Homework Collection 2

Table of Contents

1 Homework 8: AMPL Exercise 18-5 . . . 2
Part A . . . 2
Part B . . . 3
Part C . . . 4
Part D . . . 5
2 Homework 9: AMPL Exercise 20-4 . . . 7
Part A . . . 7
Part B . . . 8
Part C . . . 9
Part D . . . 9
Part E . . . 10
Part F . . . 11
3 Homework 10 . . . 12
Problem . . . 12
Solution . . . 12
4 Homework 11 . . . 13
Problem . . . 13
Solution . . . 13
5 Homework 12 . . . 15
Problem . . . 15
6 Homework 13 . . . 16
Problem . . . 16
Relevant Definitions . . . 16
Solution . . . 16
7 Homework 14 . . . 18
Problem . . . 18
Solution . . . 18

1 Homework 8: AMPL Exercise 18-5

A caterer has booked dinners for the next T days, and has as a result a requirement for a certain number of napkins each day. He has a certain initial stock of napkins, and can buy new ones each day at a certain price. In addition, used napkins can be laundered either at a slow service that takes 4 days, or at a faster but more expensive service that takes 2 days. The caterer's problem is to find the most economical combination of puchase and laundering that will meet the forthcoming demand.

Part A

Problem

The book provides the following decision variables:

```
var Buy[t]        # clean napkins bought for day t
var Carry[t]      # clean napkins still on hand at the end of day t.
var Wash2[t]      # used napkins sent to the fast laundry after day t
var Wash4[t]      # used napkins sent to the slow laundry after day t
var Trash[t]      # used napkins discarded after day t
```

There are 2 collections of constraints on these variables, described as such:

1. The number of clean napkins acquired through purchase, carryover and laundering on day t must equal the number sent the laundering, discarded or carried over *after* day t.
2. The number of used napkins laundered or discarded after day t must equal the number that were required for that day's catering.

Solution

For this problem I'll be building an AMPL model from scratch. The goal here was to build a set of parameters that would play nice with the provided variables.

Parameters

```
# Parameters ---
param T integer > 0;
set DAYS := 1..T;
param initial_stock integer >= 0;           # napkins available for start of day 1
param demand{DAYS} integer >= 0;            # Number of napkins needed on day t
param napkin_price > 0;                     # Buy price per single napkin
param wash2_price > 0;                      # Price per napkin for fast laundry
param wash4_price > 0;                      # price per napkin for slow laundry
```

Here we try to keep things simple. I make sure to set the days as an integer and let the prices for the napkins and laundering to be floats. Note here that I assume the price of trashing napkins is free.

Variables

```
# Decision Variables ---
var Buy{t in DAYS} integer >= 0;      # clean napkins bought for day t
var Carry{t in 0..T} integer >= 0;    # clean napkins after day t
var Wash2{t in DAYS} integer >= 0;    # used napkins sent to the fast laundry
var Wash4{t in DAYS} integer >= 0;    # used napkins sent to the slow laundry
var Trash{t in DAYS} integer >= 0;    # used napkins discarded after day t
```

Here I make the decision to force only integer values for the napkins. We allow `Carry` to index on 0 so it can function as our initial inventory.

Objective Function

```
minimize Total_Cost:
        sum {t in DAYS} (napkinprice * Buy[t]
        + slowprice * Wash4[t]
        + fastprice * Wash2[t]);
```

Nothing too crazy here. We add up all the stuff that costs money and try to minimize our spend.

Constraints

```
subject to InitialCarry:
        Carry[0] = initial_stock;

# The number of clean napkins acquired through purchase, carryover and laundering on
day t
# must equal the number sent to laundering, discarded or carried over after day t.
# Basically, this check ensures that we're tracking all the untrashed napkins
correctly
subject to HandlingInventory {t in DAYS}:
        Buy[t]
        + Carry[t-1]
        + (if t-2 >= 1 then Wash2[t-2] else 0)
        + (if t-4 >= 1 then Wash4[t-4] else 0)
        = demand[t] + Carry[t];

# The number of used napkins laundered or discarded after day t
# must equal the number that were required for that day’s catering.
subject to UsedNapkins {t in DAYS}:
        Wash2[t] + Wash4[t] + Trash[t] = demand[t];
```

The interesting part with the constraints is making sure that the scheduling for laundering works out appropriately. We want to avoid any indexing issues. Aside from that though I just follow the contraints to the letter given in the problem statement.

Part B

Problem

Formulate an alternative network linear programming model for this problem. Write it in AMPL using node and arc declarations.

Solution

This one took me a little while to figure out. According to the book, as a general statement, we use `arc` and `node` to take the place of `var` and `subject to` respectively. So there’s a lot of reworking here. The objective function simplifies a lot here as well which is nice.

Parameters

For starters, these are all identical to the previous model. These do not need to change whatsoever.

Nodes

These function as our variables in this context. I visualized this network as having two components to each day. We have an available node per day and a used node per day. This allows the napkins demanded for a day have an organic way to flow out for laundry and trash. The laundry simply has

the used node point to the correct available node down the line. From there we have a node for initial stock, one for a store to buy napkins from and another one for the trash.

```
node Available {t in DAYS}: net_out = 0;
node Used {t in DAYS}: net_out = 0;
node Stock: net_out = initial_stock;        # functions basically as carry[0]
node Store: net_out >= 0;                   # Provides napkins to days
node Trash: net_in >= 0;
```

The `net_out` for available and used ensure that any napkins taken into those nodes are moved out. The store can push out as many napkins as necessary and the trash can take in as many.

Objective Function

```
minimize Total_Cost;
```

Yup, that's it. The arcs will handle this value automatically.

Arcs

This is how we dictate the flow of napkins. We need a lot of these to handle all of our options. They're all pretty intuitive though.

```
arc InitialNapkins <= initial_stock,
    from Stock, to Available[1];

arc Demand {t in DAYS} >= demand[t],
    from Available[t], to Used[t];

arc Carry {t in 1..T-1} >=0,
    from Available[t], to Available[t+1];

arc Buy {t in DAYS} >= 0,
    from Store, to Available[t], obj Total_Cost napkin_price;

arc FastLaundry {t in 1..T-2} >= 0,
    from Used[t], to Available[t+2], obj Total_Cost wash2_price;

arc SlowLaundry {t in 1..T-4} >= 0,
    from Used[t], to Available[t+4], obj Total_Cost wash4_price;

arc TrashFlow {t in DAYS} >= 0,
    from Used[t], to Trash, obj Total_Cost 0;
```

We can see that some of these arcs have a cost associated with them, those will be what alter the value of the objective function.

Part C

Problem

The "caterer problem" was introduced in a 1954 paper by Walter Jacobs of the U.S. Air Force. Although it has been presented in countless books on linear and network programming, it does not seem to have ever been used by any caterer. In what application do you suppose it really originated?

Solution

There is an interesting article on this man titled **Eloge: Walter W. Jacobs, 1914-1982** by Joseph Blum et al. written in 1984. It actually brings up his history in the Air Force and the exact motivation behind this problem. To oversimplify, the problem involved the engines of aircraft.

1. Engines are procured for aircraft.
2. An engine flies a certain number of hours and then is shipped to an overhaul facility. The shipment can be done by slow surface transportation or fast airlift. The fast method is assumed to be more expensive.
3. At some time the aircraft are phased out, and an inventory of engines must be written off.

Robert L. Kirby

This is a very similar setup, it's kind of funny. It's pretty easy to see how this was modified to become napkins and laundry. This is unsurprising, many mathematical advances in optimization came about because of military applications.

Full source of the article: Annals of the History of Computing, Volume 6, Number 2, April 1984.

Part D

Problem

Since this is an artificial problem, you might as well make up your own data for it. Use your data to check that the formulations in (a) and (b) give the same optimal value.

Solution

I kept the data pretty simple here and just randomly generated some daily demands.

```
data;

param T := 7;
param napkin_price := 1.00;
param wash2_price := 0.50;
param wash4_price := 0.25;
param initial_stock := 50;

param demand :=
    1 294
    2 27
    3 129
    4 246
    5 111
    6 251
    7 182;
```

All of these values are totally arbitrarily.

Traditional AMPL model Output

```
Total_Cost = 754.25

: demand   Buy Carry Wash2 Wash4 Trash    :=
0     .     .    50     .     .     .
1   294    244    0    183   111     0
2    27     27    0     22     5     0
```

```
3   129      0   54      0   129     0
4   246    170    0    246     0     0
5   111      0    0     53     0    58
6   251      0    0      0     0   251
7   182      0    0      0     0   182
```

Network Model Output

```
Total_Cost = 754.25

: demand   Buy Carry FastLaundry SlowLaundry TrashFlow    :=
1   294    244     0      241         53           0
2    27     27     0       22          5           0
3   129      0   112       58         71           0
4   246    112     0      246          .           0
5   111      0     0      111          .           0
6   251      0     0        .          .         251
7   182      0     .        .          .         182
```

What we see here is that, though there are some slight difference in strategy on day 4, that both solutions reach the same total cost.

2 Homework 9: AMPL Exercise 20-4

Part A

Problem

Formulate an AMPL model for the kanpsack problem using the provided information.

Solution

I'll keep it simple here.

DATA

```
set OBJECTS:= a b c d e f g h i j;
param weight_limit := 100;

param: weight :=
    a 55
    b 50
    c 40
    d 35
    e 30
    f 30
    g 15
    h 15
    i 10
    j 5 ;

param: value :=
    a 1000
    b 800
    c 750
    d 700
    e 600
    f 550
    g 250
    h 200
    i 200
    j 150 ;
```

MODEL

```
set OBJECTS;
param weight {OBJECTS} > 0;
param value {OBJECTS} > 0;
param weight_limit > 0;

var Use {OBJECTS} integer >= 0, <= 1; # ensure item can only be used once.

maximize Total_Value:
    sum {i in OBJECTS} value[i] * Use[i];

subject to Weight_Limit:
    sum {i in OBJECTS} weight[i] * Use[i] <= weight_limit;
```

Overall this solution is very straightforward.

OUTPUT

```
Total_Value = 2000

: Use  value weight    :=
a   0   1000    55
b   0    800    50
c   0    750    40
d   1    700    35
e   1    600    30
f   1    550    30
g   0    250    15
h   0    200    15
i   0    200    10
j   1    150     5
```

I don't have a lot to say here. It works. We're picking all of the most efficient items for value/weight we can fit into the given weight limit.

Part B

Problem

Suppose instead you want to fill several identical knapsacks. Formulate an AMPL model for this situation while keeping in mind that an item can only go into a single knapsack. For this specific problem, solve for 2 knapsacks each with a weight limit of 50.

Solution

A few updates to make to the model mostly. We mostly need to make sure indexing across knapsacks works properly and that an item is limited to being in a single knapsack.

MODEL UPDATES

```
param knapsacks integer > 0;

var Use {i in OBJECTS, k in 1..knapsacks} integer >= 0, <= 1;

subject to One_Knapsack{i in OBJECTS}:
    sum {k in 1..knapsacks} Use[i, k] <= 1;
```

aside from those changes we just make sure that any time `Use` comes up we include that additional `k` index. This allows us to check the weight limits for all the individual knapsacks. Like so.

```
subject to Weight_Limit{k in 1..knapsacks}:
    sum {i in OBJECTS} weight[i] * Use[i, k] <= weight_limit;
```

DATA UPDATES

The only modifications here are adding a new parameter and updating the global weight limit.

```
param weight_limit := 50;
param knapsacks := 2;
```

OUTPUT

Here we get a total value of 1950. Below is a cleaned up table to show the items used per bag.

Item	Bag 1	Bag 2
a	0	0
b	0	0
c	1	0
d	0	0
e	0	1
f	0	0
g	0	1
h	0	0
i	1	0
j	0	1

Interestingly enough we have a slightly lower total value than part A. This is because of that restrictive 50 weight limit for each bag. We can't use D E and F in our solution as any combination of those items puts us over either bags limit. So we make do with a slightly lower total value.

Part C

Problem

Superficially, the preceding knapsack problem resembles an assignment problem; we have a collection of objects and a collection of knapsacks, and we want to make an optimal assignment from members of the former to members of the latter. What is the essential difference between the kinds of assignment problems described in Section 15.2, and the knapsack problem described in (b)?

Solution

The knapsack problem, in my opinion, closesly matches the bipartite network graph they use for some assignment examples. Thinking about which objects best go into which knapsacks is, in a way, very similar to the example used for assigning employees to the offices they best match. One item can only go to one knapsack much like one employee can only work at a single office. In our case, we would have a node for each item and then on the other side a node for each knapsack. It's very similar conceptually.

The key difference here, I believe, is that the knapsacks in (b) are identical to one another. Whereas in the assignment problems there would be different "weights" or "scores" for the different destinations of the given items. So in 15.2 we would have a knapsack an item best matches, which we don't have here.

Part D

Problem

Modify the formulation from part (a) so that it accommodates a volume limit for the knapsack as well as a weight limit.

How do the total weight, volume and value of this solution compare to those of the solution you found in (a).

Solution

A volume limit wasn't actually provided for this problem so I made up one of 6.

Updates here pretty much just mirror how weight limit was set up initially.

MODEL UPDATES

```
param volume_limit > 0;

subject to Volume_Limit:
    sum {i in OBJECTS} volume[i] * Use[i] <= volume_limit;
```

DATA UPDATES

```
param volume_limit := 6;

param: volume :=
    a 3
    b 3
    c 3
    d 2
    e 2
    f 2
    g 2
    h 1
    i 1
    j 1 ;
```

OUTPUT

```
Total_Value = 1900

: Use  value volume weight    :=
a   1   1000    3      55
b   0    800    3      50
c   0    750    3      40
d   1    700    2      35
e   0    600    2      30
f   0    550    2      30
g   0    250    2      15
h   0    200    1      15
i   1    200    1      10
j   0    150    1       5
```

We see yet another drop in total value here. This is because the solution in (a) is only possible with a volume limit of 7 or larger. This actually prompts item a from being used for once which is interesting. Of note that we still hit the weight limit of 100 exactly here.

Part E

Problem

How can the media selection problem of Exercise 20-1 be viewed as a knapsack problem like the one in (d)?

Solution

Exercise 20-1 is all about an advertising campaign and choosing what kind of mediums to use for advertising. There is a limited budget, and a variety of parameters assigned to each of the mediums. Our choice is whether to use a certain medium or not. The goal is to maximize the audience subject to budget and time constraints.

This is pretty much a more complicated version of what we did in part (d). We have a knapsack that, instead of weight and volume limits, has money and time constraints. It is more of a theoretical container but it still works. The choice of medium is also binary, much like the choice of item. Do we put the item in the bag or not? Do we use a given medium for advertising or not? Instead of maximizing item "value" we're maximizing the audience we reach. It's a near identical setup really.

Part F

Problem

Suppose that you can put up to 3 of each object in the knapsack, instead of just 1. Revise the model, discuss how the solution changes.

Solution

All we change is:

```
var Use {OBJECTS} integer >= 0, <= 3;
```

OUTPUT

```
Total_Value = 2150

: Use  value weight    :=
a   0   1000    55
b   0    800    50
c   0    750    40
d   1    700    35
e   1    600    30
f   0    550    30
g   0    250    15
h   0    200    15
i   2    200    10
j   3    150     5
```

Not as big of a jump in total value as I expected, but it is still 150 higher than in (a). Here the only items we use multiples of are the very light and efficient items `i` and `j`. These are very easy to fit into any weight limit and provide far more bang for their buck than the others. Note that we no longer use `f` as both `i` and `j` have a better value/rate ratio than it. With 2 additional `j` AND `i` objects we get a value of 700 from that 30 weight instead of `f`'s 550.

3 Homework 10

Problem

Explain in your own words why the model in Vanberbei Ch 23-2 (pg 393) is a correct TSP model.

Discuss what is better and worse compared to models 1 and 2 from lecture slides 12.

Solution

Without writing out the model, the books handles the traveling salesman problem using an integer program. At its core, we have a set of cities and weighted connections between those cities. The goal of this model is to minimize the overall cost of visiting each city once and then returning to the start. We refer to this closed cycle as a **tour**.

This model using binary decision variables x_{ij} where i is the "from" city and j is the "to" city. The variable $x_{ij} = 1$ indicates that the connect from city i to city j was used, and the variable is 0 otherwise. Each of these connections has an associated cost, so the objective function sums up the costs from the connections used.

Each tour is looked at as a sequence of cities visited, and so a lot of the constraints focus on making sure the sequence is well behaved. There are **go to** and **come from** constraints that ensure a certain city is only visited once.

$$\sum_{j=1}^{n} x_{ij} = 1 \text{ Only go to each city once}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \text{ Only leave each city once}$$

Additional constraints enforce that the tour order is well-behaved and prevents subtours. This is done using variables t_i, which represents the position of city i in the tour. The ordering constraints require that if the tour goes from city i to city j, then

$$t_j \geq t_i + 1$$

This forces a consistent ordering of visits. Any subtour that ignores any cities would violate these constraints.

This setup is intuitive to set up and it has all of the constraints necessary to ensure the correct behavior for the TSP. How this differs from model 1 and model 2 in the slides is that focus on sequence. The graph solutions meanwhile focus on framing this as a network.

For example, the graph approach forces each city to have two edges used each. This makes sense based on the integer solution, we need an edge for leaving a city and one for approaching it. Subtours are handled using constraints like $\sum_{e \in \delta(S)} \geq 2$ that require enough edges leaving any given subset are used.

Really all three of these models tackle the same thing. They just use slightly different ways of approaching the problem. The integer problem focuses on the behavior of sequences and the graph solution focuses on the behavior of networks.

4 Homework 11

Problem

Prove the following theorem:

> **Theorem 4.1**
>
> A cone K is convex if and only if $K + K \subseteq K$.

Solution

Some definitions:

> **Definition 4.2** — **Convex Set**
>
> A set $S \subseteq \mathbb{R}^n$ is convex if, for every two points, $x_1, x_2 \in S$ and every scalar $\lambda \in (0, 1)$, the point $\lambda x_1 + (1 - \lambda)x_2 \in S$.

> **Definition 4.3** — **Cone**
>
> A set K in $\mathbb{R}^n$ is called a cone if:
>
> $$\lambda K \subseteq K \forall \lambda \geq 0$$

Let K be a cone.

We will first show that K is convex $\Rightarrow K + K \subseteq K$.

Suppose K is convex, then for all $x_1, x_2 \in K$ and any $\lambda \in (0, 1)$,

$$\lambda x_1 + (1 - \lambda)x_2 \in K$$

Let $\lambda = \frac{1}{2}$.

It follows then that,

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 \in K$$

Also, as K is a cone, we also know that $\theta K \subseteq K : \forall \theta \geq 0$.

Letting $\theta = 2$, we have

$$2\left(\frac{1}{2}x_1 + \frac{1}{2}x_2\right) \in K$$
$$x_1 + x_2 \in K$$

As x_1, x_2 were arbitrary elements of K, It follows that $K + K \subseteq K$.

Therefore, K is convex $\Rightarrow K + K \subseteq K$.

Now, we show the other direction : $K + K \subseteq K \Rightarrow K$ is convex.

Assume that $K + K \subseteq K$

Let $x_1, x_2 \in K$ and $\lambda \in [0, 1]$

From this, building off of the given assumption,

$$\lambda x_1 \in K : \forall \lambda$$
$$(1-\lambda)x_2 \in K : \forall \lambda$$

From this,

$$\lambda x_1 + (1-\lambda)x_2 \in K$$

As x_1, x_2 were arbitrary elements of K, this holds for all elements in K and all $\lambda \in [0,1]$.

Therefore, $K + K \subseteq K \Rightarrow K$ is convex.

5 Homework 12

Problem

State and explain and LP to determine if $p^T x \geq q$ is redundant for a system $Ax \geq b$.

Solution

Definition 5.1 **Redundancy**

A constraint is said to be **redundant** for a system of constraints $Ax \geq b$ if its addition or removal does not alter the set.

The general approach here is to optimize the suspected constraint using the feasible set provided by the other constraints. We know that $p^T x$ has a lower bound at q. This approach will allow us to compare to q to draw our conclusions.

$$\min p^T x$$
$$Ax \geq b$$

This gives us an optimal set of variables x^*, so that gives us an optimal value to compare against.

If $p^T x^* \geq q$, this tells us that the other constraints already satisfy everything that the candidate constraint would provide. As such we would deem the candidate redundant.

If $p^T x^* < q$, then that means the candidate constraint provided would restrict the optimal value more. This would indicate that the candidate is not redundant.

6 Homework 13

Problem

Prove the following theorem:

> **Theorem 6.1**
>
> A point $\bar{x}$ is a vertex of S if and only if it is an extreme point of S.

Relevant Definitions

> **Definition 6.2** — **Vertex**
>
> A feasible point $x \in S$ is called a vertex of S if:
>
> $$\text{Rank } A_{\bar{x}} = n$$
>
> (for a dimension of space $\mathbb{R}^n$)

> **Definition 6.3** — **Extreme Point**
>
> A point $\bar{x}$ in a convex set S is called an extreme point of S if it cannot be written as a strict convex combination of any other two points in S.
>
> $$x = \lambda y + (1-\lambda)z \text{ for some } 0 < \lambda < 1$$
> $$\Rightarrow x = y = z$$

> **Definition 6.4** — **Active Constraints**
>
> Given a feasible point $\bar{x} \in S$, a constraint $a_i^T x \geq b_i$ is said to be:
> - Active at $\bar{x}$ if $a_i^T \bar{x} = b_i$ or
> - Inactive at $\bar{x}$ if $a_i^T \bar{x} > b_i$

Solution

($A \Rightarrow B$): Assume $\bar{x}$ is a vertex of S and that it is **not** an extreme point.

Let y, z also be feasible points in S such that $y \neq z$. Also let $\lambda \in [0,1]$. Because $\bar{x}$ is not an extreme point, it can be represented as a a convex combination of two other points in S. Therefore we can state that

$$\bar{x} = \lambda y + (1-\lambda)z$$

As $\bar{x}$ is a vertex, we know that it has active constraints such that $a_i^T \bar{x} = b_i$. Plugging in the equality for $\bar{x}$ gives us:

$$a_i^T \bar{x} = \lambda a_i^T y + a_i^T (1-\lambda) z = b_i$$

As for y and z, since they're feasible points we know that:

$$a_i^T y \geq b_i \qquad a_i^T z \geq b_i$$
$$\lambda a_i^T y \geq \lambda b_i \qquad (1-\lambda) a_i^T z \geq (1-\lambda) b_i$$

It follows from this that

$$\begin{aligned}\lambda a_i^T y + (1-\lambda) a_i^T z &\geq \lambda b_i + (1-\lambda) b_i \\ &\geq b_{i(\lambda+1-\lambda)} \\ &\geq b_i\end{aligned}$$

This is only ever an equality when $y = z$ which is contradicted by our construction of y, z where $y \neq z$.

We now prove $B \Rightarrow A$ using the contrapositive $\neg A \Rightarrow \neg B$.

If $\bar{x}$ is not a vector and $\bar{x} \in S$, then $\text{Rank}(A_{\bar{x}}) < n$.

In other words, the columns of $A_{\bar{x}}$ are linearly dependent. Hence there is a direction vector $y \neq 0 \in \mathbb{R}^n$ such that $A_{\bar{x}} y = 0$. If the columns were linearly independent, only the trivial solution $y = 0$ would work.

We want to show for some $\varepsilon > 0$,

$$\bar{x} + \varepsilon y \in S \text{ and } \bar{x} - \varepsilon y \in S$$

So, let $x_1 = \bar{x} + \varepsilon$ and $x_2 = \bar{x} - \varepsilon$.

$$\begin{aligned}x_1 + x_2 &= \bar{x} + \varepsilon + \bar{x} - \varepsilon \\ &= 2\bar{x}\end{aligned}$$

Now, if include $\lambda = \frac{1}{2}$:

$$\begin{aligned}\frac{1}{2}x_1 + \left(1 - \frac{1}{2}\right)x_2 &= \frac{1}{2}(\bar{x} + \varepsilon) + \frac{1}{2}(\bar{x} - \varepsilon) \\ &= \bar{x}\end{aligned}$$

So we have shown that we can write $\bar{x}$ as a convex combination of two feasible points in S. Therefore, $\bar{x}$ not being a vertex means that it cannot be an extreme point.

This completes the proof.

7 Homework 14

Problem

State the given corollary for all canonical form LPs. Visualize all 4 cases with a small sketch. No explanation or justification is needed.

> **Corollary 7.1**
>
> A feasible point $\bar{x}$ is optimal for (P) if and only if there exists a vector $y \geq 0$ such that $A_{\bar{x}}^T y = c$.

Solution