to the LP solution, which may or may not actually be the optimal integer solution. Therefore, the optimal cost of the rounding up method will always be greater than or equal to that of the strict integer program for the greater than or equal to demand constraint, i.e., $Z_{IP} \leq Z_{rounding\ up}$.

Thus, for the problem with the greater than or equal to demand constraint: $Z_{LP} \leq Z_{IP} \leq Z_{rounding\ up}$. The linear program will produce the best solution in terms of optimal cost. The strict integer program will produce the best solution if the paper company requires that solutions must be integer.

Problem 2.3: MinWaste Objective

1. The new formulation of the problem is the following:

$$\boldsymbol{min} \sum_{j=1}^{N} \boldsymbol{x_j} * (\boldsymbol{W} - \sum_{i=1}^{m} \boldsymbol{a_{ij} w_i})$$

$$\boldsymbol{s.t.} \sum_{j=1}^{N} \boldsymbol{a_{ij} x_j} = \boldsymbol{b_i} \ \forall \ \boldsymbol{i} \in \{\boldsymbol{1}, \ldots, \boldsymbol{m}\}$$

$$\boldsymbol{x_j} \geq \boldsymbol{0} \ \forall \ \boldsymbol{j} \in \{\boldsymbol{1} \ldots \boldsymbol{N}\}$$

2. The RMP is:

$$\boldsymbol{min} \sum_{j \in I} \boldsymbol{x_j} * (\boldsymbol{W} - \sum_{i=1}^{m} \boldsymbol{a_{ij} w_i})$$

$$\boldsymbol{s.t.} \sum_{j \in I} \boldsymbol{a_{ij} x_j} = \boldsymbol{b_i} \ \forall \ \boldsymbol{i} \in \{\boldsymbol{1}, \ldots, \boldsymbol{m}\}$$

$$x_j \geq 0 \; \forall \; j \in I$$

New pricing subproblem for this question is now:

$$min \; (W - \sum_{i=1}^{m} a_i w_i - \sum_{i=1}^{m} y_i a_i)$$
$$s.t. \sum_{i=1}^{m} a_i w_i \leq W$$
$$a_i \geq 0 \; \forall \; i \in \{1, \ldots, m\}$$
$$a_i \; is \; integer \; \forall \; i \in \{1, \ldots, m\}$$

3. See file: OUTPUT-.cs.colgen.min-waste.mod

Equality Demand Constraint:

File Name	*LP Solution*	*Rounding up*	*Constraining to Integer*
cs1.dat	339.78 rolls	344 rolls	371 rolls
cs2.dat	47.93 rolls	52 rolls	99 rolls
cs3.dat	14.7 rolls	17 rolls	19 rolls
kant1.dat	17.8736 rolls	19 rolls	22 rolls

4. For each test case, does the LP solution of CS-min-waste have the same optimal objective with that of the original cutting- stock problem? What about the optimal LP solution? Does the integer solution of CS-min-waste have the same objective with the original problem? Compare the total amount of wasted paper of the integer solution of the cs-min-waste model with that of the original model? How much different are they?

File Name	*LP Solution Original*	*LP Solution Min Waste*	*% Difference*
cs1.dat	339.78 rolls	339.78 rolls	0%
cs2.dat	47.93 rolls	47.93 rolls	0%
cs3.dat	14.7 rolls	14.7 rolls	0%
kant1.dat	17.8736 rolls	17.8736 rolls	0%

The LP Solutions for the original cutting stock problem are the same for the CS-min-waste problem. The percent difference is shown in the table above.

File Name	*LP Waste - Original*	*LP Waste - Min Waste*	*% Difference*
cs1.dat	0 waste	0 waste	0%
cs2.dat	0 waste	0 waste	0%
cs3.dat	0 waste	0 waste	0%
kant1.dat	7.24138 waste	7.24138 waste	0%

The wastes are the same for the LP solution for both of the problems.

File Name	*IP Solution Original*	*IP Solution Min Waste*	*% Difference*
cs1.dat	364 rolls	371 rolls	1.90%
cs2.dat	99 rolls	99 rolls	0%
cs3.dat	19 rolls	19 rolls	0%
kant1.dat	22 rolls	22 rolls	0%

File Name	*IP Waste - Original*	*IP Waste - Min Waste*	*% Difference*
cs1.dat	3633 waste	4683 waste	25.3%
cs2.dat	4086 waste	4086 waste	0%
cs3.dat	1075 waste	1075 waste	0%
kant1.dat	750 waste	750 waste	0%

The IP solutions are only different for cs1.dat. The rest of the integer solutions are similar. The percent difference is shown in the table above. The waste is also the same for all the IP solutions except for cs1 and the difference is shown above in the waste table.

5. After you run the test cases, does your results suggest some interesting relation between the LP solutions of the two cutting stock models? Prove that these two models (both the LP relaxation and the integer version of the CS-min-waste and the original cutting-stock problems) are in fact equivalent, thus

always give the same optimal objective value. Or argue that your experimental results disprove the above statement.

From our test cases we see that the LP relation between the two cutting stock models is the same.

Let's assume that the two models are equivalent. Then it should be the case that the Integer and LP Solutions are the same.

However, we see clearly that the solution for cs1 when comparing to IP Original Solution and IP Min Waste are not the same.

Thus by contradiction we disprove the above statement.

6. See file: OUTPUT-cs.colgen.min-waste.mod.greater-equal constraint

Greater than or Equal to Demand Constraint:

File Name	*LP Solution*	*Rounding up*	*Constraining to Integer*
cs1.dat	391.68 rolls	394 rolls	455 rolls
cs2.dat	80.66 rolls	84 rolls	85 rolls
cs3.dat	15 rolls	16 rolls	16 rolls
kant1.dat	31.67 rolls	32 rolls	31 rolls

For each test case, does the LP solution of CS-min-waste have the same optimal objective with that of the original cutting- stock problem? What about the optimal LP solution? Does the integer solution of CS-min-waste have the same objective with the original problem? Compare the total amount of wasted paper of the integer solution of the cs-min-waste model with that of the original model? How much different are they?

File Name	*LP Solution Original*	*LP Solution Min Waste*	*% Difference*
cs1.dat	339.78 rolls	391.68 rolls	14.19%
cs2.dat	47.925 rolls	80.66 rolls	50.92%
cs3.dat	14.7 rolls	15 rolls	2.02%

kant1.dat	17.8736 rolls	31.67 rolls	55.71%

File Name	*LP Waste - Original*	*LP Waste - Min Waste*	*% Difference*
cs1.dat	0 waste	0 waste	0%
cs2.dat	0 waste	0 waste	0%
cs3.dat	0 waste	0 waste	0%
kant1.dat	7.24138 waste	0 waste	100%

LP Solution for the original cutting stock solution and the Min Waste solution for the greater than equal to constraint are not the same. The percent differences are show in the table above.

The waste is the same except for Kant1 for which it is 100%.

File Name	*IP Solution Original*	*IP Solution Min Waste*	*% Difference*
cs1.dat	342 rolls	455 rolls	28.36%
cs2.dat	51rolls	85 rolls	50%
cs3.dat	15 rolls	16 rolls	6.45%
kant1.dat	19 rolls	31 rolls	48%

File Name	*IP Waste - Original*	*IP Waste - Min Waste*	*% Difference*
cs1.dat	10 waste	0 waste	100%
cs2.dat	73 waste	0 waste	100%
cs3.dat	30 waste	0 waste	100%

kant1.dat	8 waste	0 waste	100%

The IP Solution for the original problem and min-waste for the greater than equal to constraint are not the same. The percent differences are shown in the table above. We get 0 waste for all the >= constraint since we are able to cut more than the demand and not waste any part of the roll.

Problem 3: TSP using Dantzig-Fulkerson-Johnson Formulation

1. **Formulation with no small subtour elimination constraints**

Let this be the Formulation (i). It took 30.292 seconds and generated 194 constraints in total.

2. **Formulation with 1-city subtour elimination constraints**

Let this be the Formulation (ii). It took 31.77 seconds and generated 145 constraints in total.

3. **Formulation with both 1-city and 2-city subtour elimination constraints**

Let this be the Formulation (iii). It took 35.717 seconds and generated 110 constraints in total.

4. **Formulation with 1-city, 2-city and 3-city subtour elimination constraints**

Let this be the Formulation (iv). It took 81.414 seconds and generated 69 constraints in total.

5. **Formulation with 1-city, 2-city, 3-city and 4-city subtour elimination constraints**

Let this be the Formulation (v). Reach time limit of 20 min. But, it took only 1864.71 seconds (31.06 minutes) and generated 62 constraints in total.

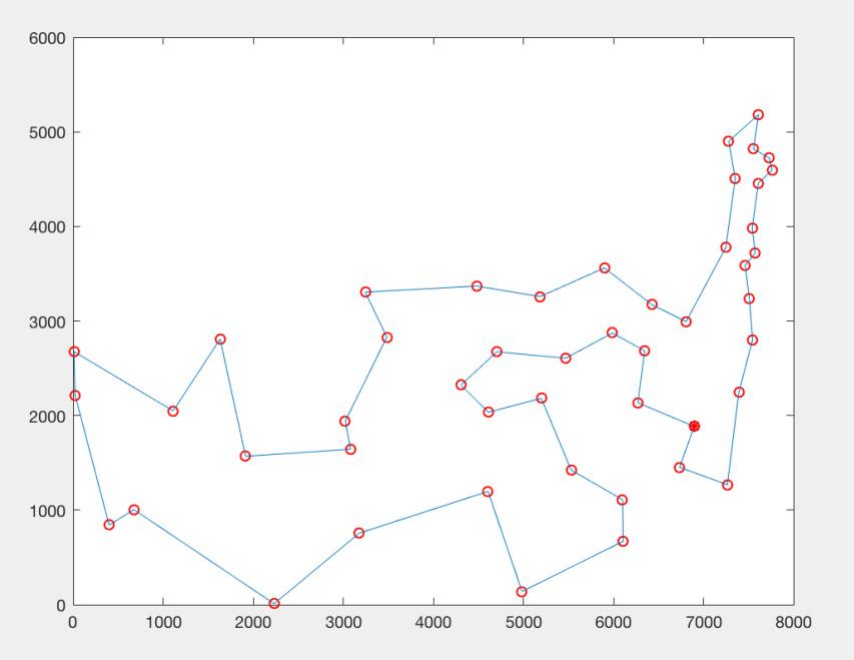
6. **For each of (i)-(v), how many subtour elimination constraints did the formulation begin with?**

There are **48** constraints to eliminate the 1-city subtours, **1128** constraints to eliminate the 2-city subtours, **17296** constraints to eliminate the 3-city subtours, and **194580** constraints to eliminate the 4-city subtours. Thus,
Formulation (i) begins with **0** subtour elimination constraint (SEC). Formulation (ii) begins with **48** SECs.
Formulation (iii) begins with 48+1128 = **1176** SECs.
Formulation (iv) begins with 1176 + 17296 = **18472** SECs.
Formulation (v) begins with 18472 + 194580 = **213052** SECs.

7. Optimal 48-city TSP tour

The optimal 48-city TSP tour distance is **33523.7**

The plot generated by Matlab is as followed :



We overlaid our optimal TSP tour plot generated from Matlab with the US map to determine the name of state capital represented by each index, as seen in the figure below.

8. Name of the state capitals in optimal 48-city TSP tour:

The optimal TSP tour for the 48 cities is as followed :

Atlanta → Nashville → Frankfort → Indianapolis → Springfield → Des Moines → Lincoln →

Topeka → Jefferson City → Little Rock → Jackson → Baton Rouge → Austin →

Oklahoma City → Santa Fe → Phoenix → Carson City → Sacramento → Salem →

Olympia → Boise → Helena → Salt Lake City → Denver → Cheyenne → Pierre →

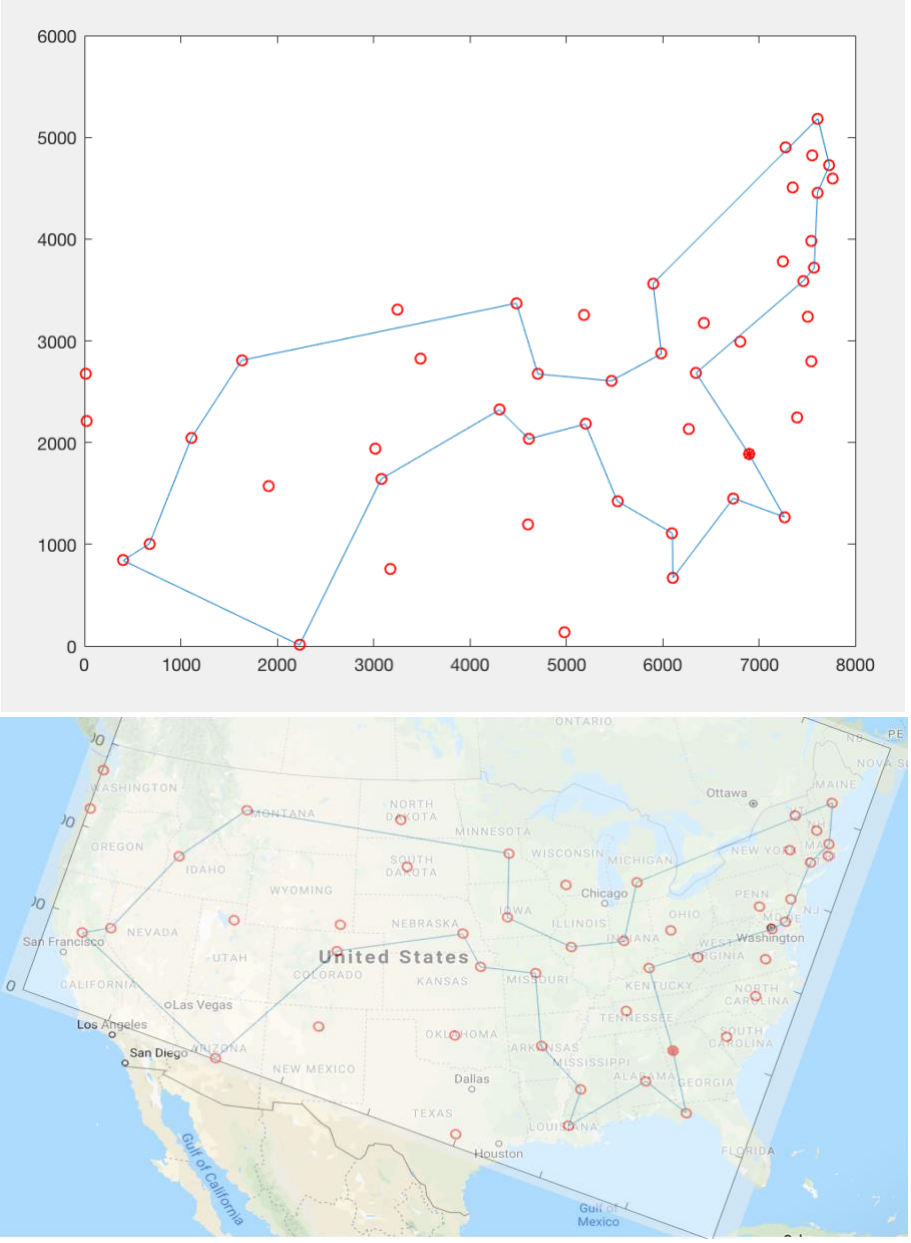
Bismarck → Saint Paul → Madison → Lansing → Columbus → Charleston → Harrisburg → Albany → Montpelier → Augusta → Concord → Boston → Providence → Hartford → Trenton → Dover → Annapolis → Richmond → Raleigh → Columbia → Tallahassee → Montgomery → Atlanta

The physical copy of code for this problem can be found in the Appendix.

9. Optimal 26-city TSP tour

We selected city index 1 to 26 to explore the 26-city TSP tour.
The data we used for the 26-city tour is as followed:
coord : [
1 6898 1885
2 2233 10
3 5530 1424
4 401 841
5 3082 1644
6 7608 4458
7 7573 3716
8 7265 1268
9 6734 1453
10 1112 2049
11 5468 2606
12 5989 2873
13 4706 2674
14 4612 2035
15 6347 2683
16 6107 669
17 7611 5184
18 7462 3590
19 7732 4723
20 5900 3561
21 4483 3369
22 6101 1110
23 5199 2182
24 1633 2809
25 4307 2322
26 675 1006]

The optimal 26-city tour distance is **24977**.
The plot generated by Matlab for the optimal 26-city TSP is as follows:



The optimal TSP tour for the 26 cities is as follows:

Atlanta → Tallahassee → Montgomery → Baton Rouge → Jackson → Little Rock → Jefferson City → Topeka → Lincoln → Denver → Phoenix → Sacramento → Carson City → Boise → Helena → Saint Paul → Des Moines → Springfield → Indianapolis → Lansing → Augusta → Boston → Hartford → Dover → Annapolis → Frankfort → Atlanta

Each group member contributed evenly Team primary responsibilities chart:	
Luffina Huang	Problem 3
Ali Mujtaba Lakdawala	Problem 1 & 2
Tess Leggio	Problem 1 & 2

The physical copy of code for these problems can be found in the Appendix.

Appendix

Problem 1

diet.colgen.V1.mos

```
model DietGen        ! Name the model

uses "mmxprs"        ! include the Xpress solver package
uses "mmodbc"        ! include package to read from Excel


NumFoods := 7146  ! declare how many foods we have
NumNutrients := 30 ! declare how many nutrients we're tracking consumption of

declarations  ! declare sets and arrays

        Foods = 1..NumFoods+1                   ! We need an extra "dummy" food as
we'll see later
        Nutrients = 1..NumNutrients

        FoodNames: array(Foods) of string       ! names of each food
        NutNames: array(Nutrients) of string    ! names of each nutrient

        Chol: array(Foods) of real              ! vector of cholesterol for each food
```

```
        Calorie: array(Foods) of real                   ! vector of calorie for each food
        Contents: array(Foods,Nutrients) of real ! matrix of food content
        Minimums: array(Nutrients) of real              ! minimum intake of each
nutrient
        Maximums: array(Nutrients) of real              ! maximum intake of each
nutrient

        Eaten: dynamic array(Foods) of mpvar    ! variables: amount of each food eaten
                                                        ! "dynamic" means
we'll be generating variables
                                                        ! instead of
defining them all up front

        minShadowPrices: array(Nutrients) of real       ! array to hold the shadow prices
of the minimum-intake constraints
        maxShadowPrices: array(Nutrients) of real       ! array to hold the shadown
prices of the maximum-intake constraints

        ColsGen: integer                ! number of columns generated in the problem so far
        BestReducedCost: real           ! what is the best reduced cost
        BestFood: integer               ! which food has the best reduced cost
        TotalCalorie: real              ! total calorie
        ReducedCost: real               ! reduced cost

end-declarations      ! end declarations section


initializations from "mmodbc.odbc:diet.xls"       ! read from Excel file

 FoodNames as "[a2:a7148]"       ! NOTE: In the Excel file, the food names
                                         ! are in cells a3:a7148, not a2:a7148.
                                         ! For some reason, Xpress makes you always
                                         ! include one row of cells above the data you
                                         ! want to read.
                                         !       You'll see the same extra row in all of
                                         ! these other things read from Excel.
 NutNames as "[b1:ae2]"
 Chol as "[ac2:ac7148]"
 Calorie as "[d2:d7148]"
 Contents as "[b2:ae7148]"
 Minimums as "[b7149:ae7150]"
 Maximums as "[b7151:ae7152]"
```

```
end-initializations


! We need to start with at least one column or else the problem
! will start off infeasible.  We'll use a "dummy" food as our
! initial column.  The dummy food will have a very high cost (1,000,000)
! and will contain all of the minimums necessary - that way we can be sure
! that by itself it will a be feasible solution.

forall(k in Nutrients)
        Contents(NumFoods+1,k) := Minimums(k)        ! create contents of "dummy" food

FoodNames(NumFoods+1) := "DUMMY FOOD"
Chol(NumFoods+1) := 1000000


! Create the new (dummy food) variable.  Because the variables were
! declared as "dynamic" none of the other variables exist in the problem yet.

create(Eaten(NumFoods+1))

! Create minimum and maximum intake constraints.
! If all the variables existed, we could say
!       sum(j in Foods) Contents(j,k)*Eaten(j) <= Maximums(k).
! However, not all the variables Eaten(j) are created yet, so as new ones
! get created we'll have to add them to the constraints.
!   Below, we add the one variable we've created so far (the dummy variable) to
! each constraint.
!   These constraints need to be named ("EatMin(k)" and "EatMax(k)") because
! we'll need to get their shadow prices later, and for us to be able to ask
! Xpress for shadow prices we need a way to refer to the constraints.

forall(k in Nutrients) do
        EatMin(k) := Contents(NumFoods+1,k)*Eaten(NumFoods+1) >= Minimums(k)
        EatMax(k) := Contents(NumFoods+1,k)*Eaten(NumFoods+1) <= Maximums(k)
end-do


! Create objective function.  Just like the intake constraints, we'll have
! to add variables to it as they get generated.
```

```
TotalChol := Chol(NumFoods+1)*Eaten(NumFoods+1)


!!!!!!!!!!!! Begin column generation !!!!!!!!!!!!!!!!!!

ColsGen := 0
repeat
        writeln(ColsGen)
        ! solve the problem using the variables that have been generated so far

        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
        minimize(TotalChol)
        writeln("Total Cholesterol = ",getobjval)



        ! get the shadow prices (dual prices) cB*Binverse
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!

        forall(k in Nutrients) do
                minShadowPrices(k) := getdual(EatMin(k))
                maxShadowPrices(k) := getdual(EatMax(k))
        end-do



        ! Find the food with the best reduced cost.
        !   As we search through each food, we'll keep track of the best reduced
        ! cost found so far in BestReducedCost, and whatever food has that best
        ! reduced cost will be stored in BestFood.

        BestReducedCost := 0
        BestFood := 0
        forall(j in Foods) do
                        ! if food j has a better (lower) reduced cost than the previous best,
                        ! then store its reduced cost in BestReducedCost and let BestFood be j.
                        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
                        !reduced cost = cj - ctransposeB * Binv * Aj
                        ReducedCost:=Chol(j)
                        forall(k in Nutrients) do
                                ReducedCost:=(ReducedCost - minShadowPrices(k) *Contents(j,
k) - maxShadowPrices(k) *Contents(j, k))
                        end-do
```

```
                    if ReducedCost < BestReducedCost then
                        BestReducedCost := ReducedCost
                        BestFood := j
                    end-if

        end-do

        writeln('minShadowPrices = ',minShadowPrices)
        writeln('maxShadowPrices = ',maxShadowPrices)

        ! If there's a variable with a good reduced cost, then
        !       create the variable
        !       update the number of columns generated
        !       update the constraints for each nutrient by adding the contents of the new food
        !       update the objective function by adding the cost of the new food

        if BestReducedCost < -0.000001 then
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!)
            writeln("Adding food ",BestFood," (",FoodNames(BestFood),") : reduced cost is
",BestReducedCost)

            !create the variable
            create(Eaten(BestFood))

            !update number of columns generated
            ColsGen += 1

            !update the constraints for each nutrient by adding the contents of the new food
            forall(k in Nutrients) do
                EatMin(k) += Contents(BestFood,k)*Eaten(BestFood)
                EatMax(k) += Contents(BestFood,k)*Eaten(BestFood)
            end-do

            !update the objective function by adding the cost of the new food
            TotalChol += Chol(BestFood)*Eaten(BestFood)


        end-if


! Here and above ("if BestReducedCost < -0.000001") we use 0.000001 instead of 0.
! The reason is that floating point roundoff errors might make the computer
```

```
! calculate a reduced cost to be something like 0.000000000001 instead of 0.  This
! is normal -- most computer software has this sort of problem, and they use the
! same sort of checks to catch it.  Specifically, any value of x for which, say,
! -0.000000001 < x < 0.000000001 will be treated as 0.  For this application, we
! can safely use 0.000001 instead.
!   This is really a CS/computer architecture issue; if you're interested in
! learning more about what causes it, let me know and I'd be happy to explain more.

until BestReducedCost >= -0.000001      ! continue until no variable has a good reduced cost


!!!!!!!!!!!! End column generation !!!!!!!!!!!!!!!!!!

! The solution can be long, so print a summary: only say what we eat.
TotalCalorie := 0
writeln("\n===============\n")
forall(j in Foods)
        if (getsol(Eaten(j)) > 0) then
                writeln(FoodNames(j)," = ",getsol(Eaten(j)))
                !writeln('Cholesterol = ', Chol(j))
                !!! you need to compute the total calorie for the chosen food !!!
                !!!!!!!!! write your code here !!!!!!!!!!
                TotalCalorie += getsol(Eaten(j)*Calorie(j))



        end-if
writeln("\nTotal Cholesterol = ",getobjval)
writeln("\nTotal Calorie = ", TotalCalorie)
writeln("\nTotal: ",ColsGen," columns generated.")

end-model
```

Problem 1

diet.colgen.V2.mos

```
model DietGen        ! Name the model

uses "mmxprs"        ! include the Xpress solver package
uses "mmodbc"        ! include package to read from Excel


NumFoods := 7146  ! declare how many foods we have
NumNutrients := 30 ! declare how many nutrients we're tracking consumption of

declarations  ! declare sets and arrays

        Foods = 1..NumFoods+1                   ! We need an extra "dummy" food as
we'll see later
        Nutrients = 1..NumNutrients

        FoodNames: array(Foods) of string       ! names of each food
        NutNames: array(Nutrients) of string    ! names of each nutrient

        Chol: array(Foods) of real              ! vector of cholesterol for each food
        Calorie: array(Foods) of real                   ! vector of calorie for each food
        Contents: array(Foods,Nutrients) of real ! matrix of food content
        Minimums: array(Nutrients) of real              ! minimum intake of each
nutrient
        Maximums: array(Nutrients) of real              ! maximum intake of each
nutrient

        Eaten: dynamic array(Foods) of mpvar    ! variables: amount of each food eaten
                                                                ! "dynamic" means
we'll be generating variables
                                                                ! instead of
defining them all up front

        minShadowPrices: array(Nutrients) of real       ! array to hold the shadow prices
of the minimum-intake constraints
        maxShadowPrices: array(Nutrients) of real       ! array to hold the shadown
prices of the maximum-intake constraints

        ColsGen: integer                ! number of columns generated in the problem so far
        BestReducedCost: real           ! what is the best reduced cost
```

```
        BestFood: integer                   ! which food has the best reduced cost
        TotalCalorie: real                  ! total calorie
        ReducedCost: real                   ! reduced cost
        TotalContent : array(Nutrients) of real                 !array to hold the total nutrient
contents

end-declarations      ! end declarations section


initializations from "mmodbc.odbc:diet.V2.xls"    ! read from Excel file

 FoodNames as "[a2:a7148]"        ! NOTE: In the Excel file, the food names
                                          ! are in cells a3:a7148, not a2:a7148.
                                          ! For some reason, Xpress makes you always
                                          ! include one row of cells above the data you
                                          ! want to read.
                                          !         You'll see the same extra row in all of
                                          ! these other things read from Excel.
 NutNames as "[b1:ae2]"
 Chol as "[ac2:ac7148]"
 Calorie as "[d2:d7148]"
 Contents as "[b2:ae7148]"
 Minimums as "[b7149:ae7150]"
 Maximums as "[b7151:ae7152]"

end-initializations


! We need to start with at least one column or else the problem
! will start off infeasible.  We'll use a "dummy" food as our
! initial column.  The dummy food will have a very high cost (1,000,000)
! and will contain all of the minimums necessary - that way we can be sure
! that by itself it will a be feasible solution.

forall(k in Nutrients)
        Contents(NumFoods+1,k) := Minimums(k)        ! create contents of "dummy" food

FoodNames(NumFoods+1) := "DUMMY FOOD"
Chol(NumFoods+1) := 1000000


! Create the new (dummy food) variable.  Because the variables were
```

```
! declared as "dynamic" none of the other variables exist in the problem yet.

create(Eaten(NumFoods+1))

! Create minimum and maximum intake constraints.
! If all the variables existed, we could say
!       sum(j in Foods) Contents(j,k)*Eaten(j) <= Maximums(k).
! However, not all the variables Eaten(j) are created yet, so as new ones
! get created we'll have to add them to the constraints.
!   Below, we add the one variable we've created so far (the dummy variable) to
! each constraint.
!   These constraints need to be named ("EatMin(k)" and "EatMax(k)") because
! we'll need to get their shadow prices later, and for us to be able to ask
! Xpress for shadow prices we need a way to refer to the constraints.

forall(k in Nutrients) do
        EatMin(k) := Contents(NumFoods+1,k)*Eaten(NumFoods+1) >= Minimums(k)
        EatMax(k) := Contents(NumFoods+1,k)*Eaten(NumFoods+1) <= Maximums(k)
end-do


! Create objective function.  Just like the intake constraints, we'll have
! to add variables to it as they get generated.

TotalChol := Chol(NumFoods+1)*Eaten(NumFoods+1)


!!!!!!!!!!!! Begin column generation !!!!!!!!!!!!!!!!!!

ColsGen := 0
repeat
        writeln(ColsGen)
        ! solve the problem using the variables that have been generated so far

        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
        minimize(TotalChol)
        writeln("Total Cholesterol = ",getobjval)


        ! get the shadow prices (dual prices) cB*Binverse
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
```

```
forall(k in Nutrients) do
        minShadowPrices(k) := getdual(EatMin(k))
        maxShadowPrices(k) := getdual(EatMax(k))
end-do


! Find the food with the best reduced cost.
!   As we search through each food, we'll keep track of the best reduced
! cost found so far in BestReducedCost, and whatever food has that best
! reduced cost will be stored in BestFood.

BestReducedCost := 0
BestFood := 0
forall(j in Foods) do
                ! if food j has a better (lower) reduced cost than the previous best,
                ! then store its reduced cost in BestReducedCost and let BestFood be j.
                !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
                !reduced cost = cj - ctransposeB * Binv * Aj
                ReducedCost := Chol(j)
                forall(k in Nutrients)do
                        ReducedCost := (ReducedCost - minShadowPrices(k) *Contents(j,
k) - maxShadowPrices(k) *Contents(j, k))
                end-do




                if ReducedCost < BestReducedCost then
                        BestReducedCost := ReducedCost
                        BestFood := j
                end-if

end-do

writeln('minShadowPrices = ',minShadowPrices)
writeln('maxShadowPrices = ',maxShadowPrices)

! If there's a variable with a good reduced cost, then
!       create the variable
!       update the number of columns generated
!       update the constraints for each nutrient by adding the contents of the new food
!       update the objective function by adding the cost of the new food
```

```
        if BestReducedCost < -0.000001 then
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!)
                writeln("Adding food ",BestFood," (",FoodNames(BestFood),") : reduced cost is
",BestReducedCost)

                !create the variable
                create(Eaten(BestFood))

                !update number of columns generated
                ColsGen += 1

                !update the constraints for each nutrient by adding the contents of the new food
                forall(k in Nutrients) do
                        EatMin(k) += Contents(BestFood,k)*Eaten(BestFood)
                        EatMax(k) += Contents(BestFood,k)*Eaten(BestFood)
                end-do

                !update the objective function by adding the cost of the new food
                TotalChol += Chol(BestFood)*Eaten(BestFood)


        end-if


! Here and above ("if BestReducedCost < -0.000001") we use 0.000001 instead of 0.
! The reason is that floating point roundoff errors might make the computer
! calculate a reduced cost to be something like 0.000000000001 instead of 0.  This
! is normal -- most computer software has this sort of problem, and they use the
! same sort of checks to catch it.  Specifically, any value of x for which, say,
! -0.000000001 < x < 0.000000001 will be treated as 0.  For this application, we
! can safely use 0.000001 instead.
!   This is really a CS/computer architecture issue; if you're interested in
! learning more about what causes it, let me know and I'd be happy to explain more.

until BestReducedCost >= -0.000001      ! continue until no variable has a good reduced cost


!!!!!!!!!!!! End column generation !!!!!!!!!!!!!!!!!!

! The solution can be long, so print a summary: only say what we eat.
TotalCalorie := 0
```

```
writeln("\n===============\n")
forall(j in Foods)
	if (getsol(Eaten(j)) > 0) then
		writeln(FoodNames(j)," = ",getsol(Eaten(j)))
		!writeln('Cholesterol = ', Chol(j))
		!!! you need to compute the total calorie for the chosen food !!!
		!!!!!!!!! write your code here !!!!!!!!!!
		TotalCalorie += getsol(Eaten(j)*Calorie(j))
		forall(nut in 1..NumNutrients)
			TotalContent(nut) += getsol(Eaten(j)*Contents(j,nut))
	end-if
writeln("\nTotal Cholesterol = ",getobjval)
writeln("\nTotal Calorie = ", TotalCalorie)
writeln("\nTotal Nutrients:")
forall(nut in 1..NumNutrients)
	writeln(NutNames(nut)," = ", TotalContent(nut))
writeln("\nTotal: ",ColsGen," columns generated.")

end-model
```

Problem 2.1

cs.Kantorovich.V1

```
model Kantorovich
uses "mmxprs"; !gain access to the Xpress-Optimizer solver

declarations
        W : integer                                     ! the width of the large rolls
        m = 4                           ! the number of different small rolls demanded
        K = 60                      ! number of available large rolls
        kRange = 1 .. K                         ! the index of large rolls
        iRange = 1 .. m                             ! the index of small rolls
        w : array(iRange) of integer                ! widths of the small rolls
        b : array(iRange) of integer                ! demand of the small rolls
        y : array(kRange) of mpvar                      ! y(k)=1 if roll k is cut, 0 ow
        x : array(kRange, iRange) of mpvar  ! x(k,i) number of times an item of width wi is cut
on roll k

end-declarations

initializations from 'kant1.dat'
        W w b
end-initializations



! objective function
!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!
objective := sum(k in kRange)(y(k))

! constraints-1
!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!
forall(i in iRange)(sum(k in kRange)(x(k,i)) = b(i))

! constraints-2
!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!
forall(k in kRange)(sum(i in iRange)(w(i)*x(k,i))  <=  (y(k)*W))



! bounds on variables and integrality constraints
! note how to constrain a variable to be integral. you will need to use it in the part 2 of this
problem.
```

```
forall (k in kRange, i in iRange) do
        x(k,i)>=0
        y(k)>=0
        y(k)<=1
        y(k) is_integer
        x(k,i) is_integer
end-do

! solve the problem
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!
minimize(objective)



! print out the objective value and the nonzero optimal variables
writeln("obj=",getobjval)
forall (k in kRange) do
        if (getsol(y(k))>0) then
                writeln("y(",k,")=", getsol(y(k)))
                forall (i in iRange) do
                        if (getsol(x(k,i))>0) then
                                writeln("x(",k,",",i,")=",getsol(x(k,i)))
                        end-if
                end-do
        end-if

end-do


end-model
```

Problem 2.2
cs.colgen.V2.2

```
model CuttingStockColGen          ! Name the model

uses "mmxprs"        ! include the Xpress solver package

! knapsack function solves the knapsack problem. You will need to write the content of the
function in the later part of the code
! here is the initial declaration of the function
forward function knapsack(y: array(range) of real, W: integer, w: array(range) of integer) : real

m := 8                                  ! the number of different small rolls

declarations                                                              ! declare sets,
arrays, constraints, and variables

        dRange = 1 .. m                         ! the index range of demand
        pRange : range                          ! the index range of patterns
        W : integer                                                       ! width of
the large roll
        w : array(dRange) of integer            ! widths of the small rolls
        b : array(dRange) of integer            ! demands of the small rolls

        initialPatterns: array(dRange, pRange) of integer  ! set of initial patterns to start the
column generation
        newPattern: array(dRange) of real          ! the newly generated pattern by the
knapsack problem
        numPatterns: integer                    ! number of patterns in the restricted master
problem
        Z : real                                ! knapsack objective value
        dualVar : array(dRange) of real         ! dual variables for the demand constraints
        DemandConstr: array(dRange) of linctr          ! Demand constraints
        numRolls : linctr                       ! objective value of cutting stock restricted
master problem
        x : dynamic array(pRange) of mpvar          ! decision variable: number of rolls cut in
each pattern
        waste : real                                                      !captures
amount of total waste

end-declarations                                                          ! end
declarations section
```

```
! initialization from data file
! test on different data files cs1.dat, cs2.dat, cs3.dat, and report results for each instance
!!!!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!

initializations from 'cs1.dat'
!initializations from 'cs2.dat'
!initializations from 'cs3.dat'
!initializations from 'kant1.dat'
        W w b
end-initializations

! Column generation needs to start from a subset of initial columns ir patterns
! As discussed in class, you can easily find a subset of m initial patterns which generates a
basic feasible solution
! Choose the subset of initial patterns and save them in the array initialPatterns
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!

forall(i in dRange) do
        initialPatterns(i,i) := 1
end-do

! set the number of initial patterns
! this number will need to be updated everytime a new pattern is generated
numPatterns := m
waste := 0

! create the corresponding x variables and set the nonnegativity constraints
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!
forall(p in dRange) do
        create(x(p))
        x(p) >= 0
end-do



! Create demand constraints using the initialPatterns and the variables x created above.
! These constraints need to be named DemandConstr(i) because we'll need to get their
shadow prices (i.e. dual variables) later,
! and for us to be able to ask Xpress for shadow prices we need a way to refer to the
constraints.
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
forall(i in dRange) do
        DemandConstr(i) := initialPatterns(i,i)*x(i) = b(i)
end-do


! Create objective function. The objective needs to be named numRolls
! Just like the demand constraints, we'll need to add variables to it as they get generated.
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
numRolls := sum(i in dRange)(x(i))


!!!!!!!!!!!! Begin column generation !!!!!!!!!!!!!!!!!!
repeat

        ! Solve the problem using the variables that have been generated so far
        ! Notice: you may need to use the XPRS_LIN parameter in the minimize command
        minimize(XPRS_LIN, numRolls)



        ! Print the objective value and the solution x
        writeln("Number of Large Rolls = ", getobjval)
        forall (j in 1..numPatterns)
                writeln("x(",j,")=",getsol(x(j)))

        ! Get the dual variables (shadow prices)
        ! Hint: Check out command "getdual" in the solution manual
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
        forall(i in dRange) do
                dualVar(i) := getdual(DemandConstr(i))
        end-do

        ! Call the knapsack function, set Z to be the knapsack objective value
        ! for example: Z=knapsack(y,W,z), but you need to figure out what y is.
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
        Z := knapsack(dualVar,W,w)


        ! If the reduced cost is less than zero (here we use -0.000001 to approximate 0 in order
to avoid numerical error)
        if 1 - Z <= -0.000001 then
```

```
                ! Print out the new pattern found by knapsack problem
                !!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!!

                writeln("Z = ",newPattern)


                !       update the number of patterns generated
                !       create the new variable
                !       update the constraints for each demand by adding the new pattern
generated by knapsack
                !       update the objective function by adding the new pattern
                !!!!!!!!!!!!!!! write your code here  !!!!!!!!!!!!!!!

                !       update the number of patterns generated
                numPatterns += 1

                !       create the new variable
                create(x(numPatterns))
                x(numPatterns) >= 0



                !       update the constraints for each demand by adding the new pattern
generated by knapsack
                forall(i in dRange) do
                        initialPatterns(i,numPatterns) := integer(newPattern(i))
                        DemandConstr(i) += initialPatterns(i,numPatterns)*x(numPatterns)
                end-do

                !       update the objective function by adding the new pattern
                numRolls += x(numPatterns)


        end-if


until 1 - Z >= -0.000001      ! continue until no variable has a good reduced cost
!!!!!!!!!!!! End column generation !!!!!!!!!!!!!!!!!!

writeln("\n===============\n")
! Print a summary of the final LP solution: selected patterns, number of rolls cut using each
selected pattern, total number of rolls
```

```
!!!!!!!!!! wrtie your code here !!!!!!!!!!!!!!!!!!

minimize(XPRS_LIN, numRolls)

writeln("\nLP:")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', (getsol(x(i))),' small rolls')
                waste += W*getsol(x(i)) - sum(j in dRange) getsol(x(i))*w(j)*initialPatterns(j,i)
        end-if
end-do

writeln('total number of big rolls = ', (getobjval))
writeln('total waste = ',waste)




! We also want to get integer solutions. There are at least two approaches to do this.
! The first approach is to round up the LP solution.
! The second approach is to re-solve the master problem as an integer program using all the
generated patterns


! Implement the first approach: rounding the LP solution and print out a summary ofthe
solution: number of rolls cut using each selected pattern, total number of rolls
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!
waste := 0
writeln("\nRound Up:")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', ceil(getsol(x(i))),' small rolls')
                waste += W*ceil(getsol(x(i))) - sum(j in dRange)
ceil(getsol(x(i)))*w(j)*initialPatterns(j,i)
        end-if
end-do

writeln('total number of big rolls = ', sum(i in 1..numPatterns)ceil(getsol(x(i))))
writeln('total waste = ',waste)
```

```
! Implement the second approach: first, constrain all variables to be integer (example: x(j) is_integer), then solve the master problem using all generated patterns
! You also need to print out a summary of the solution: number of rolls cut using each selected pattern, total number of rolls
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!
forall(i in 1..numPatterns)do
        x(i) is_integer
end-do


minimize(numRolls)

waste := 0
writeln("\nIP:")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', getsol(x(i)),' small rolls')
                waste += W*getsol(x(i)) - sum(j in dRange) getsol(x(i))*w(j)*initialPatterns(j,i)
        end-if
end-do

writeln('total number of big rolls = ', getobjval)
writeln('total waste = ',waste)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!! Knapsack Problem !!!!!!!!!!!!!!!!!!!!
!! implement a function that solves the knapsack problem   !!
!! Inputs: objective coefficient array y                   !!
!!        width of the large roll W                      !!
!!        widths of the small rolls array w              !!
!! Return: optimal objective function value               !!
!!        optimal solution (i.e. the new pattern)        !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
function knapsack(y: array(range) of real, W: integer, w: array(range) of integer) : real

        ! declare decision variable
        ! form constraints and objective
        ! do not forget the integrality constraints on the variables
        ! solve the problem
        !!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!
```

```
	! declare decision variable
	declarations
		len_y = 1..getsize(y)
		a : array(len_y) of mpvar
	end-declarations


	! form objective
	objective := sum(i in len_y)y(i) * a(i)


	! form constraints
	sum(i in len_y)(w(i)*a(i)) <= W
	forall(i in len_y) a(i) >= 0
	forall(i in len_y) a(i) is_integer

	! solve the problem
	maximize(objective)

	! return the optimal objective value
	! the keyword returned is reserved for this purpose by Xpress
	returned := getobjval

	! return the optimal solution, save it in the array newPattern defined in the beginning of
this model file
	!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!
	forall(i in len_y) do
		newPattern(i) := getsol(a(i))
	end-do


end-function


end-model
```

Problem 2.2

cs.colgen.V2.2_greaterequal

```
model CuttingStockColGen		! Name the model

uses "mmxprs"		! include the Xpress solver package
```

```
! knapsack function solves the knapsack problem. You will need to write the content of the
function in the later part of the code
! here is the initial declaration of the function
forward function knapsack(y: array(range) of real, W: integer, w: array(range) of integer) : real

m := 8                                  ! the number of different small rolls

declarations                                                              ! declare sets,
arrays, constraints, and variables

        dRange = 1 .. m                 ! the index range of demand
        pRange : range                  ! the index range of patterns
        W : integer                                                       ! width of
the large roll
        w : array(dRange) of integer    ! widths of the small rolls
        b : array(dRange) of integer    ! demands of the small rolls

        initialPatterns: array(dRange, pRange) of integer  ! set of initial patterns to start the
column generation
        newPattern: array(dRange) of real       ! the newly generated pattern by the
knapsack problem
        numPatterns: integer            ! number of patterns in the restricted master
problem
        Z : real                        ! knapsack objective value
        dualVar : array(dRange) of real         ! dual variables for the demand constraints
        DemandConstr: array(dRange) of linctr   ! Demand constraints
        numRolls : linctr               ! objective value of cutting stock restricted
master problem
        x : dynamic array(pRange) of mpvar      ! decision variable: number of rolls cut in
each pattern
        waste : real                                                      !captures
amount of total waste

end-declarations                                                          ! end
declarations section


! initialization from data file
! test on different data files cs1.dat, cs2.dat, cs3.dat, and report results for each instance
!!!!!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!

!initializations from 'cs1.dat'
```

```
!initializations from 'cs2.dat'
!initializations from 'cs3.dat'
initializations from 'kant1.dat'
        W w b
end-initializations

! Column generation needs to start from a subset of initial columns ir patterns
! As discussed in class, you can easily find a subset of m initial patterns which generates a
basic feasible solution
! Choose the subset of initial patterns and save them in the array initialPatterns
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!

forall(i in dRange) do
        initialPatterns(i,i) := 1
end-do

! set the number of initial patterns
! this number will need to be updated everytime a new pattern is generated
numPatterns := m
waste := 0

! create the corresponding x variables and set the nonnegativity constraints
!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!
forall(p in dRange) do
        create(x(p))
        x(p) >= 0
end-do


! Create demand constraints using the initialPatterns and the variables x created above.
! These constraints need to be named DemandConstr(i) because we'll need to get their
shadow prices (i.e. dual variables) later,
! and for us to be able to ask Xpress for shadow prices we need a way to refer to the
constraints.
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!
forall(i in dRange) do
        DemandConstr(i) := initialPatterns(i,i)*x(i) >= b(i)
end-do


! Create objective function. The objective needs to be named numRolls
! Just like the demand constraints, we'll need to add variables to it as they get generated.
```

```
!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!!
numRolls := sum(i in dRange)(x(i))


!!!!!!!!!!! Begin column generation !!!!!!!!!!!!!!!!!
repeat

        ! Solve the problem using the variables that have been generated so far
        ! Notice: you may need to use the XPRS_LIN parameter in the minimize command
        minimize(XPRS_LIN, numRolls)


        ! Print the objective value and the solution x
        writeln("Number of Large Rolls = ", getobjval)
        forall (j in 1..numPatterns)
                writeln("x(",j,")=",getsol(x(j)))

        ! Get the dual variables (shadow prices)
        ! Hint: Check out command "getdual" in the solution manual
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!
        forall(i in dRange) do
                dualVar(i) := getdual(DemandConstr(i))
        end-do

        ! Call the knapsack function, set Z to be the knapsack objective value
        ! for example: Z=knapsack(y,W,z), but you need to figure out what y is.
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!

        Z := knapsack(dualVar,W,w)


        ! If the reduced cost is less than zero (here we use -0.000001 to approximate 0 in order
to avoid numerical error)
        if 1 - Z <= -0.000001 then

                ! Print out the new pattern found by knapsack problem
                !!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!

                writeln("Z = ",newPattern)


                !       update the number of patterns generated
```

```
                !       create the new variable
                !       update the constraints for each demand by adding the new pattern
generated by knapsack
                !       update the objective function by adding the new pattern
                !!!!!!!!!!!!!!! write your code here  !!!!!!!!!!!!!!

                !       update the number of patterns generated
                numPatterns += 1

                !       create the new variable
                create(x(numPatterns))
                x(numPatterns) >= 0



                !       update the constraints for each demand by adding the new pattern
generated by knapsack
                forall(i in dRange) do
                        initialPatterns(i,numPatterns) := integer(newPattern(i))
                        DemandConstr(i) += initialPatterns(i,numPatterns)*x(numPatterns)
                end-do

                !       update the objective function by adding the new pattern
                numRolls += x(numPatterns)


        end-if


until 1 - Z >= -0.000001      ! continue until no variable has a good reduced cost
!!!!!!!!!!!! End column generation !!!!!!!!!!!!!!!!!!

writeln("\n===============\n")
! Print a summary of the final LP solution: selected patterns, number of rolls cut using each
selected pattern, total number of rolls
!!!!!!!!!!! wrtie your code here !!!!!!!!!!!!!!!!!!

minimize(XPRS_LIN, numRolls)

writeln("\nLP:")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
```

```
            writeln('selected pattern: x(',i,') = ', (getsol(x(i))),' small rolls')
            waste += W*getsol(x(i)) - sum(j in dRange) getsol(x(i))*w(j)*initialPatterns(j,i)
        end-if
end-do

writeln('total number of big rolls = ', (getobjval))
writeln('total waste = ',waste)



! We also want to get integer solutions. There are at least two approaches to do this.
! The first approach is to round up the LP solution.
! The second approach is to re-solve the master problem as an integer program using all the
generated patterns


! Implement the first approach: rounding the LP solution and print out a summary ofthe
solution: number of rolls cut using each selected pattern, total number of rolls
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!
waste := 0
writeln("\nRound Up:")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
            writeln('selected pattern: x(',i,') = ', ceil(getsol(x(i))),' small rolls')
            waste += W*ceil(getsol(x(i))) - sum(j in dRange)
ceil(getsol(x(i)))*w(j)*initialPatterns(j,i)
        end-if
end-do

writeln('total number of big rolls = ', sum(i in 1..numPatterns)ceil(getsol(x(i))))
writeln('total waste = ',waste)


! Implement the second approach: first, constrain all variables to be integer (example: x(j)
is_integer), then solve the master problem using all generated patterns
! You also need to print out a summary of the solution: number of rolls cut using each
selected pattern, total number of rolls
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!
forall(i in 1..numPatterns)do
        x(i) is_integer
end-do
```

```
minimize(numRolls)

waste := 0
writeln("\nIP:")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', getsol(x(i)),' small rolls')
                waste += W*getsol(x(i)) - sum(j in dRange) getsol(x(i))*w(j)*initialPatterns(j,i)
        end-if
end-do

writeln('total number of big rolls = ', getobjval)
writeln('total waste = ',waste)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!! Knapsack Problem !!!!!!!!!!!!!!!!!!!!
!! implement a function that solves the knapsack problem   !!
!! Inputs: objective coefficient array y                   !!
!!        width of the large roll W                      !!
!!        widths of the small rolls array w              !!
!! Return: optimal objective function value               !!
!!        optimal solution (i.e. the new pattern)        !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
function knapsack(y: array(range) of real, W: integer, w: array(range) of integer) : real

        ! declare decision variable
        ! form constraints and objective
        ! do not forget the integrality constraints on the variables
        ! solve the problem
        !!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!

        ! declare decision variable
        declarations
                len_y = 1..getsize(y)
                a : array(len_y) of mpvar
        end-declarations


        ! form objective
```

```
        objective := sum(i in len_y)y(i) * a(i)


        ! form constraints
        sum(i in len_y)(w(i)*a(i)) <= W
        forall(i in len_y) a(i) >= 0
        forall(i in len_y) a(i) is_integer

        ! solve the problem
        maximize(objective)

        ! return the optimal objective value
        ! the keyword returned is reserved for this purpose by Xpress
        returned := getobjval

        ! return the optimal solution, save it in the array newPattern defined in the beginning of
this model file
        !!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!
        forall(i in len_y) do
                newPattern(i) := getsol(a(i))
        end-do

end-function



end-model
```

Xpress Code for 2.3 (Including greater equal to constraint)

```
model CuttingStockColGen          ! Name the model

uses "mmxprs"        ! include the Xpress solver package

! Pricing function solves the Pricing problem. You will need to write the content of the function
in the later part of the code
! here is the initial declaration of the function
forward function Pricing(y: array(range) of real, W: integer, w: array(range) of integer) : real

m := 8                                  ! the number of different small rolls

declarations                                                                    ! declare sets,
arrays, constraints, and variables
```

```
        dRange = 1 .. m                          ! the index range of demand
        pRange : range                           ! the index range of patterns
        W : integer                                                    ! width of
the large roll
        w : array(dRange) of integer             ! widths of the small rolls
        b : array(dRange) of integer             ! demands of the small rolls

        initialPatterns: array(dRange, pRange) of integer  ! set of initial patterns to start the
column generation
        newPattern: array(dRange) of real        ! the newly generated pattern by the
Pricing problem
        numPatterns: integer                     ! number of patterns in the restricted master
problem
        Z : real                                 ! Pricing objective value
        dualVar : array(dRange) of real          ! dual variables for the demand constraints
        DemandConstr: array(dRange) of linctr    ! Demand constraints
        waste : linctr                           ! objective value of cutting stock restricted master
problem
        x : dynamic array(pRange) of mpvar       ! decision variable: number of rolls cut in
each pattern

end-declarations                                                   ! end
declarations section


! initialization from data file
! test on different data files cs1.dat, cs2.dat, cs3.dat, and report results for each instance
!!!!!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!
initializations from 'cs1.dat'
!initializations from 'cs2.dat'
!initializations from 'cs3.dat'
!initializations from 'kant1.dat'
        W w b
end-initializations

! Column generation needs to start from a subset of initial columns ir patterns
! As discussed in class, you can easily find a subset of m initial patterns which generates a
basic feasible solution
! Choose the subset of initial patterns and save them in the array initialPatterns
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!

forall(i in dRange) do
```

```
        initialPatterns(i,i) := 1
end-do

! set the number of initial patterns
! this number will need to be updated everytime a new pattern is generated
numPatterns := m

! create the corresponding x variables and set the nonnegativity constraints
!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!
forall(p in dRange) do
        create(x(p))
        x(p) >= 0
end-do



! Create demand constraints using the initialPatterns and the variables x created above.
! These constraints need to be named DemandConstr(i) because we'll need to get their
shadow prices (i.e. dual variables) later,
! and for us to be able to ask Xpress for shadow prices we need a way to refer to the
constraints.
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!
forall(i in dRange) do
        DemandConstr(i) := sum(k in 1..numPatterns) initialPatterns(i,k)*x(k) >= b(i)
end-do
!sum(i in dRange)(x(i)*w(i)) <= W



! Create objective function. The objective needs to be named numRolls
! Just like the demand constraints, we'll need to add variables to it as they get generated.
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!!!!!!
waste := 0.0
forall(k in 1..numPatterns) do
waste += W*x(k) - sum(i in dRange) x(k)*w(i)*initialPatterns(i,k)
end-do

!!!!!!!!!!!! Begin column generation !!!!!!!!!!!!!!!!!!
repeat

        ! Solve the problem using the variables that have been generated so far
        ! Notice: you may need to use the XPRS_LIN parameter in the minimize command
        minimize(XPRS_LIN, waste)
```

```
! Print the objective value and the solution x
writeln("Number of Large Rolls = ", getobjval)
forall (j in 1..numPatterns)
        writeln("x(",j,")=",getsol(x(j)))

! Get the dual variables (shadow prices)
! Hint: Check out command "getdual" in the solution manual
!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!
forall(i in dRange) do
        dualVar(i) := getdual(DemandConstr(i))
end-do

! Call the Pricing function, set Z to be the Pricing objective value
! for example: Z=Pricing(y,W,z), but you need to figure out what y is.
!!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!

Z := Pricing(dualVar,W,w)
writeln("Z = ",Z)

! If the reduced cost is less than zero (here we use -0.000001 to approximate 0 in order
to avoid numerical error)
if  Z <= -0.000001 then

        ! Print out the new pattern found by Pricing problem
        !!!!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!!!!

        writeln("NEW PATTERN = ",newPattern)


        !       update the number of patterns generated
        !       create the new variable
        !       update the constraints for each demand by adding the new pattern
generated by Pricing
        !       update the objective function by adding the new pattern
        !!!!!!!!!!!!!! write your code here  !!!!!!!!!!!!!!

        !       update the number of patterns generated
        numPatterns += 1

        !       create the new variable
        create(x(numPatterns))
```

```
                x(numPatterns) >= 0



                !       update the constraints for each demand by adding the new pattern
generated by Pricing
                forall(i in dRange) do
                        initialPatterns(i,numPatterns) := integer(newPattern(i))
                        DemandConstr(i) += initialPatterns(i,numPatterns)*x(numPatterns)
                end-do

                !       update the objective function by adding the new pattern
                waste += x(numPatterns) * (W- sum(i in dRange)
w(i)*initialPatterns(i,numPatterns))


        end-if


until   Z >= -0.000001          ! continue until no variable has a good reduced cost
!!!!!!!!!!! End column generation !!!!!!!!!!!!!!!!!

writeln("\n===============\n")
! Print a summary of the final LP solution: selected patterns, number of rolls cut using each
selected pattern, total number of rolls
!!!!!!!!!! wrtie your code here !!!!!!!!!!!!!!!!!

minimize(XPRS_LIN, waste)

writeln("")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', (getsol(x(i))),' rolls')
                forall(j in 1..m)
                        writeln('pattern =', (initialPatterns(j,i)))
        end-if
end-do

writeln('total waste = ', (getobjval))
```

```
! We also want to get integer solutions. There are at least two approaches to do this.
! The first approach is to round up the LP solution.
! The second approach is to re-solve the master problem as an integer program using all the
generated patterns


! Implement the first approach: rounding the LP solution and print out a summary ofthe
solution: number of rolls cut using each selected pattern, total number of rolls
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!
writeln("")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', ceil(getsol(x(i))),' rolls')
        end-if
end-do

writeln('total waste = ', getobjval)


! Implement the second approach: first, constrain all variables to be integer (example: x(j)
is_integer), then solve the master problem using all generated patterns
! You also need to print out a summary of the solution: number of rolls cut using each
selected pattern, total number of rolls
!!!!!!!!!!!!!!!!!! Write your code here !!!!!!!!!!!!!!!!!!!!!!!!
forall(i in 1..numPatterns)do
        x(i) is_integer
end-do


minimize(waste)


writeln("")
forall(i in 1..numPatterns) do
        if getsol(x(i)) > 0 then
                writeln('selected pattern: x(',i,') = ', getsol(x(i)),' rolls')
        end-if
end-do

writeln('total waste = ', getobjval)
```

```
function Pricing(y: array(range) of real, W: integer, w: array(range) of integer) : real

        ! declare decision variable
        ! form constraints and objective
        ! do not forget the integrality constraints on the variables
        ! solve the problem
        !!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!

        ! declare decision variable
        declarations
                len_y = 1..getsize(y)
                a : array(len_y) of mpvar
        end-declarations



        ! form objective
        objective := W - sum(i in len_y)w(i) * a(i) - sum(i in len_y)y(i)*a(i)



        ! form constraints
        sum(i in len_y)(w(i)*a(i)) <= W
        forall(i in len_y) a(i) >= 0
        forall(i in len_y) a(i) is_integer

        ! solve the problem
        minimize(objective)

        ! return the optimal objective value
        ! the keyword returned is reserved for this purpose by Xpress
        returned := getobjval

        ! return the optimal solution, save it in the array newPattern defined in the beginning of
this model file
        !!!!!!!!!!!! write your code here !!!!!!!!!!!!!!!
        forall(i in len_y) do
                newPattern(i) := getsol(a(i))
        end-do
```

```
end-function
```

```
end-model
```

Problem 3

Xpress code for 48-city TSP tour :

```
model ModelName
uses "mmxprs"; !gain access to the Xpress-Optimizer solver
uses "mmsystem" ! include package to operating systems

N := 48  ! number of cities
declarations
        Cities = 1 .. N                         ! set of cities
        coord: array(Cities,1..3) of real       ! array of coordinates of cities, to be read from US48.dat
        dist: array(Cities,Cities) of real      ! distance between each pair of cities
        x : array(Cities,Cities) of mpvar       ! decision variables
        flag : integer                          ! flag=0: not optimal yet; flag=1: optimal
        ind : range                             ! dynamic range
        numSubtour : integer                    ! number of generated subtours
        numSubtourCities : integer              ! number of cities on a generated subtour
        SubtourCities : array(Cities) of integer        ! SubtourCities(i)=1 means city i is on the subtour
        subtourCtr : dynamic array(ind) of linctr       ! dynamic array of subtour elimination constraints
        Next_City: array(Cities) of integer             ! next neighbor of each city
        Subtour: set of integer                         ! set of the cities in a arbitary subtour
        allSubtour : set of integer                     ! set of the cities in the subtours explored so far
        starttime: real                                 ! record starting time
end-declarations

! record initial time
starttime:= gettime

! initialization part is given
initializations from "US48.dat"
    coord
end-initializations

! compute dist(i,j) the distance between each pair of cities using (x,y) coordinates of the cities, which
are in the array coord
  ! you may need square root function sqrt()
  !!!!!!!!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!!!!!
  forall ( i,j in Cities ) do
        if ( j>i ) then
                dist(i,j) := sqrt( (coord(i,2)-coord(j,2))^2 + (coord(i,3)-coord(j,3))^2 )
                dist(j,i) := dist(i,j)
        elif ( i = j ) then
          dist(i,j):= 0
        end-if
  end-do
```

```
  !!!!!!!!!!!! objective: total distance of a tour
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  TotalDist := sum(i, j in Cities) x(i,j)*dist(i,j)

!!!!!!!!!! write constraint: x(i,j) is binary !!!!!!!!!!!!!!!!
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  forall(i,j in Cities ) do
        x(i,j) is_binary
  end-do

  !!!!!!!!!! write assignment constraints: in and out constraints for each city !!!!!!!!!!!!!!!!!!
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  ! constraint for only one out path for each city
  forall(i in Cities) do
        sum ( j in Cities ) x(i,j) = 1
  end-do
  ! constraint for only one into path for each city
  forall(j in Cities) do
        sum ( i in Cities ) x(i,j) = 1
  end-do

  !!!!!!!!!! write 1-city subtour elimination constraints here !!!!!!!!!!!!!!!!!
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  forall(i in Cities) do
        x(i,i) = 0
  end-do

  !!!!!!!!!!! write 2-city subtour elimination constraints here !!!!!!!!!!!!!!!!!!!
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  forall(i,j in Cities ) do
        if ( j> i ) then
              x(i,j) + x(j,i) <= 1
        end-if
  end-do

  !!!!!!!!!! write 3-city subtour elimination constraints here !!!!!!!!!!!!!!
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  ! forall(i,j,k in Cities ) do
  !       if ( k> j and j> i ) then
  !             x(i,j) + x(j,i) + x(i,k) + x(k,i) + x(j,k) + x(k,j) <= 2
  !       end-if
  ! end-do

  !!!!!!!!!! write 4-city subtour elimination constraints here !!!!!!!!!!!!!!
  !!!!!!!!! fill in your code here !!!!!!!!!!!!!!!!
  ! forall(i,j,k,l in Cities ) do
```

```
!	if ( l>k and k>j and j>i ) then
!		x(i,j) + x(j,i) + x(i,k) + x(k,i) + x(j,k) + x(k,j) +
!		x(i,l) + x(l,i) + x(l,k) + x(k,l) + x(j,l) + x(l,j) <= 3
!	end-if
! end-do

!!!!!!!!!!!! constraint generation algorithm !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
numSubtour := 0   ! number of added subtour elimination constraints is zero
flag := 0 ! initalize flag to be 0, so no optimal solution has been found yet
epoch:=0
repeat
	epoch+=1
	writeln("\nEpoch: ", epoch)
	!!!!!!!!!!!!!! Solve the restricted master problem  !!!!!!!!!!
	minimize(TotalDist)

	! Output the solution of the restricted master problem
	writeln("the restricted master problem is solved:")
	forall (i in Cities, j in Cities) do
		if abs(getsol(x(i,j))-1)<0.1 then
		! note here we could have simply written "if getsol(x(i,j))=1 then",
		! but I found cases where Xpress doesn't output all such x(i,j)'s.
		! So this is a quick and ugly fix.
		! You can use this trick in the later part when you need to check if x(i,j) is 1 or not
		! Also, feel free to develop your own solution
			writeln("x(",i,",",j,")=",getsol(x(i,j)))
		end-if
	end-do

	!!!!!!!!!!!!!!!!!!!!  find a subtour !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
	! We want to find a subtour starting at city 1 (Atlanta) and ending at City 1 (such a subtour always exists!)
	! First, initialize a few things:
	numSubtourCities := 0    ! the number of cities on the subtour
	forall (i in Cities) do  ! SubtourCities(i)=1 if city i is on the subtour, initialize all entries to zero
		SubtourCities(i):=0  ! need to change entries when city i is found on the tour
	end-do
	subtourCities(1) := 1  ! City 1 (Atlanta) is always on the subtour

	! Start the procedure to look for a subtour starting and ending at City 1. The basic algorithm is discussed in the hand-out
	! Note you need to update SubtourCities for cities that are on the subtour
	! You also need to keep track of the number of cities numSubtourCities on the subtour
	!!!!!!!! fill in your code here !!!!!!!!!!!!!!
	! find the next neighbor for each city
	forall (i , j in Cities) do
```

```
        if abs(getsol(x(i,j))-1)<0.1 then
        Next_City(i):= j
        end-if
    end-do


        city_trace := 1             ! temparary index to trace the city on a subtour.
                                    ! Initialized as 1, which means this subtour start from city 1
        Subtour:={}                 ! set of the cities in a arbitary subtour, here is the one start from city 1
        allSubtour:={}              ! set of the cities in the subtours explored so far
        repeat
                Subtour+={city_trace}
                SubtourCities(city_trace):=1
                numSubtourCities+=1
       city_trace:=Next_City(city_trace)
        until city_trace = 1
        allSubtour+=Subtour

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        ! output the subtour you found
        writeln("\nFound a tour of distance ", getobjval, " and ", numSubtourCities, " cities")
        writeln("Cities on the subtour are:")
        forall (i in Cities | SubtourCities(i)=1) do
                write(i, " ")
        end-do

        !!!!!!!!!!!!!!!!!!! add the subtour elimination constraint !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        ! If the subtour found above is indeed a subtour (i.e. has fewer than 48 cities),
        ! then add the corresponding subtour elimination constraint to the problem
        ! otherwise, if the subtour has 48 cities, then it's a TSP tour and optimal,
        ! terminate the constraint generation by setting the flag to 1
        !!!!!! fill in you code !!!!!!!!!!
        if ( numSubtourCities = N ) then
                flag := 1
        elif ( numSubtourCities < N ) then
                !Generate the constranit that makes the subtour containing City 1 infeasible
                numSubtour+=1
                create(subtourCtr(numSubtour))
                subtourCtr(numSubtour):= sum(k in Subtour) x(k,Next_City(k)) <= getsize(Subtour) -1
                writeln("subtour Size : ", getsize(Subtour), "  All subtour Size : ", getsize(allSubtour), "
constraints generated : ", numSubtour)

        ! Generate constranits that makes all other subtours infeasible
        writeln("Generate constranits that makes all other subtours infeasible :")
        forall (i in Cities | SubtourCities(i)=0) do
                if (i not in allSubtour) then
```

```
                        Subtour:={}
                        city_trace:=i
                        repeat
                        Subtour+={city_trace}
                        city_trace:=Next_City(city_trace)
                        until city_trace=i
                        numSubtour+=1
                        create(subtourCtr(numSubtour))
                        subtourCtr(numSubtour):= sum(k in Subtour) x(k,Next_City(k)) <=
getsize(Subtour) -1
                        allSubtour += Subtour
 writeln("subtour Size : ", getsize(Subtour), "  All subtour Size : ", getsize(allSubtour), "  constraints
generated : ", numSubtour)
                end-if
        end-do

        ! Generate the constranit that makes at least one edge leave the subtour containing City1 and
enter other subtours
        numSubtour += 1
                create(subtourCtr(numSubtour))
                subtourCtr(numSubtour):= sum (i in Cities | SubtourCities(i)=1, j in Cities |
SubtourCities(j)=0 ) x(i, j) >= 1
                writeln( "Constraints generated : ", numSubtour)
        end-if
   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
until flag = 1

!!!!!!!!!!!!!!!!!!! end of the constraint generation algorithm !!!!!!!!!!!!!!!!!!!!!!!!!
        writeln("\nOptimal TSP distance = ", getobjval)
        writeln("Total: ",numSubtour," constraints generated.")
        writeln("Time in Secs : ", gettime-starttime)
        forall (i in Cities, j in Cities) do
                if abs(getsol(x(i,j))-1)<0.1 then
                        writeln("x(",i,",",j,")=",getsol(x(i,j)))
                end-if
        end-do

        ! write the solution to an output file
        ! then run matlab code US48TourPlot.m to plot the tour
        fopen("US48.output",F_OUTPUT)
        forall (i in Cities, j in Cities) do
                if abs(getsol(x(i,j))-1)<0.1 then
                        writeln(i,"\t",j)
                end-if
        end-do
        fclose(F_OUTPUT)
```

```
writeln("End running model")
end-model
```

Matlab code for 26-city TSP tour :

```
clear all;

outputfile = 'US26.output';
f = fopen(outputfile,'r');
x = fscanf(f, '%d %d', [2, inf]);
fclose(f);
x = x';

coordfile = 'US48.input';
f = fopen(coordfile,'r');
coord = fscanf(f, '%d %f %f', [3, inf]);
fclose(f);
coord = coord';

tour = zeros(26,1);
tour(1) = 1;
fromCity = 1;
for k = 1 : 26
   tour(k+1) = x(fromCity,2);
   fromCity = x(fromCity,2);
end
figure(1);
plot(coord(:,2),coord(:,3),'rO');
hold on;
plot([coord(tour(:,1),2);coord(1,2)],[coord(tour(:,1),3);coord(1,3)]);
plot(coord(1,2),coord(1,3),'r*'); % the star marks Atlanta
```